

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	offsetof
_Complex_I	false	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:

alignas	bool	thread_local	true
alignof	static_assert	false	

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**.

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:

```
#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:

```
#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

<u>alignas</u>	extern	sizeof	__Alignof
<u>alignof</u>	<u>false</u>	static	<u>__Atomic</u>
auto	float	<u>static_assert</u>	__Bool
<u>bool</u>	for	struct	<u>__Complex</u>
break	goto	switch	<u>__Decimal128</u>
case	if	<u>thread_local</u>	<u>__Decimal32</u>
char	inline	<u>true</u>	<u>__Decimal64</u>
const	int	typedef	<u>__Generic</u>
continue	long	union	<u>__Imaginary</u>
default	register	unsigned	<u>__Noreturn</u>
do	restrict	void	__Static_assert
double	return	volatile	__Thread_local
else	short	while	
enum	signed	__Alignas	

Constraints

- 2 The keywords

<u>alignas</u>	<u>bool</u>	<u>static_assert</u>	<u>true</u>
<u>alignof</u>	<u>false</u>	<u>thread_local</u>	

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.⁷⁴⁾

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.⁷⁵⁾
- 4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.⁷⁶⁾

<u>keyword</u>	<u>alternative spelling</u>
<u>alignas</u>	__Alignas
<u>alignof</u>	__Alignof
<u>bool</u>	__Bool
<u>static_assert</u>	__Static_assert
<u>thread_local</u>	__Thread_local

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 correspondig 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

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

Constraints

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

Semantics

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

6.4.4.1 Integer constants

Syntax

- 1 *integer-constant*:
- decimal-constant integer-suffix_{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

```
1  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

1 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

```
1  string-literal:
    encoding-prefixopt " s-char-sequenceopt "
```

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.

- 2 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

- 1 [The keywords](#)

<code>alignas</code>	<code>bool</code>	<code>static_assert</code>	<code>true</code>
<code>alignof</code>	<code>false</code>	<code>thread_local</code>	

[optionally are also predefined macro names that expand to unspecified tokens.](#)

6.10.9 Pragma operator

Semantics

- 1 A unary operator expression of the form:

`_Pragma (string-literal)`

is processed as follows: The string literal is *destringized* by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence `\` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

- 2 **EXAMPLE** A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\listing.dir\"")
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING (..\listing.dir)
```

7.2 Diagnostics <assert.h>

- 1 The header <assert.h> defines the **assert** and **static_assert** macros [macro](#) 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.

- 2 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

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

##### Description

- 2 The **assert** macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if *expression* (which shall have a scalar type) is false (that is, compares equal to 0), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros **\_\_FILE\_\_** and **\_\_LINE\_\_** and of the identifier **\_\_func\_\_**) on the standard error stream in an implementation-defined format.<sup>207)</sup> It then calls the **abort** function.

##### Returns

- 3 The **assert** macro returns no value.

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

<sup>207)</sup>The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```

## 7.15 Alignment `<stdalign.h>`

~~The header defines four macros:~~

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

```
__alignas_is_defined
```

and

```
__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**

```
1 #include <stdatomic.h>
bool atomic_is_lock_free(const volatile A *obj);
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)~~ `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.<sup>280)</sup>

**7.17.6 Atomic integer types**

- 1 For each line in the following table,<sup>281)</sup> the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.<sup>282)</sup>

| Atomic type name                  | Direct type                                                     |
|-----------------------------------|-----------------------------------------------------------------|
| <code>atomic_bool</code>          | <del><code>_Atomic _Bool</code></del> <code>_Atomic bool</code> |
| <code>atomic_char</code>          | <code>_Atomic char</code>                                       |
| <code>atomic_schar</code>         | <code>_Atomic signed char</code>                                |
| <code>atomic_uchar</code>         | <code>_Atomic unsigned char</code>                              |
| <code>atomic_short</code>         | <code>_Atomic short</code>                                      |
| <code>atomic_ushort</code>        | <code>_Atomic unsigned short</code>                             |
| <code>atomic_int</code>           | <code>_Atomic int</code>                                        |
| <code>atomic_uint</code>          | <code>_Atomic unsigned int</code>                               |
| <code>atomic_long</code>          | <code>_Atomic long</code>                                       |
| <code>atomic_ulong</code>         | <code>_Atomic unsigned long</code>                              |
| <code>atomic_llong</code>         | <code>_Atomic long long</code>                                  |
| <code>atomic_ullong</code>        | <code>_Atomic unsigned long long</code>                         |
| <code>atomic_char16_t</code>      | <code>_Atomic char16_t</code>                                   |
| <code>atomic_char32_t</code>      | <code>_Atomic char32_t</code>                                   |
| <code>atomic_wchar_t</code>       | <code>_Atomic wchar_t</code>                                    |
| <code>atomic_int_least8_t</code>  | <code>_Atomic int_least8_t</code>                               |
| <code>atomic_uint_least8_t</code> | <code>_Atomic uint_least8_t</code>                              |

<sup>280)</sup> `obj` can be a null pointer.

<sup>281)</sup> See “future library directions” (7.31.11).

<sup>282)</sup> 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

- 1 The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.<sup>339)</sup>
- 2 Implementations that define the macro `__STDC_NO_THREADS__` need not provide this header nor support any of its facilities.

which expands to the keyword `_Thread_local`; The macros are

`ONCE_FLAG_INIT`

which expands to a value that can be used to initialize an object of type `once_flag`; and

`TSS_DTOR_ITERATIONS`

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

- 4 The types are

`cond_t`

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

`thrd_t`

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

`tss_t`

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

`mtx_t`

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

`tss_dtor_t`

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

`thrd_start_t`

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

`once_flag`

which is a complete object type that holds a flag for use by `call_once`.

- 5 The enumeration constants are

`mtx_plain`

which is passed to `mtx_init` to create a mutex object that does not support timeout;

`mtx_recursive`

<sup>339)</sup>See “future library directions” (7.31.18).

|                        |                          |                        |                       |                      |
|------------------------|--------------------------|------------------------|-----------------------|----------------------|
| <code>cracosh</code>   | <code>cratanh</code>     | <code>crexp10</code>   | <code>crlog1p</code>  | <code>crrootn</code> |
| <code>cracospi</code>  | <code>cratanpi</code>    | <code>crexp2m1</code>  | <code>crlog2p1</code> | <code>crrsqrt</code> |
| <code>cracos</code>    | <code>cratan</code>      | <code>crexp2</code>    | <code>crlog2</code>   | <code>crsinh</code>  |
| <code>crasinh</code>   | <code>crcompoundn</code> | <code>crexpm1</code>   | <code>crlogp1</code>  | <code>crsinpi</code> |
| <code>crasinpi</code>  | <code>crsinh</code>      | <code>crexp</code>     | <code>crlog</code>    | <code>crsin</code>   |
| <code>crasin</code>    | <code>cracospi</code>    | <code>crhypot</code>   | <code>crpown</code>   | <code>crtanh</code>  |
| <code>cratan2pi</code> | <code>cracos</code>      | <code>crlog10p1</code> | <code>crpowr</code>   | <code>crtanpi</code> |
| <code>cratan2</code>   | <code>crexp10m1</code>   | <code>crlog10</code>   | <code>crpow</code>    | <code>crtan</code>   |

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>`

- 1 Macros that begin with either `SIG` and an uppercase letter or `SIG_` and an uppercase letter may be added to the macros defined in the `<signal.h>` header.

### 7.31.10 Alignment `<stdalign.h>`

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

### 7.31.11 Atomics `<stdatomic.h>`

- 1 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 declarations in the `<stdatomic.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.
- 2 The macro `ATOMIC_VAR_INIT` is an obsolescent feature.

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

- 1 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>`

- 1 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>`

- 1 Lowercase letters may be added to the conversion specifiers and length modifiers in `fprintf` and `fscanf`. Other characters may be used in extensions.
- 2 The use of `ungetc` on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

### 7.31.15 General utilities `<stdlib.h>`

- 1 Function names that begin with `str` or `wcs` and a lowercase letter may be added to the declarations in the `<stdlib.h>` header.
- 2 Invoking `realloc` with a `size` argument equal to zero is an obsolescent feature.

### 7.31.16 String handling `<string.h>`

- 1 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

|                |              |                      |                           |
|----------------|--------------|----------------------|---------------------------|
| <u>alignas</u> | extern       | sizeof               | <del>_Alignof</del>       |
| <u>alignof</u> | <u>false</u> | static               | _Atomic                   |
| auto           | float        | <u>static_assert</u> | <del>_Bool</del>          |
| <u>bool</u>    | for          | struct               | _Complex                  |
| break          | goto         | switch               | _Decimal128               |
| case           | if           | <u>thread_local</u>  | _Decimal32                |
| char           | inline       | <u>true</u>          | _Decimal64                |
| const          | int          | typedef              | _Generic                  |
| continue       | long         | union                | _Imaginary                |
| default        | register     | unsigned             | _Noreturn                 |
| do             | restrict     | void                 | <del>_Static_assert</del> |
| double         | return       | volatile             | <del>_Thread_local</del>  |
| else           | short        | while                |                           |
| enum           | signed       | <del>_Alignas</del>  |                           |

#### 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*:

```

\u hex-quad
\U hex-quad hex-quad

```

(6.4.3) *hex-quad*:

```
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit
```

## A.1.5 Constants

(6.4.4) *constant*:

```

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

```

(6.4.4.1) *integer-constant*:

```

decimal-constant integer-suffixopt
octal-constant integer-suffixopt
hexadecimal-constant integer-suffixopt

```

(6.4.4.1) *decimal-constant*:

```

nonzero-digit
decimal-constant digit

```

(6.4.4.1) *octal-constant*:

```

0
octal-constant octal-digit

```

(6.4.4.1) *hexadecimal-constant*:

```

hexadecimal-prefix hexadecimal-digit
hexadecimal-constant 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

```

(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

\ ' \" \? \\

\a \b \f \n \r \t \v

(6.4.4.4) *octal-escape-sequence*:

\ *octal-digit*

\ *octal-digit* *octal-digit*

\ *octal-digit* *octal-digit* *octal-digit*

(6.4.4.4) *hexadecimal-escape-sequence*:

\x *hexadecimal-digit*

*hexadecimal-escape-sequence* *hexadecimal-digit*

### A.1.5.1 Predefined constants

(6.4.4.5) *predefined-constant*:

**false**

**true**

### A.1.6 String literals

(6.4.5) *string-literal*:

*encoding-prefix*<sub>opt</sub> " *s-char-sequence*<sub>opt</sub> "

(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*<sub>opt</sub> )  
*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<sub>opt</sub>*  
*declaration-specifier declaration-specifiers*

(6.7) *declaration-specifier*:

*storage-class-specifier*  
*type-specifier-qualifier*  
*function-specifier*

(6.7) *init-declarator-list*:

*init-declarator*  
*init-declarator-list , init-declarator*

(6.7) *init-declarator*:

*declarator*  
*declarator = initializer*

(6.7) *attribute-declaration*:

*attribute-specifier-sequence ;*

(6.7.1) *storage-class-specifier*:

**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<sub>opt</sub> identifier<sub>opt</sub> { member-declaration-list }*  
*struct-or-union attribute-specifier-sequence<sub>opt</sub> identifier*

(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*:

*attribute-specifier-sequence*<sub>opt</sub> *specifier-qualifier-list* *member-declarator-list*<sub>opt</sub> ;  
*static\_assert-declaration*

(6.7.2.1) *specifier-qualifier-list*:

*type-specifier-qualifier* *attribute-specifier-sequence*<sub>opt</sub>  
*type-specifier-qualifier* *specifier-qualifier-list*

(6.7.2.1) *type-specifier-qualifier*:

*type-specifier*  
*type-qualifier*  
*alignment-specifier*

(6.7.2.1) *member-declarator-list*:

*member-declarator*  
*member-declarator-list* , *member-declarator*

(6.7.2.1) *member-declarator*:

*declarator*  
*declarator*<sub>opt</sub> : *constant-expression*

(6.7.2.2) *enum-specifier*:

**enum** *attribute-specifier-sequence*<sub>opt</sub> *identifier*<sub>opt</sub> { *enumerator-list* }  
**enum** *attribute-specifier-sequence*<sub>opt</sub> *identifier*<sub>opt</sub> { *enumerator-list* , }  
**enum** *identifier*

(6.7.2.2) *enumerator-list*:

*enumerator*  
*enumerator-list* , *enumerator*

(6.7.2.2) *enumerator*:

*enumeration-constant* *attribute-specifier-sequence*<sub>opt</sub>  
*enumeration-constant* *attribute-specifier-sequence*<sub>opt</sub> = *constant-expression*

(6.7.2.4) *atomic-type-specifier*:

**\_Atomic** ( *type-name* )

(6.7.3) *type-qualifier*:

**const**  
**restrict**  
**volatile**  
**\_Atomic**

(6.7.4) *function-specifier*:

**inline**  
**\_Noreturn**

(6.7.5) *alignment-specifier*:

~~**Alignas**~~ **alignas** ( *type-name* )  
~~**Alignas**~~ **alignas** ( *constant-expression* )

(6.7.6) *declarator*:

*pointer*<sub>opt</sub> *direct-declarator*

(6.7.6) *direct-declarator*:

*identifier* *attribute-specifier-sequence*<sub>opt</sub>  
( *declarator* )  
*array-declarator* *attribute-specifier-sequence*<sub>opt</sub>  
*function-declarator* *attribute-specifier-sequence*<sub>opt</sub>  
*direct-declarator* ( *identifier-list*<sub>opt</sub> )

(6.7.6) *array-declarator*:

*direct-declarator* [ *type-qualifier-list*<sub>opt</sub> *assignment-expression*<sub>opt</sub> ]  
*direct-declarator* [ **static** *type-qualifier-list*<sub>opt</sub> *assignment-expression* ]  
*direct-declarator* [ *type-qualifier-list* **static** *assignment-expression* ]  
*direct-declarator* [ *type-qualifier-list*<sub>opt</sub> \* ]

(6.7.9) *designator*:

[ *constant-expression* ]  
 . *identifier*

(6.7.10) *static\_assert-declaration*:

~~Static\_assert~~ static\_assert ( *constant-expression* , *string-literal* ) ;  
~~Static\_assert~~ static\_assert ( *constant-expression* ) ;

(6.7.11.1) *attribute-specifier-sequence*:

*attribute-specifier-sequence*<sub>opt</sub> *attribute-specifier*

(6.7.11.1) *attribute-specifier*:

[ [ *attribute-list* ] ]

(6.7.11.1) *attribute-list*:

*attribute*<sub>opt</sub>  
*attribute-list* , *attribute*<sub>opt</sub>

(6.7.11.1) *attribute*:

*attribute-token* *attribute-argument-clause*<sub>opt</sub>

(6.7.11.1) *attribute-token*:

*standard-attribute*  
*attribute-prefixed-token*

(6.7.11.1) *standard-attribute*:

*identifier*

(6.7.11.1) *attribute-prefixed-token*:

*attribute-prefix* :: *identifier*

(6.7.11.1) *attribute-prefix*:

*identifier*

(6.7.11.1) *attribute-argument-clause*:

( *balanced-token-sequence*<sub>opt</sub> )

(6.7.11.1) *balanced-token-sequence*:

*balanced-token*  
*balanced-token-sequence* *balanced-token*

(6.7.11.1) *balanced-token*:

( *balanced-token-sequence*<sub>opt</sub> )  
 [ *balanced-token-sequence*<sub>opt</sub> ]  
 { *balanced-token-sequence*<sub>opt</sub> }

any token other than a parenthesis, a bracket, or a brace

### A.2.3 Statements

(6.8) *statement*:

*labeled-statement*  
*expression-statement*  
*attribute-specifier-sequence*<sub>opt</sub> *compound-statement*  
*attribute-specifier-sequence*<sub>opt</sub> *selection-statement*  
*attribute-specifier-sequence*<sub>opt</sub> *iteration-statement*  
*attribute-specifier-sequence*<sub>opt</sub> *jump-statement*

(6.8.1) *labeled-statement*:

*attribute-specifier-sequence*<sub>opt</sub> *identifier* : *statement*  
*attribute-specifier-sequence*<sub>opt</sub> **case** *constant-expression* : *statement*  
*attribute-specifier-sequence*<sub>opt</sub> **default** : *statement*

(6.8.2) *compound-statement*:

{ *block-item-list*<sub>opt</sub> }

(6.8.2) *block-item-list*:

*block-item*  
*block-item-list* *block-item*