Introduce the nullptr constant

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2394**

June 9, 2019 **v 1**

Introduce the nullptr constant
proposal for C2x

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 split-off of N2368 that builds on the approval of the "keywords" patches.

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.
All of this is not strictly necessary for C: we can just define `nullptr` to be `((void*)0)` to avoid the principal defaults of `NULL` as mentioned above.

As a matter of fact we see two possible strategies for existing implementations to implement a `nullptr` feature:

— Implementations that have also C++ front-ends will want to reuse their code for `nullptr` from there. Such an implementation then can provide detailed diagnostics when the constant is used erroneously.

— Implementations that have yet no special provisions for null pointer constants besides `NULL` and will hesitate to embark on implementing it in a short time frame. For these a simple cost-effective migration path to C2x should be provided.

Therefore we propose a specification that attempts to bridge the gap between those two principal strategies:

— We don't require a special type of `nullptr`, but only that it is not an arithmetic type. Thereby, an implementation as predefined macro with expansion `((void*)0)` or similar is legit.
— We forbid the same use cases that C++, but not all of them can be formulated as constraint violations. In fact, two of the forbidden use cases are not easily detectable at compile time if `nullptr` is expanded to `((void*)0)` by the preprocessor, namely
  — conversion to a non-pointer type, and
  — the usage of `nullptr` and an integer 0 as second and third argument of a conditional operator.

3. PROPOSED CHANGES

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: there is a list of contexts where it appears **as if** it had a pointer type. This ensures that in all other contexts (which are not listed) the use of `nullptr` constitutes a constraint violation.
Then, some special arrangments are necessary:

(1) For pointer conversion (6.3.2.3) we add it to the list of null pointer constants and insist that `nullptr` must not be converted to a non-pointer type.
(2) For default function argument promotions (6.5.2.2) we stipulate that `nullptr` is replaced by a null pointer of type `void*`.
(3) `sizeof nullptr` is `sizeof(void*)` (6.5.3.4).
(4) We make special provisions for `nullptr` constants that are used in equality testing (6.9 p6) and conditional operators (6.5.15).

A second set of changes concern `NULL`. The new constant `nullptr` is introduced to phase this one out, so `NULL` should be deprecated (7.19 p3, 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. REFERENCE IMPLEMENTATION

To add minimal support for the proposed changes, an implementation would have to add a definition that is equivalent to the following line to their startup code:

```
#define nullptr       ((void*)0)
```

At the other end of the spectrum, an implementation that implements `nullptr` as first-class construct (*e.g* by using its existing C++ mechanism) can simply have a definition that is the the token identity:

```
#define nullptr       nullptr
```

Appendix: pages with diffmarks of the proposed changes against the keywords proposal N2392.

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

When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

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

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

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

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

6.3.2.2 void

1 The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 Pointers

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

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

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

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

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

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

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

[71] The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with

6.4.1 Keywords

Syntax

1 *keyword:* one of

alignas	**enum**	**return**	**void**
alignof	**extern**	**short**	**volatile**
auto	**false**	**signed**	**while**
bool	**float**	**sizeof**	**_Atomic**
break	**for**	**static**	**_Complex**
case	**goto**	**static_assert**	**_Decimal128**
char	**if**	**struct**	**_Decimal32**
const	**inline**	**switch**	**_Decimal64**
continue	**int**	**thread_local**	**_Generic**
default	**long**	**true**	**_Imaginary**
do	**nullptr**	**typedef**	**_Noreturn**
double	**register**	**union**	
else	**restrict**	**unsigned**	

Constraints

2 The keywords

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

are also predefined macro names (6.10.8). None of these shall be the subject of a **#define** or a **#undef** preprocessing directive and their spelling inside expressions that are subject to the **#** and **##** preprocessing operators is unspecified.[74)]

Semantics

3 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords except in an attribute token, and shall not be used otherwise. The keyword **_Imaginary** is reserved for specifying imaginary types.[75)]

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

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

6.4.2 Identifiers

6.4.2.1 General

Syntax

1 *identifier:*

identifier-nondigit
identifier identifier-nondigit
identifier digit

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

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

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

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 **int** that are suitable for use as are integer literals. Their values are 0 for **false** and 1 for **true**.[86)]

6.4.4.5.2 The **nullptr** constant

Description

1 The keyword **nullptr** represents a null pointer constant that has an unspecified type that is not an arithmetic type. Unless specified otherwise, it is a suitable primary expression wherever a constant operand of pointer type is allowed for initialization, assignment, conversion, function argument, equality testing, the **sizeof** operator, logical operators, and as a controlling expression.[87)] If **nullptr** is used in any other context, the behavior is undefined.[88)]

2 **NOTE** Because its type is unspecified, using **nullptr** as a controlling expression in a generic selection can lead to non-portable results.

Recommended practice

3 Although a possible implementation of **nullptr** is as a predefined macro that expands to **((void*)0)**, implementations are encouraged to implement **nullptr** with a type that is not a scalar type, that is incompatible to any other type and such that no objects of that type can be formed. They should diagnose the use of **nullptr**

- in any context where its use is undefined;
- as the controlling expression of a generic selection, unless that generic selection is itself not evaluated or the resulting type of the expression is independent of the effective choice;
- in a conversion to a type that is not a pointer type;
- as a second or third operand of a conditional operator if the other (second or third) operand has arithmetic type.

6.4.5 String literals

Syntax

1 *string-literal:*

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

encoding-prefix:

u8

[86)] Thus, the keywords **false** and **true** are usable in preprocessor directives.

[87)] Thus **nullptr** may or may not have pointer type, but if it appears in one of the listed contexts it behaves as if it had pointer type.

[88)] In particular this prohibits the use of **nullptr** for any type of arithmetic operation, relational comparison, or in an operation that requires an lvalue.

u
U
L

s-char-sequence:
s-char
s-char-sequence s-char

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

Constraints

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

Description

3 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "xyz". A UTF–*8 string literal* is the same, except prefixed by **u8**. A *wide string literal* is the same, except prefixed by the letter **L**, **u**, or **U**.

4 The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF–8 string literal) or a wide character constant (for a wide string literal), except that the single-quote ' is representable either by itself or by the escape sequence \', but the double-quote " shall be represented by the escape sequence \".

Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[89] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence. For UTF–8 string literals, the array elements have type **char**, and are initialized with the characters of the multibyte character sequence, as encoded in UTF–8. For wide string literals prefixed by the letter **L**, the array elements have type **wchar_t** and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the **mbstowcs** function with an implementation-defined current locale. For wide string literals prefixed by the letter **u** or **U**, the array elements have type **char16_t** or **char32_t**, respectively, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by successive calls to the **mbrtoc16**, or **mbrtoc32** function as appropriate for its type, with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

8 **EXAMPLE 1** This pair of adjacent character string literals

[89] A string literal might not be a string (see 7.1.1), because a null character can be embedded in it by a \0 escape sequence.

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, **nullptr** is converted to **void***, 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 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.[107)]

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).

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,[108)] and is an lvalue if the first expression is

[107)] In other words, function executions do not "interleave" with each other.

[108)] 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.

```
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
      if (p1->m  <  0)
            p2->m = -p2->m;
      return p1->m;
}
int g()
{
      union {
            struct t1 s1;
            struct t2 s2;
      } u;
      /* ... */
      return f(&u.s1, &u.s2);
}
```

Forward references: address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1).

6.5.2.4 Postfix increment and decrement operators

Constraints

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

Semantics

2 The result of the postfix `++` operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately-sequenced function call, the operation of postfix `++` is a single evaluation. Postfix `++` on an object with atomic type is a read-modify-write operation with `memory_order_seq_cst` memory order semantics.[111)]

3 The postfix `--` operator is analogous to the postfix `++` operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

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

6.5.2.5 Compound literals

Constraints

1 The type name shall specify a complete object type or an array of unknown size, but not a variable length array type.

2 All the constraints for initializer lists in 6.7.9 also apply to compound literals.

[111)] Where a pointer to an atomic object can be formed and `E` has integer type, `E++` is equivalent to the following code sequence where *T* is the type of `E`:

```
T *addr = &E;
T old = *addr;
T new;
do {
      new = old + 1;
} while (!atomic_compare_exchange_strong(addr, &old, new));
```

with `old` being the result of the operation.
Special care is necessary if `E` has floating type; see 6.5.16.2.

Semantics

3 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.[112)]

4 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

5 The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

6 All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.[113)]

7 String literals, and compound literals with const-qualified types, need not designate distinct objects.[114)]

8 **EXAMPLE 1** The file scope definition

```
int *p = (int []){2, 4};
```

initializes `p` to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

9 **EXAMPLE 2** In contrast, in

```
void f(void)
{
      int *p;
      /*...*/
      p = (int [2]){*p};
      /*...*/
}
```

`p` is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by `p` and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

10 **EXAMPLE 3** Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
      (struct point){.x=3, .y=4});
```

Or, if `drawline` instead expected pointers to **`struct`** `point`:

```
drawline(&(struct point){.x=1, .y=1},
      &(struct point){.x=3, .y=4});
```

11 **EXAMPLE 4** A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

12 **EXAMPLE 5** The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

[112)] Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or **`void`** only, and the result of a cast expression is not an lvalue.

[113)] For example, subobjects without explicit initializers are initialized to zero.

[114)] This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

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.

Semantics

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

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

4 When **`sizeof`** is applied to an operand that has type **`char`**, **`unsigned char`**, or **`signed char`**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[116)] When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. When applied to the **`nullptr`** constant the result is **`sizeof(void*)`**.

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

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

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

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

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

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

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

```
#include <stddef.h>

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

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

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

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

Semantics

4 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.[121] 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.

5 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.

6 If both of the operands are null pointer constants, they compare equal.

7 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 pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.

8 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[122]

9 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.

6.5.10 Bitwise AND operator

Syntax

1 *AND-expression:*
equality-expression
AND-expression **&** *equality-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

6.5.11 Bitwise exclusive OR operator

Syntax

1 *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression **^** *AND-expression*

Constraints

2 Each of the operands shall have integer type.

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

[122] Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

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**.

4 If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

5 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.[123)]

6 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.

7 If both the second and third operands are **nullptr** the result has the same type and value as **nullptr**. Otherwise, if either of the second or third operands is **nullptr**, and the other is an integer constant expression of value 0 the behavior is undefined.[124)]

8 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**.

9 **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.

10 Given the declarations

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

[124)] If the other operand has arithmetic type but is not constant and 0, a constraint is violated.

6.10.8 Predefined macro names

1 The values of the predefined macros listed in the following subclauses[194] (except for **__FILE__** and **__LINE__**) remain constant throughout the translation unit.

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

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

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

1 In addition to the keywords

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

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

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

__FILE__ The presumed name of the current source file (a character string literal).[195]

__LINE__ The presumed line number (within the current source file) of the current source line (an integer constant).[195]

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

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

__STDC_VERSION__ The integer constant *yyyymm*`L`.[196]

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

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

6.10.8.2 Environment macros

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

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

[194] See "future language directions" (6.11.10).
[195] The presumed source file name and line number can be changed by the **#line** directive.
[196] See Annex M for the values in previous revisions. The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this document.

7.19 Common definitions <stddef.h>

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

2 The types are

```
ptrdiff_t
```

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

```
size_t
```

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

```
max_align_t
```

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

```
wchar_t
```

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

3 The macros are

```
NULL
```

which expands to an implementation-defined null pointer constant;[286] 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 subobject (designated by *member-designator*), from the beginning of any object of type *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 *type* defines a new type or 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** .

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

7.31.10 Alignment <stdalign.h>

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

7.31.11 Atomics <stdatomic.h>

1 Macros that begin with **ATOMIC_** and an uppercase letter may be added to the macros defined in the <stdatomic.h> header. Typedef names that begin with either **atomic_** or **memory_**, and a lowercase letter may be added to the declarations in the <stdatomic.h> header. Enumeration constants that begin with **memory_order_** and a lowercase letter may be added to the definition of the **memory_order** type in the <stdatomic.h> header. Function names that begin with **atomic_** and a lowercase letter may be added to the declarations in the <stdatomic.h> header.

2 The macro **ATOMIC_VAR_INIT** is an obsolescent feature.

7.31.12 Common definitions <stddef.h>

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

7.31.13 Boolean type and values <stdbool.h>

1 The header <stdbool.h> together with its defined macro **__bool_true_false_are_defined** is an obsolescent feature.

7.31.14 Integer types <stdint.h>

1 Typedef names beginning with **int** or **uint** and ending with **_t** may be added to the types defined in the <stdint.h> header. Macro names beginning with **INT** or **UINT** and ending with **_MAX**, **_MIN**, **_WIDTH**, or **_C** may be added to the macros defined in the <stdint.h> header.

7.31.15 Input/output <stdio.h>

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

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

7.31.16 General utilities <stdlib.h>

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

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

7.31.17 String handling <string.h>

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

7.31.18 Date and time <time.h>

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

7.31.19 Threads <threads.h>

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

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

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

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

Annex A
(informative)
Language syntax summary

1 **NOTE** The notation is described in 6.1.

A.1 Lexical grammar

A.1.1 Lexical elements

(6.4) *token:*
keyword
identifier
constant
string-literal
punctuator

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

A.1.2 Keywords

(6.4.1) *keyword:* one of

alignas	**enum**	**return**	**void**
alignof	**extern**	**short**	**volatile**
auto	**false**	**signed**	**while**
bool	**float**	**sizeof**	**_Atomic**
break	**for**	**static**	**_Complex**
case	**goto**	**static_assert**	**_Decimal128**
char	**if**	**struct**	**_Decimal32**
const	**inline**	**switch**	**_Decimal64**
continue	**int**	**thread_local**	**_Generic**
default	**long**	**true**	**_Imaginary**
do	**nullptr**	**typedef**	**_Noreturn**
double	**register**	**union**	
else	**restrict**	**unsigned**	

A.1.3 Identifiers

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

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

(6.4.4.3) *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

Annex J
(informative)
Portability issues

1 This annex collects some information about portability that appears in this document.

J.1 Unspecified behavior

1 The following are unspecified:

— The manner and timing of static initialization (5.1.2).

— The termination status returned to the hosted environment if the return type of **main** is not compatible with int (5.1.2.2.3).

— The values of objects that are neither lock-free atomic objects nor of type **volatile sig_atomic_t** and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).

— The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).

— The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

— The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

— How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).

— Many aspects of the representations of types (6.2.6).

— The value of padding bytes when storing values in structures or unions (6.2.6.1).

— The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).

— The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).

— The values of any padding bits in integer representations (6.2.6.2).

— Whether certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (6.2.6.2).

— The type of the **nullptr** constant (??).

— Whether two string literals result in distinct arrays (6.4.5).

— The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?:`, and comma operators (6.5).

— The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

— The order of side effects among compound literal initialization list expressions (6.5.2.5).

— The order in which the operands of an assignment operator are evaluated (6.5.16).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

— An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **`void`**) is applied to a void expression (6.3.2.2).

— Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

— Conversion of **`nullptr`** to a type that is not a pointer type (6.3.2.3).

— Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

— A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

— An unmatched `'` or `"` character is encountered on a logical source line during tokenization (6.4).

— A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

— A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

— The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— The identifier **`__func__`** is explicitly declared (6.4.2.2).

— The program attempts to modify a string literal (6.4.5).

— The characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, or the characters `'`, `\`, `//`, or `/*` occur in the sequence between the `"` delimiters, in a header name preprocessing token (6.4.7).

— A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

— An exceptional condition occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

— For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after default argument promotion are not compatible with the types of the parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after default argument promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2).

— A 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 (6.5.2.2).

— A member of an atomic structure or union is accessed (6.5.2.3).

— The operand of the unary `*` operator has an invalid value (6.5.3.2).

— A pointer is converted to other than an integer or pointer type (6.5.4).

— The value of the second operand of the `/` or `%` operator is zero (6.5.5).

— If the quotient `a/b` is not representable, the behavior of both `a/b` and `a%b` (6.5.5).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

— Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration **`int`** `a[4][5]`) (6.5.6).

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

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

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

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

— Either of the second or third operands of a conditional operator is **`nullptr`**, and the other is an integer constant expression of value 0 (6.5.15).

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

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

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

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

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

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

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

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

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