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

Nyyyy v2
P2312R0

February 15, 2021

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 without putting much strain on implementations.
This is a follow-up of N2394 (which has been a split-off of N2368) that builds on the approval of N2654 and N2655.

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 seemed to be too timid. So instead of a primarily macro facility, we now propose a complete typed solution, that is capable to do most deductions at translation time and that also integrates well with C's `_Generic` feature.

© 2021 by the author(s). Distributed under a Creative Commons Attribution 4.0 International License

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 about the **nullptr_t** type 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`
`x ? nullptr : nullptr`	`nullptr`	`x` may be evaluated or not
`nullptr`	`false`	controlling expression
`!nullptr`	`true`	
`nullptr == nullptr`	`true`	
`nullptr != nullptr`	`false`	

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, not even to objects of **nullptr_t**,
— 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 evaluated. Such an evaluation is unnecessary since the type has exactly one possible value, namely **nullptr**.

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, for example. Also, necessary ABI additions are minimized. The only specification that implementations have to agree upon is if a parameter of such a type is even 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

p2 is a list of contexts in which **nullptr** is allowed.

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 p5) 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 used in equality testing (6.5.9 p2 and new p5) and the conditional operator (6.5.15 p3 and new p6).

As an optional change we add a new clause (6.11.3) to mark other constructs for null pointer constants obsolescent.

4.2. NULL

A second 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.

4.3. 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 few details of this type then are added in a new subclause (7.19.1). It constrains the use of the type to function parameters and inhibits any evaluation of such a parameter. The presentation of the new type is then accomplished by a sequence of three examples that illustrate the connection of **nullptr_t** and **_Generic**.

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 Nyyyy into C23?*

Question 2. *Does WG14 want to mark the constructs for other null pointer constants as obsolescent as proposed in Nyyyy for C23?*

Question 3. *Does WG14 want to mark the* **NULL** *macro as obsolescent as proposed in Nyyyy for C23?*

Question 4. *Does WG14 want to have a* **nullptr_t** *type along the lines of Nyyyy in C23?*

Question 5. *Does WG14 want to integrate* **nullptr_t** *as proposed in Nyyyy into C23?*

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.

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`**; if it appears as both the first and the second operand of an equality operator, the corresponding expression is adjusted to **`true`** (for `==`) or **`false`** (for `!=`); if it appears as both the second and third operand of a conditional operator, the corresponding expression is adjusted to **`nullptr`**.

2 After such adjustments, the **`nullptr`** constant shall only be used as follows: as a null pointer constant when it appears as the operand of a conversion to a pointer type; as the only possible value for a function call argument to a parameter of type **`nullptr_t`**; as a **`void`** expression; when used for the determination of a Boolean value or a type in a controlling expression; as an operand of an equality operator, of a conversion to **`bool`** or of a logical negation.

Description

3 The keyword **`nullptr`** represents a null pointer constant of type **`nullptr_t`**.

4 If a conditional operator is adjusted as in the constraints, it is unspecified if the first operand of the operator is evaluated.

5 **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)

6 **EXAMPLE**

```
(x ? nullptr : nullptr);         // valid, adjusted to nullptr, x may be evaluated
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*
```

6.4.5 String literals

Syntax

1 *string-literal:*

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

[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.

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.[115] 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.[116] 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 **`true`** (for `==`) or **`false`** (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

[115] 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`".

[116] Because of the precedences, `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth-value.

— 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.[118)]

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 is adjusted to **nullptr** and it is unspecified if the first operand is evaluated, 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

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

__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).[187)]

__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)
```

[187)] 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.

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;[269] 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**.

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

7.19.1 The `nullptr_t` type

Constraints

1 The **`nullptr_t`** type shall only be used to declare objects if they are function parameters. Such a parameter with **`nullptr_t`** type shall not be subject to lvalue conversion.

Description

2 The **`nullptr_t`** type is a complete object type that is not a pointer type itself and that is different from any other object type. It has only a very limited use in contexts where this type is needed to distinguish **`nullptr`** from other expression types. By the constraints, objects of this type may only occur as parameters, and even as such their particular value (known to be **`nullptr`**) will never be read.

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