ISO/IEC JTC 1/SC 22/WG14
WG 21, SG 22

N2692 v3
P2312R1

2021-3-31

Introduce the nullptr constant
proposal for C23

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Since more than a decade C++ has already replaced the problematic definition of `NULL` which might be either of integer type or `void*`. By using a new constant `nullptr`, they achieve a more constrained specification, that allows much better diagnosis of user code. We propose to integrate this concept into C as far as possible by imposing only minimal ABI additions.
This is a follow-up of N2394 (which has been a split-off of N2368) that builds on the approval of N2654 and N2655.
Changes:

v3/R1. this document, integrating feedback from different sources
— make the type of `nullptr` incomplete and incompletable
— move most of the type information to `nullptr` itself and insist that it has as type that is different from any other standard type or type that could be defined by users code
— since `nullptr` does not have a scalar type, add it explicitly to contexts such as `if` or similar that so far only had scalars
— change the adjustment rules to result in `int` of value `0` and `1` for contexts where logical evaluation still has that type
— insist that the first operand of a ternary or comma expression is evaluated
— insist that primary expressions such as `()` or `_Generic` also are constant expressions or null pointer constants if the respective operands are
— add `nullptr_t` to generic selection
— don't allow `nullptr_t` as the last parameter before a ...
— only allow `nullptr_t` parameters without names
v2/R0. a complete rewrite as a proper language feature instead of a shallow macro solution

1. INTRODUCTION

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) 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.

2. POSSIBLE SPECIFICATIONS FOR A MORE RESTRICTIVE NULL POINTER CONSTANT

Because of such problems, C++ has long shifted to a different setting, namely the keyword `nullptr`. They use a special type `nullptr_t` for this constant, which allows to analyze and constrain the use of the constant more precisely:

— The constant can only be used in specific contexts, namely for conversion to pointer type, initialization, assignment and equality testing. It cannot be used in arithmetic or comparison.
— This type cannot be converted to an arithmetic type.
— No object can be created with this type.

In addition, a specific **nullptr_t** type allows C++ to provide neat overloading facilities such that optimized versions of functions can be accessed when their argument is a null pointer constant.
WG14 has expressed a clear position (15-1-1) to introduce the **nullptr** constant into the C language, but the path to achieve that as proposed in N2394 was considered not to go far enough. So instead of a primarily macro facility, we now propose a typed solution, that is capable to do most deductions at translation time and that also integrates well with C's **_Generic** feature.

3. DESIGN CHOICES

The most important choice in the design of the new feature is to draw the line between properties of the **nullptr** constant and its type, **nullptr_t**. It seemed to us that the most important feature here is the **nullptr** constant itself, and that the type is of much less importance. Therefore we opted to attach most properties to the constant; otherwise all places in the standard that make sophisticated choices according to a null pointer constant and/or about a type **void*** would have to be amended.
To avoid having to talk too much about the type of **nullptr** and expressions with that type we chose to adjust expressions that contain **nullptr** as much as possible, see Table I.

Table I. Proposed adjustment rules for **nullptr** expressions

expression	adjustment	context
`(nullptr)`	`nullptr`	
`_Generic(X, T: nullptr, ...)`	`nullptr`	if `X` has type `T`
`nullptr`	`0`	controlling expression for **if**, **do**, **for**, **while**, `&&` and `||` operators
`nullptr != nullptr`	`0`	
`nullptr`	`false`	conversion to **bool**
`!nullptr`	`1`	
`nullptr == nullptr`	`1`	
`x ? nullptr : nullptr`	`nullptr`	`x` is evaluated, result may not be a null pointer constant
`x , nullptr`		

Otherwise, **nullptr** is only allowed where a null pointer constant is converted to a pointer type, namely, pointer equality, pointer initialization and pointer assignment, including function arguments for functions with a pointer parameter *in their prototype*. Forbidden are the following uses.

— operand of an arithmetic operator,
— operand of relational comparison,
— as second or third operand in a `?:` choice, unless the other expression has pointer type,
— any conversion to a type other than a pointer type or **bool**,
— initialization or assignment other than for a pointer type,
— argument to a function parameter without prototype.

Only once these choices have been made, we have to minimally design the **nullptr_t** such that it is usable in generic selection and function declaration. Thereby it can be used to

improve the possibility of translation time choices through existing C features for type-generic programming.
As a consequence:

— Declarations of objects of type **nullptr_t** are only allowed for function parameters.
— Even if defined as a function parameter such an object cannot be named.

By this tricks we avoid any difficulties that could arise if the representation of such an object would be manipulated. There is no need for a specification of trap representations for the type, the **sizeof** and **alignof** operators cannot be applied, and the type cannot be used for the last parameter before a ... list or as an argument to such a list. Also, necessary ABI additions are minimized. The only specification that implementations have to agree upon is if a parameter of such a type is passed along, and, if it even is, the size or hardware register for such a parameter.

4. PROPOSED CHANGES

4.1. nullptr

First, we have to anchor **nullptr** in the syntax. This is not very difficult and requires additions of **nullptr** to 6.4.1 p1 and p2, 6.4.4.5 p1, 6.10.8.1 and Annex A.
The most important change is a new clause (6.5.4.4.2) that describes the main properties of the new constant. In particular it describes the main mechanism according to which it is used:

p1 is a list of contexts where expressions with **nullptr** are adjusted to simplify them at translation time, and that lead to null pointer constants or integer constant expressions
p2 gives two contexts that can lead to expressions of the same type and value as **nullptr**, but that are not necessarily null pointer constants.
p3 is a list of contexts in which **nullptr** is allowed after such adjustments.

The description part (p4) then describes the type of that constant. Namely it is an incomplete type that cannot be used in much other contexts than conversion to a pointer type and can be used with none of the usual type derivations. None of this specification makes use of the **nullptr_t** name, such that the decision to add that or not can be made independently.
Because the type is incomplete it follows that **nullptr** cannot be used for lvalue conversion, **sizeof**, **alignof**, **alignas**, declarations, arithmetic etc.
It also cannot be used as function parameter and in **_Generic**, but that restriction is lifted below if we introduce **nullptr_t**.
Then, some special arrangments are necessary:

(1) For pointer conversion (6.3.2.3) we add it to the list of null pointer constants.
(2) A note (6.4.4.5.2 p6) explains the contexts in which **nullptr** is not allowed by the constraints. In particular, for default function argument promotions we stipulate that **nullptr** is not a valid argument.
(3) We make the special provisions for **nullptr** constants that are adjusted, operator ! (6.5.3.3 p1), equality (6.5.9 p2 and new p5), operator && (6.5.13), operator || (6.5.14), conditional operator (6.5.15 p3 and new p6).
(4) Also rules for the controlling expressions of **if** (6.8.4.1 p1 and p2) and iterations statements (6.8.5 p2 and p4) adapted.

Note that at this point the function call operator needs no change, because **nullptr** arguments to pointer parameters are already covered by conversion.
As an optional change we add a new clause (6.11.3) to mark other constructs for null pointer constants obsolescent.

4.2. `nullptr_t`

In this proposal the type **`nullptr_t`** only plays a minor role and is only needed for user code that explicitly requests it. Therefore we propose to add it as semantic type to `<stddef.h>` much as **`size_t`** or **`ptrdiff_t`** (7.19 p2). The very few details of this type then are added in a new subclause (7.19.1).
The presentation of the new type is then accomplished by a sequence of three examples that illustrate the connection of **`nullptr_t`** and **`_Generic`**.
The intent is to use this type for type-generic interfaces, so special provisions are added to **`_Generic`** (6.5.1 p2), function calls (6.5.2.2 p2) and function declarators (6.7.6.3 p10).

4.3. `NULL`

Another set of changes concern **`NULL`**. The new constant **`nullptr`** is introduced to phase this one out, so **`NULL`** should be deprecated and replaced (7.19 p3 and new p5, 7.31.12). In the future even existing usage of **`NULL`** should provide all the possible diagnostics, so its expansion should preferably be set to **`nullptr`** (7.19 p5).
In addition, all uses of **`NULL`** should be replaced by **`nullptr`**. These changes are mainly text replacement, so we don't list them in the diffmarks, below.

5. QUESTIONS FOR WG14

As WG14 has already clearly expressed the wish to add **`nullptr`** to the C standard, we don't repeat that question here.

QUESTION 1. *Does WG14 want to integrate* **`nullptr`** *as proposed in N2692 into C23?*

QUESTION 2. *Does WG14 want to have a* **`nullptr_t`** *type along the lines of N2692 in C23?*

QUESTION 3. *Does WG14 want to integrate* **`nullptr_t`** *as proposed in N2692 into C23?*

QUESTION 4. *Does WG14 want to mark the constructs for other null pointer constants as obsolescent as proposed in N2692 for C23?*

QUESTION 5. *Does WG14 want to mark the* **`NULL`** *macro as obsolescent as proposed in N2692 for C23?*

Ackowledgments

Many thanks to Joseph Myers for the very detailed review and feedback.

Appendix: pages with diffmarks of the proposed changes against proposals N2654 and N2655.

The following page numbers are from the particular snapshot and may vary once the changes are integrated.
Necessary text replacement of **`NULL`** by **`nullptr`** is not listed and should be applied additionally.

6.3.2.3 Pointers

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

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

3 An integer constant expression with the value 0, ~~or~~ such an expression cast to type **`void`** `*`, or the constant **`nullptr`**, is called a *null pointer constant*.[68] 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.[69]

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[70] 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: the **`nullptr`** constant (6.4.4.5.2), 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).

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

[68] 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.

[69] 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.

[70] 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.

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.[71)] 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).

6.4.1 Keywords

Syntax

1 *keyword:* one of

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

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

Constraints

2 The keywords

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

may optionally be predefined macro names (6.10.8.4). None of these shall be the subject of a **#define** or a **#undef** preprocessing directive.

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 reserved for specifying imaginary types.[72)]

4 The following table provides alternate spellings for certain keywords. These can be used wherever the keyword can.[73)]

keyword	alternative spelling
alignas	**_Alignas**
alignof	**_Alignof**
bool	**_Bool**
static_assert	**_Static_assert**
thread_local	**_Thread_local**

5 The spelling of keywords that are also predefined macros and that are subject to the **#** and **##** preprocessing operators is unspecified.[74)]

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

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

digit: one of
0 1 2 3 4 5 6 7 8 9

Semantics

2 An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as

[72)] One possible specification for imaginary types appears in Annex G.
[73)] These alternative keywords are obsolescent features and should not be used for new code.
[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.

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**~~
~~**true**~~**false** **nullptr** **true**

Description

Some keywords represent constants of a specific value and type.

6.4.4.5.1 The `false` and `true` constants

Description

1 The keywords **`false`** and **`true`** represent constants of type **`bool`** that are suitable for use as are integer literals. Their values are 0 for **`false`** and 1 for **`true`**.[82] When used in preprocessor conditional expressions, the keywords **`false`** and **`true`** behave as if replaced with the pp-numbers `0` and `1`, respectively.[83]

6.4.4.5.2 The `nullptr` constant

Constraints

1 If the **`nullptr`** constant appears as the subject expression of a parenthesized expression or as the chosen assignment expression of a generic selection, the corresponding expression is adjusted to **`nullptr`**; in a conversion to **`bool`** it is adjusted to **`false`**; if it appears as both the first and the second operand of an equality operator, the corresponding expression is adjusted to values of type **`int`**, `1` (for `==`) or `0` (for `!=`); as a controlling expression of an **`if`**, **`while`**, **`for`** or **`do`** statement or of a conditional operator, or as an operand of a logical negation, conjunction or disjunction it is adjusted to `0`. The result of such an adjustments is a null pointer constant or integer constant expression, respectively.

2 If the **`nullptr`** constant appears as both the second and third operand of a conditional operator, or as the second operand of a comma operator, the expression has the same type and value as **`nullptr`**.

3 After such adjustments, the **`nullptr`** constant shall only be used as follows: as a null pointer constant when it is the operand of a conversion to a pointer type; as a function call argument to a parameter of type **`nullptr_t`**; or as a void expression.

Description

4 The keyword **`nullptr`** represents a null pointer constant of an incomplete type that is not an array or pointer type and that is different from any other object type specified by this document or declared by any translation unit that constitutes the program.[84] The type is never completed and it shall not be used in any type derivations, such as pointer types, function types, atomic types, or arrays. Values of this type shall only be formed by using the **`nullptr`** constant, either directly or by the adjustments or evaluations as described in the constraints.

5 If a conditional operator or a comma operator result in the same value and type as **`nullptr`** as indicated in the constraints, the resulting expression is a null pointer constant only if the first operand of the operator is a null pointer constant or an integer constant expression.

6 **NOTE** The syntactical adjustments ensure that the use of **`nullptr`** as a null pointer constant can be detected and acted upon during translation time. The constraints prohibit the use of **`nullptr`** as an operand of any arithmetic operation, relational comparison, initialization, assignment or as an argument to a function parameter for which no prototype is visible.

Forward references: the **`nullptr_t`** type (7.19.1)

7 **EXAMPLE**

```
double * a = nullptr;           // implicit conversion to double*, valid
free(nullptr);                  // implicit conversion to void*, useless, but valid
printf("%p\n", nullptr);        // invalid, use of nullptr without conversion
printf("%p\n", (void*)nullptr); // valid, explicit conversion to void*
```

[82] When used in arithmetic expressions after translation phase 4 the values of the keywords are promoted to type **`int`**.

[83] Therefore, arithmetic with **`false`** and **`true`** in translation phase 4 presents results that are generally consistent with later translation phases.

[84] This implies that the type of the **`nullptr`** constant is neither a pointer type nor an arithmetic type and that the type cannot accidentally be formed by type derivations that are at the disposal of the program, such as pointer, array, structure or union types.

— a type that is the signed or unsigned type corresponding to the effective type of the object,

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

8 A floating expression may be *contracted*, that is, evaluated as though it were a single operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[97] The `FP_CONTRACT` pragma in `<math.h>` provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.[98]

Forward references: the `FP_CONTRACT` pragma (7.12.2), copying functions (7.24.2).

6.5.1 Primary expressions

Syntax

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

Semantics

2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[99]

3 A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4 A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

5 A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, a constant expression, a null pointer constant, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, a constant expression, a null pointer constant, or a void expression.

6 A generic selection is a primary expression. Its type and value depend on the selected generic association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax

1 *generic-selection:*
_Generic (*assignment-expression* **,** *generic-assoc-list* **)**
generic-assoc-list:
generic-association
generic-assoc-list **,** *generic-association*
generic-association:
type-name **:** *assignment-expression*

[97] The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the final operation is rounded to the format determined by the expression evaluation method. A contracted expression might also omit the raising of floating-point exceptions.

[98] This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

[99] Thus, an undeclared identifier is a violation of the syntax.

`default :` *assignment-expression*

Constraints

2 A generic selection shall have no more than one **`default`** generic association. The type name in a generic association shall specify a complete object type other than a variably modified type or it shall be **`nullptr_t`** . No two generic associations in the same generic selection shall specify compatible types. The type of the controlling expression is the type of the expression as if it had undergone an lvalue conversion,[100] array to pointer conversion, or function to pointer conversion. That type shall be compatible with at most one of the types named in the generic association list. If a generic selection has no **`default`** generic association, its controlling expression shall have type compatible with exactly one of the types named in its generic association list.

Semantics

3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic association with a type name that is compatible with the type of the controlling expression, then the result expression of the generic selection is the expression in that generic association. Otherwise, the result expression of the generic selection is the expression in the **`default`** generic association. None of the expressions from any other generic association of the generic selection is evaluated.

4 The type and value of a generic selection are identical to those of its result expression. It is an lvalue, a function designator, a constant expression, a null pointer constant, or a void expression if its result expression is, respectively, an lvalue, a function designator, a constant expression, a null pointer constant, or a void expression.

5 **EXAMPLE** The **`cbrt`** type-generic macro could be implemented as follows:

```
#define cbrt(X) _Generic((X),                   \
                         long double: cbrtl,    \
                         default: cbrt,         \
                         float: cbrtf           \
                         )(X)
```

6.5.2 Postfix operators

Syntax

1 *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* **`,`** **`}`**

argument-expression-list:

assignment-expression
argument-expression-list **`,`** *assignment-expression*

6.5.2.1 Array subscripting

Constraints

1 One of the expressions shall have type "pointer to complete object *type*", the other expression shall have integer type, and the result has type "*type*".

[100] An lvalue conversion drops type qualifiers.

Semantics

2 A postfix expression followed by an expression in square brackets `[]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary `+` operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2` -th element of `E1` (counting from zero).

3 Successive subscript operators designate an element of a multidimensional array object. If `E` is an n-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$, then `E` (used as other than an lvalue) is converted to a pointer to an $(n-1)$-dimensional array with dimensions $j \times \cdots \times k$. If the unary $*$ operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n-1)$-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4 **EXAMPLE** Consider the array object defined by the declaration

```
int x[3][5];
```

Here `x`

is a 3×5 array of

`int` s; more precisely, `x` is an array of three element objects, each of which is an array of five **`int`** s. In the expression `x[i]`, which is equivalent to `(*((x)+(i)))`, `x` is first converted to a pointer to the initial array of five **`int`** s. Then `i` is adjusted according to the type of `x`, which conceptually entails multiplying `i` by the size of the object to which the pointer points, namely an array of five **`int`** objects. The results are added and indirection is applied to yield an array of five **`int`** s. When used in the expression `x[i][j]`, that array is in turn converted to a pointer to the first of the **`int`** s, so `x[i][j]` yields an **`int`**.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints

1 The expression that denotes the called function[101)] shall have type pointer to function returning **`void`** or returning a complete object type other than an array type.

2 If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter, or shall be of type **`nullptr_t`** if the corresponding parameter has that type.

Semantics

3 A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

4 An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[102)]

5 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **`void`**.

6 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types

[101)] Most often, this is the result of converting an identifier that is a function designator.

[102)] A function can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

— one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;

— both types are pointers to qualified or unqualified versions of a character type or **`void`**.

7 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

8 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

9 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

10 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[103)]

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

12 **EXAMPLE** In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

Forward references: function declarators (including prototypes) (6.7.6.3), function definitions (6.9.1), the **`return`** statement (6.8.6.4), simple assignment (6.5.16.1)~~.~~, the **`nullptr_t`** type (7.19.1).

6.5.2.3 Structure and union members

Constraints

1 The first operand of the `.` operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2 The first operand of the`->` operator shall have type “pointer to atomic, qualified, or unqualified structure” or “pointer to atomic, qualified, or unqualified union”, and the second operand shall name a member of the type pointed to.

Semantics

3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member,[104)] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

4 A postfix expression followed by the`->` operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression

[103)] In other words, function executions do not “interleave” with each other.

[104)] If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called “type punning”). This might be a trap representation.

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.[111)]

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 or **`nullptr`**.

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 (or is **`nullptr`**, see 6.4.4.5.2). The result has type **`int`**. The expression `!E` is equivalent to `(0==E)`.

6.5.3.4 The `sizeof` and `alignof` operators

Constraints

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

[111)] 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.

4 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression `P` points to an element of an array object and the expression `Q` points to the last element of the same array object, the pointer expression `Q+1` compares greater than `P`. In all other cases, the behavior is undefined.

6 Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[116] The result has type **`int`**.

6.5.9 Equality operators

Syntax

1 *equality-expression:*
relational-expression
equality-expression **`==`** *relational-expression*
equality-expression **`!=`** *relational-expression*

Constraints

2 One of the following shall hold:

— both operands have arithmetic type;

— both operands are **`nullptr`**;

— both operands are pointers to qualified or unqualified versions of compatible types;

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **`void`**; or

— one operand is a pointer and the other is a null pointer constant.

Semantics

3 The `==` (equal to) and `!=` (not equal to) operators are analogous to the relational operators except for their lower precedence.[117] Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type **`int`**. For any pair of operands, exactly one of the relations is true.

4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

5 If both operands are **`nullptr`** the expression is adjusted to `0` (for `!=`) or `1` (for `==`), see 6.4.4.5.2.

6 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a

[116] The expression `a<b<c` is not interpreted as in ordinary mathematics. As the syntax indicates, it means `(a<b)<c`; in other words, "if `a` is less than `b`, compare 1 to `c`; otherwise, compare 0 to `c`".
[117] Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

6.5.13 Logical AND operator

Syntax

1 *logical-AND-expression:*
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

Constraints

2 Each of the operands shall have scalar type or be **nullptr** .

Semantics

3 The && operator shall yield 1 if both of its operands are scalars and compare unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

6.5.14 Logical OR operator

Syntax

1 *logical-OR-expression:*
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

Constraints

2 Each of the operands shall have scalar type or be **nullptr** .

Semantics

3 The || operator shall yield 1 if either of its operands ~~compare~~ is scalar and compares unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.

6.5.15 Conditional operator

Syntax

1 *conditional-expression:*
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

Constraints

2 The first operand shall have scalar type.

3 One of the following shall hold for the second and third operands:

— both operands have arithmetic type;

— both operands have the same structure or union type;

— both operands have void type;

— both operands are pointers to qualified or unqualified versions of compatible types;

— both operands are **nullptr**;

— one operand is a pointer and the other is a null pointer constant; or

— one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**.

Semantics

4 The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.[119)]

5 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

6 If both the second and third operands are **nullptr**, the expression has the type and value of **nullptr**, see 6.4.4.5.2.

7 If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**.

8 **EXAMPLE** The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

9 Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

```
c_vp   c_ip   const void *
v_ip   0      volatile int *
c_ip   v_ip   const volatile int *
vp     c_cp   const void *
ip     c_ip   const int *
vp     ip     void *
```

6.5.16 Assignment operators

Syntax

1 *assignment-expression:*
conditional-expression
unary-expression assignment-operator assignment-expression

[119)] A conditional expression does not yield an lvalue.

```
        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* `)`

or

`D (` *identifier-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* function returning the unqualified version of *T*".

6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

7 A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword **`static`** also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

8 A declaration of a parameter as "function returning *type*" shall be adjusted to "pointer to function returning *type*", as in 6.3.2.1.

9 If the list terminates with an ellipsis (`, ...`), no information about the number or types of the parameters after the comma is supplied.[152)]

10 The special case of an unnamed parameter of type **`nullptr_t`** as item in the list specifies that a call to the function expects an argument of that type in the corresponding position. If the parameter type list terminates with an ellipsis (`, ...`), the rightmost parameter declaration shall not have type **`nullptr_t`**.

11 The special case of an unnamed parameter of type **`void`** as the only item in the list specifies that the function has no parameters.

12 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

13 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

14 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

[152)] The macros defined in the `<stdarg.h>` header (7.16) can be used to access arguments that correspond to the ellipsis.

Forward references: iteration statements (6.8.5).

6.8.4 Selection statements

Syntax

1 *selection-statement:*

`if` **`(`** *expression* **`)`** *statement*
`if` **`(`** *expression* **`)`** *statement* **`else`** *statement*
`switch` **`(`** *expression* **`)`** *statement*

Semantics

2 A selection statement selects among a set of statements depending on the value of a controlling expression.

3 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

6.8.4.1 The `if` statement

Constraints

1 The controlling expression of an **`if`** statement shall have scalar type or be **`nullptr`** .

Semantics

2 In both forms, the first substatement is executed if the expression is scalar and compares unequal to 0. In the **`else`** form, the second substatement is executed if the expression compares equal to ~~0.~~ 0 (or is **`nullptr`** see 6.4.4.5.2). If the first substatement is reached via a label, the second substatement is not executed.

3 An **`else`** is associated with the lexically nearest preceding **`if`** that is allowed by the syntax.

6.8.4.2 The `switch` statement

Constraints

1 The controlling expression of a **`switch`** statement shall have integer type.

2 If a **`switch`** statement has an associated **`case`** or **`default`** label within the scope of an identifier with a variably modified type, the entire **`switch`** statement shall be within the scope of that identifier.[162)]

3 The expression of each **`case`** label shall be an integer constant expression and no two of the **`case`** constant expressions in the same **`switch`** statement shall have the same value after conversion. There may be at most one **`default`** label in a **`switch`** statement. (Any enclosed **`switch`** statement may have a **`default`** label or **`case`** constant expressions with values that duplicate **`case`** constant expressions in the enclosing **`switch`** statement.)

Semantics

4 A **`switch`** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **`default`** label and the values of any **`case`** labels on or in the switch body. A **`case`** or **`default`** label is accessible only within the closest enclosing **`switch`** statement.

5 The integer promotions are performed on the controlling expression. The constant expression in each **`case`** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **`case`** label. Otherwise, if there is a **`default`** label, control jumps to the labeled statement. If no converted **`case`** constant expression matches and there is no **`default`** label, no part of the switch body is executed.

[162)] That is, the declaration either precedes the **`switch`** statement, or it follows the last **`case`** or **`default`** label associated with the **`switch`** that is in the block containing the declaration.

7 **EXAMPLE** In the artificial program fragment

```
switch (expr)
{
      int i = 4;
      f(i);
case 0:
      i = 17;
      /* falls through into default code */
default:
      printf("%d\n", i);
}
```

the object whose identifier is `i` exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **`printf`** function will access an indeterminate value. Similarly, the call to the function `f` cannot be reached.

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*

`while` **`(`** *expression* **`)`** *statement*
`do` *statement* **`while`** **`(`** *expression* **`)`** **`;`**
`for` **`(`** $expression_{opt}$ **`;`** $expression_{opt}$ **`;`** $expression_{opt}$ **`)`** *statement*
`for` **`(`** *declaration* $expression_{opt}$ **`;`** $expression_{opt}$ **`)`** *statement*

Constraints

2 The controlling expression of an iteration statement shall have scalar type or be **`nullptr`** .

3 The declaration part of a **`for`** statement shall only declare identifiers for objects having storage class **`auto`** or **`register`**.

Semantics

4 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to ~~0.~~ 0 (or is **`nullptr`** see 6.4.4.5.2). The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[163)]

5 An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

6 An iteration statement may be assumed by the implementation to terminate if its controlling expression is not a constant expression,[164)] and none of the following operations are performed in its body, controlling expression or (in the case of a **`for`** statement) its *expression-3*:[165)]

— input/output operations

— accessing a volatile object

— synchronization or atomic operations.

6.8.5.1 The `while` statement

1 The evaluation of the controlling expression takes place before each execution of the loop body.

6.8.5.2 The `do` statement

1 The evaluation of the controlling expression takes place after each execution of the loop body.

[163)] Code jumped over is not executed. In particular, the controlling expression of a **`for`** or **`while`** statement is not evaluated before entering the loop body, nor is *clause-1* of a **`for`** statement.

[164)] An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

[165)] This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.

__STDC_IEC_559__ The integer constant `1`, intended to indicate conformance to the specifications in Annex F (IEC 60559 floating-point arithmetic).

__STDC_IEC_559_COMPLEX__ The integer constant `1`, intended to indicate adherence to the specifications in Annex G (IEC 60559 compatible complex arithmetic).

__STDC_LIB_EXT1__ The integer constant *`202101`*`L`, intended to indicate support for the extensions defined in Annex K (Bounds-checking interfaces).[188)]

__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 defines **__STDC_NO_COMPLEX__** shall not define **__STDC_IEC_559_COMPLEX__**.

6.10.8.4 Optional macros

1 The keywords

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

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

6.10.9 Pragma operator

Semantics

1 A unary operator expression of the form:

_Pragma (*string-literal* **)**

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

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

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

can also be expressed as:

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

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

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

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

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

6.11 Future language directions

6.11.1 Floating types

1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **`long double`**.

6.11.2 Linkages of identifiers

1 Declaring an identifier with internal linkage at file scope without the **`static`** storage-class specifier is an obsolescent feature.

6.11.3 Null pointer constants

1 The property of integer constant expressions with the value 0, and such expressions cast to type **`void`**`*`, to stand in as a null pointer constant is an obsolescent feature.

6.11.4 External names

1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.5 Character escape sequences

1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.6 Storage-class specifiers

1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.7 Function declarators

1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

6.11.8 Function definitions

1 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

6.11.9 Pragma directives

1 Pragmas whose first preprocessing token is **`STDC`** are reserved for future standardization.

6.11.10 Predefined macro names

1 Macro names beginning with **`__STDC_`** are reserved for future standardization.

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 (7.22.1.3).

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.[250] 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.[251]

7.12.12 Maximum, minimum, and positive difference functions

7.12.12.1 The fdim functions

Synopsis

1
```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

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

[251] 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.

7.16 Variable arguments <stdarg.h>

1 The header <stdarg.h> declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

2 A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.[263)]

3 The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros **va_start**, **va_arg**, **va_end**, and **va_copy**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type **va_list**. The object ap may be passed as an argument to another function; if that function invokes the **va_arg** macro with parameter ap, the value of ap in the calling function is indeterminate and shall be passed to the **va_end** macro prior to any further reference to ap.[264)]

7.16.1 Variable argument list access macros

1 The **va_start** and **va_arg** macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether **va_copy** and **va_end** are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the **va_start** and **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro in the same function.

7.16.1.1 The va_arg macro

Synopsis

1
```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

2 The **va_arg** macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter ap shall have been initialized by the **va_start** or **va_copy** macro (without an intervening invocation of the **va_end** macro for the same ap). Each invocation of the **va_arg** macro modifies ap so that the values of successive arguments are returned in turn. The parameter *type* shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a $*$ to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

— one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;

— one type is pointer to **void** and the other is a pointer to a character type.

Returns

3 The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

Forward references: the **nullptr_t** type (7.19.1).

7.16.1.2 The va_copy macro

[263)] This parameter does not have type **nullptr_t**, see 6.7.6.3.

[264)] It is permitted to create a pointer to a **va_list** and pass that pointer to another function, in which case the original function can make further use of the original list after the other function returns.

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

```
nullptr_t
```

which is the type of the **nullptr** constant, see below;

```
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;[271] 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.

5 The macro **NULL** should expand to **nullptr**.

[271] The **NULL** macro is an obsolescent feature.

7.19.1 The `nullptr_t` type

Description

1 The **`nullptr_t`** type is type of the **`nullptr`** constant. It has only a very limited use in contexts where this type is needed to distinguish **`nullptr`** from other expression types.

2 Although it is an incomplete type that is not an array type, this type may occur as a function parameter but, if so, it is not named, see 6.7.6.3. Therefore such parameters can never be addressed or evaluated.

3 **EXAMPLE 1** Consider a function `func` that receives a pointer parameter that can either be valid or a null pointer to indicate a default choice.

```
// header "func.h"
void func(toto*);

// define a default action
// no parameter name, parameter is never read
inline void func_nullptr(nullptr_t) {
  ...
}

#define func(P)                  \
   _Generic((P),                 \
         nullptr_t: func_nullptr, \
         default:   func)(P)
------------------------------------------------------------
// one translation unit
#include "func.h"
// emit an external definition
extern void func_nullptr(nullptr_t);

// define the general action
void (func)(toto* p) {
  // p may still have value null
  if (!p) func_nullptr(nullptr);    // may only be called with nullptr
  else {
    ...
  }
}
```

Here, a function `func_nullptr` is defined that receives a **`nullptr_t`** type. The function needs no access to the parameter, since that parameter can only hold one specific value and it may not even be evaluated. A type-generic macro **`func`** then chooses this function or the general function `func`. The translation unit that defines `func` may then emit an external definition of `func_nullptr` and also use it within the definition for the case that `func` receives a parameter value that is null without being recognized as such at translation time of the call.

4 **EXAMPLE 2**

```
#include "func.h"
...
   func(0);        // ok, but uses the general function and may issue a diagnostic
   func(nullptr);  // uses default action directly
```

The use of the macro with a null pointer constant of integer type then uses the general function and sets the parameter to null; implementations that chose to diagnose the use of null pointer constants of integer type may do so for this call. In contrast to that, a call that uses **`nullptr`** as an argument directly resolves to `func_nullptr`, may or may not inline the corresponding action, and will not trigger such a diagnosis.

5 **EXAMPLE 3**

```
#define func_strict(P)           \
   _Generic((P),                 \
         nullptr_t: func_nullptr, \
         toto*:     func)(P)
...
```

```
func_strict(0);       // invalid, int not a valid choice, constraint violation
func_strict(nullptr); // uses default action directly
```

The emission of a diagnosis can be forced by restricting the admissible type as shown in the definition of **`func_strict`**.

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.

(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`~~
~~`true`~~`false` `nullptr` `true`

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