

Align spelling of keywords with C++ and make them feature tests proposal for C2x

Jens Gustedt

▶ **To cite this version:**

Jens Gustedt. Align spelling of keywords with C++ and make them feature tests proposal for C2x. [Technical Report] n2368, ISO JCT1/SC22/WG14. 2019. hal-02089925

HAL Id: hal-02089925
https://inria.hal.science/hal-02089925

Submitted on 4 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ISO/IEC JTC 1/SC 22/WG14 **N 2368**

v 1

April 1, 2019

Align spelling of keywords with C++ and make them feature tests

proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Over time C has integrated certain features in coordination with C++, but the strategy to integrate the corresponding keywords has not be entirely consistent: some were integrated with the same syntax as C++ (`const`, `inline`) others were integrated in an underscore-capitalized form. Compatibility with C++ is then ensured via a set of library header files that provide the C++ spelling. 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 same primary spelling as for C++.

1. INTRODUCTION

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

`_Alignas`	`_Bool`	`_Noreturn`
`_Alignof`	`_Complex`	`_Static_assert`
`_Atomic`	`_Imaginary`	`_Thread_local`

Most 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 are provided through headers and have not achieved the status of first class language constructs:

`NULL`	`_Imaginary_I`	`true`
`_Complex_I`	`false`	

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.
In addition to that, there is a specific problem with the `NULL` macro that is underspecified and which C++, also since years, has replaced with an unambiguous keyword `nullptr`.

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:

— Require the keywords to be also macros that can be tested.
— 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.

© 2019 by the author(s). Distributed under a Creative Commons Attribution 4.0 International License

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

— 13 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`. The integration of the constants **_Complex_I** and **_Imaginary_I** is direct and we think that they do not need much explanations.

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**.
Since for C++, these constants are of type **bool**, we propose to do it the same. For implementations that do not want to integrate these constants in their parser infrastructure, yet, this can easily be achieved by definitions similar to the following

```
#define false ((bool)+0)
#define true  ((bool)+1)
```

which, by preprocesor magic, expands to values `0` and `1` when used in preprocessor conditionals, and to constant expressions of type **bool** in other places. In most operations these will then undergo promotion to **int**, such that there should be no difference for code that uses them in arithmetic.

3.2. Null pointer constants

The macro **NULL** that goes back quite early, was meant to provide a tool to specify a null pointer constant such that it is easily visible and such that it makes the intention of the programmer to specifier a pointer value clear. Unfortunately, the definition as it is given in the standard misses that goal, because the constant that is hidden behind the macro can be of very different nature.
A *null pointer constant* can be any integer constant of value 0 or such a constant converted to **void***. Thereby several types are possible for **NULL**. Commonly used are `0` with **int**, `0L` with **long** and `(void*)0` with **void***.

(1) Similar to the problems with **false** and **true** above, this may lead to surprises when invoking a type-generic macro with a **NULL** argument.
(2) Conditional expressions such as `(1 ? 0 : NULL)` and `(1 ? 1 : NULL)` have different status depending how **NULL** is defined. Whereas the first is always defined, the second is a constraint violation if **NULL** has type **void***, and defined otherwise.

(3) A `NULL` argument that is passed to a `va_arg` function that expects a pointer can have severe consequences. On many architectures nowadays `int` and `void*` have different size, and so if `NULL` is just `0`, a wrongly sized arguments is passed to the function.

Because of such problems, C++ has long shifted to a different setting, namely the keyword `nullptr`. Because of the different conversion rules for `void*`, C++ specification is a bit complicated. In particular there is a special type `nullptr_t` for this constant. This is not necessary for C: we can just define `nullptr` to be `((void*)0)` and have the same properties as in C++.

4. FEATURE TESTS

As additional effect of having the keywords to be macros, too, the macros `complex` and `imaginary` can be used as feature tests for complex and imaginary types, respectively. This can eventually be useful for constrained implementations that want to implement complex types, without providing full library support.

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         ((bool)+0)
#define noreturn      _Noreturn
#define nullptr       ((void*)0)
#define static_assert _Static_assert
#define thread_local  _Thread_local
#define true          ((bool)+1)
#ifndef __STDC_NO_COMPLEX__
# define _Complex_I    // implementation specific
# define _Imaginary_I  // implementation specific
# define complex       _Complex
# define imaginary     _Imaginary
#endif
```

Appendix: pages with diffmarks of the proposed changes against the March 2019 working draft.

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

Foreword

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

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

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

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

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

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

7 This fifth edition cancels and replaces the fourth edition, ISO/IEC 9899:2018. Major changes from the previous edition include:

- ~~added~~ add a one-argument version of ~~`_Static_assert`~~ **`static_assert`**, make it a keyword and deprecate the underscore-capital form
- an integration of floating-point technical specification TS 18661-1
- add a universal null pointer constant **`nullptr`**
- change **`bool`**, **`false`** and **`true`** to keywords and make the constants type **`bool`**
- change **`alignas`**, **`alignof`**, **`complex`**, **`imaginary`**, **`noreturn`** and **`thread_local`** to be keywords and deprecate the underscore-capital forms
- change **`_Complex_I`** and **`_Imaginary_I`** to be keywords

8 A complete change history can be found in Annex M.

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

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[35] and retains its last-stored value throughout its lifetime.[36] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

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

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

5 An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*, as do some compound literals. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

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

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[37] If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary lifetime*.[38] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime results in undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. Such an object need not have a unique address.

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), initialization (6.7.9), statements (6.8), effective type (6.5).

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

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

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

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

6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).[39)]

2 An object declared as type ~~`_Bool`~~ **`bool`** is large enough to store the values ~~0 and 1~~ **`false`** and **`true`**.

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

4 There are five *standard signed integer types*, designated as **`signed char`**, **`short int`**, **`int`**, **`long int`**, and **`long long int`**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.[40)] The standard and extended signed integer types are collectively called *signed integer types*.[41)]

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

6 For each of the signed integer types, there is a *corresponding* (but different) *unsigned integer type* (designated with the keyword **`unsigned`**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type ~~`_Bool`~~ **`bool`** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard and extended unsigned integer types are collectively called *unsigned integer types*.[42)]

7 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

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

9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[43)] A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

10 There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**.[44)] The set of values of the type **`float`** is a subset of the set of values of the type **`double`**; the set of values of the type **`double`** is a subset of the set of values of the type **`long double`**.

11 There are three *complex types*, designated as ~~`float _Complex`~~, **`float complex`**, ~~`double _Complex`~~, **`double complex`**, and **`long double`** ~~`_Complex`~~ **`complex`**.[45)] ~~(~~Complex types are a conditional

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

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

[41)] Therefore, any statement in this document about signed integer types also applies to the extended signed integer types.

[42)] Therefore, any statement in this document about unsigned integer types also applies to the extended unsigned integer types.

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

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

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

feature that implementations ~~need not support ; see 6.10.8.3.)~~ support if and only if the macro **complex** is defined.[46] The real floating and complex types are collectively called the *floating types*.

12 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword ~~**_Complex**~~ **complex** from the type name.

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

14 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[47]

15 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[48]

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

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

18 Integer and floating types are collectively called *arithmetic types*.[49] Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

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

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

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

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

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

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

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced

[46] See also 6.10.8.3.

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

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

[49] Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

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

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

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

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

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

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.

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

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

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

Forward references: array declarators (6.7.6.2).

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

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

The resulting composite type for the function is:

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

6.2.8 Alignment of objects

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

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

— all atomic, qualified, or unqualified basic types;

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

— all atomic, qualified, or unqualified enumerated types;

— all atomic, qualified, or unqualified pointer types;

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

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

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

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

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

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

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

7 Comparing alignments is meaningful and provides the obvious results:

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

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

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

6.3 Conversions

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

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

Forward references: cast operators (6.5.4).

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If an **int** can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.[62] All other types are unchanged by the integer promotions.

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

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

6.3.1.2 Boolean type

1 When any scalar value is converted to **bool**, the result is ~~0~~ **false** if the value compares equal to 0; otherwise, the result is ~~1~~**true**.[63]

6.3.1.3 Signed and unsigned integers

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

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

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

6.3.1.4 Real floating and integer

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

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

[63] NaNs do not compare equal to 0 and thus convert to **true**.

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

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

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

6.3.1.5 Real floating types

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

6.3.1.6 Complex types

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

6.3.1.7 Real and complex

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

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

6.3.1.8 Usual arithmetic conversions

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

> First, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**.
>
> Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**.
>
> Otherwise, if the corresponding real type of either operand is **`float`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`float`**.[67]
>
> Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
> > If both operands have the same type, then no further conversion is needed.
> >
> > Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

[66] See 6.3.1.2.

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

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

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

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

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

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

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

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

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

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions `<stddef.h>` (7.19), initialization (6.7.9), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** and ~~_Alignof~~ **alignof** operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression.

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

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

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

If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 Pointers

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

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

3 ~~An~~ The keyword **nullptr**, an integer constant expression with the value 0, or such an expression cast to type **void** *, ~~is~~ are called a *null pointer constant*.[70] If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

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

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

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

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

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

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.16.1).

[70] The obsolescent macro **NULL** is defined in <stddef.h> (and other headers) as a null pointer constant; see 7.19, but new code should prefer the keyword **nullptr** wherever a null pointer constant is specified.

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

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

6.4 Lexical elements

Syntax

1 *token:*
keyword
identifier
constant
string-literal
punctuator

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

Constraints

2 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.

Semantics

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

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

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

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

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5).

[73)] An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files.

6.4.1 Keywords

Syntax

1 *keyword:* one of

alignas	**extern**	**short**	~~**_Alignas**~~
alignof	**false**	**signed**	~~**_Alignof**~~
auto	**float**	**sizeof**	**_Atomic**
bool	**for**	**static**	~~**_Bool**~~
break	**goto**	**static_assert**	~~**_Complex**~~
case	**if**	**struct**	**_Generic**
char	**imaginary**	**switch**	~~**_Imaginary**~~
complex	**inline**	**thread_local**	**_Complex_I**
const	**int**	**true**	~~**_Noreturn**~~
continue	**long**	**typedef**	~~**_Static_assert**~~
default	**noreturn**	**union**	~~**_Thread_local**~~
do	**nullptr**	**unsigned**	**_Imaginary_I**
double	**register**	**void**	
else	**restrict**	**volatile**	
enum	**return**	**while**	

Constraints

2 The keywords

alignas	**false**	**static_assert**	**complex**	**_Imaginary_I**
alignof	**noreturn**	**thread_local**	**imaginary**	
bool	**nullptr**	**true**	**_Complex_I**	

are also predefined macro names (6.10.8). 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, and shall not be used otherwise. The ~~keyword **_Imaginary** is~~ keywords **complex**, **imaginary**, **_Complex_I**, and **_Imaginary_I** are reserved for specifying ~~imaginary types.~~the optional complex and imaginary[75)] types and imaginary units, respectively.

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

keyword	alternative spelling
alignas	~~**_Alignas**~~
alignof	~~**_Alignof**~~
bool	~~**_Bool**~~
complex	**_Complex**
imaginary	**_Imaginary**
noreturn	~~**_Noreturn**~~
static_assert	~~**_Static_assert**~~
thread_local	~~**_Thread_local**~~

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

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$_{\text{opt}}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{\text{opt}}$
long-long-suffix unsigned-suffix$_{\text{opt}}$

8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[84)]

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

Prefix	Corresponding Type
none	**`unsigned char`**
`L`	the unsigned type corresponding to **`wchar_t`**
`u`	**`char16_t`**
`U`	**`char32_t`**

Semantics

10 An integer character constant has type **`int`**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., `'ab'`), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **`char`** whose value is that of the single character or escape sequence is converted to type **`int`**.

11 A wide character constant prefixed by the letter **`L`** has type **`wchar_t`**, an integer type defined in the `<stddef.h>` header; a wide character constant prefixed by the letter **`u`** or **`U`** has type **`char16_t`** or **`char32_t`**, respectively, unsigned integer types defined in the `<uchar.h>` header. The value of a wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **`mbtowc`**, **`mbrtoc16`**, or **`mbrtoc32`** function as appropriate for its type, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

12 **EXAMPLE 1** The construction `'\0'` is commonly used to represent the null character.

13 **EXAMPLE 2** Consider implementations that use two's complement representation for integers and eight bits for objects that have type **`char`**. In an implementation in which type **`char`** has the same range of values as **`signed char`**, the integer character constant `'\xFF'` has the value −1; if type **`char`** has the same range of values as **`unsigned char`**, the character constant `'\xFF'` has the value +255.

14 **EXAMPLE 3** Even if eight bits are used for objects that have type **`char`**, the construction `'\x123'` specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are `'\x12'` and `'3'`, the construction `'\0223'` can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

15 **EXAMPLE 4** Even if 12 or more bits are used for objects that have type **`wchar_t`**, the construction `L'\1234'` specifies the implementation-defined value that results from the combination of the values `0123` and `'4'`.

Forward references: common definitions `<stddef.h>` (7.19), the **`mbtowc`** function (7.22.7.2), Unicode utilities `<uchar.h>` (7.28).

6.4.4.5 Predefined constants

Syntax

1 *predefined-constant:* one of

`false` **`nullptr`** **`true`** **`_Complex_I`** **`_Imaginary_I`**

[84)] The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See "future language directions" (6.11.4).

Description

2 Some keywords represent constants of a specific value and type.

6.4.4.5.1 The `nullptr` constant

Description

1 The keyword **`nullptr`** represents a null pointer constant of type **`void`**`*` that is a suitable primary expression for initialization, assignment and comparison of any pointer type.[85)]

6.4.4.5.2 The `false` and `true` constants

Description

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

6.4.4.5.3 The `_Complex_I` constant

Description

1 If complex types are supported, the keyword

```
_Complex_I
```

represents a constant of type **`const float complex`**, with the value of the imaginary unit.[87)]

6.4.4.5.4 The `_Imaginary_I` constant

Description

1 If imaginary types are supported, the keyword

```
_Imaginary_I
```

represents a constant of type **`const float imaginary`**, with the value of the imaginary unit.

6.4.5 String literals

Syntax

1 *string-literal:*

encoding-prefix$_{\text{opt}}$ " *s-char-sequence*$_{\text{opt}}$ "

[85)] A suitable implementation for **`nullptr`** is a predefined macro that expands to `((void*)0)`

[86)] Thus, the keywords **`false`** and **`true`** are usable in preprocessor directives. Suitable implementations for **`false`** and **`true`** are predefined macros that expand to `((bool)+0)` and `((bool)+1)`, respectively.

[87)] The imaginary unit is a number i such that $i^2 = -1$.

might yield 1 if the literals' storage is shared.

14 **EXAMPLE 7** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

15 **EXAMPLE 8** Each compound literal creates only a single object in a given scope:

```
struct s { int i; };

int f (void)
{
      struct s *p = 0, *q;
      int j = 0;

again:
      q = p, p = &((struct s){ j++ });
      if (j  <  2) goto again;

      return p == q && q->i == 1;
}
```

The function `f()` always returns the value 1.

16 Note that if an iteration statement were used instead of an explicit **goto** and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

Forward references: type names (6.7.7), initialization (6.7.9).

6.5.3 Unary operators

Syntax

1 *unary-expression:*
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof **(** *type-name* **)**
~~**_Alignof**~~ **alignof** **(** *type-name* **)**

unary-operator: one of
& * + - ~ !

6.5.3.1 Prefix increment and decrement operators

Constraints

1 The operand of the prefix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics

2 The value of the operand of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix `--` operator is analogous to the prefix `++` operator, except that the value of the operand is

decremented.

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary & operator shall be either a function designator, the result of a `[]` or unary `*` operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **`register`** storage-class specifier.

2 The operand of the unary `*` operator shall have pointer type.

Semantics

3 The unary & operator yields the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary `*` operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a `[]` operator, neither the & operator nor the unary `*` that is implied by the `[]` is evaluated and the result is as if the & operator were removed and the `[]` operator were changed to a `+` operator. Otherwise, the result is a pointer to the object or function designated by its operand.

4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.[114)]

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

6.5.3.3 Unary arithmetic operators

Constraints

1 The operand of the unary `+` or `-` operator shall have arithmetic type; of the `~` operator, integer type; of the `!` operator, scalar type.

Semantics

2 The result of the unary `+` operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

3 The result of the unary `-` operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

4 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.

6.5.3.4 The `sizeof` and `alignof` operators

Constraints

1 The **`sizeof`** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The ~~`_Alignof`~~ **`alignof`** operator shall not be applied to a function type or an incomplete type.

[114)] Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary & operator, `*&E` is a function designator or an lvalue equal to `E`. If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Semantics

2 The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

3 The ~~_Alignof~~ **alignof** operator yields the alignment requirement of its operand type. The operand is not evaluated and the result is an integer constant. When applied to an array type, the result is the alignment requirement of the element type.

4 When **sizeof** is applied to an operand that has type **char**, **unsigned char**, or **signed char**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[115)] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is **size_t**, defined in <stddef.h> (and other headers).

6 **EXAMPLE 1** A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the alloc function presumably ensures that its return value is aligned suitably for conversion to a pointer to **double**.

7 **EXAMPLE 2** Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

8 **EXAMPLE 3** In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
      char b[n+3];       // variable length array
      return sizeof b;   // execution time sizeof
}

int main()
{
      size_t size;
      size = fsize3(10); // fsize3 returns 13
      return 0;
}
```

Forward references: common definitions <stddef.h> (7.19), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.7), array declarators (6.7.6.2).

6.5.4 Cast operators

Syntax

1 *cast-expression:*

unary-expression
(*type-name* **)** *cast-expression*

[115)] When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1).

Semantics

3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment,[123] but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[124]

— the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;

— the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of **void**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

— the left operand is an atomic, qualified, or unqualified pointer, and the right is a null pointer constant; or

— the left operand has type atomic, qualified, or unqualified ~~_Bool~~ **bool**, and the right is a pointer.

Semantics

2 In *simple assignment* (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

4 **EXAMPLE 1** In the program fragment

```
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
      /* ... */
```

the **int** value returned by the function could be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable c would be declared as **int**.

5 **EXAMPLE 2** In the fragment:

[123] The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

[124] The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **int volatile * const**).

6.6 Constant expressions

Syntax

1 *constant-expression:*
conditional-expression

Description

2 A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[126)]

4 Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Semantics

5 An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic range and precision shall be at least as great as if the expression were being evaluated in the execution environment.[127)]

6 An *integer constant expression*[128)] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, ~~_Alignof~~ **alignof** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **sizeof** or ~~_Alignof~~ **alignof** operator.

7 More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:

— an arithmetic constant expression,

— a null pointer constant,

— an address constant, or

— an address constant for a complete object type plus or minus an integer constant expression.

8 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, and ~~_Alignof~~ **alignof** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **sizeof** or ~~_Alignof~~ **alignof** operator.

9 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript [] and member-access . and -> operators, the address & and indirection * unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

10 An implementation may accept other forms of constant expressions.

[126)] The operand of a **sizeof** or **alignof** operator is usually not evaluated (6.5.3.4).

[127)] The use of evaluation formats as characterized by **FLT_EVAL_METHOD** also applies to evaluation in the translation environment.

[128)] An integer constant expression is required in a number of contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

6.7 Declarations

Syntax

1 *declaration:*
declaration-specifiers *init-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration
declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{\text{opt}}$
type-specifier *declaration-specifiers*$_{\text{opt}}$
type-qualifier *declaration-specifiers*$_{\text{opt}}$
function-specifier *declaration-specifiers*$_{\text{opt}}$
alignment-specifier *declaration-specifiers*$_{\text{opt}}$
init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*
init-declarator:
declarator
declarator **=** *initializer*

Constraints

2 A declaration other than a ~~static_assert~~ **static_assert** declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:

— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;

— tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

5 A declaration specifies the interpretation and attributes of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:

— for an object, causes storage to be reserved for that object;

— for a function, includes the function body;[130)]

— for an enumeration constant, is the (only) declaration of the identifier;

— for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared.

7 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters (including in prototypes), it is the adjusted type (see 6.7.6.3) that is required to be complete.

Forward references: declarators (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.9), type names (6.7.7), type qualifiers (6.7.3).

[130)] Function definitions have a different syntax, described in 6.9.1.

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

typedef
extern
static
~~**_Thread_local**~~ **thread_local**
auto
register

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that ~~**_Thread_local**~~ **thread_local** may appear with **static** or **extern**.[131)]

3 In the declaration of an object with block scope, if the declaration specifiers include ~~**_Thread_local**~~ **thread_local**, they shall also include either **static** or **extern**. If ~~**_Thread_local**~~ **thread_local** appears in any declaration of an object, it shall be present in every declaration of that object.

4 ~~**_Thread_local**~~ **thread_local** shall not appear in the declaration specifiers of a function declaration.

Semantics

5 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

6 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[132)]

7 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

8 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

Forward references: type definitions (6.7.8).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

void
char
short
int
long
float
double
signed
unsigned
~~**_Bool**~~ **bool**
~~**_Complex**~~ **complex**

[131)] See "future language directions" (6.11.5).

[132)] The implementation can treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary & operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

atomic-type-specifier
struct-or-union-specifier
enum-specifier
typedef-name

Constraints

2 At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each member declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— **void**
— **char**
— **signed char**
— **unsigned char**
— **short**, **signed short**, **short int**, or **signed short int**
— **unsigned short**, or **unsigned short int**
— **int**, **signed**, or **signed int**
— **unsigned**, or **unsigned int**
— **long**, **signed long**, **long int**, or **signed long int**
— **unsigned long**, or **unsigned long int**
— **long long**, **signed long long**, **long long int**, or **signed long long int**
— **unsigned long long**, or **unsigned long long int**
— **float**
— **double**
— **long double**
— ~~**_Bool**~~ **bool**
— ~~**float _Complex**~~ **float complex**
— ~~**double _Complex**~~ **double complex**
— ~~**long double _Complex**~~ **long double complex**
— atomic type specifier
— struct or union specifier
— enum specifier
— typedef name

3 The type specifier ~~**_Complex**~~ **complex** shall not be used other than as the operand of a **defined** operator if the implementation does not support complex types (see 6.10.8.3).

Semantics

4 Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5.

5 Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **int** designates the same type as **signed int** or the same type as **unsigned int**.

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8).

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
struct-or-union *identifier*$_{\text{opt}}$ **{** *member-declaration-list* **}**
struct-or-union *identifier*
struct-or-union:
struct
union
member-declaration-list:
member-declaration
member-declaration-list *member-declaration*
member-declaration:
specifier-qualifier-list *member-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{\text{opt}}$
type-qualifier *specifier-qualifier-list*$_{\text{opt}}$
alignment-specifier *specifier-qualifier-list*$_{\text{opt}}$
member-declarator-list:
member-declarator
member-declarator-list **,** *member-declarator*
member-declarator:
declarator
declarator$_{\text{opt}}$ **:** *constant-expression*

Constraints

2 A member declaration that does not declare an anonymous structure or anonymous union shall contain a member declarator list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted.[133] If the value is zero, the declaration shall have no declarator.

5 A bit-field shall have a type that is a qualified or unqualified version of ~~**_Bool**~~ **bool**, **signed int**, **unsigned int**, or some other implementation-defined type. It is implementation-defined whether atomic types are permitted.

Semantics

6 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

7 Structure and union specifiers have the same form. The keywords **struct** and **union** indicate that the type being specified is, respectively, a structure type or a union type.

8 The presence of a member declaration list in a struct-or-union-specifier declares a new type, within a translation unit. The member declaration list is a sequence of declarations for the members of the structure or union. If the member declaration list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is

[133] While the number of bits in a **bool** object is at least **CHAR_BIT**, the width (number of sign and value bits) of a **bool** can be just 1 bit.

incomplete until immediately after the } that terminates the list, and complete thereafter.

9 A member of a structure or union may have any complete object type other than a variably modified type.[134] In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[135] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[136] If the value 0 or 1 is stored into a nonzero-width bit-field of type ~~_Bool~~ **bool**, the value of the bit-field shall compare equal to the value stored; a ~~_Bool~~ **bool** bit-field has the semantics of a ~~_Bool~~ **bool**.

11 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[137] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

13 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

14 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

15 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

16 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

17 There may be unnamed padding at the end of a structure or union.

18 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.

[134] A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[135] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[136] As specified in 6.7.2 above, if the actual type specifier used is **int** or a typedef-name defined as **int**, then it is implementation-defined whether the bit-field is signed or unsigned.

[137] An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

15 **EXAMPLE 7** The least effective alternative is:

```
void f(int n, int * restrict p, int *q) { /* ... */ }
```

Here the translator can make the no-aliasing inference only by analyzing the body of the function and proving that `q` cannot become based on `p`. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator is required to assume that aliases are present because assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and thus need to conservatively assume that aliases are present.

6.7.4 Function specifiers

Syntax

1 *function-specifier:*

inline
~~**_Noreturn**~~ **noreturn**

Constraints

2 Function specifiers shall be used only in the declaration of an identifier for a function.

3 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static or thread storage duration, and shall not contain a reference to an identifier with internal linkage.

4 In a hosted environment, no function specifier(s) shall appear in a declaration of **main**.

Semantics

5 A function specifier may appear more than once; the behavior is the same as if it appeared only once.

6 A function declared with an **inline** function specifier is an *inline function*. Making a function an inline function suggests that calls to the function be as fast as possible.[149] The extent to which such suggestions are effective is implementation-defined.[150]

7 Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an **inline** function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the **inline** function specifier without **extern**, then the definition in that translation unit is an *inline definition*. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.[151]

8 A function declared with a ~~**_Noreturn**~~ **noreturn** function specifier shall not return to its caller.

Recommended practice

9 The implementation should produce a diagnostic message for a function declared with a ~~**_Noreturn**~~ **noreturn** function specifier that appears to be capable of returning to its caller.

10 **EXAMPLE 1** The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit.

[149] By using, for example, an alternative to the usual function call mechanism, such as "inline substitution". Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

[150] For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **inline** declaration.

[151] Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

```
inline double fahr(double t)
{
      return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
      return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double);   // creates an external definition

double convert(int is_fahr, double temp)
{
      /* A translator may perform inline substitutions */
      return is_fahr ? cels(temp): fahr(temp);
}
```

11 Note that the definition of `fahr` is an external definition because `fahr` is also declared with **`extern`**, but the definition of `cels` is an inline definition. Because `cels` has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either can be used for the call.

12 **EXAMPLE 2**

```
~~_Noreturn void f () {~~
noreturn void f () {
      abort(); // ok
}

~~_Noreturn void g (int i) { // causes undefined behavior if i <= 0~~
noreturn void g (int i) { // causes undefined behavior if i <= 0
      if (i  >  0) abort();
}
```

Forward references: function definitions (6.9.1).

6.7.5 Alignment specifier

Syntax

1 *alignment-specifier:*

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

Constraints

2 An alignment specifier shall appear only in the declaration specifiers of a declaration, or in the *specifier-qualifier* list of a member declaration, or in the type name of a compound literal. An alignment specifier shall not be used in conjunction with either of the storage-class specifiers **`typedef`** or **`register`**, nor in a declaration of a function or bit-field.

3 The constant expression shall be an integer constant expression. It shall evaluate to a valid fundamental alignment, or to a valid extended alignment supported by the implementation for an object of the storage duration (if any) being declared, or to zero.

4 An object shall not be declared with an over-aligned type with an extended alignment requirement not supported by the implementation for an object of that storage duration.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the type of the object or member being declared.

Semantics

6 The first form is equivalent to ~~`_Alignas(_Alignof(`~~ `alignas(alignof(`*type-name*`))`.

7 The alignment requirement of the declared object or member is taken to be the specified alignment. An alignment specification of zero has no effect.[152] When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

8 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.6 Declarators

Syntax

1 *declarator:*
pointer$_{\text{opt}}$ *direct-declarator*

direct-declarator:
identifier
`(` *declarator* `)`
direct-declarator `[` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ `]`
direct-declarator `[` `static` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* `]`
direct-declarator `[` *type-qualifier-list* `static` *assignment-expression* `]`
direct-declarator `[` *type-qualifier-list*$_{\text{opt}}$ `*` `]`
direct-declarator `(` *parameter-type-list* `)`
direct-declarator `(` *identifier-list*$_{\text{opt}}$ `)`

pointer:
`*` *type-qualifier-list*$_{\text{opt}}$
`*` *type-qualifier-list*$_{\text{opt}}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list *type-qualifier*

parameter-type-list:
parameter-list
parameter-list `,` `...`

parameter-list:
parameter-declaration
parameter-list `,` *parameter-declaration*

parameter-declaration:
declaration-specifiers *declarator*
declaration-specifiers *abstract-declarator*$_{\text{opt}}$

identifier-list:
identifier
identifier-list `,` *identifier*

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

[152] An alignment specification of zero also does not affect other alignment specifications in the same declaration.

6.7.6.2 Array declarators

Constraints

1 In addition to optional type qualifiers and the keyword **static**, the [and] may delimit an expression or ∗. If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

2 If an identifier is declared as having a variably modified type, it shall be an ordinary identifier (as defined in 6.2.3), have no linkage, and have either block scope or function prototype scope. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

Semantics

3 If, in the declaration "`T D1`", `D1` has one of the forms:

`D [` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression*$_{\text{opt}}$ `]`
`D [` *type-qualifier-list*$_{\text{opt}}$ *assignment-expression* `]`
`D [` *type-qualifier-list* **static** *assignment-expression* `]`
`D [` *type-qualifier-list*$_{\text{opt}}$ `*` `]`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[153] (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword **static**.)

4 If the size is not present, the array type is an incomplete type. If the size is ∗ instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations or type names with function prototype scope;[154] such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. (Variable length arrays are a conditional feature that implementations need not support; see 6.10.8.3.)

5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by ∗; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of an ~~_Alignof~~ **alignof** operator, that expression is not evaluated.

6 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

7 **EXAMPLE 1**

```
float fa[11], *afp[17];
```

declares an array of **float** numbers and an array of pointers to **float** numbers.

8 **EXAMPLE 2** Note the distinction between the declarations

```
extern int *x;
extern int y[];
```

The first declares `x` to be a pointer to **int**; the second declares `y` to be an array of **int** of unspecified size (an incomplete type), the storage for which is defined elsewhere.

[153] When several "array of" specifications are adjacent, a multidimensional array is declared.
[154] Thus, ∗ can be used only in function declarations that are not definitions (see 6.7.6.3).

9 **EXAMPLE 3** The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;

void fcompat(void)
{
      int a[n][6][m];
      int (*p)[4][n+1];
      int c[n][n][6][m];
      int (*r)[n][n][n+1];
      p = a;      // invalid:  not compatible because 4 != 6
      r = c;      // compatible, but defined behavior only if
                  // n == 6 and m == n+1
}
```

10 **EXAMPLE 4** All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the ~~_Thread_local~~ `thread_local`, **static**, or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```
extern int n;
int A[n];                           // invalid:  file scope VLA
extern int (*p2)[n];                // invalid:  file scope VM
int B[100];                         // valid:  file scope but not VM

void fvla(int m, int C[m][m]);      // valid:  VLA with prototype scope

void fvla(int m, int C[m][m])       // valid:  adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];        // valid:  block scope typedef VLA

      struct tag {
            int (*y)[n];            // invalid:  y not ordinary identifier
            int z[n];               // invalid:  z not ordinary identifier
      };
      int D[m];                     // valid:  auto VLA
      static int E[m];              // invalid:  static block scope VLA
      extern int F[m];              // invalid:  F has linkage and is VLA
      int (*s)[m];                  // valid:  auto pointer to VLA
      extern int (*r)[m];           // invalid:  r has linkage and points to VLA
      static int (*q)[m] = &B;      // valid:  q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.9).

6.7.6.3 Function declarators (including prototypes)

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

2 The only storage-class specifier that shall occur in a parameter declaration is **register**.

3 An identifier list in a function declarator that is not part of a definition of that function shall be empty.

4 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

Semantics

5 If, in the declaration "`T D1`", `D1` has the form

`D (` *parameter-type-list* `)`

```
union { /* ... */ } u = {.any_member = 42 };
```

Forward references: common definitions <stddef.h> (7.19).

6.7.10 Static assertions

Syntax

1 *static_assert-declaration:*

~~_Static_assert~~ static_assert (*constant-expression* , *string-literal*) ;
~~_Static_assert~~ static_assert (*constant-expression*) ;

Constraints

2 The constant expression shall compare unequal to 0.

Semantics

3 The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, if present, except that characters not in the basic source character set are not required to appear in the message.

Forward references: diagnostics (7.2).

6.9 External definitions

Syntax

1 *translation-unit:*
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

Constraints

2 The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** or ~~**_Alignof**~~ **alignof** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** or ~~**_Alignof**~~ **alignof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[172)]

6.9.1 Function definitions

Syntax

1 *function-definition:*
declaration-specifiers declarator declaration-list$_{\text{opt}}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[173)]

3 The return type of a function shall be **void** or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier. No declaration list shall follow.

[172)] Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

 # error *pp-tokens*$_{\text{opt}}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

6.10.6 Pragma directive

Semantics

1 A preprocessing directive of the form

 # pragma *pp-tokens*$_{\text{opt}}$ *new-line*

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)[186] causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored.

2 If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[187] whose meanings are described elsewhere:

 # pragma STDC FP_CONTRACT *on-off-switch*
 # pragma STDC FENV_ACCESS *on-off-switch*
 # pragma STDC CX_LIMITED_RANGE *on-off-switch*

on-off-switch: one of
ON OFF DEFAULT

Forward references: the **FP_CONTRACT** pragma (7.12.2), the **FENV_ACCESS** pragma (7.6.1), the **CX_LIMITED_RANGE** pragma (7.3.4).

6.10.7 Null directive

Semantics

1 A preprocessing directive of the form

*new-line*

has no effect.

6.10.8 Predefined macro names

1 The values of the predefined macros listed in the following subclauses[188] (except for **__FILE__** and **__LINE__**) remain constant throughout the translation unit.

2 None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

3 The implementation shall not predefine the macro **__cplusplus**, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

1 ~~The~~ In addition to the keywords

[186] An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.
[187] See "future language directions" (6.11.8).
[188] See "future language directions" (6.11.9).

alignas	**bool**	**noreturn**	**static_assert**	**true**
alignof	**false**	**nullptr**	**thread_local**	

which are object-like macros that expand to unspecified tokens, the following macro names shall be defined by the implementation:

__DATE__ The date of translation of the preprocessing translation unit: a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the **asctime** function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

__FILE__ The presumed name of the current source file (a character string literal).[189)]

__LINE__ The presumed line number (within the current source file) of the current source line (an integer constant).[189)]

__STDC__ The integer constant `1`, intended to indicate a conforming implementation.

__STDC_HOSTED__ The integer constant `1` if the implementation is a hosted implementation or the integer constant `0` if it is not.

__STDC_VERSION__ The integer constant *yyyymm*L.[190)]

__TIME__ The time of translation of the preprocessing translation unit: a character string literal of the form `"hh:mm:ss"` as in the time generated by the **asctime** function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the **asctime** function (7.27.3.1).

6.10.8.2 Environment macros

1 The following macro names are conditionally defined by the implementation:

__STDC_ISO_10646__ An integer constant of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **wchar_t**, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

__STDC_MB_MIGHT_NEQ_WC__ The integer constant `1`, intended to indicate that, in the encoding for **wchar_t**, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.

__STDC_UTF_16__ The integer constant `1`, intended to indicate that values of type **char16_t** are UTF–16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

__STDC_UTF_32__ The integer constant `1`, intended to indicate that values of type **char32_t** are UTF–32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

Forward references: common definitions (7.19), unicode utilities (7.28).

[189)] The presumed source file name and line number can be changed by the **#line** directive.

[190)] See Annex M for the values in previous revisions. The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this document.

6.10.8.3 Conditional feature macros

1 The following macro names are conditionally defined by the implementation:

complex and **_Complex_I** are defined to unspecified values if and only if complex types are provided.

imaginary and **_Imaginary_I** are defined to unspecified values if and only if the implementation adheres to the specifications in G.1 to G.5 (IEC 60559 compatible complex arithmetic) providing imaginary types.

__STDC_ANALYZABLE__ The integer constant `1`, intended to indicate conformance to the specifications in Annex L (Analyzability).

__STDC_IEC_60559_BFP__ The integer constant *yyyymm*`L`, intended to indicate conformance to Annex F (IEC 60559 binary floating-point arithmetic).

__STDC_IEC_559__ The integer constant `1`, intended to indicate conformance to the specifications in Annex F (IEC 60559 floating-point arithmetic). Use of this macro is an obsolescent feature.

__STDC_IEC_60559_COMPLEX__ The integer constant *yyyymm*`L`, intended to indicate conformance to the specifications in Annex G (IEC 60559 compatible complex arithmetic).

__STDC_IEC_559_COMPLEX__ The integer constant `1`, intended to indicate adherence to the specifications in Annex G (IEC 60559 compatible complex arithmetic). Use of this macro is an obsolescent feature.

__STDC_LIB_EXT1__ The integer constant *yyyymm*`L`, intended to indicate support for the extensions defined in Annex K (Bounds-checking interfaces).[191)]

__STDC_NO_ATOMICS__ The integer constant `1`, intended to indicate that the implementation does not support atomic types (including the **_Atomic** type qualifier) and the `<stdatomic.h>` header.

__STDC_NO_COMPLEX__ The integer constant `1`, intended to indicate that the implementation does not support complex types or the `<complex.h>` header.

__STDC_NO_THREADS__ The integer constant `1`, intended to indicate that the implementation does not support the `<threads.h>` header.

__STDC_NO_VLA__ The integer constant `1`, intended to indicate that the implementation does not support variable length arrays or variably modified types.

2 An implementation that does not define **complex** shall define **__STDC_NO_COMPLEX__**. An implementation that defines **__STDC_NO_COMPLEX__** shall not define **imaginary**, **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__**.

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:

[191)] The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this document.

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.[204] It then calls the **abort** function.

Returns

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

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

[204] The message written might be of the form:

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

7.3 Complex arithmetic <complex.h>

7.3.1 Introduction

1 The header <complex.h> defines macros and declares functions that support complex arithmetic.[205)]

2 Implementations that define the macro __STDC_NO_COMPLEX__ need not provide this header nor support any of its facilities.

3 Each synopsis, other than for the **CMPLX** macros, specifies a family of functions consisting of a principal function with one or more **double complex** parameters and a **double complex** or **double** return value; and other functions with the same name but with **f** and **l** suffixes which are corresponding functions with **float** and **long double** parameters and return values.

4 The macro

```
I
```

expands to either **_Imaginary_I** or **_Complex_I**. If **_Imaginary_I** is not defined, **I** shall expand to **_Complex_I**.

5 Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the ~~macros **complex**, **imaginary**, and~~ macro **I**.

Forward references: the **CMPLX** macros (7.3.9.3), IEC 60559-compatible complex arithmetic (Annex G).

7.3.2 Conventions

1 Values are interpreted as radians, not degrees. An implementation may set **errno** but is not required to.

7.3.3 Branch cuts

1 Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications of Annex G, the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part `+0`, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part `-0`, maps to the negative imaginary axis.

2 Implementations that do not support a signed zero (see Annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

7.3.4 The CX_LIMITED_RANGE pragma

Synopsis

1

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Description

2 The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The **CX_LIMITED_RANGE** pragma can be used to inform the implementation that (where the state is "on")

[205)] See "future library directions" (7.31.1).

Returns

3 The **copysign** functions return a value with the magnitude of x and the sign of y.

7.12.11.2 The nan functions

Synopsis

1
```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

2 The **nan**, **nanf**, and **nanl** functions convert the string pointed to by tagp according to the following rules. The call **nan**("*n-char-sequence*") is equivalent to **strtod**("NAN(*n-char-sequence*)", ~~(char**)NULL)~~ nullptr); the call **nan**("") is equivalent to ~~**strtod**("NAN()",(char**)NULL)~~ **strtod**("NAN()", nullptr). If tagp does not point to an n-char sequence or an empty string, the call is equivalent to ~~**strtod**("NAN",(char**)NULL)~~ **strtod**("NAN", nullptr). Calls to **nanf** and **nanl** are equivalent to the corresponding calls to **strtof** and **strtold**.

Returns

3 The **nan** functions return a quiet NaN, if available, with content indicated through tagp. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.4).

7.12.11.3 The nextafter functions

Synopsis

1
```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Description

2 The **nextafter** functions determine the next representable value, in the type of the function, after x in the direction of y, where x and y are first converted to the type of the function.[254] The **nextafter** functions return y if x equals y. A range error may occur if the magnitude of x is the largest finite value representable in the type and the result is infinite or not representable in the type.

Returns

3 The **nextafter** functions return the next representable value in the specified format after x in the direction of y.

7.12.11.4 The nexttoward functions

Synopsis

1
```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Description

2 The **nexttoward** functions are equivalent to the **nextafter** functions except that the second parameter has type **long double** and the functions return y converted to the type of the function if x equals y.[255]

[254] The argument values are converted to the type of the function, even by a macro implementation of the function.

[255] The result of the **nexttoward** functions is determined in the type of the function, without loss of range or precision in a floating second argument.

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.[275)]

7.17.6 Atomic integer types

1 For each line in the following table,[276)] the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.[277)]

Atomic type name	Direct type
atomic_bool	~~**_Atomic _Bool**~~ **_Atomic bool**
atomic_char	**_Atomic char**
atomic_schar	**_Atomic signed char**
atomic_uchar	**_Atomic unsigned char**
atomic_short	**_Atomic short**
atomic_ushort	**_Atomic unsigned short**
atomic_int	**_Atomic int**
atomic_uint	**_Atomic unsigned int**
atomic_long	**_Atomic long**
atomic_ulong	**_Atomic unsigned long**
atomic_llong	**_Atomic long long**
atomic_ullong	**_Atomic unsigned long long**
atomic_char16_t	**_Atomic char16_t**
atomic_char32_t	**_Atomic char32_t**
atomic_wchar_t	**_Atomic wchar_t**
atomic_int_least8_t	**_Atomic int_least8_t**
atomic_uint_least8_t	**_Atomic uint_least8_t**

[275)] `obj` can be a null pointer.
[276)] See "future library directions" (7.31.9).
[277)] The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

Description

2 The `order` argument shall not be **`memory_order_release`** nor **`memory_order_acq_rel`**. Memory is affected according to the value of `order`.

Returns

3 Atomically returns the value pointed to by `object`.

7.17.7.3 The `atomic_exchange` generic functions

Synopsis

1
```
      #include <stdatomic.h>
      C atomic_exchange(volatile A *object, C desired);
      C atomic_exchange_explicit(volatile A *object,
            C desired, memory_order order);
```

Description

2 Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (5.1.2.4).

Returns

3 Atomically returns the value pointed to by `object` immediately before the effects.

7.17.7.4 The `atomic_compare_exchange` generic functions

Synopsis

1
```
      #include <stdatomic.h>
~~~~~~_Bool atomic_compare_exchange_strong(volatile A *object,~~~~~
      bool atomic_compare_exchange_strong(volatile A *object,
            C *expected, C desired);
~~~~~~_Bool atomic_compare_exchange_strong_explicit(~~~~~~~~~~~~~~~
      bool atomic_compare_exchange_strong_explicit(
            volatile A *object, C *expected, C desired,
            memory_order success, memory_order failure);
~~~~~~_Bool atomic_compare_exchange_weak(volatile A *object,~~~~~~~
      bool atomic_compare_exchange_weak(volatile A *object,
            C *expected, C desired);
~~~~~~_Bool atomic_compare_exchange_weak_explicit(~~~~~~~~~~~~~~~~~
      bool atomic_compare_exchange_weak_explicit(
            volatile A *object, C *expected, C desired,
            memory_order success, memory_order failure);
```

Description

2 The `failure` argument shall not be **`memory_order_release`** nor **`memory_order_acq_rel`**. The `failure` argument shall be no stronger than the success argument.

3 Atomically, compares the contents of the memory pointed to by `object` for equality with that pointed to by `expected`, and if true, replaces the contents of the memory pointed to by `object` with `desired`, and if false, updates the contents of the memory pointed to by `expected` with that pointed to by `object`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. These operations are atomic read-modify-write operations (5.1.2.4).

4 **NOTE 1** For example, the effect of **`atomic_compare_exchange_strong`** is

```
      if (memcmp(object, expected, sizeof (*object)) == 0)
            memcpy(object, &desired, sizeof (*object));
      else
            memcpy(expected, object, sizeof (*object));
```

5 **EXAMPLE**

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

7.17.8.1 The `atomic_flag_test_and_set` functions

Synopsis

1

```
#include <stdatomic.h>
_Bool atomic_flag_test_and_set(
bool atomic_flag_test_and_set(
      volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit(
bool atomic_flag_test_and_set_explicit(
      volatile atomic_flag *object, memory_order order);
```

Description

2 Atomically places the atomic flag pointed to by `object` in the set state and returns the value corresponding to the immediately preceding state. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

3 The **`atomic_flag_test_and_set`** functions return the value that corresponds to the state of the atomic flag immediately before the effects. The return value ~~true~~ **`true`** corresponds to the set state and the return value ~~false~~ **`false`** corresponds to the clear state.

7.17.8.2 The `atomic_flag_clear` functions

Synopsis

1

```
#include <stdatomic.h>
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(
      volatile atomic_flag *object, memory_order order);
```

Description

2 The `order` argument shall not be **`memory_order_acquire`** nor **`memory_order_acq_rel`**. Atomically places the atomic flag pointed to by `object` into the clear state. Memory is affected according to the value of `order`.

Returns

3 The **`atomic_flag_clear`** functions return no value.

7.18 Boolean type and values <stdbool.h>

1 The obsolescent header <stdbool.h> defines ~~four macros.~~

~~expands to **_Bool**.~~

~~Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros bool~~the following macro which is suitable for use in conditional preprocessing directives:

```
__bool_true_false_are_defined
```

It expands to the constant **true**. ~~, **true**, and **false**.~~

7.19 Common definitions <stddef.h>

1 The header <stddef.h> defines the following macros and declares the following types. Some are also defined in other headers, as noted in their respective subclauses.

2 The types are

```
ptrdiff_t
```

which is the signed integer type of the result of subtracting two pointers;

```
size_t
```

which is the unsigned integer type of the result of the **sizeof** operator;

```
max_align_t
```

which is an object type whose alignment is the greatest fundamental alignment; and

```
wchar_t
```

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. Each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant if an implementation does not define **__STDC_MB_MIGHT_NEQ_WC__**.

3 The macros are

```
NULL
```

which expands to an implementation-defined null pointer constant;[278] and

```
offsetof(type, member-designator)
```

which expands to an integer constant expression that has type **size_t**, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The type and member designator shall be such that given

```
static type t;
```

then the expression `&(t.` *member-designator*`)` evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Recommended practice

4 The types used for **size_t** and **ptrdiff_t** should not have an integer conversion rank greater than that of **signed long int** unless the implementation supports objects large enough to make this necessary.

[278] The **NULL** macro is an obsolescent feature. Because of its underspecification, **NULL** can expand to an expression with an integer type or of type **void***. In particular, **sizeof** **NULL** can be different from **sizeof**(**void***) which can lead to unexpected or undefined behavior when used as argument to variable-argument or type-generic functions, conditional operators, or in the context of a type-generic primary expression. New code should use the keyword **nullptr** instead.

Description

2 The **atof** function converts the initial portion of the string pointed to by `nptr` to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
strtod(nptr, nullptr)
```

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.4).

7.22.1.2 The atoi, atol, and atoll functions

Synopsis

1

```
#include <stdlib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by `nptr` to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
atoi: (int)strtol(nptr, (char **)NULL, 10)
atol: strtol(nptr, (char **)NULL, 10)
atoll: strtoll(nptr, (char **)NULL, 10)
atoi: (int)strtol(nptr, nullptr, 10)
atol: strtol(nptr, nullptr, 10)
atoll: strtoll(nptr, nullptr, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.5).

7.22.1.3 The strfromd, strfromf, and strfroml functions

Synopsis

1

```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdlib.h>
int strfromd(char *restrict s, size_t n, const char *restrict format, double fp);
int strfromf(char *restrict s, size_t n, const char *restrict format, float fp);
int strfroml(char *restrict s, size_t n, const char *restrict format, long double fp);
```

Description

2 The **strfromd**, **strfromf**, and **strfroml** functions are equivalent to **snprintf**`(s, n, format, fp)` (7.21.6.5), except that the format string shall only contain the character `%`, an optional precision that does not contain an asterisk `*`, and one of the conversion specifiers `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, which applies to the type (**double**, **float**, or **long double**) indicated by the function suffix (rather than by a length modifier).

Returns

The **strfromd**, **strfromf**, and **strfroml** functions return the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than `n`.

7.23 `noreturn` `<stdnoreturn.h>`

1 The obsolescent header `<stdnoreturn.h>` ~~defines the macro which expands to `_Noreturn`~~ contains no declarations or definitions.

7.24.4.4 The `strncmp` function

Synopsis

1
```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

2 The **strncmp** function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Returns

3 The **strncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

7.24.4.5 The `strxfrm` function

Synopsis

1
```
#include <string.h>
size_t strxfrm(char * restrict s1,
      const char * restrict s2,
      size_t n);
```

Description

2 The **strxfrm** function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3 The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is `n` or more, the contents of the array pointed to by `s1` are indeterminate.

4 **EXAMPLE** The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by `s`.

```
~~1 + strxfrm(NULL, s, 0)~~
1 + strxfrm(nullptr, s, 0)
```

7.24.5 Search functions

7.24.5.1 The `memchr` function

Synopsis

1
```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Description

2 The **memchr** function locates the first occurrence of `c` (converted to an **unsigned char**) in the initial `n` characters (each interpreted as **unsigned char**) of the object pointed to by `s`. The implementation shall behave as if it reads the characters sequentially and stops as soon as a matching character is found.

6 The **strtok** function is not required to avoid data races with other calls to the **strtok** function.[330] The implementation shall behave as if no library function calls the **strtok** function.

Returns

7 The **strtok** function returns a pointer to the first character of a token, or a null pointer if there is no token.

8 **EXAMPLE**

```
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

t = strtok(str, "?");         // t points to the token "a"
t = strtok(NULL, ",");        // t points to the token "??b"
t = strtok(NULL, "#,");       // t points to the token "c"
t = strtok(NULL, "?");        // t is a null pointer
t = strtok(nullptr, ",");     // t points to the token "??b"
t = strtok(nullptr, "#,");    // t points to the token "c"
t = strtok(nullptr, "?");     // t is a null pointer
```

Forward references: The **strtok_s** function (K.3.7.3.1).

7.24.6 Miscellaneous functions

7.24.6.1 The memset function

Synopsis

1
```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Description

2 The **memset** function copies the value of c (converted to an **unsigned char**) into each of the first n characters of the object pointed to by s.

Returns

3 The **memset** function returns the value of s.

7.24.6.2 The strerror function

Synopsis

1
```
#include <string.h>
char *strerror(int errnum);
```

Description

2 The **strerror** function maps the number in errnum to a message string. Typically, the values for errnum come from **errno**, but **strerror** shall map any value of type **int** to a message.

3 The **strerror** function is not required to avoid data races with other calls to the **strerror** function.[331] The implementation shall behave as if no library function calls the **strerror** function.

Returns

4 The **strerror** function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **strerror** function.

Forward references: The **strerror_s** function (K.3.7.4.2).

[330] The **strtok_s** function can be used instead to avoid data races.
[331] The **strerror_s** function can be used instead to avoid data races.

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.[335)]

2 Implementations that define the macro __STDC_NO_THREADS__ need not provide this header nor support any of its facilities.

~~which expands to _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

```
cnd_t
```

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

```
thrd_t
```

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

```
tss_t
```

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

```
mtx_t
```

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

```
tss_dtor_t
```

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

```
thrd_start_t
```

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

```
once_flag
```

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

5 The enumeration constants are

```
mtx_plain
```

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

```
mtx_recursive
```

[335)] See "future library directions" (7.31.17).

7.28 Unicode utilities <uchar.h>

1 The header <uchar.h> declares types and functions for manipulating Unicode characters.

2 The types declared are **mbstate_t** (described in 7.29.1) and **size_t** (described in 7.19);

```
char16_t
```

which is an unsigned integer type used for 16-bit characters and is the same type as **uint_least16_t** (described in 7.20.1.2); and

```
char32_t
```

which is an unsigned integer type used for 32-bit characters and is the same type as **uint_least32_t** (also described in 7.20.1.2).

7.28.1 Restartable multibyte/wide character conversion functions

1 These functions have a parameter, ps, of type pointer to **mbstate_t** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence, which the functions alter as necessary. If ps is a null pointer, each function uses its own internal **mbstate_t** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for ps.

7.28.1.1 The mbrtoc16 function

Synopsis

1
```
#include <uchar.h>
size_t mbrtoc16(char16_t * restrict pc16,
      const char * restrict s, size_t n,
      mbstate_t * restrict ps);
```

Description

2 If s is a null pointer, the **mbrtoc16** function is equivalent to the call:

```
mbrtoc16(NULL, "", 1, ps)
mbrtoc16(nullptr, "", 1, ps)
```

In this case, the values of the parameters pc16 and n are ignored.

3 If s is not a null pointer, the **mbrtoc16** function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if pc16 is not a null pointer, stores the value of the first (or only) such character in the object pointed to by pc16. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **mbrtoc16** function returns the first of the following that applies (given the current conversion state):

0 if the next n or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* n *inclusive* if the next n or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

(size_t)(−3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(size_t)(−2) if the next n bytes contribute to an incomplete (but potentially valid) multibyte character, and all n bytes have been processed (no value is stored).[344)]

(size_t)(−1) if an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **EILSEQ** is stored in **errno**, and the conversion state is unspecified.

7.28.1.2 The c16rtomb function

Synopsis

1

```
#include <uchar.h>
size_t c16rtomb(char * restrict s, char16_t c16,
      mbstate_t * restrict ps);
```

Description

2 If s is a null pointer, the **c16rtomb** function is equivalent to the call

```
c16rtomb(buf, L'\0', ps)
```

where buf is an internal buffer.

3 If s is not a null pointer, the **c16rtomb** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given or completed by c16 (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s, or stores nothing if c16 does not represent a complete character. At most **MB_CUR_MAX** bytes are stored. If c16 is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

4 The **c16rtomb** function returns the number of bytes stored in the array object (including any shift sequences). When c16 is not a valid wide character, an encoding error occurs: the function stores the value of the macro **EILSEQ** in **errno** and returns **(size_t)**(-1); the conversion state is unspecified.

7.28.1.3 The mbrtoc32 function

Synopsis

1

```
#include <uchar.h>
size_t mbrtoc32(char32_t * restrict pc32,
      const char * restrict s, size_t n,
      mbstate_t * restrict ps);
```

Description

2 If s is a null pointer, the **mbrtoc32** function is equivalent to the call:

```
mbrtoc32(NULL, "", 1, ps)
mbrtoc32(nullptr, "", 1, ps)
```

In this case, the values of the parameters pc32 and n are ignored.

3 If s is not a null pointer, the **mbrtoc32** function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is

[344)] When n has at least the value of the **MB_CUR_MAX** macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

`s1` are indeterminate.

4 **EXAMPLE** The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by `s`:

```
~~1 + wcsxfrm(NULL, s, 0)~~
1 + wcsxfrm(nullptr, s, 0)
```

7.29.4.4.5 The `wmemcmp` function

Synopsis

1
```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2,
      size_t n);
```

Description

2 The **`wmemcmp`** function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`.

Returns

3 The **`wmemcmp`** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

7.29.4.5 Wide string search functions

7.29.4.5.1 The `wcschr` function

Synopsis

1
```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

2 The **`wcschr`** function locates the first occurrence of `c` in the wide string pointed to by `s`. The terminating null wide character is considered to be part of the wide string.

Returns

3 The **`wcschr`** function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

7.29.4.5.2 The `wcscspn` function

Synopsis

1
```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

2 The **`wcscspn`** function computes the length of the maximum initial segment of the wide string pointed to by `s1` which consists entirely of wide characters *not* from the wide string pointed to by `s2`.

Returns

3 The **`wcscspn`** function returns the length of the segment.

7.29.4.5.7 The `wcstok` function

Synopsis

1

```
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1,
      const wchar_t * restrict s2,
      wchar_t ** restrict ptr);
```

Description

2 A sequence of calls to the **`wcstok`** function breaks the wide string pointed to by `s1` into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by `s2`. The third argument points to a caller-provided **`wchar_t`** pointer into which the **`wcstok`** function stores information necessary for it to continue scanning the same wide string.

3 The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by `ptr`. Subsequent calls in the sequence have a null first argument and the object pointed to by `ptr` is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by `s2` may be different from call to call.

4 The first call in the sequence searches the wide string pointed to by `s1` for the first wide character that is *not* contained in the current separator wide string pointed to by `s2`. If no such wide character is found, then there are no tokens in the wide string pointed to by `s1` and the **`wcstok`** function returns a null pointer. If such a wide character is found, it is the start of the first token.

5 The **`wcstok`** function then searches from there for a wide character that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by `s1`, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

6 In all cases, the **`wcstok`** function stores sufficient information in the pointer pointed to by `ptr` so that subsequent calls, with a null pointer for `s1` and the unmodified pointer value for `ptr`, shall start searching just past the element overwritten by a null wide character (if any).

Returns

7 The **`wcstok`** function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

8 **EXAMPLE**

```
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;

t = wcstok(str1, L"?", &ptr1);         // t points to the token L"a"
t = wcstok(NULL, L",", &ptr1);         // t points to the token L"??b"
t = wcstok(nullptr, L",", &ptr1);      // t points to the token L"??b"
t = wcstok(str2, L" \t", &ptr2);       // t is a null pointer
t = wcstok(NULL, L"#,", &ptr1);        // t points to the token L"c"
t = wcstok(NULL, L"?", &ptr1);         // t is a null pointer
t = wcstok(nullptr, L"#,", &ptr1);     // t points to the token L"c"
t = wcstok(nullptr, L"?", &ptr1);      // t is a null pointer
```

7.29.4.5.8 The `wmemchr` function

Synopsis

1

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c,
      size_t n);
```

7.29.6.2 Conversion state functions

7.29.6.2.1 The `mbsinit` function

Synopsis

1
```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

2 If `ps` is not a null pointer, the **`mbsinit`** function determines whether the referenced **`mbstate_t`** object describes an initial conversion state.

Returns

3 The **`mbsinit`** function returns nonzero if `ps` is a null pointer or if the referenced object describes an initial conversion state; otherwise, it returns zero.

7.29.6.3 Restartable multibyte/wide character conversion functions

1 These functions differ from the corresponding multibyte character functions of 7.22.7 (**`mblen`**, **`mbtowc`**, and **`wctomb`**) in that they have an extra parameter, `ps`, of type pointer to **`mbstate_t`** that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a null pointer, each function uses its own internal **`mbstate_t`** object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

2 Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.29.6.3.1 The `mbrlen` function

Synopsis

1
```
#include <wchar.h>
size_t mbrlen(const char * restrict s,
      size_t n,
      mbstate_t * restrict ps);
```

Description

2 The **`mbrlen`** function is equivalent to the call:

```
~~mbrtowc(NULL, s, n, ps != NULL ? ps: &internal)~~
mbrtowc(nullptr, s, n, ps != nullptr ? ps: &internal)
```

where `internal` is the **`mbstate_t`** object for the **`mbrlen`** function, except that the expression designated by `ps` is evaluated only once.

Returns

3 The **`mbrlen`** function returns a value between zero and `n`, inclusive, **`(size_t)`**(-2), or **`(size_t)`**(-1).

Forward references: the **`mbrtowc`** function (7.29.6.3.2).

7.29.6.3.2 The `mbrtowc` function

Synopsis

1
```
#include <wchar.h>
size_t mbrtowc(wchar_t * restrict pwc,
      const char * restrict s,
      size_t n,
      mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the **`mbrtowc`** function is equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
mbrtowc(nullptr, "", 1, ps)
```

In this case, the values of the parameters `pwc` and `n` are ignored.

3 If `s` is not a null pointer, the **`mbrtowc`** function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

4 The **`mbrtowc`** function returns the first of the following that applies (given the current conversion state):

0 if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 *and* `n` *inclusive* if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

`(size_t)(-2)` if the next `n` bytes contribute to an incomplete (but potentially valid) multibyte character, and all `n` bytes have been processed (no value is stored).[370)]

`(size_t)(-1)` if an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro **`EILSEQ`** is stored in **`errno`**, and the conversion state is unspecified.

7.29.6.3.3 The `wcrtomb` function

Synopsis

1
```
#include <wchar.h>
size_t wcrtomb(char * restrict s,
      wchar_t wc,
      mbstate_t * restrict ps);
```

Description

2 If `s` is a null pointer, the **`wcrtomb`** function is equivalent to the call

```
wcrtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer.

3 If `s` is not a null pointer, the **`wcrtomb`** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s`. At most **`MB_CUR_MAX`** bytes are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

[370)] When `n` has at least the value of the **`MB_CUR_MAX`** macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

7.31 Future library directions

1 The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

7.31.1 Complex arithmetic <complex.h>

1 The function names

```
cerf            cexpm1          clog2
cerfc           clog10          clgamma
cexp2           clog1p          ctgamma
```

and the same names suffixed with **f** or **l** may be added to the declarations in the <complex.h> header.

7.31.2 Character handling <ctype.h>

1 Function names that begin with either **is** or **to**, and a lowercase letter may be added to the declarations in the <ctype.h> header.

7.31.3 Errors <errno.h>

1 Macros that begin with **E** and a digit or **E** and an uppercase letter may be added to the macros defined in the <errno.h> header.

7.31.4 Floating-point environment <fenv.h>

1 Macros that begin with **FE_** and an uppercase letter may be added to the macros defined in the <fenv.h> header.

7.31.5 Format conversion of integer types <inttypes.h>

1 Macros that begin with either **PRI** or **SCN**, and either a lowercase letter or X may be added to the macros defined in the <inttypes.h> header.

7.31.6 Localization <locale.h>

1 Macros that begin with **LC_** and an uppercase letter may be added to the macros defined in the <locale.h> header.

7.31.7 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.

7.31.8 Alignment <stdalign.h>

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

7.31.9 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.

7.31.10 Common definitions <stddef.h>

1 The macro **NULL** is an obsolescent feature.

7.31.11 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.

7.31.12 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.

7.31.13 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.

7.31.14 General utilities <stdlib.h>

1 Function names that begin with **str** 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.

7.31.15 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.

7.31.16 Date and time <time.h>

Macros beginning with **TIME_** and an uppercase letter may be added to the macros in the <time.h> header.

7.31.17 Threads <threads.h>

1 Function names, type names, and enumeration constants that begin with either **cnd_**, **mtx_**, **thrd_**, or **tss_**, and a lowercase letter may be added to the declarations in the <threads.h> header.

7.31.18 Extended multibyte and wide character utilities <wchar.h>

1 Function names that begin with **wcs** and a lowercase letter may be added to the declarations in the <wchar.h> header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.

7.31.19 Wide character classification and mapping utilities <wctype.h>

1 Function names that begin with **is** or **to** and a lowercase letter may be added to the declarations in the <wctype.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	**extern**	**short**	~~**_Alignas**~~
alignof	**false**	**signed**	~~**_Alignof**~~
auto	**float**	**sizeof**	**_Atomic**
bool	**for**	**static**	~~**_Bool**~~
break	**goto**	**static_assert**	~~**_Complex**~~
case	**if**	**struct**	**_Generic**
char	**imaginary**	**switch**	~~**_Imaginary**~~
complex	**inline**	**thread_local**	**_Complex_I**
const	**int**	**true**	~~**_Noreturn**~~
continue	**long**	**typedef**	~~**_Static_assert**~~
default	**noreturn**	**union**	~~**_Thread_local**~~
do	**nullptr**	**unsigned**	**_Imaginary_I**
double	**register**	**void**	
else	**restrict**	**volatile**	
enum	**return**	**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:*

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

(6.4.3) *hex-quad:*

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

A.1.5 Constants

(6.4.4) *constant:*

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

(6.4.4.1) *integer-constant:*

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

(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

(6.4.4.1) *integer-suffix:*
unsigned-suffix *long-suffix*$_{\text{opt}}$
unsigned-suffix *long-long-suffix*
long-suffix *unsigned-suffix*$_{\text{opt}}$
long-long-suffix *unsigned-suffix*$_{\text{opt}}$

(6.4.4.1) *unsigned-suffix:* one of
u U

(6.4.4.1) *long-suffix:* one of
l L

(6.4.4.1) *long-long-suffix:* one of
ll LL

(6.4.4.2) *floating-constant:*
decimal-floating-constant
hexadecimal-floating-constant

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

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

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

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

(6.4.4.2) *sign:* one of
+ -

(6.4.4.2) *digit-sequence:*
digit
digit-sequence *digit*

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

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

(6.4.4.2) *hexadecimal-digit-sequence:*
hexadecimal-digit
hexadecimal-digit-sequence *hexadecimal-digit*

(6.4.4.2) *floating-suffix:* one of
f l F L

(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:* one of
`false` **`nullptr`** **`true`** **`_Complex_I`** **`_Imaginary_I`**

A.1.6 String literals

(6.4.5) *string-literal:*
encoding-prefix$_{\text{opt}}$ `"` *s-char-sequence*$_{\text{opt}}$ `"`

(6.4.5) *encoding-prefix:*
`u8`
`u`
`U`
`L`

(6.4.5) *s-char-sequence:*
s-char
s-char-sequence s-char

(6.4.5) *s-char:*
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

A.1.7 Punctuators

(6.4.6) *punctuator:* one of

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

A.1.8 Header names

(6.4.7) *header-name:*
< *h-char-sequence* **>**
" *q-char-sequence* **"**

(6.4.7) *h-char-sequence:*
h-char
h-char-sequence h-char

(6.4.7) *h-char:*
any member of the source character set except
the new-line character and >

(6.4.7) *q-char-sequence:*
q-char
q-char-sequence q-char

(6.4.7) *q-char:*
any member of the source character set except
the new-line character and **"**

A.1.9 Preprocessing numbers

(6.4.8) *pp-number:*
digit
. *digit*
pp-number digit
pp-number identifier-nondigit
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **p** *sign*
pp-number **P** *sign*
pp-number **.**

A.2 Phrase structure grammar

A.2.1 Expressions

(6.5.1) *primary-expression:*
identifier
constant
string-literal
(*expression* **)**
generic-selection

(6.5.1.1) *generic-selection:*
_Generic (*assignment-expression* **,** *generic-assoc-list* **)**

(6.5.1.1) *generic-assoc-list:*
generic-association
generic-assoc-list **,** *generic-association*

(6.5.1.1) *generic-association:*
type-name **:** *assignment-expression*
default : *assignment-expression*

(6.5.2) *postfix-expression:*
primary-expression
postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{\text{opt}}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**
(*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*

(6.5.6) *additive-expression:*
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

(6.5.7) *shift-expression:*
additive-expression
shift-expression **«** *additive-expression*
shift-expression **»** *additive-expression*
(6.5.8) *relational-expression:*
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*
(6.5.9) *equality-expression:*
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

(6.5.10) *AND-expression:*
equality-expression
AND-expression **&** *equality-expression*
(6.5.11) *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression **^** *AND-expression*
(6.5.12) *inclusive-OR-expression:*
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*
(6.5.13) *logical-AND-expression:*
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*
(6.5.14) *logical-OR-expression:*
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*
(6.5.15) *conditional-expression:*
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

(6.5.16) *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

(6.5.16) *assignment-operator:* one of
= *= /= %= += -= <<= >>= &= ^= |=
(6.5.17) *expression:*
assignment-expression
expression **,** *assignment-expression*
(6.6) *constant-expression:*
conditional-expression

A.2.2 Declarations

(6.7) *declaration:*
declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**
static_assert-declaration

Annex B
(informative)
Library summary

B.1 Diagnostics <assert.h>

```
NDEBUG
static_assert

void assert(scalar expression);
```

B.2 Complex <complex.h>

__STDC_NO_COMPLEX__
complex
_Complex_I
imaginary
_Imaginary_I
I

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
double complex cexp(double complex z);
float complex cexpf(float complex z);
```

```
__STDC_WANT_IEC_60559_BFP_EXT__

int totalorder(double x, double y);
int totalorderf(float x, float y);
int totalorderl(long double x, long double y);
int totalordermag(double x, double y);
int totalordermagf(float x, float y);
int totalordermagl(long double x, long double y);
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
int setpayload(double *res, double pl);
int setpayloadf(float *res, float pl);
int setpayloadl(long double *res, long double pl);
int setpayloadsig(double *res, double pl);
int setpayloadsigf(float *res, float pl);
int setpayloadsigl(long double *res, long double pl);
```

B.12 Nonlocal jumps <setjmp.h>

jmp_buf

```
jmp_buf

int setjmp(jmp_buf env);
_Noreturn void longjmp(jmp_buf env, int val);
noreturn void longjmp(jmp_buf env, int val);
```

B.13 Signal handling <signal.h>

sig_atomic_t **SIG_IGN** **SIGILL** **SIGTERM**
SIG_DFL **SIGABRT** **SIGINT**
SIG_ERR **SIGFPE** **SIGSEGV**

```
void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```

B.14 Alignment <stdalign.h>

alignas **alignof** **__alignas_is_defined** **__alignof_is_defined**

B.15 Variable arguments <stdarg.h>

```
va_list

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, parmN);
```

B.16 Atomics <stdatomic.h>

```
__STDC_NO_ATOMICS__          memory_order_seq_cst        atomic_uint_least16_t
ATOMIC_BOOL_LOCK_FREE        atomic_bool                 atomic_int_least32_t
ATOMIC_CHAR_LOCK_FREE        atomic_char                 atomic_uint_least32_t
ATOMIC_CHAR16_T_LOCK_FREE    atomic_schar                atomic_int_least64_t
ATOMIC_CHAR32_T_LOCK_FREE    atomic_uchar                atomic_uint_least64_t
ATOMIC_WCHAR_T_LOCK_FREE     atomic_short                atomic_int_fast8_t
ATOMIC_SHORT_LOCK_FREE       atomic_ushort               atomic_uint_fast8_t
ATOMIC_INT_LOCK_FREE         atomic_int                  atomic_int_fast16_t
ATOMIC_LONG_LOCK_FREE        atomic_uint                 atomic_uint_fast16_t
ATOMIC_LLONG_LOCK_FREE       atomic_long                 atomic_int_fast32_t
ATOMIC_POINTER_LOCK_FREE     atomic_ulong                atomic_uint_fast32_t
ATOMIC_FLAG_INIT             atomic_llong                atomic_int_fast64_t
memory_order                 atomic_ullong               atomic_uint_fast64_t
atomic_flag                  atomic_char16_t             atomic_intptr_t
memory_order_relaxed         atomic_char32_t             atomic_uintptr_t
memory_order_consume         atomic_wchar_t              atomic_size_t
memory_order_acquire         atomic_int_least8_t         atomic_ptrdiff_t
memory_order_release         atomic_uint_least8_t        atomic_intmax_t
memory_order_acq_rel         atomic_int_least16_t        atomic_uintmax_t
```

```
#define ATOMIC_VAR_INIT(C value)
void atomic_init(volatile A *obj, C value);
type kill_dependency(type y);
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);
_Bool atomic_is_lock_free(const volatile A *obj);
bool atomic_is_lock_free(const volatile A *obj);
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
C atomic_load(const volatile A *object);
C atomic_load_explicit(const volatile A *object, memory_order order);
C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
_Bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(
bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_strong_explicit(
      volatile A *object, C *expected, C desired,
      memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(
bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
bool atomic_compare_exchange_weak_explicit(
      volatile A *object, C *expected, C desired,
      memory_order success, memory_order failure);
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
_Bool atomic_flag_test_and_set(volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit(
bool atomic_flag_test_and_set(volatile atomic_flag *object);
bool atomic_flag_test_and_set_explicit(
      volatile atomic_flag *object, memory_order order);
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *object, memory_order order);
```

B.17 Boolean type and values <stdbool.h>

```
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs, size_t n);
```

```
__STDC_WANT_LIB_EXT1__
errno_t
rsize_t
constraint_handler_t

constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
void abort_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);
void ignore_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);
errno_t getenv_s(
      size_t *restrict len, char *restrict value, rsize_t maxsize,
      const char *restrict name);
void *bsearch_s(
      const void *key, const void *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *k, const void *y, void *context),
      void *context);
errno_t qsort_s(
      void *base, rsize_t nmemb, rsize_t size,
      int (*compar)(const void *x, const void *y, void *context),
      void *context);
errno_t wctomb_s(int *restrict status, char *restrict s, rsize_t smax, wchar_t wc);
errno_t mbstowcs_s(
      size_t *restrict retval, wchar_t *restrict dst, rsize_t dstmax,
       const char *restrict src, rsize_t len);
errno_t wcstombs_s(
      size_t *restrict retval, char *restrict dst, rsize_t dstmax,
      const wchar_t *restrict src, rsize_t len);
```

B.22 `noreturn` <stdnoreturn.h>

~~`noreturn`~~

This header is empty.

B.23 String handling <string.h>

`size_t` `NULL`

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *restrict s1, const char *restrict s2);
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
char *strcat(char *restrict s1, const char *restrict s2);
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
void *memchr(const void *s, int c, size_t n);
```

```
      __STDC_WANT_IEC_60559_BFP_EXT__

        totalorder
        totalordermag
```

B.25 Threads <threads.h>

```
__STDC_NO_THREADS__
~~thread_local~~
ONCE_FLAG_INIT
TSS_DTOR_ITERATIONS
cnd_t
thrd_t
tss_t
mtx_t
tss_dtor_t
thrd_start_t
once_flag
mtx_plain
mtx_recursive
mtx_timed
thrd_timedout
thrd_success
thrd_busy
thrd_error
thrd_nomem
```

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
      const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
~~_Noreturn void thrd_exit(int res);~~
noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
void thrd_yield(void);
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

B.26 Date and time <time.h>

```
NULL
CLOCKS_PER_SEC
TIME_UTC
size_t
clock_t
time_t
struct timespec
struct tm
```

```
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
int timespec_get(timespec *ts, int base);
```

Annex G
(normative)
IEC 60559-compatible complex arithmetic

G.1 Introduction

1 This annex supplements Annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. An implementation that defines **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__** shall conform to the specifications in this annex.[401] An implementation that defines **imaginary** shall conform to the specifications in clauses G.1 to G.5.

G.2 Types

1 There is a new keyword ~~**_Imaginary**~~and macro **imaginary**, which is used to specify and test for imaginary types. It is used as a type specifier within declaration specifiers in the same way as ~~**_Complex**~~ **complex** is (thus, ~~**_Imaginary float**~~ **imaginary float** is a valid type name).

2 There are three *imaginary types*, designated as ~~**float _Imaginary,**~~ **float imaginary**, ~~**double _Imaginary,**~~ **double imaginary**, and **long double** ~~**_Imaginary.**~~ **imaginary**. The imaginary types (along with the real floating and complex types) are floating types.

3 For imaginary types, the corresponding real type is given by deleting the keyword ~~**_Imaginary**~~ **imaginary** from the type name.

4 Each imaginary type has the same representation and alignment requirements as the corresponding real type. The value of an object of imaginary type is the value of the real representation times the imaginary unit.

5 The *imaginary type domain* comprises the imaginary types.

G.3 Conventions

1 A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a quiet NaN). A complex or imaginary value is a *finite number* if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a *zero* if each of its parts is a zero.

G.4 Conversions

G.4.1 Imaginary types

1 Conversions among imaginary types follow rules analogous to those for real floating types.

G.4.2 Real and imaginary

1 When a value of imaginary type is converted to a real type other than ~~**_Bool**~~ **bool**,[402] the result is a positive zero.

2 When a value of real type is converted to an imaginary type, the result is a positive imaginary zero.

G.4.3 Imaginary and complex

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

2 When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for the corresponding real types.

[401] Implementations that do not define **imaginary**, **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__** are not required to conform to these specifications. The use of **__STDC_IEC_559_COMPLEX__** as a feature test macro is obsolescent and should be avoided in new code.

[402] See 6.3.1.2.

G.5 Binary operators

1 The following subclauses supplement 6.5 in order to specify the type of the result for an operation with an imaginary operand.

2 For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow; in these cases the result satisfies certain properties (specified in G.5.1), but is not completely determined.

G.5.1 Multiplicative operators

Semantics

1 If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has real type. (If either operand has complex type, then the result has complex type.)

2 If the operands are not both complex, then the result and floating-point exception behavior of the $*$ operator is defined by the usual mathematical formula:

$*$	u	iv	$u + iv$
x	xu	$i(xv)$	$(xu) + i(xv)$
iy	$i(yu)$	$(-y)v$	$((-y)v) + i(yu)$
$x + iy$	$(xu) + i(yu)$	$((-y)v) + i(xv)$	

3 If the second operand is not complex, then the result and floating-point exception behavior of the / operator is defined by the usual mathematical formula:

/	u	iv
x	x/u	$i((-x)/v)$
iy	$i(y/u)$	y/v
$x + iy$	$(x/u) + i(y/u)$	$(y/v) + i((-x)/v)$

4 The $*$ and / operators satisfy the following infinity properties for all real, imaginary, and complex operands:[403)]

— if one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the $*$ operator is an infinity;

— if the first operand is an infinity and the second operand is a finite number, then the result of the / operator is an infinity;

— if the first operand is a finite number and the second operand is an infinity, then the result of the / operator is a zero;

— if the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the / operator is an infinity.

5 If both operands of the $*$ operator are complex or if the second operand of the / operator is complex, the operator raises floating-point exceptions if appropriate for the calculation of the parts of the result, and may raise spurious floating-point exceptions.

6 **EXAMPLE 1** Multiplication of ~~`double _Complex`~~ `double complex` operands could be implemented as follows. Note that the imaginary unit `I` has imaginary type (see G.6).

```
#include <math.h>
#include <complex.h>

/* Multiply z * w ...*/
double complex _Cmultd(double complex z, double complex w)
```

[403)] These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for `CX_LIMITED_RANGE` is "off").

```
{
      #pragma STDC FP_CONTRACT OFF
      double a, b, c, d, ac, bd, ad, bc, x, y;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      ac = a * c;   bd = b * d;
      ad = a * d;   bc = b * c;
      x = ac - bd;  y = ad + bc;
      if (isnan(x) && isnan(y)) {
            /* Recover infinities that computed as NaN+iNaN ... */
            int recalc = 0;
            if (isinf(a) || isinf(b)) { // z is infinite
                  /* "Box" the infinity and change NaNs in the other factor to 0 */
                  a = copysign(isinf(a) ? 1.0: 0.0, a);
                  b = copysign(isinf(b) ? 1.0: 0.0, b);
                  if (isnan(c)) c = copysign(0.0, c);
                  if (isnan(d)) d = copysign(0.0, d);
                  recalc = 1;
            }
            if (isinf(c) || isinf(d)) { // w is infinite
                  /* "Box" the infinity and change NaNs in the other factor to 0 */
                  c = copysign(isinf(c) ? 1.0: 0.0, c);
                  d = copysign(isinf(d) ? 1.0: 0.0, d);
                  if (isnan(a)) a = copysign(0.0, a);
                  if (isnan(b)) b = copysign(0.0, b);
                  recalc = 1;
            }
            if (!recalc && (isinf(ac) || isinf(bd) ||
                        isinf(ad) || isinf(bc))) {
                  /* Recover infinities from overflow by changing NaNs to 0 ... */
                  if (isnan(a)) a = copysign(0.0, a);
                  if (isnan(b)) b = copysign(0.0, b);
                  if (isnan(c)) c = copysign(0.0, c);
                  if (isnan(d)) d = copysign(0.0, d);
                  recalc = 1;
            }
            if (recalc) {
                  x = INFINITY * (a * c - b * d);
                  y = INFINITY * (a * d + b * c);
            }
      }
      return x + I * y;
}
```

7 This implementation achieves the required treatment of infinities at the cost of only one **isnan** test in ordinary (finite) cases. It is less than ideal in that undue overflow and underflow could occur.

8 **EXAMPLE 2** Division of two ~~**double _Complex**~~ **double complex** operands could be implemented as follows.

```
#include <math.h>
#include <complex.h>

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
      #pragma STDC FP_CONTRACT OFF
      double a, b, c, d, logbw, denom, x, y;
      int ilogbw = 0;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      logbw = logb(fmax(fabs(c), fabs(d)));
      if (isfinite(logbw)) {
```

```
                ilogbw = (int)logbw;
                c = scalbn(c, -ilogbw); d = scalbn(d, -ilogbw);
        }
        denom = c * c + d * d;
        x = scalbn((a * c + b * d) / denom, -ilogbw);
        y = scalbn((b * c - a * d) / denom, -ilogbw);

        /* Recover infinities and zeros that computed as NaN+iNaN;          */
        /* the only cases are nonzero/zero, infinite/finite, and finite/infinite, ...    */

        if (isnan(x) && isnan(y)) {
                if ((denom == 0.0) &&
                    (!isnan(a) || !isnan(b))) {
                        x = copysign(INFINITY, c) * a;
                        y = copysign(INFINITY, c) * b;
                }
                else if ((isinf(a) || isinf(b)) &&
                    isfinite(c) && isfinite(d)) {
                        a = copysign(isinf(a) ? 1.0: 0.0, a);
                        b = copysign(isinf(b) ? 1.0: 0.0, b);
                        x = INFINITY * (a * c + b * d);
                        y = INFINITY * (b * c - a * d);
                }
                else if ((logbw == INFINITY) &&
                    isfinite(a) && isfinite(b)) {
                        c = copysign(isinf(c) ? 1.0: 0.0, c);
                        d = copysign(isinf(d) ? 1.0: 0.0, d);
                        x = 0.0 * (a * c + b * d);
                        y = 0.0 * (b * c - a * d);
                }
        }
        return x + I * y;
}
```

9 Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the **scalbn** function, instead of with division, provides better roundoff characteristics.

G.5.2 Additive operators

Semantics

1 If both operands have imaginary type, then the result has imaginary type. (If one operand has real type and the other operand has imaginary type, or if either operand has complex type, then the result has complex type.)

2 In all cases the result and floating-point exception behavior of a + or - operator is defined by the usual mathematical formula:

+ or -	u	iv	$u + iv$
x	$x \pm u$	$x \pm iv$	$(x \pm u) \pm iv$
iy	$\pm u + iy$	$i(y \pm v)$	$\pm u + i(y \pm v)$
$x + iy$	$(x \pm u) + iy$	$x + i(y \pm v)$	$(x \pm u) + i(y \pm v)$

G.6 Complex arithmetic <complex.h>

~~are defined, respectively, as **_Imaginary** and a constant expression of type **const float _Imaginary** with the value of the imaginary unit. The macro is defined to be **_Imaginary_I** (not **_Complex_I** as stated in 7.3). Notwithstanding the provisions of 7.1.3, a program may undefine and then perhaps redefine the macro **imaginary**.~~

1 This subclause contains specifications for the <complex.h> functions that are particularly suited to IEC 60559 implementations. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol "±"

Annex H
(informative)
Language independent arithmetic

H.1 Introduction

1 This annex documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1). LIA–1 is more general than IEC 60559 (Annex F) in that it covers integer and diverse floating-point arithmetics.

H.2 Types

1 The relevant C arithmetic types meet the requirements of LIA–1 types if an implementation adds notification of exceptional arithmetic operations and meets the 1 unit in the last place (ULP) accuracy requirement (LIA–1 subclause 5.2.8).

H.2.1 Boolean type

1 The LIA–1 data type Boolean is implemented by the C data type ~~bool~~**bool** with values of **true** and **false** ~~, all from .~~.

H.2.2 Integer types

1 The signed C integer types **int**, **long int**, **long long int**, and the corresponding unsigned types are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values "integer_overflow" and "undefined", then those types are LIA–1 conformant types. C's unsigned integer types are "modulo" in the LIA–1 sense in that overflows or out-of-bounds results silently wrap. An implementation that defines signed integer types as also being modulo need not detect integer overflow, in which case, only integer divide-by-zero need be detected.

2 The parameters for the integer data types can be accessed by the following:

maxint **INT_MAX**, **LONG_MAX**, **LLONG_MAX**, **UINT_MAX**, **ULONG_MAX**, **ULLONG_MAX**

minint **INT_MIN**, **LONG_MIN**, **LLONG_MIN**

3 The parameter "bounded" is always true, and is not provided. The parameter "minint" is always 0 for the unsigned types, and is not provided for those types.

H.2.2.1 Integer operations

1 The integer operations on integer types are the following:

addI `x + y`

subI `x - y`

mulI `x * y`

divI, divtI `x / y`

remI, remtI `x % y`

negI `-x`

absI `abs(x), labs(x), llabs(x)`

eqI `x == y`

neqI `x != y`

lssI `x < y`

leqI `x <= y`

— The result of subtracting two pointers is not representable in an object of type **ptrdiff_t** (6.5.6).

— An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

— An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type (6.5.7).

— Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

— An object is assigned to an inexactly overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

— An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, ~~_Alignof~~ **alignof** expressions, or immediately-cast floating constants; or contains casts (outside operands to **sizeof** and ~~_Alignof~~ **alignof** operators) other than conversions of arithmetic types to integer types (6.6).

— A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

— An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, or ~~_Alignof~~ **alignof** expressions; or contains casts (outside operands to **sizeof** or ~~_Alignof~~ **alignof** operators) other than conversions of arithmetic types to arithmetic types (6.6).

— The value of an object is accessed by an array-subscript `[]`, member-access `.` or `->`, address `&`, or indirection `*` operator or a pointer cast in creating an address constant (6.6).

— An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

— A function is declared at block scope with an explicit storage-class specifier other than **extern** (6.7.1).

— A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

— An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

— When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).

— An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).

— The specification of a function type includes any type qualifiers (6.7.3).

— Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).

— An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).

— A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

— A function with external linkage is declared with an **inline** function specifier, but is not also defined in the same translation unit (6.7.4).

— A function declared with a ~~_Noreturn~~ **noreturn** function specifier returns to its caller (6.7.4).

— The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).

— Declarations of an object in different translation units have different alignment specifiers (6.7.5).

— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).

— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).

— In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).

— A declaration of an array parameter includes the keyword **static** within the [and] and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).

— A storage-class specifier or type qualifier modifies the keyword **void** as a function parameter type list (6.7.6.3).

— In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipsis terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.6.3).

— The value of an unnamed member of a structure or union is used (6.7.9).

— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).

— The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).

— The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.9).

— An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).

— A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1).

— An adjusted parameter type in a function definition is not a complete object type (6.9.1).

— A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

— The } that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

— An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

— How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

— Whether and how floating expressions are contracted when not disallowed by the **FP_CONTRACT** pragma (6.5).

— The default state for the **FENV_ACCESS** pragma (7.6.1).

— Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

— The default state for the **FP_CONTRACT** pragma (7.12.2).

J.3.7 Arrays and pointers

1 — The result of converting a pointer to an integer or vice versa (6.3.2.3).

— The size of the result of subtracting two pointers to elements of the same array (6.5.6).

J.3.8 Hints

1 — The extent to which suggestions made by using the **register** storage-class specifier are effective (6.7.1).

— The extent to which suggestions made by using the **inline** function specifier are effective (6.7.4).

J.3.9 Structures, unions, enumerations, and bit-fields

1 — Whether a "plain" **int** bit-field is treated as a **signed int** bit-field or as an **unsigned int** bit-field (6.7.2, 6.7.2.1).

— Allowable bit-field types other than ~~_Bool~~ **bool**, **signed int**, and **unsigned int** (6.7.2.1).

— Whether atomic types are permitted for bit-fields (6.7.2.1).

— Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

— The order of allocation of bit-fields within a unit (6.7.2.1).

— The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

— The integer type compatible with each enumerated type (6.7.2.2).

J.3.10 Qualifiers

1 — What constitutes an access to an object that has volatile-qualified type (6.7.3).

J.3.11 Preprocessing directives

1 — The locations within **#pragma** directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

— How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

— Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

— Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

— The local time zone and Daylight Saving Time (7.27.1).

— The era for the **clock** function (7.27.2.1).

— The **TIME_UTC** epoch (7.27.2.5).

— The replacement string for the %Z specifier to the **strftime**, and **wcsftime** functions in the "C" locale (7.27.3.5, 7.29.5.1).

— Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

J.3.13 Architecture

1 — The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.20.2, 7.20.3).

— The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

— The number, order, and encoding of bytes in any object (when not explicitly specified in this document) (6.2.6.1).

— Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

— Valid alignment values other than those returned by an ~~_Alignof~~ **alignof** expression for fundamental types, if any (6.2.8).

— The value of the result of the **sizeof** and ~~_Alignof~~ **alignof** operators (6.5.3.4).

J.4 Locale-specific behavior

1 The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

— Additional members of the source and execution character sets beyond the basic character set (5.2.1).

— The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

— The shift states used for the encoding of multibyte characters (5.2.1.2).

— The direction of writing of successive printing characters (5.2.2).

— The decimal-point character (7.1.1).

— The set of printing characters (7.4, 7.30.2).

— The set of control characters (7.4, 7.30.2).

— The sets of characters tested for by the **isalpha**, **isblank**, **islower**, **ispunct**, **isspace**, **isupper**, **iswalpha**, **iswblank**, **iswlower**, **iswpunct**, **iswspace**, or **iswupper** functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).

— The native environment (7.11.1.1).

— Additional subject sequences accepted by the numeric conversion functions (7.22.1, 7.29.4.1).

— The collation sequence of the execution character set (7.24.4.3, 7.29.4.4.2).

— The contents of the error message strings set up by the **strerror** function (7.24.6.2).

— The formats for time and date (7.27.3.5, 7.29.5.1).

— Character mappings that are supported by the **towctrans** function (7.30.1).

— Character classifications that are supported by the **iswctype** function (7.30.1).

J.5 Common extensions

1 The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, extra library functions declared in standard headers, or predefined macros with names that do not begin with an underscore.

J.5.1 Environment arguments

1 In a hosted environment, the **main** function receives a third argument, **char** *envp[], that points to a null-terminated array of pointers to **char**, each of which points to a string that provides information about the environment for this execution of the program (5.1.2.2.1).

J.5.2 Specialized identifiers

1 Characters other than the underscore _, letters, and digits, that are not part of the basic source character set (such as the dollar sign $, or characters in national character sets) may appear in an identifier (6.4.2).

J.5.3 Lengths and cases of identifiers

1 All characters in identifiers (with or without external linkage) are significant (6.4.2).

J.5.4 Scopes of identifiers

1 A function identifier, or the identifier of an object the declaration of which contains the keyword **extern**, has file scope (6.2.1).

J.5.5 Writable string literals

1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

J.5.6 Other arithmetic types

1 Additional arithmetic types, such as __int128 or **double double**, and their appropriate conversions are defined (6.2.5, 6.3.1). Additional floating types may have more range or precision than **long double**, may be used for evaluating expressions of other floating types, and may be used to define **float_t** or **double_t**. Additional floating types may also have less range or precision than **float**.

J.5.7 Function pointer casts

1 A pointer to an object or to **void** may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4).

2 A pointer to a function may be cast to a pointer to an object or to **void**, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).

J.5.8 Extended bit-field types

1 A bit-field may be declared with a type other than ~~_Bool~~ **bool**, **unsigned int**, or **signed int**, with an appropriate maximum width (6.7.2.1).

J.5.9 The **fortran** keyword

1 The fortran function specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that a different representation for the external name is to be generated (6.7.4).

J.5.10 The **asm** keyword

1 The asm keyword may be used to insert assembly language directly into the translator output (6.8). The most common implementation is via a statement of the form:

```
asm (character-string-literal);
```

wcsrchr
wcsrtombs
wcsrtombs_s
wcsspn
wcsstr
wcstod
wcstof
wcstoimax
wcstok
wcstok_s
wcstol
wcstold
wcstoll
wcstombs
wcstombs_s
wcstoul
wcstoull
wcstoumax
wcsxfrm
_WIDTH

J.6.2 Particular identifiers or keywords

1 The following ~~808~~ 809 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

abort
abort_handler_s
abs
acos
acosf
acosh
acoshf
acoshl
acosl
alignas
aligned_alloc
alignof
and
and_eq
asctime
asctime_s
asin
asinf
asinh
asinhf
asinhl
asinl
assert
atan
atan2
atan2f
atan2l
atanf
atanh
atanhf
atanhl
atanl
atexit
atof
atoi
atol
atoll
at_quick_exit
auto
bitand
bitor
bool
break
bsearch
bsearch_s
btowc
BUFSIZ
c16rtomb
c32rtomb
cabs
cabsf
cabsl
cacos
cacosf
cacosh
cacoshf
cacoshl
cacosl
calloc
call_once
canonicalize
canonicalizef
canonicalizel
carg
cargf
cargl
case
casin
casinf
casinh
casinhf
casinhl
casinl
catan
catanf
catanh
catanhf
catanhl
catanl
cbrt
cbrtf
cbrtl
ccos
ccosf
ccosh
ccoshf
ccoshl
ccosl
ceil
ceilf
ceill
cerf
cerfc
cexp
cexp2
cexpf
cexpl
cexpm1
char
char16_t
char32_t
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHAR_WIDTH
cimag
cimagf
cimagl
clearerr
clgamma
clock
CLOCKS_PER_SEC
clock_t
clog
clog10
clog1p
clog2
clogf
clogl
CMPLX

lgammal
line
llabs
lldiv
lldiv_t
llogb
llogbf
llogbl
LLONG_MAX
LLONG_MIN
LLONG_WIDTH
llrint
llrintf
llrintl
llround
llroundf
llroundl
localeconv
localtime
localtime_s
log
log10
log10f
log10l
log1p
log1pf
log1pl
log2
log2f
log2l
logb
logbf
logbl
logf
logl
long
longjmp
LONG_MAX
LONG_MIN
LONG_WIDTH
lrint
lrintf
lrintl
lround
lroundf
lroundl
L_tmpnam
L_tmpnam_s
main
malloc
MATH_ERREXCEPT
math_errhandling
MATH_ERRNO
max_align_t
MB_CUR_MAX
mblen
MB_LEN_MAX
mbrlen
mbrtoc16
mbrtoc32
mbrtowc
mbsinit
mbsrtowcs
mbsrtowcs_s
mbstate_t
mbstowcs
mbstowcs_s
mbtowc
mktime
modf
modff
modfl
mon_decimal_point
mon_grouping
mon_thousands_sep
nan
nanf
nanl
n_cs_precedes
NDEBUG
nearbyint
nearbyintf
nearbyintl
negative_sign
nextafter
nextafterf
nextafterl
nextdown
nextdownf
nextdownl
nexttoward
nexttowardf
nexttowardl
nextup
nextupf
nextupl
noreturn
not
not_eq
n_sep_by_space
n_sign_posn
NULL
nullptr
OFF
offsetof
ON
once_flag
ONCE_FLAG_INIT
or
or_eq
p_cs_precedes
perror
positive_sign
pow
powf
powl
pragma
printf
printf_s
p_sep_by_space
p_sign_posn
PTRDIFF_MAX
PTRDIFF_MIN
ptrdiff_t
PTRDIFF_WIDTH
putc
putchar
puts
putwc
putwchar
qsort
qsort_s
quick_exit
raise
rand
RAND_MAX
realloc
register
remainder
remainderf
remainderl
remove
remquo
remquof
remquol
rename
restrict
return
rewind
rint
rintf
rintl
round
roundeven
roundevenf
roundevenl
roundf
roundl
RSIZE_MAX
rsize_t
scalbln
scalblnf
scalblnl
scalbn
scalbnf
scalbnl
scanf
scanf_s

10 **EXAMPLE**

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
static char str1[] = "?a???b,,,#c";
static char str2[] = "\t \t";
char *t, *ptr1, *ptr2;
rsize_t max1 = sizeof (str1);
rsize_t max2 = sizeof (str2);

t = strtok_s(str1, &max1, "?", &ptr1);       // t points to the token "a"
t = strtok_s(NULL, &max1, ",", &ptr1);       // t points to the token "??b"
t = strtok_s(nullptr, &max1, ",", &ptr1);    // t points to the token "??b"
t = strtok_s(str2, &max2, " \t", &ptr2);     // t is a null pointer
t = strtok_s(NULL, &max1, "#", &ptr1);       // t points to the token "c"
t = strtok_s(NULL, &max1, "?", &ptr1);       // t is a null pointer
t = strtok_s(nullptr, &max1, "#", &ptr1);    // t points to the token "c"
t = strtok_s(nullptr, &max1, "?", &ptr1);    // t is a null pointer
```

K.3.7.4 Miscellaneous functions

K.3.7.4.1 The `memset_s` function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
```

Runtime-constraints

2 `s` shall not be a null pointer. Neither `smax` nor `n` shall be greater than **`RSIZE_MAX`**. `n` shall not be greater than `smax`.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `smax` is not greater than **`RSIZE_MAX`**, the **`memset_s`** function stores the value of `c` (converted to an **`unsigned char`**) into each of the first `smax` characters of the object pointed to by `s`.

Description

4 The **`memset_s`** function copies the value of `c` (converted to an **`unsigned char`**) into each of the first `n` characters of the object pointed to by `s`. Unlike **`memset`**, any call to the **`memset_s`** function shall be evaluated strictly according to the rules of the abstract machine as described in (5.1.2.3). That is, any call to the **`memset_s`** function shall assume that the memory indicated by `s` and `n` may be accessible in the future and thus contains the values indicated by `c`.

Returns

5 The **`memset_s`** function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.4.2 The `strerror_s` function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strerror_s(char *s, rsize_t maxsize,
      errno_t errnum);
```

Runtime-constraints

2 `s` shall not be a null pointer. `maxsize` shall not be greater than **`RSIZE_MAX`**. `maxsize` shall not equal zero.

3 If there is a runtime-constraint violation, then the array (if any) pointed to by `s` is not modified.

Description

4 A sequence of calls to the **wcstok_s** function breaks the wide string pointed to by s1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by s2. The fourth argument points to a caller-provided **wchar_t** pointer into which the **wcstok_s** function stores information necessary for it to continue scanning the same wide string.

5 The first call in a sequence has a non-null first argument and s1max points to an object whose value is the number of elements in the wide character array pointed to by the first argument. The first call stores an initial value in the object pointed to by ptr and updates the value pointed to by s1max to reflect the number of elements that remain in relation to ptr. Subsequent calls in the sequence have a null first argument and the objects pointed to by s1max and ptr are required to have the values stored by the previous call in the sequence, which are then updated. The separator wide string pointed to by s2 may be different from call to call.

6 The first call in the sequence searches the wide string pointed to by s1 for the first wide character that is *not* contained in the current separator wide string pointed to by s2. If no such wide character is found, then there are no tokens in the wide string pointed to by s1 and the **wcstok_s** function returns a null pointer. If such a wide character is found, it is the start of the first token.

7 The **wcstok_s** function then searches from there for the first wide character in s1 that *is* contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by s1, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

8 In all cases, the **wcstok_s** function stores sufficient information in the pointer pointed to by ptr so that subsequent calls, with a null pointer for s1 and the unmodified pointer value for ptr, shall start searching just past the element overwritten by a null wide character (if any).

Returns

9 The **wcstok_s** function returns a pointer to the first wide character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

10 **EXAMPLE**

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;
rsize_t max1 = wcslen(str1)+1;
rsize_t max2 = wcslen(str2)+1;

t = wcstok_s(str1, &max1, "?", &ptr1);       // t points to the token "a"
t = wcstok_s(NULL, &max1, ",", &ptr1);       // t points to the token "??b"
t = wcstok_s(nullptr, &max1, ",", &ptr1);    // t points to the token "??b"
t = wcstok_s(str2, &max2, " \t", &ptr2);     // t is a null pointer
t = wcstok_s(NULL, &max1, "#,", &ptr1);      // t points to the token "c"
t = wcstok_s(NULL, &max1, "?", &ptr1);       // t is a null pointer
t = wcstok_s(nullptr, &max1, "#,", &ptr1);   // t points to the token "c"
t = wcstok_s(nullptr, &max1, "?", &ptr1);    // t is a null pointer
```

K.3.9.2.4 Miscellaneous functions

K.3.9.2.4.1 The wcsnlen_s function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```

Annex M
(informative)
Change History

M.1 Fifth Edition

1 Major changes in this fifth edition (**__STDC_VERSION__** *yyyymm*L) include:

- ~~added~~ add a one-argument version of ~~_Static_assert~~ **static_assert**, make it a keyword and deprecate the underscore-capital form
- an integration of floating-point technical specification TS 18661-1
- add a universal null pointer constant **nullptr**
- change **bool**, **false** and **true** to keywords and make the constants type **bool**
- change **alignas**, **alignof**, **complex**, **imaginary**, **noreturn** and **thread_local** to be keywords and deprecate the underscore-capital forms
- change **_Complex_I** and **_Imaginary_I** to be keywords

M.2 Fourth Edition

1 There were no major changes in the fourth edition (**__STDC_VERSION__** 201710L), only technical corrections and clarifications.

M.3 Third Edition

1 Major changes in the third edition (**__STDC_VERSION__** 201112L) included:

- conditional (optional) features (including some that were previously mandatory)
- support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (<stdatomic.h> and <threads.h>)
- additional floating-point characteristic macros (<float.h>)
- querying and specifying alignment of objects (<stdalign.h>, <stdlib.h>)
- Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 19769:2004)
- type-generic expressions
- static assertions
- anonymous structures and unions
- no-return functions
- macros to create complex numbers (<complex.h>)
- support for opening files for exclusive access
- removed the **gets** function (<stdio.h>)
- added the **aligned_alloc**, **at_quick_exit**, and **quick_exit** functions (<stdlib.h>)
- (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731–1:2007)
- (conditional) support for analyzability