INTERNATIONAL STANDARD **©ISO/IEC** **ISO/IEC 9899:*2023***

Programming languages — C

Reply To: JeanHeyd Meneide <wg14@soasis.org>

Freek Wiedijk <freek@cs.ru.nl>

Abstract

(This cover sheet to be replaced by ISO.)

This document specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems.

Clauses are included that detail the C language itself and the contents of the C language execution library. Annexes summarize aspects of both of them, and enumerate factors that influence the portability of C programs.

Although this document is intended to guide knowledgeable C language programmers as well as implementors of C language translation systems, the document itself is not designed to serve as a tutorial.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

The following documents, for all intents and purposes, have been applied to this draft from before and during the October 2019 Meeting:

DR 476	volatile semantics for lvalues
DR 488	**c16rtomb()** on wide characters encoded as multiple **char16_t**
DR 494	Part 1: Alignment specifier expression evaluation
DR 496	**offsetof** and subobjects (with editorial modification)
DR 497	"white-space character" defined in two places
DR 499	Anonymous structure in union behavior
DR 500	Ambiguous specification for **FLT_EVAL_METHOD**
DR 501	make **DECIMAL_DIG** obsolescent
FP DR 13	**totalorder** parameters
FP DR 20	changes for obsolescing **DECIMAL_DIG**
FP DR 21	**printf** of one-digit character string
FP DR 22	changes for obsolescing **DECIMAL_DIG**, Part 2
FP DR 23	**llquantexp** invalid case
FP DR 24	**remainder** NaN case
FP DR 25	**totalorder** parameters
N2124 and N2319	rounding direction macro **FE_TONEARESTFROMZERO**
N2186	Alternative to N2166
N2212	type generic **cbrt** (with editorial changes)

N2260	Clarifying the **restrict** Keyword v2
N2265	Harmonizing **static_assert** with C++
N2267	**nodiscard** attribute
N2270	**maybe_unused** attribute
N2271	CR for **pow** divide-by-zero case
N2293	Alignment requirements for memory management functions
N2314	TS 18661-1 plus CR/DRs for C2X
N2322	preprocessor line numbers unspecified
N2325	**DBL_NORM_MAX** etc
N2326	floating-point zero and other normalization
N2334	**deprecated** attribute
N2335	attributes
N2337	**strftime**, with 'b' and 'B' swapped
N2338	error indicator for encoding errors in **fgetwc**
N2341	TS 18661-2 plus CR/DRs for C2X
N2345	editors, resolve ambiguity of a semicolon
N2349	the **memccpy** function
N2350	defining new types in **offsetof**
N2353	the **strdup** and **strndup** functions
N2356	update for payload functions
N2358	no internal state for **mblen**
N2359	part 2 (remove `WANT` macros from numbered clauses) and part 3 (version macros for changed library clauses)
N2401	TS 18661-4a for C2X
N2408	The **fallthrough** attribute
N2412	Two's complement sign representation for C2x
N2417	Section 6: Add time conversion functions that are relatively thread-safe
N2418	Adding the `u8` character prefix
N2432	Remove support for function definitions with identifier lists
N2508	Free Positioning of Labels Inside Compound Statements
N2554	Minor attribute wording cleanups

The following documents have been applied to this draft from the October 2019 Meeting:

N2379	`*_IS_IEC_60559` Feature Test Macros.
N2416	Floating Point Negation and Conversion.
N2384	Annex F.8 Update for Implementation Extensions and Rounding.
N2424	Why **logp1** as a Function Name.
N2406	Signaling NaN Initializers.
N2393	**_Bool** Definitions For **true** and **false**.

The following documents have been applied to this draft from the March/April 2020 Virtual Meeting:

N2444	More optionally per-thread state for the library.
N2446	**printf** of **NAN**().
N2448	`[[Nodiscard("should have a reason")]]`.
N2459	Add an interface to query resolution of time bases, v3.
N2464	Zero-size Reallocations are Undefined Behavior.

N2476	Names and Locations of Floating Point Entities.
N2480	Allowing unnamed parameters in function definitions.
N2490	Why no wide string `strfrom` functions.

The following documents have been applied to this draft from the August 2020 Virtual Meeting:

N2491	`powr` justification
N2492	Note About Math Function Properties.
N2506	Range Errors in Math Functions.
N2508	Free Positioning of Labels.
N2517	Clarification Request for C17 Example of Undefined Behavior.
N2532	Min-max Functions.
N2553	Querying Attribute Support.
N2554	Minor Attribute Wording Cleanup.

The following documents have been applied to this draft from the October and November 2020 Virtual Meetings:

N2546	Missing `DEC_EVAL_METHOD`
N2547	Missing `const` in decimal getpayload functions
N2548	`intmax_t` removal from FP functions
N2549	Binary Literals
N2552	Editorial cleanup for rounding macros
N2557	Allow Duplicate Attributes
N2560	FP hex formatting precision
N2562	Unclear type relationship between a format specifier and its argument
N2563	Character encoding of diagnostic text
N2564	Range errors and math functions (updated previous version, N2506)
N2570	Feature and `WANT` macros for Annex F functions
N2571	`snprintf` nonnegative clarification
N2572	What We Think We Reserve
N2580	Decimal Floating Point Triples
N2586	Sufficient Formatting Precision
N2594	Remove Mixed Wide String Literal Concatenation
N2559	Update to IEC 60559:2020
N2600	Update to IEC 60559:2020 (updates previous version, N2559)
N2602	Infinity/NAN Macros, Editorial Fixes
N2607	Compatibility of Pointers to Arrays with Qualifiers

The following documents have been applied to this draft from the March/April 2021 Virtual Meeting:

N2524	String Functions for Freestanding Implementations
N2626	Digit Separators
N2630	Formatting Input/Output of Binary Integer Numbers
N2640	Missing `DEC_EVAL_METHOD`, Take 2
N2641	Missing `+(x)` in Table
N2643	Negative vs. Less Than Zero
N2645	Add Support for Preprocessing Directives `#elifdef` and `#elifndef`
N2680	Specific Width Length Modifier for Formatting

The following documents have been applied to this draft from the June 2021 Virtual Meeting:

N2651	`fabs` and `copysign` Cleanup
N2662	`[[maybe_unused]]` for Labels
N2665	Zero-size Reallocations Are No Longer an Obsolescent Feature
N2670	Zeros Compare Equal
N2671	Negative Values
N2672	§5.2.4.2.2 Cleanup
N2683	Towards Integer Safety
N2751	`signbit` Cleanup
N2763	Adding a Fundamental Type for N-bit Integers

The following documents have been applied to this draft from the August/September 2021 Virtual Meeting:

N2686	`#warning` Directive
N2688	Sterile Characters
N2710	`SNAN` Fixes
N2711	`fmin`, `fmax`
N2713	Integer Constant Expressions
N2714	`hypot` Changes
N2715	`cr_` Prefix Potentially Reserved for Identifiers
N2716	Fix "numerically"/"numerically equal" Usage
N2726	`_Imaginary_I` and `_Complex_I` Qualifiers
N2728	`char16_t` & `char32_t` String Literals Shall be UTF-16 & UTF-32
N2745	Range Error Definition
N2748	Effects of `fenv` Exception Functions
N2749	IEC 60559 Bindings
N2755	Static Initialization of Decimal Floating Point
N2776	`ckd_*` Identifiers Should be Potentially Reserved Identifiers
N2799	`__has_include` for C

The following documents have been applied to this draft from the November/December 2021 Virtual Meeting:

N2747	Annex F Overflow and Underflow
N2770	Remove UB from Incomplete Types in Function Parameters
N2778	Require Variably-Modified Types
N2781	Types do not have Types (with meeting-agreed changes plus some editorial changes)
N2790	“remquo“ Changes
N2805	Overflow and Underflow Definitions
N2806	§5.2.4.2.2 Cleanup, Again
N2808	Allow 16-bit `ptrdiff_t`
N2823	Freestanding CFP Functions
N2838	Types and Sizes
N2837	Clarifying Integer Terms (also, delete Annex H and replace with the Floating Point TS / Annex merge)
N2842	Normal and Subnormal Classification
N2843	Clarification of Max Exponent Macros
N2845	feraiseexcept Update

N2846	Clarification about Expression Transformations
N2848	INFINITY Macro Contradictions (Wording 1 only!)
N2872	Require Exact-Width Integer Type Interfaces, Part I (Change from proposal's §3.1 only)

The following documents have been applied to this draft from the January/February 2022 Virtual Meeting, Parts 1 and 2:

N2653	`char8_t`: A type for UTF-8 characters and strings
N2701	`@`, `$`, and ' in the source/execution character set
N2754	Decimal Floating Point: Quantum Exponent of NaN
N2762	Fixes for Potentially Reserved Identifiers
N2764	The `_Noreturn` Attribute
N2775	Literal Suffixes for Bit-Precise Integers
N2797	`*_HAS_SUBNORM == 0` Implies What?
N2810	`calloc` Overflow Handling
N2819	Disambiguate the Storage Class of Some Compound Literals
N2826	`unreachable()`
N2828	Unicode Sequences More Than 21 Bits are a Constraint Violation
N2829	Make `assert()` user friendly in C
N2836	Unicode Syntax Identifiers for C
N2840	Make `call_once()` Mandatory
N2841	No Function Declarators without Prototypes
N2844	Remove default promotions for `_FloatN` Types
N2847	Revised Suggestions of Change for Numerically Equal / Equivalent
N2879	5.2.4.2.2 Cleanup, Again Again
N2880	Overflow and Underflow Definitions Update
N2881	Normal and Subnormal Classification Update
N2882	Clarification for the Max Exponent Macros
N2900	Consistent, Warningless, and Intuitive Initialization with `{}`
N2927	Not-So-Magic: `typeof(...)`
N2931	Macros and Macro Spellings from C Floating Point Integration
N2934	Revised Spelling of Keywords
N2935	Make `false` and `true` Language Features
N2937	Properly Define Blocks in the Grammar

The following documents have been applied to this draft from the May 2022 Virtual Meeting:

N2601	Annex X (replacing Annex H) for IEC 60599 Interchange (ratified early 2021 but integrated over a long period of time).
N2861	Indeterminate Values and Trap Representations
N2867	Checked N-Bit Integers? (Not Now)
N2886	Remove `ATOMIC_VAR_INIT`
N2888	Require Exact-width Integer Type Interfaces, Part II
N2897	`memset_explicit`
N2992	Wording Clarification for Variably-Modified Types

The following documents have been applied to this draft from the July 2022 Virtual Meeting:

N2930	Change `remove_quals` to `typeof_unqual`
N2939	Identifier Syntax Fixes

N2940	Remove Trigraphs??!
N2969	Bit-Precise Bit Fields
N2974	Queryable Pointer Alignment
N3029	Improved Normal Enumerations
N2975	Relax requirements for `va_start`
N2993	Make `*_HAS_SUBNORM` Obsolete
N3011	Oops, Empty Initializers in Compound Literals
N3030	Enhanced Enumerations
N2951	Freestanding C and IEC 60559 Conformance Scope Reduction
N2956	Unsequenced Functions
N3033	Comma Ommission and Deletion (`__VA_OPT__` and Preprocessor Wording Improvements)
N3035	`_BitInt(...)` Fixes
N3006	Underspecified Object Declarations
N3007	Type Inference for Object Declarations
N3018	constexpr for Object Definitions
N3038	Introduce Storage Class Specifiers for Compound Literals
N3034	Identifier Primary Expressions
N3042	Introduce the `nullptr_t` constant, `nullptr`
N2929	Memory Layout of `union` s
N3037	Improved Tag Compatibility
N3020	Qualifier-preserving Standard Functions
N3022	Modern Bit Utilities - without Rotate Left/Right, Memory Reversal ("byteswap"), or Endian-Aware Load/Store
N3017	`#embed`
N2957	New Optional Time Bases

In addition to these, the document has undergone some editorial changes, including the following.

— The synopsis lists in Annex B are now generated automatically and classified according to the feature test or `WANT` macros that are required to make them available.

— A new non-normative clause J.6 added to Annex J categorizes identifiers used by this document.

— Renaming of the syntax term "*struct declaration*", "*struct declaration list*" "*struct declarator*", and "*struct declarator list*" to the more appropriate "*member declaration*", "*member declaration list*", "*member declarator*" and "*member declarator list*", respectively.

— Misspelling of "invokation" fixed to "invocation".

— A positional reference to a table was changed to be a more direct reference due to unfortunate page breaks.

— Missing macros were added to from `<float.h>` and `<limits.h>`.

— A footnote added for simple atomic assignment (6.5.16).

— An issue with "modifying object" being removed from an earlier draft was fixed. This was a mistake: side effects do include modifying an object.

— The Decimal Floating Point Initialization text was not well-worded. It was fixed after the paper adding the wording was integrated.

— Examples using poor phrasing for objects and their types were fixed to say "object(s) of type **`int`**" and similar.

— The terms "floating-point type" and "floating-point constant" were changed to just be "floating type" and "floating constant", as are defined in the standard, respectively.

— The wording “thread-local storage” was normalized to be “thread storage” everywhere, as intended (this is the word defined by the standard, the other just fell naturally out of casual usage and thought).

— A footnote clarifying the role for valid pointers with zero size was added to the library frontmatter, specifically concerning functions like `memcpy` and `memset`.

— Various duplicate spellings (e.g. "function functions" and similar) were removed and typos were fixed (e.g., “stirng” and similar).

— The *pp-number* production was incorrect for digit separators. Adjusted and fixed.

— The wording for freestanding heads for `<string.h>` were very poorly done. It was changed to have better wording.

— The introductory sentence for the implementation limits was very wordy and deeply confusing to normal users. The sentence was adjusted to read much better and more clearly.

— In a sentence using “respectively” for `fmin` and `fmax` descriptions, the order of the respective items was swapped. This gave the wrong definitions to each item. They were put in the proper order.

— A missing closing parenthesis in Annex J was fixed.

— The term “floating-point multiply add” was changed to “fused multiply add”, matching naming conventions in reality.

Contents

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

3 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

4 Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see the following URL: www.iso.org/iso/foreword.html.

6 This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

7 This fifth edition cancels and replaces the fourth edition, ISO/IEC 9899:2018. A complete change history can be found in Annex M.

Introduction

1 With the introduction of new devices and extended character sets, new features could be added to this document. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, could conflict with future additions.

2 Certain features are *obsolescent*, which means that they could be considered for withdrawal in future revisions of this document. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.33]) is discouraged.

3 This document is divided into four major subdivisions:

— preliminary elements (Clauses 1–4);

— the characteristics of environments that translate and execute C programs (Clause 5);

— the language syntax, constraints, and semantics (Clause 6);

— the library facilities (Clause 7).

4 Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this document. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes define optional features, provide additional information and summarize the information contained in this document. A bibliography lists documents that were referred to during the preparation of this document.

5 The language clause (Clause 6) is derived from "The C Reference Manual".

6 The library clause (Clause 7) is based on the *1984 /usr/group Standard*.

7 The Working Group responsible for this document (WG 14) maintains a site on the World Wide Web at `http://www.open-std.org/JTC1/SC22/WG14/` containing ancillary information that may be of interest to some readers such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.

INTERNATIONAL STANDARD **©ISO/IEC** **ISO/IEC 9899:*2023***

Programming languages — C

1. Scope

1 This document specifies the form and establishes the interpretation of programs written in the C programming language.[1)] It specifies

- the representation of C programs;
- the syntax and constraints of the C language;
- the semantic rules for interpreting C programs;
- the representation of input data to be processed by C programs;
- the representation of output data produced by C programs;
- the restrictions and limits imposed by a conforming implementation of C.

2 This document does not specify

- the mechanism by which C programs are transformed for use by a data-processing system;
- the mechanism by which C programs are invoked for use by a data-processing system;
- the mechanism by which input data are transformed for use by a C program;
- the mechanism by which output data are transformed after being produced by a C program;
- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

[1)] This document is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and programmers. Annex J gives an overview of portability issues that a C program might encounter.

2. Normative references

1 The following documents are referred to in the text in such a way that some or all their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2 ISO/IEC 2382:2015, *Information technology — Vocabulary*. Available from the *ISO online browsing platform* at `http://www.iso.org/obp`.

3 ISO 4217, *Codes for the representation of currencies and funds*.

4 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

5 ISO/IEC 10646, *Information technology —Universal Coded Character Set (UCS)*. Available from the ISO/IEC Information Technology Task Force (ITTF) web site at `http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm`.

6 ISO/IEC 60559:2020, *Floating-point arithmetic*.

7 ISO 80000–2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*.

8 ISO 80000–3, *Quantities and units — Part 3: Space and time*.

9 The Unicode Consortium. *Unicode Standard Annex, UAX #44, Unicode Character Database [online]*. Edited by Ken Whistler and Laurentiu Iancu. Available at `http://www.unicode.org/reports/tr44`.

10 The Unicode Consortium. *Unicode Standard Annex, UAX #31, Unicode Character Database [online]*. Edited by Ken Whistler and Laurentiu Iancu. Available at `http://www.unicode.org/reports/tr31`.

11 The Unicode Consortium. *The Unicode Standard, Derived Core Properties*. Available at `https://www.unicode.org/Public/UCD/latest/ucd/DerivedCoreProperties.txt`.

3. Terms, definitions, and symbols

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000–2, and the following apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at `https://www.iso.org/obp`

— IEC Electropedia: available at `http://www.electropedia.org/`

3 Additional terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere.

3.1

1 **access (verb)**

⟨execution-time action⟩ to read or modify the value of an object

2 **Note 1 to entry:** Where only one of these two actions is meant, "read" or "modify" is used.

3 **Note 2 to entry:** "Modify" includes the case where the new value being stored is the same as the previous value.

4 **Note 3 to entry:** Expressions that are not evaluated do not access objects.

3.2

1 **alignment**

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.3

1 **argument**

actual argument (DEPRECATED: actual parameter)

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4

1 **behavior**

external appearance or action

3.4.1

1 **implementation-defined behavior**

unspecified behavior where each implementation documents how the choice is made

2 **Note 1 to entry:** J.3 gives an overview over properties of C programs that lead to implementation-defined behavior.

3 **EXAMPLE** An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2

1 **locale-specific behavior**

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

2 **Note 1 to entry:** J.4 gives an overview over properties of C programs that lead to locale-specific behavior.

3 **EXAMPLE** An example of locale-specific behavior is whether the **`islower`** function returns true for characters other than the 26 lowercase Latin letters.

3.4.3

1 **undefined behavior**

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes no requirements

2 **Note 1 to entry:** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

3 **Note 2 to entry:** J.2 gives an overview over properties of C programs that lead to undefined behavior.

4 **EXAMPLE** An example of undefined behavior is the behavior on dereferencing a null pointer.

3.4.4

1 **unspecified behavior**

behavior, that results from the use of an unspecified value, or other behavior upon which this document provides two or more possibilities and imposes no further requirements on which is chosen in any instance

2 **Note 1 to entry:** J.1 gives an overview over properties of C programs that lead to unspecified behavior.

3 **EXAMPLE** An example of unspecified behavior is the order in which the arguments to a function are evaluated.

3.5

1 **bit**

unit of data storage in the execution environment large enough to hold an object that can have one of two values

2 **Note 1 to entry:** It need not be possible to express the address of each individual bit of an object.

3.6

1 **byte**

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

2 **Note 1 to entry:** It is possible to express the address of each individual byte of an object uniquely.

3 **Note 2 to entry:** A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

3.7

1 **character**

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

3.7.1

1 **character**

single-byte character

⟨C⟩ bit representation that fits in a byte

3.7.2

1 **multibyte character**

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

2 **Note 1 to entry:** The extended character set is a superset of the basic character set.

3.7.3

1 **wide character**

value representable by an object of type `wchar_t`, capable of representing any character in the current locale

3.8

1 **constraint**

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

3.9

1 **correctly rounded result**

representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

2 **Note 1 to entry:** In this document, when the words "correctly rounded" are not immediately followed by "result", this is the intended usage.

3 **Note 2 to entry:** IEC 60559 or implementation-defined rules apply for extreme magnitude results if the result format contains infinity.

3.10

1 **diagnostic message**

message belonging to an implementation-defined subset of the implementation's message output

3.11

1 **forward reference**

reference to a later subclause of this document that contains additional information relevant to this subclause

3.12

1 **implementation**

particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13

1 **implementation limit**

restriction imposed upon programs by the implementation

3.14

1 **memory location**

either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

2 **Note 1 to entry:** Two threads of execution can update and access separate memory locations without interfering with each other.

3 **Note 2 to entry:** A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (nonzero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

4 **EXAMPLE** A structure declared as

```
struct {
      char a;
      int b:5, c:11,:0, d:8;
```

```
        struct { int ee:8; } e;
    }
```

contains four separate memory locations: The member a, and bit-fields d and e.ee are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c cannot be concurrently modified, but b and a, for example, can be.

3.15

1 **object**

region of data storage in the execution environment, the contents of which can represent values

2 **Note 1 to entry:** When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **parameter**

formal parameter

DEPRECATED: formal argument

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.17

1 **recommended practice**

specification that is strongly recommended as being in keeping with the intent of the standard, but that might be impractical for some implementations

3.18

1 **runtime-constraint**

requirement on a program when calling a library function

2 **Note 1 to entry:** Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 **Note 2 to entry:** Implementations that support the extensions in Annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

4 **Note 3 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.19

1 **value**

precise meaning of the contents of an object when interpreted as having a specific type

3.19.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.19.2

1 **indeterminate representation**

object representation that either represents an unspecified value or is a non-value representation

3.19.3

1 **unspecified value**

valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance

3.19.4

1 **non-value representation**

an object representation that does not represent a value of the object type

3.19.5

1 **perform a trap**

interrupt execution of the program such that no further operations are performed[2)]

2 **Note 1 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.20

1 $\lceil x \rceil$

ceiling of x

the least integer greater than or equal to x

2 **EXAMPLE** $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2.

3.21

1 $\lfloor x \rfloor$

floor of x

the greatest integer less than or equal to x

2 **EXAMPLE** $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3.

3.22

1 **wraparound**

the process by which a value is reduced modulo 2^N, where N is the width of the resulting type

[2)]Note that fetching a non-value representation might perform a trap but is not required to (see 6.2.6.1).

4. Conformance

1 In this document, "shall" is to be interpreted as a requirement on an implementation or on a program; conversely, "shall not" is to be interpreted as a prohibition.

2 If a "shall" or "shall not" requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this document by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined".

3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

4 The implementation shall not successfully translate a preprocessing translation unit containing a **`#error`** preprocessing directive unless it is part of a group skipped by conditional inclusion.

5 A *strictly conforming program* shall use only those features of the language and library specified in this document.[3] It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

6 The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library clause (Clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbit.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>`. Additionally, a *conforming freestanding implementation* shall accept any strictly conforming program where:

— the features specified in the header `<string.h>` are used, except the following functions: **`strdup`**, **`strndup`**, **`strcoll`**, **`strxfrm`**, **`strerror`**; and/or,

the selected function **`memalignment`** from `<stdlib.h>` is used.

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[4]

7 The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines **`__STDC_IEC_60559_BFP__`** or **`__STDC_IEC_60559_DFP__`** may also use features in the contents of the standard headers `<fenv.h>`, `<math.h>`, and the **`strto`**∗ floating-point numeric conversion functions (7.24.1) of the standard header `<stdlib.h>`, provided the program does not set the state of the **`FENV_ACCESS`** pragma to "**ON**".

All identifiers that are reserved when `<stdlib.h>` is included in a hosted implementation are reserved when it is included in a freestanding implementation.

8 A *conforming program* is one that is acceptable to a conforming implementation. [5]

---

[3] A strictly conforming program can use conditional features (see 6.10.9.3) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```
#ifdef __STDC_IEC_60559_BFP__ /* FE_UPWARD defined */
    /* ... */
    fesetround(FE_UPWARD);
    /* ... */
#endif
```

[4] This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this document.

[5] Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs can depend upon nonportable features of a conforming implementation.

9 An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.6), characteristics of floating types `<float.h>` (7.7), alternative spellings `<iso646.h>` (7.9), sizes of integer types `<limits.h>` (7.10), alignment `<stdalign.h>` (7.15), variable arguments `<stdarg.h>` (7.16), boolean type and values `<stdbool.h>` (7.19), common definitions `<stddef.h>` (7.21), integer types `<stdint.h>` (7.22), `<stdnoreturn.h>` (7.25).

5. Environment

1 An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this document. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In this clause, only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

1 A C program need not all be translated at the same time. The text of the program is kept in units called *source files*, (or *preprocessing files*) in this document. A source file together with all the headers and source files included via the preprocessing directive **`#include`** is known as a *preprocessing translation unit*. After preprocessing, a preprocessing translation unit is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

1 The precedence among the syntax rules of translation is specified by the following phases.[6)]

1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

3. The source file is decomposed into preprocessing tokens[7)] and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed, macro invocations are expanded, and **`_Pragma`** unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.4.3), the behavior is undefined. A **`#include`** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

[6)] This requires implementations to behave as if these separate phases occur, even though many are typically folded together in practice. Source files, translation units, and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

[7)] As described in 6.4, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of **`<`** within a **`#include`** preprocessing directive.

5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set. Each instance of a source character or escape sequence for which there is no corresponding member is converted in an implementation-defined manner to some member of the execution character set other than the null (wide) character.[8)]

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), external definitions (6.9).

5.1.1.3 Diagnostics

1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.[9)]

2 **EXAMPLE** An implementation is required to issue a diagnostic for the translation unit:

```
char i;
int i;
```

because in those cases where wording in this document describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error is still required to be diagnosed.

5.1.2 Execution environments

1 Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects with static storage duration shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

Forward references: storage durations of objects (6.2.4), initialization (6.7.10).

5.1.2.1 Freestanding environment

1 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by Clause 4, are implementation-defined.

2 The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

1 A hosted environment need not be provided, but shall conform to the following specifications if present.

[8)]An implementation may convert each instance of the same non-corresponding source character to a different member of the execution character set.

[9)]An implementation is encouraged to identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostic messages, often referred to as warnings, as long as a valid program is still correctly translated. It can also successfully translate an invalid program. Annex I lists a few of the more common warnings.

5.1.2.2.1 Program startup

1 The function called at program startup is named **main**. The implementation declares no prototype for this function. It shall be defined with a return type of **int** and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[]) { /* ... */ }
```

or equivalent[10]; or in some other implementation-defined manner.

2 If they are declared, the parameters to the **main** function shall obey the following constraints:

— The value of `argc` shall be nonnegative.

— `argv[argc]` shall be a null pointer.

— If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

— If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the *program name*; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the *program parameters*.

— The parameters `argc` and `argv` and the strings pointed to by the `argv` array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

5.1.2.2.2 Program execution

1 In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (Clause 7).

5.1.2.2.3 Program termination

1 If the return type of the **main** function is a type compatible with **int**, a return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument;[11] reaching the `}` that terminates the **main** function returns a value of 0. If the return type is not compatible with **int**, the termination status returned to the host environment is unspecified.

Forward references: definition of terms (7.1.1), the **exit** function (7.24.4.4).

5.1.2.3 Program execution

1 The semantic descriptions in this document describe the behavior of an abstract machine in which issues of optimization are irrelevant.

2 An access to an object through the use of an lvalue of volatile-qualified type is a *volatile access*. A volatile access to an object, modifying an object, modifying a file, or calling a function that does any

[10] Thus, **int** can be replaced by a typedef name defined as **int**, or the type of `argv` can be written as `char ** argv`, and so on.

[11] In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **main** will have ended in the former case, even where they would not have in the latter.

of those operations are all *side effects*[12], which are changes in the state of the execution environment.

Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

3 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B*, then the execution of *A* shall precede the execution of *B*. (Conversely, if *A* is sequenced before *B*, then *B* is *sequenced after A*.) If *A* is not sequenced before or after *B*, then *A* and *B* are *unsequenced*. Evaluations *A* and *B* are *indeterminately sequenced* when *A* is sequenced either before or after *B*, but it is unspecified which.[13] The presence of a *sequence point* between the evaluation of expressions *A* and *B* implies that every value computation and side effect associated with *A* is sequenced before every value computation and side effect associated with *B*. (A summary of the sequence points is given in Annex C.)

4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or through volatile access to an object).

5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` are unspecified, as is the state of the dynamic floating-point environment. The representation of any object modified by the handler that is neither a lock-free atomic object nor of type `volatile sig_atomic_t` becomes indeterminate when the handler exits, as does the state of the dynamic floating-point environment if it is modified by the handler and not restored to its original state.

6 The least requirements on a conforming implementation are:

— Volatile accesses to objects are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place as specified in 7.23.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages appear prior to a program waiting for input.

This is the *observable behavior* of the program.

7 What constitutes an interactive device is implementation-defined.

8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.

9 **EXAMPLE 1** An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword `volatile` would then be redundant.

10 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function

[12]The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see Annex F for details. The floating-point environment library `<fenv.h>` provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

[13]The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the `signal` function would require explicit specification of `volatile` storage, as well as other implementation-defined restrictions.

11 **EXAMPLE 2** In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the "integer promotions" require that the abstract machine promote the value of each variable to `int` size and then add the two `int`s and truncate the sum. Provided the addition of two `char`s can be done without integer overflow, or with integer overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

12 **EXAMPLE 3** Similarly, in the fragment

```
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
```

the multiplication can be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if `d` were replaced by the constant `2.0`, which has type `double`).

13 **EXAMPLE 4** Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to `d1` and `d2` are required to have been converted to `float`.

14 **EXAMPLE 5** Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.9).

```
double x, y, z;
/* ... */
x = (x * y) * z;  // not equivalent to x *= y * z;
z = (x - y) + y;  // not equivalent to z = x;
z = x + x * y;    // not equivalent to z = x * (1.0 + y);
y = x / 5.0;      // not equivalent to y = x * 0.2;
```

15 **EXAMPLE 6** To illustrate the grouping behavior of expressions, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum `(a + 32760)` is next added to `b`, and that result is then added to `5` which results in the value assigned to `a`. On a machine in which integer overflows produce an explicit trap and in which the range of values representable by an **`int`** is $[-32768, +32767]$, the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for `a` and `b` were, respectively, -32754 and -15, the sum `a + b` would produce a trap while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However, on a machine in which integer overflow silently generates some value and where positive and negative integer overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

16 **EXAMPLE 7** The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of `p` can occur at any time between the previous sequence point and the next sequence point (the `;`), and the call to **`getchar`** can occur at any point prior to the need of its returned value.

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment `<fenv.h>` (7.6), the **`signal`** function (7.14), files (7.23.3).

5.1.2.4 Multi-threaded executions and data races

1 Under a hosted implementation, a program can have more than one *thread of execution* (or *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all its threads.[14] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 The value of an object visible to a thread T at a particular point is the initial value of the object, a value stored in the object by T, or a value stored in the object by another thread, according to the rules below.

3 **NOTE 1** In some cases, there could instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs.

[14]The execution can usually be viewed as an interleaving of all the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below.

4 Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.

5 The library defines *atomic operations* (7.17) and operations on mutexes (7.28.4) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A *synchronization operation* on one or more memory locations is one of an *acquire operation*, a *release operation*, both an acquire and release operation, or a *consume operation*. A synchronization operation without an associated memory location is a *fence* and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are *relaxed atomic operations*, which are not synchronization operations, and atomic *read-modify-write operations*, which have special characteristics.

6 **NOTE 2** For example, a call that acquires a mutex will perform an acquire operation on the locations composing the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform an acquire or consume operation on A. Relaxed atomic operations are not included as synchronization operations although, like synchronization operations, they cannot contribute to data races.

7 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M, and A happens before B, then A shall precede B in the modification order of M, which is defined below.

8 **NOTE 3** This states that the modification orders are expected to respect the "happens before" relation.

9 **NOTE 4** There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different variables in inconsistent orders.

10 A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A and every subsequent operation either is performed by the same thread that performed the release or is an atomic read-modify-write operation.

11 Certain library calls *synchronize with* other library calls performed by another thread. In particular, an atomic operation A that performs a release operation on an object M synchronizes with an atomic operation B that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by A.

12 **NOTE 5** Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation.

13 **NOTE 6** The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release.

14 An evaluation A *carries a dependency*[15)] to an evaluation B if:

— the value of A is used as an operand of B, unless:

- B is an invocation of the `kill_dependency` macro,
- A is the left operand of a && or `||` operator,
- A is the left operand of a `?:` operator, or
- A is the left operand of a `,` operator;

or

— A writes a scalar object or bit-field M, B reads from M the value written by A, and A is sequenced before B, or

— for some evaluation X, A carries a dependency to X and X carries a dependency to B.

15 An evaluation A is *dependency-ordered before*[16)] an evaluation B if:

15)The "carries a dependency" relation is a subset of the "sequenced before" relation, and is similarly strictly intra-thread.
16)The "dependency-ordered before" relation is analogous to the "synchronizes with" relation, but uses release/consume in place of release/acquire.

— A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

— for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

16 An evaluation A *inter-thread happens before* an evaluation B if A synchronizes with B, A is dependency-ordered before B, or, for some evaluation X:

— A synchronizes with X and X is sequenced before B,

— A is sequenced before X and X inter-thread happens before B, or

— A inter-thread happens before X and X inter-thread happens before B.

17 **NOTE 7** The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with", and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before".

18 An evaluation A *happens before* an evaluation B if A is sequenced before B or A inter-thread happens before B. The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation.

19 **NOTE 8** This cycle would otherwise be possible only through the use of consume operations.

20 A *visible side effect* A on an object M with respect to a value computation B of M satisfies the conditions:

— A happens before B, and

— there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object M, as determined by evaluation B, shall be the value stored by the visible side effect A.

21 **NOTE 9** If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.

22 **NOTE 10** This states that operations on ordinary variables are not visibly reordered. This is not detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.

23 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

24 **NOTE 11** The set of side effects from which a given evaluation might take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below.

25 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M.

26 **NOTE 12** The requirement above is known as "write-write coherence".

27 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M.

28 **NOTE 13** The requirement above is known as "read-read coherence".

29 If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.

30 **NOTE 14** The requirement above is known as "read-write coherence".

31 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

32 **NOTE 15** The requirement above is known as "write-read coherence".

33 **NOTE 16** This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are "relaxed" loads. By doing so, it effectively makes the "cache coherence" guarantee provided by most hardware available to C atomic operations.

34 **NOTE 17** The value observed by a load of an atomic object depends on the "happens before" relation, which in turn depends on the values observed by loads of atomic objects. The intended reading is that there exists an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here.

35 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

36 **NOTE 18** It can be shown that programs that correctly use simple mutexes and `memory_order_seq_cst` operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result necessarily has undefined behavior even before such a transformation is applied.

37 **NOTE 19** Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence requirements.

38 **NOTE 20** Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

1 Two sets of characters and their associated *collating sequences* shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. The combined set is also called the *extended character set*. The values of the members of the execution character set are implementation-defined.

2 In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash `\` followed by one or more characters. A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string.

3 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the *Latin alphabet*

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
```

the 26 *lowercase letters* of the Latin alphabet

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

the 10 decimal *digits*

```
0 1 2 3 4 5 6 7 8 9
```

the following 29 *graphic characters*

```
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed. The representation of each member of the source and execution basic character sets shall fit in a byte. In both the source and execution basic character sets, the value of each character after `0` in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this document treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined.

4 A *letter* is an uppercase letter or a lowercase letter as defined above; in this document the term does not include other characters that are letters in other alphabets.

5 The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1).

5.2.1.1 Multibyte characters

1 The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:

— The basic character set, @, $, and ` shall be present and each character shall be encoded as a single byte.

— The presence, meaning, and representation of any additional members is locale-specific.

— A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

— A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.

2 For source files, the following shall hold:

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

1 The *active position* is that location on a display device where the next character output by the **`fputc`** function would appear. The intent of writing a printing character (as defined by the **`isprint`** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

2 Alphabetic escape sequences representing non-graphic characters in the execution character set are intended to produce actions on display devices as follows:

`\a` (*alert*) Produces an audible or visible alert without changing the active position.

`\b` (*backspace*) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.

`\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page.

`\n` (*new line*) Moves the active position to the initial position of the next line.

`\r` (*carriage return*) Moves the active position to the initial position of the current line.

`\t` (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.

`\v` (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

3 Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single **`char`** object. The external representations in a text file need not be identical to the internal representations, and are outside the scope of this document.

Forward references: the **`isprint`** function (7.4.1.8), the **`fputc`** function (7.23.7.3).

5.2.3 Signals and interrupts

1 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis.

5.2.4 Environmental limits

1 Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the language-related environmental limits on a conforming implementation; the library-related limits are discussed in Clause 7.

5.2.4.1 Translation limits

1 The implementation shall be able to translate and execute a program that uses but does not exceed the following limitations for these constructs and entities[17]:

— 127 nesting levels of blocks

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or **`void`** type in a declaration

— 63 nesting levels of parenthesized declarators within a full declarator

— 63 nesting levels of parenthesized expressions within a full expression

— 63 significant initial characters in an internal identifier or a macro name(each universal character name or extended source character is considered a single character)

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[18]

[17]Implementations are encouraged to avoid imposing fixed translation limits whenever possible.
[18]See "future language directions" (6.11.3).

— 4095 external identifiers in one translation unit

— 511 identifiers with block scope declared in one block

— 4095 macro identifiers simultaneously defined in one preprocessing translation unit

— 127 parameters in one function definition

— 127 arguments in one function call

— 127 parameters in one macro definition

— 127 arguments in one macro invocation

— 4095 characters in a logical source line

— 4095 characters in a string literal (after concatenation)

— 32767 bytes in an object (in a hosted environment only)

— 15 nesting levels for **#include**d files

— 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)

— 1023 members in a single structure or union

— 1023 enumeration constants in a single enumeration

— 63 levels of nested structure or union definitions in a single member declaration list

5.2.4.2 Numerical limits

1 An implementation is required to document all the limits specified in this subclause, which are specified in the headers `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`.

Forward references: integer types `<stdint.h>` (7.22).

5.2.4.2.1 Characteristics of integer types <limits.h>

1 The values given below shall be replaced by constant expressions suitable for use in conditional expression inclusion preprocessing directives. Their implementation-defined values shall be equal or greater to those shown.

— width for an object of type **bool**[19]

```
BOOL_WIDTH                     1
```

— number of bits for smallest object that is not a bit-field (byte)

```
CHAR_BIT                       8
```

The macros **CHAR_WIDTH**, **SCHAR_WIDTH**, and **UCHAR_WIDTH** that represent the width of the types **char**, **signed char** and **unsigned char** shall expand to the same value as **CHAR_BIT**.

— width for an object of type **unsigned short int**

```
USHRT_WIDTH                    16
```

The macro **SHRT_WIDTH** represents the width of the type **short int** and shall expand to the same value as **USHRT_WIDTH**.

— width for an object of type **unsigned int**

[19]This value is exact.

```
UINT_WIDTH                    16
```

The macro **INT_WIDTH** represents the width of the type **int** and shall expand to the same value as **UINT_WIDTH**.

— width for an object of type **unsigned long int**

```
ULONG_WIDTH                   32
```

The macro **LONG_WIDTH** represents the width of the type **long int** and shall expand to the same value as **ULONG_WIDTH**.

— width for an object of type **unsigned long long int**

```
ULLONG_WIDTH                  64
```

The macro **LLONG_WIDTH** represents the width of the type **long long int** and shall expand to the same value as **ULLONG_WIDTH**.

— maximum width for an object of type **_BitInt** or **unsigned _BitInt**

```
BITINT_MAXWIDTH     /* see below */
```

The macro **BITINT_MAXWIDTH** represents the maximum width N supported by the declaration of a bit-precise integer (6.2.5) in the type specifier **_BitInt**(N). The value **BITINT_MAXWIDTH** shall expand to a value that is greater than or equal to the value of **ULLONG_WIDTH**.

— maximum number of bytes in a multibyte character, for any supported locale

```
MB_LEN_MAX                    1
```

2 For all unsigned integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix **_WIDTH** holding its width N, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that is representable by the type and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. If the value is in the range of the type **uintmax_t** (7.22.1.5) the macro is suitable for use in conditional expression inclusion preprocessing directives.

3 For all signed integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix **_WIDTH** holding its width N, there are macros with suffix **_MIN** and **_MAX** holding the minimal and maximal values -2^{N-1} and $2^{N-1} - 1$ that are representable by the type and that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. If the values are in the range of the type **intmax_t** (7.22.1.5) the macros are suitable for use in conditional expression inclusion preprocessing directives.

4 If an object of type **char** can hold negative values, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**. Otherwise, the value of **CHAR_MIN** shall be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.[20]

Forward references: representations of types (6.2.6), conditional inclusion (6.10.1), integer types `<stdint.h>` (7.22).

5.2.4.2.2 Characteristics of floating types `<float.h>`

1 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and allows other values. The characteristics provide information about an implementation's floating-point arithmetic.[21] An implementation that de-

[20] See 6.2.5.

[21] The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical.

fines `__STDC_IEC_60559_BFP__` or `__STDC_IEC_559__` shall implement floating types and arithmetic conforming to IEC 60559 as specified in Annex F. An implementation that defines `__STDC_IEC_60559_COMPLEX__` or `__STDC_IEC_559_COMPLEX__` shall implement complex types and arithmetic conforming to IEC 60559 as specified in Annex G.

2 The following parameters are used to define the model for each floating type:

s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

For each floating type, the parameters b, p, $e_{\min}$, and $e_{\max}$ are fixed constants.

3 For each floating type, a *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

4 Model floating-point numbers x with $f_1 > 0$ are called *normalized floating-point numbers*.

5 Model floating-point numbers $x \neq 0$ with $f_1 = 0$ and $e = e_{min}$ are called *subnormal floating-point numbers*.

6 Model floating-point numbers $x \neq 0$ with $f_1 = 0$ and $e > e_{min}$ are called *unnormalized floating-point numbers*.

7 Model floating-point numbers x with all $f_k = 0$ are zeros.

8 Floating types shall be able to represent signed zeros or an unsigned zero and all normalized floating-point numbers. In addition, floating types may be able to contain other kinds of floating-point numbers[22], such as subnormal floating-point numbers and unnormalized floating-point numbers, and values that are not floating-point numbers, such as NaNs and (signed and unsigned) infinities. A *NaN* is a value signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand[23].

9 Wherever values are unsigned, any requirement in this document to get the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored, unless otherwise specified.[24]

10 Whether and in what cases subnormal numbers are treated as zeros is implementation-defined. Subnormal numbers that in some cases are treated by arithmetic operations as zeros are properly classified as subnormal. However, object representations that could represent subnormal numbers but that are always treated by arithmetic operations as zeros are non-canonical zeros, and the values are properly classified as zero, not subnormal. IEC 60559 arithmetic (with default exception handling) always treats subnormal numbers as nonzero.

11 A value is negative if and only if it compares less than 0. Thus, negative zeros and NaNs are not negative values.

12 An implementation may prefer particular representations of values that have multiple representations in a floating type, 6.2.6.1 not withstanding.[25] The preferred representations of a floating type, including unique representations of values in the type, are called *canonical*. A floating type may also contain *non-canonical* representations, for example, redundant representations of some or all its values, or representations that are extraneous to the floating-point model.[26] Typically, floating-point

[22]Some implementations have types that include finite numbers with range and/or precision that are not covered by the model.

[23]IEC 60559 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559, the terms quiet NaN and signaling NaN are intended to apply to values with similar behavior.

[24]Bit representations of floating-point values might include a sign bit, even if the values can be regarded as unsigned. IEC 60559 NaNs are such values.

[25]The library operations `iscanonical` and `canonicalize` distinguish canonical (preferred) representations, but this distinction alone does not imply that canonical and non-canonical representations are of different values.

[26]Some of the values in the IEC 60559 decimal formats have non-canonical representations (as well as a canonical representation).

operations deliver results with canonical representations. IEC 60559 operations deliver results with canonical representations, unless specified otherwise.

13 The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. In addition, if negative infinity is representable in a type, the range of that type is extended to all negative real numbers; likewise, if positive infinity is representable in a type, the range of that type is extended to all positive real numbers.

14 The accuracy of the floating-point operations ($+$, $-$, $*$, `/`) and of most of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown. Decimal floating-point operations have stricter requirements.

15 All integer values in the `<float.h>` header, except **`FLT_ROUNDS`**, shall be constant expressions suitable for use in conditional expression inclusion preprocessing directives; all floating values shall be constant expressions. All except **`CR_DECIMAL_DIG`** (F.5), **`DECIMAL_DIG`**, **`DEC_EVAL_METHOD`**, **`FLT_EVAL_METHOD`**, **`FLT_RADIX`**, and **`FLT_ROUNDS`** have separate names for all floating types. The floating-point model representation is provided for all values except **`DEC_EVAL_METHOD`**, **`FLT_EVAL_METHOD`** and **`FLT_ROUNDS`**.

16 The remainder of this subclause specifies characteristics of standard floating types.

17 The rounding mode for floating-point addition for standard floating types is characterized by the implementation-defined value of **`FLT_ROUNDS`**. Evaluation of **`FLT_ROUNDS`** correctly reflects any execution-time change of rounding mode through the function **`fesetround`** in `<fenv.h>`.

-1	indeterminable
0	toward zero
1	to nearest, ties to even
2	toward positive infinity
3	toward negative infinity
4	to nearest, ties away from zero

All other values for **`FLT_ROUNDS`** characterize implementation-defined rounding behavior.

18 Whether a type matches an IEC 60559 format (and perhaps, operations) is characterized by the implementation-defined values of **`FLT_IS_IEC_60559`**, **`DBL_IS_IEC_60559`**, and **`LDBL_IS_IEC_60559`** (this does not imply conformance to Annex F):

0 type does not match an IEC 60559 format

1 type matches an IEC 60559 format

2 type matches an IEC 60559 format and operations

19 The values of floating type yielded by operators subject to the usual arithmetic conversions, including the values yielded by the implicit conversion of operands, and the values of floating constants are evaluated to a format whose range and precision may be greater than required by the type. Such a format is called an *evaluation format*. In all cases, assignment and cast operators yield values in the format of the type. The extent to which evaluation formats are used is characterized by the value of **`FLT_EVAL_METHOD`**:[27)]

[27)]The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **`FLT_EVAL_METHOD`** is 1, then the product of two **`float _Complex`** operands is represented in the **`double _Complex`** format, and its parts are evaluated to **`double`**.

−1 indeterminable;

0 evaluate all operations and constants just to the range and precision of the type;

1 evaluate operations and constants of type **float** and **double** to the range and precision of the **double** type, evaluate **long double** operations and constants to the range and precision of the **long double** type;

2 evaluate all operations and constants to the range and precision of the **long double** type.

All other negative values for **FLT_EVAL_METHOD** characterize implementation-defined behavior. The value of **FLT_EVAL_METHOD** does not characterize values returned by function calls (see 6.8.6.4, F.6).

20 The presence or absence of subnormal numbers is characterized by the implementation-defined values of **FLT_HAS_SUBNORM**, **DBL_HAS_SUBNORM**, and **LDBL_HAS_SUBNORM**:

−1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

The use of **FLT_HAS_SUBNORM**, **DBL_HAS_SUBNORM**, and **LDBL_HAS_SUBNORM** macros is an obsolescent feature.

21 Each of the signaling NaN macros

```
FLT_SNAN
DBL_SNAN
LDBL_SNAN
```

is defined if and only if the respective type contains signaling NaNs. They expand to a constant expression of the respective type representing a signaling NaN. If an optional unary + or − operator followed by a signaling NaN macro is used as the initializer for initializing an object of the same type that has static or thread storage duration, the object is initialized with a signaling NaN value.

22 The macro

```
INFINITY
```

is defined if and only if the implementation supports an infinity for the type **float**. It expands to a constant expression of type **float** representing positive or unsigned infinity.

23 The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet NaN.

24 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

— radix of exponent representation, b

```
FLT_RADIX                         2
```

— number of base-**FLT_RADIX** digits in the floating-point significand, p

```
FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG
```

— number of decimal digits, n, such that any floating-point number with p radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
FLT_DECIMAL_DIG                  6
DBL_DECIMAL_DIG                 10
LDBL_DECIMAL_DIG                10
```

— number of decimal digits, n, such that any floating-point number in the widest of the supported floating types and the supported IEC 60559 encodings with $p_{\max}$ radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p_{\max} \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
DECIMAL_DIG                     10
```

This is an obsolescent feature, see 7.33.8.

— number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

```
FLT_DIG                          6
DBL_DIG                         10
LDBL_DIG                        10
```

— minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, $e_{\min}$

```
FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP
```

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil log_{10} b^{e_{\min}-1} \rceil$

```
FLT_MIN_10_EXP                 -37
DBL_MIN_10_EXP                 -37
LDBL_MIN_10_EXP                -37
```

— maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number; if that representable finite floating-point number is normalized, the value of the macro is $e_{\max}$

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor log_{10}((1 - b^{-p})b^{e_{\max}})\rfloor$

```
FLT_MAX_10_EXP                 +37
DBL_MAX_10_EXP                 +37
LDBL_MAX_10_EXP                +37
```

25 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown:

— maximum representable finite floating-point number; if that number is normalized, its value is $(1 - b^{-p})b^{e_{\max}}$

```
FLT_MAX                        1E+37
DBL_MAX                        1E+37
LDBL_MAX                       1E+37
```

— maximum normalized floating-point number, $(1 - b^{-p})b^{e_{\max}}$

```
FLT_NORM_MAX                   1E+37
DBL_NORM_MAX                   1E+37
LDBL_NORM_MAX                  1E+37
```

26 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

— the difference between 1 and the least normalized value greater than 1 that is representable in the given floating type, b^{1-p}

```
FLT_EPSILON                     1E-5
DBL_EPSILON                     1E-9
LDBL_EPSILON                    1E-9
```

— minimum normalized positive floating-point number, $b^{e_{\min}-1}$

```
FLT_MIN                        1E-37
DBL_MIN                        1E-37
LDBL_MIN                       1E-37
```

— minimum positive floating-point number[28)]

```
FLT_TRUE_MIN                   1E-37
DBL_TRUE_MIN                   1E-37
LDBL_TRUE_MIN                  1E-37
```

[28)]If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.

Recommended practice

27 Conversion between real floating type and decimal character sequence with at most *T*`_DECIMAL_DIG` digits should be correctly rounded, where *T* is the macro prefix for the type. This assures conversion from real floating type to decimal character sequence with *T*`_DECIMAL_DIG` digits and back, using to-nearest rounding, is the identity function.

28 **EXAMPLE 1** The following describes an artificial floating-point representation that meets the minimum requirements of this document, and the appropriate values in a `<float.h>` header for type **`float`**:

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
FLT_RADIX                         16
FLT_MANT_DIG                       6
FLT_EPSILON           9.53674316E-07F
FLT_DECIMAL_DIG                    9
FLT_DIG                            6
FLT_MIN_EXP                      -31
FLT_MIN               2.93873588E-39F
FLT_MIN_10_EXP                   -38
FLT_MAX_EXP                      +32
FLT_MAX               3.40282347E+38F
FLT_MAX_10_EXP                   +38
```

29 **EXAMPLE 2** The following describes floating-point representations that also meet the requirements for single-precision and double-precision numbers in IEC 60559,[29] and the appropriate values in a `<float.h>` header for types **`float`** and **`double`**:

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_IS_IEC_60559                   2
FLT_RADIX                          2
FLT_MANT_DIG                      24
FLT_EPSILON           1.19209290E-07F // decimal constant
FLT_EPSILON                 0X1P-23F // hex constant
FLT_DECIMAL_DIG                    9
FLT_DIG                            6
FLT_MIN_EXP                     -125
FLT_MIN               1.17549435E-38F // decimal constant
FLT_MIN                    0X1P-126F // hex constant
FLT_TRUE_MIN          1.40129846E-45F // decimal constant
FLT_TRUE_MIN               0X1P-149F // hex constant
FLT_HAS_SUBNORM                    1
FLT_MIN_10_EXP                   -37
FLT_MAX_EXP                     +128
FLT_MAX               3.40282347E+38F // decimal constant
FLT_MAX               0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                   +38
DBL_MANT_DIG                      53
DBL_IS_IEC_60559                   2
DBL_EPSILON    2.2204460492503131E-16 // decimal constant
DBL_EPSILON                  0X1P-52 // hex constant
```

[29]The floating-point model in that standard sums powers of *b* from zero, so the values of the exponent limits are one less than shown here.

```
DBL_DECIMAL_DIG                      17
DBL_DIG                              15
DBL_MIN_EXP                       -1021
DBL_MIN      2.2250738585072014E-308 // decimal constant
DBL_MIN                     0X1P-1022 // hex constant
DBL_TRUE_MIN 4.9406564584124654E-324 // decimal constant
DBL_TRUE_MIN                0X1P-1074 // hex constant
DBL_HAS_SUBNORM                       1
DBL_MIN_10_EXP                     -307
DBL_MAX_EXP                       +1024
DBL_MAX      1.7976931348623157E+308 // decimal constant
DBL_MAX       0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP                     +308
```

Forward references: conditional inclusion (6.10.1), predefined macro names (6.10.9), complex arithmetic `<complex.h>` (7.3), extended multibyte and wide character utilities `<wchar.h>` (7.31), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.24), input/output `<stdio.h>` (7.23), mathematics `<math.h>` (7.12), IEC 60559 floating-point arithmetic (Annex F), IEC 60559-compatible complex arithmetic (Annex G).

5.2.4.2.3 Characteristics of decimal floating types in `<float.h>`

1 This subclause specifies macros in `<float.h>` that provide characteristics of decimal floating types (an optional feature) in terms of the model presented in 5.2.4.2.2. An implementation shall provide these macros if and only if defines `__STDC_IEC_60559_DFP__`. The prefixes `DEC32_`, `DEC64_`, and `DEC128_` denote the types `_Decimal32`, `_Decimal64`, and `_Decimal128` respectively.

2 `DEC_EVAL_METHOD` is the decimal floating-point analog of `FLT_EVAL_METHOD` (5.2.4.2.2). Its implementation-defined value characterizes the use of evaluation formats for decimal floating types:

-1 indeterminable;

0 evaluate all operations and constants just to the range and precision of the type;

1 evaluate operations and constants of type `_Decimal32` and `_Decimal64` to the range and precision of the `_Decimal64` type, evaluate `_Decimal128` operations and constants to the range and precision of the `_Decimal128` type;

2 evaluate all operations and constants to the range and precision of the `_Decimal128` type.

3 Each of the decimal signaling NaN macros

```
DEC32_SNAN
DEC64_SNAN
DEC128_SNAN
```

expands to a constant expression of the respective decimal floating type representing a signaling NaN. If an optional unary $+$ or $-$ operator followed by a signaling NaN macro is used for initializing an object of the same type that has static or thread storage duration, the object is initialized with a signaling NaN value.

4 The macro

```
DEC_INFINITY
```

expands to a constant expression of type `_Decimal32` representing positive infinity.

5 The macro

```
DEC_NAN
```

expands to a constant expression of type **_Decimal32** representing a quiet NaN.

6 The integer values given in the following lists shall be replaced by constant expressions suitable for use in conditional expression inclusion preprocessing directives:

— radix of exponent representation, b(=10)

For the standard floating types, this value is implementation-defined and is specified by the macro **FLT_RADIX**. For the decimal floating types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever **FLT_RADIX** appears in a description of a function that has versions that operate on decimal floating types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than **FLT_RADIX**.

— number of digits in the coefficient

```
DEC32_MANT_DIG     7
DEC64_MANT_DIG     16
DEC128_MANT_DIG    34
```

— minimum exponent

```
DEC32_MIN_EXP      -94
DEC64_MIN_EXP      -382
DEC128_MIN_EXP     -6142
```

— maximum exponent

```
DEC32_MAX_EXP      97
DEC64_MAX_EXP      385
DEC128_MAX_EXP     6145
```

— maximum representable finite decimal floating-point number (there are 6, 15 and 33 9's after the decimal points respectively)

```
DEC32_MAX          9.999999E96DF
DEC64_MAX          9.999999999999999E384DD
DEC128_MAX         9.999999999999999999999999999999999E6144DL
```

— the difference between 1 and the least value greater than 1 that is representable in the given floating type

```
DEC32_EPSILON      1E-6DF
DEC64_EPSILON      1E-15DD
DEC128_EPSILON     1E-33DL
```

— minimum normalized positive decimal floating-point number

```
DEC32_MIN          1E-95DF
DEC64_MIN          1E-383DD
DEC128_MIN         1E-6143DL
```

— minimum positive subnormal decimal floating-point number

```
DEC32_TRUE_MIN     0.000001E-95DF
DEC64_TRUE_MIN     0.000000000000001E-383DD
DEC128_TRUE_MIN    0.000000000000000000000000000000001E-6143DL
```

7 For decimal floating-point arithmetic, it is often convenient to consider an alternate equivalent model where the significand is represented with integer rather than fraction digits. With s, b, e, p, and f_k as defined in 5.2.4.2.2, a floating-point number x is defined by the model:

$$x = s \cdot b^{(e-p)} \sum_{k=1}^{p} f_k \cdot b^{(p-k)}$$

8 With b fixed to 10, a decimal floating-point number x is thus:

$$x = s \cdot 10^{(e-p)} \sum_{k=1}^{p} f_k \cdot 10^{(p-k)}$$

The *quantum exponent* is $q = e - p$ and the *coefficient* is $c = f_1 f_2 \cdots f_p$, which is an integer between 0 and $10^{(p-1)}$, inclusive. Thus, $x = s \cdot c \cdot 10^q$ is represented by the triple of integers (s, c, q). The *quantum* of x is 10^q, which is the value of a unit in the last place of the coefficient.

Quantum exponent ranges

Type	`_Decimal32`	`_Decimal64`	`_Decimal128`
Maximum Quantum Exponent (q_{max})	90	369	6111
Minimum Quantum Exponent (q_{min})	-101	-398	-6176

9 For binary floating-point arithmetic following IEC 60559, representations in the model described in 5.2.4.2.2 that have the same numerical value are indistinguishable in the arithmetic. However, for decimal floating-point arithmetic, representations that have the same numerical value but different quantum exponents, e.g., $(+1, 10, -1)$ representing 1.0 and $(+1, 100, -2)$ representing 1.00, are distinguishable. To facilitate exact fixed-point calculation, operation results that are of decimal floating type have a *preferred quantum exponent*, as specified in IEC 60559, which is determined by the quantum exponents of the operands if they have decimal floating types (or by specific rules for conversions from other types). The table below gives rules for determining preferred quantum exponents for results of IEC 60559 operations, and for other operations specified in this document. When exact, these operations produce a result with their preferred quantum exponent, or as close to it as possible within the limitations of the type. When inexact, these operations produce a result with the least possible quantum exponent. For example, the preferred quantum exponent for addition is the minimum of the quantum exponents of the operands. Hence $(+1, 123, -2) + (+1, 4000, -3) = (+1, 5230, -3)$ or $1.23 + 4.000 = 5.230$.

10 The following table shows, for each operation delivering a result in decimal floating-point format, how the preferred quantum exponents of the operands, $Q(\texttt{x})$, $Q(\texttt{y})$, etc., determine the preferred quantum exponent of the operation result, provided the table formula is defined for the arguments. For the cases where the formula is undefined and the function result is $\pm\infty$, the preferred quantum exponent is immaterial because the quantum exponent of $\pm\infty$ is defined to be infinity. For the other cases where the formula is undefined and the function result is finite, the preferred quantum exponent is unspecified.[30]

Preferred quantum exponents

Operation	Preferred quantum exponent of result
`roundeven`, **`round`**, **`trunc`**, **`ceil`**, **`floor`**, **`rint`**, **`nearbyint`**	$\max(Q(\texttt{x}), 0)$
`nextup`, **`nextdown`**, **`nextafter`**, **`nexttoward`**	least possible
`remainder`	$\min(Q(\texttt{x}), Q(\texttt{y}))$

[30] Although unspecified in IEC 60559, a preferred quantum exponent of 0 for these cases would be a reasonable implementation choice.

fmin, **fmax**, **fminimum**, **fmaximum**, **fminimum_mag**, **fmaximum_mag**, **fminimum_num**, **fmaximum_num**, **fminimum_mag_num**, **fmaximum_mag_num**	Q(x) if x gives the result, Q(y) if y gives the result
scalbn, **scalbln**	$Q(\texttt{x}) + \texttt{n}$
ldexp	$Q(\texttt{x}) + \texttt{p}$
logb	0
postfix ++ operator, postfix -- operator, prefix ++ operator, prefix -- operator	`min(Q(x), 0)`
$+$, **d32add**, **d64add**	$\min(Q(\texttt{x}), Q(\texttt{y}))$
$-$, **d32sub**, **d64sub**	$\min(Q(\texttt{x}), Q(\texttt{y}))$
$*$, **d32mul**, **d64mul**	$Q(\texttt{x}) + Q(\texttt{y})$
/, **d32div**, **d64div**	$Q(\texttt{x}) - Q(\texttt{y})$
sqrt, **d32sqrt**, **d64sqrt**	$\lfloor Q(\texttt{x})/2 \rfloor$
fma, **d32fma**, **d64fma**	$\min(Q(\texttt{x}) + Q(\texttt{y}), Q(\texttt{z}))$
conversion from integer type	0
exact conversion from non-decimal floating type	0
inexact conversion from non-decimal floating type	least possible
conversion between decimal floating types	Q(x)
$*$cx returned by **canonicalize**	Q($*$x)
strto, **wcsto**, **scanf**, floating constants of decimal floating type	see 7.24.1.6
-(x), +(x)	Q(x)
fabs	Q(x)
copysign	Q(x)
quantize	Q(y)
quantum	Q(x)
$*$encptr returned by **encodedec**, **encodebin**	Q($*$xptr)
$*$xptr returned by **decodedec**, **decodebin**	Q($*$encptr)
fmod	$\min(Q(\texttt{x}), Q(\texttt{y}))$
fdim	$\min((Q(\texttt{x}), Q(\texttt{y}))$ if $\texttt{x} > \texttt{y}$, 0 if $\texttt{x} \leq \texttt{y}$
cbrt	$\lfloor Q(\texttt{x})/3 \rfloor$
hypot	$\min(Q(\texttt{x}), Q(\texttt{y}))$
pow	$\lfloor \texttt{y} \times Q(\texttt{x}) \rfloor$
modf	Q(value)
$*$iptr returned by **modf**	$\max(Q(\texttt{value}), 0)$
frexp	Q(value) if value $= 0$, –(length of coefficient of value) otherwise
$*$res returned by **setpayload**, **setpayloadsig**	0 if pl does not represent a valid payload, not applicable otherwise (NaN returned)
getpayload	0 if $*$x is a NaN, unspecified otherwise
compoundn	$\lfloor n \times \min(0, Q(x)) \rfloor$
pown	$\lfloor n \times Q(x) \rfloor$
powr	$\lfloor y \times Q(x) \rfloor$
rootn	$\lfloor Q(x)/n \rfloor$
rsqrt	$-\lfloor Q(x)/2 \rfloor$
transcendental functions	0

A function family listed in the table above indicates the functions for all decimal floating types, where the function family is represented by the name of the functions without a suffix. For example, **ceil** indicates the functions **ceild32**, **ceild64**, and **ceild128**.

Forward references: extended multibyte and wide character utilities `<wchar.h>` (7.31), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.24), input/output `<stdio.h>` (7.23), mathematics `<math.h>` (7.12), type-generic mathematics `<tgmath.h>` (7.27), IEC 60559 floating-point arithmetic (Annex F).

6. Language

6.1 Notation

1 In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

{ *expression*$_{\text{opt}}$ **}**

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote:

— a standard attribute, an attribute prefix, or an attribute name;

— an object; a function;

— a tag or a member of a structure, union, or enumeration;

— a typedef name;

— a label name;

— a macro name;

— or, a macro parameter.

The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function.)

3 A label name is the only kind of identifier that has *function scope*. It can be used (in a **`goto`** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a `:` and a statement).

4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function

declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

5 Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. An ordinary identifier that has an underspecified definition has scope that starts when the definition is completed; if the same ordinary identifier declares another entity with a scope that encloses the current block, that declaration is hidden as soon as the inner declarator is completed.[31] Any other identifier has scope that begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.4), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*[32]. There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for:

— an object contains any of the storage-class specifiers **`static`** or **`constexpr`**;

— or, a function contains the storage-class specifier **`static`**,

then the identifier has internal linkage.[33]

4 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible[34], if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **`extern`**. If the declaration of an identifier for an object has file scope and no storage-class specifier or only the specifier **`auto`**, its linkage is external.

6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **`extern`**.

[31]That means, that the outer declaration is not visible for the initializer.
[32]There is no linkage between different identifiers.
[33]A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1.
[34]As specified in 6.2.1, the later declaration might hide the prior declaration.

7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

— *label names* (disambiguated by the syntax of the label declaration and use);

— the *tags* of structures, unions, and enumerations (disambiguated by following any[35] of the keywords **`struct`**, **`union`**, or **`enum`**);

— the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);

— standard attributes and attribute prefixes (disambiguated by the syntax of the attribute specifier and name of the attribute token) (6.7.12);

— the trailing identifier in an attribute prefixed token; each attribute prefix has a separate name space for the implementation-defined attributes that it introduces (disambiguated by the attribute prefix and the trailing identifier token);

— all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the **`goto`** statement (6.8.6.1).

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.24.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address.[36], and retains its last-stored value throughout its lifetime[37] If an object is referred to outside of its lifetime, the behavior is undefined. If a pointer value is used in an evaluation after the object the pointer points to (or just past) reaches the end of its lifetime, the behavior is undefined. The representation of a pointer object becomes indeterminate when the object the pointer points to (or just past) reaches the end of its lifetime.

3 An object whose identifier is declared without the storage-class specifier **`thread_local`**, and either with external or internal linkage or with the storage-class specifier **`static`**, has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier **`thread_local`** has *thread storage duration*. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*, as do some compound literals. The result of attempting to indirectly

[35]There is only one name space for tags even though three are possible.

[36]The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

[37]In the case of a volatile object, the last store need not be explicit in the program.

access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial representation of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the representation of the object becomes indeterminate each time the declaration is reached.

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[38] If the scope is entered recursively, a new instance of the object is created each time. The initial representation of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary lifetime*.[39] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime results in undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. Such an object need not have a unique address.

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), initialization (6.7.10), statements (6.8), effective type (6.5).

6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete*[40] (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information)[41].

2 An object declared as type **`bool`** is large enough to store the values **`false`** and **`true`**.

3 An object declared as type **`char`** is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a **`char`** object, its value is guaranteed to be nonnegative. If any other character is stored in a **`char`** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

4 There are five *standard signed integer types*, designated as **`signed char`**, **`short int`**, **`int`**, **`long int`**, and **`long long int`**. (These and other types may be designated in several additional ways, as described in 6.7.2.)

5 A *bit-precise signed integer type* is designated as **`_BitInt`**`(`*N*`)` where *N* is an integer constant expression that specifies the number of bits that are used to represent the type, including the sign bit. Each value of *N* designates a distinct type.[42].

6 There may also be implementation-defined *extended signed integer types* [43]. The standard signed

[38]Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

[39]The address of such an object is taken implicitly when an array member is accessed.

[40]An incomplete type can only be used when the size of an object of that type is not needed. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. The specification has to be complete before such a function is called or defined.

[41]A type can be incomplete or complete throughout an entire translation unit, or it can change states at different points within a translation unit.

[42]Thus, **`_BitInt`**`(3)` is not the same type as **`_BitInt`**`(4)`.

[43]Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

integer types, bit-precise signed integer types, and extended signed integer types are collectively called *signed integer types* [44].

7 An object declared as type **signed char** occupies the same amount of storage as a "plain" **char** object. A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header `<limits.h>`).

8 For each of the signed integer types, there is a *corresponding* (but different) *unsigned integer type* (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type **bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. In addition to the unsigned integer types that correspond to the bit-precise signed integer types there is the type **unsigned _BitInt**(1), which uses one bit to represent the type. Collectively, **unsigned _BitInt**(1) and the unsigned integer types that correspond to the bit-precise signed integer types are the *bit-precise unsigned integer types*. The standard unsigned integer types, bit-precise unsigned integer types, and extended unsigned integer types are collectively called *unsigned integer types*.[45]

9 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the bit-precise signed integer types and bit-precise unsigned integer types are collectively called the *bit-precise integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

10 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

11 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[46] The range of representable values for the unsigned type is 0 to $2^N - 1$ (inclusive). A computation involving unsigned operands can never produce an overflow, because arithmetic for the unsigned type is performed modulo 2^N.

12 There are three *standard floating types*, designated as **float**, **double**, and **long double**. [47] The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.

13 There are three *decimal floating types*, designated as **_Decimal32**, **_Decimal64**, and **_Decimal128**. Respectively, they have the IEC 60559 formats: decimal32[48], decimal64, and decimal128. Decimal floating types are real floating types. (Decimal floating types are a conditional feature that implementations need not support; see 6.10.9.3.)

14 The standard floating types and the decimal floating types are collectively called the real floating types.

15 There are three *complex types*, designated as **float _Complex**, **double _Complex**, and **long double _Complex**.[49] (Complex types are a conditional feature that implementations need not support; see 6.10.9.3.) The real floating and complex types are collectively called the *floating types*.

16 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword

[44] Any statement in this document about signed integer types also applies to the bit-precise signed integer types and the extended signed integer types, unless otherwise noted.

[45] Any statement in this document about unsigned integer types also applies to the bit-precise unsigned integer types and the extended unsigned integer types, unless otherwise specified.

[46] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

[47] See "future language directions" (6.11.1).

[48] IEC 60559 specifies decimal32 as a data-interchange format that does not require arithmetic support; however, **_Decimal32** is a fully supported arithmetic type.

[49] A specification for imaginary types is in Annex G.

_Complex from the type name.

17 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

18 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

19 **NOTE 1** An implementation can define new keywords that provide alternative ways to designate a basic (or any other) type; this does not violate the requirement that all basic types be different. Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

20 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[50)]

21 An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

22 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. The integer and real floating types are collectively called *real types*.

23 Integer and floating types are collectively called *arithmetic types*. Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

24 The **void** type comprises an empty set of values; it is an incomplete object type that cannot be completed.

25 Any number of *derived types* can be constructed from the object and function types, as follows:

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*". The construction of an array type from an element type is called "array type derivation".

— A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

— A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". The construction of a function type from a return type is called "function type derivation".

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **_Atomic**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.9.3.)

[50)] **CHAR_MIN**, defined in `<limits.h>`, will have one of the values 0 or **SCHAR_MIN**, and this can be used to distinguish the two options. Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either.

These methods of constructing derived types can be applied recursively.

26 Arithmetic types, pointer types, and the **`nullptr_t`** type are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*[51].

27 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

28 A complete type shall have a size that is less than or equal to **`SIZE_MAX`**. A type has *known constant size* if it is complete and is not a variable length array type.

29 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

30 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

31 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type[52], corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[53] An array and its element type are always considered to be identically qualified.[54] Any other derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

32 Further, there is the **`_Atomic`** qualifier. The presence of the **`_Atomic`** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic, qualified, or unqualified type" whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.

33 A pointer to **`void`** shall have the same representation and alignment requirements as a pointer to a character type.[53] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

34 **EXAMPLE 1** The type designated as "`float *`" has type "pointer to **`float`**". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "`float * const`" whereas the type designated as "`const float *`" is not a qualified type — its type is "pointer to const-qualified **`float`**" and is a pointer to a qualified type.

35 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type **`float`**. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

---

[51]Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[52]See 6.7.3 regarding qualified array and function types.

[53]The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

[54]This does not apply to the **`_Atomic`** qualifier. Note that qualifiers do not have any direct effect on the array type itself, but affect conversion rules for pointer types that reference an array type.

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause.

2 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

3 Values stored in unsigned bit-fields and objects of type `unsigned char` shall be represented using a pure binary notation.

4 **NOTE 1** A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems.*) A byte contains `CHAR_BIT` bits, and the values of type `unsigned char` range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

5 Values stored in non-bit-field objects of any other object type are represented using $n \times$ `CHAR_BIT` bits, where n is the size of an object of that type, in bytes. An object that has the value may be copied into an object of type `unsigned char [`n`]` (e.g., by `memcpy`); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

6 Certain object representations need not represent a value of the object type. If such a representation is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[55)] Such a representation is called a non-value representation.

7 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[56)] The object representation of a structure or union object is never a non-value representation, even though the byte range corresponding to a member of the structure or union object may be a non-value representation for that member.

8 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

9 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[57)] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a non-value representation shall not be generated.

10 Loads and stores of objects with atomic types are done with `memory_order_seq_cst` semantics.

Forward references: declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

6.2.6.2 Integer types

1 For unsigned integer types the bits of the object representation shall be divided into two groups: value bits and padding bits. If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified. The number of value bits N is called the *width* of the unsigned integer type. The type `bool` shall have one value bit and `(sizeof(bool)*CHAR_BIT)- 1` padding bits. Otherwise, there need not be any padding bits; `unsigned char` shall not have any

[55)]Thus, an automatic variable can be initialized to a non-value representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[56)]Thus, for example, structure assignment need not copy any padding bits.

[57)]It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. In particular, if `==` is defined for type `T`, then `x == y` does not imply that `memcmp(&x, &y, sizeof(T))== 0`. Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

padding bits.

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. If the corresponding unsigned type has width N, the signed type uses the same number of N bits, its *width*, as value bits and sign bit. $N - 1$ are value bits and the remaining bit is the sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type. If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, it has value $-(2^{N-1})$. There need not be any padding bits; **`signed char`** shall not have any padding bits.

3 The values of any padding bits are unspecified. A valid object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

4 The *precision* of an integer type is the number of value bits.

5 **NOTE 1** Some combinations of padding bits might generate non-value representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a non-value representation other than as part of an exceptional condition such as an integer overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

6 **NOTE 2** The sign representation defined in this document is called *two's complement*. Previous revisions of this document additionally allowed other sign representations.

7 **NOTE 3** For unsigned integer types the width and precision are the same, while for signed integer types the width is one greater than the precision.

6.2.7 Compatible type and composite type

1 Two types are *compatible types* if they are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.[58] Moreover, two complete structure, union, or enumerated types declared with the same tag are compatible if members satisfy the following requirements:

— there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types;

— if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier;

— and, if one member of the pair is declared with a name, the other is declared with the same name.

For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values; if one has a fixed underlying type, then the other shall have a compatible fixed underlying type. For determining type compatibility, anonymous structures and unions are considered a regular member of the containing structure or union type, and the type of an anonymous structure or union is considered compatible to the type of another anonymous structure or union, respectively, if their members fulfill the above requirements.

Furthermore, two structure, union, or enumerated types declared in separate translation units are compatible in the following cases:

— both are declared without tags and they fulfill the requirements above;

— both have the same tag and are completed somewhere in their respective translation units and they fulfill the requirements above;

— both have the same tag and at least one of the two types is not completed in its translation unit.

[58] Two types need not be identical to be compatible.

Otherwise, the structure, union, or enumerated types are incompatible.[59)]

2 All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.

3 A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

— If both types are array types, the following rules are applied:

- If one type is an array of known constant size, the composite type is an array of that size.
- Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
- Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
- Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
- Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

The element type of the composite type is the composite type of the two element types.

— If both types are function types, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

— If one of the types has a standard attribute, the composite type also has that attribute.

These rules apply recursively to the types from which the two types are derived.

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible[60)], if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

5 **EXAMPLE** Given the following two file scope declarations:

```
int f(int (*)(char *), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

Forward references: array declarators (6.7.6.2).

6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the **alignas** keyword.

2 A *fundamental alignment* is a valid alignment less than or equal to `alignof(max_align_t)`. Fundamental alignments shall be supported by the implementation for objects of all storage durations. The alignment requirements of the following types shall be fundamental alignments:

— all atomic, qualified, or unqualified basic types;

[59)]A structure, union, or enumerated type without a tag or an incomplete structure, union or enumerated type is not compatible with any other structure, union or enum type declared in the same translation unit.

[60)]As specified in 6.2.1, the later declaration might hide the prior declaration.

— all atomic, qualified, or unqualified enumerated types;

— all atomic, qualified, or unqualified pointer types;

— all array types whose element type has a fundamental alignment requirement;

— all types specified in Clause 7 as complete object types;

— all structure or union types whose elements have types with fundamental alignment requirements and none of whose elements have an alignment specifier specifying an alignment that is not a fundamental alignment.

3 An *extended alignment* is represented by an alignment greater than `alignof(max_align_t)`. It is implementation-defined whether any extended alignments are supported and the storage durations for which they are supported. A type having an extended alignment requirement is an *over-aligned* type.[61)]

4 Alignments are represented as values of the type `size_t`. Valid alignments include only fundamental alignments, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an `alignof` expression. The types `char`, `signed char`, and `unsigned char` shall have the weakest alignment requirement.

7 Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

6.2.9 Encodings

1 The *literal encoding* is an implementation-defined mapping of the characters of the execution character set to the values in a character constant (6.4.4.4) or string literal (6.4.5). It shall support a mapping from all the basic execution character set values into the implementation-defined encoding. It may contain multibyte character sequences (5.2.1.1).

2 The *wide literal encoding* is an implementation-defined mapping of the characters of the execution character set to the values in a `wchar_t` character constant (6.4.4.4) or a `wchar_t` string literal (6.4.5). It shall support a mapping from all the basic execution character set values into the implementation-defined encoding. The mapping shall produce values identical to the literal encoding for all the basic execution character set values if an implementation does not define `__STDC_MB_MIGHT_NEQ_WC__`. One or more values may map to one or more values of the extended execution character set.

6.3 Conversions

1 Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.

2 Unless explicitly stated otherwise, conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (6.5.4).

61)Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

1 Every integer type has an *integer conversion rank* defined as follows:

— No two signed integer types shall have the same rank, even if they have the same representation.

— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

— The rank of **long long int** shall be greater than the rank of **long int**, which shall be greater than the rank of **int**, which shall be greater than the rank of **short int**, which shall be greater than the rank of **signed char**.

— The rank of a bit-precise signed integer type shall be greater than the rank of any standard integer type with less width or any bit-precise integer type with less width.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width or bit-precise integer type with the same width.

— The rank of any bit-precise integer type relative to an extended integer type of the same width is implementation-defined.

— The rank of **char** shall equal the rank of **signed char** and **unsigned char**.

— The rank of **bool** shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

2 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

— An object or expression with an integer type (other than **int** or **unsigned int**) whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**.

— A bit-field of type **bool**, **int**, **signed int**, or **unsigned int**.

The value from a bit-field of a bit-precise integer type is converted to the corresponding bit-precise integer type. If the original type is not a bit-precise integer type (6.2.5): if an **int** can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an **int**[62]; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*[63]. All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a "plain" **char** can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

[62]E.g., **unsigned _BitInt**(7): 2 is a bit-field that can hold the values 0, 1, 2, 3, and converts to **unsigned _BitInt**(7).

[63]The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, −, and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses.

6.3.1.2 Boolean type

1 When any scalar value is converted to **bool**, the result is **false** if the value is a zero (for arithmetic types), null (for pointer types), or the scalar has type **nullptr_t**; otherwise, the result is **true**.

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than **bool**, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[64)]

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6.3.1.4 Real floating and integer

1 When a finite value of standard floating type is converted to an integer type other than **bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[65)]

2 When a finite value of decimal floating type is converted to an integer type other than **bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the "invalid" floating-point exception shall be raised and the result of the conversion is unspecified.

3 When a value of integer type is converted to a standard floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

4 When a value of integer type is converted to a decimal floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEC 60559.

6.3.1.5 Real floating types

1 When a value of real floating type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged.

2 When a value of real floating type is converted to a standard floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.

3 When a value of real floating type is converted to a decimal floating type, if the value being converted cannot be represented exactly, the result is correctly rounded with exceptions raised as specified in IEC 60559.

4 Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

---

64) The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

65) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. Thus, the range of portable real floating values is $(-1,$ **U***type*_**MAX** $+ 1)$.

6.3.1.6 Complex types

1 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

6.3.1.7 Real and complex

1 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

2 When a value of complex type is converted to a real type other than **bool**,[66] the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

> If one operand has decimal floating type, the other operand shall not have standard floating, complex, or imaginary type.
>
> First, if the type of either operand is **_Decimal128**, the other operand is converted to **_Decimal128**.
>
> Otherwise, if the type of either operand is **_Decimal64**, the other operand is converted to **_Decimal64**.
>
> Otherwise, if the type of either operand is **_Decimal32**, the other operand is converted to **_Decimal32**.
>
> Otherwise, if the corresponding real type of either operand is **long double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **long double**.
>
> Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**.
>
> Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.[67]
>
> Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
> > If both operands have the same type, then no further conversion is needed.
> >
> > Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
> >
> > Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
> >
> > Otherwise, if the type of the operand with signed integer type can represent all the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

---

[66] See 6.3.1.2.

[67] For example, addition of a **double _Complex** and a **float** entails just the conversion of the **float** operand to **double** (and yields a **double _Complex** result).

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

2 The values of floating operands and of the results of floating expressions may be represented in greater range and precision than that required by the type; the types are not changed thereby. See 5.2.4.2.2 regarding evaluation formats.

3 **EXAMPLE 1** One consequence of `_BitInt` being exempt from the integer promotion rules (6.3.1) is that a `_BitInt` operand of a binary operator is not always promoted to an `int` or `unsigned int` as part of the usual arithmetic conversions. Instead, a lower-ranked operand is converted to the higher-rank operand type and the result of the operation is the higher-ranked type.

```
_BitInt(2) a2 = 1;
_BitInt(3) a3 = 2;
_BitInt(33) a33 = 1;
char c = 3;

a2 * a3 /* As part of the multiplication, a2 is converted to
            _BitInt(3) and the result type is _BitInt(3). */
a2 * c /* As part of the multiplication, c is promoted to int,
            a2 is converted to int and the result type is int. */
a33 * c /* As part of the multiplication, c is promoted to int,
            then converted to _BitInt(33) and the result type
            is _BitInt(33). */

void func(_BitInt(8) a1, _BitInt(24) a2) {
      /* Cast one of the operands to 32-bits to guarantee the
         result of the multiplication can contain all possible
         values. */
      _BitInt(32) a3 = a1 * (_BitInt(32))a2;
}
```

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression (with an object type other than **void**) that potentially designates an object;[68] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the **sizeof** operator, or the typeof operators, the unary & operator, the ++ operator, the -- operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the lvalue designates an object of automatic storage duration that could have been declared with the **register** storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

3 Except when it is the operand of the **sizeof** operator, or typeof operators, or the unary & operator,

[68]The name "lvalue" comes originally from the assignment expression `E1 = E2`, in which the left operand `E1` is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this document described as the "value of an expression".

An obvious example of an lvalue is an identifier of an object. As a further example, if `E` is a unary expression that is a pointer to an object, `*E` is an lvalue that designates the object to which `E` points.

or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

4 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator[69], a typeof operator, or the unary & operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*".

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions `<stddef.h>` (7.21), initialization (6.7.10), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and **alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 Pointers

1 A pointer to **void** may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.

2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, such an expression cast to type **void** *, or the predefined constant **nullptr** is called a *null pointer constant*[70]. If a null pointer constant or a value of the type **nullptr_t** (which is necessarily the value **nullptr**) is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might produce an indeterminate representation when stored into an object.[71]

6 Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

7 A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned[72] for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

8 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

---

[69] Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraints in 6.5.3.4

[70] The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.21.

[71] The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

[72] In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

6.3.2.4 `nullptr_t`

1 The type **`nullptr_t`** may be converted to **`bool`** or to a pointer type. The result is **`false`** or a null pointer value, respectively.

2 The type **`nullptr_t`** may be converted to itself.

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.22.1.4), simple assignment (6.5.16.1), the **`nullptr_t`** type (7.21.2).

6.4 Lexical elements

Syntax

1 *token:*

> *keyword*
> *identifier*
> *constant*
> *string-literal*
> *punctuator*

preprocessing-token:

> *header-name*
> *identifier*
> *pp-number*
> *character-constant*
> *string-literal*
> *punctuator*
> each universal-character-name that cannot be one of the above
> each non-white-space character that cannot be one of the above

Constraints

2 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator. A single universal character name shall match one of the other preprocessing token categories.

Semantics

3 A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and both single universal character names as well as single non-white-space characters that do not lexically match the other preprocessing token categories.[73)] If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

4 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. There is one exception to this rule: header name preprocessing tokens are recognized only within **`# include`** and **`#embed`** preprocessing directives, in **`__has_include`** and **`__has_embed`** expressions, as well as in implementation-defined locations within **`#pragma`** directives. In such contexts, a sequence of characters that could be either a header name or a string literal is recognized as the former.

5 **EXAMPLE 1** The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating constant token), whether or not `E` is a macro name.

6 **EXAMPLE 2** The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression.

73)An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.4.3); it cannot occur in source files.

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.4), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5).

6.4.1 Keywords

Syntax

1 *keyword:* one of

alignas	**enum**	**short**	**void**
alignof	**extern**	**signed**	**volatile**
auto	**false**	**sizeof**	**while**
bool	**float**	**static**	**_Atomic**
break	**for**	**static_assert**	**_BitInt**
case	**goto**	**struct**	**_Complex**
char	**if**	**switch**	**_Decimal128**
const	**inline**	**thread_local**	**_Decimal32**
constexpr	**int**	**true**	**_Decimal64**
continue	**long**	**typedef**	**_Generic**
default	**nullptr**	**typeof**	**_Imaginary**
do	**register**	**typeof_unqual**	**_Noreturn**
double	**restrict**	**union**	
else	**return**	**unsigned**	

Semantics

2 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords except in an attribute token, and shall not be used otherwise. The keyword **_Imaginary** is reserved for specifying imaginary types.[74]

3 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.[75]

Keyword	Alternative Spelling
alignas	**_Alignas**
alignof	**_Alignof**
bool	**_Bool**
static_assert	**_Static_assert**
thread_local	**_Thread_local**

The spelling of these keywords, their alternate forms, and of **false** and **true** inside expressions that are subject to the **#** and **##** preprocessing operators is unspecified.[76]

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*

identifier-start
identifier identifier-continue

identifier-start:

nondigit
XID_Start character
universal-character-name of class XID_Start

[74]One possible specification for imaginary types appears in Annex G.
[75]These alternative keywords are obsolescent features and should not be used for new code and development.
[76]The intent of this specification is to allow but not force the implementation of the corresponding feature by means of a predefined macro.

identifier-continue:
digit
nondigit
XID_Continue character
universal-character-name of class XID_Continue

nondigit: one of

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

Semantics

2 An XID_Start character is an implementation-defined character whose corresponding code point in ISO/IEC 10646 has the XID_Start property. An XID_Continue character is an implementation-defined character whose corresponding code point in ISO/IEC 10646 has the XID_Continue property. An identifier is a sequence of one identifier start character followed by 0 or more identifier continue characters, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

3 The character classes XID_Start and XID_Continue are Derived Core Properties as described by UAX #44[77]. Each character and universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 has the XID_Continue property. The initial character (which may be a universal character name) shall designate a character whose encoding in ISO/IEC 10646 has the XID_Start property. An identifier shall conform to Normalization Form C as specified in ISO/IEC 10646. Annex D provides an overview of the conforming identifiers.

4 **NOTE 1** Uppercase and lowercase letters are considered different for all identifiers.

5 **NOTE 2** In translation phase 4 (4), the term identifier also includes those preprocessing tokens (6.4.8) differentiated as keywords (6.4.1) in the later translation phase 7 (7).

6 When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword except in an attribute token.

7 Some identifiers are reserved.

- All identifiers that begin with a double underscore (`__`) or begin with an underscore (`_`) followed by an uppercase letter are reserved for any use, except those identifiers which are lexically identical to keywords.[78]
- All identifiers that begin with an underscore are reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

Other identifiers may be reserved, see 7.1.3.

8 If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), the behavior is undefined.

[77]On systems that cannot accept extended characters in external identifiers, an encoding of the universal-character-name may be used in forming such identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `u` in a universal character name.

[78]This allows a reserved identifier that matches the spelling of a keyword to be used as a macro name by the program.

9 If the program defines a reserved identifier or attribute token described in 6.7.12.1 as a macro name, or removes (with **`#undef`**) any macro definition of an identifier in the first group listed above or attribute token described in 6.7.12.1, the behavior is undefined.

10 Some identifiers may be potentially reserved. A *potentially reserved identifier* is an identifier which is not reserved unless made so by an implementation providing the identifier (7.1.3) but is anticipated to become reserved by an implementation or a future version of this document.

Recommended Practice

11 Implementations are encouraged to issue a diagnostic message when a potentially reserved identifier is declared or defined for any use that is not implementation-compatible (see below) in a context where the potentially reserved identifier may be reserved under a conforming implementation. This brings attention to a potential conflict when porting a program to a future revision of this document.

12 An implementation-compatible use of a potentially reserved identifier is a declaration of an external name where the name is provided by the implementation as an external name and where the declaration declares an object or function with a type that is compatible with the type of the object or function provided by the implementation under that name.

Implementation limits

13 As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.

14 Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

Forward references: universal character names (6.4.3), macro replacement (6.10.4), reserved library identifiers (7.1.3), use of library functions (7.1.4), attributes (6.7.12.1).

6.4.2.2 Predefined identifiers

Semantics

1 The identifier **`__func__`** shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.[79)]

2 This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5.

3 **EXAMPLE** Consider the code fragment:

```
#include <stdio.h>
void myfunc(void)
{
      printf("%s\n", __func__);
      /* ... */
}
```

Each time the function is called, it will print to the standard output stream:

```
myfunc
```

Forward references: function definitions (6.9.1).

[79)]Since the name **`__func__`** is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name **`__func__`**, the behavior is undefined.

6.4.3 Universal character names

Syntax

1 *universal-character-name:*

`\u` *hex-quad*

`\U` *hex-quad hex-quad*

hex-quad:

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

Constraints

2 A universal character name shall not designate a code point where the hexadecimal value is:

— less than 00A0 other than 0024 ($), 0040 (@), or 0060 (`);

— in the range D800 through DFFF inclusive; or

— greater than 10FFFF[80].

Description

3 Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Semantics

4 The universal character name `\U`*nnnnnnnn* designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *nnnnnnnn*.[81] Similarly, the universal character name `\u`*nnnn* designates the character whose four-digit short identifier is *nnnn* (and whose eight-digit short identifier is 0000*nnnn*).

[80]The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, the S-zone (reserved for use by UTF-16), and characters too large to be encoded by ISO/IEC 10646. Disallowed universal character escape sequences can still be specified with hexadecimal and octal escape sequences (6.4.4.4).

[81]Short identifiers for characters were first specified in ISO/IEC 10646–1:1993/Amd 9:1997.

6.4.4 Constants

Syntax

1 *constant:*

integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

Constraints

2 Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

3 Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

1 *integer-constant:*

decimal-constant integer-suffix$_{\text{opt}}$
octal-constant integer-suffix$_{\text{opt}}$
hexadecimal-constant integer-suffix$_{\text{opt}}$
binary-constant integer-suffix$_{\text{opt}}$

decimal-constant:

nonzero-digit
decimal-constant `'`$_{\text{opt}}$ *digit*

octal-constant:

`0`
octal-constant `'`$_{\text{opt}}$ *octal-digit*

hexadecimal-constant:

hexadecimal-prefix hexadecimal-digit-sequence

binary-constant:

binary-prefix binary-digit
binary-constant `'`$_{\text{opt}}$ *binary-digit*

hexadecimal-prefix: one of

`0x 0X`

binary-prefix: one of

`0b 0B`

nonzero-digit: one of
    `1 2 3 4 5 6 7 8 9`

octal-digit: one of
    `0 1 2 3 4 5 6 7`

hexadecimal-digit-sequence:
    hexadecimal-digit
    hexadecimal-digit-sequence **`'`**$_{\text{opt}}$ *hexadecimal-digit*

hexadecimal-digit: one of
    `0 1 2 3 4 5 6 7 8 9`
    `a b c d e f`
    `A B C D E F`

binary-digit: one of
    `0 1`

integer-suffix:
    unsigned-suffix long-suffix$_{\text{opt}}$
    unsigned-suffix long-long-suffix
    unsigned-suffix bit-precise-int-suffix
    long-suffix unsigned-suffix$_{\text{opt}}$
    long-long-suffix unsigned-suffix$_{\text{opt}}$
    bit-precise-int-suffix unsigned-suffix$_{\text{opt}}$

bit-precise-int-suffix: one of
    `wb WB`

unsigned-suffix: one of
    `u U`

long-suffix: one of
    `l L`

long-long-suffix: one of
    `ll LL`

Description

2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type. An optional separating single quote character (`'`) in an integer or floating constant is called a digit separator. Digit separators are ignored when determining the value of the constant.

3 **EXAMPLE** The following integer constants use digit separators; the comment associated with each constant shows the equivalent constant without digit separators.

```
0b11'10'11'01 /* 0b11101101 */
'1'2 /* character constant '1' followed by integer constant 2,
        not the integer constant 12 */
11'22 /* 1122 */
0x'FFFF'FFFF /* invalid hexadecimal constant (' cannot appear after 0x) */
0x1'2'3'4AB'C'D /* 0x1234ABCD */
```

4 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix **`0`** optionally followed by a sequence of the digits **`0`** through **`7`** only. A hexadecimal constant consists of the prefix **`0x`** or **`0X`** followed by a sequence of the decimal digits and the letters **`a`** (or **`A`**) through **`f`** (or **`F`**) with values 10 through 15 respectively. A binary constant consists of the prefix **`0b`** or **`0B`** followed by a sequence of the digits **`0`** or **`1`**.

Semantics

5 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16; that of a binary constant, base 2. The lexically first digit is the most significant.

6 The type of an integer constant is the first of the corresponding list in which its value can be represented.

Suffix	Decimal Constant	Octal, Hexadecimal or Binary Constant
none	**`int`** **`long int`** **`long long int`**	**`int`** **`unsigned int`** **`long int`** **`unsigned long int`** **`long long int`** **`unsigned long long int`**
`u` or **`U`**	**`unsigned int`** **`unsigned long int`** **`unsigned long long int`**	**`unsigned int`** **`unsigned long int`** **`unsigned long long int`**
`l` or **`L`**	**`long int`** **`long long int`**	**`long int`** **`unsigned long int`** **`long long int`** **`unsigned long long int`**
Both **`u`** or **`U`** and **`l`** or **`L`**	**`unsigned long int`** **`unsigned long long int`**	**`unsigned long int`** **`unsigned long long int`**
`ll` or **`LL`**	**`long long int`**	**`long long int`** **`unsigned long long int`**
Both **`u`** or **`U`** and **`ll`** or **`LL`**	**`unsigned long long int`**	**`unsigned long long int`**
`wb` or **`WB`**	**`_BitInt(N)`** where the width **`N`** is the smallest **`N`** greater than 1 which can accommodate the value and the sign bit.	**`_BitInt(N)`** where the width **`N`** is the smallest **`N`** greater than 1 which can accommodate the value and the sign bit.
Both **`u`** or **`U`** and **`wb`** or **`WB`**	**`unsigned _BitInt(N)`** where the width **`N`** is the smallest **`N`** greater than 0 which can accommodate the value.	**`unsigned _BitInt(N)`** where the width **`N`** is the smallest **`N`** greater than 0 which can accommodate the value.

7 If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all the types in the list for the constant are signed, the extended integer type shall be signed. If all the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and

unsigned types, the extended integer type may be signed or unsigned. If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

8 **EXAMPLE 1** The `wb` suffix results in an **`_BitInt`** that includes space for the sign bit even if the value of the constant is positive or was specified in hexadecimal or octal notation.

```
-3wb /* Yields an _BitInt(3) that is then negated; two value
        bits, one sign bit */
-0x3wb /* Yields an _BitInt(3) that is then negated; two value
          bits, one sign bit */
3wb /* Yields an _BitInt(3); two value bits, one sign bit */
3uwb /* Yields an unsigned _BitInt(2) */
-3uwb /* Yields an unsigned _BitInt(2) that is then negated,
         resulting in wraparound */
```

Forward references: preprocessing numbers (6.4.8), numeric conversion functions (7.24.1).

6.4.4.2 Floating constants

Syntax

1 *floating-constant:*
  decimal-floating-constant
  hexadecimal-floating-constant

decimal-floating-constant:
  fractional-constant exponent-part$_{\text{opt}}$ *floating-suffix*$_{\text{opt}}$
  digit-sequence exponent-part floating-suffix$_{\text{opt}}$

hexadecimal-floating-constant:
  hexadecimal-prefix hexadecimal-fractional-constant
    binary-exponent-part floating-suffix$_{\text{opt}}$
  hexadecimal-prefix hexadecimal-digit-sequence
    binary-exponent-part floating-suffix$_{\text{opt}}$

fractional-constant:
  digit-sequence$_{\text{opt}}$ **.** *digit-sequence*
  digit-sequence **.**

exponent-part:
  e *sign*$_{\text{opt}}$ *digit-sequence*
  E *sign*$_{\text{opt}}$ *digit-sequence*

sign: one of
  + -

digit-sequence:
  digit
  digit-sequence **'**$_{\text{opt}}$ *digit*

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{\text{opt}}$ **.** *hexadecimal-digit-sequence*
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{\text{opt}}$ *digit-sequence*
P *sign*$_{\text{opt}}$ *digit-sequence*

floating-suffix: one of
f l F L df dd dl DF DD DL

Constraints

2 A floating suffix **df**, **dd**, **dl**, **DF**, **DD**, or **DL** shall not be used in a hexadecimal floating constant.

Description

3 A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (**.**), followed by a digit sequence representing the fraction part. Digit separators (6.4.4.1) are ignored when determining the value of the constant. The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

Semantics

4 The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when **FLT_RADIX** is a power of 2, the result is correctly rounded.

5 An unsuffixed floating constant has type **double**. If suffixed by a floating suffix it has a type according to the following table:

Suffixes for floating constants

Suffix	Type
f, **F**	**float**
l, **L**	**long double**
df, **DF**	**_Decimal32**
dd, **DD**	**_Decimal64**
dl, **DL**	**_Decimal128**

6 The values of floating constants may be represented in greater range and precision than that required by the type (determined by the suffix); the types are not changed thereby. See 5.2.4.2.2 regarding evaluation formats. [82)]

[82)]Hexadecimal floating constants can be used to obtain exact values in the semantic type that are independent of the

7 Floating constants of decimal floating type that have the same numerical value but different quantum exponents have distinguishable internal representations. The value shall be correctly rounded as specified in IEC 60559. The coefficient c and the quantum exponent q of a finite converted decimal floating-point number (see 5.2.4.2.3) are determined as follows:

— q is set to the value of *sign*$_{opt}$ *digit-sequence* in the exponent part, if any, or to 0, otherwise.

— If there is a fractional constant, q is decreased by the number of digits to the right of the period and the period is removed to form a digit sequence.

— c is set to the value of the digit sequence (after any period has been removed).

— Rounding required because of insufficient precision or range in the type of the result will round c to the full precision available in the type, and will adjust q accordingly within the limits of the type, provided the rounding does not yield an infinity (in which case the result is an appropriately signed internal representation of infinity). If the full precision of the type would require q to be smaller than the minimum for the type, then q is pinned at the minimum and c is adjusted through the subnormal range accordingly, perhaps to zero.

8 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. All floating constants of the same source form [83] shall convert to the same internal format with the same value.

9 **EXAMPLE** Following are floating constants of type `_Decimal64` and their values as triples (s, c, q). Note that for `_Decimal64`, the precision (maximum coefficient length) is 16 and the quantum exponent range is $-398 \le q \le 369$.

`0.dd`	$(+1, 0, 0)$
`0.00dd`	$(+1, 0, -2)$
`123.dd`	$(+1, 123, 0)$
`1.23E3dd`	$(+1, 123, 1)$
`1.23E+3dd`	$(+1, 123, 1)$
`12.3E+7dd`	$(+1, 123, 6)$
`12.0dd`	$(+1, 120, -1)$
`12.3dd`	$(+1, 123, -1)$
`0.00123dd`	$(+1, 123, -5)$
`1.23E-12dd`	$(+1, 123, -14)$
`1234.5E-4dd`	$(+1, 12345, -5)$
`0E+7dd`	$(+1, 0, 7)$
`12345678901234567890.dd`	$(+1, 1234567890123457, 4)$ assuming default rounding and **`DEC_EVAL_METHOD`** is 0 or 1[84]
`1234E-400dd`	$(+1, 12, -398)$ assuming default rounding and **`DEC_EVAL_METHOD`** is 0 or 1
`1234E-402dd`	$(+1, 0, -398)$ assuming default rounding and **`DEC_EVAL_METHOD`** is 0 or 1
`1000.dd`	$(+1, 1000, 0)$
`.0001dd`	$(+1, 1, -4)$
`1000.e0dd`	$(+1, 1000, 0)$
`.0001e0dd`	$(+1, 1, -4)$
`1000.0dd`	$(+1, 10000, -1)$
`0.0001dd`	$(+1, 1, -4)$
`1000.00dd`	$(+1, 100000, -2)$
`00.0001dd`	$(+1, 1, -4)$

evaluation format. Casts produce values in the semantic type, though depend on the rounding mode and may raise the inexact floating-point exception.

[83] `1.23`, `1.230`, `123e-2`, `123e-02`, and `1.23L` are all different source forms and thus need not convert to the same internal format and value.

`001000.dd`	$(+1, 1000, 0)$
`001000.0dd`	$(+1, 10000, -1)$
`001000.00dd`	$(+1, 100000, -2)$
`00.00dd`	$(+1, 0, -2)$
`00.dd`	$(+1, 0, 0)$
`.00dd`	$(+1, 0, -2)$
`00.00e-5dd`	$(+1, 0, -7)$
`00.e-5dd`	$(+1, 0, -5)$
`.00e-5dd`	$(+1, 0, -7)$

Recommended practice

10 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.

11 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as **`strtod`**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding. [85)]

12 **NOTE 1** Floating constants do not include a sign and are negated by the unary − operator (6.5.3.3) which negates the rounded value of the constant. In contrast, the numeric conversion functions in the **`strto`** family (7.24.1.5, 7.24.1.6) may include the sign as part of the input value and convert and round the negated input; Annex F requires this behavior. Negating before rounding and negating after rounding might yield different results, depending on the rounding direction and whether the results are correctly rounded. For example, the results are the same when both are correctly rounded using rounding to nearest or rounding toward zero, but the results are different when they are inexact and correctly rounded using rounding toward positive infinity or rounding toward negative infinity.

Conversions yielding exact results require no rounding, so are not affected by the order of negating and rounding. For types with radix 10, decimal floating constants expressed within the precision and range of the evaluation format convert exactly. For types whose radix is a power of 2, hexadecimal floating constants expressed within the precision and range of the evaluation format convert exactly.

Forward references: preprocessing numbers (6.4.8), numeric conversion functions (7.24.1), the **`strto`** function family (7.24.1.5, 7.24.1.6).

6.4.4.3 Enumeration constants

Syntax

1 *enumeration-constant:*
        identifier

Semantics

2 An identifier declared as an enumeration constant for an enumeration without a fixed underlying type has either type **`int`** or the enumerated type, as defined in 6.7.2.2. An identifier declared as an enumeration constant for an enumeration with a fixed underlying type has the associated enumeration type.

3 An enumeration constant may be used in an expression (or constant expression) wherever a value of an integer type may be used.

Forward references: enumeration specifiers (6.7.2.2).

6.4.4.4 Character constants

Syntax

1 *character-constant:*
        encoding-prefix$_{\text{opt}}$ **'** *c-char-sequence* **'**

[84)] That is, assuming the default translation rounding-direction mode is not changed by an **`FENV_DEC_ROUND`** pragma (7.6.3).
[85)] The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.24.1.5).

encoding-prefix: one of
`u8`
`u` `U` `L`

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except
the single-quote `'`, backslash `\`, or new-line character
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

simple-escape-sequence: one of
`\' \" \? \\`
`\a \b \f \n \r \t \v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

Description

2 An *integer character constant* is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`. A *UTF-8 character constant* is the same, except prefixed by `u8`. A ***`wchar_t`*** *character constant* is prefixed by the letter `L`. A *UTF-16 character constant* is prefixed by the letter `u`. A *UTF-32 character constant* is prefixed by the letter `U`. Collectively, **`wchar_t`**, UTF-16, and UTF-32 character constants are called *wide character constants*. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

3 The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences:

single quote `'`	`\'`
double quote `"`	`\"`
question mark `?`	`\?`
backslash `\`	`\\`
octal character	`\`*octal digits*
hexadecimal character	`\x` *hexadecimal digits*

4 The double-quote `"` and question-mark `?` are representable either by themselves or by the escape sequences `\"` and `\?`, respectively, but the single-quote `'` and the backslash `\` shall be represented, respectively, by the escape sequences `\'` and `\\`.

5 The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

6 The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

8 In addition, characters not in the basic character set are representable by universal character names and certain non-graphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[86]

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

Prefix	Corresponding Type
none	**`unsigned char`**
`u8`	**`char8_t`**
`L`	the unsigned type corresponding to **`wchar_t`**
`u`	**`char16_t`**
`U`	**`char32_t`**

10 A UTF-8, UTF-16, or UTF-32 character constant shall not contain more than one character.[87] The value shall be representable with a single UTF-8, UTF-16, or UTF-32 code unit, respectively.

Semantics

11 An integer character constant has type **`int`**. The value of an integer character constant containing a single character that maps to a single value in the literal encoding (6.2.9) is the numerical value of the representation of the mapped character in the literal encoding interpreted as an integer. The value of an integer character constant containing more than one character (e.g., `'ab'`), or containing a character or escape sequence that does not map to a single value in the literal encoding, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **`char`** whose value is that of the single character or escape sequence is converted to type **`int`**.

12 A UTF-8 character constant has type **`char8_t`**. If the UTF-8 character constant is not produced through a hexadecimal or octal escape sequence, the value of a UTF-8 character constant is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-8 code unit. Otherwise, the value of the UTF-8 character constant is the numeric value specified in the hexadecimal or octal escape sequence.

13 A UTF-16 character constant has type **`char16_t`** which is an unsigned integer types defined in the `<uchar.h>` header. If the UTF-16 character constant is not produced through a hexadecimal or octal escape sequence, the value of a UTF-16 character constant is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-16 code unit. Otherwise, the value of the UTF-16 character constant is the numeric value specified in the hexadecimal or octal escape sequence.

14 A UTF-32 character constant has type **`char32_t`** which is an unsigned integer types defined in the

[86] The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See "future language directions" (6.11.4).

[87] For example `u8'ab'` violates this constraint.

<uchar.h> header. If the UTF-32 character constant is not produced through a hexadecimal or octal escape sequence, the value of a UTF-32 character constant is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-32 code unit. Otherwise, the value of the UTF-32 character constant is the numeric value specified in the hexadecimal or octal escape sequence.

15 A **wchar_t** character constant prefixed by the letter L has type **wchar_t**, an integer type defined in the <stddef.h> header. The value of a **wchar_t** character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character in the implementation-defined wide literal encoding (6.2.9). The value of a **wchar_t** character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

16 **EXAMPLE 1** The construction '\0' is commonly used to represent the null character.

17 **EXAMPLE 2** Consider implementations that use eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant '\xFF' has the value -1; if type **char** has the same range of values as **unsigned char**, the character constant '\xFF' has the value $+255$.

18 **EXAMPLE 3** Even if eight bits are used for objects that have type **char**, the construction '\x123' specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are '\x12' and '3', the construction '\0223' can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

19 **EXAMPLE 4** Even if 12 or more bits are used for objects that have type **wchar_t**, the construction L'\1234' specifies the implementation-defined value that results from the combination of the values 0123 and '4'.

Forward references: common definitions <stddef.h> (7.21), the **mbtowc** function (7.24.7.2), Unicode utilities <uchar.h> (7.30).

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:*

```
false
true
nullptr
```

Description

2 Some keywords represent constants of a specific value and type.

3 The keywords **false** and **true** are constants of type **bool** with a value of 0 for **false** and 1 for **true**[88].

4 The keyword **nullptr** represents a null pointer constant. Details of its type are described in 7.21.2.

6.4.5 String literals

Syntax

1 *string-literal:*

encoding-prefix$_{opt}$ " *s-char-sequence*$_{opt}$ "

[88]The constants **false** and **true** promote to type **int**, see 6.3.1.1. When used for arithmetic, in translation phase 4, they are signed values and the result of such arithmetic is consistent with the results of later translation phases.

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote `"`, backslash `\`, or new-line character
escape-sequence

Constraints

2 If a sequence of adjacent string literal tokens includes prefixed string literal tokens, the prefixed tokens shall all have the same prefix.

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`. A *UTF-8 string literal* is the same, except prefixed by **`u8`**. A ***`wchar_t`*** *string literal* is the same, except prefixed by `L`. A *UTF-16 string literal* is the same, except prefixed by `u`. A *UTF-32 string literal* is the same, except prefixed by `U`. Collectively, **`wchar_t`**, UTF-16, and UTF-32 string literals are called *wide string literals*.

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF-8 string literal) or a wide character constant (for a wide string literal), except that the single-quote `'` is representable either by itself or by the escape sequence `\'`, but the double-quote `"` shall be represented by the escape sequence `\"`.

Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals. [89] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **`char`**, and are initialized with the individual bytes of the multibyte character sequence corresponding to the literal encoding (6.2.9). For UTF-8 string literals, the array elements have type **`char8_t`**, and are initialized with the characters of the multibyte character sequence, as encoded in UTF-8. For wide string literals prefixed by the letter **`L`**, the array elements have type **`wchar_t`** and are initialized with the sequence of wide characters corresponding to the wide literal encoding. For wide string literals prefixed by the letter **`u`** or **`U`**, the array elements have type **`char16_t`** or **`char32_t`**, respectively, and are initialized sequence of wide characters corresponding to UTF-16 and UTF-32 encoded text, respectively. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined. Any hexadecimal escape sequence or octal escape sequence specified in a `u8`, `u`, or `U` string specifies a single **`char8_t`**, **`char16_t`**, or **`char32_t`** value and may result in the full character sequence not being valid UTF-8, UTF-16, or UTF-32.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

8 **EXAMPLE 1** This pair of adjacent character string literals

```
"\x12" "3"
```

[89] A string literal might not be a string (see 7.1.1), because a null character can be embedded in it by a `\0` escape sequence.

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 **EXAMPLE 2** Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions `<stddef.h>` (7.21), the **`mbstowcs`** function (7.24.8.1), Unicode utilities `<uchar.h>` (7.30).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

```
[  ]  (  )  {  }  .  ->
++  --  &  *  +  -  ~  !
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
?  :  ::  ;  ...
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #  ##
<:  :>  <%  %>  %:  %:%:
```

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

3 In all aspects of the language, the six tokens[90)]

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

```
[  ]  {  }  #  ##
```

except for their spelling.[91)]

[90)]These tokens are sometimes called "digraphs".

[91)]Thus `[` and `<:` behave differently when "stringized" (see 6.10.4.2), but can otherwise be freely interchanged.

Forward references: expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8).

6.4.7 Header names

Syntax

1 *header-name:*

  < *h-char-sequence* **>**
  " *q-char-sequence* **"**

h-char-sequence:

  h-char
  h-char-sequence h-char

h-char:

  any member of the source character set except
    the new-line character and >

q-char-sequence:

  q-char
  q-char-sequence q-char

q-char:

  any member of the source character set except
    the new-line character and **"**

Semantics

2 The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2.

3 If the characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the < and > delimiters, the behavior is undefined. Similarly, if the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, the behavior is undefined.[92] Header name preprocessing tokens are recognized only within **`#include`** preprocessing directives and in implementation-defined locations within **`#pragma`** directives.[93]

4 **EXAMPLE** The following sequence of characters:

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a { on the left and a } on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

[92]Thus, sequences of characters that resemble escape sequences cause undefined behavior.

[93]For an example of a header name preprocessing token used in a **`#pragma`** directive, see 6.10.10.

Forward references: source file inclusion (6.10.2).

6.4.8 Preprocessing numbers

Syntax

1 *pp-number:*

digit
. *digit*
pp-number identifier-continue
pp-number **'** *digit*
pp-number **'** *nondigit*
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

Description

2 A preprocessing number begins with a digit optionally preceded by a period (`.`) and may be followed by valid identifier characters and the character sequences **e+**, **e-**, **E+**, **E-**, **p+**, **p-**, **P+**, or **P-**.

3 Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

4 A preprocessing number does not have type or a value; it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

6.4.9 Comments

1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[94)]

2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

3 **EXAMPLE**

```
"a//b"                  // four-character string literal
#include "//e"          // undefined behavior
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();          // syntax error, not comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

[94)]Thus, `/* ... */` comments do not nest.

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[95)]

3 The grouping of operators and operands is indicated by the syntax.[96)] Except as specified later, side effects and value computations of subexpressions are unsequenced.[97)]

4 Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[98)] If a value is stored into an object having no declared type through an lvalue having a type that is not a non-atomic character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using **memcpy** or **memmove**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[99)]

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

— a type that is the signed or unsigned type corresponding to the effective type of the object,

[95)]This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

[96)]The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses () (6.5.1), subscripting brackets [] (6.5.2.1), function-call parentheses () (6.5.2.2), and the conditional operator ?: (6.5.15).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

[97)]In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

[98)]Allocated objects have no declared type.

[99)]The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

8 A floating expression may be *contracted*, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[100] The `FP_CONTRACT` pragma in `<math.h>` provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[101]

9 Operators involving decimal floating types are evaluated according to the semantics of IEC 60559, including production of results with the preferred quantum exponent as specified in IEC 60559.

Forward references: the `FP_CONTRACT` pragma (7.12.2), copying functions (7.26.2).

6.5.1 Primary expressions

Syntax

1 *primary-expression:*
  identifier
  constant
  string-literal
  (*expression* **)**
  generic-selection

Constraints

The identifier in an identifier primary expression shall have a visible declaration as an ordinary identifier that declares an object or a function.[102]

Semantics

2 An identifier primary expression designating an object is an lvalue. An identifier primary expression designating a function is a function designator.

3 A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4 A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

5 A *parenthesized expression* is a primary expression. Its type, value, and semantics are identical to those of the unparenthesized expression.

6 A generic selection is a primary expression. Its type, value, and semantics depend on the selected generic association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax

1 *generic-selection:*
  `_Generic` **(** *assignment-expression* **,** *generic-assoc-list* **)**
generic-assoc-list:
  generic-association

[100]The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions.

[101]This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

[102]An identifier designating an enumeration constant is a primary expression through the constant production, not the identifier production.

generic-assoc-list **,** *generic-association*

generic-association:

type-name **:** *assignment-expression*

`default` **:** *assignment-expression*

Constraints

2 A generic selection shall have no more than one **`default`** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type. No two generic associations in the same generic selection shall specify compatible types. The type of the controlling expression is the type of the expression as if it had undergone an lvalue conversion,[103] array to pointer conversion, or function to pointer conversion. That type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no **`default`** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

4 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, or a void expression if its result expression is, respectively, an lvalue, a function designator, or a void expression.

5 **EXAMPLE** A **`cbrt`** type-generic macro could be implemented as follows:

```
#define cbrt(X) _Generic((X),                   \
                        long double: cbrtl,     \
                        default: cbrt,          \
                        float: cbrtf            \
                        )(X)
```

7.27 shows how such a macro could be implemented with the required rounding properties.

6.5.2 Postfix operators

Syntax

1 *postfix-expression:*

primary-expression

postfix-expression **[** *expression* **]**

postfix-expression **(** *argument-expression-list*$_{opt}$ **)**

postfix-expression **.** *identifier*

postfix-expression **->** *identifier*

postfix-expression **++**

postfix-expression **--**

compound-literal

argument-expression-list:

assignment-expression

argument-expression-list **,** *assignment-expression*

103) An lvalue conversion drops type qualifiers.

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type "pointer to complete object *type*", the other expression shall have integer type, and the result has type "*type*".

Semantics

2 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary $+$ operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2` -th element of `E1` (counting from zero).

3 Successive subscript operators designate an element of a multidimensional array object. If `E` is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary $*$ operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4 **EXAMPLE** Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x` is a 3×5 array of objects of type **`int`**; more precisely, `x` is an array of three element objects, each of which is an array of five objects of type **`int`**. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five objects of type **`int`**. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five **`int`** objects. The results are added and indirection is applied to yield an array of five objects of type **`int`**. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the objects of type **`int`**, so `x[i][j]` yields an **`int`**.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[104)] shall have type pointer to function returning **`void`** or returning a complete object type other than an array type.

2 The number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

4 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[105)]

5 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **`void`**.

104) Most often, this is the result of converting an identifier that is a function designator.

105) A function can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.7.6.3.

6 The arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter, if present. The integer promotions are performed on each trailing argument, and trailing arguments that have type **`float`** are promoted to **`double`**. These are called the *default argument promotions*. No other conversions are performed implicitly.

7 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

8 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[106)]

9 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

10 **EXAMPLE** In the function call

```
    (*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.16.1).

6.5.2.3 Structure and union members

Constraints

1 The first operand of the `.` operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2 The first operand of the `->` operator shall have type "pointer to atomic, qualified, or unqualified structure" or "pointer to atomic, qualified, or unqualified union", and the second operand shall name a member of the type pointed to.

Semantics

3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member,[107)] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

4 A postfix expression followed by the `->` operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.[108)] If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

5 Accessing a member of an atomic structure or union object results in undefined behavior.[109)]

6 One special guarantee is made to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a *common initial*

[106)] In other words, function executions do not interleave with each other.

[107)] If the member used to read the contents of a union object is not the same as the member last used to store a value in the object the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called type punning). This might be a non-value representation.

[108)] If `&E` is a valid pointer expression (where `&` is the address of operator, which generates a pointer to its operand), the expression `(&E)->MOS` is the same as `E.MOS`.

[109)] For example, a data race would occur if access to the entire structure or union in one thread conflicts with access to a member from another thread, where at least one access is a modification. Members can be safely accessed using a non-atomic object which is assigned to or from the atomic object.

sequence if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

7 **EXAMPLE 1** If `f` is a function returning a structure or union, and `x` is a member of that structure or union, `f().x` is a valid postfix expression but is not an lvalue.

8 **EXAMPLE 2** In:

```
struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;
```

the various members have the types:

```
s.i     int
s.ci    const int
cs.i    const int
cs.ci   const int
vs.i    volatile int
vs.ci   volatile const int
```

9 **EXAMPLE 3** The following is a valid fragment:

```
union {
      struct {
            int    alltypes;
      } n;
      struct {
            int    type;
            int    intnode;
      } ni;
      struct {
            int    type;
            double doublenode;
      } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
      if (sin(u.nf.doublenode) == 0.0)
            /* ... */
```

The following is not a valid fragment (because the union type is not visible within function `f`):

```
struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
      if (p1->m  <  0)
            p2->m = -p2->m;
      return p1->m;
}
int g()
{
      union {
            struct t1 s1;
            struct t2 s2;
      } u;
      /* ... */
      return f(&u.s1, &u.s2);
```

```
    }
```

Forward references: address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1).

6.5.2.4 Postfix increment and decrement operators

Constraints

1 The operand of the postfix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

2 The result of the postfix ++ operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately sequenced function call, the operation of postfix ++ is a single evaluation. Postfix ++ on an object with atomic type is a read-modify-write operation with `memory_order_seq_cst` memory order semantics.[110)]

3 The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.2.5 Compound literals

Syntax

1 *compound-literal:*

  (*storage-class-specifiers*$_{\text{opt}}$ *type-name* **)** *braced-initializer*

storage-class-specifiers:

  storage-class-specifier

  storage-class-specifiers storage-class-specifier

Constraints

2 The type name shall specify a complete object type or an array of unknown size, but not a variable length array type.

3 All the constraints for initializer lists in 6.7.10 also apply to compound literals.

4 If the compound literal is evaluated outside the body of a function and outside of any parameter list, it is associated with file scope; otherwise, it is associated with the enclosing block. Depending on this association, the storage-class specifiers `SC` (possibly empty)[111)], type name `T`, and initializer list, if any, shall be such that they are valid specifiers for an object definition in file scope or block scope,

[110)] Where a pointer to an atomic object can be formed and `E` has integer type, `E++` is equivalent to the following code sequence where *T* is the type of `E`:

```
    T *addr = &E;
    T old = *addr;
    T new;
    do {
        new = old + 1;
    } while (!atomic_compare_exchange_strong(addr, &old, new));
```

with `old` being the result of the operation.

Special care is necessary if `E` has floating type; see 6.5.16.2.

[111)] If the storage-class specifiers contain the same storage-class specifier more than once, the following constraint is violated.

respectively, of the following form,

```
    SC typeof(T) ID = { IL };
```

where ID is an identifier that is unique for the whole program and where IL is a (possibly empty) initializer list with nested structure, designators, values and types as the initializer list of the compound literal. All the constraints for storage class specifiers in 6.7.1 also apply correspondingly to compound literals.

Semantics

5 A *compound literal* provides an unnamed object whose value, type, storage duration and other properties are as if given by the definition syntax in the constraints; if the storage duration is automatic, the lifetime of the instance of the unnamed object is the current execution of the enclosing block.[112)] If the storage-class specifiers contain other specifiers than **constexpr**, **static**, **register**, or **thread_local** the behavior is undefined.

6 The value of the compound literal is that of an lvalue corresponding to the unnamed object.

7 All the semantic rules for initializer lists in 6.7.10 also apply to compound literals.[113)]

8 **EXAMPLE 1** Consider the following 2 functions:

```
int f(int*);
int g(char * para[f((int[27]){ 0, })]) {
    /* ... */
    return 0;
}
```

Here, each call to g creates an unnamed object of type int[27] to determine the variably-modified type of para for the duration of the call. During that determination, a pointer to the object is passed into a call to the function f. If a pointer to the object is kept by f, access to that object is possible during the whole execution of the call to g. The lifetime of the object ends with the end of the call to g; for any access after that, the behavior is undefined.

9 String literals, and compound literals with const-qualified types, need not designate distinct objects.[114)]

10 **EXAMPLE 2** The file scope definition

```
    int *p = (int []){2, 4};
```

initializes p to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

11 **EXAMPLE 3** In contrast, in

```
    void f(void)
    {
        int *p;
        /*...*/
        p = (int [2]){*p};
        /*...*/
    }
```

p is assigned the address of the first element of an array of two ints, the first having the value

[112)] Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or **void** only, and the result of a cast expression is not an lvalue.

[113)] For example, subobjects without explicit initializers are initialized to zero.

[114)] This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

previously pointed to by p and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

12 **EXAMPLE 4** Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
      (struct point){.x=3, .y=4});
```

Or, if drawline instead expected pointers to **struct** point:

```
drawline(&(struct point){.x=1, .y=1},
      &(struct point){.x=3, .y=4});
```

13 **EXAMPLE 5** A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

14 **EXAMPLE 6** The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

15 **EXAMPLE 7** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

16 **EXAMPLE 8** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object endless_zeros below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

17 **EXAMPLE 9** Each compound literal creates only a single object in a given scope:

```
struct s { int i; };

int f (void)
{
      struct s *p = 0, *q;
      int j = 0;

again:
      q = p, p = &((struct s){ j++ });
      if (j  <  2) goto again;

      return p == q && q->i == 1;
}
```

The function `f()` always returns the value 1.

18 Note that if an iteration statement were used instead of an explicit **`goto`** and a label, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have indeterminate representation, which would result in undefined behavior.

Forward references: type names (6.7.7), initialization (6.7.10).

6.5.3 Unary operators

Syntax

1 *unary-expression:*

> *postfix-expression*
> **`++`** *unary-expression*
> **`--`** *unary-expression*
> *unary-operator cast-expression*
> **`sizeof`** *unary-expression*
> **`sizeof (`** *type-name* **`)`**
> **`alignof (`** *type-name* **`)`**

unary-operator: one of

> **`& * + - ~ !`**

6.5.3.1 Prefix increment and decrement operators

Constraints

1 The operand of the prefix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

2 The value of the operand of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`, where the value `1` is of the appropriate type. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix `--` operator is analogous to the prefix `++` operator, except that the value of the operand is decremented.

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary `&` operator shall be either a function designator, the result of a `[]` or unary `*` operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **`register`** storage-class specifier.

2 The operand of the unary `*` operator shall have pointer type.

Semantics

3 The unary `&` operator yields the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary `*` operator, neither that operator nor the `&` operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary `*` that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a $+$ operator. Otherwise, the result is a pointer to the object or function designated by its operand.

4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function

designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.[115)]

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

6.5.3.3 Unary arithmetic operators

Constraints

1 The operand of the unary + or − operator shall have arithmetic type; of the ~ operator, integer type; of the `!` operator, scalar type.

Semantics

2 The result of the unary + operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

3 The result of the unary − operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

4 The result of the ~ operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.

6.5.3.4 The `sizeof` and `alignof` operators

Constraints

1 The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The **`alignof`** operator shall not be applied to a function type or an incomplete type.

Semantics

2 The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

3 The **`alignof`** operator yields the alignment requirement of its operand type. The operand is not evaluated and the result is an integer constant expression. When applied to an array type, the result is the alignment requirement of the element type.

4 When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[116)] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is **`size_t`**, defined in `<stddef.h>` (and other headers).

6 **EXAMPLE 1** A principal use of the **`sizeof`** operator is in communication with routines such as

---

[115)] Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `*&E` is a function designator or an lvalue equal to `E`. If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

[116)] When applied to a parameter declared to have array or function type, the **`sizeof`** operator yields the size of the adjusted (pointer) type (see 6.9.1).

storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the `alloc` function presumably ensures that its return value is aligned suitably for conversion to a pointer to **double**.

7 **EXAMPLE 2** Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

8 **EXAMPLE 3** In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
      char b[n+3];       // variable length array
      return sizeof b;   // execution time sizeof
}

int main(void)
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

Forward references: common definitions `<stddef.h>` (7.21), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.7), array declarators (6.7.6.2).

6.5.4 Cast operators

Syntax

1 *cast-expression:*
  unary-expression
  (*type-name* **)** *cast-expression*

Constraints

2 Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type.

3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

4 A pointer type shall not be converted to any floating type. A floating type shall not be converted to any pointer type. The type **nullptr_t** shall not be converted to any type other than **void**, **bool** or a pointer type. No type other than **nullptr_t** shall be converted to **nullptr_t**.

Semantics

5 Preceding an expression by a parenthesized type name converts the value of the expression to the unqualified version of the named type. This construction is called a *cast*[117]. A cast that specifies no conversion has no effect on the type or value of an expression.

[117] A cast does not yield an lvalue.

6 If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

Forward references: equality operators (6.5.9), function declarators (6.7.6.3), simple assignment (6.5.16.1), type names (6.7.7).

6.5.5 Multiplicative operators

Syntax

1 *multiplicative-expression:*
cast-expression
multiplicative-expression **`*`** *cast-expression*
multiplicative-expression **`/`** *cast-expression*
multiplicative-expression **`%`** *cast-expression*

Constraints

2 Each of the operands shall have arithmetic type. The operands of the `%` operator shall have integer type.

3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

4 The usual arithmetic conversions are performed on the operands.

5 The result of the binary `*` operator is the product of the operands.

6 The result of the `/` operator is the quotient from the division of the first operand by the second; the result of the `%` operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

7 When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[118] If the quotient `a/b` is representable, the expression `(a/b)*b + a%b` shall equal `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

6.5.6 Additive operators

Syntax

1 *additive-expression:*
multiplicative-expression
additive-expression **`+`** *multiplicative-expression*
additive-expression **`-`** *multiplicative-expression*

Constraints

2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3 For subtraction, one of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible complete object types; or

— the left operand is a pointer to a complete object type and the right operand has integer type.

[118]This is often called “truncation toward zero”.

(Decrementing is equivalent to subtracting 1.)

4 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

5 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

6 The result of the binary $+$ operator is the sum of the operands.

7 The result of the binary $-$ operator is the difference resulting from the subtraction of the second operand from the first.

8 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

9 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression `P` points to the i-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If the pointer operand and the result do not point to elements of the same array object or one past the last element of the array object, the behavior is undefined. If the addition or subtraction produces an overflow, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary $*$ operator that is evaluated.

10 When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **`ptrdiff_t`** defined in the `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the i-th and j-th elements of an array object, the expression `(P)-(Q)` has the value $i-j$ provided the value fits in an object of type **`ptrdiff_t`**. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.[119)]

11 **EXAMPLE** Pointer arithmetic is well defined with pointers to variable length array types.

```
{
        int n = 4, m = 3;
        int a[n][m];
        int (*p)[m] = a;  // p == &a[0]
        p += 1;           // p == &a[1]
        (*p)[2] = 99;     // a[1][2] == 99
        n = p - a;        // n == 1
}
```

[119)] Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which can overlap another object in the program) just after the end of the object to satisfy the "one past the last element" requirements.

12 If array `a` in the above example were declared to be an array of known constant size, and pointer `p` were declared to be a pointer to an array of the same known constant size (pointing to `a`), the results would be the same.

Forward references: array declarators (6.7.6.2), common definitions `<stddef.h>` (7.21).

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
  additive-expression
  shift-expression **<<** *additive-expression*
  shift-expression **>>** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is $\mathtt{E1} \times 2^{\mathtt{E2}}$, wrapped around. If `E1` has a signed type and nonnegative value, and $\mathtt{E1} \times 2^{\mathtt{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\mathtt{E1}/2^{\mathtt{E2}}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

6.5.8 Relational operators

Syntax

1 *relational-expression:*
  shift-expression
  relational-expression **<** *shift-expression*
  relational-expression **>** *shift-expression*
  relational-expression **<=** *shift-expression*
  relational-expression **>=** *shift-expression*

Constraints

2 One of the following shall hold:

— both operands have real type; or

— both operands are pointers to qualified or unqualified versions of compatible object types.

3 If either operand has decimal floating type, the other operand shall not have standard floating type.

Semantics

4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Positive zeros compare equal to negative zeros.

5 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

6 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point

one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`. In all other cases, the behavior is undefined.

7 Each of the operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[120]. The result has type **`int`**.

6.5.9 Equality operators

Syntax

1 *equality-expression:*
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

Constraints

2 One of the following shall hold:

— both operands have arithmetic type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **`void`**;

— both operands have type **`nullptr_t`**;

— one operand has type **`nullptr_t`** and the other is a null pointer constant; or,

— one operand is a pointer and the other is a null pointer constant.

3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

4 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence[121] Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type **`int`**. For any pair of operands, exactly one of the relations is true.

5 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Positive zeros compare equal to negative zeros. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal. If both operands have type **`nullptr_t`** or one operand has type **`nullptr_t`** and the other is a null pointer constant, they compare equal.

6 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **`void`**, the former is converted to the type of the latter.

[120]The expression `a<b<c` is not interpreted as in ordinary mathematics. As the syntax indicates, it means `(a<b)<c`; in other words, "if `a` is less than `b`, compare 1 to `c`; otherwise, compare 0 to `c`".
[121]Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

7 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[122)]

8 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

6.5.10 Bitwise AND operator

Syntax

1 *AND-expression:*
equality-expression
AND-expression **&** *equality-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

6.5.11 Bitwise exclusive OR operator

Syntax

1 *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression **^** *AND-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

6.5.12 Bitwise inclusive OR operator

Syntax

1 *inclusive-OR-expression:*
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

Constraints

2 Each of the operands shall have integer type.

[122)]Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the `|` operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

6.5.13 Logical AND operator

Syntax

1 *logical-AND-expression:*
  inclusive-OR-expression
  logical-AND-expression **&&** *inclusive-OR-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The && operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **`int`**.

4 Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

6.5.14 Logical OR operator

Syntax

1 *logical-OR-expression:*
  logical-AND-expression
  logical-OR-expression **||** *logical-AND-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The `||` operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **`int`**.

4 Unlike the bitwise `|` operator, the `||` operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.

6.5.15 Conditional operator

Syntax

1 *conditional-expression:*
  logical-OR-expression
  logical-OR-expression **?** *expression* **:** *conditional-expression*

Constraints

2 The first operand shall have scalar type.

3 One of the following shall hold for the second and third operands[123]:

— both operands have arithmetic type;

[123] If a second or third operand of type **`nullptr_t`** is used that is not a null pointer constant and the other operand is not a pointer or does not have type **`nullptr_t`** itself, a constraint is violated even if that other operand is a null pointer constant such as `0`.

— both operands have the same structure or union type;

— both operands have void type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— both operands have `nullptr_t` type;

— one operand is a pointer and the other is a null pointer constant or has type `nullptr_t`; or

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`.

4 If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

5 The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.[124)]

6 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

7 If both the second and third operands are pointers, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands; if one is a null pointer constant (other than a pointer) or has type `nullptr_t` and the other is a pointer, the result type is the pointer type; if both the second and third operands have `nullptr_t` type, the result also has that type. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to `void` or a qualified version of `void`, in which case the result type is a pointer to an appropriately qualified version of `void`.

8 **EXAMPLE** The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

9 Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

```
c_vp   c_ip   const void *
v_ip   0      volatile int *
c_ip   v_ip   const volatile int *
vp     c_cp   const void *
ip     c_ip   const int *
vp     ip     void *
```

124) A conditional expression does not yield an lvalue.

6.5.16 Assignment operators

Syntax

1 *assignment-expression:*
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator: one of
  `= *= /= %= += -= <<= >>= &= ^= |=`

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment,[125] but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold[126]:

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right operand has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right operand;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of **void**, and the type pointed to by the left operand has all the qualifiers of the type pointed to by the right operand;

— the left operand has an atomic, qualified, or unqualified version of the **nullptr_t** type and the type of the right is **nullptr_t**[127];

— the left operand is an atomic, qualified, or unqualified pointer, and the type of the right operand is **nullptr_t**;

— the left operand is an atomic, qualified, or unqualified **bool**, and the type of the right operand is **nullptr_t**;

— the left operand is an atomic, qualified, or unqualified pointer, and the right operand is a null pointer constant; or

— the left operand has type atomic, qualified, or unqualified **bool**, and the right is a pointer.

[125]The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

[126]The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **int volatile * const**).

[127]The assignment of an object of type **nullptr_t** with a value of another type, even if the value is a null pointer constant, is a constraint violation.

Semantics

2 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand. [128)]

3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall exactly match and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

4 **EXAMPLE 1** In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
      /* ... */
```

the **int** value returned by the function could be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable `c` would be declared as **int**.

5 **EXAMPLE 2** In the fragment:

```
char c;
int i;
long l;

l = (c = i);
```

the value of `i` is converted to the type of the assignment expression `c = i`, that is, **char** type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, **long int** type.

6 **EXAMPLE 3** Consider the fragment:

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;       // constraint violation
*cpp = &c;      // valid
*p = 0;         // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object `c`.

6.5.16.2 Compound assignment

Constraints

1 For the operators += and -= only, either the left operand shall be an atomic, qualified, or unqualified pointer to a complete object type, and the right shall have integer type; or the left operand shall have atomic, qualified, or unqualified arithmetic type, and the right shall have arithmetic type.

2 For the other operators, the left operand shall have atomic, qualified, or unqualified arithmetic type, and (considering the type the left operand would have after lvalue conversion) each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

[128)] As described in 6.2.6.1, a store to an object with atomic type is done with **`memory_order_seq_cst`** semantics.

Semantics

4 A *compound assignment* of the form `E1` *op*`=` `E2` is equivalent to the simple assignment expression `E1 = E1` *op* `(E2)`, except that the lvalue `E1` is evaluated only once, and with respect to an indeterminately sequenced function call, the operation of a compound assignment is a single evaluation. If `E1` has an atomic type, compound assignment is a read-modify-write operation with **`memory_order_seq_cst`** memory order semantics.

5 **NOTE 1** Where a pointer to an atomic object can be formed and `E1` and `E2` have integer type, this is equivalent to the following code sequence where *T1* is the type of `E1` and *T2* is the type of `E2`:

```
T1 *addr = &E1;
T2 val = (E2);
T1 old = *addr;
T1 new;
do {
      new = old op val;
} while (!atomic_compare_exchange_strong(addr, &old, new));
```

with `new` being the result of the operation.

If `E1` or `E2` has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of `new` would also be discarded to satisfy the equivalence of `E1` *op*`=` `E2` and `E1 = E1` *op* `(E2)`. For example, if Annex F is in effect, the floating types involved have IEC 60559 binary formats, and **`FLT_EVAL_METHOD`** is 0, the equivalent code would be:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* ... */
      fenv_t fenv;
      T1 *addr = &E1;
      T2 val = E2;
      T1 old = *addr;
      T1 new;
      feholdexcept(&fenv);
      for (;;) {
            new = old op val;
            if (atomic_compare_exchange_strong(addr, &old, new))
                        break;
            feclearexcept(FE_ALL_EXCEPT);
      }
      feupdateenv(&fenv);
```

If **`FLT_EVAL_METHOD`** is not 0, then *T2* is expected to be a type with the range and precision to which `E2` is evaluated to satisfy the equivalence.

6.5.17 Comma operator

Syntax

1 *expression:*

assignment-expression
expression **,** *assignment-expression*

Semantics

2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.[129)]

3 **EXAMPLE** As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within

[129)] A comma operator does not yield an lvalue.

the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Forward references: initialization (6.7.10).

6.6 Constant expressions

Syntax

1 *constant-expression:*
    conditional-expression

Description

2 A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[130]

4 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Semantics

5 An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic range and precision shall be at least as great as if the expression were being evaluated in the execution environment. [131]

6 A compound literal with storage-class specifier `constexpr` is a *compound literal constant*. A compound literal constant is a constant expression with the type and value of the unnamed object.

7 An identifier that is:

— an enumeration constant,

— a predefined constant, or

— declared with storage-class specifier `constexpr` and has an object type,

is a *named constant*, as is a postfix expression that applies the `.` member access operator to a named constant of structure or union type, even recursively. For enumeration and predefined constants, their value and type are defined in the respective clauses; for `constexpr` objects, such a named constant is a constant expression with the type and value of the declared object.

8 An *integer constant expression*[132] shall have integer type and shall only have operands that are integer constants, named and compound literal constants of integer type, character constants, `sizeof` expressions whose results are integer constants, `alignof` expressions, and floating, named, or compound literal constants of arithmetic type that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the typeof operators, `sizeof` operator, or `alignof` operator.

9 More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:

— a named constant,

— a compound literal constant,

— an arithmetic constant expression,

[130] The operand of a typeof (6.7.2.5), `sizeof`, or `alignof` operator is usually not evaluated (6.5.3.4).

[131] The use of evaluation formats as characterized by `FLT_EVAL_METHOD` and `DEC_EVAL_METHOD` also applies to evaluation in the translation environment.

[132] An integer constant expression is required in contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

— a null pointer constant,

— an address constant, or

— an address constant for a complete object type plus or minus an integer constant expression.

10 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, named or compound literal constants of arithmetic type, character constants, **`sizeof`** expressions whose results are integer constants, and **`alignof`** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the typeof operators, **`sizeof`** operator, or **`alignof`** operator.

11 An *address constant* is a null pointer[133], a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary `&` operator or an integer constant cast to pointer type, or implicitly using an expression of array or function type.

12 The array-subscript `[]` and member-access `->` operator, the address `&` and indirection `*` unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.[134]

13 A *structure or union constant* is a named constant or compound literal constant with structure or union type, respectively.

14 An implementation may accept other forms of constant expressions; however, they are not an integer constant expression.[135]

15 Starting from a structure or union constant, the member-access `.` operator may be used to form a named constant or compound literal constant as described above.

16 If the member-access operator `.` accesses a member of a union constant, the accessed member shall be the same as the member that is initialized by the union constant's initializer.

17 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions[136].

Forward references: array declarators (6.7.6.2), initialization (6.7.10).

---

[133] A named constant or compound literal constant of integer type and value zero is a null pointer constant. A named constant or compound literal constant with a pointer type and a value null is a null pointer but not a null pointer constant; it may only be used to initialize a pointer object if its type implicitly converts to the target type.

[134] Named constants or compound literal constants with arithmetic type, including names of **`constexpr`** objects, are valid in offset computations such as array subscripts or in pointer casts, as long as the expressions in which they occur form integer constant expressions. In contrast, names of other objects, even if **`const`**-qualified and with static storage duration, are not valid.

[135] For example, in the statement `int arr_or_vla[(int)+1.0];`, while possible to be computed by some implementations as an array with a size of one, still results in a variable length array declaration of automatic storage duration.

[136] Thus, in the following initialization,

```
static int i = 2 || 1 / 0;
```

the expression is a valid integer constant expression with value one.

6.7 Declarations

Syntax

1 *declaration:*

    declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**
    attribute-specifier-sequence declaration-specifiers init-declarator-list **;**
    static_assert-declaration
    attribute-declaration

declaration-specifiers:

    declaration-specifier attribute-specifier-sequence$_{\text{opt}}$
    declaration-specifier declaration-specifiers

declaration-specifier:

    storage-class-specifier
    type-specifier-qualifier
    function-specifier

init-declarator-list:

    init-declarator
    init-declarator-list **,** *init-declarator*

init-declarator:

    declarator
    declarator **=** *initializer*

attribute-declaration:

    attribute-specifier-sequence **;**

Constraints

2 A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— enumeration constants and tags may be redeclared as specified in 6.7.2.2 and 6.7.2.3, respectively.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

5 In an underspecified declaration all declared identifiers that do not have a prior declaration shall be ordinary identifiers.

Semantics

6 A declaration specifies the interpretation and properties of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that for:

— an object, causes storage to be reserved for that object,

— a function, includes the function body[137)],

— an enumeration constant, is the only declaration of the identifier, or

— a typedef name, is the first (or only) declaration of the identifier.

[137)] Function definitions have a different syntax, described in 6.9.1.

7 The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence. The declaration specifiers indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init declarator list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared. The optional attribute specifier sequence in a declaration appertains to each of the entities declared by the declarators of the init declarator list.

8 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer. In the case of function parameters, it is the adjusted type (see 6.7.6.3) that is required to be complete.

9 The optional attribute specifier sequence terminating a sequence of declaration specifiers appertains to the type determined by the preceding sequence of declaration specifiers. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

10 Except where specified otherwise, the meaning of an attribute declaration is implementation-defined.

11 **EXAMPLE** In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the identifier for that declaration.

```
[[deprecated]] void f [[deprecated]] (void); // valid
```

12 A declaration such that the declaration specifiers contain no type specifier or that is declared with **`constexpr`** is said to be *underspecified*. If such a declaration is not a definition, if it declares no or more than one ordinary identifier, if the declared identifier already has a declaration in the same scope, or if the declared entity is not an object, the behavior is undefined.

Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.10), type names (6.7.7), type qualifiers (6.7.3).

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

```
auto
constexpr
extern
register
static
thread_local
typedef
```

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that:

— **`thread_local`** may appear with **`static`** or **`extern`**,

— **`auto`** may appear with all the others except **`typedef`**[138], and

— **`constexpr`** may appear with **`auto`**, **`register`**, or **`static`**.

3 In the declaration of an object with block scope, if the declaration specifiers include **`thread_local`**, they shall also include either **`static`** or **`extern`**. If **`thread_local`** appears in any declaration of an object, it shall be present in every declaration of that object.

[138] See "future language directions" (6.11.5).

4 **thread_local** shall not appear in the declaration specifiers of a function declaration. **auto** shall only appear in the declaration specifiers of an identifier with file scope or along with other storage class specifiers if the type is to be inferred from an initializer.

5 An object declared with storage-class specifier **constexpr** or any of its members, even recursively, shall not have an atomic type, or a variably modified type, or a type that is **volatile** or **restrict** qualified. The declaration shall be a definition and shall have an initializer.[139] The value of any constant expressions or of any character in a string literal of the initializer shall be exactly representable in the corresponding target type; no change of value shall be applied[140]. If an object or subobject declared with storage-class specifier **constexpr** has pointer, integer, or arithmetic type, the implicit or explicit initializer value for it shall be a null pointer constant[141], an integer constant expression, or an arithmetic constant expression, respectively.

Semantics

6 Storage-class specifiers specify various properties of identifiers and declared features:

— storage duration (**static** in block scope, **thread_local**, **auto**, **register**),

— linkage (**extern**, **static** and **constexpr** in file scope, **typedef**),

— value (**constexpr**), and

— type (**typedef**).

7 The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4, **typedef** is discussed in 6.7.8, and type inference using **auto** is discussed in 6.7.9.

8 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined[142].

9 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

10 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, including recursively for any aggregate or union member objects.

11 If **auto** appears with another storage-class specifier, or if it appears in a declaration at file scope, it is ignored for the purposes of determining a storage duration or linkage. In this case, it indicates only that the declared type may be inferred.

12 An object declared with a storage-class specifier **constexpr** has its value permanently fixed at translation-time; if not yet present, a **const**-qualification is implicitly added to the object's type. The declared identifier is considered a constant expression of the respective kind, see 6.6.

13 **NOTE 1** An object declared in block scope with a storage-class specifier **constexpr** and without **static** has automatic storage duration, the identifier has no linkage, and each instance of the object has a unique address obtainable with & (if it is not declared with the **register** specifier), if any. Such an object in file scope has static storage duration, the corresponding identifier has internal linkage, and each translation unit that sees the same textual definition implements a separate object with a distinct address.

14 **NOTE 2** The constraints for **constexpr** objects are intended to enforce checks for portability at translation time.

---

[139] All assignment expressions of such an initializer, if any, are constant expressions or string literals, see 6.7.10.

[140] In the context of arithmetic conversions, 6.3.1 describes the details of changes of value that occur if values of arithmetic expressions are stored in the objects that for example have a different signedness, excess precision or quantum exponent. Whenever such a change of value is necessary, the constraint is violated.

[141] The named constant or compound literal constant corresponding to an object declared with storage-class specifier **constexpr** and pointer type is a constant expression with a value null, and thus a null pointer and an address constant. However, even if it has type **void*** it is not a null pointer constant.

[142] The implementation can treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary & operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof** and the typeof operators.

```
constexpr unsigned int minusOne    = -1;              // constraint violation
constexpr unsigned int uint_max    = -1U;             // ok
constexpr char string[]            = { "\xFF", };     // ok
constexpr unsigned char unstring[] = { "\xFF", };     // possible constraint
                                                      // violation
constexpr char8_t u8string[]       = { u8"\xFF", };   // ok
constexpr double onethird          = 1.0/3.0;         // possible constraint
                                                      // violation
constexpr double onethirdtrunc     = (double)(1.0/3.0);  // ok
constexpr _Decimal32 small         = DEC64_TRUE_MIN * 0; // constraint violation
```

Using an octal or hexadecimal escape character sequence with a value greater than the largest representable value of the target character type (such as for `unstring`) possibly violates a constraint. Equally, an implementation that uses excess precision for floating constants violates the constraint for `onethird`; a diagnostic is required if a truncation of the mantissa occurs. In contrast to that, the explicit conversion in the initializer for `onethirdtrunc` ensures that the definition is valid. Similarly, the initializer of `small` has a quantum exponent that is larger than the largest possible quantum exponent for **`_Decimal32`**.

15 **EXAMPLE 1** An identifier declared with the **`constexpr`** specifier may have its value used in constant expressions:

```
constexpr int K = 47;
enum {
      A = K,              // valid, constant initialization
};
constexpr int L = K;     // valid, constexpr initialization
static int b    = K + 1; // valid, static initialization
int array[K];            // not a VLA
```

16 **EXAMPLE 2** An object declared with the **`constexpr`** specifier stores the exact value of its initializer, no implicit value change is applied:

```
#include <float.h>

constexpr int A          = 42LL;          // valid, 42 always fits in an int
constexpr signed short B = ULLONG_MAX;    // constraint violation, value never
                                          // fits
constexpr float C        = 47u;           // valid, exactly representable
                                          // in single precision

#if FLT_MANT_DIG > 24
constexpr float D = 536900000;            // constraint violation if float is
                                          // 32-bit single-precision IEC 60559
#endif

#if (FLT_MANT_DIG == DBL_MANT_DIG) && (0 <= FLT_EVAL_METHOD) && (FLT_EVAL_METHOD
    <= 1)
constexpr float E = 1.0 / 3.0;            // only valid if double expressions
                                          // and float objects have the same
                                          // precision
#endif

#if FLT_EVAL_METHOD == 0
constexpr float F = 1.0f / 3.0f;          // valid, same type and precision
#else
constexpr float F = (float)(1.0f / 3.0f); // needs cast to truncate the
                                          // excess precision
#endif
```

17 **EXAMPLE 3** This recursively applies to initializers for all elements of an aggregate object declared with the **`constexpr`** specifier:

```
constexpr static unsigned short array[] = {
      3000,    // valid, fits in unsigned short range
      300000,  // constraint violation if short is 16-bit
      -1       // constraint violation, target type is unsigned
};

struct S {
      int x, y;
};
constexpr struct S s = {
      .x = INT_MAX,         // valid
      .y = UINT_MAX,        // constraint violation
};
```

Forward references: type definitions (6.7.8), type definitions (6.7.9).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

```
void
char
short
int
long
float
double
signed
unsigned
_BitInt ( constant-expression )
bool
_Complex
_Decimal32
_Decimal64
_Decimal128
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
typeof-specifier
```

Constraints

2 Except where the type is inferred (6.7.9), at least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each member declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **void**

— **char**

— **signed char**

— **unsigned char**

— **short**, **signed short**, **short int**, or **signed short int**

— **unsigned short**, or **unsigned short int**

— **`int`**, **`signed`**, or **`signed int`**

— **`unsigned`**, or **`unsigned int`**

— **`long`**, **`signed long`**, **`long int`**, or **`signed long int`**

— **`unsigned long`**, or **`unsigned long int`**

— **`long long`**, **`signed long long`**, **`long long int`**, or **`signed long long int`**

— **`unsigned long long`**, or **`unsigned long long int`**

— **`_BitInt(`** *constant-expression* **`)`**, or **`signed _BitInt(`** *constant-expression* **`)`**

— **`unsigned _BitInt(`** *constant-expression* **`)`**

— **`float`**

— **`double`**

— **`long double`**

— **`_Decimal32`**

— **`_Decimal64`**

— **`_Decimal128`**

— **`bool`**

— **`float _Complex`**

— **`double _Complex`**

— **`long double _Complex`**

— atomic type specifier

— struct or union specifier

— enum specifier

— typedef name

— typeof specifier

3 The type specifier **`_Complex`** shall not be used if the implementation does not support complex types, and the type specifiers **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** shall not be used if the implementation does not support decimal floating types (see 6.10.9.3).

4 The parenthesized constant expression that follows the **`_BitInt`** keyword shall be an integer constant expression N that specifies the width (6.2.6.2) of the type. The value of N for **`unsigned _BitInt`** shall be greater than or equal to 1. The value of N for **`_BitInt`** shall be greater than or equal to 2. The value of N shall be less than or equal to the value of **`BITINT_MAXWIDTH`** (see 5.2.4.2.1).

Semantics

5 Specifiers for structures, unions, enumerations, atomic types, and typeof specifiers are discussed in 6.7.2.1 through 6.7.2.5. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5.

6 For a declaration such that the declaration specifiers contain no type specifier a mechanism to infer the type from an initializer is discussed in 6.7.9. In such a declaration, optional elements, if any, of a sequence of declaration specifiers appertain to the inferred type (for qualifiers and attribute specifiers) or to the declared objects (for alignment specifiers).

7 Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **`int`** designates the same type as **`signed int`** or the same type as **`unsigned int`**.

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
    struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*$_{opt}$ **{** *member-declaration-list* **}**
    struct-or-union attribute-specifier-sequence$_{opt}$ *identifier*

struct-or-union:
    `struct`
    `union`

member-declaration-list:
    member-declaration
    member-declaration-list member-declaration

member-declaration:
    attribute-specifier-sequence$_{opt}$ *specifier-qualifier-list member-declarator-list*$_{opt}$ **;**
    static_assert-declaration

specifier-qualifier-list:
    type-specifier-qualifier attribute-specifier-sequence$_{opt}$
    type-specifier-qualifier specifier-qualifier-list

type-specifier-qualifier:
    type-specifier
    type-qualifier
    alignment-specifier

member-declarator-list:
    member-declarator
    member-declarator-list **,** *member-declarator*

member-declarator:
    declarator
    declarator$_{opt}$ **:** *constant-expression*

Constraints

2 A member declaration that does not declare an anonymous structure or anonymous union shall contain a member declarator list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted[143]. If the value is zero, the declaration shall have no declarator.

5 A bit-field shall have a type that is a qualified or unqualified version of **`bool`**, **`signed int`**, **`unsigned`**

[143]While the number of bits in a **`bool`** object is at least **`CHAR_BIT`**, the width of a **`bool`** is just 1 bit.

`int`, a bit-precise integer type, or other implementation-defined type. It is implementation-defined whether atomic types are permitted.

6 An attribute specifier sequence shall not appear in a struct-or-union specifier without a member declaration list, except in a declaration of the form:

struct-or-union attribute-specifier-sequence identifier **`;`**

The attributes in the attribute specifier sequence, if any, are thereafter considered attributes of the **`struct`** or **`union`** whenever it is named.

Semantics

7 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

8 Structure and union specifiers have the same form. The keywords **`struct`** and **`union`** indicate that the type being specified is, respectively, a structure type or a union type.

9 The optional attribute specifier sequence in a struct-or-union specifier appertains to the structure or union type being declared. The optional attribute specifier sequence in a member declaration appertains to each of the members declared by the member declarator list; it shall not appear if the optional member declarator list is omitted. The optional attribute specifier sequence in a specifier qualifier list appertains to the type denoted by the preceding type specifier qualifiers. The attribute specifier sequence affects the type only for the member declaration or type name it appears in, not other types or declarations involving the same type.

10 The member declaration list is a sequence of declarations for the members of the structure or union. If the member declaration list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined[144).

11 A member of a structure or union may have any complete object type other than a variably modified type.[145) In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field* [146); its width is preceded by a colon.

12 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits[147). If the value 0 or 1 is stored into a nonzero-width bit-field of type **`bool`**, the value of the bit-field shall compare equal to the value stored; a **`bool`** bit-field has the semantics of a **`bool`**.

13 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

14 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[148) As a special case, a bit-field structure member with a width of zero indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

15 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

[144)For further rules affecting compatibility and completeness of structure or union types, see 6.2.7 and 6.7.2.3.

[145)A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[146)The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[147)As specified in 6.7.2 above, if the actual type specifier used is **`int`** or a typedef-name defined as **`int`**, then it is implementation-defined whether the bit-field is signed or unsigned. This includes an **`int`** type specifier produced using the typeof specifiers (6.7.2.5).

[148)An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

16 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

17 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

18 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa. The members of a union object overlap in such a way that pointers to them when converted to pointers to character type point to the same byte. There may be unnamed padding at the end of a union object, but not at its beginning.

19 There may be unnamed padding at the end of a structure or union.

20 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

21 **EXAMPLE 1** The following declarations illustrate the behavior when an attribute is written on a tag declaration:

```
struct [[deprecated]] S; // valid, [[deprecated]] appertains to struct S
void f(struct S *s);     // valid, the struct S type has the [[deprecated]]
                         // attribute
struct S {               // valid, struct S inherits the [[deprecated]] attribute
      int a;              // from the previous declaration
};
void g(struct [[deprecated]] S s); // invalid
```

22 **EXAMPLE 2** The following illustrates anonymous structures and unions:

```
struct v {
      union {     // anonymous union
            struct { int i, j; };   // anonymous structure
            struct { long k, l; } w;
      };
      int m;
} v1;

v1.i = 2;   // valid
v1.k = 3;   // invalid:  inner structure is not anonymous
v1.w.k = 5; // valid
```

23 **EXAMPLE 3** After the declaration:

```
struct s { int n; double d[]; };
```

the structure `struct s` has a flexible array member `d`. A typical way to use this is:

```
int m = /* some value */;
struct s *p = malloc(sizeof(struct s) + sizeof(double [m]));
```

and assuming that the call to `malloc` succeeds, the object pointed to by `p` behaves, for most purposes, as if `p` had been declared as:

```
struct { int n; double d[m]; } *p;
```

(there are circumstances in which this equivalence is broken; in particular, the offsets of member `d` might not be the same).

24 Following the above declaration:

```
struct s t1 = { 0 };          // valid
struct s t2 = { 1, { 4.2 }};  // invalid
t1.n = 4;                     // valid
t1.d[0] = 4.2;                // might be undefined behavior
```

The initialization of `t2` is invalid (and violates a constraint) because `struct s` is treated as if it did not contain member `d`. The assignment to `t1.d[0]` is probably undefined behavior, but it is possible that

```
sizeof(struct s) >= offsetof(struct s, d) + sizeof(ouble)
```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

25 After the further declaration:

```
struct ss { int n; };
```

the expressions:

```
sizeof(struct s) >= sizeof(struct ss)
sizeof(struct s) >= offsetof(struct s, d)
```

are always equal to 1.

26 If `sizeof(double)` is 8, then after the following code is executed:

```
struct s *s1;
struct s *s2;
s1 = malloc(sizeof(struct s) + 64);
s2 = malloc(sizeof(struct s) + 46);
```

and assuming that the calls to `malloc` succeed, the objects pointed to by `s1` and `s2` behave, for most purposes, as if the identifiers had been declared as:

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

27 Following the further successful assignments:

```
s1 = malloc(sizeof(struct s) + 10);
s2 = malloc(sizeof(struct s) +  6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]); // valid
*dp = 42;         // valid
dp = &(s2->d[0]); // valid
*dp = 42;         // undefined behavior
```

28 The assignment:

```
*s1 = *s2;
```

only copies the member `n`; if any of the array elements are within the first `sizeof(struct s)` bytes of the structure, they are set to an indeterminate representation, that may or may not coincide with a copy of the representation of the elements of the source array.

29 **EXAMPLE 4** Because members of anonymous structures and unions are considered to be members of the containing structure or union, `struct s` in the following example has more than one named member and thus the use of a flexible array member is valid:

```
struct s {
      struct { int i; };
      int a[];
};
```

Forward references: declarators (6.7.6), tags (6.7.2.3).

6.7.2.2 Enumeration specifiers

Syntax

1 *enum-specifier:*
  `enum` *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ *enum-type-specifier*$_{opt}$ **`{`** *enumerator-list* **`}`**
  `enum` *attribute-specifier-sequence*$_{opt}$ *identifier*$_{opt}$ *enum-type-specifier*$_{opt}$ **`{`** *enumerator-list* **`,`** **`}`**
  `enum` *identifier* *enum-type-specifier*$_{opt}$

enumerator-list:
  enumerator
  enumerator-list **`,`** *enumerator*

enumerator:
  enumeration-constant *attribute-specifier-sequence*$_{opt}$
  enumeration-constant *attribute-specifier-sequence*$_{opt}$ **`=`** *constant-expression*

enum-type-specifier:
  `:` *specifier-qualifier-list*

2 All enumerations have an *underlying type*. The underlying type can be explicitly specified using an enum type specifier and is its *fixed underlying type*. If it is not explicitly specified, the underlying type is the enumeration's compatible type, which is either **`char`** or a standard or extended signed or unsigned integer type.

Constraints

3 For an enumeration with a fixed underlying type, the integer constant expression defining the value of the enumeration constant shall be representable in that fixed underlying type. The definition of an enumeration constant without a defining constant expression shall neither overflow nor wraparound the fixed underlying type by adding 1 to the previous enumeration constant.

4 For an enumeration without a fixed underlying type, the expression that defines the value of an enumeration constant shall be an integer constant expression. For all the integer constant expressions which make up the values of the enumeration constants, there shall be a type capable of representing

all the values that is a standard or extended signed or unsigned integer type, or **`char`**.

5 If an enum type specifier is present, then the longest possible sequence of tokens that can be interpreted as a specifier qualifier list is interpreted as part of the enum type specifier. It shall name an integer type that is neither an enumeration nor a bit-precise integer type.

6 An enum specifier of the form

> **`enum`** *identifier* *enum-type-specifier*

may not appear except in a declaration of the form

> **`enum`** *identifier* *enum-type-specifier* `;`

unless it is immediately followed by an opening brace, an enumerator list (with an optional ending comma), and a closing brace.

7 If two enum specifiers that include an enum type specifier declare the same type, the underlying types shall be compatible.

Semantics

8 The optional attribute specifier sequence in the **`enum`** specifier appertains to the enumeration; the attributes in that attribute specifier sequence are thereafter considered attributes of the enumeration whenever it is named. The optional attribute specifier sequence in the enumerator appertains to that enumerator.

9 The identifiers in an enumerator list are declared as constants of the types specified below and may appear wherever such are permitted.[149] An enumerator with `=` defines its enumeration constant as the value of the constant expression. If the first enumerator has no `=`, the value of its enumeration constant is zero. Each subsequent enumerator with no `=` defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with `=` may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

10 The type for the members of an enumeration is called the *enumeration member type*.

11 During the processing of each enumeration constant in the enumerator list, the type of the enumeration constant shall be:

- the previously declared type, if it is a redeclaration of the same enumeration constant; or,
- the enumerated type, for an enumeration with fixed underlying type; or,
- **`int`**, if there are no previous enumeration constants in the enumerator list and no explicit `=` with a defining integer constant expression; or,
- **`int`**, if given explicitly with `=` and the value of the integer constant expression is representable by an **`int`**; or,
- the type of the integer constant expression, if given explicitly with `=` and if the value of the integer constant expression is not representable by **`int`**; or,
- the type of the value from the previous enumeration constant with one added to it. If such an integer constant expression would overflow or wraparound the value of the previous enumeration constant from the addition of one, the type takes on either:
  - a suitably sized signed integer type (excluding the bit-precise signed integer types) capable of representing the value of the previous enumeration constant plus one; or,

[149] Thus, the identifiers of enumeration constants declared in the same scope are all required to be distinct from each other and from other identifiers declared in ordinary declarators.

— a suitably sized unsigned integer type (excluding the bit-precise unsigned integer types) capable of representing the value of the previous enumeration constant plus one.

A signed integer type is chosen if the previous enumeration constant being added is of signed integer type. An unsigned integer type is chosen if the previous enumeration constant is of unsigned integer type. If there is no suitably sized integer type described previously which can represent the new value, then the enumeration has no type which can represent all its values.[150)]

12 For all enumerations without a fixed underlying type, each enumerated type shall be compatible with **`char`**, a signed integer type, or an unsigned integer type (excluding the bit-precise integer types). The choice of type is implementation-defined[151)], but shall be capable of representing the values of all the members of the enumeration[152)].

13 Enumeration constants can be redefined in the same scope with the same value as part of a redeclaration of the same enumerated type.

14 The enumeration member type for an enumerated type without fixed underlying type upon completion is:

— **`int`** if all the values of the enumeration are representable as an **`int`**; or,

— the enumerated type.[153)]

15 The enumeration member type for an enumerated type with fixed underlying type is the enumerated type. The enumerated type is compatible with the underlying type of the enumeration. After possible lvalue conversion a value of the enumerated type behaves the same as the value with the underlying type, in particularly with all aspects of promotion, conversion, and arithmetic[154)].

16 **EXAMPLE** The following fragment:

```
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
col = claret;
cp = &col;
if (*cp != burgundy)
      /* ... */
```

makes `hue` the tag of an enumeration, and then declares `col` as an object that has that type and `cp` as a pointer to an object that has that type. The enumerated values are in the set $\{0, 1, 20, 21\}$.

17 **EXAMPLE** Even if the value of an enumeration constant is generated by the implicit addition of one, an enumeration with a fixed underlying type does not exhibit typical overflow behavior:

```
#include <limits.h>

enum us : unsigned short {
      us_max = USHRT_MAX,
      us_violation, /* Constraint violation:
                       USHRT_MAX + 1 would wraparound. */
      us_violation_2 = us_max + 1, /* Maybe constraint violation:
                                      USHRT_MAX + 1 may be promoted to "int", and
                                      result is too wide for the
```

150)Therefore, a constraint has been violated.

151)An implementation can delay the choice of which integer type until all enumeration constants have been seen.

152)For further rules affecting compatibility and completeness of enumerated types see 6.2.7 and 6.7.2.3.

153)The integer type selected during processing of the enumerator list (before completion) of the enumeration may not be the same as the compatible implementation-defined integer type selected for the completed enumeration.

154)This means in particular that if the compatible type is **`bool`**, values of the enumerated type behave in all aspects the same as **`bool`** and the members only have values **`false`** and **`true`**. If it is a signed integer type and the constant expression of an enumeration constant overflows, a constraint for constant expressions (6.6) is violated.

```
                                     underlying type. */
        us_wraparound_to_zero = (unsigned short)(USHRT_MAX + 1) /* Okay:
                              conversion done in constant expression
                              before conversion to underlying type:
                              unsigned semantics okay. */
};

enum ui : unsigned int {
        ui_max = UINT_MAX,
        ui_violation, /* Constraint violation:
                         UINT_MAX + 1 would wraparound. */
        ui_no_violation = ui_max + 1, /* Okay: Arithmetic performed as typical
                                         unsigned integer arithmetic: conversion
                                         from a value that is already 0 to 0. */
        ui_wraparound_to_zero = (unsigned int)(UINT_MAX + 1) /* Okay: conversion
                              done in constant expression before conversion to
                              underlying type: unsigned semantics okay. */
};

int main () {
        // Same as return 0;
        return ui_wraparound_to_zero + us_wraparound_to_zero;
}
```

18 **EXAMPLE** The following fragment:

```
#include <limits.h>

enum E1: short;
enum E2: short;
enum E3; /* Constraint violation: E3 forward declaration. */
enum E4 : unsigned long long;

enum E1 : short { m11, m12 };
enum E1 x = m11;

enum E2 : long { m21, m22 }; /* Constraint violation: different underlying types
    */

enum E3 {
      m31,
      m32,
      m33 = sizeof(enum E3) /* Constraint violation: E3 is not complete here. */
};
enum E3 : int; /* Constraint violation: E3 previously had no underlying type */

enum E4 : unsigned long long {
      m40 = sizeof(enum E4),
      m41 = ULLONG_MAX,
      m42 /* Constraint violation: unrepresentable value (wraparound) */
};

enum E5 y; /* Constraint violation: incomplete type */
enum E6 : long int z; /* Constraint violation: enum-type-specifier
                         with identifier in declarator */
enum E7 : long int = 0; /* Syntax violation:
                           enum-type-specifier with initializer */
```

demonstrates many of the properties of multiple declarations of enumerations with underlying types. Particularly, `enum E3` is declared and defined without an underlying type first, therefore a redeclaration with an underlying type second is a violation. Because it not complete at that time within its enumerator list, `sizeof(enum E3)` is a constraint violation within the `enum E3` definition. `enum E4` is complete as it is being defined, therefore `sizeof(enum E4)` is not a constraint violation.

19 **EXAMPLE** The following fragment:

```
enum no_underlying {
      a0
};

int main (void) {
      int a = _Generic(a0,
            int: 2,
            unsigned char: 1,
            default: 0
      );
      int b = _Generic((enum no_underlying)a0,
            int: 2,
            unsigned char: 1,
            default: 0
      );
      return a + b;
}
```

demonstrates the implementation-defined nature of the underlying type of enumerations using generic selection (6.5.1.1). The value of `a` after its initialization is `2`. The value of `b` after its initialization is implementation-defined: the enumeration must be compatible with a type large enough to fit the values of its enumeration constants. Because the only value is `0` for `a0`, `b` may hold any of `2`, `1`, or `0`.

Now, consider a similar fragment, but using a fixed underlying type:

```
enum underlying : unsigned char {
      b0
};

int main (void) {
      int a = _Generic(b0,
            int: 2,
            unsigned char: 1,
            default: 0
      );
      int b = _Generic((enum underlying)b0,
            int: 2,
            unsigned char: 1,
            default: 0
      );
      return 0;
}
```

Here, we are guaranteed that `a` and `b` are both initialized to one. This makes enumerations with a fixed underlying type more portable.

20 **EXAMPLE** Enumerations with a fixed underlying type must have their braces and the enumerator list specified as part of their declaration if they are not a standalone declaration:

```
void f1 (enum a : long b); /* Constraint violation */
void f2 (enum c : long { x } d);
enum e : int f3(); /* Constraint violation */

typedef enum t u; /* Constraint violation: forward declaration of t. */
typedef enum v : short W; /* Constraint violation */
typedef enum q : short { s } R;
```

```
struct s1 {
      int x;
      enum e : int : 1; /* Constraint violation */
      int y;
};

enum forward; /* Constraint violation */
extern enum forward fwd_val0; /* Constraint violation: incomplete type */
extern enum forward* fwd_ptr0; /* Constraint violation: enums cannot be
                                  used like other incomplete types */
extern int* fwd_ptr0; /* Constraint violation: incompatible
                         with incomplete type. */

enum forward1 : int;
extern enum forward1 fwd_val1;
extern int fwd_val1;
extern enum forward1* fwd_ptr1;
extern int* fwd_ptr1;

int main () {
      enum e : short;
      enum e : short f = 0; /* Constraint violation */
      enum g : short { y } h = y;
      return 0;
}
```

21 **EXAMPLE** Enumerations with a fixed underlying type are complete when the enum type specifier for that specific enumeration is complete. The enumeration `e` in this snippet:

```
enum e : typeof ((enum e : short { A })0, (short)0);
```

`enum e` is considered complete by the first opening brace within the **`typeof`** in this snippet.

Forward references: generic selection (6.5.1.1), tags (6.7.2.3), declarations (6.7), declarators (6.7.6), function declarators (6.7.6.3), type names (6.7.7).

6.7.2.3 Tags

Constraints

1 Where two declarations that use the same tag declare the same type, they shall both use the same choice of **`struct`**, **`union`**, or **`enum`**. If two declarations of the same type have a member-declaration or enumerator-list, one shall not be nested within the other and both declarations shall fulfill all requirements of compatible types (6.2.7) with the additional requirement that corresponding members of structure or union types shall have the same (and not merely compatible) types.

2 A type specifier of the form

> **`enum`** *identifier*

without an enumerator list shall only appear after the type it specifies is complete.

3 A type specifier of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

shall not contain an attribute specifier sequence[155].

Semantics

4 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type.

[155] As specified in 6.7.2.1 above, the type specifier may be followed by a `;` or a member declaration list.

5 Irrespective of whether there is a tag or what other declarations of the type are in the same translation unit, the type (except enumerated types with a fixed underlying type) is incomplete until immediately after the closing brace of the list defining the content for the first time and complete thereafter.

6 Enumerated types with fixed underlying type (6.7.2.2) are complete immediately after their first associated enum type specifier ends.

7 **EXAMPLE 1** The following example shows allowed redeclarations of the same structure, union, or enumerated type in the same scope:

```
struct foo { struct { int x; }; };
struct foo { struct { int x; }; };
union bar { int x; float y; };
union bar { float y; int x; };
typedef struct q { int x; } q_t;
typedef struct q { int x; } q_t;
void foo(void)
{
      struct S { int x; };
      struct T { struct S s; };
      struct S { int x; };
      struct T { struct S s; };
}
enum X { A = 1, B = 1 + 1 };
enum X { B = 2, A = 1 };
```

8 **EXAMPLE 2** The following example shows invalid redeclarations of the same structure, union, or enumerated type in the same scope:

```
struct foo { int (*p)[3]; };
struct foo { int (*p)[]; };    // member has different type

union bar { int x; float y; };
union bar { int z; float y; }; // member has different name

typedef struct { int x; } q_t;
typedef struct { int x; } q_t; // not the same type

struct S { int x; };

void foo(void)
{
      struct T { struct S s; };
      struct S { int x; };
      struct T { struct S s; }; // struct S not the same type
}

enum X { A = 1, B = 2 };
enum X { A = 1, B = 3 };       // different enumeration constant

enum R { C = 1 };
enum Q { C = 1 };              // conflicting enumeration constant
enum Q { C = C };              // ok!
```

9 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

10 A type specifier of the form

struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}** or

> **enum** *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**

or

> **enum** *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *enumerator-list* **, }**

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided[156], the type specifier also declares the identifier to be the tag of that type. The optional attribute specifier sequence appertains to the structure, union, or enumeration type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure, union, or enumeration type whenever it is named.

11 A declaration of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier* **;**

specifies a structure or union type and declares the identifier as a tag of that type[157]. The optional attribute specifier sequence appertains to the structure or union type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure or union type whenever it is named.

12 If a type specifier of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.[157]

13 If a type specifier of the form

> *struct-or-union attribute-specifier-sequence*$_{\text{opt}}$ *identifier*

or

> **enum** *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

14 **EXAMPLE 3** This mechanism allows declaration of a self-referential structure.

```
struct tnode {
      int count;
      struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be an object of the given type and `sp` to be a pointer to an object of the given type. With these declarations, the expression `sp->left` refers to the left `struct tnode` pointer of the object to which `sp` points; the expression `s.right->count` designates the `count` member of the right `struct tnode` pointed to from `s`.

15 The following alternative formulation uses the **typedef** mechanism:

```
typedef struct tnode TNODE;
struct tnode {
```

[156] If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

[157] A similar construction with **enum** does not exist.

```
        int count;
        TNODE *left, *right;
    };
    TNODE s, *sp;
```

16 **EXAMPLE 4** To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

```
    struct s1 { struct s2 *s2p; /* ... */ }; // D1
    struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to *it*, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the declaration

```
    struct s2;
```

can be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

Forward references: declarators (6.7.6), type definitions (6.7.8).

6.7.2.4 Atomic type specifiers

Syntax

1 *atomic-type-specifier:*
    `_Atomic` **`(`** *type-name* **`)`**

Constraints

2 Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.9.3).

3 The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, or a qualified type.

Semantics

4 The properties associated with atomic types are meaningful only for expressions that are lvalues. If the **`_Atomic`** keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.

6.7.2.5 Typeof specifiers

Syntax

1 *typeof-specifier:*
    `typeof` **`(`** *typeof-specifier-argument* **`)`**
    `typeof_unqual` **`(`** *typeof-specifier-argument* **`)`**
typeof-specifier-argument:
    expression
    type-name

2 The **`typeof`** and **`typeof_unqual`** tokens are collectively called the *typeof operators*.

Constraints

3 The typeof operators shall not be applied to an expression that designates a bit-field member.

Semantics

4 The typeof specifier applies the typeof operators to an *expression* (6.5) or a type name. If the typeof operators are applied to an expression, they yield the type name representing the type of their

operand[158]. Otherwise, they produce the type name with any nested typeof specifier evaluated.[159] If the type of the operand is a variably modified type, the operand is evaluated; otherwise, the operand is not evaluated.

5 The result of the **typeof_unqual** operation is the non-atomic unqualified version of the type name that would result from the **typeof** operation[160]. The **typeof** operator preserves all qualifiers.

6 **EXAMPLE 1** Type of an expression.

```
typeof(1+1) main () {
      return 0;
}
```

is equivalent to this program:

```
int main () {
      return 0;
}
```

7 **EXAMPLE 2** The following program:

```
const _Atomic int purr = 0;
const int meow = 1;
const char* const animals[] = {
      "aardvark",
      "bluejay",
      "catte",
};

typeof_unqual(meow) main (int argc, char* argv[]) {
      typeof_unqual(purr)          plain_purr;
      typeof(_Atomic typeof(meow)) atomic_meow;
      typeof(animals)              animals_array;
      typeof_unqual(animals)       animals2_array;
      return 0;
}
```

is equivalent to this program:

```
const _Atomic int purr = 0;
const int meow = 1;
const char* const animals[] = {
      "aardvark",
      "bluejay",
      "catte",
};

int main (int argc, char* argv[]) {
      int                 plain_purr;
      const _Atomic int atomic_meow;
      const char* const animals_array[3];
      const char*       animals2_array[3];
      return 0;
}
```

[158]When applied to a parameter declared to have array or function type, the typeof operators yield the adjusted (pointer) type (see 6.9.1).

[159]If the typeof specifier argument is itself a typeof specifier, the operand will be evaluated before evaluating the current typeof operator. This happens recursively until a typeof specifier is no longer the operand.

[160]`_Atomic ( type-name )`, with parentheses, is considered an `_Atomic`-qualified type.

8 **EXAMPLE 3** The equivalence between **sizeof** and **typeof**'s deduction of the type means this program has no constraint violations:

```
int main (int argc, char* argv[]) {
      static_assert(sizeof(typeof('p')) == sizeof(int));
      static_assert(sizeof(typeof('p')) == sizeof('p'));
      static_assert(sizeof(typeof((char)'p')) == sizeof(char));
      static_assert(sizeof(typeof((char)'p')) == sizeof((char)'p'));
      static_assert(sizeof(typeof("meow")) == sizeof(char[5]));
      static_assert(sizeof(typeof("meow")) == sizeof("meow"));
      static_assert(sizeof(typeof(argc)) == sizeof(int));
      static_assert(sizeof(typeof(argc)) == sizeof(argc));
      static_assert(sizeof(typeof(argv)) == sizeof(char**));
      static_assert(sizeof(typeof(argv)) == sizeof(argv));

      static_assert(sizeof(typeof_unqual('p')) == sizeof(int));
      static_assert(sizeof(typeof_unqual('p')) == sizeof('p'));
      static_assert(sizeof(typeof_unqual((char)'p')) == sizeof(char));
      static_assert(sizeof(typeof_unqual((char)'p')) == sizeof((char)'p'));
      static_assert(sizeof(typeof_unqual("meow")) == sizeof(char[5]));
      static_assert(sizeof(typeof_unqual("meow")) == sizeof("meow"));
      static_assert(sizeof(typeof_unqual(argc)) == sizeof(int));
      static_assert(sizeof(typeof_unqual(argc)) == sizeof(argc));
      static_assert(sizeof(typeof_unqual(argv)) == sizeof(char**));
      static_assert(sizeof(typeof_unqual(argv)) == sizeof(argv));
      return 0;
}
```

9 **EXAMPLE 4** The following program with nested **typeof**(...):

```
int main (int argc, char*[]) {
      float val = 6.0f;
      return (typeof(typeof_unqual(typeof(argc))))val;
}
```

is equivalent to this program:

```
int main (int argc, char*[]) {
      float val = 6.0f;
      return (int)val;
}
```

10 **EXAMPLE 5** Variable length arrays with typeof operators performs the operation at execution time rather than translation time.

```
#include <stddef.h>

size_t vla_size (int n) {
      typedef char vla_type[n + 3];
      vla_type b; // variable length array
      return sizeof(
            typeof_unqual(b)
      ); // execution-time sizeof, translation-time typeof operation
}

int main () {
      return (int)vla_size(10); // vla_size returns 13
}
```

11 **EXAMPLE 6** Nested typeof operators, arrays, and pointers do not perform array to pointer decay.

```
int main () {
      typeof(typeof(const char*)[4]) y = {
            "a",
            "b",
            "c",
            "d"
      }; // 4-element array of "pointer to const char"
      return 0;
}
```

12 **EXAMPLE 7** Function, pointer, and array types may be substituted with typeof operations.

```
void f(int);

typeof(f(5)) g(double x) {          // g has type "void(double)"
      printf("value %g\n", x);
}

typeof(g)* h;                       // h has type "void(*)(double)"
typeof(true ? g : NULL) k;          // k has type "void(*)(double)"

void j(double A[5], typeof(A)* B); // j has type "void(double*, double**)"

extern typeof(double[]) D;          // D has an incomplete type
typeof(D) C = { 0.7, 99 };          // C has type "double[2]"

typeof(D) D = { 5, 8.9, 0.1, 99 }; // D is now completed to "double[4]"
typeof(D) E;                        // E has type "double[4]" from D's completed type
```

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

```
const
restrict
volatile
_Atomic
```

Constraints

2 Types other than pointer types whose referenced type is an object type and (possibly multi-dimensional) array types with such pointer types as element type shall not be restrict-qualified.

3 The **_Atomic** qualifier shall not be used if the implementation does not support atomic types (see 6.10.9.3).

4 The type modified by the **_Atomic** qualifier shall not be an array type or a function type.

Semantics

5 The properties associated with qualified types are meaningful only for expressions that are lvalues.[161)]

6 If the same qualifier appears more than once in the same specifier-qualifier list or as declaration specifiers, either directly, via one or more typeof specifiers, or via one or more **typedef**s, the behavior is the same as if it appeared only once. If other qualifiers appear along with the **_Atomic** qualifier the resulting type is the so-qualified atomic type.

7 If an attempt is made to modify an object defined with a const-qualified type through use of an

[161)]The implementation can place a **const** object that is not **volatile** in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.[162)]

8 An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore, any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[163)] What constitutes an access to an object that has volatile-qualified type is implementation-defined.

9 An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that pointer.[164)] The intended use of the **restrict** qualifier (like the **register** storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).

10 If the specification of an array type includes any type qualifiers, both the array and the element type are so-qualified. If the specification of a function type includes any type qualifiers, the behavior is undefined.[165)]

11 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

12 **EXAMPLE 1** An object declared

```
extern const volatile int real_time_clock;
```

might be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

13 **EXAMPLE 2** The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

```
const struct s { int mem; } cs = { 1 };
struct s ncs;  // the object ncs is modifiable
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;      // valid
cs = ncs;      // violates modifiable lvalue constraint for =
pi = &ncs.mem; // valid
pi = &cs.mem;  // violates type constraints for =
pci = &cs.mem; // valid
pi = a[0];     // invalid:  a[0] has type "const int *"
```

14 **EXAMPLE 3** The declaration

```
_Atomic volatile int *p;
```

162)This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

163)A **volatile** declaration can be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared are not allowed to be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

164)For example, a statement that assigns a value returned by **malloc** to a single pointer establishes this association between the allocated object and the pointer.

165)This can occur with **typedef** s. Note that this rule does not apply to the **_Atomic** qualifier, and that qualifiers do not have any direct effect on the array type itself, but affect conversion rules for pointer types that reference an array type.

specifies that `p` has the type "pointer to volatile atomic int", a pointer to a volatile-qualified atomic type.

6.7.3.1 Formal definition of `restrict`

1 Let `D` be a declaration of an ordinary identifier that provides a means of designating an object `P` as a restrict-qualified pointer to type `T`.

2 If `D` appears inside a block and does not have storage class `extern`, let `B` denote the block. If `D` appears in the list of parameter declarations of a function definition, let `B` denote the associated block. Otherwise, let `B` denote the block of `main` (or the block of whatever function is called at program startup in a freestanding environment).

3 In what follows, a pointer expression `E` is said to be *based* on object `P` if (at some sequence point in the execution of `B` prior to the evaluation of `E`) modifying `P` to point to a copy of the array object into which it formerly pointed would change the value of `E`.[166] Note that "based" is defined only for expressions with pointer types.

4 During each execution of `B`, let `L` be any lvalue that has `&L` based on `P`. If `L` is used to access the value of the object `X` that it designates, and `X` is also modified (by any means), then the following requirements apply: `T` shall not be const-qualified. Every other lvalue used to access the value of `X` shall also have its address based on `P`. Every access that modifies `X` shall be considered also to modify `P`, for the purposes of this subclause. If `P` is assigned the value of a pointer expression `E` that is based on another restricted pointer object `P2`, associated with block `B2`, then either the execution of `B2` shall begin before the execution of `B`, or the execution of `B2` shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

5 Here an execution of `B` means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with `B`.

6 A translator is free to ignore any or all aliasing implications of uses of `restrict`.

7 **EXAMPLE 1** The file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of `a`, `b`, or `c`, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

8 **EXAMPLE 2** The function parameter declarations in the following example

```
void f(int n, int * restrict p, int * restrict q)
{
      while (n--  >  0)
            *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other. The translator can make this no-aliasing inference based on the parameter declarations alone, without analyzing the function body.

9 The benefit of the `restrict` qualifiers is that they enable a translator to make an effective dependence analysis of function `f` without examining any of the calls of `f` in the program. The cost is that the programmer has to examine all those calls to ensure that none give undefined behavior. For example, the second call of `f` in `g` has undefined behavior because each of `d[1]` through `d[49]` is accessed through both `p` and `q`.

```
void g(void)
{
      extern int d[100];
      f(50, d + 50, d); // valid
```

[166] In other words, `E` depends on the value of `P` itself rather than on the value of an object referenced indirectly through `P`. For example, if identifier `p` has type `(int **restrict)`, then the pointer expressions `p` and `p+1` are based on the restricted pointer object designated by `p`, but the pointer expressions `*p` and `p[1]` are not.

```
        f(50, d +  1, d); // undefined behavior
    }
```

10 **EXAMPLE 3** The function parameter declarations

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
      int i;
      for (i = 0; i  <  n; i++)
            p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. If `a` and `b` are disjoint arrays, a call of the form `h(100, a, b, b)` has defined behavior, because array `b` is not modified within function `h`.

11 **EXAMPLE 4** The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only "outer-to-inner" assignments between restricted pointers declared in nested blocks have defined behavior.

```
{
      int * restrict p1;
      int * restrict q1;
      p1 = q1; // undefined behavior
      {
            int * restrict p2 = p1; // valid
            int * restrict q2 = q1; // valid
            p1 = q2;                // undefined behavior
            p2 = q2;                // undefined behavior
      }
}
```

12 The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`.

```
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
      vector t;
      t.n = n;
      t.v = malloc(n * sizeof (float));
      return t;
}
```

13 **EXAMPLE 5** Suppose that a programmer knows that references of the form `p[i]` and `q[j]` are never aliases in the body of a function:

```
void f(int n, int *p, int *q) { /* ... */ }
```

There are several ways that this information could be conveyed to a translator using the **`restrict`** qualifier. Example 2 shows the most effective way, qualifying all pointer parameters, and can be used provided that neither `p` nor `q` becomes based on the other in the function body. A potentially effective alternative is:

```
void f(int n, int * restrict p, int * const q) { /* ... */ }
```

Again, it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use subtler reasoning: that the const-qualification of `q` precludes it becoming based on `p`. There is also a requirement that `q` is not modified, so this alternative cannot be used for the function in Example 2, as written.

14 **EXAMPLE 6** Another potentially effective alternative is:

```
void f(int n, int *p, int const * restrict q) { /* ... */ }
```

Again, it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use even subtler reasoning: that this combination of **restrict** and **const** means that objects referenced using q cannot be modified, and so no modified object can be referenced using both p and q.

15 **EXAMPLE 7** The least effective alternative is:

```
void f(int n, int * restrict p, int *q) { /* ... */ }
```

Here the translator can make the no-aliasing inference only by analyzing the body of the function and proving that `q` cannot become based on `p`. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator is required to assume that aliases are present because assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and consequently need to conservatively assume that aliases are present.

6.7.4 Function specifiers

Syntax

1 *function-specifier:*
    `inline`
    `_Noreturn`

Constraints

2 Function specifiers shall be used only in the declaration of an identifier for a function.

3 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static or thread storage duration, and shall not contain a reference to an identifier with internal linkage.

4 In a hosted environment, no function specifier(s) shall appear in a declaration of **`main`**.

Semantics

5 A function specifier may appear more than once; the behavior is the same as if it appeared only once.

6 A function declared with an **`inline`** function specifier is an *inline function*. Making a function an inline function suggests that calls to the function be as fast as possible.[167] The extent to which such suggestions are effective is implementation-defined.[168]

7 Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an **`inline`** function specifier, then it shall also be defined in the same translation unit. If all the file scope declarations for a function in a translation unit include the **`inline`** function specifier without **`extern`**, then the definition in that translation unit is an *inline definition*. An inline definition does not provide an external definition for the function and does not forbid an external definition in another translation unit. Inline definitions provide an alternative to external definitions, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.[169]

8 A function declared with a **`_Noreturn`** function specifier shall not return to its caller. The attribute `[[`**`noreturn`**`]]` provides similar semantics. The **`_Noreturn`** function specifier is an obsolescent feature (6.7.12.6).

Recommended practice

9 The implementation should produce a diagnostic message for a function declared with a **`_Noreturn`** function specifier that appears to be capable of returning to its caller.

[167]By using, for example, an alternative to the usual function call mechanism, such as "inline substitution". Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

[168]For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **`inline`** declaration.

[169]Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

10 **EXAMPLE 1** The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit.

```
inline double fahr(double t)
{
      return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
      return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double);   // creates an external definition

double convert(int is_fahr, double temp)
{
      /* A translator may perform inline substitutions */
      return is_fahr ? cels(temp): fahr(temp);
}
```

11 Note that the definition of `fahr` is an external definition because `fahr` is also declared with **extern**, but the definition of `cels` is an inline definition. Because `cels` has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either can be used for the call.

Forward references: function definitions (6.9.1).

6.7.5 Alignment specifier

Syntax

1 *alignment-specifier:*

  alignas (** *type-name* **)
  alignas (** *constant-expression* **)

Constraints

2 An alignment specifier shall appear only in the declaration specifiers of a declaration, or in the *specifier-qualifier* list of a member declaration, or in the type name of a compound literal. An alignment specifier shall not be used in conjunction with either of the storage-class specifiers **typedef** or **register**, nor in a declaration of a function or bit-field.

3 The constant expression shall be an integer constant expression. It shall evaluate to a valid fundamental alignment, or to a valid extended alignment supported by the implementation for an object of the storage duration (if any) being declared, or to zero.

4 An object shall not be declared with an over-aligned type with an extended alignment requirement not supported by the implementation for an object of that storage duration.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the type of the object or member being declared.

Semantics

6 The first form is equivalent to **alignas(alignof(** *type-name* **))**.

7 The alignment requirement of the declared object or member is taken to be the specified alignment.

An alignment specification of zero has no effect.[170] When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

8 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.6 Declarators

Syntax

1 *declarator:*
    pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator:
    identifier *attribute-specifier-sequence*$_{\text{opt}}$
    `(` *declarator* `)`
    array-declarator *attribute-specifier-sequence*$_{\text{opt}}$
    function-declarator *attribute-specifier-sequence*$_{\text{opt}}$

array-declarator:
    direct-declarator `[` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ `]`
    direct-declarator `[` `static` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* `]`
    direct-declarator `[` *type-qualifier-list* `static` *assignment-expression* `]`
    direct-declarator `[` *type-qualifier-list*$_{\text{opt}}$ `*` `]`

function-declarator:
    direct-declarator `(` *parameter-type-list*$_{opt}$ `)`

pointer:
    `*` *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$
    `*` *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *pointer*

type-qualifier-list:
    type-qualifier
    type-qualifier-list *type-qualifier*

parameter-type-list:
    parameter-list
    parameter-list `, ...`
    `...`

parameter-list:
    parameter-declaration
    parameter-list `,` *parameter-declaration*

parameter-declaration:
    attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers* *declarator*
    attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers* *abstract-declarator*$_{\text{opt}}$

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

3 A *full declarator* is a declarator that is not part of another declarator. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified.

[170] An alignment specification of zero also does not affect other alignment specifications in the same declaration.

4 In the following subclauses, consider a declaration

`T D1`

where `T` contains the declaration specifiers that specify a type *T* (such as **`int`**) and `D1` is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

5 If, in the declaration "`T D1`", `D1` has the form

identifier attribute-specifier-sequence$_{\text{opt}}$

then the type specified for *ident* is *T* and the optional attribute specifier sequence appertains to the entity that is declared.

6 If, in the declaration "`T D1`", `D1` has the form

`( D )`

then *ident* has the type specified by the declaration "`T D`". Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Implementation limits

7 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or **`void`** type, either directly or via one or more **`typedef`** s.

Forward references: array declarators (6.7.6.2), type definitions (6.7.8).

6.7.6.1 Pointer declarators

Semantics

1 If, in the declaration "`T D1`", `D1` has the form

`*` *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ `D`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*". For each type qualifier in the list, *ident* is a so-qualified pointer. The optional attribute specifier sequence appertains to the pointer and not the object pointed to.

2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

3 **EXAMPLE** The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value".

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of any object pointed to by `ptr_to_constant` cannot be modified through that pointer, but `ptr_to_constant` itself can be changed to point to another object. Similarly, the contents of the **`int`** pointed to by `constant_ptr` can be modified, but `constant_ptr` itself always points to the same location.

4 The declaration of the constant pointer `constant_ptr` can be clarified by including a definition for the type "pointer to **`int`**".

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

declares `constant_ptr` as an object that has type "const-qualified pointer to **`int`**".

6.7.6.2 Array declarators

Constraints

1 In addition to optional type qualifiers and the keyword **static**, the [and] may delimit an expression or ∗. If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

2 If an identifier is declared as having a variably modified type, it shall be an ordinary identifier (as defined in 6.2.3), have no linkage, and have either block scope or function prototype scope. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

Semantics

3 If, in the declaration "`T D1`", `D1` has one of the forms:

> `D [` *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ `]` *attribute-specifier-sequence*$_{opt}$
> `D [ static` *type-qualifier-list*$_{opt}$ *assignment-expression* `]` *attribute-specifier-sequence*$_{opt}$
> `D [` *type-qualifier-list* `static` *assignment-expression* `]` *attribute-specifier-sequence*$_{opt}$
> `D [` *type-qualifier-list*$_{opt}$ `* ]` *attribute-specifier-sequence*$_{opt}$

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[171)][172)] The optional attribute specifier sequence appertains to the array. (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword **static**.)

4 If the size is not present, the array type is an incomplete type. If the size is ∗ instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations or type names with function prototype scope[173)]; such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. (Variable length arrays with automatic storage duration are a conditional feature that implementations need not support; see 6.10.9.3.)

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by ∗; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a typeof or **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of an **alignof** operator, that expression is not evaluated.

6 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

7 **EXAMPLE 1**

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

8 **EXAMPLE 2** Note the distinction between the declarations

```
extern int *x;
extern int y[];
```

171)When several "array of" specifications are adjacent, a multidimensional array is declared.

172)The array is considered identically qualified to `T` according to 6.2.5.

173)Thus, ∗ can be used only in function declarations that are not definitions (see 6.7.6.3).

The first declares `x` to be a pointer to **`int`**; the second declares `y` to be an array of **`int`** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

9 **EXAMPLE 3** The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;      // invalid: not compatible because 4 != 6
      r = c;      // compatible, but defined behavior only if
                  // n == 6 and m == n+1
}
```

10 **EXAMPLE 4** All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **`thread_local`**, **`static`**, or **`extern`** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **`static`** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                           // invalid: file scope VLA
extern int (*p2)[n];                // invalid: file scope VM
int B[100];                         // valid: file scope but not VM

void fvla(int m, int C[m][m]);      // valid: VLA with prototype scope

void fvla(int m, int C[m][m])       // valid: adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];        // valid: block scope typedef VLA

      struct tag {
            int (*y)[n];            // invalid: y not ordinary identifier
            int z[n];               // invalid: z not ordinary identifier
      };
      int D[m];                     // valid: auto VLA
      static int E[m];              // invalid: static block scope VLA
      extern int F[m];              // invalid: F has linkage and is VLA
      int (*s)[m];                  // valid: auto pointer to VLA
      extern int (*r)[m];           // invalid: r has linkage and points to VLA
      static int (*q)[m] = &B;      // valid: q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.10).

6.7.6.3 Function declarators

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is **`register`**.

3 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

Semantics

4 If, in the declaration "`T D1`", `D1` has the form

`D (` *parameter-type-list*$_{\text{opt}}$ `)` *attribute-specifier-sequence*$_{\text{opt}}$

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* function returning the unqualified version of *T*". The optional attribute specifier sequence appertains to the function type.

5 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

6 A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

7 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

8 If the list terminates with an ellipsis (`...`), no information about the number or types of the parameters after the comma is supplied. [174)]

9 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

10 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

11 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

12 The storage class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. The optional attribute specifier sequence in a parameter declaration appertains to the parameter.

13 For a function declarator without a parameter type list: the effect is as if it were declared with a parameter type list consisting of the keyword **`void`**. A function declarator provides a prototype for the function[175)].

14 For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists shall agree in the number of parameters and in use of the final ellipsis; corresponding parameters shall have compatible types. In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

15 **EXAMPLE 1** The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function `f` with no parameters returning an **`int`**, a function `fip` with no parameters returning a pointer to an **`int`**, and a pointer `pfi` to a function with no parameters returning an **`int`**. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the pointer result to yield an **`int`**. In the declarator `(*pfi)()`, the extra parentheses are necessary to indicate that indirection through a pointer to a function

174)The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.

175)This implies that a function definition without a parameter list provides a prototype, and that subsequent calls to that function in the same translation unit are constrained not to provide any argument to the function call. Thus a definition of a function without parameter list and one that has such a list consisting of the keyword **`void`** are fully equivalent.

yields a function designator, which is then used to call the function; it returns an **int**.

16 If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions f and fip have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer pfi has block scope and no linkage.

17 **EXAMPLE 2** The declaration

```
int (*apfi[3])(int *x, int *y);
```

declares an array apfi of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers x and y are declared for descriptive purposes only and go out of scope at the end of the declaration of apfi.

18 **EXAMPLE 3** The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function fpfi that returns a pointer to a function returning an **int**. The function fpfi has two parameters: a pointer to a function returning an **int** (with one parameter of type **long int**), and an **int**. The pointer returned by fpfi points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

19 **EXAMPLE 4** The following prototype has a variably modified parameter.

```
void addscalar(int n, int m,
      double a[n][n*m+300], double x);

int main(void)
{
      double b[4][308];
      addscalar(4, 2, b, 2.17);
      return 0;
}

void addscalar(int n, int m,
      double a[n][n*m+300], double x)
{
      for (int i = 0; i  <  n; i++)
            for (int j = 0, k = n*m+300; j  <  k; j++)
                  // a is a pointer to a VLA with n*m+300 elements
                  a[i][j] += x;
}
```

20 **EXAMPLE 5** The following are all compatible function prototype declarators.

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

(Note that the last declaration also specifies that the argument corresponding to a in any call to f can be expected to be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Forward references: function definitions (6.9.1), type names (6.7.7).

6.7.7 Type names

Syntax

1 *type-name:*
   specifier-qualifier-list abstract-declarator$_{\text{opt}}$

abstract-declarator:
   pointer
   pointer$_{\text{opt}}$ *direct-abstract-declarator*

direct-abstract-declarator:
   (*abstract-declarator* **)**
   array-abstract-declarator attribute-specifier-sequence$_{\text{opt}}$
   function-abstract-declarator attribute-specifier-sequence$_{\text{opt}}$

array-abstract-declarator:
   direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**
   direct-abstract-declarator$_{\text{opt}}$ **[static** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**
   direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**
   direct-abstract-declarator$_{\text{opt}}$ **[*]**

function-abstract-declarator:
   direct-abstract-declarator$_{\text{opt}}$ **(** *parameter-type-list*$_{\text{opt}}$ **)**

Semantics

2 In several contexts, it is necessary to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[176] The optional attribute specifier sequence in a direct abstract declarator appertains to the preceding array or function type. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

3 **EXAMPLE** The constructions

```
(a)     int
(b)     int *
(c)     int *[3]
(d)     int (*)[3]
(e)     int (*)[*]
(f)     int *()
(g)     int (*)(void)
(h)     int (*const [])(unsigned int, ...)
```

name respectively the types (a) **int**, (b) pointer to **int**, (c) array of three pointers to **int**, (d) pointer to an array of three **int** s, (e) pointer to a variable length array of an unspecified number of **int** s, (f) function with no parameters returning a pointer to **int**, (g) pointer to function with no parameters returning an **int**, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type **unsigned int** and an unspecified number of other parameters, returning an **int**.

6.7.8 Type definitions

Syntax

1 *typedef-name:*
   identifier

[176] As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification", rather than redundant parentheses around the omitted identifier.

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope.

Semantics

3 In a declaration whose storage-class specifier is **`typedef`**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.6. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A **`typedef`** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

`type_ident` is defined as a typedef name with the type specified by the declaration specifiers in `T` (known as *T*), and the identifier in `D` has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of `D`. A typedef name shares the same name space as other identifiers declared in ordinary declarators. If the identifier is redeclared in an enclosed block, the type of the inner declaration shall not be inferred (6.7.9).

4 **EXAMPLE 1** After

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of `distance` is **`int`**, that of `metricp` is "pointer to function with no parameters returning **`int`**", and that of `x` and `z` is the specified structure; `zp` is a pointer to such a structure. The object `distance` has a type compatible with any other **`int`** object.

5 **EXAMPLE 2** After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type `t1` and the type pointed to by `tp1` are compatible. Type `t1` is also compatible with type **`struct`** `s1`, but not compatible with the types **`struct`** `s2`, `t2`, the type pointed to by `tp2`, or **`int`**.

6 **EXAMPLE 3** The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
      unsigned t:4;
      const t:5;
      plain r:5;
};
```

declare a typedef name `t` with type **`signed int`**, a typedef name `plain` with type **`int`**, and a structure with three bit-field members, one named `t` that contains values in the range $[0, 15]$, an unnamed const-qualified bit-field which (if it could be accessed) would contain values in the range

$[-16, +15]$, and one named `r` that contains values in one of the ranges $[0, 31]$ or $[-16, +15]$. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **`unsigned`** is a type specifier (which forces `t` to be the name of a structure member), while **`const`** is a type qualifier (which modifies `t` which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function `f` is declared with type "function returning **`signed int`** with one unnamed parameter with type pointer to function returning **`signed int`** with one unnamed parameter with type **`signed int`**", and an identifier `t` with type **`long int`**.

7 **EXAMPLE 4** On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **`signal`** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

8 **EXAMPLE 5** If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
      typedef int B[n];   // B is n ints, n evaluated now
      n += 1;
      B a;                // a is n ints, n without += 1
      int b[n];           // a and b are different sizes
      for (int i = 1; i  <  n; i++)
            a[i-1] = b[i];
}
```

6.7.9 Type inference

Constraints

1 A declaration for which the type is inferred shall contain the storage-class specifier **`auto`**.

Description

2 For such a declaration that is the definition of an object the init-declarator shall have one of the forms

direct-declarator **=** *assignment-expression*
direct-declarator **=** **{** *assignment-expression* **}**
direct-declarator **=** **{** *assignment-expression* **, }**

The declared type is the type of the assignment expression after lvalue, array to pointer or function to pointer conversion, additionally qualified by qualifiers and amended by attributes as they appear in the declaration specifiers, if any[177]. If the direct declarator is not of the form

identifier attribute-specifier-sequence$_{\text{opt}}$

optionally enclosed in balanced pairs of parentheses, the behavior is undefined.

3 **NOTE 1** Such a declaration that also defines a structure or union type violates a constraint. Here, the identifier `x` which is not ordinary but in the name space of the structure type is declared.

[177]The scope rules as described in 6.2.1 also prohibit the use of the identifier of the declarator within the assignment expression.

```
auto p = (struct { int x; } *)0;
```

Even a forward declaration of a structure tag

```
struct s;
auto p = (struct s { int x; } *)0;
```

would not change that situation. A direct use of the structure definition as the type specifier ensures the validity of the declaration.

```
struct s { int x; } * p = 0;
```

4 **EXAMPLE 1** Consider the following file scope definitions:

```
static auto a = 3.5;
auto p = &a;
```

They are interpreted as if they had been written as:

```
static double a = 3.5;
double * p = &a;
```

So effectively `a` is a **`double`** and `p` is a **`double`**`*`. Note that the restrictions on the syntax of such declarations does not allow the declarator to be `*p`, but that the final type here nevertheless is a pointer type.

5 **EXAMPLE 2** The scope of the identifier for which the type is inferred only starts after the end of the initializer (6.2.1), so the assignment expression cannot use the identifier to refer to the object or function that is declared, for example to take its address. Any use of the identifier in the initializer is invalid, even if an entity with the same name exists in an outer scope.

```
{
        double a = 7;
        double b = 9;
        {
                double b = b * b;  // undefined, uses uninitialized
                                   // variable without address
                printf("%g\n", a); // valid, uses "a" from outer scope, prints 7
                auto a    = a * a; // invalid, "a" from outer scope is not
                                   // visible during variable initialization
        }
        {
                auto b    = a * a; // valid, uses "a" from outer scope
                auto a    = b;     // valid, "a" from outer scope not visible now
                // ...
                printf("%g\n", a); // valid, uses "a" from inner scope, prints 49
        }
        // ...
}
```

6 **EXAMPLE 3** In the following, declarations of `pA` and `qA` are valid. The type of `A` after array-to-pointer conversion is a pointer type, and `qA` is a pointer to array.

```
double A[3] = { 0 };
auto pA = A;
auto qA = &A;
```

7 **EXAMPLE 4** Type inference can be used to capture the type of a call to a type-generic function. It ensures that the same type as the argument `x` is used.

```
#include <tgmath.h>
auto y = cos(x);
```

If instead the type of y is explicitly specified to a different type than x, a diagnosis of the mismatch is not enforced.

8 **EXAMPLE 5** A type-generic macro that generalizes the **div** functions (7.24.6.2) is defined and used as follows.

```
#define div(X, Y) _Generic((X)+(Y),\
       int: div,\
       long: ldiv,\
       long long: lldiv)((X), (Y))
auto z = div(x, y);
auto q = z.quot;
auto r = z.rem;
```

9 **EXAMPLE 6** Definitions of objects with inferred type are valid in all contexts that allow the initializer syntax as described. In particular they can be used to ensure type safety of **for**-loop controlling expressions.

```
for (auto i = j; i < 2*j; ++i) {
      // ...
}
```

Here, regardless of the integer rank or signedness of the type of j, i will have the non-atomic unqualified type of j. So, after lvalue conversion and possible promotion, the two operands of the < operator in the controlling expression are guaranteed to have the same type, and, in particular, the same signedness.

6.7.10 Initialization

Syntax

1 *braced-initializer:*

> **{ }**
> **{** *initializer-list* **}**
> **{** *initializer-list* **, }**

initializer:

> *assignment-expression*
> *braced-initializer*

initializer-list:

> $designation_{opt}$ *initializer*
> *initializer-list* **,** $designation_{opt}$ *initializer*

designation:

> *designator-list* **=**

designator-list:

> *designator*
> *designator-list designator*

designator:

> **[** *constant-expression* **]**
> **.** *identifier*

2 An empty brace pair ({}) is called an *empty initializer* and is referred to as *empty initialization*.

Constraints

3 No initializer shall attempt to provide a value for an object not contained within the entity being initialized.

4 The type of the entity to be initialized shall be an array of unknown size or a complete object type. An entity of variable length array type shall not be initialized except by an empty initializer. An array of unknown size shall not be initialized by an empty initializer.

5 All the expressions in an initializer for an object that has static or thread storage duration or is declared with the **`constexpr`** storage-class specifier shall be constant expressions or string literals.

6 If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

7 If a designator has the form

[*constant-expression* **]**

then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

8 If a designator has the form

. *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

Semantics

9 An initializer specifies the initial value stored in an object. For objects with atomic type additional restrictions apply, see 7.17.2 and 7.17.8.

10 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate representation even after initialization.

11 If an object that has automatic storage duration is initialized with an empty initializer, its value is the same as the initialization of a static storage duration object. Otherwise, if an object that has automatic storage duration is not initialized explicitly, its representation is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, or is initialized with an empty initializer, then *default initialization*:

— if it has pointer type, it is initialized to a null pointer;

— if it has decimal floating type, it is initialized to positive zero, and the quantum exponent is implementation-defined[178)];

— if it has arithmetic type, and it does not have decimal floating type, it is initialized to (positive or unsigned) zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

— if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

12 The initializer for a scalar shall be a single expression, optionally enclosed in braces, or it shall be an empty initializer. If the initializer is the empty initializer, the initial value is the same as the initialization of a static storage duration object. Otherwise, the initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

13 The rest of this subclause deals with initializers for objects that have aggregate or union type.

[178)] A representation with all bits zero results in a decimal floating-point zero with the most negative exponent.

14 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.

15 An array of character type may be initialized by a character string literal or UTF-8 string literal, optionally enclosed in braces. Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

16 An array with element type compatible with a qualified or unqualified version of `wchar_t`, `char16_t`, or `char32_t` may be initialized by a wide string literal with the corresponding encoding prefix (`L`, `u`, or `U`, respectively), optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

17 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.

18 Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.[179] In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization then continues forward in order, beginning with the next subobject after that described by the designator.[180]

19 Each designator list begins its description with the current object associated with the closest surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.[181] The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

20 The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[182] all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

21 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

22 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

23 If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.

24 The evaluations of the initialization list expressions are indeterminately sequenced with respect to one another and thus the order in which any side effects occur is unspecified.[183]

---

[179] If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

[180] After a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

[181] Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

[182] Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

[183] In particular, the evaluation order need not be the same as the order of subobject initialization.

25 **EXAMPLE 1** Provided that `<complex.h>` has been **`#include`**d, the declarations

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize `i` with the value 3 and `c` with the value $5.0 + i3.0$.

26 **EXAMPLE 2** The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

27 **EXAMPLE 3** The declaration

```
int y[4][3] = {
      { 1, 3, 5 },
      { 2, 4, 6 },
      { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of `y` (the array object `y[0]`), namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, so `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
      1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`.

28 **EXAMPLE 4** The declaration

```
int z[4][3] = {
      { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `z` as specified and initializes the rest with zeros.

29 **EXAMPLE 5** The declaration

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: `w[0].a[0]` is 1 and `w[1].a[0]` is 2; all the other elements are zero.

30 **EXAMPLE 6** The declaration

```
short q[4][3][2] = {
      { 1 },
      { 2, 3 },
      { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0]` does not begin with a left brace, so up to six items from the current list could be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for `q[1][0]` and `q[2][0]` do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
short q[4][3][2] = {
      1, 0, 0, 0, 0, 0,
      2, 3, 0, 0, 0, 0,
      4, 5, 6
};
```

or by:

```
short q[4][3][2] = {
      {
            { 1 },
      },
      {
            { 2, 3 },
      },
      {
            { 4, 5 },
            { 6 },
      }
};
```

in a fully bracketed form.

31 Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

32 **EXAMPLE 7** One form of initialization that completes array types involves typedef names. Given the declaration

```
typedef int A[];  // OK - declared with block scope
```

the declaration

```
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

33 **EXAMPLE 8** The declaration

```
char s[] = "abc", t[3] = "abc";
```

defines "plain" **`char`** array objects `s` and `t` whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
     t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines `p` with type "pointer to **char**" and initializes it to point to an object with type "array of **char**" with length 4 whose elements are initialized with a character string literal. If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

34 **EXAMPLE 9** Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
      [member_two] = "member two",
      [member_one] = "member one",
};
```

35 **EXAMPLE 10** Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = {.quot = 2, .rem = -1 };
```

36 **EXAMPLE 11** Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] =
      { [0].a = {1}, [1].a[0] = 2 };
```

37 **EXAMPLE 12**

```
struct T {
      int k;
      int l;
};

struct S {
      int i;
      struct T t;
};

struct T x = {.l = 43, .k = 42, };

void f(void)
{
      struct S l = { 1, .t = x, .t.l = 41, };
}
```

The value of `l.t.k` is 42, because implicit initialization does not override explicit initialization.

38 **EXAMPLE 13** Space can be "allocated" from both ends of an array by using a single designator:

```
int a[MAX] = {
      1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

39 In the above, if `MAX` is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

40 **EXAMPLE 14** Any member of a union can be initialized:

```
union { /* ... */ } u = {.any_member = 42 };
```

Forward references: common definitions `<stddef.h>` (7.21).

6.7.11 Static assertions

Syntax

1 *static_assert-declaration:*

  static_assert (*constant-expression* **,** *string-literal* **) ;**
  static_assert (*constant-expression* **) ;**

Constraints

2 The constant expression shall compare unequal to 0.

Semantics

3 The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message which should include the text of the string literal, if present.

Forward references: diagnostics (7.2).

6.7.12 Attributes

1 Attributes specify additional information for various source constructs such as types, variables, identifiers, or blocks. They are identified by an *attribute token*, which can either be a *attribute prefixed token* (for implementation-specific attributes) or a *standard attribute* specified by an identifier (for attributes specified in this document).

2 Support for any of the standard attributes specified in this document is implementation-defined and optional. For an attribute token (including an attribute prefixed token) not specified in this document, the behavior is implementation-defined. Any attribute token that is not supported by the implementation is ignored.

3 Attributes are said to appertain to some source construct, identified by the syntactic context where they appear, and for each individual attribute, the corresponding clause constrains the syntactic context in which this appertainance is valid. The attribute specifier sequence appertaining to some source construct shall contain only attributes that are allowed to apply to that source construct.

4 In all aspects of the language, a standard attribute specified by this document as an identifier `attr` and an identifier of the form `__attr__` shall behave the same when used as an attribute token, except for the spelling.[184)]

Recommended practice

5 It is recommended that implementations support all standard attributes as defined in this document.

6.7.12.1 General

Syntax

1 *attribute-specifier-sequence:*

  attribute-specifier-sequence$_{\text{opt}}$ *attribute-specifier*

attribute-specifier:

  [[*attribute-list* **]]**

attribute-list:

  attribute$_{\text{opt}}$
  attribute-list **,** *attribute*$_{\text{opt}}$

attribute:

  attribute-token *attribute-argument-clause*$_{\text{opt}}$

attribute-token:

  standard-attribute
  attribute-prefixed-token

---

184) Thus, the attributes `[[nodiscard]]` and `[[__nodiscard__]]` can be freely interchanged. Implementations are encouraged to behave similarly for attribute tokens (including attribute prefixed tokens) they provide.

standard-attribute:
    identifier

attribute-prefixed-token:
    attribute-prefix **::** *identifier*
attribute-prefix:
    identifier
attribute-argument-clause:
    (*balanced-token-sequence*$_{opt}$ **)**
balanced-token-sequence:
    balanced-token
    balanced-token-sequence balanced-token
balanced-token:
    (*balanced-token-sequence*$_{opt}$ **)**
    [*balanced-token-sequence*$_{opt}$ **]**
    { *balanced-token-sequence*$_{opt}$ **}**
    any token other than a parenthesis, a bracket, or a brace

Constraints

2 The identifier in a standard attribute shall be one of:

`deprecated`	`maybe_unused`	`noreturn`	`unsequenced`
`fallthrough`	`nodiscard`	`_Noreturn`	`reproducible`

Semantics

3 An attribute specifier that contains no attributes has no effect. The order in which attribute tokens appear in an attribute list is not significant. If a keyword (6.4.1) that satisfies the syntactic requirements of an identifier (6.4.2) is contained in an attribute token, it is considered an identifier. A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute.[185)]

4 **NOTE 1** For each standard attribute, the form of the balanced token sequence, if any, will be specified.

Recommended Practice

5 Each implementation should choose a distinctive name for the attribute prefix in an attribute prefixed token. Implementations should not define attributes without an attribute prefix unless it is a standard attribute as specified in this document.

6 **EXAMPLE 1** Suppose that an implementation chooses the attribute prefix `hal` and provides specific attributes named `daisy` and `rosie`.

```
[[deprecated, hal::daisy]] double nine1000(double);
[[deprecated]] [[hal::daisy]] double nine1000(double);
[[deprecated]] double nine1000 [[hal::daisy]] (double);
```

Then all the following declarations should be equivalent aside from the spelling:

```
[[__deprecated__, __hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] [[__hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] double nine1000 [[__hal__::__daisy__]] (double);
```

These use the alternate spelling that is required for all standard attributes and recommended for prefixed attributes. These may be better-suited for use in header files, where the use of the alternate spelling avoids naming conflicts with user-provided macros.

185)Standard attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.

7 **EXAMPLE 2** For the same implementation, the following two declarations are equivalent, because the ordering inside attribute lists is not important.

```
[[hal::daisy, hal::rosie]] double nine999(double);
[[hal::rosie, hal::daisy]] double nine999(double);
```

On the other hand the following two declarations are not equivalent, because the ordering of different attribute specifiers may affect the semantics.

```
[[hal::daisy]] [[hal::rosie]] double nine999(double);
[[hal::rosie]] [[hal::daisy]] double nine999(double); // may have different semantics
```

6.7.12.2 The **nodiscard** attribute

Constraints

1 The **nodiscard** attribute shall be applied to the identifier in a function declaration or to the definition of a structure, union, or enumeration type. If an attribute argument clause is present, it shall have the form:

(*string-literal* **)**

Semantics

2 The **__has_c_attribute** conditional inclusion expression (6.10.1) shall return the value 202003L when given **nodiscard** as the pp-tokens operand.

3 A name or entity declared without the **nodiscard** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.

Recommended Practice

4 A nodiscard call is a function call expression that calls a function previously declared with attribute **nodiscard**, or whose return type is a structure, union, or enumeration type marked with attribute **nodiscard**. Evaluation of a nodiscard call as a void expression (6.8.3) is discouraged unless explicitly cast to **void**. Implementations are encouraged to issue a diagnostic in such cases. This is typically because immediately discarding the return value of a **nodiscard** call has surprising consequences.

5 The diagnostic message should include text provided by the string literal within the attribute argument clause of any **nodiscard** attribute applied to the name or entity.

6 **EXAMPLE 1**

```
struct [[nodiscard]] error_info { /*...*/ };
struct error_info enable_missile_safety_mode(void);
void launch_missiles(void);
void test_missiles(void) {
      enable_missile_safety_mode();
      launch_missiles();
}
```

A diagnostic for the call to `enable_missile_safety_mode` is encouraged.

7 **EXAMPLE 2**

```
[[nodiscard]] int important_func(void);
void call(void) {
      int i = important_func();
}
```

No diagnostic for the call to `important_func` is encouraged despite the value of `i` not being used.

8 **EXAMPLE 3**

```
      [[nodiscard("must check armed state")]]
```

```
    bool arm_detonator(int within);

    void call(void) {
          arm_detonator(3);
          detonate();
    }
```

A diagnostic for the call to `arm_detonator` using the string literal `"must check armed state"` from the attribute argument clause is encouraged.

6.7.12.3 The `maybe_unused` attribute

Constraints

1 The **`maybe_unused`** attribute shall be applied to the declaration of a structure, a union, a **`typedef`** name, a variable, a structure or union member, a function, an enumeration, an enumerator, or a label. No attribute argument clause shall be present.

Semantics

2 The **`maybe_unused`** attribute indicates that a name or entity is possibly intentionally unused.

3 The **`__has_c_attribute`** conditional inclusion expression (6.10.1) shall return the value 202106L when given **`maybe_unused`** as the pp-tokens operand.

A name or entity declared without the **`maybe_unused`** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

Recommended Practice

4 For an entity marked **`maybe_unused`**, implementations are encouraged not to emit a diagnostic that the entity is unused, or that the entity is used despite the presence of the attribute.

5 **EXAMPLE**

```
[[maybe_unused]] void f([[maybe_unused]] int i) {
      [[maybe_unused]] int j = i + 100;
      assert(j);
}
```

Implementations are encouraged not to diagnose that `j` is unused, even if **`NDEBUG`** is defined.

6.7.12.4 The `deprecated` attribute

Constraints

1 The **`deprecated`** attribute shall be applied to the declaration of a structure, a union, a **`typedef`** name, a variable, a structure or union member, a function, an enumeration, or an enumerator.

2 If an attribute argument clause is present, it shall have the form:

(*string-literal* **)**

Semantics

3 The **`deprecated`** attribute can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [186)]

4 The **`__has_c_attribute`** conditional inclusion expression (6.10.1) shall return the value 201904L when given **`deprecated`** as the pp-tokens operand.

5 A name or entity declared without the **`deprecated`** attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

186)In particular, **`deprecated`** is appropriate for names and entities that are obsolescent, insecure, unsafe, or otherwise unfit for purpose.

Recommended Practice

6 Implementations should use the **deprecated** attribute to produce a diagnostic message in case the program refers to a name or entity other than to declare it, after a declaration that specifies the attribute, when the reference to the name or entity is not within the context of a related deprecated entity. The diagnostic message should include text provided by the string literal within the attribute argument clause of any **deprecated** attribute applied to the name or entity.

7 **EXAMPLE**

```
struct [[deprecated]] S {
      int a;
};

enum [[deprecated]] E1 {
      one
};

enum E2 {
      two [[deprecated("use 'three' instead")]],
      three
};

[[deprecated]] typedef int Foo;

void f1(struct S s) { // Diagnose use of S
      int i = one; // Diagnose use of E1
      int j = two; // Diagnose use of two: "use 'three' instead"
      int k = three;
      Foo f; // Diagnose use of Foo
}

[[deprecated]] void f2(struct S s) {
      int i = one;
      int j = two;
      int k = three;
      Foo f;
}

struct [[deprecated]] T {
      Foo f;
      struct S s;
};
```

Implementations are encouraged to diagnose the use of deprecated entities within a context which is not itself deprecated, as indicated for function `f1`, but not to diagnose within function `f2` and `struct T`, as they are themselves deprecated.

6.7.12.5 The `fallthrough` attribute

Constraints

1 The attribute token **fallthrough** shall only appear in an attribute declaration (6.7); such a declaration is a *fallthrough declaration*. No attribute argument clause shall be present. A fallthrough declaration may only appear within an enclosing **switch** statement (6.8.4.2). The next block item (6.8.2) that would be encountered after a fallthrough declaration shall be a **case** label or **default** label associated with the smallest enclosing **switch** statement.

Semantics

2 The **__has_c_attribute** conditional inclusion expression (6.10.1) shall return the value 201910L when given **fallthrough** as the pp-tokens operand.

Recommended Practice

3 The use of a fallthrough declaration is intended to suppress a diagnostic that an implementation might otherwise issue for a **case** or **default** label that is reachable from another **case** or **default** label along some path of execution. Implementations are encouraged to issue a diagnostic if a fallthrough declaration is not dynamically reachable.

4 **EXAMPLE**

```
void f(int n) {
      void g(void), h(void), i(void);
      switch (n) {
      case 1: /* diagnostic on fallthrough discouraged */
      case 2:
            g();
            [[fallthrough]];
      case 3: /* diagnostic on fallthrough discouraged */
            h();
      case 4: /* fallthrough diagnostic encouraged */
            i();
            [[fallthrough]]; /* constraint violation */
      }
}
```

6.7.12.6 The `noreturn` and `_Noreturn` attributes

Description

1 When `_Noreturn` is used as an attribute token (instead of a function specifier), the constraints and semantics are identical to that of the `noreturn` attribute token. Use of `_Noreturn` as an attribute token is an obsolescent feature[187].

Constraints

2 The `noreturn` attribute shall be applied to the identifier in a function declaration. No attribute argument clause shall be present.

Semantics

3 The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the behavior is undefined.

4 If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined.

5 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value 202202L when given `noreturn` as the pp-tokens operand.

Recommended Practice

6 The implementation should produce a diagnostic message for a function declared with a `noreturn` attribute that appears to be capable of returning to its caller.

7 **EXAMPLE**

```
[[noreturn]] void f(void) {
      abort(); // ok
}

[[noreturn]] void g(int i) { // causes undefined behavior if i <= 0
      if (i > 0) abort();
}
```

[187] `[[_Noreturn]]` and `[[noreturn]]` are equivalent attributes to support code that includes `<stdnoreturn.h>`, because that header defines `noreturn` as a macro that expands to `_Noreturn`.

```
[[noreturn]] int h(void);
```

Implementations are encouraged to diagnose the definition of `g()` because it is capable of returning to its caller. Implementations are similarly encouraged to diagnose the declaration of `h()` because it appears capable of returning to its caller due to the non-**`void`** return type.

6.7.12.7 Standard attributes `unsequenced` and `reproducible` for function types

Constraints

1 The identifier in a standard function type attribute shall be one of:

`unsequenced` `reproducible`

2 An attribute for a function type shall be applied to a function declarator[188) or to a type specifier that has a function type. The corresponding attribute is a property of the referred function type.[189) No attribute argument clause shall be present.

Description

3 The main purpose of the function type properties and attributes defined in this clause is to provide the translator with information about the access of objects by a function such that certain properties of function calls can be deduced; the properties distinguish read operations (stateless and independent) and write operations (effectless, idempotent and reproducible) or a combination of both (unsequenced). Although semantically attached to a function type, the attributes described are not part of the prototype of a such annotated function, and redeclarations and conversions that drop such an attribute are valid and constitute compatible types. Conversely, if a definition that does not have the asserted property is accessed by a function declaration or a function pointer with a type that has the attribute, the behavior is undefined.[190)

4 To allow reordering of calls to functions as they are described here, possible access to objects with a lifetime that starts before or ends after a call has to be restricted; effects on all objects that are accessed during a function call are restricted to the same thread as the call and the based-on relation between pointer parameters and lvalues (6.7.3.1) models the fact that objects do not change inadvertently during the call. In the following, an operation is said to be sequenced *during* a function call if it is sequenced after the start of the function call[191) and before the call terminates. An object definition of an object X in a function f *escapes* if an access to X happens while no call to f is active. An object is *local* to a call to a function f if its lifetime starts and ends during the call or if it is defined by f but does not escape. A function call and an object X *synchronize* if all accesses to X that are not sequenced during the call happen before or after the call. Execution state that is described in the library clause, such as the floating-point environment, conversion state, locale, input/output streams, external files or **`errno`** account as objects; operations that allow to query this state, even indirectly, account as lvalue conversions, and operations that allow to change this state account as store operations.

5 A function definition f is *stateless* if any definition of an object of static or thread storage duration in f or in a function that is called by f is **`const`** but not **`volatile`** qualified.

6 An object X is *observed* by a function call if both synchronize, if X is not local to the call, if X has a lifetime that starts before the function call and if an access of X is sequenced during the call; the last value of X, if any, that is stored before the call is said to be the value of X that is observed by the call. A function pointer value f is *independent* if for any object X that is observed by some call to f

188)That is, they appear in the attributes right after the closing parenthesis of the parameter list, independently if the function type is, for example, used directly to declare a function or if it is used in a pointer to function type.

189)If several declarations of the same function or function pointer are visible, regardless whether an attribute is present at several or just one of the declarators, it is attached to the type of the corresponding function definition, function pointer object, or function pointer value.

190)That is, the fact that a function has one of these properties is in general not determined by the specification of the translation unit in which it is found; other translation units and specific run time conditions also condition the possible assertion of the properties.

191)The initializations of the parameters is sequenced during the function call.

through an lvalue that is not based on a parameter of the call, then all accesses to X in all calls to f during the same program execution observe the same value; otherwise if the access is based on a pointer parameter, there shall be a unique such pointer parameter P such that any access to X shall be to an lvalue that is based on P. A function definition is independent if the derived function pointer value is independent.

7 A store operation to an object X that is sequenced during a function call such that both synchronize is said to be *observable* if X is not local to the call, if the lifetime of X ends after the call, if the stored value is different from the value observed by the call, if any, and if it is the last value written before the termination of the call. An evaluation of a function call[192] is *effectless* if any store operation that is sequenced during the call is the modification of an object that synchronizes with the call; if additionally the operation is observable, there shall be a unique pointer parameter P of the function such that any access to X shall be to an lvalue that is based on P. A function pointer value f is effectless if any evaluation of a function call that calls f is effectless. A function definition is effectless if the derived function pointer value is effectless.

8 An evaluation E is *idempotent* if a second evaluation of E can be sequenced immediately after the original one without changing the resulting value, if any, or the observable state of the execution. A function pointer value f is idempotent if any evaluation of a function call[193] that calls f is idempotent. A function definition is idempotent if the derived function pointer value is idempotent.

9 A function is *reproducible* if it is effectless and idempotent; it is *unsequenced* if it is stateless, effectless, idempotent and independent.[194]

10 **NOTE 1** The synchronization requirements with respect to any accessed object X for the independence of functions provide boundaries up to which a function call may safely be reordered without changing the semantics of the program. If X is `const` but not `volatile` qualified the reordering is unconstrained. If it is an object that is conditioned in an initialization phase, for a single threaded program a synchronization is provided by the sequenced before relation and the reordering may, in principle, move the call just after the initialization. For a multi-threaded program, synchronization guarantees can be given by calls to synchronizing functions of the `<threads.h>` header or by an appropriate call to `atomic_thread_fence` at the end of the initialization phase. If a function is known to be independent or effectless, adding `restrict` qualifications to the declarations of all pointer parameters does not change the semantics of any call. Similarly, changing the memory order to `memory_order_relaxed` for all atomic operations during a call to such a function preserves semantics.

11 **NOTE 2** In general the functions provided by the `<math.h>` header do not have the properties that are defined above; many of them change the floating-point state or `errno` when they encounter an error (so they have observable side effects) and the results of most of them depend on execution wide state such as the rounding direction mode (so they are not independent). Whether a particular C library function is reproducible or unsequenced additionally often depends on properties of the implementation, such as implementation-defined behavior for certain error conditions.

Recommended Practice

12 If possible, it is recommended that implementations diagnose if an attribute of this clause is applied to a function definition that does not have the corresponding property. It is recommended that applications that assert the independent or effectless properties for functions qualify pointer parameters with `restrict`.

Forward references: errors `<errno.h>` (7.5), floating-point environment `<fenv.h>` (7.6), localization `<locale.h>` (7.11), mathematics `<math.h>` (7.12), fences (7.17.4), input/output `<stdio.h>` (7.23), threads `<threads.h>` (7.28), extended multibyte and wide character utilities `<wchar.h>` (7.31).

6.7.12.7.1 The `reproducible` type attribute

Description

1 The `reproducible` type attribute asserts that a function or pointed-to function with that type is reproducible.

[192]This considers the evaluation of the function call itself, not the evaluation of a full function call expression. Such an evaluation is sequenced after all evaluations that determine f and the call arguments, if any, have been performed.

[193]This considers the evaluation of the function call itself, not the evaluation of a full function call expression. Such an evaluated is sequenced after all evaluations that determine f and the call arguments, if any, have been performed.

[194]A function call of an unsequenced function can be executed as early as the function pointer value, the values of the arguments and all objects that are accessible through them, and all values of globally accessible state have been determined, and it can be executed as late as the arguments and the objects they possibly target are unchanged and as any of its return value or modified pointed-to arguments are accessed.

2 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value 202207L when given `reproducible` as the pp-tokens operand.

3 **EXAMPLE 1** The attribute in the following function declaration asserts that two consecutive calls to the function will result in the same return value. Changes to the abstract state during the call are possible as long as they are not observable, but no other side effects will occur. Thus the function definition may for example use local objects of static or thread storage duration to keep track of the arguments for which the function has been called and cache their computed return values.

```
size_t hash(char const[static 32]) [[reproducible]];
```

6.7.12.7.2 The `unsequenced` type attribute

Description

1 The `unsequenced` type attribute asserts that a function or pointed-to function with that type is unsequenced.

2 The `__has_c_attribute` conditional inclusion expression (6.10.1) shall return the value 202207L when given `unsequenced` as the pp-tokens operand.

3 **NOTE 1** The unsequenced type attribute asserts strong properties for the such typed function, in particular that certain sequencing requirements for function calls can be relaxed without affecting the state of the abstract machine. Thereby, calls to such functions are natural candidates for optimization techniques such as common subexpression elimination, local memoization or lazy evaluation.

4 **NOTE 2** A proof of validity of the annotation of a function type with the `unsequenced` attribute may depend on the property if a derived function pointer escapes the translation unit or not. For a function with internal linkage where no function pointer escapes the translation unit, all calling contexts are known and it is possible, in principle, to prove that no control flow exists such that a library function is called with arguments that trigger an exceptional condition. For a function with external linkage such a proof may not be possible and the use of such a function then has to ensure that no exceptional condition results from the provided arguments.

5 **NOTE 3** The unsequenced property does not necessarily imply that the function is reentrant or that calls can be executed concurrently. This is because an unsequenced function can read from and write to objects of static storage duration, as long as no change is observable after a call terminates.

6 **EXAMPLE 1** The attribute in the following function declaration asserts that it doesn't depend on any modifiable state of the abstract machine. Calls to the function can be executed out of sequence before the return value is needed and two calls to the function with the same argument value will result in the same return value.

```
bool tendency(signed char) [[unsequenced]];
```

Therefore such a call for a given argument value needs only to be executed once and the returned value can be reused when appropriate. For example, calls for all possible argument values can be executed during program startup and tabulated.

7 **EXAMPLE 2** The attribute in the following function declaration asserts that it doesn't depend on any modifiable state of the abstract machine. Within the same thread, calls to the function can be executed out of sequence before the return value is needed and two calls to the function will result in the same pointer return value. Therefore such a call needs only to be executed once in a given thread and the returned pointer value can be reused when appropriate. For example, a single call can be executed during thread startup and the return value `p` and the value of the object `*p` of type `toto const` can be cached.

```
typedef struct toto toto;
toto const* toto_zero(void) [[unsequenced]];
```

8 **EXAMPLE 3** The unsequenced property of a function f can be locally asserted within a function g that uses it. For example the library function `sqrt` is in generally not unsequenced because a negative argument will raise a domain error and because the result may depend on the rounding mode. Nevertheless in contexts similar to the following function a user can prove that it will not be called with invalid arguments, and, that the floating-point environment has the same value for all calls.

```
#include <math.h>
#include <fenv.h>

inline double distance (double const x[static 2]) [[reproducible]] {
      #pragma FP_CONTRACT OFF
      #pragma FENV_ROUND  FE_TONEAREST
      // We assert that sqrt will not be called with invalid arguments
      // and the result only depends on the argument value.
      extern typeof(sqrt) [[unsequenced]] sqrt;
      return sqrt(x[0]*x[0] + x[1]*x[1]);
}
```

The function `distance` potentially has the side effect of changing the floating-point environment. Nevertheless the floating environment is thread local, thus a change to that state outside the function is sequenced with the change within and additionally the observed value is restored when the function returns. Thus this side effect is not observable for a caller. Overall the function `distance` is stateless, effectless and idempotent and in particular it is reproducible as the attribute indicates. Because the function can be called in a context where the floating-point environment has different state, `distance` is not independent and thus it is also not unsequenced. Nevertheless, adding an unsequenced attribute where this is justified may introduce optimization opportunities.

```
double g (double y[static 1], double const x[static 2]) {
      // We assert that distance will not see different states of the floating
      // point environment.
      extern double distance (double const x[static 2]) [[unsequenced]];
      y[0] = distance(x);
      ...
      return distance(x);   // replacement by y[0] is valid
}
```

6.8 Statements and blocks

Syntax

1 *statement:*
labeled-statement
unlabeled-statement

unlabeled-statement:
expression-statement
attribute-specifier-sequence$_{\text{opt}}$ *primary-block*
attribute-specifier-sequence$_{\text{opt}}$ *jump-statement*

primary-block:
compound-statement
selection-statement
iteration-statement

secondary-block:
statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence. The optional attribute specifier sequence appertains to the respective statement.

3 A *block* is either a primary block, a secondary block, or the block associated with a function definition; it allows a set of declarations and statements to be grouped into one syntactic unit. Whenever a block *B* appears in the syntax production as part of the definition of an enclosing block *A*, scopes of identifiers and lifetimes of objects that are associated with *B* do not extend to the parts of *A* that are outside of *B*. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (the representation of objects without an initializer becomes indeterminate) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 **NOTE 1** Each of the following is a full expression:

— a full declarator for a variably modified type,
— an initializer that is not part of a compound literal,
— the expression in an expression statement,
— the controlling expression of a selection statement (**`if`** or **`switch`**),
— the controlling expression of a **`while`** or **`do`** statement,
— each of the (optional) expressions of a **`for`** statement,
— the (optional) expression in a **`return`** statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *label:*
attribute-specifier-sequence$_{\text{opt}}$ *identifier* **`:`**

attribute-specifier-sequence$_{opt}$ **`case`** *constant-expression* **`:`**
attribute-specifier-sequence$_{opt}$ **`default :`**

labeled-statement:
label statement

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement. Further constraints on such labels are discussed under the **`switch`** statement.

3 Label names shall be unique within a function.

Semantics

4 Any statement may be preceded by a prefix that declares an identifier as a label name. The optional attribute specifier sequence appertains to the label. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Forward references: the **`goto`** statement (6.8.6.1), the **`switch`** statement (6.8.4.2) .

6.8.2 Compound statement

Syntax

1 *compound-statement:*
`{` *block-item-list*$_{opt}$ **`}`**

block-item-list:
block-item
block-item-list block-item

block-item:
declaration
unlabeled-statement
label

Semantics

2 A *compound statement* that is a function body together with the parameter type list and the optional attribute specifier sequence between them forms the block associated with the function definition in which it appears. Otherwise, it is a block that is different from any other block. A label shall be translated as if it were followed by a null statement.

6.8.3 Expression and null statements

Syntax

1 *expression-statement:*
expression$_{opt}$ **`;`**
attribute-specifier-sequence expression **`;`**

Semantics

2 The attribute specifier sequence appertains to the expression. The expression in an expression statement is evaluated as a void expression for its side effects.[195)]

3 A *null statement* (consisting of just a semicolon) performs no operations.

4 **EXAMPLE 1** If a function call is evaluated as an expression statement for its side effects only, the discarding of its value can be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
```

[195)]Such as assignments, and function calls which have side effects.

```
(void)p(0);
```

5 **EXAMPLE 2** In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

Forward references: iteration statements (6.8.5).

6.8.4 Selection statements

Syntax

1 *selection-statement:*

  if (*expression* **)** *secondary-block*
  if (*expression* **)** *secondary-block* **else** *secondary-block*
  switch (*expression* **)** *secondary-block*

Semantics

2 A selection statement selects among a set of secondary blocks depending on the value of a controlling expression.

6.8.4.1 The `if` statement

Constraints

1 The controlling expression of an **if** statement shall have scalar type.

Semantics

2 In both forms, the first substatement is executed if the expression compares unequal to 0. In the **else** form, the second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

3 An **else** is associated with the lexically nearest preceding **if** that is allowed by the syntax.

6.8.4.2 The `switch` statement

Constraints

1 The controlling expression of a **switch** statement shall have integer type.

2 If a **switch** statement has an associated **case** or **default** label within the scope of an identifier with a variably modified type, the entire **switch** statement shall be within the scope of that identifier.[196)]

3 The expression of each **case** label shall be an integer constant expression and no two of the **case** constant expressions associated to the same **switch** statement shall have the same value after conversion. There may be at most one **default** label associated to a **switch** statement. (Any enclosed **switch** statement may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.)

Semantics

4 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement.

5 The integer promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement or declaration

196)That is, the declaration either precedes the **switch** statement, or it follows the last **case** or **default** label associated with the **switch** that is in the block containing the declaration.

following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the statement or declaration following the default label. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

Implementation limits

6 As discussed in 5.2.4.1, the implementation may limit the number of **case** values in a **switch** statement.

7 **EXAMPLE** In the artificial program fragment

```
switch (expr)
{
      int i = 4;
      f(i);
case 0:
      i = 17;
      /* falls through into default code */
default:
      printf("%d\n", i);
}
```

the object whose identifier is `i` exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an object with an indeterminate representation. Similarly, the call to the function `f` cannot be reached.

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*

  while (*expression* **)** *secondary-block*
  do *secondary-block* **while (** *expression* **) ;**
  for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *secondary-block*
  for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *secondary-block*

Constraints

2 The controlling expression of an iteration statement shall have scalar type.

3 The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**.

Semantics

4 An iteration statement causes a secondary block called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump[197].

5 An iteration statement may be assumed by the implementation to terminate if its controlling expression is not a constant expression[198], and none of the following operations are performed in its body, controlling expression or (in the case of a **for** statement) its *expression-3*[199]:

— input/output operations

— accessing a volatile object

— synchronization or atomic operations.

[197]Code jumped over is not executed. In particular, the controlling expression of a **for** or **while** statement is not evaluated before entering the loop body, nor is *clause-1* (6.8.5.3) of a **for** statement.

[198]An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

[199]This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.

6.8.5.1 The `while` statement

1 The evaluation of the controlling expression takes place before each execution of the loop body.

6.8.5.2 The `do` statement

1 The evaluation of the controlling expression takes place after each execution of the loop body.

6.8.5.3 The `for` statement

1 The statement

```
for (clause-1; expression-2; expression-3) statement
```

behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.[200)]

2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

6.8.6 Jump statements

Syntax

1 *jump-statement:*
    `goto` *identifier* **`;`**
    `continue ;`
    `break ;`
    `return` *expression*$_{\text{opt}}$ **`;`**

Semantics

2 A jump statement causes an unconditional jump to another place.

6.8.6.1 The `goto` statement

Constraints

1 The identifier in a **`goto`** statement shall name a label located somewhere in the enclosing function. A **`goto`** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

Semantics

2 A **`goto`** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

3 **EXAMPLE 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by **`continue`** statements, for example.)

```
/* ... */
```

200) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

```
    goto first_time;
    for (;;) {
          // determine next operation
          /* ... */
          if (need to reinitialize) {
                // reinitialize-only code
                /* ... */
          first_time:
                // general initialization code
                /* ... */
                continue;
          }
          // handle other operations
          /* ... */
    }
```

4 **EXAMPLE 2** A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```
    goto lab3;              // invalid:  going INTO scope of VLA.
    {
          double a[n];
          a[j] = 4.4;
    lab3:
          a[j] = 3.3;
          goto lab4;        // valid:  going WITHIN scope of VLA.
          a[j] = 5.5;
    lab4:
          a[j] = 6.6;
    }
    goto lab4;              // invalid:  going INTO scope of VLA.
```

6.8.6.2 The continue statement

Constraints

1 A **continue** statement shall appear only in or as a loop body.

Semantics

2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:
}
```

```
do {
   /* ...  */
   continue;
   /* ...  */
contin:;
} while (/* ...  */);
```

```
for (/* ...  */) {
   /* ...  */
   continue;
   /* ...  */
contin:
}
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to `goto contin;` [201].

6.8.6.3 The break statement

Constraints

1 A **break** statement shall appear only in or as a switch body or loop body.

[201]Following the `contin:` label in the 2nd example is a null statement. The null statement in the first and third example is implied by the label (6.8.2).

Semantics

2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

6.8.6.4 The return statement

Constraints

1 A **return** statement with an expression shall not appear in a function whose return type is **void**. A **return** statement without an expression shall only appear in a function whose return type is **void**.

Semantics

2 A **return** statement terminates execution of the current function and returns control to its caller. A function may have any number of **return** statements.

3 If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[202)]

4 **EXAMPLE** In:

```
struct s { double i; } f(void);
union {
      struct {
            int f1;
            struct s f2;
      } u1;
      struct {
            struct s f3;
            int f4;
      } u2;
} g;

struct s f(void)
{
      return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

[202)]The **return** statement is not an assignment. The overlap restriction of 6.5.16.1 does not apply to the case of function return. The representation of floating-point values can have wider range or precision than implied by the type; a cast can be used to remove this extra range and precision.

6.9 External definitions

Syntax

1 *translation-unit:*
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

Constraints

2 The storage-class specifier **`register`** shall not appear in the declaration specifiers in an external declaration. The storage-class specifier **`auto`** shall only appear in the declaration specifiers in an external declaration if the type is inferred.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression there shall be exactly one external definition for the identifier in the translation unit, unless it is:

— part of the operand of a **`sizeof`** operator whose result is an integer constant;

— part of the operand of an **`alignof`** operator whose result is an integer constant;

— or, part of the operand of any typeof operator whose result is not a variably modified type.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a typeof operator whose result is not a variably modified type, or a **`sizeof`** or **`alignof`** operator whose result is an integer constant expression), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[203)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
  attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers declarator function-body*

function-body:
  compound-statement

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.

3 The return type of a function shall be **`void`** or a complete object type other than array type.

[203)] Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **`extern`** or **`static`**.

5 If the parameter list consists of a single parameter of type **`void`**, the parameter declarator shall not include an identifier.

Semantics

6 The optional attribute specifier sequence in a function definition appertains to the function.

7 The declarator in a function definition specifies the name of the function being defined and the types (and optionally the names) of all the parameters; the declarator also serves as a function prototype for later calls to the same function in the same translation unit. The type of each parameter is adjusted as described in 6.7.6.3.

8 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

9 The parameter type list, the attribute specifier sequence of the declarator that follows the parameter type list, and the compound statement of the function body form a single block.[204] Each parameter has automatic storage duration; its identifier, if any[205], is an lvalue.[206] The layout of the storage for parameters is unspecified.

10 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

11 After all parameters have been assigned, the compound statement of the function body is executed.

12 Unless otherwise specified, if the } that terminates the function body is reached, and the value of the function call is used by the caller, the behavior is undefined.

13 **NOTE 1** In a function definition, the type of the function and its prototype cannot be inherited from a typedef:

```
typedef int F(void);            // type F is "function with no parameters
                                // returning int"
F f, g;                         // f and g both have type compatible with F
F f { /* ... */ }               // WRONG: syntax/constraint error
F g() { /* ... */ }             // WRONG: declares that g returns a function
int f(void) { /* ... */ }       // RIGHT: f has type compatible with F
int g() { /* ... */ }           // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }        // e returns a pointer to a function
F *((e))(void) { /* ... */ }    // same:  parentheses irrelevant
int (*fp)(void);                // fp points to a function that has type F
F *Fp;                          // Fp points to a function that has type F
```

14 **EXAMPLE 1** In the following:

```
extern int max(int a, int b)
{
      return a  >  b ? a: b;
}
```

`extern` is the storage-class specifier and **`int`** is the type specifier; `max(int a, int b)` is the function declarator; and

```
{ return a  >  b ? a: b; }
```

[204]The visibility scope of a parameter in a function definition starts when its declaration is completed, extends to following parameter declarations, to possible attributes that follow the parameter type list, and then to the entire function body. The lifetime of each instance of a parameter starts when the declaration is evaluated starting a call and ends when that call terminates.

[205]A parameter that has no declared name is inaccessible within the function body.

[206]A parameter identifier cannot be redeclared in the function body except in an enclosed block.

is the function body.

15 **EXAMPLE 2** To pass one function to another, one might say

```
        int f(void);
        /* ... */
        g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
        /* ... */
        (*funcp)(); /* or funcp(); ...*/
}
```

or, equivalently,

```
void g(int func(void))
{
        /* ... */
        func(); /* or (*func)(); ...*/
}
```

6.9.2 External object definitions

Semantics

1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **`static`**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to `{ 0 }`.

3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

4 **EXAMPLE 1**

```
int i1 = 1;          // definition, external linkage
static int i2 = 2;  // definition, internal linkage
extern int i3 = 3;  // definition, external linkage
int i4;             // tentative definition, external linkage
static int i5;      // tentative definition, internal linkage

int i1;             // valid tentative definition, refers to previous
int i2;             // 6.2.2 renders undefined, linkage disagreement
int i3;             // valid tentative definition, refers to previous
int i4;             // valid tentative definition, refers to previous
int i5;             // 6.2.2 renders undefined, linkage disagreement

extern int i1;      // refers to previous, whose linkage is external
extern int i2;      // refers to previous, whose linkage is internal
extern int i3;      // refers to previous, whose linkage is external
extern int i4;      // refers to previous, whose linkage is external
extern int i5;      // refers to previous, whose linkage is internal
```

5 **EXAMPLE 2** If at the end of the translation unit containing

```
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

6.10 Preprocessing directives

Syntax

1 *preprocessing-file:*
group$_{\text{opt}}$

group:
group-part
group group-part

group-part:
if-section
control-line
text-line
*non-directive*

if-section:
if-group elif-groups$_{\text{opt}}$ *else-group*$_{\text{opt}}$ *endif-line*

if-group:
 # if *constant-expression new-line group*$_{\text{opt}}$
 # ifdef *identifier new-line group*$_{\text{opt}}$
 # ifndef *identifier new-line group*$_{\text{opt}}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
 # elif *constant-expression new-line group*$_{\text{opt}}$
 # elifdef *identifier new-line group*$_{\text{opt}}$
 # elifndef *identifier new-line group*$_{\text{opt}}$

else-group:
 # else *new-line group*$_{\text{opt}}$

endif-line:
 # endif *new-line*

control-line:
 # include *pp-tokens new-line*
 # embed *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list*$_{\text{opt}}$ **)** *replacement-list new-line*
 # define *identifier lparen* **...)** *replacement-list new-line*
 # define *identifier lparen identifier-list* **, ...)** *replacement-list new-line*
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens*$_{\text{opt}}$ *new-line*
 # warning *pp-tokens*$_{\text{opt}}$ *new-line*
 # pragma *pp-tokens*$_{\text{opt}}$ *new-line*
*new-line*

text-line:
pp-tokens$_{\text{opt}}$ *new-line*

non-directive:
pp-tokens new-line

lparen:
a **(** character not immediately preceded by white space

replacement-list:
pp-tokens$_{\text{opt}}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

identifier-list:
identifier
identifier-list **,** *identifier*

pp-parameter:
pp-parameter-name pp-parameter-clause$_{\text{opt}}$

pp-parameter-name:
pp-standard-parameter
pp-prefixed-parameter

pp-standard-parameter:
identifier

pp-prefixed-parameter:
identifier **::** *identifier*

pp-parameter-clause:
(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**

pp-balanced-token-sequence:
pp-balanced-token
pp-balanced-token-sequence pp-balanced-token

pp-balanced-token:
(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**
[*pp-balanced-token-sequence*$_{\text{opt}}$ **]**
{ *pp-balanced-token-sequence*$_{\text{opt}}$ **}**
any pp-token other than a parenthesis, a bracket, or a brace

embed-parameter-sequence:
pp-parameter
embed-parameter-sequence pp-parameter

Description

2 A *preprocessing directive* consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a **#** preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence. [207]

[207]Thus, preprocessing directives are commonly called "lines". These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the **#** character string literal creation operator

A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

3 A text line shall not begin with a **#** preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

4 Some preprocessing directives take additional information using preprocessor parameters. A *preprocessing parameter* (pp-parameter) shall be either a *preprocessor prefixed parameter* (identified by a pp-prefixed-parameter, for implementation-defined preprocessor parameters) or a *preprocessor standard parameter* (identified with a pp-standard-parameter, for pp-parameters specified by this document).

5 In all aspects, a preprocessor standard parameter specified by this document as an identifier `pp_param` and an identifier of the form `__pp_param__` shall behave the same when used as a preprocessor parameter, except for the spelling.

6 **EXAMPLE 1** Thus, the preprocessor parameters on the two binary resource inclusion directives (6.10.3.1):

```
#embed "boop.h" limit(5)
#embed "boop.h" __limit__(5)
```

behave the same, and can be freely interchanged. Implementations are encouraged to behave similarly for preprocessor parameters (including preprocessor prefixed parameters) they provide.

7 When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

8 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing **#** preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

9 A preprocessor parameter shall be either a preprocessor standard parameter, or an implementation-defined preprocessor prefixed parameter[208].

Semantics

10 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

11 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

12 **EXAMPLE** In:

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a **#** at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced.

13 The execution of a non-directive preprocessing directive results in undefined behavior.

6.10.1 Conditional inclusion

Syntax

1 *defined-macro-expression*:
  `defined` *identifier*

---

in 6.10.4.2, for example).

[208] An unrecognized preprocessor prefixed parameter is a constraint violation, except within has_embed expressions (6.10.1).

defined (*identifier* **)**

h-preprocessing-token:
any *preprocessing-token* other than **>**

h-pp-tokens:
h-preprocessing-token
h-pp-tokens h-preprocessing-token

header-name-tokens:
string-literal
< *h-pp-tokens* **>**

has-include-expression:
__has_include (*header-name* **)**
__has_include (*header-name-tokens* **)**

has-embed-expression:
__has_embed (*header-name embed-parameter-sequence*$_{\mathrm{opt}}$ **)**
__has_embed (*header-name-tokens pp-balanced-token-sequence*$_{\mathrm{opt}}$ **)**

has-c-attribute-express:
__has_c_attribute (*pp-tokens* **)**

2 The **#if** and **#elif** directives are collectively known as the *conditional expression inclusion preprocessing directives*. The conditional expression inclusion preprocessing directives, **#ifdef**, **#ifndef**, **#elifdef**, and **#elifndef** directives are collectively known as the *conditional inclusion preprocessing directives*.

Constraints

3 The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below[209] and it may contain zero or more defined macro expressions, has_include expressions, has_embed expressions, and/or has_c_attribute expressions as unary operator expressions.

4 A defined macro expression evaluates to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

5 The second form of the has_include expression and has_embed expression is considered only if the first form does not match, in which case the preprocessing tokens are processed just as in normal text.

6 The header or source file identified by the parenthesized preprocessing token sequence in each contained has_include expression is searched for as if that preprocessing token were the pp-tokens in a **#include** directive, except that no further macro expansion is performed. Such a directive shall satisfy the syntactic requirements of a **#include** directive. The has_include expression evaluates to 1 if the search for the source file succeeds, and to 0 if the search fails.

7 The resource (6.10.3.1) identified by the header-name preprocessing token sequence in each contained has_embed expression is searched for as if those preprocessing token were the pp-tokens in a **#embed** directive, except that no further macro expansion is performed. Such a directive shall satisfy the syntactic requirements of a **#embed** directive. The has_embed expression evaluates to:

— `0` if the search fails or if any of the embed parameters in the embed parameter sequence specified are not supported by the implementation for the **#embed** directive; or,

— `1` if the search for the resource succeeds and all embed parameters in the embed parameter sequence specified are supported by the implementation for the **#embed** directive and the resource is not empty; or,

[209]Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

— 2 if the search for the resource succeeds and all embed parameters in the embed parameter sequence specified are supported by the implementation for the **`#embed`** directive and the resource is empty.

8 **NOTE 1** Unrecognized preprocessor prefixed parameters in has_embed expressions is not a constraint violation and instead causes the expression to be evaluate to 0, as specified above.

9 Each has_c_attribute expression is replaced by a nonzero pp-number matching the form of an integer constant if the implementation supports an attribute with the name specified by interpreting the pp-tokens as an attribute token, and by 0 otherwise. The pp-tokens shall match the form of an attribute token.

10 Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a token (6.4).

Semantics

11 The **`#ifdef`**, **`#ifndef`**, **`#elifdef`**, and **`#elifndef`** directives, and the **`defined`** conditional inclusion operator, shall treat **`__has_include`**, **`__has_embed`** and **`__has_c_attribute`** as if they were the name of defined macros. The identifiers **`__has_include`**, **`__has_embed`**, and **`__has_c_attribute`** shall not appear in any context not mentioned in this subclause.

12 Preprocessing directives of the forms

`# if` *constant-expression new-line group*$_{opt}$
`# elif` *constant-expression new-line group*$_{opt}$

check whether the controlling constant expression evaluates to nonzero.

13 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **`defined`** unary operator), just as in normal text. If the token **`defined`** is generated as a result of this replacement process or use of the **`defined`** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and evaluations of defined macro expressions, has_include expressions, has_embed expressions, and has_c_attribute expressions have been performed, all remaining identifiers other than **`true`** (including those lexically identical to keywords such as **`false`**) are replaced with the pp-number `0`, **`true`** is replaced with pp-number `1`, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **`intmax_t`** and **`uintmax_t`** defined in the header `<stdint.h>`.[210] This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **`#if`** or **`#elif`** directive) is implementation-defined[211].

Also, whether a single-character character constant may have a negative value is implementation-defined.

14 Preprocessing directives of the forms

`# ifdef` *identifier new-line group*$_{opt}$
`# ifndef` *identifier new-line group*$_{opt}$

[210]Thus, on an implementation where **`INT_MAX`** is `0x7FFF` and **`UINT_MAX`** is `0xFFFF`, the constant `0x8000` is signed and positive within a **`#if`** expression even though it would be unsigned in translation phase 7.

[211]Thus, the constant expression in the following **`#if`** directive and **`if`** statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

 # elifdef *identifier new-line group*$_{opt}$
 # elifndef *identifier new-line group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier*, **#if !defined** *identifier*, **#elif defined** *identifier*, and **#elif !defined** *identifier* respectively.

15 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped. [212)]

16 **EXAMPLE** This demonstrates a way to include a header file only if it is available.

```
#if __has_include(<optional.h>)
#     include <optional.h>
#     define have_optional 1
#elif __has_include(<experimental/optional.h>)
#     include <experimental/optional.h>
#     define have_optional 1
#     define have_experimental_optional 1
#endif
#ifndef have_optional
#     define have_optional 0
#endif
```

17 **EXAMPLE**

```
/* Fallback for compilers not yet implementing this feature. */
#ifndef __has_c_attribute
#define __has_c_attribute(x) 0
#endif /* __has_c_attribute */

#if __has_c_attribute(fallthrough)
/* Standard attribute is available, use it. */
#define FALLTHROUGH [[fallthrough]]
#elif __has_c_attribute(vendor::fallthrough)
/* Vendor attribute is available, use it. */
#define FALLTHROUGH [[vendor::fallthrough]]
#else
/* Fallback implementation. */
#define FALLTHROUGH
#endif
```

18 **EXAMPLE**

```
#ifdef __STDC__
#define TITLE "ISO C Compilation"
#elifndef __cplusplus
#define TITLE "Non-ISO C Compilation"
#else
/* C++ */
#define TITLE "C++ Compilation"
#endif
```

[212)] As indicated by the syntax, no preprocessing tokens are allowed to follow a **#else** or **#endif** directive before the terminating new-line character. However, comments can appear anywhere in a source file, including within a preprocessing directive.

19 **EXAMPLE 1** A combination of `__FILE__` (6.10.9.1) and `__has_embed` could be used to check for support of specific implementation extensions for the `#embed` (6.10.3.1) directive's parameters.

```
#if __has_embed(__FILE__ ext::token(0xB055))
#define DESCRIPTION "Supports extended token embed"
#else
#define DESCRIPTION "Does not support extended token embed"
#endif
```

20 **EXAMPLE 2** The snippet below uses `__has_embed` to check for support of a specific implementation-defined embed parameter, and otherwise uses standard behavior to produce the same effect.

```
void parse_into_s(short* ptr, unsigned char* ptr_bytes, unsigned long long size);

int main () {
#if __has_embed ("bits.bin" ds9000::element_type(short))
      /* Implementation extension: create short integers from the */
      /* translation environment resource into */
      /*  a sequence of integer constants */
      short meow[] = {
#embed "bits.bin" ds9000::element_type(short)
      };
#elif __has_embed ("bits.bin")
      /* no support for implementation-specific */
      /* ds9000::element_type(short) parameter */
      const unsigned char meow_bytes[] = {
#embed "bits.bin"
      };
      short meow[sizeof(meow_bytes) / sizeof(short)] = {};
      /* parse meow_bytes into short values by-hand! */
      parse_into_s(meow, meow_bytes, sizeof(meow_bytes));
#else
#error "cannot find bits.bin resource"
#endif
      return (int)(meow[0] + meow[(sizeof(meow) / sizeof(*meow)) - 1]);
}
```

21 **EXAMPLE 3** If the search for the resource is successful, this resource is always considered empty due to the `limit(0)` embed parameter, including in `__has_embed` expressions.

```
int main () {
#if __has_embed(<infinite-resource> limit(0)) == 2
      // if <infinite-resource> exists, this
      // token sequence is always taken.
      return 0;
#else
      // the 'infinite-resource' resource does not exist
      #error "The resource does not exist"
#endif
}
```

Forward references: macro replacement (6.10.4), source file inclusion (6.10.2), mandatory macros (6.10.9.1), largest integer types (7.22.1.5).

6.10.2 Source file inclusion

Constraints

1 A `#include` directive shall identify a header or source file that can be processed by the implementation.

Semantics

2 A preprocessing directive of the form

 # include < *h-char-sequence* **>** *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

 # include " *q-char-sequence* **"** *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

 # include < *h-char-sequence* **>** *new-line*

with the identical contained sequence (including > characters, if any) from the original directive.

4 A preprocessing directive of the form

 # include *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.[213] The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (6.4.2.1) followed by a period (.) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

6 A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

7 **EXAMPLE 1** The most common uses of **#include** preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

8 **EXAMPLE 2** This illustrates macro-replaced **#include** directives:

```
#if VERSION == 1
      #define INCFILE  "vers1.h"
#elif VERSION == 2
      #define INCFILE  "vers2.h"   // and so on
#else
      #define INCFILE  "versN.h"
#endif
#include INCFILE
```

Forward references: macro replacement (6.10.4).

[213]Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

6.10.3 Binary resource inclusion

6.10.3.1 `#embed` preprocessing directive

Description

1 A *resource* is a source of data accessible from the translation environment. An *embed parameter* is a single preprocessor parameter in the embed parameter sequence. It has an *implementation resource width*, which is the implementation-defined size in bits of the located resource. It also has a *resource width*, which is either:

— the number of bits as computed from the optionally-provided `limit` embed parameter (6.10.3.2), if present; or,

— the implementation resource width.

2 An *embed parameter sequence* is a whitespace-delimited list of preprocessor parameters which may modify the result of the replacement for the `#embed` preprocessing directive.

Constraints

3 An `#embed` directive shall identify a resource that can be processed by the implementation as a binary data sequence given the provided embed parameters.

4 Embed parameters not specified in this document shall be implementation-defined. Implementation-defined embed parameters may change the below-defined semantics of the directive; otherwise, `#embed` directives which do not contain implementation-defined embed parameters shall behave as described in this document.

5 A resource is considered empty when its resource width is zero.

6 Let *embed element width* be either:

— an integer constant expression greater than zero determined by an implementation-defined embed parameter; or,

— `CHAR_BIT` (5.2.4.2.1).

The result of $(resource\ width)\ \%\ (embed\ element\ width)$ shall be zero[214].

Semantics

7 The expansion of a `#embed` directive is a token sequence formed from the list of integer constant expressions described below. The group of tokens for each integer constant expression in the list is separated in the token sequence from the group of tokens for the previous integer constant expression in the list by a comma. The sequence neither begins nor ends in a comma. If the list of integer constant expressions is empty, the token sequence is empty. The directive is replaced by its expansion and, with the presence of certain embed parameters, additional or replacement token sequences.

8 A preprocessing directive of the form

`# embed <` *h-char-sequence* **`>`** *embed-parameter-sequence*$_{\text{opt}}$ *new-line*

searches a sequence of implementation-defined places for a resource identified uniquely by the specified sequence between the < and >. The search for the named resource is done in an implementation-defined manner.

9 A preprocessing directive of the form

`# embed "` *q-char-sequence* **`"`** *embed-parameter-sequence*$_{\text{opt}}$ *new-line*

searches a sequence of implementation-defined places for a resource identified uniquely by the specified sequence between the `"` delimiters. The search for the named resource is done in an

214)This constraint helps ensure data is neither filled with padding values nor truncated in a given environment, and helps ensure the data is portable with respect to usages of `memcpy` (7.26.2.1) with character type arrays initialized from the data.

implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

 # embed < *h-char-sequence* **>** *embed-parameter-sequence*$_{\text{opt}}$ *new-line*

with the identical contained q-char-sequence (including > characters, if any) from the original directive.

10 Either form of the **#embed** directive specified previously behave as specified below. The values of the integer constant expressions in the expanded sequence are determined by an implementation-defined mapping of the resource's data. Each integer constant expression's value is in the range from 0 to $(2^{embed\ element\ width}) - 1$, inclusive.[215] If:

— the list of integer constant expressions is used to initialize an array of a type compatible with **unsigned char**, or compatible with **char** if **char** cannot hold negative values; and,

— the embed element width is equal to **CHAR_BIT** (5.2.4.2.1),

then the contents of the initialized elements of the array are as-if the resource's binary data is **fread** (7.23.8.1) into the array at translation time.

11 A preprocessing directive of the form

 # embed *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **embed** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms[216]. The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single resource name preprocessing token is implementation-defined.

12 An embed parameter with a preprocessor parameter token that is one of the following is a *standard embed parameter*:

limit **prefix** **suffix** **if_empty**

The significance of these standard embed parameters is specified below.

Recommended practice

13 The **#embed** directive is meant to translate binary data in a resource to a sequence of integer constant expressions in a way that preserves the value of the resource's bit stream where possible.

14 A mechanism similar to, but distinct from, the implementation-defined search paths used for source file inclusion (6.10.2) is encouraged.

15 Implementations should take into account translation-time bit and byte orders as well as execution time bit and byte orders to more appropriately represent the resource's binary data from the directive. This maximizes the chance that, if the resource referenced at translation time through the **#embed** directive is the same one accessed through execution-time means, the data that is e.g. **fread** or similar into contiguous storage will compare bit-for-bit equal to an array of character type initialized from an **#embed** directive's expanded contents.

16 **EXAMPLE 1** Placing a small image resource.

```
#include <stddef.h>

void have_you_any_wool(const unsigned char*, size_t);
```

[215] For example, an embed element width of 8 will yield a range of values from 0 to 255, inclusive.

[216] Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

```
int main (int, char*[]) {
      static const unsigned char baa_baa[] = {
#embed "black_sheep.ico"
      };

      have_you_any_wool(baa_baa, sizeof(baa_baa));

      return 0;
}
```

17 **EXAMPLE 2** This snippet:

```
int main (int, char*[]) {
      static const unsigned char coefficients[] = {
#embed "only_8_bits.bin" // potential constraint violation
      };

      return 0;
}
```

may violate the constraint that $(resource\ width)\ \%\ (embed\ element\ width)$ must be 0. The 8 bits might not be evenly divisible by the embed element width (e.g., on a system where `CHAR_BIT` is 16). Issuing a diagnostic in this case may aid in portability by calling attention to potentially incompatible expectations between implementations and their resources.

18 **EXAMPLE 3** Initialization of non-arrays.

```
int main () {
      /* Braces may be kept or elided as per normal initialization rules */
      int i = {
#embed "i.dat"
      }; /* i value is [0, 2^(embed element width)) first entry */
      int i2 =
#embed "i.dat"
      ; /* valid if i.dat produces 1 value,
            i2 value is [0, 2^(embed element width)) */
      struct s {
            double a, b, c;
            struct { double e, f, g; };
            double h, i, j;
      };
      struct s x = {
      /* initializes each element in order according to initialization
      rules with comma-separated list of integer constant expressions
      inside of braces */
#embed "s.dat"
      };
      return 0;
}
```

Non-array types can still be initialized since the directive produces a comma-delimited list of integer constant expressions, a single integer constant expression, or nothing.

19 **EXAMPLE 4** Equivalency of bit sequence and bit order between a translation-time read and an execution-time read of the same resource/file.

```
#include <string.h>
#include <stddef.h>
#include <stdio.h>

int main(void) {
```

```
        static const unsigned char embed_data[] = {
#embed <data.dat>
        };

        const size_t f_size = sizeof(embed_data);
        unsigned char f_data[f_size];
        FILE* f_source = fopen("data.dat", "rb");
        if (f_source == NULL);
                return 1;
        char* f_ptr = (char*)&f_data[0];
        if (fread(f_ptr, 1, f_size, f_source) != f_size) {
                fclose(f_source);
                return 1;
        }
        fclose(f_source);

        int is_same = memcmp(&embed_data[0], f_ptr, f_size);
        // if both operations refers to the same resource/file at
        // execution time and translation time, "is_same" should be 0
        return is_same == 0 ? 0 : 1;
}
```

6.10.3.2 `limit` parameter

Constraints

1 The **`limit`** standard embed parameter may appear zero times or one time in the embed parameter sequence. Its preprocessor argument clause shall be present and have the form:

(*constant-expression* **)**

and shall be an integer constant expression. The integer constant expression shall not evaluate to a value less than 0.

2 The token **`defined`** shall not appear within the constant expression.

Semantics

3 The embed parameter with a preprocessor parameter token limit denotes a balanced preprocessing token sequence that will be used to compute the resource width. Independently of any macro replacement done previously (e.g. when matching the form of **`#embed`**), the constant expression is evaluated after the balanced preprocessing token sequence is processed as in normal text, using the rules specified for conditional inclusion (6.10.1), with the exception that any defined macro expressions are not permitted.

4 The resource width is:

— 0, if the integer constant expression evaluates to 0; or,

— the implementation resource width if it is less than the embed element width multiplied by the integer constant expression; or,

— the embed element width multiplied by the integer constant expression, if it is less than or equal to the implementation resource width.

5 **EXAMPLE 1** Checking the first 4 elements of a sound resource.

```
#include <assert.h>

int main (int, char*[]) {
        static const char sound_signature[] = {
#embed <sdk/jump.wav> limit(2+2)
        };
        static_assert((sizeof(sound_signature) / sizeof(*sound_signature)) == 4,
```

```
                "There should only be 4 elements in this array.");

        // verify PCM WAV resource
        assert(sound_signature[0] == 'R');
        assert(sound_signature[1] == 'I');
        assert(sound_signature[2] == 'F');
        assert(sound_signature[3] == 'F');
        assert(sizeof(sound_signature) == 4);

        return 0;
}
```

6 **EXAMPLE 2** Similar to a previous example, except it illustrates macro expansion specifically done for the `limit(...)` parameter.

```
#include <assert.h>

#define TWO_PLUS_TWO 2+2

int main (int, char*[]) {
        const char sound_signature[] = {
        /* the token sequence within the parentheses
        for the "limit" parameter undergoes macro
        expansion, at least once, resulting in
#embed <sdk/jump.wav> limit(2+2)
        */
#embed <sdk/jump.wav> limit(TWO_PLUS_TWO)
        };
        static_assert((sizeof(sound_signature) / sizeof(*sound_signature)) == 4,
                "There should only be 4 elements in this array.");

        // verify PCM WAV resource
        assert(sound_signature[0] == 'R');
        assert(sound_signature[1] == 'I');
        assert(sound_signature[2] == 'F');
        assert(sound_signature[3] == 'F');
        assert(sizeof(sound_signature) == 4);

        return 0;
}
```

7 **EXAMPLE 3** A potential constraint violation from a resource that may not have enough information in an environment that has a `CHAR_BIT` greater than 24.

```
int main (int, char*[]) {
        const unsigned char arr[] = {
#embed "24_bits.bin" limit(1) // may be a constraint violation
        };

        return 0;
}
```

8 **EXAMPLE 4** A potential constraint violation from a resource that may not have enough information in an environment that has a `CHAR_BIT` greater than 24.

```
int main (int, char*[]) {
        const unsigned char arr[] = {
#embed "24_bits.bin" limit(1) // may be a constraint violation
        };

        return 0;
```

```
}
```

9 **EXAMPLE 5** Resources interfacing with certain implementations may have an infinite stream of data, such as the `</owo/uwurandom>` resource used in the snippet below:

```
int main (int, char*[]) {
      const unsigned char arr[] = {
#embed </owo/uwurandom>  limit(513)
      };

      return 0;
}
```

The `limit` parameter may help process only a portion of that information and prevent exhaustion of an implementation's internal resources when processing such data.

6.10.3.3 `suffix` parameter

Constraints

The suffix standard embed parameter may appear zero times or one time in the embed parameter sequence. Its preprocessor argument clause shall be present and have the form:

(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**

Semantics

1 The embed parameter with a preprocessing parameter token `suffix` denotes a balanced preprocessing token sequence within its preprocessor argument clause that will be placed immediately after the result of the associated `#embed` directive's expansion.

2 If the resource is empty, then `suffix` has no effect and is ignored.

3 **EXAMPLE 1** Extra elements added to array initializer.

```
#include <string.h>

#ifndef SHADER_TARGET
#define SHADER_TARGET "edith-impl.glsl"
#endif

extern char* null_term_shader_data;

void fill_in_data () {
      const char internal_data[] = {
#embed SHADER_TARGET \
            suffix(,)
            0
      };

      strcpy(null_term_shader_data, internal_data);
}
```

6.10.3.4 `prefix` parameter

Constraints

1 The `prefix` standard embed parameter may appear zero times or one time in the embed parameter sequence. Its preprocessor parameter clause shall be present and have the form:

(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**

Semantics

2 The embed parameter with a preprocessor parameter token `prefix` denotes a balanced preprocessing token sequence within its preprocessor argument clause that will be placed immediately before the

result of the associated **#embed** directive's expansion, if any.

3 If the resource is empty, then **prefix** has no effect and is ignored.

4 **EXAMPLE 1** A null-terminated character array with prefixed and suffixed additional tokens when the resource is not empty, providing null termination and a byte order mark.

```
#include <string.h>
#include <assert.h>

#ifndef SHADER_TARGET
#define SHADER_TARGET "ches.glsl"
#endif

extern char* merp;

void init_data () {
      const char whl[] = {
#embed SHADER_TARGET                                 \
           prefix(0xEF, 0xBB, 0xBF, ) /* UTF-8 BOM */ \
           suffix(,)
           0
      };
      // always null terminated,
      // contains BOM if not-empty
      int is_good = (sizeof(whl) == 1 && whl[0] == '\0')
      || (whl[0] == '\xEF' && whl[1] == '\xBB'
      && whl[2] == '\xBF' && whl[sizeof(whl) - 1] == '\0');
      assert(is_good);
      strcpy(merp, whl);
}
```

6.10.3.5 if_empty parameter

Constraints

The **if_empty** standard embed parameter may appear zero times or one time in the embed parameter sequence. Its preprocessor argument clause shall be present and have the form:

(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**

Semantics

1 The embed parameter with a preprocessing parameter token **if_empty** denotes a balanced preprocessing token sequence within its preprocessor argument clause that will replace the **#embed** directive entirely.

If the resource is not empty, then **if_empty** has no effect and is ignored.

2 **EXAMPLE 1** If the search for the resource is successful, this resource is always considered empty due to the `limit(0)` embed parameter. This program always returns 0, even if the resource is searched for and found successfully by the implementation and has an implementation resource width greater than 0.

```
int main () {
      return
#embed <some_resource> limit(0) prefix(1) if_empty(0)
      ;
      // becomes:
      // return 0;
}
```

3 **EXAMPLE 2** An example similar to using the suffix **embed** parameter, but changed slightly.

```
#include <string.h>
```

```
#ifndef SHADER_TARGET
#define SHADER_TARGET "edith-impl.glsl"
#endif

extern char* null_term_shader_data;

void fill_in_data () {
      const char internal_data[] = {
#embed SHADER_TARGET   \
            suffix(, 0) \
            if_empty(0)
      };

      strcpy(null_term_shader_data, internal_data);
}
```

4 **EXAMPLE 3** This resource is considered empty due to the **limit**(0) embed parameter, meaning an **if_empty** expression replaces the directive as specified above. A constraint is still violated if the search for the resource is unsuccessful.

```
int main () {
      return
            #embed <infinite-resource> limit(0) if_empty(45540)
      ;
}
```

becomes:

```
int main () {
      return 45540;
}
```

6.10.4 Macro replacement

Constraints

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

3 There shall be white space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be at least as many arguments in the invocation as there are parameters in the macro definition (excluding the `...`). There shall exist a `)` preprocessing token that terminates the invocation.

5 The identifiers **__VA_ARGS__** and **__VA_OPT__** shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics

7 The identifier immediately following the **`define`** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

8 If a **`#`** preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

`# define` *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name[217] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

`# define` *identifier lparen identifier-list*$_{opt}$ **`)`** *replacement-list new-line*
`# define` *identifier lparen* **`... )`** *replacement-list new-line*
`# define` *identifier lparen identifier-list* **`, ... )`** *replacement-list new-line*

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **`#define`** preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,[218] the behavior is undefined.

12 If there is a `...` in the identifier-list in the macro definition, then the trailing arguments (if any), including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the `...`), except that if there are as many arguments as named parameters, the macro invocation behaves as if a comma token has been appended to the argument list such that variable arguments are formed that contain no pp-tokens.

6.10.4.1 Argument substitution

Syntax

1 *va-opt-replacement*:

`__VA_OPT__ (` *pp-tokens*$_{opt}$ **`)`**

Description

2 Argument substitution is a process during macro expansion in which identifiers corresponding to the parameters of the macro definition and the special constructs **`__VA_ARGS__`** and **`__VA_OPT__`** are replaced with token sequences from the arguments of the macro invocation and possibly of the

[217] Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

[218] Despite the name, a non-directive is a preprocessing directive.

argument of the feature `__VA_OPT__`. The latter process allows to control a substitute token sequence that is only expanded if the argument list that corresponds to a trailing `...` of the parameter list is present and has a non-empty substitution.

Constraints

3 The identifier `__VA_OPT__` shall always occur as part of the preprocessing token sequence va-opt-replacement; its closing `)` is determined by skipping intervening pairs of matching left and right parentheses in its pp-tokens. The pp-tokens of a va-opt-replacement shall not contain `__VA_OPT__`. The pp-tokens shall form a valid replacement list for the current function-like macro.

Semantics

4 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A va-opt-replacement is treated as if it were a parameter. For each parameter in the replacement list that is neither preceded by a **#** or **##** preprocessing token nor followed by a **##** preprocessing token, the preprocessing tokens naming the parameter are replaced by a token sequence determined as follows:

— If the parameter is of the form va-opt-replacement, the replacement preprocessing tokens are the preprocessing token sequence for the corresponding argument, as specified below.

— Otherwise, the replacement preprocessing tokens are the preprocessing tokens of the corresponding argument after all macros contained therein have been expanded. The argument's preprocessing tokens are completely macro replaced before being substituted as if they formed the rest of the preprocessing file with no other preprocessing tokens being available.

5 **EXAMPLE 1**

```
#define LPAREN() (
#define G(Q) 42
#define F(R, X, ...)  __VA_OPT__(G R X) )
int x = F(LPAREN(), 0, <:-);     // replaced by int x = 42;
```

6 An identifier `__VA_ARGS__` that occurs in the replacement list is treated as if it were a parameter, and the variable arguments form the preprocessing tokens used to replace it.

7 The preprocessing token sequence for the corresponding argument of a va-opt-replacement is defined as follows. If a (hypothetical) substitution of `__VA_ARGS__` as neither an operand of **#** nor **##** consists of no preprocessing tokens, the argument consists of a single placemarker preprocessing token (6.10.4.3, 6.10.4.4). Otherwise, the argument consists of the results of the expansion of the contained pp-tokens as the replacement list of the current function-like macro before removal of placemarker tokens, rescanning, and further replacement.

8 **NOTE 1** The placemarker tokens are removed before stringization (6.10.4.2), and can be removed by rescanning and further replacement (6.10.4.4).

9 **EXAMPLE 2**

```
#define F(...)           f(0 __VA_OPT__(,) __VA_ARGS__)
#define G(X, ...)        f(0, X __VA_OPT__(,) __VA_ARGS__)
#define SDEF(sname, ...) S sname __VA_OPT__(= { __VA_ARGS__ })
#define EMP

F(a, b, c)         // replaced by f(0, a, b, c)
F()                // replaced by f(0)
F(EMP)             // replaced by f(0)

G(a, b, c)         // replaced by f(0, a, b, c)
G(a, )             // replaced by f(0, a)
G(a)               // replaced by f(0, a)

SDEF(foo);         // replaced by S foo;
```

```
SDEF(bar, 1, 2);  // replaced by S bar = { 1, 2 };

#define H1(X, ...)    X __VA_OPT__(##) __VA_ARGS__
                  // error: ## on line above
                  // may not appear at the beginning of a replacement
                  // list (6.10.4.3)

#define H2(X, Y, ...) __VA_OPT__(X ## Y,) __VA_ARGS__

H2(a, b, c, d)    // replaced by ab, c, d

#define H3(X, ...)    #__VA_OPT__(X##X X##X)
H3(, 0)           // replaced by ""

#define H4(X, ...)    __VA_OPT__(a X ## X) ## b
H4(, 1)           // replaced by a b

#define H5A(...)      __VA_OPT__()/**/__VA_OPT__()
#define H5B(X)        a ## X ## b
#define H5C(X)        H5B(X)
H5C(H5A())        // replaced by ab
```

6.10.4.2 The # operator

Constraints

1 Each **#** preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

2 If, in the replacement list, a parameter is immediately preceded by a **#** preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument (excluding placemarker tokens). Let the *stringizing argument* be the preprocessing token sequence for the corresponding argument with placemarker tokens removed. Each occurrence of white space between the stringizing argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the stringizing argument is deleted. Otherwise, the original spelling of each preprocessing token in the stringizing argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a `\` character is inserted before each `"` and `\` character of a character constant or string literal (including the delimiting `"` characters), except that it is implementation-defined whether a `\` character is inserted before the `\` character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty stringizing argument is `""`. The order of evaluation of **#** and **##** operators is unspecified.

6.10.4.3 The ## operator

Constraints

1 A **##** preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a **##** preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[219)]

3 For both object-like and function-like macro invocations, before the replacement list is reexamined

[219)] Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

for more macro names to replace, each instance of a **##** preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of **##** operators is unspecified.

4 **EXAMPLE** In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the **##** operator.

6.10.4.4 Rescanning and further replacement

1 After all parameters in the replacement list have been substituted and **#** and **##** processing has taken place, all placemarker preprocessing tokens are removed. The resulting preprocessing token sequence is then rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.10 below.

4 **EXAMPLE** There are cases where it is not clear whether a replacement is nested or not. For example, given the following macro definitions:

```
#define f(a) a*g
#define g(a) f(a)
```

the invocation

```
f(2)(9)
```

could expand to either

```
2*f(9)
```

or

```
2*9*g
```

Strictly conforming programs are not permitted to depend on such unspecified behavior.

6.10.4.5 Scope of macro definitions

1 A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

2 A preprocessing directive of the form

 # undef *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 **EXAMPLE 1** The simplest use of this facility is to define a "manifest constant", as in

```
#define TABSIZE 100

int table[TABSIZE];
```

4 **EXAMPLE 2** The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a)  >  (b) ? (a): (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5 **EXAMPLE 3** To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)   f(x * (a))
#undef  x
#define x      2
#define g      f
#define z      z[0]
#define h      g(\~{ }
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
      (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (\~{ } 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5,  };
char c[2][6] = { "hello", "" };
```

6 **EXAMPLE 4** To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                        x ## s, x ## t)
#define INCFILE(n)  vers ## n
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
      == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
  "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
  s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the **#** and **##** tokens in the macro definition is optional.

7 **EXAMPLE 5** To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
            10, 11, 12,  };
```

8 **EXAMPLE 6** To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)   (a)
#define FUNC_LIKE(a)(   /* note the white space */ \
                        a /* other stuff on this line
                           */)
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)     // different token sequence
#define OBJ_LIKE    (1 - 1) // different white space
#define FUNC_LIKE(b) (a)    // different parameter usage
#define FUNC_LIKE(b) (b)    // different parameter spelling
```

9 **EXAMPLE 7** Finally, to show the variable argument list macro facilities:

```
#define debug(...)      fprintf(stderr, __VA_ARGS__)
#define showlist(...)   puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
            printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr,  "Flag");
fprintf(stderr,  "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"):
            printf("x is %d but y is %d", x, y));
```

6.10.5 Line control

Constraints

1 The string literal of a **#line** directive, if present, shall be a character string literal.

Semantics

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token.

3 If a preprocessing token (in particular **__LINE__**) spans two or more physical lines, it is unspecified which of those line numbers is associated with that token. If a preprocessing directive spans two or more physical lines, it is unspecified which of those line numbers is associated with the preprocessing directive. If a macro invocation spans multiple physical or logical lines, it is unspecified which of those line numbers is associated with that invocation. The line number of a preprocessing token is independent of the context (in particular, as a macro argument or in a preprocessing directive). The line number of a **__LINE__** in a macro body is the line number of the macro invocation.

4 A preprocessing directive of the form

 # line *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer, ignoring any optional digit separators (6.4.4.1) in the digit sequence). The digit sequence shall not specify zero, nor a number greater than 2147483647.

5 A preprocessing directive of the form

 # line *digit-sequence* **"** *s-char-sequence*$_{\text{opt}}$ **"** new-line

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

6 A preprocessing directive of the form

 # line *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.[220)]

Recommended practice

7 The line number associated with a *pp-token* should be the line number of the first character of the *pp-token*. The line number associated with a preprocessing directive should be the line number of the line with the first **#** token. The line number associated with a macro invocation should be the line number of the first character of the macro name in the invocation.

6.10.6 Diagnostic directives

Semantics

1 A preprocessing directive of either form

 # error *pp-tokens*$_{\text{opt}}$ *new-line*
 # warning *pp-tokens*$_{\text{opt}}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

6.10.7 Pragma directive

Semantics

1 A preprocessing directive of the form

 # pragma *pp-tokens*$_{\text{opt}}$ *new-line*

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)[221)] causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored.

2 If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[222)] whose meanings are described elsewhere:

standard-pragma:

 # pragma STDC FP_CONTRACT *on-off-switch*
 # pragma STDC FENV_ACCESS *on-off-switch*
 # pragma STDC FENV_DEC_ROUND *dec-direction*
 # pragma STDC FENV_ROUND *direction*
 # pragma STDC CX_LIMITED_RANGE *on-off-switch*

---

220)Because a new-line is explicitly included as part of the **#line** directive, the number of new-line characters read while processing to the first *pp-token* can be different depending on whether the implementation uses a one-pass preprocessor. Therefore, there are two possible values for the line number following a directive of the form **#line __LINE__** *new-line*.

221)An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

222)See "future language directions" (6.11.6).

on-off-switch: one of
`ON` `OFF` `DEFAULT`

direction: one of
`FE_DOWNWARD` `FE_TONEAREST` `FE_TONEARESTFROMZERO`
`FE_TOWARDZERO` `FE_UPWARD` `FE_DYNAMIC`

dec-direction: one of
`FE_DEC_DOWNWARD` `FE_DEC_TONEAREST` `FE_DEC_TONEARESTFROMZERO`
`FE_DEC_TOWARDZERO` `FE_DEC_UPWARD` `FE_DEC_DYNAMIC`

Forward references: the `FP_CONTRACT` pragma (7.12.2), the `FENV_ACCESS` pragma (7.6.1), the `FENV_DEC_ROUND` pragma (7.6.3), the `FENV_ROUND` pragma (7.6.2), the `CX_LIMITED_RANGE` pragma (7.3.4).

6.10.8 Null directive

Semantics

1 A preprocessing directive of the form

*new-line*

has no effect.

6.10.9 Predefined macro names

1 The values of the predefined macros listed in the following subclauses[223] (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.

2 None of these macro names, nor the identifiers **`defined`**, `__has_c_attribute`, `__has_include`, or `__has_embed` shall be the subject of a **`#define`** or a **`#undef`** preprocessing directive. Any other predefined macro names: shall begin with a leading underscore followed by an uppercase letter; or, a second underscore; or, shall be any of the identifiers **`alignas`**, **`alignof`**, **`bool`**, **`false`**, **`static_assert`**, **`thread_local`**, or **`true`**.

3 The implementation shall not predefine the macro `__cplusplus`, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.9.1 Mandatory macros

1 The following macro names shall be defined by the implementation:

`__DATE__` The date of translation of the preprocessing translation unit: a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

`__FILE__` The presumed name of the current source file (a character string literal).[224]

`__LINE__` The presumed line number (within the current source file) of the current source line (an integer constant).[224]

`__STDC__` The integer constant `1`, intended to indicate a conforming implementation.

`__STDC_HOSTED__` The integer constant `1` if the implementation is a hosted implementation or the integer constant `0` if it is not.

[223] See "future language directions" (6.11.7).

[224] The presumed source file name and line number can be changed by the **`#line`** directive.

`__STDC_UTF_16__` The integer constant `1`, intended to indicate that values of type **`char16_t`** are UTF–16 encoded.

`__STDC_UTF_32__` The integer constant `1`, intended to indicate that values of type **`char32_t`** are UTF–32 encoded.

`__STDC_VERSION__` The integer constant *`202311`*`L`.[225)]

`__TIME__` The time of translation of the preprocessing translation unit: a character string literal of the form `"hh:mm:ss"` as in the time generated by the **`asctime`** functions. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the **`asctime`** functions (7.29.3.1).

6.10.9.2 Environment macros

1 The following macro names are conditionally defined by the implementation:

`__STDC_ISO_10646__` An integer constant of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

`__STDC_MB_MIGHT_NEQ_WC__` The integer constant `1`, intended to indicate that, in the encoding for **`wchar_t`**, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.

Forward references: common definitions (7.21), Unicode utilities (7.30).

6.10.9.3 Conditional feature macros

1 The following macro names are conditionally defined by the implementation:

`__STDC_ANALYZABLE__` The integer constant `1`, intended to indicate conformance to the specifications in Annex L (Analyzability).

`__STDC_IEC_60559_BFP__` The integer constant *`202311`*`L`, intended to indicate conformance to Annex F (IEC 60559 floating-point arithmetic) for binary floating-point arithmetic.

`__STDC_IEC_559__` The integer constant `1`, intended to indicate conformance to the specifications in Annex F (IEC 60559 floating-point arithmetic) for binary floating-point arithmetic. Use of this macro is an obsolescent feature.

`__STDC_IEC_60559_DFP__` The integer constant *`202311`*`L`, intended to indicate support of decimal floating types and conformance to Annex F (IEC 60559 floating-point arithmetic) for decimal floating-point arithmetic.

`__STDC_IEC_60559_COMPLEX__` The integer constant *`202311`*`L`, intended to indicate conformance to the specifications in Annex G (IEC 60559 compatible complex arithmetic).

`__STDC_IEC_60559_TYPES__` The integer constant *`202311`*`L`, intended to indicate conformance to the specification in Annex H (IEC 60559 interchange and extended types).

`__STDC_IEC_559_COMPLEX__` The integer constant `1`, intended to indicate adherence to the specifications in Annex G (IEC 60559 compatible complex arithmetic). Use of this macro is an obsolescent feature.

[225)]See Annex M for the values in previous revisions. The intention is that this will remain an integer constant of type **`long int`** that is increased with each revision of this document.

`__STDC_LIB_EXT1__` The integer constant `202311L`, intended to indicate support for the extensions defined in Annex K (Bounds-checking interfaces)[226].

`__STDC_NO_ATOMICS__` The integer constant `1`, intended to indicate that the implementation does not support atomic types (including the `_Atomic` type qualifier) and the `<stdatomic.h>` header.

`__STDC_NO_COMPLEX__` The integer constant `1`, intended to indicate that the implementation does not support complex types or the `<complex.h>` header.

`__STDC_NO_THREADS__` The integer constant `1`, intended to indicate that the implementation does not support the `<threads.h>` header.

`__STDC_NO_VLA__` The integer constant `1`, intended to indicate that the implementation does not support variable length arrays with automatic storage duration. Parameters declared with variable length array types are adjusted and then define objects of automatic storage duration with pointer types. Thus, support for such declarations is mandatory.

2 An implementation that defines `__STDC_NO_COMPLEX__` shall not define `__STDC_IEC_60559_COMPLEX__` or `__STDC_IEC_559_COMPLEX__`.

6.10.10 Pragma operator

Semantics

1 A unary operator expression of the form:

`_Pragma (` *string-literal* `)`

is processed as follows: The string literal is *destringized* by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 **EXAMPLE** A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\\listing.dir\"")
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x)  _Pragma(#x)

LISTING (..\listing.dir)
```

[226]The intention is that this will remain an integer constant of type `long int` that is increased with each revision of this document.

6.11 Future language directions

6.11.1 Floating types

1 Future standardization may include additional floating types, including those with greater range, precision, or both than **long double**.

6.11.2 Linkages of identifiers

1 Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature.

6.11.3 External names

1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Storage-class specifiers

1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.6 Pragma directives

1 Pragmas whose first preprocessing token is **STDC** are reserved for future standardization.

6.11.7 Predefined macro names

1 Macro names beginning with **__STDC_** are reserved for future standardization.

2 Uses of the **__STDC_IEC_559__** and **__STDC_IEC_559_COMPLEX__** macros are obsolescent features.

7. Library

7.1 Introduction

7.1.1 Definitions of terms

1 A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The *length of a string* is the number of bytes preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

2 The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[227] It is represented in the text and examples by a period, but may be changed by the **setlocale** function.

3 A *null wide character* is a wide character with code value zero.

4 A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

5 A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state (see 5.2.1.1). A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.[228] In this clause, "*white-space character*" refers to (execution) white-space character as defined by **isspace**. "*White-space wide character*" refers to (execution) white-space wide character as defined by **iswspace**.

Forward references: character handling (7.4), the **setlocale** function (7.11.1.1).

7.1.2 Standard headers

1 Each library function is declared in a *header*,[229] whose contents are made available by the **#include** preprocessing directive. The header declares a set of related functions, plus any types and additional macros needed to facilitate the use of such related functions. In addition to the provisions given in this clause, an implementation that defines **__STDC_LIB_EXT1__** shall conform to the specifications in Annex K and Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of this clause. Declarations of types described here or in Annex K shall not include type qualifiers, unless explicitly stated otherwise.

2 An implementation that does not support decimal floating types (6.10.9.3) need not support interfaces or aspects of interfaces that are specific to these types.

3 The standard headers are[230]

<assert.h>	<fenv.h>	<limits.h>
<complex.h>	<float.h>	<locale.h>
<ctype.h>	<inttypes.h>	<math.h>
<errno.h>	<iso646.h>	<setjmp.h>

[227] The functions that make use of the decimal-point character are the numeric conversion functions (7.24.1, 7.31.4.1) and the formatted input/output functions (7.23.6, 7.31.2).

[228] For state-dependent encodings, the values for **MB_CUR_MAX** and **MB_LEN_MAX** are thus required to be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation's choice.

[229] A header is not necessarily a source file, nor are the < and > delimited sequences in header names necessarily valid source file names.

[230] The headers <complex.h>, <stdatomic.h>, and <threads.h> are conditional features that implementations need not support; see 6.10.9.3.

`<signal.h>`	`<stddef.h>`	`<tgmath.h>`
`<stdalign.h>`	`<stdint.h>`	`<threads.h>`
`<stdarg.h>`	`<stdio.h>`	`<time.h>`
`<stdatomic.h>`	`<stdlib.h>`	`<uchar.h>`
`<stdbool.h>`	`<stdnoreturn.h>`	`<wchar.h>`
`<stdckdint.h>`	`<string.h>`	`<wctype.h>`

4 If a file with the same name as one of the above < and > delimited sequences, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files, the behavior is undefined.

5 Standard headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including `<assert.h>` depends on the definition of **`NDEBUG`** (see 7.2). If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion of the header or when any macro defined in the header is expanded.

6 Some standard headers define or declare identifiers that had not been present in previous versions of this document. To allow implementations and users to adapt to that situation, they also define a version macro for feature test of the form **`__STDC_VERSION_`***`XXXX`***`_H__`** which expands to *`202311`*`L`, where `XXXX` is the all-caps spelling of the corresponding header `<xxxx.h>`.

7 Any definition of an object-like macro described in this clause or Annex K shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

8 Any declaration of a library function shall have external linkage.

9 A summary of the contents of the standard headers is given in Annex B.

Forward references: diagnostics (7.2).

7.1.3 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

— All potentially reserved identifiers (including ones listed in the future library directions) that are provided by an implementation with an external definition are reserved for any use. An implementation shall not provide an external definition of a potentially reserved identifier unless that identifier is reserved for a use where it would have external linkage.[231] All other potentially reserved identifiers that are provided by an implementation (including in the form of a macro) are reserved for any use when the associated header is included. No other potentially reserved identifiers are reserved.[232]

— Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 7.1.4).

— All identifiers with external linkage in any of the following subclauses (including the future library directions) and **`errno`** are always reserved for use as identifiers with external linkage[233].

[231] All library functions have external linkage.

[232] A potentially reserved identifier becomes a reserved identifier when an implementation begins using it or a future standard reserves it, but is otherwise available for use by the programmer.

[233] The list of reserved identifiers with external linkage includes **`math_errhandling`**, **`setjmp`**, **`va_copy`**, and **`va_end`**.

— Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

7.1.4 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow:

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after default argument promotion) not expected by a function with a variable number of arguments, the behavior is undefined.

— If a function argument is described as being an array, the pointer passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are valid.[234)]

— Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[235)] The use of **`#undef`** to remove any macro definition will also ensure that an actual function is referred to.

— Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.[236)]

— Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called. [237)]

— All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in conditional expression inclusion preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

[234)] This includes, for example, passing a valid pointer that points one-past-the-end of an array along with a size of 0, or using any valid pointer with a size of 0.

[235)] This means that an implementation is required to provide an actual function for each library function, even if it also provides a macro for that function.

[236)] Such macros might not contain the sequence points that the corresponding function calls do.

[237)] Because external identifiers and some macro names beginning with an underscore are reserved, implementations can provide special semantics for such names. For example, the identifier `_BUILTIN_abs` could be used to indicate generation of in-line code for the **`abs`** function. Thus, the appropriate header could specify

```
#define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as **`abs`** will be a genuine function can write

```
#undef abs
```

whether the implementation's header provides a macro implementation of **`abs`** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration. [238)]

5 Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments. [239)] Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

6 Unless otherwise specified, library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.[240)]

7 **EXAMPLE** The function **atoi** can be used in any of several ways:

— by use of its associated header (possibly generating a macro expansion)

```
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

— by use of its associated header (assuredly generating a true function reference)

```
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

— by explicit declaration

```
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

238)Thus, a signal handler cannot, in general, call standard library functions.

239)This means, for example, that an implementation is not permitted to use a **static** object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. Similarly, an implementation of **memcpy** is not permitted to copy bytes beyond the specified length of the destination object and then restore the original values because it could cause a data race if the program shared those bytes between threads.

240)This allows implementations to parallelize operations if there are no visible side effects.

7.2 Diagnostics <assert.h>

1 The header <assert.h> defines the **assert** and __STDC_VERSION_ASSERT_H__ macros and refers to another macro,

```
NDEBUG
```

which is *not* defined by <assert.h>. If **NDEBUG** is defined as a macro name at the point in the source file where <assert.h> is included, the **assert** macro is defined simply as

```
#define assert(...) ((void)0)
```

The **assert** macro is redefined according to the current state of **NDEBUG** each time that <assert.h> is included.

2 The **assert** macro shall be implemented as a macro with an ellipsis parameter, not as an actual function. If the macro definition is suppressed to access an actual function, the behavior is undefined.

3 The macro

```
__STDC_VERSION_ASSERT_H__
```

is an integer constant expression with a value equivalent to *202311*L.

7.2.1 Program diagnostics

7.2.1.1 The assert macro

Synopsis

1
```
#include <assert.h>
void assert(scalar expression);
```

Description

2 The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if expression (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros **__FILE__** and **__LINE__** and of the identifier **__func__**) on the standard error stream in an implementation-defined format.[241]

It then calls the **abort** function.

Returns

3 The **assert** macro returns no value.

Forward references: the **abort** function (7.24.4.1).

[241]The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

7.3 Complex arithmetic `<complex.h>`

7.3.1 Introduction

1 The header `<complex.h>` defines macros and declares functions that support complex arithmetic.[242]

2 Implementations that define the macro `__STDC_NO_COMPLEX__` need not provide this header nor support any of its facilities.

3 The macro

```
__STDC_VERSION_COMPLEX_H__
```

is an integer constant expression with a value equivalent to `202311L`.

4 Each synopsis, other than for the `CMPLX` macros, specifies a family of functions consisting of a principal function with one or more `double complex` parameters and a `double complex` or `double` return value; and other functions with the same name but with `f` and `l` suffixes which are corresponding functions with `float` and `long double` parameters and return values.

5 The macro

```
complex
```

expands to `_Complex`; the macro

```
_Complex_I
```

expands to a constant expression of type `float _Complex`, with the value of the imaginary unit.[243]

6 The macros

```
imaginary
```

and

```
_Imaginary_I
```

are defined if and only if the implementation supports imaginary types[244]; and, if defined, they expand to `_Imaginary` and a constant expression of type `float _Imaginary` with the value of the imaginary unit.

7 The macro

```
I
```

expands to either `_Imaginary_I` or `_Complex_I`. If `_Imaginary_I` is not defined, `I` shall expand to `_Complex_I`.

8 Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros `complex`, `imaginary`, and `I`.

Forward references: the `CMPLX` macros (7.3.9.3), IEC 60559-compatible complex arithmetic (Annex G).

7.3.2 Conventions

1 Values are interpreted as radians, not degrees. An implementation may set `errno` but is not required to do so.

---

[242] See "future library directions" (7.33.1).
[243] The imaginary unit is a number i such that $i^2 = -1$.
[244] A specification for imaginary types is inAnnex G.

7.3.3 Branch cuts

1 Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications of Annex G, the sign of zero distinguishes one side of a cut from another so that the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part `+0`, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part `-0`, maps to the negative imaginary axis.

2 Implementations that do not support a signed zero (see Annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so that the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, in the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so that the cut maps to the positive imaginary axis.

7.3.4 The CX_LIMITED_RANGE pragma

Synopsis

1
```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Description

2 The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The `CX_LIMITED_RANGE` pragma can be used to inform the implementation that (where the state is "on") the usual mathematical formulas are acceptable.[245] The pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is "off".

7.3.5 Trigonometric functions

7.3.5.1 The cacos functions

Synopsis

1
```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

Description

2 The `cacos` functions compute the complex arc cosine of `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis.

[245]The purpose of the pragma is to allow the implementation to use the formulas:

$$\begin{aligned} (x+iy)\times(u+iv) &= (xu-yv)+i(yu+xv) \\ (x+iy)\;/\;(u+iv) &= [(xu+yv)+i(yu-xv)]/(u^2+v^2) \\ |x+iy| &= \sqrt{x^2+y^2} \end{aligned}$$

where the programmer can determine they are safe.

Returns

3 The **cacos** functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis.

7.3.5.2 The casin functions

Synopsis

1
```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Description

2 The casin functions compute the complex arc sine of z, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

3 The casin functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ along the real axis.

7.3.5.3 The catan functions

Synopsis

1
```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Description

2 The catan functions compute the complex arc tangent of z, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

3 The catan functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ along the real axis.

7.3.5.4 The ccos functions

Synopsis

1
```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Description

2 The ccos functions compute the complex cosine of z.

Returns

3 The ccos functions return the complex cosine value.

7.3.5.5 The csin functions

Synopsis

1
```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

Description

2 The csin functions compute the complex sine of z.

Returns

3 The `csin` functions return the complex sine value.

7.3.5.6 The `ctan` functions

Synopsis

1
```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

Description

2 The `ctan` functions compute the complex tangent of `z`.

Returns

3 The `ctan` functions return the complex tangent value.

7.3.6 Hyperbolic functions

7.3.6.1 The `cacosh` functions

Synopsis

1
```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Description

2 The `cacosh` functions compute the complex arc hyperbolic cosine of `z`, with a branch cut at values less than 1 along the real axis.

Returns

3 The `cacosh` functions return the complex arc hyperbolic cosine value, in the range of a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.6.2 The `casinh` functions

Synopsis

1
```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Description

2 The `casinh` functions compute the complex arc hyperbolic sine of `z`, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

Returns

3 The `casinh` functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-\frac{i\pi}{2}, +\frac{i\pi}{2}]$ along the imaginary axis.

7.3.6.3 The `catanh` functions

Synopsis

1
```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Description

2 The **catanh** functions compute the complex arc hyperbolic tangent of z, with branch cuts outside the interval $[-1, +1]$ along the real axis.

Returns

3 The **catanh** functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-\frac{i\pi}{2}, +\frac{i\pi}{2}]$ along the imaginary axis.

7.3.6.4 The ccosh functions

Synopsis

1
```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Description

2 The **ccosh** functions compute the complex hyperbolic cosine of z.

Returns

3 The **ccosh** functions return the complex hyperbolic cosine value.

7.3.6.5 The csinh functions

Synopsis

1
```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Description

2 The **csinh** functions compute the complex hyperbolic sine of z.

Returns

3 The **csinh** functions return the complex hyperbolic sine value.

7.3.6.6 The ctanh functions

Synopsis

1
```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Description

2 The **ctanh** functions compute the complex hyperbolic tangent of z.

Returns

3 The **ctanh** functions return the complex hyperbolic tangent value.

7.3.7 Exponential and logarithmic functions

7.3.7.1 The cexp functions

Synopsis

1
```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
```

```
        long double complex cexpl(long double complex z);
```

Description

2 The **cexp** functions compute the complex base-e exponential of `z`.

Returns

3 The **cexp** functions return the complex base-e exponential value.

7.3.7.2 The clog functions

Synopsis

1
```
        #include <complex.h>
        double complex clog(double complex z);
        float complex clogf(float complex z);
        long double complex clogl(long double complex z);
```

Description

2 The **clog** functions compute the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

Returns

3 The **clog** functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

7.3.8 Power and absolute-value functions

7.3.8.1 The cabs functions

Synopsis

1
```
        #include <complex.h>
        double cabs(double complex z);
        float cabsf(float complex z);
        long double cabsl(long double complex z);
```

Description

2 The **cabs** functions compute the complex absolute value (also called norm, modulus, or magnitude) of `z`.

Returns

3 The **cabs** functions return the complex absolute value.

7.3.8.2 The cpow functions

Synopsis

1
```
        #include <complex.h>
        double complex cpow(double complex x, double complex y);
        float complex cpowf(float complex x, float complex y);
        long double complex cpowl(long double complex x, long double complex y);
```

Description

2 The **cpow** functions compute the complex power function $\mathtt{x}^{\mathtt{y}}$, with a branch cut for the first parameter along the negative real axis.

Returns

3 The **cpow** functions return the complex power function value.

7.3.8.3 The `csqrt` functions

Synopsis

1
```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Description

2 The `csqrt` functions compute the complex square root of `z`, with a branch cut along the negative real axis.

Returns

3 The `csqrt` functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

7.3.9 Manipulation functions

7.3.9.1 The `carg` functions

Synopsis

1
```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Description

2 The `carg` functions compute the argument (also called phase (which is an angle)) of `z`, with a branch cut along the negative real axis.

Returns

3 The `carg` functions return the value of the argument in the interval $[-\pi, +\pi]$.

7.3.9.2 The `cimag` functions

Synopsis

1
```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

Description

2 The `cimag` functions compute the imaginary part of `z`.[246]

Returns

3 The `cimag` functions return the imaginary part value (as a real).

7.3.9.3 The `CMPLX` macros

Synopsis

1
```
#include <complex.h>
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

[246]For a variable `z` of complex type, `z == creal(z)+cimag(z)*I`.

Description

2 The **CMPLX** macros expand to an expression of the specified complex type, with the real part having the (converted) value of x and the imaginary part having the (converted) value of y. The resulting expression shall be suitable for use as an initializer for an object with static or thread storage duration, provided both arguments are likewise suitable.

Returns

3 The **CMPLX** macros return the complex value x + iy.

4 **NOTE 1** These macros act as if the implementation supported imaginary types and the definitions were:

```
#define CMPLX(x, y)  ((double complex)((double)(x) + \
                              _Imaginary_I * (double)(y)))
#define CMPLXF(x, y) ((float complex)((float)(x) + \
                              _Imaginary_I * (float)(y)))
#define CMPLXL(x, y) ((long double complex)((long double)(x) + \
                              _Imaginary_I * (long double)(y)))
```

7.3.9.4 The conj functions

Synopsis

1
```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Description

2 The **conj** functions compute the complex conjugate of z, by negating the sign of its imaginary part.

Returns

3 The **conj** functions return the complex conjugate value.

7.3.9.5 The cproj functions

Synopsis

1
```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Description

2 The **cproj** functions compute a projection of z onto the Riemann sphere where z projects to z except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, then **cproj**(z) is equivalent to

```
INFINITY + I * copysign(0.0, cimag(z))
```

Returns

3 The **cproj** functions return the value of the projection onto the Riemann sphere.

7.3.9.6 The creal functions

Synopsis

1
```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Description

2 The **creal** functions compute the real part of z.[247)]

Returns

3 The **creal** functions return the real part value.

[247)] For a variable z of complex type, `z == creal(z)+cimag(z)*I`.

7.4 Character handling `<ctype.h>`

1 The header `<ctype.h>` declares several functions useful for classifying and mapping characters.[248] In all cases the argument is an **`int`**, the value of which shall be representable as an **`unsigned char`** or shall equal the value of the macro **`EOF`**. If the argument has any other value, the behavior is undefined.

2 The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the `"C"` locale are noted below.

3 The term *printing character* refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.[249] All letters and digits are printing characters.

Forward references: **`EOF`** (7.23.1), localization (7.11).

7.4.1 Character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument `c` conforms to that in the description of the function.

7.4.1.1 The `isalnum` function

Synopsis

1
```
#include <ctype.h>
int isalnum(int c);
```

Description

2 The **`isalnum`** function tests for any character for which **`isalpha`** or **`isdigit`** is true.

7.4.1.2 The `isalpha` function

Synopsis

1
```
#include <ctype.h>
int isalpha(int c);
```

Description

2 The **`isalpha`** function tests for any character for which **`isupper`** or **`islower`** is true, or any character that is one of a locale-specific set of alphabetic characters for which none of **`iscntrl`**, **`isdigit`**, **`ispunct`**, or **`isspace`** is true.[250] In the `"C"` locale, **`isalpha`** returns true only for the characters for which **`isupper`** or **`islower`** is true.

7.4.1.3 The `isblank` function

Synopsis

1
```
#include <ctype.h>
int isblank(int c);
```

Description

2 The **`isblank`** function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which **`isspace`** is true and that is used to separate words within a line of text. The standard blank characters are the following: space (`' '`), and horizontal tab (`'\t'`). In the `"C"` locale, **`isblank`** returns true only for the standard blank characters.

[248] See “future library directions” (7.33.2).

[249] In an implementation that uses the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde); the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL).

[250] The functions **`islower`** and **`isupper`** test true or false separately for each of these additional characters; all four combinations are possible.

7.4.1.4 The iscntrl function

Synopsis

1
```
#include <ctype.h>
int iscntrl(int c);
```

Description

2 The **iscntrl** function tests for any control character.

7.4.1.5 The isdigit function

Synopsis

1
```
#include <ctype.h>
int isdigit(int c);
```

Description

2 The **isdigit** function tests for any decimal-digit character (as defined in 5.2.1).

7.4.1.6 The isgraph function

Synopsis

1
```
#include <ctype.h>
int isgraph(int c);
```

Description

2 The **isgraph** function tests for any printing character except space (`' '`).

7.4.1.7 The islower function

Synopsis

1
```
#include <ctype.h>
int islower(int c);
```

Description

2 The **islower** function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the `"C"` locale, **islower** returns true only for the lowercase letters (as defined in 5.2.1).

7.4.1.8 The isprint function

Synopsis

1
```
#include <ctype.h>
int isprint(int c);
```

Description

2 The **isprint** function tests for any printing character including space (`' '`).

7.4.1.9 The ispunct function

Synopsis

1
```
#include <ctype.h>
int ispunct(int c);
```

Description

2 The **ispunct** function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither **isspace** nor **isalnum** is true. In the `"C"` locale, **ispunct** returns true for every printing character for which neither **isspace** nor **isalnum** is true.

7.4.1.10 The `isspace` function

Synopsis

1
```
#include <ctype.h>
int isspace(int c);
```

Description

2 The `isspace` function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which `isalnum` is false. The standard white-space characters are the following: space (`' '`), form feed (`'\f'`), new-line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). In the `"C"` locale, `isspace` returns true only for the standard white-space characters.

7.4.1.11 The `isupper` function

Synopsis

1
```
#include <ctype.h>
int isupper(int c);
```

Description

2 The `isupper` function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the `"C"` locale, `isupper` returns true only for the uppercase letters (as defined in 5.2.1).

7.4.1.12 The `isxdigit` function

Synopsis

1
```
#include <ctype.h>
int isxdigit(int c);
```

Description

2 The `isxdigit` function tests for any hexadecimal-digit character (as defined in 6.4.4.1).

7.4.2 Character case mapping functions

7.4.2.1 The `tolower` function

Synopsis

1
```
#include <ctype.h>
int tolower(int c);
```

Description

2 The `tolower` function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a character for which `isupper` is true and there are one or more corresponding characters, as specified by the current locale, for which `islower` is true, the `tolower` function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.4.2.2 The `toupper` function

Synopsis

1
```
#include <ctype.h>
int toupper(int c);
```

Description

2 The `toupper` function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a character for which **`islower`** is true and there are one or more corresponding characters, as specified by the current locale, for which **`isupper`** is true, the **`toupper`** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.5 Errors <errno.h>

1 The header <errno.h> defines several macros, all relating to the reporting of error conditions.

2 The macros are

```
EDOM
EILSEQ
ERANGE
```

which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in conditional expression inclusion preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue[251] that has type **int** and thread storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed to access an actual object, or a program defines an identifier with the name **errno**, the behavior is undefined.

3 The value of **errno** in the initial thread is zero at program startup (the initial representation of the object designated by **errno** in other threads is indeterminate), but is never set to zero by any library function[252]. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in this document.

4 Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter,[253] may also be specified by the implementation.

[251]The macro **errno** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, *errno()).

[252]Thus, a program that uses **errno** for error checking would set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **errno** on entry and then set it to zero, as long as the original value is restored if **errno**'s value is still zero just before the return.

[253]See "future library directions" (7.33.3).

7.6 Floating-point environment `<fenv.h>`

1 The header `<fenv.h>` defines several macros, and declares types and functions that provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[254)]

A *floating-point status flag* is a system variable whose value is set (but never cleared) when a *floating-point exception* is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information.[255)] A *floating-point control mode* is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

2 A floating-point control mode may be *constant* (7.6.2) or *dynamic*. The *dynamic floating-point environment* includes the dynamic floating-point control modes and the floating-point status flags.

3 The dynamic floating-point environment has thread storage duration. The initial state for a thread's dynamic floating-point environment is the current state of the dynamic floating-point environment of the thread that creates it. It is initialized at the at the time of the thread's creation.

4 Certain programming conventions support the intended model of use for the dynamic floating-point environment:[256)]

— a function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented;

— a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;

— a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

5 The feature test macro `__STDC_VERSION_FENV_H__` expands to the token `202311L`.

6 The type

```
fenv_t
```

represents the entire dynamic floating-point environment.

7 The type

```
femode_t
```

represents the collection of dynamic floating-point control modes supported by the implementation, including the dynamic rounding direction mode.

8 The type

```
fexcept_t
```

represents the floating-point status flags collectively, including any status the implementation associates with the flags.

9 Each of the macros

254)This header is designed to support the floating-point exception status flags and rounding-direction control modes required by IEC 60559, and other similar floating-point state information. It is also designed to facilitate code portability among all systems.

255)A floating-point status flag is not an object and can be set more than once within an expression.

256)With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
```

is defined if and only if the implementation supports the floating-point exception by means of the functions in 7.6.4.[257] Additional implementation-defined floating-point exceptions, with macro definitions beginning with **FE_** and an uppercase letter,[258] may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values, and furthermore, bitwise ANDs of all combinations of the macros result in zero.[259]

10 Decimal floating-point operations and IEC 60559 binary floating-point operations (Annex F) access the same floating-point exception status flags.

11 The macro

```
FE_DFL_MODE
```

represents the default state for the collection of dynamic floating-point control modes supported by the implementation – and has type "pointer to const-qualified **femode_t**". Additional implementation-defined states for the dynamic mode collection, with macro definitions beginning with **FE_** and an uppercase letter, and having type "pointer to const-qualified **femode_t**", may also be specified by the implementation.

12 The macro

```
FE_ALL_EXCEPT
```

is the bitwise OR of all floating-point exception macros defined by the implementation. If no such macros are defined, **FE_ALL_EXCEPT** shall be defined as `0`.

13 Each of the macros

```
FE_DOWNWARD
FE_TONEAREST
FE_TONEARESTFROMZERO
FE_TOWARDZERO
FE_UPWARD
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the **fegetround** and **fesetround** functions. Additional implementation-defined rounding directions, with macro definitions beginning with **FE_** and an uppercase letter,[260] may also be specified by the implementation.[261]

14 If the implementation supports decimal floating types, each of the macros

```
FE_DEC_DOWNWARD
FE_DEC_TONEAREST
FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO
FE_DEC_UPWARD
```

[257]The implementation supports a floating-point exception if there are circumstances where a call to at least one of the functions in 7.6.4, using the macro as the appropriate argument, will succeed. It is not necessary for all the functions to succeed all the time.

[258]See "future library directions" (7.33.4).

[259]The macros are typically distinct powers of two.

[260]See "future library directions" (7.33.4).

[261]Even though the rounding direction macros might expand to constants corresponding to the values of **FLT_ROUNDS**, they are not required to do so.

is defined for use with the **fe_dec_getround** and **fe_dec_setround** functions for getting and setting the dynamic rounding direction mode, and with the **FENV_DEC_ROUND** rounding control pragma (7.6.3) for specifying a constant rounding direction, for decimal floating-point operations. The decimal rounding direction affects all (inexact) operations that produce a result of decimal floating type and all operations that produce an integer or character sequence result and have an operand of decimal floating type, unless stated otherwise. The macros expand to integer constant expressions whose values are distinct nonnegative values.

15 During translation, constant rounding direction modes for decimal floating-point arithmetic are in effect where specified. Elsewhere, during translation the decimal rounding direction mode is **FE_DEC_TONEAREST**.

16 At program startup the dynamic rounding direction mode for decimal floating-point arithmetic is initialized to **FE_DEC_TONEAREST**.

17 The macro

```
FE_DFL_ENV
```

represents the default dynamic floating-point environment — the one installed at program startup — and has type "pointer to const-qualified **fenv_t**". It can be used as an argument to `<fenv.h>` functions that manage the dynamic floating-point environment.

18 Additional implementation-defined environments, with macro definitions beginning with **FE_** and an uppercase letter,[262] and having type "pointer to const-qualified **fenv_t**", may also be specified by the implementation.

7.6.1 The FENV_ACCESS pragma

Synopsis

1
```
#include <fenv.h>
#pragma STDC FENV_ACCESS on-off-switch
```

Description

2 The **FENV_ACCESS** pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.[263] The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FENV_ACCESS** pragma is encountered (including within a nested compound statement), or until the end of the compound statement. At the end of a compound statement, the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags or establishes non-default floating-point mode settings using any means other than the **FENV_ROUND** pragmas, but was translated with the state for the **FENV_ACCESS** pragma "off", the behavior is undefined. The default state ("on" or "off") for the pragma is implementation-defined. (When execution passes from a part of the program translated with **FENV_ACCESS** "off" to a part translated with **FENV_ACCESS** "on", the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

3 **EXAMPLE**

```
#include <fenv.h>
void f(double x)
```

[262]See "future library directions" (7.33.4).

[263]The purpose of the **FENV_ACCESS** pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of **FENV_ACCESS** is "off", the translator can assume that the flags are not tested, and that default modes are in effect, except where specified otherwise by an **FENV_ROUND** pragma.

```
{
        #pragma STDC FENV_ACCESS ON
        void g(double);
        void h(double);
        /* ... */
        g(x + 1);
        h(x + 1);
        /* ... */
}
```

4 If the function `g` might depend on status flags set as a side effect of the first `x + 1`, or if the second `x + 1` might depend on control modes set as a side effect of the call to function `g`, then the program has to contain an appropriately placed invocation of `#pragma STDC FENV_ACCESS ON` as shown.[264)]

7.6.2 The `FENV_ROUND` pragma

Synopsis

1
```
#include <fenv.h>
#pragma STDC FENV_ROUND direction
#pragma STDC FENV_ROUND FE_DYNAMIC
```

Description

2 The `FENV_ROUND` pragma provides a means to specify a constant rounding direction for floating-point operations for standard floating types within a translation unit or compound statement. The pragma shall occur either outside external declarations or before all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FENV_ROUND` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ROUND` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the static rounding mode is restored to its condition just before the compound statement. If this pragma is used in any other context, its behavior is undefined.

3 *direction* shall be one of the names of the supported rounding direction macros for operations for standard floating types (7.6), or `FE_DYNAMIC`. If any other value is specified, the behavior is undefined. If no `FENV_ROUND` pragma is in effect, or the specified constant rounding mode is `FE_DYNAMIC`, rounding is according to the mode specified by the dynamic floating-point environment, which is the dynamic rounding mode that was established either at thread creation or by a call to `fesetround`, `fesetmode`, `fesetenv`, or `feupdateenv`. If the `FE_DYNAMIC` mode is specified and `FENV_ACCESS` is "off", the translator may assume that the default rounding mode is in effect.

4 The `FENV_ROUND` pragma affects operations for standard floating types. Within the scope of an `FENV_ROUND` pragma establishing a mode other than `FE_DYNAMIC`, floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic types to its semantic type, as done by classification macros), and invocations of functions indicated in the table below, for which macro replacement has not been suppressed (7.1.4), shall be evaluated according to the specified constant rounding mode (as though no constant mode was specified and the corresponding dynamic rounding mode had been established by a call to `fesetround`). Invocations of functions for which macro replacement has been suppressed and invocations of functions other than those indicated in the table below shall not be affected by constant rounding modes – they are affected by (and affect) only the dynamic mode. Floating constants (6.4.4.2) of a standard floating type that occur in the scope of a constant rounding mode shall be interpreted according to that mode.

[264)]The side effects impose a temporal ordering that requires two evaluations of `x + 1`. On the other hand, without the `#pragma STDC FENV_ACCESS ON` pragma, and assuming the default state is "off", just one evaluation of `x + 1` would suffice.

Functions affected by constant rounding modes – for standard floating types

Header	Function families
<math.h>	**acos**, **acospi**, **asin**, **asinpi**, **atan**, **atan2**, **atan2pi**, **atanpi**
<math.h>	**cos**, **cospi**, **sin**, **sinpi**, **tan**, **tanpi**
<math.h>	**acosh**, **asinh**, **atanh**
<math.h>	**cosh**, **sinh**, **tanh**
<math.h>	**exp**, **exp10**, **exp10m1**, **exp2**, **exp2m1**, **expm1**
<math.h>	**log**, **log10**, **log10p1**, **log1p**, **log2**, **log2p1**, **logp1**
<math.h>	**scalbn**, **scalbln**, **ldexp**
<math.h>	**cbrt**, **compoundn**, **hypot**, **pow**, **pown**, **powr**, **rootn**, **rsqrt**, **sqrt**
<math.h>	**erf**, **erfc**
<math.h>	**lgamma**, **tgamma**
<math.h>	**rint**, **nearbyint**, **lrint**, **llrint**
<math.h>	**fdim**
<math.h>	**fma**
<math.h>	**fadd**, **dadd**, **fsub**, **dsub**, **fmul**, **dmul**, **fdiv**, **ddiv**, **ffma**, **dfma**, **fsqrt**, **dsqrt**
<stdlib.h>	**atof**, **strfrom**, **strto**
<wchar.h>	**wcsto**
<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

A function family listed in the table above indicates the functions for all standard floating types, where the function family is represented by the name of the functions without a suffix. For example, **acos** indicates the functions **acos**, **acosf**, and **acosl**.

5 **NOTE 1** Constant rounding modes (other than `FE_DYNAMIC`) could be implemented using dynamic rounding modes as illustrated in the following example:

```
{
        #pragma STDC FENV_ROUND direction
        // compiler inserts:
        // #pragma STDC FENV_ACCESS ON
        // int __savedrnd;
        // __savedrnd = __swapround(direction);
        ...  operations affected by constant rounding mode ...
        // compiler inserts:
        // __savedrnd = __swapround(__savedrnd);
        ...  operations not affected by constant rounding mode ...
        // compiler inserts:
        // __savedrnd = __swapround(__savedrnd);
        ...  operations affected by constant rounding mode ...
        // compiler inserts:
        // __swapround(__savedrnd);
}
```

where `__swapround` is defined by:

```
static inline int __swapround(const int new) {
        const int old = fegetround();
        fesetround(new);
        return old;
}
```

7.6.3 The FENV_DEC_ROUND pragma

Synopsis

1
```
#include <fenv.h>
```

```
#ifdef __STDC_IEC_60559_DFP__
#pragma STDC FENV_DEC_ROUND dec-direction
#endif
```

Description

2 The `FENV_DEC_ROUND` pragma is a decimal floating-point analog of the `FENV_ROUND` pragma. If `FLT_RADIX` is not 10, the `FENV_DEC_ROUND` pragma affects operators, functions, and floating constants only for decimal floating types. The affected functions are listed in the table below. If `FLT_RADIX` is 10, whether the `FENV_ROUND` and `FENV_DEC_ROUND` pragmas alter the rounding direction of both standard and decimal floating-point operations is implementation-defined. *dec-direction* shall be one of the decimal rounding direction macro names (`FE_DEC_DOWNWARD`, `FE_DEC_TONEAREST`, `FE_DEC_TONEARESTFROMZERO`, `FE_DEC_TOWARDZERO`, and `FE_DEC_UPWARD`) defined in 7.6, to specify a constant rounding mode, or `FE_DEC_DYNAMIC`, to specify dynamic rounding. The corresponding dynamic rounding mode can be established by a call to `fe_dec_setround`.

Functions affected by constant rounding modes – for decimal floating types

Header	Function families
`<math.h>`	`acos`, `acospi`, `asin`, `asinpi`, `atan`, `atan2`, `atan2pi`, `atanpi`
`<math.h>`	`cos`, `cospi`, `sin`, `sinpi`, `tan`, `tanpi`
`<math.h>`	`acosh`, `asinh`, `atanh`
`<math.h>`	`cosh`, `sinh`, `tanh`
`<math.h>`	`exp`, `exp10`, `exp10m1`, `exp2`, `exp2m1`, `expm1`
`<math.h>`	`log`, `log10`, `log10p1`, `log1p`, `log2`, `log2p1`, `logp1`
`<math.h>`	`scalbn`, `scalbln`, `ldexp`
`<math.h>`	`cbrt`, `compoundn`, `hypot`, `pow`, `pown`, `powr`, `rootn`, `rsqrt`, `sqrt`
`<math.h>`	`erf`, `erfc`
`<math.h>`	`lgamma`, `tgamma`
`<math.h>`	`rint`, `nearbyint`, `lrint`, `llrint`
`<math.h>`	`quantize`
`<math.h>`	`fdim`
`<math.h>`	`fma`
`<math.h>`	`d32add`, `d64add`, `d32sub`, `d64sub`, `d32mul`, `d64mul`, `d32div`, `d64div`, `d32fma`, `d64fma`, `d32sqrt`, `d64sqrt`
`<stdlib.h>`	`strfrom`, `strto`
`<wchar.h>`	`wcsto`
`<stdio.h>`	`printf` and `scanf` families
`<wchar.h>`	`wprintf` and `wscanf` families

A function family listed in the table above indicates the functions for all decimal floating types, where the function family is represented by the name of the functions without a suffix. For example, `acos` indicates the functions `acosd32`, `acosd64`, and `acosd128`.

7.6.4 Floating-point exceptions

1 The following functions provide access to the floating-point status flags.[265] The `int` input argument for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise OR of one or more floating-point exception macros, for example `FE_OVERFLOW | FE_INEXACT`. For other argument values, the behavior of these functions is undefined.

7.6.4.1 The `feclearexcept` function

[265]The functions `fetestexcept`, `feraiseexcept`, and `feclearexcept` support the basic abstraction of flags that are either set or clear. An implementation can endow floating-point status flags with more information — for example, the address of the code which first raised the floating-point exception; the functions `fegetexceptflag` and `fesetexceptflag` deal with the full content of flags.

Synopsis

1
```
#include <fenv.h>
int feclearexcept(int excepts);
```

Description

2 The `feclearexcept` function attempts to clear the supported floating-point exceptions represented by its argument.

Returns

3 The `feclearexcept` function returns zero if the `excepts` argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.

7.6.4.2 The `fegetexceptflag` function

Synopsis

1
```
#include <fenv.h>
int fegetexceptflag(fexcept_t *flagp, int excepts);
```

Description

2 The `fegetexceptflag` function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by the argument `flagp`.

Returns

3 The `fegetexceptflag` function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.

7.6.4.3 The `feraiseexcept` function

Synopsis

1
```
#include <fenv.h>
int feraiseexcept(int excepts);
```

Description

2 The `feraiseexcept` function attempts to raise the supported floating-point exceptions represented by its argument. [266] The order in which these floating-point exceptions are raised is unspecified, except as stated in F.8.6. Whether the `feraiseexcept` function additionally raises the "inexact" floating-point exception whenever it raises the "overflow" or "underflow" floating-point exception is implementation-defined.

Returns

3 The `feraiseexcept` function returns zero if the `excepts` argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.

Recommended Practice

Implementation extensions associated with raising a floating-point exception (for example, enabled traps or IEC 60559 alternate exception handling) should be honored by this function.

7.6.4.4 The `fesetexcept` function

Synopsis

1
```
#include <fenv.h>
int fesetexcept(int excepts);
```

[266]The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, implementation extensions associated with raising a floating-point exception (for example, enabled traps or IEC 60559 alternate exception handling) should be honored. The specification in F.8.6 is in the same spirit.

Description

2 The `fesetexcept` function attempts to set the supported floating-point exception flags represented by its argument. This function does not clear any floating-point exception flags. This function changes the state of the floating-point exception flags, but does not cause any other side effects that might be associated with raising floating-point exceptions. [267)]

Returns

3 The `fesetexcept` function returns zero if all the specified exceptions were successfully set or if the excepts argument is zero. Otherwise, it returns a nonzero value.

7.6.4.5 The `fesetexceptflag` function

Synopsis

1
```
#include <fenv.h>
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

Description

2 The `fesetexceptflag` function attempts to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. Like `fesetexcept`, this function does not raise floating-point exceptions, but only sets the state of the flags.

Returns

3 The `fesetexceptflag` function returns zero if the `excepts` argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.

7.6.4.6 The `fetestexceptflag` function

Synopsis

1
```
#include <fenv.h>
int fetestexceptflag(const fexcept_t * flagp, int excepts);
```

Description

2 The `fetestexceptflag` function determines which of a specified subset of the floating-point exception flags are set in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. The `excepts` argument specifies the floating-point status flags to be queried.

Returns

3 The `fetestexceptflag` function returns the value of the bitwise OR of the floating-point exception macros included in `excepts` corresponding to the floating-point exceptions set in `*flagp`.

7.6.4.7 The `fetestexcept` function

Synopsis

1
```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

2 The `fetestexcept` function determines which of a specified subset of the floating-point exception flags are currently set. The `excepts` argument specifies the floating-point status flags to be queried.[268)]

[267)]Implementation extensions like traps for floating-point exceptions and IEC 60559 exception handling do not occur.
[268)]This mechanism allows testing several floating-point exceptions with just one function call.

Returns

3 The **fetestexcept** function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in excepts.

4 **EXAMPLE** Call f if "invalid" is set, then g if "overflow" is set:

```
#include <fenv.h>
/* ... */
{
        #pragma STDC FENV_ACCESS ON
        int set_excepts;
        feclearexcept(FE_INVALID | FE_OVERFLOW);
        // maybe raise exceptions
        set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
        if (set_excepts & FE_INVALID) f();
        if (set_excepts & FE_OVERFLOW) g();
        /* ... */
}
```

7.6.5 Rounding and other control modes

1 The **fegetround** and **fesetround** functions provide control of rounding direction modes. The **fegetmode** and **fesetmode** functions manage all the implementation's dynamic floating-point control modes collectively.

7.6.5.1 The fegetmode function

Synopsis

1
```
#include <fenv.h>
int fegetmode(femode_t *modep);
```

Description

2 The **fegetmode** function attempts to store all the dynamic floating-point control modes in the object pointed to by modep.

Returns

3 The **fegetmode** function returns zero if the modes were successfully stored. Otherwise, it returns a nonzero value.

7.6.5.2 The fegetround function

Synopsis

1
```
#include <fenv.h>
int fegetround(void);
```

Description

2 The **fegetround** function gets the current value of the dynamic rounding direction mode.

Returns

3 The **fegetround** function returns the value of the rounding direction macro representing the current dynamic rounding direction or a negative value if there is no such rounding direction macro or the current dynamic rounding direction is not determinable.

7.6.5.3 The fe_dec_getround function

Synopsis

1
```
#include <fenv.h>
#ifdef __STDC_IEC_60559_DFP__
int fe_dec_getround(void);
```

```
#endif
```

Description

2 The **fe_dec_getround** function gets the current value of the dynamic rounding direction mode for decimal floating-point operations.

Returns

3 The **fe_dec_getround** function returns the value of the rounding direction macro representing the current dynamic rounding direction for decimal floating-point operations, or a negative value if there is no such rounding macro or the current rounding direction is not determinable.

7.6.5.4 The fesetmode function

Synopsis

1
```
#include <fenv.h>
int fesetmode(const femode_t *modep);
```

Description

2 The **fesetmode** function attempts to establish the dynamic floating-point modes represented by the object pointed to by modep. The argument modep shall point to an object set by a call to **fegetmode**, or equal **FE_DFL_MODE** or a dynamic floating-point mode state macro defined by the implementation.

Returns

The fesetmode **fesetmode** function returns zero if the modes were successfully established. Otherwise, it returns a nonzero value.

7.6.5.5 The fesetround function

Synopsis

1
```
#include <fenv.h>
int fesetround(int rnd);
```

Description

2 The **fesetround** function establishes the rounding direction represented by its argument rnd. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Returns

3 The **fesetround** function returns zero if and only if the dynamic rounding direction mode was set to the requested rounding direction.

4 **EXAMPLE** Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>

void f(int rnd_dir)
{
      #pragma STDC FENV_ACCESS ON
      int save_round;
      int setround_ok;
      save_round = fegetround();
      setround_ok = fesetround(rnd_dir);
      assert(setround_ok == 0);
      /* ... */
      fesetround(save_round);
      /* ... */
```

```
    }
```

7.6.5.6 The fe_dec_setround function

Synopsis

1
```
#include <fenv.h>
#ifdef __STDC_IEC_60559_DFP__
int fe_dec_setround(int rnd);
#endif
```

Description

2 The **fe_dec_setround** function sets the dynamic rounding direction mode for decimal floating-point operations to be the rounding direction represented by its argument `rnd`. If the argument is not equal to the value of a decimal rounding direction macro, the rounding direction is not changed.

3 If **FLT_RADIX** is not 10, the rounding direction altered by the **fesetround** function is independent of the rounding direction altered by the **fe_dec_setround** function; otherwise if **FLT_RADIX** is 10, whether the **fesetround** and **fe_dec_setround** functions alter the rounding direction of both standard and decimal floating-point operations is implementation-defined.

Returns

4 The **fe_dec_setround** function returns a zero value if and only if the argument is equal to a decimal rounding direction macro (that is, if and only if the dynamic rounding direction mode for decimal floating-point operations was set to the requested rounding direction).

7.6.6 Environment

1 The functions in this section manage the floating-point environment — status flags and control modes — as one entity.

7.6.6.1 The fegetenv function

Synopsis

1
```
#include <fenv.h>
int fegetenv(fenv_t *envp);
```

Description

2 The **fegetenv** function attempts to store the current dynamic floating-point environment in the object pointed to by `envp`.

Returns

3 The **fegetenv** function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.

7.6.6.2 The feholdexcept function

Synopsis

1
```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

2 The **feholdexcept** function saves the current dynamic floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and then installs a *non-stop* (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.[269)]

[269)]IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the **feholdexcept** function can be used in conjunction with the **feupdateenv** function to write routines that hide spurious floating-point exceptions from their callers.

Returns

3 The **feholdexcept** function returns zero if and only if non-stop floating-point exception handling was successfully installed.

7.6.6.3 The fesetenv function

Synopsis

1
```
#include <fenv.h>
int fesetenv(const fenv_t *envp);
```

Description

2 The **fesetenv** function attempts to establish the dynamic floating-point environment represented by the object pointed to by envp. The argument envp shall point to an object set by a call to **fegetenv** or **feholdexcept**, or equal a dynamic floating-point environment macro. Note that **fesetenv** merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.

Returns

3 The **fesetenv** function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.

7.6.6.4 The feupdateenv function

Synopsis

1
```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

Description

2 The **feupdateenv** function attempts to save the currently raised floating-point exceptions in its automatic storage, install the dynamic floating-point environment represented by the object pointed to by envp, and then raise the saved floating-point exceptions. The argument envp shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a dynamic floating-point environment macro.

Returns

3 The **feupdateenv** function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.

4 **EXAMPLE** Hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
{
      #pragma STDC FENV_ACCESS ON
      double result;
      fenv_t save_env;
      if (feholdexcept(&save_env))
            return /* indication of an environmental problem */;
      // compute result
      if (/* test spurious underflow */)
            if (feclearexcept(FE_UNDERFLOW))
                  return /* indication of an environmental problem */;
      if (feupdateenv(&save_env))
            return /* indication of an environmental problem */;
      return result;
}
```

7.7 Characteristics of floating types <float.h>

1 The header <float.h> defines several macros that expand to various limits and parameters of the real floating types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.2 and 5.2.4.2.3. A summary is given in Annex E.

7.8 Format conversion of integer types `<inttypes.h>`

1 The header `<inttypes.h>` includes the header `<stdint.h>` and extends it with additional facilities provided by hosted implementations.

2 It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the type

```
imaxdiv_t
```

which is a structure type that is the type of the value returned by the **`imaxdiv`** function. For each type declared in `<stdint.h>`, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[270)]

Forward references: integer types `<stdint.h>` (7.22), formatted input/output functions (7.23.6), formatted wide character input/output functions (7.31.2).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of **`PRI`** (character string literals for the **`fprintf`** and **`fwprintf`** family) or **`SCN`** (character string literals for the **`fscanf`** and **`fwscanf`** family),[271)] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.22.1. In these names, *N* represents the width of the type as described in 7.22.1. For example, **`PRIdFAST32`** can be used in a format string to print the value of an integer of type **`int_fast32_t`**.

2 The **`fprintf`** macros for signed integers are:

`PRId`*N*	`PRIdLEAST`*N*	`PRIdFAST`*N*	`PRIdMAX`	`PRIdPTR`
`PRIi`*N*	`PRIiLEAST`*N*	`PRIiFAST`*N*	`PRIiMAX`	`PRIiPTR`

3 The **`fprintf`** macros for unsigned integers are:

`PRIo`*N*	`PRIoLEAST`*N*	`PRIoFAST`*N*	`PRIoMAX`	`PRIoPTR`
`PRIu`*N*	`PRIuLEAST`*N*	`PRIuFAST`*N*	`PRIuMAX`	`PRIuPTR`
`PRIx`*N*	`PRIxLEAST`*N*	`PRIxFAST`*N*	`PRIxMAX`	`PRIxPTR`
`PRIX`*N*	`PRIXLEAST`*N*	`PRIXFAST`*N*	`PRIXMAX`	`PRIXPTR`

4 The **`fscanf`** macros for signed integers are:

`SCNd`*N*	`SCNdLEAST`*N*	`SCNdFAST`*N*	`SCNdMAX`	`SCNdPTR`
`SCNi`*N*	`SCNiLEAST`*N*	`SCNiFAST`*N*	`SCNiMAX`	`SCNiPTR`

5 The **`fscanf`** macros for unsigned integers are:

`SCNo`*N*	`SCNoLEAST`*N*	`SCNoFAST`*N*	`SCNoMAX`	`SCNoPTR`
`SCNu`*N*	`SCNuLEAST`*N*	`SCNuFAST`*N*	`SCNuMAX`	`SCNuPTR`
`SCNx`*N*	`SCNxLEAST`*N*	`SCNxFAST`*N*	`SCNxMAX`	`SCNxPTR`

6 For each type that the implementation provides in `<stdint.h>`, the corresponding **`fprintf`** macros shall be defined and the corresponding **`fscanf`** macros shall be defined unless the implementation does not have a suitable **`fscanf`** length modifier for the type.

7 **EXAMPLE**

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
      uintmax_t i = UINTMAX_MAX;    // this type always exists
      wprintf(L"The largest integer value is %020"
```

[270)]See "future library directions" (7.33.6).

[271)]Separate macros are given for use with **`fprintf`** and **`fscanf`** functions because, in the general case, different format specifiers might be required for **`fprintf`** and **`fscanf`**, even when the type is the same.

```
                PRIxMAX "\n", i);
        return 0;
    }
```

7.8.2 Functions for greatest-width integer types

7.8.2.1 The imaxabs function

Synopsis

1
```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

Description

2 The **imaxabs** function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[272]

Returns

3 The **imaxabs** function returns the absolute value.

7.8.2.2 The imaxdiv function

Synopsis

1
```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

2 The **imaxdiv** function computes `numer / denom` and `numer % denom` in a single operation.

Returns

3 The **imaxdiv** function returns a structure of type **imaxdiv_t** comprising both the quotient and the remainder. The structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type **intmax_t**. If either part of the result cannot be represented, the behavior is undefined.

7.8.2.3 The strtoimax and strtoumax functions

Synopsis

1
```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
```

Description

2 The **strtoimax** and **strtoumax** functions are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions, except that the initial portion of the string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Returns

3 The **strtoimax** and **strtoumax** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.24.1.7).

7.8.2.4 The wcstoimax and wcstoumax functions

[272] The absolute value of the most negative number may not be representable.

Synopsis

1
```
#include <stddef.h>            // for wchar_t
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
```

Description

2 The `wcstoimax` and `wcstoumax` functions are equivalent to the `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions except that the initial portion of the wide string is converted to `intmax_t` and `uintmax_t` representation, respectively.

Returns

3 The `wcstoimax` function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `INTMAX_MAX`, `INTMAX_MIN`, or `UINTMAX_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

Forward references: the `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions (7.31.4.1.4).

7.9 Alternative spellings <iso646.h>

1 The header `<iso646.h>` defines the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

```
and       &&
and_eq    &=
bitand    &
bitor     |
compl     ~
not       !
not_eq    !=
or        ||
or_eq     |=
xor       ^
xor_eq    ^=
```

7.10 Characteristics of integer types <limits.h>

1 The header `<limits.h>` defines several macros that expand to various limits and parameters of the standard integer types.

2 The macro

```
__STDC_VERSION_LIMITS_H__
```

is an integer constant expression with a value equivalent to `202311L`.

3 The rest of the macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.1. A summary is given in Annex E.

7.11 Localization <locale.h>

1 The header <locale.h> declares two functions, one type, and defines several macros.

2 The type is

```
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained in 7.11.2.1. In the "C" locale, the members shall have the values specified in the comments.

```
char *decimal_point;        // "."
char *thousands_sep;        // ""
char *grouping;             // ""
char *mon_decimal_point;    // ""
char *mon_thousands_sep;    // ""
char *mon_grouping;         // ""
char *positive_sign;        // ""
char *negative_sign;        // ""
char *currency_symbol;      // ""
char frac_digits;           // CHAR_MAX
char p_cs_precedes;         // CHAR_MAX
char n_cs_precedes;         // CHAR_MAX
char p_sep_by_space;        // CHAR_MAX
char n_sep_by_space;        // CHAR_MAX
char p_sign_posn;           // CHAR_MAX
char n_sign_posn;           // CHAR_MAX
char *int_curr_symbol;      // ""
char int_frac_digits;       // CHAR_MAX
char int_p_cs_precedes;     // CHAR_MAX
char int_n_cs_precedes;     // CHAR_MAX
char int_p_sep_by_space;    // CHAR_MAX
char int_n_sep_by_space;    // CHAR_MAX
char int_p_sign_posn;       // CHAR_MAX
char int_n_sign_posn;       // CHAR_MAX
```

3 The macros defined are NULL (described in 7.21); and

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the setlocale function.[273] Additional macro definitions, beginning with the characters LC_ and an uppercase letter,[274] may also be specified by the implementation.

7.11.1 Locale control

7.11.1.1 The setlocale function

Synopsis

1
```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

[273] ISO/IEC 9945–2 specifies locale and charmap formats that can be used to specify locales for C.

[274] See "future library directions" (7.33.7).

Description

2 The **setlocale** function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for category names the program's entire locale; the other values for category name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions[275] and the multibyte and wide character functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** and **wcsftime** functions.

3 A value of "C" for locale specifies the minimal environment for C translation; a value of "" for locale specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

4 At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

5 A call to the **setlocale** function may introduce a data race with other calls to the **setlocale** function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the **setlocale** function.

Returns

6 If a pointer to a string is given for locale and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified category for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

7 A null pointer for locale causes the **setlocale** function to return a pointer to the string associated with the category for the program's current locale; the program's locale is not changed.[276]

8 The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program. The behavior is undefined if the returned value is used after a subsequent call to the **setlocale** function, or after the thread which called the **setlocale** function to obtain the returned value has exited.

Forward references: formatted input/output functions (7.23.6), multibyte/wide character conversion functions (7.24.7), multibyte/wide string conversion functions (7.24.8), numeric conversion functions (7.24.1), the **strcoll** function (7.26.4.3), the **strftime** function (7.29.3.5), the **strxfrm** function (7.26.4.5).

7.11.2 Numeric formatting convention inquiry

7.11.2.1 The localeconv function

Synopsis

1
```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

2 The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

[275]The only functions in 7.4 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.

[276]The implementation is thus required to arrange to encode in a string the various categories due to a heterogeneous locale when category has the value **LC_ALL**.

3 The members of the structure with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. Apart from `grouping` and `mon_grouping`, the strings shall start and end in the initial shift state. The members with type `char` are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The members include the following:

`char *decimal_point`
: The decimal-point character used to format nonmonetary quantities.

`char *thousands_sep`
: The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

`char *grouping`
: A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

`char *mon_decimal_point`
: The decimal-point used to format monetary quantities.

`char *mon_thousands_sep`
: The separator for groups of digits before the decimal-point in formatted monetary quantities.

`char *mon_grouping`
: A string whose elements indicate the size of each group of digits in formatted monetary quantities.

`char *positive_sign`
: The string used to indicate a nonnegative-valued formatted monetary quantity.

`char *negative_sign`
: The string used to indicate a negative-valued formatted monetary quantity.

`char *currency_symbol`
: The local currency symbol applicable to the current locale.

`char frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

`char p_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

`char n_cs_precedes`
: Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

`char p_sep_by_space`
: Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a nonnegative locally formatted monetary quantity.

`char n_sep_by_space`
: Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative locally formatted monetary quantity.

`char p_sign_posn`
: Set to a value indicating the positioning of the `positive_sign` for a nonnegative locally formatted monetary quantity.

char `n_sign_posn`
: Set to a value indicating the positioning of the `negative_sign` for a negative locally formatted monetary quantity.

char `*int_curr_symbol`
: The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char `int_frac_digits`
: The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char `int_p_cs_precedes`
: Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

char `int_n_cs_precedes`
: Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

char `int_p_sep_by_space`
: Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

char `int_n_sep_by_space`
: Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a negative internationally formatted monetary quantity.

char `int_p_sign_posn`
: Set to a value indicating the positioning of the `positive_sign` for a nonnegative internationally formatted monetary quantity.

char `int_n_sign_posn`
: Set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity.

4 The elements of `grouping` and `mon_grouping` are interpreted according to the following:

`CHAR_MAX` No further grouping is to be performed.

`0` The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

5 The values of `p_sep_by_space`, `n_sep_by_space`, `int_p_sep_by_space`, and `int_n_sep_by_space` are interpreted according to the following:

`0` No space separates the currency symbol and value.

`1` If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

`2` If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is used instead of a space.

6 The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

`0` Parentheses surround the quantity and currency symbol.

`1` The sign string precedes the quantity and currency symbol.

`2` The sign string succeeds the quantity and currency symbol.

`3` The sign string immediately precedes the currency symbol.

`4` The sign string immediately succeeds the currency symbol.

7 The implementation shall behave as if no library function calls the **localeconv** function.

Returns

8 The **localeconv** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **localeconv** function. In addition, calls to the **setlocale** function with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure.

9 **EXAMPLE 1** The following table illustrates rules which might well be used by four countries to format monetary quantities.

	Local format		International format	
Country	Positive	Negative	Positive	Negative
Country1	`1.234,56 mk`	`-1.234,56 mk`	`FIM 1.234,56`	`FIM -1.234,56`
Country2	`L.1.234`	`-L.1.234`	`ITL 1.234`	`-ITL 1.234`
Country3	`ƒ 1.234,56`	`ƒ -1.234,56`	`NLG 1.234,56`	`NLG -1.234,56`
Country4	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`	`CHF 1,234.56C`

10 For these four countries, the respective values for the monetary members of the structure returned by **localeconv** could be:

	`Country1`	`Country2`	`Country3`	`Country4`
mon_decimal_point	`","`	`""`	`","`	`"."`
mon_thousands_sep	`"."`	`"."`	`"."`	`","`
mon_grouping	`"\3"`	`"\3"`	`"\3"`	`"\3"`
positive_sign	`""`	`""`	`""`	`""`
negative_sign	`"-"`	`"-"`	`"-"`	`"C"`
currency_symbol	`"mk"`	`"L."`	`"\u0192"`	`"SFrs."`
frac_digits	`2`	`0`	`2`	`2`
p_cs_precedes	`0`	`1`	`1`	`1`
n_cs_precedes	`0`	`1`	`1`	`1`
p_sep_by_space	`1`	`0`	`1`	`0`
n_sep_by_space	`1`	`0`	`2`	`0`
p_sign_posn	`1`	`1`	`1`	`1`
n_sign_posn	`1`	`1`	`4`	`2`
int_curr_symbol	`"FIM "`	`"ITL "`	`"NLG "`	`"CHF "`
int_frac_digits	`2`	`0`	`2`	`2`
int_p_cs_precedes	`1`	`1`	`1`	`1`
int_n_cs_precedes	`1`	`1`	`1`	`1`
int_p_sep_by_space	`1`	`1`	`1`	`1`
int_n_sep_by_space	`2`	`1`	`2`	`1`
int_p_sign_posn	`1`	`1`	`1`	`1`
int_n_sign_posn	`4`	`1`	`4`	`2`

11 **EXAMPLE 2** The following table illustrates how the `cs_precedes`, `sep_by_space`, and `sign_posn` members affect the formatted value.

		p_sep_by_space		
p_cs_precedes	p_sign_posn	0	1	2
0	0	(1.25$)	(1.25 $)	(1.25$)
	1	+1.25$	+1.25 $	+ 1.25$
	2	1.25$+	1.25 $+	1.25$ +
	3	1.25+$	1.25 +$	1.25+ $
	4	1.25$+	1.25 $+	1.25$ +
1	0	($1.25)	($ 1.25)	($1.25)
	1	+$1.25	+$ 1.25	+ $1.25
	2	$1.25+	$ 1.25+	$1.25 +
	3	+$1.25	+$ 1.25	+ $1.25
	4	$+1.25	$+ 1.25	$ +1.25

7.12 Mathematics <math.h>

1 The header `<math.h>` declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more **double** parameters, a **double** return value, or both; and other functions with the same name but with **f** and **l** suffixes, which are corresponding functions with **float** and **long double** parameters, return values, or both.[277] Integer arithmetic functions and conversion functions are discussed later.

2 The feature test macro **__STDC_VERSION_MATH_H__** expands to the token *202311*L.

3 The types

```
float_t
double_t
```

are floating types at least as wide as **float** and **double**, respectively, and such that **double_t** is at least as wide as **float_t**. If **FLT_EVAL_METHOD** equals 0, **float_t** and **double_t** are **float** and **double**, respectively; if **FLT_EVAL_METHOD** equals 1, they are both **double**; if **FLT_EVAL_METHOD** equals 2, they are both **long double**; and for other values of **FLT_EVAL_METHOD**, they are otherwise implementation-defined.[278]

4 The types

```
_Decimal32_t
_Decimal64_t
```

are decimal floating types at least as wide as **_Decimal32** and **_Decimal64**, respectively, and such that **_Decimal64_t** is at least as wide as **_Decimal32_t**. They are present only if the implementation defines **__STDC_IEC_60559_DFP__** and additionally the user code defines **__STDC_WANT_IEC_60559_EXT__** before any inclusion of `<math.h>`. If **DEC_EVAL_METHOD** equals 0, **_Decimal32_t** and **_Decimal64_t** are **_Decimal32** and **_Decimal64**, respectively; if **DEC_EVAL_METHOD** equals 1, they are both **_Decimal64**; if **DEC_EVAL_METHOD** equals 2, they are both **_Decimal128**; and for other values of **DEC_EVAL_METHOD**, they are otherwise implementation-defined.

5 The macro

```
HUGE_VAL
```

expands to a **double** constant expression, not necessarily representable as a **float**, whose value is the maximum value returned by library functions when a floating result of type **double** overflows under the default rounding mode, either maximum finite number in the type or positive or unsigned infinity. The macros

```
HUGE_VALF
HUGE_VALL
```

are respectively **float** and **long double** analogs of **HUGE_VAL**[279].

6 The macros in this paragraph are only present if the implementation defines **__STDC_IEC_60559_DFP__** and additionally the user code defines **__STDC_WANT_IEC_60559_EXT__** before any inclusion of `<math.h>`. The macro

```
HUGE_VAL_D32
```

[277]Particularly on systems with wide expression evaluation, a `<math.h>` function might pass arguments and return values in wider format than the synopsis prototype indicates.

[278]The types **float_t** and **double_t** are intended to be the implementation's most efficient types at least as wide as **float** and **double**, respectively. For **FLT_EVAL_METHOD** equal 0, 1, or 2, the type **float_t** is the narrowest type used by the implementation to evaluate floating expressions.

[279]**HUGE_VAL**, **HUGE_VALF**, and **HUGE_VALL** can be positive infinities in an implementation that supports infinities.

expands to a constant expression of type **_Decimal32** representing positive infinity. The macros

```
HUGE_VAL_D64
HUGE_VAL_D128
```

are respectively **_Decimal64** and **_Decimal128** analogs of **HUGE_VAL_D32**.

7 The macro

```
INFINITY
```

is defined if and only if the implementation supports an infinity for the type **float**. It expands to a constant expression of type **float** representing positive or unsigned infinity.

8 The macro

```
DEC_INFINITY
```

expands to a constant expression of type **_Decimal32** representing positive infinity.

9 The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet NaN.

10 The macro

```
DEC_NAN
```

expands to a constant expression of type **_Decimal32** representing a quiet NaN.

11 Use of the macros **INFINITY**, **DEC_INFINITY**, **NAN**, and **DEC_NAN** in `<math.h>` is an obsolescent feature. Instead, use the same macros in `<float.h>`.

12 The *number classification macros*

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

represent mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with **FP_** and an uppercase letter, may also be specified by the implementation.

13 The *math rounding direction macros*

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

represent the rounding directions of the functions **ceil**, **floor**, **trunc**, **round**, and **roundeven**, respectively, that convert to integral values in floating-point formats. They expand to integer constant expressions with distinct values suitable for use as the second argument to the **fromfp**, **ufromfp**, **fromfpx**, and **ufromfpx** functions.

14 The macro

```
FP_FAST_FMA
```

is optionally defined. If defined, it indicates that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of **double** operands.[280] The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, **float** and **long double** analogs of **FP_FAST_FMA**. If defined, these macros expand to the integer constant 1.

15 The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

are, respectively, **_Decimal32**, **_Decimal64**, and **_Decimal128** analogs of **FP_FAST_FMA**.

16 Each of the macros

FP_FAST_FADD	**FP_FAST_DSUBL**	**FP_FAST_FDIVL**	**FP_FAST_FFMA**
FP_FAST_FADDL	**FP_FAST_FMUL**	**FP_FAST_DDIVL**	**FP_FAST_FFMAL**
FP_FAST_DADDL	**FP_FAST_FMULL**	**FP_FAST_FSQRT**	**FP_FAST_DFMAL**
FP_FAST_FSUB	**FP_FAST_DMULL**	**FP_FAST_FSQRTL**	
FP_FAST_FSUBL	**FP_FAST_FDIV**	**FP_FAST_DSQRTL**	

is optionally defined. If defined, it indicates that the corresponding function generally executes about as fast, or faster, than the corresponding operation or function of the argument type with result type the same as the argument type followed by conversion to the narrower type. For **FP_FAST_FFMA**, **FP_FAST_FFMAL**, and **FP_FAST_DFMAL**, the comparison is to a call to **fma** or **fmal** followed by a conversion, not to separate multiply, add, and conversion. If defined, these macros expand to the integer constant 1.

17 The macros

FP_FAST_D32ADDD64	**FP_FAST_D32MULD64**	**FP_FAST_D32FMAD64**
FP_FAST_D32ADDD128	**FP_FAST_D32MULD128**	**FP_FAST_D32FMAD128**
FP_FAST_D64ADDD128	**FP_FAST_D64MULD128**	**FP_FAST_D64FMAD128**
FP_FAST_D32SUBD64	**FP_FAST_D32DIVD64**	**FP_FAST_D32SQRTD64**
FP_FAST_D32SUBD128	**FP_FAST_D32DIVD128**	**FP_FAST_D32SQRTD128**
FP_FAST_D64SUBD128	**FP_FAST_D64DIVD128**	**FP_FAST_D64SQRTD128**

are analogs of **FP_FAST_FADD**, **FP_FAST_FADDL**, **FP_FAST_DADDL**, etc., for decimal floating types.

18 The macros

```
FP_ILOGB0
FP_ILOGBNAN
```

expand to integer constant expressions whose values are returned by **ilogb**(x) if x is zero or NaN, respectively. The value of **FP_ILOGB0** shall be either **INT_MIN** or -**INT_MAX**. The value of **FP_ILOGBNAN** shall be either **INT_MAX** or **INT_MIN**.

19 The macros

[280] Typically, the **FP_FAST_FMA** macro is defined if and only if the **fma** function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.

```
FP_LLOGB0
FP_LLOGBNAN
```

expand to integer constant expressions whose values are returned by `llogb(x)` if `x` is zero or NaN, respectively. The value of `FP_LLOGB0` shall be `LONG_MIN` if the value of `FP_ILOGB0` is `INT_MIN`, and shall be `-LONG_MAX` if the value of `FP_ILOGB0` is `-INT_MAX`. The value of `FP_LLOGBNAN` shall be `LONG_MAX` if the value of `FP_ILOGBNAN` is `INT_MAX`, and shall be `LONG_MIN` if the value of `FP_ILOGBNAN` is `INT_MIN`.

20 The macros

```
MATH_ERRNO
MATH_ERREXCEPT
```

expand to the integer constants 1 and 2, respectively; the macro

```
math_errhandling
```

expands to an expression that has type `int` and the value `MATH_ERRNO`, `MATH_ERREXCEPT`, the bitwise OR of both, or 0; the value shall not be 0 in a hosted implementation. The value of `math_errhandling` is constant for the duration of the program. It is unspecified whether `math_errhandling` is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name `math_errhandling`, the behavior is undefined. If the expression `math_errhandling & MATH_ERREXCEPT` can be nonzero, the implementation shall define the macros `FE_DIVBYZERO`, `FE_INVALID`, and `FE_OVERFLOW` in `<fenv.h>`.

7.12.1 Treatment of error conditions

1 The behavior of each of the functions in `<math.h>` is specified for all representable values of its input arguments, except where explicitly stated otherwise. Each function shall execute as if it were a single operation without raising `SIGFPE` and without generating any of the floating-point exceptions "invalid", "divide-by-zero", or "overflow" except to reflect the result of the function.

2 For all functions, a *domain error* occurs if and only if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[281] Whether a signaling NaN input causes a domain error is implementation-defined. On a domain error, the function returns an implementation-defined value; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `EDOM`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "invalid" floating-point exception is raised.

3 Similarly, a *pole error* (also known as a singularity or infinitary) occurs if and only if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit (for example, `log(0.0)`). The description of each function lists any required pole errors; an implementation may define additional pole errors, provided that such errors are consistent with the mathematical definition of the function. On a pole error, the function returns an implementation-defined value; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "divide-by-zero" floating-point exception is raised.

4 Likewise, a *range error* occurs if and only if the result overflows or underflows, as defined below. The description of each function lists any required range errors; an implementation may define additional range errors, provided that such errors are consistent with the mathematical definition of the function and are the result of either overflow or underflow.[282]

[281] In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

[282] Range errors that are required or implementation-defined shall or may be reported, as specified in this subclause.

5 A floating result overflows if a finite result value with ordinary accuracy[283] would have magnitude (absolute value) too large for the representation with full precision in the specified type. A result that is exactly an infinity does not overflow. If a floating result overflows and default rounding is in effect, then the function returns the value of the macro **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** according to the return type, with the same sign as the correct value of the function; however, for the types with reduced-precision representations of numbers beyond the overflow threshold, the function may return a representation of the result with less than full precision for the type. If a floating result overflows and the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression **errno** acquires the value **ERANGE**. If a floating result overflows and the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "overflow" floating-point exception is raised.

6 The result underflows if a nonzero result value with ordinary accuracy would have magnitude (absolute value) less than the minimum normalized number in the type; however a zero result that is specified to be an exact zero does not underflow. Also, a result with ordinary accuracy and the magnitude of the minimum normalized number may underflow[284]. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether **errno** acquires the value **ERANGE** is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

7 If a domain, pole, or range error occurs and the integer expression `math_errhandling & MATH_ERRNO` is zero,[285] then **errno** shall either be set to the value corresponding to the error or left unmodified. If no such error occurs, **errno** shall be left unmodified regardless of the setting of **math_errhandling**.

7.12.2 The FP_CONTRACT pragma

Synopsis

1
```
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

Description

2 The **FP_CONTRACT** pragma can be used to allow (if the state is "on") or disallow (if the state is "off") the implementation to contract expressions (6.5). Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FP_CONTRACT** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state ("on" or "off") for the pragma is implementation-defined.

7.12.3 Classification macros

1 Floating-point values can be classified as NaN, infinite, normal, subnormal, or zero, or into other implementation-defined categories. Numbers whose magnitude is at least $b^{e_{min}-1}$ (the minimum magnitude of normalized floating-point numbers in the type) and at most $(1-b^{-p})b^{e_{max}}$ (the maximum magnitude of normalized floating-point numbers in the type), where b, p, e_{min}, and e_{max} are as in 5.2.4.2.2, are classified as normal. Larger magnitude finite numbers represented with full precision in the type may also be classified as normal. Nonzero numbers whose magnitude is less than $b^{e_{min}-1}$ are classified as subnormal.

[283]Ordinary accuracy is determined by the implementation. It refers to the accuracy of the function where results are not compromised by extreme magnitude.

[284]The term underflow here is intended to encompass both "gradual underflow" as in IEC 60559 and also "flush-to-zero" underflow. IEC 60559 underflow can occur in cases where the magnitude of the rounded result (accurate to the full precision of the type) equals the minimum normalized number in the format.

[285]Math errors are being indicated by the floating-point exception flags rather than by **errno**.

2 In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type.

7.12.3.1 The `fpclassify` macro

Synopsis

1
```
#include <math.h>
int fpclassify(real-floating x);
```

Description

2 The **`fpclassify`** macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[286)]

Returns

3 The **`fpclassify`** macro returns the value of the number classification macro appropriate to the value of its argument.

7.12.3.2 The `iscanonical` macro

Synopsis

1
```
#include <math.h>
int iscanonical(real-floating x);
```

Description

2 The **`iscanonical`** macro determines whether its argument value is canonical (5.2.4.2.2). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then, determination is based on the type of the argument.

Returns

3 The **`iscanonical`** macro returns a nonzero value if and only if its argument is canonical.

7.12.3.3 The `isfinite` macro

Synopsis

1
```
#include <math.h>
int isfinite(real-floating x);
```

Description

2 The **`isfinite`** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **`isfinite`** macro returns a nonzero value if and only if its argument has a finite value.

7.12.3.4 The `isinf` macro

Synopsis

1
```
#include <math.h>
int isinf(real-floating x);
```

[286)]Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal **`long double`** value might become subnormal when converted to **`double`**, and zero when converted to **`float`**.

Description

2 The **isinf** macro determines whether its argument value is (positive or negative) infinity. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **isinf** macro returns a nonzero value if and only if its argument has an infinite value.

7.12.3.5 The isnan macro

Synopsis

1
```
#include <math.h>
int isnan(real-floating x);
```

Description

2 The **isnan** macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.[287)]

Returns

3 The **isnan** macro returns a nonzero value if and only if its argument has a NaN value.

7.12.3.6 The isnormal macro

Synopsis

1
```
#include <math.h>
int isnormal(real-floating x);
```

Description

2 The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **isnormal** macro returns a nonzero value if and only if its argument has a normal value.

7.12.3.7 The signbit macro

Synopsis

1
```
#include <math.h>
int signbit(real-floating x);
```

Description

2 The **signbit** macro determines whether the sign of its argument value is negative[288)]. If the argument value is an unsigned zero, its sign is regarded as positive. Otherwise, if the argument value is unsigned, the result value (zero or nonzero) is implementation-defined.

Returns

3 The **signbit** macro returns a nonzero value if and only if the sign of its argument value is determined to be negative.

7.12.3.8 The issignaling macro

[287)]For the **isnan** macro, the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

[288)]The **signbit** macro determines the sign of all values, including infinities, zeros, and NaNs.

Synopsis

1
```
#include <math.h>
int issignaling(real-floating x);
```

Description

2 The **issignaling** macro determines whether its argument value is a signaling NaN.

Returns

3 The **issignaling** macro returns a nonzero value if and only if its argument is a signaling NaN.[289]

7.12.3.9 The issubnormal macro

Synopsis

1
```
#include <math.h>
int issubnormal(real-floating x);
```

Description

2 The **issubnormal** macro determines whether its argument value is subnormal. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

3 The **issubnormal** macro returns a nonzero value if and only if its argument is subnormal.

7.12.3.10 The iszero macro

Synopsis

1
```
#include <math.h>
int iszero(real-floating x);
```

Description

2 The **iszero** macro determines whether its argument value is (positive, negative, or unsigned) zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then, determination is based on the type of the argument.

Returns

3 The **iszero** macro returns a nonzero value if and only if its argument is zero.

7.12.4 Trigonometric functions

7.12.4.1 The acos functions

Synopsis

1
```
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 acosd32(_Decimal32 x);
_Decimal64 acosd64(_Decimal64 x);
_Decimal128 acosd128(_Decimal128 x);
#endif
```

289) F.3 specifies that **issignaling** (and all the other classification macros), raise no floating-point exception if the argument is a variable, or any other expression whose value is represented in the format of its semantic type, even if the value is a signaling NaN.

Description

2 The **acos** functions compute the principal value of the arc cosine of x. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **acos** functions return $\arccos$ x in the interval $[0, \pi]$ radians.

7.12.4.2 The asin functions

Synopsis

1
```
#include <math.h>
double asin(double x);
float asinf(float x);
long double asinl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 asind32(_Decimal32 x);
_Decimal64 asind64(_Decimal64 x);
_Decimal128 asind128(_Decimal128 x);
#endif
```

Description

2 The **asin** functions compute the principal value of the arc sine of x. A domain error occurs for arguments not in the interval $[-1, +1]$. A range error occurs if nonzero x is too close to zero.

Returns

3 The **asin** functions return $\arcsin$ x in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

7.12.4.3 The atan functions

Synopsis

1
```
#include <math.h>
double atan(double x);
float atanf(float x);
long double atanl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 atand32(_Decimal32 x);
_Decimal64 atand64(_Decimal64 x);
_Decimal128 atand128(_Decimal128 x);
#endif
```

Description

2 The **atan** functions compute the principal value of the arc tangent of x. A range error occurs if nonzero x is too close to zero.

Returns

3 The **atan** functions return $\arctan$ x in the interval $[-\frac{\pi}{2}, +\frac{\pi}{2}]$ radians.

7.12.4.4 The atan2 functions

Synopsis

1
```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
_Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
_Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);
#endif
```

Description

2 The **atan2** functions compute the value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero. A range error occurs if x is positive and nonzero $\frac{y}{x}$ is too close to zero.

Returns

3 The **atan2** functions return $\arctan(y/x)$ in the interval $[-\pi, +\pi]$ radians.

7.12.4.5 The cos functions

Synopsis

1
```
#include <math.h>
double cos(double x);
float cosf(float x);
long double cosl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 cosd32(_Decimal32 x);
_Decimal64 cosd64(_Decimal64 x);
_Decimal128 cosd128(_Decimal128 x);
#endif
```

Description

2 The **cos** functions compute the cosine of x (measured in radians).

Returns

3 The **cos** functions return $\cos x$.

7.12.4.6 The sin functions

Synopsis

1
```
#include <math.h>
double sin(double x);
float sinf(float x);
long double sinl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 sind32(_Decimal32 x);
_Decimal64 sind64(_Decimal64 x);
_Decimal128 sind128(_Decimal128 x);
#endif
```

Description

2 The **sin** functions compute the sine of x (measured in radians). A range error occurs if nonzero x is too close to zero.

Returns

3 The **sin** functions return $\sin x$.

7.12.4.7 The tan functions

Synopsis

1
```
#include <math.h>
double tan(double x);
float tanf(float x);
long double tanl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 tand32(_Decimal32 x);
_Decimal64 tand64(_Decimal64 x);
_Decimal128 tand128(_Decimal128 x);
#endif
```

Description

2 The **tan** functions return the tangent of x (measured in radians). A range error occurs if nonzero x is too close to zero.

Returns

3 The **tan** functions return $\tan$ x.

7.12.4.8 The acospi functions

Synopsis

1
```
#include <math.h>
double acospi(double x);
float acospif(float x);
long double acospil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 acospid32(_Decimal32 x);
_Decimal64 acospid64(_Decimal64 x);
_Decimal128 acospid128(_Decimal128 x);
#endif
```

Description

2 The **acospi** functions compute the principal value of the arc cosine of x, divided by π, thus measuring the angle in half-revolutions. A domain error occurs for arguments not in the interval $[-1, +1]$.

Returns

3 The **acospi** functions return $\arccos(\texttt{x})/\pi$ in the interval $[0, 1]$.

7.12.4.9 The asinpi functions

Synopsis

1
```
#include <math.h>
double asinpi(double x);
float asinpif(float x);
long double asinpil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 asinpid32(_Decimal32 x);
_Decimal64 asinpid64(_Decimal64 x);
_Decimal128 asinpid128(_Decimal128 x);
#endif
```

Description

2 The **asinpi** functions compute the principal value of the arc sine of x, divided by π, thus measuring the angle in half-revolutions. A domain error occurs for arguments not in the interval $[-1, +1]$. A range error occurs if nonzero x is too close to zero.

Returns

3 The **asinpi** functions return $\arcsin(\texttt{x})/\pi$ in the interval $[-\frac{1}{2}, +\frac{1}{2}]$.

7.12.4.10 The atanpi functions

Synopsis

1
```
#include <math.h>
double atanpi(double x);
float atanpif(float x);
long double atanpil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 atanpid32(_Decimal32 x);
_Decimal64 atanpid64(_Decimal64 x);
_Decimal128 atanpid128(_Decimal128 x);
#endif
```

Description

2 The **atanpi** functions compute the principal value of the arc tangent of x, divided by π, thus measuring the angle in half-revolutions. A range error occurs if nonzero x is too close to zero.

Returns

3 The **atanpi** functions return $\arctan(\mathtt{x})/\pi$. in the interval $[-\frac{1}{2}, +\frac{1}{2}]$.

7.12.4.11 The atan2pi functions

Synopsis

1
```
#include <math.h>
double atan2pi(double y, double x);
float atan2pif(float y, float x);
long double atan2pil(long double y, long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 atan2pid32(_Decimal32 y, _Decimal32 x);
_Decimal64 atan2pid64(_Decimal64 y, _Decimal64 x);
_Decimal128 atan2pid128(_Decimal128 y, _Decimal128 x);
#endif
```

Description

2 The **atan2pi** functions compute the angle, measured in half-revolutions, subtended at the origin by the point $(\mathtt{x}, \mathtt{y})$ and the positive x-axis. Thus, the **atan2pi** functions compute $\arctan(\frac{y}{x})/\pi$, in the range $[-1, +1]$. A domain error may occur if both arguments are zero. A range error occurs if x is positive and nonzero $\frac{\mathtt{y}}{\mathtt{x}}$ is too close to zero.

Returns

3 The **atan2pi** functions return the computed angle, in the interval $[-1, +1]$.

7.12.4.12 The cospi functions

Synopsis

1
```
#include <math.h>
double cospi(double x);
float cospif(float x);
long double cospil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 cospid32(_Decimal32 x);
_Decimal64 cospid64(_Decimal64 x);
_Decimal128 cospid128(_Decimal128 x);
#endif
```

Description

2 The **cospi** functions compute the cosine of $\pi \times \mathtt{x}$, thus regarding x as a measurement in half-revolutions.

Returns

3 The **cospi** functions return $\cos(\pi \times \mathtt{x})$.

7.12.4.13 The sinpi functions

Synopsis

1
```
#include <math.h>
double sinpi(double x);
float sinpif(float x);
long double sinpil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 sinpid32(_Decimal32 x);
_Decimal64 sinpid64(_Decimal64 x);
```

```
    _Decimal128 sinpid128(_Decimal128 x);
    #endif
```

Description

2 The **sinpi** functions compute the sine of $\pi \times$ x, thus regarding x as a measurement in half-revolutions. A range error occurs if nonzero x is too close to zero.

Returns

3 The **sinpi** functions return $\sin(\pi \times$ x$)$.

7.12.4.14 The tanpi functions

Synopsis

1
```
    #include <math.h>
    double tanpi(double x);
    float tanpif(float x);
    long double tanpil(long double x);
    #ifdef __STDC_IEC_60559_DFP__
    _Decimal32 tanpid32(_Decimal32 x);
    _Decimal64 tanpid64(_Decimal64 x);
    _Decimal128 tanpid128(_Decimal128 x);
    #endif
```

Description

2 The **tanpi** functions compute the tagent of $\pi \times$ x, thus regarding x as a measurement in half-revolutions. A range error occurs if nonzero x is too close to zero.

Returns

3 The **tanpi** functions return $\tan(\pi \times$ x$)$.

7.12.5 Hyperbolic functions

7.12.5.1 The acosh functions

Synopsis

1
```
    #include <math.h>
    double acosh(double x);
    float acoshf(float x);
    long double acoshl(long double x);
    #ifdef __STDC_IEC_60559_DFP__
    _Decimal32 acoshd32(_Decimal32 x);
    _Decimal64 acoshd64(_Decimal64 x);
    _Decimal128 acoshd128(_Decimal128 x);
    #endif
```

Description

2 The **acosh** functions compute the (nonnegative) arc hyperbolic cosine of x. A domain error occurs for arguments less than 1.

Returns

3 The **acosh** functions return arcosh x in the interval $[0, +\infty]$.

7.12.5.2 The asinh functions

Synopsis

1
```
    #include <math.h>
    double asinh(double x);
    float asinhf(float x);
    long double asinhl(long double x);
```

```
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 asinhd32(_Decimal32 x);
_Decimal64 asinhd64(_Decimal64 x);
_Decimal128 asinhd128(_Decimal128 x);
#endif
```

Description

2 The `asinh` functions compute the arc hyperbolic sine of `x`. A range error occurs if nonzero `x` is too close to zero.

Returns

3 The `asinh` functions return $\operatorname{arsinh} \texttt{x}$.

7.12.5.3 The `atanh` functions

Synopsis

1
```
#include <math.h>
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 atanhd32(_Decimal32 x);
_Decimal64 atanhd64(_Decimal64 x);
_Decimal128 atanhd128(_Decimal128 x);
#endif
```

Description

2 The `atanh` functions compute the arc hyperbolic tangent of `x`. A domain error occurs for arguments not in the interval $[-1, +1]$. A pole error may occur if the argument equals `-1` or `+1`. A range error occurs if nonzero `x` is too close to zero.

Returns

3 The `atanh` functions return $\operatorname{artanh} \texttt{x}$.

7.12.5.4 The `cosh` functions

Synopsis

1
```
#include <math.h>
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 coshd32(_Decimal32 x);
_Decimal64 coshd64(_Decimal64 x);
_Decimal128 coshd128(_Decimal128 x);
#endif
```

Description

2 The `cosh` functions compute the hyperbolic cosine of `x`. A range error occurs if the magnitude of finite `x` is too large.

Returns

3 The `cosh` functions return $\cosh \texttt{x}$.

7.12.5.5 The `sinh` functions

Synopsis

1
```
#include <math.h>
```

```
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 sinhd32(_Decimal32 x);
_Decimal64 sinhd64(_Decimal64 x);
_Decimal128 sinhd128(_Decimal128 x);
#endif
```

Description

2 The `sinh` functions compute the hyperbolic sine of `x`. A range error occurs if the magnitude of finite `x` is too large or if nonzero `x` is too close to zero.

Returns

3 The `sinh` functions return $\sinh$ `x`.

7.12.5.6 The `tanh` functions

Synopsis

1
```
#include <math.h>
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 tanhd32(_Decimal32 x);
_Decimal64 tanhd64(_Decimal64 x);
_Decimal128 tanhd128(_Decimal128 x);
#endif
```

Description

2 The `tanh` functions compute the hyperbolic tangent of `x`. A range error occurs if nonzero `x` is too close to zero.

Returns

3 The `tanh` functions return $\tanh$ `x`.

7.12.6 Exponential and logarithmic functions

7.12.6.1 The `exp` functions

Synopsis

1
```
#include <math.h>
double exp(double x);
float expf(float x);
long double expl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 expd32(_Decimal32 x);
_Decimal64 expd64(_Decimal64 x);
_Decimal128 expd128(_Decimal128 x);
#endif
```

Description

2 The `exp` functions compute the base-e exponential of `x`. A range error occurs if the magnitude of finite `x` is too large.

Returns

3 The `exp` functions return $e^{\mathtt{x}}$.

7.12.6.2 The `exp10` functions

Synopsis

1
```
#include <math.h>
double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 exp10d32(_Decimal32 x);
_Decimal64 exp10d64(_Decimal64 x);
_Decimal128 exp10d128(_Decimal128 x);
#endif
```

Description

2 The **exp10** functions compute the base-10 exponential of x. A range error occurs if the magnitude of finite x is too large.

Returns

3 The **exp10** functions return 10^{x}.

7.12.6.3 The exp10m1 functions

Synopsis

1
```
#include <math.h>
double exp10m1(double x);
float exp10m1f(float x);
long double exp10m1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 exp10m1d32(_Decimal32 x);
_Decimal64 exp10m1d64(_Decimal64 x);
_Decimal128 exp10m1d128(_Decimal128 x);
#endif
```

Description

2 The **exp10m1** functions compute the base-10 exponential of the argument, minus 1. A range error occurs if positive finite x is too large or if nonzero x is too close to zero.

Returns

3 The **exp10m1** functions return $10^{x} - 1$.

7.12.6.4 The exp2 functions

Synopsis

1
```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 exp2d32(_Decimal32 x);
_Decimal64 exp2d64(_Decimal64 x);
_Decimal128 exp2d128(_Decimal128 x);
#endif
```

Description

2 The **exp2** functions compute the base-2 exponential of x. A range error occurs if the magnitude of finite x is too large.

Returns

3 The **exp2** functions return 2^{x}.

7.12.6.5 The exp2m1 functions

Synopsis

1
```
#include <math.h>
double exp2m1(double x);
float exp2m1f(float x);
long double exp2m1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 exp2m1d32(_Decimal32 x);
_Decimal64 exp2m1d64(_Decimal64 x);
_Decimal128 exp2m1d128(_Decimal128 x);
#endif
```

Description

2 The **exp2m1** functions compute the base-2 exponential of the argument, minus 1. A range error occurs if positive finite x is too large or if nonzero x is too close to zero.

Returns

3 The **exp2m1** functions return $2^{x} - 1$.

7.12.6.6 The expm1 functions

Synopsis

1
```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 expm1d32(_Decimal32 x);
_Decimal64 expm1d64(_Decimal64 x);
_Decimal128 expm1d128(_Decimal128 x);
#endif
```

Description

2 The **expm1** functions compute the base-e exponential of the argument, minus 1. A range error occurs if positive finite x is too large or if nonzero x is too close to zero. [290)]

Returns

3 The **expm1** functions return $e^{x} - 1$.

7.12.6.7 The frexp functions

Synopsis

1
```
#include <math.h>
double frexp(double value, int *p);
float frexpf(float value, int *p);
long double frexpl(long double value, int *p);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 frexpd32(_Decimal32 value, int *p);
_Decimal64 frexpd64(_Decimal64 value, int *p);
_Decimal128 frexpd128(_Decimal128 value, int *p);
#endif
```

Description

2 The **frexp** functions break a floating-point number into a normalized fraction and an integer exponent. They store the integer in the **int** object pointed to by p. If the type of the function is a

[290)]For small magnitude x, `expm1(x)` is expected to be more accurate than `exp(x)-1`.

standard floating type, the exponent is an integral power of 2. If the type of the function is a decimal floating type, the exponent is an integral power of 10.

Returns

3 If `value` is not a floating-point number or if the integral power is outside the range of **`int`**, the results are unspecified. Otherwise, the **`frexp`** functions return the value `x`, such that `x` has a magnitude in the interval $[\frac{1}{2}, 1)$ or zero, and `value` equals `x` $\times\ 2^{*\texttt{p}}$, when the type of the function is a standard floating type; or `x` has a magnitude in the interval $[1/10, 1)$ or zero, and `value` equals `x` $\times\ 10^{*\texttt{p}}$, when the type of the function is a decimal floating type. If `value` is zero, both parts of the result are zero.

7.12.6.8 The `ilogb` functions

Synopsis

1
```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
#ifdef __STDC_IEC_60559_DFP__
int ilogbd32(_Decimal32 x);
int ilogbd64(_Decimal64 x);
int ilogbd128(_Decimal128 x);
#endif
```

Description

2 The **`ilogb`** functions extract the exponent of `x` as a signed **`int`** value. If `x` is zero they compute the value **`FP_ILOGB0`**; if `x` is infinite they compute the value **`INT_MAX`**; if `x` is a NaN they compute the value **`FP_ILOGBNAN`**; otherwise, they are equivalent to calling the corresponding **`logb`** function and converting the returned value to type **`int`**. A domain error or range error may occur if `x` is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **`ilogb`** functions return the exponent of `x` as a signed **`int`** value.

Forward references: the **`logb`** functions (7.12.6.17).

7.12.6.9 The `ldexp` functions

Synopsis

1
```
#include <math.h>
double ldexp(double x, int p);
float ldexpf(float x, int p);
long double ldexpl(long double x, int p);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 ldexpd32(_Decimal32 x, int p);
_Decimal64 ldexpd64(_Decimal64 x, int p);
_Decimal128 ldexpd128(_Decimal128 x, int p);
#endif
```

Description

2 The **`ldexp`** functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type. A range error occurs for some finite `x`, depending on `p`.

Returns

3 The **`ldexp`** functions return `x` $\times\ 2^{\texttt{p}}$ when the type of the function is a standard floating type, or return `x` $\times\ 10^{\texttt{p}}$ when the type of the function is a decimal floating type.

7.12.6.10 The `llogb` functions

Synopsis

1
```
#include <math.h>
long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
#ifdef __STDC_IEC_60559_DFP__
long int llogbd32(_Decimal32 x);
long int llogbd64(_Decimal64 x);
long int llogbd128(_Decimal128 x);
#endif
```

Description

2 The `llogb` functions extract the exponent of `x` as a signed `long int` value. If `x` is zero they compute the value `FP_LLOGB0`; if `x` is infinite they compute the value `LONG_MAX`; if `x` is a NaN they compute the value `FP_LLOGBNAN`; otherwise, they are equivalent to calling the corresponding `logb` function and converting the returned value to type `long int`. A domain error or range error may occur if `x` is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified.

Returns

3 The `llogb` functions return the exponent of `x` as a signed `long int` value.

Forward references: the `logb` functions (7.12.6.17).

7.12.6.11 The `log` functions

Synopsis

1
```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 logd32(_Decimal32 x);
_Decimal64 logd64(_Decimal64 x);
_Decimal128 logd128(_Decimal128 x);
#endif
```

Description

2 The `log` functions compute the base-e (natural) logarithm of `x`. A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

3 The `log` functions return $\log_e$ `x`.

7.12.6.12 The `log10` functions

Synopsis

1
```
#include <math.h>
double log10(double x);
float log10f(float x);
long double log10l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log10d32(_Decimal32 x);
_Decimal64 log10d64(_Decimal64 x);
_Decimal128 log10d128(_Decimal128 x);
#endif
```

Description

2 The **log10** functions compute the base-10 (common) logarithm of x. A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

3 The **log10** functions return $\log_{10}$ x.

7.12.6.13 The log10p1 functions

Synopsis

1
```
#include <math.h>
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log10p1d32(_Decimal32 x);
_Decimal64 log10p1d64(_Decimal64 x);
_Decimal128 log10p1d128(_Decimal128 x);
#endif
```

Description

2 The **log10p1** functions compute the base-10 logarithm of 1 plus the argument. A domain error occurs if the argument is less than -1. A pole error may occur if the argument equals -1. A range error occurs if nonzero x is too close to zero.

Returns

3 The **log10p1** functions return $\log_{10}(1 + \mathtt{x})$.

7.12.6.14 The log1p and logp1 functions

Synopsis

1
```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
double logp1(double x);
float logp1f(float x);
long double logp1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log1pd32(_Decimal32 x);
_Decimal64 log1pd64(_Decimal64 x);
_Decimal128 log1pd128(_Decimal128 x);
_Decimal32 logp1d32(_Decimal32 x);
_Decimal64 logp1d64(_Decimal64 x);
_Decimal128 logp1d128(_Decimal128 x);
#endif
```

Description

2 The **log1p** functions are equivalent to the **logp1** functions.[291] These functions compute the base-e (natural) logarithm of 1 plus the argument.[292] A domain error occurs if the argument is less than -1. A pole error may occur if the argument equals -1. A range error occurs if nonzero x is too close to zero.

Returns

3 The **log1p** and **logp1** functions return $\log_e(1 + \mathtt{x})$.

---

[291] The **logp1** functions are preferred for name consistency with the **log10p1** and **log2p1** functions.

[292] For small magnitude x, `logp1(x)` is expected to be more accurate than `log(1 + x)`.

7.12.6.15 The log2 functions

Synopsis

1
```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log2d32(_Decimal32 x);
_Decimal64 log2d64(_Decimal64 x);
_Decimal128 log2d128(_Decimal128 x);
#endif
```

Description

2 The **log2** functions compute the base-2 logarithm of x. A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

3 The **log2** functions return $\log_2$ x.

7.12.6.16 The log2p1 functions

Synopsis

1
```
#include <math.h>
double log2p1(double x);
float log2p1f(float x);
long double log2p1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log2p1d32(_Decimal32 x);
_Decimal64 log2p1d64(_Decimal64 x);
_Decimal128 log2p1d128(_Decimal128 x);
#endif
```

Description

2 The **log2p1** functions compute the base-2 logarithm of 1 plus the argument. A domain error occurs if the argument is less than -1. A pole error may occur if the argument equals -1. A range error occurs if nonzero x is too close to zero.

Returns

3 The **log2p1** functions return $\log_2$(1+x).

7.12.6.17 The logb functions

Synopsis

1
```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 logbd32(_Decimal32 x);
_Decimal64 logbd64(_Decimal64 x);
_Decimal128 logbd128(_Decimal128 x);
#endif
```

Description

2 The **logb** functions extract the exponent of x, as a signed integer value in floating-point format. If x is subnormal it is treated as though it were normalized; thus, for positive finite x,

$$1 \leq \mathtt{x} \times b^{-\mathtt{logb(x)}} < b$$

where $b =$ FLT_RADIX if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type. A domain error or pole error may occur if the argument is zero.

Returns

3 The logb functions return the signed exponent of x.

7.12.6.18 The modf functions

Synopsis

1
```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 modfd32(_Decimal32 x, _Decimal32 *iptr);
_Decimal64 modfd64(_Decimal64 x, _Decimal64 *iptr);
_Decimal128 modfd128(_Decimal128 x, _Decimal128 *iptr);
#endif
```

Description

2 The modf functions break the argument value into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in floating-point format) in the object pointed to by iptr.

Returns

3 The modf functions return the signed fractional part of value.

7.12.6.19 The scalbn and scalbln functions

Synopsis

1
```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 scalbnd32(_Decimal32 x, int n);
_Decimal64 scalbnd64(_Decimal64 x, int n);
_Decimal128 scalbnd128(_Decimal128 x, int n);
_Decimal32 scalblnd32(_Decimal32 x, long int n);
_Decimal64 scalblnd64(_Decimal64 x, long int n);
_Decimal128 scalblnd128(_Decimal128 x, long int n);
#endif
```

Description

2 The scalbn and scalbln functions compute $\texttt{x} \times b^{\texttt{n}}$, where $b =$ FLT_RADIX if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type. A range error occurs for some finite x, depending on n.

Returns

3 The scalbn and scalbln functions return $\texttt{x} \times b^{\texttt{n}}$.

7.12.7 Power and absolute-value functions

7.12.7.1 The cbrt functions

Synopsis

1

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 cbrtd32(_Decimal32 x);
_Decimal64 cbrtd64(_Decimal64 x);
_Decimal128 cbrtd128(_Decimal128 x);
#endif
```

Description

2 The **cbrt** functions compute the real cube root of x.

Returns

3 The **cbrt** functions return $x^{\frac{1}{3}}$.

7.12.7.2 The compoundn functions

Synopsis

1
```
#include <stdint.h>
#include <math.h>
double compoundn(double x, long long int n);
float compoundnf(float x, long long int n);
long double compoundnl(long double x, long long int n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 compoundnd32(_Decimal32 x, long long int n);
_Decimal64 compoundnd64(_Decimal64 x, long long int n);
_Decimal128 compoundnd128(_Decimal128 x, long long int n);
#endif
```

Description

2 The **compoundn** functions compute 1 plus x, raised to the power n. A domain error occurs if $x < -1$. Depending on n, a range error occurs if either positive finite x is too large or if x is too near but not equal to -1. A pole error may occur if x equals -1 and $n < 0$.

Returns

3 The **compoundn** functions return $(1 + x)^n$.

7.12.7.3 The fabs functions

Synopsis

1
```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fabsd32(_Decimal32 x);
_Decimal64 fabsd64(_Decimal64 x);
_Decimal128 fabsd128(_Decimal128 x);
#endif
```

Description

2 The **fabs** functions compute the absolute value of x.

Returns

3 The **fabs** functions return $|x|$.

7.12.7.4 The hypot functions

Synopsis

1

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal64 hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **hypot** functions compute the square root of the sum of the squares of x and y, without undue overflow or underflow. A range error occurs for some finite arguments.

3

Returns

4 The **hypot** functions return $\sqrt{x^2 + y^2}$.

7.12.7.5 The pow functions

Synopsis

1

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 powd32(_Decimal32 x, _Decimal32 y);
_Decimal64 powd64(_Decimal64 x, _Decimal64 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **pow** functions compute x raised to the power y. A domain error occurs if x is finite and less than zero and y is finite and not an integer value. A domain error may occur if x is zero and y is zero. Depending on y, a range error occurs if either the magnitude of nonzero finite x is too large or too near zero. A domain error or pole error may occur if x is zero and y is less than zero.

Returns

3 The **pow** functions return x^y.

7.12.7.6 The pown functions

Synopsis

1

```
#include <stdint.h>
#include <math.h>
double pown(double x, long long int n);
float pownf(float x, long long int n);
long double pownl(long double x, long long int n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 pownd32(_Decimal32 x, long long int n);
_Decimal64 pownd64(_Decimal64 x, long long int n);
_Decimal128 pownd128(_Decimal128 x, long long int n);
#endif
```

Description

2 The **pown** functions compute `x` raised to the `n`th power. A pole error may occur if `x` equals 0 and `n` < 0. Depending on `n`, a range error occurs if either the magnitude of nonzero finite `x` is too large or too near zero.

Returns

3 The **pown** functions return $\mathtt{x}^{\mathtt{n}}$.

7.12.7.7 The powr functions

Synopsis

1
```
#include <math.h>
double powr(double y, double x);
float powrf(float y, float x);
long double powrl(long double y, long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 powrd32(_Decimal32 y, _Decimal32 x);
_Decimal64 powrd64(_Decimal64 y, _Decimal64 x);
_Decimal128 powrd128(_Decimal128 y, _Decimal128 x);
#endif
```

Description

2 The **powr** functions compute `x` raised to the power `y` as $e^{\mathtt{y} \log_e \mathtt{x}}$.[293] A domain error occurs if `x` < 0 or if `x` and `y` are both zero. Depending on `y`, a range error occurs if either positive nonzero finite `x` is too large or too near zero. A pole error may occur if `x` equals zero and finite `y` < 0.

Returns

3 The **powr** functions return $e^{\mathtt{y} \log_e \mathtt{x}}$.

7.12.7.8 The rootn functions

Synopsis

1
```
#include <stdint.h>
#include <math.h>
double rootn(double x, long long int n);
float rootnf(float x, long long int n);
long double rootnl(long double x, long long int n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 rootnd32(_Decimal32 x, long long int n);
_Decimal64 rootnd64(_Decimal64 x, long long int n);
_Decimal128 rootnd128(_Decimal128 x, long long int n);
#endif
```

Description

2 The **rootn** functions compute the principal `n`th root of `x`. A domain error occurs if `n` is 0 or if `x` < 0 and `n` is even. If `n` is -1, a range error occurs if either the magnitude of nonzero finite `x` is too large or too near zero. A pole error may occur if `x` equals zero and `n` < 0.

Returns

3 The **rootn** functions return $\mathtt{x}^{\frac{1}{\mathtt{n}}}$.

7.12.7.9 The rsqrt functions

Synopsis

1
```
#include <math.h>
double rsqrt(double x);
```

[293] Restricting the domain to that of the formula $e^{\mathtt{y} \log_e \mathtt{x}}$ is intended to better meet expectations for a continuous power function and to allow implementations with fewer tests for special cases.

```
float rsqrtf(float x);
long double rsqrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 rsqrtd32(_Decimal32 x);
_Decimal64 rsqrtd64(_Decimal64 x);
_Decimal128 rsqrtd128(_Decimal128 x);
#endif
```

Description

2 The **rsqrt** functions compute the reciprocal of the nonnegative square root of the argument. A domain error occurs if the argument is less than zero. A pole error may occur if the argument equals zero.

Returns

3 The **rsqrt** functions return $\frac{1}{\sqrt{x}}$.

7.12.7.10 The sqrt functions

Synopsis

1
```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 sqrtd32(_Decimal32 x);
_Decimal64 sqrtd64(_Decimal64 x);
_Decimal128 sqrtd128(_Decimal128 x);
#endif
```

Description

2 The **sqrt** functions compute the nonnegative square root of x. A domain error occurs if the argument is less than zero.

Returns

3 The **sqrt** functions return $\sqrt{x}$.

7.12.8 Error and gamma functions

7.12.8.1 The erf functions

Synopsis

1
```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 erfd32(_Decimal32 x);
_Decimal64 erfd64(_Decimal64 x);
_Decimal128 erfd128(_Decimal128 x);
#endif
```

Description

2 The **erf** functions compute the error function of x. A range error occurs if nonzero x is too close to zero.

Returns

3 The **erf** functions return $\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$.

7.12.8.2 The erfc functions

Synopsis

1
```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 erfcd32(_Decimal32 x);
_Decimal64 erfcd64(_Decimal64 x);
_Decimal128 erfcd128(_Decimal128 x);
#endif
```

Description

2 The **erfc** functions compute the complementary error function of x. A range error occurs if positive finite x is too large.

Returns

3 The **erfc** functions return $\operatorname{erfc} \mathtt{x} = 1 - \operatorname{erf} \mathtt{x} = \frac{2}{\sqrt{\pi}} \int_{\mathtt{x}}^{\infty} e^{-t^2}\, dt$.

7.12.8.3 The lgamma functions

Synopsis

1
```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 lgammad32(_Decimal32 x);
_Decimal64 lgammad64(_Decimal64 x);
_Decimal128 lgammad128(_Decimal128 x);
#endif
```

Description

2 The **lgamma** functions compute the natural logarithm of the absolute value of gamma of x. A range error occurs if positive finite x is too large. A pole error may occur if x is a negative integer or zero.

Returns

3 The **lgamma** functions return $\log_e |\Gamma(\mathtt{x})|$.

7.12.8.4 The tgamma functions

Synopsis

1
```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 tgammad32(_Decimal32 x);
_Decimal64 tgammad64(_Decimal64 x);
_Decimal128 tgammad128(_Decimal128 x);
#endif
```

Description

2 The **tgamma** functions compute the gamma function of x. A domain error or pole error may occur if x is a negative integer or zero. A range error occurs for some finite x less than zero, if positive finite x is too large, or nonzero x is too close to zero.

Returns

3 The **tgamma** functions return $\Gamma(x)$.

7.12.9 Nearest integer functions

7.12.9.1 The ceil functions

Synopsis

1
```
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 ceild32(_Decimal32 x);
_Decimal64 ceild64(_Decimal64 x);
_Decimal128 ceild128(_Decimal128 x);
#endif
```

Description

2 The **ceil** functions compute the smallest integer value not less than x.

Returns

3 The **ceil** functions return $\lceil x \rceil$, expressed as a floating-point number.

7.12.9.2 The floor functions

Synopsis

1
```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 floord32(_Decimal32 x);
_Decimal64 floord64(_Decimal64 x);
_Decimal128 floord128(_Decimal128 x);
#endif
```

Description

2 The **floor** functions compute the largest integer value not greater than x.

Returns

3 The **floor** functions return $\lfloor x \rfloor$, expressed as a floating-point number.

7.12.9.3 The nearbyint functions

Synopsis

1
```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nearbyintd32(_Decimal32 x);
_Decimal64 nearbyintd64(_Decimal64 x);
_Decimal128 nearbyintd128(_Decimal128 x);
#endif
```

Description

2 The **nearbyint** functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the "inexact" floating-point exception.

Returns

3 The **nearbyint** functions return the rounded integer value.

7.12.9.4 The rint functions

Synopsis

1
```
#include <math.h>
double rint(double x);
float rintf(float x);
long double rintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 rintd32(_Decimal32 x);
_Decimal64 rintd64(_Decimal64 x);
_Decimal128 rintd128(_Decimal128 x);
#endif
```

Description

2 The **rint** functions differ from the **nearbyint** functions (7.12.9.3) only in that the **rint** functions may raise the "inexact" floating-point exception if the result differs in value from the argument.

Returns

3 The **rint** functions return the rounded integer value.

7.12.9.5 The lrint and llrint functions

Synopsis

1
```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
long int lrintd32(_Decimal32 x);
long int lrintd64(_Decimal64 x);
long int lrintd128(_Decimal128 x);
long long int llrintd32(_Decimal32 x);
long long int llrintd64(_Decimal64 x);
long long int llrintd128(_Decimal128 x);
#endif
```

Description

2 The **lrint** and **llrint** functions round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lrint** and **llrint** functions return the rounded integer value.

7.12.9.6 The round functions

Synopsis

1
```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 roundd32(_Decimal32 x);
```

```
_Decimal64 roundd64(_Decimal64 x);
_Decimal128 roundd128(_Decimal128 x);
#endif
```

Description

2 The **round** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Returns

3 The **round** functions return the rounded integer value.

7.12.9.7 The lround and llround functions

Synopsis

1
```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
#ifdef __STDC_IEC_60559_DFP__
long int lroundd32(_Decimal32 x);
long int lroundd64(_Decimal64 x);
long int lroundd128(_Decimal128 x);
long long int llroundd32(_Decimal32 x);
long long int llroundd64(_Decimal64 x);
long long int llroundd128(_Decimal128 x);
#endif
```

Description

2 The **lround** and **llround** functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lround** and **llround** functions return the rounded integer value.

7.12.9.8 The roundeven functions

Synopsis

1
```
#include <math.h>
double roundeven(double x);
float roundevenf(float x);
long double roundevenl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 roundevend32(_Decimal32 x);
_Decimal64 roundevend64(_Decimal64 x);
_Decimal128 roundevend128(_Decimal128 x);
#endif
```

Description

2 The **roundeven** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases to even (that is, to the nearest value that is an even integer), regardless of the current rounding direction.

Returns

3 The **roundeven** functions return the rounded integer value.

7.12.9.9 The trunc functions

Synopsis

1
```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 truncd32(_Decimal32 x);
_Decimal64 truncd64(_Decimal64 x);
_Decimal128 truncd128(_Decimal128 x);
#endif
```

Description

2 The **trunc** functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

3 The **trunc** functions return the truncated integer value.

7.12.9.10 The fromfp and ufromfp functions

Synopsis

1
```
#include <stdint.h>
#include <math.h>
double fromfp(double x, int rnd, unsigned int width);
float fromfpf(float x, int rnd, unsigned int width);
long double fromfpl(long double x, int rnd, unsigned int width);
double ufromfp(double x, int rnd, unsigned int width);
float ufromfpf(float x, int rnd, unsigned int width);
long double ufromfpl(long double x, int rnd, unsigned int width);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fromfpd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 fromfpd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 fromfpd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 ufromfpd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 ufromfpd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 ufromfpd128(_Decimal128 x, int rnd, unsigned int width);
#endif
```

Description

2 The **fromfp** and **ufromfp** functions round `x`, using the math rounding direction indicated by `rnd`, to a signed or unsigned integer, respectively. If `width` is nonzero and the resulting integer is within the range

— $[-2^{(\texttt{width}-1)}, 2^{(\texttt{width}-1)} - 1]$, for signed

— $[0, 2^{\texttt{width}} - 1]$, for unsigned

then the functions return the integer value (represented in floating type). Otherwise, if `width` is zero or `x` does not round to an integer within the range, the functions return a `NaN` (of the type of the `x` argument, if available), else the value of `x`, and a domain error occurs. If the value of the `rnd` argument is not equal to the value of a math rounding direction macro (7.12), the direction of rounding is unspecified. The **fromfp** and **ufromfp** functions do not raise the "inexact" floating-point exception.

Returns

3 The **fromfp** and **ufromfp** functions return the rounded integer value.

4 **EXAMPLE** Upward rounding of **double** x to type **int**, without raising the "inexact" floating-point exception, is achieved by

```
(int)fromfp(x, FP_INT_UPWARD, INT_WIDTH)
```

5 **EXAMPLE** Unsigned integer wrapping is not performed in

```
ufromfp(-3.0, FP_INT_UPWARD, UINT_WIDTH) /* domain error */
```

7.12.9.11 The fromfpx and ufromfpx functions

Synopsis

1
```
#include <stdint.h>
#include <math.h>
double fromfpx(double x, int rnd, unsigned int width);
float fromfpxf(float x, int rnd, unsigned int width);
long double fromfpxl(long double x, int rnd, unsigned int width);
double ufromfpx(double x, int rnd, unsigned int width);
float ufromfpxf(float x, int rnd, unsigned int width);
long double ufromfpxl(long double x, int rnd, unsigned int width);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fromfpxd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 fromfpxd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 fromfpxd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 ufromfpxd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 ufromfpxd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 ufromfpxd128(_Decimal128 x, int rnd, unsigned int width);
#endif
```

Description

2 The **fromfpx** and **ufromfpx** functions differ from the **fromfp** and **ufromfp** functions, respectively, only in that the **fromfpx** and **ufromfpx** functions raise the "inexact" floating-point exception if a rounded result not exceeding the specified width differs in value from the argument x.

Returns

3 The **fromfpx** and **ufromfpx** functions return the rounded integer value.

4 **NOTE 1** Conversions to integer types that are not required to raise the inexact exception can be done simply by rounding to integral value in floating type and then converting to the target integer type. For example, the conversion of **long double** x to **uint64_t**, using upward rounding, is done by

```
(uint64_t)ceill(x)
```

7.12.10 Remainder functions

7.12.10.1 The fmod functions

Synopsis

1
```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fmod** functions compute the floating-point remainder of x/y.

Returns

3 The **fmod** functions return the value x $-$ ny, for some integer n such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the **fmod** functions return zero is implementation-defined.

7.12.10.2 The remainder functions

Synopsis

1
```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 remainderd32(_Decimal32 x, _Decimal32 y);
_Decimal64 remainderd64(_Decimal64 x, _Decimal64 y);
_Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **remainder** functions compute the remainder x REM y required by IEC 60559. [294)]

Returns

3 The **remainder** functions return x REM y. If y is zero, whether a domain error occurs or the functions return zero is implementation-defined.

7.12.10.3 The remquo functions

Synopsis

1
```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

Description

2 The **remquo** functions compute the same remainder as the **remainder** functions. In the object pointed to by quo they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of x/y, where n is an implementation-defined integer greater than or equal to 3.

Returns

3 The **remquo** functions return x REM y. If y is zero, the value stored in the object pointed to by quo is unspecified and whether a domain error occurs or the functions return zero is implementation-defined.

4 **NOTE 1** There are no decimal floating-point versions of the **remquo** functions.

7.12.11 Manipulation functions

7.12.11.1 The copysign functions

Synopsis

1
```
#include <math.h>
double copysign(double x, double y);
```

[294)]"When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of $\frac{x}{y}$; whenever $|n - \frac{x}{y}| = \frac{1}{2}$, then n is even. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

```
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 copysignd32(_Decimal32 x, _Decimal32 y);
_Decimal64 copysignd64(_Decimal64 x, _Decimal64 y);
_Decimal128 copysignd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The `copysign` functions produce a value with the magnitude of `x` and the sign of `y`. If `x` or `y` is an unsigned value, the sign (if any) of the result is implementation-defined. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the `copysign` functions should regard the sign of zero as positive.

Returns

3 The `copysign` functions return a value with the magnitude of `x` and the sign of `y`.

7.12.11.2 The `nan` functions

Synopsis

1
```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nand32(const char *tagp);
_Decimal64 nand64(const char *tagp);
_Decimal128 nand128(const char *tagp);
#endif
```

Description

2 The `nan`, `nanf`, and `nanl` functions convert the string pointed to by `tagp` according to the following rules. The call `nan("`*n-char-sequence*`")` is equivalent to `strtod("NAN(`*n-char-sequence*`)", nullptr)`; the call `nan("")` is equivalent to `strtod("NAN()", nullptr)`. If `tagp` does not point to an empty string or an n-char sequence, the call is equivalent to `strtod("NAN", nullptr)`. Calls to `nanf` and `nanl` are equivalent to the corresponding calls to `strtof` and `strtold`.

Returns

3 The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the `strtod`, `strtof`, and `strtold` functions (7.24.1.5).

7.12.11.3 The `nextafter` functions

Synopsis

1
```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nextafterd32(_Decimal32 x, _Decimal32 y);
_Decimal64 nextafterd64(_Decimal64 x, _Decimal64 y);
_Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **nextafter** functions determine the next representable value, in the type of the function, after x in the direction of y, where x and y are first converted to the type of the function.[295] The **nextafter** functions return y if x equals y.

A range error occurs if the magnitude of x is the largest finite value representable in the type and the result is infinite or not representable in the type. If x != y, a range error occurs for either subnormal or zero results.

Returns

3 The **nextafter** functions return the next representable value in the specified format after x in the direction of y.

7.12.11.4 The nexttoward functions

Synopsis

1
```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nexttowardd32(_Decimal32 x, _Decimal128 y);
_Decimal64 nexttowardd64(_Decimal64 x, _Decimal128 y);
_Decimal128 nexttowardd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **nexttoward** functions are equivalent to the **nextafter** functions except that the second parameter has type **long double** or **_Decimal128** and the functions return y converted to the type of the function if x equals y.[296]

7.12.11.5 The nextup functions

Synopsis

1
```
#include <math.h>
double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nextupd32(_Decimal32 x);
_Decimal64 nextupd64(_Decimal64 x);
_Decimal128 nextupd128(_Decimal128 x);
#endif
```

Description

2 The **nextup** functions determine the next representable value, in the type of the function, greater than x. If x is the negative number of least magnitude in the type of x, **nextup**(x) is -0 if the type has signed zeros and is 0 otherwise. If x is zero, **nextup**(x) is the positive number of least magnitude in the type of x. If x is the positive number (finite or infinite) of maximum magnitude in the type, **nextup**(x) is x.

Returns

3 The nextup functions return the next representable value in the specified type greater than x.

7.12.11.6 The nextdown functions

[295] The argument values are converted to the type of the function, even by a macro implementation of the function.
[296] The result of the **nexttoward** functions is determined in the type of the function, without loss of range or precision in a floating second argument.

Synopsis

1
```
#include <math.h>
double nextdown(double x);
float nextdownf(float x);
long double nextdownl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 nextdownd32(_Decimal32 x);
_Decimal64 nextdownd64(_Decimal64 x);
_Decimal128 nextdownd128(_Decimal128 x);
#endif
```

Description

2 The `nextdown` functions determine the next representable value, in the type of the function, less than `x`. If `x` is the positive number of least magnitude in the type of `x`, `nextdown(x)` is `+0` if the type has signed zeros and is `0` otherwise. If `x` is zero, `nextdown(x)` is the negative number of least magnitude in the type of `x`. If `x` is the negative number (finite or infinite) of maximum magnitude in the type, `nextdown(x)` is `x`.

Returns

3 The `nextdown` functions return the next representable value in the specified type less than `x`.

7.12.11.7 The `canonicalize` functions

Synopsis

1
```
#include <math.h>
int canonicalize(double * cx, const double * x);
int canonicalizef(float * cx, const float * x);
int canonicalizel(long double * cx, const long double * x);
#ifdef __STDC_IEC_60559_DFP__
int canonicalized32(_Decimal32 * cx, const _Decimal32 * x);
int canonicalized64(_Decimal64 * cx, const _Decimal64 * x);
int canonicalized128(_Decimal128 * cx, const _Decimal128 * x);
#endif
```

Description

2 The `canonicalize` functions attempt to produce a canonical version of the floating-point representation in the object pointed to by the argument `x`, as if to a temporary object of the specified type, and store the canonical result in the object pointed to by the argument `cx`.[297] If the input `*x` is a signaling NaN, the `canonicalize` functions are intended to store a canonical quiet NaN. If a canonical result is not produced the object pointed to by `cx` is unchanged.

Returns

3 The `canonicalize` functions return zero if a canonical result is stored in the object pointed to by `cx`. Otherwise they return a nonzero value.

7.12.12 Maximum, minimum, and positive difference functions

7.12.12.1 The `fdim` functions

Synopsis

1
```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fdimd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fdimd64(_Decimal64 x, _Decimal64 y);
```

[297] Arguments `x` and `cx` may point to the same object.

```
_Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fdim** functions determine the *positive difference* between their arguments:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \leq y \end{cases}$$

A range error occurs for some finite arguments.

Returns

3 The **fdim** functions return the positive difference value.

7.12.12.2 The fmax functions

Synopsis

1
```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmaxd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaxd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fmax** functions determine the maximum numeric value of their arguments.[298)]

Returns

3 The **fmax** functions return the maximum numeric value of their arguments.

7.12.12.3 The fmin functions

Synopsis

1
```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmind32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmind64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmind128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fmin** functions determine the minimum numeric value of their arguments. [299)]

Returns

3 The **fmin** functions return the minimum numeric value of their arguments.

4 **NOTE 1** The **fmax** and **fmin** functions are similar to the **fmaximum_num** and **fminimum_num** functions, though may differ in which signed zero is returned when the arguments are differently signed zeros and in their treatment of signaling NaNs (see F.10.9.5).

[298)]Quiet NaN arguments are treated as missing data: if one argument is a quiet NaN and the other numeric, then the **fmax** functions choose the numeric value. See F.10.9.2.

[299)]The **fmin** functions are analogous to the **fmax** functions in their treatment of quiet NaNs.

7.12.12.4 The fmaximum functions

Synopsis

1
```
#include <math.h>
double fmaximum(double x, double y);
float fmaximumf(float x, float y);
long double fmaximuml(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmaximumd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximumd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximumd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The `fmaximum` functions determine the maximum value of their arguments. For these functions, $+0$ is considered greater than -0. These functions differ from the `fmaximum_num` functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The `fmaximum` functions return the maximum value of their arguments.

7.12.12.5 The fminimum functions

Synopsis

1
```
#include <math.h>
double fminimum(double x, double y);
float fminimumf(float x, float y);
long double fminimuml(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fminimumd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimumd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimumd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The `fminimum` functions determine the minimum value of their arguments. For these functions, -0 is considered less than $+0$. These functions differ from the `fminimum_num` functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The `fminimum` functions return the minimum value of their arguments.

7.12.12.6 The fmaximum_mag functions

Synopsis

1
```
#include <math.h>
double fmaximum_mag(double x, double y);
float fmaximum_magf(float x, float y);
long double fmaximum_magl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmaximum_magd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_magd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_magd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The `fmaximum_mag` functions determine the value of the argument of maximum magnitude: `x` if $|x| > |y|$, `y` if $|y| > |x|$, and `fmaximum(x, y)` otherwise. These functions differ from the `fmaximum_mag_num` functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fmaximum_mag** functions return the value of the argument of maximum magnitude.

7.12.12.7 The fminimum_mag functions

Synopsis

1
```
#include <math.h>
double fminimum_mag(double x, double y);
float fminimum_magf(float x, float y);
long double fminimum_magl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fminimum_magd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_magd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_magd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fminimum_mag** functions determine the value of the argument of minimum magnitude: `x` if $|$`x`$| < |$`y`$|$, `y` if $|$`y`$| < |$`x`$|$, and **fminimum(**`x`**,** `y`**)** otherwise. These functions differ from the **fminimum_mag_num** functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fminimum_mag** functions return the value of the argument of minimum magnitude.

7.12.12.8 The fmaximum_num functions

Synopsis

1
```
#include <math.h>
double fmaximum_num(double x, double y);
float fmaximum_numf(float x, float y);
long double fmaximum_numl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmaximum_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_numd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fmaximum_num** functions determine the maximum value of their numeric arguments. They determine the number if one argument is a number and the other is a NaN. These functions differ from the **fmaximum** functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fmaximum_num** functions return the maximum value of their numeric arguments.

7.12.12.9 The fminimum_num functions

Synopsis

1
```
#include <math.h>
double fminimum_num(double x, double y);
float fminimum_numf(float x, float y);
long double fminimum_numl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fminimum_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_numd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fminimum_num** functions determine the minimum value of their numeric arguments. They determine the number if one argument is a number and the other is a NaN. These functions differ from the **fminimum** functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fminimum_num** functions return the minimum value of their numeric arguments.

7.12.12.10 The fmaximum_mag_num functions

Synopsis

1
```
#include <math.h>
double fmaximum_mag_num(double x, double y);
float fmaximum_mag_numf(float x, float y);
long double fmaximum_mag_numl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmaximum_mag_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_mag_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_mag_numd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fmaximum_mag_num** functions determine the value of a numeric argument of maximum magnitude. They determine the number if one argument is a number and the other is a NaN. These functions differ from the **fmaximum_mag** functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fmaximum_mag_num** functions return the value of a numeric argument of maximum magnitude.

7.12.12.11 The fminimum_mag_num functions

Synopsis

1
```
#include <math.h>
double fminimum_mag_num(double x, double y);
float fminimum_mag_numf(float x, float y);
long double fminimum_mag_numl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fminimum_mag_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_mag_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_mag_numd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **fminimum_mag_num** functions determine the value of a numeric argument of minimum magnitude. They determine the number if one argument is a number and the other is a NaN. These functions differ from the **fminimum_mag** functions only in their treatment of NaN arguments (see F.10.9.4, F.10.9.5).

Returns

3 The **fminimum_mag_num** functions return the value of a numeric argument of minimum magnitude.

7.12.13 Fused multiply-add

7.12.13.1 The fma functions

Synopsis

1
```
#include <math.h>
double fma(double x, double y, double z);
```

```
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
_Decimal64 fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
#endif
```

Description

2 The **fma** functions compute $(x \times y) + z$, rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error occurs for some infinite arguments.

Returns

3 The **fma** functions return $(x \times y) + z$, rounded as one ternary operation.

7.12.14 Functions that round result to narrower type

1 The functions in this subclause round their results to a type typically narrower[300] than the parameter types.

7.12.14.1 Add and round to narrower type

Synopsis

1
```
#include <math.h>
float fadd(double x, double y);
float faddl(long double x, long double y);
double daddl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32addd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32addd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64addd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 These functions compute the sum of $x + y$, rounded to the type of the function. They compute the sum (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error may occur for infinite arguments.

Returns

3 These functions return the sum of $x + y$, rounded to the type of the function.

7.12.14.2 Subtract and round to narrower type

Synopsis

1
```
#include <math.h>
float fsub(double x, double y);
float fsubl(long double x, long double y);
double dsubl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32subd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32subd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64subd128(_Decimal128 x, _Decimal128 y);
#endif
```

[300] In some cases the destination type might not be narrower than the parameter types. For example, double might not be narrower than long double.

Description

2 These functions compute the difference of x − y, rounded to the type of the function. They compute the difference (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error may occur for infinite arguments.

Returns

3 These functions return the difference of x − y, rounded to the type of the function.

7.12.14.3 Multiply and round to narrower type

Synopsis

1
```
#include <math.h>
float fmul(double x, double y);
float fmull(long double x, long double y);
double dmull(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32muld64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32muld128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64muld128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 These functions compute the product x × y, rounded to the return type of the function. They compute the product (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error occurs for one infinite argument and one zero argument.

Returns

3 These functions return the product of x × y, rounded to the return type of the function.

7.12.14.4 Divide and round to narrower type

Synopsis

1
```
#include <math.h>
float fdiv(double x, double y);
float fdivl(long double x, long double y);
double ddivl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32divd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32divd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64divd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 These functions compute the quotient x ÷ y, rounded to the return type of the function. They compute the quotient (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error occurs for either both arguments infinite or both arguments zero. A pole error occurs for a finite x and a zero y.

Returns

3 These functions return the quotient x ÷ y, rounded to the type of the function.

7.12.14.5 Fused multiply-add and round to narrower type

Synopsis

1
```
#include <math.h>
float ffma(double x, double y, double z);
```

```
float ffmal(long double x, long double y, long double z);
double dfmal(long double x, long double y, long double z);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal32 d32fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
_Decimal64 d64fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
#endif
```

Description

2 These functions compute $(\mathtt{x} \times \mathtt{y}) + \mathtt{z}$, rounded to the return type of the function. They compute $(\mathtt{x} \times \mathtt{y}) + \mathtt{z}$ (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite arguments. A domain error may occur for an infinite argument.

Returns

3 These functions return $(\mathtt{x} \times \mathtt{y}) + \mathtt{z}$, rounded to the return type of the function.

7.12.14.6 Square root rounded to narrower type

Synopsis

1
```
#include <math.h>
float fsqrt(double x);
float fsqrtl(long double x);
double dsqrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32sqrtd64(_Decimal64 x);
_Decimal32 d32sqrtd128(_Decimal128 x);
_Decimal64 d64sqrtd128(_Decimal128 x);
#endif
```

Description

2 These functions compute the square root of x, rounded to the type of the function. They compute the square root (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error occurs for some finite positive arguments. A domain error occurs if the argument is less than zero.

Returns

3 These functions return the nonnegative square root of x, rounded to the return type of the function.

7.12.15 Quantum and quantum exponent functions

7.12.15.1 The quantizedN functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 quantized32(_Decimal32 x, _Decimal32 y);
_Decimal64 quantized64(_Decimal64 x, _Decimal64 y);
_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The quantized*N* functions compute, if possible, a value with the numerical value of x and the quantum exponent of y. If the quantum exponent is being increased, the value shall be correctly rounded; if the result does not have the same value as x, the "inexact" floating-point exception shall be raised. If the quantum exponent is being decreased and the significand of the result has more digits than the type would allow, the result is NaN, the "invalid" floating-point exception is raised, and a domain error occurs. If one or both operands are NaN the result is NaN. Otherwise if only one

operand is infinite, the result is NaN, the "invalid" floating-point exception is raised, and a domain error occurs. If both operands are infinite, the result is **DEC_INFINITY** with the sign of x, converted to the type of the function. The **quantized***N* functions do not raise the "overflow" and "underflow" floating-point exceptions.

Returns

3 The **quantized***N* functions return a value with the numerical value of x (except for any rounding) and the quantum exponent of y.

7.12.15.2 The samequantumd*N* functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
bool samequantumd32(_Decimal32 x, _Decimal32 y);
bool samequantumd64(_Decimal64 x, _Decimal64 y);
bool samequantumd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **samequantumd***N* functions determine if the quantum exponents of x and y are the same. If both x and y are NaN, or both infinite, they have the same quantum exponents; if exactly one operand is infinite or exactly one operand is NaN, they do not have the same quantum exponents. The **samequantumd***N* functions raise no floating-point exception.

Returns

3 The **samequantumd***N* functions return nonzero (**true**) when x and y have the same quantum exponents, zero (**false**) otherwise.

7.12.15.3 The quantumd*N* functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 quantumd32(_Decimal32 x);
_Decimal64 quantumd64(_Decimal64 x);
_Decimal128 quantumd128(_Decimal128 x);
#endif
```

Description

2 The **quantumd***N* functions compute the quantum (5.2.4.2.3) of a finite argument. If x is infinite, the result is $+\infty$.

Returns

3 The **quantumd***N* functions return the quantum of x.

7.12.15.4 The llquantexpd*N* functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
long long int llquantexpd32(_Decimal32 x);
long long int llquantexpd64(_Decimal64 x);
long long int llquantexpd128(_Decimal128 x);
#endif
```

Description

2 The **llquantexpd**N functions compute the quantum exponent (5.2.4.2.3) of a finite argument. If `x` is infinite or NaN, they compute **LLONG_MIN**, the "invalid" floating-point exception is raised, and a domain error occurs.

Returns

3 The **llquantexpd**N functions return the quantum exponent of `x`.

7.12.16 Decimal re-encoding functions

1 IEC 60559 specifies two different schemes to encode significands in the object representation of a decimal floating-point object: one based on decimal encoding (which packs three decimal digits into 10 bits), the other based on binary encoding (as a binary integer). An implementation may use either of these encoding schemes for its decimal floating types. The re-encoding functions in this subclause provide conversions between external decimal data with a given encoding scheme and the implementation's corresponding decimal floating type.

7.12.16.1 The encodedecd$N$ functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
void encodedecd32(unsigned char encptr[restrict static 4],
      const _Decimal32*restrict xptr);
void encodedecd64(unsigned char encptr[restrict static 8],
      const _Decimal64*restrict xptr);
void encodedecd128(unsigned char encptr[restrict static 16],
      const _Decimal128*restrict xptr);
#endif
```

Description

2 The **encodedecd**N functions convert `*xptr` into an IEC 60559 decimal$N$ encoding in the encoding scheme based on decimal encoding of the significand and store the resulting encoding as an $N/8$ element array, with 8 bits per array element, in the object pointed to by `encptr`. The order of bytes in the array is implementation-defined. These functions preserve the value of `*xptr` and raise no floating-point exceptions. If `*xptr` is non-canonical, these functions may or may not produce a canonical encoding.

Returns

3 The **encodedecd**N functions return no value.

7.12.16.2 The decodedecd$N$ functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
void decodedecd32(_Decimal32 * restrict xptr,
      const unsigned char encptr[restrict static 4]);
void decodedecd64(_Decimal64 * restrict xptr,
      const unsigned char encptr[restrict static 8]);
void decodedecd128(_Decimal128 * restrict xptr,
      const unsigned char encptr[restrict static 16]);
#endif
```

Description

15

2 The **decodedecd**N functions interpret the $N/8$ element array pointed to by `encptr` as an IEC 60559 decimal$N$ encoding, with 8 bits per array element, in the encoding scheme based on decimal

encoding of the significand. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a value of the decimal floating type, and store the result in the object pointed to by `xptr`. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns

3 The **decodedecd***N* functions return no value.

7.12.16.3 The encodebind*N* functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
void encodebind32(unsigned char encptr[restrict static 4],
      const _Decimal32 * restrict xptr);
void encodebind64(unsigned char encptr[restrict static 8],
      const _Decimal64 * restrict xptr);
void encodebind128(unsigned char encptr[restrict static 16],
      const _Decimal128 * restrict xptr);
#endif
```

Description

2 The **encodebind***N* functions convert `*xptr` into an IEC 60559 decimal*N* encoding in the encoding scheme based on binary encoding of the significand and store the resulting encoding as an $N/8$ element array, with 8 bits per array element, in the object pointed to by `encptr`. The order of bytes in the array is implementation-defined. These functions preserve the value of `*xptr` and raise no floating-point exceptions. If `*xptr` is non-canonical, these functions may or may not produce a canonical encoding.

Returns

3 The **encodebind***N* functions return no value.

7.12.16.4 The decodebind*N* functions

Synopsis

1
```
#include <math.h>
#ifdef __STDC_IEC_60559_DFP__
void decodebind32(_Decimal32 * restrict xptr,
      const unsigned char encptr[restrict static 4]);
void decodebind64(_Decimal64 * restrict xptr,
      const unsigned char encptr[restrict static 8]);
void decodebind128(_Decimal128 * restrict xptr,
      const unsigned char encptr[restrict static 16]);
#endif
```

Description

2 The **decodebind***N* functions interpret the $N/8$ element array pointed to by `encptr` as an IEC 60559 decimal*N* encoding, with 8 bits per array element, in the encoding scheme based on binary encoding of the significand. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a value of decimal floating type, and store the result in the object pointed to by `xptr`. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns

3 The **decodebind***N* functions return no value.

7.12.17 Comparison macros

1 The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships — *less*, *greater*, and *equal* — is true. Relational operators may raise the "invalid" floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[301] Subclauses 7.12.17.1 through 7.12.17.6 provide macros that are quiet versions of the relational operators: the macros do not raise the "invalid" floating-point exception as an effect of quiet NaN arguments. The comparison macros facilitate writing efficient code that accounts for quiet NaNs without suffering the "invalid" floating-point exception. In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type[302] (both arguments need not have the same type).[303] If either argument has decimal floating type, the other argument shall have decimal floating type as well.

7.12.17.1 The `isgreater` macro

Synopsis

1
```
#include <math.h>
int isgreater(real-floating x, real-floating y);
```

Description

2 The `isgreater` macro determines whether its first argument is greater than its second argument. The value of `isgreater(x,y)` is always equal to `(x)> (y)`; however, unlike `(x)> (y)`, `isgreater(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns

3 The `isgreater` macro returns the value of `(x)> (y)`.

7.12.17.2 The `isgreaterequal` macro

Synopsis

1
```
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

Description

2 The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x,y)` is always equal to `(x)>= (y)`; however, unlike `(x)>= (y)`, `isgreaterequal(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns

3 The `isgreaterequal` macro returns the value of `(x)>= (y)`.

7.12.17.3 The `isless` macro

Synopsis

1
```
#include <math.h>
int isless(real-floating x, real-floating y);
```

Description

2 The `isless` macro determines whether its first argument is less than its second argument. The value of `isless(x,y)` is always equal to `(x)< (y)`; however, unlike `(x)< (y)`, `isless(x,y)` does not

[301] IEC 60559 requires that the built-in relational operators raise the "invalid" floating-point exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

[302] If any argument is of integer type, or any other type that is not a real floating type, the behavior is undefined.

[303] Whether an argument represented in a format wider than its semantic type is converted to the semantic type is unspecified.

raise the "invalid" floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns

3 The **`isless`** macro returns the value of (`x`) < (`y`).

7.12.17.4 The `islessequal` macro

Synopsis

1
```
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

Description

2 The **`islessequal`** macro determines whether its first argument is less than or equal to its second argument. The value of `islessequal(x,y)` is always equal to `(x)<= (y)`; however, unlike `(x)<= (y)`, `islessequal(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns

3 The **`islessequal`** macro returns the value of `(x)<= (y)`.

7.12.17.5 The `islessgreater` macro

Synopsis

1
```
#include <math.h>
int islessgreater(real-floating x, real-floating y);
```

Description

2 The **`islessgreater`** macro determines whether its first argument is less than or greater than its second argument. The `islessgreater(x,y)` macro is similar to `(x)< (y)|| (x)> (y)`; however, `islessgreater(x,y)` does not raise the "invalid" floating-point exception when `x` and `y` are unordered and neither is a signaling NaN (nor does it evaluate `x` and `y` twice).

Returns

3 The **`islessgreater`** macro returns the value of `(x)< (y)|| (x)> (y)`.

7.12.17.6 The `isunordered` macro

Synopsis

1
```
#include <math.h>
int isunordered(real-floating x, real-floating y);
```

Description

2 The **`isunordered`** macro determines whether its arguments are unordered. It raises no floating-point exceptions if neither argument is a signaling NaN.

Returns

3 The **`isunordered`** macro returns 1 if its arguments are unordered and 0 otherwise.

7.12.17.7 The `iseqsig` macro

Synopsis

1
```
#include <math.h>
int iseqsig(real-floating x, real-floating y);
```

Description

2 The **`iseqsig`** macro determines whether its arguments are equal. If an argument is a NaN, a domain error occurs for the macro, as if a domain error occurred for a function (7.12.1).

Returns

3 The **iseqsig** macro returns 1 if its arguments are equal and 0 otherwise.

7.13 Non-local jumps `<setjmp.h>`

1 The header `<setjmp.h>` defines the macros **`setjmp`** and `__STDC_VERSION_SETJMP_H__`, and declares one function and one type, for bypassing the normal function call and return discipline.[304]

2 The macro

```
__STDC_VERSION_SETJMP_H__
```

is an integer constant expression with a value equivalent to *`202311`*`L`.

3 The type declared is

```
jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to the **`setjmp`** macro consists of information sufficient for a call to the **`longjmp`** function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

4 It is unspecified whether **`setjmp`** is a macro or an identifier declared with external linkage. If a macro definition is suppressed to access an actual function, or a program defines an external identifier with the name **`setjmp`**, the behavior is undefined.

7.13.1 Save calling environment

7.13.1.1 The `setjmp` macro

Synopsis

1
```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

2 The **`setjmp`** macro saves its calling environment in its **`jmp_buf`** argument for later use by the **`longjmp`** function.

Returns

3 If the return is from a direct invocation, the **`setjmp`** macro returns the value zero. If the return is from a call to the **`longjmp`** function, the **`setjmp`** macro returns a nonzero value.

Environmental limits

4 An invocation of the **`setjmp`** macro shall appear only in one of the following contexts:

— the entire controlling expression of a selection or iteration statement;

— one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;

— the operand of a unary `!` operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or

— the entire expression of an expression statement (possibly cast to **`void`**).

5 If the invocation appears in any other context, the behavior is undefined.

7.13.2 Restore calling environment

7.13.2.1 The `longjmp` function

[304]These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

Synopsis

1
```
#include <setjmp.h>
[[noreturn]] void longjmp(jmp_buf env, int val);
```

Description

2 The `longjmp` function restores the environment saved by the most recent invocation of the `setjmp` macro in the same invocation of the program with the corresponding `jmp_buf` argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the `setjmp` macro has terminated execution[305] in the interim, or if the invocation of the `setjmp` macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine[306] have state, as of the time the `longjmp` function was called, except that the representation of objects of automatic storage duration that are local to the function containing the invocation of the corresponding `setjmp` macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call is indeterminate.

Returns

4 After `longjmp` is completed, thread execution continues as if the corresponding invocation of the `setjmp` macro had just returned the value specified by `val`. The `longjmp` function cannot cause the `setjmp` macro to return the value 0; if `val` is 0, the `setjmp` macro returns the value 1.

5 **EXAMPLE** The `longjmp` function that returns control back to the point of the `setjmp` invocation might cause memory associated with a variable length array object to be squandered.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
      int x[n];           // valid:  f is not terminated
      setjmp(buf);
      g(n);
}

void g(int n)
{
      int a[n];           // a may remain allocated
      h(n);
}

void h(int n)
{
      int b[n];           // b may remain allocated
      longjmp(buf, 2);    // might cause memory loss
}
```

[305]For example, by executing a `return` statement or because another `longjmp` call has caused a transfer to a `setjmp` invocation in a function earlier in the set of nested calls.

[306]This includes, but is not limited to, the floating-point status flags and the state of open files.

7.14 Signal handling `<signal.h>`

1 The header `<signal.h>` declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

2 The type defined is

```
sig_atomic_t
```

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

3 The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the `signal` function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type `int` and distinct values that are the signal numbers, each corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the `abort` function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

4 An implementation need not generate any of these signals, except as a result of explicit calls to the `raise` function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters `SIG` and an uppercase letter or with `SIG_` and an uppercase letter,[307] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

7.14.1 Specify signal handling

7.14.1.1 The `signal` function

Synopsis

1
```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

2 The `signal` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise, `func` shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library),[308] is called a *signal handler*.

[307] See "future library directions" (7.33.9). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

[308] This includes functions called indirectly via standard library functions (e.g., a `SIGABRT` handler called via the `abort` function).

3 When a signal occurs and `func` points to a function, it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of `SIGILL`, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

4 If the signal occurs as the result of calling the `abort` or `raise` function, the signal handler shall not call the `raise` function.

5 If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or the signal handler calls any function in the standard library other than

— the `abort` function,

— the `_Exit` function,

— the `quick_exit` function,

— the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,

— the `atomic_is_lock_free` function with any atomic argument, or

— the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the `signal` function results in a `SIG_ERR` return, the object designated by `errno` has an indeterminate representation.[309)]

6 At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

7 Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the `signal` function.

Returns

8 If the request can be honored, the `signal` function returns the value of `func` for the most recent successful call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

Forward references: the `abort` function (7.24.4.1), the `exit` function (7.24.4.4), the `_Exit` function (7.24.4.5), the `quick_exit` function (7.24.4.7).

7.14.2 Send signal

7.14.2.1 The `raise` function

Synopsis

1
```
#include <signal.h>
int raise(int sig);
```

309)If any signal is generated by an asynchronous signal handler, the behavior is undefined.

Description

2 The `raise` function carries out the actions described in 7.14.1.1 for the signal `sig`. If a signal handler is called, the `raise` function shall not return until after the signal handler does.

Returns

3 The `raise` function returns zero if successful, nonzero if unsuccessful.

7.15 Alignment `<stdalign.h>`

1 The header `<stdalign.h>` provides no content.

7.16 Variable arguments `<stdarg.h>`

1 The header `<stdarg.h>` declares a type and defines five macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

2 The macro

```
__STDC_VERSION_STDARG_H__
```

is an integer constant expression with a value equivalent to `202311L`.

3 A function may be called with a variable number of arguments of varying types if its parameter type list ends with an ellipsis.

4 The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros **`va_start`**, **`va_arg`**, **`va_end`**, and **`va_copy`**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as `ap` in this subclause) having type **`va_list`**. The object `ap` may be passed as an argument to another function; if that function invokes the **`va_arg`** macro with parameter `ap`, the representation of `ap` in the calling function is indeterminate and shall be passed to the **`va_end`** macro prior to any further reference to `ap`[310].

7.16.1 Variable argument list access macros

1 The **`va_start`** and **`va_arg`** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **`va_copy`** and **`va_end`** are macros or identifiers declared with external linkage. If a macro definition is suppressed to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the **`va_start`** and **`va_copy`** macros shall be matched by a corresponding invocation of the **`va_end`** macro in the same function.

7.16.1.1 The `va_arg` macro

Synopsis

1
```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

2 The **`va_arg`** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter `ap` shall have been initialized by the **`va_start`** or **`va_copy`** macro (without an intervening invocation of the **`va_end`** macro for the same `ap`). Each invocation of the **`va_arg`** macro modifies `ap` so that the values of successive arguments are returned in turn. The behavior is undefined if there is no actual next argument. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a $*$ to *type*. If *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

— both types are pointers to qualified or unqualified versions of compatible types;

— one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;

— one type is pointer to qualified or unqualified **`void`** and the other is a pointer to a qualified or unqualified character type;

[310]It is permitted to create a pointer to a **`va_list`** and pass that pointer to another function, in which case the original function can make further use of the original list after the other function returns.

— or, the type of the next argument is **nullptr_t** and *type* is a pointer type that has the same representation and alignment requirements as a pointer to a character type.[311)]

Returns

3 The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the first argument without an explicit parameter, which matches the position of the `...` in the parameter list. Successive invocations return the values of the remaining arguments in succession.

7.16.1.2 The va_copy macro

Synopsis

1
```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

2 The **va_copy** macro initializes `dest` as a copy of `src`, as if the **va_start** macro had been applied to `dest` followed by the same sequence of uses of the **va_arg** macro as had previously been used to reach the present state of `src`. Neither the **va_copy** nor **va_start** macro shall be invoked to reinitialize `dest` without an intervening invocation of the **va_end** macro for the same `dest`.

Returns

3 The **va_copy** macro returns no value.

7.16.1.3 The va_end macro

Synopsis

1
```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

2 The **va_end** macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the **va_start** macro, or the function containing the expansion of the **va_copy** macro, that initialized the **va_list** `ap`. The **va_end** macro may modify `ap` so that it is no longer usable (without being reinitialized by the **va_start** or **va_copy** macro). If there is no corresponding invocation of the **va_start** or **va_copy** macro, or if the **va_end** macro is not invoked before the return, the behavior is undefined.

Returns

3 The **va_end** macro returns no value.

7.16.1.4 The va_start macro

Synopsis

1
```
#include <stdarg.h>
void va_start(va_list ap, ...);
```

Description

2 The **va_start** macro shall be invoked before any access to the unnamed arguments.

3 The **va_start** macro initializes `ap` for subsequent use by the **va_arg** and **va_end** macros. Neither the **va_start** nor **va_copy** macro shall be invoked to reinitialize `ap` without an intervening invocation of the **va_end** macro for the same `ap`.

4 Only the first argument passed to **va_start** is evaluated. Any additional arguments are not used by the macro and will not be expanded or evaluated for any reason.

5 **NOTE 1** The macro allows additional arguments to be passed for **va_start** for compatibility with older versions of the library only.

[311)] Such types are in particular pointers to qualified or unqualified versions of **void**.

Returns

6 The **va_start** macro returns no value.

7 **EXAMPLE 1** The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
#define MAXARGS   31

void f1(int n_ptrs, ...)
{
      va_list ap;
      char *array[MAXARGS];
      int ptr_no = 0;

      if (n_ptrs > MAXARGS)
            n_ptrs = MAXARGS;
      va_start(ap);
      while (ptr_no  <  n_ptrs)
            array[ptr_no++] = va_arg(ap, char *);
      va_end(ap);
      f2(n_ptrs, array);
}
```

Each call to f1 is required to have visible the definition of the function or a declaration such as

```
void f1(int, ...);
```

8 **EXAMPLE 2** The function f3 is similar, but saves the status of the variable argument list after the indicated number of arguments; after f2 has been called once with the whole list, the trailing part of the list is gathered again and passed to function f4.

```
#include <stdarg.h>
#define MAXARGS 31

void f3(int n_ptrs, int f4_after, ...)
{
      va_list ap, ap_save;
      char *array[MAXARGS];
      int ptr_no = 0;
      if (n_ptrs > MAXARGS)
            n_ptrs = MAXARGS;
      va_start(ap);
      while (ptr_no < n_ptrs) {
            array[ptr_no++] = va_arg(ap, char *);
            if (ptr_no == f4_after)
                  va_copy(ap_save, ap);
      }
      va_end(ap);
      f2(n_ptrs, array);

      // Now process the saved copy.

      n_ptrs -= f4_after;
      ptr_no = 0;
      while (ptr_no < n_ptrs)
            array[ptr_no++] = va_arg(ap_save, char *);
      va_end(ap_save);
      f4(n_ptrs, array);
}
```

9 **EXAMPLE 3** The function f5 is similar to f1, but instead of passing an explicit number of strings as the first argument, the argument list is terminated with a null pointer.

```
#include <stdarg.h>

#define MAXARGS 31

void f5(...)
{
        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;
        va_start(ap);
        while (ptr_no < MAXARGS)
        {
                char *ptr = va_arg(ap, char *);
                if (!ptr)
                        break;
                array[ptr_no++] = ptr;
        }
        va_end(ap);
        f6(ptr_no, array);
}
```

Each call to f5 is required to have visible the definition of the function or a declaration such as

```
void f5(...);
```

and implicitly requires the last argument to be a null pointer.

7.17 Atomics <stdatomic.h>

7.17.1 Introduction

1 The header `<stdatomic.h>` defines several macros and declares several types and functions for performing atomic operations on data shared between threads.[312)]

2 Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide this header nor support any of its facilities.

3 The macro

```
__STDC_VERSION_STDATOMIC_H__
```

is an integer constant expression with a value equivalent to `202311L`.

4 The macros defined are the *atomic lock-free macros*

```
ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR8_T_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
```

which expand to constant expressions suitable for use in conditional expression inclusion preprocessing directives and which indicate the lock-free property of the corresponding atomic types (both signed and unsigned); and

```
ATOMIC_FLAG_INIT
```

which expands to an initializer for an object of type `atomic_flag`.

5 The types include

```
memory_order
```

which is an enumerated type whose enumerators identify memory ordering constraints;

```
atomic_flag
```

which is a structure type representing a lock-free, primitive atomic flag; and several atomic analogs of integer types.

6 In the following synopses:

— An *A* refers to an atomic type.

— A *C* refers to its corresponding non-atomic type.

— An *M* refers to the type of the other argument for arithmetic operations. For atomic integer types, *M* is *C*. For atomic pointer types, *M* is `ptrdiff_t`.

— The functions not ending in `_explicit` have the same semantics as the corresponding `_explicit` function with `memory_order_seq_cst` for the `memory_order` argument.

312)See "future library directions" (7.33.10).

7 It is unspecified whether any generic function declared in <stdatomic.h> is a macro or an identifier declared with external linkage. If a macro definition is suppressed to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is undefined.

8 **NOTE 1** Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects.

7.17.2 Initialization

1 An atomic object with automatic storage duration that is not initialized or such an object with allocated storage duration initially has an indeterminate representation; equally, a non-atomic store to any byte of the representation (either directly or, for example, by calls to `memcpy` or `memset`) makes any atomic object have an indeterminate representation. Explicit or default initialization for atomic objects with static or thread storage duration that do not have the type `atomic_flag` is guaranteed to produce a valid state.[313].

2 Concurrent access to an atomic object before it is set to a valid state, even via an atomic operation, constitutes a data race. If a signal occurs other than as the result of calling the `abort` or `raise` functions, the behavior is undefined if the signal handler reads or modifies an atomic object that has an indeterminate representation.

3 **EXAMPLE** The following definition ensure valid states for `guide` and `head` regardless if these are found in file scope or block scope. Thus any atomic operation that is performed on them after their initialization has been met is well defined.

```
_Atomic int guide = 42;
static _Atomic(void*) head;
```

7.17.2.1 The `atomic_init` generic function

Synopsis

1
```
#include <stdatomic.h>
void atomic_init(volatile A *obj, C value);
```

Description

2 The `atomic_init` generic function initializes the atomic object pointed to by `obj` to the value `value`, while also initializing any additional state that the implementation might need to carry for the atomic object. If the object has no declared type, after the call the effective type is the atomic type *A*.

3 Although this function initializes an atomic object, it does not avoid data races; concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

4 If a signal occurs other than as the result of calling the `abort` or `raise` functions, the behavior is undefined if the signal handler calls the `atomic_init` generic function.

Returns

5 The `atomic_init` generic function returns no value.

6 **EXAMPLE**

```
atomic_int guide;
atomic_init(&guide, 42);
```

7.17.3 Order and consistency

1 The enumerated type `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations as defined in 5.1.2.4 and may provide for operation ordering. Its enumeration constants are as follows:[314]

[313]See "future library directions" (7.33.10).
[314]See "future library directions" (7.33.10).

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

2 For `memory_order_relaxed`, no operation orders memory.

3 For `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`, a store operation performs a release operation on the affected memory location.

4 For `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`, a load operation performs an acquire operation on the affected memory location.

5 For `memory_order_consume`, a load operation performs a consume operation on the affected memory location.

6 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

— the result of the last modification A of M that precedes B in S, if it exists, or

— if A exists, the result of some modification of M that is not `memory_order_seq_cst` and that does not happen before A, or

— if A does not exist, the result of some modification of M that is not `memory_order_seq_cst`.

7 **NOTE 1** Although it is not explicitly required that S include lock operations, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering.

8 **NOTE 2** Atomic operations specifying `memory_order_relaxed` are relaxed only with respect to memory ordering. Implementations still guarantee that any given atomic access to a particular atomic object is indivisible with respect to all other atomic accesses to that object.

9 For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

10 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

11 For atomic modifications A and B of an atomic object M, B occurs later than A in the modification order of M if:

— there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, or

— there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, or

— there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

12 Atomic read-modify-write operations shall always read the last value (in the modification order) stored before the write associated with the read-modify-write operation.

13 An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that

— If an evaluation B observes a value computed by A in a different thread, then B does not happen before A.

— If an evaluation A is included in the sequence, then all evaluations that assign to the same variable and happen before A are also included.

14 **NOTE 3** The second requirement disallows "out-of-thin-air", or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations can appear in this sequence out of thread order. For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, 42, memory_order_relaxed);
```

is allowed to produce `r1 == 42 && r2 == 42`. The sequence of evaluations justifying this consists of:

```
atomic_store_explicit(&y, 42, memory_order_relaxed);
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);
r2 = atomic_load_explicit(&x, memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = atomic_load_explicit(&y, memory_order_relaxed);
atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&x, memory_order_relaxed);
atomic_store_explicit(&y, r2, memory_order_relaxed);
```

is not allowed to produce `r1 == 42 && r2 == 42`, since there is no sequence of evaluations that results in the computation of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact.

Recommended practice

15 The requirements do not forbid `r1 == 42 && r2 == 42` in the following example, with `x` and `y` initially zero:

```
// Thread 1:
r1 = atomic_load_explicit(&x, memory_order_relaxed);
if (r1 == 42)
      atomic_store_explicit(&y, r1, memory_order_relaxed);

// Thread 2:
r2 = atomic_load_explicit(&y, memory_order_relaxed);
if (r2 == 42)
      atomic_store_explicit(&x, 42, memory_order_relaxed);
```

However, this is not useful behavior, and implementations should not allow it.

16 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

7.17.3.1 The `kill_dependency` macro

Synopsis

1
```
#include <stdatomic.h>
type kill_dependency(type y);
```

Description

2 The `kill_dependency` macro terminates a dependency chain; the argument does not carry a dependency to the return value.

Returns

3 The `kill_dependency` macro returns the value of `y`.

7.17.4 Fences

1 This subclause introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*; a fence with release semantics is called a *release fence*.

2 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

3 A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

4 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

7.17.4.1 The `atomic_thread_fence` function

Synopsis

1
```
#include <stdatomic.h>
void atomic_thread_fence(memory_order order);
```

Description

2 Depending on the value of `order`, this operation:

— has no effects, if `order == memory_order_relaxed`;

— is an acquire fence, if `order == memory_order_acquire` or `order == memory_order_consume` ;

— is a release fence, if `order == memory_order_release`;

— is both an acquire fence and a release fence, if `order == memory_order_acq_rel`;

— is a sequentially consistent acquire and release fence, if `order == memory_order_seq_cst`.

Returns

3 The `atomic_thread_fence` function returns no value.

7.17.4.2 The `atomic_signal_fence` function

Synopsis

1
```
#include <stdatomic.h>
void atomic_signal_fence(memory_order order);
```

Description

2 Equivalent to `atomic_thread_fence(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

3 NOTE 1 The **atomic_signal_fence** function can be used to specify the order in which actions performed by the thread become visible to the signal handler.

4 NOTE 2 Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with **atomic_thread_fence**, but the hardware fence instructions that **atomic_thread_fence** would have inserted are not emitted.

Returns

5 The **atomic_signal_fence** function returns no value.

7.17.5 Lock-free property

1 The atomic lock-free macros indicate the lock-free property of integer and address atomic types. A value of 0 indicates that the type is never lock-free; a value of 1 indicates that the type is sometimes lock-free; a value of 2 indicates that the type is always lock-free.

Recommended practice

2 Operations that are lock-free should also be *address-free*. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication via memory mapped into a process more than once and memory shared between two processes.

7.17.5.1 The atomic_is_lock_free generic function

Synopsis

1
```
#include <stdatomic.h>
bool atomic_is_lock_free(const volatile A *obj);
```

Description

2 The **atomic_is_lock_free** generic function indicates whether atomic operations on objects of the type pointed to by `obj` are lock-free.

Returns

3 The **atomic_is_lock_free** generic function returns nonzero (true) if and only if atomic operations on objects of the type pointed to by the argument are lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.[315)]

7.17.6 Atomic integer types

1 For each line in the following table,[316)] the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.[317)]

Atomic type name	Direct type
`atomic_bool`	`_Atomic bool`
`atomic_char`	`_Atomic char`
`atomic_schar`	`_Atomic signed char`
`atomic_uchar`	`_Atomic unsigned char`
`atomic_short`	`_Atomic short`
`atomic_ushort`	`_Atomic unsigned short`
`atomic_int`	`_Atomic int`
`atomic_uint`	`_Atomic unsigned int`
`atomic_long`	`_Atomic long`
`atomic_ulong`	`_Atomic unsigned long`
`atomic_llong`	`_Atomic long long`
`atomic_ullong`	`_Atomic unsigned long long`
`atomic_char8_t`	`_Atomic char8_t`

[315)] `obj` can be a null pointer.

[316)] See "future library directions" (7.33.10).

[317)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

Atomic type name	Direct type
atomic_char16_t	_Atomic char16_t
atomic_char32_t	_Atomic char32_t
atomic_wchar_t	_Atomic wchar_t
atomic_int_least8_t	_Atomic int_least8_t
atomic_uint_least8_t	_Atomic uint_least8_t
atomic_int_least16_t	_Atomic int_least16_t
atomic_uint_least16_t	_Atomic uint_least16_t
atomic_int_least32_t	_Atomic int_least32_t
atomic_uint_least32_t	_Atomic uint_least32_t
atomic_int_least64_t	_Atomic int_least64_t
atomic_uint_least64_t	_Atomic uint_least64_t
atomic_int_fast8_t	_Atomic int_fast8_t
atomic_uint_fast8_t	_Atomic uint_fast8_t
atomic_int_fast16_t	_Atomic int_fast16_t
atomic_uint_fast16_t	_Atomic uint_fast16_t
atomic_int_fast32_t	_Atomic int_fast32_t
atomic_uint_fast32_t	_Atomic uint_fast32_t
atomic_int_fast64_t	_Atomic int_fast64_t
atomic_uint_fast64_t	_Atomic uint_fast64_t
atomic_intptr_t	_Atomic intptr_t
atomic_uintptr_t	_Atomic uintptr_t
atomic_size_t	_Atomic size_t
atomic_ptrdiff_t	_Atomic ptrdiff_t
atomic_intmax_t	_Atomic intmax_t
atomic_uintmax_t	_Atomic uintmax_t

Recommended practice

2 The representation of an atomic integer type is not required to have the same size as the corresponding regular type but it should have the same size whenever possible, as it eases effort required to port existing code.

7.17.7 Operations on atomic types

1 There are only a few kinds of operations on atomic types, though there are many instances of those kinds. This subclause specifies each general kind.

7.17.7.1 The atomic_store generic functions

Synopsis

1
```
#include <stdatomic.h>
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_acquire`, `memory_order_consume`, nor `memory_order_acq_rel`. Atomically replace the value pointed to by `object` with the value of `desired`. Memory is affected according to the value of `order`.

Returns

3 The `atomic_store` generic functions return no value.

7.17.7.2 The atomic_load generic functions

Synopsis

1
```
#include <stdatomic.h>
C atomic_load(const volatile A *object);
C atomic_load_explicit(const volatile A *object, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. Memory is affected according to the value of `order`.

Returns

3 Atomically returns the value pointed to by `object`.

7.17.7.3 The `atomic_exchange` generic functions

Synopsis

1
```
#include <stdatomic.h>
C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
```

Description

2 Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (5.1.2.4).

Returns

3 Atomically returns the value pointed to by `object` immediately before the effects.

7.17.7.4 The `atomic_compare_exchange` generic functions

Synopsis

1
```
#include <stdatomic.h>
bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected,
      C desired, memory_order success, memory_order failure);
bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected,
      C desired, memory_order success, memory_order failure);
```

Description

2 The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the success argument.

3 Atomically, compares the contents of the memory pointed to by `object` for equality with that pointed to by `expected`, and if true, replaces the contents of the memory pointed to by `object` with `desired`, and if false, updates the contents of the memory pointed to by `expected` with that pointed to by `object`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. These operations are atomic read-modify-write operations (5.1.2.4).

4 **NOTE 1** For example, the effect of `atomic_compare_exchange_strong` is

```
if (memcmp(object, expected, sizeof(*object)) == 0)
      memcpy(object, &desired, sizeof(*object));
else
      memcpy(expected, object, sizeof(*object));
```

5 A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.

6 **NOTE 2** This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.

7 **EXAMPLE** A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```
exp = atomic_load(&cur);
do {
      des = function(exp);
} while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

Returns

8 The result of the comparison.

7.17.7.5 The `atomic_fetch` and modify generic functions

1 The following operations perform arithmetic and bitwise computations. All these operations are applicable to an object of any atomic integer type. None of these operations is applicable to **`atomic_bool`**. The key, operator, and computation correspondence is:

key	op	computation
add	+	addition
sub	−	subtraction
or	|	bitwise inclusive or
xor	^	bitwise exclusive or
and	&	bitwise and

Synopsis

2
```
#include <stdatomic.h>
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
```

Description

3 Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4). For signed integer types, arithmetic performs silent wraparound on integer overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

Returns

4 Atomically, the value pointed to by `object` immediately before the effects.

5 **NOTE 1** The operation of the `atomic_fetch` and modify generic functions are nearly equivalent to the operation of the corresponding *op*= compound assignment operators. The only differences are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated value of the object, whereas the value returned by the `atomic_fetch` and modify generic functions is the previous value of the atomic object.

7.17.8 Atomic flag type and operations

1 The **`atomic_flag`** type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type **`atomic_flag`** shall be lock free.

3 **NOTE 1** Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the **`atomic_flag`** type is the minimum hardware-implemented type needed to conform to this document. The remaining types can be emulated with **`atomic_flag`**, though with less than ideal properties.

4 The macro **`ATOMIC_FLAG_INIT`** may be used to initialize an **`atomic_flag`** to the clear state. An

`atomic_flag` that is not explicitly initialized with `ATOMIC_FLAG_INIT` has initially an indeterminate representation.

5 **EXAMPLE**

```
    atomic_flag guard = ATOMIC_FLAG_INIT;
```

7.17.8.1 The `atomic_flag_test_and_set` functions

Synopsis

1
```
#include <stdatomic.h>
    bool atomic_flag_test_and_set(volatile atomic_flag *object);
    bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,
        memory_order order);
```

Description

2 Atomically places the atomic flag pointed to by `object` in the set state and returns the value corresponding to the immediately preceding state. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 The `atomic_flag_test_and_set` functions return the value that corresponds to the state of the atomic flag immediately before the effects. The return value true corresponds to the set state and the return value false corresponds to the clear state.

7.17.8.2 The `atomic_flag_clear` functions

Synopsis

1
```
    #include <stdatomic.h>
    void atomic_flag_clear(volatile atomic_flag *object);
    void atomic_flag_clear_explicit(volatile atomic_flag *object,
        memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`. Atomically places the atomic flag pointed to by `object` into the clear state. Memory is affected according to the value of `order`.

Returns

3 The `atomic_flag_clear` functions return no value.

7.18 Bit and byte utilities <stdbit.h>

7.18.1 General

1 The header <stdbit.h> defines the following macros, types, and functions, to work with the byte and bit representation of many types, typically integer types. This header makes available the **size_t** type name (7.21) and any uintN_t, intN_t, uint_leastN_t, or int_leastN_t type names defined by the implementation (7.22).

2 The macro

```
__STDC_VERSION_STDBIT_H__
```

is an integer constant expression with a value equivalent to 202311L.

3 The *most significant index* is the 0-based index counting from the most significant bit, 0, to the least significant bit, $w - 1$, where w is the width of the type that is having its most significant index computed.

4 The *least significant index* is the 0-based index counting from the least significant bit, 0, to the most significant bit, $w - 1$, where w is the width of the type that is having its least significant index computed.

5 It is unspecified whether any generic function declared in <stdbit.h> is a macro or an identifier declared with external linkage. If a macro definition is suppressed to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is unspecified.

7.18.2 Endian

1 Two common methods of byte ordering in multi-byte scalar types are *little-endian* and *big-endian*. Little-endian is a format for storage or transmission of binary data in which the least significant byte is placed first, with the rest in ascending order. Or, that the least significant byte is stored at the smallest memory address. Big-endian is a format for storage or transmission of binary data in which the most significant byte is placed first, with the rest in descending order. Or, that the most significant byte is stored at the smallest memory address. Other byte orderings are also possible.

2 The macros are:

```
__STDC_ENDIAN_LITTLE__
```

which represents a method of byte order storage in which the least significant byte is placed first and the rest are in ascending order, and is an integer constant expression;

```
__STDC_ENDIAN_BIG__
```

which represents a method of byte order storage in which the most significant byte is placed first and the rest are in descending order, and is an integer constant expression;

```
__STDC_ENDIAN_NATIVE__ /* see below */
```

which represents the method of byte order storage for the execution environment and is an integer constant expression. **__STDC_ENDIAN_NATIVE__** describes the endianness of the execution environment with respect to bit-precise integer types, standard integer types, and extended integer types which do not have padding bits.

3 **__STDC_ENDIAN_NATIVE__** shall expand to an integer constant expression whose value is equivalent to the value of **__STDC_ENDIAN_LITTLE__** if the execution environment is little-endian. Otherwise, **__STDC_ENDIAN_NATIVE__** shall expand to an integer constant expression whose value is equivalent to the value of **__STDC_ENDIAN_BIG__** if the execution environment is big-endian. If the execution environment is neither little-endian nor big-endian, it then has some other implementation-defined byte order and the macro **__STDC_ENDIAN_NATIVE__** shall expand to an integer constant expression whose value is different from the values of

__STDC_ENDIAN_LITTLE__ and __STDC_ENDIAN_BIG__. The values of the integer constant expressions for __STDC_ENDIAN_LITTLE__ and __STDC_ENDIAN_BIG__ are not equal.

7.18.3 Count Leading Zeros

Synopsis

1
```
#include <stdbit.h>
int stdc_leading_zerosuc(unsigned char value);
int stdc_leading_zerosus(unsigned short value);
int stdc_leading_zerosui(unsigned int value);
int stdc_leading_zerosul(unsigned long value);
int stdc_leading_zerosull(unsigned long long value);
generic_return_type stdc_leading_zeros(generic_value_type value);
```

Returns

Returns the number of consecutive 0 bits in `value`, starting from the most significant bit.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.4 Count Leading Ones

Synopsis

1
```
#include <stdbit.h>
int stdc_leading_onesuc(unsigned char value);
int stdc_leading_onesus(unsigned short value);
int stdc_leading_onesui(unsigned int value);
int stdc_leading_onesul(unsigned long value);
int stdc_leading_onesull(unsigned long long value);
generic_return_type stdc_leading_ones(generic_value_type value);
```

Returns

Returns the number of consecutive 1 bits in `value`, starting from the most significant bit.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.5 Count Trailing Zeros

Synopsis

1
```
#include <stdbit.h>
int stdc_trailing_zerosuc(unsigned char value);
int stdc_trailing_zerosus(unsigned short value);
int stdc_trailing_zerosui(unsigned int value);
int stdc_trailing_zerosul(unsigned long value);
int stdc_trailing_zerosull(unsigned long long value);
generic_return_type stdc_trailing_zeros(generic_value_type value);
```

Returns

Returns the number of consecutive 0 bits in `value`, starting from the least significant bit.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.6 Count Trailing Ones

Synopsis

1
```
#include <stdbit.h>
int stdc_trailing_onesuc(unsigned char value);
int stdc_trailing_onesus(unsigned short value);
int stdc_trailing_onesui(unsigned int value);
int stdc_trailing_onesul(unsigned long value);
int stdc_trailing_onesull(unsigned long long value);
generic_return_type stdc_trailing_ones(generic_value_type value);
```

Returns

Returns the number of consecutive 1 bits in `value`, starting from the least significant bit.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.7 First Leading Zero

Synopsis

1
```
#include <stdbit.h>
int stdc_first_leading_zerouc(unsigned char value);
int stdc_first_leading_zerous(unsigned short value);
int stdc_first_leading_zeroui(unsigned int value);
```

```
int stdc_first_leading_zeroul(unsigned long value);
int stdc_first_leading_zeroull(unsigned long long value);
generic_return_type stdc_first_leading_zero(generic_value_type value);
```

Returns

Returns the most significant index of the first 0 bit in `value`, plus 1. If it is not found, this function returns 0.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.8 First Leading One

Synopsis

1

```
#include <stdbit.h>
int stdc_first_leading_oneuc(unsigned char value);
int stdc_first_leading_oneus(unsigned short value);
int stdc_first_leading_oneui(unsigned int value);
int stdc_first_leading_oneul(unsigned long value);
int stdc_first_leading_oneull(unsigned long long value);
generic_return_type stdc_first_leading_one(generic_value_type value);
```

Returns

Returns the most significant index of the first 1 bit in `value`, plus 1. If it is not found, this function returns 0.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.9 First Trailing Zero

Synopsis

1

```
#include <stdbit.h>
int stdc_first_trailing_zerouc(unsigned char value);
int stdc_first_trailing_zerous(unsigned short value);
int stdc_first_trailing_zeroui(unsigned int value);
int stdc_first_trailing_zeroul(unsigned long value);
int stdc_first_trailing_zeroull(unsigned long long value);
generic_return_type stdc_first_trailing_zero(generic_value_type value);
```

Returns

Returns the least significant index of the first 0 bit in `value`, plus 1. If it is not found, this function returns 0.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.10 First Trailing One

Synopsis

1
```
#include <stdbit.h>
int stdc_first_trailing_oneuc(unsigned char value);
int stdc_first_trailing_oneus(unsigned short value);
int stdc_first_trailing_oneui(unsigned int value);
int stdc_first_trailing_oneul(unsigned long value);
int stdc_first_trailing_oneull(unsigned long long value);
generic_return_type stdc_first_trailing_one(generic_value_type value);
```

Returns

Returns the least significant index of the first 1 bit in `value`, plus 1. If it is not found, this function returns 0.

The type-generic function (marked by its **`generic_value_type`** argument) returns the appropriate value based on the type of the input value, so long as it is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.11 Count Zeros

Synopsis

1
```
#include <stdbit.h>
int stdc_count_zerosuc(unsigned char value);
int stdc_count_zerosus(unsigned short value);
int stdc_count_zerosui(unsigned int value);
int stdc_count_zerosul(unsigned long value);
int stdc_count_zerosull(unsigned long long value);
generic_return_type stdc_count_zeros(generic_value_type value);
```

Returns

Returns the total number of 0 bits within the given `value`.

The type-generic function (marked by its **`generic_value_type`** argument) returns the previously described result for a given input value so long as the **`generic_value_type`** is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type for the type-generic function need not be the same as the type of `value`. It shall be a suitably large signed integer type capable of representing the computed result.

7.18.12 Count Ones

Synopsis

1
```
#include <stdbit.h>
int stdc_count_onesuc(unsigned char value);
int stdc_count_onesus(unsigned short value);
int stdc_count_onesui(unsigned int value);
int stdc_count_onesul(unsigned long value);
int stdc_count_onesull(unsigned long long value);
generic_return_type stdc_count_ones(generic_value_type value);
```

Returns

Returns the total number of 1 bits within the given `value`.

The type-generic function (marked by its **`generic_value_type`** argument) returns the previously described result for a given input value so long as the **`generic_value_type`** is a:

— standard unsigned integer type, excluding **`bool`**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **`bool`**.

The **`generic_return_type`** type shall be a suitably large signed integer type capable of representing the computed result.

7.18.13 Single-bit Check

Synopsis

1
```
#include <stdbit.h>
bool stdc_has_single_bituc(unsigned char value);
bool stdc_has_single_bitus(unsigned short value);
bool stdc_has_single_bitui(unsigned int value);
bool stdc_has_single_bitul(unsigned long value);
bool stdc_has_single_bitull(unsigned long long value);
bool stdc_has_single_bit(generic_value_type value);
```

Returns

The **`stdc_has_single_bit`** functions return **`true`** if and only if there is a single 1 bit in `value`.

The type-generic function (marked by its **`generic_value_type`** argument) returns the previously described result for a given input value so long as the **`generic_value_type`** is a:

— standard unsigned integer type, excluding **bool**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **bool**.

7.18.14 Bit Width

Synopsis

1
```
#include <stdbit.h>
int stdc_bit_widthuc(unsigned char value);
int stdc_bit_widthus(unsigned short value);
int stdc_bit_widthui(unsigned int value);
int stdc_bit_widthul(unsigned long value);
int stdc_bit_widthull(unsigned long long value);
generic_return_type stdc_bit_width(generic_value_type value);
```

Description

The `stdc_bit_width` functions compute the smallest number of bits needed to store `value`.

Returns

The `stdc_bit_width` functions return 0 if `value` is 0. Otherwise, they return $1 + \lfloor \log_2(\texttt{value}) \rfloor$.

The type-generic function (marked by its `generic_value_type` argument) returns the previously described result for a given input value so long as the `generic_value_type` is a:

— standard unsigned integer type, excluding **bool**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **bool**.

The `generic_return_type` type for the type-generic function need not be the same as the type of `value`. It shall be a suitably large signed integer type capable of representing the computed result.

7.18.15 Bit Floor

Synopsis

1
```
#include <stdbit.h>
unsigned char stdc_bit_flooruc(unsigned char value);
unsigned short stdc_bit_floorus(unsigned short value);
unsigned int stdc_bit_floorui(unsigned int value);
unsigned long stdc_bit_floorul(unsigned long value);
unsigned long long stdc_bit_floorull(unsigned long long value);
generic_value_type stdc_bit_floor(generic_value_type value);
```

Description

The `stdc_bit_floor` functions compute the largest integral power of 2 that is not greater than `value`.

Returns

The `stdc_bit_floor` functions return 0 if `value` is 0. Otherwise, they return the largest integral power of 2 that is not greater than `value`.

The type-generic function (marked by its `generic_value_type` argument) returns the previously described result for a given input value so long as the `generic_value_type` is a:

— standard unsigned integer type, excluding **bool**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **bool**.

7.18.16 Bit Ceiling

Synopsis

1
```
#include <stdbit.h>
unsigned char stdc_bit_ceiluc(unsigned char value);
unsigned short stdc_bit_ceilus(unsigned short value);
unsigned int stdc_bit_ceilui(unsigned int value);
unsigned long stdc_bit_ceilul(unsigned long value);
unsigned long long stdc_bit_ceilull(unsigned long long value);
generic_value_type stdc_bit_ceil(generic_value_type value);
```

Description

The **stdc_bit_ceil** functions compute the smallest integral power of 2 that is not less than `value`. If the computation does not fit in the given return type, the behavior is undefined.

Returns

The **stdc_bit_ceil** functions return the smallest integral power of 2 that is not less than `value`.

The type-generic function (marked by its **generic_value_type** argument) returns the previously described result for a given input value so long as the **generic_value_type** is a:

— standard unsigned integer type, excluding **bool**;

— extended unsigned integer type;

— or, bit-precise unsigned integer type whose width matches a standard or extended integer type, excluding **bool**.

7.19 Boolean type and values <stdbool.h>

1 The header <stdbool.h> provides the obsolescent macro **__bool_true_false_are_defined** which expands to the integer constant 1.

7.20 Checked Integer Arithmetic `<stdckdint.h>`

1 The header `<stdckdint.h>` defines several macros for performing checked integer arithmetic.

2 The macro

```
__STDC_VERSION_STDCKDINT_H__
```

is an integer constant expression with a value equivalent to `202311L`.

7.20.1 The `ckd_` Checked Integer Operation Macros

Synopsis

1
```
#include <stdckdint.h>
bool ckd_add(type1 *result, type2 a, type3 b);
bool ckd_sub(type1 *result, type2 a, type3 b);
bool ckd_mul(type1 *result, type2 a, type3 b);
```

Description

2 These macros perform addition, subtraction, or multiplication of the mathematical values of `a` and `b`, storing the result of the operation in `*result`, (that is, `*result` is assigned the result of computing `a + b`, `a - b`, or `a * b`). Each operation is performed as if both operands were represented in a signed integer type with infinite range, and the result was then converted from this integer type to `type1`.

3 Both `type2` and `type3` shall be any integer type other than plain **`char`**, **`bool`**, a bit-precise integer type, or an enumeration type, and they need not be the same. `*result` shall be a modifiable lvalue of any integer type other than plain **`char`**, **`bool`**, a bit-precise integer type, or an enumeration type.

Recommended practice

4 It is recommended to produce a diagnostic message if `type2` or `type3` are not suitable integer types, or if `*result` is not a modifiable lvalue of a suitable integer type.

Returns

5 If these macros return **`false`**, the value assigned to `*result` correctly represents the mathematical result of the operation. Otherwise, these macros return **`true`**. In this case, the value assigned to `*result` is the mathematical result of the operation wrapped around to the width of `*result`.

6 **EXAMPLE 1** If `a` and `b` are values of type **`signed int`**, and `result` is a **`signed long`**, then

```
        ckd_sub(&result, a, b);
```

will indicate if `a - b` can be expressed as a **`signed long`**. If **`signed long`** has a greater width than **`signed int`**, this will always be possible and this macro will return **`false`**.

7.21 Common definitions <stddef.h>

1 The header <stddef.h> defines the following macros and declares the following types. Some are also defined in other headers, as noted in their respective subclauses.

2 The macro

```
__STDC_VERSION_STDDEF_H__
```

is an integer constant expression with a value equivalent to 202311L.

3 The types are

```
ptrdiff_t
```

which is the signed integer type of the result of subtracting two pointers;

```
size_t
```

which is the unsigned integer type of the result of the **sizeof** operator;

```
max_align_t
```

which is an object type whose alignment is the greatest fundamental alignment;

```
wchar_t
```

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. Each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant if an implementation does not define **__STDC_MB_MIGHT_NEQ_WC__**; and,

```
nullptr_t
```

which is the type of the **nullptr** predefined constant, see below.

4 The macros are

```
NULL
```

which expands to an implementation-defined null pointer constant;

```
unreachable()
```

which expands to a void expression that invokes undefined behavior if it is reached during execution; and

```
offsetof(type, member-designator)
```

which expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the subobject (designated by *member-designator*), from the beginning of any object of type *type*. The type and member designator shall be such that given

```
static type t;
```

then the expression &(t. *member-designator*) evaluates to an address constant. If the specified *type* defines a new type or if the specified member is a bit-field, the behavior is undefined.

Recommended practice

5 The types used for **size_t** and **ptrdiff_t** should not have an integer conversion rank greater than that of **signed long int** unless the implementation supports objects large enough to make this necessary.

7.21.1 The unreachable macro

Synopsis

1
```
#include <stddef.h>
void unreachable(void);
```

Description

2 A call to the function-like macro **unreachable** indicates that the particular flow control that leads to the call will never be taken; it receives no arguments and expands to a void expression. The program execution shall not reach such a call.

Returns

3 If a macro call **unreachable()** is reached during execution, the behavior is undefined.

4 **EXAMPLE 1** The following program assumes that each execution is provided with at least one command line argument. The behavior of an execution with no arguments is undefined.

```
#include <stddef.h>
#include <stdio.h>

int main (int argc, char* argv[static argc + 1]) {
      if (argc <= 2)
            unreachable();
      else
            return printf("%s: we see %s", argv[0], argv[1]);

      return puts("this should never be reached");
}
```

Here, the **static** array size expression and the annotation of the control flow with **unreachable** indicates that the pointed-to parameter array **argv** will hold at least three elements, regardless of the circumstances. A possible optimization is that the resulting executable never performs the comparison and unconditionally executes a tail call to **printf** that never returns to the **main** function. In particular, the entire call and reference to **puts** can be omitted from the executable. No diagnostic is expected.

7.21.2 The nullptr_t type

Synopsis

1
```
#include <stddef.h>
typedef typeof_unqual(nullptr) nullptr_t;
```

Description

2 The **nullptr_t** type is the type of the **nullptr** predefined constant. It has only a very limited use in contexts where this type is needed to distinguish **nullptr** from other expression types. It is an unqualified complete scalar type that is different from all pointer or arithmetic types and is neither an atomic or array type and has exactly one value, **nullptr**. Default initialization of an object of this type is equivalent to an initialization by **nullptr**.

3 The size and alignment of **nullptr_t** is the same as for a pointer to character type. An object representation of the value **nullptr** is the same as the object representation of a null pointer value of type **void***. An lvalue conversion of an object of type **nullptr_t** with such an object representation

has the value **nullptr**; if the object representation is different, the behavior is undefined[318].

4 **NOTE 1** Because it is considered to be a scalar type, **nullptr_t** may appear in many context where (**void***)0 would be valid, for example,

— as the operand of **alignas**, **sizeof** or typeof operators,

— as the operand of an implicit or explicit conversion to a pointer type,

— as the assignment expression in an assignment or initialization of an object of type **nullptr_t**,

— as an argument to a parameter of type **nullptr_t** or in a variable argument list,

— as a void expression,

— as the operand of an implicit or explicit conversion to **bool**,

— as an operand of a **_Generic** primary expression,

— as an operand of the !, &&, || or conditional operators, or

— as the controlling expression of an **if** or iteration statement.

[318]Thus, during the whole program execution an object of type **nullptr_t** evaluates to the assumed value **nullptr**.

7.22 Integer types <stdint.h>

1 The header <stdint.h> declares sets of integer types having specified widths, and defines corresponding sets of macros.[319] It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

2 Types are defined in the following categories:

- integer types having certain exact widths;
- integer types having at least certain specified widths;
- fastest integer types having at least certain specified widths;
- integer types wide enough to hold pointers to objects;
- integer types having greatest width.

(Some of these types may denote the same type.)

3 Corresponding macros specify limits of the declared types and construct suitable constants.

4 For each type described herein that the implementation provides,[320] <stdint.h> shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, <stdint.h> shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as "required", but need not provide any of the others (described as "optional"). None of the types shall be defined as a synonym for a bit-precise integer type.

5 The feature test macro __STDC_VERSION_STDINT_H__ expands to the token *202311*L.

7.22.1 Integer types

1 When typedef names differing only in the absence or presence of the initial u are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

2 In the following descriptions, the symbol *N* represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

7.22.1.1 Exact-width integer types

1 The typedef name **int*N*_t** designates a signed integer type with width *N* and no padding bits. Thus, **int8_t** denotes such a signed integer type with a width of exactly 8 bits.

2 The typedef name **uint*N*_t** designates an unsigned integer type with width *N* and no padding bits. Thus, uint24_t denotes such an unsigned integer type with a width of exactly 24 bits.

3 If an implementation provides standard or extended integer types with a particular width and no padding bits, it shall define the corresponding typedef names.

7.22.1.2 Minimum-width integer types

1 The typedef name **int_least*N*_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width. Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits.

2 The typedef name **uint_least*N*_t** designates an unsigned integer type with a width of at least *N*, such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.

3 If the typedef name **int*N*_t** is defined, **int_least*N*_t** designates the same type. If the typedef name **uint*N*_t** is defined, **uint_least*N*_t** designates the same type.

4 The following types are required:

[319] See "future library directions" (7.33.14).

[320] Some of these types might denote implementation-defined extended integer types.

```
int_least8_t          uint_least8_t
int_least16_t         uint_least16_t
int_least32_t         uint_least32_t
int_least64_t         uint_least64_t
```

All other types of this form are optional.

7.22.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[321)] to operate with among all integer types that have at least the specified width.

2 The typedef name **int_fast***N***_t** designates the fastest signed integer type with a width of at least *N*. The typedef name **uint_fast***N***_t** designates the fastest unsigned integer type with a width of at least *N*.

3 The following types are required:

```
int_fast8_t           uint_fast8_t
int_fast16_t          uint_fast16_t
int_fast32_t          uint_fast32_t
int_fast64_t          uint_fast64_t
```

All other types of this form are optional.

7.22.1.4 Integer types capable of holding object pointers

1 The following type designates a signed integer type, other than a bit-precise integer type, with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type, other than a bit-precise integer type, with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

7.22.1.5 Greatest-width integer types

1 The following type designates a signed integer type, other than a bit-precise integer type, capable of representing any value of any signed integer type with the possible exceptions of signed bit-precise integer types and of signed extended integer types that are wider than **long long** and that are referred by the type definition for an exact width integer type:

```
intmax_t
```

The following type designates the unsigned integer type that corresponds to **intmax_t**[322)]:

```
uintmax_t
```

These types are required.

[321)]The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

[322)]Thus this type is capable of representing any value of any unsigned integer type with the possible exception of particular extended integer types that are wider than **unsigned long long**.

7.22.2 Widths of specified-width integer types

1 The following object-like macros specify the width of the types declared in `<stdint.h>`. Each macro name corresponds to a similar type name in 7.22.1.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **`#if`** preprocessing directives. Its implementation-defined value shall be equal to or greater than the value given below, except where stated to be exactly the given value. An implementation shall define only the macros corresponding to those typedef names it actually provides.[323)]

7.22.2.1 Width of exact-width integer types

1

`INTN_WIDTH`	exactly N
`UINTN_WIDTH`	exactly N

7.22.2.2 Width of minimum-width integer types

1

`INT_LEASTN_WIDTH`	exactly `UINT_LEASTN_WIDTH`
`UINT_LEASTN_WIDTH`	N

7.22.2.3 Width of fastest minimum-width integer types

1

`INT_FASTN_WIDTH`	exactly `UINT_FASTN_WIDTH`
`UINT_FASTN_WIDTH`	N

7.22.2.4 Width of integer types capable of holding object pointers

1

`INTPTR_WIDTH`	exactly `UINTPTR_WIDTH`
`UINTPTR_WIDTH`	16

7.22.2.5 Width of greatest-width integer types

1

`INTMAX_WIDTH`	exactly `UINTMAX_WIDTH`
`UINTMAX_WIDTH`	64

7.22.3 Width of other integer types

1 The following object-like macros specify the width of integer types corresponding to types defined in other standard headers.

2 Each instance of these macros shall be replaced by a constant expression suitable for use in **`#if`** preprocessing directives. Its implementation-defined value shall be equal to or greater than the corresponding value given below. An implementation shall define only the macros corresponding to those typedef names it actually provides.[324)]

7.22.3.1 Width of `ptrdiff_t`

1

`PTRDIFF_WIDTH`	16

7.22.3.2 Width of `sig_atomic_t`

1

`SIG_ATOMIC_WIDTH`	8

[323)]The exact-width and pointer-holding integer types are optional.
[324)]A freestanding implementation need not provide all these types.

7.22.3.3 Width of size_t

1
```
SIZE_WIDTH                          16
```

7.22.3.4 Width of wchar_t

1
```
WCHAR_WIDTH                          8
```

7.22.3.5 Width of wint_t

1
```
WINT_WIDTH                          16
```

7.22.4 Macros for integer constants

1 The following function-like macros expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in `<stdint.h>`. Each macro name corresponds to a similar type name in 7.22.1.2 or 7.22.1.5.

2 The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

3 Each invocation of one of these macros shall expand to an integer constant expression. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument. If the value is in the range of the type **intmax_t** (for a signed type) or the type **uintmax_t** (for an unsigned type), see 7.22.1.5, the expression is suitable for use in conditional expression inclusion preprocessing directives.

7.22.4.1 Macros for minimum-width integer constants

1 The macro **INT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **int_least***N***_t**. The macro **UINT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **uint_least***N***_t**. For example, if **uint_least64_t** is a name for the type **unsigned long long int**, then **UINT64_C**(0x123) might expand to the integer constant 0x123ULL.

7.22.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type **intmax_t**:

```
INTMAX_C(value)
```

The following macro expands to an integer constant expression having the value specified by its argument and the type **uintmax_t**:

```
UINTMAX_C(value)
```

7.22.5 Maximal and minimal values of integer types

1 For all integer types for which there is a macro with suffix **_WIDTH** holding the width, maximum macros with suffix **_MAX** and, for all signed types, minimum macros with suffix **_MIN** are defined as by 5.2.4.2. If it is unspecified if a type is signed or unsigned and the implementation has it as an unsigned type, a minimum macro with extension **_MIN**, and value 0 of the corresponding type is defined.

7.23 Input/output <stdio.h>

7.23.1 Introduction

1 The header <stdio.h> defines several macros, and declares three types and many functions for performing input and output.

2 The macro

```
__STDC_VERSION_STDIO_H__
```

is an integer constant expression with a value equivalent to 202311L.

3 The types declared are **size_t** (described in 7.21);

```
FILE
```

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

```
fpos_t
```

which is a complete object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

4 The macros are **NULL** (described in 7.21);

```
_IOFBF
_IOLBF
_IONBF
```

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

```
BUFSIZ
```

which expands to an integer constant expression that is the size of the buffer used by the **setbuf** function;

```
EOF
```

which expands to an integer constant expression, with type **int** and a negative value, that is returned by several functions to indicate *end-of-file*, that is, no more input from a stream;

```
FOPEN_MAX
```

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

```
FILENAME_MAX
```

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened or, if the implementation imposes no practical limit on the length of file name strings, the recommended size of an array intended to hold a file name string[325] ;

[325]Of course, file name string contents are subject to other system-specific constraints; therefore *all* possible strings of length **FILENAME_MAX** cannot be expected to be opened successfully.

```
_PRINTF_NAN_LEN_MAX
```

which expands to an integer constant expression (suitable for use in conditional expression inclusion preprocessing directives) that is the maximum number of characters output for any

[-]NAN(*n-char-sequence*)

sequence.[326] If an implementation has no support for NaNs, it shall be 0. **_PRINTF_NAN_LEN_MAX** shall be less than 64;

```
L_tmpnam
```

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam** function;

```
SEEK_CUR
SEEK_END
SEEK_SET
```

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

```
TMP_MAX
```

which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the **tmpnam** function;

```
stderr
stdin
stdout
```

which are expressions of type "pointer to **FILE**" that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

5 The header <wchar.h> declares functions for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of generalized multibyte characters, as described further in 7.23.3.

6 The input/output functions are given the following collective terms:

— The *wide character input functions* — those functions described in 7.31 that perform input into wide characters and wide strings: **fgetwc**, **fgetws**, **getwc**, **getwchar**, **fwscanf**, **wscanf**, **vfwscanf**, and **vwscanf**.

— The *wide character output functions* — those functions described in 7.31 that perform output from wide characters and wide strings: **fputwc**, **fputws**, **putwc**, **putwchar**, **fwprintf**, **wprintf**, **vfwprintf**, and **vwprintf**.

— The *wide character input/output functions* — the union of the **ungetwc** function, the wide character input functions, and the wide character output functions.

— The *byte input/output functions* — those functions described in this subclause that perform input/output: **fgetc**, **fgets**, **fprintf**, **fputc**, **fputs**, **fread**, **fscanf**, **fwrite**, **getc**, **getchar**, **printf**, **putc**, **putchar**, **puts**, **scanf**, **ungetc**, **vfprintf**, **vfscanf**, **vprintf**, and **vscanf**.

Forward references: files (7.23.3), the **fseek** function (7.23.9.2), streams (7.23.2), the **tmpnam** function (7.23.4.4), <wchar.h> (7.31).

[326]If the implementation only uses the [-]NAN style, then **_PRINTF_NAN_LEN_MAX** would have the value 4.

7.23.2 Streams

1 Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.[327)]

2 A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

4 Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is unoriented. Once a wide character input/output function has been applied to an unoriented stream, the stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to an unoriented stream, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)[328)]

5 Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written may henceforth not consist of valid multibyte characters.

6 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

7 Each stream has an associated lock that is used to prevent data races when multiple threads of execution access a stream, and to restrict the interleaving of stream operations performed by multiple threads. Only one thread may hold this lock at a time. The lock is reentrant: a single thread may hold the lock multiple times at a given time.

8 All functions that read, write, position, or query the position of a stream lock the stream before accessing it. They release the lock associated with the stream when the access is complete.

---

327) An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

328) The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

Environmental limits

9 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Forward references: the **freopen** function (7.23.5.4), the **fwide** function (7.31.3.5), **mbstate_t** (7.31.1), the **fgetpos** function (7.23.9.1), the **fsetpos** function (7.23.9.3).

7.23.3 Files

1 A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.23.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **setbuf** and **setvbuf** functions.

4 A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The lifetime of a **FILE** object ends when the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

5 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

6 The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not serve in place of the original.

7 At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

9 Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

— Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

— A file need not begin nor end in the initial shift state.[329)]

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's own **mbstate_t** object. The byte input functions read characters from the stream as if by successive calls to the **fgetc** function.

12 The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the **fputwc** function. Each conversion occurs as if by a call to the **wcrtomb** function, with the conversion state described by the stream's own **mbstate_t** object. The byte output functions write characters to the stream as if by successive calls to the **fputc** function.

13 In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the **mbrtowc** and **wcrtomb** functions.

14 An *encoding error* occurs if the character sequence presented to the underlying **mbrtowc** function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying **wcrtomb** does not correspond to a valid (generalized) multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro **EILSEQ** in **errno** if and only if an encoding error occurs.

Environmental limits

15 The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Forward references: the **exit** function (7.24.4.4), the **fgetc** function (7.23.7.1), the **fopen** function (7.23.5.3), the **fputc** function (7.23.7.3), the **setbuf** function (7.23.5.5), the **setvbuf** function (7.23.5.6), the **fgetwc** function (7.31.3.1), the **fputwc** function (7.31.3.3), conversion state (7.31.6), the **mbrtowc** function (7.31.6.3.2), the **wcrtomb** function (7.31.6.3.3).

7.23.4 Operations on files

7.23.4.1 The remove function

Synopsis

1
```
#include <stdio.h>
int remove(const char *filename);
```

Description

2 The **remove** function causes the file whose name is the string pointed to by `filename` to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is implementation-defined.

Returns

3 The **remove** function returns zero if the operation succeeds, nonzero if it fails.

7.23.4.2 The rename function

Synopsis

1
```
#include <stdio.h>
int rename(const char *old, const char *new);
```

[329)]Setting the file position indicator to end-of-file, as with `fseek(file, 0, SEEK_END)`, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

Description

2 The **rename** function causes the file whose name is the string pointed to by `old` to be henceforth known by the name given by the string pointed to by `new`. The file named `old` is no longer accessible by that name. If a file named by the string pointed to by `new` exists prior to the call to the **rename** function, the behavior is implementation-defined.

Returns

3 The **rename** function returns zero if the operation succeeds, nonzero if it fails,[330] in which case if the file existed previously it is still known by its original name.

7.23.4.3 The `tmpfile` function

Synopsis

1
```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

2 The **tmpfile** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with `"wb+"` mode.

Recommended practice

3 It should be possible to open at least **TMP_MAX** temporary files during the lifetime of the program (this limit may be shared with **tmpnam**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

4 The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Forward references: the **fopen** function (7.23.5.3).

7.23.4.4 The `tmpnam` function

Synopsis

1
```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

2 The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.[331] The function is potentially capable of generating at least **TMP_MAX** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

3 The **tmpnam** function generates a different string each time it is called.

4 Calls to the **tmpnam** function with a null pointer argument may introduce data races with each other. The implementation shall behave as if no library function calls the **tmpnam** function.

Returns

5 If no suitable string can be generated, the **tmpnam** function returns a null pointer. Otherwise, if the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the **tmpnam** function may modify the same object).

---

[330] Among the reasons the implementation could cause the **rename** function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

[331] Files created using strings generated by the **tmpnam** function are temporary only in the sense that their names are not expected to collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination.

If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` `char`s; the `tmpnam` function writes its result in that array and returns the argument as its value.

Environmental limits

6 The value of the macro `TMP_MAX` shall be at least 25.

7.23.5 File access functions

7.23.5.1 The `fclose` function

Synopsis

1
```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

2 A successful call to the `fclose` function causes the stream pointed to by `stream` to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether the call succeeds or not, the stream is disassociated from the file and any buffer set by the `setbuf` or `setvbuf` function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

3 The `fclose` function returns zero if the stream was successfully closed, or `EOF` if any errors were detected.

7.23.5.2 The `fflush` function

Synopsis

1
```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

2 If `stream` points to an output stream or an update stream in which the most recent operation was not input, the `fflush` function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

3 If `stream` is a null pointer, the `fflush` function performs this flushing action on all streams for which the behavior is defined above.

Returns

4 The `fflush` function sets the error indicator for the stream and returns `EOF` if a write error occurs, otherwise it returns zero.

Forward references: the `fopen` function (7.23.5.3).

7.23.5.3 The `fopen` function

Synopsis

1
```
#include <stdio.h>
FILE *fopen(const char * restrict filename, const char * restrict mode);
```

Description

2 The `fopen` function opens the file whose name is the string pointed to by `filename`, and associates a stream with it.

3 The argument `mode` points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.[332]

[332] If the string begins with one of the listed mode sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in 7.23.2).

`r`	open text file for reading
`w`	truncate to zero length or create text file for writing
`wx`	create text file for writing
`a`	append; open or create text file for writing at end-of-file
`rb`	open binary file for reading
`wb`	truncate to zero length or create binary file for writing
`wbx`	create binary file for writing
`ab`	append; open or create binary file for writing at end-of-file
`r+`	open text file for update (reading and writing)
`w+`	truncate to zero length or create text file for update
`w+x`	create text file for update
`a+`	append; open or create text file for update, writing at end-of-file
`r+b` *or* `rb+`	open binary file for update (reading and writing)
`w+b` *or* `wb+`	truncate to zero length or create binary file for update
`w+bx` *or* `wb+x`	create binary file for update
`a+b` *or* `ab+`	append; open or create binary file for update, writing at end-of-file

4 Opening a file with read mode (`'r'` as the first character in the `mode` argument) fails if the file does not exist or cannot be read.

5 Opening a file with exclusive mode (`'x'` as the last character in the `mode` argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.

6 Opening a file with append mode (`'a'` as the first character in the `mode` argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the **`fseek`** function. In some implementations, opening a binary file with append mode (`'b'` as the second or third character in the above list of `mode` argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

7 When a file is opened with update mode (`'+'` as the second or third character in the above list of `mode` argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the **`fflush`** function or to a file positioning function (**`fseek`**, **`fsetpos`**, or **`rewind`**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

8 When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

9 The **`fopen`** function returns a pointer to the object controlling the stream. If the open operation fails, **`fopen`** returns a null pointer.

Forward references: file positioning functions (7.23.9).

7.23.5.4 The `freopen` function

Synopsis

1
```
#include <stdio.h>
FILE *freopen(const char * restrict filename, const char * restrict mode,
      FILE * restrict stream);
```

Description

2 The **freopen** function opens the file whose name is the string pointed to by `filename` and associates the stream pointed to by `stream` with it. The `mode` argument is used just as in the **fopen** function.[333)]

3 If `filename` is a null pointer, the **freopen** function attempts to change the mode of the stream to that specified by `mode`, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

4 The **freopen** function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Returns

5 The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of `stream`.

7.23.5.5 The setbuf function

Synopsis

1
```
#include <stdio.h>
void setbuf(FILE * restrict stream, char * restrict buf);
```

Description

2 Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for `mode` and **BUFSIZ** for `size`, or (if `buf` is a null pointer), with the value **_IONBF** for `mode`.

Returns

3 The **setbuf** function returns no value.

Forward references: the **setvbuf** function (7.23.5.6).

7.23.5.6 The setvbuf function

Synopsis

1
```
#include <stdio.h>
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

Description

2 The **setvbuf** function may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument `mode` determines how `stream` will be buffered, as follows:

_IOFBF causes input/output to be fully buffered;

_IOLBF causes input/output to be line buffered;

_IONBF causes input/output to be unbuffered.

If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function[334)] and the argument `size` specifies the size of the array; otherwise, `size` may determine the size of a buffer allocated by the **setvbuf** function. The members of the array at any time have unspecified values.

[333)]The primary use of the **freopen** function is to change the file associated with a standard text stream (**stderr**, **stdin**, or **stdout**), as those identifiers need not be modifiable lvalues to which the value returned by the **fopen** function could be assigned.

[334)]The buffer has to have a lifetime at least as great as the open stream, so not closing the stream before a buffer that has automatic storage duration is deallocated upon block exit results in undefined behavior.

Returns

3 The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for `mode` or if the request cannot be honored.

7.23.6 Formatted input/output functions

1 The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[335)]

7.23.6.1 The fprintf function

Synopsis

1
```
#include <stdio.h>
int fprintf(FILE * restrict stream, const char * restrict format, ...);
```

Description

2 The **fprintf** function writes output to the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a nonnegative decimal integer. [336)]

— An optional *precision* that gives the minimum number of digits to appear for the `b`, `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of bytes to be written for `s` conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag characters and their meanings are:

[335)]The **fprintf** functions perform writes to memory for the `%n` specifier.

[336)]Note that `0` is taken as a flag, not as the beginning of a field width.

- `-` The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

- `+` The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a value with a negative sign is converted if this flag is not specified.) [337)]

- *space* If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and `+` flags both appear, the *space* flag is ignored.

- `#` The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `b` conversion, a nonzero result has `0b` prefixed to it. For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

- `0` For `b`, `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

- `hh` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`signed char`** or **`unsigned char`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`signed char`** or **`unsigned char`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`signed char`** argument.

- `h` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`short int`** or **`unsigned short int`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`short int`** or **`unsigned short int`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`short int`** argument.

- `l` (ell) Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long int`** or **`unsigned long int`** argument; that a following `n` conversion specifier applies to a pointer to a **`long int`** argument; that a following `c` conversion specifier applies to a **`wint_t`** argument; that a following `s` conversion specifier applies to a pointer to a **`wchar_t`** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

- `ll` (ell-ell) Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long long int`** or **`unsigned long long int`** argument; or that a following `n` conversion specifier applies to a pointer to a **`long long int`** argument.

- `j` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **`intmax_t`** or **`uintmax_t`** argument; or that a following `n` conversion specifier applies to a pointer to an **`intmax_t`** argument.

- `z` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`size_t`** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **`size_t`** argument.

[337)] The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

`t` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`ptrdiff_t`** or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a **`ptrdiff_t`** argument.

`w`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an integer argument with a specific width where *N* is a positive decimal integer with no leading zeros (the argument will have been promoted according to the integer promotions, but its value shall be converted to the unpromoted type); or that a following `n` conversion specifier applies to a pointer to an integer type argument with a width of *N* bits. All minimum-width integer types (7.22.1.2) and exact-width integer types (7.22.1.1) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`wf`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a fastest minimum-width integer argument with a specific width where *N* is a positive decimal integer with no leading zeros (the argument will have been promoted according to the integer promotions, but its value shall be converted to the unpromoted type); or that a following `n` conversion specifier applies to a pointer to a fastest minimum-width integer type argument with a width of *N* bits. All fastest minimum-width integer types (7.22.1.3) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`long double`** argument.

`H` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal32`** argument.

`D` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal64`** argument.

`DD` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal128`** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

`d,i` The **`int`** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`b,o,u,x,X` The **`unsigned int`** argument is converted to unsigned binary (b), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f,F` A **`double`** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **`#`** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **`double`** argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A **`double`** argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-char-sequence*`)` — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively.[338)]

`e,E` A **`double`** argument representing a floating-point number is converted in the style *[-]d.ddd*`e`$\pm$*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g,G` A **`double`** argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of X:

if $P > X \geq -4$, the conversion is with style `f` (or `F`) and precision $P - (X + 1)$.

otherwise, the conversion is with style `e` (or `E`) and precision $P - 1$.

Finally, unless the `#` flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a,A` A **`double`** argument representing a floating-point number is converted in the style *[-]*`0x`*h.hhhh*`p`$\pm$*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[339)] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **`FLT_RADIX`** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **`FLT_RADIX`** is not a power of 2, then the precision is sufficient to distinguish[340)] values of type **`double`**, except that trailing zeros may be omitted; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

338) When applied to infinite and NaN values, the `-`, `+`, and *space* flag characters have their usual meaning; the `#` and `0` flag characters have no effect.

339) Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries. This implementation choice affects numerical values printed with a precision P that is insufficient to represent all values exactly. Implementations with different conventions about the most significant hexadecimal digit will round at different places, affecting the numerical value of the hexadecimal result. For example, possible printed output for the code

```
#include <stdio.h>
/* ... */
double x = 123.0;
printf("%.1a", x);
```

include "`0x1.fp+6`" and "`0xf.6p+3`" whose numerical values are 124 and 123, respectively. Portable code seeking identical numerical results on different platforms should avoid precisions P that require rounding.

340) The formatting precision P is sufficient to distinguish values of the source type if $16^P > b^p$ where b (not a power of 2) and p are the base and precision of the source type (5.2.4.2.2). A smaller P might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

If an `H`, `D`, or `DD` modifier is present and the precision is missing, then for a decimal floating type argument represented by a triple of integers (s, c, q), where n is the number of significant digits in the coefficient c,

— if $-(n+5) \leq q \leq 0$, use style `f` (or style `F` in the case of an `A` conversion specifier) with formatting precision equal to $-q$,

— otherwise, use style `e` (or style `E` in the case of an `A` conversion specifier) with formatting precision equal to $n-1$, with the exceptions that if $c = 0$ then the digit-sequence in the exponent-part shall have the value q (rather than 0), and that the exponent is always expressed with the minimum number of digits required to represent its value (the exponent never contains a leading zero).

If the precision P is present (in the conversion specification) and is zero or at least as large as the precision p (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision P is present (and nonzero) and less than the precision p of the decimal floating type, the conversion first obtains an intermediate result as follows, where n is the number of significant digits in the coefficient:

— If $n \leq P$, set the intermediate result to the input.

— If $n > P$, round the input value, according to the current rounding direction for decimal floating-point operations, to P decimal digits, with unbounded exponent range, representing the result with a P-digit integer coefficient when in the form (s, c, q).

Convert the intermediate result in the manner described above for the case where the precision is missing.

`c` If no `l` length modifier is present, the **`int`** argument is converted to an **`unsigned char`**, and the resulting character is written.

If an `l` length modifier is present, the **`wint_t`** argument is converted as if by an `ls` conversion specification with no precision and an argument that points to storage suitably sized for at least two **`wchar_t`** elements, the first element containing the **`wint_t`** argument to the `lc` conversion specification and the second a null wide character.

`s` If no `l` length modifier is present, the argument shall be a pointer to storage of character type.[341] Characters from the storage are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the storage, the storage shall contain a null character.

If an `l` length modifier is present, the argument shall be a pointer to storage of **`wchar_t`** type. Wide characters from the storage are converted to multibyte characters (each as if by a call to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the storage shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the storage shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[342]

`p` The argument shall be a pointer to **`void`** or a pointer to a character type. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

[341] No special provisions are made for multibyte characters.

[342] Redundant shift sequences can result if multibyte characters have a state-dependent encoding.

`n` The argument shall be a pointer to signed integer whose type is specified by the length modifiers, if any, for the conversion specification, or shall be **`int`** if no length modifiers are specified for the conversion specification. The number of characters written to the output stream so far by this call to **`fprintf`** is stored into the integer object pointed to by the argument. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%` A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

9 If a conversion specification is invalid, the behavior is undefined.[343] **`fprintf`** shall behave as if it uses **`va_arg`** with a type argument naming the type resulting from applying the default argument promotions to the type corresponding to the conversion specification and then converting the result of the **`va_arg`** expansion to the type corresponding to the conversion specification.[344]

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For `a` and `A` conversions, if **`FLT_RADIX`** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For `a` and `A` conversions, if **`FLT_RADIX`** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most the maximum value M of the *T*__`_DECIMAL_DIG`__ macros (defined in `<float.h>`), then the result should be correctly rounded.[345] If the number of significant decimal digits is more than M but the source value is exactly representable with M digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having M significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

14 An uppercase `B` format specifier is not covered by the description above, because it used to be available for extensions in previous versions of this standard.

Implementations that did not use an uppercase `B` as their own extension before are encouraged to implement it similar to conversion specifier `b` as standardized above, with the alternative form (`#B`) generating `0B` as prefix for nonzero values.

Returns

15 The **`fprintf`** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred or if the implementation does not support a specified width length modifier.

Environmental limits

16 The number of characters that can be produced by any single conversion shall be at least 4095.

17 **EXAMPLE 1** To print a date and time in the form "Sunday, July 3, 10:02" followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month;    // pointers to strings
```

[343]See "future library directions" (7.33.15).
[344]The behavior is undefined when the types differ as specified for **`va_arg`** 7.16.1.1.
[345]For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

```
    int day, hour, min;
    fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
          weekday, month, day, hour, min);
    fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

18 **EXAMPLE 2** In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter.

19 Given the following wide string with length seven,

```
    static wchar_t wstr[] = L"□X□Yabc□Z□W";
```

the seven calls

```
    fprintf(stdout, "|1234567890123|\n");
    fprintf(stdout, "|%13ls|\n", wstr);
    fprintf(stdout, "|%-13.9ls|\n", wstr);
    fprintf(stdout, "|%13.10ls|\n", wstr);
    fprintf(stdout, "|%13.11ls|\n", wstr);
    fprintf(stdout, "|%13.15ls|\n", &wstr[2]);
    fprintf(stdout, "|%13lc|\n", (wint_t) wstr[5]);
```

will print the following seven lines:

```
    |1234567890123|
    |  □X□Yabc□Z□W|
    |□X□Yabc□Z    |
    |    □X□Yabc□Z|
    |  □X□Yabc□Z□W|
    |      abc□Z□W|
    |           □Z|
```

20 **EXAMPLE 3** Following are representations of `_Decimal64` arguments as triples (s, c, q) and the corresponding character sequences `fprintf` produces with `"%Da"`:

$(+1, 123, 0)$	`123`
$(-1, 123, 0)$	`-123`
$(+1, 123, -2)$	`1.23`
$(+1, 123, 1)$	`1.23e+3`
$(-1, 123, 1)$	`-1.23e+3`
$(+1, 123, -8)$	`0.00000123`
$(+1, 123, -9)$	`1.23e-7`
$(+1, 120, -8)$	`0.00000120`
$(+1, 120, -9)$	`1.20e-7`
$(+1, 1234567890123456, 0)$	`1234567890123456`
$(+1, 1234567890123456, 1)$	`1.234567890123456e+16`
$(+1, 1234567890123456, -1)$	`123456789012345.6`
$(+1, 1234567890123456, -21)$	`0.000001234567890123456`
$(+1, 1234567890123456, -22)$	`1.234567890123456e-7`
$(+1, 0, 0)$	`0`
$(-1, 0, 0)$	`-0`
$(+1, 0, -6)$	`0.000000`
$(+1, 0, -7)$	`0e-7`
$(+1, 0, 2)$	`0e+2`
$(+1, 5, -6)$	`0.000005`
$(+1, 50, -7)$	`0.0000050`
$(+1, 5, -7)$	`5e-7`

To illustrate the effects of a precision specification, the sequence:

```
_Decimal32 x = 6543.00DF;       // (+1, 654300, -2)
fprintf(stdout, "%Ha\n", x);
fprintf(stdout, "%.6Ha\n", x);
fprintf(stdout, "%.5Ha\n", x);
fprintf(stdout, "%.4Ha\n", x);
fprintf(stdout, "%.3Ha\n", x);
fprintf(stdout, "%.2Ha\n", x);
fprintf(stdout, "%.1Ha\n", x);
fprintf(stdout, "%.0Ha\n", x);
```

assuming default rounding, results in:

```
6543.00
6543.00
6543.0
6543
6.54e+3
6.5e+3
7e+3
6543.00
```

To illustrate the effects of the exponent range, the sequence:

```
_Decimal32 x = 9543210e87DF;    // (+1, 9543210, 87)
_Decimal32 y = 9500000e90DF;    // (+1, 9500000, 90)
fprintf(stdout, "%.6Ha\n", x);
fprintf(stdout, "%.5Ha\n", x);
fprintf(stdout, "%.4Ha\n", x);
fprintf(stdout, "%.3Ha\n", x);
fprintf(stdout, "%.2Ha\n", x);
fprintf(stdout, "%.1Ha\n", x);
fprintf(stdout, "%.1Ha\n", y);
```

assuming default rounding, results in:

```
9.54321e+93
9.5432e+93
9.543e+93
9.54e+93
9.5e+93
1e+94
1e+97
```

To further illustrate the effects of the exponent range, the sequence:

```
_Decimal32 x = 9512345e90DF;     // (+1, 9512345, 90)
_Decimal32 y = 9512345e86DF;     // (+1, 9512345, 86)
fprintf(stdout, "%.3Ha\n", x);
fprintf(stdout, "%.2Ha\n", x);
fprintf(stdout, "%.1Ha\n", x);
fprintf(stdout, "%.2Ha\n", y);
```

assuming default rounding, results in:

```
9.51e+96
9.5e+96
1e+97
9.5e+92
```

Forward references: conversion state (7.31.6), the `wcrtomb` function (7.31.6.3.3).

7.23.6.2 The `fscanf` function

Synopsis

1
```
#include <stdio.h>
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

Description

2 The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state.The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

— An optional assignment-suppressing character `*`.

— An optional decimal integer greater than zero that specifies the maximum field width (in characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* character that specifies the type of conversion to be applied.

4 The `fscanf` function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. The directive never fails.

6 A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive,the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space characters are skipped, unless the specification includes a `[`, `c`, or `n` specifier.[346)]

9 An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[347)] The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless

[346)] These white-space characters are not counted against a specified field width.

[347)] `fscanf` pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to `strtod`, `strtol`, etc., are unacceptable to `fscanf`.

assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

`hh` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`signed char`** or **`unsigned char`**.

`h` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`short int`** or **`unsigned short int`**.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`long int`** or **`unsigned long int`**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`double`**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **`wchar_t`**.

`ll` (ell-ell) Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`long long int`** or **`unsigned long long int`**.

`j` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`intmax_t`** or **`uintmax_t`**.

`z` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`size_t`** or the corresponding signed integer type.

`t` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **`ptrdiff_t`** or the corresponding unsigned integer type.

`w`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X`, or `n` conversion specifier applies to an argument which is a pointer to an integer with a specific width where *N* is a positive decimal integer with no leading zeros. All minimum-width integer types (7.22.1.2) and exact-width integer types (7.22.1.1) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`wf`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X`, or `n` conversion specifier applies to an argument which is a pointer to a fastest minimum-width integer with a specific width where *N* is a positive decimal integer with no leading zeros. All fastest minimum-width integer types (7.22.1.3) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`long double`**.

`H` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`_Decimal32`**.

`D` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`_Decimal64`**.

`DD` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **`_Decimal128`**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 In the following, the type of the corresponding argument for a conversion specifier shall be a pointer to a type determined by the length modifiers, if any, or specified by the conversion specifier. The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **int**.

b Matches an optionally signed binary integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 2 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **int**.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. Unless a length modifier is specified, the corresponding argument shall be a pointer to **float**.

c Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).
[348)]

If no l length modifier is present, the corresponding argument shall be a pointer to **char**, **signed char**, **unsigned char**, or **void** that points to storage large enough to accept the sequence. No null character is added.

If an l length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to storage of **wchar_t** large enough to accept the resulting sequence of wide characters.No null wide character is added.

s Matches a sequence of non-white-space characters.[348)]

If no l length modifier is present, the corresponding argument shall be a pointer to **char**, **signed char**, **unsigned char**, or **void** that points to storage large enough to accept the sequence and a terminating null character, which will be added automatically.

If an l length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to storage of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[348)]No special provisions are made for multibyte characters in the matching rules used by the c, s, and [conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[348)]

If no `l` length modifier is present, the corresponding argument shall be a pointer to **`char`**, **`signed char`**, **`unsigned char`**, or **`void`** that points to storage large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer that points to storage of **`wchar_t`** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the **`fprintf`** function. The corresponding argument shall be a pointer to a pointer of **`void`**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n` No input is consumed. The corresponding argument shall be a pointer of a signed integer type. The number of characters read from the input stream so far by this call to the **`fscanf`** function is stored into the integer object pointed to by the argument. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the **`fscanf`** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

`%` Matches a single `%` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[349)]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

15 Trailing white-space characters (including new-line characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns

16 The **`fscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure or if the implementation does not support a specific width length modifier.

17 **EXAMPLE 1** The call:

349)See "future library directions" (7.33.15).

```
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `thompson\0`.

18 **EXAMPLE 2** The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip `0123`, and will assign to `name` the sequence `56\0`. The next character read from the input stream will be `a`.

19 **EXAMPLE 3** To accept repeatedly from `stdin` a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21], item[21];
do {
      count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
      fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

20 If the `stdin` stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS     of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

21 **EXAMPLE 4** In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to `d1` and the value 3 to `n1`. Because `%n` can never get an input failure, the value of 3 is also assigned to `n2`. The value of `d2` is not affected. The value 1 is assigned to `i`.

22 **EXAMPLE 5** The call:

```
#include <stdio.h>
/* ... */
int n, i;
n = sscanf("foo  %bar  42", "foo%%bar%d", &i);
```

will assign to `n` the value 1 and to `i` the value 42 because input white-space characters are skipped for both the `%` and `d` conversion specifiers.

23 **EXAMPLE 6** In these examples, multibyte characters do have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a □ and the second by an uppercase letter, but are only recognized as such when in the alternate shift state. The shift sequences are denoted by ↑ and ↓, in which the first causes entry into the alternate shift state.

24 After the call:

```
#include <stdio.h>
/* ... */
char str[50];
fscanf(stdin, "a%s", str);
```

with the input line:

```
a↑□X□Y↓ bc
```

`str` will contain `↑□X□Y↓\\0` assuming that none of the bytes of the shift sequences (or of the multibyte characters, in the more general case) appears to be a single-byte white-space character.

25 In contrast, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a%ls", wstr);
```

with the same input line, `wstr` will contain the two wide characters that correspond to `□X` and `□Y` and a terminating null wide character.

26 However, the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□X↓%ls", wstr);
```

with the same input line will return zero due to a matching failure against the ↓ sequence in the format string.

27 Assuming that the first byte of the multibyte character □X is the same as the first byte of the multibyte character □Y, after the call:

```
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a↑□Y↓%ls", wstr);
```

with the same input line, zero will again be returned, but **stdin** will be left with a partially consumed multibyte character.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.24.1.5), the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.24.1.7), conversion state (7.31.6), the **wcrtomb** function (7.31.6.3.3).

7.23.6.3 The printf function

Synopsis

1
```
#include <stdio.h>
int printf(const char * restrict format, ...);
```

Description

2 The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Returns

3 The **printf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.23.6.4 The scanf function

Synopsis

1
```
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

Description

2 The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Returns

3 The **scanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.23.6.5 The snprintf function

Synopsis

1
```
#include <stdio.h>
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```

Description

2 The **snprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by argument `s`) rather than to a stream. If `n` is zero, nothing is written, and `s` may be a null pointer. Otherwise, output characters beyond the `n-1`st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **snprintf** function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than n.

7.23.6.6 The sprintf function

Synopsis

1
```
#include <stdio.h>
int sprintf(char * restrict s, const char * restrict format, ...);
```

Description

2 The **sprintf** function is equivalent to **fprintf**, except that the output is written into an array (specified by the argument s) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.23.6.7 The sscanf function

Synopsis

1
```
#include <stdio.h>
int sscanf(const char * restrict s, const char * restrict format, ...);
```

Description

2 The **sscanf** function is equivalent to **fscanf**, except that input is obtained from a string (specified by the argument s) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.23.6.8 The vfprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);
```

Description

2 The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the **va_end** macro[350].

Returns

3 The **vfprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

[350] As the functions **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, and **vsscanf** invoke the **va_arg** macro, arg after the return has an indeterminate representation.

4 **EXAMPLE** The following shows the use of the **vfprintf** function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>

void error(char *function_name, char *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fprintf(stderr, "ERROR in %s: ", function_name);
      // print out remainder of message
      vfprintf(stderr, format, args);
      va_end(args);
}
```

7.23.6.9 The vfscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);
```

Description

2 The **vfscanf** function is equivalent to **fscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf** function does not invoke the **va_end** macro.[350]

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.23.6.10 The vprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char * restrict format, va_list arg);
```

Description

2 The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[350]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.23.6.11 The vscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format, va_list arg);
```

Description

2 The **vscanf** function is equivalent to **scanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf** function does not invoke the **va_end** macro.[350)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.23.6.12 The vsnprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list
    arg);
```

Description

2 The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro.[350)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsnprintf** function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than n.

7.23.6.13 The vsprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s, const char * restrict format, va_list arg);
```

Description

2 The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro.[350)] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **vsprintf** function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.23.6.14 The vsscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
```

Description

2 The `vsscanf` function is equivalent to `sscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsscanf` function does not invoke the `va_end` macro.[350)]

Returns

3 The `vsscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vsscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.23.7 Character input/output functions

7.23.7.1 The `fgetc` function

Synopsis

1
```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by `stream` is not set and a next character is present, the `fgetc` function obtains that character as an `unsigned char` converted to an `int` and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the `fgetc` function returns `EOF`. Otherwise, the `fgetc` function returns the next character from the input stream pointed to by `stream`. If a read error occurs, the error indicator for the stream is set and the `fgetc` function returns `EOF`.[351)]

7.23.7.2 The `fgets` function

Synopsis

1
```
#include <stdio.h>
char *fgets(char * restrict s, int n, FILE * restrict stream);
```

Description

2 The `fgets` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns

3 The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the members of the array have unspecified values and a null pointer is returned.

7.23.7.3 The `fputc` function

Synopsis

1
```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

[351)] An end-of-file and a read error can be distinguished by use of the `feof` and `ferror` functions.

Description

2 The **fputc** function writes the character specified by c (converted to an **unsigned char**) to the output stream pointed to by stream, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

7.23.7.4 The fputs function

Synopsis

1
```
#include <stdio.h>
int fputs(const char * restrict s, FILE * restrict stream);
```

Description

2 The **fputs** function writes the string pointed to by s to the stream pointed to by stream. The terminating null character is not written.

Returns

3 The **fputs** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.23.7.5 The getc function

Synopsis

1
```
#include <stdio.h>
int getc(FILE *stream);
```

Description

2 The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it may evaluate stream more than once, so the argument should never be an expression with side effects.

Returns

3 The **getc** function returns the next character from the input stream pointed to by stream. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

7.23.7.6 The getchar function

Synopsis

1
```
#include <stdio.h>
int getchar(void);
```

Description

2 The **getchar** function is equivalent to **getc** with the argument **stdin**.

Returns

3 The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

7.23.7.7 The putc function

Synopsis

1
```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description

2 The `putc` function is equivalent to `fputc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so that argument should never be an expression with side effects.

Returns

3 The `putc` function returns the character written. If a write error occurs, the error indicator for the stream is set and `putc` returns `EOF`.

7.23.7.8 The `putchar` function

Synopsis

1
```
#include <stdio.h>
int putchar(int c);
```

Description

2 The `putchar` function is equivalent to `putc` with the second argument `stdout`.

Returns

3 The `putchar` function returns the character written. If a write error occurs, the error indicator for the stream is set and `putchar` returns `EOF`.

7.23.7.9 The puts function

Synopsis

1
```
#include <stdio.h>
int puts(const char *s);
```

Description

2 The **puts** function writes the string pointed to by s to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

Returns

3 The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

7.23.7.10 The ungetc function

Synopsis

1
```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

2 The **ungetc** function pushes the character specified by c (converted to an **unsigned char**) back onto the input stream pointed to by stream. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

3 One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of c equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

5 A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back.[352)] For a text stream, the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it has an indeterminate representation after the call[353)].

Returns

6 The **ungetc** function returns the character pushed back after conversion, or **EOF** if the operation fails.

Forward references: file positioning functions (7.23.9).

7.23.8 Direct input/output functions

7.23.8.1 The fread function

Synopsis

1
```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
      FILE * restrict stream);
```

---

352)Note that a file positioning function could further modify the file position indicator after discarding any pushed-back characters.

353)See "future library directions" (7.33.15).

Description

2 The **fread** function reads, into the array pointed to by ptr, up to nmemb elements whose size is specified by size, from the stream pointed to by stream. For each object, size calls are made to the **fgetc** function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting representation of the file position indicator for the stream is indeterminate. If a partial element is read, its representation is indeterminate.

Returns

3 The **fread** function returns the number of elements successfully read, which may be less than nmemb if a read error or end-of-file is encountered. If size or nmemb is zero, **fread** returns zero and the contents of the array and the state of the stream remain unchanged.

7.23.8.2 The fwrite function

Synopsis

1
```
#include <stdio.h>
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
     FILE * restrict stream);
```

Description

2 The **fwrite** function writes, from the array pointed to by ptr, up to nmemb elements whose size is specified by size, to the stream pointed to by stream. For each object, size calls are made to the **fputc** function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting representation of the file position indicator for the stream is indeterminate.

Returns

3 The **fwrite** function returns the number of elements successfully written, which will be less than nmemb only if a write error is encountered. If size or nmemb is zero, **fwrite** returns zero and the state of the stream remains unchanged.

7.23.9 File positioning functions

7.23.9.1 The fgetpos function

Synopsis

1
```
#include <stdio.h>
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

Description

2 The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by stream in the object pointed to by pos. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Returns

3 If successful, the **fgetpos** function returns zero; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Forward references: the **fsetpos** function (7.23.9.3).

7.23.9.2 The fseek function

Synopsis

1
```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

2 The **fseek** function sets the file position indicator for the stream pointed to by `stream`. If a read or write error occurs, the error indicator for the stream is set and **fseek** fails.

3 For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`. The specified position is the beginning of the file if `whence` is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a `whence` value of **SEEK_END**.

4 For a text stream, either `offset` shall be zero, or `offset` shall be a value returned by an earlier successful call to the **ftell** function on a stream associated with the same file and `whence` shall be **SEEK_SET**.

5 After determining the new position, a successful call to the **fseek** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful **fseek** call, the next operation on an update stream may be either input or output.

Returns

6 The **fseek** function returns nonzero only for a request that cannot be satisfied.

Forward references: the **ftell** function (7.23.9.4).

7.23.9.3 The fsetpos function

Synopsis

1
```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

2 The **fsetpos** function sets the **mbstate_t** object (if any) and file position indicator for the stream pointed to by `stream` according to the value of the object pointed to by `pos`, which shall be a value obtained from an earlier successful call to the **fgetpos** function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and **fsetpos** fails.

3 A successful call to the **fsetpos** function undoes any effects of the **ungetc** function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful **fsetpos** call, the next operation on an update stream may be either input or output.

Returns

4 If successful, the **fsetpos** function returns zero; on failure, the **fsetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

7.23.9.4 The ftell function

Synopsis

1
```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

2 The **ftell** function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of characters from the beginning of the file.

For a text stream, its file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

Returns

3 If successful, the `ftell` function returns the current value of the file position indicator for the stream. On failure, the `ftell` function returns -1L and stores an implementation-defined positive value in `errno`.

7.23.9.5 The `rewind` function

Synopsis

1
```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

2 The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns

3 The `rewind` function returns no value.

7.23.10 Error-handling functions

7.23.10.1 The `clearerr` function

Synopsis

1
```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

2 The `clearerr` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

Returns

3 The `clearerr` function returns no value.

7.23.10.2 The `feof` function

Synopsis

1
```
#include <stdio.h>
int feof(FILE *stream);
```

Description

2 The `feof` function tests the end-of-file indicator for the stream pointed to by `stream`.

Returns

3 The `feof` function returns nonzero if and only if the end-of-file indicator is set for `stream`.

7.23.10.3 The `ferror` function

Synopsis

1
```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

2 The `ferror` function tests the error indicator for the stream pointed to by `stream`.

Returns

3 The `ferror` function returns nonzero if and only if the error indicator is set for `stream`.

7.23.10.4 The `perror` function

Synopsis

1
```
#include <stdio.h>
void perror(const char *s);
```

Description

2 The `perror` function maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first (if `s` is not a null pointer and the character pointed to by `s` is not the null character), the string pointed to by `s` followed by a colon (`:`) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the `strerror` function with argument `errno`.

Returns

3 The `perror` function returns no value.

Forward references: the `strerror` function (7.26.6.3).

7.24 General utilities <stdlib.h>

1 The header `<stdlib.h>` declares five types and several functions of general utility, and defines several macros.[354)]

2 The feature test macro `__STDC_VERSION_STDLIB_H__` expands to the token `202311L`.

3 The types declared are `size_t` and `wchar_t` (both described in 7.21), `once_flag` (described in 7.28),

```
div_t
```

which is a structure type that is the type of the value returned by the `div` function,

```
ldiv_t
```

which is a structure type that is the type of the value returned by the `ldiv` function, and

```
lldiv_t
```

which is a structure type that is the type of the value returned by the `lldiv` function.

4 The macros defined are `NULL` (described in 7.21); `ONCE_FLAG_INIT` (described in 7.28);

```
EXIT_FAILURE
```

and

```
EXIT_SUCCESS
```

which expand to integer constant expressions that can be used as the argument to the `exit` function to return unsuccessful or successful termination status, respectively, to the host environment;

```
RAND_MAX
```

which expands to an integer constant expression that is the maximum value returned by the `rand` function; and

```
MB_CUR_MAX
```

which expands to a positive integer expression with type `size_t` that is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category `LC_CTYPE`), which is never greater than `MB_LEN_MAX`.

5 The function

```
#include <stdlib.h>
void call_once(once_flag *flag, void (*func)(void));
```

is described in 7.28.2.

7.24.1 Numeric conversion functions

1 The functions `atof`, `atoi`, `atol`, and `atoll` need not affect the value of the integer expression `errno` on an error. If the value of the result cannot be represented, the behavior is undefined.

7.24.1.1 The atof function

Synopsis

1
```
#include <stdlib.h>
double atof(const char *nptr);
```

[354)]See "future library directions" (7.33.16).

Description

2 The **atof** function converts the initial portion of the string pointed to by nptr to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, nullptr)
```

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.24.1.5).

7.24.1.2 The atoi, atol, and atoll functions

Synopsis

1
```
#include <stdlib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by nptr to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi:  (int)strtol(nptr, nullptr, 10)
atol:  strtol(nptr, nullptr, 10)
atoll: strtoll(nptr, nullptr, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.24.1.7).

7.24.1.3 The strfromd, strfromf, and strfroml functions

Synopsis

1
```
#include <stdlib.h>
int strfromd(char *restrict s, size_t n, const char *restrict format, double fp);
int strfromf(char *restrict s, size_t n, const char *restrict format, float fp);
int strfroml(char *restrict s, size_t n, const char *restrict format, long double fp);
```

Description

2 The **strfromd**, **strfromf**, and **strfroml** functions are equivalent to **snprintf**(s, n, format, fp) (7.23.6.5), except that the format string shall only contain the character %, an optional precision that does not contain an asterisk *, and one of the conversion specifiers a, A, e, E, f, F, g, or G, which applies to the type (**double**, **float**, or **long double**) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other format string results in undefined behavior.

Returns

3 The **strfromd**, **strfromf**, and **strfroml** functions return the number of characters that would have been written had n been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than n.

7.24.1.4 The strfromd*N* functions

Synopsis

1
```
#include <stdlib.h>
```

```
#ifdef __STDC_IEC_60559_DFP__
int strfromd32(char*restrict s, size_t n, const char*restrict format, _Decimal32 fp);
int strfromd64(char*restrict s, size_t n, const char*restrict format, _Decimal64 fp);
int strfromd128(char*restrict s, size_t n, const char*restrict format, _Decimal128 fp);
#endif
```

Description

2 The **strfromd***N* functions are equivalent to `snprintf(s, n, format, fp)` (7.23.6.5), except the format string contains only the character `%`, an optional precision that does not contain an asterisk `*`, and one of the conversion specifiers `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, which applies to the type (**_Decimal32**, **_Decimal64**, or **_Decimal128**) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other format string results in undefined behavior.

Returns

3 The **strfromd***N* functions return the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than `n`.

7.24.1.5 The `strtod`, `strtof`, and `strtold` functions

Synopsis

1
```
#include <stdlib.h>
double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

Description

2 The **strtod**, **strtof**, and **strtold** functions convert the initial portion of the string pointed to by `nptr` to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling a floating constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2, excluding any digit separators (6.4.4.1);

— a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part as defined in 6.4.4.2, excluding any digit separators;

— `INF` or `INFINITY`, ignoring case

— `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)`, ignoring case in the `NAN` part, where:

n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated[355].

A character sequence `INF` or `INFINITY` is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)` is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequence is implementation-defined.[356] A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

5 If the subject sequence has the hexadecimal form and **`FLT_RADIX`** is a power of 2, the value resulting from the conversion is correctly rounded.

6 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Recommended practice

8 If the subject sequence has the hexadecimal form, **`FLT_RADIX`** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

9 If the subject sequence has the decimal form and at most M significant digits, where M is the maximum value of the *T*`_DECIMAL_DIG` macros (defined in `<float.h>`), the result should be correctly rounded. If the subject sequence D has the decimal form and more than M significant digits, consider the two bounding, adjacent decimal strings L and U, both having M significant digits, such that the values of L, D, and U satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.[357]

Returns

10 The functions return the converted value, if any. If no conversion could be performed, zero is returned.

If the correct value overflows and default rounding is in effect (7.12.1), plus or minus **`HUGE_VAL`**, **`HUGE_VALF`**, or **`HUGE_VALL`** is returned (according to the return type and sign of the value); if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, the integer expression **`errno`** acquires the value of **`ERANGE`**; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, the "overflow" floating-point exception is raised.

If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero, whether **`errno`** acquires the value **`ERANGE`** is implementation-defined; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

[355] It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods could yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

[356] An implementation can use the n-char sequence to determine extra information to be represented in the NaN's significand.

[357] M is sufficiently large that L and U will usually correctly round to the same internal floating value, but if not will correctly round to adjacent values.

7.24.1.6 The `strtod`*N* functions

Synopsis

1
```
#include <stdlib.h>
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 strtod32(const char * restrict nptr, char ** restrict endptr);
_Decimal64 strtod64(const char * restrict nptr,char ** restrict endptr);
_Decimal128 strtod128(const char * restrict nptr,char ** restrict endptr);
#endif
```

Description

2 The `strtod`*N* functions convert the initial portion of the string pointed to by `nptr` to decimal floating type representation. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters; a subject sequence resembling a floating constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2, excluding any digit separators (6.4.4.1)

— `INF` or `INFINITY`, ignoring case

— `NAN` or `NAN(`*d-char-sequence*$_{opt}$`)`, ignoring case in the `NAN` part, where:

d-char-sequence:
digit
nondigit
d-char-sequence digit
d-char-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, including correct rounding and determination of the coefficient c and the quantum exponent q, with the following exceptions:

— It is not a hexadecimal floating number.

— The decimal-point character is used in place of a period.

— If neither an exponent part nor a decimal-point character appears in a decimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated before rounding and the sign s is set to -1, else s is set to 1. A character sequence `INF` or `INFINITY` is interpreted as an infinity. A character sequence `NAN` or `NAN(`*d-char-sequence*$_{opt}$`)`, is interpreted as a quiet NaN; the meaning of the d-char sequence is implementation-defined.[358] A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

[358]An implementation may use the d-char sequence to determine extra information to be represented in the NaN's significand.

5 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

6 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns

7 The **`strtod`**N functions return the correctly rounded converted value, if any. If no conversion could be performed, the value of the triple $(+1, 0, 0)$ is returned. If the correct value overflows:

— the value of the macro **`ERANGE`** is stored in errno if the integer expression **`math_errhandling`** & **`MATH_ERRNO`** is nonzero;

— the "overflow" floating-point exception is raised if the integer expression **`math_errhandling`** & **`MATH_ERREXCEPT`** is nonzero.

If the result underflows (7.12.1), whether **`errno`** acquires the value **`ERANGE`** if the integer expression **`math_errhandling`** & **`MATH_ERRNO`** is nonzero is implementation-defined; if the integer expression **`math_errhandling`** & **`MATH_ERREXCEPT`** is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

8 **EXAMPLE** Following are subject sequences of the decimal form and the resulting triples (s, c, q) produced by **`strtod64`**. Note that for **`_Decimal64`**, the precision (maximum coefficient length) is 16 and the quantum exponent range is $-398 \leq q \leq 369$.

`"0"`	$(+1, 0, 0)$
`"0.00"`	$(+1, 0, -2)$
`"123"`	$(+1, 123, 0)$
`"-123"`	$(-1, 123, 0)$
`"1.23E3"`	$(+1, 123, 1)$
`"1.23E+3"`	$(+1, 123, 1)$
`"12.3E+7"`	$(+1, 123, 6)$
`"12.0"`	$(+1, 120, -1)$
`"12.3"`	$(+1, 123, -1)$
`"0.00123"`	$(+1, 123, -5)$
`"-1.23E-12"`	$(-1, 123, -14)$
`"1234.5E-4"`	$(+1, 12345, -5)$
`"-0"`	$(-1, 0, 0)$
`"-0.00"`	$(-1, 0, -2)$
`"0E+7"`	$(+1, 0, 7)$
`"-0E-7"`	$(-1, 0, -7)$
`"12345678901234567890"`	$(+1, 1234567890123457, 4)$ or $(+1, 1234567890123456, 4)$ depending on rounding mode
`"1234E-400"`	$(+1, 12, -398)$ or $(+1, 13, -398)$ depending on rounding mode
`"1234E-402"`	$(+1, 0, -398)$ or $(+1, 1, -398)$ depending on rounding mode
`"1000."`	$(+1, 1000, 0)$
`".0001"`	$(+1, 1, -4)$
`"1000.e0"`	$(+1, 1000, 0)$
`".0001e0"`	$(+1, 1, -4)$
`"1000.0"`	$(+1, 10000, -1)$
`"0.0001"`	$(+1, 1, -4)$
`"1000.00"`	$(+1, 100000, -2)$
`"00.0001"`	$(+1, 1, -4)$
`"001000."`	$(+1, 1000, 0)$
`"001000.0"`	$(+1, 10000, -1)$
`"001000.00"`	$(+1, 100000, -2)$
`"00.00"`	$(+1, 0, -2)$
`"00."`	$(+1, 0, 0)$
`".00"`	$(+1, 0, -2)$

`"00.00e-5"`	$(+1, 0, -7)$
`"00.e-5"`	$(+1, 0, -5)$
`".00e-5"`	$(+1, 0, -7)$
`"0x1.8p+4"`	$(+1, 0, 0)$, and a pointer to `"x1.8p+4"` is stored in the object pointed to by `endptr`, provided `endptr` is not a null pointer
`"infinite"`	infinity, and a pointer to `"inite"` is stored in the object pointed to by `endptr`, provided `endptr` is not a null pointer

7.24.1.7 The `strtol`, `strtoll`, `strtoul`, and `strtoull` functions

Synopsis

1
```c
#include <stdlib.h>
long int strtol(const char *restrict nptr, char **restrict endptr, int base);
long long int strtoll(const char *restrict nptr, char **restrict endptr, int base);
unsigned long int strtoul(const char *restrict nptr, char **restrict endptr, int base);
unsigned long long int strtoull(const char *restrict nptr, char **restrict endptr, int
    base);
```

Description

2 The **`strtol`**, **`strtoll`**, **`strtoul`**, and **`strtoull`** functions convert the initial portion of the string pointed to by `nptr` to **`long int`**, **`long long int`**, **`unsigned long int`**, and **`unsigned long long int`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters, a subject sequence resembling an integer represented in some radix determined by the value of `base`, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

3 If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant as described in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix or any optional digit separators. If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix or any optional digit separators. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of base is 2, the characters `0b` or `0B` may optionally precede the sequence of letters and digits, following the sign if present. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

4 The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

5 If the subject sequence has the expected form and the value of `base` is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

6 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns

8 The **`strtol`**, **`strtoll`**, **`strtoul`**, and **`strtoull`** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of

representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.24.2 Pseudo-random sequence generation functions

7.24.2.1 The rand function

Synopsis

1
```
#include <stdlib.h>
int rand(void);
```

Description

2 The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX** inclusive.

3 The **rand** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **rand** function.

Recommended practice

4 There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Returns

5 The **rand** function returns a pseudo-random integer.

Environmental limits

6 The value of the **RAND_MAX** macro shall be at least 32767.

7.24.2.2 The srand function

Synopsis

1
```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

2 The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence shall be generated as when **srand** is first called with a seed value of 1.

3 The **srand** function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the **srand** function.

Returns

4 The **srand** function returns no value.

5 **EXAMPLE** The following functions define a portable implementation of **rand** and **srand**.

```
static unsigned long int next = 1;

int rand(void)   //  RAND_MAX assumed to be 32767
{
      next = next * 1103515245 + 12345;
      return (unsigned int)(next/65536) % 32768;
}
```

```
void srand(unsigned int seed)
{
        next = seed;
}
```

7.24.3 Memory management functions

1 The order and contiguity of storage allocated by successive calls to the **aligned_alloc**, **calloc**, **malloc**, and **realloc** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested. It may then be used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

2 For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only memory locations accessible through their arguments and not other static duration storage. These functions may, however, visibly modify the storage that they allocate or deallocate. Calls to these functions that allocate or deallocate a particular region of memory shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.

7.24.3.1 The aligned_alloc function

Synopsis

1
```
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```

Description

2 The **aligned_alloc** function allocates space for an object whose alignment is specified by `alignment`, whose size is specified by `size`, and whose representation is indeterminate. If the value of `alignment` is not a valid alignment supported by the implementation the function shall fail by returning a null pointer.

Returns

3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated space.

7.24.3.2 The calloc function

Synopsis

1
```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

2 The **calloc** function allocates space for an array of `nmemb` objects, each of whose size is `size`. The space is initialized to all bits zero.[359)]

Returns

3 The **calloc** function returns either a pointer to the allocated space or a null pointer if the space cannot be allocated or if the product `nmemb * size` would wraparound **size_t**.

[359)]Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

7.24.3.3 The `free` function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
```

Description

2 The **free** function causes the space pointed to by `ptr` to be deallocated, that is, made available for further allocation. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.24.3.4 The `free_sized` function

Synopsis

1
```
#include <stdlib.h>
void free_sized(void *ptr, size_t size);
```

Description

2 If `ptr` is a null pointer or the result obtained from a call to **malloc**, **realloc**, or **calloc**, where `size` size is equal to the requested allocation size, this function is equivalent to **free**(`ptr`). Otherwise, the behavior is undefined.

3 **NOTE 1** A conforming implementation may ignore `size` and call **free**.

4 **NOTE 2** The result of an **aligned_alloc** call may not be passed to **free_sized**.

Recommended practice

5 Implementations may provide extensions to query the usable size of an allocation, or to determine the usable size of the allocation that would result if a request for some other size were to succeed. Such implementations should allow passing the resulting usable size as the `size` parameter, and provide functionality equivalent to **free** in such cases.

Returns

6 The **free_sized** function returns no value.

7.24.3.5 The `free_aligned_sized` function

Synopsis

1
```
#include <stdlib.h>
void free_aligned_sized(void *ptr, size_t alignment, size_t size);
```

Description

2 If `ptr` is a null pointer or the result obtained from a call to **aligned_alloc**, where `alignment` is equal to the requested allocation alignment and `size` is equal to the requested allocation size, this function is equivalent to **free**(`ptr`). Otherwise, the behavior is undefined.

3 **NOTE 1** A conforming implementation may ignore `alignment` and `size` and call **free**.

4 **NOTE 2** The result of an **malloc**, **calloc**, or **realloc** call may not be passed to **free_aligned_sized**.

Recommended practice

5 Implementations may provide extensions to query the usable size of an allocation, or to determine the usable size of the allocation that would result if a request for some other size were to succeed. Such implementations should allow passing the resulting usable size as the `size` parameter, and provide functionality equivalent to **free** in such cases.

Returns

6 The `free_aligned_sized` function returns no value.

7.24.3.6 The `malloc` function

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

2 The `malloc` function allocates space for an object whose size is specified by `size` and whose representation is indeterminate.

Returns

3 The `malloc` function returns either a null pointer or a pointer to the allocated space.

7.24.3.7 The `realloc` function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Description

2 The `realloc` function deallocates the old object pointed to by `ptr` and returns a pointer to a new object that has the size specified by `size`. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have unspecified values.

3 If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to the `free` or `realloc` function, or if the `size` is zero, the behavior is undefined. If memory for the new object is not allocated, the old object is not deallocated and its value is unchanged.

Returns

4 The `realloc` function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object has not been allocated.

7.24.4 Communication with the environment

7.24.4.1 The `abort` function

Synopsis

1
```
#include <stdlib.h>
[[noreturn]] void abort(void);
```

Description

2 The `abort` function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call `raise(SIGABRT)`.

Returns

3 The `abort` function does not return to its caller.

7.24.4.2 The `atexit` function

Synopsis

1
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The `atexit` function registers the function pointed to by `func`, to be called without arguments at normal program termination.[360] It is unspecified whether a call to the `atexit` function that does not happen before the `exit` function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The `atexit` function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the `at_quick_exit` function (7.24.4.3), the `exit` function (7.24.4.4).

7.24.4.3 The `at_quick_exit` function

Synopsis

1
```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

Description

2 The `at_quick_exit` function registers the function pointed to by `func`, to be called without arguments should `quick_exit` be called.[361] It is unspecified whether a call to the `at_quick_exit` function that does not happen before the `quick_exit` function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The `at_quick_exit` function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the `quick_exit` function (7.24.4.7).

7.24.4.4 The `exit` function

Synopsis

1
```
#include <stdlib.h>
[[noreturn]] void exit(int status);
```

Description

2 The `exit` function causes normal program termination to occur. No functions registered by the `at_quick_exit` function are called. If a program calls the `exit` function more than once, or calls the `quick_exit` function in addition to the `exit` function, the behavior is undefined.

3 First, all functions registered by the `atexit` function are called, in the reverse order of their registration,[362] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the `longjmp` function is made that would terminate the call to the registered function, the behavior is undefined.

[360]The `atexit` function registrations are distinct from the `at_quick_exit` registrations, so applications might need to call both registration functions with the same argument.
[361]The `at_quick_exit` function registrations are distinct from the `atexit` registrations, so applications might need to call both registration functions with the same argument.
[362]Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

4 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

5 Finally, control is returned to the host environment. If the value of status is zero or **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of status is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

6 The **exit** function cannot return to its caller.

7.24.4.5 The _Exit function

Synopsis

1
```
#include <stdlib.h>
[[noreturn]] void _Exit(int status);
```

Description

2 The **_Exit** function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the **atexit** function, the **at_quick_exit** function, or signal handlers registered by the **signal** function are called. The status returned to the host environment is determined in the same way as for the **exit** function (7.24.4.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

Returns

3 The **_Exit** function cannot return to its caller.

7.24.4.6 The getenv function

Synopsis

1
```
#include <stdlib.h>
char *getenv(const char *name);
```

Description

2 The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by name. The set of environment names and the method for altering the environment list are implementation-defined. The **getenv** function need not avoid data races with other threads of execution that modify the environment list.[363)]

3 The implementation shall behave as if no library function calls the **getenv** function.

Returns

4 The **getenv** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **getenv** function. If the specified name cannot be found, a null pointer is returned.

7.24.4.7 The quick_exit function

Synopsis

1
```
#include <stdlib.h>
[[noreturn]] void quick_exit(int status);
```

Description

2 The **quick_exit** function causes normal program termination to occur. No functions registered by the **atexit** function or signal handlers registered by the **signal** function are called. If a program calls the **quick_exit** function more than once, or calls the **exit** function in addition to the **quick_exit**

363)Many implementations provide non-standard functions that modify the environment list.

function, the behavior is undefined. If a signal is raised while the **quick_exit** function is executing, the behavior is undefined.

3 The **quick_exit** function first calls all functions registered by the **at_quick_exit** function, in the reverse order of their registration,[364] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

4 Then control is returned to the host environment by means of the function call `_Exit(status)`.

Returns

5 The **quick_exit** function cannot return to its caller.

7.24.4.8 The `system` function

Synopsis

1
```
#include <stdlib.h>
int system(const char *string);
```

Description

2 If `string` is a null pointer, the **system** function determines whether the host environment has a *command processor*. If `string` is not a null pointer, the **system** function passes the string pointed to by `string` to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling **system** to behave in a non-conforming manner or to terminate.

Returns

3 If the argument is a null pointer, the **system** function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the **system** function does return, it returns an implementation-defined value.

7.24.5 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as `size_t nmemb` specifies the length of the array for a function, `nmemb` can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. Pointer arguments on such a call shall still have valid values, as described in 7.1.4.

2 The implementation shall ensure that the second argument of the comparison function (when called from **bsearch**), or both arguments (when called from **qsort**), are pointers to elements of the array.[365] The first argument when called from **bsearch** shall equal `key`.

3 The comparison function shall not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.

4 When the same objects (consisting of `size` bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for **qsort** they shall define a total ordering on the array, and for **bsearch** the same object shall always compare the same way with the key.

---

[364] Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

[365] That is, if the value passed is `p`, then the following expressions are always nonzero:

```
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p  <  (char *)base + nmemb * size
```

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

7.24.5.1 The `bsearch` generic function

Synopsis

1
```
#include <stdlib.h>
QVoid *bsearch(const void *key, QVoid *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Description

2 The `bsearch` generic function searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element of the array is specified by `size`.

3 The comparison function pointed to by `compar` is called with two arguments that point to the `key` object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.[366)]

Returns

4 The `bsearch` generic function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

5 The `bsearch` function is generic in the qualification of the type pointed to by the argument `base`. If this argument is a pointer to a `const`-qualified object type, the returned pointer will be a pointer to `const`-qualified `void`. Otherwise, the argument shall be a pointer to an unqualified object type or a null pointer constant[367)], and the returned pointer will be a pointer to unqualified `void`.

The external declaration of `bsearch` has the concrete type:

```
void * (const void *, const void *, size_t, size_t, int (*) (const void *, const
    void *))
```

which supports all correct uses. If a macro definition of this generic function is suppressed to access an actual function, the external declaration with this concrete type is visible.[368)]

7.24.5.2 The `qsort` function

Synopsis

1
```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
```

Description

2 The `qsort` function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.

3 The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

[366)] In practice, the entire array is sorted according to the comparison function.
[367)] If the argument is a null pointer and the call is executed, the behavior is undefined.
[368)] This is an obsolescent feature.

Returns

5 The `qsort` function returns no value.

7.24.6 Integer arithmetic functions

7.24.6.1 The `abs`, `labs`, and `llabs` functions

Synopsis

1
```
#include <stdlib.h>
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Description

2 The `abs`, `labs`, and `llabs` functions compute the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[369)]

Returns

3 The `abs`, `labs`, and `llabs`, functions return the absolute value.

7.24.6.2 The `div`, `ldiv`, and `lldiv` functions

Synopsis

1
```
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Description

2 The `div`, `ldiv`, and `lldiv`, functions compute `numer/denom` and `numer%denom` in a single operation.

Returns

3 The `div`, `ldiv`, and `lldiv` functions return a structure of type `div_t`, `ldiv_t`, and `lldiv_t`, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has the same type as the arguments `numer` and `denom`. If either part of the result cannot be represented, the behavior is undefined.

7.24.7 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the `LC_CTYPE` category of the current locale. For a state-dependent encoding, each of the `mbtowc` and `wctomb` functions is placed into its initial conversion state prior to the first call to the function and can be returned to that state by a call for which its character pointer argument, `s`, is a null pointer. Subsequent calls with `s` as other than a null pointer cause the internal conversion state of the function to be altered as necessary. It is implementation-defined whether internal conversion state has thread storage duration, and whether a newly created thread has the same state as the current thread at the time of creation, or the initial conversion state. A call with `s` as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise. [370)]

Changing the `LC_CTYPE` category causes the internal object describing the conversion state of the `mbtowc` and `wctomb` functions to have an indeterminate representation.

7.24.7.1 The `mblen` function

Synopsis

1

[369)] The absolute value of the most negative number may not be representable.

[370)] If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

2 If s is not a null pointer, the **mblen** function determines the number of bytes contained in the multibyte character pointed to by s. Except that the conversion state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, (const char *)0, 0);
mbtowc((wchar_t *)0, s, n);
```

Returns

3 If s is a null pointer, the **mblen** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the **mblen** function either returns 0 (if s points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

Forward references: the **mbtowc** function (7.24.7.2).

7.24.7.2 The mbtowc function

Synopsis

1
```
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

Description

2 If s is not a null pointer, the **mbtowc** function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if pwc is not a null pointer, stores that value in the object pointed to by pwc. If the corresponding wide character is the null wide character, the function is left in the initial conversion state.

3 The implementation shall behave as if no library function calls the **mbtowc** function.

Returns

4 If s is a null pointer, the **mbtowc** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the **mbtowc** function either returns 0 (if s points to the null character), or returns the number of bytes that are contained in the converted multibyte character (if the next n or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

5 In no case will the value returned be greater than n or the value of the **MB_CUR_MAX** macro.

7.24.7.3 The wctomb function

Synopsis

1
```
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

Description

2 The **wctomb** function determines the number of bytes needed to represent the multibyte character corresponding to the wide character given by wc (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s (if s is not a null pointer). At most **MB_CUR_MAX** characters are stored. If wc is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

3 The implementation shall behave as if no library function calls the **wctomb** function.

Returns

4 If s is a null pointer, the **wctomb** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the **wctomb** function returns -1 if the value of wc does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of wc.

5 In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

7.24.8 Multibyte/wide string conversion functions

1 The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

7.24.8.1 The mbstowcs function

Synopsis

1
```
#include <stdlib.h>
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

Description

2 The **mbstowcs** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by s into a sequence of corresponding wide characters and stores not more than n wide characters into the array pointed to by pwcs. No multibyte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multibyte character is converted as if by a call to the **mbtowc** function, except that the conversion state of the **mbtowc** function is not affected.

3 No more than n elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If an invalid multibyte character is encountered, the **mbstowcs** function returns (size_t)(-1). Otherwise, the **mbstowcs** function returns the number of array elements modified, not including a terminating null wide character, if any.[371)]

[371)] The array will not be null-terminated if the value returned is n.

7.24.8.2 The wcstombs function

Synopsis

1
```
#include <stdlib.h>
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

Description

2 The **wcstombs** function converts a sequence of wide characters from the array pointed to by pwcs into a sequence of corresponding multibyte characters that begins in the initial shift state, and stores these multibyte characters into the array pointed to by s, stopping if a multibyte character would exceed the limit of n total bytes or if a null character is stored. Each wide character is converted as if by a call to the **wctomb** function, except that the conversion state of the **wctomb** function is not affected.

3 No more than n bytes will be modified in the array pointed to by s. If copying takes place between objects that overlap, the behavior is undefined.

Returns

4 If a wide character is encountered that does not correspond to a valid multibyte character, the **wcstombs** function returns (**size_t**)(-1). Otherwise, the **wcstombs** function returns the number of bytes modified, not including a terminating null character, if any.[371)]

7.24.9 Alignment of memory

7.24.9.1 The memalignment function

Synopsis

1
```
#include <stdlib.h>
size_t memalignment(const void * p);
```

Description

2 The **memalignment** function accepts a pointer to any object and returns the maximum alignment satisfied by its address value. The alignment may be an extended alignment and may also be beyond the range supported by the implementation for explicit use by **alignas**[372)]. If so, it will satisfy all alignments usable by the implementation. The value returned can be compared to the result of **alignof**, and if it is greater or equal, the alignment requirement for the type operand is satisfied.

Returns

3 The alignment of the pointer p, which is a power of two. If p is a null pointer, an alignment of zero is returned.

4 **NOTE 1** An alignment of zero indicates that the tested pointer cannot be used to access an object of any type.

[372)] The actual alignment of an object may be stricter than the alignment requested for an object by **alignas** or (implicitly) by an allocation function, but will always satisfy it.

7.25 _Noreturn <stdnoreturn.h>

1 The header `<stdnoreturn.h>` defines the macro

```
noreturn
```

which expands to **`_Noreturn`**.

2 The **`noreturn`** macro and the `<stdnoreturn.h>` header are obsolescent features.

7.26 String handling <string.h>

7.26.1 String function conventions

1 The header <string.h> declares one type, several functions, several type-generic functions, and defines two macros useful for manipulating arrays of character type and other objects treated as arrays of character type.[373] The type is **size_t** and one of the macros is **NULL** (both described in 7.21). Various methods are used for determining the lengths of the arrays, but in all cases a **char** * or **void** * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 The macro

```
__STDC_VERSION_STRING_H__
```

is an integer constant expression with a value equivalent to *202311*L.

3 Where an argument declared as **size_t** n specifies the length of the array for a function, n can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

4 For all functions in this subclause, each character shall be interpreted as if it had the type **unsigned char** (and therefore every possible object representation is valid and has a different value).

7.26.2 Copying functions

7.26.2.1 The memcpy function

Synopsis

1
```
#include <string.h>
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Description

2 The **memcpy** function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **memcpy** function returns the value of s1.

7.26.2.2 The memccpy function

Synopsis

1
```
#include <string.h>
void *memccpy(void * restrict s1, const void * restrict s2, int c, size_t n);
```

Description

2 The **memccpy** function copies characters from the object pointed to by s2 into the object pointed to by s1, stopping after the first occurrence of character c (converted to an **unsigned char**) is copied, or after n characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **memccpy** function returns a pointer to the character after the copy of c in s1, or a null pointer if c was not found in the first n characters of s2.

7.26.2.3 The memmove function

[373]See "future library directions" (7.33.17).

Synopsis

1
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

2 The `memmove` function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` characters from the object pointed to by `s2` are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` characters from the temporary array are copied into the object pointed to by `s1`.

Returns

3 The `memmove` function returns the value of `s1`.

7.26.2.4 The `strcpy` function

Synopsis

1
```
#include <string.h>
char *strcpy(char * restrict s1, const char * restrict s2);
```

Description

2 The `strcpy` function copies the string pointed to by `s2` (including the terminating null character) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The `strcpy` function returns the value of `s1`.

7.26.2.5 The `strncpy` function

Synopsis

1
```
#include <string.h>
char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
```

Description

2 The `strncpy` function copies not more than `n` characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.[374] If copying takes place between objects that overlap, the behavior is undefined.

3 If the array pointed to by `s2` is a string that is shorter than `n` characters, null characters are appended to the copy in the array pointed to by `s1`, until `n` characters in all have been written.

Returns

4 The `strncpy` function returns the value of `s1`.

7.26.2.6 The `strdup` function

Synopsis

1
```
#include <string.h>
char *strdup(const char *s);
```

Description

2 The `strdup` function creates a copy of the string pointed to by `s` in a space allocated as if by a call to `malloc`.

[374] Thus, if there is no null character in the first `n` characters of the array pointed to by `s2`, the result will not be null-terminated.

Returns

3 The **strdup** function returns a pointer to the first character of the duplicate string. The returned pointer can be passed to **free**. If no space can be allocated the **strdup** function returns a null pointer.

7.26.2.7 The strndup function

Synopsis

1
```
#include <string.h>
char *strndup(const char *s, size_t size);
```

Description

2 The **strndup** function creates a string initialized with no more than `size` initial characters of the array pointed to by `s` and up to the first null character, whichever comes first, in a space allocated as if by a call to **malloc**. If the array pointed to by `s` does not contain a null within the first `size` characters, a null is appended to the copy of the array.

Returns

3 The **strndup** function returns a pointer to the first character of the created string. The returned pointer can be passed to **free**. If space cannot be allocated the **strndup** function returns a null pointer.

7.26.3 Concatenation functions

7.26.3.1 The strcat function

Synopsis

1
```
#include <string.h>
char *strcat(char * restrict s1, const char * restrict s2);
```

Description

2 The **strcat** function appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strcat** function returns the value of `s1`.

7.26.3.2 The strncat function

Synopsis

1
```
#include <string.h>
char *strncat(char * restrict s1, const char * restrict s2, size_t n);
```

Description

2 The **strncat** function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result.[375] If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strncat** function returns the value of `s1`.

Forward references: the **strlen** function (7.26.6.4).

---

[375] Thus, the maximum number of characters that can end up in the array pointed to by `s1` is `strlen(s1)+n+1`.

7.26.4 Comparison functions

1 The sign of a nonzero value returned by the comparison functions **memcmp**, **strcmp**, and **strncmp** is determined by the sign of the difference between the values of the first pair of characters (both interpreted as **unsigned char**) that differ in the objects being compared.

7.26.4.1 The memcmp function

Synopsis

1
```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

2 The **memcmp** function compares the first n characters of the object pointed to by s1 to the first n characters of the object pointed to by s2[376)].

Returns

3 The **memcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

7.26.4.2 The strcmp function

Synopsis

1
```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description

2 The **strcmp** function compares the string pointed to by s1 to the string pointed to by s2.

Returns

3 The **strcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

7.26.4.3 The strcoll function

Synopsis

1
```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description

2 The **strcoll** function compares the string pointed to by s1 to the string pointed to by s2, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Returns

3 The **strcoll** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 when both are interpreted as appropriate to the current locale.

7.26.4.4 The strncmp function

Synopsis

1
```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

[376)]The unused bytes used as padding for purposes of alignment within structure objects take on unspecified values when a value is stored in the object (see 6.2.6.1). Strings shorter than their allocated space and unions can also cause problems in comparison.

Description

2 The **strncmp** function compares not more than n characters (characters that follow a null character are not compared) from the array pointed to by s1 to the array pointed to by s2.

Returns

3 The **strncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2.

7.26.4.5 The strxfrm function

Synopsis

1
```
#include <string.h>
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Description

2 The **strxfrm** function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. If n is zero, s1 is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, the members of the array pointed to by s1 have an indeterminate representation.

4 **EXAMPLE** The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by s.

```
1 + strxfrm(NULL, s, 0)
```

7.26.5 Search functions

7.26.5.1 Introduction

1 The stateless search functions in this section (**memchr**, **strchr**, **strpbrk**, **strrchr**, **strstr**) are *generic functions*. These functions are generic in the qualification of the array to be searched and will return a result pointer to an element with the same qualification as the passed array. If the array to be searched is **const**-qualified, the result pointer will be to a **const**-qualified element. If the array to be searched is not **const**-qualified[377], the result pointer will be to an unqualified element.

2 The external declarations of these generic functions have a concrete function type that returns a pointer to an unqualified element (of type **char** when specified as ***QChar***, and **void** when specified as ***QVoid***), and accepts a pointer to a **const**-qualified array of the same type to search. This signature supports all correct uses. If a macro definition of any of these generic functions is suppressed to access an actual function, the external declaration with the corresponding concrete type is visible.[378]

3 The **volatile** and **restrict** qualifiers are not accepted on the elements of the array to search.

7.26.5.2 The memchr generic function

Synopsis

1
```
#include <string.h>
QVoid *memchr(QVoid *s, int c, size_t n);
```

[377] The null pointer constant is not a pointer to a **const**-qualified type, and therefore the result expression has the type of a pointer to an unqualified element; however, evaluating such a call is undefined.

[378] This is an obsolescent feature.

Description

2 The **memchr** generic function locates the first occurrence of c (converted to an **unsigned char**) in the initial n characters (each interpreted as **unsigned char**) of the object pointed to by s. The implementation shall behave as if it reads the characters sequentially and stops as soon as a matching character is found.

Returns

3 The **memchr** generic function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

7.26.5.3 The strchr generic function

Synopsis

1
```
#include <string.h>
QChar *strchr(QChar *s, int c);
```

Description

2 The **strchr** generic function locates the first occurrence of c (converted to a **char**) in the string pointed to by s. The terminating null character is considered to be part of the string.

Returns

3 The **strchr** generic function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

7.26.5.4 The strcspn function

Synopsis

1
```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description

2 The **strcspn** function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters *not* from the string pointed to by s2.

Returns

3 The **strcspn** function returns the length of the segment.

7.26.5.5 The strpbrk generic function

Synopsis

1
```
#include <string.h>
QChar *strpbrk(QChar *s1, const char *s2);
```

Description

2 The **strpbrk** generic function locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2.

Returns

3 The **strpbrk** generic function returns a pointer to the character, or a null pointer if no character from s2 occurs in s1.

7.26.5.6 The strrchr generic function

Synopsis

1
```
#include <string.h>
QChar *strrchr(QChar *s, int c);
```

Description

2 The **strrchr** generic function locates the last occurrence of c (converted to a **char**) in the string pointed to by s. The terminating null character is considered to be part of the string.

Returns

3 The **strrchr** generic function returns a pointer to the character, or a null pointer if c does not occur in the string.

7.26.5.7 The strspn function

Synopsis

1
```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description

2 The **strspn** function computes the length of the maximum initial segment of the string pointed to by s1 which consists entirely of characters from the string pointed to by s2.

Returns

3 The **strspn** function returns the length of the segment.

7.26.5.8 The strstr generic function

Synopsis

1
```
#include <string.h>
QChar *strstr(QChar *s1, const char *s2);
```

Description

2 The **strstr** generic function locates the first occurrence in the string pointed to by s1 of the sequence of characters (excluding the terminating null character) in the string pointed to by s2.

Returns

3 The **strstr** generic function returns a pointer to the located string, or a null pointer if the string is not found. If s2 points to a string with zero length, the function returns s1.

7.26.5.9 The strtok function

Synopsis

1
```
#include <string.h>
char *strtok(char * restrict s1, const char * restrict s2);
```

Description

2 A sequence of calls to the **strtok** function breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a character from the string pointed to by s2. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. If any of the subsequent calls in the sequence is made by a different thread than the first, the behavior is undefined. The separator string pointed to by s2 may be different from call to call.

3 The first call in the sequence searches the string pointed to by s1 for the first character that is *not* contained in the current separator string pointed to by s2. If no such character is found, then there are no tokens in the string pointed to by s1 and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

4 The **strtok** function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.

5 Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

6 The **strtok** function is not required to avoid data races with other calls to the **strtok** function.[379)] The implementation shall behave as if no library function calls the **strtok** function.

Returns

7 The **strtok** function returns a pointer to the first character of a token, or a null pointer if there is no token.

8 **EXAMPLE**

```
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");   // t points to the token "a"
t = strtok(NULL, ",");  // t points to the token "??b"
t = strtok(NULL, "#,"); // t points to the token "c"
t = strtok(NULL, "?");  // t is a null pointer
```

Forward references: The **strtok_s** function (K.3.7.3.1).

7.26.6 Miscellaneous functions

7.26.6.1 The memset function

Synopsis

1
```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Description

2 The **memset** function copies the value of c (converted to an **unsigned char**) into each of the first n characters of the object pointed to by s.

Returns

3 The **memset** function returns the value of s.

7.26.6.2 The memset_explicit function

Synopsis

1
```
#include <string.h>
void *memset_explicit(void *s, int c, size_t n);
```

Description

2 The **memset_explicit** function copies the value of c (converted to an **unsigned char**) into each of the first n characters of the object pointed to by s. The purpose of this function is to make sensitive information stored in the object inaccessible[380)].

Returns

3 The **memset_explicit** function returns the value of s.

7.26.6.3 The strerror function

Synopsis

1
```
#include <string.h>
char *strerror(int errnum);
```

[379)] The **strtok_s** function can be used instead to avoid data races.

[380)] The intention is that the memory store is always performed (i.e., never elided), regardless of optimizations. This is in contrast to calls to the **memset** function (7.26.6.1)

Description

2 The **strerror** function maps the number in errnum to a message string. Typically, the values for errnum come from **errno**, but **strerror** shall map any value of type **int** to a message.

3 The **strerror** function is not required to avoid data races with other calls to the **strerror** function.[381] The implementation shall behave as if no library function calls the **strerror** function.

Returns

4 The **strerror** function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program. The behavior is undefined if the returned value is used after a subsequent call to the strerror function, or after the thread which called the function to obtain the returned value has exited.

Forward references: The **strerror_s** function (K.3.7.4.2).

7.26.6.4 The strlen function

Synopsis

1
```
#include <string.h>
size_t strlen(const char *s);
```

Description

2 The **strlen** function computes the length of the string pointed to by s.

Returns

3 The **strlen** function returns the number of characters that precede the terminating null character.

[381]The **strerror_s** function can be used instead to avoid data races.

7.27 Type-generic math `<tgmath.h>`

1 The header `<tgmath.h>` includes the headers `<math.h>` and `<complex.h>` and defines several type-generic macros.

2 The feature test macro **`__STDC_VERSION_TGMATH_H__`** expands to the token *`202311L`*.

3 This clause specifies a many-to-one correspondence of functions in `<math.h>` and `<complex.h>` with *type-generic macros*.[382] Use of a type-generic macro invokes a corresponding function whose type is determined by the types of the arguments for particular parameters called the *generic parameters*.[383]

4 Of the `<math.h>` and `<complex.h>` functions without an **`f`** (**`float`**) or **`l`** (**`long double`**) suffix, several have one or more parameters whose corresponding real type is **`double`**. For each such function, except the functions that round result to narrower type (7.12.14) (which are covered below) and **`modf`**, there is a corresponding type-generic macro. The parameters whose corresponding real type is **`double`** in the function synopsis are generic parameters.

5 Some of the `<math.h>` functions for decimal floating types have no unsuffixed counterpart. Of these functions with a **`d64`** suffix, some have one or more parameters whose type is **`_Decimal64`**. For each such function, except **`decodedecd64`**, **`encodedecd64`**, **`decodebind64`**, and **`encodebind64`**, there is a corresponding type-generic macro. The parameters whose real type is **`_Decimal64`** in the function synopsis are generic parameters.

6 If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type, the behavior is undefined.

7 Except for the macros for functions that round result to a narrower type (7.12.14), use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the types of the arguments for the generic parameters as follows:

— Arguments of integer type are regarded as having type **`_Decimal64`** if any argument has decimal floating type, and as having type **`double`** otherwise.

— If the function has exactly one generic parameter, the type determined is the corresponding real type of the argument for the generic parameter.

— If the function has exactly two generic parameters, the type determined is the corresponding real type determined by the usual arithmetic conversions (6.3.1.8) applied to the arguments for the generic parameters.

— If the function has more than two generic parameters, the type determined is the corresponding real type determined by repeatedly applying the usual arithmetic conversions, first to the first two arguments for generic parameters, then to that result type and the next argument for a generic parameter, and so forth until the usual arithmetic conversions have been applied to the last argument for a generic parameter.

If neither `<math.h>` and `<complex.h>` define a function whose generic parameters have the determined corresponding real type, the behavior is undefined.

8 For each unsuffixed function in `<math.h>` for which there is a function in `<complex.h>` with the same name except for a **`c`** prefix, the corresponding type-generic macro (for both functions) has the same name as the function in `<math.h>`. The corresponding type-generic macro for **`fabs`** and **`cabs`** is **`fabs`**.

[382] Like other function-like macros in standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

[383] If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

<math.h> function	<complex.h> function	type-generic macro
acos	cacos	acos
asin	casin	asin
atan	catan	atan
acosh	cacosh	acosh
asinh	casinh	asinh
atanh	catanh	atanh
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
cosh	ccosh	cosh
sinh	csinh	sinh
tanh	ctanh	tanh
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

9 For each unsuffixed function in <math.h> without a **c**-prefixed counterpart in <complex.h> (except functions that round result to narrower type, **modf**, and **canonicalize**), the corresponding type-generic macro has the same name as the function. These type-generic macros are:

acospi	exp2	fmod	log2	rootn
asinpi	expm1	frexp	logb	roundeven
atan2pi	fdim	fromfpx	logp1	round
atan2	floor	fromfp	lrint	rsqrt
atanpi	fmax	hypot	lround	scalbln
cbrt	fmaximum	ilogb	nearbyint	scalbn
ceil	fmaximum_mag	ldexp	nextafter	sinpi
compoundn	fmaximum_num	lgamma	nextdown	tanpi
copysign	fmaximum_mag_num	llogb	nexttoward	tgamma
cospi	fma	llrint	nextup	trunc
erfc	fmin	llround	pown	ufromfpx
erf	fminimum	log10p1	powr	ufromfp
exp10m1	fminimum_mag	log10	remainder	
exp10	fminimum_num	log1p	remquo	
exp2m1	fminimum_mag_num	log2p1	rint	

If all arguments for generic parameters are real, then use of the macro invokes a real function (provided <math.h> defines a function of the determined type); otherwise, use of the macro is undefined.

10 For each unsuffixed function in <complex.h> that is not a **c**-prefixed counterpart to a function in <math.h>, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

carg	cimag	conj	cproj	creal

Use of the macro with any argument of standard floating or complex type invokes a complex function. Use of the macro with an argument of decimal floating type is undefined.

11 The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with **f** or **d** prefix are:

fadd	**fsub**	**fmul**	**fdiv**	**ffma**	**fsqrt**
dadd	**dsub**	**dmul**	**ddiv**	**dfma**	**dsqrt**

and the macros with **d32** or **d64** prefix are:

d32add	**d32sub**	**d32mul**	**d32div**	**d32fma**	**d32sqrt**
d64add	**d64sub**	**d64mul**	**d64div**	**d64fma**	**d64sqrt**

All arguments shall be real. If the macro prefix is **f** or **d**, use of an argument of decimal floating type is undefined. If the macro prefix is **d32** or **d64**, use of an argument of standard floating type is undefined. The function invoked is determined as follows:

— If any argument has type **_Decimal128**, or if the macro prefix is **d64**, the function invoked has the name of the macro, with a **d128** suffix.

— Otherwise, if the macro prefix is **d32**, the function invoked has the name of the macro, with a **d64** suffix.

— Otherwise, if any argument has type **long double**, or if the macro prefix is **d**, the function invoked has the name of the macro, with an **l** suffix.

— Otherwise, the function invoked has the name of the macro (with no suffix).

12 For each **d64**-suffixed function in `<math.h>`, except **decodedecd64**, **encodedecd64**, **decodebind64**, and **encodebind64**, that does not have an unsuffixed counterpart, the corresponding type-generic macro has the name of the function, but without the suffix. These type-generic macros are:

`<math.h>` function	type-generic macro
quantized*N*	**quantize**
samequantumd*N*	**samequantum**
quantumd*N*	**quantum**
llquantexpd*N*	**llquantexp**

Use of the macro with an argument of standard floating or complex type or with only integer type arguments is undefined.

13 A type-generic macro corresponding to a function indicated in the table in 7.6.2 is affected by constant rounding modes (7.6.4).

14 **NOTE 1** The type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows:

```
#define cbrt(X)                         \
    _Generic((X),                       \
       long double: _Roundwise_cbrtl, \
       default:     _Roundwise_cbrt,  \
       float:       _Roundwise_cbrtf  \
    )(X)
```

where `_Roundwise_cbrtl`, `_Roundwise_cbrt`, and `_Roundwise_cbrtf` are pointers to functions that are equivalent to **cbrtl**, **cbrt**, and **cbrtf**, respectively, but that are guaranteed to be affected by constant rounding modes (7.6.2).

15 **EXAMPLE** With the declarations

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32;
_Decimal64 d64;
_Decimal128 d128;
#endif
```

functions invoked by use of type-generic macros are shown in the following table:

macro use	invocation
`exp(n)`	`exp(n)`, the function
`acosh(f)`	`acoshf(f)`
`sin(d)`	`sin(d)`, the function
`atan(ld)`	`atanl(ld)`
`log(fc)`	`clogf(fc)`
`sqrt(dc)`	`csqrt(dc)`
`pow(ldc, f)`	`cpowl(ldc, f)`
`remainder(n, n)`	`remainder(n, n)`, the function
`nextafter(d, f)`	`nextafter(d, f)`, the function
`nexttoward(f, ld)`	`nexttowardf(f, ld)`
`copysign(n, ld)`	`copysignl(n, ld)`
`ceil(fc)`	undefined
`rint(dc)`	undefined
`fmaximum(ldc, ld)`	undefined
`carg(n)`	`carg(n)`, the function
`cproj(f)`	`cprojf(f)`
`creal(d)`	`creal(d)`, the function
`cimag(ld)`	`cimagl(ld)`
`fabs(fc)`	`cabsf(fc)`
`carg(dc)`	`carg(dc)`, the function
`cproj(ldc)`	`cprojl(ldc)`
`fsub(f, ld)`	`fsubl(f, ld)`
`fdiv(d, n)`	`fdiv(d, n)`, the function
`dfma(f, d, ld)`	`dfmal(f, d, ld)`
`dadd(f, f)`	`daddl(f, f)`
`dsqrt(dc)`	undefined
`exp(d64)`	`expd64(d64)`
`sqrt(d32)`	`sqrtd32(d32)`
`fmaximum(d64, d128)`	`fmaximumd128(d64, d128)`
`pow(d32, n)`	`powd64(d32, n)`
`remainder(d64, d)`	undefined
`creal(d64)`	undefined
`remquo(d32, d32, &n)`	undefined
`llquantexp(d)`	undefined
`quantize(dc)`	undefined
`samequantum(n, n)`	undefined
`d32sub(d32, d128)`	`d32subd128(d32, d128)`
`d32div(d64, n)`	`d32divd64(d64, n)`

`d64fma(d32, d64, d128)`	`d64fmad128(d32, d64, d128)`
`d64add(d32, d32)`	`d64addd128(d32, d32)`
`d64sqrt(d)`	undefined
`dadd(n, d64)`	undefined

7.28 Threads <threads.h>

7.28.1 Introduction

1 The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution[384].

2 Implementations that define the macro __STDC_NO_THREADS__ need not provide this header nor support any of its facilities.

3 The macros are

```
ONCE_FLAG_INIT
```

which expands to a value that can be used to initialize an object of type once_flag; and

```
TSS_DTOR_ITERATIONS
```

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

4 The types are

```
cnd_t
```

which is a complete object type that holds an identifier for a condition variable;

```
thrd_t
```

which is a complete object type that holds an identifier for a thread;

```
tss_t
```

which is a complete object type that holds an identifier for a thread-specific storage pointer;

```
mtx_t
```

which is a complete object type that holds an identifier for a mutex;

```
tss_dtor_t
```

which is the function pointer type void (*)(void*), used for a destructor for a thread-specific storage pointer;

```
thrd_start_t
```

which is the function pointer type int (*)(void*) that is passed to thrd_create to create a new thread; and

```
once_flag
```

which is a complete object type that holds a flag for use by call_once.

5 The enumeration constants are

```
mtx_plain
```

which is passed to mtx_init to create a mutex object that does not support timeout;

```
mtx_recursive
```

[384] See "future library directions" (7.33.19).

which is passed to **mtx_init** to create a mutex object that supports recursive locking;

```
mtx_timed
```

which is passed to **mtx_init** to create a mutex object that supports timeout;

```
thrd_timedout
```

which is returned by a timed wait function to indicate that the time specified in the call was reached without acquiring the requested resource;

```
thrd_success
```

which is returned by a function to indicate that the requested operation succeeded;

```
thrd_busy
```

which is returned by a function to indicate that the requested operation failed because a resource requested by a test and return function is already in use;

```
thrd_error
```

which is returned by a function to indicate that the requested operation failed; and

```
thrd_nomem
```

which is returned by a function to indicate that the requested operation failed because it was unable to allocate memory.

Forward references: date and time (7.29).

7.28.2 Initialization functions

7.28.2.1 The call_once function

Synopsis

1
```
#include <threads.h>
void call_once(once_flag *flag, void (*func)(void));
```

Description

2 The **call_once** function uses the **once_flag** pointed to by `flag` to ensure that `func` is called exactly once, the first time the **call_once** function is called with that value of `flag`. Completion of an effective call to the **call_once** function synchronizes with all subsequent calls to the **call_once** function with the same value of `flag`.

Returns

3 The **call_once** function returns no value.

7.28.3 Condition variable functions

7.28.3.1 The cnd_broadcast function

Synopsis

1
```
#include <threads.h>
int cnd_broadcast(cnd_t *cond);
```

Description

2 The **cnd_broadcast** function unblocks all the threads that are blocked on the condition variable pointed to by cond at the time of the call. If no threads are blocked on the condition variable pointed to by cond at the time of the call, the function does nothing.

Returns

3 The **cnd_broadcast** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.28.3.2 The cnd_destroy function

Synopsis

1
```
#include <threads.h>
void cnd_destroy(cnd_t *cond);
```

Description

2 The **cnd_destroy** function releases all resources used by the condition variable pointed to by cond. The **cnd_destroy** function requires that no threads be blocked waiting for the condition variable pointed to by cond.

Returns

3 The **cnd_destroy** function returns no value.

7.28.3.3 The cnd_init function

Synopsis

1
```
#include <threads.h>
int cnd_init(cnd_t *cond);
```

Description

2 The **cnd_init** function creates a condition variable. If it succeeds it sets the variable pointed to by cond to a value that uniquely identifies the newly created condition variable. A thread that calls **cnd_wait** on a newly created condition variable will block.

Returns

3 The **cnd_init** function returns **thrd_success** on success, or **thrd_nomem** if no memory could be allocated for the newly created condition, or **thrd_error** if the request could not be honored.

7.28.3.4 The cnd_signal function

Synopsis

1
```
#include <threads.h>
int cnd_signal(cnd_t *cond);
```

Description

2 The **cnd_signal** function unblocks one of the threads that are blocked on the condition variable pointed to by cond at the time of the call. If no threads are blocked on the condition variable at the time of the call, the function does nothing and returns success.

Returns

3 The **cnd_signal** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.28.3.5 The cnd_timedwait function

Synopsis

1
```
#include <threads.h>
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
      const struct timespec *restrict ts);
```

Description

2 The **cnd_timedwait** function atomically unlocks the mutex pointed to by `mtx` and blocks until the condition variable pointed to by `cond` is signaled by a call to **cnd_signal** or to **cnd_broadcast**, or until after the **TIME_UTC**-based calendar time pointed to by `ts`, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the variable pointed to by `mtx` before it returns. The **cnd_timedwait** function requires that the mutex pointed to by `mtx` be locked by the calling thread.

Returns

3 The **cnd_timedwait** function returns **thrd_success** upon success, or **thrd_timedout** if the time specified in the call was reached without acquiring the requested resource, or **thrd_error** if the request could not be honored.

7.28.3.6 The cnd_wait function

Synopsis

1
```
#include <threads.h>
int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Description

2 The **cnd_wait** function atomically unlocks the mutex pointed to by `mtx` and blocks until the condition variable pointed to by `cond` is signaled by a call to **cnd_signal** or to **cnd_broadcast**, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the mutex pointed to by `mtx` before it returns. The **cnd_wait** function requires that the mutex pointed to by `mtx` be locked by the calling thread.

Returns

3 The **cnd_wait** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.28.4 Mutex functions

1 For purposes of determining the existence of a data race, lock and unlock operations behave as atomic operations. All lock and unlock operations on a particular mutex occur in some particular total order.

2 **NOTE 1** This total order can be viewed as the modification order of the mutex.

7.28.4.1 The mtx_destroy function

Synopsis

1
```
#include <threads.h>
void mtx_destroy(mtx_t *mtx);
```

Description

2 The **mtx_destroy** function releases any resources used by the mutex pointed to by `mtx`. No threads can be blocked waiting for the mutex pointed to by `mtx`.

Returns

3 The **mtx_destroy** function returns no value.

7.28.4.2 The mtx_init function

Synopsis

1
```
#include <threads.h>
int mtx_init(mtx_t *mtx, int type);
```

Description

2 The **mtx_init** function creates a mutex object with properties indicated by type, which shall have one of these values:

mtx_plain for a simple non-recursive mutex,

mtx_timed for a non-recursive mutex that supports timeout,

mtx_plain | mtx_recursive for a simple recursive mutex, or

mtx_timed | mtx_recursive for a recursive mutex that supports timeout.

3 If the **mtx_init** function succeeds, it sets the mutex pointed to by mtx to a value that uniquely identifies the newly created mutex.

Returns

4 The **mtx_init** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.28.4.3 The mtx_lock function

Synopsis

1
```
#include <threads.h>
int mtx_lock(mtx_t *mtx);
```

Description

2 The **mtx_lock** function blocks until it locks the mutex pointed to by mtx. If the mutex is non-recursive, it shall not be locked by the calling thread. Prior calls to **mtx_unlock** on the same mutex synchronize with this operation.

Returns

3 The **mtx_lock** function returns **thrd_success** on success, or **thrd_error** if the request could not be honored.

7.28.4.4 The mtx_timedlock function

Synopsis

1
```
#include <threads.h>
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
```

Description

2 The **mtx_timedlock** function endeavors to block until it locks the mutex pointed to by mtx or until after the **TIME_UTC**-based calendar time pointed to by ts. The specified mutex shall support timeout. If the operation succeeds, prior calls to **mtx_unlock** on the same mutex synchronize with this operation.

Returns

3 The **mtx_timedlock** function returns **thrd_success** on success, or **thrd_timedout** if the time specified was reached without acquiring the requested resource, or **thrd_error** if the request could not be honored.

7.28.4.5 The mtx_trylock function

Synopsis

1
```
#include <threads.h>
int mtx_trylock(mtx_t *mtx);
```

Description

2 The `mtx_trylock` function endeavors to lock the mutex pointed to by `mtx`. If the mutex is already locked, the function returns without blocking. If the operation succeeds, prior calls to `mtx_unlock` on the same mutex synchronize with this operation.

Returns

3 The `mtx_trylock` function returns `thrd_success` on success, or `thrd_busy` if the resource requested is already in use, or `thrd_error` if the request could not be honored. `mtx_trylock` may spuriously fail to lock an unused resource, in which case it returns `thrd_busy`.

7.28.4.6 The `mtx_unlock` function

Synopsis

1
```
#include <threads.h>
int mtx_unlock(mtx_t *mtx);
```

Description

2 The `mtx_unlock` function unlocks the mutex pointed to by `mtx`. The mutex pointed to by `mtx` shall be locked by the calling thread.

Returns

3 The `mtx_unlock` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

7.28.5 Thread functions

7.28.5.1 The `thrd_create` function

Synopsis

1
```
#include <threads.h>
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Description

2 The `thrd_create` function creates a new thread executing `func(arg)`. If the `thrd_create` function succeeds, it sets the object pointed to by `thr` to the identifier of the newly created thread. (A thread's identifier may be reused for a different thread once the original thread has exited and either been detached or joined to another thread.) The completion of the `thrd_create` function synchronizes with the beginning of the execution of the new thread.

3 Returning from `func` has the same behavior as invoking `thrd_exit` with the value returned from `func`.

Returns

4 The `thrd_create` function returns `thrd_success` on success, or `thrd_nomem` if no memory could be allocated for the thread requested, or `thrd_error` if the request could not be honored.

7.28.5.2 The `thrd_current` function

Synopsis

1
```
#include <threads.h>
thrd_t thrd_current(void);
```

Description

2 The `thrd_current` function identifies the thread that called it.

Returns

3 The `thrd_current` function returns the identifier of the thread that called it.

7.28.5.3 The thrd_detach function

Synopsis

1
```
#include <threads.h>
int thrd_detach(thrd_t thr);
```

Description

2 The **thrd_detach** function tells the operating system to dispose of any resources allocated to the thread identified by thr when that thread terminates. The thread identified by thr shall not have been previously detached or joined with another thread.

Returns

3 The **thrd_detach** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.28.5.4 The thrd_equal function

Synopsis

1
```
#include <threads.h>
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

2 The **thrd_equal** function will determine whether the thread identified by thr0 refers to the thread identified by thr1.

Returns

3 The **thrd_equal** function returns zero if the thread thr0 and the thread thr1 refer to different threads. Otherwise the **thrd_equal** function returns a nonzero value.

7.28.5.5 The thrd_exit function

Synopsis

1
```
#include <threads.h>
[[noreturn]] void thrd_exit(int res);
```

Description

2 For every thread-specific storage key which was created with a non-null destructor and for which the value is non-null, **thrd_exit** sets the value associated with the key to a null pointer value and then invokes the destructor with its previous value. The order in which destructors are invoked is unspecified.

3 If after this process there remain keys with both non-null destructors and values, the implementation repeats this process up to **TSS_DTOR_ITERATIONS** times.

4 Following this, the **thrd_exit** function terminates execution of the calling thread and sets its result code to res.

5 The program terminates normally after the last thread has been terminated. The behavior is as if the program called the **exit** function with the status **EXIT_SUCCESS** at thread termination time.

Returns

6 The **thrd_exit** function returns no value.

7.28.5.6 The thrd_join function

Synopsis

1
```
#include <threads.h>
int thrd_join(thrd_t thr, int *res);
```

Description

2 The **thrd_join** function joins the thread identified by `thr` with the current thread by blocking until the other thread has terminated. If the parameter `res` is not a null pointer, it stores the thread's result code in the integer pointed to by `res`. The termination of the other thread synchronizes with the completion of the **thrd_join** function. The thread identified by `thr` shall not have been previously detached or joined with another thread.

Returns

3 The **thrd_join** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.28.5.7 The thrd_sleep function

Synopsis

1
```
#include <threads.h>
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
```

Description

2 The **thrd_sleep** function suspends execution of the calling thread until either the interval specified by `duration` has elapsed or a signal which is not being ignored is received. If interrupted by a signal and the `remaining` argument is not null, the amount of time remaining (the requested interval minus the time actually slept) is stored in the interval it points to. The `duration` and `remaining` arguments may point to the same object.

3 The suspension time may be longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than that specified, as measured by the system clock **TIME_UTC**.

Returns

4 The **thrd_sleep** function returns zero if the requested time has elapsed, -1 if it has been interrupted by a signal, or a negative value (which may also be -1) if it fails.

7.28.5.8 The thrd_yield function

Synopsis

1
```
#include <threads.h>
void thrd_yield(void);
```

Description

2 The **thrd_yield** function endeavors to permit other threads to run, even if the current thread would ordinarily continue to run.

Returns

3 The **thrd_yield** function returns no value.

7.28.6 Thread-specific storage functions

7.28.6.1 The tss_create function

Synopsis

1
```
#include <threads.h>
int tss_create(tss_t *key, tss_dtor_t dtor);
```

Description

2 The **tss_create** function creates a thread-specific storage pointer with destructor `dtor`, which may be null.

3 A null pointer value is associated with the newly created key in all existing threads. Upon subsequent thread creation, the value associated with all keys is initialized to a null pointer value in the new thread.

4 Destructors associated with thread-specific storage are not invoked at program termination.

5 The **tss_create** function shall not be called from within a destructor.

Returns

6 If the **tss_create** function is successful, it sets the thread-specific storage pointed to by key to a value that uniquely identifies the newly created pointer and returns **thrd_success**; otherwise, **thrd_error** is returned and the thread-specific storage pointed to by key is set to an indeterminate representation.

7.28.6.2 The tss_delete function

Synopsis

1
```
#include <threads.h>
void tss_delete(tss_t key);
```

Description

2 The **tss_delete** function releases any resources used by the thread-specific storage identified by key. The **tss_delete** function shall only be called with a value for key that was returned by a call to **tss_create** before the thread commenced executing destructors.

3 If **tss_delete** is called while another thread is executing destructors, whether this will affect the number of invocations of the destructor associated with key on that thread is unspecified.

4 Calling **tss_delete** will not result in the invocation of any destructors.

Returns

5 The **tss_delete** function returns no value.

7.28.6.3 The tss_get function

Synopsis

1
```
#include <threads.h>
void *tss_get(tss_t key);
```

Description

2 The **tss_get** function returns the value for the current thread held in the thread-specific storage identified by key. The **tss_get** function shall only be called with a value for key that was returned by a call to **tss_create** before the thread commenced executing destructors.

Returns

3 The **tss_get** function returns the value for the current thread if successful, or zero if unsuccessful.

7.28.6.4 The tss_set function

Synopsis

1
```
#include <threads.h>
int tss_set(tss_t key, void *val);
```

Description

2 The **tss_set** function sets the value for the current thread held in the thread-specific storage identified by key to val. The **tss_set** function shall only be called with a value for key that was returned by a call to **tss_create** before the thread commenced executing destructors.

3 This action will not invoke the destructor associated with the key on the value being replaced.

Returns

4 The **tss_set** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.29 Date and time <time.h>

7.29.1 Components of time

1 The header <time.h> defines several macros, and declares types and functions for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

2 The feature test macro `__STDC_VERSION_TIME_H__` expands to the token `202311L`. The other macros defined are `NULL` (described in 7.21);

```
CLOCKS_PER_SEC
```

which expands to an expression with type `clock_t` (described below) that is the number per second of the value returned by the `clock` function;

```
TIME_UTC
TIME_MONOTONIC
```

which expand to integer constants greater than 0 designating the calendar time and monotonic time bases, respectively. Additional time base macro definitions, beginning with `TIME_` and an uppercase letter, may also be specified by the implementation[385]; and,

```
TIME_ACTIVE
TIME_THREAD_ACTIVE
```

which, if defined, expand to integer values, designating overall execution and thread-specific active processing time bases, respectively.

3 The definition of macros for time bases other than `TIME_UTC` are optional. If defined, the corresponding time bases are supported by `timespec_get` and `timespec_getres`, and their values are positive. If defined, the value of the optional macro `TIME_ACTIVE` shall be different from the constants `TIME_UTC` and `TIME_MONOTONIC` and shall not change during the same program invocation. The optional macro `TIME_THREAD_ACTIVE` shall not be defined if the implementation does not support threads; its value shall be different from `TIME_UTC`, `TIME_MONOTONIC`, and `TIME_ACTIVE`, it shall be the same for all expansions of the macro for the same thread, and the value provided for one thread shall not be used by a different thread as the `base` argument of `timespec_get` or `timespec_getres`.

4 The types declared are `size_t` (described in 7.21);

```
clock_t
```

and

```
time_t
```

which are real types capable of representing times;

```
struct timespec
```

which holds an interval specified in seconds and nanoseconds (which may represent a calendar time based on a particular epoch); and

```
struct tm
```

[385]See future library directions (7.33). Implementations can define additional time bases, but are only required to support a real time clock based on UTC.

which holds the components of a calendar time, called the *broken-down time*.

5 The range and precision of times representable in **clock_t** and **time_t** are implementation-defined. The **timespec** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[386)]

```
time_t tv_sec;  // whole seconds -- ≥ 0
long   tv_nsec; // nanoseconds -- [0, 999999999]
```

The **tm** structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.[387)]

```
int tm_sec;   // seconds after the minute -- [0, 60]
int tm_min;   // minutes after the hour -- [0, 59]
int tm_hour;  // hours since midnight -- [0, 23]
int tm_mday;  // day of the month -- [1, 31]
int tm_mon;   // months since January -- [0, 11]
int tm_year;  // years since 1900
int tm_wday;  // days since Sunday -- [0, 6]
int tm_yday;  // days since January 1 -- [0, 365]
int tm_isdst; // Daylight Saving Time flag
```

The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

7.29.2 Time manipulation functions

7.29.2.1 The clock function

Synopsis

1
```
#include <time.h>
clock_t clock(void);
```

Description

2 The **clock** function determines the processor time used.

Returns

3 The **clock** function returns the implementation's best approximation of the active processing time associated with the program execution since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock** function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available, the function returns the value **(clock_t)**(-1). If the value cannot be represented, the function returns an unspecified value[388)].

7.29.2.2 The difftime function

Synopsis

1
```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Description

2 The **difftime** function computes the difference between two calendar times: `time1 - time0`.

Returns

3 The **difftime** function returns the difference expressed in seconds as a **double**.

[386)] The **tv_sec** member is a linear count of seconds and might not have the normal semantics of a **time_t**.

[387)] The range [0, 60] for **tm_sec** allows for a positive leap second.

[388)] This could be due to overflow of the **clock_t** type.

7.29.2.3 The mktime function

Synopsis

1
```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description

2 The mktime function converts the broken-down time, expressed as local time, in the structure pointed to by timeptr into a calendar time value with the same encoding as that of the values returned by the time function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. [389] On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of tm_mday is not set until tm_mon and tm_year are determined.

Returns

3 The mktime function returns the specified calendar time encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value (time_t)(-1).

4 **EXAMPLE** What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>
static const char *const wday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/* ... */

time_str.tm_year   = 2001 - 1900;
time_str.tm_mon    = 7 - 1;
time_str.tm_mday   = 4;
time_str.tm_hour   = 0;
time_str.tm_min    = 0;
time_str.tm_sec    = 1;
time_str.tm_isdst  = -1;
if (mktime(&time_str) == (time_t)(-1))
      time_str.tm_wday = 7;
printf("%s\n", wday[time_str.tm_wday]);
```

7.29.2.4 The timegm function

Synopsis

1
```
#include <time.h>
time_t timegm(struct tm *timeptr);
```

Description

2 The timegm function converts the broken-down time, expressed as UTC time, in the structure pointed to by timeptr into a calendar time value with the same encoding as that of the values returned by the time function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified

[389]Thus, a positive or zero value for tm_isdst causes the mktime function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value causes it to attempt to determine whether Daylight Saving Time is in effect for the specified time.

calendar time, but with their values forced to the ranges indicated above; the final value of `tm_mday` is not set until `tm_mon` and `tm_year` are determined.

Returns

3 The `timegm` function returns the specified calendar time encoded as a value of type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t)(-1)`.

7.29.2.5 The `time` function

Synopsis

1
```
#include <time.h>
time_t time(time_t *timer);
```

Description

2 The `time` function determines the current calendar time. The encoding of the value is unspecified.

Returns

3 The `time` function returns the implementation's best approximation to the current calendar time. The value `(time_t)`(-1) is returned if the calendar time is not available. If `timer` is not a null pointer, the return value is also assigned to the object it points to.

7.29.2.6 The `timespec_get` function

Synopsis

1
```
#include <time.h>
int timespec_get(struct timespec *ts, int base);
```

Description

2 The `timespec_get` function sets the interval pointed to by `ts` to hold the current calendar time based on the specified time base.

3 If `base` is `TIME_UTC`, the `tv_sec` member is set to the number of seconds since an implementation-defined *epoch*, truncated to a whole value and the `tv_nsec` member is set to the integral number of nanoseconds, rounded to the resolution of the system clock.[390] The optional time base `TIME_MONOTONIC` is the same, but the reference point is an implementation-defined time point; different program invocations need not refer to the same reference points.[391] For the same program invocation, the results of two calls to `timespec_get` with `TIME_MONOTONIC` such that the first happens before the second shall not be decreasing. It is implementation-defined if `TIME_MONOTONIC` accounts for time during which the execution environment is suspended.[392] For the optional time bases `TIME_ACTIVE` and `TIME_THREAD_ACTIVE` the result is similar, but the call measures the amount of active processing time associated with the whole program invocation or with the calling thread, respectively.

Returns

4 If the `timespec_get` function is successful it returns the nonzero value `base`; otherwise, it returns zero.

Recommended practice

5 It is recommended practice that timing results of calls to `timespec_get` with `TIME_ACTIVE`, if defined, and of calls to `clock` are as close to each other as their types, value ranges, and resolutions (obtained with `timespec_getres` and `CLOCKS_PER_SEC`, respectively) allow. Because of its wider value range and improved indications on error, `timespec_get` with time base `TIME_ACTIVE` should be used instead of `clock` by new code whenever possible.

---

[390] Although a `struct timespec` object describes times with nanosecond resolution, the available resolution is system dependent and could even be greater than 1 second.

[391] Commonly, this reference point is the boot time of the execution environment or the start of the execution.

[392] The execution environment may, for example, not be able to track physical time that elapsed during suspension in a low power consumption mode.

7.29.2.7 The timespec_getres function

Synopsis

1
```
#include <time.h>
int timespec_getres(struct timespec *ts, int base);
```

Description

2 If ts is non-null and base is supported by the timespec_get function, the timespec_getres function returns the resolution of the time provided by the timespec_get function for base in the timespec structure pointed to by ts. For each supported base, multiple calls to the timespec_getres function during the same program execution shall have identical results.

Returns

3 If the value base is supported by the timespec_get function, the timespec_getres function returns the nonzero value base; otherwise, it returns zero.

7.29.3 Time conversion functions

1 Functions with a _r suffix place the result of the conversion into the buffer referred by buf and return that pointer. These functions and the function strftime shall not be subject to data races, unless the time or calendar state is changed in a multi-thread execution.[393)]

2 Functions asctime, ctime, gmtime, and localtime are the same as their counterparts suffixed with _r. In place of the parameter buf, these functions use a pointer to an object and return it: one or two broken-down time structures (for gmtime and localtime) or an array of char (commonly used by asctime and ctime). Execution of any of the functions that return a pointer to one of these objects may overwrite the information returned from any previous call to one of these functions that uses the same object. These functions are not reentrant and are not required to avoid data races with each other. Accessing the returned pointer after the thread that called the function that returned it has exited results in undefined behavior. The implementation shall behave as if no other library functions call these functions.

7.29.3.1 The asctime function

Synopsis

1
```
#include <time.h>
[[deprecated]] char *asctime(const struct tm *timeptr);
```

Description

2 This function is obsolescent and should be avoided in new code.

3 The asctime function converts the broken-down time in the structure pointed to by timeptr into a string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
[[deprecated]] char *asctime(const struct tm *timeptr)
{
      static const char wday_name[7][3] = {
            "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
      };
      static const char mon_name[12][3] = {
            "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
```

393)This does not mean that these functions may not read global state that describes the time and calendar settings of the execution, such as the LC_TIME locale or the implementation-defined specification of the local time zone. Only the setting of that state by setlocale or by means of implementation-defined functions may constitute races.

```
    };
    static char result[26];

    snprintf(result, 26, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

4 If any of the members of the broken-down time contain values that are outside their normal ranges[394], the behavior of the **asctime** function is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined.

Returns

5 The **asctime** function returns a pointer to the string.

7.29.3.2 The ctime function

Synopsis

1
```
#include <time.h>
[[deprecated]] char *ctime(const time_t *timer);
```

Description

2 This function is obsolescent and should be avoided in new code.

3 The **ctime** function converts the calendar time pointed to by `timer` to local time in the form of a string. They are equivalent to:

```
asctime(localtime(timer))
```

Returns

4 The **ctime** function returns the pointer returned by the **asctime** functions with that broken-down time as argument.

Forward references: the **localtime** functions (7.29.3.4).

7.29.3.3 The gmtime functions

Synopsis

1
```
#include <time.h>
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *timer, struct tm *buf);
```

Description

2 The **gmtime** functions convert the calendar time pointed to by `timer` into a broken-down time, expressed as UTC.

Returns

3 The **gmtime** functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

7.29.3.4 The localtime functions

[394]See 7.29.1.

Synopsis

1
```
#include <time.h>
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *timer, struct tm *buf);
```

Description

2 The `localtime` functions converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time.

Returns

3 The `localtime` functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

7.29.3.5 The `strftime` function

Synopsis

1
```
#include <time.h>
size_t strftime(char * restrict s, size_t maxsize, const char * restrict format,
        const struct tm * restrict timeptr);
```

Description

2 The `strftime` function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The `format` string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a `%` character, possibly followed by an `E` or `O` modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array.

3 Each conversion specifier shall be replaced by appropriate characters as described in the following list. The appropriate characters shall be determined using the `LC_TIME` category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

`%a` is replaced by the locale's abbreviated weekday name. [`tm_wday`]

`%A` is replaced by the locale's full weekday name. [`tm_wday`]

`%b` is replaced by the locale's abbreviated month name. [`tm_mon`]

`%B` is replaced by the locale's full month name. [`tm_mon`]

`%c` is replaced by the locale's appropriate date and time representation. [all specified in 7.29.1]

`%C` is replaced by the year divided by 100 and truncated to an integer, as a decimal number (`00`–`99`). [`tm_year`]

`%d` is replaced by the day of the month as a decimal number (`01`–`31`). [`tm_mday`]

`%D` is equivalent to "`%m/%d/%y`". [`tm_mon`, `tm_mday`, `tm_year`]

`%e` is replaced by the day of the month as a decimal number (`1`–`31`); a single digit is preceded by a space. [`tm_mday`]

`%F` is equivalent to "`%Y-%m-%d`" (the ISO 8601 date format). [`tm_year`, `tm_mon`, `tm_mday`]

`%g` is replaced by the last 2 digits of the week-based year (see below) as a decimal number (`00`–`99`). [`tm_year`, `tm_wday`, `tm_yday`]

`%G` is replaced by the week-based year (see below) as a decimal number (e.g., 1997). [`tm_year`, `tm_wday`, `tm_yday`]

`%h` is equivalent to "`%b`". [`tm_mon`]

`%H` is replaced by the hour (24-hour clock) as a decimal number (`00`–`23`). [**`tm_hour`**]

`%I` is replaced by the hour (12-hour clock) as a decimal number (`01`–`12`). [**`tm_hour`**]

`%j` is replaced by the day of the year as a decimal number (`001`–`366`). [**`tm_yday`**]

`%m` is replaced by the month as a decimal number (`01`–`12`). [**`tm_mon`**]

`%M` is replaced by the minute as a decimal number (`00`–`59`). [**`tm_min`**]

`%n` is replaced by a new-line character.

`%p` is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock. [**`tm_hour`**]

`%r` is replaced by the locale's 12-hour clock time. [**`tm_hour`**, **`tm_min`**, **`tm_sec`**]

`%R` is equivalent to "`%H:%M`". [**`tm_hour`**, **`tm_min`**]

`%S` is replaced by the second as a decimal number (`00`–`60`). [**`tm_sec`**]

`%t` is replaced by a horizontal-tab character.

`%T` is equivalent to "`%H:%M:%S`" (the ISO 8601 time format). [**`tm_hour`**, **`tm_min`**, **`tm_sec`**]

`%u` is replaced by the ISO 8601 weekday as a decimal number (`1`–`7`), where Monday is 1. [**`tm_wday`**]

`%U` is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (`00`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]

`%V` is replaced by the ISO 8601 week number (see below) as a decimal number (`01`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]

`%w` is replaced by the weekday as a decimal number (`0`–`6`), where Sunday is 0. [**`tm_wday`**]

`%W` is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (`00`–`53`). [**`tm_year`**, **`tm_wday`**, **`tm_yday`**]

`%x` is replaced by the locale's appropriate date representation. [all specified in 7.29.1]

`%X` is replaced by the locale's appropriate time representation. [all specified in 7.29.1]

`%y` is replaced by the last 2 digits of the year as a decimal number (`00`–`99`). [**`tm_year`**]

`%Y` is replaced by the year as a decimal number (e.g., `1997`). [**`tm_year`**]

`%z` is replaced by the offset from UTC in the ISO 8601 format "`-0430`" (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [**`tm_isdst`**]

`%Z` is replaced by the locale's time zone name or abbreviation, or by no characters if no time zone is determinable. [**`tm_isdst`**]

`%%` is replaced by `%`.

4 Some conversion specifiers can be modified by the inclusion of an `E` or `O` modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

`%Ec` is replaced by the locale's alternative date and time representation.

`%EC` is replaced by the name of the base year (period) in the locale's alternative representation.

`%Ex` is replaced by the locale's alternative date representation.

`%EX` is replaced by the locale's alternative time representation.

`%Ey` is replaced by the offset from `%EC` (year only) in the locale's alternative representation.

`%EY` is replaced by the locale's full alternative year representation.

`%Ob` is replaced by the locale's abbreviated alternative month name.

`%OB` is replaced by the locale's alternative appropriate full month name.

`%Od` is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

`%Oe` is replaced by the day of the month, using the locale's alternative numeric symbols (filled as needed with leading spaces).

`%OH` is replaced by the hour (24-hour clock), using the locale's alternative numeric symbols.

`%OI` is replaced by the hour (12-hour clock), using the locale's alternative numeric symbols.

`%Om` is replaced by the month, using the locale's alternative numeric symbols.

`%OM` is replaced by the minutes, using the locale's alternative numeric symbols.

`%OS` is replaced by the seconds, using the locale's alternative numeric symbols.

`%Ou` is replaced by the ISO 8601 weekday as a number in the locale's alternative representation, where Monday is 1.

`%OU` is replaced by the week number, using the locale's alternative numeric symbols.

`%OV` is replaced by the ISO 8601 week number, using the locale's alternative numeric symbols.

`%Ow` is replaced by the weekday as a number, using the locale's alternative numeric symbols.

`%OW` is replaced by the week number of the year, using the locale's alternative numeric symbols.

`%Oy` is replaced by the last 2 digits of the year, using the locale's alternative numeric symbols.

5 `%g`, `%G`, and `%V` give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, `%G` is replaced by `1998` and `%V` is replaced by `53`. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, `%G` is replaced by `1998` and `%V` is replaced by `01`.

6 If a conversion specifier is not one of the above, the behavior is undefined.

7 In the `"C"` locale, the `E` and `O` modifiers are ignored and the replacement strings for the following specifiers are:

`%a` the first three characters of `%A`.

`%A` one of "`Sunday`", "`Monday`", . . . , "`Saturday`".

`%b` the first three characters of `%B`.

`%B` one of "`January`", "`February`", . . . , "`December`".

`%c` equivalent to "`%a %b %e %T %Y`".

`%p` one of "`AM`" or "`PM`".

`%r` equivalent to "`%I:%M:%S %p`".

`%x` equivalent to "`%m/%d/%y`".

`%X` equivalent to `%T`.

`%Z` implementation-defined.

Returns

8 If the total number of resulting characters including the terminating null character is not more than `maxsize`, the **`strftime`** function returns the number of characters placed into the array pointed to by `s` not including the terminating null character. Otherwise, zero is returned and the members of the array have an indeterminate representation.

7.30 Unicode utilities <uchar.h>

1 The header <uchar.h> declares one macro, a few types, and several functions for manipulating Unicode characters.

2 The macro

```
__STDC_VERSION_UCHAR_H__
```

is an integer constant expression with a value equivalent to 202311L.

3 The types declared are mbstate_t (described in 7.31.1) and size_t (described in 7.21);

```
char8_t
```

which is an unsigned integer type used for 8-bit characters and is the same type as unsigned char;

```
char16_t
```

which is an unsigned integer type used for 16-bit characters and is the same type as uint_least16_t (described in 7.22.1.2); and

```
char32_t
```

which is an unsigned integer type used for 32-bit characters and is the same type as uint_least32_t (also described in 7.22.1.2).

7.30.1 Restartable multibyte/wide character conversion functions

1 These functions have a parameter, ps, of type pointer to mbstate_t that points to an object that can completely describe the current conversion state of the associated multibyte character sequence, which the functions alter as necessary. If ps is a null pointer, each function uses its own internal mbstate_t object instead, which is initialized prior to the first call to the function to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. It is implementation-defined whether the internal mbstate_t object has thread storage duration; if it has thread storage duration, it is initialized to the initial conversion state prior to the first call to the function on the new thread. The implementation behaves as if no library function calls these functions with a null pointer for ps.

2 When used in the functions in this subclause, the encoding of char8_t, char16_t, and char32_t objects, and sequences of such objects, is UTF-8, UTF-16, and UTF-32, respectively. Similarly, the encoding of char and wchar_t, and sequences of such objects, is the execution and wide execution encodings (6.2.9), respectively.

7.30.1.1 The mbrtoc8 function

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc8(char8_t * restrict pc8, const char * restrict s, size_t n,
      mbstate_t * restrict ps);
```

Description

2 If s is a null pointer, the mbrtoc8 function is equivalent to the call:

```
      mbrtoc8(NULL, "", 1, ps)
```

In this case, the values of the parameters pc8 and n are ignored.

3 If s is not a null pointer, the mbrtoc8 function function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character

is complete and valid, it determines the values of the corresponding characters and then, if `pc8` is not a null pointer, stores the value of the first (or only) such character in the object pointed to by `pc8`. Subsequent calls will store successive characters without consuming any additional input until all the characters have been stored. If the corresponding character is the null character, the resulting state described is the initial conversion state.

Returns

4 The `mbrtoc8` function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)`(-3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

`(size_t)`(-2) if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[395]

`(size_t)`(-1) if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

7.30.1.2 The `c8rtomb` function

Synopsis

1
```
#include <uchar.h>
size_t c8rtomb(char * restrict s, char8_t c8, mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `c8rtomb` function is equivalent to the call

```
c8rtomb(buf, u8'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the `c8rtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the character given or completed by `c8` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s, or stores nothing if `c8` does not represent a complete character. At most `MB_CUR_MAX` bytes are stored. If `c8` is a null character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The `c8rtomb` function returns the number of bytes stored in the array object (including any shift sequences). When `c8` is not a valid character, an encoding error occurs: the function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)`(-1); the conversion state is unspecified.

7.30.1.3 The `mbrtoc16` function

[395]When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `mbrtoc16` function is equivalent to the call:

```
mbrtoc16(NULL, "", 1, ps)
```

In this case, the values of the parameters `pc16` and `n` are ignored.

3 If `s` is not a null pointer, the `mbrtoc16` function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if `pc16` is not a null pointer, stores the value of the first (or only) such character in the object pointed to by `pc16`. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The `mbrtoc16` function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)`(-3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

`(size_t)`(-2) if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[396)]

`(size_t)`(-1) if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

7.30.1.4 The `c16rtomb` function

Synopsis

1
```
#include <uchar.h>
size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `c16rtomb` function is equivalent to the call

```
c16rtomb(buf, u'\0', ps)
```

where `buf` is an internal buffer.

[396)]When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

3 If `s` is not a null pointer, the **`c16rtomb`** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given or completed by `c16` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s, or stores nothing if `c16` does not represent a complete character. At most **`MB_CUR_MAX`** bytes are stored. If `c16` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **`c16rtomb`** function returns the number of bytes stored in the array object (including any shift sequences). When `c16` is not a valid wide character, an encoding error occurs: the function stores the value of the macro **`EILSEQ`** in **`errno`** and returns `(size_t)`(-1); the conversion state is unspecified.

7.30.1.5 The `mbrtoc32` function

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the **`mbrtoc32`** function is equivalent to the call:

```
mbrtoc32(NULL, "", 1, ps)
```

In this case, the values of the parameters `pc32` and `n` are ignored.

3 If `s` is not a null pointer, the **`mbrtoc32`** function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if `pc32` is not a null pointer, stores the value of the first (or only) such character in the object pointed to by `pc32`. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **`mbrtoc32`** function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)`(-3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

`(size_t)`(-2) if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[397)]

`(size_t)`(-1) if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **`EILSEQ`** is stored in **`errno`**, and the conversion state is unspecified.

[397)] When `n` has at least the value of the **`MB_CUR_MAX`** macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.30.1.6 The c32rtomb function

Synopsis

1
```
#include <uchar.h>
size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
```

Description

2 If s is a null pointer, the c32rtomb function is equivalent to the call

```
c32rtomb(buf, U'\0', ps)
```

where buf is an internal buffer.

3 If s is not a null pointer, the c32rtomb function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by c32 (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s. At most MB_CUR_MAX bytes are stored. If c32 is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The c32rtomb function returns the number of bytes stored in the array object (including any shift sequences). When c32 is not a valid wide character, an encoding error occurs: the function stores the value of the macro EILSEQ in errno and returns (size_t) (-1);the conversion state is unspecified.

7.31 Extended multibyte and wide character utilities <wchar.h>

7.31.1 Introduction

1 The header <wchar.h> defines five macros, and declares four data types, one tag, and many functions.[398)]

2 The macro

```
__STDC_VERSION_WCHAR_H__
```

is an integer constant expression with a value equivalent to *202311*L.

3 The types declared are **wchar_t** and **size_t** (both described in 7.21);

```
mbstate_t
```

which is a complete object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

```
wint_t
```

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **WEOF** below);[399)] and

```
struct tm
```

which is declared as an incomplete structure type (the contents are described in 7.29.1).

4 The macros defined are **NULL** (described in 7.21); **WCHAR_MIN**, **WCHAR_MAX**, and **WCHAR_WIDTH** (described in 7.22); and

```
WEOF
```

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set.[400)] It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

5 The functions declared are grouped as follows:

- Functions that perform input and output of wide characters, or multibyte characters, or both;
- Functions that provide wide string numeric conversion;
- Functions that perform general wide string manipulation;
- Functions for wide string date and time conversion; and
- Functions that provide extended capabilities for conversion between multibyte and wide character sequences.

6 Arguments to the functions in this subclause may point to arrays containing **wchar_t** values that do not correspond to members of the extended character set. Such values shall be processed according to the specified semantics, except that it is unspecified whether an encoding error occurs if such a value appears in the format string for a function in 7.31.2 or 7.31.5 and the specified semantics do not require that value to be processed by **wcrtomb**.

7 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

[398)] See "future library directions" (7.33.20).

[399)] **wchar_t** and **wint_t** can be the same integer type.

[400)] The value of the macro **WEOF** can differ from that of **EOF** and need not be negative.

7.31.2 Formatted wide character input/output functions

1 The formatted wide character input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[401]

7.31.2.1 The fwprintf function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Description

2 The **fwprintf** function writes output to the stream pointed to by stream, under control of the wide string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fwprintf** function returns when the end of the format string is encountered.

3 The format is composed of zero or more directives: ordinary wide characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the wide character %. After the %, the following appear in sequence:

— Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

— An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a nonnegative decimal integer. [402]

— An optional *precision* that gives the minimum number of digits to appear for the b, d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point wide character for a, A, e, E, f, and F conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of wide characters to be written for s conversions. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

— An optional *length modifier* that specifies the size of the argument.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag wide characters and their meanings are:

- \- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
- \+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a value with a negative sign is converted if this flag is not specified.) [403]

[401]The **fwprintf** functions perform writes to memory for the %n specifier.
[402]Note that 0 is taken as a flag, not as the beginning of a field width.
[403]The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

space If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is prefixed to the result. If the *space* and `+` flags both appear, the *space* flag is ignored.

`#` The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `b` conversion, a nonzero result has `0b` prefixed to it. For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result of converting a floating-point number always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

`0` For `b`, `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

`hh` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`signed char`** or **`unsigned char`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`signed char`** or **`unsigned char`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`signed char`** argument.

`h` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`short int`** or **`unsigned short int`** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **`short int`** or **`unsigned short int`** before printing); or that a following `n` conversion specifier applies to a pointer to a **`short int`** argument.

`l` (ell) Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long int`** or **`unsigned long int`** argument; that a following `n` conversion specifier applies to a pointer to a **`long int`** argument; that a following `c` conversion specifier applies to a **`wint_t`** argument; that a following `s` conversion specifier applies to a pointer to a **`wchar_t`** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell) Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`long long int`** or **`unsigned long long int`** argument; or that a following `n` conversion specifier applies to a pointer to a **`long long int`** argument.

`j` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **`intmax_t`** or **`uintmax_t`** argument; or that a following `n` conversion specifier applies to a pointer to an **`intmax_t`** argument.

`z` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`size_t`** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **`size_t`** argument.

`t` Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **`ptrdiff_t`** or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a **`ptrdiff_t`** argument.

`w`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an integer argument with a specific width where *N* is a positive decimal integer with no leading zeros (the argument will have been promoted according to the integer promotions, but its value shall be converted to the unpromoted type); or that a following `n` conversion specifier applies to a pointer to an integer type argument with a width of *N* bits. All minimum-width integer types (7.22.1.2) and exact-width integer types (7.22.1.1) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`wf`*N* Specifies that a following `b`, `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a fastest minimum-width integer argument with a specific width where *N* is a positive decimal integer with no leading zeros (the argument will have been promoted according to the integer promotions, but its value shall be converted to the unpromoted type); or that a following `n` conversion specifier applies to a pointer to a fastest minimum-width integer type argument with a width of *N* bits. All fastest minimum-width integer types (7.22.1.3) defined in the header `<stdint.h>` shall be supported. Other supported values of *N* are implementation-defined.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`long double`** argument.

`H` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal32`** argument.

`D` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal64`** argument.

`DD` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **`_Decimal128`** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

8 The conversion specifiers and their meanings are:

`d,i` The **`int`** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

`b,o,u,x,X` The **`unsigned int`** argument is converted to unsigned binary (b), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

`f,F` A **`double`** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **`#`** flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **`double`** argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A **`double`** argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-wchar-sequence*`)` — which style, and the meaning of any *n-wchar-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively.[404)]

`e,E` A **`double`** argument representing a floating-point number is converted in the style *[-]d.ddd*`e`*±dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point wide character and the number of digits after it is equal to the

[404)]When applied to infinite and NaN values, the `-`, `+`, and *space* flag wide characters have their usual meaning; the **`#`** and `0` flag wide characters have no effect.

precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g,G` A **double** argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of X:

if $P > X \geq -4$, the conversion is with style `f` (or `F`) and precision $P - (X + 1)$.

otherwise, the conversion is with style `e` (or `E`) and precision $P - 1$.

Finally, unless the `#` flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point wide character is removed if there is no fractional portion remaining.

A **double** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a,A` A **double** argument representing a floating-point number is converted in the style *[-]*`0x`*h.hhhh*`p`$\pm$*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character[405] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **`FLT_RADIX`** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **`FLT_RADIX`** is not a power of 2, then the precision is sufficient to distinguish[406] values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the `#` flag is not specified, no decimal-point wide character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

If an `H`, `D`, or `DD` modifier is present and the precision is missing, then for a decimal floating type argument represented by a triple of integers (s, c, q), where n is the number of significant digits in the coefficient c,

— if $-(n + 5) \leq q \leq 0$, use style `f` (or style `F` in the case of an `A` conversion specifier) with formatting precision equal to $-q$,

[405] Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries. This implementation choice affects numerical values printed with a precision P that is insufficient to represent all values exactly. Implementations with different conventions about the most significant hexadecimal digit will round at different places, affecting the numerical value of the hexadecimal result. For example, possible printed output for the code

```
#include <stdio.h>
/* ... */
double x = 123.0;
printf("%.1a", x);
```

include "`0x1.fp+6`" and "`0xf.6p+3`" whose numerical values are 124 and 123, respectively. Portable code seeking identical numerical results on different platforms should avoid precisions P that require rounding.

[406] The formatting precision P is sufficient to distinguish values of the source type if $16^P > b^p$ where b (not a power of 2) and p are the base and precision of the source type (5.2.4.2.2). A smaller P might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point wide character.

— otherwise, use style `e` (or style `E` in the case of an `A` conversion specifier) with formatting precision equal to $n - 1$, with the exceptions that if $c = 0$ then the digit-sequence in the exponent-part shall have the value q (rather than 0), and that the exponent is always expressed with the minimum number of digits required to represent its value (the exponent never contains a leading zero).

If the precision P is present (in the conversion specification) and is zero or at least as large as the precision p (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision P is present (and nonzero) and less than the precision p of the decimal floating type, the conversion first obtains an intermediate result as follows, where n is the number of significant digits in the coefficient:

— If $n \leq P$, set the intermediate result to the input.

— If $n > P$, round the input value, according to the current rounding direction for decimal floating-point operations, to P decimal digits, with unbounded exponent range, representing the result with a P-digit integer coefficient when in the form (s, c, q).

Convert the intermediate result in the manner described above for the case where the precision is missing.

`c` If no `l` length modifier is present, the **`int`** argument is converted to a wide character as if by calling **`btowc`** and the resulting wide character is written.

If an `l` length modifier is present, the **`wint_t`** argument is converted to **`wchar_t`** and written.

`s` If no `l` length modifier is present, the argument shall be a pointer to storage of character type containing a multibyte character sequence beginning in the initial shift state. Characters from the storage are converted as if by repeated calls to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted storage, the converted storage shall contain a null wide character.

If an `l` length modifier is present, the argument shall be a pointer to storage of **`wchar_t`** type. Wide characters from the storage are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the storage shall contain a null wide character.

`p` The argument shall be a pointer to **`void`** or a pointer to a character type. The value of the pointer is converted to a sequence of printing wide characters, in an implementation-defined manner.

`n` The argument shall be a pointer to signed integer whose type is specified by the length modifiers, if any, for the conversion specification, or shall be **`int`** if no length modifiers are specified for the conversion specification. The number of wide characters written to the output stream so far by this call to **`fwprintf`** is stored into the integer object pointed to by the argument. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%` A `%` wide character is written. No argument is converted. The complete conversion specification shall be `%%`.

9 If a conversion specification is invalid, the behavior is undefined.[407] **`fwprintf`** shall behave as if it uses **`va_arg`** with a type argument naming the type resulting from applying the default argument promotions to the type corresponding to the conversion specification and then converting the result of the **`va_arg`** expansion to the type corresponding to the conversion specification.[408]

[407]See "future library directions" (7.33.20).

[408]The behavior is undefined when the types differ as specified for **`va_arg`** 7.16.1.1.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For `a` and `A` conversions, if **`FLT_RADIX`** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For `a` and `A` conversions, if **`FLT_RADIX`** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

13 For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most the maximum value M of the T**`_DECIMAL_DIG`** macros (defined in `<float.h>`), then the result should be correctly rounded.[409] If the number of significant decimal digits is more than M but the source value is exactly representable with M digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having M significant digits; the value of the resultant decimal string D should satisfy $L \le D \le U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

14 An uppercase `B` format specifier is not covered by the description above, because it used to be available for extensions in previous versions of this standard.

Implementations that did not use an uppercase `B` as their own extension before are encouraged to implement it similar to conversion specifier `b` as standardized above, with the alternative form (`#B`) generating `0B` as prefix for nonzero values.

Returns

15 The **`fwprintf`** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred or if the implementation does not support a specified width length modifier.

Environmental limits

16 The number of wide characters that can be produced by any single conversion shall be at least 4095.

17 **EXAMPLE** To print a date and time in the form "Sunday, July 3, 10:02" followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
#include <wchar.h>
/* ... */
wchar_t *weekday, *month;  // pointers to wide strings
int day, hour, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
      weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));
```

Forward references: the **`btowc`** function (7.31.6.1.1), the **`mbrtowc`** function (7.31.6.3.2).

7.31.2.2 The `fwscanf` function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
```

[409] For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

Description

2 The `fwscanf` function reads input from the stream pointed to by `stream`, under control of the wide string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

3 The format is composed of zero or more directives: one or more white-space wide characters, an ordinary wide character (neither `%` nor a white-space wide character), or a conversion specification. Each conversion specification is introduced by the wide character `%`. After the `%`, the following appear in sequence:

— An optional assignment-suppressing wide character `*`.

— An optional decimal integer greater than zero that specifies the maximum field width (in wide characters).

— An optional *length modifier* that specifies the size of the receiving object.

— A *conversion specifier* wide character that specifies the type of conversion to be applied.

4 The `fwscanf` function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

5 A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read. The directive never fails.

6 A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If that wide character differs from the directive,the directive fails and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive fails.

7 A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

8 Input white-space wide characters are skipped, unless the specification includes a `[`, `c`, or `n` specifier.[410)]

9 An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input wide characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[411)] The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

10 Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

11 The length modifiers and their meanings are:

---

410)These white-space wide characters are not counted against a specified field width.

411)`fwscanf` pushes back at most one input wide character onto the input stream. Therefore, some sequences that are acceptable to `wcstod`, `wcstol`, etc., are unacceptable to `fwscanf`.

hh Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long int** or **unsigned long int**; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **double**; or that a following c, s, or [conversion specifier applies to an argument with type pointer to **wchar_t**.

ll (ell-ell) Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

j Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following b, d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

w*N* Specifies that a following b, d, i, o, u, x, or X, or n conversion specifier applies to an argument which is a pointer to an integer with a specific width where *N* is a positive decimal integer with no leading zeros. All minimum-width integer types (7.22.1.2) and exact-width integer types (7.22.1.1) defined in the header <stdint.h> shall be supported. Other supported values of *N* are implementation-defined.

wf*N* Specifies that a following b, d, i, o, u, x, or X, or n conversion specifier applies to an argument which is a pointer to a fastest minimum-width integer with a specific width where *N* is a positive decimal integer with no leading zeros. All fastest minimum-width integer types (7.22.1.3) defined in the header <stdint.h> shall be supported. Other supported values of *N* are implementation-defined.

L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **long double**.

H Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **_Decimal32**.

D Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **_Decimal64**.

DD Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to **_Decimal128**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 In the following, the type of the corresponding argument for a conversion specifier shall be a pointer to a type determined by the length modifiers, if any, or specified by the conversion specifier. The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 10 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **int**.

b Matches an optionally signed binary integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 2 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **wcstol** function with the value 0 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **int**.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 8 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 10 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **wcstoul** function with the value 16 for the base argument. Unless a length modifier is specified, the corresponding argument shall be a pointer to **unsigned int**.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **wcstod** function. Unless a length modifier is specified, the corresponding argument shall be a pointer to **float**.

c Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the directive).

If no l length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to **char**, **signed char**, **unsigned char**, or **void** that points to storage large enough to accept the sequence. No null character is added.

If an l length modifier is present, the corresponding argument shall be a pointer to storage of **wchar_t** large enough to accept the sequence.No null wide character is added.

s Matches a sequence of non-white-space wide characters.

If no l length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to **char**, **signed char**, **unsigned char**, or **void** that points to storage large enough to accept the sequence and a terminating null character, which will be added automatically.

If an l length modifier is present, the corresponding argument shall be a pointer to storage of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*).

If no l length modifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to **char**, **signed char**, **unsigned char**, or **void** that points to storage large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the corresponding argument shall be a pointer that points to storage of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the `format` string, up to and including the matching right bracket (`]`). The wide characters between the brackets (the *scanlist*) compose the scanset, unless the wide character after the left bracket is a circumflex (`^`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket wide character is in the scanlist and the next following right bracket wide character is the matching right bracket that ends the specification; otherwise the first following right bracket wide character is the one that ends the specification. If a `-` wide character is in the scanlist and is not the first, nor the second where the first wide character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fwprintf` function. The corresponding argument shall be a pointer to a pointer of `void`. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n` No input is consumed. The corresponding argument shall be a pointer of a signed integer type. The number of wide characters read from the input stream so far by this call to the `fwscanf` function is stored into the integer object pointed to by the argument. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `fwscanf` function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

`%` Matches a single `%` wide character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[412)]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

15 Trailing white-space wide characters (including new-line wide characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns

16 The `fwscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure or if the implementation does not support a specific width length modifier.

17 **EXAMPLE 1** The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int n, i; float x; wchar_t name[50];
n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

with the input line:

[412)]See "future library directions" (7.33.20).

```
25 54.32E-1 thompson
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `thompson\0`.

18 **EXAMPLE 2** The call:

```
#include <stdio.h>
#include <wchar.h>
/* ... */
int i; float x; double y;
fwscanf(stdin, L"%2d%f%*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip past `0123`, and will assign to `y` the value 56.0. The next wide character read from the input stream will be `a`.

Forward references: the **`wcstod`**, **`wcstof`**, and **`wcstold`** functions (7.31.4.1.2), the **`wcstol`**, **`wcstoll`**, **`wcstoul`**, and **`wcstoull`** functions (7.31.4.1.4), the **`wcrtomb`** function (7.31.6.3.3).

7.31.2.3 The `swprintf` function

Synopsis

1
```
#include <wchar.h>
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
      ...);
```

Description

2 The **`swprintf`** function is equivalent to **`fwprintf`**, except that the argument `s` specifies an array of wide characters into which the generated output is to be written, rather than written to a stream. No more than `n` wide characters are written, including a terminating null wide character, which is always added (unless `n` is zero).

Returns

3 The **`swprintf`** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be written.

7.31.2.4 The `swscanf` function

Synopsis

1
```
#include <wchar.h>
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Description

2 The **`swscanf`** function is equivalent to **`fwscanf`**, except that the argument `s` specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **`fwscanf`** function.

Returns

3 The **`swscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **`swscanf`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.31.2.5 The vfwprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
```

Description

2 The `vfwprintf` function is equivalent to `fwprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwprintf` function does not invoke the `va_end` macro[413].

Returns

3 The `vfwprintf` function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

4 **EXAMPLE** The following shows the use of the `vfwprintf` function in a general error-reporting routine.

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

void error(char *function_name, wchar_t *format, ...)
{
      va_list args;

      va_start(args, format);
      // print out name of function causing error
      fwprintf(stderr, L"ERROR in %s: ", function_name);
      // print out remainder of message
      vfwprintf(stderr, format, args);
      va_end(args);
}
```

7.31.2.6 The vfwscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
```

Description

2 The `vfwscanf` function is equivalent to `fwscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwscanf` function does not invoke the `va_end` macro.[413]

Returns

3 The `vfwscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vfwscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

[413] As the functions `vfwprintf`, `vswprintf`, `vfwscanf`, `vwprintf`, `vwscanf`, and `vswscanf` invoke the `va_arg` macro, the representation of `arg` after the return is indeterminate.

7.31.2.7 The vswprintf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **vswprintf** function is equivalent to **swprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswprintf** function does not invoke the **va_end** macro.[413]

Returns

3 The **vswprintf** function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if n or more wide characters were requested to be generated.

7.31.2.8 The vswscanf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,
     va_list arg);
```

Description

2 The **vswscanf** function is equivalent to **swscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vswscanf** function does not invoke the **va_end** macro.[413]

Returns

3 The **vswscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vswscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.31.2.9 The vwprintf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t * restrict format, va_list arg);
```

Description

2 The **vwprintf** function is equivalent to **wprintf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vwprintf** function does not invoke the **va_end** macro.[413]

Returns

3 The **vwprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.31.2.10 The vwscanf function

Synopsis

1
```
#include <stdarg.h>
#include <wchar.h>
```

```
int vwscanf(const wchar_t * restrict format, va_list arg);
```

Description

2 The **vwscanf** function is equivalent to **wscanf**, with the variable argument list replaced by arg, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vwscanf** function does not invoke the **va_end** macro.[413)]

Returns

3 The **vwscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vwscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.31.2.11 The wprintf function

Synopsis

1
```
#include <wchar.h>
int wprintf(const wchar_t * restrict format, ...);
```

Description

2 The **wprintf** function is equivalent to **fwprintf** with the argument **stdout** interposed before the arguments to **wprintf**.

Returns

3 The **wprintf** function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.31.2.12 The wscanf function

Synopsis

1
```
#include <wchar.h>
int wscanf(const wchar_t * restrict format, ...);
```

Description

2 The **wscanf** function is equivalent to **fwscanf** with the argument **stdin** interposed before the arguments to **wscanf**.

Returns

3 The **wscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **wscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.31.3 Wide character input/output functions

7.31.3.1 The fgetwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

Description

2 If the end-of-file indicator for the input stream pointed to by stream is not set and a next wide character is present, the **fgetwc** function obtains that wide character as a **wchar_t** converted to a **wint_t** and advances the associated file position indicator for the stream (if defined).

Returns

3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the **fgetwc** function returns **WEOF**. Otherwise, the **fgetwc** function returns the next wide character from the input stream pointed to by stream. If a read error occurs, the error indicator for the stream is set and the **fgetwc** function returns **WEOF**. If an encoding error occurs (including too few bytes), the error indicator for the stream is set and the value of the macro **EILSEQ** is stored in **errno** and the **fgetwc** function returns **WEOF**.[414]

7.31.3.2 The fgetws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

Description

2 The **fgetws** function reads at most one less than the number of wide characters specified by n from the stream pointed to by stream into the array pointed to by s. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

Returns

3 The **fgetws** function returns s if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array members have an indeterminate representation and a null pointer is returned.

7.31.3.3 The fputwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t c, FILE *stream);
```

Description

2 The **fputwc** function writes the wide character specified by c to the output stream pointed to by stream, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns

3 The **fputwc** function returns the wide character written. If a write error occurs, the error indicator for the stream is set and **fputwc** returns **WEOF**. If an encoding error occurs, the value of the macro **EILSEQ** is stored in **errno** and **fputwc** returns **WEOF**.

7.31.3.4 The fputws function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t * restrict s, FILE * restrict stream);
```

Description

2 The **fputws** function writes the wide string pointed to by s to the stream pointed to by stream. The terminating null wide character is not written.

[414] An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions. Also, **errno** will be set to **EILSEQ** by input/output functions only if an encoding error occurs.

Returns

3 The **fputws** function returns **EOF** if a write or encoding error occurs; otherwise, it returns a nonnegative value.

7.31.3.5 The fwide function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

2 The **fwide** function determines the orientation of the stream pointed to by `stream`. If `mode` is greater than zero, the function first attempts to make the stream wide oriented. If `mode` is less than zero, the function first attempts to make the stream byte oriented.[415] Otherwise, `mode` is zero and the function does not alter the orientation of the stream.

Returns

3 The **fwide** function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

7.31.3.6 The getwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

2 The **getwc** function is equivalent to **fgetwc**, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects.

Returns

3 The **getwc** function returns the next wide character from the input stream pointed to by `stream`, or **WEOF**.

7.31.3.7 The getwchar function

Synopsis

1
```
#include <wchar.h>
wint_t getwchar(void);
```

Description

2 The **getwchar** function is equivalent to **getwc** with the argument **stdin**.

Returns

3 The **getwchar** function returns the next wide character from the input stream pointed to by **stdin**, or **WEOF**.

7.31.3.8 The putwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t c, FILE *stream);
```

[415]If the orientation of the stream has already been determined, **fwide** does not change it.

Description

2 The **putwc** function is equivalent to **fputwc**, except that if it is implemented as a macro, it may evaluate `stream` more than once, so that argument should never be an expression with side effects.

Returns

3 The **putwc** function returns the wide character written, or **WEOF**.

7.31.3.9 The putwchar function

Synopsis

1
```
#include <wchar.h>
wint_t putwchar(wchar_t c);
```

Description

2 The **putwchar** function is equivalent to **putwc** with the second argument **stdout**.

Returns

3 The **putwchar** function returns the character written, or **WEOF**.

7.31.3.10 The ungetwc function

Synopsis

1
```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

2 The **ungetwc** function pushes the wide character specified by `c` back onto the input stream pointed to by `stream`. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file positioning function (**fseek**, **fsetpos**, or **rewind**) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

3 One wide character of pushback is guaranteed, even if the call to the **ungetwc** function follows just after a call to a formatted wide character input function **fwscanf**, **vfwscanf**, **vwscanf**, or **wscanf**. If the **ungetwc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of `c` equals that of the macro **WEOF**, the operation fails and the input stream is unchanged.

5 A successful call to the **ungetwc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.[416] For a text or binary stream, the value of its file position indicator after a successful call to the **ungetwc** function is unspecified until all pushed-back wide characters are read or discarded.

Returns

6 The **ungetwc** function returns the wide character pushed back, or **WEOF** if the operation fails.

7.31.4 General wide string utilities

1 The header `<wchar.h>` declares functions for wide string manipulation. Various methods are used for determining the lengths of the arrays, but in all cases a **wchar_t*** argument points to the initial (lowest addressed) element of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 Where an argument declared as **size_t** `n` determines the length of the array for a function, `n` can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of

[416]Note that a file positioning function could further modify the file position indicator after discarding any pushed-back wide characters.

a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

7.31.4.1 Wide string numeric conversion functions

7.31.4.1.1 General

This subclause describes wide string analogs of the **strtod** family of functions (7.24.1.5, 7.24.1.6)[417].

7.31.4.1.2 The wcstod, wcstof, and wcstold functions

```
#include <wchar.h>
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

1 The **wcstod**, **wcstof**, and **wcstold** functions convert the initial portion of the wide string pointed to by `nptr` to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters, a subject sequence resembling a floating constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

2 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in 6.4.4.2, excluding any digit separators (6.4.4.1);

— a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part as defined in 6.4.4.2, excluding any digit separators (6.4.4.1);

— `INF` or `INFINITY`, or any other wide string equivalent except for case

— `NAN` or `NAN(`*n-wchar-sequence*$_{opt}$`)`, or any other wide string equivalent except for case in the `NAN` part, where:

[417]Wide string analogs of the **strfromd** family of functions (7.24.1.5, 7.24.1.6) are not provided because those conversions can be done by using **mbstowcs** (7.24.8.1) to convert the result of **strfromd**, **strfromf**, and similar to wide string. For example, the following converts **double** `d` to wide string `ws` with at most `n-1` non-null wide characters, using style `g` formatting, and computes the number `nc` of wide characters that would have been written had `n` been sufficiently large, not counting the terminating null wide character.

```
#include <stdlib.h>
const size_t n = 20;
double d;
//...
// convert d to single-byte character string s
char s[n];
int nc = strfromd(s, n, "%g", d);
// convert s (regarded as a multi-byte character
// string) to wide string ws
wchar_t ws[n];
(void)mbstowcs(ws, s, n);
```

n-wchar-sequence:
digit
nondigit
n-wchar-sequence digit
n-wchar-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

3 If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated.[418)]

A wide character sequence `INF` or `INFINITY` is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence `NAN` or `NAN(`*n-wchar-sequence*$_{opt}$`)` is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequence is implementation-defined.[419)]

A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

4 If the subject sequence has the hexadecimal form and **`FLT_RADIX`** is a power of 2, the value resulting from the conversion is correctly rounded.

5 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

6 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Recommended practice

7 If the subject sequence has the hexadecimal form, **`FLT_RADIX`** is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

8 If the subject sequence has the decimal form and at most M significant digits, where M is the maximum value of the *T*_**`DECIMAL_DIG`** macros (defined in `<float.h>`), the result should be correctly rounded. If the subject sequence D has the decimal form and more than M significant digits, consider the two bounding, adjacent decimal strings L and U, both having M significant digits, such that the values of L, D, and U satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.[420)]

[418)] It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods could yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

[419)] An implementation can use the n-wchar sequence to determine extra information to be represented in the NaN's significand.

[420)] M is sufficiently large that L and `U` will usually correctly round to the same internal floating value, but if not will correctly round to adjacent values.

Returns

9 The functions return the converted value, if any. If no conversion could be performed, zero is returned.

If the correct value overflows and default rounding is in effect (7.12.1), plus or minus `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` is returned (according to the return type and sign of the value); if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value of `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the "overflow" floating-point exception is raised.

If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether `errno` acquires the value `ERANGE` is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

7.31.4.1.3 The `wcstod`*N* functions

Synopsis

1
```
#include <wchar.h>
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 wcstod32(const wchar_t * restrict nptr, char ** restrict endptr);
_Decimal64 wcstod64(const wchar_t * restrict nptr,char ** restrict endptr);
_Decimal128 wcstod128(const wchar_t * restrict nptr,char ** restrict endptr);
#endif
```

Description

2 The `wcstod`*N* functions convert the initial portion of the wide string pointed to by `nptr` to decimal floating type representation. First, they decompose the input wide string into three parts: an initial, possibly empty, sequence of white-space wide characters; a subject sequence resembling a floating constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

— a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined in 6.4.4.2, excluding any digit separators (6.4.4.1)

— `INF` or `INFINITY`, ignoring case

— `NAN` or `NAN(`*d-wchar-sequence*$_{opt}$`)`, ignoring case in the `NAN` part, where:

d-wchar-sequence:
  digit
  nondigit
  d-wchar-sequence digit
  d-wchar-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, including correct rounding and determination of the coefficient c and the quantum exponent q, with the following exceptions:

— It is not a hexadecimal floating number.

— The decimal-point wide character is used in place of a period.

— If neither an exponent part nor a decimal-point wide character appears in a decimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the wide string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated before rounding and the sign s is set to -1, else s is set to 1. A wide character sequence `INF` or `INFINITY` is interpreted as an infinity. A wide character sequence `NAN` or `NAN(`*d-wchar-sequence*$_{opt}$`)`, is interpreted as a quiet NaN; the meaning of the d-wchar sequence is implementation-defined.[421] A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

5 In other than the `"C"` locale, additional locale-specific subject sequence forms may be accepted.

6 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns

7 The **`wcstod`***N* functions return the correctly rounded converted value, if any. If no conversion could be performed, the value of the triple $(+1, 0, 0)$ is returned. If the correct value overflows:

— the value of the macro **`ERANGE`** is stored in errno if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero;

— the "overflow" floating-point exception is raised if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero.

If the result underflows (7.12.1), whether **`errno`** acquires the value **`ERANGE`** if the integer expression **`math_errhandling`** `&` **`MATH_ERRNO`** is nonzero is implementation-defined; if the integer expression **`math_errhandling`** `&` **`MATH_ERREXCEPT`** is nonzero, whether the "underflow" floating-point exception is raised is implementation-defined.

7.31.4.1.4 The `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions

Synopsis

1
```
#include <wchar.h>
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
      int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
      int base);
unsigned long int wcstoul(const wchar_t * restrict nptr,
      wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr,
      wchar_t ** restrict endptr, int base);
```

Description

2 The **`wcstol`**, **`wcstoll`**, **`wcstoul`**, and **`wcstoull`** functions convert the initial portion of the wide string pointed to by `nptr` to **`long int`**, **`long long int`**, **`unsigned long int`**, and **`unsigned long long int`** representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters, a subject sequence resembling an integer represented in some radix determined by the value of `base`, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.

[421]An implementation may use the d-wchar sequence to determine extra information to be represented in the NaN's significand.

3 If the value of base is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix or any optional digit separators (6.4.4.1). If the value of base is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix or any optional digit separators. The letters from a (or A) through z (or Z) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of base are permitted. If the value of base is 2, the characters 0b or 0B may optionally precede the sequence of letters and digits, following the sign if present. If the value of base is 16, the wide characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

4 The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white-space wide characters, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

5 If the subject sequence has the expected form and the value of base is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

6 In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Returns

8 The **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MIN**, **LONG_MAX**, **LLONG_MIN**, **LLONG_MAX**, **ULONG_MAX**, or **ULLONG_MAX** is returned (according to the return type sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

7.31.4.2 Wide string copying functions

7.31.4.2.1 The wcscpy function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Description

2 The **wcscpy** function copies the wide string pointed to by s2 (including the terminating null wide character) into the array pointed to by s1.

Returns

3 The **wcscpy** function returns the value of s1.

7.31.4.2.2 The wcsncpy function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description

2 The **wcsncpy** function copies not more than n wide characters (those that follow a null wide character are not copied) from the array pointed to by s2 to the array pointed to by s1.[422]

3 If the array pointed to by s2 is a wide string that is shorter than n wide characters, null wide characters are appended to the copy in the array pointed to by s1, until n wide characters in all have been written.

Returns

4 The **wcsncpy** function returns the value of s1.

7.31.4.2.3 The wmemcpy function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description

2 The **wmemcpy** function copies n wide characters from the object pointed to by s2 to the object pointed to by s1.

Returns

3 The **wmemcpy** function returns the value of s1.

7.31.4.2.4 The wmemmove function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

2 The **wmemmove** function copies n wide characters from the object pointed to by s2 to the object pointed to by s1. Copying takes place as if the n wide characters from the object pointed to by s2 are first copied into a temporary array of n wide characters that does not overlap the objects pointed to by s1 or s2, and then the n wide characters from the temporary array are copied into the object pointed to by s1.

Returns

3 The **wmemmove** function returns the value of s1.

7.31.4.3 Wide string concatenation functions

7.31.4.3.1 The wcscat function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Description

2 The **wcscat** function appends a copy of the wide string pointed to by s2 (including the terminating null wide character) to the end of the wide string pointed to by s1. The initial wide character of s2 overwrites the null wide character at the end of s1.

Returns

3 The **wcscat** function returns the value of s1.

[422]Thus, if there is no null wide character in the first n wide characters of the array pointed to by s2, the result will not be null-terminated.

7.31.4.3.2 The wcsncat function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description

2 The `wcsncat` function appends not more than `n` wide characters (a null wide character and those that follow it are not appended) from the array pointed to by `s2` to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`. A terminating null wide character is always appended to the result.[423]

Returns

3 The `wcsncat` function returns the value of `s1`.

7.31.4.4 Wide string comparison functions

1 Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integer type designated by `wchar_t`.

7.31.4.4.1 The wcscmp function

Synopsis

1
```
#include <wchar.h>
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The `wcscmp` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`.

Returns

3 The `wcscmp` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2`.

7.31.4.4.2 The wcscoll function

Synopsis

1
```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The `wcscoll` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Returns

3 The `wcscoll` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2` when both are interpreted as appropriate to the current locale.

7.31.4.4.3 The wcsncmp function

Synopsis

1
```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

2 The `wcsncmp` function compares not more than `n` wide characters (those that follow a null wide character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

[423]Thus, the maximum number of wide characters that can end up in the array pointed to by `s1` is `wcslen(s1)+n+1`.

Returns

3 The **wcsncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2.

7.31.4.4.4 The wcsxfrm function

Synopsis

1
```
#include <wchar.h>
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description

2 The **wcsxfrm** function transforms the wide string pointed to by s2 and places the resulting wide string into the array pointed to by s1. The transformation is such that if the **wcscmp** function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **wcscoll** function applied to the same two original wide strings. No more than n wide characters are placed into the resulting array pointed to by s1, including the terminating null wide character. If n is zero, s1 is permitted to be a null pointer.

Returns

3 The **wcsxfrm** function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is n or greater, the members of the array pointed to by s1 have an indeterminate representation.

4 **EXAMPLE** The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by s:

```
1 + wcsxfrm(NULL, s, 0)
```

7.31.4.4.5 The wmemcmp function

Synopsis

1
```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

2 The **wmemcmp** function compares the first n wide characters of the object pointed to by s1 to the first n wide characters of the object pointed to by s2.

Returns

3 The **wmemcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

7.31.4.5 Wide string search functions

7.31.4.6 Introduction

1 The stateless search functions in this section (**wcschr**, **wcspbrk**, **wcsrchr**, **wmemchr**, **wcsstr**) are *generic functions*. These functions are generic in the qualification of the array to be searched and will return a result pointer to an element with the same qualification as the passed array. If the array to be searched is **const**-qualified, the result pointer will be to a **const**-qualified element. If the array to be searched is not **const**-qualified[424], the result pointer will be to an unqualified element.

2 The external declarations of these generic functions have a concrete function type that returns a pointer to an unqualified element of type **wchar_t** (named ***QWchar_t***), and accepts a pointer to a **const**-qualified array of the same type to search. This signature supports all correct uses. If a macro

[424]The null pointer constant is not a pointer to a **const**-qualified type, and therefore the result expression has the type of a pointer to an unqualified element; however, evaluating such a call is undefined.

definition of any of these generic functions is suppressed to access an actual function, the external declaration with this concrete type is visible.[425)]

3 The **volatile** and **restrict** qualifiers are not accepted on the elements of the array to search.

7.31.4.6.1 The wcschr generic function

Synopsis

1
```
#include <wchar.h>
QWchar_t *wcschr(QWchar_t *s, wchar_t c);
```

Description

2 The **wcschr** generic function locates the first occurrence of c in the wide string pointed to by s. The terminating null wide character is considered to be part of the wide string.

Returns

3 The **wcschr** generic function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

7.31.4.6.2 The wcscspn function

Synopsis

1
```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **wcscspn** function computes the length of the maximum initial segment of the wide string pointed to by s1 which consists entirely of wide characters *not* from the wide string pointed to by s2.

Returns

3 The **wcscspn** function returns the length of the segment.

425)This is an obsolescent feature.

7.31.4.6.3 The wcspbrk generic function

Synopsis

1
```
#include <wchar.h>
QWchar_t *wcspbrk(QWchar_t *s1, const wchar_t *s2);
```

Description

2 The wcspbrk generic function locates the first occurrence in the wide string pointed to by s1 of any wide character from the wide string pointed to by s2.

Returns

3 The wcspbrk generic function returns a pointer to the wide character in s1, or a null pointer if no wide character from s2 occurs in s1.

7.31.4.6.4 The wcsrchr generic function

Synopsis

1
```
#include <wchar.h>
QWchar_t *wcsrchr(QWchar_t *s, wchar_t c);
```

Description

2 The wcsrchr generic function locates the last occurrence of c in the wide string pointed to by s. The terminating null wide character is considered to be part of the wide string.

Returns

3 The wcsrchr generic function returns a pointer to the wide character, or a null pointer if c does not occur in the wide string.

7.31.4.6.5 The wcsspn function

Synopsis

1
```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The wcsspn function computes the length of the maximum initial segment of the wide string pointed to by s1 which consists entirely of wide characters from the wide string pointed to by s2.

Returns

3 The wcsspn function returns the length of the segment.

7.31.4.6.6 The wcsstr generic function

Synopsis

1
```
#include <wchar.h>
QWchar_t *wcsstr(QWchar_t *s1, const wchar_t *s2);
```

Description

2 The wcsstr generic function locates the first occurrence in the wide string pointed to by s1 of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by s2.

Returns

3 The wcsstr generic function returns a pointer to the located wide string, or a null pointer if the wide string is not found. If s2 points to a wide string with zero length, the function returns s1.

7.31.4.6.7 The wcstok function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2,
      wchar_t ** restrict ptr);
```

Description

2 A sequence of calls to the **wcstok** function breaks the wide string pointed to by s1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by s2. The third argument points to a caller-provided **wchar_t** pointer into which the **wcstok** function stores information necessary for it to continue scanning the same wide string.

3 The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by ptr. Subsequent calls in the sequence have a null first argument and the object pointed to by ptr is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by s2 may be different from call to call.

4 The first call in the sequence searches the wide string pointed to by s1 for the first wide character that is *not* contained in the current separator wide string pointed to by s2. If no such wide character is found, then there are no tokens in the wide string pointed to by s1 and the **wcstok** function returns a null pointer. If such a wide character is found, it is the start of the first token.

5 The **wcstok** function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by s1, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

6 In all cases, the **wcstok** function stores sufficient information in the pointer pointed to by ptr so that subsequent calls, with a null pointer for s1 and the unmodified pointer value for ptr, shall start searching just past the element overwritten by a null wide character (if any).

Returns

7 The **wcstok** function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

8 **EXAMPLE**

```
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;

t = wcstok(str1, L"?", &ptr1);      // t points to the token L"a"
t = wcstok(NULL, L",", &ptr1);      // t points to the token L"??b"
t = wcstok(str2, L" \t", &ptr2);    // t is a null pointer
t = wcstok(NULL, L"#,", &ptr1);     // t points to the token L"c"
t = wcstok(NULL, L"?", &ptr1);      // t is a null pointer
```

7.31.4.6.8 The wmemchr generic function

Synopsis

1
```
#include <wchar.h>
QWchar_t *wmemchr(QWchar_t *s, wchar_t c, size_t n);
```

Description

2 The **wmemchr** generic function locates the first occurrence of c in the initial n wide characters of the object pointed to by s.

Returns

3 The `wmemchr` generic function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

7.31.4.7 Miscellaneous functions

7.31.4.7.1 The `wcslen` function

Synopsis

1
```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

Description

2 The `wcslen` function computes the length of the wide string pointed to by `s`.

Returns

3 The `wcslen` function returns the number of wide characters that precede the terminating null wide character.

7.31.4.7.2 The `wmemset` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

2 The `wmemset` function copies the value of `c` into each of the first `n` wide characters of the object pointed to by `s`.

Returns

3 The `wmemset` function returns the value of `s`.

7.31.5 Wide character time conversion functions

7.31.5.1 The `wcsftime` function

Synopsis

1
```
#include <time.h>
#include <wchar.h>
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
      const wchar_t * restrict format, const struct tm * restrict timeptr);
```

Description

2 The `wcsftime` function is equivalent to the `strftime` function, except that:

— The argument `s` points to the initial element of an array of wide characters into which the generated output is to be placed.

— The argument `maxsize` indicates the limiting number of wide characters.

— The argument `format` is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.

— The return value indicates the number of wide characters.

Returns

3 If the total number of resulting wide characters including the terminating null wide character is not more than `maxsize`, the `wcsftime` function returns the number of wide characters placed into the array pointed to by `s` not including the terminating null wide character. Otherwise, zero is returned and the members of the array have an indeterminate representation.

7.31.6 Extended multibyte/wide character conversion utilities

1 The header `<wchar.h>` declares an extended set of functions useful for conversion between multibyte characters and wide characters.

2 Most of the following functions — those that are listed as "restartable", 7.31.6.3 and 7.31.6.4 — take as a last argument a pointer to an object of type **mbstate_t** that is used to describe the current *conversion state* from a particular multibyte character sequence to a wide character sequence (or the reverse) under the rules of a particular setting for the **LC_CTYPE** category of the current locale.

3 The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new multibyte character in the initial shift state. A zero-valued **mbstate_t** object is (at least) one way to describe an initial conversion state. A zero-valued **mbstate_t** object can be used to initiate conversion involving any multibyte character sequence, in any **LC_CTYPE** category setting. If an **mbstate_t** object has been altered by any of the functions described in this subclause, and is then used with a different multibyte character sequence, or in the other conversion direction, or with a different **LC_CTYPE** category setting than on earlier function calls, the behavior is undefined.[426)]

4 On entry, each function takes the described conversion state (either internal or pointed to by an argument) as current. The conversion state described by the referenced object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

7.31.6.1 Single-byte/wide character conversion functions

7.31.6.1.1 The btowc function

Synopsis

1
```
#include <wchar.h>
wint_t btowc(int c);
```

Description

2 The **btowc** function determines whether `c` constitutes a valid single-byte character in the initial shift state.

Returns

3 The **btowc** function returns **WEOF** if `c` has the value **EOF** or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state. Otherwise, it returns the wide character representation of that character.

7.31.6.1.2 The wctob function

Synopsis

1
```
#include <wchar.h>
int wctob(wint_t c);
```

Description

2 The **wctob** function determines whether `c` corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Returns

3 The **wctob** function returns **EOF** if `c` does not correspond to a multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an **unsigned char** converted to an **int**.

7.31.6.2 Conversion state functions

7.31.6.2.1 The mbsinit function

Synopsis

1
```
#include <wchar.h>
```

[426)]Thus, a particular **mbstate_t** object can be used, for example, with both the **mbrtowc** and **mbsrtowcs** functions as long as they are used to step sequentially through the same multibyte character string.

```
int mbsinit(const mbstate_t *ps);
```

Description

2 If `ps` is not a null pointer, the `mbsinit` function determines whether the referenced `mbstate_t` object describes an initial conversion state.

Returns

3 The `mbsinit` function returns nonzero if `ps` is a null pointer or if the referenced object describes an initial conversion state; otherwise, it returns zero.

7.31.6.3 Restartable multibyte/wide character conversion functions

1 These functions differ from the corresponding multibyte character functions of 7.24.7 (`mblen`, `mbtowc`, and `wctomb`) in that they have an extra parameter, `ps`, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a null pointer, each function uses its own internal `mbstate_t` object instead, which is initialized prior to the first call to the function to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. It is implementation-defined whether the internal `mbstate_t` object has thread storage duration; if it has thread storage duration, it is initialized to the initial conversion state prior to the first call to the function on the new thread. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

2 Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.31.6.3.1 The `mbrlen` function

Synopsis

1
```
#include <wchar.h>
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

2 The `mbrlen` function is equivalent to the call:

```
mbrtowc(NULL, s, n, ps != NULL ? ps: &internal)
```

where `internal` is the `mbstate_t` object for the `mbrlen` function, except that the expression designated by `ps` is evaluated only once.

Returns

3 The `mbrlen` function returns a value between zero and `n`, inclusive, `(size_t)`(-2), or `(size_t)`(-1).

Forward references: the `mbrtowc` function (7.31.6.3.2).

7.31.6.3.2 The `mbrtowc` function

Synopsis

1
```
#include <wchar.h>
size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n,
     mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `mbrtowc` function is equivalent to the call:

```
    mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the parameters `pwc` and `n` are ignored.

3 If `s` is not a null pointer, the `mbrtowc` function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The `mbrtowc` function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)`(-2) if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[427)]

`(size_t)(-1)` if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro `EILSEQ` is stored in `errno`, and the conversion state is unspecified.

7.31.6.3.3 The `wcrtomb` function

Synopsis

1
```
#include <wchar.h>
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the `wcrtomb` function is equivalent to the call

```
wcrtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the `wcrtomb` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most `MB_CUR_MAX` bytes are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The `wcrtomb` function returns the number of bytes stored in the array object (including any shift sequences). When `wc` is not a valid wide character, an encoding error occurs: the function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)`(-1); the conversion state is unspecified.

7.31.6.4 Restartable multibyte/wide string conversion functions

1 These functions differ from the corresponding multibyte string functions of 7.24.8 (`mbstowcs` and `wcstombs`) in that they have an extra parameter, `ps`, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a null pointer, each function uses its own internal `mbstate_t` object instead, which is initialized prior to the first call to the function to the initial conversion state; the

[427)] When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

functions are not required to avoid data races with other calls to the same function in this case. It is implementation-defined whether the internal **mbstate_t** object has thread storage duration; if it has thread storage duration, it is initialized to the initial conversion state prior to the first call to the function on the new thread. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

2 Also unlike their corresponding functions, the conversion source parameter, `src`, has a pointer-to-pointer type. When the function is storing the results of conversions (that is, when `dst` is not a null pointer), the pointer object pointed to by this parameter is updated to reflect the amount of the source processed by that invocation.

7.31.6.4.1 The `mbsrtowcs` function

Synopsis

1
```
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len,
     mbstate_t * restrict ps);
```

Description

2 The **mbsrtowcs** function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by `ps`, from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters are stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.[428] Each conversion takes place as if by a call to the **mbrtowc** function.

3 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if `dst` is not a null pointer, the resulting state described is the initial conversion state.

Returns

4 If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **mbsrtowcs** function stores the value of the macro **EILSEQ** in **errno** and returns `(size_t)(-1)`; the conversion state is unspecified. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null character (if any).

7.31.6.4.2 The `wcsrtombs` function

Synopsis

1
```
#include <wchar.h>
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len,
     mbstate_t * restrict ps);
```

Description

2 The **wcsrtombs** function converts a sequence of wide characters from the array indirectly pointed to by `src` into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters are then stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multibyte character, or (if `dst` is not a null pointer) when the next multibyte character would exceed the limit of `len` total bytes to be stored into the array pointed to by `dst`. Each conversion takes place as if by a call to the **wcrtomb** function.[429]

[428] Thus, the value of `len` is ignored if `dst` is a null pointer.

[429] If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.

3 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Returns

4 If conversion stops because a wide character is reached that does not correspond to a valid multibyte character, an encoding error occurs: the **`wcsrtombs`** function stores the value of the macro **`EILSEQ`** in **`errno`** and returns **`(size_t)`**(-1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

7.32 Wide character classification and mapping utilities <wctype.h>

7.32.1 Introduction

1 The header <wctype.h> defines one macro, and declares three data types and many functions.[430)]

2 The types declared are **wint_t** described in 7.31.1;

```
wctrans_t
```

which is a scalar type that can hold values which represent locale-specific character mappings; and

```
wctype_t
```

which is a scalar type that can hold values which represent locale-specific character classifications.

3 The macro defined is **WEOF** (described in 7.31.1).

4 The functions declared are grouped as follows:

— Functions that provide wide character classification;

— Extensible functions that provide wide character classification;

— Functions that provide wide character case mapping;

— Extensible functions that provide wide character mapping.

5 For all functions described in this subclause that accept an argument of type **wint_t**, the value shall be representable as a **wchar_t** or shall equal the value of the macro **WEOF**. If this argument has any other value, the behavior is undefined.

6 The behavior of these functions is affected by the **LC_CTYPE** category of the current locale.

7.32.2 Wide character classification utilities

1 The header <wctype.h> declares several functions useful for classifying wide characters.

2 The term *printing wide character* refers to a member of a locale-specific set of wide characters, each of which occupies at least one printing position on a display device. The term *control wide character* refers to a member of a locale-specific set of wide characters that are not printing wide characters.

7.32.2.1 Wide character classification functions

1 The functions in this subclause return nonzero (true) if and only if the value of the argument `wc` conforms to that in the description of the function.

2 Each of the following functions returns true for each wide character that corresponds (as if by a call to the **wctob** function) to a single-byte character for which the corresponding character classification function from 7.4.1 returns true, except that the **iswgraph** and **iswpunct** functions may differ with respect to wide characters other than `L' '` that are both printing and white-space wide characters.[431)]

Forward references: the **wctob** function (7.31.6.1.2).

[430)]See "future library directions" (7.33.21).

[431)]For example, if the expression `isalpha(wctob(wc))` evaluates to true, then the call `iswalpha(wc)` also returns true. But, if the expression `isgraph(wctob(wc))` evaluates to true (which cannot occur for `wc == L' '` of course), then either `iswgraph(wc)` or `iswprint(wc)&& iswspace(wc)` is true, but not both.

7.32.2.1.1 The iswalnum function

Synopsis

1
```
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

2 The **iswalnum** function tests for any wide character for which **iswalpha** or **iswdigit** is true.

7.32.2.1.2 The iswalpha function

Synopsis

1
```
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

2 The **iswalpha** function tests for any wide character for which **iswupper** or **iswlower** is true, or any wide character that is one of a locale-specific set of alphabetic wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.[432)]

7.32.2.1.3 The iswblank function

Synopsis

1
```
#include <wctype.h>
int iswblank(wint_t wc);
```

Description

2 The **iswblank** function tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which **iswspace** is true and that is used to separate words within a line of text. The standard blank wide characters are the following: space (`L' '`), and horizontal tab (`L'\t'`). In the `"C"` locale, **iswblank** returns true only for the standard blank characters.

7.32.2.1.4 The iswcntrl function

Synopsis

1
```
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

2 The **iswcntrl** function tests for any control wide character.

7.32.2.1.5 The iswdigit function

Synopsis

1
```
#include <wctype.h>
int iswdigit(wint_t wc);
```

Description

2 The **iswdigit** function tests for any wide character that corresponds to a decimal-digit character (as defined in 5.2.1).

[432)]The functions **iswlower** and **iswupper** test true or false separately for each of these additional wide characters; all four combinations are possible.

7.32.2.1.6 The iswgraph function

Synopsis

1
```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

2 The **iswgraph** function tests for any wide character for which **iswprint** is true and **iswspace** is false.[433)]

7.32.2.1.7 The iswlower function

Synopsis

1
```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

2 The **iswlower** function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

7.32.2.1.8 The iswprint function

Synopsis

1
```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

2 The **iswprint** function tests for any printing wide character.

7.32.2.1.9 The iswpunct function

Synopsis

1
```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

2 The **iswpunct** function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither **iswspace** nor **iswalnum** is true.[433)]

7.32.2.1.10 The iswspace function

Synopsis

1
```
#include <wctype.h>
int iswspace(wint_t wc);
```

Description

2 The **iswspace** function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of **iswalnum**, **iswgraph**, or **iswpunct** is true.

7.32.2.1.11 The iswupper function

Synopsis

1
```
#include <wctype.h>
int iswupper(wint_t wc);
```

[433)]Note that the behavior of the **iswgraph** and **iswpunct** functions can differ from their corresponding functions in 7.4.1 with respect to printing, white-space, single-byte execution characters other than `' '`.

Description

2 The **iswupper** function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of **iswcntrl**, **iswdigit**, **iswpunct**, or **iswspace** is true.

7.32.2.1.12 The iswxdigit function

Synopsis

1
```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Description

2 The **iswxdigit** function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in 6.4.4.1).

7.32.2.2 Extensible wide character classification functions

1 The functions **wctype** and **iswctype** provide extensible wide character classification as well as testing equivalent to that performed by the functions described in the previous subclause (7.32.2.1).

7.32.2.2.1 The iswctype function

Synopsis

1
```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description

2 The **iswctype** function determines whether the wide character `wc` has the property described by `desc`. The current setting of the **LC_CTYPE** category shall be the same as during the call to **wctype** that returned the value `desc`.

3 Each of the following expressions has a truth-value equivalent to the call to the wide character classification function (7.32.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum"))    // iswalnum(wc)
iswctype(wc, wctype("alpha"))    // iswalpha(wc)
iswctype(wc, wctype("blank"))    // iswblank(wc)
iswctype(wc, wctype("cntrl"))    // iswcntrl(wc)
iswctype(wc, wctype("digit"))    // iswdigit(wc)
iswctype(wc, wctype("graph"))    // iswgraph(wc)
iswctype(wc, wctype("lower"))    // iswlower(wc)
iswctype(wc, wctype("print"))    // iswprint(wc)
iswctype(wc, wctype("punct"))    // iswpunct(wc)
iswctype(wc, wctype("space"))    // iswspace(wc)
iswctype(wc, wctype("upper"))    // iswupper(wc)
iswctype(wc, wctype("xdigit"))   // iswxdigit(wc)
```

Returns

4 The **iswctype** function returns nonzero (true) if and only if the value of the wide character `wc` has the property described by `desc`. If `desc` is zero, the **iswctype** function returns zero (false).

Forward references: the **wctype** function (7.32.2.2.2).

7.32.2.2.2 The wctype function

Synopsis

1
```
#include <wctype.h>
wctype_t wctype(const char *property);
```

Description

2 The `wctype` function constructs a value with type `wctype_t` that describes a class of wide characters identified by the string argument `property`.

3 The strings listed in the description of the `iswctype` function shall be valid in all locales as `property` arguments to the `wctype` function.

Returns

4 If `property` identifies a valid class of wide characters according to the `LC_CTYPE` category of the current locale, the `wctype` function returns a nonzero value that is valid as the second argument to the `iswctype` function; otherwise, it returns zero.

7.32.3 Wide character case mapping utilities

1 The header `<wctype.h>` declares several functions useful for mapping wide characters.

7.32.3.1 Wide character case mapping functions

7.32.3.1.1 The `towlower` function

Synopsis

1
```
#include <wctype.h>
wint_t towlower(wint_t wc);
```

Description

2 The `towlower` function converts an uppercase letter to a corresponding lowercase letter.

Returns

3 If the argument is a wide character for which `iswupper` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswlower` is true, the `towlower` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.32.3.1.2 The `towupper` function

Synopsis

1
```
#include <wctype.h>
wint_t towupper(wint_t wc);
```

Description

2 The `towupper` function converts a lowercase letter to a corresponding uppercase letter.

Returns

3 If the argument is a wide character for which `iswlower` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswupper` is true, the `towupper` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.32.3.2 Extensible wide character case mapping functions

1 The functions `wctrans` and `towctrans` provide extensible wide character mapping as well as case mapping equivalent to that performed by the functions described in the previous subclause (7.32.3.1).

7.32.3.2.1 The `towctrans` function

Synopsis

1
```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

2 The **towctrans** function maps the wide character `wc` using the mapping described by `desc`. The current setting of the **LC_CTYPE** category shall be the same as during the call to **wctrans** that returned the value `desc`.

3 Each of the following expressions behaves the same as the call to the wide character case mapping function (7.32.3.1) in the comment that follows the expression:

```
towctrans(wc, wctrans("tolower"))    // towlower(wc)
towctrans(wc, wctrans("toupper"))    // towupper(wc)
```

Returns

4 The **towctrans** function returns the mapped value of `wc` using the mapping described by `desc`. If `desc` is zero, the **towctrans** function returns the value of `wc`.

7.32.3.2.2 The wctrans function

Synopsis

1
```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

Description

2 The **wctrans** function constructs a value with type **wctrans_t** that describes a mapping between wide characters identified by the string argument `property`.

3 The strings listed in the description of the **towctrans** function shall be valid in all locales as `property` arguments to the **wctrans** function.

Returns

4 If `property` identifies a valid mapping of wide characters according to the **LC_CTYPE** category of the current locale, the **wctrans** function returns a nonzero value that is valid as the second argument to the **towctrans** function; otherwise, it returns zero.

7.33 Future library directions

1 Although grouped under individual headers, all the external names identified as reserved identifiers or potentially reserved identifiers in this subclause remain so regardless of which headers are included in the program.

7.33.1 Complex arithmetic <complex.h>

1 The function names

`cacospi`	`cexp10m1`	`clog10`	`crootn`
`casinpi`	`cexp10`	`clog1p`	`crsqrt`
`catanpi`	`cexp2m1`	`clog2p1`	`csinpi`
`ccompoundn`	`cexp2`	`clog2`	`ctanpi`
`ccospi`	`cexpm1`	`clogp1`	`ctgamma`
`cerfc`	`clgamma`	`cpown`	
`cerf`	`clog10p1`	`cpowr`	

and the same names suffixed with `f` or `l` are potentially reserved identifiers and may be added to the declarations in the `<complex.h>` header.

7.33.2 Character handling <ctype.h>

1 Function names that begin with either `is` or `to`, and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<ctype.h>` header.

7.33.3 Errors <errno.h>

1 Macros that begin with `E` and a digit or `E` and an uppercase letter may be added to the macros defined in the `<errno.h>` header by a future revision of this document or by an implementation.

7.33.4 Floating-point environment <fenv.h>

1 Macros that begin with `FE_` and an uppercase letter may be added to the macros defined in the `<fenv.h>` header by a future revision of this document.

7.33.5 Characteristics of floating types <float.h>

1 Macros that begin with `DBL_`, `DEC32_`, `DEC64_`, `DEC128_`, `DEC_`, `FLT_`, or `LDBL_` and an uppercase letter are potentially reserved identifiers and may be added to the macros defined in the `<float.h>` header.

2 Use of the `DECIMAL_DIG` macro is an obsolescent feature. A similar type-specific macro, such as `LDBL_DECIMAL_DIG`, can be used instead.

3 The use of `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM`, and `LDBL_HAS_SUBNORM` macros is an obsolescent feature.

7.33.6 Format conversion of integer types <inttypes.h>

1 Macros that begin with either `PRI` or `SCN`, and either a lowercase letter or `X` are potentially reserved identifiers and may be added to the macros defined in the `<inttypes.h>` header.

2 Function names that begin with `str`, or `wcs` and a lowercase letter are potentially reserved identifiers may be added to the declarations in the `<inttypes.h>` header.

7.33.7 Localization <locale.h>

1 Macros that begin with `LC_` and an uppercase letter may be added to the macros defined in the `<locale.h>` header by a future revision of this document or by an implementation.

7.33.8 Mathematics <math.h>

1 Macros that begin with `FP_` and an uppercase letter may be added to the macros defined in the `<math.h>` header by a future revision of this document or by an implementation.

2 Macros that begin with MATH_ and an uppercase letter are potentially reserved identifiers and may be added to the macros in the <math.h> header.

3 Function names that begin with is and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the <math.h> header.

4 Function names that begin with cr_ are potentially reserved identifiers and may be added to the <math.h> header. The cr_ prefix is intended to indicate a correctly rounded version of the function.

5 Use of the macros INFINITY, DEC_INFINITY, NAN, and DEC_NAN in <math.h> is an obsolescent feature. Instead, use the same macros in <float.h>.

7.33.9 Signal handling <signal.h>

1 Macros that begin with either SIG and an uppercase letter or SIG_ and an uppercase letter may be added to the macros defined in the <signal.h> header by a future revision of this document or by an implementation.

7.33.10 Atomics <stdatomic.h>

1 Macros that begin with ATOMIC_ and an uppercase letter are potentially reserved identifiers and may be added to the macros defined in the <stdatomic.h> header. Typedef names that begin with either atomic_ or memory_, and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the <stdatomic.h> header. Enumeration constants that begin with memory_order_ and a lowercase letter are potentially reserved identifiers and may be added to the definition of the memory_order type in the <stdatomic.h> header. Function names that begin with atomic_ and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the <stdatomic.h> header.

7.33.11 Boolean type and values <stdbool.h>

1 The macro __bool_true_false_are_defined is an obsolescent feature.

7.33.12 Bit and byte utilities <stdbit.h>

1 Type and function names that begin with stdc_ are potentially reserved identifiers and may be added to the declarations in the <stdbit.h> header.

7.33.13 Checked Arithmetic Functions <stdckdint.h>

1 Type and function names that begin with ckd_ are potentially reserved identifiers and may be added to the declarations in the <stdckdint.h> header.

7.33.14 Integer types <stdint.h>

1 Typedef names beginning with int or uint and ending with _t are potentially reserved identifiers and may be added to the types defined in the <stdint.h> header. Macro names beginning with INT or UINT and ending with _MAX, _MIN, _WIDTH, or _C are potentially reserved identifiers and may be added to the macros defined in the <stdint.h> header.

7.33.15 Input/output <stdio.h>

1 Lowercase letters may be added to the conversion specifiers and length modifiers in fprintf and fscanf. Other characters may be used in extensions.

2 The use of ungetc on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.33.16 General utilities <stdlib.h>

1 Function names that begin with str or wcs and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the <stdlib.h> header.

2 Suppressing the macro definition of bsearch to access the actual function is an obsolescent feature.

7.33.17 String handling `<string.h>`

1 Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<string.h>` header.

2 Suppressing the macro definitions of **memchr**, **strchr**, **strpbrk**, **strrchr**, or **strstr** to access the corresponding actual function is an obsolescent feature.

7.33.18 Date and time `<time.h>`

1 Macros beginning with **TIME_** and an uppercase letter may be added to the macros in the `<time.h>` header by a future revision of this document or by an implementation.

2 The time bases **TIME_MONOTONIC**, **TIME_ACTIVE** and **TIME_THREAD_ACTIVE** may become mandatory in future versions of this standard.

7.33.19 Threads `<threads.h>`

1 Function names, type names, and enumeration constants that begin with either **cnd_**, **mtx_**, **thrd_**, or **tss_**, and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<threads.h>` header.

7.33.20 Extended multibyte and wide character utilities `<wchar.h>`

1 Function names that begin with **wcs** and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<wchar.h>` header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.

3 Suppressing the macro definitions of **wcschr**, **wcspbrk**, **wcsrchr**, **wmemchr**, or **wcsstr** to access the corresponding actual function is an obsolescent feature.

7.33.21 Wide character classification and mapping utilities `<wctype.h>`

1 Function names that begin with **is** or **to** and a lowercase letter are potentially reserved identifiers and may be added to the declarations in the `<wctype.h>` header.

Annex A
(informative)
Language syntax summary

1 **NOTE 1** The notation is described in 6.1.

A.1 Lexical grammar

A.1.1 Lexical elements

(6.4) *token:*
keyword
identifier
constant
string-literal
punctuator

(6.4) *preprocessing-token:*
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each universal-character-name that cannot be one of the above
each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) *keyword:* one of

alignas	**enum**	**short**	**void**
alignof	**extern**	**signed**	**volatile**
auto	**false**	**sizeof**	**while**
bool	**float**	**static**	**_Atomic**
break	**for**	**static_assert**	**_BitInt**
case	**goto**	**struct**	**_Complex**
char	**if**	**switch**	**_Decimal128**
const	**inline**	**thread_local**	**_Decimal32**
constexpr	**int**	**true**	**_Decimal64**
continue	**long**	**typedef**	**_Generic**
default	**nullptr**	**typeof**	**_Imaginary**
do	**register**	**typeof_unqual**	**_Noreturn**
double	**restrict**	**union**	
else	**return**	**unsigned**	

A.1.3 Identifiers

(6.4.2.1) *identifier:*
identifier-start
identifier identifier-continue

(6.4.2.1) *identifier-start:*
nondigit
XID_Start character
universal-character-name of class XID_Start

(6.4.2.1) *identifier-continue:*
digit
nondigit
XID_Continue character
universal-character-name of class XID_Continue

(6.4.2.1) *nondigit:* one of

```
_ a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

(6.4.2.1) *digit:* one of

```
0 1 2 3 4 5 6 7 8 9
```

A.1.4 Universal character names

(6.4.3) *universal-character-name:*

`\u` *hex-quad*
`\U` *hex-quad hex-quad*

(6.4.3) *hex-quad:*

hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

A.1.5 Constants

(6.4.4) *constant:*

integer-constant
floating-constant
enumeration-constant
character-constant
predefined-constant

(6.4.4.1) *integer-constant:*

decimal-constant integer-suffix$_{\text{opt}}$
octal-constant integer-suffix$_{\text{opt}}$
hexadecimal-constant integer-suffix$_{\text{opt}}$
binary-constant integer-suffix$_{\text{opt}}$

(6.4.4.1) *decimal-constant:*

nonzero-digit
decimal-constant `'`$_{\text{opt}}$ *digit*

(6.4.4.1) *octal-constant:*

`0`
octal-constant `'`$_{\text{opt}}$ *octal-digit*

(6.4.4.1) *hexadecimal-constant:*

hexadecimal-prefix hexadecimal-digit-sequence

(6.4.4.1) *binary-constant:*

binary-prefix binary-digit
binary-constant `'`$_{\text{opt}}$ *binary-digit*

(6.4.4.1) *hexadecimal-prefix:* one of

```
0x 0X
```

(6.4.4.1) *binary-prefix:* one of

```
0b 0B
```

(6.4.4.1) *nonzero-digit:* one of

```
1 2 3 4 5 6 7 8 9
```

(6.4.4.1) *octal-digit:* one of

```
0 1 2 3 4 5 6 7
```

hexadecimal-digit-sequence:

hexadecimal-digit
hexadecimal-digit-sequence `'`$_{\text{opt}}$ *hexadecimal-digit*

(6.4.4.1) *hexadecimal-digit:* one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

(6.4.4.1) *binary-digit:* one of
`0 1`

(6.4.4.1) *integer-suffix:*
unsigned-suffix long-suffix$_{\text{opt}}$
unsigned-suffix long-long-suffix
unsigned-suffix bit-precise-int-suffix
long-suffix unsigned-suffix$_{\text{opt}}$
long-long-suffix unsigned-suffix$_{\text{opt}}$
bit-precise-int-suffix unsigned-suffix$_{\text{opt}}$

(6.4.4.1) *bit-precise-int-suffix:* one of
`wb WB`

(6.4.4.1) *unsigned-suffix:* one of
`u U`

(6.4.4.1) *long-suffix:* one of
`l L`

(6.4.4.1) *long-long-suffix:* one of
`ll LL`

(6.4.4.2) *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

(6.4.4.2) *decimal-floating-constant:*
fractional-constant exponent-part$_{\text{opt}}$ *floating-suffix*$_{\text{opt}}$
digit-sequence exponent-part floating-suffix$_{\text{opt}}$

(6.4.4.2) *hexadecimal-floating-constant:*
hexadecimal-prefix hexadecimal-fractional-constant
binary-exponent-part floating-suffix$_{\text{opt}}$
hexadecimal-prefix hexadecimal-digit-sequence
binary-exponent-part floating-suffix$_{\text{opt}}$

(6.4.4.2) *fractional-constant:*
digit-sequence$_{\text{opt}}$ **`.`** *digit-sequence*
digit-sequence **`.`**

(6.4.4.2) *exponent-part:*
`e` *sign*$_{\text{opt}}$ *digit-sequence*
`E` *sign*$_{\text{opt}}$ *digit-sequence*

(6.4.4.2) *sign:* one of
`+ -`

(6.4.4.2) *digit-sequence:*
digit
digit-sequence **`'`**$_{\text{opt}}$ *digit*

(6.4.4.2) *hexadecimal-fractional-constant:*
hexadecimal-digit-sequence$_{\text{opt}}$ **`.`** *hexadecimal-digit-sequence*
hexadecimal-digit-sequence **`.`**

(6.4.4.2) *binary-exponent-part:*
`p` *sign*$_{\text{opt}}$ *digit-sequence*
`P` *sign*$_{\text{opt}}$ *digit-sequence*

(6.4.4.2) *floating-suffix:* one of
: **`f l F L df dd dl DF DD DL`**

(6.4.4.3) *enumeration-constant:*
: *identifier*

(6.4.4.4) *character-constant:*
: *encoding-prefix*$_{\text{opt}}$ **`'`** *c-char-sequence* **`'`**

(6.4.4.4) *encoding-prefix:* one of
: **`u8`**
: **`u`** **`U`** **`L`**

(6.4.4.4) *c-char-sequence:*
: *c-char*
: *c-char-sequence c-char*

(6.4.4.4) *c-char:*
: any member of the source character set except
: the single-quote `'`, backslash `\`, or new-line character
: *escape-sequence*

(6.4.4.4) *escape-sequence:*
: *simple-escape-sequence*
: *octal-escape-sequence*
: *hexadecimal-escape-sequence*
: *universal-character-name*

(6.4.4.4) *simple-escape-sequence:* one of
: `\' \" \? \\`
: `\a \b \f \n \r \t \v`

(6.4.4.4) *octal-escape-sequence:*
: `\` *octal-digit*
: `\` *octal-digit octal-digit*
: `\` *octal-digit octal-digit octal-digit*

(6.4.4.4) *hexadecimal-escape-sequence:*
: `\x` *hexadecimal-digit*
: *hexadecimal-escape-sequence hexadecimal-digit*

(6.4.4.5) *predefined-constant:*
: **`false`**
: **`true`**
: **`nullptr`**

A.1.6 String literals

(6.4.5) *string-literal:*
: *encoding-prefix*$_{\text{opt}}$ **`"`** *s-char-sequence*$_{\text{opt}}$ **`"`**

(6.4.5) *s-char-sequence:*
: *s-char*
: *s-char-sequence s-char*

(6.4.5) *s-char:*
: any member of the source character set except
: the double-quote `"`, backslash `\`, or new-line character
: *escape-sequence*

A.1.7 Punctuators

(6.4.6) *punctuator:* one of

```
[   ]   (  )   {   }   .    ->
++   --   &   *   +   -   ~   !
/   %   <<   >>   <   >   <=   >=   ==   !=   ^   |   &&   ||
?   :   ::   ;   ...
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
,   #   ##
<:   :>   <%   %>   %:   %:%:
```

A.1.8 Header names

(6.4.7) *header-name:*

< *h-char-sequence* **>**
" *q-char-sequence* **"**

(6.4.7) *h-char-sequence:*

h-char
h-char-sequence h-char

(6.4.7) *h-char:*

any member of the source character set except
the new-line character and >

(6.4.7) *q-char-sequence:*

q-char
q-char-sequence q-char

(6.4.7) *q-char:*

any member of the source character set except
the new-line character and **"**

A.1.9 Preprocessing numbers

(6.4.8) *pp-number:*

digit
. *digit*
pp-number identifier-continue
pp-number **'** *digit*
pp-number **'** *nondigit*
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

A.2 Phrase structure grammar

A.2.1 Expressions

(6.5.1) *primary-expression:*

identifier
constant
string-literal
(*expression* **)**
generic-selection

(6.5.1.1) *generic-selection:*

_Generic (*assignment-expression* **,** *generic-assoc-list* **)**

(6.5.1.1) *generic-assoc-list:*

generic-association
generic-assoc-list **,** *generic-association*

(6.5.1.1) *generic-association:*
type-name **:** *assignment-expression*
default : *assignment-expression*

(6.5.2) *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{\text{opt}}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**
compound-literal

(6.5.2) *argument-expression-list:*
assignment-expression
argument-expression-list **,** *assignment-expression*

(6.5.2.5) *compound-literal:*
(*storage-class-specifiers*$_{\text{opt}}$ *type-name* **)** *braced-initializer*

(6.5.2.5) *storage-class-specifiers:*
storage-class-specifier
storage-class-specifiers storage-class-specifier

(6.5.3) *unary-expression:*
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof (*type-name* **)**
alignof (*type-name* **)**

(6.5.3) *unary-operator:* one of
& * + - ~ !

(6.5.4) *cast-expression:*
unary-expression
(*type-name* **)** *cast-expression*

(6.5.5) *multiplicative-expression:*
cast-expression
multiplicative-expression ***** *cast-expression*
multiplicative-expression **/** *cast-expression*
multiplicative-expression **%** *cast-expression*

(6.5.6) *additive-expression:*
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

(6.5.7) *shift-expression:*
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

(6.5.8) *relational-expression:*
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

(6.5.9) *equality-expression:*
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

(6.5.10) *AND-expression:*
equality-expression
AND-expression **&** *equality-expression*

(6.5.11) *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression **^** *AND-expression*

(6.5.12) *inclusive-OR-expression:*
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

(6.5.13) *logical-AND-expression:*
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

(6.5.14) *logical-OR-expression:*
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

(6.5.15) *conditional-expression:*
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

(6.5.16) *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

(6.5.16) *assignment-operator:* one of
= *= /= %= += -= <<= >>= &= ^= |=

(6.5.17) *expression:*
assignment-expression
expression **,** *assignment-expression*

(6.6) *constant-expression:*
conditional-expression

A.2.2 Declarations

(6.7) *declaration:*
declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**
attribute-specifier-sequence declaration-specifiers init-declarator-list **;**
static_assert-declaration
attribute-declaration

(6.7) *declaration-specifiers:*
declaration-specifier attribute-specifier-sequence$_{\text{opt}}$
declaration-specifier declaration-specifiers

(6.7) *declaration-specifier:*
storage-class-specifier
type-specifier-qualifier
function-specifier

(6.7) *init-declarator-list:*
init-declarator
init-declarator-list **,** *init-declarator*

(6.7) *init-declarator:*

declarator
declarator **=** *initializer*

(6.7) *attribute-declaration:*

attribute-specifier-sequence **;**

(6.7.1) *storage-class-specifier:*

auto
constexpr
extern
register
static
thread_local
typedef

(6.7.2) *type-specifier:*

void
char
short
int
long
float
double
signed
unsigned
_BitInt (*constant-expression*)
bool
_Complex
_Decimal32
_Decimal64
_Decimal128
atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name
typeof-specifier

(6.7.2.1) *struct-or-union-specifier:*

struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**
struct-or-union attribute-specifier-sequence$_{\text{opt}}$ *identifier*

(6.7.2.1) *struct-or-union:*

struct
union

[-2ex]

(6.7.2.1) *member-declaration-list:*

member-declaration
member-declaration-list member-declaration

(6.7.2.1) *member-declaration:*

attribute-specifier-sequence$_{\text{opt}}$ *specifier-qualifier-list member-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration

(6.7.2.1) *specifier-qualifier-list:*

type-specifier-qualifier attribute-specifier-sequence$_{\text{opt}}$
type-specifier-qualifier specifier-qualifier-list

(6.7.2.1) *type-specifier-qualifier:*
type-specifier
type-qualifier
alignment-specifier

(6.7.2.1) *member-declarator-list:*
member-declarator
member-declarator-list **,** *member-declarator*

(6.7.2.1) *member-declarator:*
declarator
declarator$_{\text{opt}}$ **:** *constant-expression*

(6.7.2.2) *enum-specifier:*
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ *enum-type-specifier*$_{\text{opt}}$
{ *enumerator-list* **}**
enum *attribute-specifier-sequence*$_{\text{opt}}$ *identifier*$_{\text{opt}}$ *enum-type-specifier*$_{\text{opt}}$
{ *enumerator-list* **, }**
enum *identifier* *enum-type-specifier*$_{\text{opt}}$

(6.7.2.2) *enumerator-list:*
enumerator
enumerator-list **,** *enumerator*

(6.7.2.2) *enumerator:*
enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$
enumeration-constant *attribute-specifier-sequence*$_{\text{opt}}$ **=** *constant-expression*
(6.7.2.2) *enum-type-specifier:*
: *specifier-qualifier-list*

(6.7.2.4) *atomic-type-specifier:*
_Atomic (*type-name* **)**

(6.7.2.5) *typeof-specifier:*
typeof (*typeof-specifier-argument* **)**
typeof_unqual (*typeof-specifier-argument* **)**
(6.7.2.5) *typeof-specifier-argument:*
expression
type-name

(6.7.3) *type-qualifier:*
const
restrict
volatile
_Atomic

(6.7.4) *function-specifier:*
inline
_Noreturn
[-7ex]

(6.7.5) *alignment-specifier:*
alignas (*type-name* **)**
alignas (*constant-expression* **)**

(6.7.6) *declarator:*
pointer$_{\text{opt}}$ *direct-declarator*

(6.7.6) *direct-declarator:*
identifier *attribute-specifier-sequence*$_{\text{opt}}$
(*declarator* **)**
array-declarator *attribute-specifier-sequence*$_{\text{opt}}$
function-declarator *attribute-specifier-sequence*$_{\text{opt}}$

(6.7.6) *array-declarator:*
direct-declarator **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**
direct-declarator **[** `static` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* `static` *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{\text{opt}}$ ***** **]**

(6.7.6) *function-declarator:*
direct-declarator **(** *parameter-type-list*$_{opt}$ **)**

(6.7.6) *pointer:*
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$
***** *attribute-specifier-sequence*$_{\text{opt}}$ *type-qualifier-list*$_{\text{opt}}$ *pointer*

(6.7.6) *type-qualifier-list:*
type-qualifier
type-qualifier-list type-qualifier

(6.7.6) *parameter-type-list:*
parameter-list
parameter-list **, ...**
...

(6.7.6) *parameter-list:*
parameter-declaration
parameter-list **,** *parameter-declaration*

(6.7.6) *parameter-declaration:*
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers declarator*
attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers abstract-declarator*$_{\text{opt}}$

(6.7.7) *type-name:*
specifier-qualifier-list abstract-declarator$_{\text{opt}}$

(6.7.7) *abstract-declarator:*
pointer
pointer$_{\text{opt}}$ *direct-abstract-declarator*

(6.7.7) *direct-abstract-declarator:*
(*abstract-declarator* **)**
array-abstract-declarator attribute-specifier-sequence$_{\text{opt}}$
function-abstract-declarator attribute-specifier-sequence$_{\text{opt}}$

(6.7.7) *array-abstract-declarator:*
direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ **]**
direct-abstract-declarator$_{\text{opt}}$ **[** `static` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* **]**
direct-abstract-declarator$_{\text{opt}}$ **[** *type-qualifier-list* `static` *assignment-expression* **]**
direct-abstract-declarator$_{\text{opt}}$ **[*]**

(6.7.7) *function-abstract-declarator:*
direct-abstract-declarator$_{\text{opt}}$ **(** *parameter-type-list*$_{\text{opt}}$ **)**

(6.7.8) *typedef-name:*
identifier

(6.7.10) *braced-initializer:*
{ }
{ *initializer-list* **}**
{ *initializer-list* **, }**

(6.7.10) *initializer:*
assignment-expression
braced-initializer

(6.7.10) *initializer-list:*
designation$_{\text{opt}}$ *initializer*
initializer-list **,** *designation*$_{\text{opt}}$ *initializer*

(6.7.10) *designation:*
designator-list **=**

(6.7.10) *designator-list:*
designator
designator-list designator

(6.7.10) *designator:*
[*constant-expression* **]**
. *identifier*

(6.7.11) *static_assert-declaration:*
static_assert (*constant-expression* **,** *string-literal* **) ;**
static_assert (*constant-expression* **) ;**

(6.7.12.1) *attribute-specifier-sequence:*
attribute-specifier-sequence$_{\text{opt}}$ *attribute-specifier*

(6.7.12.1) *attribute-specifier:*
[[*attribute-list* **]]**

(6.7.12.1) *attribute-list:*
attribute$_{\text{opt}}$
attribute-list **,** *attribute*$_{\text{opt}}$

(6.7.12.1) *attribute:*
attribute-token attribute-argument-clause$_{\text{opt}}$

(6.7.12.1) *attribute-token:*
standard-attribute
attribute-prefixed-token

(6.7.12.1) *standard-attribute:*
identifier

(6.7.12.1) *attribute-prefixed-token:*
attribute-prefix **::** *identifier*

(6.7.12.1) *attribute-prefix:*
identifier

(6.7.12.1) *attribute-argument-clause:*
(*balanced-token-sequence*$_{\text{opt}}$ **)**

(6.7.12.1) *balanced-token-sequence:*
balanced-token
balanced-token-sequence balanced-token

(6.7.12.1) *balanced-token:*
(*balanced-token-sequence*$_{\text{opt}}$ **)**
[*balanced-token-sequence*$_{\text{opt}}$ **]**
{ *balanced-token-sequence*$_{\text{opt}}$ **}**
any token other than a parenthesis, a bracket, or a brace

A.2.3 Statements

(6.8) *statement:*
labeled-statement
unlabeled-statement

(6.8) *unlabeled-statement:*
expression-statement
attribute-specifier-sequence$_{\text{opt}}$ *primary-block*
attribute-specifier-sequence$_{\text{opt}}$ *jump-statement*

(6.8) *primary-block:*
compound-statement
selection-statement
iteration-statement

(6.8) *secondary-block:*
statement

(6.8.1) *label:*
attribute-specifier-sequence$_{\text{opt}}$ *identifier* **:**
attribute-specifier-sequence$_{\text{opt}}$ **case** *constant-expression* **:**
attribute-specifier-sequence$_{\text{opt}}$ **default :**

(6.8.1) *labeled-statement:*
label statement

(6.8.2) *compound-statement:*
{ *block-item-list*$_{\text{opt}}$ **}**

(6.8.2) *block-item-list:*
block-item
block-item-list block-item

(6.8.2) *block-item:*
declaration
unlabeled-statement
label

(6.8.3) *expression-statement:*
expression$_{\text{opt}}$ **;**
attribute-specifier-sequence expression **;**

[-6ex]

(6.8.4) *selection-statement:*
if (*expression* **)** *secondary-block*
if (*expression* **)** *secondary-block* **else** *secondary-block*
switch (*expression* **)** *secondary-block*

[-6ex]

(6.8.5) *iteration-statement:*
while (*expression* **)** *secondary-block*
do *secondary-block* **while (** *expression* **) ;**
for (*expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **)** *secondary-block*
for (*declaration expression*$_{\text{opt}}$ **;** *expression*$_{\text{opt}}$ **)** *secondary-block*

[-6ex]

(6.8.6) *jump-statement:*
goto *identifier* **;**
continue ;
break ;
return *expression*$_{\text{opt}}$ **;**

[-6ex]

A.2.4 External definitions

(6.9) *translation-unit:*
external-declaration
translation-unit external-declaration

(6.9) *external-declaration:*
function-definition
declaration

(6.9.1) *function-definition:*

attribute-specifier-sequence$_{\text{opt}}$ *declaration-specifiers declarator function-body*

(6.9.1) *function-body:*

compound-statement

A.3 Preprocessing directives

(6.10) *preprocessing-file:*

group$_{\text{opt}}$

(6.10) *group:*

group-part
group group-part

(6.10) *group-part:*

if-section
control-line
text-line
*non-directive*

(6.10) *if-section:*

if-group elif-groups$_{\text{opt}}$ *else-group*$_{\text{opt}}$ *endif-line*

(6.10) *if-group:*

 # if *constant-expression new-line group*$_{\text{opt}}$
 # ifdef *identifier new-line group*$_{\text{opt}}$
 # ifndef *identifier new-line group*$_{\text{opt}}$

(6.10) *elif-groups:*

elif-group
elif-groups elif-group

(6.10) *elif-group:*

 # elif *constant-expression new-line group*$_{\text{opt}}$
 # elifdef *identifier new-line group*$_{\text{opt}}$
 # elifndef *identifier new-line group*$_{\text{opt}}$

(6.10) *else-group:*

 # else *new-line group*$_{\text{opt}}$

(6.10) *endif-line:*

 # endif *new-line*

(6.10) *control-line:*

 # include *pp-tokens new-line*
 # embed *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list*$_{\text{opt}}$ **)** *replacement-list new-line*
 # define *identifier lparen* **...)** *replacement-list new-line*
 # define *identifier lparen identifier-list* **, ...)** *replacement-list new-line*
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens*$_{\text{opt}}$ *new-line*
 # warning *pp-tokens*$_{\text{opt}}$ *new-line*
 # pragma *pp-tokens*$_{\text{opt}}$ *new-line*
*new-line*

(6.10) *text-line:*

pp-tokens$_{\text{opt}}$ *new-line*

(6.10) *non-directive:*

pp-tokens new-line

(6.10) lparen:

a **(** character not immediately preceded by white space

(6.10) replacement-list:

pp-tokens$_{\text{opt}}$

(6.10) pp-tokens:

preprocessing-token
pp-tokens preprocessing-token

(6.10) new-line:

the new-line character

(6.10) identifier-list:

identifier
identifier-list **,** *identifier*

(6.10) pp-parameter:

pp-parameter-name pp-parameter-clause$_{\text{opt}}$

(6.10) pp-parameter-name:

pp-standard-parameter
pp-prefixed-parameter

(6.10) pp-standard-parameter:

identifier

(6.10) pp-prefixed-parameter:

identifier **::** *identifier*

(6.10) pp-parameter-clause:

(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**

(6.10) pp-balanced-token-sequence:

pp-balanced-token
pp-balanced-token-sequence pp-balanced-token

(6.10) pp-balanced-token:

(*pp-balanced-token-sequence*$_{\text{opt}}$ **)**
[*pp-balanced-token-sequence*$_{\text{opt}}$ **]**
{ *pp-balanced-token-sequence*$_{\text{opt}}$ **}**
any pp-token other than a parenthesis, a bracket, or a brace

(6.10) embed-parameter-sequence:

pp-parameter
embed-parameter-sequence pp-parameter

defined-macro-expression:
  `defined` *identifier*
  `defined ( ` *identifier* **` )`**

h-preprocessing-token:
  any *preprocessing-token* other than **`>`**

h-pp-tokens:
  h-preprocessing-token
  h-pp-tokens h-preprocessing-token

header-name-tokens:
  string-literal
  `<` *h-pp-tokens* **`>`**

has-include-expression:
  `__has_include (` *header-name* **`)`**
  `__has_include (` *header-name-tokens* **`)`**

has-embed-expression:
  `__has_embed (` *header-name embed-parameter-sequence*$_{opt}$ **`)`**
  `__has_embed (` *header-name-tokens pp-balanced-token-sequence*$_{opt}$ **`)`**

has-c-attribute-express:
  `__has_c_attribute (` *pp-tokens* **`)`**

va-opt-replacement:
  `__VA_OPT__ (` *pp-tokens*$_{opt}$ **`)`**

(6.10.7) *standard-pragma:*
  `# pragma` `STDC FP_CONTRACT` *on-off-switch*
  `# pragma` `STDC FENV_ACCESS` *on-off-switch*
  `# pragma` `STDC FENV_DEC_ROUND` *dec-direction*
  `# pragma` `STDC FENV_ROUND` *direction*
  `# pragma` `STDC CX_LIMITED_RANGE` *on-off-switch*

(6.10.7) *on-off-switch:* one of
  `ON` `OFF` `DEFAULT`

(6.10.7) *direction:* one of
  `FE_DOWNWARD` `FE_TONEAREST` `FE_TONEARESTFROMZERO`
  `FE_TOWARDZERO` `FE_UPWARD` `FE_DYNAMIC`

(6.10.7) *dec-direction:* one of
  `FE_DEC_DOWNWARD` `FE_DEC_TONEAREST` `FE_DEC_TONEARESTFROMZERO`
  `FE_DEC_TOWARDZERO` `FE_DEC_UPWARD` `FE_DEC_DYNAMIC`

A.4 Floating-point subject sequence

A.4.1 NaN `char` sequence

(7.24.1.5) *n-char-sequence*:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit

A.4.2 NaN `wchar_t` sequence

(7.31.4.1.2) *n-wchar-sequence*:
  digit
  nondigit
  n-wchar-sequence digit
  n-wchar-sequence nondigit

A.5 Decimal floating-point subject sequence

A.5.1 NaN decimal `char` sequence

(7.24.1.6) *d-char-sequence*:
digit
nondigit
d-char-sequence digit
d-char-sequence nondigit

A.5.2 NaN decimal `wchar_t` sequence

(7.31.4.1.3) *d-wchar-sequence*:
digit
nondigit
d-wchar-sequence digit
d-wchar-sequence nondigit

Annex B
(informative)
Library summary

B.1 Diagnostics <assert.h>

NDEBUG

```
void assert(scalar expression);
```

B.2 Complex <complex.h>

__STDC_NO_COMPLEX__
complex
_Complex_I
imaginary
_Imaginary_I
I

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
double complex clog(double complex z);
float complex clogf(float complex z);
```

```
long double complex clogl(long double complex z);
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

B.3 Character handling <ctype.h>

```
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

B.4 Errors <errno.h>

`EDOM` `EILSEQ` `ERANGE` `errno`

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<errno.h>`:

`errno_t`

B.5 Floating-point environment <fenv.h>

```
fenv_t              FE_OVERFLOW         FE_TOWARDZERO
fexcept_t           FE_UNDERFLOW        FE_UPWARD
FE_DIVBYZERO        FE_ALL_EXCEPT       FE_DFL_ENV
FE_INEXACT          FE_DOWNWARD
FE_INVALID          FE_TONEAREST
```

```
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC FENV_ROUND direction
#pragma STDC FENV_ROUND FE_DYNAMIC
int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexceptflag(const fexcept_t * flagp, int excepts);
int fetestexcept(int excepts);
int fegetmode(femode_t *modep);
int fegetround(void);
int fesetmode(const femode_t *modep);
int fesetround(int rnd);
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

Only if the implementation defines `__STDC_IEC_60559_DFP__`:

```
FE_DEC_DOWNWARD     FE_DEC_TONEARESTFROMZERO    FE_DEC_UPWARD
FE_DEC_TONEAREST    FE_DEC_TOWARDZERO
```

```
#pragma STDC FENV_DEC_ROUND dec-direction
int fe_dec_getround(void);
int fe_dec_setround(int rnd);
```

B.6 Characteristics of floating types `<float.h>`

```
FLT_ROUNDS          LDBL_DIG            DBL_NORM_MAX
FLT_EVAL_METHOD     FLT_MIN_EXP         LDBL_NORM_MAX
FLT_HAS_SUBNORM     DBL_MIN_EXP         FLT_EPSILON
DBL_HAS_SUBNORM     LDBL_MIN_EXP        DBL_EPSILON
LDBL_HAS_SUBNORM    FLT_MIN_10_EXP      LDBL_EPSILON
FLT_RADIX           DBL_MIN_10_EXP      FLT_MIN
FLT_MANT_DIG        LDBL_MIN_10_EXP     DBL_MIN
DBL_MANT_DIG        FLT_MAX_EXP         LDBL_MIN
LDBL_MANT_DIG       DBL_MAX_EXP         FLT_SNAN
FLT_DECIMAL_DIG     LDBL_MAX_EXP        DBL_SNAN
DBL_DECIMAL_DIG     FLT_MAX_10_EXP      LDBL_SNAN
LDBL_DECIMAL_DIG    DBL_MAX_10_EXP      FLT_TRUE_MIN
DECIMAL_DIG         LDBL_MAX_10_EXP     DBL_TRUE_MIN
FLT_IS_IEC_60559    FLT_MAX             LDBL_TRUE_MIN
DBL_IS_IEC_60559    DBL_MAX             INFINITY
FLT_DIG             LDBL_MAX            NAN
DBL_DIG             FLT_NORM_MAX
```

The following macro is provided only if the program defines `__STDC_WANT_IEC_60559_EXT__` before inclusion of the header `<float.h>`:

CR_DECIMAL_DIG

B.6.1 Characteristics of decimal floating types

1 The following macros are provided only if the implementation defines __STDC_IEC_60559_DFP__. *N* is 32, 64 and 128.

DEC_INFINITY	DEC*N*_MANT_DIG	DEC*N*_MIN_EXP	DEC*N*_SNAN
DEC_NAN	DEC*N*_MAX_EXP	DEC*N*_MIN	
DEC*N*_EPSILON	DEC*N*_MAX	DEC*N*_TRUE_MIN	

B.7 Format conversion of integer types <inttypes.h>

imaxdiv_t

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR
PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR
SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR
SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

```
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
```

B.8 Alternative spellings <iso646.h>

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

B.9 Sizes of integer types <limits.h>

BOOL_WIDTH	UINT_WIDTH	UCHAR_MAX	INT_MAX
CHAR_BIT	LONG_WIDTH	CHAR_MIN	UINT_MAX
CHAR_WIDTH	ULONG_WIDTH	CHAR_MAX	LONG_MIN
SCHAR_WIDTH	LLONG_WIDTH	MB_LEN_MAX	LONG_MAX
UCHAR_WIDTH	ULLONG_WIDTH	SHRT_MIN	ULONG_MAX
SHRT_WIDTH	BOOL_MAX	SHRT_MAX	LLONG_MIN
USHRT_WIDTH	SCHAR_MIN	USHRT_MAX	LLONG_MAX
INT_WIDTH	SCHAR_MAX	INT_MIN	ULLONG_MAX

B.10 Localization <locale.h>

```
struct lconv     LC_ALL           LC_CTYPE         LC_NUMERIC
NULL             LC_COLLATE       LC_MONETARY      LC_TIME
```

```
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);
```

B.11 Mathematics <math.h>

```
float_t          FP_INFINITE      FP_FAST_FMAL
double_t         FP_NAN           FP_ILOGB0
HUGE_VAL         FP_NORMAL        FP_ILOGBNAN
HUGE_VALF        FP_SUBNORMAL     MATH_ERRNO
HUGE_VALL        FP_ZERO          MATH_ERREXCEPT
INFINITY         FP_FAST_FMA      math_errhandling
NAN              FP_FAST_FMAF
```

```
#pragma STDC FP_CONTRACT on-off-switch
int fpclassify(real-floating x);
int iscanonical(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);
int signbit(real-floating x);
int issignaling(real-floating x);
int issubnormal(real-floating x);
int iszero(real-floating x);
double acos(double x);
float acosf(float x);
long double acosl(long double x);
double asin(double x);
float asinf(float x);
long double asinl(long double x);
double atan(double x);
float atanf(float x);
long double atanl(long double x);
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
double cos(double x);
float cosf(float x);
long double cosl(long double x);
double sin(double x);
float sinf(float x);
long double sinl(long double x);
double tan(double x);
float tanf(float x);
long double tanl(long double x);
double acospi(double x);
float acospif(float x);
long double acospil(long double x);
double asinpi(double x);
float asinpif(float x);
long double asinpil(long double x);
double atanpi(double x);
float atanpif(float x);
long double atanpil(long double x);
double atan2pi(double y, double x);
```

```
float atan2pif(float y, float x);
long double atan2pil(long double y, long double x);
double cospi(double x);
float cospif(float x);
long double cospil(long double x);
double sinpi(double x);
float sinpif(float x);
long double sinpil(long double x);
double tanpi(double x);
float tanpif(float x);
long double tanpil(long double x);
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
double exp(double x);
float expf(float x);
long double expl(long double x);
double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
double exp10m1(double x);
float exp10m1f(float x);
long double exp10m1l(long double x);
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
double exp2m1(double x);
float exp2m1f(float x);
long double exp2m1l(long double x);
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
double frexp(double value, int *p);
float frexpf(float value, int *p);
long double frexpl(long double value, int *p);
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
double ldexp(double x, int p);
float ldexpf(float x, int p);
long double ldexpl(long double x, int p);
long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
double log(double x);
float logf(float x);
```

```
long double logl(long double x);
double log10(double x);
float log10f(float x);
long double log10l(long double x);
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
double logp1(double x);
float logp1f(float x);
long double logp1l(long double x);
double log2(double x);
float log2f(float x);
long double log2l(long double x);
double log2p1(double x);
float log2p1f(float x);
long double log2p1l(long double x);
double logb(double x);
float logbf(float x);
long double logbl(long double x);
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
double compoundn(double x, long long int n);
float compoundnf(float x, long long int n);
long double compoundnl(long double x, long long int n);
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
double pown(double x, long long int n);
float pownf(float x, long long int n);
long double pownl(long double x, long long int n);
double powr(double y, double x);
float powrf(float y, float x);
long double powrl(long double y, long double x);
double rootn(double x, long long int n);
float rootnf(float x, long long int n);
long double rootnl(long double x, long long int n);
double rsqrt(double x);
float rsqrtf(float x);
long double rsqrtl(long double x);
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

```
double erf(double x);
float erff(float x);
long double erfl(long double x);
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
double floor(double x);
float floorf(float x);
long double floorl(long double x);
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
double rint(double x);
float rintf(float x);
long double rintl(long double x);
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
double round(double x);
float roundf(float x);
long double roundl(long double x);
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
double roundeven(double x);
float roundevenf(float x);
long double roundevenl(long double x);
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
double fromfp(double x, int rnd, unsigned int width);
float fromfpf(float x, int rnd, unsigned int width);
long double fromfpl(long double x, int rnd, unsigned int width);
double ufromfp(double x, int rnd, unsigned int width);
float ufromfpf(float x, int rnd, unsigned int width);
long double ufromfpl(long double x, int rnd, unsigned int width);
double fromfpx(double x, int rnd, unsigned int width);
float fromfpxf(float x, int rnd, unsigned int width);
long double fromfpxl(long double x, int rnd, unsigned int width);
double ufromfpx(double x, int rnd, unsigned int width);
float ufromfpxf(float x, int rnd, unsigned int width);
long double ufromfpxl(long double x, int rnd, unsigned int width);
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
double remainder(double x, double y);
```

```
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);
double nextdown(double x);
float nextdownf(float x);
long double nextdownl(long double x);
int canonicalize(double * cx, const double * x);
int canonicalizef(float * cx, const float * x);
int canonicalizel(long double * cx, const long double * x);
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
double fmaximum(double x, double y);
float fmaximumf(float x, float y);
long double fmaximuml(long double x, long double y);
double fminimum(double x, double y);
float fminimumf(float x, float y);
long double fminimuml(long double x, long double y);
double fmaximum_mag(double x, double y);
float fmaximum_magf(float x, float y);
long double fmaximum_magl(long double x, long double y);
double fminimum_mag(double x, double y);
float fminimum_magf(float x, float y);
long double fminimum_magl(long double x, long double y);
double fmaximum_num(double x, double y);
float fmaximum_numf(float x, float y);
long double fmaximum_numl(long double x, long double y);
double fminimum_num(double x, double y);
float fminimum_numf(float x, float y);
long double fminimum_numl(long double x, long double y);
double fmaximum_mag_num(double x, double y);
float fmaximum_mag_numf(float x, float y);
long double fmaximum_mag_numl(long double x, long double y);
double fminimum_mag_num(double x, double y);
float fminimum_mag_numf(float x, float y);
long double fminimum_mag_numl(long double x, long double y);
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
```

```
long double fmal(long double x, long double y, long double z);
float fadd(double x, double y);
float faddl(long double x, long double y);
double daddl(long double x, long double y);
float fsub(double x, double y);
float fsubl(long double x, long double y);
double dsubl(long double x, long double y);
float fmul(double x, double y);
float fmull(long double x, long double y);
double dmull(long double x, long double y);
float fdiv(double x, double y);
float fdivl(long double x, long double y);
double ddivl(long double x, long double y);
float ffma(double x, double y, double z);
float ffmal(long double x, long double y, long double z);
double dfmal(long double x, long double y, long double z);
float fsqrt(double x);
float fsqrtl(long double x);
double dsqrtl(long double x);
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
int iseqsig(real-floating x, real-floating y);
```

Only if the implementation defines `__STDC_IEC_60559_DFP__`:

```
_Decimal32 acosd32(_Decimal32 x);
_Decimal64 acosd64(_Decimal64 x);
_Decimal128 acosd128(_Decimal128 x);
_Decimal32 asind32(_Decimal32 x);
_Decimal64 asind64(_Decimal64 x);
_Decimal128 asind128(_Decimal128 x);
_Decimal32 atand32(_Decimal32 x);
_Decimal64 atand64(_Decimal64 x);
_Decimal128 atand128(_Decimal128 x);
_Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
_Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
_Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);
_Decimal32 cosd32(_Decimal32 x);
_Decimal64 cosd64(_Decimal64 x);
_Decimal128 cosd128(_Decimal128 x);
_Decimal32 sind32(_Decimal32 x);
_Decimal64 sind64(_Decimal64 x);
_Decimal128 sind128(_Decimal128 x);
_Decimal32 tand32(_Decimal32 x);
_Decimal64 tand64(_Decimal64 x);
_Decimal128 tand128(_Decimal128 x);
_Decimal32 acospid32(_Decimal32 x);
_Decimal64 acospid64(_Decimal64 x);
_Decimal128 acospid128(_Decimal128 x);
_Decimal32 asinpid32(_Decimal32 x);
_Decimal64 asinpid64(_Decimal64 x);
_Decimal128 asinpid128(_Decimal128 x);
_Decimal32 atanpid32(_Decimal32 x);
_Decimal64 atanpid64(_Decimal64 x);
_Decimal128 atanpid128(_Decimal128 x);
_Decimal32 atan2pid32(_Decimal32 y, _Decimal32 x);
_Decimal64 atan2pid64(_Decimal64 y, _Decimal64 x);
```

```
_Decimal128 atan2pid128(_Decimal128 y, _Decimal128 x);
_Decimal32 cospid32(_Decimal32 x);
_Decimal64 cospid64(_Decimal64 x);
_Decimal128 cospid128(_Decimal128 x);
_Decimal32 sinpid32(_Decimal32 x);
_Decimal64 sinpid64(_Decimal64 x);
_Decimal128 sinpid128(_Decimal128 x);
_Decimal32 tanpid32(_Decimal32 x);
_Decimal64 tanpid64(_Decimal64 x);
_Decimal128 tanpid128(_Decimal128 x);
_Decimal32 acoshd32(_Decimal32 x);
_Decimal64 acoshd64(_Decimal64 x);
_Decimal128 acoshd128(_Decimal128 x);
_Decimal32 asinhd32(_Decimal32 x);
_Decimal64 asinhd64(_Decimal64 x);
_Decimal128 asinhd128(_Decimal128 x);
_Decimal32 atanhd32(_Decimal32 x);
_Decimal64 atanhd64(_Decimal64 x);
_Decimal128 atanhd128(_Decimal128 x);
_Decimal32 coshd32(_Decimal32 x);
_Decimal64 coshd64(_Decimal64 x);
_Decimal128 coshd128(_Decimal128 x);
_Decimal32 sinhd32(_Decimal32 x);
_Decimal64 sinhd64(_Decimal64 x);
_Decimal128 sinhd128(_Decimal128 x);
_Decimal32 tanhd32(_Decimal32 x);
_Decimal64 tanhd64(_Decimal64 x);
_Decimal128 tanhd128(_Decimal128 x);
_Decimal32 expd32(_Decimal32 x);
_Decimal64 expd64(_Decimal64 x);
_Decimal128 expd128(_Decimal128 x);
_Decimal32 exp10d32(_Decimal32 x);
_Decimal64 exp10d64(_Decimal64 x);
_Decimal128 exp10d128(_Decimal128 x);
_Decimal32 exp10m1d32(_Decimal32 x);
_Decimal64 exp10m1d64(_Decimal64 x);
_Decimal128 exp10m1d128(_Decimal128 x);
_Decimal32 exp2d32(_Decimal32 x);
_Decimal64 exp2d64(_Decimal64 x);
_Decimal128 exp2d128(_Decimal128 x);
_Decimal32 exp2m1d32(_Decimal32 x);
_Decimal64 exp2m1d64(_Decimal64 x);
_Decimal128 exp2m1d128(_Decimal128 x);
_Decimal32 expm1d32(_Decimal32 x);
_Decimal64 expm1d64(_Decimal64 x);
_Decimal128 expm1d128(_Decimal128 x);
_Decimal32 frexpd32(_Decimal32 value, int *p);
_Decimal64 frexpd64(_Decimal64 value, int *p);
_Decimal128 frexpd128(_Decimal128 value, int *p);
int ilogbd32(_Decimal32 x);
int ilogbd64(_Decimal64 x);
int ilogbd128(_Decimal128 x);
_Decimal32 ldexpd32(_Decimal32 x, int p);
_Decimal64 ldexpd64(_Decimal64 x, int p);
_Decimal128 ldexpd128(_Decimal128 x, int p);
long int llogbd32(_Decimal32 x);
long int llogbd64(_Decimal64 x);
long int llogbd128(_Decimal128 x);
_Decimal32 logd32(_Decimal32 x);
_Decimal64 logd64(_Decimal64 x);
_Decimal128 logd128(_Decimal128 x);
```

```
_Decimal32 log10d32(_Decimal32 x);
_Decimal64 log10d64(_Decimal64 x);
_Decimal128 log10d128(_Decimal128 x);
_Decimal32 log10p1d32(_Decimal32 x);
_Decimal64 log10p1d64(_Decimal64 x);
_Decimal128 log10p1d128(_Decimal128 x);
_Decimal32 log1pd32(_Decimal32 x);
_Decimal64 log1pd64(_Decimal64 x);
_Decimal128 log1pd128(_Decimal128 x);
_Decimal32 logp1d32(_Decimal32 x);
_Decimal64 logp1d64(_Decimal64 x);
_Decimal128 logp1d128(_Decimal128 x);
_Decimal32 log2d32(_Decimal32 x);
_Decimal64 log2d64(_Decimal64 x);
_Decimal128 log2d128(_Decimal128 x);
_Decimal32 log2p1d32(_Decimal32 x);
_Decimal64 log2p1d64(_Decimal64 x);
_Decimal128 log2p1d128(_Decimal128 x);
_Decimal32 logbd32(_Decimal32 x);
_Decimal64 logbd64(_Decimal64 x);
_Decimal128 logbd128(_Decimal128 x);
_Decimal32 modfd32(_Decimal32 x, _Decimal32 *iptr);
_Decimal64 modfd64(_Decimal64 x, _Decimal64 *iptr);
_Decimal128 modfd128(_Decimal128 x, _Decimal128 *iptr);
_Decimal32 scalbnd32(_Decimal32 x, int n);
_Decimal64 scalbnd64(_Decimal64 x, int n);
_Decimal128 scalbnd128(_Decimal128 x, int n);
_Decimal32 scalblnd32(_Decimal32 x, long int n);
_Decimal64 scalblnd64(_Decimal64 x, long int n);
_Decimal128 scalblnd128(_Decimal128 x, long int n);
_Decimal32 cbrtd32(_Decimal32 x);
_Decimal64 cbrtd64(_Decimal64 x);
_Decimal128 cbrtd128(_Decimal128 x);
_Decimal32 compoundnd32(_Decimal32 x, long long int n);
_Decimal64 compoundnd64(_Decimal64 x, long long int n);
_Decimal128 compoundnd128(_Decimal128 x, long long int n);
_Decimal32 fabsd32(_Decimal32 x);
_Decimal64 fabsd64(_Decimal64 x);
_Decimal128 fabsd128(_Decimal128 x);
_Decimal32 hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal64 hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);
_Decimal32 powd32(_Decimal32 x, _Decimal32 y);
_Decimal64 powd64(_Decimal64 x, _Decimal64 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);
_Decimal32 pownd32(_Decimal32 x, long long int n);
_Decimal64 pownd64(_Decimal64 x, long long int n);
_Decimal128 pownd128(_Decimal128 x, long long int n);
_Decimal32 powrd32(_Decimal32 y, _Decimal32 x);
_Decimal64 powrd64(_Decimal64 y, _Decimal64 x);
_Decimal128 powrd128(_Decimal128 y, _Decimal128 x);
_Decimal32 rootnd32(_Decimal32 x, long long int n);
_Decimal64 rootnd64(_Decimal64 x, long long int n);
_Decimal128 rootnd128(_Decimal128 x, long long int n);
_Decimal32 rsqrtd32(_Decimal32 x);
_Decimal64 rsqrtd64(_Decimal64 x);
_Decimal128 rsqrtd128(_Decimal128 x);
_Decimal32 sqrtd32(_Decimal32 x);
_Decimal64 sqrtd64(_Decimal64 x);
_Decimal128 sqrtd128(_Decimal128 x);
_Decimal32 erfd32(_Decimal32 x);
```

```
_Decimal64 erfd64(_Decimal64 x);
_Decimal128 erfd128(_Decimal128 x);
_Decimal32 erfcd32(_Decimal32 x);
_Decimal64 erfcd64(_Decimal64 x);
_Decimal128 erfcd128(_Decimal128 x);
_Decimal32 lgammad32(_Decimal32 x);
_Decimal64 lgammad64(_Decimal64 x);
_Decimal128 lgammad128(_Decimal128 x);
_Decimal32 tgammad32(_Decimal32 x);
_Decimal64 tgammad64(_Decimal64 x);
_Decimal128 tgammad128(_Decimal128 x);
_Decimal32 ceild32(_Decimal32 x);
_Decimal64 ceild64(_Decimal64 x);
_Decimal128 ceild128(_Decimal128 x);
_Decimal32 floord32(_Decimal32 x);
_Decimal64 floord64(_Decimal64 x);
_Decimal128 floord128(_Decimal128 x);
_Decimal32 nearbyintd32(_Decimal32 x);
_Decimal64 nearbyintd64(_Decimal64 x);
_Decimal128 nearbyintd128(_Decimal128 x);
_Decimal32 rintd32(_Decimal32 x);
_Decimal64 rintd64(_Decimal64 x);
_Decimal128 rintd128(_Decimal128 x);
long int lrintd32(_Decimal32 x);
long int lrintd64(_Decimal64 x);
long int lrintd128(_Decimal128 x);
long long int llrintd32(_Decimal32 x);
long long int llrintd64(_Decimal64 x);
long long int llrintd128(_Decimal128 x);
_Decimal32 roundd32(_Decimal32 x);
_Decimal64 roundd64(_Decimal64 x);
_Decimal128 roundd128(_Decimal128 x);
long int lroundd32(_Decimal32 x);
long int lroundd64(_Decimal64 x);
long int lroundd128(_Decimal128 x);
long long int llroundd32(_Decimal32 x);
long long int llroundd64(_Decimal64 x);
long long int llroundd128(_Decimal128 x);
_Decimal32 roundevend32(_Decimal32 x);
_Decimal64 roundevend64(_Decimal64 x);
_Decimal128 roundevend128(_Decimal128 x);
_Decimal32 truncd32(_Decimal32 x);
_Decimal64 truncd64(_Decimal64 x);
_Decimal128 truncd128(_Decimal128 x);
_Decimal32 fromfpd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 fromfpd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 fromfpd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 ufromfpd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 ufromfpd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 ufromfpd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 fromfpxd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 fromfpxd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 fromfpxd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 ufromfpxd32(_Decimal32 x, int rnd, unsigned int width);
_Decimal64 ufromfpxd64(_Decimal64 x, int rnd, unsigned int width);
_Decimal128 ufromfpxd128(_Decimal128 x, int rnd, unsigned int width);
_Decimal32 fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);
_Decimal32 remainderd32(_Decimal32 x, _Decimal32 y);
_Decimal64 remainderd64(_Decimal64 x, _Decimal64 y);
```

```
_Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);
_Decimal32 copysignd32(_Decimal32 x, _Decimal32 y);
_Decimal64 copysignd64(_Decimal64 x, _Decimal64 y);
_Decimal128 copysignd128(_Decimal128 x, _Decimal128 y);
_Decimal32 nand32(const char *tagp);
_Decimal64 nand64(const char *tagp);
_Decimal128 nand128(const char *tagp);
_Decimal32 nextafterd32(_Decimal32 x, _Decimal32 y);
_Decimal64 nextafterd64(_Decimal64 x, _Decimal64 y);
_Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);
_Decimal32 nexttowardd32(_Decimal32 x, _Decimal128 y);
_Decimal64 nexttowardd64(_Decimal64 x, _Decimal128 y);
_Decimal128 nexttowardd128(_Decimal128 x, _Decimal128 y);
_Decimal32 nextupd32(_Decimal32 x);
_Decimal64 nextupd64(_Decimal64 x);
_Decimal128 nextupd128(_Decimal128 x);
_Decimal32 nextdownd32(_Decimal32 x);
_Decimal64 nextdownd64(_Decimal64 x);
_Decimal128 nextdownd128(_Decimal128 x);
int canonicalized32(_Decimal32 * cx, const _Decimal32 * x);
int canonicalized64(_Decimal64 * cx, const _Decimal64 * x);
int canonicalized128(_Decimal128 * cx, const _Decimal128 * x);
_Decimal32 fdimd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fdimd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmaxd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaxd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmind32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmind64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmind128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmaximumd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximumd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximumd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fminimumd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimumd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimumd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmaximum_magd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_magd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_magd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fminimum_magd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_magd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_magd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmaximum_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_numd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fminimum_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_numd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmaximum_mag_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmaximum_mag_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaximum_mag_numd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fminimum_mag_numd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminimum_mag_numd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminimum_mag_numd128(_Decimal128 x, _Decimal128 y);
_Decimal32 fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
_Decimal64 fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
_Decimal32 d32addd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32addd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64addd128(_Decimal128 x, _Decimal128 y);
```

```
_Decimal32 d32subd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32subd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64subd128(_Decimal128 x, _Decimal128 y);
_Decimal32 d32muld64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32muld128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64muld128(_Decimal128 x, _Decimal128 y);
_Decimal32 d32divd64(_Decimal64 x, _Decimal64 y);
_Decimal32 d32divd128(_Decimal128 x, _Decimal128 y);
_Decimal64 d64divd128(_Decimal128 x, _Decimal128 y);
_Decimal32 d32fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal32 d32fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
_Decimal64 d64fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
_Decimal32 d32sqrtd64(_Decimal64 x);
_Decimal32 d32sqrtd128(_Decimal128 x);
_Decimal64 d64sqrtd128(_Decimal128 x);
_Decimal32 quantized32(_Decimal32 x, _Decimal32 y);
_Decimal64 quantized64(_Decimal64 x, _Decimal64 y);
_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);
bool samequantumd32(_Decimal32 x, _Decimal32 y);
bool samequantumd64(_Decimal64 x, _Decimal64 y);
bool samequantumd128(_Decimal128 x, _Decimal128 y);
_Decimal32 quantumd32(_Decimal32 x);
_Decimal64 quantumd64(_Decimal64 x);
_Decimal128 quantumd128(_Decimal128 x);
long long int llquantexpd32(_Decimal32 x);
long long int llquantexpd64(_Decimal64 x);
long long int llquantexpd128(_Decimal128 x);
void encodedecd32(unsigned char encptr[restrict static 4],
      const _Decimal32*restrict xptr);
void encodedecd64(unsigned char encptr[restrict static 8],
      const _Decimal64*restrict xptr);
void encodedecd128(unsigned char encptr[restrict static 16],
      const _Decimal128*restrict xptr);
void decodedecd32(_Decimal32 * restrict xptr,
      const unsigned char encptr[restrict static 4]);
void decodedecd64(_Decimal64 * restrict xptr,
      const unsigned char encptr[restrict static 8]);
void decodedecd128(_Decimal128 * restrict xptr,
      const unsigned char encptr[restrict static 16]);
void encodebind32(unsigned char encptr[restrict static 4],
      const _Decimal32 * restrict xptr);
void encodebind64(unsigned char encptr[restrict static 8],
      const _Decimal64 * restrict xptr);
void encodebind128(unsigned char encptr[restrict static 16],
      const _Decimal128 * restrict xptr);
void decodebind32(_Decimal32 * restrict xptr,
      const unsigned char encptr[restrict static 4]);
void decodebind64(_Decimal64 * restrict xptr,
      const unsigned char encptr[restrict static 8]);
void decodebind128(_Decimal128 * restrict xptr,
      const unsigned char encptr[restrict static 16]);
```

Only if the implementation defines `__STDC_IEC_60559_BFP__` or `__STDC_IEC_559__` and additionally the user code defines `__STDC_WANT_IEC_60559_EXT__` before any inclusion of `<math.h>`:

```
int totalorder(const double *x, const double *y);
int totalorderf(const float *x, const float *y);
int totalorderl(const long double *x, const long double *y);
int totalordermag(const double *x, const double *y);
int totalordermagf(const float *x, const float *y);
int totalordermagl(const long double *x, const long double *y);
```

```
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
int setpayload(double *res, double pl);
int setpayloadf(float *res, float pl);
int setpayloadl(long double *res, long double pl);
int setpayloadsig(double *res, double pl);
int setpayloadsigf(float *res, float pl);
int setpayloadsigl(long double *res, long double pl);
```

Only if the implementation defines `__STDC_IEC_60559_DFP__` and additionally the user code defines `__STDC_WANT_IEC_60559_EXT__` before any inclusion of `<math.h>`:

`_Decimal32_t`	`_Decimal64_t`	`HUGE_VAL_D32`	`HUGE_VAL_D64`	`HUGE_VAL_D128`

```
int totalorderd32(const _Decimal32 *x, const _Decimal32 *y);
int totalorderd64(const _Decimal64 *x, const _Decimal64 *y);
int totalorderd128(const _Decimal128 *x, const _Decimal128 *y);
int totalordermagd32(const _Decimal32 *x, const _Decimal32 *y);
int totalordermagd64(const _Decimal64 *x, const _Decimal64 *y);
int totalordermagd128(const _Decimal128 *x, const _Decimal128 *y);
_Decimal32 getpayloadd32(const _Decimal32 *x);
_Decimal64 getpayloadd64(const _Decimal64 *x);
_Decimal128 getpayloadd128(const _Decimal128 *x);
int setpayloadd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadd128(_Decimal128 *res, _Decimal128 pl);
int setpayloadsigd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadsigd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadsigd128(_Decimal128 *res, _Decimal128 pl);
```

B.12 Non-local jumps `<setjmp.h>`

`jmp_buf`

```
int setjmp(jmp_buf env);
[[noreturn]] void longjmp(jmp_buf env, int val);
```

B.13 Signal handling `<signal.h>`

`sig_atomic_t`	`SIG_IGN`	`SIGILL`	`SIGTERM`
`SIG_DFL`	`SIGABRT`	`SIGINT`	
`SIG_ERR`	`SIGFPE`	`SIGSEGV`	

```
void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```

B.14 Alignment `<stdalign.h>`

The header `<stdalign.h>` provides no content.

B.15 Variable arguments `<stdarg.h>`

`va_list`

```
type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
```

```
void va_end(va_list ap);
void va_start(va_list ap, ...);
```

B.16 Atomics <stdatomic.h>

```
__STDC_NO_ATOMICS__          memory_order_seq_cst     atomic_uint_least16_t
ATOMIC_BOOL_LOCK_FREE        atomic_bool              atomic_int_least32_t
ATOMIC_CHAR_LOCK_FREE        atomic_char              atomic_uint_least32_t
ATOMIC_CHAR16_T_LOCK_FREE    atomic_schar             atomic_int_least64_t
ATOMIC_CHAR32_T_LOCK_FREE    atomic_uchar             atomic_uint_least64_t
ATOMIC_WCHAR_T_LOCK_FREE     atomic_short             atomic_int_fast8_t
ATOMIC_SHORT_LOCK_FREE       atomic_ushort            atomic_uint_fast8_t
ATOMIC_INT_LOCK_FREE         atomic_int               atomic_int_fast16_t
ATOMIC_LONG_LOCK_FREE        atomic_uint              atomic_uint_fast16_t
ATOMIC_LLONG_LOCK_FREE       atomic_long              atomic_int_fast32_t
ATOMIC_POINTER_LOCK_FREE     atomic_ulong             atomic_uint_fast32_t
ATOMIC_FLAG_INIT             atomic_llong             atomic_int_fast64_t
memory_order                 atomic_ullong            atomic_uint_fast64_t
atomic_flag                  atomic_char16_t          atomic_intptr_t
memory_order_relaxed         atomic_char32_t          atomic_uintptr_t
memory_order_consume         atomic_wchar_t           atomic_size_t
memory_order_acquire         atomic_int_least8_t      atomic_ptrdiff_t
memory_order_release         atomic_uint_least8_t     atomic_intmax_t
memory_order_acq_rel         atomic_int_least16_t     atomic_uintmax_t
```

```
void atomic_init(volatile A *obj, C value);
type kill_dependency(type y);
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);
bool atomic_is_lock_free(const volatile A *obj);
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
C atomic_load(const volatile A *object);
C atomic_load_explicit(const volatile A *object, memory_order order);
C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected,
      C desired, memory_order success, memory_order failure);
bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected,
      C desired, memory_order success, memory_order failure);
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
bool atomic_flag_test_and_set(volatile atomic_flag *object);
bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,
      memory_order order);
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *object,
      memory_order order);
```

B.17 Bit and byte utilities <stdbit.h>

```
__STDC_ENDIAN_BIG__          __STDC_ENDIAN_LITTLE__     __STDC_ENDIAN_NATIVE__
```

```
int stdc_leading_zerosuc(unsigned char value);
```

```
int stdc_leading_zerosus(unsigned short value);
int stdc_leading_zerosui(unsigned int value);
int stdc_leading_zerosul(unsigned long value);
int stdc_leading_zerosull(unsigned long long value);
generic_return_type stdc_leading_zeros(generic_value_type value);
int stdc_leading_onesuc(unsigned char value);
int stdc_leading_onesus(unsigned short value);
int stdc_leading_onesui(unsigned int value);
int stdc_leading_onesul(unsigned long value);
int stdc_leading_onesull(unsigned long long value);
generic_return_type stdc_leading_ones(generic_value_type value);
int stdc_trailing_zerosuc(unsigned char value);
int stdc_trailing_zerosus(unsigned short value);
int stdc_trailing_zerosui(unsigned int value);
int stdc_trailing_zerosul(unsigned long value);
int stdc_trailing_zerosull(unsigned long long value);
generic_return_type stdc_trailing_zeros(generic_value_type value);
int stdc_trailing_onesuc(unsigned char value);
int stdc_trailing_onesus(unsigned short value);
int stdc_trailing_onesui(unsigned int value);
int stdc_trailing_onesul(unsigned long value);
int stdc_trailing_onesull(unsigned long long value);
generic_return_type stdc_trailing_ones(generic_value_type value);
int stdc_first_leading_zerouc(unsigned char value);
int stdc_first_leading_zerous(unsigned short value);
int stdc_first_leading_zeroui(unsigned int value);
int stdc_first_leading_zeroul(unsigned long value);
int stdc_first_leading_zeroull(unsigned long long value);
generic_return_type stdc_first_leading_zero(generic_value_type value);
int stdc_first_leading_oneuc(unsigned char value);
int stdc_first_leading_oneus(unsigned short value);
int stdc_first_leading_oneui(unsigned int value);
int stdc_first_leading_oneul(unsigned long value);
int stdc_first_leading_oneull(unsigned long long value);
generic_return_type stdc_first_leading_one(generic_value_type value);
int stdc_first_trailing_zerouc(unsigned char value);
int stdc_first_trailing_zerous(unsigned short value);
int stdc_first_trailing_zeroui(unsigned int value);
int stdc_first_trailing_zeroul(unsigned long value);
int stdc_first_trailing_zeroull(unsigned long long value);
generic_return_type stdc_first_trailing_zero(generic_value_type value);
int stdc_first_trailing_oneuc(unsigned char value);
int stdc_first_trailing_oneus(unsigned short value);
int stdc_first_trailing_oneui(unsigned int value);
int stdc_first_trailing_oneul(unsigned long value);
int stdc_first_trailing_oneull(unsigned long long value);
generic_return_type stdc_first_trailing_one(generic_value_type value);
int stdc_count_zerosuc(unsigned char value);
int stdc_count_zerosus(unsigned short value);
int stdc_count_zerosui(unsigned int value);
int stdc_count_zerosul(unsigned long value);
int stdc_count_zerosull(unsigned long long value);
generic_return_type stdc_count_zeros(generic_value_type value);
int stdc_count_onesuc(unsigned char value);
int stdc_count_onesus(unsigned short value);
int stdc_count_onesui(unsigned int value);
int stdc_count_onesul(unsigned long value);
int stdc_count_onesull(unsigned long long value);
generic_return_type stdc_count_ones(generic_value_type value);
bool stdc_has_single_bituc(unsigned char value);
bool stdc_has_single_bitus(unsigned short value);
```

```
bool stdc_has_single_bitui(unsigned int value);
bool stdc_has_single_bitul(unsigned long value);
bool stdc_has_single_bitull(unsigned long long value);
bool stdc_has_single_bit(generic_value_type value);
int stdc_bit_widthuc(unsigned char value);
int stdc_bit_widthus(unsigned short value);
int stdc_bit_widthui(unsigned int value);
int stdc_bit_widthul(unsigned long value);
int stdc_bit_widthull(unsigned long long value);
generic_return_type stdc_bit_width(generic_value_type value);
unsigned char stdc_bit_flooruc(unsigned char value);
unsigned short stdc_bit_floorus(unsigned short value);
unsigned int stdc_bit_floorui(unsigned int value);
unsigned long stdc_bit_floorul(unsigned long value);
unsigned long long stdc_bit_floorull(unsigned long long value);
generic_value_type stdc_bit_floor(generic_value_type value);
unsigned char stdc_bit_ceiluc(unsigned char value);
unsigned short stdc_bit_ceilus(unsigned short value);
unsigned int stdc_bit_ceilui(unsigned int value);
unsigned long stdc_bit_ceilul(unsigned long value);
unsigned long long stdc_bit_ceilull(unsigned long long value);
generic_value_type stdc_bit_ceil(generic_value_type value);
```

B.18 Boolean types and values <stdbool.h>

__bool_true_false_are_defined

B.19 Common definitions <stddef.h>

ptrdiff_t	size_t	wchar_t
nullptr_t	max_align_t	NULL

```
offsetof(type, member-designator)
```

Only if the implementation defines __STDC_LIB_EXT1__ and additionally the user code defines __STDC_WANT_LIB_EXT1__ before any inclusion of <stddef.h>:

rsize_t

B.20 Integer types <stdint.h>

int*N*_t	INT_LEAST*N*_MAX	INTMAX_WIDTH
uint*N*_t	INT_LEAST*N*_WIDTH	UINTMAX_MAX
int_least*N*_t	UINT_LEAST*N*_MAX	UINTMAX_WIDTH
uint_least*N*_t	UINT_LEAST*N*_WIDTH	PTRDIFF_MIN
int_fast*N*_t	INT_FAST*N*_MIN	PTRDIFF_MAX
uint_fast*N*_t	INT_FAST*N*_MAX	SIG_ATOMIC_MIN
intptr_t	INT_FAST*N*_WIDTH	SIG_ATOMIC_MAX
uintptr_t	UINT_FAST*N*_MAX	SIG_ATOMIC_WIDTH
intmax_t	UINT_FAST*N*_WIDTH	SIZE_MAX
uintmax_t	INTPTR_MIN	SIZE_WIDTH
INT*N*_MIN	INTPTR_MAX	WCHAR_MIN
INT*N*_MAX	INTPTR_WIDTH	WCHAR_MAX
INT*N*_WIDTH	UINTPTR_MAX	WCHAR_WIDTH
UINT*N*_MAX	UINTPTR_WIDTH	WINT_MIN
UINT*N*_WIDTH	INTMAX_MIN	WINT_MAX
INT_LEAST*N*_MIN	INTMAX_MAX	WINT_WIDTH

```
INTN_C( value )        INTMAX_C( value )
UINTN_C( value )       UINTMAX_C( value )
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<stdint.h>`:

`RSIZE_MAX`

B.21 Input/output <stdio.h>

```
size_t      _IONBF        SEEK_CUR    stdout
FILE        BUFSIZ        SEEK_END    _PRINTF_NAN_LEN_MAX
fpos_t      EOF           SEEK_SET
NULL        FOPEN_MAX     TMP_MAX
_IOFBF      FILENAME_MAX  stderr
_IOLBF      L_tmpnam      stdin
```

```
int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char * restrict filename, const char * restrict mode);
FILE *freopen(const char * restrict filename, const char * restrict mode,
      FILE * restrict stream);
void setbuf(FILE * restrict stream, char * restrict buf);
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
int printf(const char * restrict format, ...);
int scanf(const char * restrict format, ...);
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
int sprintf(char * restrict s, const char * restrict format, ...);
int sscanf(const char * restrict s, const char * restrict format, ...);
int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);
int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);
int vprintf(const char * restrict format, va_list arg);
int vscanf(const char * restrict format, va_list arg);
int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);
int vsprintf(char * restrict s, const char * restrict format, va_list arg);
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char * restrict s, int n, FILE * restrict stream);
int fputc(int c, FILE *stream);
int fputs(const char * restrict s, FILE * restrict stream);
int getc(FILE *stream);
int getchar(void);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
      FILE * restrict stream);
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
      FILE * restrict stream);
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
```

```
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
int fprintf(FILE * restrict stream, const char * restrict format, ...);
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<stdio.h>`:

`L_tmpnam_s`	`TMP_MAX_S`	`errno_t`	`rsize_t`

```
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
errno_t tmpnam_s(char *s, rsize_t maxsize);
errno_t fopen_s(FILE * restrict * restrict streamptr,
      const char * restrict filename, const char * restrict mode);
errno_t freopen_s(FILE * restrict * restrict newstreamptr,
      const char * restrict filename, const char * restrict mode,
      FILE * restrict stream);
int fprintf_s(FILE * restrict stream, const char * restrict format, ...);
int fscanf_s(FILE * restrict stream, const char * restrict format, ...);
int printf_s(const char * restrict format, ...);
int scanf_s(const char * restrict format, ...);
int snprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
int sprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
int sscanf_s(const char * restrict s, const char * restrict format, ...);
int vfprintf_s(FILE *restrict stream, const char *restrict format, va_list arg);
int vfscanf_s(FILE *restrict stream, const char *restrict format, va_list arg);
int vprintf_s(const char * restrict format, va_list arg);
int vscanf_s(const char * restrict format, va_list arg);
int vsnprintf_s(char *restrict s, rsize_t n, const char *restrict format,
      va_list arg);
int vsprintf_s(char * restrict s, rsize_t n, const char * restrict format,
      va_list arg);
int vsscanf_s(const char *restrict s, const char *restrict format, va_list arg);
char *gets_s(char *s, rsize_t n);
```

B.22 General utilities `<stdlib.h>`

`size_t`	`div_t`	`lldiv_t`	`EXIT_FAILURE`	`RAND_MAX`
`wchar_t`	`ldiv_t`	`NULL`	`EXIT_SUCCESS`	`MB_CUR_MAX`

```
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
int strfromd(char *restrict s, size_t n, const char *restrict format, double fp);
int strfromf(char *restrict s, size_t n, const char *restrict format, float fp);
int strfroml(char *restrict s, size_t n, const char *restrict format, long double fp);

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
long int strtol(const char *restrict nptr, char **restrict endptr, int base);
long long int strtoll(const char *restrict nptr, char **restrict endptr, int base);
unsigned long int strtoul(const char *restrict nptr, char **restrict endptr, int base);
unsigned long long int strtoull(const char *restrict nptr, char **restrict endptr, int
    base);
```

```
int rand(void);
void srand(unsigned int seed);
void *aligned_alloc(size_t alignment, size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void free_sized(void *ptr, size_t size);
void free_aligned_sized(void *ptr, size_t alignment, size_t size);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
[[noreturn]] void abort(void);
int atexit(void (*func)(void));
int at_quick_exit(void (*func)(void));
[[noreturn]] void exit(int status);
[[noreturn]] void _Exit(int status);
char *getenv(const char *name);
[[noreturn]] void quick_exit(int status);
int system(const char *string);
QVoid *bsearch(const void *key, QVoid *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *));
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
int wctomb(char *s, wchar_t wc);
size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
size_t memalignment(const void * p);
```

Only if the implementation defines `__STDC_IEC_60559_DFP__`:

```
int strfromd32(char*restrict s, size_t n, const char*restrict format, _Decimal32 fp);
int strfromd64(char*restrict s, size_t n, const char*restrict format, _Decimal64 fp);
int strfromd128(char*restrict s, size_t n, const char*restrict format, _Decimal128 fp);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<stdlib.h>`:

```
errno_t                 rsize_t                 constraint_handler_t
```

```
constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
void abort_handler_s(const char * restrict msg, void * restrict ptr,
      errno_t error);
void ignore_handler_s(const char * restrict msg, void * restrict ptr,
      errno_t error);
errno_t getenv_s(size_t * restrict len, char * restrict value, rsize_t maxsize,
      const char * restrict name);
QVoid *bsearch_s(const void *key, QVoid *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *k, const void *y, void *context),
      void *context);
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *x, const void *y, void *context),
      void *context);
errno_t wctomb_s(int *restrict status, char *restrict s, rsize_t smax,
      wchar_t wc);
```

```
errno_t mbstowcs_s(size_t *restrict retval, wchar_t *restrict dst,
      rsize_t dstmax, const char * restrict src, rsize_t len);
errno_t wcstombs_s(size_t * restrict retval, char * restrict dst, rsize_t dstmax,
      const wchar_t * restrict src, rsize_t len);
```

B.23 `_Noreturn <stdnoreturn.h>`

`noreturn`

B.24 `ckd_` Checked Integer Operations `<stdckdint.h>`

```
bool ckd_add(type1 *result, type2 a, type3 b);
bool ckd_sub(type1 *result, type2 a, type3 b);
bool ckd_mul(type1 *result, type2 a, type3 b);
```

B.25 String handling `<string.h>`

`size_t` `NULL`

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memccpy(void * restrict s1, const void * restrict s2, int c, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char * restrict s1, const char * restrict s2);
char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
char *strdup(const char *s);
char *strndup(const char *s, size_t size);
char *strcat(char * restrict s1, const char * restrict s2);
char *strncat(char * restrict s1, const char * restrict s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
QVoid *memchr(QVoid *s, int c, size_t n);
QChar *strchr(QChar *s, int c);
size_t strcspn(const char *s1, const char *s2);
QChar *strpbrk(QChar *s1, const char *s2);
QChar *strrchr(QChar *s, int c);
size_t strspn(const char *s1, const char *s2);
QChar *strstr(QChar *s1, const char *s2);
char *strtok(char * restrict s1, const char * restrict s2);
void *memset(void *s, int c, size_t n);
void *memset_explicit(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<string.h>`:

`errno_t` `rsize_t`

```
errno_t memcpy_s(void * restrict s1, rsize_t s1max, const void * restrict s2,
      rsize_t n);
errno_t memmove_s(void *s1, rsize_t s1max, const void *s2, rsize_t n);
errno_t strcpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
errno_t strncpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2,
      rsize_t n);
errno_t strcat_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
```

```
errno_t strncat_s(char * restrict s1, rsize_t s1max, const char * restrict s2,
      rsize_t n);
char *strtok_s(char * restrict s1, rsize_t * restrict s1max,
      const char * restrict s2, char ** restrict ptr);
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
errno_t strerror_s(char *s, rsize_t maxsize, errno_t errnum);
size_t strerrorlen_s(errno_t errnum);
size_t strnlen_s(const char *s, size_t maxsize);
```

B.26 Type-generic math <tgmath.h>

acos	atanpi	fmin	logb	tanpi
asin	cbrt	fminimum	logp1	tgamma
atan	ceil	fminimum_mag	lrint	trunc
acosh	compoundn	fminimum_num	lround	ufromfpx
asinh	copysign	fminimum_mag_num	nearbyint	ufromfp
atanh	cospi	fmod	nextafter	fadd
cos	erfc	frexp	nextdown	dadd
sin	erf	fromfpx	nexttoward	fsub
tan	exp10m1	fromfp	nextup	dsub
cosh	exp10	hypot	pown	fmul
sinh	exp2m1	ilogb	powr	dmul
tanh	exp2	ldexp	remainder	fdiv
exp	expm1	lgamma	remquo	ddiv
log	fdim	llogb	rint	ffma
pow	floor	llrint	rootn	dfma
sqrt	fmax	llround	roundeven	fsqrt
fabs	fmaximum	log10p1	round	dsqrt
acospi	fmaximum_mag	log10	rsqrt	
asinpi	fmaximum_num	log1p	scalbln	
atan2pi	fmaximum_mag_num	log2p1	scalbn	
atan2	fma	log2	sinpi	

Only if the implementation does not define `__STDC_NO_COMPLEX__`:

carg	cimag	conj	cproj	creal

Only if the implementation defines `__STDC_IEC_60559_DFP__`:

d32add	d64sub	d32div	d64fma	quantize	llquantexp
d64add	d32mul	d64div	d32sqrt	samequantum	
d32sub	d64mul	d32fma	d64sqrt	quantum	

B.27 Threads <threads.h>

__STDC_NO_THREADS__	mtx_t	thrd_timedout
thread_local	tss_dtor_t	thrd_success
ONCE_FLAG_INIT	thrd_start_t	thrd_busy
TSS_DTOR_ITERATIONS	once_flag	thrd_error
cnd_t	mtx_plain	thrd_nomem
thrd_t	mtx_recursive	
tss_t	mtx_timed	

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
```

```
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
      const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
[[noreturn]] void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
void thrd_yield(void);
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

B.28 Date and time <time.h>

```
NULL                    size_t                  struct timespec
CLOCKS_PER_SEC          clock_t                 struct tm
TIME_UTC                time_t
```

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t timegm(struct tm *timeptr);
time_t time(time_t *timer);
int timespec_get(struct timespec *ts, int base);
int timespec_getres(struct timespec *ts, int base);
[[deprecated]] char *asctime(const struct tm *timeptr);
[[deprecated]] char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *timer, struct tm *buf);
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *timer, struct tm *buf);
size_t strftime(char * restrict s, size_t maxsize, const char * restrict format,
      const struct tm * restrict timeptr);
```

Only if supported by the implementation:

```
TIME_MONOTONIC
TIME_ACTIVE
```

Only if threads are supported and it is supported by the implementation:

```
TIME_THREAD_ACTIVE
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<time.h>`:

`errno_t` `rsize_t`

```
errno_t asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);
errno_t ctime_s(char *s, rsize_t maxsize, const time_t *timer);
struct tm *gmtime_s(const time_t * restrict timer, struct tm * restrict result);
struct tm *localtime_s(const time_t *restrict timer, struct tm *restrict result);
```

B.29 Unicode utilities <uchar.h>

`mbstate_t` `size_t` `char16_t` `char32_t`

```
size_t mbrtoc8(char8_t * restrict pc8, const char * restrict s, size_t n,
      mbstate_t * restrict ps);
size_t c8rtomb(char * restrict s, char8_t c8, mbstate_t * restrict ps);
size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n,
      mbstate_t * restrict ps);
size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n,
      mbstate_t * restrict ps);
size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
```

B.30 Extended multibyte/wide character utilities <wchar.h>

`wchar_t`	`wint_t`	`WCHAR_MAX`
`size_t`	`struct tm`	`WCHAR_MIN`
`mbstate_t`	`NULL`	`WEOF`

```
int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
      ...);
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
      va_list arg);
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,
      va_list arg);
int vwprintf(const wchar_t * restrict format, va_list arg);
int vwscanf(const wchar_t * restrict format, va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s, FILE * restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
      int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr,
```

```
      int base);
unsigned long int wcstoul(const wchar_t * restrict nptr,
      wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr,
      wchar_t ** restrict endptr, int base);
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
QWchar_t *wcschr(QWchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
QWchar_t *wcspbrk(QWchar_t *s1, const wchar_t *s2);
QWchar_t *wcsrchr(QWchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
QWchar_t *wcsstr(QWchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2,
      wchar_t ** restrict ptr);
QWchar_t *wmemchr(QWchar_t *s, wchar_t c, size_t n);
size_t wcslen(const wchar_t *s);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
      const wchar_t * restrict format, const struct tm * restrict timeptr);
wint_t btowc(int c);
int wctob(wint_t c);
int mbsinit(const mbstate_t *ps);
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n,
      mbstate_t * restrict ps);
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len,
      mbstate_t * restrict ps);
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len,
      mbstate_t * restrict ps);
```

Only if the implementation defines **__STDC_LIB_EXT1__** and additionally the user code defines **__STDC_WANT_LIB_EXT1__** before any inclusion of `<wchar.h>`:

`errno_t` `rsize_t`

```
int fwprintf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int snwprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format,
      ...);
int swprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format,
      ...);
int swscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, ...);
int vfwprintf_s(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
int vfwscanf_s(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
int vsnwprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format,
      va_list arg);
int vswprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format,
```

```
      va_list arg);
int vswscanf_s(const wchar_t * restrict s, const wchar_t * restrict format,
      va_list arg);
int vwprintf_s(const wchar_t * restrict format, va_list arg);
int vwscanf_s(const wchar_t * restrict format, va_list arg);
int wprintf_s(const wchar_t * restrict format, ...);
int wscanf_s(const wchar_t * restrict format, ...);
errno_t wcscpy_s(wchar_t *restrict s1, rsize_t s1max,
      const wchar_t *restrict s2);
errno_t wcsncpy_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2, rsize_t n);
errno_t wmemcpy_s(wchar_t *restrict s1, rsize_t s1max,
      const wchar_t *restrict s2, rsize_t n);
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max, const wchar_t *s2, rsize_t n);
errno_t wcscat_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2);
errno_t wcsncat_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2, rsize_t n);
wchar_t *wcstok_s(wchar_t * restrict s1, rsize_t * restrict s1max,
      const wchar_t * restrict s2, wchar_t ** restrict ptr);
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
errno_t wcrtomb_s(size_t * restrict retval, char * restrict s, rsize_t smax,
      wchar_t wc, mbstate_t * restrict ps);
errno_t mbsrtowcs_s(size_t * restrict retval, wchar_t * restrict dst,
      rsize_t dstmax, const char ** restrict src, rsize_t len,
      mbstate_t * restrict ps);
errno_t wcsrtombs_s(size_t * restrict retval, char * restrict dst,
      rsize_t dstmax, const wchar_t ** restrict src, rsize_t len,
      mbstate_t * restrict ps);
```

B.31 Wide character classification and mapping utilities <wctype.h>

`wint_t` `wctrans_t` `wctype_t` `WEOF`

```
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char *property);
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
```

Annex C
(informative)
Sequence points

1 The following are the sequence points described in 5.1.2.3:

— Between the evaluations of the function designator and actual arguments in a function call and the actual call. (6.5.2.2).

— Between the evaluations of the first and second operands of the following operators: logical AND && (6.5.13); logical OR `||` (6.5.14); comma `,` (6.5.17).

— Between the evaluations of the first operand of the conditional `?:` operator and whichever of the second and third operands is evaluated (6.5.15).

— Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions: a full declarator for a variably modified type; an initializer that is not part of a compound literal (6.7.10); the expression in an expression statement (6.8.3); the controlling expression of a selection statement (**`if`** or **`switch`**) (6.8.4); the controlling expression of a **`while`** or **`do`** statement (6.8.5); each of the (optional) expressions of a **`for`** statement (6.8.5.3); the (optional) expression in a **`return`** statement (6.8.6.4).

— Immediately before a library function returns (7.1.4).

— After the actions associated with each formatted input/output function conversion specifier (7.23.6, 7.31.2).

— Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call (7.24.5).

Annex D
(informative)
Universal character names for identifiers

1 This subclause describes the choices made in application of UAX #31 ("Unicode Identifier and Pattern Syntax") to C of the requirements from UAX #31 and how they do or do not apply to C. For UAX #31, C conforms by meeting the requirements "Default Identifiers" (D.1) and "Equivalent Normalized Identifiers" (D.1). The other requirements, also listed below, are either alternatives not taken or do not apply to C.

D.1 Default Identifiers

1 UAX #31 specifies a default syntax for identifiers based on properties from the Unicode Character Database, UAX #44. The general syntax is

```
<Identifier> := <Start> <Continue>* (<Medial> <Continue>+)*
```

where <Start> has the XID_Start property, <Continue> has the XID_Continue property, and <Medial> is a list of characters permitted between continue characters. For C we add the character U+005F, LOW LINE, or _, to the set of permitted Start characters, the Medial set is empty, and the Continue characters are unmodified. In the grammar used in UAX #31, this is

```
<Identifier> := <Start> <Continue>*
<Start> := XID_Start + U+005F
<Continue> := <Start> + XID_Continue
```

This is described in the C grammar (6.4.2.1), where *identifier* is formed from *identifier-start* or *identifier* followed by *identifier-continue*.

D.1.1 Restricted Format Characters

1 If an implementation of UAX #31 wishes to allow format characters such as ZERO WIDTH JOINER or ZERO WIDTH NON-JOINER it must define a profile allowing them, or describe precisely which combinations are permitted.

2 C does not allow format characters in identifiers, so this does not apply.

D.1.2 Stable Identifiers

1 An implementation of UAX #31 may choose to guarantee that identifiers are stable across versions of the Unicode Standard. Once a string qualifies as an identifier it does so in all future versions. C does not make this guarantee, except to the extent that UAX #31 guarantees the stability of the XID_Start and XID_Continue properties.

D.2 Immutable Identifiers

1 An implementation may choose to guarantee that the set of identifiers will never change by fixing the set of code points allowed in identifiers forever.

2 C does not choose to make this guarantee. As scripts are added to Unicode, additional characters in those scripts may become available for use in identifiers.

D.3 Pattern_White_Space and Pattern_Syntax Characters

1 UAX #31 describes how languages that use or interpret patterns of characters, such as regular expressions or number formats, may describe that syntax with Unicode properties.

2 C does not do this as part of the language, deferring to library components for such usage of patterns. This requirement does not apply to C.

D.4 Equivalent Normalized Identifiers

1 UAX #31 requires that implementations describe how identifiers are compared and considered equivalent.

2 C requires that identifiers be in Normalization Form C and therefore identifiers that compare the same under NFC are equivalent. This is described in subclause 6.4.2.

D.5 Equivalent Case-Insensitive Identifiers

1 C considers case to be significant in identifier comparison, and does not do any case folding. This requirement does not apply to C

D.6 Filtered Normalized Identifiers

1 If any characters are excluded from normalization, UAX #31 requires a precise specification of those exclusions.

2 C does not make any such exclusions.

D.7 Filtered Case-Insensitive Identifiers

1 C identifiers are case sensitive, and therefore this requirement does not apply.

D.8 Hashtag Identifiers

1 There are no hashtags in C, so this requirement does not apply.

Annex E
(informative)
Implementation limits

1 The contents of the header `<limits.h>` are given below. The values shall all be constant expressions suitable for use in conditional expression inclusion preprocessing directives. The components are described further in 5.2.4.2.1.

2 For the following macros, the minimum values shown shall be replaced by implementation-defined values.

```
#define BOOL_WIDTH                  1
#define CHAR_BIT                    8
#define USHRT_WIDTH                16
#define UINT_WIDTH                 16
#define ULONG_WIDTH                32
#define ULLONG_WIDTH               64
#define BITINT_MAXWIDTH            ULLONG_WIDTH // at minimum as large
                                                // as unsigned long long
#define MB_LEN_MAX                  1
```

3 For the following macros, the minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign that are deduced from the macros above as indicated.[434)]

```
#define BOOL_MAX                       1   // 2^BOOL_WIDTH − 1
#define CHAR_MAX      UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN              0 or SCHAR_MIN
#define CHAR_WIDTH                     8   // CHAR_BIT
#define INT_MAX                   +32767   // 2^(INT_WIDTH−1) − 1
#define INT_MIN                   -32768   // −2^(INT_WIDTH−1)
#define INT_WIDTH                     16   // UINT_WIDTH
#define LONG_MAX             +2147483647   // 2^(LONG_WIDTH−1) − 1
#define LONG_MIN             -2147483648   // −2^(LONG_WIDTH−1)
#define LONG_WIDTH                    32   // ULONG_WIDTH
#define LLONG_MAX   +9223372036854775807   // 2^(LLONG_WIDTH−1) − 1
#define LLONG_MIN   -9223372036854775808   // −2^(LLONG_WIDTH−1)
#define LLONG_WIDTH                   64   // ULLONG_WIDTH
#define SCHAR_MAX                   +127   // 2^(SCHAR_WIDTH−1) − 1
#define SCHAR_MIN                   -128   // −2^(SCHAR_WIDTH−1)
#define SCHAR_WIDTH                    8   // CHAR_BIT
#define SHRT_MAX                  +32767   // 2^(SHRT_WIDTH−1) − 1
#define SHRT_MIN                  -32768   // −2^(SHRT_WIDTH−1)
#define UCHAR_MAX                    255   // 2^UCHAR_WIDTH − 1
#define UCHAR_WIDTH                    8   // CHAR_BIT
#define USHRT_MAX                  65535   // 2^USHRT_WIDTH − 1
#define UINT_MAX                   65535   // 2^UINT_WIDTH − 1
#define ULONG_MAX             4294967295   // 2^ULONG_WIDTH − 1
#define ULLONG_MAX  18446744073709551615   // 2^ULLONG_WIDTH − 1
```

4 The contents of the header `<float.h>` are given below. All integer values, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. The components are described further in 5.2.4.2.2 and 5.2.4.2.3.

5 The values given in the following list shall be replaced by implementation-defined expressions:

```
#define FLT_EVAL_METHOD
#define FLT_ROUNDS
```

434)For the minimum value of a signed integer type there is no expression consisting of a minus sign and a decimal literal of that same type. The numbers in the table are only given as indications for the values and do not represent suitable expressions to be used for these macros.

```
#ifdef __STDC_IEC_60559_DFP__
#define DEC_EVAL_METHOD
#endif
```

6 The values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

```
#define DBL_DECIMAL_DIG                10
#define DBL_DIG                        10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP                +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP                -37
#define DBL_MIN_EXP
#define DECIMAL_DIG                    10
#define FLT_DECIMAL_DIG                 6
#define FLT_DIG                         6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP                +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP                -37
#define FLT_MIN_EXP
#define FLT_RADIX                       2
#define LDBL_DECIMAL_DIG               10
#define LDBL_DIG                       10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP               +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP               -37
#define LDBL_MIN_EXP
```

7 The values given in the following list shall be replaced by implementation-defined constant expressions with values that are greater than or equal to those shown:

```
#define DBL_MAX                     1E+37
#define DBL_NORM_MAX                1E+37
#define FLT_MAX                     1E+37
#define FLT_NORM_MAX                1E+37
#define LDBL_MAX                    1E+37
#define LDBL_NORM_MAX               1E+37
```

8 The values given in the following list shall be replaced by implementation-defined constant expressions with (positive) values that are less than or equal to those shown:

```
#define DBL_EPSILON                  1E-9
#define DBL_MIN                     1E-37
#define FLT_EPSILON                  1E-5
#define FLT_MIN                     1E-37
#define LDBL_EPSILON                 1E-9
#define LDBL_MIN                    1E-37
```

9 If the implementation supports decimal floating types, the following macros provide the parameters of these types as exact values.

```
#ifdef __STDC_IEC_60559_DFP__
#define DEC32_EPSILON        1E-6DF
#define DEC32_MANT_DIG       7
#define DEC32_MAX            9.999999E96DF
#define DEC32_MAX_EXP        97
```

```
#define DEC32_MIN            1E-95DF
#define DEC32_MIN_EXP        -94
#define DEC32_TRUE_MIN       0.000001E-95DF
#define DEC64_EPSILON        1E-15DD
#define DEC64_MANT_DIG       16
#define DEC64_MAX            9.999999999999999E384DD
#define DEC64_MAX_EXP        385
#define DEC64_MIN            1E-383DD
#define DEC64_MIN_EXP        -382
#define DEC64_TRUE_MIN       0.000000000000001E-383DD
#define DEC128_EPSILON       1E-33DL
#define DEC128_MANT_DIG      34
#define DEC128_MAX           9.999999999999999999999999999999999E6144DL
#define DEC128_MAX_EXP       6145
#define DEC128_MIN           1E-6143DL
#define DEC128_MIN_EXP       -6142
#define DEC128_TRUE_MIN      0.000000000000000000000000000000001E-6143DL
#endif
```

Annex F
(normative)
IEC 60559 floating-point arithmetic

F.1 Introduction

1 This annex specifies C language support for the *IEC 60559* floating-point standard. The *IEC 60559 floating-point standard* is specifically Floating-point arithmetic (ISO/IEC 60559:2020), also designated as *IEEE Standard for Floating-Point Arithmetic* (IEEE 754–2019). IEC 60559 generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc.

2 The IEC 60559 floating-point standard is a minor upgrade to IEC 60559:2011 (IEEE 754-2008). IEC 60559:2011 was a major upgrade to IEC 60559:1989 (IEEE 754–1985), specifying decimal as well as binary floating-point arithmetic.

3 An implementation that defines `__STDC_IEC_60559_BFP__` to `202311L` shall conform to the specifications in this annex for binary floating-point arithmetic and shall also define `__STDC_IEC_559__` to `1`.[435]

4 An implementation that defines `__STDC_IEC_60559_DFP__` to `202311L` shall conform to the specifications for decimal floating-point arithmetic in the following subclauses of this annex:

— F.2.1 Infinities and NaNs
— F.3 Operations
— F.4 Floating to integer conversions
— F.6 The **`return`** statement
— F.7 Contracted expressions
— F.8 Floating-point environment
— F.9 Optimization
— F.10 Mathematics `<math.h>` and `<tgmath.h>`

For the purpose of specifying these conformance requirements, the macros, functions, and values mentioned in the subclauses listed above are understood to refer to the corresponding macros, functions, and values for decimal floating types. Likewise, the "rounding direction mode" is understood to refer to the rounding direction mode for decimal floating-point arithmetic.

5 Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

F.2 Types

1 The C floating types match the IEC 60559 formats as follows:

— The **`float`** type matches the IEC 60559 binary32 format.

— The **`double`** type matches the IEC 60559 binary64 format.

— The **`long double`** type matches the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format, [436] else a non-IEC 60559 extended format, else the IEC 60559 binary64 format.

Any non-IEC 60559 extended format used for the **`long double`** type shall have more precision than IEC 60559 binary64 and at least the range of IEC 60559 binary64.[437] The value of `FLT_ROUNDS` applies to all IEC 60559 types supported by the implementation, but need not apply to non-IEC 60559 types.

[435] Implementations that do not define either of `__STDC_IEC_60559_BFP__` and `__STDC_IEC_559__` are not required to conform to these specifications. New code should not use the obsolescent macro `__STDC_IEC_559__` to test for conformance to this annex.

[436] IEC 60559 binary64-extended formats include the common 80-bit IEC 60559 format.

[437] A non-IEC 60559 **`long double`** type is required to provide infinity and NaNs, as its values include all **`double`** values.

Recommended practice

2 The **long double** type should match the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format.

F.2.1 Infinities and NaNs

1 Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values (5.2.4.2.2).

2 The **NAN** and **INFINITY** macros in `<float.h>` and the **nan** functions in `<math.h>` provide designations for IEC 60559 quiet NaNs and infinities. The **FLT_SNAN**, **DBL_SNAN**, and **LDBL_SNAN** macros in `<float.h>` provide designations for IEC 60559 signaling NaNs.

3 This annex does not require the full support for signaling NaNs specified in IEC 60559. This annex uses the term NaN, unless explicitly qualified, to denote quiet NaNs. Where specification of signaling NaNs is not provided, the behavior of signaling NaNs is implementation-defined (either treated as an IEC 60559 quiet NaN or treated as an IEC 60559 signaling NaN). [438)]

4 Any operator or `<math.h>` function that raises an "invalid" floating-point exception, if delivering a floating type result, shall return a quiet NaN, unless explicitly specified otherwise.

5 To support signaling NaNs as specified in IEC 60559, an implementation should adhere to the following recommended practice.

Recommended practice

6 Any floating-point operator or `<math.h>` function or macro with a signaling NaN input, unless explicitly specified otherwise, raises an "invalid" floating-point exception.

7 **NOTE 1** Some functions do not propagate quiet NaN arguments. For example, **hypot**(x, y) returns infinity if x or y is infinite and the other is a quiet NaN. The recommended practice in this subclause specifies that such functions (and others) raise the "invalid" floating-point exception if an argument is a signaling NaN, which also implies they return a quiet NaN in these cases.

8 The `<fenv.h>` header defines the macro **FE_SNANS_ALWAYS_SIGNAL** if and only if the implementation follows the recommended practice in this subclause. If defined, **FE_SNANS_ALWAYS_SIGNAL** expands to the integer constant `1`.

F.3 Operations

1 C operators, functions, and function-like macros provide operations specified by IEC 60559 as shown in the following table. In the table, C functions are represented by the function name without a type suffix. Specifications for the C facilities are provided in the listed clauses. The C specifications are intended to match IEC 60559, unless stated otherwise.

Operation binding

IEC 60559 operation	C operation	Clause
roundToIntegralTiesToEven	**roundeven**	7.12.9.8, F.10.6.8
roundToIntegralTiesAway	**round**	7.12.9.6, F.10.6.6
roundToIntegralTowardZero	**trunc**	7.12.9.9, F.10.6.9
roundToIntegralTowardPositive	**ceil**	7.12.9.1, F.10.6.1
roundToIntegralTowardNegative	**floor**	7.12.9.2, F.10.6.2
roundToIntegralExact	**rint**	7.12.9.4, F.10.6.4
nextUp	**nextup**	7.12.11.5, F.10.8.5
nextDown	**nextdown**	7.12.11.6, F.10.8.6
getPayload	**getpayload**	F.10.13.1
setPayload	**setpayload**	F.10.13.2
setPayloadSignaling	**setpayloadsig**	F.10.13.3
quantize	**quantize**	7.12.15.1
sameQuantum	**samequantum**	7.12.15.2

438) Since NaNs created by IEC 60559 arithmetic operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

quantum	`quantum`	7.12.15.3
encodeDecimal	`encodedec`	7.12.16.1
decodeDecimal	`decodedec`	7.12.16.2
encodeBinary	`encodebin`	7.12.16.3
decodeBinary	`decodebin`	7.12.16.4
remainder	`remainder`, `remquo`	7.12.10.2, F.10.7.2, 7.12.10.3, F.10.7.3
maximum	`fmaximum`	7.12.12.4, F.10.9.4
minimum	`fminimum`	7.12.12.5, F.10.9.4
maximumMagnitude	`fmaximum_mag`	7.12.12.6, F.10.9.4
minimumMagnitude	`fminimum_mag`	7.12.12.7, F.10.9.4
maximumNumber	`fmaximum_num`	7.12.12.8, F.10.9.5
minimumNumber	`fminimum_num`	7.12.12.9, F.10.9.5
maximumMagnitudeNumber	`fmaximum_mag_num`	7.12.12.10, F.10.9.5
minimumMagnitudeNumber	`fminimum_mag_num`	7.12.12.11, F.10.9.5
scaleB	`scalbn`, `scalbln`	7.12.6.19, F.10.3.19
logB	`logb`, `ilogb`, `llogb`	7.12.6.17, F.10.3.17, 7.12.6.8, F.10.3.8, 7.12.6.10, F.10.3.10
addition	+, `fadd`, `faddl`, `daddl`	6.5.6, 7.12.14.1, F.10.11
subtraction	−, `fsub`, `fsubl`, `dsubl`	6.5.6, 7.12.14.2, F.10.11
multiplication	∗, `fmul`, `fmull`, `dmull`	6.5.5, 7.12.14.3, F.10.11
division	/, `fdiv`, `fdivl`, `ddivl`	6.5.5, 7.12.14.4, F.10.11
squareRoot	`sqrt`, `fsqrt`, `fsqrtl`, `dsqrtl`	7.12.7.10, F.10.4.10, 7.12.14.6, F.10.11
fusedMultiplyAdd	`fma`, `ffma`, `ffmal`, `dfmal`	7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4
convertToIntegerTiesToEven convertToIntegerTowardZero convertToIntegerTowardPositive convertToIntegerTowardNegative	`fromfp`, `ufromfp`	7.12.9.10, F.10.6.10
convertToIntegerTiesToAway	`fromfp`, `ufromfp`, `lround`, `llround`	7.12.9.10, F.10.6.10, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven convertToIntegerExactTowardZero convertToIntegerExactTowardPositive convertToIntegerExactTowardNegative convertToIntegerExactTiesToAway	`fromfpx`, `ufromfpx`	7.12.9.11, F.10.6.11
convertFormat - different formats	cast and implicit conversions	6.3.1.5, 6.5.4
convertFormat - same format	`canonicalize`	7.12.11.7, F.10.8.7
convertFromDecimalCharacter	`strtod`, `wcstod`, `scanf`, `wscanf`, decimal floating constants	7.24.1.5, 7.31.4.1.2, 7.23.6.4, 7.31.2.12, F.5
convertToDecimalCharacter	`printf`, `wprintf`, `strfromd`	7.23.6.3, 7.31.2.11, 7.24.1.3, F.5
convertFromHexCharacter	`strtod`, `wcstod`, `scanf`, `wscanf`, hexadecimal floating constants	7.24.1.5, 7.31.4.1.2, 7.23.6.4, 7.31.2.12, F.5

convertToHexCharacter	`printf`, `wprintf`, `strfromd`	7.23.6.3, 7.31.2.11, 7.24.1.3, F.5
copy	`memcpy`, `memmove`, `+(x)`	7.26.2.1, 7.26.2.3
negate	`-(x)`	6.5.3.3
abs	`fabs`	7.12.7.3, F.10.4.3
copySign	`copysign`	7.12.11.1, F.10.8.1
compareQuietEqual	`==`	6.5.9, F.9.3
compareQuietNotEqual	`!=`	6.5.9, F.9.3
compareSignalingEqual	`iseqsig`	7.12.17.7, F.10.14.1
compareSignalingGreater	`>`	6.5.8, F.9.3
compareSignalingGreaterEqual	`>=`	6.5.8, F.9.3
compareSignalingLess	`<`	6.5.8, F.9.3
compareSignalingLessEqual	`<=`	6.5.8, F.9.3
compareSignalingNotEqual	`! iseqsig(x)`	7.12.17.7, F.10.14.1
compareSignalingNotGreater	`! (x > y)`	6.5.8, F.9.3
compareSignalingLessUnordered	`! (x >= y)`	6.5.8, F.9.3
compareSignalingNotLess	`! (x < y)`	6.5.8, F.9.3
compareSignalingGreaterUnordered	`! (x <= y)`	6.5.8, F.9.3
compareQuietGreater	`isgreater`	7.12.17.1
compareQuietGreaterEqual	`isgreaterequal`	7.12.17.2
compareQuietLess	`isless`	7.12.17.3
compareQuietLessEqual	`islessequal`	7.12.17.4
compareQuietUnordered	`isunordered`	7.12.17.6
compareQuietNotGreater	`! isgreater(x, y)`	7.12.17.1
compareQuietLessUnordered	`! isgreaterequal(x, y)`	7.12.17.2
compareQuietNotLess	`! isless(x, y)`	7.12.17.3
compareQuietGreaterUnordered	`! islessequal(x, y)`	7.12.17.4
compareQuietOrdered	`! isunordered(x, y)`	7.12.17.6
class	`fpclassify`, `signbit`, `issignaling`	7.12.3.1, 7.12.3.7, 7.12.3.8
isSignMinus	`signbit`	7.12.3.7
isNormal	`isnormal`	7.12.3.6
isFinite	`isfinite`	7.12.3.3
isZero	`iszero`	7.12.3.10
isSubnormal	`issubnormal`	7.12.3.9
isInfinite	`isinf`	7.12.3.4
isNaN	`isnan`	7.12.3.5
isSignaling	`issignaling`	7.12.3.8
isCanonical	`iscanonical`	7.12.3.2
radix	`FLT_RADIX`	5.2.4.2.2
totalOrder	`totalorder`	F.10.12.1
totalOrderMag	`totalordermag`	F.10.12.2
lowerFlags	`feclearexcept`	7.6.4.1
raiseFlags	`fesetexcept`	7.6.4.4
testFlags	`fetestexcept`	7.6.4.7
testSavedFlags	`fetestexceptflag`	7.6.4.6
restoreFlags	`fesetexceptflag`	7.6.4.5
saveAllFlags	`fegetexceptflag`	7.6.4.2
getBinaryRoundingDirection	`fegetround`	7.6.5.2
setBinaryRoundingDirection	`fesetround`	7.6.5.5
saveModes	`fegetmode`	7.6.5.1
restoreModes	`fesetmode`	7.6.5.4
defaultModes	`fesetmode(FE_DFL_MODE)`	7.6.5.4, 7.6

2 The IEC 60559 requirement that certain of its operations be provided for operands of different formats (of the same radix) is satisfied by C's usual arithmetic conversions (6.3.1.8) and function-call argument conversions (6.5.2.2). For example, the following operations take **float** f and **double** d inputs and produce a **long double** result:

```
(long double)f * d
powl(f, d)
```

3 The functions **fmin** and **fmax** have been superseded by **fminimum_num** and **fmaximum_num**. The **fmin** and **fmax** functions provide the minNum and maxNum operations specified in (the superseded) IEC 60559:2011.

4 Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation[439)] is implementation-defined, even if `<fenv.h>` defines the macro **FE_SNANS_ALWAYS_SIGNAL** (F.2.1). If the return expression of a **return** statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression.

5 The unary $+$ and $-$ operators raise no floating-point exceptions, even if the operand is a signaling NaN.

6 The C classification macros **fpclassify**, **iscanonical**, **isfinite**, **isinf**, **isnan**, **isnormal**, **issignaling**, **issubnormal**, **iszero**, and **signbit** provide the IEC 60559 operations indicated in the table above provided their arguments are in the format of their semantic type. Then these macros raise no floating-point exceptions, even if an argument is a signaling NaN.

7 The **signbit** macro, providing the IEC 60559 `isSignMinus` operation, determines the sign of its argument value as the sign bit of the value's representation. This applies to all values, including NaNs whose sign bit is not generally interpreted by IEC 60559.

8 The C **nearbyint** functions (7.12.9.3, F.10.6.3) provide the nearbyinteger function recommended in the Appendix to (superseded) ANSI/IEEE 854.

9 The C **nextafter** (7.12.11.3, F.10.8.3) and **nexttoward** (7.12.11.4, F.10.8.4) functions provide the **nextafter** function recommended in the Appendix to (superseded) IEC 60559:1989 (but with a minor change to better handle signed zeros).

10 The macros (7.6) **FE_DOWNWARD**, **FE_TONEAREST**, **FE_TONEARESTFROMZERO**, **FE_TOWARDZERO**, and **FE_UPWARD**, which are used in conjunction with the **fegetround** and **fesetround** functions and the **FENV_ROUND** pragma, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTiesToAway, roundTowardZero, and roundTowardPositive, respectively, for binary floating-point arithmetic. Support for the roundTiesToAway attribute for binary floating-point arithmetic, and hence for the **FE_TONEARESTFROMZERO** macro, is optional.

11 The C **fegetenv** (7.6.6.1), **feholdexcept** (7.6.6.2), **fesetenv** (7.6.6.3) and **feupdateenv** (7.6.6.4) functions provide a facility to manage the dynamic floating-point environment, comprising the IEC 60559 status flags and dynamic control modes.

12 IEC 60559 requires operations with specified operand and result formats. Therefore, math functions that are bound to IEC 60559 operations (see table above) must remove any extra range and precision from arguments or results.

13 IEC 60559 requires operations that round their result to formats the same as and wider than the operands, in addition to the operations that round their result to narrower formats (see 7.12.14). Operators ($+$, $-$, $*$, and /) whose evaluation formats are wider than the semantic type (5.2.4.2.2) might not support some of the IEEE 60559 operations, because getting a result in a given format might require a cast that could introduce an extra rounding error. The functions that round result to narrower type (7.12.14) provide the IEC 60559 operations that round result to same and wider (as well as narrower) formats, in those cases where built-in operators and casts do not. For example,

[439)] Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the "invalid" floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.

ddivl(x, y) computes a correctly rounded **double** divide of **float** x by **float** y, regardless of the evaluation method.

14 Decimal versions of the **remquo** library function are not provided. (The decimal **remainder** functions provide the remainder operation defined by IEC 60559.)

15 The binding for the convertFormat operation applies to all conversions among IEC 60559 formats. Therefore, for implementations that conform to Annex F, conversions between decimal floating types and standard floating types with IEC 60559 formats are correctly rounded and raise floating-point exceptions as specified in IEC 60559.

16 IEC 60559 specifies the convertFromHexCharacter and convertToHexCharacter operations only for binary floating-point arithmetic.

17 The integer constant 10 provides the radix operation defined in IEC 60559 for decimal floating-point arithmetic.

18 The **fe_dec_getround** (7.6.5.3) and **fe_dec_setround** (7.6.5.6) functions provide the getDecimalRoundingDirection and setDecimalRoundingDirection operations defined in IEC 60559 for decimal floating-point arithmetic. The macros (7.6) **FE_DEC_DOWNWARD**, **FE_DEC_TONEAREST**, **FE_DEC_TONEARESTFROMZERO**, **FE_DEC_TOWARDZERO**, and **FE_DEC_UPWARD**, which are used in conjunction with the **fe_dec_getround** and **fe_dec_setround** functions and the **FENV_DEC_ROUND** pragma, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTiesToAway, roundTowardZero, and roundTowardPositive, respectively, for decimal floating-point arithmetic.

19 The **llquantexpd***N* (7.12.15.4) functions compute the (quantum) exponent q defined in IEC 60559 for decimal numbers viewed as having integer significands.

20 The C functions in the following table correspond to mathematical operations recommended by IEC 60559. However, correct rounding, which IEC 60559 specifies for its operations, is not required for the C functions in the table. 7.33.8 (potentially) reserves cr_ prefixed names for functions fully matching the IEC 60559 mathematical operations. In the table, the C functions are represented by the function name without a type suffix.

IEC 60559 operation	C function	Clause
exp	**exp**	7.12.6.1, F.10.3.1
expm1	**expm1**	7.12.6.6, F.10.3.6
exp2	**exp2**	7.12.6.4, F.10.3.4
exp2m1	**exp2m1**	7.12.6.5, F.10.3.5
exp10	**exp10**	7.12.6.2, F.10.3.2
exp10m1	**exp10m1**	7.12.6.3, F.10.3.3
log	**log**	7.12.6.11, F.10.3.11
log2	**log2**	7.12.6.15, F.10.3.15
log10	**log10**	7.12.6.12, F.10.3.12
logp1	**log1p**, **logp1**	7.12.6.14, F.10.3.14
log2p1	**log2p1**	7.12.6.16, F.10.3.16
log10p1	**log10p1**	7.12.6.13, F.10.3.13
hypot	**hypot**	7.12.7.4, F.10.4.4
rSqrt	**rsqrt**	7.12.7.9, F.10.4.9
compound	**compoundn**	7.12.7.2, F.10.4.2
rootn	**rootn**	7.12.7.8, F.10.4.8
pown	**pown**	7.12.7.6, F.10.4.6
pow	**pow**	7.12.7.5, F.10.4.5
powr	**powr**	7.12.7.7, F.10.4.7
sin	**sin**	7.12.4.6, F.10.1.6
cos	**cos**	7.12.4.5, F.10.1.5
tan	**tan**	7.12.4.7, F.10.1.7
sinPi	**sinpi**	7.12.4.13, F.10.1.13
cosPi	**cospi**	7.12.4.12, F.10.1.12
... continued ...		

... continued ...		
IEC 60559 operation	C function	Clause
tanPi	`tanpi`	7.12.4.14, F.10.1.14
asinPi	`asinpi`	7.12.4.9, F.10.1.9
acosPi	`acospi`	7.12.4.8, F.10.1.8
atanPi	`atanpi`	7.12.4.10, F.10.1.10
atan2Pi	`atan2pi`	7.12.4.11, F.10.1.11
asin	`asin`	7.12.4.2, F.10.1.2
acos	`acos`	7.12.4.1, F.10.1.1
atan	`atan`	7.12.4.3, F.10.1.3
atan2	`atan2`	7.12.4.4, F.10.1.4
sinh	`sinh`	7.12.5.5, F.10.2.5
cosh	`cosh`	7.12.5.4, F.10.2.4
tanh	`tanh`	7.12.5.6, F.10.2.6
asinh	`asinh`	7.12.5.2, F.10.2.2
acosh	`acosh`	7.12.5.1, F.10.2.1
atanh	`atanh`	7.12.5.3, F.10.2.3

F.4 Floating to integer conversion

1 If the integer type is `bool`, 6.3.1.2 applies and the conversion raises no floating-point exceptions if the floating-point value is not a signaling NaN. Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the "invalid" floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the "inexact" floating-point exception is unspecified.[440)]

F.5 Conversions between binary floating types and decimal character sequences

1 The `<float.h>` header defines the macro

```
CR_DECIMAL_DIG
```

if and only if `__STDC_WANT_IEC_60559_EXT__` is defined as a macro at the point in the source file where `<float.h>` is first included. If defined, `CR_DECIMAL_DIG` expands to an integer constant expression suitable for use in conditional expression inclusion preprocessing directives whose value is a number such that conversions between all supported IEC 60559 binary formats and character sequences with at most `CR_DECIMAL_DIG` significant decimal digits are correctly rounded. The value of `CR_DECIMAL_DIG` shall be at least $M + 3$, where M is the maximum value of the *T*`_DECIMAL_DIG` macros for IEC 60559 binary formats. If the implementation correctly rounds for all numbers of significant decimal digits, then `CR_DECIMAL_DIG` shall have the value of the macro `UINTMAX_MAX`.

2 Conversions of types with IEC 60559 binary formats to character sequences with more than `CR_DECIMAL_DIG` significant decimal digits shall correctly round to `CR_DECIMAL_DIG` significant digits and pad zeros on the right.

3 Conversions from character sequences with more than `CR_DECIMAL_DIG` significant decimal digits to types with IEC 60559 binary formats shall correctly round to an intermediate character sequence with `CR_DECIMAL_DIG` significant decimal digits, according to the applicable rounding direction, and correctly round the intermediate result (having `CR_DECIMAL_DIG` significant decimal digits) to the destination type. The "inexact" floating-point exception is raised (once) if either conversion is inexact.[441)] (The second conversion may raise the "overflow" or "underflow" floating-point

[440)]IEC 60559 recommends that implicit floating-to-integer conversions raise the "inexact" floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the "inexact" floating- point exception. See `fromfp`, `ufromfp`, `fromfpx`, `ufromfpx`, `rint`, `lrint`, `llrint`, and `nearbyint` in `<math.h>`.

[441)]The intermediate conversion is exact only if all input digits after the first `CR_DECIMAL_DIG` digits are 0.

exception.)

4 The specification in this subclause assures conversion between IEC 60559 binary format and decimal character sequence follows all pertinent recommended practice. It also assures conversion from IEC 60559 format to decimal character sequence with at least *T*`_DECIMAL_DIG` digits and back, using to-nearest rounding, is the identity function, where *T* is the macro prefix for the format.

5 Functions such as `strtod` that convert character sequences to floating types honor the rounding direction. Hence, if the rounding direction might be upward or downward, the implementation cannot convert a minus-signed sequence by negating the converted unsigned sequence.

6 **NOTE 1** IEC 60559 specifies that conversion to one-digit character strings using roundTiesToEven when both choices have an odd least significant digit, shall produce the value with the larger magnitude. For example, this can happen with `9.5e2` whose nearest neighbors are `9.e2` and `1.e3`, both of which have a single odd digit in the significand part.

F.6 The `return` statement

If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment[442) to the return type of the function and the resulting value is returned to the caller.

F.7 Contracted expressions

1 A contracted expression is correctly rounded (once) and treats infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by IEC 60559.

Recommended practice

2 A contracted expression should raise floating-point exceptions in a manner generally consistent with the basic arithmetic operations.

F.8 Floating-point environment

1 The floating-point environment defined in `<fenv.h>` includes the IEC 60559 floating-point exception status flags and rounding-direction control modes. It may also include other floating-point status or modes that the implementation provides as extensions.[443)]

2 This annex does not include support for IEC 60559's optional alternate exception handling. The specification in this annex assumes IEC 60559 default exception handling: the flag is set, a default result is delivered, and execution continues. Implementations might provide alternate exception handling as an extension.

F.8.1 Environment management

1 IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. These changes to the floating-point state are treated as side effects which respect sequence points.[444)]

F.8.2 Translation

1 During translation, constant rounding direction modes (7.6.2) are in effect where specified. Elsewhere, during translation the IEC 60559 default modes are in effect:

— The rounding direction mode is rounding to nearest.

— The rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

442)Assignment removes any extra range and precision.

443)Dynamic rounding precision and trap enablement modes are examples of such extensions.

444)If the state for the `FENV_ACCESS` pragma is "off", the implementation is free to assume the dynamic floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see F.9).

Recommended practice

2 The implementation should produce a diagnostic message for each translation-time floating-point exception, other than "inexact";[445] the implementation should then proceed with the translation of the program.

F.8.3 Execution

1 At program startup the dynamic floating-point environment is initialized as prescribed by IEC 60559:

— All floating-point exception status flags are cleared.

— The dynamic rounding direction mode is rounding to nearest.

— The dynamic rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all floating-point exceptions.

F.8.4 Constant expressions

1 An arithmetic constant expression of floating type, other than one in an initializer for an object that has static or thread storage duration, is evaluated (as if) during execution; thus, it is affected by any operative floating-point control modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is "on").[446]

2 **EXAMPLE**

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
      float w[] = { 0.0/0.0 };  // raises an exception
      static float x = 0.0/0.0; // does not raise an exception
      float y = 0.0/0.0;        // raises an exception
      double z = 0.0/0.0;       // raises an exception
      /* ... */
}
```

3 For the static initialization, the division is done at translation time, raising no (execution-time) floating-point exceptions. On the other hand, for the three automatic initializations the invalid division occurs at execution time.

F.8.5 Initialization

1 All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the **FENV_ACCESS** pragma is "on"). All computation for initialization of objects that have static or thread storage duration is done (as if) at translation time.

2 **EXAMPLE**

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
```

[445] As floating constants are converted to appropriate internal representations at translation time, their conversion is subject to constant or default rounding modes and raises no execution-time floating-point exceptions (even where the state of the **FENV_ACCESS** pragma is "on"). Library functions, for example **strtod**, provide execution-time conversion of numeric strings.

[446] Where the state for the **FENV_ACCESS** pragma is "on", results of inexact expressions like `1.0/3.0` are affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` generate execution-time floating-point exceptions. The programmer can achieve the efficiency of translation-time evaluation through static initialization, such as

```
const static double one_third = 1.0/3.0;
```

```
{
        float u[] = { 1.1e75 };  // raises exceptions
        static float v = 1.1e75; // does not raise exceptions
        float w = 1.1e75;        // raises exceptions
        double x = 1.1e75;       // may raise exceptions
        float y = 1.1e75f;       // may raise exceptions
        long double z = 1.1e75;  // does not raise exceptions
        /* ... */
}
```

3 The static initialization of `v` raises no (execution-time) floating-point exceptions because its computation is done at translation time. The automatic initialization of `u` and `w` require an execution-time conversion to **`float`** of the wider value `1.1e75`, which raises floating-point exceptions. The automatic initializations of `x` and `y` entail execution-time conversion; however, in some expression evaluation methods, the conversions is not to a narrower format, in which case no floating-point exception is raised.[447] The automatic initialization of `z` entails execution-time conversion, but not to a narrower format, so no floating-point exception is raised. Note that the conversions of the floating constants `1.1e75` and `1.1e75f` to their internal representations occur at translation time in all cases.

F.8.6 Changing the environment

1 Operations defined in 6.5 and functions and macros defined for the standard libraries change floating-point status flags and control modes just as indicated by their specifications (including conformance to IEC 60559). They do not change flags or modes (so as to be detectable by the user) in any other cases.

2 If the floating-point exceptions represented by the argument to the **`feraiseexcept`** function in `<fenv.h>` include both "overflow" and "inexact", then "overflow" is raised before "inexact". Similarly, if the represented exceptions include both "underflow" and "inexact", then "underflow" is raised before "inexact".

F.9 Optimization

1 This section identifies code transformations that might subvert IEC 60559-specified behavior, and others that do not.

F.9.1 Global transformations

1 Floating-point arithmetic operations and external function calls may entail side effects which optimization shall honor, at least where the state of the **`FENV_ACCESS`** pragma is "on". The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations ($+$, $*$, etc.) implicitly read the modes and write the flags.

2 Concern about side effects may inhibit code motion and removal of seemingly useless code. For example, in

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(double x)
{
        /* ... */
        for (i = 0; i  <  n; i++) x + 1;
        /* ... */
}
```

`x+1` might raise floating-point exceptions, so cannot be removed. And since the loop body might not

[447]Use of **`float_t`** and **`double_t`** variables increases the likelihood of translation-time computation. For example, the automatic initialization

```
double_t x = 1.1e75;
```

could be done at translation time, regardless of the expression evaluation method.

execute (maybe $0 \geq$ `n`), `x+1` cannot be moved out of the loop. (Of course these optimizations are valid if the implementation can rule out the nettlesome cases.)

3 This specification does not require support for trap handlers that maintain information about the order or count of floating-point exceptions. Therefore, between function calls, floating-point exceptions need not be precise: the actual order and number of occurrences of floating-point exceptions (> 1) may vary from what the source code expresses. Thus, the preceding loop could be treated as

```
if (0 < n) x + 1;
```

F.9.2 Expression transformations

1 Valid expression transformations must preserve numerical values.

2 The equivalences noted below apply to expressions of standard floating types.

$x/2 \leftrightarrow x \times 0.5$	Although similar transformations involving inexact constants generally do not yield equivalent expressions, if the constants are exact then such transformations can be made on IEC 60559 machines and others that round perfectly.
$1 \times x$ and $x/1 \rightarrow x$	The expressions $1 \times x$, $x/1$, and x may be regarded as equivalent (on IEC 60559 machines, among others).[448]
$x/x \rightarrow 1.0$	The expressions x/x and 1.0 are not equivalent if x can be zero, infinite, or NaN.
$x - y \leftrightarrow x + (-y)$	The expressions $x - y$, $x + (-y)$, and $(-y) + x$ are equivalent (on IEC 60559 machines, among others).
$x - y \leftrightarrow -(y - x)$	The expressions $x - y$ and $-(y - x)$ are not equivalent because $1 - 1$ is $+0$ but $-(1 - 1)$ is -0 (in the default rounding direction).[449]
$x - x \rightarrow 0.0$	The expressions $x - x$ and 0.0 are not equivalent if x is a NaN or infinite.
$0 \times x \rightarrow 0.0$	The expressions $0 \times x$ and 0.0 are not equivalent if x is a NaN, infinite, or -0.
$x + 0 \rightarrow x$	The expressions $x + 0$ and x are not equivalent if x is -0, because $(-0) + (+0)$ yields $+0$ (in the default rounding direction), not -0.
$x - 0 \rightarrow x$	$(+0) - (+0)$ yields -0 when rounding is downward (toward $-\infty$), but $+0$ otherwise, and $(-0) - (+0)$ always yields -0; so, if the state of the **FENV_ACCESS** pragma is "off", promising default rounding, then the implementation can replace $x - 0$ by x, even if x might be zero.
$-x \leftrightarrow 0 - x$	The expressions $-x$ and $0 - x$ are not equivalent if x is $+0$, because $-(+0)$ yields -0, but $0 - (+0)$ yields $+0$ (unless rounding is downward).

3 For expressions of decimal floating types, transformations must preserve quantum exponents, as well as numerical values (5.2.4.2.3).

4 **EXAMPLE** $1. \times x \rightarrow x$ is valid for decimal floating-point expressions x, but $1.0 \times x \rightarrow x$ is not:

$$\begin{array}{lclclcl} 1. \times 12.34 & = & (+1, 1, 0) \times (+1, 1234, -2) & \textit{yields} & (+1, 1234, -2) & = & 12.34 \\ 1.0 \times 12.34 & = & (+1, 10, -1) \times (+1, 1234, -2) & \textit{yields} & (+1, 12340, -3) & = & 12.340 \end{array}$$

In the second case, the factor 12.34 and the result 12.340 have different quantum exponents, demonstrating that $1.0 \times x$ and x are not equivalent expressions.

[448] Implementations might have non-required features that invalidate these and other transformations that remove arithmetic operators. Examples include strict support for signaling NaNs (an optional feature) and alternate exception handling (not included in this specification).

[449] IEC 60559 prescribes a signed zero to preserve mathematical identities across certain discontinuities. Examples include:

$1/(1/\pm\infty)$ is $\pm\infty$

and

conj(csqrt(z)) is csqrt(conj(z)),

for complex z.

F.9.3 Relational operators

1 $x \neq x \to$ **false** The expression $x \neq x$ is true if x is a NaN.

$x = x \to$ **true** The expression $x = x$ is false if x is a NaN.

$x < y \to$ **isless**(x, y) (and similarly for $\leq$, $>$, $\geq$) Though equal, these expressions are not equivalent because of side effects when x or y is a NaN and the state of the **FENV_ACCESS** pragma is "on". This transformation, which would be desirable if extra code were required to cause the "invalid" floating-point exception for unordered cases, could be performed provided the state of the **FENV_ACCESS** pragma is "off".

The sense of relational operators shall be maintained. This includes handling unordered cases as expressed by the source code.

2 **EXAMPLE**

```
// calls g and raises "invalid" if a and b are unordered
if (a  <  b)
      f();
else
      g();
```

is not equivalent to

```
// calls f and raises "invalid" if a and b are unordered
if (a >= b)
      g();
else
      f();
```

nor to

```
// calls f without raising "invalid" if a and b are unordered
if (isgreaterequal(a,b))
      g();
else
      f();
```

nor, unless the state of the **FENV_ACCESS** pragma is "off", to

```
// calls g without raising "invalid" if a and b are unordered
if (isless(a,b))
      f();
else
      g();
```

but is equivalent to

```
if (!(a  <  b))
      g();
else
      f();
```

F.9.4 Constant arithmetic

1 The implementation shall honor floating-point exceptions raised by execution-time constant arithmetic wherever the state of the **FENV_ACCESS** pragma is "on". (See F.8.4 and F.8.5.) An operation on constants that raises no floating-point exception can be folded during translation, except, if the state of the **FENV_ACCESS** pragma is "on", a further check is required to assure that changing the

rounding direction to downward does not alter the sign of the result,[450] and implementations that support dynamic rounding precision modes shall assure further that the result of the operation raises no floating-point exception when converted to the semantic type of the operation.

F.10 Mathematics `<math.h>` and `<tgmath.h>`

1 This subclause contains specifications of `<math.h>` and `<tgmath.h>` facilities that are particularly suited for IEC 60559 implementations.

2 The Standard C macro **`HUGE_VAL`** and its **`float`** and **`long double`** analogs, **`HUGE_VALF`** and **`HUGE_VALL`**, expand to expressions whose values are positive infinities.

3 For each single-argument function `f` in `<math.h>` whose mathematical counterpart is symmetric (even), `f(-x)` is `f(x)` for all rounding modes and for all `x` in the (valid) domain of the function. For each single-argument function `f` in `<math.h>` whose mathematical counterpart is antisymmetric (odd), `f(-x)` is `-f(x)` for the IEC 60559 rounding modes roundTiesToEven, roundTiesToAway, and roundTowardZero, and for all `x` in the (valid) domain of the function. The **`atan2`** and **`atan2pi`** functions are odd in their first argument.

4 Special cases for functions in `<math.h>` are covered directly or indirectly by IEC 60559. The functions that IEC 60559 specifies directly are identified in F.3. The other functions in `<math.h>` treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the **`FENV_ACCESS`** pragma is "on") the floating-point status flags in a manner consistent with IEC 60559 operations.

5 The expression **`math_errhandling & MATH_ERREXCEPT`** shall evaluate to a nonzero value.

6 The functions bound to operations in IEC 60559 (F.3) are fully specified by IEC 60559, including rounding behaviors and floating-point exceptions.

7 The "invalid" and "divide-by-zero" floating-point exceptions are raised as specified in subsequent subclauses of this annex.

8 The "overflow" floating-point exception is raised whenever an infinity — or, because of rounding direction, a maximal-magnitude finite number — is returned in lieu of a finite value whose magnitude is too large.

9 The "underflow" floating-point exception is raised whenever a computed result is tiny[451] and the returned result is inexact.

10 Whether or when library functions not listed in the "Operation binding" table in F.3 raise the "inexact" floating-point exception is unspecified, unless stated otherwise.

11 Whether or when library functions not listed in the "Operation binding" table in F.3 raise a spurious "underflow" floating-point exception is not specified by this annex.[452]

12 As implied by F.8.6, library functions do not raise spurious "invalid", "overflow", or "divide-by-zero" floating-point exceptions (detectable by the user).

13 Whether the functions not listed in the "Operation binding" table in F.3 honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

14 Functions with a NaN argument return a NaN result and raise no floating-point exception, except where explicitly stated otherwise.

15 The specifications in the following subclauses append to the definitions in `<math.h>`. For families of functions, the specifications apply to all the functions even though only the principal function is shown. Unless otherwise specified, where the symbol "±" occurs in both an argument and the result, the result has the same sign as the argument.

[450] `0-0` yields `-0` instead of `+0` just when the rounding direction is downward.

[451] Tiny generally indicates having a magnitude in the subnormal range. See IEC 60559 for details about detecting tininess.

[452] It is intended that spurious "underflow" and "inexact" floating-point exceptions are raised only if avoiding them would be too costly. 7.12.1 specifies that if **`math_errhandling & MATH_ERREXCEPT`** is nonzero, then an "underflow" floating-point exception shall not be raised unless an underflow range error occurs.

Recommended practice

16 IEC 60559 specifies correct rounding for the operations in the F.3 table of operations recommended by IEC 60559, and thereby preserves useful mathematical properties such as symmetry, monotonicity, and periodicity. The corresponding functions with (potentially) reserved `cr_`-prefixed names (7.33.8) do the same. The C functions in the table, however, are not required to be correctly rounded, but implementations should still preserve as many of these useful mathematical properties as possible.

17 If a function with one or more NaN arguments returns a NaN result, the result should be the same as one of the NaN arguments (after possible type conversion), except perhaps for the sign.

F.10.1 Trigonometric functions

F.10.1.1 The acos functions

1
— **acos**(1) returns +0.

— **acos**(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.2 The asin functions

1
— **asin**(± 0) returns ± 0.

— **asin**(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.3 The atan functions

1
— **atan**(± 0) returns ± 0.

— **atan**$(\pm\infty)$ returns $\pm\frac{\pi}{2}$.

F.10.1.4 The atan2 functions

1
— **atan2**$(\pm 0, -0)$ returns $\pm\pi$.[453]

— **atan2**$(\pm 0, +0)$ returns ± 0.

— **atan2**$(\pm 0, x)$ returns $\pm\pi$ for $x < 0$.

— **atan2**$(\pm 0, x)$ returns ± 0 for $x > 0$.

— **atan2**$(y, \pm 0)$ returns $-\frac{\pi}{2}$ for $y < 0$.

— **atan2**$(y, \pm 0)$ returns $\frac{\pi}{2}$ for $y > 0$.

— **atan2**$(\pm y, -\infty)$ returns $\pm\pi$ for finite $y > 0$.

— **atan2**$(\pm y, +\infty)$ returns ± 0 for finite $y > 0$.

— **atan2**$(\pm\infty, x)$ returns $\pm\frac{\pi}{2}$ for finite x.

— **atan2**$(\pm\infty, -\infty)$ returns $\pm\frac{3\pi}{4}$.

— **atan2**$(\pm\infty, +\infty)$ returns $\pm\frac{\pi}{4}$.

F.10.1.5 The cos functions

1
— **cos**(± 0) returns 1.

— **cos**$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.1.6 The sin functions

1
— **sin**(± 0) returns ± 0.

— **sin**$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

[453] **atan2**$(0, 0)$ does not raise the "invalid" floating-point exception, nor does **atan2**$(y, 0)$ raise the "divide-by-zero" floating-point exception.

F.10.1.7 The `tan` functions

1 — `tan`(± 0) returns ± 0.

— `tan`$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.1.8 The `acospi` functions

1 — `acospi`$(+1)$ returns +0.

— `acospi`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.9 The `asinpi` functions

1 — `asinpi`(± 0) returns ± 0.

— `asinpi`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.1.10 The `atanpi` functions

1 — `atanpi`(± 0) returns ± 0.

— `atanpi`$(\pm\infty)$ returns $\pm\frac{1}{2}$.

F.10.1.11 The `atan2pi` functions

1 — `atan2pi`$(\pm 0, -0)$ returns ± 1.[454)]

— `atan2pi`$(\pm 0, +0)$ returns ± 0.

— `atan2pi`$(\pm 0, x)$ returns ± 1 for $x < 0$.

— `atan2pi`$(\pm 0, x)$ returns ± 0 for $x > 0$.

— `atan2pi`$(y, \pm 0)$ returns $-\frac{1}{2}$ for $y < 0$.

— `atan2pi`$(y, \pm 0)$ returns $+\frac{1}{2}$ for $y > 0$.

— `atan2pi`$(\pm y, -\infty)$ returns ± 1 for finite $y > 0$.

— `atan2pi`$(\pm y, +\infty)$ returns ± 0 for finite $y > 0$.

— `atan2pi`$(\pm\infty, x)$ returns $\pm\frac{1}{2}$ for finite x.

— `atan2pi`$(\pm\infty, -\infty)$ returns $\pm\frac{3}{4}$.

— `atan2pi`$(\pm\infty, +\infty)$ returns $\pm\frac{1}{4}$.

F.10.1.12 The `cospi` functions

1 — `cospi`(± 0) returns 1.

— `cospi`$(n + \frac{1}{2})$ returns $+0$, for integers n.

— `cospi`$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.1.13 The `sinpi` functions

1 — `sinpi`(± 0) returns ± 0.

— `sinpi`$(\pm n)$ returns ± 0, for positive integers n.

— `sinpi`$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

[454)] `atan2pi`$(0, 0)$ does not raise the "invalid" floating-point exception, nor does `atan2pi`$(y, 0)$ raise the "divide-by-zero" floating-point exception.

F.10.1.14 The `tanpi` functions

1 — `tanpi`(± 0) returns ± 0.

— `tanpi`(n) returns $+0$, for positive even and negative odd integers n.

— `tanpi`(n) returns -0, for positive odd and negative even integers n.

— `tanpi`$(n + \frac{1}{2})$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception, for even integers n.

— `tanpi`$(n + \frac{1}{2})$ returns $-\infty$ and raises the "divide-by-zero" floating-point exception, for odd integers n.

— `tanpi`$(\pm\infty)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.2 Hyperbolic functions

F.10.2.1 The `acosh` functions

1 — `acosh`(1) returns $+0$.

— `acosh`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 1$.

— `acosh`$(+\infty)$ returns $+\infty$.

F.10.2.2 The `asinh` functions

1 — `asinh`(± 0) returns ± 0.

— `asinh`$(\pm\infty)$ returns $\pm\infty$.

F.10.2.3 The `atanh` functions

1 — `atanh`(± 0) returns ± 0.

— `atanh`(± 1) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception.

— `atanh`(x) returns a NaN and raises the "invalid" floating-point exception for $|x| > 1$.

F.10.2.4 The `cosh` functions

1 — `cosh`(± 0) returns 1.

— `cosh`$(\pm\infty)$ returns $+\infty$.

F.10.2.5 The `sinh` functions

1 — `sinh`(± 0) returns ± 0.

— `sinh`$(\pm\infty)$ returns $\pm\infty$.

F.10.2.6 The `tanh` functions

1 — `tanh`(± 0) returns ± 0.

— `tanh`$(\pm\infty)$ returns ± 1.

F.10.3 Exponential and logarithmic functions

F.10.3.1 The `exp` functions

1 — `exp`(± 0) returns 1.

— `exp`$(-\infty)$ returns $+0$.

— `exp`$(+\infty)$ returns $+\infty$.

F.10.3.2 The `exp10` functions

1 — `exp10`(± 0) returns 1.

— `exp10`($-\infty$) returns $+0$.

— `exp10`($+\infty$) returns $+\infty$.

F.10.3.3 The `exp10m1` functions

1 — `exp10m1`(± 0) returns ± 0.

— `exp10m1`($-\infty$) returns -1.

— `exp10m1`($+\infty$) returns $+\infty$.

F.10.3.4 The `exp2` functions

1 — `exp2`(± 0) returns 1.

— `exp2`($-\infty$) returns $+0$.

— `exp2`($+\infty$) returns $+\infty$.

F.10.3.5 The `exp2m1` functions

1 — `exp2m1`(± 0) returns ± 0.

— `exp2m1`($-\infty$) returns -1.

— `exp2m1`($+\infty$) returns $+\infty$.

F.10.3.6 The `expm1` functions

1 — `expm1`(± 0) returns ± 0.

— `expm1`($-\infty$) returns -1.

— `expm1`($+\infty$) returns $+\infty$.

F.10.3.7 The `frexp` functions

1 — `frexp`($\pm 0, exp$) returns ± 0, and stores 0 in the object pointed to by `exp`.

— `frexp`($\pm\infty, exp$) returns $\pm\infty$, and stores an unspecified value in the object pointed to by `exp`.

— `frexp`(NaN, exp) stores an unspecified value in the object pointed to by `exp` (and returns a NaN).

2 `frexp` raises no floating-point exceptions if `value` is not a signaling NaN.

3 The returned value is independent of the current rounding direction mode.

4 On a binary system, the body of the `frexp` function might be

```
{
        *exp = (value == 0) ? 0: (int)(1 + logb(value));
        return scalbn(value, -(*exp));
}
```

F.10.3.8 The `ilogb` functions

1 When the correct result is representable in the range of the return type, the returned value is exact and is independent of the current rounding direction mode.

2 If the correct result is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised.

3 `ilogb`(x), for x zero, infinite, or NaN, raises the "invalid" floating-point exception and returns the value specified in 7.12.6.8.

F.10.3.9 The `ldexp` functions

1 On a binary system, `ldexp(x, exp)` is equivalent to `scalbn(x, exp)`.

F.10.3.10 The `llogb` functions

1 The `llogb` functions are equivalent to the `ilogb` functions, except that the `llogb` functions determine a result in the `long int` type.

F.10.3.11 The `log` functions

1
- `log`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log`(1) returns +0.
- `log`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- `log`$(+\infty)$ returns $+\infty$.

F.10.3.12 The `log10` functions

1
- `log10`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log10`(1) returns $+0$.
- `log10`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- `log10`$(+\infty)$ returns $+\infty$.

F.10.3.13 The `log10p1` functions

1
- `log10p1`(± 0) returns ± 0.
- `log10p1`(-1) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log10p1`(x) returns a NaN and raises the "invalid" floating-point exception for $x < -1$.
- `log10p1`$(+\infty)$ returns $+\infty$.

F.10.3.14 The `log1p` and `logp1` functions

1
- `logp1`(± 0) returns ± 0.
- `logp1`(-1) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `logp1`(x) returns a NaN and raises the "invalid" floating-point exception for $x < -1$.
- `logp1`$(+\infty)$ returns $+\infty$.

The `log1p` functions are equivalent to the `logp1` functions.

F.10.3.15 The `log2` functions

1
- `log2`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log2`(1) returns $+0$.
- `log2`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- `log2`$(+\infty)$ returns $+\infty$.

F.10.3.16 The `log2p1` functions

1
- `log2p1`(± 0) returns ± 0.
- `log2p1`(-1) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.
- `log2p1`(x) returns a NaN and raises the "invalid" floating-point exception for $x < -1$.
- `log2p1`$(+\infty)$ returns $+\infty$.

F.10.3.17 The `logb` functions

1 — `logb`(± 0) returns $-\infty$ and raises the "divide-by-zero" floating-point exception.

— `logb`$(\pm\infty)$ returns $+\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.3.18 The `modf` functions

1 — `modf`$(\pm x, iptr)$ returns a result with the same sign as x.

— `modf`$(\pm\infty, iptr)$ returns ± 0 and stores $\pm\infty$ in the object pointed to by $iptr$.

— `modf`$(\mathrm{NaN}, iptr)$ stores a NaN in the object pointed to by $iptr$ (and returns a NaN).

2 The returned values are exact and are independent of the current rounding direction mode.

3 `modf` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double modf(double value, double *iptr)
{
      int save_round = fegetround();
      fesetround(FE_TOWARDZERO);
      *iptr = nearbyint(value);
      fesetround(save_round);
      return copysign(
            isinf(value) ? 0.0:
                  value - (*iptr), value);
}
```

F.10.3.19 The `scalbn` and `scalbln` functions

1 — `scalbn`$(\pm 0, n)$ returns ± 0.

— `scalbn`$(x, 0)$ returns x.

— `scalbn`$(\pm\infty, n)$ returns $\pm\infty$.

2 If the calculation does not overflow or underflow, the returned value is exact and independent of the current rounding direction mode.

F.10.4 Power and absolute value functions

F.10.4.1 The `cbrt` functions

1 — `cbrt`(± 0) returns ± 0.

— `cbrt`$(\pm\infty)$ returns $\pm\infty$.

F.10.4.2 The `compoundn` functions

1 — `compoundn`$(x, 0)$ returns 1 for $x \geq -1$ or x a NaN.

— `compoundn`(x, n) returns a NaN and raises the "invalid" floating-point exception for $x < -1$.

— `compoundn`$(-1, n)$ returns $+\infty$ and raises the divide-by-zero floating-point exception for $n < 0$.

— `compoundn`$(-1, n)$ returns $+0$ for $n > 0$.

F.10.4.3 The `fabs` functions

1
- `fabs`(± 0) returns $+0$.
- `fabs`$(\pm\infty)$ returns $+\infty$.

2 `fabs(x)` raises no floating-point exceptions, even if `x` is a signaling NaN. The returned value is independent of the current rounding direction mode.

F.10.4.4 The `hypot` functions

1
- `hypot`(x, y), `hypot`(y, x), and `hypot`$(x, -y)$ are equivalent.
- `hypot`$(x, \pm 0)$ returns the absolute value of `x`, if `x` is not a NaN.
- `hypot`$(\pm\infty, y)$ returns $+\infty$, even if y is a NaN.
- `hypot`(x, NaN) returns a NaN, if `x` is not $\pm\infty$.

F.10.4.5 The `pow` functions

1
- `pow`$(\pm 0, y)$ returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for y an odd integer < 0.
- `pow`$(\pm 0, y)$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception for $y < 0$, finite, and not an odd integer.
- `pow`$(\pm 0, -\infty)$ returns $+\infty$.
- `pow`$(\pm 0, y)$ returns ± 0 for y an odd integer > 0.
- `pow`$(\pm 0, y)$ returns $+0$ for $y > 0$ and not an odd integer.
- `pow`$(-1, \pm\infty)$ returns 1.
- `pow`$(+1, y)$ returns 1 for any y, even a NaN.
- `pow`$(x, \pm 0)$ returns 1 for any x, even a NaN.
- `pow`(x, y) returns a NaN and raises the "invalid" floating-point exception for finite $x < 0$ and finite non-integer y.
- `pow`$(x, -\infty)$ returns $+\infty$ for $|x| < 1$.
- `pow`$(x, -\infty)$ returns $+0$ for $|x| > 1$.
- `pow`$(x, +\infty)$ returns $+0$ for $|x| < 1$.
- `pow`$(x, +\infty)$ returns $+\infty$ for $|x| > 1$.
- `pow`$(-\infty, y)$ returns -0 for y an odd integer < 0.
- `pow`$(-\infty, y)$ returns $+0$ for $y < 0$ and not an odd integer.
- `pow`$(-\infty, y)$ returns $-\infty$ for y an odd integer > 0.
- `pow`$(-\infty, y)$ returns $+\infty$ for $y > 0$ and not an odd integer.
- `pow`$(+\infty, y)$ returns $+0$ for $y < 0$.
- `pow`$(+\infty, y)$ returns $+\infty$ for $y > 0$.

F.10.4.6 The `pown` functions

1 — `pown`$(x, 0)$ returns 1 for all x not a signalling NaN.

— `pown`$(\pm 0, n)$ returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for odd $n < 0$.

— `pown`$(\pm 0, n)$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception for even $n < 0$.

— `pown`$(\pm 0, n)$ returns $+0$ for even $n > 0$.

— `pown`$(\pm 0, n)$ returns ± 0 for odd $n > 0$.

— `pown`$(\pm\infty, n)$ is equivalent to `pown`$(\pm 0, -n)$ for n not 0, except that the "divide-by-zero" floating-point exception is not raised.

F.10.4.7 The `powr` functions

1 — `powr`$(x, \pm 0)$ returns 1 for finite $x > 0$.

— `powr`$(\pm 0, y)$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception for finite $y < 0$.

— `powr`$(\pm 0, -\infty)$ returns $+\infty$.

— `powr`$(\pm 0, y)$ returns $+0$ for $y > 0$.

— `powr`$(+1, y)$ returns 1 for finite y.

— `powr`$(+1, y)$

— `powr`(x, y) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.

— `powr`$(\pm 0, \pm 0)$ returns a NaN and raises the "invalid" floating-point exception.

— `powr`$(+\infty, \pm 0)$ returns a NaN and raises the "invalid" floating-point exception.

F.10.4.8 The `rootn` functions

1 — `rootn`$(\pm 0, n)$ returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for odd $n < 0$.

— `rootn`$(\pm 0, n)$ returns $+\infty$ and raises the "divide-by-zero" floating-point exception for even $n < 0$.

— `rootn`$(\pm 0, n)$ returns $+0$ for even $n > 0$.

— `rootn`$(\pm 0, n)$ returns ± 0 for odd $n > 0$.

— `rootn`$(+\infty, n)$ returns $+\infty$ for $n > 0$.

— `rootn`$(-\infty, n)$ returns $-\infty$ for odd $n > 0$.

— `rootn`$(-\infty, n)$ returns a NaN and raises the "invalid" floating-point exception for even $n > 0$.

— `rootn`$(+\infty, n)$ returns $+0$ for $n < 0$.

— `rootn`$(-\infty, n)$ returns -0 for odd $n < 0$.

— `rootn`$(-\infty, n)$ returns a NaN and raises the "invalid" floating-point exception for even $n < 0$.

— `rootn`$(x, 0)$ returns a NaN and raises the "invalid" floating-point exception for all x (including NaN).

— `rootn`(x, n) returns a NaN and raises the "invalid" floating-point exception for $x < 0$ and n even.

F.10.4.9 The `rsqrt` functions

1 — `rsqrt`(± 0) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception.

— `rsqrt`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.

— `rsqrt`$(+\infty)$ returns $+0$.

F.10.4.10 The `sqrt` functions

1 — `sqrt`(± 0) returns ± 0.

— `sqrt`$(+\infty)$ returns $+\infty$.

— `sqrt`(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.

2 The returned value is dependent on the current rounding direction mode.

F.10.5 Error and gamma functions

F.10.5.1 The `erf` functions

1 — `erf`(± 0) returns ± 0.

— `erf`$(\pm\infty)$ returns ± 1.

F.10.5.2 The `erfc` functions

1 — `erfc`$(-\infty)$ returns 2.

— `erfc`$(+\infty)$ returns $+0$.

F.10.5.3 The `lgamma` functions

1 — `lgamma`(1) returns +0.

— `lgamma`(2) returns +0.

— `lgamma`(x) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for x a negative integer or zero.

— `lgamma`$(-\infty)$ returns $+\infty$.

— `lgamma`$(+\infty)$ returns $+\infty$.

F.10.5.4 The `tgamma` functions

1 — `tgamma`(± 0) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception.

— `tgamma`(x) returns a NaN and raises the "invalid" floating-point exception for x a negative integer.

— `tgamma`$(-\infty)$ returns a NaN and raises the "invalid" floating-point exception.

— `tgamma`$(+\infty)$ returns $+\infty$.

F.10.6 Nearest integer functions

F.10.6.1 The `ceil` functions

1 — `ceil`(± 0) returns ± 0.

— `ceil`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

3 The **`double`** version of `ceil` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double ceil(double x)
{
        double result;
        int save_round = fegetround();
        fesetround(FE_UPWARD);
        result = nearbyint(x);
        fesetround(save_round);
        return result;
}
```

F.10.6.2 The floor functions

1 — **floor**(± 0) returns ± 0.

— **floor**($\pm\infty$) returns $\pm\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

3 See the sample implementation for **ceil** in F.10.6.1.

F.10.6.3 The nearbyint functions

1 The **nearbyint** functions use IEC 60559 rounding according to the current rounding direction. They do not raise the "inexact" floating-point exception if the result differs in value from the argument.

— **nearbyint**(± 0) returns ± 0 (for all rounding directions).

— **nearbyint**($\pm\infty$) returns $\pm\infty$ (for all rounding directions).

F.10.6.4 The rint functions

1 The **rint** functions differ from the **nearbyint** functions only in that they do raise the "inexact" floating-point exception if the result differs in value from the argument.

F.10.6.5 The lrint and llrint functions

1 The **lrint** and **llrint** functions provide floating-to-integer conversion as prescribed by IEC 60559. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the "inexact" floating-point exception.

F.10.6.6 The round functions

1 — **round**(± 0) returns ± 0.

— **round**($\pm\infty$) returns $\pm\infty$.

2 The returned value is independent of the current rounding direction mode.

3 The **double** version of **round** behaves as though implemented by[455)]

```
#include <math.h>
#include <fenv.h>
  #pragma STDC FENV_ACCESS ON
  double round(double x)
  {
      double result;
      fenv_t save_env;
```

[455)]This code does not handle signaling NaNs as required of implementations that define **FE_SNANS_ALWAYS_SIGNAL**.

```
        feholdexcept(&save_env);
        result = rint(x);
        if (fetestexcept(FE_INEXACT)) {
            fesetround(FE_TOWARDZERO);
            result = rint(copysign(0.5 + fabs(x), x));
            feclearexcept(FE_INEXACT);
        }
        feupdateenv(&save_env);
        return result;
    }
```

F.10.6.7 The `lround` and `llround` functions

1 The `lround` and `llround` functions differ from the `lrint` and `llrint` functions with the default rounding direction just in that the `lround` and `llround` functions round halfway cases away from zero and need not raise the "inexact" floating-point exception for non-integer arguments that round to within the range of the return type.

F.10.6.8 The `roundeven` functions

1

— `roundeven`(± 0) returns ± 0.

— `roundeven`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

3 See the sample implementation for `ceil` in F.10.6.1.

F.10.6.9 The `trunc` functions

1 The `trunc` functions use IEC 60559 rounding toward zero (regardless of the current rounding direction).

— `trunc`(± 0) returns ± 0.

— `trunc`$(\pm\infty)$ returns $\pm\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.6.10 The `fromfp` and `ufromfp` functions

1 The `fromfp` and `ufromfp` functions raise the "invalid" floating-point exception and return a NaN if the argument `width` is zero or if the floating-point argument `x` is infinite or NaN or rounds to an integral value that is outside the range determined by the argument `width` (see 7.12.9.10).

2 These functions do not raise the "inexact" floating-point exception.

F.10.6.11 The `fromfpx` and `ufromfpx` functions

1 The `fromfpx` and `ufromfpx` functions raise the "invalid" floating-point exception and return a NaN if the argument `width` is zero or if the floating-point argument `x` is infinite or NaN or rounds to an integral value that is outside the range determined by the argument `width` (see 7.12.9.11).

2 These functions raise the "inexact" floating-point exception if a valid result differs in value from the floating-point argument `x`.

F.10.7 Remainder functions

F.10.7.1 The `fmod` functions

1 — `fmod`$(\pm 0, y)$ returns ± 0 for y not zero.

— `fmod`(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).

— **fmod**$(x, \pm\infty)$ returns x for x finite x.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

3 The **double** version of **fmod** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double fmod(double x, double y)
{
      double result;
      result = remainder(fabs(x), (y = fabs(y)));
      if (signbit(result)) result += y;
      return copysign(result, x);
}
```

F.10.7.2 The remainder functions

1
- **remainder**$(\pm 0, y)$ returns ± 0 for y not zero.
- **remainder**(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).
- **remainder**$(x, \pm\infty)$ returns x for finite x.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.7.3 The remquo functions

1 The **remquo** functions follow the specifications for the **remainder** functions.

2 If a NaN is returned, the value stored in the object pointed to by `quo` is unspecified.

3 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.8 Manipulation functions

F.10.8.1 The copysign functions

1 **copysign** is specified in the Appendix to IEC 60559.

2 **copysign**(x, y) raises no floating-point exceptions, even if x or y is a signaling NaN. The returned value is independent of the current rounding direction mode.

F.10.8.2 The nan functions

1 All IEC 60559 implementations support quiet NaNs, in all floating formats.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.8.3 The nextafter functions

1
- **nextafter**(x, y) raises the "overflow" and "inexact" floating-point exceptions for x finite and the function value infinite.
- **nextafter**(x, y) raises the "underflow" and "inexact" floating-point exceptions for the function value subnormal or zero and $x \neq y$.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.8.4 The nexttoward functions

1 No additional requirements beyond those on **nextafter**.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

F.10.8.5 The nextup functions

1
- **nextup**$(+\infty)$ returns $+\infty$.
- **nextup**$(-\infty)$ returns the largest-magnitude negative finite number in the type of the function.

2 **nextup**(x) raises no floating-point exceptions if x is not a signaling NaN. The returned value is independent of the current rounding direction mode.

F.10.8.6 The nextdown functions

1
- **nextdown**$(-\infty)$ returns $-\infty$.
- **nextdown**$(+\infty)$ returns the largest-magnitude positive finite number in the type of the function.

2 **nextdown**(x) raises no floating-point exceptions if x is not a signaling NaN. The returned value is independent of the current rounding direction mode.

F.10.8.7 The canonicalize functions

1 The canonicalize functions produce[456] the canonical version of the representation in the object pointed to by the argument x. If the input *x is a signaling NaN, the "invalid" floating-point exception is raised and a (canonical) quiet NaN (which should be the canonical version of that signaling NaN made quiet) is produced. For quiet NaN, infinity, and finite inputs, the functions raise no floating-point exceptions.

F.10.9 Maximum, minimum, and positive difference functions

F.10.9.1 The fdim functions

1 No additional requirements.

F.10.9.2 The fmax functions

1 If just one argument is a NaN, the **fmax** functions return the other argument (if both arguments are NaNs, the functions return a NaN).

2 The returned value is exact and is independent of the current rounding direction mode.

3 The body of the **fmax** function might be[457]

```
{
      double r = (isgreaterequal(x, y) || isnan(y)) ? x : y;
      (void) canonicalize(&r, &r);
      return r;
}
```

F.10.9.3 The fmin functions

1 The **fmin** functions are analogous to the **fmax** functions (see F.10.9.2).

2 The returned value is exact and is independent of the current rounding direction mode.

[456] As if *x * 1e0 were computed. Note also that this implementation does not handle signaling NaNs as required of implementations that define **FE_SNANS_ALWAYS_SIGNAL**.

[457] Ideally, **fmax** would be sensitive to the sign of zero, for example **fmax**$(-0.0, +0.0)$ would return $+0$; however, implementation in software might be impractical.

F.10.9.4 The fmaximum, fminimum, fmaximum_mag, and fminimum_mag functions

1 These functions treat NaNs like other functions in `<math.h>` (see F.10). They differ from the corresponding **fmaximum_num**, **fminimum_num**, **fmaximum_mag_num**, and **fminimum_mag_num** functions only in their treatment of NaNs.

F.10.9.5 The fmaximum_num, fminimum_num, fmaximum_mag_num, and fminimum_mag_num functions

1 These functions return the number if one argument is a number and the other is a quiet or signaling NaN. If both arguments are NaNs, a quiet NaN is returned. If an argument is a signaling NaN, the "invalid" floating-point exception is raised (even though the function returns the number when the other argument is a number).

F.10.10 Fused multiply-add

F.10.10.1 The fma functions

1
- **fma**(x, y, z) computes $xy + z$, correctly rounded once.
- **fma**(x, y, z) returns a NaN and optionally raises the "invalid" floating-point exception if one of x and y is infinite, the other is zero, and z is a NaN.
- **fma**(x, y, z) returns a NaN and raises the "invalid" floating-point exception if one of x and y is infinite, the other is zero, and z is not a NaN.
- **fma**(x, y, z) returns a NaN and raises the "invalid" floating-point exception if x times y is an exact infinity and z is also an infinity but with the opposite sign.

F.10.11 Functions that round result to narrower type

1 The functions that round their result to narrower type (7.12.14) are fully specified in IEC 60559. The returned value is dependent on the current rounding direction mode.

2 These functions treat zero and infinite arguments like the corresponding operation or function: $+$, $-$, $*$, `/`, **fma**, or **sqrt**.

F.10.12 Total order functions

1 This subclause specifies the total order functions required by IEC 60559.

2 **NOTE 1** These functions are specified only in Annex F because they depend on details of IEC 60559 formats that might not be supported if **__STDC_IEC_60559_BFP__** is not defined.

F.10.12.1 The totalorder functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_EXT__
#include <math.h>
#ifdef __STDC_IEC_60559_BFP__
int totalorder(const double *x, const double *y);
int totalorderf(const float *x, const float *y);
int totalorderl(const long double *x, const long double *y);
#endif
#ifdef __STDC_IEC_60559_DFP__
int totalorderd32(const _Decimal32 *x, const _Decimal32 *y);
int totalorderd64(const _Decimal64 *x, const _Decimal64 *y);
int totalorderd128(const _Decimal128 *x, const _Decimal128 *y);
#endif
```

Description

2 The **totalorder** functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of `*x`, `*y`. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if `*x` or `*y` is a signaling NaN.

Returns

3 The **totalorder** functions return nonzero if and only if the total order relation is true for the ordered pair of `*x`, `*y`.

F.10.12.2 The totalordermag functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_EXT__
#include <math.h>
#ifdef __STDC_IEC_60559_BFP__
int totalordermag(const double *x, const double *y);
int totalordermagf(const float *x, const float *y);
int totalordermagl(const long double *x, const long double *y);
#endif
#ifdef __STDC_IEC_60559_DFP__
int totalordermagd32(const _Decimal32 *x, const _Decimal32 *y);
int totalordermagd64(const _Decimal64 *x, const _Decimal64 *y);
int totalordermagd128(const _Decimal128 *x, const _Decimal128 *y);
#endif
```

Description

2 The **totalordermag** functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of the magnitudes of `*x`, `*y`. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if `*x` or `*y` is a signaling NaN.

Returns

3 The **totalordermag** functions return nonzero if and only if the total order relation is true for the ordered pair of the magnitudes of `*x`, `*y`.

F.10.13 Payload functions

1 IEC 60559 defines the payload to be information contained in a quiet or signaling NaN. The payload is intended for implementation-defined diagnostic information about the NaN, such as where or how the NaN was created. The implementation interprets the payload as a nonnegative integer suitable for use with the functions in this subclause, which get and set payloads. The implementation may restrict which payloads are admissible for the user to set.

2 **NOTE 1** These functions are specified only in Annex F because they depend on details of IEC 60559 formats that might not be supported if **__STDC_IEC_60559_BFP__** is not defined.

F.10.13.1 The getpayload functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_EXT__
#include <math.h>
#ifdef __STDC_IEC_60559_BFP__
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
#endif
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 getpayloadd32(const _Decimal32 *x);
_Decimal64 getpayloadd64(const _Decimal64 *x);
_Decimal128 getpayloadd128(const _Decimal128 *x);
#endif
```

Description

2 The **getpayload** functions extract the payload of a quiet or signaling NaN input and return it as a positive-signed floating-point integer. If `*x` is not a NaN, the return result is -1. These functions

raise no floating-point exceptions, even if *x is a signaling NaN.

Returns

3 The **getpayload** functions return the payload of the NaN input as a positive-signed floating-point integer.

F.10.13.2 The setpayload functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_EXT__
#include <math.h>
#ifdef __STDC_IEC_60559_BFP__
int setpayload(double *res, double pl);
int setpayloadf(float *res, float pl);
int setpayloadl(long double *res, long double pl);
#endif
#ifdef __STDC_IEC_60559_DFP__
int setpayloadd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadd128(_Decimal128 *res, _Decimal128 pl);
#endif
```

Description

2 The **setpayload** functions create a quiet NaN with the payload specified by pl and a zero sign bit and store that NaN in the object pointed to by *res. If pl is not a floating-point integer representing an admissible payload, *res is set to $+0$.

Returns

3 If the **setpayload** functions stored the specified NaN, they return a zero value, otherwise a nonzero value (and *res is set to $+0$).

F.10.13.3 The setpayloadsig functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_EXT__
#include <math.h>
#ifdef __STDC_IEC_60559_BFP__
int setpayloadsig(double *res, double pl);
int setpayloadsigf(float *res, float pl);
int setpayloadsigl(long double *res, long double pl);
#endif
#ifdef __STDC_IEC_60559_DFP__
int setpayloadsigd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadsigd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadsigd128(_Decimal128 *res, _Decimal128 pl);
#endif
```

Description

2 The **setpayloadsig** functions create a signaling NaN with the payload specified by pl and a zero sign bit and store that NaN in the object pointed to by *res. If pl is not a floating-point integer representing an admissible payload, *res is set to $+0$.

Returns

3 If the **setpayloadsig** functions stored the specified NaN, they return a zero value, otherwise a nonzero value (and *res is set to $+0$).

F.10.14 Comparison macros

1 Relational operators and their corresponding comparison macros (7.12.17) produce equivalent result values, even if argument values are represented in wider formats. Thus, comparison macro argu-

ments represented in formats wider than their semantic types are not converted to the semantic types, unless the wide evaluation method converts operands of relational operators to their semantic types. The standard wide evaluation methods characterized by **FLT_EVAL_METHOD** and **DEC_EVAL_METHOD** equal to 1 or 2 (5.2.4.2.2, 5.2.4.2.3), do not convert operands of relational operators to their semantic types.

F.10.14.1 The iseqsig macro

1 The equality operator == and the **iseqsig** macro produce equivalent results, except that the **iseqsig** macro raises the "invalid" floating-point exception if an argument is a NaN.

Annex G
(normative)
IEC 60559-compatible complex arithmetic

G.1 Introduction

1 This annex supplements Annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. An implementation that defines `__STDC_IEC_60559_COMPLEX__` or `__STDC_IEC_559_COMPLEX__` shall conform to the specifications in this annex.[458]

G.2 Types

1 There is a new keyword **`_Imaginary`**, which is used to specify imaginary types. It is used as a type specifier within declaration specifiers in the same way as **`_Complex`** is (thus, **`_Imaginary float`** is a valid type name).

2 There are three *imaginary types*, designated as **`float _Imaginary`**, **`double _Imaginary`**, and **`long double _Imaginary`**. The imaginary types (along with the real floating and complex types) are floating types.

3 For imaginary types, the corresponding real type is given by deleting the keyword **`_Imaginary`** from the type name.

4 Each imaginary type has the same representation and alignment requirements as the corresponding real type. The value of an object of imaginary type is the value of the real representation times the imaginary unit.

5 The *imaginary type domain* comprises the imaginary types.

G.3 Conventions

1 A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a quiet NaN). A complex or imaginary value is a *finite number* if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a *zero* if each of its parts is a zero.

G.4 Conversions

G.4.1 Imaginary types

1 Conversions among imaginary types follow rules analogous to those for real floating types.

G.4.2 Real and imaginary

1 When a value of imaginary type is converted to a real type other than **`bool`**[459], the result is a positive zero.

2 When a value of real type is converted to an imaginary type, the result is a positive imaginary zero.

G.4.3 Imaginary and complex

1 When a value of imaginary type is converted to a complex type, the real part of the complex result value is a positive zero and the imaginary part of the complex result value is determined by the conversion rules for the corresponding real types.

2 When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for the corresponding real types.

[458] Implementations that do not define `__STDC_IEC_60559_COMPLEX__` or `__STDC_IEC_559_COMPLEX__` are not required to conform to these specifications. The use of `__STDC_IEC_559_COMPLEX__` for this purpose is obsolescent and should be avoided in new code.

[459] See 6.3.1.2.

G.5 Binary operators

1 The following subclauses supplement 6.5 to specify the type of the result for an operation with an imaginary operand.

2 For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow; in these cases the result satisfies certain properties (specified in G.5.1), but is not completely determined.

G.5.1 Multiplicative operators

Semantics

1 If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has real type. (If either operand has complex type, then the result has complex type.)

2 If the operands are not both complex, then the result and floating-point exception behavior of the * operator is defined by the usual mathematical formula:

$*$	u	iv	$u + iv$
x	xu	$i(xv)$	$(xu) + i(xv)$
iy	$i(yu)$	$(-y)v$	$((-y)v) + i(yu)$
$x + iy$	$(xu) + i(yu)$	$((-y)v) + i(xv)$	

3 If the second operand is not complex, then the result and floating-point exception behavior of the / operator is defined by the usual mathematical formula:

/	u	iv
x	x/u	$i((-x)/v)$
iy	$i(y/u)$	y/v
$x + iy$	$(x/u) + i(y/u)$	$(y/v) + i((-x)/v)$

4 The * and / operators satisfy the following infinity properties for all real, imaginary, and complex operands:[460)]

— if one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the * operator is an infinity;

— if the first operand is an infinity and the second operand is a finite number, then the result of the / operator is an infinity;

— if the first operand is a finite number and the second operand is an infinity, then the result of the / operator is a zero;

— if the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the / operator is an infinity.

5 If both operands of the * operator are complex or if the second operand of the / operator is complex, the operator raises floating-point exceptions if appropriate for the calculation of the parts of the result, and may raise spurious floating-point exceptions.

6 **EXAMPLE 1** Multiplication of **double _Complex** operands could be implemented as follows. Note that the imaginary unit **I** has imaginary type (see G.6).

```
#include <math.h>
#include <complex.h>

/* Multiply z * w ...*/
double complex _Cmultd(double complex z, double complex w)
```

[460)]These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for **CX_LIMITED_RANGE** is “off”).

```
{
        #pragma STDC FP_CONTRACT OFF
        double a, b, c, d, ac, bd, ad, bc, x, y;
        a = creal(z); b = cimag(z);
        c = creal(w); d = cimag(w);
        ac = a * c;   bd = b * d;
        ad = a * d;   bc = b * c;
        x = ac - bd;  y = ad + bc;
        if (isnan(x) && isnan(y)) {
                /* Recover infinities that computed as NaN+iNaN ... */
                int recalc = 0;
                if (isinf(a) || isinf(b)) { // z is infinite
                        /* "Box" the infinity and change NaNs in the other factor to 0 */
                        a = copysign(isinf(a) ? 1.0: 0.0, a);
                        b = copysign(isinf(b) ? 1.0: 0.0, b);
                        if (isnan(c)) c = copysign(0.0, c);
                        if (isnan(d)) d = copysign(0.0, d);
                        recalc = 1;
                }
                if (isinf(c) || isinf(d)) { // w is infinite
                        /* "Box" the infinity and change NaNs in the other factor to 0 */
                        c = copysign(isinf(c) ? 1.0: 0.0, c);
                        d = copysign(isinf(d) ? 1.0: 0.0, d);
                        if (isnan(a)) a = copysign(0.0, a);
                        if (isnan(b)) b = copysign(0.0, b);
                        recalc = 1;
                }
                if (!recalc && (isinf(ac) || isinf(bd) ||
                                isinf(ad) || isinf(bc))) {
                        /* Recover infinities from overflow by changing NaNs to 0 ... */
                        if (isnan(a)) a = copysign(0.0, a);
                        if (isnan(b)) b = copysign(0.0, b);
                        if (isnan(c)) c = copysign(0.0, c);
                        if (isnan(d)) d = copysign(0.0, d);
                        recalc = 1;
                }
                if (recalc) {
                        x = INFINITY * (a * c - b * d);
                        y = INFINITY * (a * d + b * c);
                }
        }
        return x + I * y;
}
```

7 This implementation achieves the required treatment of infinities at the cost of only one `isnan` test in ordinary (finite) cases. It is less than ideal in that undue overflow and underflow could occur.

8 **EXAMPLE 2** Division of two `double _Complex` operands could be implemented as follows.

```
#include <math.h>
#include <complex.h>

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
        #pragma STDC FP_CONTRACT OFF
        double a, b, c, d, logbw, denom, x, y;
        int ilogbw = 0;
        a = creal(z); b = cimag(z);
        c = creal(w); d = cimag(w);
        logbw = logb(fmaximum_num(fabs(c), fabs(d)));
        if (isfinite(logbw)) {
```

```
                ilogbw = (int)logbw;
                c = scalbn(c, -ilogbw); d = scalbn(d, -ilogbw);
        }
        denom = c * c + d * d;
        x = scalbn((a * c + b * d) / denom, -ilogbw);
        y = scalbn((b * c - a * d) / denom, -ilogbw);

        /* Recover infinities and zeros that computed as NaN+iNaN;           */
        /* the only cases are nonzero/zero, infinite/finite, and finite/infinite, ...    */

        if (isnan(x) && isnan(y)) {
                if ((denom == 0.0) &&
                        (!isnan(a) || !isnan(b))) {
                        x = copysign(INFINITY, c) * a;
                        y = copysign(INFINITY, c) * b;
                }
                else if ((isinf(a) || isinf(b)) &&
                        isfinite(c) && isfinite(d)) {
                        a = copysign(isinf(a) ? 1.0: 0.0, a);
                        b = copysign(isinf(b) ? 1.0: 0.0, b);
                        x = INFINITY * (a * c + b * d);
                        y = INFINITY * (b * c - a * d);
                }
                else if ((logbw == INFINITY) &&
                        isfinite(a) && isfinite(b)) {
                        c = copysign(isinf(c) ? 1.0: 0.0, c);
                        d = copysign(isinf(d) ? 1.0: 0.0, d);
                        x = 0.0 * (a * c + b * d);
                        y = 0.0 * (b * c - a * d);
                }
        }
        return x + I * y;
}
```

9 Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the **scalbn** function, instead of with division, provides better roundoff characteristics.

G.5.2 Additive operators

Semantics

1 If both operands have imaginary type, then the result has imaginary type. (If one operand has real type and the other operand has imaginary type, or if either operand has complex type, then the result has complex type.)

2 In all cases the result and floating-point exception behavior of a $+$ or $-$ operator is defined by the usual mathematical formula:

$+$ or $-$	u	iv	$u+iv$
x	$x \pm u$	$x \pm iv$	$(x \pm u) \pm iv$
iy	$\pm u + iy$	$i(y \pm v)$	$\pm u + i(y \pm v)$
$x+iy$	$(x \pm u) + iy$	$x + i(y \pm v)$	$(x \pm u) + i(y \pm v)$

G.6 Complex arithmetic <complex.h>

1 The macros

```
imaginary
```

and

```
_Imaginary_I
```

are defined, respectively, as **_Imaginary** and a constant expression of type **float _Imaginary** with the value of the imaginary unit. The macro

```
I
```

is defined to be **_Imaginary_I** (not **_Complex_I** as stated in 7.3). Notwithstanding the provisions of 7.1.3, a program may undefine and then perhaps redefine the macro **imaginary**.

2 This subclause contains specifications for the `<complex.h>` functions that are particularly suited to IEC 60559 implementations. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol "±" occurs in both an argument and the result, the result has the same sign as the argument.

3 The functions are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, **csqrt**$(-2 \pm i0) = \pm i\sqrt{2}$.

4 Since complex and imaginary values are composed of real values, each function may be regarded as computing real values from real values. Except as noted, the functions treat real infinities, NaNs, signed zeros, subnormals, and the floating-point exception flags in a manner consistent with the specifications for real functions in F.10.[461)]

5 In subsequent subclauses in G.6 "NaN" refers to a quiet NaN. The behavior of signaling NaNs in Annex G is implementation-defined.

6 The functions **cimag**, **conj**, **cproj**, and **creal** are fully specified for all implementations, including IEC 60559 ones, in 7.3.9. These functions raise no floating-point exceptions.

7 Each of the functions **cabs** and **carg** is specified by a formula in terms of a real function (whose special cases are covered in Annex F):

```
cabs(x + iy) = hypot(x, y)
carg(x + iy) = atan2(y, x)
```

8 Each of the functions **casin**, **catan**, **ccos**, **csin**, and **ctan** is specified implicitly by a formula in terms of other complex functions (whose special cases are specified below):

```
casin(z) = −i casinh(iz)
catan(z) = −i catanh(iz)
ccos(z)  = ccosh(iz)
csin(z)  = −i csinh(iz)
ctan(z)  = −i ctanh(iz)
```

9 For the other functions, the following subclauses specify behavior for special cases, including treatment of the "invalid" and "divide-by-zero" floating-point exceptions. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. For a function f satisfying $f(\mathrm{conj}(z)) = \mathrm{conj}(f(z))$, the specifications for the upper half-plane imply the specifications for the lower half-plane; if the function f is also either even, $f(-z) = f(z)$, or odd, $f(-z) = -f(z)$, then the specifications for the first quadrant imply the specifications for the other three quadrants.

10 In the following subclauses, $\mathrm{cis}(y)$ is defined as $\cos(y) + i\sin(y)$.

G.6.1 Trigonometric functions

G.6.1.1 The cacos functions

1 — **cacos**(**conj**(z)) = **conj**(**cacos**(z)).

— **cacos**$(\pm 0 + i0)$ returns $\frac{\pi}{2} - i0$.

461) As noted in G.3, a complex value with at least one infinite part is regarded as an infinity even if its other part is a quiet NaN.

— `cacos`$(\pm 0 + i\text{NaN})$ returns $\frac{\pi}{2} + i\text{NaN}$.

— `cacos`$(x + i\infty)$ returns $\frac{\pi}{2} - i\infty$, for finite x.

— `cacos`$(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the "invalid" floating-point exception, for nonzero finite x.

— `cacos`$(-\infty + iy)$ returns $pi - i\infty$, for positive-signed finite y.

— `cacos`$(+\infty + iy)$ returns $+0 - i\infty$, for positive-signed finite y.

— `cacos`$(-\infty + i\infty)$ returns $3\frac{\pi}{4} - i\infty$.

— `cacos`$(+\infty + i\infty)$ returns $\frac{\pi}{4} - i\infty$.

— `cacos`$(\pm\infty + i\text{NaN})$ returns $\text{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).

— `cacos`$(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the "invalid" floating-point exception, for finite y.

— `cacos`$(\text{NaN} + i\infty)$ returns $\text{NaN} - i\infty$.

— `cacos`$(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

G.6.2 Hyperbolic functions

G.6.2.1 The `cacosh` functions

1 — `cacosh`(`conj`(z)) = `conj`(`cacosh`(z)).

— `cacosh`$(\pm 0 + i0)$ returns $+0 + \frac{i\pi}{2}$.

— `cacosh`$(x + i\infty)$ returns $+\infty + \frac{i\pi}{2}$, for finite x.

— `cacosh`$(0 + i\text{NaN})$ returns $\text{NaN} \pm \frac{i\pi}{2}$ (where the sign of the imaginary part of the result is unspecified).

— `cacosh`$(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the "invalid" floating-point exception, for finite nonzero x.

— `cacosh`$(-\infty + iy)$ returns $+\infty + i\pi$, for positive-signed finite y.

— `cacosh`$(+\infty + iy)$ returns $+\infty + i0$, for positive-signed finite y.

— `cacosh`$(-\infty + i\infty)$ returns $+\infty + i\frac{3\pi}{4}$.

— `cacosh`$(+\infty + i\infty)$ returns $+\infty + \frac{i\pi}{4}$.

— `cacosh`$(\pm\infty + i\text{NaN})$ returns $+\infty + i\text{NaN}$.

— `cacosh`$(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the "invalid" floating-point exception, for finite y.

— `cacosh`$(\text{NaN} + i\infty)$ returns $+\infty + i\text{NaN}$.

— `cacosh`$(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

G.6.2.2 The `casinh` functions

1 — `casinh`(`conj`(z)) = `conj`(`casinh`(z)). and `casinh` is odd.

— `casinh`($+0 + i0$) returns $0 + i0$.

— `casinh`($x + i\infty$) returns $+\infty + \frac{i\pi}{2}$ for positive-signed finite x.

— `casinh`($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite x.

— `casinh`($+\infty + iy$) returns $+\infty + i0$ for positive-signed finite y.

— `casinh`($+\infty + i\infty$) returns $+\infty + \frac{i\pi}{4}$.

— `casinh`($+\infty + i$NaN) returns $+\infty + i$NaN.

— `casinh`(NaN + $i0$) returns NaN + $i0$.

— `casinh`(NaN + iy) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite nonzero y.

— `casinh`(NaN + $i\infty$) returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified).

— `casinh`(NaN + iNaN) returns NaN + iNaN.

G.6.2.3 The `catanh` functions

1 — `catanh`(`conj`(z)) = `conj`(`catanh`(z)). and `catanh` is odd.

— `catanh`($+0 + i0$) returns $+0 + i0$.

— `catanh`($+0 + i$NaN) returns $+0 + i$NaN.

— `catanh`($+1 + i0$) returns $+\infty + i0$ and raises the "divide-by-zero" floating-point exception.

— `catanh`($x + i\infty$) returns $+0 + \frac{i\pi}{2}$, for finite positive-signed x.

— `catanh`($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for nonzero finite x.

— `catanh`($+\infty + iy$) returns $+0 + \frac{i\pi}{2}$, for finite positive-signed y.

— `catanh`($+\infty + i\infty$) returns $+0 + \frac{i\pi}{2}$.

— `catanh`($+\infty + i$NaN) returns $+0 + i$NaN.

— `catanh`(NaN + iy) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite y.

— `catanh`(NaN + $i\infty$) returns $\pm 0 + \frac{i\pi}{2}$ (where the sign of the real part of the result is unspecified).

— `catanh`(NaN + iNaN) returns NaN + iNaN.

G.6.2.4 The `ccosh` functions

1 — `ccosh`(`conj`(z)) = `conj`(`ccosh`(z)) and `ccosh` is even.

— `ccosh`($+0 + i0$) returns $1 + i0$.

— `ccosh`($+0 + i\infty$) returns NaN$\pm i0$ (where the sign of the imaginary part of the result is unspecified) and raises the "invalid" floating-point exception.

— `ccosh`($+0 + i$NaN) returns NaN$\pm i0$ (where the sign of the imaginary part of the result is unspecified).

— `ccosh`$(x + i\infty)$ returns NaN + iNaN and raises the "invalid" floating-point exception, for finite nonzero x.

— `ccosh`$(x + i\text{NaN})$ returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite nonzero x.

— `ccosh`$(+\infty + i0)$ returns $+\infty + i0$.

— `ccosh`$(+\infty + iy)$ returns $+\infty\ \text{cis}(y)$, for finite nonzero y.

— `ccosh`$(+\infty+i\infty)$ returns $\pm\infty+i$NaN (where the sign of the real part of the result is unspecified) and raises the "invalid" floating-point exception.

— `ccosh`$(+\infty + i\text{NaN})$ returns $+\infty + i$NaN.

— `ccosh`$(\text{NaN} + i0)$ returns NaN$\pm i0$ (where the sign of the imaginary part of the result is unspecified).

— `ccosh`$(\text{NaN} + iy)$ returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for all nonzero numbers y.

— `ccosh`$(\text{NaN} + i\text{NaN})$ returns NaN + iNaN.

G.6.2.5 The `csinh` functions

1 — `csinh(conj(`z`))` $=$ `conj(csinh(`z`))`. and `csinh` is odd.

— `csinh`$(+0 + i0)$ returns $+0 + i0$.

— `csinh`$(+0 + i\infty)$ returns $\pm 0 + i$NaN (where the sign of the real part of the result is unspecified) and raises the "invalid" floating-point exception.

— `csinh`$(+0 + i\text{NaN})$ returns $\pm 0 + i$NaN (where the sign of the real part of the result is unspecified).

— `csinh`$(x + i\infty)$ returns NaN + iNaN and raises the "invalid" floating-point exception, for positive finite x.

— `csinh`$(x + i\text{NaN})$ returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite nonzero x.

— `csinh`$(+\infty + i0)$ returns $+\infty + i0$.

— `csinh`$(+\infty + iy)$ returns $+\infty\ \text{cis}(y)$, for positive finite y.

— `csinh`$(+\infty+i\infty)$ returns $\pm\infty+i$NaN (where the sign of the real part of the result is unspecified) and raises the "invalid" floating-point exception.

— `csinh`$(+\infty + i\text{NaN})$ returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified).

— `csinh`$(\text{NaN} + i0)$ returns NaN + $i0$.

— `csinh`$(\text{NaN} + iy)$ returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for all nonzero numbers y.

— `csinh`$(\text{NaN} + i\text{NaN})$ returns NaN + iNaN.

G.6.2.6 The `ctanh` functions

1 — `ctanh`(`conj`(z)) = `conj`(`ctanh`(z)) and `ctanh` is odd.

— `ctanh`($+0 + i0$) returns $+0 + i0$.

— `ctanh`($0 + i\infty$) returns $0 + i$NaN and raises the "invalid" floating-point exception.

— `ctanh`($x + i\infty$) returns NaN + iNaN and raises the "invalid" floating-point exception, for finite nonzero x.

— `ctanh`($0 + i$NaN) returns $0 + i$NaN.

— `ctanh`($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite nonzero x.

— `ctanh`($+\infty + iy$) returns $1 + i0 sin(2y)$, for positive-signed finite y.

— `ctanh`($+\infty + i\infty$) returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).

— `ctanh`($+\infty + i$NaN) returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).

— `ctanh`(NaN $+ i0$) returns NaN $+ i0$.

— `ctanh`(NaN $+ iy$) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for all nonzero numbers y.

— `ctanh`(NaN + iNaN) returns NaN + iNaN.

G.6.3 Exponential and logarithmic functions

G.6.3.1 The `cexp` functions

1 — `cexp`(`conj`(z)) = `conj`(`cexp`(z)).

— `cexp`($\pm 0 + i0$) returns $1 + i0$.

— `cexp`($x + i\infty$) returns NaN + iNaN and raises the "invalid" floating-point exception, for finite x.

— `cexp`($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite x.

— `cexp`($+\infty + i0$) returns $+\infty + i0$.

— `cexp`($-\infty + iy$) returns $+0 \ \mathrm{cis}(y)$, for finite y.

— `cexp`($+\infty + iy$) returns $+\infty \ \mathrm{cis}(y)$, for finite nonzero y.

— `cexp`($-\infty + i\infty$) returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).

— `cexp`($+\infty + i\infty$) returns $\pm\infty + i$NaN and raises the "invalid" floating-point exception (where the sign of the real part of the result is unspecified).

— `cexp`($-\infty + i$NaN) returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).

— `cexp`($+\infty + i$NaN) returns $\pm\infty + i$NaN (where the sign of the real part of the result is unspecified).

— `cexp`(NaN $+ i0$) returns NaN $+ i0$.

— `cexp`(NaN $+ iy$) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for all nonzero numbers y.

— `cexp`(NaN + iNaN) returns NaN + iNaN.

G.6.3.2 The clog functions

1
- **clog**(**conj**(z)) = **conj**(**clog**(z)).
- **clog**($-0 + i0$) returns $-\infty + i\pi$ and raises the "divide-by-zero" floating-point exception.
- **clog**($+0 + i0$) returns $-\infty + i0$ and raises the "divide-by-zero" floating-point exception.
- **clog**($x + i\infty$) returns $+\infty + \frac{i\pi}{2}$, for finite x.
- **clog**($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite x.
- **clog**($-\infty + iy$) returns $+\infty + i\pi$, for finite positive-signed y.
- **clog**($+\infty + iy$) returns $+\infty + i0$, for finite positive-signed y.
- **clog**($-\infty + i\infty$) returns $+\infty + i\frac{3\pi}{4}$.
- **clog**($+\infty + i\infty$) returns $+\infty + \frac{i\pi}{4}$.
- **clog**($\pm\infty + i$NaN) returns $+\infty + i$NaN.
- **clog**(NaN $+ iy$) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite y.
- **clog**(NaN $+ i\infty$) returns $+\infty + i$NaN.
- **clog**(NaN + iNaN) returns NaN + iNaN.

G.6.4 Power and absolute-value functions

G.6.4.1 The cpow functions

1 The **cpow** functions raise floating-point exceptions if appropriate for the calculation of the parts of the result, and may also raise spurious floating-point exceptions.[462)]

G.6.4.2 The csqrt functions

1
- **csqrt**(**conj**(z)) = **conj**(**csqrt**(z)).
- **csqrt**($\pm 0 + i0$) returns $+0 + i0$.
- **csqrt**($x + i\infty$) returns $+\infty + i\infty$, for all x (including NaN).
- **csqrt**($x + i$NaN) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite x.
- **csqrt**($-\infty + iy$) returns $+0 + i\infty$, for finite positive-signed y.
- **csqrt**($+\infty + iy$) returns $+\infty + i0$, for finite positive-signed y.
- **csqrt**($-\infty + i$NaN) returns NaN$\pm i\infty$ (where the sign of the imaginary part of the result is unspecified).
- **csqrt**($+\infty + i$NaN) returns $+\infty + i$NaN.
- **csqrt**(NaN $+ iy$) returns NaN + iNaN and optionally raises the "invalid" floating-point exception, for finite y.
- **csqrt**(NaN + iNaN) returns NaN + iNaN.

462) This allows **cpow**(z, c) to be implemented as **cexp**(c**clog**(z)) without precluding implementations that treat special cases more carefully.

G.7 Type-generic math `<tgmath.h>`

1 Type-generic macros that accept complex arguments also accept imaginary arguments. If an argument is imaginary, the macro expands to an expression whose type is real, imaginary, or complex, as appropriate for the particular function: if the argument is imaginary, then the types of **cos**, **cosh**, **fabs**, **carg**, **cimag**, and **creal** are real; the types of **sin**, **tan**, **sinh**, **tanh**, **asin**, **atan**, **asinh**, and **atanh** are imaginary; and the types of the others are complex.

2 Given an imaginary argument, each of the type-generic macros **cos**, **sin**, **tan**, **cosh**, **sinh**, **tanh**, **asin**, **atan**, **asinh**, **atanh** is specified by a formula in terms of real functions:

$$
\begin{aligned}
\texttt{cos}(iy) &= \texttt{cosh}(y)\\
\texttt{sin}(iy) &= i\ \texttt{sinh}(y)\\
\texttt{tan}(iy) &= i\ \texttt{tanh}(y)\\
\texttt{cosh}(iy) &= \texttt{cos}(y)\\
\texttt{sinh}(iy) &= i\ \texttt{sin}(y)\\
\texttt{tanh}(iy) &= i\ \texttt{tan}(y)\\
\texttt{asin}(iy) &= i\ \texttt{asinh}(y)\\
\texttt{atan}(iy) &= i\ \texttt{atanh}(y)\\
\texttt{asinh}(iy) &= i\ \texttt{asin}(y)\\
\texttt{atanh}(iy) &= i\ \texttt{atan}(y)
\end{aligned}
$$

Annex H
(normative)
IEC 60559 interchange and extended types

H.1 Introduction

1 This annex specifies extension types for programming language C that have the arithmetic interchange and extended floating-point formats specified in IEC 60559. This annex also includes functions that support the non-arithmetic interchange formats in that standard. This annex was adapted from ISO/IEC TS 18661-3:2015, Floating-point extensions for C —Interchange and extended types.

2 An implementation that defines `__STDC_IEC_60559_TYPES__` to `202311L` shall conform to the specifications in this annex. An implementation may define `__STDC_IEC_60559_TYPES__` only if it defines `__STDC_IEC_60559_BFP__`, indicating support for IEC 60559 binary floating-point arithmetic, or defines `__STDC_IEC_60559_DFP__`, indicating support for IEC 60559 decimal floating-point arithmetic (or defines both). Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

H.2 Types

This clause specifies types that support IEC 60559 arithmetic interchange and extended formats. The encoding conversion functions (H.11.3) and numeric conversion functions for encodings (H.12.3, H.12.4) support the non-arithmetic interchange formats specified in IEC 60559.

H.2.1 Interchange floating types

IEC 60559 specifies interchange formats, and their encodings, which can be used for the exchange of floating-point data between implementations. These formats are identified by their radix (binary or decimal) and their storage width *N*. The two tables below give the C floating-point model parameters[463] (5.2.4.2.2) for the IEC 60559 interchange formats, where the function round() rounds to the nearest integer.

Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary$N$ ($N \geq 128$)
N, storage width in bits	16	32	64	128	N, a multiple of 32
p, precision in bits	11	24	53	113	$N - \mathrm{round}(4 \times \log_2(N)) + 13$
e_{max}, maximum exponent e	16	128	1024	16384	$2^{(N-p-1)}$
e_{min}, minimum exponent e	-13	-125	-1021	-16381	$3 - e_{max}$

Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal$N$ ($N \geq 32$)
N, storage width in bits	32	64	128	N, a multiple of 32
p, precision in bits	7	16	34	$9 \times (N \div 32) - 2$
e_{max}, maximum exponent e	97	385	6145	$3 \times 2^{((N \div 16)+3)} + 1$
e_{min}, minimum exponent e	-94	-382	-6142	$3 - e_{max}$

1 **EXAMPLE** For the binary160 format, $p = 144$, $e_{max} = 32678$ and $e_{min} = -32765$. For the decimal160 format, $p = 43$, $e_{max} = 24577$ and $e_{min} = -24574$.

2 Types designated:

```
_FloatN
```

where N is 16, 32, 64, or ≥ 128 and a multiple of 32; and, types designated

[463] In IEC 60559, normal floating-point numbers are expressed with the first significant digit to the left of the radix point. Hence the exponent in the C model (shown in the tables) is 1 more than the exponent of the same number in the IEC 60559 model.

```
_DecimalN
```

where $N \geq 32$ and a multiple of 32, are collectively called the *interchange floating types*. Each interchange floating type has the IEC 60559 interchange format corresponding to its width (N) and radix (2 for **_FloatN**, 10 for **_DecimalN**). Each interchange floating type is not compatible with any other type.

3 An implementation that defines **__STDC_IEC_60559_BFP__** and **__STDC_IEC_60559_TYPES__** shall provide **_Float32** and **_Float64** as interchange floating types with the same representation and alignment requirements as **float** and **double**, respectively. If the implementation's **long double** type supports an IEC 60559 interchange format of width $N > 64$, then the implementation shall also provide the type **_FloatN** as an interchange floating type with the same representation and alignment requirements as **long double**. The implementation may provide other radix-2 interchange floating types **_FloatN**; the set of such types supported is implementation-defined.

4 An implementation that defines **__STDC_IEC_60559_DFP__** provides the decimal floating types **_Decimal32**, **_Decimal64**, and **_Decimal128** (6.2.5). If the implementation also defines **__STDC_IEC_60559_TYPES__**, it may provide other radix-10 interchange floating types **_DecimalN**; the set of such types supported is implementation-defined.

H.2.2 Non-arithmetic interchange formats

1 An implementation supports IEC 60559 non-arithmetic interchange formats by providing the associated encoding-to-encoding conversion functions (H.11.3.2) in `<math.h>` and the string-from-encoding functions (H.12.3) and string-to-encodng functions (H.12.4) in `<stdlib.h>`.

2 An implementation that defines **__STDC_IEC_60559_BFP__** and **__STDC_IEC_60559_TYPES__** supports some IEC 60559 radix-2 interchange formats as arithmetic formats by providing types **_FloatN** (as well as **float** and **double**) with those formats. The implementation may support other IEC 60559 radix-2 interchange formats as non-arithmetic formats; the set of such formats supported is implementation-defined.

3 An implementation that defines **__STDC_IEC_60559_DFP__** and **__STDC_IEC_60559_TYPES__** supports some IEC 60559 radix-10 interchange formats as arithmetic formats by providing types **_DecimalN** with those formats. The implementations may support other IEC 60559 radix-10 interchange formats as non-arithmetic formats; the set of such formats supported is implementation-defined.

H.2.3 Extended floating types

1 For each of its basic formats, IEC 60559 specifies an extended format whose maximum exponent and precision exceed those of the basic format it is associated with. Extended formats are intended for arithmetic with more precision and exponent range than is available in the basic formats used for the input data. The extra precision and range often mitigate round-off error and eliminate overflow and underflow in intermediate computations. The table below gives the minimum values of these parameters, as defined for the C floating-point model (5.2.4.2.2). For all IEC 60559 extended (and interchange) formats, $e_{min} = 3 - e_{max}$.

Extended format parameters for floating-point numbers

	Extended formats associated with:				
Parameter	binary32	binary64	binary128	decimal64	decimal128
p digits $\geq$	32	64	128	22	40
$e_{max} \geq$	1024	16384	65536	6145	24577

2 Types designated **_Float32x**, **_Float64x**, **_Float128x**, **_Decimal64x**, and **_Decimal128x** support the corresponding IEC 60559 extended formats and are collectively called the *extended floating types*. The set of values of **_Float32x** is a subset of the set of values of **_Float64x**; the set of values of **_Float64x** is a subset of the set of values of **_Float128x**. The set of values of **_Decimal64x** is a subset of the set of values of **_Decimal128x**. Each extended floating type is not compatible with any other type. An implementation that defines **__STDC_IEC_60559_BFP__**

and **__STDC_IEC_60559_TYPES__** shall provide **_Float32x**, and may provide one or both of the types **_Float64x** and **_Float128x**. An implementation that defines **__STDC_IEC_60559_DFP__** and **__STDC_IEC_60559_TYPES__** shall provide **_Decimal64x**, and may provide **_Decimal128x**. Which (if any) of the optional extended floating types are provided is implementation-defined.

3 **NOTE 1** IEC 60559 does not specify an extended format associated with the decimal32 format, nor does this annex specify an extended type associated with the **_Decimal32** type.

4 **NOTE 2** The **_Float32x** type may have the same format as **double**. The **_Decimal64x** type may have the same format as **_Decimal128**.

H.2.4 Classification of real floating types

1 6.2.5 defines standard floating types as a collective name for the types **float**, **double** and **long double** and it defines decimal floating types as a collective name for the types **_Decimal32**, **_Decimal64**, and **_Decimal128**.

2 H.2.1 defines interchange floating types and H.2.3 defines extended floating types.

3 The types **_Float*N*** and **_Float*N*x** are collectively called *binary floating types*.

4 This subclause broadens *decimal floating types* to include the types **_Decimal*N*** and **_Decimal*N*x**, introduced in this annex, as well as **_Decimal32**, **_Decimal64**, and **_Decimal128**.

5 This sublcause broadens *real floating types* to include all interchange floating types and extended floating types, as well as standard floating types.

6 Thus, in this annex, real floating types are classified as follows:

— standard floating types, composed of **float**, **double**, **long double**;

— decimal floating types, composed of **_Decimal*N***, **_Decimal*N*x**;

— binary floating types, composed of **_Float*N***, **_Float*N*x**;

— interchange floating types, composed of **_Float*N***, **_Decimal*N***; and,

— extended floating types, composed of **_Float*N*x**, **_Decimal*N*x**.

7 **NOTE 1** Standard floating types (which have an implementation-defined radix) are not included in either binary floating types (which always have radix 2) or decimal floating types (which always have radix 10).

H.2.5 Complex types

1 This subclause broadens the C complex types (6.2.5) to also include similar types whose corresponding real parts have binary floating types. For the types **_Float*N*** and **_Float*N*x**, there are complex types designated respectively as **_Float*N* _Complex** and **_Float*N*x _Complex**. (Complex types are a conditional feature that implementations need not support; see 6.10.9.3.)

H.2.6 Imaginary types

1 This subclause broadens the C imaginary types (G.2) to also include similar types whose corresponding real parts have binary floating types. For the types **_Float*N*** and **_Float*N*x**, there are imaginary types designated respectively as **_Float*N* _Imaginary** and **_Float*N*x _Imaginary**. The imaginary types (along with the real floating and complex types) are floating types. (Annex G, including imaginary types, is a conditional feature that implementations need not support; see 6.10.9.3.)

H.3 Characteristics in <float.h>

1 This subclause enhances the **FLT_EVAL_METHOD** and **DEC_EVAL_METHOD** macros to apply to the types introduced in this annex.

2 If **FLT_RADIX** is 2, the value of **FLT_EVAL_METHOD** (5.2.4.2.2) characterizes the use of evaluation formats for standard floating types and for binary floating types:

-1 indeterminable;

`0` evaluate all operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`float`**, to the range and precision of **`float`**; evaluate all other operations and constants to the range and precision of the semantic type;

`1` evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`double`**, to the range and precision of **`double`**; evaluate all other operations and constants to the range and precision of the semantic type;

`2` evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`long double`**, to the range and precision of **`long double`**; evaluate all other operations and constants to the range and precision of the semantic type;

N where **`_FloatN`** is a supported interchange floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_FloatN`**, to the range and precision of **`_FloatN`**; evaluate all other operations and constants to the range and precision of the semantic type;

$N+$`1` where **`_FloatN`**`x` is a supported extended floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_FloatN`**`x`, to the range and precision of **`_FloatN`**`x`; evaluate all other operations and constants to the range and precision of the semantic type.

If **`FLT_RADIX`** is not `2`, the use of evaluation formats for operations and constants of binary floating types is implementation-defined.

3 The implementation-defined value of **`DEC_EVAL_METHOD`** (5.2.4.2.3) characterizes the use of evaluation formats for decimal floating types:

`-1` indeterminable;

`0` evaluate all operations and constants just to the range and precision of the type;

`1` evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_Decimal64`**, to the range and precision of **`_Decimal64`**; evaluate all other operations and constants to the range and precision of the semantic type;

`2` evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_Decimal128`**, to the range and precision of **`_Decimal128`**; evaluate all other operations and constants to the range and precision of the semantic type;

N where **`_DecimalN`** is a supported interchange floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_DecimalN`**, to the range and precision of **`_DecimalN`**; evaluate all other operations and constants to the range and precision of the semantic type;

$N+$`1` where **`_DecimalN`**`x` is a supported extended floating type, evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **`_DecimalN`**`x`, to the range and precision of **`_DecimalN`**`x`; evaluate all other operations and constants to the range and precision of the semantic type.

4 This subclause also specifies `<float.h>` macros, analogous to the macros for standard floating types, that characterize binary floating types in terms of the model presented in 5.2.4.2.2. This subclause generalizes the specification of characteristics in 5.2.4.2.3 to include the decimal floating types introduced in this annex. The prefix **`FLTN_`** indicates the type **`_FloatN`** or the non-arithmetic binary interchange format of width N. The prefix **`FLTNX_`** indicates the type **`_FloatN`**`x`. The prefix **`DECN_`** indicates the type **`_DecimalN`** or the non-arithmetic decimal interchange format of width N. The prefix **`DECNX_`** indicates the type **`_DecimalN`**`x`. The type parameters p, e_{max}, and e_{min} for

extended floating types are for the extended floating type itself, not for the basic format that it extends.

5 If `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined (by the user) at the point in the code where `<float.h>` is first included, the following applies (H.8). For each interchange or extended floating type that the implementation provides, `<float.h>` shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, `<float.h>` shall not define the associated macros in the following list, except, the implementation shall define the macros **FLT*N*_DECIMAL_DIG** and **FLT*N*_DIG** if it supports the IEC 60559 non-arithmetic binary interchange format of width N (H.2.2).

6 The signaling NaN macros

```
FLTN_SNAN
DECN_SNAN
FLTNX_SNAN
DECNX_SNAN
```

expand to constant expressions of types **_Float*N***, **_Decimal*N***, **_Float*N*x**, and **_Decimal*N*x** respectively, representing a signaling NaN. If an optional unary $+$ or $-$ operator followed by a signaling NaN macro is used for initializing an object of the same type that has static or thread storage duration, the object is initialized with a signaling NaN value.

7 The integer values given in the following lists shall be replaced by integer constant expressions:

— radix of exponent representation, b (2 for binary, 10 for decimal)

For the standard floating types, this value is implementation-defined and is specified by the macro **FLT_RADIX**. For the interchange and extended floating types there is no corresponding macro; the radix is an inherent property of the types.

— The number of bits in the floating-point significand, p

```
FLTN_MANT_DIG
FLTNX_MANT_DIG
```

— The number of digits in the coefficient, p

```
DECN_MANT_DIG
DECNX_MANT_DIG
```

— number of decimal digits, n, such that any floating-point number with p bits can be rounded to a floating-point number with n decimal digits and back again without change to the value, $\lceil 1 + p \log_{10}(2) \rceil$

```
FLTN_DECIMAL_DIG
FLTNX_DECIMAL_DIG
```

— number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded to a floating-point number with p bits and back again without a change to the q decimal digits, $\lfloor (p-1) \log_{10}(2) \rfloor$

```
FLTN_DIG
FLTNX_DIG
```

— minimum negative integer such that the radix raised to one less than that power is a normalized floating-point number, e_{min}

```
FLTN_MIN_EXP
FLTNX_MIN_EXP
DECN_MIN_EXP
DECNX_MIN_EXP
```

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10}(2)^{e_{min}-1} \rceil$

```
FLTN_MIN_10_EXP
FLTNX_MIN_10_EXP
```

— maximum negative integer such that the radix raised to one less than that power is a representable finite floating-point number, e_{max}

```
FLTN_MAX_EXP
FLTNX_MAX_EXP
DECN_MAX_EXP
DECNX_MAX_EXP
```

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1-2^{-p})2^{e_{max}}) \rfloor$

```
FLTN_MAX_10_EXP
FLTNX_MAX_10_EXP
```

— maximum representable finite floating-pointer number, $(1-b^{-p})b^{e_{max}}$

```
FLTN_MAX
FLTNX_MAX
DECN_MAX
DECNX_MAX
```

— the difference between 1 and the least value greater than 1 that is representable in the given floating type, b^{1-p}

```
FLTN_EPSILON
FLTNX_EPSILON
DECN_EPSILON
DECNX_EPSILON
```

— minimum normalized positive floating-point number, $b^{e_{min}-1}$

```
FLTN_MIN
FLTNX_MIN
DECN_MIN
DECNX_MIN
```

— minimum positive floating-point number, $b^{e_{min}-p}$

```
FLTN_TRUE_MIN
FLTNX_TRUE_MIN
DECN_TRUE_MIN
DECNX_TRUE_MIN
```

H.4 Conversions

1 This subclause enhances the usual arithmetic conversions (6.3.1.8) to handle interchange and extended floating types. It supports the IEC 60559 recommendation against allowing implicit conversions of operands to obtain a common type where the conversion is between types where neither is a subset of (or equivalent to) the other.

2 This subclause also broadens the operation binding in F.3 for the IEC 60559 convertFormat operation to apply to IEC 60559 arithmetic and non-arithmetic formats.

H.4.1 Real floating and integer

1 When a finite value of interchange or extended floating type is converted to an integer type other than **`bool`**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the "invalid" floating-point exception shall be raised and the result of the conversion is unspecified.

2 When a value of integer type is converted to an interchange or extended floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEC 60559.

H.4.2 Usual arithmetic conversions

1 If either operand is of floating type, the common real type is determined as follows:

— If one operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

— If only one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

— If both operands have the same corresponding real type, no further conversion is needed.

— If both operands have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

— Otherwise, if both operands are floating types and the sets of values of their corresponding real types are not equivalent, the operand whose set of values of its corresponding real type is a strict subset of the set of values of the corresponding real type of the other operand is converted, without change of type domain, to a type with the corresponding real type of that other operand.

— Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

- If the corresponding real type of either operand is an interchange floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same interchange floating type.
- Otherwise, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**.
- Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**[464].
- Otherwise, if the corresponding real type of either operand is **`_Float128x`** or **`_Decimal128x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float128x`** or **`_Decimal128x`**, respectively.
- Otherwise, if the corresponding real type of either operand is **`_Float64x`** or **`_Decimal64x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float64x`** or **`_Decimal64x`**, respectively.

[464]All cases where **`float`** might have the same format as another type are covered above.

H.4.3 Arithmetic and non-arithmetic formats

1 The operation binding in F.3 for the IEC 60559 convertFormat operation applies to IEC 60559 arithmetic and non-arithmetic formats as follows:

— For conversions between arithmetic formats supported by floating types (same or different radix) – casts and implicit conversions.

— For same-radix conversions between non-arithmetic interchange formats – encoding-to-encoding conversion functions (H.11.3.2).

— For conversions between non-arithmetic interchange formats (same or different radix) – compositions of string-from-encoding functions (H.12.3) (converting exactly) and string-to-encoding functions (H.12.4).

— For same-radix conversions from interchange formats supported by interchange floating types to non-arithmetic interchange formats – compositions of encode functions (H.11.3.1.1, 7.12.16.1, 7.12.16.3) and encoding-to-encoding functions (H.11.3.2).

— For same radix conversions from non-arithmetic interchange formats to interchange formats supported by interchange floating types – compositions of encoding-to-encoding conversion functions (H.11.3.2) and decode functions (H.11.3.1.2, 7.12.16.2, 7.12.16.4). See the example in H.11.3.2.1.

— For conversions from non-arithmetic interchange formats to arithmetic formats supported by floating types (same or different radix) – compositions of string-from-encoding functions (H.12.3) (converting exactly) and numeric conversion functions `strtod`, etc. (7.24.1.5, 7.24.1.6). See the example in H.12.2.

— For conversions from arithmetic formats supported by floating types to non-arithmetic interchange formats (same or different radix) – compositions of numeric conversion functions `strfromd`, etc. (7.24.1.3, 7.24.1.4) (converting exactly) and string-to-encoding functions (H.12.4).

H.5 Lexical Elements

H.5.1 Keywords

1 This subclause expands the list of keywords (6.4.1) to also include:

— `_Float`*N*, where N is 16, 32, 64, or $\geq$ 128 and a multiple of 32

— `_Float32x`

— `_Float64x`

— `_Float128x`

— `_Decimal`*N*, where N is 96 or $>$ 128 and a multiple of 32

— `_Decimal64x`

— `_Decimal128x`

H.5.2 Constants

1 This subclause specifies constants of interchange and extended floating types.

2 This subclause expands floating-suffix (6.4.4.2) to also include: `f`*N*, `F`*N*, `f`*N*`x`, `F`*N*`x`, `d`*N*, `D`*N*, `d`*N*`x`, or `D`*N*`x`.

3 A floating suffix `d`*N*, `D`*N*, `d`*N*`x`, or `D`*N*`x` shall not be used in a hexadecimal-floating-constant.

4 A floating suffix shall not designate a type that the implementation does not provide.

5 If a floating constant is suffixed by `f`*N* or `F`*N*, it has type **`_Float`*N***. If suffixed by `f`*N*`x` or `F`*N*`x`, it has type **`_Float`*N*`x`**. If suffixed by `d`*N* or `D`*N*, it has type **`_Decimal`*N***. If suffixed by `d`*N*`x` or `D`*N*`x`, it has type **`_Decimal`*N*`x`**.

6 The quantum exponent of a floating constant of decimal floating type is the same as for the result value of the corresponding **`strtod`*N*** or **`strtod`*N*`x`** function (H.12.2) for the same numeric string.

7 **NOTE 1** For N = 32, 64, and 128, the suffixes `d`*N* and `D`*N* in this subclause for constants of type **`_Decimal`*N*** are equivalent alternatives to the suffixes `df`, `dd`, `dl`, `DF`, `DD`, and `DL` in 6.4.4.2 for the same types.

H.6 Expressions

1 This subclause expands the specification of expressions to also cover interchange and extended floating types.

2 Operators involving operands of interchange or extended floating type are evaluated according to the semantics of IEC 60559, including production of decimal floating-point results with the preferred quantum exponent as specified in IEC 60559 (see 5.2.4.2.3).

3 For multiplicative operators (6.5.5), additive operators (6.5.6), relational operators (6.5.8), equality operators (6.5.9), and compound assignment operators (6.5.16.2), if either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

4 For conditional operators (6.5.15), if the second or third operand has decimal floating type, the other of those operands shall not have standard floating type, binary floating type, complex type, or imaginary type.

5 The equivalence of expressions noted in F.9.2 apply to expressions of binary floating types, as well as standard floating types.

H.7 Declarations

1 This subclause expands the list of type specifiers (6.7.2) to also include:

- **`_Float`*N***, where N is 16, 32, 64, or $\geq$ 128 and a multiple of 32
- **`_Float32x`**
- **`_Float64x`**
- **`_Float128x`**
- **`_Decimal`*N***, where N is 96 or $>$ 128 and a multiple of 32
- **`_Decimal64x`**
- **`_Decimal128x`**

2 The type specifiers **`_Float`*N*** (where N is 16, 32, 64, or $\geq$ 128 and a multiple of 32), **`_Float32x`**, **`_Float64x`**, **`_Float128x`**, **`_Decimal`*N*** (where N is 96 or $>$ 128 and a multiple of 32), **`_Decimal64x`**, and **`_Decimal128x`** shall not be used if the implementation does not support the corresponding types (see 6.10.9.3 and H.2).

3 This subclause also expands the list under Constraints in 6.7.2 to also include:

- **`_Float`*N***, where N is 16, 32, 64, or $\geq$ 128 and a multiple of 32
- **`_Float32x`**
- **`_Float64x`**
- **`_Float128x`**
- **`_Decimal`*N***, where N is 96 or $>$ 128 and a multiple of 32

— `_Decimal64x`

— `_Decimal128x`

— `_FloatN _Complex`, where N is 16, 32, 64, or $\geq$ 128 and a multiple of 32

— `_Float32x _Complex`

— `_Float64x _Complex`

— `_Float128x _Complex`

H.8 Identifiers in standard headers

1 The identifiers added to library headers by this annex are defined or declared by their respective headers only if the macro `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined (by the user) at the point in the code where the appropriate header is first included.

H.9 Complex arithmetic `<complex.h>`

1 This subclause specifies complex functions for corresponding real types that are binary floating types.

2 Each function synopsis in 7.3 specifies a family of functions including a principal function with one or more **double complex** parameters and a **double complex** or **double** return value. This subclause expands the synopsis to also include other functions, with the same name as the principal function but with `fN` and `fNx` suffixes, which are corresponding functions whose parameters and return values have corresponding real types `_FloatN` and `_FloatNx`.

3 The following function prototypes are added to the synopses of the respective subclauses in 7.3. For each binary floating type that the implementation provides, `<complex.h>` shall declare the associated functions (see H.8). Conversely, for each such type that the implementation does not provide, `<complex.h>` shall not declare the associated functions.

7.3.5 Trigonometric functions

```
_FloatN complex cacosfN(_FloatN complex z);
_FloatNx complex cacosfNx(_FloatNx complex z);
_FloatN complex casinfN(_FloatN complex z);
_FloatNx complex casinfNx(_FloatNx complex z);
_FloatN complex catanfN(_FloatN complex z);
_FloatNx complex catanfNx(_FloatNx complex z);
_FloatN complex ccosfN(_FloatN complex z);
_FloatNx complex ccosfNx(_FloatNx complex z);
_FloatN complex csinfN(_FloatN complex z);
_FloatNx complex csinfNx(_FloatNx complex z);
_FloatN complex ctanfN(_FloatN complex z);
_FloatNx complex ctanfNx(_FloatNx complex z);
```

7.3.6 Hyperbolic functions

```
_FloatN complex cacoshfN(_FloatN complex z);
_FloatNx complex cacoshfNx(_FloatNx complex z);
_FloatN complex casinhfN(_FloatN complex z);
_FloatNx complex casinhfNx(_FloatNx complex z);
_FloatN complex catanhfN(_FloatN complex z);
_FloatNx complex catanhfNx(_FloatNx complex z);
_FloatN complex ccoshfN(_FloatN complex z);
_FloatNx complex ccoshfNx(_FloatNx complex z);
_FloatN complex csinhfN(_FloatN complex z);
_FloatNx complex csinhfNx(_FloatNx complex z);
_FloatN complex ctanhfN(_FloatN complex z);
_FloatNx complex ctanhfNx(_FloatNx complex z);
```

7.3.7 Exponential and logarithmic functions

```
_FloatN complex cexpfN(_FloatN complex z);
_FloatNx complex cexpfNx(_FloatNx complex z);
_FloatN complex clogfN(_FloatN complex z);
_FloatNx complex clogfNx(_FloatNx complex z);
```

7.3.8 Power and absolute value functions

```
_FloatN cabsfN(_FloatN complex z);
_FloatNx cabsfNx(_FloatNx complex z);
_FloatN complex cpowfN(_FloatN complex x, _FloatN complex y);
_FloatNx complex cpowfNx(_FloatNx complex x, _FloatNx complex y);
_FloatN complex csqrtfN(_FloatN complex z);
_FloatNx complex csqrtfNx(_FloatNx complex z);
```

7.3.9 Manipulation functions

```
_FloatN cargfN(_FloatN complex z);
_FloatNx cargfNx(_FloatNx complex z);
_FloatN cimagfN(_FloatN complex z);
_FloatNx cimagfNx(_FloatNx complex z);
_FloatN complex CMPLXFN(_FloatN x, _FloatN y);
_FloatNx complex CMPLXFNX(_FloatNx x, _FloatNx y);
_FloatN complex conjfN(_FloatN complex z);
_FloatNx complex conjfNx(_FloatNx complex z);
_FloatN complex cprojfN(_FloatN complex z);
_FloatNx complex cprojfNx(_FloatNx complex z);
_FloatN crealfN(_FloatN complex z);
_FloatNx crealfNx(_FloatNx complex z);
```

4 For the functions listed in "future library directions" for `<complex.h>` (7.33.1), the possible suffixes are expanded to also include **f*N*** and **f*N*x**.

H.10 Floating-point environment

1 This subclause broadens the effects of the floating-point environment (7.6) to apply to types and formats specified in this annex.

2 The same floating-point status flags are used by floating-point operations for all floating types, including those types introduced in this annex, and by conversions for IEC 60559 non-arithmetic interchange formats.

3 Both the dynamic rounding direction mode accessed by **fegetround** and **fesetround** and the **FENV_ROUND** rounding control pragma apply to operations for binary floating types, as well as for standard floating types, and also to conversions for radix-2 non-arithmetic interchange formats. Likewise, both the dynamic rounding direction mode accessed by **fe_dec_getround** and **fe_dec_setround** and the **FENV_DEC_ROUND** rounding control pragmas apply to operations for all the decimal floating types, including those decimal floating types introduced in this annex, and to conversions for radix-10 non-arithmetic interchange formats.

4 In 7.6.2, the table of functions affected by constant rounding modes for standard floating types applies also for binary floating types. Each `<math.h>` function family listed in the table indicates the family of functions of all standard and binary floating types (for example, the **acos** family includes **acosf**, **acosl**, **acosf*N***, and **acosf*N*x** as well as **acos**). The **f*M*encf*N***, **strfromencf*N***, and **strtoencf*N*** functions are also affected by these constant rounding modes.

5 In 7.6.3, in the table of functions affected by constant rounding modes for decimal floating types, each `<math.h>` function family indicates the family of functions of all decimal floating types (for example, the **acos** family includes **acosd*N*** and **acosd*N*x**). The **d*M*encbind*N***, **d*M*encdecd*N***, **strfromencbind*N***, **strfromencdecd*N***, **strtoencbind*N***, and **strtoencdecd*N*** functions are also affected by these constant rounding modes.

H.11 Mathematics <math.h>

1 This subclause specifies types, functions, and macros for interchange and extended floating types, generally corresponding to those specified in 7.12 and F.10.

2 All classification macros (7.12.3) and comparison macros (7.12.17) naturally extend to handle interchange and extended floating types. For comparison macros, if neither of the sets of values of the argument formats is a subset of (or equivalent to) the other, the behavior is undefined.

3 This subclause also specifies encoding conversion functions that are part of support for the non-arithmetic interchange formats in IEC 60559 (see H.2.2).

4 Most function synopses in 7.12 specify a family of functions including a principal function with one or more **double** parameters, a **double** return value, or both. The synopses are expanded to also include functions with the same name as the principal function but with f*N*, f*N*x, d*N*, and d*N*x suffixes, which are corresponding functions whose parameters, return values, or both are of types **_Float*N***, **_Float*N*x**, **_Decimal*N***, and **_Decimal*N*x**, respectively.

5 For each interchange or extended floating type that the implementation provides, <math.h> shall define the associated types and macros and declare the associated functions (see H.8). Conversely, for each such type that the implementation does not provide, <math.h> shall not define the associated types and macros or declare the associated functions unless explicitly specified otherwise.

6 With the types

```
float_t
double_t
```

in 7.12 are included the type

```
long_double_t
```

and for each supported type **_Float*N***, the type

```
_FloatN_t
```

and for each supported type _DecimalN, the type

```
_DecimalN_t
```

These are floating types, such that:

— each of the types has at least the range and precision of the corresponding real floating type;

— **long_double_t** has at least the range and precision of **double_t**;

— **_Float*N*_t**

has at least the range and precision of **_Float*M*_t** if $N > M$;

— **_Decimal*N*_t**

has at least the range and precision of **_Decimal*M*_t** if $N > M$.

If **FLT_RADIX** is 2 and **FLT_EVAL_METHOD** (H.3) is nonnegative, then each of the types corresponding to a standard or binary floating type is the type whose range and precision are specified by **FLT_EVAL_METHOD** to be used for evaluating operations and constants of that standard or binary floating type. If **DEC_EVAL_METHOD** (H.3) is nonnegative, then each of the types corresponding to a decimal floating type is the type whose range and precision are specified by **DEC_EVAL_METHOD** to be used for evaluating operations and constants of that decimal floating type.

7 **EXAMPLE** If the supported standard and binary floating types are

Type	IEC 60559 format
`_Float16`	binary16
`float`, `_Float32`	binary32
`double`, `_Float64`, `_Float32x`	binary64
`long double`, `_Float64x`	80-bit binary64-extended
`_Float128`	binary128

then the following tables gives the types with `_t` suffixes for various values for a `FLT_EVAL_METHOD` of a given value m:

`_t` type/m	0	1	2	32
`_Float16_t`	`float`	`double`	`long double`	`_Float32`
`float_t`	`float`	`double`	`long double`	`float`
`_Float32_t`	`_Float32`	`double`	`long double`	`_Float32`
`double_t`	`double`	`double`	`long double`	`double`
`_Float64_t`	`_Float64`	`_Float64`	`long double`	`_Float64`
`long_double_t`	`long double`	`long double`	`long double`	`long double`
`_Float128_t`	`_Float128`	`_Float128`	`_Float128`	`_Float128`

`_t` type/m	64	128	33	65
`_Float16_t`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`float_t`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`_Float32_t`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`double_t`	`double`	`_Float128`	`double`	`_Float64x`
`_Float64_t`	`_Float64`	`_Float128`	`_Float64`	`_Float64x`
`long_double_t`	`long double`	`_Float128`	`long double`	`long double`
`_Float128_t`	`_Float128`	`_Float128`	`_Float128`	`_Float128`

H.11.1 Macros

1 This subclause adds macros in 7.12 as follows.

2 The macros

```
HUGE_VAL_FN
HUGE_VAL_DN
HUGE_VAL_FNX
HUGE_VAL_DNX
```

expand to constant expressions of types `_FloatN`, `_DecimalN`, `_FloatNx`, and `_DecimalNx`, respectively, representing positive infinity.

3 The macros

```
FP_FAST_FMAFN
FP_FAST_FMADN
FP_FAST_FMAFNX
FP_FAST_FMADNX
```

are, respectively, `_FloatN`, `_DecimalN`, `_FloatNx`, and `_DecimalNx` analogues of `FP_FAST_FMA`.

4 The macros in the following lists are interchange and extended floating type analogues of `FP_FAST_FADD`, `FP_FAST_FADDL`, `FP_FAST_DADDL`, etc.

5 For $M < N$, the macros

```
FP_FAST_FMADDFN
FP_FAST_FMSUBFN
FP_FAST_FMMULFN
FP_FAST_FMDIVFN
```

```
FP_FAST_FMFMAFN
FP_FAST_FMSQRTFN
FP_FAST_DMADDDN
FP_FAST_DMSUBDN
FP_FAST_DMMULDN
FP_FAST_DMDIVDN
FP_FAST_DMFMADN
FP_FAST_DMSQRTDN
```

characterize the corresponding functions whose arguments are of an interchange floating type of width N and whose return type is an interchange floating type of width M.

6 For $M \leq N$, the macros

```
FP_FAST_FMADDFNX
FP_FAST_FMSUBFNX
FP_FAST_FMMULFNX
FP_FAST_FMDIVFNX
FP_FAST_FMFMAFNX
FP_FAST_FMSQRTFNX
FP_FAST_DMADDDNX
FP_FAST_DMSUBDNX
FP_FAST_DMMULDNX
FP_FAST_DMDIVDNX
FP_FAST_DMFMADNX
FP_FAST_DMSQRTDNX
```

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width N and whose return type is an interchange floating type of width M.

7 For $M < N$, the macros

```
FP_FAST_FMXADDFN
FP_FAST_FMXSUBFN
FP_FAST_FMXMULFN
FP_FAST_FMXDIVFN
FP_FAST_FMXFMAFN
FP_FAST_FMXSQRTFN
FP_FAST_DMXADDDN
FP_FAST_DMXSUBDN
FP_FAST_DMXMULDN
FP_FAST_DMXDIVDN
FP_FAST_DMXFMADN
FP_FAST_DMXSQRTDN
```

characterize the corresponding functions whose arguments are of an interchange floating type of width N and whose return type is an extended floating type that extends a format of width M.

8 For $M < N$, the macros

```
FP_FAST_FMXADDFNX
FP_FAST_FMXSUBFNX
FP_FAST_FMXMULFNX
FP_FAST_FMXDIVFNX
FP_FAST_FMXFMAFNX
FP_FAST_FMXSQRTFNX
FP_FAST_DMXADDDNX
FP_FAST_DMXSUBDNX
FP_FAST_DMXMULDNX
FP_FAST_DMXDIVDNX
FP_FAST_DMXFMADNX
FP_FAST_DMXSQRTDNX
```

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width N and whose return type is an extended floating type that extends a format of width M.

H.11.2 Functions

1 This sublause adds the following functions to the synopses of the respective subclauses in 7.12.

7.12.4 Trigonometric functions

```
_FloatN acosfN(_FloatN x);
_FloatNx acosfNx(_FloatNx x);
_DecimalN acosdN(_DecimalN x);
_DecimalNx acosdNx(_DecimalNx x);

_FloatN asinfN(_FloatN x);
_FloatNx asinfNx(_FloatNx x);
_DecimalN asindN(_DecimalN x);
_DecimalNx asindNx(_DecimalNx x);

_FloatN atanfN(_FloatN x);
_FloatNx atanfNx(_FloatNx x);
_DecimalN atandN(_DecimalN x);
_DecimalNx atandNx(_DecimalNx x);

_FloatN atan2fN(_FloatN y, _FloatN x);
_FloatNx atan2fNx(_FloatNx y, _FloatNx x);
_DecimalN atan2dN(_DecimalN y, _DecimalN x);
_DecimalNx atan2dNx(_DecimalNx y, _DecimalNx x);

_FloatN cosfN(_FloatN x);
_FloatNx cosfNx(_FloatNx x);
_DecimalN cosdN(_DecimalN x);
_DecimalNx cosdNx(_DecimalNx x);

_FloatN sinfN(_FloatN x);
_FloatNx sinfNx(_FloatNx x);
_DecimalN sindN(_DecimalN x);
_DecimalNx sindNx(_DecimalNx x);

_FloatN tanfN(_FloatN x);
_FloatNx tanfNx(_FloatNx x);
_DecimalN tandN(_DecimalN x);
_DecimalNx tandNx(_DecimalNx x);

_FloatN acospifN(_FloatN x);
_FloatNx acospifNx(_FloatNx x);
_DecimalN acospidN(_DecimalN x);
_DecimalNx acospidNx(_DecimalNx x);

_FloatN asinpifN(_FloatN x);
_FloatNx asinpifNx(_FloatNx x);
_DecimalN asinpidN(_DecimalN x);
_DecimalNx asinpidNx(_DecimalNx x);

_FloatN atanpifN(_FloatN x);
_FloatNx atanpifNx(_FloatNx x);
_DecimalN atanpidN(_DecimalN x);
_DecimalNx atanpidNx(_DecimalNx x);

_FloatN atan2pifN(_FloatN y, _FloatN x);
_FloatNx atan2pifNx(_FloatNx y, _FloatNx x);
```

```
_DecimalN atan2pidN(_DecimalN y, _DecimalN x);
_DecimalNx atan2pidNx(_DecimalNx y, _DecimalNx x);

_FloatN cospifN(_FloatN x);
_FloatNx cospifNx(_FloatNx x);
_DecimalN cospidN(_DecimalN x);
_DecimalNx cospidNx(_DecimalNx x);

_FloatN sinpifN(_FloatN x);
_FloatNx sinpifNx(_FloatNx x);
_DecimalN sinpidN(_DecimalN x);
_DecimalNx sinpidNx(_DecimalNx x);

_FloatN tanpifN(_FloatN x);
_FloatNx tanpifNx(_FloatNx x);
_DecimalN tanpidN(_DecimalN x);
_DecimalNx tanpidNx(_DecimalNx x);
```

7.12.5 Hyperbolic functions

```
_FloatN acoshfN(_FloatN x);
_FloatNx acoshfNx(_FloatNx x);
_DecimalN acoshdN(_DecimalN x);
_DecimalNx acoshdNx(_DecimalNx x);

_FloatN asinhfN(_FloatN x);
_FloatNx asinhfNx(_FloatNx x);
_DecimalN asinhdN(_DecimalN x);
_DecimalNx asinhdNx(_DecimalNx x);

_FloatN atanhfN(_FloatN x);
_FloatNx atanhfNx(_FloatNx x);
_DecimalN atanhdN(_DecimalN x);
_DecimalNx atanhdNx(_DecimalNx x);

_FloatN coshfN(_FloatN x);
_FloatNx coshfNx(_FloatNx x);
_DecimalN coshdN(_DecimalN x);
_DecimalNx coshdNx(_DecimalNx x);

_FloatN sinhfN(_FloatN x);
_FloatNx sinhfNx(_FloatNx x);
_DecimalN sinhdN(_DecimalN x);
_DecimalNx sinhdNx(_DecimalNx x);

_FloatN tanhfN(_FloatN x);
_FloatNx tanhfNx(_FloatNx x);
_DecimalN tanhdN(_DecimalN x);
_DecimalNx tanhdNx(_DecimalNx x);
```

7.12.6 Exponential and logarithmic functions

```
_FloatN expfN(_FloatN x);
_FloatNx expfNx(_FloatNx x);
_DecimalN expdN(_DecimalN x);
_DecimalNx expdNx(_DecimalNx x);

_FloatN exp10fN(_FloatN x);
_FloatNx exp10fNx(_FloatNx x);
_DecimalN exp10dN(_DecimalN x);
_DecimalNx exp10dNx(_DecimalNx x);
```

```
_FloatN exp10m1fN(_FloatN x);
_FloatNx exp10m1fNx(_FloatNx x);
_DecimalN exp10m1dN(_DecimalN x);
_DecimalNx exp10m1dNx(_DecimalNx x);

_FloatN exp2fN(_FloatN x);
_FloatNx exp2fNx(_FloatNx x);
_DecimalN exp2dN(_DecimalN x);
_DecimalNx exp2dNx(_DecimalNx x);

_FloatN exp2m1fN(_FloatN x);
_FloatNx exp2m1fNx(_FloatNx x);
_DecimalN exp2m1dN(_DecimalN x);
_DecimalNx exp2m1dNx(_DecimalNx x);

_FloatN expm1fN(_FloatN x);
_FloatNx expm1fNx(_FloatNx x);
_DecimalN expm1dN(_DecimalN x);
_DecimalNx expm1dNx(_DecimalNx x);

_FloatN frexpfN(_FloatN value, int *exp);
_FloatNx frexpfNx(_FloatNx value, int *exp);
_DecimalN frexpdN(_DecimalN value, int *exp);
_DecimalNx frexpdNx(_DecimalNx value, int *exp);

int ilogbfN(_FloatN x);
int ilogbfNx(_FloatNx x);
int ilogbdN(_DecimalNx x);
int ilogbdNx(_DecimalNx x);

_FloatN ldexpfN(_FloatN value, int exp);
_FloatNx ldexpfNx(_FloatNx value, int exp);
_DecimalN ldexpdN(_DecimalN value, int exp);
_DecimalNx ldexpdNx(_DecimalNx value, int exp);

long int llogbfN(_FloatN x);
long int llogbfNx(_FloatNx x);
long int llogbdN(_DecimalN x);
long int llogbdNx(_DecimalNx x);

_FloatN logfN(_FloatN x);
_FloatNx logfNx(_FloatNx x);
_DecimalN logdN(_DecimalN x);
_DecimalNx logdNx(_DecimalNx x);

_FloatN log10fN(_FloatN x);
_FloatNx log10fNx(_FloatNx x);
_DecimalN log10dN(_DecimalN x);
_DecimalNx log10dNx(_DecimalNx x);

_FloatN log10p1fN(_FloatN x);
_FloatNx log10p1fNx(_FloatNx x);
_DecimalN log10p1dN(_DecimalN x);
_DecimalNx log10p1dNx(_DecimalNx x);

_FloatN log1pfN(_FloatN x);
_FloatNx log1pfNx(_FloatNx x);
_FloatN logp1fN(_FloatN x);
_FloatNx logp1fNx(_FloatNx x);
_DecimalN log1pdN(_DecimalN x);
```

```
_DecimalNx log1pdNx(_DecimalNx x);
_DecimalN logp1dN(_DecimalN x);
_DecimalNx logp1dNx(_DecimalNx x);

_FloatN log2fN(_FloatN x);
_FloatNx log2fNx(_FloatNx x);
_DecimalN log2dN(_DecimalN x);
_DecimalNx log2dNx(_DecimalNx x);

_FloatN log2p1fN(_FloatN x);
_FloatNx log2p1fNx(_FloatNx x);
_DecimalN log2p1dN(_DecimalN x);
_DecimalNx log2p1dNx(_DecimalNx x);

_FloatN logbfN(_FloatN x);
_FloatNx logbfNx(_FloatNx x);
_DecimalN logbdN(_DecimalN x);
_DecimalNx logbdNx(_DecimalNx x);

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatNx modffNx(_FloatNx x, _FloatNx *iptr);
_DecimalN modfdN(_DecimalN x, _DecimalN *iptr);
_DecimalNx modfdNx(_DecimalNx x, _DecimalNx *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatNx scalbnfNx(_FloatNx value, int exp);
_DecimalN scalbndN(_DecimalN value, int exp);
_DecimalNx scalbndNx(_DecimalNx value, int exp);

_FloatN scalblnfN(_FloatN value, long int exp);
_FloatNx scalblnfNx(_FloatNx value, long int exp);
_DecimalN scalblndN(_DecimalN value, long int exp);
_DecimalNx scalblndNx(_DecimalNx value, long int exp);
```

7.12.7 Power and absolute-value functions

```
_FloatN cbrtfN(_FloatN x);
_FloatNx cbrtfNx(_FloatNx x);
_DecimalN cbrtdN(_DecimalN x);
_DecimalNx cbrtdNx(_DecimalNx x);

_FloatN compoundnfN(_FloatN x, long long int n);
_FloatNx compoundnfNx(_FloatNx x, long long int n);
_DecimalN compoundndN(_DecimalN x, long long int n);
_DecimalNx compoundndNx(_DecimalNx x, long long int n);

_FloatN fabsfN(_FloatN x);
_FloatNx fabsfNx(_FloatNx x);
_DecimalN fabsdN(_DecimalN x);
_DecimalNx fabsdNx(_DecimalNx x);

_FloatN hypotfN(_FloatN x, _FloatN y);
_FloatNx hypotfNx(_FloatNx x, _FloatNx y);
_DecimalN hypotdN(_DecimalN x, _DecimalN y);
_DecimalNx hypotdNx(_DecimalNx x, _DecimalNx y);

_FloatN powfN(_FloatN x, _FloatN y);
_FloatNx powfNx(_FloatNx x, _FloatNx y);
_DecimalN powdN(_DecimalN x, _DecimalN y);
_DecimalNx powdNx(_DecimalNx x, _DecimalNx y);
```

```
_FloatN pownfN(_FloatN x, long long int n);
_FloatNx pownfNx(_FloatNx x, long long int n);
_DecimalN powndN(_DecimalN x, long long int n);
_DecimalNx powndNx(_DecimalNx x, long long int n);

_FloatN powrfN(_FloatN x, _FloatN y);
_FloatNx powrfNx(_FloatNx x, _FloatNx y);
_DecimalN powrdN(_DecimalN x, _DecimalN y);
_DecimalNx powrdNx(_DecimalNx x, _DecimalNx y);

_FloatN rootnfN(_FloatN x, long long int n);
_FloatNx rootnfNx(_FloatNx x, long long int n);
_DecimalN rootndN(_DecimalN x, long long int n);
_DecimalNx rootndNx(_DecimalNx x, long long int n);

_FloatN rsqrtfN(_FloatN x);
_FloatNx rsqrtfNx(_FloatNx x);
_DecimalN rsqrtdN(_DecimalN x);
_DecimalNx rsqrtdNx(_DecimalNx x);

_FloatN sqrtfN(_FloatN x);
_FloatNx sqrtfNx(_FloatNx x);
_DecimalN sqrtdN(_DecimalN x);
_DecimalNx sqrtdNx(_DecimalNx x);
```

7.12.8 Error and gamma functions

```
_FloatN erffN(_FloatN x);
_FloatNx erffNx(_FloatNx x);
_DecimalN erfdN(_DecimalN x);
_DecimalNx erfdNx(_DecimalNx x);

_FloatN erfcfN(_FloatN x);
_FloatNx erfcfNx(_FloatNx x);
_DecimalN erfcdN(_DecimalN x);
_DecimalNx erfcdNx(_DecimalNx x);

_FloatN lgammafN(_FloatN x);
_FloatNx lgammafNx(_FloatNx x);
_DecimalN lgammadN(_DecimalN x);
_DecimalNx lgammadNx(_DecimalNx x);

_FloatN tgammafN(_FloatN x);
_FloatNx tgammafNx(_FloatNx x);
_DecimalN tgammadN(_DecimalN x);
_DecimalNx tgammadNx(_DecimalNx x);
```

7.12.9 Nearest integer functions

```
_FloatN ceilfN(_FloatN x);
_FloatNx ceilfNx(_FloatNx x);
_DecimalN ceildN(_DecimalN x);
_DecimalNx ceildNx(_DecimalNx x);

_FloatN floorfN(_FloatN x);
_FloatNx floorfNx(_FloatNx x);
_DecimalN floordN(_DecimalN x);
_DecimalNx floordNx(_DecimalNx x);

_FloatN nearbyintfN(_FloatN x);
_FloatNx nearbyintfNx(_FloatNx x);
```

```
_DecimalN nearbyintdN(_DecimalN x);
_DecimalNx nearbyintdNx(_DecimalNx x);

_FloatN rintfN(_FloatN x);
_FloatNx rintfNx(_FloatNx x);
_DecimalN rintdN(_DecimalN x);
_DecimalNx rintdNx(_DecimalNx x);

long int lrintfN(_FloatN x);
long int lrintfNx(_FloatNx x);
long int lrintdN(_DecimalN x);
long int lrintdNx(_DecimalNx x);

long long int llrintfN(_FloatN x);
long long int llrintfNx(_FloatNx x);
long long int llrintdN(_DecimalN x);
long long int llrintdNx(_DecimalNx x);

_FloatN roundfN(_FloatN x);
_FloatNx roundfNx(_FloatNx x);
_DecimalN rounddN(_DecimalN x);
_DecimalNx rounddNx(_DecimalNx x);

long int lroundfN(_FloatN x);
long int lroundfNx(_FloatNx x);
long int lrounddN(_DecimalN x);
long int lrounddNx(_DecimalNx x);

long long int llroundfN(_FloatN x);
long long int llroundfNx(_FloatNx x);
long long int llrounddN(_DecimalN x);
long long int llrounddNx(_DecimalNx x);

_FloatN roundevenfN(_FloatN x);
_FloatNx roundevenfNx(_FloatNx x);
_DecimalN roundevendN(_DecimalN x);
_DecimalNx roundevendNx(_DecimalNx x);

_FloatN truncfN(_FloatN x);
_FloatNx truncfNx(_FloatNx x);
_DecimalN truncdN(_DecimalN x);
_DecimalNx truncdNx(_DecimalNx x);

_FloatN fromfpfN(_FloatN x, int rnd, unsigned int width);
_FloatNx fromfpfNx(_FloatNx x, int rnd, unsigned int width);
_DecimalN fromfpdN(_DecimalN x, int rnd, unsigned int width);
_DecimalNx fromfpdNx(_DecimalNx x, int rnd, unsigned int width);
_FloatN ufromfpfN(_FloatN x, int rnd, unsigned int width);
_FloatNx ufromfpfNx(_FloatNx x, int rnd, unsigned int width);
_DecimalN ufromfpdN(_DecimalN x, int rnd, unsigned int width);
_DecimalNx ufromfpdNx(_DecimalNx x, int rnd, unsigned int width);

_FloatN fromfpxfN(_FloatN x, int rnd, unsigned int width);
_FloatNx fromfpxfNx(_FloatNx x, int rnd, unsigned int width);
_DecimalN fromfpxdN(_DecimalN x, int rnd, unsigned int width);
_DecimalNx fromfpxdNx(_DecimalNx x, int rnd, unsigned int width);
_FloatN ufromfpxfN(_FloatN x, int rnd, unsigned int width);
_FloatNx ufromfpxfNx(_FloatNx x, int rnd, unsigned int width);
_DecimalN ufromfpxdN(_DecimalN x, int rnd, unsigned int width);
_DecimalNx ufromfpxdNx(_DecimalNx x, int rnd, unsigned int width);
```

7.12.10.2 Remainder functions

```
_FloatN fmodfN(_FloatN x, _FloatN y);
_FloatNx fmodfNx(_FloatNx x, _FloatNx y);
_DecimalN fmoddN(_DecimalN x, _DecimalN y);
_DecimalNx fmoddNx(_DecimalNx x, _DecimalNx y);

_FloatN remainderfN(_FloatN x, _FloatN y);
_FloatNx remainderfNx(_FloatNx x, _FloatNx y);
_DecimalN remainderdN(_DecimalN x, _DecimalN y);
_DecimalNx remainderdNx(_DecimalNx x, _DecimalNx y);

_FloatN remquofN(_FloatN x, _FloatN y, int *quo);
_FloatNx remquofNx(_FloatNx x, _FloatNx y, int *quo);
```

7.12.11 Manipulation functions

```
_FloatN copysignfN(_FloatN x, _FloatN y);
_FloatNx copysignfNx(_FloatNx x, _FloatNx y);
_DecimalN copysigndN(_DecimalN x, _DecimalN y);
_DecimalNx copysigndNx(_DecimalNx x, _DecimalNx y);

_FloatN nanfN(const char *tagp);
_FloatNx nanfNx(const char *tagp);
_DecimalN nandN(const char *tagp);
_DecimalNx nandNx(const char *tagp);

_FloatN nextafterfN(_FloatN x, _FloatN y);
_FloatNx nextafterfNx(_FloatNx x, _FloatNx y);
_DecimalN nextafterdN(_DecimalN x, _DecimalN y);
_DecimalNx nextafterdNx(_DecimalNx x, _DecimalNx y);

_FloatN nextupfN(_FloatN x);
_FloatNx nextupfNx(_FloatNx x);
_DecimalN nextupdN(_DecimalN x);
_DecimalNx nextupdNx(_DecimalNx x);

_FloatN nextdownfN(_FloatN x);
_FloatNx nextdownfNx(_FloatNx x);
_DecimalN nextdowndN(_DecimalN x);
_DecimalNx nextdowndNx(_DecimalNx x);

int canonicalizefN(_FloatN * cx, const _FloatN * x);
int canonicalizefNx(_FloatNx * cx, const _FloatNx * x);
int canonicalizedN(_DecimalN * cx, const _DecimalN * x);
int canonicalizedNx(_DecimalNx * cx, const _DecimalNx * x);
```

7.12.12 Maximum, minimum, and positive difference functions

```
_FloatN fdimfN(_FloatN x, _FloatN y);
_FloatNx fdimfNx(_FloatNx x, _FloatNx y);
_DecimalN fdimdN(_DecimalN x, _DecimalN y);
_DecimalNx fdimdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaximumfN(_FloatN x, _FloatN y);
_FloatNx fmaximumfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaximumdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaximumdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminimumfN(_FloatN x, _FloatN y);
_FloatNx fminimumfNx(_FloatNx x, _FloatNx y);
```

```
_DecimalN fminimumdN(_DecimalN x, _DecimalN y);
_DecimalNx fminimumdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaximum_magfN(_FloatN x, _FloatN y);
_FloatNx fmaximum_magfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaximum_magdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaximum_magdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminimum_magfN(_FloatN x, _FloatN y);
_FloatNx fminimum_magfNx(_FloatNx x, _FloatNx y);
_DecimalN fminimum_magdN(_DecimalN x, _DecimalN y);
_DecimalNx fminimum_magdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaximum_numfN(_FloatN x, _FloatN y);
_FloatNx fmaximum_numfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaximum_numdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaximum_numdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminimum_numfN(_FloatN x, _FloatN y);
_FloatNx fminimum_numfNx(_FloatNx x, _FloatNx y);
_DecimalN fminimum_numdN(_DecimalN x, _DecimalN y);
_DecimalNx fminimum_numdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaximum_mag_numfN(_FloatN x, _FloatN y);
_FloatNx fmaximum_mag_numfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaximum_mag_numdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaximum_mag_numdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminimum_mag_numfN(_FloatN x, _FloatN y);
_FloatNx fminimum_mag_numfNx(_FloatNx x, _FloatNx y);
_DecimalN fminimum_mag_numdN(_DecimalN x, _DecimalN y);
_DecimalNx fminimum_mag_numdNx(_DecimalNx x, _DecimalNx y);
```

7.12.13.1 Fused multiply-add

```
_FloatN fmafN(_FloatN x, _FloatN y, _FloatN z);
_FloatNx fmafNx(_FloatNx x, _FloatNx y, _FloatNx z);
_DecimalN fmadN(_DecimalN x, _DecimalN y, _DecimalN z);
_DecimalNx fmadNx(_DecimalNx x, _DecimalNx y, _DecimalNx z);
```

7.12.14 Functions that round result to narrower type

```
_FloatM fMaddfN(_FloatN x, _FloatN y); // M < N
_FloatM fMaddfNx(_FloatNx x, _FloatNx y); // M ≤ N
_FloatMx fMxaddfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxaddfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMadddN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMadddNx(_DecimalNx x, _DecimalNx y); // M ≤ N
_DecimalMx dMxadddN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxadddNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMsubfN(_FloatN x, _FloatN y); // M < N
_FloatM fMsubfNx(_FloatNx x, _FloatNx y); // M ≤ N
_FloatMx fMxsubfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxsubfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMsubdN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMsubdNx(_DecimalNx x, _DecimalNx y); // M ≤ N
_DecimalMx dMxsubdN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxsubdNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMmulfN(_FloatN x, _FloatN y); // M < N
```

```
_FloatM fMmulfNx(_FloatNx x, _FloatNx y); // M ≤ N
_FloatMx fMxmulfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxmulfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMmuldN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMmuldNx(_DecimalNx x, _DecimalNx y); // M ≤ N
_DecimalMx dMxmuldN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxmuldNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMdivfNx(_FloatNx x, _FloatNx y); // M ≤ N
_FloatMx fMxdivfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxdivfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMdivdN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMdivdNx(_DecimalNx x, _DecimalNx y); // M ≤ N
_DecimalMx dMxdivdN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxdivdNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmafNx(_FloatNx x, _FloatNx y, _FloatNx z); // M ≤ N
_FloatMx fMxfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatMx fMxfmafNx(_FloatNx x, _FloatNx y, _FloatNx z); // M < N
_DecimalM dMfmadN(_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_DecimalM dMfmadNx(_DecimalNx x, _DecimalNx y, _DecimalNx z); // M ≤ N
_DecimalMx dMxfmadN(_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_DecimalMx dMxfmadNx(_DecimalNx x, _DecimalNx y, _DecimalNx z); // M < N

_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfNx(_FloatNx x); // M ≤ N
_FloatMx fMxsqrtfN(_FloatN x); // M < N
_FloatMx fMxsqrtfNx(_FloatNx x); // M < N
_DecimalM dMsqrtdN(_DecimalN x); // M < N
_DecimalM dMsqrtdNx(_DecimalNx x); // M ≤ N
_DecimalMx dMxsqrtdN(_DecimalN x); // M < N
_DecimalMx dMxsqrtdNx(_DecimalNx x); // M < N
```

7.12.15 Quantum and quantum exponent functions

```
_DecimalN quantizedN(_DecimalN x, _DecimalN y);
_DecimalNx quantizedNx(_DecimalNx x, _DecimalNx y);

bool samequantumdN(_DecimalN x, _DecimalN y);
bool samequantumdNx(_DecimalNx x, _DecimalNx y);

_DecimalN quantumdN(_DecimalN x);
_DecimalNx quantumdNx(_DecimalNx x);

long long int llquantexpdN(_DecimalN x);
long long int llquantexpdNx(_DecimalNx x);
```

7.12.16 Decimal re-encoding functions

```
void encodedecdN(unsigned char * restrict encptr,
      const _DecimalN * restrict xptr);
void decodedecdN(_DecimalN * restrict xptr,
      const unsigned char * restrict encptr);
void encodebindN(unsigned char * restrict encptr,
      const _DecimalN * restrict xptr);
void decodebindN(_DecimalN * restrict xptr,
      const unsigned char * restrict encptr);
```

F.10.12 Total order functions

```
int totalorderfN(const _FloatN *x, const _FloatN *y);
int totalorderfNx(const _FloatNx *x, const _FloatNx *y);
int totalorderdN(const _DecimalN *x, const _DecimalN *y);
int totalorderdNx(const _DecimalNx *x, const _DecimalNx *y);

int totalordermagfN(const _FloatN *x, const _FloatN *y);
int totalordermagfNx(const _FloatNx *x, const _FloatNx *y);
int totalordermagdN(const _DecimalN *x, const _DecimalN *y);
int totalordermagdNx(const _DecimalNx *x, const _DecimalNx *y);
```

F.10.13 Payload functions

```
_FloatN getpayloadfN(const _FloatN *x);
_FloatNx getpayloadfNx(const _FloatNx *x);
_DecimalN getpayloaddN(const _DecimalN *x);
_DecimalNx getpayloaddNx(const _DecimalNx *x);

int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfNx(_FloatNx *res, _FloatNx pl);
int setpayloaddN(_DecimalN *res, _DecimalN pl);
int setpayloaddNx(_DecimalNx *res, _DecimalNx pl);

int setpayloadsigfN(_FloatN *res, _FloatN pl);
int setpayloadsigfNx(_FloatNx *res, _FloatNx pl);
int setpayloadsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadsigdNx(_DecimalNx *res, _DecimalNx pl);
```

2 The specification of the **frexp** functions (7.12.6.7) applies to the functions for binary floating types like those for standard floating types: the exponent is an integral power of 2 and, when applicable, `value` equals $x \times 2^{*\texttt{exp}}$.

3 The specification of the **ldexp** functions (7.12.6.9) applies to the functions for binary floating types like those for standard floating types: they return $x \times 2^{\texttt{exp}}$.

4 The specification of the **logb** functions (7.12.6.17) applies to binary floating types, with $b = 2$.

5 The specification of the **scalbn** and **scalbln** functions (7.12.6.19) applies to binary floating types, with $b = 2$.

H.11.3 Encoding conversion functions

1 This subclause introduces `<math.h>` functions that, together with the numerical conversion functions for encodings in H.12, support the non-arithmetic interchange formats specified by IEC 60559. Support for these formats is an optional feature of this annex. Implementations that do not support non-arithmetic interchange formats need not declare the functions in this subclause.

2 Non-arithmetic interchange formats are not associated with floating types. Arrays of element type **unsigned char** are used as parameters for conversion functions, to represent encodings in interchange formats that might be non-arithmetic formats.

H.11.3.1 Encode and decode functions

1 This subclause specifies functions to map representations in binary floating types to and from encodings in **unsigned char** arrays.

H.11.3.1.1 The encodefN functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void encodefN(unsigned char encptr[restrict static N/8],
      const _FloatN * restrict xptr);
```

Description

2 The `encodefN` functions convert `*xptr` into an IEC 60559 binary$N$ encoding and store the resulting encoding as an $N/8$ element array, with 8 bits per array element, in the object pointed to by `encptr`. The order of bytes in the array is implementation-defined. These functions preserve the value of `*xptr` and raise no floating-point exceptions. If `*xptr` is non-canonical, these functions may or may not produce a canonical encoding.

Returns

3 The `encodefN` functions return no value.

H.11.3.1.2 The `decodefN` functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void decodefN(_FloatN * restrict xptr,
      const unsigned char encptr[restrict static N/8]);
```

Description

2 The `decodefN` functions interpret the $N/8$ element array pointed to by `encptr` as an IEC 60559 binary$N$ encoding, with 8 bits per array element. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a representation in the type `_FloatN`, and store the result in the object pointed to by `xptr`. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns

3 The `decodefN` functions return no value.

4 See **EXAMPLE** in H.11.3.2.1.

H.11.3.2 Encoding-to-encoding conversion functions

1 An implementation shall declare an `fMencfN` function for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format, $M \neq N$. An implementation shall provide both `dMencdecdN` and `dMencbindN`functions for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format, $M \neq N$.

H.11.3.2.1 The `fMencfN` functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void fMencfN(unsigned char encMptr[restrict static M/8],
      const unsigned char encNptr[restrict static N/8]);
```

Description

2 The `fMencfN` functions convert between IEC 60559 binary interchange formats. These functions interpret the $N/8$ element array pointed to by `encNptr` as an encoding of width N bits. They convert the encoding to an encoding of width M bits and store the resulting encoding as an $M/8$ element array in the object pointed to by `encMptr`. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

3 These functions return no value.

4 **EXAMPLE** If the IEC 60559 binary16 format is supported as a non-arithmetic format, data in binary16 format can be converted to type **float** as follows:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
unsigned char b16[2]; // for input binary16 datum
float f; // for result
unsigned char b32[4];
_Float32 f32;

// store input binary16 datum in array b16
...
f32encf16(b32, b16);
decodef32(&f32, b32);
f = f32;
...
```

H.11.3.2.2 The dMencdecdN and dMencbindN functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void dMencdecdN(unsigned char encMptr[restrict static M/8],
      const unsigned char encNptr[restrict static N/8]);
void dMencbindN(unsigned char encMptr[restrict static M/8],
      const unsigned char encNptr[restrict static N/8]);
```

Description

2 The dMencdecdN and dMencbindN functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The **d*M*encdecd*N*** functions convert between formats using the encoding scheme based on decimal encoding of the significand. The **d*M*encbind*N*** functions convert between formats using the encoding scheme based on binary encoding of the significand. These functions interpret the $N/8$ element array pointed to by enc*N*ptr as an encoding of width N bits. They convert the encoding to an encoding of width M bits and store the resulting encoding as an $M/8$ element array in the object pointed to by enc*M*ptr. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

3 These functions return no value.

H.12 Numeric conversion functions <stdlib.h>

1 This clause expands the specification of numeric conversion functions in <stdlib.h> (7.24.1) to also include conversions of strings from and to interchange and extended floating types. The conversions from floating are provided by functions analogous to the **strfromd** function. The conversions to floating are provided by functions analogous to the **strtod** function.

2 This clause also specifies functions to convert strings from and to IEC 60559 interchange format encodings.

3 For each interchange or extended floating type that the implementation provides, <stdlib.h> shall declare the associated functions specified below in H.12.1 and H.12.2 (see H.8). Conversely, for each such type that the implementation does not provide, <stdlib.h> shall not declare the associated functions.

4 For each IEC 60559 arithmetic or non-arithmetic format that the implementation supports, <stdlib.h> shall declare the associated functions specified below in H.12.3 and H.12.4 (see H.8). Conversely, for each such format that the implementation does not provide, <stdlib.h> shall not declare the associated functions.

H.12.1 String from floating

1 This subclause expands 7.24.1.3 and 7.24.1.4 to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.24.1.3 the prototypes

```
int strfromfN(char * restrict s, size_t n,
      const char * restrict format, _FloatN fp);
int strfromfNx(char * restrict s, size_t n,
      const char * restrict format, _FloatNx fp);
```

It encompasses the prototypes in 7.24.1.4 by replacing them with

```
int strfromdN(char * restrict s, size_t n,
      const char * restrict format, _DecimalN fp);
int strfromdNx(char * restrict s, size_t n,
      const char * restrict format, _DecimalNx fp);
```

2 The descriptions and returns for the added functions are analogous to the ones in 7.24.1.3 and 7.24.1.4.

H.12.2 String to floating

1 This subclause expands 7.24.1.5 and 7.24.1.6 to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.24.1.5 the prototypes

```
_FloatN strtofN(const char * restrict nptr,
      char ** restrict endptr);
_FloatNx strtofNx(const char * restrict nptr,
      char ** restrict endptr);
```

It encompasses the prototypes in 7.24.1.6 by replacing them with

```
_DecimalN strtodN(const char * restrict nptr,
      char ** restrict endptr);
_DecimalNx strtodNx(const char * restrict nptr,
      char ** restrict endptr);
```

2 The descriptions and returns for the added functions are analogous to the ones in 7.24.1.5 and 7.24.1.6.

3 For implementations that support both binary and decimal floating types and a (binary or decimal) non-arithmetic interchange format, the **strtod*N*** and **strtod*N*x** functions (and hence the **strtoencdecd*N*** and **strtoencbind*N*** functions in H.12.4.2) shall accept subject sequences that have the form of hexadecimal floating numbers (excluding any digit separators (6.4.4.1)) and otherwise meet the requirements of subject sequences (7.24.1.6). Then the decimal results shall be correctly rounded if the subject sequence has at most M significant hexadecimal digits, where $M \geq \lceil (P-1)/4 \rceil + 1$ is implementation-defined, and P is the maximum precision of the supported binary floating types and binary non-arithmetic formats. If all subject sequences of hexadecimal form are correctly rounded, M may be regarded as infinite. If the subject sequence has more than M significant hexadecimal digits, the implementation may first round to M significant hexadecimal digits according to the applicable rounding direction mode, signaling exceptions as though converting from a wider format, then correctly round the result of the shortened hexadecimal input to the result type.

4 **EXAMPLE** If the IEC 60559 binary128 format is supported as a non-arithmetic format, data in binary128 format can be converted to type **_Decimal128** as follows:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
#define MAXSIZE 41 // > intermediate hex string length
unsigned char b128[16]; // for input binary128 datum
```

```
_Decimal128 d128; // for result
char s[MAXSIZE];
// store input binary128 datum in array b128
...
strfromencf128(s, MAXSIZE, "%a", b128);
d128 = strtod128(s, NULL);
...
```

Use of `"%a"` for formatting assures an exact conversion of the value in binary format to character sequence. The value of that character sequence will be correctly rounded to `_Decimal128`, as specified above in this subclause. The array `s` for the output of `strfromencf128` need have no greater size than 41, which is the maximum length of strings of the form

$[-]0\mathrm{x}h.h\ldots h\mathrm{p}\pm d$

where there are up to 29 hexadecimal digits h and d has 5 digits plus 1 for the null character.

H.12.3 String from encoding

1 An implementation shall declare the **`strfromencfN`** function for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the **`strfromencdecdN`** and **`strfromencbindN`** functions for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

H.12.3.1 The `strfromencf` *N* functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>

int strfromencfN(char * restrict s, size_t n, const char * restrict format,
      const unsigned char encptr[restrict static N/8]);
```

Description

2 The **`strfromencfN`** functions are similar to the **`strfromfN`** functions, except the input is the value of the $N/8$ element array pointed to by `encptr`, interpreted as an IEC 60559 binary$N$ encoding. The order of bytes in the arrays is implementation-defined.

Returns

3 The **`strfromencfN`** functions return the same values as corresponding **`strfromfN`** functions.

H.12.3.2 The `strfromencdecdN` and `strfromencbindN` functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>

int strfromencdecdN(char * restrict s, size_t n, const char * restrict format,
      const unsigned char encptr[restrict static N/8]);
int strfromencbindN(char * restrict s, size_t n, const char * restrict format,
      const unsigned char encptr[restrict static N/8]);
```

Description

2 The **`strfromencdecdN`** functions are similar to the **`strfromdN`** functions except the input is the value of the $N/8$ element array pointed to by `encptr`, interpreted as an IEC 60559 decimal$N$ encoding in the coding scheme based on decimal encoding of the significand. The **`strfromencbindN`** functions are similar to the **`strfromdN`** functions except the input is the value of the $N/8$ element array pointed to by `encptr`, interpreted as an IEC 60559 decimal$N$ encoding in the coding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

3 The **strfromencdecd*N*** and **strfromencbind*N*** functions return the same values as corresponding **strfromd*N*** functions.

H.12.4 String to encoding

1 An implementation shall declare the **strtoencf*N*** function for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the **strtoencdecd*N*** and **strtoencbind*N*** functions for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

H.12.4.1 The strtoencf*N* functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>

void strtoencfN(unsigned char encptr[restrict static N/8],
      const char * restrict nptr, char ** restrict endptr);
```

Description

2 The **strtoencf*N*** functions are similar to the **strtof*N*** functions, except they store an IEC 60559 encoding of the result as an $N/8$ element array in the object pointed to by `encptr`. The order of bytes in the arrays is implementation-defined.

Returns

3 These functions return no value.

H.12.4.2 The strtoencdecd*N* and strtoencbind*N* functions

Synopsis

1
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>

void strtoencdecdN(unsigned char encptr[restrict static N/8],
      const char * restrict nptr, char ** restrict endptr);
void strtoencbindN(unsigned char encptr[restrict static N/8],
      const char * restrict nptr, char ** restrict endptr);
```

Description

2 The **strtoencdecd*N*** and **strtoencbind*N*** functions are similar to the **strtod*N*** functions, except they store an IEC 60559 encoding of the result as an $N/8$ element array in the object pointed to by `encptr`. The **strtoencdecd*N*** functions produce an encoding in the encoding scheme based on decimal encoding of the significand. The **strtoencbind*N*** functions produce an encoding in the encoding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

3 These functions return no value.

H.13 Type-generic macros <tgmath.h>

1 This clause enhances the specification of type-generic macros in <tgmath.h> (7.27) to apply to interchange and extended floating types, as well as standard floating types.

2 If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is a standard floating type or a binary floating type and another argument is of decimal floating type, the behavior is undefined.

3 The treatment of arguments of integer type in 7.27 is expanded to cases where another argument

has extended type. Arguments of integer type are regarded as having type:

— **_Decimal64x**, if any argument has a decimal extended type; otherwise

— **_Float32x**, if any argument has a binary extended type; otherwise

— **_Decimal64**, if any argument has decimal type; otherwise

— **double**

4 Use of the macros **carg**, **cimag**, **conj**, **cproj**, or **creal** with any argument of standard floating type, binary floating type, complex type, or imaginary type invokes a complex function. Use of the macro with an argument of a decimal floating type results in undefined behavior.

5 The functions that round results to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with `f` or `d` prefix are (as in 7.27):

fadd	**fmul**	**ffma**
dadd	**dmul**	**dfma**
fsub	**fdiv**	**fsqrt**
dsub	**ddiv**	**dsqrt**

and the macros with `f`*`M`*, `f`*`M`*`x`, `d`*`M`*, or `d`*`M`*`x` prefix are:

f*M*add	f*M*xmul	d*M***fma**
f*M*sub	f*M*xdiv	d*M***sqrt**
f*M*mul	f*M*xfma	d*M*xadd
f*M***div**	f*M*xsqrt	d*M*xsub
f*M***fma**	d*M*add	d*M*xmul
f*M***sqrt**	d*M*sub	d*M*xdiv
f*M*xadd	d*M*mul	d*M*xfma
f*M*xsub	d*M***div**	d*M*xsqrt

All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. The following specification uses the notation $type1 \subseteq type2$ to mean the values of $type1$ are a subset of (or the same as) the values of $type2$. The generic parameter type T for the function invoked by the macro is determined as follows:

— First, obtain a preliminary type P for the generic parameters: if all arguments are of integer type, then P is **double** if the macro prefix is `f`, `d`, `f`***`N`***, or `f`***`N`***`x` and P is **_Decimal64** if the macro prefix is `d`***`N`*** or `d`***`N`***`x`; otherwise (if some argument is not of integer type), apply the rules (for determining the corresponding real type of the generic parameters) in 7.27 for macros that do not round result to narrower type, using the usual arithmetic conversion rules in H.4.2, to obtain P.

— If there exists a corresponding function whose generic parameters have type P, then T is P.

— Otherwise, T is determined from P and the macro prefix as follows:

- For prefix `f`: if P is a standard or binary floating type, then T is the first standard floating type of either **double** or **long double**, such that $P \subseteq T$, if such a type T exists. Otherwise (if no such type T exists or P is a decimal floating type), the behavior is undefined.
- For prefix `d`: if P is a standard or binary floating type, then T is **long double** if $P \subseteq$ **long double**. Otherwise (if $P \subseteq$ **long double** is **false** or P is a decimal floating type), the behavior is undefined.

- For prefix `fM`: if P is a standard or binary floating type, then T is **`_FloatN`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is **`_FloatNx`** for minimum $N \geq M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (if no such **`_FloatN`** or **`_FloatNx`** is supported or P is a decimal floating type), the behavior is undefined.

- For prefix `fMx`: if P is a standard or binary floating type, then T is **`_FloatNx`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is **`_FloatN`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (if no such **`_FloatNx`** or **`_FloatN`** is supported or P is a decimal floating type), the behavior is undefined.

- For prefix `dM`: if P is a decimal floating type, then T is **`_DecimalN`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is **`_DecimalNx`** for minimum $N \geq M$ such that $P \subseteq T$. Otherwise (P is a standard or binary floating type), the behavior is undefined.

- For prefix `dMx`: if P is a decimal floating type, then T is **`_DecimalNx`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is **`_DecimalN`** for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (P is a standard or binary floating type), the behavior is undefined.

6 **EXAMPLE** With the declarations

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__

#include <tgmath.h>

int n;
double d;
long double ld;
double complex dc;
_Float32x f32x;
_Float64 f64;
_Float64x f64x;
_Float128 f128;
_Float64x complex f64xc;
```

functions invoked by use of type-generic macros are shown in the following table, where $type1 \subseteq type2$ means the values of $type1$ are a subset of (or the same as) the values of $type2$, and $type1 \subset type2$ means the values of $type1$ are a strict subset of the values of $type2$:

macro use	invokes
`cos(f64xc)`	`ccosf64x`
`pow(dc, f128)`	`cpowf128`
`pow(f64, d)`	`powf64`
`pow(d, f32x)`	`pow`, the function, if **`_Float32x`** $\subseteq$ **`double`**, else `powf32x` if **`double`** $\subset$ **`_Float32x`**, else undefined
`pow(f32, n)`	`pow`, the function
`pow(f32x, n)`	`pow32x`

Macros that round the result to a narrower type. . .

macro use	invokes
`fsub(d, ld)`	`fsubl`
`dsub(d, f32)`	`dsubl`
`fmul(dc, d)`	undefined
`ddiv(ld, f128)`	`ddivl` if `_Float128` $\subseteq$ `long double`, else undefined
`f32add(f64x, f64)`	`f32addf64x`
`f32xsqrt(n)`	`f32xsqrtf64`
`f32mul(f128, f32x)`	`f32mulf128` if `_Float32x` $\subseteq$ `_Float128`, else `f32mulf32x` if `_Float128` $\subset$ `_Float32x`, else undefined
`f32fma(f32x, n, f32x)`	`f32fmaf32x`
`f32add(f32, f32)`	`f32addf64`
`f32xsqrt(f32)`	`f32xsqrtf64x`, as declaration above shows `_Float64x` is supported
`f64div(f32x, f32x)`	`f64divf128` if `_Float32x` $\subseteq$ `_Float128`, else `f64divf64x`

Annex I
(informative)
Common warnings

1 An implementation may generate warnings in many situations, none of which are specified as part of this document. The following are a few of the more common situations.

2
- A new **`struct`** or **`union`** type appears in a function prototype (6.2.1, 6.7.2.3).
- A block with initialization of an object that has automatic storage duration is jumped into (6.2.4).
- An implicit narrowing conversion is encountered, such as the assignment of a **`long int`** or a **`double`** to an **`int`**, or a pointer to **`void`** to a pointer to any type other than a character type (6.3).
- A hexadecimal floating constant cannot be represented exactly in its evaluation format (6.4.4.2).
- An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (6.4.4.4).
- The characters `/*` are found in a comment (6.4.7).
- An "unordered" binary operator (not comma, &&, or `||`) contains a side effect to an lvalue in one operand, and a side effect to, or an access to the value of, the identical lvalue in the other operand (6.5).
- An object is defined but not used (6.7).
- A value is given to an object of an enumerated type other than by assignment of an enumeration constant that is a member of that type, or an enumeration object that has the same type, or the value of a function that returns the same enumerated type (6.7.2.2).
- An aggregate has a partly bracketed initialization (6.7.8).
- A statement cannot be reached (6.8).
- A statement with no apparent effect is encountered (6.8).
- A constant expression is used as the controlling expression of a selection statement (6.8.4).
- An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (6.10.1).
- An unrecognized **`#pragma`** directive is encountered (6.10.7).

Annex J
(informative)
Portability issues

1 This annex collects some information about portability that appears in this document.

J.1 Unspecified behavior

1 The following are unspecified:

(1) The manner and timing of static initialization (5.1.2).

(2) The termination status returned to the hosted environment if the return type of `main` is not compatible with `int` (5.1.2.2.3).

(3) The values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).

(4) The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).

(5) The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).

(6) The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

(7) The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

(8) How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).

(9) Many aspects of the representations of types (6.2.6).

(10) The value of padding bytes when storing values in structures or unions (6.2.6.1).

(11) The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).

(12) The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).

(13) The values of any padding bits in integer representations (6.2.6.2).

(14) Whether two string literals result in distinct arrays (6.4.5).

(15) The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, &&, `||`, `?:`, and comma operators (6.5).

(16) The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

(17) The order of side effects among compound literal initialization list expressions (6.5.2.5).

(18) The order in which the operands of an assignment operator are evaluated (6.5.16).

(19) The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

(20) Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

(21) Whether a size expression is evaluated when it is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator (6.7.6.2).

(22) The order in which any side effects occur among the initialization list expressions in an initializer (6.7.10).

(23) The layout of storage for function parameters (6.9.1).

(24) When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `(`, and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `(`, whether that is considered a nested replacement (6.10.4).

(25) The order in which **#** and **##** operations are evaluated during macro substitution (6.10.4.2, 6.10.4.3).

(26) The line number of a preprocessing token, in particular `__LINE__`, that spans multiple physical lines (6.10.5).

(27) The line number of a preprocessing directive that spans multiple physical lines (6.10.5).

(28) The line number of a macro invocation that spans multiple physical or logical lines (6.10.5).

(29) The line number following a directive of the form `#line __LINE__` *new-line* (6.10.5).

(30) The state of the floating-point status flags when execution passes from a part of the program translated with `FENV_ACCESS` "off" to a part translated with `FENV_ACCESS` "on" (7.6.1).

(31) The order in which `feraiseexcept` raises floating-point exceptions, except as stated in F.8.6 (7.6.4.3).

(32) Whether `math_errhandling` is a macro or an identifier with external linkage (7.12).

(33) The results of the `frexp` functions when the specified value is not a floating-point number (7.12.6.7).

(34) The numeric result of the `ilogb` functions when the correct value is outside the range of the return type (7.12.6.8, F.10.3.8).

(35) The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F.10.6.5).

(36) The value stored by the `remquo` functions in the object pointed to by `quo` when `y` is zero (7.12.10.3).

(37) Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.17).

(38) Whether `setjmp` is a macro or an identifier with external linkage (7.13).

(39) Whether `va_copy` and `va_end` are macros or identifiers with external linkage (7.16.1).

(40) The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an `a` or `A` conversion specifier (7.23.6.1, 7.31.2.1).

(41) The value of the file position indicator after a successful call to the `ungetc` function for a text stream, or the `ungetwc` function for any stream, until all pushed-back characters are read or discarded (7.23.7.10, 7.31.3.10).

(42) The details of the value stored by the `fgetpos` function (7.23.9.1).

(43) The details of the value returned by the `ftell` function for a text stream (7.23.9.4).

(44) Whether the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, and **wcstold** functions convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.24.1.5, 7.31.4.1.2).

(45) The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, **realloc**, and **aligned_alloc** functions (7.24.3).

(46) The amount of storage allocated by a successful call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function when 0 bytes was requested (7.24.3).

(47) Whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed (7.24.4.2).

(48) Whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed (7.24.4.3).

(49) Which of two elements that compare as equal is matched by the **bsearch** function (7.24.5.1).

(50) The order of two elements that compare as equal in an array sorted by the **qsort** function (7.24.5.2).

(51) The order in which destructors are invoked by **thrd_exit** (7.28.5.5).

(52) Whether calling **tss_delete** on a key while another thread is executing destructors affects the number of invocations of the destructors associated with the key on that thread (7.28.6.2).

(53) The encoding of the calendar time returned by the **time** function (7.29.2.5).

(54) The characters stored by the **strftime** or **wcsftime** function if any of the time values being converted is outside the normal range (7.29.3.5, 7.31.5.1).

(55) Whether an encoding error occurs if a **wchar_t** value that does not correspond to a member of the extended character set appears in the format string for a function in 7.31.2 or 7.31.5 and the specified semantics do not require that value to be processed by **wcrtomb** (7.31.1).

(56) The conversion state after an encoding error occurs (7.31.6.3.2, 7.31.6.3.3,c7.31.6.4.1, 7.31.6.4.2, 7.30.1.1, 7.30.1.2, 7.30.1.3, 7.30.1.4, 7.30.1.5, 7.30.1.6).

(57) The resulting value when the "invalid" floating-point exception is raised during IEC 60559 floating to integer conversion (F.4).

(58) Whether conversion of non-integer IEC 60559 floating values to integer raises the "inexact" floating-point exception (F.4).

(59) Whether or when library functions in `<math.h>` raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.10).

(60) Whether or when library functions in `<math.h>` raise an undeserved "underflow" floating-point exception in an IEC 60559 conformant implementation (F.10).

(61) The exponent value stored by **frexp** for a NaN or infinity (F.10.3.7).

(62) The numeric result returned by the **lrint**, **llrint**, **lround**, and **llround** functions if the rounded value is outside the range of the return type (F.10.6.5, F.10.6.7).

(63) The sign of one part of the **complex** result of several math functions for certain special cases in IEC 60559 compatible implementations (G.6.1.1, G.6.2.2, G.6.2.3, G.6.2.4, G.6.2.5, G.6.2.6, G.6.3.1, G.6.4.2).

J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

(1) A "shall" or "shall not" requirement that appears outside of a constraint is violated (Clause 4).

(2) A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

(3) Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

(4) A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).

(5) The execution of a program contains a data race (5.1.2.4).

(6) A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

(7) An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.1).

(8) The same identifier has both internal and external linkage in the same translation unit (6.2.2).

(9) An object is referred to outside of its lifetime (6.2.4).

(10) The value of a pointer to an object whose lifetime has ended is used (6.2.4).

(11) The value of an object with automatic storage duration is used while the object has an indeterminate representation (6.2.4, 6.7.10, 6.8).

(12) A non-value representation is read by an lvalue expression that does not have character type (6.2.6.1).

(13) A non-value representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

(14) Two declarations of the same object or function specify types that are not compatible (6.2.7).

(15) A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).

(16) Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

(17) Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

(18) An lvalue does not designate an object when evaluated (6.3.2.1).

(19) A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).

(20) An lvalue designating an object of automatic storage duration that could have been declared with the `register` storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

(21) An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

(22) An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to `void`) is applied to a void expression (6.3.2.2).

(23) Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

(24) Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

(25) A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

(26) An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).

(27) A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

(28) A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

(29) The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

(30) Two identifiers differ only in nonsignificant characters (6.4.2.1).

(31) The identifier `__func__` is explicitly declared (6.4.2.2).

(32) The program attempts to modify a string literal (6.4.5).

(33) The characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).

(34) A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

(35) An exceptional condition occurs during the evaluation of an expression (6.5).

(36) An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

(37) A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

(38) A member of an atomic structure or union is accessed (6.5.2.3).

(39) The operand of the unary `*` operator has an invalid value (6.5.3.2).

(40) A pointer is converted to other than an integer or pointer type (6.5.4).

(41) The value of the second operand of the `/` or `%` operator is zero (6.5.5).

(42) If the quotient `a/b` is not representable, the behavior of both `a/b` and `a%b` (6.5.5).

(43) Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

(44) Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).

(45) Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

(46) An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.5.6).

(47) The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).

(48) An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

(49) An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type (6.5.7).

(50) Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

(51) An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

(52) An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, predefined constants, **`sizeof`** expressions whose results are integer constants, **`alignof`** expressions, or immediately-cast floating constants; or contains casts (outside operands to **`sizeof`** and **`alignof`** operators) other than conversions of arithmetic types to integer types (6.6).

(53) A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

(54) An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, predefined constants, **`sizeof`** expressions whose results are integer constants, or **`alignof`** expressions; or contains casts (outside operands to **`sizeof`** or **`alignof`** operators) other than conversions of arithmetic types to arithmetic types (6.6).

(55) The value of an object is accessed by an array-subscript `[]`, member-access `.` or `->`, address `&`, or indirection `*` operator or a pointer cast in creating an address constant (6.6).

(56) An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

(57) A function is declared at block scope with an explicit storage-class specifier other than **`extern`** (6.7.1).

(58) A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

(59) An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

(60) When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

(61) An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

(62) An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

(63) The specification of a function type includes any type qualifiers (6.7.3).

(64) Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

(65) An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

(66) A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

(67) A function with external linkage is declared with an **`inline`** function specifier, but is not also defined in the same translation unit (6.7.4).

(68) A function declared with a **`_Noreturn`** function specifier returns to its caller (6.7.4).

(69) The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

(70) Declarations of an object in different translation units have different alignment specifiers (6.7.5).

(71) Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

(72) The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

(73) In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

(74) A declaration of an array parameter includes the keyword **`static`** within the `[` and `]` and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

(75) A storage-class specifier or type qualifier modifies the keyword **`void`** as a function parameter type list (6.7.6.3).

(76) In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list) (6.7.6.3).

(77) A declaration for which a type is inferred contains a pointer, array, or function declarators (6.7.9).

(78) A declaration for which a type is inferred contains no or more than one declarators (6.7.9).

(79) The value of an unnamed member of a structure or union is used (6.7.10).

(80) The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.10).

(81) The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.10).

(82) The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.10).

(83) A function definition that does not have the asserted property is called by a function declaration or a function pointer with a type that has the **`unsequenced`** or **`reproducible`** attribute (6.7.12.7).

(84) An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

(85) A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

(86) The `}` that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

(87) An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

(88) A non-directive preprocessing directive is executed (6.10).

(89) The token **`defined`** is generated during the expansion of a **`#if`** or **`#elif`** preprocessing directive, or the use of the **`defined`** unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

(90) The **`#include`** preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

(91) The character sequence in an **`#include`** preprocessing directive does not start with a letter (6.10.2).

(92) There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.4).

(93) The result of the preprocessing operator **`#`** is not a valid character string literal (6.10.4.2).

(94) The result of the preprocessing operator **`##`** is not a valid preprocessing token (6.10.4.3).

(95) The **`#line`** preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.5).

(96) A non-**`STDC #pragma`** preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.7).

(97) A **`#pragma STDC`** preprocessing directive does not match one of the well-defined forms (6.10.7).

(98) The name of a predefined macro, or the identifier **`defined`**, is the subject of a **`#define`** or **`#undef`** preprocessing directive (6.10.9).

(99) An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., **`memmove`**) (Clause 7).

(100) A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

(101) A header is included within an external declaration or definition (7.1.2).

(102) A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

(103) A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

(104) The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

(105) The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

(106) The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).

(107) An argument to a library function has an invalid value or a type not expected by a function with a variable number of arguments (7.1.4).

(108) The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

(109) The macro definition of **`assert`** is suppressed to access an actual function (7.2).

(110) The argument to the **`assert`** macro does not have a scalar type (7.2).

(111) The **CX_LIMITED_RANGE**, **FENV_ACCESS**, or **FP_CONTRACT** pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

(112) The value of an argument to a character handling function is neither equal to the value of **EOF** nor representable as an **unsigned char** (7.4).

(113) A macro definition of **errno** is suppressed to access an actual object, or the program defines an identifier with the name **errno** (7.5).

(114) Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma "off" (7.6.1).

(115) The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.4).

(116) The **fesetexceptflag** function is used to set floating-point status flags that were not specified in the call to the **fegetexceptflag** function that provided the value of the corresponding **fexcept_t** object (7.6.4.5).

(117) The argument to **fesetenv** or **feupdateenv** is neither an object set by a call to **fegetenv** or **feholdexcept**, nor is it an environment macro (7.6.6.3, 7.6.6.4).

(118) The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.24.6.1, 7.24.6.2, 7.24.1).

(119) The program modifies the string pointed to by the value returned by the **setlocale** function (7.11.1.1).

(120) A pointer returned by the **setlocale** function is used after a subsequent call to the function, or after the calling thread has exited (7.11.1.1).

(121) The program modifies the structure pointed to by the value returned by the **localeconv** function (7.11.2.1).

(122) A macro definition of **math_errhandling** is suppressed or the program defines an identifier with the name **math_errhandling** (7.12).

(123) An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.17).

(124) A macro definition of **setjmp** is suppressed to access an actual function, or the program defines an external identifier with the name **setjmp** (7.13).

(125) An invocation of the **setjmp** macro occurs other than in an allowed context (7.13.2.1).

(126) The **longjmp** function is invoked to restore a nonexistent environment (7.13.2.1).

(127) After a **longjmp**, there is an attempt to access the value of an object of automatic storage duration that does not have volatile-qualified type, local to the function containing the invocation of the corresponding **setjmp** macro, that was changed between the **setjmp** invocation and **longjmp** call (7.13.2.1).

(128) The program specifies an invalid pointer to a signal handler function (7.14.1.1).

(129) A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

(130) A signal handler called in response to **SIGFPE**, **SIGILL**, **SIGSEGV**, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

(131) A signal occurs as the result of calling the **abort** or **raise** function, and the signal handler calls the **raise** function (7.14.1.1).

(132) A signal occurs other than as the result of calling the **abort** or **raise** function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as **volatile sig_atomic_t**, or calls any function in the standard library other than the **abort** function, the **_Exit** function, the **quick_exit** function, the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free, the **atomic_is_lock_free** function with any atomic argument, or the **signal** function (for the same signal number) (7.14.1.1).

(133) The value of **errno** is referred to after a signal occurred other than as the result of calling the **abort** or **raise** function and the corresponding signal handler obtained a **SIG_ERR** return from a call to the **signal** function (7.14.1.1).

(134) A signal is generated by an asynchronous signal handler (7.14.1.1).

(135) The **signal** function is used in a multi-threaded program (7.14.1.1).

(136) A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized **va_list** object, or before the **va_start** macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

(137) The macro **va_arg** is invoked using the parameter `ap` that was passed to a function that invoked the macro **va_arg** with the same parameter (7.16).

(138) A macro definition of **va_start**, **va_arg**, **va_copy**, or **va_end** is suppressed to access an actual function, or the program defines an external identifier with the name **va_copy** or **va_end** (7.16.1).

(139) The **va_start** or **va_copy** macro is invoked without a corresponding invocation of the **va_end** macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

(140) The type parameter to the **va_arg** macro is not such that a pointer to an object of that type can be obtained simply by postfixing a * (7.16.1.1).

(141) The **va_arg** macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

(142) Using a null pointer constant in form of an integer expression as an argument to a `...` function and then interpreting it as a **void*** or **char*** (7.16.1.1).

(143) The **va_copy** or **va_start** macro is called to initialize a **va_list** that was previously initialized by either macro without an intervening invocation of the **va_end** macro for the same **va_list** (7.16.1.2, 7.16.1.4).

(144) The macro definition of a generic function is suppressed to access an actual function (7.17.1, 7.18).

(145) The *type* parameter of an **offsetof** macro defines a new type (7.21).

(146) When program execution reaches an **unreachable()** macro call (7.21.1).

(147) Arbitrarily copying or changing the bytes of or copying from a non-null pointer into a **nullptr_t** object and then reading that object (7.21.2).

(148) The *member-designator* parameter of an **offsetof** macro is an invalid right operand of the `.` operator for the *type* parameter, or designates a bit-field (7.21).

(149) The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.22.4).

(150) A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.23.2).

(151) Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.23.2).

(152) The value of a pointer to a `FILE` object is used after the associated file is closed (7.23.3).

(153) The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.23.5.2).

(154) The string pointed to by the `mode` argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.23.5.3).

(155) An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.23.5.3).

(156) An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.23.5.6).

(157) There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(158) The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.23.6.1, 7.23.6.2, 7.29.3.5, 7.31.2.1, 7.31.2.2, 7.31.5.1).

(159) In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.23.6.1, 7.31.2.1).

(160) A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.23.6.1, 7.31.2.1).

(161) A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.23.6.1, 7.31.2.1).

(162) A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(163) An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.23.6.1, 7.31.2.1).

(164) An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(165) A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%%` (7.23.6.1, 7.23.6.2, 7.31.2.1, 7.31.2.2).

(166) An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.23.6.1, 7.23.6.2, 7.29.3.5, 7.31.2.1, 7.31.2.2, 7.31.5.1).

(167) The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.23.6.1, 7.31.2.1).

(168) The number of input items assigned by a formatted input function is greater than `INT_MAX` (7.23.6.2, 7.31.2.2).

(169) The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.23.6.2, 7.31.2.2).

(170) A `c`, `s`, or `[` conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is `s` or `[`) (7.23.6.2, 7.31.2.2).

(171) A `c`, `s`, or `[` conversion specifier with an `l` qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.23.6.2, 7.31.2.2).

(172) The input item for a `%p` conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.23.6.2, 7.31.2.2).

(173) The **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, **vsscanf**, **vfwprintf**, **vfwscanf**, **vswprintf**, **vswscanf**, **vwprintf**, or **vwscanf** function is called with an improperly initialized **va_list** argument, or the argument is used (other than in an invocation of **va_end**) after the function returns (7.23.6.8, 7.23.6.9, 7.23.6.10, 7.23.6.11, 7.23.6.12, 7.23.6.13, 7.23.6.14, 7.31.2.5, 7.31.2.6, 7.31.2.7, 7.31.2.8, 7.31.2.9, 7.31.2.10).

(174) The contents of the array supplied in a call to the **fgets** or **fgetws** function are used after a read error occurred (7.23.7.2, 7.31.3.2).

(175) The file position indicator for a binary stream is used after a call to the **ungetc** function where its value was zero before the call (7.23.7.10).

(176) The file position indicator for a stream is used after an error occurred during a call to the **fread** or **fwrite** function (7.23.8.1, 7.23.8.2).

(177) A partial element read by a call to the **fread** function is used (7.23.8.1).

(178) The **fseek** function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the **ftell** function on a stream associated with the same file or `whence` is not **SEEK_SET** (7.23.9.2).

(179) The **fsetpos** function is called to set a position that was not returned by a previous successful call to the **fgetpos** function on a stream associated with the same file (7.23.9.3).

(180) A non-null pointer returned by a call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function with a zero requested size is used to access an object (7.24.3).

(181) The value of a pointer that refers to space deallocated by a call to the **free** or **realloc** function is used (7.24.3).

(182) The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to **free** or **realloc** (7.24.3.3, 7.24.3.7).

(183) The value of the object allocated by the **malloc** function is used (7.24.3.6).

(184) The values of any bytes in a new object allocated by the **realloc** function beyond the size of the old object are used (7.24.3.7).

(185) The program calls the **exit** or **quick_exit** function more than once, or calls both functions (7.24.4.4, 7.24.4.7).

(186) During the call to a function registered with the **atexit** or **at_quick_exit** function, a call is made to the **longjmp** function that would terminate the call to the registered function (7.24.4.4, 7.24.4.7).

(187) The string set up by the **getenv** or **strerror** function is modified by the program (7.24.4.6, 7.26.6.3).

(188) A signal is raised while the **quick_exit** function is executing (7.24.4.7).

(189) A command is executed through the **system** function in a way that is documented as causing termination or some other form of undefined behavior (7.24.4.8).

(190) A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.24.5).

(191) The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.24.5).

(192) The array being searched by the **bsearch** function does not have its elements in proper order (7.24.5.1).

(193) The current conversion state is used by a multibyte/wide character conversion function after changing the **LC_CTYPE** category (7.24.7).

(194) A string or wide string utility function is instructed to access an array beyond the end of an object (7.26.1, 7.31.4).

(195) A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.26.1, 7.31.4).

(196) The contents of the destination array are used after a call to the **strxfrm**, **strftime**, **wcsxfrm**, or **wcsftime** function in which the specified length was too small to hold the entire null-terminated result (7.26.4.5, 7.29.3.5, 7.31.4.4.4, 7.31.5.1).

(197) A sequence of calls of the **strtok** function is made from different threads (7.26.5.9).

(198) The first argument in the very first call to the **strtok** or **wcstok** is a null pointer (7.26.5.9, 7.31.4.6.7).

(199) A pointer returned by the **strerror** function is used after a subsequent call to the function, or after the calling thread has exited (7.26.6.3).

(200) The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.27).

(201) Arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type (7.27).

(202) Arguments for generic parameters of a type-generic macro are such that neither `<math.h>` and `<complex.h>` define a function whose generic parameters have the determined corresponding real type (7.27).

(203) A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.27).

(204) A decimal floating argument is supplied for a generic parameter of a type-generic macro that expects a complex argument (7.27).

(205) A standard floating or complex argument is supplied for a generic parameter of a type-generic macro that expects a decimal floating type argument (7.27).

(206) A non-recursive mutex passed to **mtx_lock** is locked by the calling thread (7.28.4.3).

(207) The mutex passed to **mtx_timedlock** does not support timeout (7.28.4.4).

(208) The mutex passed to **mtx_unlock** is not locked by the calling thread (7.28.4.6).

(209) The thread passed to **thrd_detach** or **thrd_join** was previously detached or joined with another thread (7.28.5.3, 7.28.5.6).

(210) The `tss_create` function is called from within a destructor (7.28.6.1).

(211) The key passed to `tss_delete`, `tss_get`, or `tss_set` was not returned by a call to `tss_create` before the thread commenced executing destructors (7.28.6.2, 7.28.6.3, 7.28.6.4).

(212) An attempt is made to access the pointer returned by the time conversion functions after the thread that originally called the function to obtain it has exited (7.29.3).

(213) At least one member of the broken-down time passed to `asctime` contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.29.3.1).

(214) The argument corresponding to an `s` specifier without an `l` qualifier in a call to the `fwprintf` function does not point to a valid multibyte character sequence that begins in the initial shift state (7.31.2.11).

(215) In a call to the `wcstok` function, the object pointed to by `ptr` does not have the value stored by the previous call for the same wide string (7.31.4.6.7).

(216) An `mbstate_t` object is used inappropriately (7.31.6).

(217) The value of an argument of type `wint_t` to a wide character classification or case mapping function is neither equal to the value of `WEOF` nor representable as a `wchar_t` (7.32.1).

(218) The `iswctype` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctype` function that returned the description (7.32.2.2.1).

(219) The `towctrans` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctrans` function that returned the description (7.32.3.2.1).

J.3 Implementation-defined behavior

1 A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

J.3.1 Translation

1 (1) How a diagnostic is identified (3.10, 5.1.1.3).

(2) Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

J.3.2 Environment

1 (1) The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

(2) The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

(3) The effect of program termination in a freestanding environment (5.1.2.1).

(4) An alternative manner in which the `main` function may be defined (5.1.2.2.1).

(5) The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

(6) What constitutes an interactive device (5.1.2.3).

(7) Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

(8) The set of signals, their semantics, and their default handling (7.14).

(9) Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

(10) Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

(11) The set of environment names and the method for altering the environment list used by the `getenv` function (7.24.4.6).

(12) The manner of execution of the string by the `system` function (7.24.4.8).

J.3.3 Identifiers

1 (1) Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

(2) The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

J.3.4 Characters

1 (1) The number of bits in a byte (3.6).

(2) The values of the members of the execution character set (5.2.1).

(3) The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

(4) The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

(5) Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

(6) The literal encoding, which maps of the characters of the execution character set to the values in a character constant or string literal (6.2.9, 6.4.4.4).

(7) The wide literal encoding, of the characters of the execution character set to the values in a `wchar_t` character constant or `wchar_t` string literal (6.2.9, 6.4.4.4).

(8) The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

(9) The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

(10) The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

(11) The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

(12) The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

(13) The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

(14) The encoding of any of `wchar_t`, `char16_t`, and `char32_t` where the corresponding standard encoding macro (`__STDC_ISO_10646__`, `__STDC_UTF_16__`, or `__STDC_UTF_32__`) is not defined (6.10.9.2).

J.3.5 Integers

1 (1) Any extended integer types that exist in the implementation (6.2.5).

(2) The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

(3) The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

(4) The results of some bitwise operations on signed integers (6.5).

J.3.6 Floating-point

1 (1) The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (5.2.4.2.2).

(2) The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in <stdio.h>, <stdlib.h>, and <wchar.h> (5.2.4.2.2).

(3) The rounding behaviors characterized by non-standard values of **FLT_ROUNDS** (5.2.4.2.2).

(4) The evaluation methods characterized by non-standard negative values of **FLT_EVAL_METHOD** (5.2.4.2.2).

(5) The evaluation methods characterized by non-standard negative values of **DEC_EVAL_METHOD** (5.2.4.2.3).

(6) If decimal floating types are supported (6.2.5).

(7) The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

(8) The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

(9) How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

(10) Whether and how floating expressions are contracted when not disallowed by the **FP_CONTRACT** pragma (6.5).

(11) The default state for the **FENV_ACCESS** pragma (7.6.1).

(12) Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

(13) The default state for the **FP_CONTRACT** pragma (7.12.2).

J.3.7 Arrays and pointers

1 (1) The result of converting a pointer to an integer or vice versa (6.3.2.3).

(2) The size of the result of subtracting two pointers to elements of the same array (6.5.6).

J.3.8 Hints

1 (1) The extent to which suggestions made by using the **register** storage-class specifier are effective (6.7.1).

(2) The extent to which suggestions made by using the **inline** function specifier are effective (6.7.4).

J.3.9 Structures, unions, enumerations, and bit-fields

1 (1) Whether a "plain" **`int`** bit-field is treated as a **`signed int`** bit-field or as an **`unsigned int`** bit-field (6.7.2, 6.7.2.1).

(2) Allowable bit-field types other than **`bool`**, **`signed int`**, **`unsigned int`**, and bit-precise integer types (6.7.2.1).

(3) Whether atomic types are permitted for bit-fields (6.7.2.1).

(4) Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

(5) The order of allocation of bit-fields within a unit (6.7.2.1).

(6) The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

(7) The integer type compatible with each enumerated type without fixed underlying type (6.7.2.2).

J.3.10 Qualifiers

1 (1) What constitutes an access to an object that has volatile-qualified type (6.7.3).

J.3.11 Preprocessing directives

1 (1) The locations within **`#pragma`** directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

(2) How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

(3) Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

(4) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

(5) The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

(6) How the named source file is searched for in an included `" "` delimited header (6.10.2).

(7) The method by which preprocessing tokens (possibly resulting from macro expansion) in a **`#include`** directive are combined into a header name (6.10.2).

(8) The nesting limit for **`#include`** processing (6.10.2).

(9) Whether the **`#`** operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.4.2).

(10) The behavior on each recognized non-**`STDC`** **`#pragma`** directive (6.10.7).

(11) The definitions for **`__DATE__`** and **`__TIME__`** when respectively, the date and time of translation are not available (6.10.9.1).

J.3.12 Library functions

1 (1) Any library facilities available to a freestanding program, other than the minimal set required by Clause 4 (5.1.2.1).

(2) The format of the diagnostic printed by the **`assert`** macro (7.2.1.1).

(3) The representation of the floating-point status flags stored by the **`fegetexceptflag`** function (7.6.4.2).

(4) Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.4.3).

(5) Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

(6) The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

(7) The types defined for `_Decimal32_t` and `_Decimal64_t` when the value of the `DEC_EVAL_METHOD` macro is less than 0 (7.12).

(8) Domain errors for the mathematics functions, other than those required by this document (7.12.1).

(9) The values returned by the mathematics functions on domain errors or pole errors (7.12.1).

(10) The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

(11) Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

(12) Whether a domain error occurs or zero is returned when a `remainder` function has a second argument of zero (7.12.10.2).

(13) The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

(14) The byte order of decimal floating type encodings (7.12.16).

(15) Whether a domain error occurs or zero is returned when a `remquo` function has a second argument of zero (7.12.10.3).

(16) Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

(17) The value of `__STDC_ENDIAN_NATIVE__` if the execution environment is not big-endian or little-endian (7.18.2)

(18) The null pointer constant to which the macro `NULL` expands (7.21).

(19) Whether the last line of a text stream requires a terminating new-line character (7.23.2).

(20) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.23.2).

(21) The number of null characters that may be appended to data written to a binary stream (7.23.2).

(22) Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.23.3).

(23) Whether a write on a text stream causes the associated file to be truncated beyond that point (7.23.3).

(24) The characteristics of file buffering (7.23.3).

(25) Whether a zero-length file actually exists (7.23.3).

(26) The rules for composing valid file names (7.23.3).

(27) Whether the same file can be simultaneously open multiple times (7.23.3).

(28) The nature and choice of encodings used for multibyte characters in files (7.23.3).

(29) The effect of the `remove` function on an open file (7.23.4.1).

(30) The effect if a file with the new name exists prior to a call to the `rename` function (7.23.4.2).

(31) Whether an open temporary file is removed upon abnormal program termination (7.23.4.3).

(32) Which changes of mode are permitted (if any), and under what circumstances (7.23.5.4).

(33) The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.23.6.1, 7.31.2.1).

(34) The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.23.6.1, 7.31.2.1).

(35) The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.23.6.2, 7.31.2.1).

(36) The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.23.6.2, 7.31.2.2).

(37) The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.23.9.1, 7.23.9.3, 7.23.9.4).

(38) The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.24.1.5, 7.31.4.1.2).

(39) Whether the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.24.1.5, 7.31.4.1.2).

(40) The meaning of any d-char or d-wchar sequence in a string representing a NaN that is converted by the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function (7.24.1.6, 7.31.4.1.3).

(41) Whether the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function sets `errno` to `ERANGE` when underflow occurs (7.24.1.6, 7.31.4.1.3).

(42) Whether the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.24.3).

(43) Whether open streams with unwritten buffered data are flushed,)open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.24.4.1, 7.24.4.5).

(44) The termination status returned to the host environment by the `abort`, `exit`, `_Exit`, or `quick_exit` function (7.24.4.1, 7.24.4.4, 7.24.4.5, 7.24.4.7).

(45) The value returned by the `system` function when its argument is not a null pointer (7.24.4.8).

(46) Whether the internal state of multibyte/wide character conversion functions has thread-storage duration, and its initial value in newly created threads (7.24.7).

(47) The range and precision of times representable in `clock_t` and `time_t` (7.29).

(48) The local time zone and Daylight Saving Time (7.29.1).

(49) Whether `TIME_MONOTONIC` or `TIME_ACTIVE` are supported time bases (7.29.1).

(50) Whether `TIME_THREAD_ACTIVE` is a supported time bases (7.29.1, 7.28.1).

(51) The era for the `clock` function (7.29.2.1).

(52) The `TIME_UTC` epoch (7.29.2.6).

(53) The replacement string for the `%Z` specifier to the **strftime**, and **wcsftime** functions in the `"C"` locale (7.29.3.5, 7.31.5.1).

(54) Whether internal **mbstate_t** objects have thread storage duration (7.30.1, 7.31.6.3, 7.31.6.4).

(55) Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

J.3.13 Architecture

1 (1) The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.22).

(2) The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

(3) The number, order, and encoding of bytes in any object (when not explicitly specified in this document) (6.2.6.1).

(4) Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

(5) Valid alignment values other than those returned by an **alignof** expression for fundamental types, if any (6.2.8).

(6) The value of the result of the **sizeof** and **alignof** operators (6.5.3.4).

J.4 Locale-specific behavior

1 The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

(1) Additional members of the source and execution character sets beyond the basic character set (5.2.1).

(2) The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.1).

(3) The shift states used for the encoding of multibyte characters (5.2.1.1).

(4) The direction of writing of successive printing characters (5.2.2).

(5) The decimal-point character (7.1.1).

(6) The set of printing characters (7.4, 7.32.2).

(7) The set of control characters (7.4, 7.32.2).

(8) The sets of characters tested for by the **isalpha**, **isblank**, **islower**, **ispunct**, **isspace**, **isupper**, **iswalpha**, **iswblank**, **iswlower**, **iswpunct**, **iswspace**, or **iswupper** functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.32.2.1.2, 7.32.2.1.3, 7.32.2.1.7, 7.32.2.1.9, 7.32.2.1.10, 7.32.2.1.11).

(9) The native environment (7.11.1.1).

(10) Additional subject sequences accepted by the numeric conversion functions (7.24.1, 7.31.4.1).

(11) The collation sequence of the execution character set (7.26.4.3, 7.31.4.4.2).

(12) The contents of the error message strings set up by the **strerror** function (7.26.6.3).

(13) The formats for time and date (7.29.3.5, 7.31.5.1).

(14) Character mappings that are supported by the **towctrans** function (7.32.1).

(15) Character classifications that are supported by the **iswctype** function (7.32.1).

J.5 Common extensions

1 The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, extra library functions declared in standard headers, or predefined macros with names that do not begin with an underscore.

J.5.1 Environment arguments

1 In a hosted environment, the `main` function receives a third argument, `char *envp[]`, that points to a null-terminated array of pointers to `char`, each of which points to a string that provides information about the environment for this execution of the program (5.1.2.2.1).

J.5.2 Specialized identifiers

1 Characters other than the underscore `_`, letters, and digits, that are not part of the basic source character set (such as the dollar sign `$`, or characters in national character sets) may appear in an identifier (6.4.2).

J.5.3 Lengths and cases of identifiers

1 All characters in identifiers (with or without external linkage) are significant (6.4.2).

J.5.4 Scopes of identifiers

1 A function identifier, or the identifier of an object the declaration of which contains the keyword `extern`, has file scope (6.2.1).

J.5.5 Writable string literals

1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

J.5.6 Other arithmetic types

1 Additional arithmetic types, such as `__int128` or `double double`, and their appropriate conversions are defined (6.2.5, 6.3.1). Additional floating types may have more range or precision than `long double`, may be used for evaluating expressions of other floating types, and may be used to define `float_t` or `double_t`. Additional floating types may also have less range or precision than `float`.

J.5.7 Function pointer casts

1 A pointer to an object or to `void` may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4).

2 A pointer to a function may be cast to a pointer to an object or to `void`, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).

J.5.8 Extended bit-field types

1 A bit-field may be declared with a type other than `bool`, `unsigned int`, `signed int`, or a bit-precise integer type, with an appropriate maximum width (6.7.2.1).

J.5.9 The `fortran` keyword

1 The `fortran` function specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that a different representation for the external name is to be generated (6.7.4).

J.5.10 The `asm` keyword

1 The `asm` keyword may be used to insert assembly language directly into the translator output (6.8). The most common implementation is via a statement of the form:

```
asm (character-string-literal);
```

J.5.11 Type inference

1 A declaration for which a type is inferred (6.7.9) may additionally accept pointer declarators, function declarators, and may have more than one declarator.

J.5.12 Multiple external definitions

1 There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern**; if the definitions disagree, or more than one is initialized, the behavior is undefined (6.9.2).

J.5.13 Predefined macro names

1 Macro names that do not begin with an underscore, describing the translation and execution environments, are defined by the implementation before translation begins (6.10.9).

J.5.14 Floating-point status flags

1 If any floating-point status flags are set on normal termination after all calls to functions registered by the **atexit** function have been made (see 7.24.4.4), the implementation writes some diagnostics indicating the fact to the **stderr** stream, if it is still open,

J.5.15 Extra arguments for signal handlers

1 Handlers for specific signals are called with extra arguments in addition to the signal number (7.14.1.1).

J.5.16 Additional stream types and file-opening modes

1 Additional mappings from files to streams are supported (7.23.2).

2 Additional file-opening modes may be specified by characters appended to the `mode` argument of the **fopen** function (7.23.5.3).

J.5.17 Defined file position indicator

1 The file position indicator is decremented by each successful call to the **ungetc** or **ungetwc** function for a text stream, except if its value was zero before a call (7.23.7.10, 7.31.3.10).

J.5.18 Math error reporting

1 Functions declared in `<complex.h>` and `<math.h>` raise **SIGFPE** to report errors instead of, or in addition to, setting **errno** or raising floating-point exceptions (7.3, 7.12).

J.6 Reserved identifiers and keywords

1 A lot of identifier preprocessing tokens are used for specific purposes in regular clauses or appendices from translation phase 3 onwards. Using any of these for a purpose different from their description in this document, even if the use is in a context where they are normatively permitted, may have an impact on the portability of code and should thus be avoided.

J.6.1 Rule based identifiers

1 The following 40 regular expressions characterize identifiers that are systematically reserved by some clause this document.

```
atomic_[a-z][a-zA-Z0-9_]*
ATOMIC_[A-Z][a-zA-Z0-9_]*
_[a-zA-Z_][a-zA-Z0-9_]*
cnd_[a-z][a-zA-Z0-9_]*
cr_[a-z][a-zA-Z0-9_]*
DBL_[A-Z][a-zA-Z0-9_]*
DEC128_[A-Z][a-zA-Z0-9_]*
DEC32_[A-Z][a-zA-Z0-9_]*
DEC64_[A-Z][a-zA-Z0-9_]*
DEC_[A-Z][a-zA-Z0-9_]*
E[0-9A-Z][a-zA-Z0-9_]*
FE_[A-Z][a-zA-Z0-9_]*
FLT_[A-Z][a-zA-Z0-9_]*
FP_[A-Z][a-zA-Z0-9_]*
INT[a-zA-Z0-9_]*_C
INT[a-zA-Z0-9_]*_MAX
INT[a-zA-Z0-9_]*_MIN
int[a-zA-Z0-9_]*_t
INT[a-zA-Z0-9_]*_WIDTH
is[a-z][a-zA-Z0-9_]*
```

```
LC_[A-Z][a-zA-Z0-9_]*
LDBL_[A-Z][a-zA-Z0-9_]*
MATH_[A-Z][a-zA-Z0-9_]*
mem[a-z][a-zA-Z0-9_]*
mtx_[a-z][a-zA-Z0-9_]*
PRI[a-zX][a-zA-Z0-9_]*
SCN[a-zX][a-zA-Z0-9_]*
SIG[A-Z][a-zA-Z0-9_]*
SIG_[A-Z][a-zA-Z0-9_]*
stdc_[a-zA-Z0-9_]*
str[a-z][a-zA-Z0-9_]*
thrd_[a-z][a-zA-Z0-9_]*
TIME_[A-Z][a-zA-Z0-9_]*
to[a-z][a-zA-Z0-9_]*
tss_[a-z][a-zA-Z0-9_]*
UINT[a-zA-Z0-9_]*_C
UINT[a-zA-Z0-9_]*_MAX
uint[a-zA-Z0-9_]*_t
UINT[a-zA-Z0-9_]*_WIDTH
wcs[a-z][a-zA-Z0-9_]*
```

2 The following 794 identifiers or keywords match these patterns and have particular semantics provided by this document.

```
_Alignas
__alignas_is_defined
_Alignof
__alignof_is_defined
_Atomic
atomic_bool
ATOMIC_BOOL_LOCK_FREE
atomic_char
atomic_char16_t
ATOMIC_CHAR16_T_LOCK_FREE
atomic_char32_t
ATOMIC_CHAR32_T_LOCK_FREE
atomic_char8_t
ATOMIC_CHAR8_T_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit
atomic_exchange
atomic_exchange_explicit
atomic_fetch_
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag
atomic_flag_clear
atomic_flag_clear_explicit
ATOMIC_FLAG_INIT
atomic_flag_test_and_set
atomic_flag_test_and_set_explicit
atomic_init
atomic_int
atomic_int_fast16_t
atomic_int_fast32_t
atomic_int_fast64_t
atomic_int_fast8_t
atomic_int_least16_t
atomic_int_least32_t
atomic_int_least64_t
atomic_int_least8_t
ATOMIC_INT_LOCK_FREE
atomic_intmax_t
atomic_intptr_t
atomic_is_lock_free
atomic_llong
ATOMIC_LLONG_LOCK_FREE
atomic_load
atomic_load_explicit
atomic_long
ATOMIC_LONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
atomic_ptrdiff_t
atomic_schar
atomic_short
ATOMIC_SHORT_LOCK_FREE
atomic_signal_fence
atomic_size_t
atomic_store
atomic_store_explicit
atomic_thread_fence
atomic_uchar
atomic_uint
atomic_uint_fast16_t
atomic_uint_fast32_t
atomic_uint_fast64_t
atomic_uint_fast8_t
atomic_uint_least16_t
atomic_uint_least32_t
atomic_uint_least64_t
atomic_uint_least8_t
atomic_uintmax_t
atomic_uintptr_t
atomic_ullong
atomic_ulong
atomic_ushort
ATOMIC_VAR_INIT
```

atomic_wchar_t
ATOMIC_WCHAR_T_LOCK_FREE
_BitInt
_Bool
__bool_true_false_are_defined
cnd_broadcast
cnd_destroy
cnd_init
cnd_signal
cnd_t
cnd_timedwait
cnd_wait
_Complex
_Complex_I
__cplusplus
__DATE__
DBL_DECIMAL_DIG
DBL_DIG
DBL_EPSILON
DBL_HAS_SUBNORM
DBL_IS_IEC_60559
DBL_MANT_DIG
DBL_MAX
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN
DBL_MIN_10_EXP
DBL_MIN_EXP
DBL_NORM_MAX
DBL_SNAN
DBL_TRUE_MIN
DEC128_EPSILON
DEC128_MANT_DIG
DEC128_MAX
DEC128_MAX_EXP
DEC128_MIN
DEC128_MIN_EXP
DEC128_SNAN
DEC128_TRUE_MIN
DEC32_EPSILON
DEC32_MANT_DIG
DEC32_MAX
DEC32_MAX_EXP
DEC32_MIN
DEC32_MIN_EXP
DEC32_SNAN
DEC32_TRUE_MIN
DEC64_EPSILON
DEC64_MANT_DIG
DEC64_MAX
DEC64_MAX_EXP
DEC64_MIN
DEC64_MIN_EXP
DEC64_SNAN
DEC64_TRUE_MIN
DEC_EVAL_METHOD
_Decimal
_Decimal128
_Decimal128x
_Decimal32
_Decimal32_t
_Decimal64
_Decimal64_t
_Decimal64x
_DECIMAL_DIG
DEC_INFINITY
DEC_NAN
__deprecated__
_DIG
EDOM
EILSEQ
EOF
EOL
_EPSILON
ERANGE
_Exit
EXIT_FAILURE
EXIT_SUCCESS
_EXT__
__fallthrough__
FE_ALL_EXCEPT
FE_DEC_DOWNWARD
FE_DEC_DYNAMIC
FE_DEC_TONEAREST
FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO
FE_DEC_UPWARD
FE_DFL_ENV
FE_DFL_MODE
FE_DIVBYZERO
FE_DOWNWARD
FE_DYNAMIC
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_SNANS_ALWAYS_SIGNAL
FE_TONEAREST
FE_TONEARESTFROMZERO
FE_TOWARDZERO
FE_UNDERFLOW
FE_UPWARD
__FILE__
_Float
_Float128
_Float128_t
_Float128x
_Float16
_Float16_t
_Float32
_Float32_t
_Float32x
_Float64

mtx_recursive
mtx_t
mtx_timed
mtx_timedlock
mtx_trylock
mtx_unlock
__nodiscard__
___Noreturn__
__noreturn__
_Noreturn
__pp_param__
_Pragma
__prefix__
PRId32
PRId64
PRIdFAST32
PRIdFAST64
PRIdLEAST32
PRIdLEAST64
PRIdMAX
PRIdPTR
PRIi32
PRIi64
PRIiFAST32
PRIiFAST64
PRIiLEAST32
PRIiLEAST64
PRIiMAX
PRIiPTR
_PRINTF_NAN_LEN_MAX
PRIo32
PRIo64
PRIoFAST32
PRIoFAST64
PRIoLEAST32
PRIoLEAST64
PRIoMAX
PRIoPTR
PRIu32
PRIu64
PRIuFAST32
PRIuFAST64
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIuPTR
PRIX32
PRIX64
PRIXFAST32
PRIXFAST64
PRIXLEAST32
PRIXLEAST64
PRIXMAX
PRIXPTR
__reproducible__
SCNdMAX
SCNdPTR
SCNiMAX
SCNiPTR
SCNoMAX
SCNoPTR
SCNuMAX
SCNuPTR
SCNxMAX
SCNxPTR
SIGABRT
SIG_ATOMIC_MAX
SIG_ATOMIC_MIN
SIG_ATOMIC_WIDTH
SIG_DFL
SIG_ERR
SIGFPE
SIG_IGN
SIGILL
SIGINT
SIGSEGV
SIGTERM
_SNAN
_Static_assert
__STDC__
__STDC_ANALYZABLE__
stdc_bit_ceil
stdc_bit_ceiluc
stdc_bit_ceilui
stdc_bit_ceilul
stdc_bit_ceilull
stdc_bit_ceilus
stdc_bit_floor
stdc_bit_flooruc
stdc_bit_floorui
stdc_bit_floorul
stdc_bit_floorull
stdc_bit_floorus
stdc_bit_width
stdc_bit_widthuc
stdc_bit_widthui
stdc_bit_widthul
stdc_bit_widthull
stdc_bit_widthus
stdc_count_ones
stdc_count_onesuc
stdc_count_onesui
stdc_count_onesul
stdc_count_onesull
stdc_count_onesus
stdc_count_zeros
stdc_count_zerosuc
stdc_count_zerosui
stdc_count_zerosul
stdc_count_zerosull
stdc_count_zerosus
__STDC_ENDIAN_BIG__

strndup
strnlen_s
strpbrk
strrchr
strspn
strstr
strto
strtod
strtod128
strtod32
strtod64
strtoencbind
strtoencdecd
strtoencf
strtof
strtoimax
strtok
strtok_s
strtol
strtold
strtoll
strtoul
strtoull
strtoumax
struct
strxfrm
__suffix__
thrd_busy
thrd_create
thrd_current
thrd_detach
thrd_equal
thrd_error
thrd_exit
thrd_join
thrd_nomem
thrd_sleep
thrd_start_t
thrd_success
thrd_t
thrd_timedout
thrd_yield
_Thread_local
__TIME__
TIME_ACTIVE
TIME_MONOTONIC
TIME_THREAD_ACTIVE
TIME_UTC
tolower
totalorder
totalorderd
totalorderd128
totalorderd32
totalorderd64
totalorderf
totalorderl
totalordermag
totalordermagd
totalordermagd128
totalordermagd32
totalordermagd64
totalordermagf
totalordermagl
toupper
towctrans
towlower
towupper
_TRUE_MIN
tss_create
tss_delete
tss_dtor_t
tss_get
tss_set
tss_t
UINT16_C
UINT16_MAX
uint16_t
UINT16_WIDTH
UINT32_C
UINT32_MAX
uint32_t
UINT32_WIDTH
UINT64_C
UINT64_MAX
uint64_t
UINT64_WIDTH
UINT8_C
UINT8_MAX
uint8_t
UINT8_WIDTH
uint_fast16_t
uint_fast32_t
uint_fast64_t
uint_fast8_t
uint_least16_t
uint_least32_t
uint_least64_t
uint_least8_t
UINT_MAX
UINTMAX_C
UINTMAX_MAX
uintmax_t
UINTMAX_WIDTH
UINTPTR_MAX
uintptr_t
UINTPTR_WIDTH
UINT_WIDTH
__unsequenced__
__VA_ARGS__
__VA_OPT__
wcscat
wcscat_s

wcschr
wcscmp
wcscoll
wcscpy
wcscpy_s
wcscspn
wcsftime
wcslen
wcsncat
wcsncat_s
wcsncmp
wcsncpy
wcsncpy_s
wcsnlen_s
wcspbrk
wcsrchr
wcsrtombs
wcsrtombs_s
wcsspn
wcsstr
wcsto
wcstod
wcstod128
wcstod32
wcstod64
wcstof
wcstoimax
wcstok
wcstok_s
wcstol
wcstold
wcstoll
wcstombs
wcstombs_s
wcstoul
wcstoull
wcstoumax
wcsxfrm

J.6.2 Particular identifiers or keywords

1 The following 1358 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

abort
abort_handler_s
abs
acos
acosd
acosd128
acosd32
acosd64
acosf
acosh
acoshd
acoshd128
acoshd32
acoshd64
acoshf
acoshl
acosl
acospi
acospid
acospid128
acospid32
acospid64
acospif
acospil
addd
addf
alignas
aligned_alloc
alignof
and
and_eq
asctime
asctime_s
asin
asind
asind128
asind32
asind64
asinf
asinh
asinhd
asinhd128
asinhd32
asinhd64
asinhf
asinhl
asinl
asinpi
asinpid
asinpid128
asinpid32
asinpid64
asinpif
asinpil
assert
atan
atan2
atan2d
atan2d128
atan2d32
atan2d64
atan2f
atan2l
atan2pi
atan2pid
atan2pid128
atan2pid32
atan2pid64
atan2pif
atan2pil
atand
atand128
atand32
atand64
atanf
atanh
atanhd
atanhd128
atanhd32
atanhd64
atanhf
atanhl
atanl
atanpi
atanpid
atanpid128
atanpid32
atanpid64
atanpif
atanpil
atexit
atof
atoi
atol
atoll
at_quick_exit

```
auto
bitand
BITINT_MAXWIDTH
bitor
bool
BOOL_MAX
BOOL_WIDTH
break
bsearch
bsearch_s
btowc
BUFSIZ
c16rtomb
c32rtomb
c8rtomb
cabs
cabsf
cabsl
cacos
cacosf
cacosh
cacoshf
cacoshl
cacosl
cacospi
calloc
call_once
canonicalize
canonicalized
canonicalized128
canonicalized32
canonicalized64
canonicalizef
canonicalizel
carg
cargf
cargl
case
casin
casinf
casinh
casinhf
casinhl
casinl
casinpi
catan
catanf
catanh
catanhf
catanhl
catanl
catanpi
cbrt
cbrtd
cbrtd128
cbrtd32
cbrtd64
cbrtf
cbrtl
ccompoundn
ccos
ccosf
ccosh
ccoshf
ccoshl
ccosl
ccospi
ceil
ceild
ceild128
ceild32
ceild64
ceilf
ceill
cerf
cerfc
cexp
cexp10
cexp10m1
cexp2
cexp2m1
cexpf
cexpl
cexpm1
char
char16_t
char32_t
char8_t
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHAR_WIDTH
cimag
cimagf
cimagl
ckd_
ckd_add
ckd_div
ckd_mul
ckd_sub
clearerr
clgamma
clock
CLOCKS_PER_SEC
clock_t
clog
clog10
clog10p1
clog1p
clog2
clog2p1
clogf
clogl
clogp1
CMPLX
CMPLXF
CMPLXL
compl
complex
compoundn
compoundnd
compoundnd128
compoundnd32
compoundnd64
compoundnf
compoundnl
conj
conjf
conjl
const
constexpr
constraint_handler_t
continue
copysign
copysignd
copysignd128
copysignd32
copysignd64
copysignf
copysignl
cos
cosd
cosd128
cosd32
cosd64
cosf
cosh
coshd
coshd128
coshd32
coshd64
coshf
coshl
cosl
cospi
cospid
cospid128
cospid32
cospid64
cospif
cospil
cpow
cpowf
cpowl
cpown
cpowr
cproj
cprojf
```

cprojl
CR_DECIMAL_DIG
creal
crealf
creall
csin
csinf
csinh
csinhf
csinhl
csinl
csinpi
csqrt
csqrtf
csqrtl
ctan
ctanf
ctanh
ctanhf
ctanhl
ctanl
ctanpi
ctgamma
ctime
ctime_s
currency_symbol
CX_LIMITED_RANGE
d32add
d32addd128
d32addd64
d32div
d32divd128
d32divd64
d32fma
d32fmad128
d32fmad64
d32mul
d32muld128
d32muld64
d32sqrt
d32sqrtd128
d32sqrtd64
d32sub
d32subd128
d32subd64
d64add
d64addd128
d64div
d64divd128
d64fma
d64fmad128
d64mul
d64muld128
d64sqrt
d64sqrtd128
d64sub
d64subd128
dadd
daddl
ddiv
ddivl
DEC
Decimal
DECIMAL_DIG
decimal_point
DECN_
DECN
decodebin
decodebind
decodebind128
decodebind32
decodebind64
decodedec
decodedecd
decodedecd128
decodedecd32
decodedecd64
decodef
DEFAULT
define
defined
deprecated
dfma
dfmal
difftime
div
divd
divf
div_t
dmul
dmull
do
double
double_t
dsqrt
dsqrtl
dsub
dsubl
elif
elifdef
elifndef
else
embed
encbind
encdecd
encf
encodebin
encodebind
encodebind128
encodebind32
encodebind64
encodedec
encodedecd
encodedecd128
encodedecd32
encodedecd64
encodef
endif
enum
erf
erfc
erfcd
erfcd128
erfcd32
erfcd64
erfcf
erfcl
erfd
erfd128
erfd32
erfd64
erff
erfl
errno
errno_t
error
exit
exp
exp10
exp10d
exp10d128
exp10d32
exp10d64
exp10f
exp10l
exp10m1
exp10m1d
exp10m1d128
exp10m1d32
exp10m1d64
exp10m1f
exp10m1l
exp2
exp2d
exp2d128
exp2d32
exp2d64
exp2f
exp2l
exp2m1
exp2m1d
exp2m1d128
exp2m1d32
exp2m1d64
exp2m1f
exp2m1l
expd
expd128

expd32
expd64
expf
expl
expm1
expm1d
expm1d128
expm1d32
expm1d64
expm1f
expm1l
extern
fabs
fabsd
fabsd128
fabsd32
fabsd64
fabsf
fabsl
fadd
faddl
fallthrough
false
fclose
fdim
fdimd
fdimd128
fdimd32
fdimd64
fdimf
fdiml
fdiv
fdivl
feclearexcept
fe_dec_getround
fe_dec_setround
fegetenv
fegetexceptflag
fegetmode
fegetround
feholdexcept
femode_t
FENV_ACCESS
FENV_DEC_ROUND
FENV_ROUND
fenv_t
feof
feraiseexcept
ferror
fesetenv
fesetexcept
fesetexceptflag
fesetmode
fesetround
fetestexcept
fetestexceptflag
feupdateenv
fexcept_t
fflush
ffma
ffmal
fgetc
fgetpos
fgets
fgetwc
fgetws
FILE
FILENAME_MAX
Float
float_t
floor
floord
floord128
floord32
floord64
floorf
floorl
FLT
FLTN_
FLTN
fma
fmad
fmad128
fmad32
fmad64
fmaf
fmal
fmax
fmaxd
fmaxd128
fmaxd32
fmaxd64
fmaxf
fmaximum
fmaximumd
fmaximumd128
fmaximumd32
fmaximumd64
fmaximumf
fmaximuml
fmaximum_mag
fmaximum_magd
fmaximum_magd128
fmaximum_magd32
fmaximum_magd64
fmaximum_magf
fmaximum_magl
fmaximum_mag_num
fmaximum_mag_numd
fmaximum_mag_numd128
fmaximum_mag_numd32
fmaximum_mag_numd64
fmaximum_mag_numf
fmaximum_mag_numl
fmaximum_num
fmaximum_numd
fmaximum_numd128
fmaximum_numd32
fmaximum_numd64
fmaximum_numf
fmaximum_numl
fmaxl
fmin
fmind
fmind128
fmind32
fmind64
fminf
fminimum
fminimumd
fminimumd128
fminimumd32
fminimumd64
fminimumf
fminimuml
fminimum_mag
fminimum_magd
fminimum_magd128
fminimum_magd32
fminimum_magd64
fminimum_magf
fminimum_magl
fminimum_mag_num
fminimum_mag_numd
fminimum_mag_numd128
fminimum_mag_numd32
fminimum_mag_numd64
fminimum_mag_numf
fminimum_mag_numl
fminimum_num
fminimum_numd
fminimum_numd128
fminimum_numd32
fminimum_numd64
fminimum_numf
fminimum_numl
fminl
fmod
fmodd
fmodd128
fmodd32
fmodd64
fmodf
fmodl
fmul
fmull
fopen
FOPEN_MAX

fopen_s
for
fpclassify
fpos_t
fprintf
fprintf_s
fputc
fputs
fputwc
fputws
frac_digits
fread
free
free_aligned_sized
free_sized
freopen
freopen_s
frexp
frexpd
frexpd128
frexpd32
frexpd64
frexpf
frexpl
fromfp
fromfpd
fromfpd128
fromfpd32
fromfpd64
fromfpf
fromfpl
fromfpx
fromfpxd
fromfpxd128
fromfpxd32
fromfpxd64
fromfpxf
fromfpxl
fscanf
fscanf_s
fseek
fsetpos
fsqrt
fsqrtl
fsub
fsubl
ftell
fwide
fwprintf
fwprintf_s
fwrite
fwscanf
fwscanf_s
generic_count_type
generic_return_type
generic_value_type
getc
getchar
getenv
getenv_s
getpayload
getpayloadd
getpayloadd128
getpayloadd32
getpayloadd64
getpayloadf
getpayloadl
gets
gets_s
getwc
getwchar
gmtime
gmtime_r
gmtime_s
goto
grouping
HUGE_VAL
HUGE_VAL_D
HUGE_VAL_D128
HUGE_VAL_D32
HUGE_VAL_D64
HUGE_VAL_F
HUGE_VALF
HUGE_VALL
hypot
hypotd
hypotd128
hypotd32
hypotd64
hypotf
hypotl
I
if
ifdef
if_empty
ifndef
ignore_handler_s
ilogb
ilogbd
ilogbd128
ilogbd32
ilogbd64
ilogbf
ilogbl
imaginary
imaxabs
imaxdiv
imaxdiv_t
include
INFINITY
inline
int_curr_symbol
int_frac_digits
int_n_cs_precedes
int_n_sep_by_space
int_n_sign_posn
int_p_cs_precedes
int_p_sep_by_space
int_p_sign_posn
jmp_buf
kill_dependency
labs
lconv
ldexp
ldexpd
ldexpd128
ldexpd32
ldexpd64
ldexpf
ldexpl
ldiv
ldiv_t
lgamma
lgammad
lgammad128
lgammad32
lgammad64
lgammaf
lgammal
limit
line
llabs
lldiv
lldiv_t
llogb
llogbd
llogbd128
llogbd32
llogbd64
llogbf
llogbl
LLONG_MAX
LLONG_MIN
LLONG_WIDTH
llquantexp
llquantexpd
llquantexpd128
llquantexpd32
llquantexpd64
llrint
llrintd
llrintd128
llrintd32
llrintd64
llrintf
llrintl
llround
llroundd

llroundd128
llroundd32
llroundd64
llroundf
llroundl
localeconv
localtime
localtime_r
localtime_s
log
log10
log10d
log10d128
log10d32
log10d64
log10f
log10l
log10p1
log10p1d
log10p1d128
log10p1d32
log10p1d64
log10p1f
log10p1l
log1p
log1pd
log1pd128
log1pd32
log1pd64
log1pf
log1pl
log2
log2d
log2d128
log2d32
log2d64
log2f
log2l
log2p1
log2p1d
log2p1d128
log2p1d32
log2p1d64
log2p1f
log2p1l
logb
logbd
logbd128
logbd32
logbd64
logbf
logbl
logd
logd128
logd32
logd64
logf
logl
logp1
logp1d
logp1d128
logp1d32
logp1d64
logp1f
logp1l
long
long_double_t
longjmp
LONG_MAX
LONG_MIN
LONG_WIDTH
lrint
lrintd
lrintd128
lrintd32
lrintd64
lrintf
lrintl
lround
lroundd
lroundd128
lroundd32
lroundd64
lroundf
lroundl
L_tmpnam
L_tmpnam_s
main
malloc
math_errhandling
max_align_t
maybe_unused
MB_CUR_MAX
mblen
MB_LEN_MAX
mbrlen
mbrtoc16
mbrtoc32
mbrtoc8
mbrtowc
mbsinit
mbsrtowcs
mbsrtowcs_s
mbstate_t
mbstowcs
mbstowcs_s
mbtowc
mktime
modf
modfd
modfd128
modfd32
modfd64
modff
modfl
mon_decimal_point
mon_grouping
mon_thousands_sep
muld
mulf
N
nan
nand
nand128
nand32
nand64
nanf
nanl
n_cs_precedes
NDEBUG
nearbyint
nearbyintd
nearbyintd128
nearbyintd32
nearbyintd64
nearbyintf
nearbyintl
negative_sign
nextafter
nextafterd
nextafterd128
nextafterd32
nextafterd64
nextafterf
nextafterl
nextdown
nextdownd
nextdownd128
nextdownd32
nextdownd64
nextdownf
nextdownl
nexttoward
nexttowardd128
nexttowardd32
nexttowardd64
nexttowardf
nexttowardl
nextup
nextupd
nextupd128
nextupd32
nextupd64
nextupf
nextupl
nodiscard
noreturn
not

not_eq
n_sep_by_space
n_sign_posn
NULL
nullptr
nullptr_t
OFF
offsetof
ON
once_flag
ONCE_FLAG_INIT
or
or_eq
p_cs_precedes
perror
positive_sign
pow
powd
powd128
powd32
powd64
powf
powl
pown
pownd
pownd128
pownd32
pownd64
pownf
pownl
powr
powrd
powrd128
powrd32
powrd64
powrf
powrl
pp_param
pragma
prefix
printf
printf_s
p_sep_by_space
p_sign_posn
PTRDIFF_MAX
PTRDIFF_MIN
ptrdiff_t
PTRDIFF_WIDTH
putc
putchar
puts
putwc
putwchar
QChar
qsort
qsort_s
quantize
quantized
quantized128
quantized32
quantized64
quantum
quantumd
quantumd128
quantumd32
quantumd64
quick_exit
QVoid
QWchar_t
raise
rand
RAND_MAX
realloc
register
remainder
remainderd
remainderd128
remainderd32
remainderd64
remainderf
remainderl
remove
remquo
remquof
remquol
rename
reproducible
restrict
return
rewind
rint
rintd
rintd128
rintd32
rintd64
rintf
rintl
rootn
rootnd
rootnd128
rootnd32
rootnd64
rootnf
rootnl
round
roundd
roundd128
roundd32
roundd64
roundeven
roundevend
roundevend128
roundevend32
roundevend64
roundevenf
roundevenl
roundf
roundl
RSIZE_MAX
rsize_t
rsqrt
rsqrtd
rsqrtd128
rsqrtd32
rsqrtd64
rsqrtf
rsqrtl
samequantum
samequantumd
samequantumd128
samequantumd32
samequantumd64
scalbln
scalblnd
scalblnd128
scalblnd32
scalblnd64
scalblnf
scalblnl
scalbn
scalbnd
scalbnd128
scalbnd32
scalbnd64
scalbnf
scalbnl
scanf
scanf_s
SCHAR_MAX
SCHAR_MIN
SCHAR_WIDTH
SEEK_CUR
SEEK_END
SEEK_SET
setbuf
set_constraint_handler_s
setjmp
setlocale
setpayload
setpayloadd
setpayloadd128
setpayloadd32
setpayloadd64
setpayloadf
setpayloadl
setpayloadsig
setpayloadsigd
setpayloadsigd128

setpayloadsigd32
setpayloadsigd64
setpayloadsigf
setpayloadsigl
setvbuf
short
SHRT_MAX
SHRT_MIN
SHRT_WIDTH
sig_atomic_t
signal
signbit
signed
sin
sind
sind128
sind32
sind64
sinf
sinh
sinhd
sinhd128
sinhd32
sinhd64
sinhf
sinhl
sinl
sinpi
sinpid
sinpid128
sinpid32
sinpid64
sinpif
sinpil
SIZE_MAX
sizeof
size_t
SIZE_WIDTH
snprintf
snprintf_s
snwprintf_s
sprintf
sprintf_s
sqrt
sqrtd
sqrtd128
sqrtd32
sqrtd64
sqrtf
sqrtl
srand
sscanf
sscanf_s
static
static_assert
STDC
stderr
stdin
stdout
subd
subf
suffix
switch
swprintf
swprintf_s
swscanf
swscanf_s
system
tan
tand
tand128
tand32
tand64
tanf
tanh
tanhd
tanhd128
tanhd32
tanhd64
tanhf
tanhl
tanl
tanpi
tanpid
tanpid128
tanpid32
tanpid64
tanpif
tanpil
tgamma
tgammad
tgammad128
tgammad32
tgammad64
tgammaf
tgammal
thousands_sep
thread_local
time
timespec
timespec_get
timespec_getres
time_t
tm
tm_hour
tm_isdst
tm_mday
tm_min
tm_mon
tmpfile
tmpfile_s
TMP_MAX
TMP_MAX_S
tmpnam
tmpnam_s
tm_sec
tm_wday
tm_yday
tm_year
true
trunc
truncd
truncd128
truncd32
truncd64
truncf
truncl
TSS_DTOR_ITERATIONS
tv_nsec
tv_sec
typedef
typeof
typeof_unqual
UCHAR_MAX
UCHAR_WIDTH
ufromfp
ufromfpd
ufromfpd128
ufromfpd32
ufromfpd64
ufromfpf
ufromfpl
ufromfpx
ufromfpxd
ufromfpxd128
ufromfpxd32
ufromfpxd64
ufromfpxf
ufromfpxl
ULLONG_MAX
ULLONG_WIDTH
ULONG_MAX
ULONG_WIDTH
undef
ungetc
ungetwc
union
unreachable
unsequenced
unsigned
USHRT_MAX
USHRT_WIDTH
va_arg
va_copy
va_end
va_list
va_start
vfprintf

vfprintf_s
vfscanf
vfscanf_s
vfwprintf
vfwprintf_s
vfwscanf
vfwscanf_s
void
volatile
vprintf
vprintf_s
vscanf
vscanf_s
vsnprintf
vsnprintf_s
vsnwprintf_s
vsprintf
vsprintf_s
vsscanf
vsscanf_s
vswprintf
vswprintf_s
vswscanf
vswscanf_s
vwprintf
vwprintf_s
vwscanf
vwscanf_s
warning
WCHAR_MAX
WCHAR_MIN
wchar_t
WCHAR_WIDTH
wcrtomb
wcrtomb_s
wctob
wctomb
wctomb_s
wctrans
wctrans_t
wctype
wctype_t
WEOF
while
WINT_MAX
WINT_MIN
wint_t
WINT_WIDTH
wmemchr
wmemcmp
wmemcpy
wmemcpy_s
wmemmove
wmemmove_s
wmemset
wprintf
wprintf_s
wscanf
wscanf_s
X_
xaddd
xaddf
X_DECIMAL_DIG
X_DIG
xdivd
xdivf
X_EPSILON
xfmad
xfmaf
X_MANT_DIG
X_MAX
X_MAX_10_EXP
X_MAX_EXP
X_MIN
X_MIN_10_EXP
X_MIN_EXP
xmuld
xmulf
xor
xor_eq
X_SNAN
xsqrtd
xsqrtf
xsubd
xsubf
X_TRUE_MIN

Annex K
(normative)
Bounds-checking interfaces

K.1 Background

1 Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.

2 A common programming style is to declare character arrays large enough to handle most practical cases. However, if these arrays are not large enough to handle the resulting strings, data can be written past the end of the array overwriting other data and program structures. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

3 Worse, this style of programming has compromised the security of computers and networks. Buffer overflows can often be exploited to run arbitrary code with the permissions of the vulnerable (defective) program.

4 If the programmer writes runtime checks to verify lengths before calling library functions, then those runtime checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.

5 This annex provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null terminated.

6 This annex also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

K.2 Scope

1 This annex specifies a series of optional extensions that can be useful in the mitigation of security vulnerabilities in programs, and comprise new functions, macros, and types declared or defined in existing standard headers.

2 An implementation that defines `__STDC_LIB_EXT1__` shall conform to the specifications in this annex.[465)]

3 Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of Clause 7.

K.3 Library

K.3.1 Introduction

K.3.1.1 Standard headers

1 The functions, macros, and types declared or defined in K.3 and its subclauses are not declared or defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant `0` at the point in the source file where the appropriate header is first included.

2 The functions, macros, and types declared or defined in K.3 and its subclauses are declared and defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant `1` at the point in the source file where the appropriate header is first

465)Implementations that do not define `__STDC_LIB_EXT1__` are not required to conform to these specifications.

included.[466)]

3 It is implementation-defined whether the functions, macros, and types declared or defined in K.3 and its subclauses are declared or defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is not defined as a macro at the point in the source file where the appropriate header is first included.[467)]

4 Within a preprocessing translation unit, `__STDC_WANT_LIB_EXT1__` shall be defined identically for all inclusions of any headers from Subclause K.3. If `__STDC_WANT_LIB_EXT1__` is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive were used.

K.3.1.2 Reserved identifiers

1 Each macro name in any of the following subclauses is reserved for use as specified if it is defined by any of its associated headers when included; unless explicitly stated otherwise (see 7.1.4).

2 All identifiers with external linkage in any of the following subclauses are reserved for use as identifiers with external linkage if any of them are used by the program. None of them are reserved if none of them are used.

3 Each identifier with file scope listed in any of the following subclauses is reserved for use as a macro name and as an identifier with file scope in the same name space if it is defined by any of its associated headers when included.

K.3.1.3 Use of errno

1 An implementation may set `errno` for the functions defined in this annex, but is not required to.

K.3.1.4 Runtime-constraint violations

1 Most functions in this annex include as part of their specification a list of runtime-constraints. These runtime-constraints are requirements on the program using the library.[468)]

2 Implementations shall verify that the runtime-constraints for a function are not violated by the program. If a runtime-constraint is violated, the implementation shall call the currently registered runtime-constraint handler (see `set_constraint_handler_s` in `<stdlib.h>`). Multiple runtime-constraint violations in the same call to a library function result in only one call to the runtime-constraint handler. It is unspecified which one of the multiple runtime-constraint violations cause the handler to be called.

3 If the runtime-constraints section for a function states an action to be performed when a runtime-constraint violation occurs, the function shall perform the action before calling the runtime-constraint handler. If the runtime-constraints section lists actions that are prohibited when a runtime-constraint violation occurs, then such actions are prohibited to the function both before calling the handler and after the handler returns.

4 The runtime-constraint handler might not return. If the handler does return, the library function whose runtime-constraint was violated shall return some indication of failure as given by the returns section in the function's specification.

K.3.2 Errors `<errno.h>`

1 The header `<errno.h>` defines a type.

2 The type is

```
errno_t
```

[466)]Future revisions of this document might define meanings for other values of `__STDC_WANT_LIB_EXT1__`.

[467)]Subclause 7.1.3 reserves certain names and patterns of names that an implementation can use in headers. All other names are not reserved, and a conforming implementation is not permitted to use them. While some of the names defined in K.3 and its subclauses are reserved, others are not. If an unreserved name is defined in a header when `__STDC_WANT_LIB_EXT1__` is defined as `0`, the implementation is not conforming.

[468)]Although runtime-constraints replace many cases of undefined behavior, undefined behavior still exists in this annex. Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior.

which is type **int**.[469]

K.3.3 Common definitions <stddef.h>

1 The header <stddef.h> defines a type.

2 The type is

```
rsize_t
```

which is the type **size_t**.[470]

K.3.4 Integer types <stdint.h>

1 The header <stdint.h> defines a macro.

2 The macro is

```
RSIZE_MAX
```

which expands to a value[471] of type **size_t**. Functions that have parameters of type **rsize_t** consider it a runtime-constraint violation if the values of those parameters are greater than **RSIZE_MAX**.

Recommended practice

3 Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like **size_t**. Also, some implementations do not support objects as large as the maximum value that can be represented by type **size_t**.

4 For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that **RSIZE_MAX** be defined as the smaller of the size of the largest object supported or (**SIZE_MAX** >> 1), even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define **RSIZE_MAX** as **SIZE_MAX**, which means that there is no object size that is considered a runtime-constraint violation.

K.3.5 Input/output <stdio.h>

1 The header <stdio.h> defines several macros and two types.

2 The macros are

```
L_tmpnam_s
```

which expands to an integer constant expression that is the size needed for an array of **char** large enough to hold a temporary file name string generated by the **tmpnam_s** function;

```
TMP_MAX_S
```

which expands to an integer constant expression that is the maximum number of unique file names that can be generated by the **tmpnam_s** function.

3 The types are

```
errno_t
```

which is type **int**; and

[469] As a matter of programming style, **errno_t** can be used as the type of something that deals only with the values that might be found in **errno**. For example, a function which returns the value of **errno** could be declared as having the return type **errno_t**.

[470] See the description of the **RSIZE_MAX** macro in <stdint.h>.

[471] The macro **RSIZE_MAX** need not expand to a constant expression.

```
rsize_t
```

which is the type **size_t**.

K.3.5.1 Operations on files

K.3.5.1.1 The tmpfile_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
```

Runtime-constraints

2 `streamptr` shall not be a null pointer.

3 If there is a runtime-constraint violation, **tmpfile_s** does not attempt to create a file.

Description

4 The **tmpfile_s** function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with `"wb+"` mode with the meaning that mode has in the **fopen_s** function (including the mode's effect on exclusive access and file permissions).

5 If the file was created successfully, then the pointer to **FILE** pointed to by `streamptr` will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to **FILE** pointed to by `streamptr` will be set to a null pointer.

Recommended practice

It should be possible to open at least **TMP_MAX_S** temporary files during the lifetime of the program (this limit may be shared with **tmpnam_s**) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (**FOPEN_MAX**).

Returns

6 The **tmpfile_s** function returns zero if it created the file. If it did not create the file or there was a runtime-constraint violation, **tmpfile_s** returns a nonzero value.

K.3.5.1.2 The tmpnam_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

Runtime-constraints

2 `s` shall not be a null pointer. `maxsize` shall be less than or equal to **RSIZE_MAX**. `maxsize` shall be greater than the length of the generated file name string.

Description

3 The **tmpnam_s** function generates a string that is a valid file name and that is not the same as the name of an existing file.[472] The function is potentially capable of generating **TMP_MAX_S** different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings shall be less than the value of the **L_tmpnam_s** macro.

4 The **tmpnam_s** function generates a different string each time it is called.

[472] Files created using strings generated by the **tmpnam_s** function are temporary only in the sense that their names are not expected to collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination.

5 It is assumed that s points to an array of at least maxsize characters. This array will be set to generated string, as specified below.

6 The implementation shall behave as if no library function except **tmpnam** calls the **tmpnam_s** function.[473)]

Recommended practice

7 After a program obtains a file name using the **tmpnam_s** function and before the program creates a file with that name, the possibility exists that someone else may create a file with that same name. To avoid this race condition, the **tmpfile_s** function should be used instead of **tmpnam_s** when possible. One situation that requires the use of the **tmpnam_s** function is when the program needs to create a temporary directory rather than a temporary file.

8 Implementations should take care in choosing the patterns used for names returned by **tmpnam_s**. For example, making a thread ID part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.

Returns

9 If no suitable string can be generated, or if there is a runtime-constraint violation, the **tmpnam_s** function:

— if s is not null and maxsize is both greater than zero and not greater than **RSIZE_MAX**, writes a null character to s[0]

— returns a nonzero value.

10 Otherwise, the **tmpnam_s** function writes the string in the array pointed to by s and returns zero.

Environmental limits

11 The value of the macro **TMP_MAX_S** shall be at least 25.

K.3.5.2 File access functions

K.3.5.2.1 The fopen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t fopen_s(FILE * restrict * restrict streamptr,
     const char * restrict filename, const char * restrict mode);
```

Runtime-constraints

2 None of streamptr, filename, or mode shall be a null pointer.

3 If there is a runtime-constraint violation, **fopen_s** does not attempt to open a file. Furthermore, if streamptr is not a null pointer, **fopen_s** sets *streamptr to the null pointer.

Description

4 The **fopen_s** function opens the file whose name is the string pointed to by filename, and associates a stream with it.

5 The mode string shall be as described for **fopen**, with the addition that modes starting with the character 'w' or 'a' may be preceded by the character 'u', see below:

uw	truncate to zero length or create text file for writing, default permissions
uwx	create text file for writing, default permissions
ua	append; open or create text file for writing at end-of-file, default permissions

[473)] An implementation can have **tmpnam** call **tmpnam_s** (perhaps so there is only one naming convention for temporary files), but this is not required.

uwb	truncate to zero length or create binary file for writing, default permissions
uwbx	create binary file for writing, default permissions
uab	append; open or create binary file for writing at end-of-file, default permissions
uw+	truncate to zero length or create text file for update, default permissions
uw+x	create text file for update, default permissions
ua+	append; open or create text file for update, writing at end-of-file, default permissions
uw+b *or* uwb+	truncate to zero length or create binary file for update, default permissions
uw+bx *or* uwb+x	create binary file for update, default permissions
ua+b *or* uab+	append; open or create binary file for update, writing at end-of-file, default permissions

6 Opening a file with exclusive mode ('x' as the last character in the mode argument) fails if the file already exists or cannot be created.

7 To the extent that the underlying system supports the concepts, files opened for writing shall be opened with exclusive (also known as non-shared) access. If the file is being created, and the first character of the mode string is not 'u', to the extent that the underlying system supports it, the file shall have a file permission that prevents other users on the system from accessing the file. If the file is being created and first character of the mode string is 'u', then by the time the file has been closed, it shall have the system default file access permissions.[474)]

8 If the file was opened successfully, then the pointer to **FILE** pointed to by streamptr will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to **FILE** pointed to by streamptr will be set to a null pointer.

Returns

9 The **fopen_s** function returns zero if it opened the file. If it did not open the file or if there was a runtime-constraint violation, **fopen_s** returns a nonzero value.

K.3.5.2.2 The freopen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t freopen_s(FILE * restrict * restrict newstreamptr,
      const char * restrict filename, const char * restrict mode,
      FILE * restrict stream);
```

Runtime-constraints

2 None of newstreamptr, mode, and stream shall be a null pointer.

3 If there is a runtime-constraint violation, **freopen_s** neither attempts to close any file associated with stream nor attempts to open a file. Furthermore, if newstreamptr is not a null pointer, **fopen_s** sets *newstreamptr to the null pointer.

Description

4 The **freopen_s** function opens the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument has the same meaning as in the **fopen_s** function (including the mode's effect on exclusive access and file permissions).

5 If filename is a null pointer, the **freopen_s** function attempts to change the mode of the stream to that specified by mode, as if the name of the file currently associated with the stream had been

474)These are the same permissions that the file would have been created with by **fopen**.

used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

6 The **freopen_s** function first attempts to close any file that is associated with stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

7 If the file was opened successfully, then the pointer to **FILE** pointed to by newstreamptr will be set to the value of stream. Otherwise, the pointer to **FILE** pointed to by newstreamptr will be set to a null pointer.

Returns

8 The **freopen_s** function returns zero if it opened the file. If it did not open the file or there was a runtime-constraint violation, **freopen_s** returns a nonzero value.

K.3.5.3 Formatted input/output functions

1 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

K.3.5.3.1 The fprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int fprintf_s(FILE * restrict stream, const char * restrict format, ...);
```

Runtime-constraints

2 Neither stream nor format shall be a null pointer. The %n specifier[475] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to **fprintf_s** corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the[476] **fprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **fprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **fprintf_s** function is equivalent to the **fprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **fprintf_s** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.2 The fscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int fscanf_s(FILE * restrict stream, const char * restrict format, ...);
```

Runtime-constraints

2 Neither stream nor format shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the[477] **fscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **fscanf_s** performed input before discovering the runtime-constraint violation.

[475] It is not a runtime-constraint violation for the characters %n to appear in sequence in the string pointed at by format when those characters are not a interpreted as a %n specifier. For example, if the entire format string was %%n.

[476] Because an implementation can treat any undefined behavior as a runtime-constraint violation, an implementation can treat any unsupported specifiers in the string pointed to by format as a runtime-constraint violation.

[477] Because an implementation can treat any undefined behavior as a runtime-constraint violation, an implementation can treat any unsupported specifiers in the string pointed to by format as a runtime-constraint violation.

Description

4 The **fscanf_s** function is equivalent to **fscanf** except that the c, s, and [conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a ∗). The first of these arguments is the same as for **fscanf**. That argument is immediately followed in the argument list by the second argument, which has type **rsize_t** and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[478)]

5 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

6 The **fscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **fscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7 **EXAMPLE 1** The call:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf_s(stdin, "%d%f%s", &i, &x, name, (rsize_t) 50);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to n the value 3, to i the value 25, to x the value 5.432, and to name the sequence thompson\0.

8 **EXAMPLE 2** The call:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
/* ... */
int n; char s[5];
n = fscanf_s(stdin, "%s", s, sizeof s);
```

with the input line:

```
hello
```

will assign to n the value 0 since a matching failure occurred because the sequence hello\0 requires an array of six characters to store it.

K.3.5.3.3 The printf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int printf_s(const char * restrict format, ...);
```

[478)]If the format is known at translation time, an implementation can issue a diagnostic for any argument used to store the result from a c, s, or [conversion specifier if that argument is not followed by an argument of a type compatible with **rsize_t**. A limited amount of checking can be done if even if the format is not known at translation time. For example, an implementation could issue a diagnostic for each argument after format that has of type pointer to one of **char**, **signed char**, **unsigned char**, or **void** that is not followed by an argument of a type compatible with **rsize_t**. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the hh length modifier, a length argument is expected to follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following format that did not have a type compatible with **rsize_t**.

Runtime-constraints

2 `format` shall not be a null pointer. The `%n` specifier[479] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **`printf_s`** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`printf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`printf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`printf_s`** function is equivalent to the **`printf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`printf_s`** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.4 The `scanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int scanf_s(const char * restrict format, ...);
```

Runtime-constraints

2 `format` shall not be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`scanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`scanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`scanf_s`** function is equivalent to **`fscanf_s`** with the argument **`stdin`** interposed before the arguments to **`scanf_s`**.

Returns

5 The **`scanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **`scanf_s`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.5 The `snprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int snprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than **`RSIZE_MAX`**. The `%n` specifier[480] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **`snprintf_s`** corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

[479] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

[480] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX`, then the `snprintf_s` function sets `s[0]` to the null character.

Description

4 The `snprintf_s` function is equivalent to the `snprintf` function except for the explicit runtime-constraints listed above.

5 The `snprintf_s` function, unlike `sprintf_s`, will truncate the result to fit within the array pointed to by `s`.

Returns

6 The `snprintf_s` function returns the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than `n`.

K.3.5.3.6 The `sprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX`. The number of characters (including the trailing null) required for the result to be written to the array pointed to by `s` shall not be greater than `n`. The `%n` specifier[481] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to `sprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX`, then the `sprintf_s` function sets `s[0]` to the null character.

Description

4 The `sprintf_s` function is equivalent to the `sprintf` function except for the parameter `n` and the explicit runtime-constraints listed above.

5 The `sprintf_s` function, unlike `snprintf_s`, treats a result too big for the array pointed to by `s` as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the `sprintf_s` function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, `sprintf_s` returns a negative value. If any other runtime-constraint violation occurred, `sprintf_s` returns zero.

[481]It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

K.3.5.3.7 The sscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sscanf_s(const char * restrict s, const char * restrict format, ...);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`sscanf_s`** function does not attempt to perform further input, and it is unspecified to what extent **`sscanf_s`** performed input before discovering the runtime-constraint violation.

Description

4 The **`sscanf_s`** function is equivalent to **`fscanf_s`**, except that input is obtained from a string (specified by the argument `s`) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **`fscanf_s`** function. If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

5 The **`sscanf_s`** function returns the value of the macro **`EOF`** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **`sscanf_s`** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.8 The vfprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfprintf_s(FILE *restrict stream, const char *restrict format, va_list arg);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. The `%n` specifier[482] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **`vfprintf_s`** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`vfprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`vfprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`vfprintf_s`** function is equivalent to the **`vfprintf`** function except for the explicit runtime-constraints listed above.

Returns

5 The **`vfprintf_s`** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

[482] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

K.3.5.3.9 The vfscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfscanf_s(FILE *restrict stream, const char *restrict format, va_list arg);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vfscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vfscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vfscanf_s** function is equivalent to **fscanf_s**, with the variable argument list replaced by `arg`, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfscanf_s** function does not invoke the **va_end** macro.[483)]

Returns

5 The **vfscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vfscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.10 The vprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vprintf_s(const char * restrict format, va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. The `%n` specifier[484)] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **vprintf_s** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **vprintf_s** function is equivalent to the **vprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vprintf_s** function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.11 The vscanf_s function

[483)]As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the representation of `arg` after the return is indeterminate.

[484)]It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vscanf_s(const char * restrict format, va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vscanf_s** function is equivalent to **scanf_s**, with the variable argument list replaced by `arg`, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf_s** function does not invoke the **va_end** macro.[485]

Returns

5 The **vscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.12 The vsnprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsnprintf_s(char *restrict s, rsize_t n, const char *restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than **RSIZE_MAX**. The `%n` specifier[486] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to **vsnprintf_s** corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than **RSIZE_MAX**, then the **vsnprintf_s** function sets `s[0]` to the null character.

Description

4 The **vsnprintf_s** function is equivalent to the **vsnprintf** function except for the explicit runtime-constraints listed above.

5 The **vsnprintf_s** function, unlike **vsprintf_s**, will truncate the result to fit within the array pointed to by `s`.

Returns

6 The **vsnprintf_s** function returns the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and

[485] As the functions **vfprintf_s**, **vfscanf_s**, **vprintf_s**, **vscanf_s**, **vsnprintf_s**, **vsprintf_s**, and **vsscanf_s** invoke the **va_arg** macro, the representation of `arg` after the return is indeterminate.

[486] It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

only if the returned value is both nonnegative and less than n.

K.3.5.3.13 The vsprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsprintf_s(char * restrict s, rsize_t n, const char * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither s nor format shall be a null pointer. n shall neither equal zero nor be greater than RSIZE_MAX. The number of characters (including the trailing null) required for the result to be written to the array pointed to by s shall not be greater than n. The %n specifier[487] (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to vsprintf_s corresponding to a %s specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if s is not a null pointer and n is greater than zero and not greater than RSIZE_MAX, then the vsprintf_s function sets s[0] to the null character.

Description

4 The vsprintf_s function is equivalent to the vsprintf function except for the parameter n and the explicit runtime-constraints listed above.

5 The vsprintf_s function, unlike vsnprintf_s, treats a result too big for the array pointed to by s as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the vsprintf_s function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, vsprintf_s returns a negative value. If any other runtime-constraint violation occurred, vsprintf_s returns zero.

K.3.5.3.14 The vsscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsscanf_s(const char *restrict s, const char *restrict format, va_list arg);
```

Runtime-constraints

2 Neither s nor format shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the vsscanf_s function does not attempt to perform further input, and it is unspecified to what extent vsscanf_s performed input before discovering the runtime-constraint violation.

Description

4 The vsscanf_s function is equivalent to sscanf_s, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsscanf_s function does not invoke the va_end macro.[488]

[487] It is not a runtime-constraint violation for the characters %n to appear in sequence in the string pointed at by format when those characters are not a interpreted as a %n specifier. For example, if the entire format string was %%n.

[488] As the functions vfprintf_s, vfscanf_s, vprintf_s, vscanf_s, vsnprintf_s, vsprintf_s, and vsscanf_s invoke the va_arg macro, the value of arg after the return is indeterminate.

Returns

5 The `vsscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.4 Character input/output functions

K.3.5.4.1 The `gets_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
char *gets_s(char *s, rsize_t n);
```

Runtime-constraints

2 `s` shall not be a null pointer. `n` shall neither be equal to zero nor be greater than `RSIZE_MAX`. A new-line character, end-of-file, or read error shall occur within reading `n-1` characters from `stdin`.[489)]

3 If there is a runtime-constraint violation, characters are read and discarded from `stdin` until a new-line character is read, or end-of-file or a read error occurs, and if `s` is not a null pointer, `s[0]` is set to the null character.

Description

4 The `gets_s` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stdin`, into the array pointed to by `s`. No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.

5 If end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then `s[0]` is set to the null character, and the other elements of `s` take unspecified values.

Recommended practice

6 The `fgets` function allows properly-written programs to safely process input lines too long to store in the result array. In general this requires that callers of `fgets` pay attention to the presence or absence of a new-line character in the result array. Consider using `fgets` (along with any needed processing based on new-line characters) instead of `gets_s`.

Returns

7 The `gets_s` function returns `s` if successful. If there was a runtime-constraint violation, or if end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then a null pointer is returned.

[489)] The `gets_s` function, unlike the historical `gets` function, makes it a runtime-constraint violation for a line of input to overflow the buffer to store it. Unlike the `fgets` function, `gets_s` maintains a one-to-one relationship between input lines and successful calls to `gets_s`. Programs that use `gets` expect such a relationship.

K.3.6 General utilities <stdlib.h>

1 The header `<stdlib.h>` defines three types.

2 The types are

```
errno_t
```

which is type **int**; and

```
rsize_t
```

which is the type **size_t**; and

```
constraint_handler_t
```

which has the following definition

```
typedef void (*constraint_handler_t)(
      const char * restrict msg,
      void * restrict ptr,
      errno_t error);
```

K.3.6.1 Runtime-constraint handling

K.3.6.1.1 The set_constraint_handler_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
```

Description

2 The **set_constraint_handler_s** function sets the runtime-constraint handler to be `handler`. The runtime-constraint handler is the function to be called when a library function detects a runtime-constraint violation. Only the most recent handler registered with **set_constraint_handler_s** is called when a runtime-constraint violation occurs.

3 When the handler is called, it is passed the following arguments in the following order:

1. A pointer to a character string describing the runtime-constraint violation.
2. A null pointer or a pointer to an implementation-defined object.
3. If the function calling the handler has a return type declared as **errno_t**, the return value of the function is passed. Otherwise, a positive value of type **errno_t** is passed.

4 The implementation has a default constraint handler that is used if no calls to the **set_constraint_handler_s** function have been made. The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort.

5 If the `handler` argument to **set_constraint_handler_s** is a null pointer, the implementation default handler becomes the current constraint handler.

Returns

6 The **set_constraint_handler_s** function returns a pointer to the previously registered handler.[490)]

[490)] If the previous handler was registered by calling **set_constraint_handler_s** with a null pointer argument, a pointer to the implementation default handler is returned (not NULL).

K.3.6.1.2 The abort_handler_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void abort_handler_s(const char * restrict msg, void * restrict ptr,
     errno_t error);
```

Description

2 A pointer to the **abort_handler_s** function shall be a suitable argument to the **set_constraint_handler_s** function.

3 The **abort_handler_s** function writes a message on the standard error stream in an implementation-defined format. The message shall include the string pointed to by msg. The **abort_handler_s** function then calls the **abort** function.[491)]

Returns

4 The **abort_handler_s** function does not return to its caller.

K.3.6.1.3 The ignore_handler_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void ignore_handler_s(const char * restrict msg, void * restrict ptr,
     errno_t error);
```

Description

2 A pointer to the **ignore_handler_s** function shall be a suitable argument to the **set_constraint_handler_s** function.

3 The **ignore_handler_s** function simply returns to its caller.[492)]

Returns

4 The **ignore_handler_s** function returns no value.

K.3.6.2 Communication with the environment

K.3.6.2.1 The getenv_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t getenv_s(size_t * restrict len, char * restrict value, rsize_t maxsize,
     const char * restrict name);
```

Runtime-constraints

2 name shall not be a null pointer. maxsize shall not be greater than **RSIZE_MAX**. If maxsize is not equal to zero, then value shall not be a null pointer.

491)Many implementations invoke a debugger when the **abort** function is called.

492)If the runtime-constraint handler is set to the **ignore_handler_s** function, any library function in which a runtime-constraint violation occurs will return to its caller. The caller can determine whether a runtime-constraint violation occurred based on the library function's specification (usually, the library function returns a nonzero **errno_t**).

3 If there is a runtime-constraint violation, the integer pointed to by `len` is set to 0 (if `len` is not null), and the environment list is not searched.

Description

4 The **`getenv_s`** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by `name`.

5 If that name is found then **`getenv_s`** performs the following actions. If `len` is not a null pointer, the length of the string associated with the matched list member is stored in the integer pointed to by `len`. If the length of the associated string is less than `maxsize`, then the associated string is copied to the array pointed to by `value`.

6 If that name is not found then **`getenv_s`** performs the following actions. If `len` is not a null pointer, zero is stored in the integer pointed to by `len`. If `maxsize` is greater than zero, then `value[0]` is set to the null character.

7 The set of environment names and the method for altering the environment list are implementation-defined. The **`getenv_s`** function need not avoid data races with other threads of execution that modify the environment list.[493)]

Returns

8 The **`getenv_s`** function returns zero if the specified `name` is found and the associated string was successfully stored in `value`. Otherwise, a nonzero value is returned.

K.3.6.3 Searching and sorting utilities

1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as **`size_t`** `nmemb` specifies the length of the array for a function, if `nmemb` has the value zero on a call to that function, then the comparison function is not called, a search finds no matching element, sorting performs no rearrangement, and the pointer to the array may be null.

2 The implementation shall ensure that the second argument of the comparison function (when called from **`bsearch_s`**), or both arguments (when called from **`qsort_s`**), are pointers to elements of the array.[494)] The first argument when called from **`bsearch_s`** shall equal `key`.

3 The comparison function shall not alter the contents of either the array or search key. The implementation may reorder elements of the array between calls to the comparison function, but shall not otherwise alter the contents of any individual element.

4 When the same objects (consisting of `size` bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for **`qsort_s`** they shall define a total ordering on the array, and for **`bsearch_s`** the same object shall always compare the same way with the key.

5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

K.3.6.3.1 The `bsearch_s` generic function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
QVoid *bsearch_s(const void *key, QVoid *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *k, const void *y, void *context),
      void *context);
```

493)Many implementations provide non-standard functions that modify the environment list.
494)That is, if the value passed is `p`, then the following expressions are always valid and nonzero:

```
((char *)p -  (char *)base) % size == 0
 (char *)p >= (char *)base
 (char *)p <  (char *)base + nmemb * size
```

Runtime-constraints

2 Neither `nmemb` nor `size` shall be greater than **`RSIZE_MAX`**. If `nmemb` is not equal to zero, then none of `key`, `base`, or `compar` shall be a null pointer.

3 If there is a runtime-constraint violation, the **`bsearch_s`** generic function does not search the array.

Description

4 The **`bsearch_s`** generic function searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element of the array is specified by `size`.

5 The comparison function pointed to by `compar` is called with three arguments. The first two point to the `key` object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.[495] The third argument to the comparison function is the `context` argument passed to **`bsearch_s`**. The sole use of `context` by **`bsearch_s`** is to pass it to the comparison function.[496]

Returns

6 The **`bsearch_s`** generic function returns a pointer to a matching element of the array, or a null pointer if no match is found or there is a runtime-constraint violation. If two elements compare as equal, which element is matched is unspecified.

7 The **`bsearch_s`** generic function is generic in the qualification of the type pointed to by the argument `base`. If this argument is a pointer to a **`const`**-qualified object type, the returned pointer will be a pointer to **`const`**-qualified **`void`**. Otherwise, the argument shall be a pointer to an unqualified object type or a null pointer constant.[497], and the returned pointer will be a pointer to unqualified **`void`**

8 The external declaration of **`bsearch_s`** has the concrete type:

```
void * (const void *, const void *, rsize_t, rsize_t, int (*) (const void *, const void
     *), void *)
```

which supports all correct uses. If a macro definition of the generic function is suppressed to access an actual function, the external declaration with this concrete type is visible.[498]

K.3.6.3.2 The `qsort_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *x, const void *y, void *context),
      void *context);
```

Runtime-constraints

2 Neither `nmemb` nor `size` shall be greater than **`RSIZE_MAX`**. If `nmemb` is not equal to zero, then neither `base` nor `compar` shall be a null pointer.

3 If there is a runtime-constraint violation, the **`qsort_s`** function does not sort the array.

---

[495] In practice, this means that the entire array has been sorted according to the comparison function.
[496] The `context` argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.
[497] If the argument is a null pointer and the call is executed, the behavior is undefined.
[498] This is an obsolescent feature.

Description

4 The `qsort_s` function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.

5 The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with three arguments. The first two point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. The third argument to the comparison function is the `context` argument passed to `qsort_s`. The sole use of `context` by `qsort_s` is to pass it to the comparison function.[499)]

6 If two elements compare as equal, their relative order in the resulting sorted array is unspecified.

Returns

7 The `qsort_s` function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.6.4 Multibyte/wide character conversion functions

1 The behavior of the multibyte character functions is affected by the `LC_CTYPE` category of the current locale. For a state-dependent encoding, each function is placed into its initial conversion state by a call for which its character pointer argument, `s`, is a null pointer. Subsequent calls with `s` as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with `s` as a null pointer causes these functions to set the `int` pointed to by their `status` argument to a nonzero value if encodings have state dependency, and zero otherwise. [500)]

Changing the `LC_CTYPE` category causes the internal object describing the conversion state of these functions to have an indeterminate representation.

K.3.6.4.1 The `wctomb_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t wctomb_s(int *restrict status, char *restrict s, rsize_t smax,
      wchar_t wc);
```

Runtime-constraints

2 Let n denote the number of bytes needed to represent the multibyte character corresponding to the wide character given by `wc` (including any shift sequences).

3 If `s` is not a null pointer, then `smax` shall not be less than n, and `smax` shall not be greater than `RSIZE_MAX`. If `s` is a null pointer, then `smax` shall equal zero.

4 If there is a runtime-constraint violation, `wctomb_s` does not modify the `int` pointed to by `status`, and if `s` is not a null pointer, no more than `smax` elements in the array pointed to by `s` will be accessed.

Description

5 The `wctomb_s` function determines n and stores the multibyte character representation of `wc` in the array whose first element is pointed to by `s` (if `s` is not a null pointer). The number of characters stored never exceeds `MB_CUR_MAX` or `smax`. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

6 The implementation shall behave as if no library function calls the `wctomb_s` function.

7 If `s` is a null pointer, the `wctomb_s` function stores into the `int` pointed to by `status` a nonzero

[499)]The `context` argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.

[500)]If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings.

8 If `s` is not a null pointer, the **`wctomb_s`** function stores into the **`int`** pointed to by `status` either n or -1 if `wc`, respectively, does or does not correspond to a valid multibyte character.

9 In no case will the **`int`** pointed to by `status` be set to a value greater than the **`MB_CUR_MAX`** macro.

Returns

10 The **`wctomb_s`** function returns zero if successful, and a nonzero value if there was a runtime-constraint violation or `wc` did not correspond to a valid multibyte character.

K.3.6.5 Multibyte/wide string conversion functions

1 The behavior of the multibyte string functions is affected by the **`LC_CTYPE`** category of the current locale.

K.3.6.5.1 The `mbstowcs_s` function

Synopsis

1
```
#include <stdlib.h>
errno_t mbstowcs_s(size_t *restrict retval, wchar_t *restrict dst,
     rsize_t dstmax, const char * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither `retval` nor `src` shall be a null pointer. If `dst` is not a null pointer, then neither `len` nor `dstmax` shall be greater than **`RSIZE_MAX`**`/`**`sizeof`**`(`**`wchar_t`**`)`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer, then `dstmax` shall not equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then a null character shall occur within the first `dstmax` multibyte characters of the array pointed to by `src`.

3 If there is a runtime-constraint violation, then **`mbstowcs_s`** does the following. If `retval` is not a null pointer, then **`mbstowcs_s`** sets `*retval` to `(`**`size_t`**`)(-1)`. If `dst` is not a null pointer and `dstmax` is greater than zero and not greater than **`RSIZE_MAX`**`/`**`sizeof`**`(`**`wchar_t`**`)`, then **`mbstowcs_s`** sets `dst[0]` to the null wide character.

Description

4 The **`mbstowcs_s`** function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by `src` into a sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters are stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.[501] If `dst` is not a null pointer and no null wide character was stored into the array pointed to by `dst`, then `dst[len]` is set to the null wide character. Each conversion takes place as if by a call to the **`mbrtowc`** function.

5 Regardless of whether `dst` is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **`mbstowcs_s`** function stores the value `(`**`size_t`**`)(-1)` into `*retval`. Otherwise, the **`mbstowcs_s`** function stores into `*retval` the number of multibyte characters successfully converted, not including the terminating null character (if any).

6 All elements following the terminating null wide character (if any) written by **`mbstowcs_s`** in the array of `dstmax` wide characters pointed to by `dst` take unspecified values when **`mbstowcs_s`** returns.[502]

7 If copying takes place between objects that overlap, the objects take on unspecified values.

---

[501] Thus, the value of `len` is ignored if `dst` is a null pointer.

[502] This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.

Returns

8 The `mbstowcs_s` function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.6.5.2 The `wcstombs_s` function

Synopsis

1
```
#include <stdlib.h>
errno_t wcstombs_s(size_t * restrict retval, char * restrict dst, rsize_t dstmax,
     const wchar_t * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither `retval` nor `src` shall be a null pointer. If `dst` is not a null pointer, then `len` shall not be greater than `RSIZE_MAX/sizeof(wchar_t)` and `dstmax` shall be nonzero and not greater than `RSIZE_MAX`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

3 If there is a runtime-constraint violation, then `wcstombs_s` does the following. If `retval` is not a null pointer, then `wcstombs_s` sets `*retval` to `(size_t)(-1)`. If `dst` is not a null pointer and `dstmax` is greater than zero and not greater than `RSIZE_MAX`, then `wcstombs_s` sets `dst[0]` to the null character.

Description

4 The `wcstombs_s` function converts a sequence of wide characters from the array pointed to by `src` into a sequence of corresponding multibyte characters that begins in the initial shift state. If `dst` is not a null pointer, the converted characters are then stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

— when a wide character is reached that does not correspond to a valid multibyte character;

— (if `dst` is not a null pointer) when the next multibyte character would exceed the limit of n total bytes to be stored into the array pointed to by `dst`. If the wide character being converted is the null wide character, then n is the lesser of `len` or `dstmax`. Otherwise, n is the lesser of `len` or `dstmax-1`.

If the conversion stops without converting a null wide character and `dst` is not a null pointer, then a null character is stored into the array pointed to by `dst` immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the `wcrtomb` function.[503)]

5 Regardless of whether `dst` is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the `wcstombs_s` function stores the value `(size_t)(-1)` into `*retval`. Otherwise, the `wcstombs_s` function stores into `*retval` the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

6 All elements following the terminating null character (if any) written by `wcstombs_s` in the array of `dstmax` elements pointed to by `dst` take unspecified values when `wcstombs_s` returns.[504)]

7 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

8 The `wcstombs_s` function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

503)If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.

504)When `len` is not less than `dstmax`, the implementation might fill the array before discovering a runtime-constraint violation.

K.3.7 String handling <string.h>

1 The header `<string.h>` defines two types.

2 The types are

```
errno_t
```

which is type **int**; and

```
rsize_t
```

which is the type **size_t**.

K.3.7.1 Copying functions

K.3.7.1.1 The memcpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memcpy_s(void * restrict s1, rsize_t s1max, const void * restrict s2,
     rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than **RSIZE_MAX**. `n` shall not be greater than `s1max`. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, the **memcpy_s** function stores zeros in the first `s1max` characters of the object pointed to by `s1` if `s1` is not a null pointer and `s1max` is not greater than **RSIZE_MAX**.

Description

4 The **memcpy_s** function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`.

Returns

5 The **memcpy_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.2 The memmove_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memmove_s(void *s1, rsize_t s1max, const void *s2, rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than **RSIZE_MAX**. `n` shall not be greater than `s1max`.

3 If there is a runtime-constraint violation, the **memmove_s** function stores zeros in the first `s1max` characters of the object pointed to by `s1` if `s1` is not a null pointer and `s1max` is not greater than **RSIZE_MAX**.

Description

4 The **memmove_s** function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. This copying takes place as if the `n` characters from the object pointed to by `s2` are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by `s1` or `s2`, and then the `n` characters from the temporary array are copied into the object pointed to by `s1`.

Returns

5 The **memmove_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.3 The strcpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strcpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
```

Runtime-constraints

2 Neither s1 nor s2 shall be a null pointer. s1max shall not be greater than RSIZE_MAX. s1max shall not equal zero. s1max shall be greater than strnlen_s(s2, s1max). Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if s1 is not a null pointer and s1max is greater than zero and not greater than RSIZE_MAX, then strcpy_s sets s1[0] to the null character.

Description

4 The strcpy_s function copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1.

5 All elements following the terminating null character (if any) written by strcpy_s in the array of s1max characters pointed to by s1 take unspecified values when strcpy_s returns.[505]

Returns

6 The strcpy_s function returns zero[506] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.4 The strncpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2,
      rsize_t n);
```

Runtime-constraints

2 Neither s1 nor s2 shall be a null pointer. Neither s1max nor n shall be greater than RSIZE_MAX. s1max shall not equal zero. If n is not less than s1max, then s1max shall be greater than strnlen_s(s2, s1max). Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if s1 is not a null pointer and s1max is greater than zero and not greater than RSIZE_MAX, then strncpy_s sets s1[0] to the null character.

Description

4 The strncpy_s function copies not more than n successive characters (characters that follow a null character are not copied) from the array pointed to by s2 to the array pointed to by s1. If no null character was copied from s2, then s1[n] is set to a null character.

5 All elements following the terminating null character (if any) written by strncpy_s in the array of s1max characters pointed to by s1 take unspecified values when strncpy_s returns a nonzero value.[507]

[505]This allows an implementation to copy characters from s2 to s1 while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of s1 before discovering that the first element was set to the null character.

[506]A zero return value implies that all the requested characters from the string pointed to by s2 fit within the array pointed to by s1 and that the result in s1 is null terminated.

[507]This allows an implementation to copy characters from s2 to s1 while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of s1 before discovering that the first element was set to the null character.

Returns

6 The `strncpy_s` function returns zero[508] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

7 **EXAMPLE 1** The `strncpy_s` function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
/* ... */
char src1[100] = "hello";
char src2[7] = {'g', 'o', 'o', 'd', 'b', 'y', 'e'};
char dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = strncpy_s(dst1, 6, src1, 100);
r2 = strncpy_s(dst2, 5, src2, 7);
r3 = strncpy_s(dst3, 5, src2, 4);
```

The first call will assign to `r1` the value zero and to `dst1` the sequence `hello\0`.

The second call will assign to `r2` a nonzero value and to `dst2` the sequence `\0`.

The third call will assign to `r3` the value zero and to `dst3` the sequence `good\0`.

K.3.7.2 Concatenation functions

K.3.7.2.1 The `strcat_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strcat_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
```

Runtime-constraints

2 Let m denote the value `s1max - strnlen_s(s1, s1max)` upon entry to `strcat_s`.

3 Neither `s1` nor `s2` shall be a null pointer. `s1max` shall not be greater than `RSIZE_MAX`. `s1max` shall not equal zero. m shall not equal zero.[509] m shall be greater than `strnlen_s`(`s2`, m). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX`, then `strcat_s` sets `s1[0]` to the null character.

Description

5 The `strcat_s` function appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character from `s2` overwrites the null character at the end of `s1`.

6 All elements following the terminating null character (if any) written by `strcat_s` in the array of `s1max` characters pointed to by `s1` take unspecified values when `strcat_s` returns.[510]

[508] A zero return value implies that all of he requested characters from the string pointed to by `s2` fit within the array pointed to by `s1` and that the result in `s1` is null terminated.

[509] Zero means that `s1` was not null terminated upon entry to `strcat_s`.

[510] This allows an implementation to append characters from `s2` to `s1` while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of `s1` before discovering that the first element was set to the null character.

Returns

7 The `strcat_s` function returns zero[511] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.2.2 The `strncat_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncat_s(char * restrict s1, rsize_t s1max, const char * restrict s2,
      rsize_t n);
```

Runtime-constraints

2 Let m denote the value `s1max - strnlen_s(s1, s1max)` upon entry to `strncat_s`.

3 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX`. `s1max` shall not equal zero. m shall not equal zero.[512] If `n` is not less than m, then m shall be greater than `strnlen_s`(`s2`, m). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX`, then `strncat_s` sets `s1[0]` to the null character.

Description

5 The `strncat_s` function appends not more than `n` successive characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character from `s2` overwrites the null character at the end of `s1`. If no null character was copied from `s2`, then `s1[s1max-` m `+n]` is set to a null character.

6 All elements following the terminating null character (if any) written by `strncat_s` in the array of `s1max` characters pointed to by `s1` take unspecified values when `strncat_s` returns.[513]

Returns

7 The `strncat_s` function returns zero[514] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

8 **EXAMPLE 1** The `strncat_s` function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
/* ... */
char s1[100] = "good";
char s2[6] = "hello";
char s3[6] = "hello";
char s4[7] = "abc";
char s5[1000] = "bye";
int r1, r2, r3, r4;
r1 = strncat_s(s1, 100, s5, 1000);
r2 = strncat_s(s2, 6, "", 1);
r3 = strncat_s(s3, 6, "X", 2);
r4 = strncat_s(s4, 7, "defghijklmn", 3);
```

511) A zero return value implies that all the requested characters from the string pointed to by `s2` were appended to the string pointed to by `s1` and that the result in `s1` is null terminated.

512) Zero means that `s1` was not null terminated upon entry to `strncat_s`.

513) This allows an implementation to append characters from `s2` to `s1` while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of `s1` before discovering that the first element was set to the null character.

514) A zero return value implies that all the requested characters from the string pointed to by `s2` were appended to the string pointed to by `s1` and that the result in `s1` is null terminated.

After the first call r1 will have the value zero and s1 will contain the sequence goodbye\0.

After the second call r2 will have the value zero and s2 will contain the sequence hello\0.

After the third call r3 will have a nonzero value and s3 will contain the sequence \0.

After the fourth call r4 will have the value zero and s4 will contain the sequence abcdef\0.

K.3.7.3 Search functions

K.3.7.3.1 The strtok_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
char *strtok_s(char * restrict s1, rsize_t * restrict s1max,
     const char * restrict s2, char ** restrict ptr);
```

Runtime-constraints

2 None of s1max, s2, or ptr shall be a null pointer. If s1 is a null pointer, then *ptr shall not be a null pointer. The value of *s1max shall not be greater than **RSIZE_MAX**. The end of the token found shall occur within the first *s1max characters of s1 for the first call, and shall occur within the first *s1max characters of where searching resumes on subsequent calls.

3 If there is a runtime-constraint violation, the **strtok_s** function does not indirect through the s1 or s2 pointers, and does not store a value in the object pointed to by ptr.

Description

4 A sequence of calls to the **strtok_s** function breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a character from the string pointed to by s2. The fourth argument points to a caller-provided **char** pointer into which the **strtok_s** function stores information necessary for it to continue scanning the same string.

5 The first call in a sequence has a non-null first argument and s1max points to an object whose value is the number of elements in the character array pointed to by the first argument. The first call stores an initial value in the object pointed to by ptr and updates the value pointed to by s1max to reflect the number of elements that remain in relation to ptr. Subsequent calls in the sequence have a null first argument and the objects pointed to by s1max and ptr are required to have the values stored by the previous call in the sequence, which are then updated. The separator string pointed to by s2 may be different from call to call.

6 The first call in the sequence searches the string pointed to by s1 for the first character that is *not* contained in the current separator string pointed to by s2. If no such character is found, then there are no tokens in the string pointed to by s1 and the **strtok_s** function returns a null pointer. If such a character is found, it is the start of the first token.

7 The **strtok_s** function then searches from there for the first character in s1 that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches in the same string for a token return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token.

8 In all cases, the **strtok_s** function stores sufficient information in the pointer pointed to by ptr so that subsequent calls, with a null pointer for s1 and the unmodified pointer value for ptr, shall start searching just past the element overwritten by a null character (if any).

Returns

9 The **strtok_s** function returns a pointer to the first character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

10 **EXAMPLE**

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
```

```
static char str1[] = "?a???b,,,#c";
static char str2[] = "\t \t";
char *t, *ptr1, *ptr2;
rsize_t max1 = sizeof(str1);
rsize_t max2 = sizeof(str2);

t = strtok_s(str1, &max1, "?", &ptr1);    // t points to the token "a"
t = strtok_s(NULL, &max1, ",", &ptr1);    // t points to the token "??b"
t = strtok_s(str2, &max2, " \t", &ptr2);  // t is a null pointer
t = strtok_s(NULL, &max1, "#,", &ptr1);   // t points to the token "c"
t = strtok_s(NULL, &max1, "?", &ptr1);    // t is a null pointer
```

K.3.7.4 Miscellaneous functions

K.3.7.4.1 The `memset_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
```

Runtime-constraints

2 `s` shall not be a null pointer. Neither `smax` nor `n` shall be greater than `RSIZE_MAX`. `n` shall not be greater than `smax`.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `smax` is not greater than `RSIZE_MAX`, the `memset_s` function stores the value of `c` (converted to an `unsigned char`) into each of the first `smax` characters of the object pointed to by `s`.

Description

4 The `memset_s` function copies the value of `c` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `s`. Unlike `memset`, any call to the `memset_s` function shall be evaluated strictly according to the rules of the abstract machine as described in (5.1.2.3). That is, any call to the `memset_s` function shall assume that the memory indicated by `s` and `n` may be accessible in the future and thus contains the values indicated by `c`.

Returns

5 The `memset_s` function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.4.2 The `strerror_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strerror_s(char *s, rsize_t maxsize, errno_t errnum);
```

Runtime-constraints

2 `s` shall not be a null pointer. `maxsize` shall not be greater than `RSIZE_MAX`. `maxsize` shall not equal zero.

3 If there is a runtime-constraint violation, then the array (if any) pointed to by `s` is not modified.

Description

4 The `strerror_s` function maps the number in `errnum` to a locale-specific message string. Typically, the values for `errnum` come from `errno`, but `strerror_s` shall map any value of type `int` to a message.

5 If the length of the desired string is less than `maxsize`, then the string is copied to the array pointed to by `s`.

6 Otherwise, if `maxsize` is greater than zero, then `maxsize-1` characters are copied from the string to the array pointed to by `s` and then `s[maxsize-1]` is set to the null character. Then, if `maxsize` is greater than 3, then `s[maxsize-2]`, `s[maxsize-3]`, and `s[maxsize-4]` are set to the character period (`.`).

Returns

7 The **`strerror_s`** function returns zero if the length of the desired string was less than `maxsize` and there was no runtime-constraint violation. Otherwise, the **`strerror_s`** function returns a nonzero value.

K.3.7.4.3 The `strerrorlen_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strerrorlen_s(errno_t errnum);
```

Description

2 The **`strerrorlen_s`** function calculates the length of the (untruncated) locale-specific message string that the **`strerror_s`** function maps to `errnum`.

Returns

3 The **`strerrorlen_s`** function returns the number of characters (not including the null character) in the full message string.

K.3.7.4.4 The `strnlen_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t maxsize);
```

Description

2 The **`strnlen_s`** function computes the length of the string pointed to by `s`.

Returns

3 If `s` is a null pointer,[515] then the **`strnlen_s`** function returns zero.

4 Otherwise, the **`strnlen_s`** function returns the number of characters that precede the terminating null character. If there is no null character in the first `maxsize` characters of `s` then **`strnlen_s`** returns `maxsize`. At most the first `maxsize` characters of `s` shall be accessed by **`strnlen_s`**.

K.3.8 Date and time `<time.h>`

1 The header `<time.h>` defines two types.

2 The types are

```
errno_t
```

which is type **`int`**; and

```
rsize_t
```

which is the type **`size_t`**.

[515]Note that the **`strnlen_s`** function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated string argument make **`strnlen_s`** useful in algorithms that gracefully handle such exceptional data.

K.3.8.1 Components of time

1 A broken-down time is *normalized* if the values of the members of the `tm` structure are in their normal rages.[516)]

K.3.8.2 Time conversion functions

1 Like the `strftime` function, the `asctime_s` and `ctime_s` functions do not return a pointer to a static object, and other library functions are permitted to call them.

K.3.8.2.1 The `asctime_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);
```

Runtime-constraints

2 Neither `s` nor `timeptr` shall be a null pointer. `maxsize` shall not be less than 26 and shall not be greater than `RSIZE_MAX`. The broken-down time pointed to by `timeptr` shall be normalized. The calendar year represented by the broken-down time pointed to by `timeptr` shall not be less than calendar year 0 and shall not be greater than calendar year 9999.

3 If there is a runtime-constraint violation, there is no attempt to convert the time, and `s[0]` is set to a null character if `s` is not a null pointer and `maxsize` is not zero and is not greater than `RSIZE_MAX`.

Description

4 The `asctime_s` function converts the normalized broken-down time in the structure pointed to by `timeptr` into a 26 character (including the null character) string in the form

```
Sun Sep 16 01:03:52 1973\n\0
```

The fields making up this string are (in order):

1. The name of the day of the week represented by `timeptr->tm_wday` using the following three character weekday names: `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, and `Sat`.
2. The character space.
3. The name of the month represented by `timeptr->tm_mon` using the following three character month names: `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, and `Dec`.
4. The character space.
5. The value of `timeptr->tm_mday` as if printed using the `fprintf` format `"%2d"`.
6. The character space.
7. The value of `timeptr->tm_hour` as if printed using the `fprintf` format `"%.2d"`.
8. The character colon.
9. The value of `timeptr->tm_min` as if printed using the `fprintf` format `"%.2d"`.
10. The character colon.
11. The value of `timeptr->tm_sec` as if printed using the `fprintf` format `"%.2d"`.
12. The character space.
13. The value of `timeptr->tm_year + 1900` as if printed using the `fprintf` format `"%4d"`.
14. The character new line.
15. The null character.

[516)]The normal ranges are defined in 7.29.1.

Recommended practice

The **strftime** function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the **asctime_s** function, consider using the **strftime** function instead.

Returns

5 The **asctime_s** function returns zero if the time was successfully converted and stored into the array pointed to by s. Otherwise, it returns a nonzero value.

K.3.8.2.2 The ctime_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t ctime_s(char *s, rsize_t maxsize, const time_t *timer);
```

Runtime-constraints

2 Neither s nor timer shall be a null pointer. maxsize shall not be less than 26 and shall not be greater than **RSIZE_MAX**.

3 If there is a runtime-constraint violation, s[0] is set to a null character if s is not a null pointer and maxsize is not equal zero and is not greater than **RSIZE_MAX**.

Description

4 The **ctime_s** function converts the calendar time pointed to by timer to local time in the form of a string. It is equivalent to

```
asctime_s(s, maxsize, localtime_s(timer, &(struct tm){ 0 }))
```

Recommended practice

The **strftime** function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the **ctime_s** function, consider using the **strftime** function instead.

Returns

5 The **ctime_s** function returns zero if the time was successfully converted and stored into the array pointed to by s. Otherwise, it returns a nonzero value.

K.3.8.2.3 The gmtime_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *gmtime_s(const time_t * restrict timer, struct tm * restrict result);
```

Runtime-constraints

2 Neither timer nor result shall be a null pointer.

3 If there is a runtime-constraint violation, there is no attempt to convert the time.

Description

4 The **gmtime_s** function converts the calendar time pointed to by timer into a broken-down time, expressed as UTC. The broken-down time is stored in the structure pointed to by result.

Returns

5 The **gmtime_s** function returns result, or a null pointer if the specified time cannot be converted to UTC or there is a runtime-constraint violation.

K.3.8.2.4 The localtime_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *localtime_s(const time_t *restrict timer, struct tm *restrict result);
```

Runtime-constraints

2 Neither `timer` nor `result` shall be a null pointer.

3 If there is a runtime-constraint violation, there is no attempt to convert the time.

Description

4 The **`localtime_s`** function converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time. The broken-down time is stored in the structure pointed to by `result`.

Returns

5 The **`localtime_s`** function returns `result`, or a null pointer if the specified time cannot be converted to local time or there is a runtime-constraint violation.

K.3.9 Extended multibyte and wide character utilities <wchar.h>

1 The header `<wchar.h>` defines two types.

2 The types are

```
errno_t
```

which is type **`int`**; and

```
rsize_t
```

which is the type **`size_t`**.

3 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

K.3.9.1 Formatted wide character input/output functions

K.3.9.1.1 The fwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int fwprintf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. The `%n` specifier[517)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to **`fwprintf_s`** corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **`fwprintf_s`** function does not attempt to produce further output, and it is unspecified to what extent **`fwprintf_s`** produced output before discovering the runtime-constraint violation.

Description

4 The **`fwprintf_s`** function is equivalent to the **`fwprintf`** function except for the explicit runtime-constraints listed above.

517)It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

Returns

5 The `fwprintf_s` function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.2 The `fwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <wchar.h>
int fwscanf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the `fwscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `fwscanf_s` performed input before discovering the runtime-constraint violation.

Description

4 The `fwscanf_s` function is equivalent to `fwscanf` except that the `c`, `s`, and `[` conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a `*`). The first of these arguments is the same as for `fwscanf`. That argument is immediately followed in the argument list by the second argument, which has type `size_t` and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.[518)]

5 A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

6 The `fwscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `fwscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.3 The `snwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int snwprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format,
      ...);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX /sizeof(wchar_t)`. The `%n` specifier[519)] (modified or not by flags, field width, or precision) shall

---

[518)] If the format is known at translation time, an implementation can issue a diagnostic for any argument used to store the result from a `c`, `s`, or `[` conversion specifier if that argument is not followed by an argument of a type compatible with `rsize_t`. A limited amount of checking can be done if even if the format is not known at translation time. For example, an implementation could issue a diagnostic for each argument after `format` that has of type pointer to one of `char`, `signed char`, `unsigned char`, or `void` that is not followed by an argument of a type compatible with `rsize_t`. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the `hh` length modifier, a length argument is expected to follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following `format` that did not have a type compatible with `rsize_t`.

[519)] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

not appear in the wide string pointed to by `format`. Any argument to `snwprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `snwprintf_s` function sets `s[0]` to the null wide character.

Description

4 The `snwprintf_s` function is equivalent to the `swprintf` function except for the explicit runtime-constraints listed above.

5 The `snwprintf_s` function, unlike `swprintf_s`, will truncate the result to fit within the array pointed to by `s`.

Returns

6 The `snwprintf_s` function returns the number of wide characters that would have been written had `n` been sufficiently large, not counting the terminating wide null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than `n`.

K.3.9.1.4 The `swprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int swprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format,
    ...);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX/sizeof(wchar_t)`. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by `s` shall not be greater than `n`. The `%n` specifier[520] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `swprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `swprintf_s` function sets `s[0]` to the null wide character.

Description

4 The `swprintf_s` function is equivalent to the `swprintf` function except for the explicit runtime-constraints listed above.

5 The `swprintf_s` function, unlike `snwprintf_s`, treats a result too big for the array pointed to by `s` as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the `swprintf_s` function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if `n` or more wide characters are requested to be written, `swprintf_s` returns a negative value. If any other runtime-constraint violation occurred, `swprintf_s` returns zero.

K.3.9.1.5 The `swscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
```

[520]It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

```
#include <wchar.h>
int swscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Runtime-constraints

2 Neither s nor format shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **swscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **swscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **swscanf_s** function is equivalent to **fwscanf_s**, except that the argument s specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the **fwscanf_s** function.

Returns

5 The **swscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **swscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.6 The vfwprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf_s(FILE * restrict stream, const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither stream nor format shall be a null pointer. The %n specifier[521] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by format. Any argument to **vfwprintf_s** corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vfwprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **vfwprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **vfwprintf_s** function is equivalent to the **vfwprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **vfwprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.7 The vfwscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
```

[521]It is not a runtime-constraint violation for the wide characters %n to appear in sequence in the wide string pointed at by format when those wide characters are not a interpreted as a %n specifier. For example, if the entire format string was L"%%n".

```
#include <wchar.h>
int vfwscanf_s(FILE * restrict stream, const wchar_t * restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vfwscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `vfwscanf_s` performed input before discovering the runtime-constraint violation.

Description

4 The `vfwscanf_s` function is equivalent to `fwscanf_s`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwscanf_s` function does not invoke the `va_end` macro.[522)]

Returns

5 The `vfwscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vfwscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.8 The `vsnwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vsnwprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format,
     va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX/sizeof(wchar_t)`. The `%n` specifier[523)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vsnwprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `vsnwprintf_s` function sets `s[0]` to the null wide character.

Description

4 The `vsnwprintf_s` function is equivalent to the `vswprintf` function except for the explicit runtime-constraints listed above.

5 The `vsnwprintf_s` function, unlike `vswprintf_s`, will truncate the result to fit within the array pointed to by `s`.

Returns

6 The `vsnwprintf_s` function returns the number of wide characters that would have been written had `n` been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is both nonnegative and less than `n`.

[522)] As the functions `vfwscanf_s`, `vwscanf_s`, and `vswscanf_s` invoke the `va_arg` macro, the representation of `arg` after the return is indeterminate.

[523)] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

K.3.9.1.9 The vswprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than **RSIZE_MAX /sizeof(wchar_t)**. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by `s` shall not be greater than `n`. The `%n` specifier[524] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to **vswprintf_s** corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than **RSIZE_MAX/sizeof(wchar_t)**, then the **vswprintf_s** function sets `s[0]` to the null wide character.

Description

4 The **vswprintf_s** function is equivalent to the **vswprintf** function except for the explicit runtime-constraints listed above.

5 The **vswprintf_s** function, unlike **vsnwprintf_s**, treats a result too big for the array pointed to by `s` as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the **vswprintf_s** function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if `n` or more wide characters are requested to be written, **vswprintf_s** returns a negative value. If any other runtime-constraint violation occurred, **vswprintf_s** returns zero.

K.3.9.1.10 The vswscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswscanf_s(const wchar_t * restrict s, const wchar_t * restrict format,
      va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **vswscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **vswscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **vswscanf_s** function is equivalent to **swscanf_s**, with the variable argument list replaced by `arg`, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg**

[524] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

calls). The `vswscanf_s` function does not invoke the `va_end` macro.[525)]

Returns

5 The `vswscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vswscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.11 The `vwprintf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwprintf_s(const wchar_t * restrict format, va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. The `%n` specifier[526)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vwprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vwprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `vwprintf_s` produced output before discovering the runtime-constraint violation.

Description

4 The `vwprintf_s` function is equivalent to the `vwprintf` function except for the explicit runtime-constraints listed above.

Returns

5 The `vwprintf_s` function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.12 The `vwscanf_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwscanf_s(const wchar_t * restrict format, va_list arg);
```

Runtime-constraints

2 `format` shall not be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vwscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `vwscanf_s` performed input before discovering the runtime-constraint violation.

Description

4 The `vwscanf_s` function is equivalent to `wscanf_s`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls).

[525)] As the functions `vfwscanf_s`, `vwscanf_s`, and `vswscanf_s` invoke the `va_arg` macro, the representation of `arg` after the return is indeterminate.

[526)] It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

The **vwscanf_s** function does not invoke the **va_end** macro.[527)]

Returns

5 The **vwscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **vwscanf_s** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.13 The wprintf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wprintf_s(const wchar_t * restrict format, ...);
```

Runtime-constraints

2 format shall not be a null pointer. The %n specifier[528)] (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by format. Any argument to **wprintf_s** corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the **wprintf_s** function does not attempt to produce further output, and it is unspecified to what extent **wprintf_s** produced output before discovering the runtime-constraint violation.

Description

4 The **wprintf_s** function is equivalent to the **wprintf** function except for the explicit runtime-constraints listed above.

Returns

5 The **wprintf_s** function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.14 The wscanf_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wscanf_s(const wchar_t * restrict format, ...);
```

Runtime-constraints

2 format shall not be a null pointer. Any argument indirected though to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the **wscanf_s** function does not attempt to perform further input, and it is unspecified to what extent **wscanf_s** performed input before discovering the runtime-constraint violation.

Description

4 The **wscanf_s** function is equivalent to **fwscanf_s** with the argument **stdin** interposed before the arguments to **wscanf_s**.

Returns

5 The **wscanf_s** function returns the value of the macro **EOF** if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the **wscanf_s** function returns

[527)] As the functions **vfwscanf_s**, **vwscanf_s**, and **vswscanf_s** invoke the **va_arg** macro, the representation of arg after the return is indeterminate.

[528)] It is not a runtime-constraint violation for the wide characters %n to appear in sequence in the wide string pointed at by format when those wide characters are not a interpreted as a %n specifier. For example, if the entire format string was L"%%n".

the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.2 General wide string utilities

K.3.9.2.1 Wide string copying functions

K.3.9.2.1.1 The wcscpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcscpy_s(wchar_t *restrict s1, rsize_t s1max,
      const wchar_t *restrict s2);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. `s1max` shall not be greater than `RSIZE_MAX/sizeof(wchar_t)`. `s1max` shall not equal zero. `s1max` shall be greater than `wcsnlen_s(s2, s1max)`. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `wcscpy_s` sets `s1[0]` to the null wide character.

Description

4 The `wcscpy_s` function copies the wide string pointed to by `s2` (including the terminating null wide character) into the array pointed to by `s1`.

5 All elements following the terminating null wide character (if any) written by `wcscpy_s` in the array of `s1max` wide characters pointed to by `s1` take unspecified values when `wcscpy_s` returns.[529)]

Returns

6 The `wcscpy_s` function returns zero[530)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.1.2 The wcsncpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcsncpy_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. `s1max` shall not equal zero. If `n` is not less than `s1max`, then `s1max` shall be greater than `wcsnlen_s(s2, s1max)`. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `wcsncpy_s` sets `s1[0]` to the null wide character.

Description

4 The `wcsncpy_s` function copies not more than `n` successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If no null wide character was copied from `s2`, then `s1[n]` is set to a null wide character.

---

529) This allows an implementation to copy wide characters from `s2` to `s1` while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of `s1` before discovering that the first element was set to the null wide character.

530) A zero return value implies that all the requested wide characters from the string pointed to by `s2` fit within the array pointed to by `s1` and that the result in `s1` is null terminated.

5 All elements following the terminating null wide character (if any) written by **wcsncpy_s** in the array of s1max wide characters pointed to by s1 take unspecified values when **wcsncpy_s** returns.[531)]

Returns

6 The **wcsncpy_s** function returns zero[532)] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

7 **EXAMPLE 1** The **wcsncpy_s** function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
/* ... */
wchar_t src1[100] = L"hello";
wchar_t src2[7] = {L'g', L'o', L'o', L'd', L'b', L'y', L'e'};
wchar_t dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
r1 = wcsncpy_s(dst1, 6, src1, 100);
r2 = wcsncpy_s(dst2, 5, src2, 7);
r3 = wcsncpy_s(dst3, 5, src2, 4);
```

The first call will assign to r1 the value zero and to dst1 the sequence of wide characters hello\0.

The second call will assign to r2 a nonzero value and to dst2 the sequence of wide characters \0.

The third call will assign to r3 the value zero and to dst3 the sequence of wide characters good\0.

K.3.9.2.1.3 The wmemcpy_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wmemcpy_s(wchar_t *restrict s1, rsize_t s1max,
     const wchar_t *restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither s1 nor s2 shall be a null pointer. Neither s1max nor n shall be greater than **RSIZE_MAX**/ **sizeof**(**wchar_t**). n shall not be greater than s1max. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, the **wmemcpy_s** function stores zeros in the first s1max wide characters of the object pointed to by s1 if s1 is not a null pointer and s1max is not greater than **RSIZE_MAX**/**sizeof**(**wchar_t**).

Description

4 The **wmemcpy_s** function copies n successive wide characters from the object pointed to by s2 into the object pointed to by s1.

Returns

5 The **wmemcpy_s** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.1.4 The wmemmove_s function

Synopsis

1

[531)]This allows an implementation to copy wide characters from s2 to s1 while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of s1 before discovering that the first element was set to the null wide character.

[532)]A zero return value implies that all the requested wide characters from the string pointed to by s2 fit within the array pointed to by s1 and that the result in s1 is null terminated.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wmemmove_s(wchar_t *s1, rsize_t s1max, const wchar_t *s2, rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. `n` shall not be greater than `s1max`.

3 If there is a runtime-constraint violation, the `wmemmove_s` function stores zeros in the first `s1max` wide characters of the object pointed to by `s1` if `s1` is not a null pointer and `s1max` is not greater than `RSIZE_MAX/sizeof(wchar_t)`.

Description

4 The `wmemmove_s` function copies `n` successive wide characters from the object pointed to by `s2` into the object pointed to by `s1`. This copying takes place as if the `n` wide characters from the object pointed to by `s2` are first copied into a temporary array of `n` wide characters that does not overlap the objects pointed to by `s1` or `s2`, and then the `n` wide characters from the temporary array are copied into the object pointed to by `s1`.

Returns

5 The `wmemmove_s` function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2 Wide string concatenation functions

K.3.9.2.2.1 The `wcscat_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcscat_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2);
```

Runtime-constraints

2 Let m denote the value `s1max - wcsnlen_s(s1, s1max)` upon entry to `wcscat_s`.

3 Neither `s1` nor `s2` shall be a null pointer. `s1max` shall not be greater than `RSIZE_MAX/sizeof(wchar_t)`. `s1max` shall not equal zero. m shall not equal zero.[533] m shall be greater than `wcsnlen_s(s2,` m`)`. Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `wcscat_s` sets `s1[0]` to the null wide character.

Description

5 The `wcscat_s` function appends a copy of the wide string pointed to by `s2` (including the terminating null wide character) to the end of the wide string pointed to by `s1`. The initial wide character from `s2` overwrites the null wide character at the end of `s1`.

6 All elements following the terminating null wide character (if any) written by `wcscat_s` in the array of `s1max` wide characters pointed to by `s1` take unspecified values when `wcscat_s` returns.[534]

[533] Zero means that `s1` was not null terminated upon entry to `wcscat_s`.

[534] This allows an implementation to append wide characters from `s2` to `s1` while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of `s1` before discovering that the first element was set to the null wide character.

Returns

7 The `wcscat_s` function returns zero[535] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2.2 The `wcsncat_s` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcsncat_s(wchar_t * restrict s1, rsize_t s1max,
      const wchar_t * restrict s2, rsize_t n);
```

Runtime-constraints

2 Let m denote the value `s1max - wcsnlen_s(s1, s1max)` upon entry to `wcsncat_s`.

3 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. `s1max` shall not equal zero. m shall not equal zero.[536] If `n` is not less than m, then m shall be greater than `wcsnlen_s`(`s2`, m). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `wcsncat_s` sets `s1[0]` to the null wide character.

Description

5 The `wcsncat_s` function appends not more than `n` successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by `s2` to the end of the wide string pointed to by `s1`. The initial wide character from `s2` overwrites the null wide character at the end of `s1`. If no null wide character was copied from `s2`, then `s1[s1max-` m `+n]` is set to a null wide character.

6 All elements following the terminating null wide character (if any) written by `wcsncat_s` in the array of `s1max` wide characters pointed to by `s1` take unspecified values when `wcsncat_s` returns.[537]

Returns

7 The `wcsncat_s` function returns zero[538] if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

8 **EXAMPLE 1** The `wcsncat_s` function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
/* ... */
wchar_t s1[100] = L"good";
wchar_t s2[6] = L"hello";
wchar_t s3[6] = L"hello";
wchar_t s4[7] = L"abc";
wchar_t s5[1000] = L"bye";
int r1, r2, r3, r4;
r1 = wcsncat_s(s1, 100, s5, 1000);
r2 = wcsncat_s(s2, 6, L"", 1);
```

[535] A zero return value implies that all the requested wide characters from the wide string pointed to by `s2` were appended to the wide string pointed to by `s1` and that the result in `s1` is null terminated.

[536] Zero means that `s1` was not null terminated upon entry to `wcsncat_s`.

[537] This allows an implementation to append wide characters from `s2` to `s1` while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of `s1` before discovering that the first element was set to the null wide character.

[538] A zero return value implies that all the requested wide characters from the wide string pointed to by `s2` were appended to the wide string pointed to by `s1` and that the result in `s1` is null terminated.

```
    r3 = wcsncat_s(s3, 6, L"X", 2);
    r4 = wcsncat_s(s4, 7, L"defghijklmn", 3);
```

After the first call `r1` will have the value zero and `s1` will be the wide character sequence `goodbye\0`.

After the second call `r2` will have the value zero and `s2` will be the wide character sequence `hello\0`.

After the third call `r3` will have a nonzero value and `s3` will be the wide character sequence `\0`.

After the fourth call `r4` will have the value zero and `s4` will be the wide character sequence `abcdef\0`.

K.3.9.2.3 Wide string search functions

K.3.9.2.3.1 The `wcstok_s` function

Synopsis

1
```
    #define __STDC_WANT_LIB_EXT1__ 1
    #include <wchar.h>
    wchar_t *wcstok_s(wchar_t * restrict s1, rsize_t * restrict s1max,
          const wchar_t * restrict s2, wchar_t ** restrict ptr);
```

Runtime-constraints

2 None of `s1max`, `s2`, or `ptr` shall be a null pointer. If `s1` is a null pointer, then `*ptr` shall not be a null pointer. The value of `*s1max` shall not be greater than `RSIZE_MAX/sizeof(wchar_t)`. The end of the token found shall occur within the first `*s1max` wide characters of `s1` for the first call, and shall occur within the first `*s1max` wide characters of where searching resumes on subsequent calls.

3 If there is a runtime-constraint violation, the `wcstok_s` function does not indirect through the `s1` or `s2` pointers, and does not store a value in the object pointed to by `ptr`.

Description

4 A sequence of calls to the `wcstok_s` function breaks the wide string pointed to by `s1` into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by `s2`. The fourth argument points to a caller-provided `wchar_t` pointer into which the `wcstok_s` function stores information necessary for it to continue scanning the same wide string.

5 The first call in a sequence has a non-null first argument and `s1max` points to an object whose value is the number of elements in the wide character array pointed to by the first argument. The first call stores an initial value in the object pointed to by `ptr` and updates the value pointed to by `s1max` to reflect the number of elements that remain in relation to `ptr`. Subsequent calls in the sequence have a null first argument and the objects pointed to by `s1max` and `ptr` are required to have the values stored by the previous call in the sequence, which are then updated. The separator wide string pointed to by `s2` may be different from call to call.

6 The first call in the sequence searches the wide string pointed to by `s1` for the first wide character that is *not* contained in the current separator wide string pointed to by `s2`. If no such wide character is found, then there are no tokens in the wide string pointed to by `s1` and the `wcstok_s` function returns a null pointer. If such a wide character is found, it is the start of the first token.

7 The `wcstok_s` function then searches from there for the first wide character in `s1` that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by `s1`, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

8 In all cases, the `wcstok_s` function stores sufficient information in the pointer pointed to by `ptr` so that subsequent calls, with a null pointer for `s1` and the unmodified pointer value for `ptr`, shall start searching just past the element overwritten by a null wide character (if any).

Returns

9 The `wcstok_s` function returns a pointer to the first wide character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

10 **EXAMPLE**

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;
rsize_t max1 = wcslen(str1)+1;
rsize_t max2 = wcslen(str2)+1;

t = wcstok_s(str1, &max1, "?", &ptr1);    // t points to the token "a"
t = wcstok_s(NULL, &max1, ",", &ptr1);    // t points to the token "??b"
t = wcstok_s(str2, &max2, " \t", &ptr2);  // t is a null pointer
t = wcstok_s(NULL, &max1, "#,", &ptr1);   // t points to the token "c"
t = wcstok_s(NULL, &max1, "?", &ptr1);    // t is a null pointer
```

K.3.9.2.4 Miscellaneous functions

K.3.9.2.4.1 The wcsnlen_s function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```

Description

2 The **wcsnlen_s** function computes the length of the wide string pointed to by s.

Returns

3 If s is a null pointer,[539] then the **wcsnlen_s** function returns zero.

4 Otherwise, the **wcsnlen_s** function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first maxsize wide characters of s then **wcsnlen_s** returns maxsize. At most the first maxsize wide characters of s shall be accessed by **wcsnlen_s**.

K.3.9.3 Extended multibyte/wide character conversion utilities

K.3.9.3.1 Restartable multibyte/wide character conversion functions

1 Unlike **wcrtomb**, **wcrtomb_s** does not permit the ps parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.1.1 The wcrtomb_s function

Synopsis

1
```
#include <wchar.h>
errno_t wcrtomb_s(size_t * restrict retval, char * restrict s, rsize_t smax,
      wchar_t wc, mbstate_t * restrict ps);
```

Runtime-constraints

2 Neither retval nor ps shall be a null pointer. If s is not a null pointer, then smax shall not equal zero and shall not be greater than **RSIZE_MAX**. If s is not a null pointer, then smax shall be not be less than the number of bytes to be stored in the array pointed to by s. If s is a null pointer, then smax shall equal zero.

3 If there is a runtime-constraint violation, then **wcrtomb_s** does the following. If s is not a null pointer and smax is greater than zero and not greater than **RSIZE_MAX**, then **wcrtomb_s** sets s[0] to the null

[539]Note that the **wcsnlen_s** function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated wide string argument make **wcsnlen_s** useful in algorithms that gracefully handle such exceptional data.

character. If `retval` is not a null pointer, then `wcrtomb_s` sets `*retval` to `(size_t)(-1)`.

Description

4 If `s` is a null pointer, the `wcrtomb_s` function is equivalent to the call

```
wcrtomb_s(&retval, buf, sizeof buf, L'\0', ps)
```

where `retval` and `buf` are internal variables of the appropriate types, and the size of `buf` is greater than `MB_CUR_MAX`.

5 If `s` is not a null pointer, the `wcrtomb_s` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most `MB_CUR_MAX` bytes are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

6 If `wc` does not correspond to a valid multibyte character, an encoding error occurs: the `wcrtomb_s` function stores the value `(size_t)(-1)` into `*retval` and the conversion state is unspecified. Otherwise, the `wcrtomb_s` function stores into `*retval` the number of bytes (including any shift sequences) stored in the array pointed to by `s`.

Returns

7 The `wcrtomb_s` function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2 Restartable multibyte/wide string conversion functions

1 Unlike `mbsrtowcs` and `wcsrtombs`, `mbsrtowcs_s` and `wcsrtombs_s` do not permit the `ps` parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.2.1 The `mbsrtowcs_s` function

Synopsis

1
```
#include <wchar.h>
errno_t mbsrtowcs_s(size_t * restrict retval, wchar_t * restrict dst,
     rsize_t dstmax, const char ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
```

Runtime-constraints

2 None of `retval`, `src`, `*src`, or `ps` shall be null pointers. If `dst` is not a null pointer, then neither `len` nor `dstmax` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer, then `dstmax` shall not equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then a null character shall occur within the first `dstmax` multibyte characters of the array pointed to by `*src`.

3 If there is a runtime-constraint violation, then `mbsrtowcs_s` does the following. If `retval` is not a null pointer, then `mbsrtowcs_s` sets `*retval` to `(size_t)(-1)`. If `dst` is not a null pointer and `dstmax` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `mbsrtowcs_s` sets `dst[0]` to the null wide character.

Description

4 The `mbsrtowcs_s` function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by `ps`, from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters are stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.[540] If `dst` is not a null pointer

[540]Thus, the value of `len` is ignored if `dst` is a null pointer.

and no null wide character was stored into the array pointed to by `dst`, then `dst[len]` is set to the null wide character. Each conversion takes place as if by a call to the **`mbrtowc`** function.

5 If `dst` is not a null pointer, the pointer object pointed to by `src` is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if `dst` is not a null pointer, the resulting state described is the initial conversion state.

6 Regardless of whether `dst` is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the **`mbsrtowcs_s`** function stores the value `(size_t)(-1)` into `*retval` and the conversion state is unspecified. Otherwise, the **`mbsrtowcs_s`** function stores into `*retval` the number of multibyte characters successfully converted, not including the terminating null character (if any).

7 All elements following the terminating null wide character (if any) written by **`mbsrtowcs_s`** in the array of `dstmax` wide characters pointed to by `dst` take unspecified values when **`mbsrtowcs_s`** returns.[541)]

8 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

9 The **`mbsrtowcs_s`** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2.2 The `wcsrtombs_s` function

Synopsis

1
```
#include <wchar.h>
errno_t wcsrtombs_s(size_t * restrict retval, char * restrict dst,
     rsize_t dstmax, const wchar_t ** restrict src, rsize_t len,
     mbstate_t * restrict ps);
```

Runtime-constraints

2 None of `retval`, `src`, `*src`, or `ps` shall be null pointers. If `dst` is not a null pointer, then neither `len` shall be greater than **`RSIZE_MAX`**`/sizeof(wchar_t)` nor `dstmax` shall be greater than **`RSIZE_MAX`**. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer, then `dstmax` shall not equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

3 If there is a runtime-constraint violation, then **`wcsrtombs_s`** does the following. If `retval` is not a null pointer, then **`wcsrtombs_s`** sets `*retval` to `(size_t)(-1)`. If `dst` is not a null pointer and `dstmax` is greater than zero and not greater than **`RSIZE_MAX`**, then **`wcsrtombs_s`** sets `dst[0]` to the null character.

Description

4 The **`wcsrtombs_s`** function converts a sequence of wide characters from the array indirectly pointed to by `src` into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by `ps`. If `dst` is not a null pointer, the converted characters are then stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

— when a wide character is reached that does not correspond to a valid multibyte character;

— (if `dst` is not a null pointer) when the next multibyte character would exceed the limit of n total bytes to be stored into the array pointed to by `dst`. If the wide character being converted is the null wide character, then n is the lesser of `len` or `dstmax`. Otherwise, n is the lesser of `len` or `dstmax-1`.

541)This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.

If the conversion stops without converting a null wide character and dst is not a null pointer, then a null character is stored into the array pointed to by dst immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the **wcrtomb** function.[542)]

5 If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

6 Regardless of whether dst is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the **wcsrtombs_s** function stores the value (**size_t**)(-1) into *retval and the conversion state is unspecified. Otherwise, the **wcsrtombs_s** function stores into *retval the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

7 All elements following the terminating null character (if any) written by **wcsrtombs_s** in the array of dstmax elements pointed to by dst take unspecified values when **wcsrtombs_s** returns.[543)]

8 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

9 The **wcsrtombs_s** function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

542)If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.

543)When len is not less than dstmax, the implementation might fill the array before discovering a runtime-constraint violation.

Annex L
(normative)
Analyzability

L.1 Scope

1 This annex specifies optional behavior that can aid in the analyzability of C programs.

2 An implementation that defines `__STDC_ANALYZABLE__` shall conform to the specifications in this annex.[544)]

L.2 Definitions

L.2.1

1 **out-of-bounds store**

an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared **volatile**, fetch) one or more bytes that lie outside the bounds permitted by this document.

L.2.2

1 **bounded undefined behavior**

undefined behavior (3.4.3) that does not perform an out-of-bounds store.

2 **Note 1 to entry:** The behavior might perform a trap.

3 **Note 2 to entry:** Any values produced might be unspecified values, and the representation of objects that are written to might become indeterminate.

L.2.3

1 **critical undefined behavior**

undefined behavior that is not bounded undefined behavior.

2 **Note 1 to entry:** The behavior might perform an out-of-bounds store or perform a trap.

L.3 Requirements

1 If the program performs a trap (3.19.5), the implementation is permitted to invoke a runtime-constraint handler. Any such semantics are implementation-defined.

2 All undefined behavior shall be limited to bounded undefined behavior, except for the following which are permitted to result in critical undefined behavior:

— An object is referred to outside of its lifetime (6.2.4).

— A store is performed to an object that has two incompatible declarations (6.2.7),

— A pointer is used to call a function whose type is not compatible with the referenced type (6.2.7, 6.3.2.3, 6.5.2.2).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— The program attempts to modify a string literal (6.4.5).

— The operand of the unary $*$ operator has an invalid value (6.5.3.2).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary $*$ operator that is evaluated (6.5.6).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

[544)]Implementations that do not define `__STDC_ANALYZABLE__` are not required to conform to these specifications.

— An argument to a function or macro defined in the standard library has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

— The **longjmp** function is called with a **jmp_buf** argument where the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument is nonexistent, or the invocation was from another thread of execution, or the function containing the invocation has terminated execution in the interim, or the invocation was within the scope of an identifier with variably modified type and execution has left that scope in the interim (7.13.2.1).

— The value of a pointer that refers to space deallocated by a call to the free or realloc function is used (7.24.3).

— A string or wide string utility function accesses an array beyond the end of an object (7.26.1, 7.31.4).

Annex M
(informative)
Change History

M.1 Fifth Edition

1 Major changes in this fifth edition (`__STDC_VERSION__` *202311*L) include:

— add new keywords such as **bool**, **static_assert**, **true**, **false**, **thread_local** and others, and allowed implementations to provide macros for the older spelling with a leading underscore followed by a capital letter as well as defining old and new keywords as macros to enable transition of programs easily;

— removed integer width constraints and obsolete sign representations (so-called "1's complement" and "sign-magnitude");

— added a one-argument version of **static_assert**;

— removed support for function definitions with identifier lists;

— mandated function declarations whose parameter list is empty be treated the same as a parameter list which only contain a single **void**;

— harmonization with ISO/IEC 9945 (POSIX):

- extended month name formats for **strftime**
- integration of functions: **gmtime_r**, **localtime_r**, **memccpy**, **strdup**, **strndup**

— harmonization with floating point standard IEC 60559:

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of mathematical functions technical specification TS 18661-4a

— made the **DECIMAL_DIG** macro obsolescent;

— added version test macros to library headers that contained changes to aid in upgrading and portability to be used alongside the **__STDC_VERSION__** macro;

— allowed placement of labels in front of declarations and at the end of compound statement;

— added the attributes feature, which includes the attributes:

- **deprecated**, for marking entities as discouraged for future use;
- **fallthrough**, for explicitly marking cases where falling through in switches or labels is intended rather than accidental;
- **maybe_unused**, for marking entities which may end up not being used;
- **nodiscard**, for marking entities which, when used, should have their value handled in some way by a program;
- **reproducible**, for marking function types for which inputs may always produce predictable output if given the same input (e.g., cached data) but for which the order of such calls still matter;
- **unsequenced**, for marking function types which always produce predictable output and have no dependencies upon other data (and other relevant caveats), and,
- **noreturn**, for indicating a function shall never return;

— added the `u8` character prefix to match the `u8` string prefix;

— mandated all `u8`, `u`, and `U` strings be UTF-8, UTF-16, and UTF-32, respectively, as defined by ISO/IEC 10646;

— separated the literal, wide literal, and UTF-8 literal, UTF-16 literal, and UTF-32 literal encodings for strings and characters and now have a solely execution-based version of these, particularly execution and wide execution encodings;

— added **`mbrtoc8`** and **`c8rtomb`** functions missing from `<uchar.h>`;

— compound literals may also include storage-class specifiers as part of the type to change the lifetime of the compound literal (and possibly turn it into a constant expression)

— added the **`constexpr`** specifier for object definitions and improved what is recognized as a constant expression in conjunction with the **`constexpr`** storage-class specifier;

— added the **`typeof`** and **`typeof_unqual`** operations for deducing the type of an expression;

— improved tag compatibility rules, enabling more types to be compatible with other types;

— added the **`_BitInt`** the bit-precise integer types;

— improved rules for handling enumerations without underlying types, in particular allowing for enumerations without fixed underlying type to have value representations that have a greater range than **`int`**;

— added a new colon-delimited type specifier for enumerations to specify a fixed underlying type (and which, subject to an implementation's definitions governing such constructs, adopt the fixed underlying type's rules for padding, alignment, and sizing within structures and unions as well as with bit-fields);

— added a new header `<stdbit.h>` and a suite of bit and byte-handling utilities for portable access to many implementation's most efficiency functionality;

— modified existing functions to preserve the **`const`**-ness of the type placed into the function;

— added a feature to embed binary data as faithfully as possible with a new preprocessor directive **`#embed`**;

— added a **`nullptr`** constant and a **`nullptr_t`** type with a well-defined underlying representation identical to a pointer to **`void`**;

— added the **`__VA_OPT__`** specifier and clarified language in the handling of macro invocation and arguments;

— mandated support for variably-modified types (but not variable-length arrays themselves);

— ellipses on functions may appear without a preceding parameter in the parameter list of functions and **`va_start`** no longer requires such an argument to be passed to it;

— Unicode identifiers allowed in syntax following Unicode Standard Annex, UAX #31;

— added the **`memset_explicit`** function for making sensitive information inaccessible;

— certain type definitions (i.e., exact-width integer types such as `int128_t`), bit-precise integer types, and extended integer types may exceed the normal boundaries of **`intmax_t`** and **`uintmax_t`** for signed and unsigned integer types, respectively;

— names of functions, macros, and variables in this document, where clarified, are potentially reserved rather than reserved to avoid undefined behavior for a large class of identifiers used by programs existing and to be created;

— mandated support for **`call_once`**;

— allowed **`ptrdiff_t`** to be an integer type of at least 16, rather than requiring an integer type with a width of at least 17;

— added the `__has_include` feature for conditional inclusion expression preprocessor directives to check if a header is available for inclusion;

— changed the type qualifiers of the `_Imaginary_I` and `_Complex_I` macros;

— added `@`, `$`, and ` (backtick) into the source and execution character set;

— enhanced the **auto** type specifier for single object definitions using type inference;

— added the **#elifdef** and **#elifndef** conditional inclusion preprocessor directives;

— added the **#warning** preprocessing directive;

— binary integer literals and appropriate formatting for input/output of binary integer numbers;

— digit separators with `'` (single quotation mark);

— removed conditional support for mixed wide and narrow string literal concatenation;

— added support for additional time bases, as well as `timespec_getres`, in `<time.h>`;

— zero-sized reallocations with `realloc` are undefined behavior;

— added an `unreachable` feature which has undefined behavior if reached during program execution.

M.2 Fourth Edition

1 There were no major changes in the fourth edition (`__STDC_VERSION__` `201710L`), only technical corrections and clarifications.

M.3 Third Edition

1 Major changes in the third edition (`__STDC_VERSION__` `201112L`) included:

— conditional (optional) features (including some that were previously mandatory)

— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread storage (`<stdatomic.h>` and `<threads.h>`)

— additional floating-point characteristic macros (`<float.h>`)

— querying and specifying alignment of objects (`<stdalign.h>`, `<stdlib.h>`)

— Unicode characters and strings (`<uchar.h>`) (originally specified in ISO/IEC TR 19769:2004)

— type-generic expressions

— static assertions

— anonymous structures and unions

— no-return functions

— macros to create complex numbers (`<complex.h>`)

— support for opening files for exclusive access

— removed the `gets` function (`<stdio.h>`)

— added the `aligned_alloc`, `at_quick_exit`, and `quick_exit` functions (`<stdlib.h>`)

— (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731–1:2007)

— (conditional) support for analyzability

M.4 Second Edition

1 Major changes in the second edition (`__STDC_VERSION__` `199901L`) included:

- restricted character set support via digraphs and `<iso646.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)
- wide character library support in `<wchar.h>` and `<wctype.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)
- more precise aliasing rules via effective type
- restricted pointers
- variable length arrays
- flexible array members
- **`static`** and type qualifiers in parameter array declarators
- complex (and imaginary) support in `<complex.h>`
- type-generic math macros in `<tgmath.h>`
- the **`long long int`** type and library functions
- extended integer types
- increased minimum translation limits
- additional floating-point characteristics in `<float.h>`
- remove implicit **`int`**
- reliable integer division
- universal character names (`\u` and `\U`)
- extended identifiers
- hexadecimal floating constants and `%a` and `%A` **`printf`**/**`scanf`** conversion specifiers
- compound literals
- designated initializers
- `//` comments
- specified width integer types and corresponding library functions in `<inttypes.h>` and `<stdint.h>`
- remove implicit function declaration
- preprocessor arithmetic done in **`intmax_t`**/**`uintmax_t`**
- mixed declarations and statements
- new block scopes for selection and iteration statements
- integer constant type rules
- integer promotion rules
- macros with a variable number of arguments (**`__VA_ARGS__`**)
- the **`vscanf`** family of functions in `<stdio.h>` and `<wchar.h>`
- additional math library functions in `<math.h>`

- treatment of error conditions by math library functions (**math_errhandling**)
- floating-point environment access in <fenv.h>
- IEC 60559 (also known as IEC 559 or IEEE 754 arithmetic) support
- trailing comma allowed in **enum** declaration
- %lf conversion specifier allowed in **printf**
- inline functions
- the **snprintf** family of functions in <stdio.h>
- boolean type in <stdbool.h>
- idempotent type qualifiers
- empty macro arguments
- new structure type compatibility rules (tag compatibility)
- additional predefined macro names
- **_Pragma** preprocessing operator
- standard pragmas
- **__func__** predefined identifier
- **va_copy** macro
- additional **strftime** conversion specifiers
- LIA compatibility annex
- deprecate **ungetc** at the beginning of a binary file
- remove deprecation of aliased array parameters
- conversion of array to pointer not limited to lvalues
- relaxed constraints on aggregate and union initialization
- relaxed restrictions on portable header names
- **return** without expression not permitted in function that returns a value (and vice versa)

M.5 First Edition, Amendment 1

1 Major changes in the amendment to the first edition (**__STDC_VERSION__** 199409L) included:

- addition of the predefined **__STDC_VERSION__** macro
- restricted character set support via digraphs and <iso646.h>
- wide character library support in <wchar.h> and <wctype.h>

Bibliography

[1] ISO/IEC 646:1991, *Information technology* — ISO *7-bit coded character set for information interchange*.

[2] ISO/IEC 9945–2:1993, *Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities*.

[3] ISO/IEC TR 10176:2003, *Information technology — Guidelines for the preparation of programming language standards*.

[4] ISO/IEC 10967–1:2012, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*.

[5] ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*.

[6] ISO/IEC TR 24731–1:2007, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*.

[7] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems*.

[8] ISO/IEC 60559:2011, *Floating-point arithmetic*.

[9] IEEE 754-1985 *IEEE Standard for Binary Floating-Point Arithmetic*.

[10] IEEE 754-2019 *IEEE Standard for Floating-Point Arithmetic*.

[11] ANSI/IEEE 854–1987, *American National Standard for Radix-Independent Floating-Point Arithmetic*.

[12] ANSI X3/TR–1–82 (1982), *American National Dictionary for Information Processing Systems*, Information Processing Systems Technical Report.

[13] "The C Reference Manual" by Dennis M. Ritchie, a version of which was published in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978). Copyright owned by AT&T.

[14] *1984 /usr/group Standard* by the /usr/group Standards Committee, Santa Clara, California, USA, November 1984.

[15] The Unicode Consortium. *Unicode Standard Annex, UAX #31, Unicode Identifier and Pattern Syntax [online]*. Edited by Mark Davis. Revision 33; issued for Unicode 13.0.0. 2020-02-13 [viewed 2020-05-27]. Available at `https://www.unicode.org/reports/tr31/tr31-33.html`.

Index