N3181

TECHNICAL SPECIFICATION

ISO/IEC TS 18661-5

Second edition
CFP Working Draft
2023-11-13

Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 5: Supplementary attributes

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — Extensions à virgule flottante pour C —

Partie 5: Attributs supplémentaires

Reference number
ISO/IEC TS 18661-5:20yy(E)

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office Case postale 56 • CH-1211 Geneva 20 Tel. + 41 22 749 01 11 Fax + 41 22 749 09 47 E-mail copyright@iso.org Web www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members 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.

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).

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).

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

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments, and system software interfaces*.

ISO/IEC TS 18661 originally consisted of the following parts, under the general title *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C*:

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplementary functions*

— *Part 5: Supplementary attributes*

Parts 1, 2, 3, and some of Part 4 are integrated into ISO/IEC 9899:2024 (C23).

ISO/IEC TS 18661 Part 4 Version 2, a separate document, supersedes ISO/IEC TS 18661-4:2015, the previous version of Part 4.

ISO/IEC TS 18661 Part 5 Version 2, this document, supersedes ISO/IEC TS 18661-5:2015, the previous version of Part 5.

Introduction

Background

IEC 60559 floating-point standard

The IEC 60559 international standard and the corresponding version of the IEEE 754 standard have equivalent content.

Floating-point standards – matching versions

IEEE 754-1985	IEC 60559:1989
IEEE 754-2008	ISO/IEC/IEEE 60559:2011
IEEE 754-2019	ISO/IEC 60559:2020

The IEEE 754-1985 standard for binary floating-point arithmetic was motivated by an expanding diversity in floating-point data representation and arithmetic, which made writing robust programs, debugging, and moving programs between systems exceedingly difficult. Now the great majority of systems provide data formats and arithmetic operations according to this standard. The stated goals of this standard were (and have remained throughout its revisions) the following, quoted from IEEE 754-1985[1], Introduction:

1 Facilitate movement of existing programs from diverse computers to those that adhere to this standard.

2 Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists.

3 Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.

4 Provide direct support for

   a. Execution-time diagnosis of anomalies

   b. Smoother handling of exceptions

   c. Interval arithmetic at a reasonable cost

5 Provide for development of

   a. Standard elementary functions such as exp and cos

   b. Very high precision (multiword) arithmetic

   c. Coupling of numerical and symbolic algebraic computation

6 Enable rather than preclude further refinements and extensions.

To these ends, the standard specified a floating-point model comprising the following:

— *formats* – for binary floating-point data, including representations for Not-a-Number (NaN) and signed infinities and zeros;

— *operations* – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system; also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations;

— *context* – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods.

The IEEE 754-2008 standard for floating-point arithmetic, which is equivalent to the ISO/IEC/IEEE 60559:2011 international standard, was a major revision. This revision:

— Specified more formats, including decimal as well as binary. It added a 128-bit binary format to its basic formats. It defined extended formats corresponding to all its basic formats. It specified data interchange formats (which may or may not be arithmetic), including a 16-bit binary format and an unbounded sequence of wider formats. To conform to the floating-point standard, an implementation must provide at least one of the basic formats, along with the required operations.

— Specified more operations. It added required operations including (among others) arithmetic operations that round their result to a narrower format than the operands (with just one rounding), more conversions with integer types, more classifications and comparisons, and more operations for managing flags and modes. It added recommended operations including an extensive set of mathematical operations and seven reduction operations for sums and scaled products.

— Placed more emphasis on reproducible results. This is reflected in its standardization of more operations. For the most part, it completely specified behaviors. It required conversions between floating-point formats and decimal character sequences to be correctly rounded for at least three more decimal digits than is necessary to distinguish all numbers in the widest supported binary format; it completely specified such conversions involving any number of decimal digits. It specified the recommended transcendental functions to be correctly rounded.

— Added a way to specify a constant rounding direction for a static portion of code, with details left to programming language standards. This feature potentially allows rounding control without incurring the overhead of runtime access to a global (or thread) rounding mode.

— Added other recommended features including alternate methods for exception handling, controls for expression evaluation (allowing or disallowing various optimizations), support for fully reproducible results, and support for program debugging.

The IEEE 754-2019 standard for floating-point arithmetic, which is equivalent to the ISO/IEC 60559:2020 international standard, was a minor revision. As such it was limited to upward-compatible editorial corrections and clarifications and minor enhancements. It added some recommended operations, including ones that might be required features in the next revision.

IEC 60559 (like IEEE 754) defines specific encodings for the exchange of floating-point data between different implementations. However, it does not define the concrete representation (specific layout in storage, or in a processor's register, for example) of data or context.

IEC 60559 (like IEEE 754) does not specify how its features are expressed in programming languages. However, its revisions have added guidance for programming language standards, recognizing that benefits of the floating-point standard, even if well supported in the hardware, are not available to users

unless the programming language provides interfaces for the features and reliable behaviors. The implementation's combination of both hardware and software determines conformance to the floating-point standard.

C support for IEC 60559

The C standard specifies floating-point arithmetic using an abstract model. The representation of a floating-point number is specified in a form where the constituent components (sign, exponent, significand) of the representation are defined but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation-defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of most floating-point operations are also implementation-defined or unspecified, including accuracy and aspects of the way special values and exceptional conditions are handled.

The reason for this approach is historical. At the time when C was first standardized, before the floating-point standard was established, there were various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of the model would have made most of the existing implementations at the time non-conforming.

Beginning with ISO/IEC 9899:1999 (C99), C has included an optional second level of specification for implementations supporting the floating-point standard. C99, in conditionally normative annex F, introduced nearly complete support for the IEC 60559:1989 standard for binary floating-point arithmetic. Also, C99's informative annex G offered a specification of complex arithmetic that is compatible with IEC 60559:1989.

ISO/IEC 9899:2011 (C11) and ISO/IEC 9899:2018 (C17) include refinements to the C99 floating-point specification, though are still based on IEC 60559:1989. C11 upgraded annex G from "informative" to "conditionally normative".

ISO/IEC TR 24732:2009 introduced partial C support for the decimal floating-point arithmetic in ISO/IEC/IEEE 60559:2011. ISO/IEC TR 24732, for which technical content was completed while IEEE 754-2008 was still in the later stages of development, specifies decimal types based on ISO/IEC/IEEE 60559:2011 decimal formats, though it does not include all the operations required by ISO/IEC/IEEE 60559:2011.

ISO/IEC TS 18661 provided a C language binding for ISO/IEC/IEEE 60559:2011, based on the C11 standard. ISO/IEC TS 18661 was organized into five parts:

ISO/IEC TS 18661-1:2014 – Binary floating-point arithmetic

ISO/IEC TS 18661-2:2015 – Decimal floating-point arithmetic, Second edition

ISO/IEC TS 18661-3:2015 – Interchange and extended types

ISO/IEC TS 18661-4:2015 – Supplementary functions

ISO/IEC TS 18661-5:2016 – Supplementary attributes

ISO/IEC 9899:2024 (C23) incorporates ISO/IEC TS 18661 Parts 1 and 2, the mathematical functions in Part 4, and, in an annex, Part 3. C23 also updates its floating-point specification to support ISO/IEC/IEEE 60559:2020.

A separate document updates ISO/IEC TS 18661-4. It retains the feature that was not incorporated into C23, namely the reduction functions. It also adds support for the augmented arithmetic operations introduced in IEC 60559:2020.

This document updates ISO/IEC TS 18661-5, which was not incorporated into C23.

Additional background on supplementary attributes

IEC 60559 defines alternatives for certain attributes of floating-point semantics and intends that programming languages provide means by which a program can specify which of the alternative semantics apply to a given block of code. The program specification of attributes is to be constant (fixed at translation time), not dynamic (changeable at execution time). These attributes are recommended (but not required) by IEC 60559. They are not supported in C23.

IEC 60559 recommends attributes for

— alternate exception handling: methods of handling floating-point exceptions

— preferredWidth: evaluation formats for floating-point operations

— value-changing optimizations: allow/disallow program transformations that might affect floating-point result values

— reproducibility: support for getting floating-point result values and exceptions that are exactly reproducible on other systems

This document provides these attributes by means of standard pragmas, where the pragma parameters represent the alternative semantics. The pragmas are similar in form to the floating-point pragmas (**`FENV_ACCESS`**, **`FP_CONTRACT`**, **`CX_LIMITED_RANGE`**) that have been in C since 1999.

The **`FENV_ROUND`** and **`FENV_DEC_ROUND`** pragmas in C23 provide the rounding direction attributes required by IEC 60559.

Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 5: Supplementary attributes

1 Scope

This document specifies extensions to programming language C to include pragmas corresponding to attributes specified and recommended in ISO/IEC 60559 but not supported in C23.

2 Conformance

An implementation may conform to any or all of four feature sets in this document. It so conforms if

a) it meets the requirements for a conforming implementation of C23;

b) it defines **__STDC_IEC_60559_BFP__** or **__STDC_IEC_60559_DFP__** or both, indicating support for IEC 60559 binary or decimal floating-point arithmetic, as specified in C23 Annex F;

and one or more of the following are true:

c) it defines **__STDC_IEC_60559_ATTRIB_EVALUATION_FORMAT__** to **20yymmL** and provides the features for evaluation formats as specified in this document (Clause 7);

d) it defines **__STDC_IEC_60559_ATTRIB_OPTIMIZATION__** to **20yymmL** and provides the features for optimization as specified in this document (Clause 8);

e) it defines **__STDC_IEC_60559_ATTRIB_REPRODUCIBLE__** to **20yymmL** and provides the features for reproducibility as specified in this document (Clause 9);

f) it defines **__STDC_IEC_60559_ATTRIB_ALTERNATE_EXCEPTION_HANDLING__** to **20yymmL** and provides the features for alternate exception handling as specified in this document (Clause 10).

3 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9899:2024, *Information technology — Programming languages — C*

ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-point arithmetic*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2024, ISO/IEC 60559:2020, and the following apply.

4.1

C23
standard ISO/IEC 9899:2024, *Information technology — Programming languages — C*

5 C standard conformance

5.1 Freestanding implementations

C23 Clause 4 allows freestanding implementations to conform to this technical specification.

5.2 Predefined macros

The implementation defines one or more of the following macros to indicate conformance to the specification in this document for support of the corresponding attributes specified and recommended in IEC 60559.

__STDC_IEC_60559_ATTRIB_EVALUATION_FORMAT__ The integer constant **20yymmL**.

__STDC_IEC_60559_ATTRIB_OPTIMIZATION__ The integer constant **20yymmL**.

__STDC_IEC_60559_ATTRIB_REPRODUCIBLE__ The integer constant **20yymmL**.

__STDC_IEC_60559_ATTRIB_ALTERNATE_EXCEPTION_HANDLING__ The integer constant **20yymmL**.

5.3 Standard headers

The identifiers specified in this document are defined or declared by the associated header if and only if the implementation defines the relevant feature macros (5.2) and

__STDC_WANT_IEC_60559_ATTRIB_EXT__

is defined as a macro at the point in the source file where the header is first included.

6 Standard pragmas

C23 provides standard pragmas (C23 6.10.7) for specifying certain attributes pertaining to floating-point behavior within a compound statement or file. This document extends this practice by introducing additional standard pragmas to support attributes recommended by IEC 60559:

#pragma STDC FP_FLT_EVAL_METHOD *width*
#pragma STDC FP_DEC_EVAL_METHOD *width*
#pragma STDC FP_ALLOW_VALUE_CHANGING_OPTIMIZATION *on-off-switch*
#pragma STDC FP_ALLOW_ASSOCIATIVE_LAW *on-off-switch*
#pragma STDC FP_ALLOW_DISTRIBUTIVE_LAW *on-off-switch*
#pragma STDC FP_ALLOW_MULTIPLY_BY_RECIPROCAL *on-off-switch*
#pragma STDC FP_ALLOW_ZERO_SUBNORMAL *on-off-switch*
#pragma STDC FP_ALLOW_CONTRACT_FMA *on-off-switch*

#pragma STDC FP_ALLOW_CONTRACT_OPERATION_CONVERSION *on-off-switch*
#pragma STDC FP_ALLOW_CONTRACT *on-off-switch*
#pragma STDC FP_REPRODUCIBLE *on-off-switch*
#pragma STDC FENV_EXCEPT *action except-list*

width: specified with the pragmas (7.1, 7.2)

on-off-switch: specified in C23 6.10.7

action, *except-list*: specified with the pragma (10.1)

7 Evaluation formats

This clause applies to implementations that define:

__STDC_IEC_60559_ATTRIB_EVALUATION_FORMAT__

C23 gives implementations the flexibility to evaluate operations to the format of the wider operand or to a still wider evaluation format. The values of the macros **FLT_EVAL_METHOD** (C23 5.2.4.2.2, H.3) and **DEC_EVAL_METHOD** (C23 5.2.4.2.3, H.3) characterize these evaluation methods. Though C23 does not provide means for the user to control the evaluation method, some implementations provide such controls as extensions. IEC 60559 recommends an attribute for this purpose. The following subclauses (7.1 and 7.2) specify pragmas in **<math.h>** to control the evaluation method. These evaluation method pragmas, like the **FENV_ROUND** (C23 7.6.2) and **FENV_DEC_ROUND** (C23 7.6.3) pragmas, affect translation-time expression evaluation and constants.

NOTE As specified in C23 6.7.1, the value of the initializer for an object declared with storage-class specifier **constexpr** is constrained to be exactly representable in the target type. Thus, an evaluation method pragma whose scope includes the declaration might affect the validity of the declaration.

7.1 Evaluation method pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_FLT_EVAL_METHOD width
```

Constraints

The *width* parameter shall be **-1**, **0**, **DEFAULT**, or another value supported by the implementation.

Description

The **FP_FLT_EVAL_METHOD** pragma sets the evaluation method for standard floating types and for binary interchange and extended floating types to the evaluation method represented by *width*. The parameter *width* is an expression in one of the forms

0
decimal-constant
- *decimal-constant*
DEFAULT

where the value of the expression is a possible value of the **FLT_EVAL_METHOD** macro, as specified in C23 5.2.4.2.2 and H.3. An expression represents the evaluation method corresponding to its value (C23 5.2.4.2.2, H.3) and **DEFAULT** designates the implementation's default evaluation method (characterized by the **FLT_EVAL_METHOD** macro). The *width* parameter may be **-1**, **0**, or **DEFAULT**. Which, if any, other values of *width* are supported is implementation-defined. 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 **FP_FLT_EVAL_METHOD** 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_FLT_EVAL_METHOD** 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.

7.2 Evaluation method pragma for decimal floating types

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_DEC_EVAL_METHOD width
```

Constraints

The *width* parameter shall be **-1**, **1**, **DEFAULT**, or another value supported by the implementation.

Description

The **FP_DEC_EVAL_METHOD** pragma sets the evaluation method for decimal interchange and extended floating types to the evaluation method represented by *width*. The parameter *width* is an expression in one of the forms

0
decimal-constant
– *decimal-constant*
DEFAULT

where the value of the expression is a possible value of the **DEC_EVAL_METHOD** macro, as specified in C23 5.2.4.2.3 and H.3. An expression represents the evaluation method corresponding to its value (C23 5.2.4.2.3, H.3) and **DEFAULT** designates the implementation's default evaluation method (characterized by the **DEC_EVAL_METHOD** macro). The *width* parameter may be **-1**, **1**, or **DEFAULT**. Which, if any, other values of *width* are supported is implementation-defined. 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 **FP_DEC_EVAL_METHOD** 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_DEC_EVAL_METHOD** 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.

7.3 Effective evaluation method macros

The **<float.h>** macros **FLT_EVAL_METHOD** (C23 5.2.4.2.2, H.3) and **DEC_EVAL_METHOD** (C23 5.2.4.2.3, H.3) characterize the default evaluation method. Their values are constant expressions, suitable for use in conditional expression inclusion preprocessing directives. They are not affected by the evaluation method pragmas, thus might not reflect the effective evaluation method.

The **<math.h>** header defines macros **FLT_EVAL_METHOD_EFFECTIVE** and **DEC_EVAL_METHOD_EFFECTIVE** that are similar to the **<float.h>** macros **FLT_EVAL_METHOD** and **DEC_EVAL_METHOD**, except that they characterize the effective evaluation method at the point in the program where the macro is used. Thus, they reflect the state of any evaluation method pragmas (7.1, 7.2) that are in effect. These macros shall not be used in conditional expression inclusion preprocessing directives.

7.4 Evaluation type macros

The **<math.h>** types with an **_t** suffix (for example, **float_t)** (C23 7.12 and H.11), which are defined to match evaluation formats, reflect the evaluation method where no evaluation method pragma is in effect. For each of these types, there is a type-like macro in **<math.h>** with the same name which expands to a designation for the type whose range and precision are used for evaluating operations and constants of the corresponding standard, binary, or decimal floating type. The macro reflects the actual evaluation method, which might be determined by an evaluation method pragma. Use of **#undef** to remove the macro definition will ensure that the actual type is referred to (as though no evaluation method pragma was in effect).

7.5 Evaluation formats for <tgmath.h>

The evaluation methods in C23 apply to floating-point operators, but not to math functions. Hence, they do not apply to the IEC 60559 operations that are provided as library functions. This subclause specifies a macro the user can define to cause the generic macros in **<tgmath.h>** to be evaluated like floating-point operators.

Except for functions that round result to a narrower type, if the macro

__STDC_TGMATH_OPERATOR_EVALUATION__

is defined at the point in the program where **<tgmath.h>** is first included, the format of the generic parameters of the function invoked by a type-generic macro is the evaluation format determined by the effective evaluation method (see C23 5.2.4.2.2, 5.2.4.2.3, H.3) applied to the types of the arguments for generic parameters. The semantic type of the expanded type-generic macro is as determined by the rules in C23 7.27 and H.13 and is unchanged by the evaluation method. Neither the arguments for generic parameters nor the result are narrowed to their semantic types. Thus, (if the macro **__STDC_TGMATH_OPERATOR_EVALUATION__** is appropriately defined) the evaluation method affects the operations provided by type-generic macros and floating-point operators in the same way. See **EXAMPLE** below.

The macro **__STDC_TGMATH_OPERATOR_EVALUATION__** does not alter the conversion of classification macro arguments to their semantic types (as specified in C23 7.12.3).

EXAMPLE The following code uses wide evaluation to avoid overflow and underflow.

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#define __STDC_TGMATH_OPERATOR_EVALUATION__
#include <tgmath.h>
{
   #pragma STDC FLT_EVAL_METHOD 1 /* to double */
   float x, y, z;
   …
   z = sqrt(x * x + y * y);
}
```

Because of the evaluation method pragma, the sum of squares, whose semantic type is **float**, is evaluated with the range and precision of **double**, hence does not overflow or underflow. The expanded **<tgmath.h>** macro **sqrt** acquires the semantic type of its argument: **float**. However, because the macro **__STDC_TGMATH_OPERATOR_EVALUATION__** is defined before the inclusion of **<tgmath.h>**, the **sqrt** macro behaves like an operator with respect to the evaluation method and does not narrow its argument to its semantic type. Without the definition of the macro **__STDC_TGMATH_OPERATOR_EVALUATION__**, the **sqrt** macro would expand to **sqrtf** and its evaluated argument would be converted to **float**, which might overflow or underflow.

8 Optimization controls

This clause applies to implementations that define:

__STDC_IEC_60559_ATTRIB_OPTIMIZATION__

IEC 60559 recommends attributes to allow and disallow value-changing optimizations, individually and collectively. C23 Annex F disallows value-changing optimizations, except for contractions (which can be controlled as a group with the **FP_CONTRACT** pragma). This clause provides pragmas to allow or disallow certain value-changing optimizations, including those mentioned in IEC 60559.

The pragmas in this clause can be used to allow the implementation to do certain floating-point optimizations that are generally disallowed because the optimization might change values of floating-point expressions. These pragmas apply to all floating types. It is unspecified whether optimizations allowed by these pragmas occur consistently, or at all. These pragmas (among other standard pragmas) apply to user code. They do not apply to code for operators or library functions that might be placed inline by the implementation.

Some of the pragmas allow optimizations based on identities of real number arithmetic that are not valid for floating-point arithmetic (C23 5.1.2.3, F.9.2). Optimizations based on identities that are valid for the implementation's floating-point arithmetic are always allowed. Optimizations based on identities derived from identities whose use is allowed (either by a standard pragma or by virtue of being valid for the implementation's floating-point arithmetic) may also be done.

These pragmas do not affect the requirements on volatile or atomic variables.

Each 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, on each optimization it controls, from its occurrence until another pragma that affects the same optimization is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect, on each optimization it controls, from its occurrence until another pragma that affects the same optimization 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 allowing each optimization controlled by the pragma is restored to its condition just before the compound statement.

8.1 The `FP_ALLOW_VALUE_CHANGING_OPTIMIZATION` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_VALUE_CHANGING_OPTIMIZATION on-off-switch
```

Description

This pragma is equivalent to all the optimization pragmas specified below, with the same value of *on-off-switch* (**ON**, **OFF**, or **DEFAULT**).

NOTE The **FP_ALLOW_VALUE_CHANGING_OPTIMIZATION** pragma does not affect the evaluation methods. Nevertheless, an evaluation method characterized by a negative value of *width* (C23 5.2.4.2.2, 5.2.4.2.3, H.3) might allow for indeterminable evaluation formats, hence unspecified result values.

8.2 The `FP_ALLOW_ASSOCIATIVE_LAW` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_ASSOCIATIVE_LAW on-off-switch
```

Description

This pragma allows or disallows optimizations based on the associative laws for addition and multiplication

$$x + (y + z) = (x + y) + z$$
$$x \times (y \times z) = (x \times y) \times z$$

where *on-off-switch* is one of

ON – allow application of the associative laws

OFF – do not allow application of the associative laws

DEFAULT – "off"

Note that this pragma allows optimizations based on similar mathematical identities involving subtraction and division. For example, for IEC 60559 floating-point arithmetic, since the identity

$$x - y = x + (-y)$$

is valid (C23 F.9.2), this pragma also allows optimizations based on

$$x + (y - z) = (x + y) - z$$

Similarly, if the states for this pragma and the **FP_ALLOW_MULTIPLY_BY_RECIPROCAL** pragma (8.4) are both "on", then optimizations based on the following are allowed:

$x \times (y / z) = (x \times y) / z$

Note also that for IEC 60559 floating-point arithmetic, since the commutative laws

$x + y = y + x$
$x \times y = y \times x$

are valid, the pragma allows optimizations based on identities derived from the associative and commutative laws, such as

$x + (z + y) = (x + y) + z$

8.3 The FP_ALLOW_DISTRIBUTIVE_LAW pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_DISTRIBUTIVE_LAW on-off-switch
```

Description

This pragma allows or disallows optimizations based on the distributive laws for multiplication and division

$x \times (y + z) = (x \times y) + (x \times z)$
$x \times (y - z) = (x \times y) - (x \times z)$
$(x + y) / z = (x / z) + (y / z)$
$(x - y) / z = (x / z) - (y / z)$

where *on-off-switch* is one of

ON – allow application of the distributive laws

OFF – do not allow application of the distributive laws

DEFAULT – "off"

8.4 The FP_ALLOW_MULTIPLY_BY_RECIPROCAL pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_MULTIPLY_BY_RECIPROCAL on-off-switch
```

Description

This pragma allows or disallows optimizations based on the mathematical equivalence of division and multiplication by the reciprocal of the denominator

$x / y = x \times (1 / y)$

where *on-off-switch* is one of

ON – allow multiply by reciprocal

OFF – do not allow multiply by reciprocal

DEFAULT – "off"

8.5 The `FP_ALLOW_ZERO_SUBNORMAL` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_ZERO_SUBNORMAL on-off-switch
```

Description

This pragma allows or disallows replacement of subnormal operands and results by zero, where *on-off-switch* is one of

ON – allow replacement of subnormals with zero

OFF – do not allow replacement of subnormals with zero

DEFAULT – "off"

Within the scope of this pragma**,** the floating-point operations affected by the pragma are all floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic type to its semantic type, as done by classification macros), and invocations of applicable functions in **<math.h>**, **<stdio.h>**, **<stdlib.h>**, and **<wchar.h>** for which macro replacement has not been suppressed (C23 7.1.4). Thus, subnormal operands and results of affected operations may be replaced by zero. Whether the replacement raises the "inexact" and "underflow" floating-point exceptions is unspecified. Functions not affected by the pragma behave as though no **FP_ALLOW_ZERO_SUBNORMAL** pragma were in effect at the site of the call.

8.6 The `FP_ALLOW_CONTRACT_FMA` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_CONTRACT_FMA on-off-switch
```

Description

This pragma allows or disallows contraction (C23 6.5) of floating-point multiply and add or subtract (with the result of the multiply)

$x * y + z$
$x * y - z$
$x + y * z$
$x - y * z$

where *on-off-switch* is one of

ON – allow contraction for floating-point multiply-add

OFF – do not allow contraction for floating-point multiply-add

DEFAULT – implementation defined whether "on" or "off"

NOTE IEC 60559 uses the term *synthesize* instead of *contract*.

8.7 The `FP_ALLOW_CONTRACT_OPERATION_CONVERSION` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_CONTRACT_OPERATION_CONVERSION on-off-switch
```

Description

This pragma allows or disallows contraction (C23 6.5) of a floating-point operation and a conversion (of the result of the operation), where *on-off-switch* is one of

ON – allow contraction for floating-point operation-conversion

OFF – do not allow contraction for floating-point operation-conversion

DEFAULT – implementation defined whether "on" or "off"

Within the scope of this pragma**,** the floating-point operations affected by the pragma are all floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic type to its semantic type, as done by classification macros), and invocations of applicable functions in **<math.h>**, **<stdio.h>**, **<stdlib.h>**, and **<wchar.h>** for which macro replacement has not been suppressed (C23 7.1.4). Thus, an affected operation may be contracted with a conversion of its result. Functions not affected by the pragma behave as though no **FP_ALLOW_CONTRACT_OPERATION_CONVERSION** pragma were in effect at the site of the call.

EXAMPLE For the code sequence

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_CONTRACT_OPERATION_CONVERSION ON
float f1, f2;
double d1, d2;
...
f1 = d1 * d2;
f2 = sqrt(d1);
```

the multiply (operation) and assignment (conversion) are allowed to be evaluated with just one rounding (to the range and precision of **float**). If the *on-off-switch* for the pragma were **OFF**, then the multiply would have to be rounded according to the evaluation method and the assignment would require a second rounding. With the given code, the **sqrt** function may be replaced by **fsqrt**, avoiding the need for a separate operation to convert the **double** result of **sqrt** to **float**.

8.8 The FP_ALLOW_CONTRACT pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_ALLOW_CONTRACT on-off-switch
```

Description

This pragma allows or disallows contraction (C23 6.5) for floating-point operations, where *on-off-switch* is one of

ON – allow contraction for floating-point operations

OFF – do not allow contraction for floating-point operations

DEFAULT – implementation defined whether “on” or “off”

The optimizations controlled by this pragma include those controlled by the **FP_ALLOW_CONTRACT_FMA** and **FP_ALLOW_CONTRACT_OPERATION_CONVERSION** pragmas.

This pragma is equivalent to the **FP_CONTRACT** pragma (C23 7.12.2), also in **<math.h>**: the two pragmas may be used interchangeably, provided the implementation defines **__STDC_WANT_IEC_60559_ATTRIBS_EXT__**.

9 Reproducibility

This clause applies to implementations that define:

__STDC_IEC_60559_ATTRIB_REPRODUCIBLE__

IEC 60559 recommends an attribute to facilitate writing programs whose floating-point results and exception flags will be reproducible on any implementation that supports the language and library features used by the program. Such code must use only those features of the language and library that support reproducible results. These features include ones with a well-defined binding to reproducible features of IEC 60559, so that no unspecified or implementation-defined behavior is admitted.

This clause provides a pragma to support the IEC 60559 attribute for reproducible results and gives requirements for programs to have reproducible results. Where the state of the pragma is "on", floating-point numerical results and exception flags are reproducible (given the same inputs, including relevant environment variables) on implementations that define

```
__STDC_IEC_60559_ATTRIB_REPRODUCIBLE__
```

and that support the language and library features used by the source code, provided the source code uses a limited set of features as described below (9.2).

An implementation that defines **__STDC_IEC_60559_ATTRIB_REPRODUCIBLE__** also defines either **__STDC_IEC_60559_BFP__** or **__STDC_IEC_60559_DFP__**, or both. If the implementation defines **__STDC_IEC_60559_BFP__**, it supports reproducible results for code using (binary) types **float** and **double**. If the implementation defines **__STDC_IEC_60559_DFP__**, it supports reproducible results for code using types **_Decimal32**, **_Decimal64**, and **_Decimal128**. If the implementation defines **__STDC_IEC_60559_TYPES__**, then it supports reproducible results for code using its interchange floating types (C23 H.2.1). If the implementation provides a set of correctly rounded math functions (C23 7.33.8), then it supports reproducible results for code using correctly rounded math functions from that set.

9.1 The FP_REPRODUCIBLE pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <math.h>
#pragma STDC FP_REPRODUCIBLE on-off-switch
```

Description

This pragma enables or disables support for reproducible results. 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 **FP_REPRODUCIBLE** 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_REPRODUCIBLE** 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 the state of the pragma is "on", then the effects of the following are implied

```
#pragma STDC FENV_ACCESS ON
#pragma STDC FP_ALLOW_VALUE_CHANGING_OPTIMIZATION OFF
```

and if **__STDC_IEC_60559_BFP__** is defined

```
#pragma STDC FP_FLT_EVAL_METHOD 0
```

and if **__STDC_IEC_60559_DFP__** is defined

```
#pragma STDC FP_DEC_EVAL_METHOD 1
```

If the **FP_REPRODUCIBLE** pragma appears with the *on-off-switch* **OFF** under the effect of a **FP_REPRODUCIBLE** pragma with *on-off-switch* **ON**, then the states of the **FENV_ACCESS** pragma, the

value-changing optimization pragmas, and the evaluation method pragmas (even an evaluation method pragma whose state was explicitly changed under the effect of the pragma with *on-off-switch* **ON**) revert to their states prior to the **FP_REPRODUCIBLE** pragma with on-off-switch **ON**. The **FP_REPRODUCIBLE** pragma with *on-off-switch* **OFF** has no effect if it occurs where the state of the pragma is "off".

The default state of the pragma is "off".

Recommended practice
The implementation is encouraged to issue a diagnostic message if, where the state of the **FP_REPRODUCIBLE** pragma is "on", the source code uses a language or library feature whose results may not be reproducible.

9.2 Reproducible code

The following properties support code sequences in producing reproducible results.[1]

— The code is under the effect of the **FP_REPRODUCIBLE** pragma (with state "on").

— All floating-point operations used by the code are bound to IEC 60559 operations, as described in C23 F.3 in the table entitled "Operation binding".

— The code does not contain any use that may result in undefined behavior. The code does not depend on any behavior that is unspecified, implementation-defined, or locale-specific.

The restrictive properties below are examples, not a complete list. See also C23 Annex J. Although the properties may not be necessary in all cases for reproducible code, the user is advised to follow the restrictions to avoid common programming practices that would undermine reproducibility.

— The code does not use the **long double** type.

— The code does not use complex or imaginary types.

— If **__STDC_IEC_60559_BFP__** is not defined by the implementation, the code does not use the **float** or **double** types.

— Even if **__STDC_IEC_60559_TYPES__** is defined, the code does not use extended floating types. (Even if **__STDC_IEC_60559_TYPES__** is defined, some interchange floating types are optional features.)

— The code does not depend on the payloads (C23 F.10.13) or sign bits of quiet NaNs.

— The code does not use signaling NaNs.

— The code does not depend on conversions between binary floating types and character sequences with more than $M + 3$ significant decimal digits, where M is 17 if **__STDC_IEC_60559_TYPES__** is not defined (by the implementation), and M is $1 + \lceil p \times \log_{10}(2) \rceil$, where p is the precision of the widest supported binary interchange floating type, if **__STDC_IEC_60559_TYPES__** is defined. Even if

[1] Of course, if the code uses optional features, results will be reproducible only on implementations that support those features.

__STDC_IEC_60559_TYPES__ is defined, support for interchange floating types wider than binary64 is an optional feature. (This specification differs from IEC 60559 which specifies that an implementation supporting reproducibility shall not limit the number of significant decimal digits for correct rounding.)

- The code does not depend on the actual character sequence in **printf** results with style **a** (or **A**), nor does it depend on numerical values of such results when the precision is not sufficient for an exact representation.
- The code does not depend on the quantum of a result for the decimal maximum and minimum functions in C23 7.12.12 when the arguments are equal.
- The code does not use the **remquo** functions.
- The code does not set the state of any pragma that allows value-changing optimizations to "on" or "default".
- The code does not set the state of the **FENV_ACCESS** pragma to "off" or "default".
- The code does not use the **FP_FLT_EVAL_METHOD** pragma with any *width* except 0 or 1. (Support for *width* equal to 1 is an optional feature.)
- The code does not use the **FP_DEC_EVAL_METHOD** pragma with any *width* except 1 or 2. (Support for *width* equal to 2 is an optional feature.)
- The code does not use an **FENV_EXCEPT** pragma (10.1) with an *action* **OPTIONAL_FLAG**, **BREAK**, **TRY**, or **CATCH**.
- The code does not depend on the "underflow" or "inexact" floating-point exceptions or flags.

10 Alternate exception handling

This clause applies to implementations that define:

__STDC_IEC_60559_ATTRIB_ALTERNATE_EXCEPTION_HANDLING__

IEC 60559 arithmetic raises floating-point exceptions to inform the program when an operation encounters problematic inputs, such that no one result would be suitable for all situations. The default exception handling in IEC 60559 is intended to be more useful in more situations than other schemes, or at least predictable. However, other exception handling is more useful in certain situations. Thus, IEC 60559 describes alternate exception handling and recommends that programming languages provide means for the program to specify which exception handling will be done.

When a floating-point exception is raised, the IEC 60559 default exception handling sets the appropriate exception flag(s), returns a specified result, and continues execution. IEC 60559 also prescribes alternate exception handling. The pragma in this clause provides a means for the program to choose the method of exception handling. The pragma applies to operations on all floating types.

For the "underflow" exception, the chosen exception handling occurs if the exception is raised, whether the default result would be exact or inexact, unless stated otherwise.

10.1 The `FENV_EXCEPT` pragma

Synopsis

```
#define __STDC_WANT_IEC_60559_ATTRIBS_EXT__
#include <fenv.h>
#pragma STDC FENV_EXCEPT action except-list
```

Description

The **FENV_EXCEPT** pragma sets the method specified by *action* for handling the exceptions represented by *except-list*.

except-list shall be a comma-separated list of distinct supported exception designations (or one supported exception designation). The supported exception designations shall include the exception macro identifiers (C23 7.6)

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
```

The **<fenv.h>** header should define macros for the following sub-exceptions and may define additional macros with the appropriate prefix (**FE_INVALID_** or **FE_DIVBYZERO_**) for other sub-exceptions. The supported exception designations shall include the defined sub-exception macro identifiers (if any). If defined, the macros expand to integer constant expressions. Sub-exceptions corresponding to defined macros occur as specified below, and not in other cases.

— "invalid" floating-point exceptions from add and subtract operators and functions that add or subtract (C23 7.12.14.1, 7.12.14.2, F.10.11), not caused by signaling NaN input
FE_INVALID_ADD

— "invalid" floating-point exceptions from divide operators and functions that divide (C23 7.12.14.4, F.10.11), not caused by signaling NaN input
FE_INVALID_DIV

— "invalid" floating-point exceptions from functions that compute multiply-add (C23 7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11) and from contracted multiply and add operators, not caused by signaling NaN input
FE_INVALID_FMA

— "invalid" floating-point exceptions from conversions from floating to integer types (C23 F.4), not caused by signaling NaN input
FE_INVALID_INT

— "invalid" floating-point exceptions from **ilogb** and **llogb** functions (C23 F.10.3.8, F.10.3.10), not caused by signaling NaN input
FE_INVALID_ILOGB

— "invalid" floating-point exceptions from multiply operators and functions that multiply (C23 7.12.14.3, F.10.11), not caused by signaling NaN input
FE_INVALID_MUL

— "invalid" floating-point exceptions from the **quantized***N* functions (C23 7.12.15.1), not caused by signaling NaN input
FE_INVALID_QUANTIZE

— "invalid" floating-point exceptions from the **remainder** and **remquo** functions (C23 F.10.7.2, F.10.7.3), not caused by signaling NaN input
FE_INVALID_REM

— "invalid" floating-point exceptions from functions that compute square root or reciprocal of square root (C23 F.10.4.9, F.10.4.10, 7.12.14.6, F.10.11), not caused by signaling NaN input
FE_INVALID_SQRT

— "invalid" floating-point exceptions caused by signaling NaN input (C23 F.2.1)
FE_INVALID_SNAN

— "invalid" floating-point exceptions from relational operators and comparison macros (C23 6.5.8, 7.12.17, F.10.14.1), not caused by signaling NaN input
FE_INVALID_UNORDERED

— "divide-by-zero" floating-point exceptions from divide operators and functions that divide (C23 7.12.14.4, F.10.11)
FE_DIVBYZERO_ZERO

— "divide-by-zero" floating-point exceptions from logarithm and **logb** functions (C23 F.10.3.11, F.10.3.12, F.10.3.13, F.10.3.14, F.10.3.15, F.10.3.16, F.10.3.17)
FE_DIVBYZERO_LOG

action shall be a designation of a supported exception handling method. The following actions shall be provided:

— default exception handling (as specified in IEC 60559).
DEFAULT

— default exception handling, but without setting the flag.
NO_FLAG

— default exception handling, but whether the flag is set (as with default exception handling), and for which operations and their occurrences, is unspecified.
OPTIONAL_FLAG

— *abrupt underflow*. If an "underflow" floating-point exception occurs (see IEC 60559), the operation delivers a result with magnitude zero or the minimum normal magnitude (for the result format) and with the same sign as the default result, sets the "underflow" floating-point exception flag, and raises the "inexact" floating-point exception. When

rounding to nearest, ties to even
rounding to nearest, ties away from zero

or

rounding toward zero

the result magnitude is zero. When rounding toward positive infinity, the result magnitude is the minimum normal magnitude if the result sign is positive, and zero if the result sign is negative. When rounding toward negative infinity, the result magnitude is the minimum

normal magnitude if the result sign is negative, and zero if the result sign is positive. Abrupt underflow has no effect on the interpretation of subnormal operands. The *action* has no effect if **FE_UNDERFLOW** is not included in *except-list*.

ABRUPT_UNDERFLOW

With one of the actions in the list above, the pragma shall occur either outside external declarations, or preceding all explicit declarations and statements inside a compound statement, which then is the compound statement associated with the pragma. When outside external declarations, the pragma action for a designated exception takes effect from the occurrence of the pragma until another **FENV_EXCEPT** pragma designating the same exception is encountered, or until the end of the translation unit. When inside a compound statement, the pragma action for a designated exception takes effect from the occurrence of the pragma until another **FENV_EXCEPT** pragma designating the same exception 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 handling each exception designated in *except-list* is restored to its condition just before the compound statement.

The actions in the lists below in this subclause affect flow of control. With one of these actions, the pragma shall precede a compound statement, which is the compound statement associated with the pragma. There shall be nothing between the pragma and its associated compound statement except perhaps white space (including comments). For a designated exception, the pragma takes effect from the beginning of the associated compound statement until another **FENV_EXCEPT** pragma designating the same exception is encountered (with a nested associated compound statement), or until the end of the compound statement. At the end of a compound statement the state for handling each exception designated in *except-list* is restored to its condition just before the compound statement.

— *break*. Terminate execution of the compound statement associated with the pragma. Then, continue execution after the associated compound statement. When termination occurs, the following apply: if the execution to completion of the associated compound statement (without the break) would at any point modify an object, the value of the object is indeterminate; if the execution would modify the state of the dynamic rounding mode or any state maintained by the standard library (e.g. in the I/O system), the state is unspecified; the values of flags for the designated exceptions are unspecified. (Thus, termination may occur as soon as possible after the exception is raised, to maximize performance.)

BREAK

The following two actions work together. A compound statement associated with a *try* action shall be paired with one or more compound statements each associated with a *catch* action. The pragmas with *catch* actions and their associated compound statements shall appear contiguously immediately below the compound statement associated with the *try* action, except for white space (including comments). Each exception designation in the pragma with a *try* action shall appear in one and only one of the pragmas with a *catch* action.

— *try*. The designated exceptions may be handled by a *catch* action. It is unspecified whether flags for designated exceptions that are set in the execution of the associated compound statement are restored to their states before the associated compound statement. The associated compound statement shall not be the *statement* of a selection (C23 6.8.4) or iteration (C23 6.8.5) statement.[2] There shall be no jumps into or out of the associated compound statement, other than to handle an exception, as specified below.

TRY

— *catch*. If any designated exception occurs in the execution of the compound statement associated with the *try* action, jump to a compound statement associated with some *catch* action with an occurring designated exception. Upon completion of the associated compound statement, continue execution after the last of the compound statements associated with *catch* actions. The jump target should be a compound statement associated with the first occurring designated exception. When the jump occurs, the following apply: if the execution of the associated compound statement to completion (without the jump) would at any point modify an object, the value of the object is indeterminate; if the execution would modify the state of the dynamic rounding mode or any state maintained by the standard library (e.g., in the I/O system), the state is unspecified. (Thus, the jump may occur as soon as possible after the exception is raised, to maximize performance). The compound statement associated with a *catch* action is executed only to handle an exception occurring in the compound statement associated with the *try* action. There shall be no other jumps into or out of the compound statement associated with a *catch* action.

CATCH

The following two actions work together. A compound statement associated with a *delayed-try* action shall be paired with one or more compound statements each associated with a *delayed-catch* action. The pragmas with *delayed-catch* actions and their associated compound statements shall appear contiguously immediately below the compound statement associated with the *delayed-try* action, except for white space (including comments). Each exception designation in the pragma with a *delayed-try* action shall appear in one and only one of the pragmas with a *delayed-catch* action. For supported sub-exceptions, the behavior of the actions listed below shall be as if the exceptions, flags, and functions in the specification were extended for sub-exceptions, though such extensions are not prescribed in this document.

— *delayed-try*. The designated exceptions may be handled by a *delayed-catch* action. Before executing the compound statement associated with the *delayed-try* action, save (as by **fegetexceptflag**) the states of the flags for the designated exceptions, and then clear (as by **feclearexcept**) the designated exceptions. After normal completion of the

[2] The compound statements associated with a *try* action and its *catch* actions (or with a *delayed-try* action and its *delayed-catch* actions), together enclosed in braces, may be the *statement* of a selection or iteration statement. For example, the following code segment is permitted:

```
for (int i = 0; i < LEN; i++)  {
   #pragma STDC FENV_EXCEPT TRY FE_OVERFLOW
   {
      y[i] = x[i] * x[i];
   }
   #pragma STDC FENV_EXCEPT CATCH FE_OVERFLOW
   {
      y[i] = DBL_MAX;
   }
}
```

associated compound statement, re-save the states of the designated exceptions. Then restore (as by **fesetexceptflag**) the designated exception flag states before the associated compound statement. The associated compound statement shall not be the *statement* of a selection (C23 6.8.4) or iteration (C23 6.8.5) statement. There shall be no jumps into or out of the associated compound statement.

DELAYED_TRY

— *delayed-catch*. Test (as by **fetestexceptflag**) the exception flag states saved after completion of the compound statement associated with the *delayed-try* action. If any exception with the same designation for the *delayed-try* action and a *delayed-catch* action occurred (as determined by flag state tests), jump to the first compound statement associated with an occurring exception with the same designation for the *delayed-try* action and a *delayed-catch* action. Upon completion of the associated compound statement, continue execution after the last of the compound statements associated with *delayed-catch* actions. Each exception designation shall be listed in at most one of the pragmas with a *delayed-catch* action. The compound statement associated with a *delayed-catch* action is executed only to handle an exception occurring in the compound statement associated with the *delayed-try* action. There shall be no other jumps into or out of the compound statement associated with a *delayed-catch* action.

DELAYED_CATCH

Within the scope of an **FENV_EXCEPT** pragma**,** the floating-point operations affected by the pragma are all floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic type to its semantic type, as done by classification macros), and invocations of applicable functions in **<math.h>**, **<stdio.h>**, **<stdlib.h>**, and **<wchar.h>** for which macro replacement has not been suppressed (7.1.4). Thus, exceptions raised by affected operations are handled according to the specified *action*. Functions not affected by the pragma behave as though no **FENV_EXCEPT** pragma were in effect at the site of the call.

Behavior is undefined if non-default **SIGFPE** handling is set on entry to code with a non-default **FENV_EXCEPT** pragma, or if code within such a scope uses the **signal** function, uses **raise(SIGFPE)**, or explicitly accesses the flag of a designated exception.

EXAMPLE 1 This example illustrates differences between *try* and *catch* actions and *delayed-try* and *delayed-catch* actions.

Code sequence with *try* and *catch* actions:

```
#pragma STDC FENV_ACCESS ON
#include <fenv.h>
#define LEN 2
double d[LEN];
float f[LEN];
…
#pragma STDC FENV_EXCEPT TRY FE_DIVBYZERO, FE_OVERFLOW
{
   for (int i=0; i<LEN; i++) {
      f[i] = 1.0 / d[i];
   }
}
#pragma STDC FENV_EXCEPT CATCH FE_DIVBYZERO
{
   printf("divide-by-zero\n");
}
#pragma STDC FENV_EXCEPT CATCH FE_OVERFLOW
{
   printf("overflow\n");
}
…
```

The same code but with *delayed-try* and *delayed-catch* actions:

```
#pragma STDC FENV_ACCESS ON
#include <fenv.h>
#define LEN 2
double d[LEN];
float f[LEN];
…
#pragma STDC FENV_EXCEPT DELAYED_TRY FE_DIVBYZERO, FE_OVERFLOW
{
   for (int i=0; i<LEN; i++) {
      f[i] = 1.0 / d[i];
   }
}
#pragma STDC FENV_EXCEPT DELAYED_CATCH FE_DIVBYZERO
{
   printf("divide-by-zero\n");
}
#pragma STDC FENV_EXCEPT DELAYED_CATCH FE_OVERFLOW
{
   printf("overflow\n");
}
…
```

The following table shows examples of inputs and results for the two code sequences above.

	try - catch	*delayed-try – delayed-catch*
Input **d**	0.5, 0.0	
Results		
f = **1**/**d**	indeterminate, indeterminate	2.0, infinity
output	"**overflow**"	"**divide-by-zero**"
"divide-by-zero" flag	unspecified (set or restored)	restored
"overflow" flag	unchanged	restored (unchanged)
Input **d**	0.5, 1e−100	
Results		
f = **1**/**d**	indeterminate, indeterminate	2.0, infinity
output	"**overflow**"	"**overflow**"
"divide-by-zero" flag	unchanged	restored (unchanged)
"overflow" flag	unspecified (set or restored)	restored
Input **d**	1e−100, 0.0	
Results		
f = **1**/**d**	indeterminate, indeterminate	infinity, infinity
output	"**overflow**" (recommended) or "**divide-by-zero**"	"**divide-by-zero**"
"divide-by-zero" flag	unspecified (set or restored)	restored
"overflow" flag	unspecified (set or restored)	restored

NOTE The *delayed-try* and *delayed-catch* actions are deterministic. They can be implemented with the floating-point exception flags. The following code sequence is equivalent to the code sequence using *delayed-try* and *delayed-catch* in the example above.

```
#pragma STDC FENV_ACCESS ON
#include <fenv.h>
#define LEN 2
double d[LEN];
float f[LEN];
...
{
   fexcept_t old_except, new_except;
   fegetexceptflag(&old_except, FE_DIVBYZERO | FE_OVERFLOW);
   feclearexcept(FE_DIVBYZERO | FE_OVERFLOW);
   {
      for (int i=0; i<LEN; i++) {
         f[i] = 1.0 / d[i];
      }
   }
   fegetexceptflag(&new_except, FE_DIVBYZERO | FE_OVERFLOW);
   fesetexceptflag(&old_except, FE_DIVBYZERO | FE_OVERFLOW);
```

```
      if (fetestexceptflag(&new_except, FE_DIVBYZERO)) {
         printf("divide-by-zero\n");
      }
      else if (fetestexceptflag(&new_except, FE_OVERFLOW)) {
         printf("overflow\n");
      }
   }
   …
```

NOTE The *try* and *catch* actions are not deterministic (see example above), which allows more implementation flexibility for better performance.

In most cases, the *try* and *catch* actions can be implemented like *delayed-try* and *delayed-catch* actions, though not for the "underflow" exception (which occurs without causing the "underflow" flag to be set, in cases of exact subnormal results). Such implementation would not always handle the first occurring designated exception, as recommended.

An implementation of *try* and *catch* actions using floating-point exception traps might well be able to handle the first occurring designated exception (including "underflow"), as recommended, and achieve better performance.

EXAMPLE 2 The following code sequence uses overlapping exception designations:

```
#pragma STDC FENV_ACCESS ON
#include <fenv.h>
double x, y;
…
#pragma STDC FENV_EXCEPT DELAYED_TRY FE_INVALID
{
   #pragma STDC FENV_EXCEPT TRY FE_INVALID_DIV
   {
      y = sin(x) / x;
   }
   #pragma STDC FENV_EXCEPT CATCH FE_INVALID_DIV
   {
      y = 1.0;
   }
}
#pragma STDC FENV_EXCEPT DELAYED_CATCH FE_INVALID
{
   printf("invalid\n");
}
…
```

The inner try/catch overrides the handling in the outer delayed-try/delayed-catch for an "invalid divide" but does not affect the handling of other "invalid" exceptions. Thus an "invalid divide", which can occur here only for **x** = 0, gives **y** the value 1.0, without printing "**invalid**". Other invalid exceptions, which can occur if **x** is infinite or a signaling NaN, cause "**invalid**" to be printed (after **y** is given a NaN value).

Note that if **DELAYED_TRY** and **DELAYED_CATCH** were changed to **TRY** and **CATCH**, an "invalid" exception other than "invalid divide" might cause a jump out of the inner *try* statement resulting in undefined behavior.

Bibliography

[1] IEEE 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*

[2] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

[3] IEEE 754-2019, *IEEE Standard for Floating-Point Arithmetic*

[4] IEEE 854-1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*

[5] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*

[6] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*

[7] ISO/IEC TR 24732:2009, *Information technology — Programming languages, their environments, and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic*

[8] ISO/IEC TS 18661-1:2014, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C* — Binary floating-point arithmetic

[9] ISO/IEC TS 18661-2:2015, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C* — Decimal floating-point arithmetic, Second edition

[10] ISO/IEC TS 18661-3:2015, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C* — Interchange and extended types

[11] ISO/IEC TS 18661-4:2015, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C* — Supplementary functions

[12] ISO/IEC TS 18661-5:2016, *Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C* — Supplementary attributes

[13] ISO/IEC 9899:1999, *Information technology — Programming languages — C*, Second edition

[14] ISO/IEC 9899:2011/Cor.1:2012, *Information technology — Programming languages — C / Technical Corrigendum 1*, Third edition

[15] ISO/IEC 9899:2018, *Information technology — Programming languages — C*