Revise spelling of keywords v3
proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Over time C has integrated some new features as keywords (some genuine, some from C++) but the naming strategy has not been entirely consistent: some were integrated using non-reserved names (const, inline) others were integrated in an underscore-capitalized form. For some of them, the use of the lower-case form then is ensured via a set of library header files. The reason for this complicated mechanism had been backwards compatibility for existing code bases. Since now years or even decades have gone by, we think that it is time to switch and to use the primary spelling.

This is a revision of papers to N2368 and N2392 where we reduce the focus to the list of keywords that found consensus in the WG14 London 2019 meeting. Other papers will build on this for those keywords or features that need more investigation.

Changes in v3:
— Remove the requirement for implementations to have these keywords as macro names and adapt title and contents accordingly.
— Update Annex B.

1. INTRODUCTION

Several keywords in current C2x have weird spellings as reserved names that have ensured backwards compatibility for existing code bases:

_alignas _bool _Decimal32 _Imaginary
_alignof _Complex _Decimal64 _Noreturn _Thread_local
_atomic _Decimal128 _Generic _Static_assert

Many of them have alternative spellings that are provided through special library headers:

alignas bool imaginary static_assert
alignof complex noreturn thread_local

In addition, several important constants or language constructs are provided through headers and have not achieved the status of first class language constructs:

NULL _Imaginary_I
_Complex_I false offsetof
_true

The use of these different keywords make C code often more difficult or unpleasant to read, and always need special care for code that is sought to be included in both languages, C and C++. For all of the features it will be ten years since their introduction when C2x comes out, a time that should be sufficient for all users of the identifiers to have upgraded to a non-conflicting form.

Some of the constructs mentioned above have their own specificities and need more coordination with WG21 and C++. E.g a common mechanism is currently sought for the derived type mechanisms for _Complex and _Atomic, or a keyword like _Noreturn might even be replaced by means of the attribute mechanism that has recently been voted into C2x.

This paper repropose those keywords of N2368 that found direct consensus in WG14, in the expectation that the thus proposed modifications can be integrated directly into C2x.
Other proposals will follow that will tackle other parts of N2368 and beyond:

— Modify `false` and `true` to be of type `bool`.
— Make `noreturn` a keyword or replace it by an attribute.
— Introduce `nullptr` and deprecate `NULL`.
— Make `complex` and `imaginary` keywords and/or provide `__complex(T)` and `__imaginary(T)` constructs for interoperability with C++.
— Make `atomic` (or `__atomic`) a keyword that resolves to the specifier form of `_Atomic(T)`.
— Replace `_Complex_I` and `_Imaginary_I` by first-class language constructs.
— Make `offsetof` a keyword.
— Make `generic` a keyword that replaces `_Generic`.
— Make `decimal32`, `decimal64` and `decimal128` (or `dec32`, `dec64` and `dec128`) keywords that replace `_Decimal32`, `_Decimal64` and `_Decimal128`.

2. PROPOSED MECHANISM OF INTEGRATION

Many code bases use in fact the underscore-capitalized form of the keywords and not the compatible ones that are provided by the library headers. Therefore we need a mechanism that makes a final transition to the new keywords seamless. We propose the following:

— Allow for the keywords to also be macros, such that implementations may have an easy transition.
— Don’t allow user code to change such macros.
— Allow the keywords to result in other spellings when they are expanded in with `#` or `##` operators.
— Keep the alternative spelling with underscore-capitalized identifiers around for a while.

With this in mind, implementing these new keywords is in fact almost trivial for any implementation that is conforming to C17.

— 7 predefined macros have to be added to the startup mechanism of the translator. They should expand to similar tokens as had been defined in the corresponding library headers.
— If some of the macros are distinct to their previous definition, the library headers have to be amended with `#ifndef` tests. Otherwise, the equivalent macro definition in a header should not harm.

Needless to say that on the long run, it would be good if implementations would switch to full support as keywords, but there is no rush, and some implementations that have no need for C++ compatibility might never do this.

3. PREDEFINED CONSTANTS

Predefined constants need a little bit more effort for the integration, because up to now C did not have named constants on the level of the language. We propose to integrate these constants by means of a new syntax term `predefined constant`.

For this proposal we only include `false` and `true`. Other proposals will follow for `nullptr` and maybe `_Complex_I` and `_Imaginary_I`.

3.1. Boolean constants

The Boolean constants `false` and `true` are a bit ambivalent because in C17 they expand to integer constants 0 and 1 that have type `int` and not `bool`. This is unfortunate when they are used as arguments to type-generic macros, because there they could trigger an unexpected expansion, namely for `int` instead of `bool`.

alignas  bool  thread_local  true
alignof  static_assert  false
Nevertheless, **int** is the type that is currently used for them, so in this consensus paper we propose to stay with this. A follow-up paper will propose to change the type to **bool**.

4. FEATURE TESTS

As additional effect of having the keywords to be macros, too, the macros **bool** and **thread_local** (and eventual future **complex** or **atomic**) can be used as feature tests that are independent of library support and of the inclusion of the corresponding header.

5. REFERENCE IMPLEMENTATION

To add minimal support for the proposed changes, an implementation would have to add definitions that are equivalent to the following lines to their startup code:

```c
#define alignas _Alignas
#define alignof _Alignof
#define bool _Bool
#define false 0
#define static_assert _Static_assert
#define thread_local _Thread_local
#define true 1
```

At the other end of the spectrum, an implementation that implements all new keywords as first-class constructs and also wants to provide them as macros (though they don’t have to) can simply have definitions that are the token identity:

```c
#define alignas alignas
#define alignof alignof
#define bool bool
#define false false
#define static_assert static_assert
#define thread_local thread_local
#define true true
```

6. MODIFICATIONS TO THE STANDARD TEXT

This proposal implies a large number of trivial modifications in the text, namely simple text processing that replaces the occurrence of one of the deprecated keywords by its new version. These modifications are not by themselves interesting and are not included in the following. WG14 members are invited to inspect them on the VC system, if they want, they are in the branch “keywords”.

The following appendix lists the non-trivial changes:

— Changes to the “Keywords” clause 6.4.1, where we replace the keywords themselves (p1) and add provisions to have the new ones as macro names (p2) and establish the old keywords as alternative spellings (p4).
— Addition of a new clause 6.4.4.5 “Predefined constants” that implement the constants **false** and **true**, and that is anchored in 6.4.4 “Constants”.
— A new subclause to 6.10.8.4 “Optional macros” that lists the new keywords that may also be macros.
— Modifications of the corresponding library clauses (7.2, 7.15, 7.18, and 7.26).
— Mark `<stdalign.h>` and `<stdbool.h>` to be obsolescent inside their specific text and in clause 7.13 “Future library directions”.
— Update Annex A.
— Update Annex B.
Appendix: pages with diffmarks of the proposed changes against the September 2019 working draft.
The following page numbers are from the particular snapshot and may vary once the changes are integrated.
6.4.1 Keywords

Syntax

1 keyword: one of

alignas extern sizeof _Alignas_
alignof false static _Aligntof_ auto bool for struct _Atomic_
break goto switch _Complex_
case if thread_local _Decimal128_
char inline true _Decimal32_
const int typedef _Generic_
continue long union _Decimal64_
default register unsigned _Noreturn_
do restrict void _Static_assert_
double return volatile _Thread_local_
else short while
enum signed _Alignof_

Constraints

2 The keywords

alignas bool static_assert true
alignof false thread_local

may optionally be predefined macro names (6.10.8.4). None of these shall be the subject of a _
#define or a _undef preprocessing directive and their spelling inside expressions that are subject to the # and ## preprocessing operators is unspecified. 

Semantics

3 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.

4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.

<table>
<thead>
<tr>
<th>keyword</th>
<th>alternative spelling</th>
</tr>
</thead>
<tbody>
<tr>
<td>alignas</td>
<td><em>Alignas</em></td>
</tr>
<tr>
<td>alignof</td>
<td><em>Alignof</em></td>
</tr>
<tr>
<td>bool</td>
<td><em>Bool</em></td>
</tr>
<tr>
<td>static_assert</td>
<td><em>Static_assert</em></td>
</tr>
<tr>
<td>thread_local</td>
<td><em>Thread_local</em></td>
</tr>
</tbody>
</table>

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 identifier:

identifier-nondigit
identifier identifier-nondigit
identifier digit

---

74) The intent of these specifications is to allow but not to force the implementation of the corresponding feature by means of a predefined macro.

75) One possible specification for imaginary types appears in Annex G.

76) These alternative keywords are obsolescent features and should not be used for new code.
6.4.4 Constants

Syntax

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

Constraints

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

Semantics

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

6.4.4.1 Integer constants

Syntax

integer-constant:
decimal-constant integer-suffix opt
octal-constant integer-suffix opt
hexadecimal-constant integer-suffix opt

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
0
octal-constant octal-digit

hexadecimal-constant:
hexadecimal-prefix hexadecimal-digit
hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of
0x 0X

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

integer-suffix:
unsigned-suffix long-suffix opt
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix opt
long-long-suffix unsigned-suffix opt
Forward references: common definitions `<stddef.h>` (7.19), the `mbtowc` function (7.22.7.2), Unicode utilities `<uchar.h>` (7.28).

6.4.4.5 Predefined constants

Syntax

```plaintext
predefined-constant:
    false
    true
```

Description

Some keywords represent constants of a specific value and type.

6.4.4.5.1 The `false` and `true` constants

Description

The keywords `false` and `true` represent constants of type `int` that are suitable for use as are integer literals. Their values are 0 for `false` and 1 for `true`.86)

6.4.5 String literals

Syntax

```plaintext
string-literal: 
    encoding-prefix_opt " s-char-sequence_opt "

encoding-prefix: 
    u8
    u
    U
    L

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 A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

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 wide string literal is the same, except prefixed by the letter `L`, `u`, or `U`.

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 \”.

---

86) Thus, the keywords `false` and `true` are usable in preprocessor directives.
An implementation that defines `__STDC_NO_COMPLEX__` shall not define `__STDC_IEC_60559_COMPLEX__` or `__STDC_IEC_559_COMPLEX__`.

### 6.10.8.4 Optional macros

The keywords

```
alignas    bool    static_assert    true
alignof    false   thread_local
```

optionally are also predefined macro names that expand to unspecified tokens.

### 6.10.9 Pragma operator

#### Semantics

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.

#### 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)
```
7.2 Diagnostics `<assert.h>`

The header `<assert.h>` defines the `assert` and `static_assert` 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(ignore) (((void)0)
```

The `assert` macro is redefined according to the current state of `NDEBUG` each time that `<assert.h>` is included.

The `assert` macro shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The macro expands to `__Static_assert`.

7.2.1 Program diagnostics

7.2.1.1 The `assert` macro

Synopsis
```
#include <assert.h>
void assert(scalar expression);
```

Description

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. It then calls the `abort` function.

Returns

The `assert` macro returns no value.

Forward references: the `abort` function (7.22.4.1).

207) The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```
7.15 Alignment `<stdalign.h>`

The obsolescent header `<stdalign.h>` defines two macros that are suitable for use in `#if` preprocessing directives. They are

```c
__alignas_is_defined
```

and

```c
__alignof_is_defined
```

which both expand to `true`. 
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**

```c
#include <stdatomic.h>

:::_Bool atomic_is_lock_free(const volatile A *obj);
```

**Description**

2 The **atomic_is_lock_free** generic function indicates whether or not 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.\(^{280}\)

### 7.17.6 Atomic integer types

1 For each line in the following table,\(^{281}\) the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.\(^{282}\)

<table>
<thead>
<tr>
<th>Atomic type name</th>
<th>Direct type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>atomic_bool</strong></td>
<td>_Atomic _Bool</td>
</tr>
<tr>
<td><strong>atomic_char</strong></td>
<td>_Atomic char</td>
</tr>
<tr>
<td><strong>atomic_schar</strong></td>
<td>_Atomic signed char</td>
</tr>
<tr>
<td><strong>atomic_uchar</strong></td>
<td>_Atomic unsigned char</td>
</tr>
<tr>
<td><strong>atomic_short</strong></td>
<td>_Atomic short</td>
</tr>
<tr>
<td><strong>atomic_ushort</strong></td>
<td>_Atomic unsigned short</td>
</tr>
<tr>
<td><strong>atomic_int</strong></td>
<td>_Atomic int</td>
</tr>
<tr>
<td><strong>atomic_uint</strong></td>
<td>_Atomic unsigned int</td>
</tr>
<tr>
<td><strong>atomic_long</strong></td>
<td>_Atomic long</td>
</tr>
<tr>
<td><strong>atomic_ulong</strong></td>
<td>_Atomic unsigned long</td>
</tr>
<tr>
<td><strong>atomic_llong</strong></td>
<td>_Atomic unsigned long long</td>
</tr>
<tr>
<td><strong>atomic_ullong</strong></td>
<td>_Atomic unsigned long long</td>
</tr>
<tr>
<td><strong>atomic_char16_t</strong></td>
<td>_Atomic char16_t</td>
</tr>
<tr>
<td><strong>atomic_char32_t</strong></td>
<td>_Atomic char32_t</td>
</tr>
<tr>
<td><strong>atomic_wchar_t</strong></td>
<td>_Atomic wchar_t</td>
</tr>
<tr>
<td><strong>atomic_int_least8_t</strong></td>
<td>_Atomic int_least8_t</td>
</tr>
<tr>
<td><strong>atomic_uint_least8_t</strong></td>
<td>_Atomic uint_least8_t</td>
</tr>
</tbody>
</table>

\(^{280}\) *obj* can be a null pointer.

\(^{281}\) See “future library directions” (7.31.11).

\(^{282}\) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.
7.26 Threads <threads.h>

7.26.1 Introduction

The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.\(^{339}\) Implementations that define the macro \texttt{__STDC_NO_THREADS__} need not provide this header nor support any of its facilities.

The header <threads.h> also includes <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.

The macros are

\begin{verbatim}
ONCE_FLAG_INIT
\end{verbatim}

which expands to a value that can be used to initialize an object of type \texttt{once_flag}; and

\begin{verbatim}
TSS_DTOR_ITERATIONS
\end{verbatim}

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

The types are

\begin{verbatim}
cnd_t
\end{verbatim}

which is a complete object type that holds an identifier for a condition variable;

\begin{verbatim}
thrd_t
\end{verbatim}

which is a complete object type that holds an identifier for a thread;

\begin{verbatim}
tss_t
\end{verbatim}

which is a complete object type that holds an identifier for a thread-specific storage pointer;

\begin{verbatim}
mtx_t
\end{verbatim}

which is a complete object type that holds an identifier for a mutex;

\begin{verbatim}
tss_dtor_t
\end{verbatim}

which is the function pointer type \texttt{void (*)(void*)}, used for a destructor for a thread-specific storage pointer;

\begin{verbatim}
thrd_start_t
\end{verbatim}

which is the function pointer type \texttt{int (*)(void*)} that is passed to \texttt{thrd_create} to create a new thread; and

\begin{verbatim}
once_flag
\end{verbatim}

which is a complete object type that holds a flag for use by \texttt{call_once}.

The enumeration constants are

\begin{verbatim}
mtx_plain
\end{verbatim}

which is passed to \texttt{mtx_init} to create a mutex object that does not support timeout;

\begin{verbatim}
mtx_recursive
\end{verbatim}

\(^{339}\)See "future library directions" (7.31.18).
and the same names suffixed with f, l, d32, d64, or d128 may be added to the <math.h> header. The cr prefix is intended to indicate a correctly rounded version of the function.

### 7.31.9 Signal handling <signal.h>

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.

### 7.31.10 Alignment <stdalign.h>

The <stdalign.h> header together with its defined macros __alignas_is_defined and __alignas_is_defined is an obsolescent feature.

### 7.31.11 Atomics <stdatomic.h>

Macros that begin with ATOMIC_ and an uppercase letter 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 may be added to the types defined in the <stdint.h> header. Enumeration constants that begin with memory_order_ and a lowercase letter 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 may be added to the declarations in the <stdatomic.h> header.

The macro ATOMIC_VAR_INIT is an obsolescent feature.

### 7.31.12 Boolean type and values <stdbool.h>

The ability to undefine and perhaps then redefine the macros bool, true, and false header <stdbool.h> together with its defined macro __bool_true_false_are_defined is an obsolescent feature.

### 7.31.13 Integer types <stdint.h>

Typedef names beginning with int or uint and ending with _t 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 may be added to the macros defined in the <stdint.h> header.

### 7.31.14 Input/output <stdio.h>

Lowercase letters may be added to the conversion specifiers and length modifiers in fprintf and fscanf. Other characters may be used in extensions.

The use of ungetc on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

### 7.31.15 General utilities <stdlib.h>

Function names that begin with str or wcs and a lowercase letter may be added to the declarations in the <stdlib.h> header.

Invoking realloc with a size argument equal to zero is an obsolescent feature.

### 7.31.16 String handling <string.h>

Function names that begin with str, mem, or wcs and a lowercase letter may be added to the declarations in the <string.h> header.
Annex A  
(informative)  
Language syntax summary

1  NOTE  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 non-white-space character that cannot be one of the above

A.1.2  Keywords

(6.4.1) keyword: one of
alignas
alignof
auto
bool
break
case
cchar
cconst
ccontinue
default
do
double
delse
denum
extern
false
float
for
goto
if
inline
int
long
register
restrict
return
short
signed
sizeof
static
static_assert
struct
switch
thread_local
ttrue
typedef
unsigned
volatile
while

A.1.3  Identifiers

(6.4.2.1) identifier:
identifier-nondigit
identifier  identifier-nondigit
identifier  digit

(6.4.2.1) identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters
(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**:

- \(\backslash u\ \text{hex-quad}\)
- \(\backslash U\ \text{hex-quad} \text{hex-quad}\)

(6.4.3) **hex-quad**: \n
- \(\text{hexadecimal-digit} \ \text{hexadecimal-digit} \ \text{hexadecimal-digit} \ \text{hexadecimal-digit}\)

### A.1.5 Constants

(6.4.4) **constant**:

- \(\text{integer-constant}\)
- \(\text{floating-constant}\)
- \(\text{enumeration-constant}\)
- \(\text{character-constant}\)
- \(\text{predefined-constant}\)

(6.4.4.1) **integer-constant**:

- \(\text{decimal-constant} \ \text{integer-suffix}_{\text{opt}}\)
- \(\text{octal-constant} \ \text{integer-suffix}_{\text{opt}}\)
- \(\text{hexadecimal-constant} \ \text{integer-suffix}_{\text{opt}}\)

(6.4.4.1) **decimal-constant**:

- \(\text{nonzero-digit}\)
- \(\text{decimal-constant} \ \text{digit}\)

(6.4.4.1) **octal-constant**:

- \(0\)
- \(\text{octal-constant} \ \text{octal-digit}\)

(6.4.4.1) **hexadecimal-constant**:

- \(\text{hexadecimal-prefix} \ \text{hexadecimal-digit}\)
- \(\text{hexadecimal-constant} \ \text{hexadecimal-digit}\)

(6.4.4.1) **hexadecimal-prefix**: one of

- \(0x\ \ 0X\)

(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

- \(0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\)

(6.4.4.1) **hexadecimal-digit**: one of

- \(0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\)
- \(a\ b\ c\ d\ e\ f\)
- \(A\ B\ C\ D\ E\ F\)

§ A.1.5 Language syntax summary 393
(6.4.4.3) enumeration-constant:
   identifier

(6.4.4.4) character-constant:
   ' c-char-sequence '
   L' c-char-sequence '
   u' c-char-sequence '
   U' c-char-sequence '

(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
   \ ' " \? \ \n \a \b \f \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

A.1.5.1 Predefined constants

(6.4.5) predefined-constant:
   ~~~~~~~~~~~~~~~~~~~ false
   ~~~~~~~~~~~~~~~~~~~ true

A.1.6 String literals

(6.4.5) string-literal:
   encoding-prefix_opt" s-char-sequence_opt"

(6.4.5) encoding-prefix:
   u8
   u
   U
   L

(6.4.5) s-char-sequence:
   s-char
   s-char-sequence s-char
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_opt )
    postfix-expression . identifier
    postfix-expression -> identifier
    postfix-expression ++
    postfix-expression --
    ( type-name ) { initializer-list }
    ( type-name ) { initializer-list , }

(6.5.2) argument-expression-list:
    assignment-expression
    argument-expression-list , assignment-expression

(6.5.3) unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
    _Alignof _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
(6.7) declaration:
  no-leading-attribute-declaration
  attribute-specifier-sequence declaration-specifiers init-declarator-list ;
  attribute-declaration

(6.7) declaration-specifiers:
  declaration-specifier attribute-specifier-sequence attribute-declaration

(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:
  typedef
  extern
  static
  __Thread_local
  thread_local
  auto
  register

(6.7.2) type-specifier:
  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  __Bool__
  _Bool_
  _Complex
  _Decimal32
  _Decimal64
  _Decimal128
  atomic-type-specifier
  struct-or-union-specifier
  enum-specifier
  typedef-name

(6.7.2.1) struct-or-union-specifier:
  struct-or-union attribute-specifier-sequence identifier { member-declaration-list }

(6.7.2.1) struct-or-union:
  struct
  union

(6.7.2.1) member-declaration-list:
  member-declaration
  member-declaration-list member-declaration
(6.7.2.1) member-declaration:
\[ \text{attribute-specifier-sequence}_{\text{opt}} \text{ specifier-qualifier-list } \text{ member-declarator-list}_{\text{opt}} ; \]
\[ \text{static assert-declaration} \]
(6.7.2.1) specifier-qualifier-list:
\[ \text{type-specifier-qualifier } \text{ attribute-specifier-sequence}_{\text{opt}} \]
\[ \text{type-specifier-qualifier } \text{ specifier-qualifier-list} \]
(6.7.2.1) type-specifier-qualifier:
\[ \text{type-specifier} \]
\[ \text{type-qualifier} \]
\[ \text{alignment-specifier} \]
(6.7.2.1) member-declarator-list:
\[ \text{member-declarator} \]
\[ \text{member-declarator-list} , \text{ member-declarator} \]
(6.7.2.1) member-declarator:
\[ \text{declarator} \]
\[ \text{declarator}_{\text{opt}} : \text{ constant-expression} \]
(6.7.2.2) enum-specifier:
\[ \text{enum } \text{ attribute-specifier-sequence}_{\text{opt}} \text{ identifier}_{\text{opt}} \{ \text{enumerator-list} \} \]
\[ \text{enum } \text{ attribute-specifier-sequence}_{\text{opt}} \text{ identifier}_{\text{opt}} \{ \text{enumerator-list} , \} \]
\[ \text{enum } \text{ identifier} \]
(6.7.2.2) enumerator-list:
\[ \text{enumerator} \]
\[ \text{enumerator-list} , \text{ enumerator} \]
(6.7.2.2) enumerator:
\[ \text{enumeration-constant } \text{ attribute-specifier-sequence}_{\text{opt}} \]
\[ \text{enumeration-constant } \text{ attribute-specifier-sequence}_{\text{opt}} = \text{ constant-expression} \]
(6.7.2.4) atomic-type-specifier:
\[ \text{\_Atomic ( type-name )} \]
(6.7.3) type-qualifier:
\[ \text{const} \]
\[ \text{restrict} \]
\[ \text{volatile} \]
\[ \text{\_Atomic} \]
(6.7.4) function-specifier:
\[ \text{inline} \]
\[ \text{\_Noreturn} \]
(6.7.5) alignment-specifier:
\[ \text{\_Alignas alignas ( type-name )} \]
\[ \text{\_Alignas alignas ( constant-expression )} \]
(6.7.6) declarator:
\[ \text{pointer}_{\text{opt}} \text{ direct-declarator} \]
(6.7.6) direct-declarator:
\[ \text{identifier } \text{ attribute-specifier-sequence}_{\text{opt}} \]
\[ ( \text{ declarator } ) \]
\[ \text{array-declarator } \text{ attribute-specifier-sequence}_{\text{opt}} \]
\[ \text{function-declarator } \text{ attribute-specifier-sequence}_{\text{opt}} \]
\[ \text{direct-declarator } ( \text{ identifier-list}_{\text{opt}} ) \]
(6.7.6) array-declarator:
\[ \text{direct-declarator [ type-qualifier-list}_{\text{opt}} \text{ assignment-expression}_{\text{opt}} \]}
\[ \text{direct-declarator [ static type-qualifier-list}_{\text{opt}} \text{ assignment-expression } \]
\[ \text{direct-declarator [ type-qualifier-list static assignment-expression ]} \]
\[ \text{direct-declarator [ type-qualifier-list}_{\text{opt}} * \]}

400 Language syntax summary § A.2.2
(6.7.9) designator:

- constant-expression
- identifier

(6.7.10) static_assert-declaration:

```c
static_assert ( constant-expression , string-literal ) ;
static_assert ( constant-expression ) ;
```

(6.7.11.1) attribute-specifier-sequence:

```c
attribute-specifier-sequenceopt attribute-specifier
```

(6.7.11.1) attribute-specifier:

```c
[ [ attribute-list ] ]
```

(6.7.11.1) attribute-list:

```c
attributeopt attribute-list , attributeopt
```

(6.7.11.1) attribute:

```c
attribute-token attribute-argument-clauseopt
```

(6.7.11.1) attribute-token:

```c
standard-attribute
attribute-prefixed-token
```

(6.7.11.1) standard-attribute:

```c
identifier
```

(6.7.11.1) attribute-prefixed-token:

```c
attribute-prefix :: identifier
```

(6.7.11.1) attribute-prefix:

```c
identifier
```

(6.7.11.1) attribute-argument-clause:

```c
( balanced-token-sequenceopt )
```

(6.7.11.1) balanced-token-sequence:

```c
balanced-token
balanced-token-sequence balanced-token
```

(6.7.11.1) balanced-token:

```c
( balanced-token-sequenceopt )
[ balanced-token-sequenceopt ]
{ balanced-token-sequenceopt }
any token other than a parenthesis, a bracket, or a brace
```

### A.2.3 Statements

(6.8) statement:

```c
labeled-statement
expression-statement
attribute-specifier-sequenceopt compound-statement
attribute-specifier-sequenceopt selection-statement
attribute-specifier-sequenceopt iteration-statement
attribute-specifier-sequenceopt jump-statement
```

(6.8.1) labeled-statement:

```c
attribute-specifier-sequenceopt identifier : statement
```

(6.8.2) compound-statement:

```c
{ block-item-listopt }
```

(6.8.2) block-item-list:

```c
block-item
block-item-list block-item
```