ISO/IEC JTC 1/SC 22/WG14 N 2362

March 30, 2019 v 1

Moving to a provenance-aware memory object model for C

proposal for C2x by the memory model study group

Jens Gustedt[1], Peter Sewell[2], Kayvan Memarian[2], Victor B. F. Gomes[2], Martin Uecker[3]

[1]INRIA and ICube, Université de Strasbourg, France
[2]University of Cambridge, UK
[3]University Medical Center, Göttingen, Germany

1. INTRODUCTION

In a committee discussion from 2004 concerning **DR260**, WG14 confirmed the concept of provenance of pointers, introduced as means to track and distinguish pointer values that represent storage instances with same address but non-overlapping lifetimes. Implementations started to use that concept, in optimisations relying on provenance-based alias analysis, without it ever being clearly or formally defined, and without it being integrated consistently with the rest of the C standard.
We now propose a resolution to this: a provenance-aware memory object model for C to put C programmers and implementers on a solid footing in this regard. We present it in three related documents:

— N2362 (this document) Moving to a provenance-aware memory model for C: proposal for C2x by the memory object model study group. Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Martin Uecker. This introduces the proposal and gives the proposed change to the standard text, presented as change-highlighted pages of the standard.
— N2363 C provenance semantics: examples. Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Jens Gustedt, Martin Uecker. This explains the proposal and its design choices with discussion of a series of examples.
— N2364 C provenance semantics: detailed semantics. Peter Sewell, Kayvan Memarian, Victor B. F. Gomes. This gives a detailed mathematical semantics for the proposal

In addition:

— At http://cerberus.cl.cam.ac.uk/cerberus we provide an executable version of the semantics, with a web interface that allows one to explore and visualise the behaviour of small test programs. Nxxxx+1 includes the results of this for the example programs and for some major compilers.

The proposal has been developed in discussion among the C memory object model study group, including the authors listed above, Hubert Tong, Martin Sebor, and Hal Finkel. It has also been discussed with the Clang/LLVM and GCC communities, and with members of WG21, with presentations and informal conversations at EuroLLVM and the GNU Tools Cauldron in 2018.
To the best of our knowledge and ability, the proposal reconciles the various demands of existing implementations and the corpus of existing C code.

2. RELATED PAPERS

The proposal is based on discussion in the following earlier WG14 notes and meetings. With respect to these, the main changes are (1) a clear preference among the study group and the compiler communities we have spoken with for a model that does not track provenance via integers (coined PNVI models rather than PVI); (2) the enhancement to the specific address-exposed variants (PNVI-ae-*), which for many seems to be more intuitive than

PNVI-plain (though it is also more complex); and (3) the refinement to the PNVI-ae-udi variant.

— N2311: Exploring C Semantics and Pointer Provenance. Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Robert N. M. Watson, Peter Sewell. Identical text to the POPL 2019 paper of the same title.
— N2294: C Memory Object Model Study Group: Progress Report. Peter Sewell. 2018-09-16

Brno 2018-04

— N2263: Clarifying Pointer Provenance v4
— N2219: Clarifying Pointer Provenance (Q1-Q20) v3

Pittsburgh 2016-10

— N2090: Clarifying Pointer Provenance (Draft Defect Report or Proposal for C2x)

London 2016-04

— N2012 Clarifying the C memory object model
— N2013 C Memory Object and Value Semantics: The Space of de facto and ISO Standards
— N2014 What is C in Practice? (Cerberus Survey v2): Analysis of Response
— N2015 What is C in practice? (Cerberus survey v2): Analysis of Responses - with Comments

3. THE BASIC IDEA

(This section is duplicated both here and in the start of Section 2 of Nxxxx+1, to make both documents readable in isolation.)
C pointer values are typically represented at runtime as simple concrete numeric values, but mainstream compilers routinely exploit information about the *provenance* of pointers to reason that they cannot alias, and hence to justify optimisations. In this section we develop a provenance semantics for simple cases of the construction and use of pointers,
For example, consider the classic test [Fea04; KW12; Kre15; CMM$^+$16; MML$^+$16] below (note that this and many of the examples below are edge-cases, exploring the boundaries of what different semantic choices allow, and sometimes what behaviour existing compilers exhibit; they are not all intended as desirable code idioms).

```c
#include <stdio.h>
#include <string.h>
int y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

Depending on the implementation, x and y might in some executions happen to be allocated in adjacent memory, in which case &x+1 and &y will have bitwise-identical representation values, the **memcmp** will succeed, and p (derived from a pointer to x) will have the same representation value as a pointer to a different object, y, at the point of the update *p=11. This can occur in practice, e.g. with GCC 8.1 -O2 on some platforms. Its output of

```
x=1 y=2 *p=11 *q=2
```

suggests that the compiler is reasoning that `*p` does not alias with `y` or `*q`, and hence that the initial value of `y=2` can be propagated to the final **`printf`**. ICC, e.g. ICC 19 -O2, also optimises here (for a variant with `x` and `y` swapped), producing

```
x=1 y=2 *p=11 *q=11.
```

In contrast, Clang 6.0 -O2 just outputs the

```
x=1 y=11 *p=11 *q=11
```

that one might expect from a concrete semantics. Note that this example does not involve type-based alias analysis, and the outcome is not affected by GCC or ICC's `-fno-strict-aliasing` flag. Note also that the mere formation of the `&x+1` one-past pointer is explicitly permitted by the ISO standard, and, because the `*p=11` access is guarded by the **`memcmp`** conditional check on the representation bytes of the pointer, it will not be attempted (and hence flag UB) in executions in which the two storage instances are not adjacent.
These GCC and ICC outcomes would not be correct with respect to a concrete semantics, and so to make the existing compiler behaviour sound it is necessary for this program to be deemed to have undefined behaviour.
The current ISO standard text does not explicitly speak to this, but the 2004 ISO WG14 C standards committee response to Defect Report 260 (DR260 CR) [Fea04] hints at a notion of provenance associated to values that keeps track of their "origins":

> *"Implementations are permitted to track the origins of a bit-pattern and [...]. They may also treat pointers based on different origins as distinct even though they are bitwise identical."*

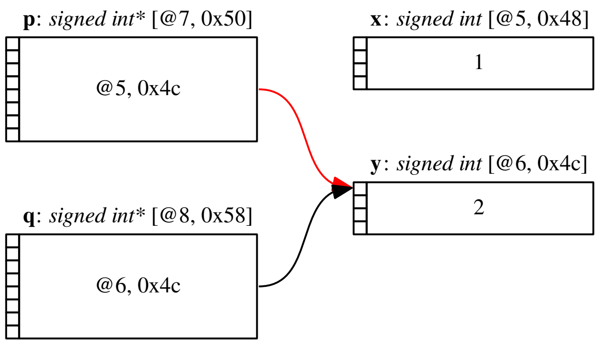
However, DR260 CR has never been incorporated in the standard text, and it gives no more detail. This leaves many specific questions unclear: it is ambiguous whether some programming idioms are allowed or not, and exactly what compiler alias analysis and optimisation are allowed to do.

Basic provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: we associate a *provenance* with every pointer value, identifying the original storage instance that the pointer is derived from. In more detail:

— We take abstract-machine pointer values to be pairs (π, a), adding a *provenance* π, either $@i$ where i is a storage instance ID, or the *empty* provenance @`empty`, to their concrete address a.
— On every creation of a storage instance (of objects with static, thread, automatic, and allocated storage duration), the abstract machine nondeterministically chooses a fresh storage instance ID i (unique across the entire execution), and the resulting pointer value carries that single storage instance ID as its provenance $@i$.
— Provenance is preserved by pointer arithmetic that adds or subtracts an integer to a pointer.
— At any access via a pointer value, its numeric address must be consistent with its provenance, with undefined behaviour otherwise. In particular:
  — access via a pointer value which has provenance a single storage instance ID $@i$ must be within the memory footprint of the corresponding original storage instance, which must still be live.
  — all other accesses, including those via a pointer value with empty provenance, are undefined behaviour.

This undefined behaviour is what justifies optimisation based on provenance alias analysis.

Below is a provenance-semantics memory-state snapshot (from the Cerberus GUI) for `provenance_basic_global_xy.c`, just before the invalid access via `p`, showing how the provenance mismatch makes it UB: at the attempted access via `p`, its pointer-value address 0x4c is not within the storage instance with the ID `@5` of the provenance of `p`.


p: *signed int** [@7, 0x50]
@5, 0x4c
x: *signed int* [@5, 0x48]
1
y: *signed int* [@6, 0x4c]
2
q: *signed int** [@8, 0x58]
@6, 0x4c

All this is for the *C abstract machine* as defined in the standard: compilers might rely on provenance in their alias analysis and optimisation, but one would not expect normal implementations to record or manipulate provenance at runtime (though dynamic or static analysis tools might), as might non-standard or bug-finding-tool implementations. Provenances therefore do not have program-accessible runtime representations in the abstract machine.
Then there are many other ways to construct and manipulate pointer values: casts to and from integers, copying with **`memcpy`**, manipulation of their representation bytes, type punning, I/O, copying with **`realloc`**, and constructing pointer values that embody knowledge established from linking. Nxxxx+1 discusses all these, and the proposal follows the **PNVI-ae-udi (PNVI exposed-address user-disambiguation)** model developed in it. Here:

— **PNVI-plain** is a semantics that tracks provenance via pointer values but not via integers. Then, at integer-to-pointer cast points, it checks whether the given address points within a live storage instance and, if so, recreates the corresponding provenance.
— **PNVI-ae (PNVI exposed-address)** is a variant of PNVI that allows integer-to-pointer casts to recreate provenance only for storage instances that have previously been *exposed*. A storage instance is deemed exposed by a conversion of a pointer to it to an integer type, by a read (at non-pointer type) of the representation of the pointer, or by an output of the pointer using `"%p"`.
— **PNVI-ae-udi (PNVI exposed-address user-disambiguation)** is a further refinement of PNVI-ae that supports roundtrip casts, from pointer to integer and back, of pointers that are one-past a storage instance. This is the currently preferred option in the C memory object model study group.

4. NEWLY INTRODUCED TERMS

4.1. Storage instance

An addressable *storage instance*[1] is the byte array that is created when either an object starts its lifetime (for static, automatic and thread storage duration) or an allocation function is called (**`malloc`**, **`calloc`** etc). Addressable storage instances are more than just an

[1]There are also storage instances that are not addressable, namely for **`register`** variables. But since provenance needs pointers, these play no role in the following and we don't discuss them, here.

address, they have a unique ID throughout the whole execution. Once their lifetime ends, another storage instance may receive the same address, but never the same ID.

4.2. Provenance

The *provenance* of a valid pointer is the storage instance to which the pointer refers (or one-past). The provenance is part of the abstract state in C's abstract machine, but not necessarily part of the object representation of the pointer itself. Thus in general it is not observable.
Valid pointers keep provenance to the encapsulating storage instance of the referred object. When the storage instance dies (falls out of scope, end of thread, `free`) the value of the pointer becomes indeterminate.

4.3. Abstract address

The concept of *abstract address* lifts the implementation defined mapping required for pointer-to-integer conversions, up the level of the memory model.

— Each byte of a storage instance has an abstract address, which is a positive integer that is constant during the whole lifetime of the storage instance.
— Abstract addresses are increasing within a storage instance.
— Storage instances are strictly ordered by the induced order of their abstract addresses.
— Storage instances don't overlap.
— The set of all abstract addresses forms the *address space* of the execution.
— There are no other ordering constraints between any pair of storage instances. In particular, no syntactic features (declaration order) or runtime features (order of allocation) can give any hint about the relative position.

This concept is completely decorrelated from the object representation of pointers: it is up to any implementation to define the relation between the two in any way that suits best. In particular, the address offset between consecutive bytes does not need to be 1 (or any other constant). There can be bumps (corresponding to segments, for example) and strides, and address sharing on the boundary between the one-past pointer of one storage instance and the start address of the next.
Compared to C17, on "ususal" architectures where `uintptr_t` exists, the abstract address of a pointer value `p` is just `(uintptr_t)p`. Architectures that don't have `uintptr_t` should be able to define an abstract address that is consistent with the other operations that they allow on pointers.

4.4. Exposure

Tracking provenance for the sake of aliasing analysis will fail if pointers can acquire an abstract address with an arbitrary provenance of which the compiler could not be aware. With the above rules for abstract addresses this is only possible with a leak of information about a storage instance `A`:

— the abstract address of `A` has been made known,
— the object representation of a pointer to `A` is inspected.

In such a case we say that `A` has been *exposed*.
There are only very restricted contexts where pointers can be constructed from scratch. We require that a storage instance of such a constructed pointer must have been exposed previously. By that we ensure that all storage instances that have *not* been exposed can be subject to a rigorous aliasing analysis, whereas pointers to potentially exposed storage instance acquire a clear "warning label" that tell the compiler to be cautious about them.
For the sake of sequencing and synchronization, exposure constitutes a side effect, even though it might not be directly observable.

5. OPERATIONS

5.1. Exposing and non-exposing operations

A storage instance is exposed once information from any valid pointer with this provenance has leaked into other parts of the program state. In C17 there are four different operations that can provide information about the address of a storage instance `A`.

— A pointer to `A` is converted to integer.
— **`printf`** (or similar) with `"%p"` is used to print the pointer value.
— A byte of the pointer representation is accessed directly.
— A byte of the pointer representation is written with **`fwrite`**.

All other C library functions (with the exception of **`tss_set`**) are guaranteed not to expose address information, unless they use a callback that does so (e.g **`qsort`** or **`exit`**). This guarantee has two different aspects:

— C library functions that receive pointers are not allowed to leak information about these pointers into global state.
— C library functions (such as **`memcpy`**, **`realloc`** or **`atomic_compare_exchange_weak`**) that copy bytes are supposed to know what they are doing. That is, if they copy the object representation of a pointer, they are supposed to transfer provenance information consistently.

5.2. Reconstructing operations

5.2.1. Lvalue conversion. Lvalue conversion for a pointer object that has somehow been constructed in memory, reads bytes of the object representation of the pointer and reinterprets them as a valid address with provenance. To be sure that we do not construct a pointer value for which the compiler has assumptions about non-aliasing, we must be sure that the provenance of that newly constructed pointer value had been exposed before.

5.2.2. Integer-to-pointer conversion. An integer-to-pointer conversion (cast) or IO (**`scanf`** with `"%p"`) is only defined if the corresponding storage instance had been exposed, and if the result is a pointer to a byte (or one-past) of the storage instance.

5.2.3. Copies. Pointer values can be copied by the usual means that is: assignment, **`memcpy`**, **`memmove`** and byte-wise copy. The first three copy over provenance in addition to the representation and the effective type.
Byte-wise copy is special, here, because up to now there is no tool to hint a transfer of a pointer value including provenance to the compiler. Therefore this works only through exposure, that is a pointer value that is copied byte-wise is first exposed (because bytes are accessed) and then reconstructed as before by lvalue conversion.

5.3. Pointer inquiry

5.3.1. Pointer equality. With the tool of abstract addresses, the description of pointer equality becomes quite simple: pointers are equal if their abstract addresses are the same.

5.3.2. Ordered comparision. Ordered comparisons (<, >, >=, <=) between pointers are only defined when the two pointers have the same provenance. They then can be defined by the relative position of the abstract addresses.
A possible extension here would be to remove the constraint that the two pointers have to have the same provenance.

5.4. Pointer arithmetic

5.4.1. Pointer addition and subtraction. Pointer arithmetic (addition or subtraction of integers) preserves provenance. The resulting pointer value is indeterminate if the result not within (or one-past) the storage instance.

5.4.2. Pointer difference. Pointer difference is only defined for pointers with the same provenance and within the same array. The latter is still necessary because pointer difference is not in byte but in number of elements of an array. The former is necessary because the one-past element of an array could be the first element of another storage instance that just happens to follow in the address space.

6. AMBIGUOUS PROVENANCE:

With the above, there is one special case where a back-converted pointer (let's just assume integer-to-pointer) could have two different provenances. This can happen when:

— `p` is the end address (one past) pointer of a storage instance `A` and the start address of another storage instance `B`, and
— both storage instances `A` and `B` are exposed, that is at some point we did a pointer-to-integer conversion with two pointers `a == b`, `a` having provenance `A`, and `b` having provenance `B`.

In such a situation, both `A` and `B` could be valid choices for the provenance. Our solution in 6.2.5 p20 is to leave which of `A` or `B` is chosen to the programmer, allowing one or the other (but not both) to be used, so long as that is done consistently.

References

David Chisnall, Justus Matthiesen, Kayvan Memarian, Kyndylan Nienhuis, Peter Sewell, and Robert N. M. Watson. C memory object and value semantics: the space of de facto and ISO standards. http://www.cl.cam.ac.uk/~pes20/cerberus/notes30.pdf (a revison of ISO SC22 WG14 N2013), March 2016.

Clive D. W. Feather. Indeterminate values and identical representations (dr260). Technical report, September 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm.

Robbert Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, December 2015.

Krebbers and Wiedijk. N1637: Subtleties of the ANSI/ISO C standard, September 2012. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1637.pdf.

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N.M. Watson, and Peter Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI 2016: 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (Santa Barbara)*, June 2016. PLDI 2016 Distinguished Paper award.

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

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

contains four separate memory locations: The member `a`, and bit-fields `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be.

3.15

1 **object**

region of data storage in the execution environment, the contents of which can represent values

2 **Note 1 to entry:** When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **parameter**

formal parameter

DEPRECATED: formal argument

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.17

1 **pointer provenance**

provenance

storage instance that holds the object to which a valid pointer value refers

3.18

1 **recommended practice**

specification that is strongly recommended as being in keeping with the intent of the standard, but that might be impractical for some implementations

3.19

1 **runtime-constraint**

requirement on a program when calling a library function

2 **Note 1 to entry:** Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 **Note 2 to entry:** Implementations that support the extensions in Annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

4 **Note 3 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.20

1 **storage instance**

a maximal region of data storage in the execution environment that is created when either an object definition or an allocation is encountered

2 **Note 1 to entry:** Storage instances are created and destroyed when specific language constructs (6.2.4) are met during program execution, including program startup, or when specific library functions (7.22.3) are called.

3 **Note 2 to entry:** A given storage instance may or may not have a memory address, and may or may not be accessible from all threads of execution.

4 **Note 3 to entry:** Storage instances have identities which are unique across the program execution.

3.21

1 **value**

precise meaning of the contents of an object when interpreted as having a specific type

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

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

6.2.4 Storage durations and object lifetimes

1 The *lifetime* of an object ~~is the portion of program execution during which storage is guaranteed~~ has a start and an end, which both constitute side effects in the abstract state machine, and is the set of all evaluations that happen after the start and before the end. An object exists, has a storage instance that is guaranteed to be reserved for it~~. An object exists,~~,[35] has a constant address,[36] if any, and retains its last-stored value throughout its lifetime.[37] ~~If~~

2 The lifetime of an object is ~~referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime~~determined by its *storage duration* . There are four storage durations: static, thread, automatic, and allocated. Allocated storage and its duration are described in 7.22.3.

3 ~~An~~ The storage instance of an object whose identifier is declared without the storage-class specifier **_Thread_local**, and either with external or internal linkage or with the storage-class specifier **static**, has *static storage duration* ~~. Its~~, as do storage instances for string literals and some compound literals. The object's lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

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

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

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

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[38] If the scope is entered recursively, a new instance of the object and associated storage is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to ~~an object~~ a *temporary object* with automatic storage duration and *temporary lifetime*.[39] Its

[35]String literals, compound literals or certain objects with temporary lifetime may share a storage instance with other such objects.

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

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

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

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

be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". The construction of a function type from a return type is called "function type derivation".

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. If the type is an object type, the pointer also carries a *provenance*, typically identifying the storage instance holding the corresponding object, if any. A pointer value is *valid* if and only if it has a non-empty provenance, there is a live storage instance for that provenance, and the address is either within or one-past the addresses of that storage instance. It is *null* to indicate that it does not refer to such a function or object,[50] and *indeterminate* otherwise. A pointer type derived from the referenced type *T* is sometimes called "pointer to *T*". The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.[51] Under certain circumstances a pointer value can have an address that is the end address of one storage instance and the start address of another. It (and any pointer value derived from it by means of arithmetic operations) shall then only be used with one and the same of these provenances as operand to subsequent operations that require a provenance.

— An *atomic type* describes the type designated by the construct **`_Atomic`**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[52]

22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[53] corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[54] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 Further, there is the **`_Atomic`** qualifier. The presence of the **`_Atomic`** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of

[50] A pointer object can be null by implicit or explicit initialization or assignment with a null pointer constant or by another null pointer value. A pointer value can be null if it is either a null pointer constant or the result of an lvalue conversion of a null pointer object. A null pointer will not appear as the result of an arithmetic operation.

[51] The provenance of a pointer value and the property that such a pointer value is indeterminate are generally not observable. In particular, in the course of the same program execution the same pointer representation (6.2.6) may refer to objects with different provenance and may sometimes be valid and sometimes be indeterminate. Yet, this information is part of the abstract state machine and may restrict the set of operations that can be performed on the pointer.

[52] Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

[53] See 6.7.3 regarding qualified array and function types.

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

the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic, qualified, or unqualified type" whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.

28 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[54] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. ~~Pointers to other types need not~~ It is implementation-defined if other groups of pointer types have the same representation or alignment requirements.[55]

29 **EXAMPLE 1** The type designated as "**float** *" has type "pointer to **float**". Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "**float** * **const**" whereas the type designated as "**const float** *" is not a qualified type — its type is "pointer to const-qualified **float**" and is a pointer to a qualified type.

30 **EXAMPLE 2** The type designated as "**struct** tag (*[5])(**float**)" has type "array of pointer to function returning **struct** tag". The array has length five and the function has a single parameter of type **float**. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in 6.2.5 and in this subclause. An object is represented (or held) by a storage instance (or part thereof) that is either created by an allocation (for allocated storage duration), at program startup (for static storage duration), at thread startup (for thread storage duration), or when the lifetime of the object starts (for automatic storage duration).

2 An addressable storage instance[56] of size m shall behave as a byte array of type **unsigned char**[m] or a qualified version thereof. All bytes of the array have an *abstract address* , which is a non-negative integer value that is determined in an implementation-defined manner. The abstract addresses of the bytes are increasing with the ordering within the array, and they shall be unique and constant during the lifetime. The address of the first byte of the array is the *start address* of the storage instance, the address one element beyond the array at index m is its *end address* . The abstract addresses of the bytes of all storage instances of a program execution form its *address space* . A storage instance Y *follows* storage instance X if the start address of Y is greater or equal than the end address of X, and it *follows immediately* if they are equal. During the common lifetime of any two distinct addressable storage instances X and Y, either Y follows X or X follows Y in the address space. This document imposes no other constraints about the relative position of addressable storage instances whenever they are created.[57]

3 Unless stated otherwise, a storage instance is *exposed* if a pointer value p of effective type T* with this provenance is used in the following contexts:[58]

— Any byte of the object representation of p is used in an expression.[59]

— Any byte of the object representation of p is passed to the **fwrite** library function.

— p is converted to an integer.

[55] An implementation might represent all pointers the same and with the same alignment requirements.

[56] All storage instances that do not originate from an object definition with **register** storage class are subject to the address-of operator & (6.5.3.2) and are addressable.

[57] This means that no relative positioning between storage instances and the objects they represent can be deduced from syntactical properties of the program (such as declaration order or order inside a parameter list) or sequencing properties of the execution (such as one instantiation happening before another).

[58] Pointer values with exposed provenance may alias in ways that cannot be predicted by simple data flow analysis.

[59] The exposure of bytes of the object representation can happen through a conversion of the address of a pointer object containing p to a character type and a subsequent access to the bytes, or by storing p in a **union** that allows access to all or parts of the object representation by means of a type that is not a pointer type or by a pointer type that gives rise to a different object representation.

— p is used as an argument to a %p conversion specifier of the **printf** family of library functions.

Other provisions of this document not withstanding, if the object representation of p is read through an lvalue of a pointer type S∗ that has the same representation and alignment requirements as T∗, that lvalue has the same provenance as p and the provenance is not exposed.[60] Exposure of a storage instance is irreversible and constitutes a side effect in the abstract state machine.

4 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

5 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[61]

6 Values stored in non-bit-field objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. ~~The value may be copied into an object of type~~ Converting a pointer of such an object to **unsigned char**∗ yields a pointer into the byte array of the storage instance such that the values of the first n ~~] (e.g., by **memcpy**); the resulting~~ bytes determine the value of the object; the offset of the first byte of these in the byte array is the *byte offset* of the object in its storage instance, the converted address is called the *byte address* of the object, and the set of bytes is called the *object representation* of the value. The object representation may be used to copy the value of the object into another object (e.g., by **memcpy**). Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations. The object representations of pointers and how they relate to the abstract addresses they represent are not further specified by this document.

7 Certain object representations need not represent a value of the object type. If ~~the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If~~ such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[62] Such a representation is called a trap representation.

8 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[63] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

9 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

10 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[64] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

11 Loads and stores of objects with atomic types are done with **memory_order_seq_cst** semantics.

Forward references: declarations (6.7), expressions (6.5), address and indirection operators (6.5.3.2), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3),

[60]This means that pointer members in a **union** can be used to reinterpret representations of different character and void pointers, different **struct** pointers, different **union** pointers or pointers with differently qualified target types.

[61]A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

[62]Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[63]Thus, for example, structure assignment need not copy any padding bits.

[64]It is possible for objects x and y with the same effective type T to have the same value when they are accessed as objects of type T, but to have different values in other contexts. In particular, if == is defined for type T, then `x == y` does not imply that `memcmp(&x, &y, sizeof (T))== 0`. Furthermore, `x == y` does not necessarily imply that x and y have the same value; other operations on values of type T might distinguish between them.

First, if the corresponding real type of either operand is `long double`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `long double`.

Otherwise, if the corresponding real type of either operand is `double`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `double`.

Otherwise, if the corresponding real type of either operand is `float`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `float`.[75)]

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

If both operands have the same type, then no further conversion is needed.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

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

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

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

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

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

1 An *lvalue* is an expression (with an object type other than `void`) that potentially designates an object;[76)] if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

2 Except when it is the operand of the `sizeof` operator, the unary & operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. ~~If~~ The behavior is undefined if the lvalue has an incomplete type~~and does not have array type, the behavior is undefined. If~~, if the object representation is a trap representation for the type,[77)] or if the lvalue designates an object of automatic storage duration that could have been declared with the `register` storage class

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

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

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

[77)]Character types have no trap representation, thus reading representation bytes of an addressable live storage instance is always defined.

(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~~ .

3 Additionally, if the type is a pointer type `T*`, a pointer value and an associated provenance, if any, is determined as follows:

— If the object representation represents a null pointer the result is a null pointer.

— If the last store to the representation array was with a pointer type `S*` that has the same representation and alignment requirements as `T*`, the result is the same address and provenance as the stored value.

— Otherwise, the object representation of the lvalue shall represent an abstract address within (or one-past) an exposed storage instance, such that the exposure happened before this lvalue conversion, and the result has that address and provenance.[78)]

The behavior is undefined if the lvalue conversion does not happen during the lifetime of the associated provenance, the address is not a valid address (or one-past) for the associated provenance, or the address is not correctly aligned for the type.

4 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.

5 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator,[79)] 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 a *null pointer constant*.[80)] 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. If the source type is signed, the operand is first converted to the corresponding unsigned type. The result is then determined in the following order:

[78)] If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

[79)] 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.

[80)] The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.19.

— The operand has a value that could have been the result of the conversion of a null pointer value. The result is a null pointer.

— The operand is an abstract address within or one past a live and exposed storage instance, such that the exposure happened before this integer-to-pointer conversion. The result is a pointer value with that address, provenance and target type.[81]

— The pointer value is indeterminate.

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.

6 Any pointer type may be converted to an integer type. ~~Except as previously specified~~For a null pointer, the result is chosen from a non-empty set of implementation-defined ~~. If the result cannot be represented in the integer~~ values.[82] If the pointer value is valid, its provenance is henceforth exposed. Except as previously specified, the result is the bit representation of the abstract address interpreted in the target type. If the abstract address has more significant bits than the width of the target type, the behavior is undefined. The result need not be in the range of values of any integer type. If the pointer is null or valid, the integer result converted back to the pointer type shall compare equal to the original pointer.[83] For two valid pointer values that compare equal, conversion to the same integer type yields identical values.

7 A pointer to an object type may be converted to a pointer to a different object type with the same provenance. If the resulting pointer is not correctly aligned[84] 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 or **`void`**, 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~~ is the byte address of the object.

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

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

9 **NOTE** If the result `p` of an lvalue conversion or integer-to-pointer conversion is the end address of an exposed storage instance A and the start address of another exposed storage instance B that happens to follow immediately in the address space, a conforming program must only use one of these provenances in any expressions that is derived from `p`, see 6.2.5.

The following three cases determine if `p` is used with one of A or B and must hence not be used otherwise:

— Operations that constitute a use of `p` with either A or B and do not prohibit a use with the other:

- any relational operator or pointer subtraction where the other operand `q` may have both provenances, that is where `q` is also the result of a similar conversion and where `p == q`;
- `q == p` and `q != p` regardless of the provenance of `q`;
- addition or subtraction of the value 0;
- conversion to integer.

For the latter, A and B must have been exposed before, and so a any choice of provenance, that would otherwise have exposed one of the storage instances, is consistent with any other use.

— Operations that, if otherwise well defined, constitute a use of `p` with A and prohibit any use with B:

- Any relational operator or pointer subtraction where the other operand `q` has provenance A and cannot have provenance B.
- `p + n` and `p[n]`, where `n` is an integer strictly less than `0`.
- `p - n`, where `n` is an integer strictly greater than `0`.

[81]If the address corresponds to more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

[82]It is recommended that 0 is a member of that set.

[83]Although such a round-trip conversion may be the identity for the pointer value, the side effect of exposing a storage instance still takes place.

[84]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.

— Operations that, if otherwise well defined, constitute a use of `p` with B and prohibit any use with A:

- Any relational operator or pointer subtraction where the other operand `q` has provenance B and cannot have provenance A.
- `p + n` and `p[n]`, where `n` is an integer strictly greater than `0`.
- `p - n`, where `n` is an integer strictly less than `0`.
- operations that access an object in B, that is indirection (`*p` or `p[n]` for `n == 0`) and member access (`p->member`).

6.5 Expressions

1 An *expression* is a sequence of operators and operands that specifies computation of a value,[101] or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If 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, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.[102]

3 The grouping of operators and operands is indicated by the syntax.[103] Except as specified later, side effects and value computations of subexpressions are unsequenced.[104]

4 Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[105] If a value is stored into an object ~~having no declared type~~ with allocated storage duration through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object ~~having no declared type~~ with allocated storage duration using **memcpy** or **memmove**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object ~~having no declared type~~with allocated storage duration, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[106]

— a type compatible with the effective type of the object,

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

[102]This paragraph renders undefined statement expressions such as

```
    i = ++i + 1;
    a[i++] = i;
```

while allowing

```
    i = i + 1;
    a[i] = i;
```

[103]The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses () (6.5.1), subscripting brackets [] (6.5.2.1), function-call parentheses () (6.5.2.2), and the conditional operator ?: (6.5.15).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

[104]In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

[105]An object with allocated storage duration has no declaration and thus no declared type.

[106]The intent of this list is to specify those circumstances in which an object can or cannot be aliased.

an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

4 A postfix expression followed by the `->` operator and an identifier designates a member of a structure or union object. The pointer value shall be valid, not be the end address of its provenance and be correctly aligned for the structure or union type. The value is that of the named member of the object to which the first expression points, and is an lvalue.[115] If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

5 Accessing a member of an atomic structure or union object results in undefined behavior.[116]

6 One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

7 **EXAMPLE 1** If `f` is a function returning a structure or union, and `x` is a member of that structure or union, `f().x` is a valid postfix expression but is not an lvalue.

8 **EXAMPLE 2** In:

```
struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;
```

the various members have the types:

```
s.i     int
s.ci    const int
cs.i    const int
cs.ci   const int
vs.i    volatile int
vs.ci   volatile const int
```

9 **EXAMPLE 3** The following is a valid fragment:

```
union {
      struct {
            int    alltypes;
      } n;
      struct {
            int    type;
            int    intnode;
      } ni;
      struct {
            int    type;
            double doublenode;
      } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
      if (sin(u.nf.doublenode) == 0.0)
            /* ... */
```

The following is not a valid fragment (because the union type is not visible within function `f`):

[115]If `&E` is a valid pointer expression (where & is the "address-of" operator, which generates a pointer to its operand), the expression `(&E)->MOS` is the same as `E.MOS`.

[116]For example, a data race would occur if access to the entire structure or union in one thread conflicts with access to a member from another thread, where at least one access is a modification. Members can be safely accessed using a non-atomic object which is assigned to or from the atomic object.

Semantics

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

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

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

6.5.3.2 Address and indirection operators

Constraints

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

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

Semantics

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

4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". ~~If an invalid value has been assigned to the pointer , the behavior of the unary * operator is undefined~~The pointer value shall be valid, not be the end address of its provenance and be correctly aligned for "*type*".[121)]

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

6.5.3.3 Unary arithmetic operators

Constraints

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

Semantics

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

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

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

5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to

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

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

6.5.6 Additive operators

Syntax

1 *additive-expression:*
  multiplicative-expression
  additive-expression **+** *multiplicative-expression*
  additive-expression **-** *multiplicative-expression*

Constraints

2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3 For subtraction, one of the following shall hold:

— both operands have arithmetic type;

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

— the left operand is a pointer to a complete object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

Semantics

4 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

5 The result of the binary `+` operator is the sum of the operands.

6 The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

7 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.

8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. ~~In other words, if the expression `P` points to the i-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object.~~ If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated. The result pointer has the same provenance as the pointer operand.[125)]

9 When two pointers are subtracted, both shall be valid. If they compare equal the result is 0. Otherwise they shall have the same provenance and point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is **`ptrdiff_t`** defined in the `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. ~~In other words, if the~~

[125)] If the pointer operand `P` had been the result of an integer-to-pointer or **`scanf`** conversion that could have two possible provenances, and the integer value added or subtracted is not 0, the provenance S for the additive operation (and henceforth other operations with `P`) must be such that the result lies in S (or one beyond).

10 **NOTE 1** If the expression P points to the i-th element of an array object, the expressions (P)+N (equivalently, N+(P)) and (P)-N (where N has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression P points to the last element of an array object, the expression (P)+1 points one past the last element of the array object, and if the expression Q points one past the last element of an array object, the expression (Q)-1 points to the last element of the array object.

11 **NOTE 2** If the expressions P and Q point to, respectively, the i-th and j-th elements of an array object, the expression (P)-(Q) has the value $i-j$ provided the value fits in an object of type **ptrdiff_t**. Moreover, if the expression P points either to an element of an array object or one past the last element of an array object, and the expression Q points to the last element of the same array object, the expression ((Q)+1)-(P) has the same value as ((Q)-(P))+1 and as-((P)-((Q)+1)), and has the value zero if the expression P points one past the last element of the array object, even though the expression (Q)+1 does not point to an element of the array object.

12 **NOTE 3** Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which can overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

13 **EXAMPLE** Pointer arithmetic is well defined with pointers to variable length array types.

```
        {
                int n = 4, m = 3;
                int a[n][m];
                int (*p)[m] = a;  // p == &a[0]
                p += 1;           // p == &a[1]
                (*p)[2] = 99;     // a[1][2] == 99
                n = p - a;        // n == 1
        }
```

14 If array a in the above example were declared to be an array of known constant size, and pointer p were declared to be a pointer to an array of the same known constant size (pointing to a), the results would be the same.

Forward references: array declarators (6.7.6.2), common definitions <stddef.h> (7.19).

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*

additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is $\texttt{E1} \times 2^{\texttt{E2}}$, reduced modulo one more than the maximum value representable in the result type. If E1 has a signed type and nonnegative value, and $\texttt{E1} \times 2^{\texttt{E2}}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $\texttt{E1}/2^{\texttt{E2}}$. If E1 has a signed type and a negative value, the resulting value is implementation-defined.

6.5.8 Relational operators

Syntax

1 *relational-expression:*
  shift-expression
  relational-expression **<** *shift-expression*
  relational-expression **>** *shift-expression*
  relational-expression **<=** *shift-expression*
  relational-expression **>=** *shift-expression*

Constraints

2 One of the following shall hold:

— both operands have real type; or

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

Semantics

3 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

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~~they shall both be valid and have the same provenance. The result depends on the relative ordering of their abstract addresses.

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.[126] 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 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

[126]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`".

— 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.[127] None of the operands shall be indeterminate. 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 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.

6 ~~Two pointers~~ If one operand is null they compare equal if and only if ~~both are nullpointers, both~~ the other operand is null. Otherwise, if both operands are pointers to ~~the same object (including a pointer to an object and a subobject at its beginning) or function, both~~ function type they compare equal if and only if they refer to the same function. Otherwise, they 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 addressspace.~~ objects and compare equal if and only if they have the same abstract address.

7 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.

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

```
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

assignment-operator: one of
  `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

Constraints

2 An assignment operator shall have a modifiable lvalue as its left operand.

Semantics

3 An assignment operator stores a value in the object designated by the left operand. If a non-null pointer is stored by an assignment operator, either directly or within a structure or union object, the stored pointer object has the same provenance as the original. An assignment expression has the value of the left operand after the assignment,[129] but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

6.5.16.1 Simple assignment

Constraints

1 One of the following shall hold:[130]

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

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

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

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

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

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

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

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

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

9 A member of a structure or union may have any complete object type other than a variably modified type.[140] In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[141] its width is preceded by a colon.

10 A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[142] If the value 0 or 1 is stored into a nonzero-width bit-field of type **_Bool**, the value of the bit-field shall compare equal to the value stored; a **_Bool** bit-field has the semantics of a **_Bool**.

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

12 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[143] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

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

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

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

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

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

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

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

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

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

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

one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations.

Semantics

7 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[180] the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.6.3 for a parameter type list; the resulting type shall be a complete object type.

8 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

9 Each parameter has automatic storage duration; its identifier is an lvalue.[181] ~~The layout of the storage for parameters is unspecified.~~[182]

10 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)

11 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

12 Unless otherwise specified, if the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.

13 **EXAMPLE 1** In the following:

```
extern int max(int a, int b)
{
      return a  >  b ? a: b;
}
```

extern is the storage-class specifier and **int** is the type specifier; `max(int a, int b)` is the function declarator; and

```
{ return a  >  b ? a: b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

[179] The intent is that the type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);            // type F is “function with no parameters
                                // returning int”
F f, g;                         // f and g both have type compatible with F
F f { /* ... */ }               // WRONG: syntax/constraint error
F g() { /* ... */ }             // WRONG: declares that g returns a function
int f(void) { /* ... */ }       // RIGHT: f has type compatible with F
int g() { /* ... */ }           // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }        // e returns a pointer to a function
F *((e))(void) { /* ... */ }    // same:  parentheses irrelevant
int (*fp)(void);                // fp points to a function that has type F
F *Fp;                          // Fp points to a function that has type F
```

[180] See “future language directions” (6.11.7).

[181] A parameter identifier cannot be redeclared in the function body except in an enclosed block.

[182] As any object with automatic storage duration, each parameter gives rise to its own storage instance. Thus the relative layout of parameters in memory is unspecified.

5 Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments.[209] Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

6 Unless otherwise specified, library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.[210]

7 Unless otherwise specified, library functions by themselves do not expose storage instances, but library functions that execute application specific callbacks[211] may expose storage instances through calls into these callbacks.

8 **EXAMPLE** The function **atoi** can be used in any of several ways:

— by use of its associated header (possibly generating a macro expansion)

```
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

— by use of its associated header (assuredly generating a true function reference)

```
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

— by explicit declaration

```
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

[209]This means, for example, that an implementation is not permitted to use a **static** object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. Similarly, an implementation of **memcpy** is not permitted to copy bytes beyond the specified length of the destination object and then restore the original values because it could cause a data race if the program shared those bytes between threads.

[210]This allows implementations to parallelize operations if there are no visible side effects.

[211]The following library functions call application specific functions that they or related functions receive as arguments: **bsearch**, **call_once**, **exit** (for **atexit** handlers), **qsort**, **quick_exit** (for **at_quick_exit** handlers), and **thrd_exit** (for thread specific storage).

Description

2 The **longjmp** function restores the environment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the **setjmp** macro has terminated execution[276] in the interim, or if the invocation of the **setjmp** macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine[277] have state, as of the time the **longjmp** function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and **longjmp** call are indeterminate.

Returns

4 After **longjmp** is completed, thread execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by `val`. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if `val` is 0, the **setjmp** macro returns the value 1.

5 EXAMPLE The **longjmp** function that returns control back to the point of the **setjmp** invocation might cause ~~memory~~ the storage instance associated with a variable length array object to be squandered.

```
#include <setjmp.h>
jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
        int x[n];           // valid:  f is not terminated
        setjmp(buf);
        g(n);
}

void g(int n)
{
        int a[n];           // a may remain allocated
        h(n);
}

void h(int n)
{
        int b[n];           // b may remain allocated
        longjmp(buf, 2);    // might cause memory loss
}
```

[276]For example, by executing a **return** statement or because another **longjmp** call has caused a transfer to a **setjmp** invocation in a function earlier in the set of nested calls.

[277]This includes, but is not limited to, the floating-point status flags and the state of open files.

`a,A` A **`double`** argument representing a floating-point number is converted in the style *[-]*`0`*xh.hhhh*`p`*±d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[307] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **`FLT_RADIX`** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **`FLT_RADIX`** is not a power of 2, then the precision is sufficient to distinguish[308] values of type **`double`**, except that trailing zeros may be omitted; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` length modifier is present, the **`int`** argument is converted to an **`unsigned char`**,and the resulting character is written.

If an `l` length modifier is present, the **`wint_t`** argument is converted as if by an `ls` conversion specification with no precision and an argument that points to the initial element of a two-element array of **`wchar_t`**, the first element containing the **`wint_t`** argument to the `lc` conversion specification and the second a null wide character.

`s` If no `l` length modifier is present, the argument shall be a pointer to the initial element of an array of character type.[309] Characters from the array arewritten up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of **`wchar_t`** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[310]

`p` The argument shall be a pointer to **`void`**. The value of the pointer shall be valid or null. It is converted to a sequence of printing characters, in an implementation-defined manner. If the value of the pointer is valid its provenance is henceforth exposed.

`n` The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **`fprintf`**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%` A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

[307]Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.
[308]The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **`FLT_RADIX`** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.
[309]No special provisions are made for multibyte characters.
[310]Redundant shift sequences can result if multibyte characters have a state-dependent encoding.

the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches ~~an~~ the same implementation-defined set of sequences ~~, which should be the same as the set of sequences~~ of characters that may be produced by the `%p` conversion of the **`fprintf`** function. The corresponding argument `ptr` shall be a pointer to a pointer to **`void`**. ~~The input item is converted to a pointer value in an implementation-defined manner.~~

- If the input ~~item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the conversion is undefined.~~ sequence could have been printed from a null pointer value, `*ptr` is assigned a null pointer value.
- Otherwise, if the input sequence could have been printed from a valid pointer x and if the address x currently refers to an exposed storage instance, a valid pointer with address x and the provenance of that storage instance is stored in `*ptr`.[316]
- Otherwise `*ptr` becomes indeterminate.

`n` No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **`fscanf`** function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the **`fscanf`** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

`%` Matches a single `%` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[317]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

15 Trailing white-space characters(including new-line characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns

16 The **`fscanf`** function returns the value of the macro **`EOF`** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

17 **EXAMPLE 1** The call:

```
        #include <stdio.h>
        /* ... */
        int n, i; float x; char name[50];
        n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
        25 54.32E-1 thompson
```

[316] Thus, the constructed pointer value has a valid provenance. Nevertheless, because the original storage instance might be dead and a new storage instance might live at the same address, this provenance can be different from the provenance that gave rise to the print operation.If x can be an address with more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

[317] See "future library directions" (7.31.11).

Description

2 The **fread** function reads, into the array pointed to by ptr, up to nmemb elements whose size is specified by size, from the stream pointed to by stream. For each object, size calls are made to the **fgetc** function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns

3 The **fread** function returns the number of elements successfully read, which may be less than nmemb if a read error or end-of-file is encountered. If size or nmemb is zero, **fread** returns zero and the contents of the array and the state of the stream remain unchanged.

7.21.8.2 The fwrite function

Synopsis

1
```
        #include <stdio.h>
        size_t fwrite(const void * restrict ptr,
              size_t size, size_t nmemb,
              FILE * restrict stream);
```

Description

2 The **fwrite** function writes, from the array pointed to by ptr, up to nmemb elements whose size is specified by size, to the stream pointed to by stream. For each object, size calls are made to the **fputc** function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

3 If the object (or part thereof) corresponding to the first size*nmemb bytes referred by ptr contains a valid pointer value with provenance x, the **fwrite** function exposes x.

Returns

4 The **fwrite** function returns the number of elements successfully written, which will be less than nmemb only if a write error is encountered. If size or nmemb is zero, **fwrite** returns zero and the state of the stream remains unchanged.

7.21.9 File positioning functions

7.21.9.1 The fgetpos function

Synopsis

1
```
        #include <stdio.h>
        int fgetpos(FILE * restrict stream,
              fpos_t * restrict pos);
```

Description

2 The **fgetpos** function stores the current values of the parse state (if any) and file position indicator for the stream pointed to by stream in the object pointed to by pos. The values stored contain unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Returns

3 If successful, the **fgetpos** function returns zero; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Forward references: the **fsetpos** function (7.21.9.3).

function.

Returns

4 The `srand` function returns no value.

5 **EXAMPLE** The following functions define a portable implementation of `rand` and `srand`.

```
        static unsigned long int next = 1;

        int rand(void)   //  RAND_MAX assumed to be 32767
        {
              next = next * 1103515245 + 12345;
              return (unsigned int)(next/65536) % 32768;
        }

        void srand(unsigned int seed)
        {
              next = seed;
        }
```

7.22.3 Storage management functions

1 The order and contiguity of storage instances allocated by successive calls to the **`aligned_alloc`**, **`calloc`**, **`malloc`**, and **`realloc`** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the ~~space~~ storage instance allocated (until the ~~space~~ storage instance is explicitly deallocated). The lifetime of an allocated ~~object~~ storage instance extends from the allocation until the deallocation. Each such allocation shall yield a pointer to ~~an object~~ a storage instance that is disjoint from any other ~~object~~storage instance. The pointer returned points to the start ~~(lowest byte address)~~ address of the allocated ~~space~~storage instance. If the ~~space~~ storage instance cannot be allocated, a null pointer is returned. If the size of the ~~space~~ storage instance requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

2 For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only ~~memory locations~~ storage instances accessible through their arguments and not other static duration storage ~~.~~ instances. These functions may, however, visibly modify the storage instance that they allocate or deallocate. Calls to these functions that allocate or deallocate storage instances in a particular region of memory (identified by its address and size) shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.[326]

7.22.3.1 The `aligned_alloc` function

Synopsis

1
```
        #include <stdlib.h>
        void *aligned_alloc(size_t alignment, size_t size);
```

Description

2 The **`aligned_alloc`** function allocates ~~space for an object~~ a storage instance whose alignment is specified by `alignment`, whose size is specified by `size`, and whose ~~value is indeterminate~~byte values are unspecified. If the value of `alignment` is not a valid alignment supported by the implementation the function shall fail by returning a null pointer.

[326]This means that an implementation may only reuse a valid address that is computed from an allocated storage instance for a different allocated storage instance if the calls to allocate and deallocate the storage instances synchronize.

Returns

3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.3.2 The calloc function

Synopsis

1
```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description

2 The **calloc** function allocates ~~space~~ a storage instance for an array of nmemb objects, each of whose size is size. The ~~space~~ storage instance is initialized to all bits zero.[327]

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.3.3 The free function

Synopsis

1
```
#include <stdlib.h>
void free(void *ptr);
```

Description

2 The **free** function causes the ~~space~~ storage instance pointed to by ptr to be deallocated, that is, made available for further ~~allocation.~~ use.[328] If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.22.3.4 The malloc function

Synopsis

1
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

2 The **malloc** function allocates ~~space for an object~~ a storage instance whose size is specified by size and whose ~~value is indeterminate~~byte values are unspecified.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated ~~space.~~ storage instance.

7.22.3.5 The realloc function

Synopsis

1
```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

[327] Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

[328] That means that the implementation may reuse the address range of the storage instance (determined by ptr and its size) for any storage instance whose instantiation synchronizes with the call.

Description

2 The **realloc** function deallocates the old ~~object~~ storage instance pointed to by ptr and returns a pointer to a new ~~object~~ storage instance that has the size specified by size. The ~~contents~~ bytes of the new ~~object shall be the same as that of the old object prior to deallocation,~~ storage instance up to the lesser of the new and old sizes ~~.~~ have the same value as the bytes in the same positions of the old storage instance.[329] Any bytes in the new ~~object~~ storage instance beyond the size of the old object have ~~indeterminate~~ unspecified values.

3 If ptr is a null pointer, the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if ptr does not match a pointer earlier returned by a ~~memory~~ storage management function, or if the ~~space~~ storage instance has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If size is nonzero and ~~memory for the new object is not~~ no storage instance is allocated, the old ~~object~~ storage instance is not deallocated. If size is zero and ~~memory for the new object is not~~ no storage instance is allocated, it is implementation-defined whether the old ~~object~~ storage instance is deallocated. If the old ~~object~~ storage instance is not deallocated, ~~its value~~ it shall be unchanged.

Returns

4 The **realloc** function returns a pointer to the new ~~object~~ storage instance (which may have the same value as a pointer to the old ~~object),~~ storage instance), or a null pointer if ~~the new object has not~~ no new storage instance has been allocated.

7.22.4 Communication with the environment

7.22.4.1 The abort function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void abort(void);
```

Description

2 The **abort** function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise**(**SIGABRT**).

Returns

3 The **abort** function does not return to its caller.

7.22.4.2 The atexit function

Synopsis

1
```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The **atexit** function registers the function pointed to by func, to be called without arguments at normal program termination.[330] It is unspecified whether a call to the **atexit** function that does not happen before the **exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

[329] Thus this initial part behaves as if it were copied by **memcpy**. In particular, the initial part of the new storage instance represents objects with same value and effective type as the initial part of the old storage instance, if any.

[330] The **atexit** function registrations are distinct from the **at_quick_exit** registrations, so applications might need to call both registration functions with the same argument.

7.24 String handling `<string.h>`

7.24.1 String function conventions

1 The header `<string.h>` declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.[340] The type is **size_t** and the macro is **NULL** (both described in 7.19). Various methods are used for determining the lengths of the arrays, but in all cases a **char** * or **void** * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 Where an argument declared as **size_t** `n` specifies the length of the array for a function, `n` can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

3 For all functions in this subclause, each character shall be interpreted as if it had the type **unsigned char** (and therefore every possible object representation is valid and has a different value).

7.24.2 Copying functions

7.24.2.1 The memcpy function

Synopsis

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

Description

2 The **memcpy** function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

3 If the object representation of a non-null pointer is copied by the **memcpy** function , either directly or within a structure or union object, the pointer copy has the same provenance as the original.

Returns

4 The **memcpy** function returns the value of `s1`.

7.24.2.2 The memmove function

Synopsis

1
```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

2 The **memmove** function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` characters from the object pointed to by `s2` are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` characters from the temporary array are copied into the object pointed to by `s1` .

3 If the object representation of a non-null pointer is copied by the **memmove** function , either directly or within a structure or union object, the pointer copy has the same provenance as the original.

Returns

4 The **memmove** function returns the value of `s1`.

7.24.2.3 The strcpy function

[340] See "future library directions" (7.31.13).

3 A null pointer value is associated with the newly created key in all existing threads. Upon subsequent thread creation, the value associated with all keys is initialized to a null pointer value in the new thread.

4 Destructors associated with thread-specific storage are not invoked at program termination.

5 The `tss_create` function shall not be called from within a destructor.

Returns

6 If the `tss_create` function is successful, it sets the thread-specific storage pointed to by `key` to a value that uniquely identifies the newly created pointer and returns `thrd_success`; otherwise, `thrd_error` is returned and the thread-specific storage pointed to by `key` is set to an indeterminate value.

7.26.6.2 The `tss_delete` function

Synopsis

1
```
        #include <threads.h>
        void tss_delete(tss_t key);
```

Description

2 The `tss_delete` function releases any resources used by the thread-specific storage identified by `key`. The `tss_delete` function shall only be called with a value for `key` that was returned by a call to `tss_create` before the thread commenced executing destructors.

3 If `tss_delete` is called while another thread is executing destructors, whether this will affect the number of invocations of the destructor associated with `key` on that thread is unspecified.

4 Calling `tss_delete` will not result in the invocation of any destructors.

Returns

5 The `tss_delete` function returns no value.

7.26.6.3 The `tss_get` function

Synopsis

1
```
        #include <threads.h>
        void *tss_get(tss_t key);
```

Description

2 The `tss_get` function returns the value for the current thread held in the thread-specific storage identified by `key`. The `tss_get` function shall only be called with a value for `key` that was returned by a call to `tss_create` before the thread commenced executing destructors.

Returns

3 The `tss_get` function returns the value for the current thread if successful, or zero if unsuccessful.

7.26.6.4 The `tss_set` function

Synopsis

1
```
        #include <threads.h>
        int tss_set(tss_t key, void *val);
```

Description

2 The `tss_set` function sets the value for the current thread held in the thread-specific storage identified by `key` to `val`. The `tss_set` function shall only be called with a value for `key` that was returned by a call to `tss_create` before the thread commenced executing destructors.

3 This action will not invoke the destructor associated with the key on the value being replaced.

4 If `val` is a valid pointer, its provenance is is henceforth exposed.

for an exact representation of the value; if the precision is missing and **`FLT_RADIX`** is not a power of 2, then the precision is sufficient to distinguish[368] values of type **`double`**, except that trailing zeros may be omitted; if the precision is zero and the **`#`** flag is not specified, no decimal-point wide character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A **`double`** argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` length modifier is present, the **`int`** argument is converted to a wide character as if by calling **`btowc`** and the resulting wide character is written.

If an `l` length modifier is present, the **`wint_t`** argument is converted to **`wchar_t`** and written.

`s` If no `l` length modifier is present, the argument shall be a pointer to the initial element of a character array containing a multibyte character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted, andwritten up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of **`wchar_t`** type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

`p` The argument shall be a pointer to **`void`**. The value of the pointer shall be valid or null. It is converted to a sequence of printing wide characters, in an implementation-defined manner. If the value of the pointer is valid its provenance is henceforth exposed.

`n` The argument shall be a pointer to signed integer into which is *written* the number of wide characters written to the output stream so far by this call to **`fwprintf`**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%` A `%` wide character is written. No argument is converted. The complete conversion specification shall be `%%`.

9 If a conversion specification is invalid, the behavior is undefined.[369] If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

10 In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

11 For `a` and `A` conversions, if **`FLT_RADIX`** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

12 For `a` and `A` conversions, if **`FLT_RADIX`** is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

[368]The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is **`FLT_RADIX`** and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point wide character.

[369]See "future library directions" (7.31.16).

`[` Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*).

If no `l` length modifier is present, characters from the input field are converted as if by repeated calls to the **`wcrtomb`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the `format` string, up to and including the matching right bracket (`]`). The wide characters between the brackets (the *scanlist*) compose the scanset, unless the wide character after the left bracket is a circumflex (`^`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket wide character is in the scanlist and the next following right bracket wide character is the matching right bracket that ends the specification; otherwise the first following right bracket wide character is the one that ends the specification. If a `-` wide character is in the scanlist and is not the first, nor the second where the first wide character is a `^`, nor the last character, the behavior is implementation-defined.

`p` Matches ~~an~~ the same implementation-defined set of sequences ~~, which should be the same as the set of sequences~~ of wide characters that may be produced by the `%p` conversion of the **`fwprintf`** function. The corresponding argument `ptr` shall be a pointer to a pointer to **`void`**. ~~The input item is converted to a pointer value in an implementation-defined manner.~~

- If the input ~~item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the conversion is undefined.~~ sequence could have been printed from a null pointer value, `*ptr` is assigned a null pointer value.
- Otherwise, if the input sequence could have been printed from a valid pointer x and if the address x currently refers to an exposed storage instance, a valid pointer with address x and the provenance of that storage instance is stored in `*ptr`.[373)]
- Otherwise `*ptr` becomes indeterminate.

`n` No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of wide characters read from the input stream so far by this call to the **`fwscanf`** function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the **`fwscanf`** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

`%` Matches a single `%` wide character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

13 If a conversion specification is invalid, the behavior is undefined.[374)]

14 The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and behave the same as, respectively, `a`, `e`, `f`, `g`, and `x`.

373) Thus, the constructed pointer value has a valid provenance. Nevertheless, because the original storage instance might be dead and a new storage instance might live at the same address, this provenance can be different from the provenance that gave rise to the print operation.If x can be an address with more than one provenance, only one of these shall be used in the sequel, see 6.2.5.

374) See "future library directions" (7.31.16).

— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

— Whether or not a size expression is evaluated when it is part of the operand of a **sizeof** operator and changing the value of the size expression would not affect the result of the operator (6.7.6.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.9).

— The relative layout of storage instances for function parameters (6.9.1).

— When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `(`, and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `(`, whether that is considered a nested replacement (6.10.3).

— The order in which **#** and **##** operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— The line number following a directive of the form **#line** **__LINE__** *new-line* (6.10.4).

— The state of the floating-point status flags when execution passes from a part of the program translated with **FENV_ACCESS** "off" to a part translated with **FENV_ACCESS** "on" (7.6.1).

— The order in which **feraiseexcept** raises floating-point exceptions, except as stated in F.8.6 (7.6.3.3).

— Whether **math_errhandling** is a macro or an identifier with external linkage (7.12).

— The results of the **frexp** functions when the specified value is not a floating-point number (7.12.6.4).

— The numeric result of the **ilogb** functions when the correct value is outside the range of the return type (7.12.6.5, F.10.3.5).

— The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F.10.6.5).

— The value stored by the **remquo** functions in the object pointed to by `quo` when `y` is zero (7.12.10.3).

— Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.15).

— Whether **setjmp** is a macro or an identifier with external linkage (7.13).

— Whether **va_copy** and **va_end** are macros or identifiers with external linkage (7.16.1).

— The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an `a` or `A` conversion specifier (7.21.6.1, 7.29.2.1).

— The value of the file position indicator after a successful call to the **ungetc** function for a text stream, or the **ungetwc** function for any stream, until all pushed-back characters are read or discarded (7.21.7.10, 7.29.3.10).

— The details of the value stored by the **fgetpos** function (7.21.9.1).

— The details of the value returned by the **ftell** function for a text stream (7.21.9.4).

— Whether the **strtod**, **strtof**, **strtold**, **wcstod**, **wcstof**, and **wcstold** functions convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.22.1.4, 7.29.4.1.1).

— A c, s, or [conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is s or [) (7.21.6.2, 7.29.2.2).

— A c, s, or [conversion specifier with an l qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

— The input item for a %p conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

— The **vfprintf**, **vfscanf**, **vprintf**, **vscanf**, **vsnprintf**, **vsprintf**, **vsscanf**, **vfwprintf**, **vfwscanf**, **vswprintf**, **vswscanf**, **vwprintf**, or **vwscanf** function is called with an improperly initialized **va_list** argument, or the argument is used (other than in an invocation of **va_end**) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

— The contents of the array supplied in a call to the **fgets** or **fgetws** function are used after a read error occurred (7.21.7.2, 7.29.3.2).

— The file position indicator for a binary stream is used after a call to the **ungetc** function where its value was zero before the call (7.21.7.10).

— The file position indicator for a stream is used after an error occurred during a call to the **fread** or **fwrite** function (7.21.8.1, 7.21.8.2).

— A partial element read by a call to the **fread** function is used (7.21.8.1).

— The **fseek** function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the **ftell** function on a stream associated with the same file or whence is not **SEEK_SET** (7.21.9.2).

— The **fsetpos** function is called to set a position that was not returned by a previous successful call to the **fgetpos** function on a stream associated with the same file (7.21.9.3).

— A non-null pointer returned by a call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function with a zero requested size is used to access an object (7.22.3).

— The value of a pointer that refers to ~~space~~ storage deallocated by a call to the **free** or **realloc** function is used (7.22.3).

— The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a ~~memory~~ storage management function, or the ~~space~~ storage has been deallocated by a call to **free** or **realloc** (7.22.3.3, 7.22.3.5).

— The value of the object allocated by the **malloc** function is used (7.22.3.4).

— The values of any bytes in a new object allocated by the **realloc** function beyond the size of the old object are used (7.22.3.5).

— The program calls the **exit** or **quick_exit** function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

— During the call to a function registered with the **atexit** or **at_quick_exit** function, a call is made to the **longjmp** function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).

— The string set up by the **getenv** or **strerror** function is modified by the program (7.22.4.6, 7.24.6.2).

— A signal is raised while the **quick_exit** function is executing (7.22.4.7).

— A command is executed through the **system** function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

— Whether the last line of a text stream requires a terminating new-line character (7.21.2).

— Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

— The number of null characters that may be appended to data written to a binary stream (7.21.2).

— Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

— Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

— The characteristics of file buffering (7.21.3).

— Whether a zero-length file actually exists (7.21.3).

— The rules for composing valid file names (7.21.3).

— Whether the same file can be simultaneously open multiple times (7.21.3).

— The nature and choice of encodings used for multibyte characters in files (7.21.3).

— The effect of the **`remove`** function on an open file (7.21.4.1).

— The effect if a file with the new name exists prior to a call to the **`rename`** function (7.21.4.2).

— Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

— Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

— The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

— The output for `%p` conversion in the **`fprintf`** or **`fwprintf`** function (7.21.6.1, 7.29.2.1).

— The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the **`fscanf`** or **`fwscanf`** function (7.21.6.2, 7.29.2.1).

— The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the **`fscanf`** or **`fwscanf`** function (7.21.6.2, 7.29.2.2).

— The value to which the macro **`errno`** is set by the **`fgetpos`**, **`fsetpos`**, or **`ftell`** functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

— The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the **`strtod`**, **`strtof`**, **`strtold`**, **`wcstod`**, **`wcstof`**, or **`wcstold`** function (7.22.1.4, 7.29.4.1.1).

— Whether or not the **`strtod`**, **`strtof`**, **`strtold`**, **`wcstod`**, **`wcstof`**, or **`wcstold`** function sets **`errno`** to **`ERANGE`** when underflow occurs (7.22.1.4, 7.29.4.1.1).

— Whether the **`calloc`**, **`malloc`**, **`realloc`**, and **`aligned_alloc`** functions return a null pointer or a pointer to ~~an allocated object~~ storage when the size requested is zero (7.22.3).

— Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the **`abort`** or **`_Exit`** function is called (7.22.4.1, 7.22.4.5).

— The termination status returned to the host environment by the **`abort`**, **`exit`**, **`_Exit`**, or **`quick_exit`** function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

— The value returned by the **`system`** function when its argument is not a null pointer (7.22.4.8).

— The range and precision of times representable in **`clock_t`** and **`time_t`** (7.27).