A Provenance-aware Memory Object Model for C

Draft Technical Specification

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

ISO TC1/SC22/WG14
document number: **N2577**
document date: 2020-10-4

ISO TC1/SC22/WG21 SG22
document number: **P2318R0**
document date: 2021-2-15

Contents

Foreword **3**

1 Introduction **4**
1.1 Previous related papers . . . 4
1.2 The basic idea . . . 5
1.3 Newly introduced terms . . . 7
1.3.1 Storage instance . . . 7
1.3.2 Provenance . . . 7
1.3.3 Abstract address . . . 7
1.3.4 Pointer exposure and synthesis . . . 7
1.4 Operations . . . 8
1.4.1 Exposing and non-exposing operations . . . 8
1.4.2 Reconstructing operations . . . 8
1.4.3 Pointer inquiry . . . 8
1.4.4 Pointer arithmetic . . . 8
1.5 Ambiguous Provenance . . . 9

2 Scope **10**

3 Normative references **11**

4 Terms and definitions **12**
4.1 pointer provenance . . . 12
4.2 storage instance . . . 12

5 Specifications **13**

A Examples (informative) **14**
A.1 Introduction . . . 14
A.2 Basic pointer provenance . . . 15
A.3 Refining the basic provenance model to support pointer construction via casts, representation accesses, etc. . . . 17
A.4 Refining the basic provenance model: phenomena and examples . . . 18
A.5 Implications of provenance semantics for optimisations . . . 26
A.6 Testing the example behaviour in Cerberus . . . 31
A.7 Testing the example behaviour in mainstream C implementations . . . 32

B Detailed semantics (informative) **33**
B.1 The PNVI-ae-udi, PNVI-ae, PNVI-plain, and PVI semantics . . . 33
B.1.1 The memory object model interface . . . 34
B.2 The memory object model state . . . 34
B.2.1 Mappings between abstract values and representation abstract-byte sequences . . . 35
B.2.2 Memory operations . . . 35
B.2.3 Pointer / Integer operations . . . 37
B.2.4 No-expose annotation . . . 40
B.2.5 Provenance of other operations . . . 40

C Modifications to ISO/IEC 9899:2018 (normative) **41**

Bibliography **89**

Foreword

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

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

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

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

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

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

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.

This Technical Specification provides a solution for this: a provenance-aware memory object model for C to put C programmers and implementers on a solid footing in this regard. This draft Technical Specification is based on, and incorporates the content of, three earlier WG14 documents:

- N2362 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 introduced the proposal and gives the proposed change to the standard text, presented as change-highlighted pages of the standard. Here, as appropriate for a Technical Specification, we instead present the proposed changes with respect to ISO/IEC 9899:2018.
- 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 the first draft of this Technical Specification, the latter two parts have identical text to those earlier N-papers. 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. Following N2363, we include 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 been discussed with WG14 (in multiple meetings) and at the March 2019 Cologne meeting of WG21, in SG12 *UB & Vulnerabilities*. Both of these have approved the overall direction, subject to implementation experience. It has also been discussed with the Clang/LLVM and GCC communities, 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.

1.1 Previous 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

1.2 The basic idea

This section follows the start of Section 2 of N2363.

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.

```
1  #include <stdio.h>
2  #include <string.h>
3  int y=2, x=1;
4  int main() {
5    int *p = &x + 1;
6    int *q = &y;
7    printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
8    if (memcmp(&p, &q, sizeof(p)) == 0) {
9      *p = 11;  // does this have undefined behaviour?
10     printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
11   }
12 }
```

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.

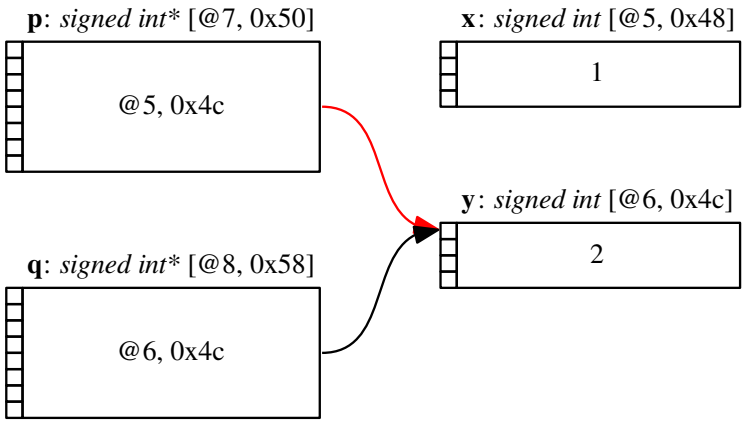
Basic provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: we associate a *provenance* with every pointer value, identifying the original storage instance that the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



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. N2363 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 from WG14 and WG21 discussions.

1.3 Newly introduced terms

1.3.1 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 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.

1.3.2 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 invalid.

1.3.3 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 "usual" architectures where **uintptr_t** exists, the abstract address of a pointer value `p` is just `(uintptr_t)p`. Architectures that do not have **uintptr_t** should be able to define an abstract address that is consistent with the other operations that they allow on pointers.

1.3.4 Pointer exposure and synthesis

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 a pointer value `p` can be constructed from scratch. In such a case we say that `p` has been *synthesized*. We require that a storage instance of such a synthesized 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.

[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.

1.4 Operations

1.4.1 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.

1.4.2 Reconstructing operations

Lvalue conversion Lvalue conversion for a pointer object that has somehow been synthesized 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 synthesize a pointer value for which the compiler has assumptions about non-aliasing, we must be sure that the provenance of that newly synthesized pointer value had been exposed before.

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.

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 synthesized as before by lvalue conversion.

1.4.3 Pointer inquiry

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.

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.

1.4.4 Pointer arithmetic

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

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.

1.5 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.

2 Scope

This document specifies the form and establishes the interpretation of programs written in the C programming language. It is not a complete specification of that language but amends ISO/IEC 9899:2018 by providing a Technical Specification that constrains and clarifies the Memory Object Model implicit there.

3 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382:2015, *Information technology – Vocabulary.* Available from the ISO online browsing platform at http://www.iso.org/obp.

ISO/IEC 9899:2018, *Programming languages – C*

ISO 80000–2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology.*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000–2, and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp
- IEC Electropedia: available at http://www.electropedia.org/

The following terms are explicitly defined in this document and are not to be presumed to refer implicitly to similar terms defined elsewhere. The clauses in the modified version of ISO/IEC 9899:2018 are 3.17 and 3.20, respectively.

4.1 pointer provenance

provenance

an entity that is associated to a pointer value in the abstract machine, which is either empty, or the identity of a storage instance

4.2 storage instance

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

5 Specifications

The normative specification is given in its entirety by modifications to ISO/IEC 9899:2018. These are provided by normative Annex C.

Prior to that, two informative annexes provide examples (Annex A) and detailed semantics (Annex B) for the different variants of the memory model that have been discussed in the introduction.

A Examples (informative)

This annex discusses the design of provenance semantics for C, looking at a series of examples. We consider the three variants of the provenance-not-via-integer (PNVI) model: PNVI plain, PNVI address-exposed (PNVI-ae) and PNVI address-exposed user-disambiguation (PNVI-ae-udi), and also the provenance-via-integers (PVI) model. The examples include those of *Exploring C Semantics and Pointer Provenance* [POPL 2019] (also available as ISO WG14 N2311 `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2311.pdf`), with several additions.

A.1 Introduction

The new material for PNVI-address-exposed and PNVI address-exposed user-disambiguation models starts in §A.3, but first we introduce introduce the problem in general and describe the basic pointer provenance semantics.

The semantics of pointers and memory objects in C has been a vexed question for many years. A priori, one might imagine two language-design extremes: a concrete model that exposes the memory semantics of the underlying hardware, with memory being simply a finite partial map from machine-word addresses to bytes and pointers that are simply machine words, and an abstract model in which the language types enforce hard distinctions, e.g. between numeric types that support arithmetic and pointer types that support dereferencing. C is neither of these. Its values are not abstract: the language intentionally permits manipulation of their underlying representations, via casts between pointer and integer types, `char*` pointers to access representation bytes, and so on, to support low-level systems programming. But C values also cannot be considered to be simple concrete values: at runtime a C pointer will typically just be a machine word, but compiler analysis reasons about abstract notions of the provenance of pointers, and compiler optimisations rely on assumptions about these for soundness. Particularly relevant here, some compiler optimisations rely on alias analysis to deduce that two pointer values do not refer to the same object, which in turn relies on assumptions that the program only constructs pointer values in "reasonable" ways (with other programs regarded as having undefined behaviour, UB). The committee response to Defect Report DR260 [Fea04] states that implementations can track the origins (or "provenance") of pointer values, *"the implementation is entitled to take account of the provenance of a pointer value when determining what actions are and are not defined"*, but exactly what this "provenance" means is left undefined, and it has never been incorporated into the standard text. Even what a memory object is is not completely clear in the standard, especially for aggregate types and for objects within heap regions.

Second, in some respects there are significant discrepancies between the ISO standard and the de facto standards, of C as it is implemented and used in practice. Major C codebases typically rely on particular compiler flags, e.g. `-fno-strict-aliasing` or `-fwrapv`, that substantially affect the semantics but which standard does not attempt to describe, and some idioms are UB in ISO C but relied on in practice, e.g. comparing against a pointer value after the lifetime-end of the object it pointed to. There is also not a unique de facto standard: in reality, one has to consider the expectations of expert C programmers and compiler writers, the behaviours of specific compilers, and the assumptions about the language implementations that the global C codebase relies upon to work correctly (in so far as it does). Our recent surveys [MML+16, MS16b] of the first revealed many discrepancies, with widely conflicting responses to specific questions.

Third, the ISO standard is a prose document, as is typical for industry standards. The lack of mathematical precision, while also typical for industry standards, has surely contributed to the accumulated confusion about C, but, perhaps more importantly, the prose standard is not *executable as a test oracle*. One would like, given small test programs, to be able to automatically compute the sets of their allowed behaviours (including whether they have UB). Instead, one has to do painstaking argument with respect to the text and concepts of the standard, a time-consuming and error-prone task that requires great expertise, and which will sometimes run up against the areas where the standard is unclear or differs with practice. One also cannot use conventional implementations to find the sets of all allowed behaviours, as (a) the standard is a loose specification, while particular compilations will resolve many nondeterministic choices, and (b) conventional implementations cannot detect all sources of undefined behaviour (that is the main point of UB in the standard, to let implementations assume that source programs do not exhibit UB, together with supporting implementation variation beyond the UB boundary). Sanitisers and other tools can detect some UB cases, but not all, and each tool builds in its own more-or-less ad hoc C semantics.

This is not just an academic problem: disagreements over exactly what is or should be permitted in C have caused considerable tensions, e.g. between OS kernel and compiler developers, as increasingly aggressive optimisations can break code that worked on earlier compiler implementations.

This note continues an exploration of the design space and two candidate semantics for pointers and memory objects in C, taking both ISO and de facto C into account. We earlier [MML+16, CMM+16] identified many design questions. We focus here on the questions concerning pointer provenance, which we revise and extend. We develop two main coherent proposals that reconcile many design concerns; both are broadly consistent with the provenance intuitions of practitioners and ISO DR260, while still reasonably simple. We highlight their pros and cons and various outstanding open questions. These proposals cover many of the interactions between abstract and concrete views in C: casts between pointers and integers, access to the byte representations of values, etc.

A.2 Basic pointer provenance

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] on the right (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).

// provenance_basic_global_yx.c (and an xy variant)

```
1  #include <stdio.h>
2  #include <string.h>
3  int y=2, x=1;
4  int main() {
5    int *p = &x + 1;
6    int *q = &y;
7    printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
8    if (memcmp(&p, &q, sizeof(p)) == 0) {
9      *p = 11;  // does this have undefined behaviour?
10     printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
11   }
12 }
```

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.

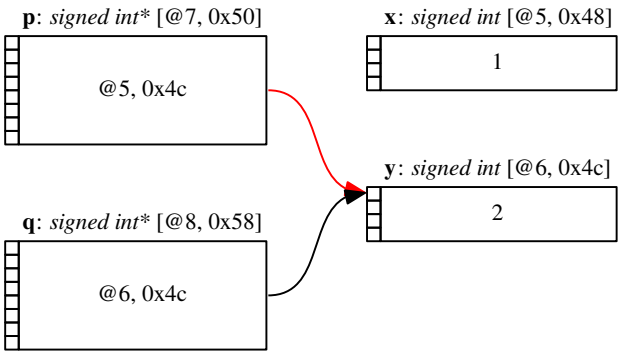
Basic provenance semantics for pointer values For simple cases of the construction and use of pointers, capturing the basic intuition suggested by DR260 CR in a precise semantics is straightforward: we associate a *provenance* with every pointer value, identifying the original storage instance the pointer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

  Regarding such accesses as undefined behaviour is necessary to make optimisation based on provenance alias analysis sound: if the standard did define behaviour for programs that make provenance-violating accesses,

e.g. by adopting a concrete semantics, optimisation based on provenance-aware alias analysis would not be sound.

On the right 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`.



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 implementations such as CHERI C). Provenances therefore do not have program-accessible runtime representations in the abstract machine.

Even for the basic provenance semantics, there are some open design questions, which we now discuss.

Can one construct out-of-bounds (by more than one) pointer values by pointer arithmetic? Consider the example below, where `q` is transiently (more than one-past) out of bounds but brought back into bounds before being used for access. In ISO C, constructing such a pointer value is clearly stated to be undefined behaviour [c1818, 6.5.6p8]. This can be captured using the provenance of the pointer value to determine the relevant bounds. There are cases where such pointer arithmetic would go wrong on some platforms (some now exotic), e.g. where pointer arithmetic subtraction overflows, or if the transient value is not aligned and only aligned values are representable at the particular pointer type, or for hardware that does bounds checking, or where pointer arithmetic might wrap at values less than the obvious word size (e.g. "near" or "huge" 8086 pointers). However, transiently out-of-bounds pointer construction seems to be common in practice. It may be desirable to make it implementation-defined whether such pointer construction is allowed. That would continue to permit implementations in which it would go wrong to forbid it, but give a clear way for other implementations to document that they do not exploit this UB in compiler optimisations that may be surprising to programmers.

```
// cheri_03_ii.c
1 int x[2];
2 int *p = &x[0];
3 int *q = p + 11; // defined behaviour?
4 q = q - 10;
5 *q = 1;
```

Inter-object pointer arithmetic The first example in this section relied on guessing (and then checking) the offset between two storage instances. What if one instead calculates the offset, with pointer subtraction; should that let one move between objects, as below? In ISO C18, the `q-p` is UB (as it is a pointer subtraction between pointers to different objects, which in some abstract-machine executions are not one-past-related). In a variant semantics that allows construction of more-than-one-past pointers (which allows the evaluation of `p + offset`), one would have to to choose whether the `*r=11` access is UB or not. The basic provenance semantics will forbid it, because `r` will retain the provenance of the `x` storage instance, but its address is not in bounds for that. This is probably the most desirable semantics: we have found very few example idioms that intentionally use inter-object pointer arithmetic, and the freedom that forbidding it gives to alias analysis and optimisation seems significant.

```
// pointer_offset_from_ptr_subtraction_global_xy.c
1  #include <stdio.h>
2  #include <string.h>
3  #include <stddef.h>
4  int x=1, y=2;
5  int main() {
6    int *p = &x;
7    int *q = &y;
8    ptrdiff_t offset = q - p;
9    int *r = p + offset;
10   if (memcmp(&r, &q, sizeof(r)) == 0) {
11     *r = 11; // is this free of UB?
12     printf("y=%d *q=%d *r=%d\n",y,*q,*r);
13   }
14 }
```

Pointer equality comparison and provenance A priori, pointer equality comparison (with == or !=) might be expected to just compare their numeric addresses, but we observe GCC 8.1 -O2 sometimes regarding two pointers with the same address but different provenance as nonequal. Unsurprisingly, this happens in some circumstances but not others, e.g. if the test is pulled into a simple separate function, but not if in a separate compilation unit. To be conservative w.r.t. current compiler behaviour, pointer equality in the semantics should give false if the addresses are not equal, but nondeterministically (at each runtime occurrence) either take provenance into account or not if the addresses are equal – this specification looseness accommodating implementation variation. Alternatively, one could require numeric comparisons, which would be a simpler semantics for programmers but force that GCC behaviour to be regarded as a bug. Cerberus supports both options. One might also imagine making it UB to compare pointers that are not strictly within their original storage instance [Kre15], but that would break loops that test against a one-past pointer, or requiring equality to *always* take provenance into account, but that would require implementations to track provenance at runtime.

// provenance_equality_global_xy.c

```c
#include <stdio.h>
#include <string.h>
int x=1, y=2;
int main() {
  int *p = &x + 1;
  int *q = &y;
  printf("Addresses: p=%p q=%p\n",(void*)p,(void*)q);
  _Bool b = (p==q);
  // can this be false even with identical addresses?
  printf("(p==q) = %s\n", b?"true":"false");
  return 0;
}
```

The current ISO C18 standard text is too strong here unless numeric comparison is required: 6.5.9p6 says *"Two pointers compare equal* ***if and only if*** *both are [...] or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space"*, which requires such pointers to compare equal – reasonable pre-DR260 CR, but debatable after it.

Pointer equality should not be confused with alias analysis: we could require == to return true for pointers with the same address but different provenance, while still permitting alias analysis to regard the two as distinct by making accesses via pointers with the wrong provenance UB.

Pointer relational comparison and provenance In ISO C (6.5.8p5), inter-object pointer relational comparison (with < etc.) is undefined behaviour. Just as for inter-object pointer subtraction, there are platforms where this would go wrong, but there are also substantial bodies of code that rely on it, e.g. for lock orderings

It may be desirable to make it implementation-defined whether such pointer construction is allowed.

A.3 Refining the basic provenance model to support pointer construction via casts, representation accesses, etc.

To support low-level systems programming, C provides many other ways to construct and manipulate pointer values:

- casts of pointers to integer types and back, possibly with integer arithmetic, e.g. to force alignment, or to store information in unused bits of pointers;
- copying pointer values with `memcpy`;
- manipulation of the representation bytes of pointers, e.g. via user code that copies them via `char*` or `unsigned char*` accesses;
- type punning between pointer and integer values;
- I/O, using either `fprintf`/`fscanf` and the `%p` format, `fwrite`/`fread` on the pointer representation bytes, or pointer/integer casts and integer I/O;
- copying pointer values with `realloc`;
- constructing pointer values that embody knowledge established from linking, and from constants that represent the addresses of memory-mapped devices.

A satisfactory semantics has to address all these, together with the implications on optimisation. We define and explore several alternatives:

- **PNVI-plain**: a semantics that does not track provenance via integers, but instead, at integer-to-pointer cast points, checks whether the given address points within a live object and, if so, recreates the corresponding provenance. We explain in the next section why this is not as damaging to optimisation as it may sound.

- **PNVI-ae (PNVI exposed-address)**: 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 cast 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)**: 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.
- **PVI**: a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/pointer casts, and making some particular choices for the provenance results of integer and pointer +/- integer operations; or

We write PNVI-* for PNVI-plain, PNVI-ae, and PNVI-ae-udi. The PNVI-plain and PVI semantics were described in the POPL 2019/N2311 paper. PNVI-ae and PNVI-ae-udi have emerged from discussions in the C memory object model study group.

We also mention other variants of PNVI that seem less desirable:

- **PNVI-address-taken**: an earlier variant of PNVI-ae that allowed integer-to-pointer casts to recreate provenance for objects whose address has been taken (irrespective of whether it has been exposed); and
- **PNVI-wildcard**: a variant that gives a "wildcard" provenance to the results of integer-to-pointer casts, delaying checks to access time.

The PVI semantics, originally developed informally in ISO WG14 working papers [MS16a, MGS18], was motivated in part by the GCC documentation [FSF18]:

> *"When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 and C11 6.5.6/8."*

which presumes there is *an* "original" pointer, and by experimental data for `uintptr_t` analogues of the first test of §A.2, which suggested that GCC and ICC sometimes track provenance via integers (see `xy` and `yx` variants). However, discussions at the 2018 GNU Tools Cauldron suggest instead that at least some key developers regard the result of casts from integer types as potentially broadly aliasing, at least in their GIMPLE IR, and such test results as long-standing bugs in the RTL backend.

A.4 Refining the basic provenance model: phenomena and examples

Pointer/integer casts The ISO standard (6.3.2.3) leaves conversions between pointer and integer types almost entirely implementation-defined, except for conversion of integer constant `0` and null pointers, and for the optional `intptr_t` and `uintptr_t` types, for which it guarantees that any *"valid pointer to* `void`*"* can be converted and back, and that *"the result will compare equal to the original pointer"*. As we have seen, in a post-DR260 CR provenance-aware semantics, *"compare equal"* is not enough to guarantee the two are interchangeable, which was clearly the intent of that phrasing. All variants of PNVI-* and PVI support this, by reconstructing or preserving the original provenance respectively.

```
// provenance_roundtrip_via_intptr_t.c
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

Inter-object integer arithmetic Below is a `uintptr_t` analogue of the §A.2 example `pointer_offset_from_ptr_subtraction_global_xy.c`, attempting to move between objects with `uintptr_t` arithmetic. In PNVI-*, this has defined behaviour. For PNVI-plain: the integer values are pure integers, and at the `int*` cast the value of `ux+offset` matches the address of `y` (live and of the right type), so the resulting pointer value takes on the provenance of the `y` storage instance. For PNVI-ae and PNVI-ae-udi, the storage instance for `y` is marked as *exposed* at the cast of `&y` to an integer, and so the above is likewise permitted there.

`// pointer_offset_from_int_subtraction_global_xy.c`

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = uy - ux;
  printf("Addresses: &x=%"PRIuPTR" &y=%"PRIuPTR\
         " offset=%"PRIuPTR" \n",ux,uy,offset);
  int *p = (int *)(ux + offset);
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11; // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

In PVI, this is UB. First, the integer values of `ux` and `uy` have the provenances of the storage instances of `x` and `y` respectively. Then `offset` is a subtraction of two integer values with non-equal single provenances; we define the result of such to have the empty provenance. Adding that empty-provenance result to `ux` preserves the original `x`-storage instance provenance of the latter, as does the cast to `int*`. Then the final `*p=11` access is via a pointer value whose address is not consistent with its provenance. Similarly, PNVI-* allows (contrary to current GCC/ICC O2) a `uintptr_t` analogue of the first test of §A.2, on the left below. PVI forbids this test.

`// provenance_basic_using_uintptr_t_global_xy.c`

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int x=1, y=2;
int main() {
  uintptr_t ux = (uintptr_t)&x;
  uintptr_t uy = (uintptr_t)&y;
  uintptr_t offset = 4;
  ux = ux + offset;
  int *p = (int *)ux; // does this have UB?
  int *q = &y;
  printf("Addresses: &x=%p p=%p &y=%"PRIxPTR\
         "\n",(void*)&x,(void*)p,uy);
  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);
  }
}
```

`// pointer_offset_xor_global.c`

```
#include <stdio.h>
#include <inttypes.h>
int x=1;
int y=2;
int main() {
  int *p = &x;
  int *q = &y;
  uintptr_t i = (uintptr_t) p;
  uintptr_t j = (uintptr_t) q;
  uintptr_t k = i ^ j;
  uintptr_t l = k ^ i;
  int *r = (int *)l;
  // are r and q now equivalent?
  *r = 11;      // does this have defined
      behaviour?
  _Bool b = (r==q);
  printf("x=%i y=%i *r=%i (r==p)=%s\n",x,y,*
      r,
         b?"true":"false");
}
```

Both choices are defensible here: PVI will permit more aggressive alias analysis for pointers computed via integers (though those may be relatively uncommon), while PNVI-* will allow not just this test, which as written is probably not idiomatic desirable C, but also the essentially identical XOR doubly linked list idiom, using only one pointer per node by storing the XOR of two, on the right above. Opinions differ as to whether that idiom matters for modern code.

There are other real-world but rare cases of inter-object arithmetic, e.g. in the implementations of Linux and FreeBSD per-CPU variables, in fixing up pointers after a `realloc`, and in dynamic linking (though arguably some of these are not between C abstract-machine objects). These are rare enough that it seems reasonable to require additional source annotation, or some other mechanism, to prevent compilers implicitly assuming that uses of such pointers as undefined.

Pointer provenance for pointer bit manipulations It is a standard idiom in systems code to use otherwise unused bits of pointers: low-order bits for pointers known to be aligned, and/or high-order bits beyond the addressable range. The example on the right (which assumes `_Alignof(int)>= 4`) does this: casting a pointer to `uintptr_t` and back, using bitwise logical operations on the integer value to store some tag bits.

To allow this, we suggest that the set of unused bits for pointer types of each alignment should be made implementation-defined. In PNVI-* the intermediate value of `q` will have empty provenance, but the value of `r` used for the access will re-acquire the correct provenance at cast time. In PVI we make the binary operations used here, combining an integer value that has some provenance ID with a pure integer, preserve that provenance.

(A separate question is the behaviour if the integer value with tag bits set is converted back to pointer type. In ISO the result is implementation-defined, per 6.3.2.3p{5,6} and 7.20.1.4.)

// provenance_tag_bits_via_uintptr_t_1.c

```c
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p = &x;
  // cast &x to an integer
  uintptr_t i = (uintptr_t) p;
  // set low-order bit
  i = i | 1u;
  // cast back to a pointer
  int *q = (int *) i; // does this have UB?
  // cast to integer and mask out low-order bits
  uintptr_t j = ((uintptr_t)q) & ~((uintptr_t)3u);
  // cast back to a pointer
  int *r = (int *) j;
  // are r and p now equivalent?
  *r = 11;           //  does this have UB?
  _Bool b = (r==p);  //  is this true?
  printf("x=%i *r=%i (r==p)=%s\n",x,*r,b?"t":"f");
}
```

Algebraic properties of integer operations The PVI definitions of the provenance results of integer operations, chosen to make `pointer_offset_from_int_subtraction_global_xy.c` forbidden and `provenance_tag_bits_via_uintptr_t_1.c` allowed, has an unfortunate consequence: it makes those operations no longer associative. Compare the examples below:

// pointer_arith_algebraic_properties_2_global.c

```c
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(((uintptr_t)&(x[0])) +
    (((uintptr_t)&(y[1]))-((uintptr_t)&(y[0]))));
  *p = 11;  // is this free of undefined behaviour?
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

// pointer_arith_algebraic_properties_3_global.c

```c
#include <stdio.h>
#include <inttypes.h>
int y[2], x[2];
int main() {
  int *p=(int*)(
    (((uintptr_t)&(x[0])) + ((uintptr_t)&(y[1])))
    -((uintptr_t)&(y[0])) );
  *p = 11;  // is this free of undefined behaviour?
  //(equivalent to the &x[0]+(&(y[1])-&(y[0])) version?)
  printf("x[1]=%d *p=%d\n",x[1],*p);
  return 0;
}
```

The latter is UB in PVI. It is unclear whether this would be acceptable in practice, either for C programmers or for compiler optimisation. One could conceivably switch to a PVI-multiple variant, allowing provenances to be finite sets of storage instance IDs. That would allow the `pointer_offset_from_int_subtraction_global_xy.c` example above, but perhaps too much else besides. The PNVI-* models do not suffer from this problem.

Copying pointer values with `memcpy()` This clearly has to be allowed, and so, to make the results usable for accessing memory without UB, `memcpy()` and similar functions have to preserve the original provenance. The ISO C18 text does not explicitly address this (in a pre-provenance semantics, before DR260, it did not need to). One could do so by special-casing `memcpy()` and similar functions to preserve provenance, but the following questions suggest less ad hoc approaches, for PNVI-plain or PVI. For PNVI-ae and PNVI-ae-udi, the best approach is not yet clear.

// pointer_copy_memcpy.c

```
#include <stdio.h>
#include <string.h>
int x=1;
int main() {
  int *p = &x;
  int *q;
  memcpy (&q, &p, sizeof p);
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

Copying pointer values bytewise, with user-`memcpy` One of the key aspects of C is that it supports manipulation of object representations, e.g. as in the following naive user implementation of a `memcpy`-like function, which constructs a pointer value from copied bytes. This too should be allowed. PNVI-plain makes it legal: the representation bytes have no provenance, but when reading a pointer value from the copied memory, the read will be from multiple representation-byte writes. We use essentially the same semantics for such reads as for integer-to-pointer casts: checking at read-time that the address is within a live object, and giving the result the corresponding provenance. For PNVI-ae and PNVI-ae-udi, the current proposal is to mark storage instances as exposed whenever representation bytes of pointers to them are read, and use the same semantics for reads of pointer values from representation-byte writes as for integer-to-pointer casts. This is attractively simple, but it does means that integer-to-pointer casts become permitted for all storage instances for which a pointer has been copied via `user_memcpy`, which is arguably too liberal. It may be possible to add additional annotations for code like `user_memcpy` to indicate (to alias analysis) that (a) their target memory should have the same provenance as their source memory, and (b) the storage instances of any copied pointers should not be marked as exposed, despite the reads of their representation bytes. This machinery has not yet been designed.

// pointer_copy_user_dataflow_direct_bytewise.c

```
#include <stdio.h>
#include <string.h>
int  x=1;
void user_memcpy(unsigned char* dest,
                 unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = *src;
    src += 1; dest += 1; n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy((unsigned char*)&q,
              (unsigned char*)&p, sizeof(int *));
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

One might instead think of recording symbolically in the semantics of integer values (e.g. for representation-byte values) whether they are of the form "byte n of pointer value v", or perhaps "byte n of pointer value of type t", and allow reads of pointer values from representation-byte writes only for such. This is more complex and rather ad hoc, arbitrarily restricting the integer computation that can be done on such bytes. If one wanted to allow (e.g.) bitwise operations on such bytes, as in `provenance_tag_bits_via_repr_byte_1.c`, one would essentially have to adopt a PVI model. However, note that to capture the 6.5p6 preservation of effective types by character-type array copy (*"If a value is copied into an object having no declared type 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."*), we might need something like a very restricted version of PVI: some effective-type information attached to integer values of character type, to say "byte n of pointer value of type t", with all integer operations except character-type stores clearing that info.

As Lee observes [private communication], to make it legal for compilers to replace user-memcpy by the library version, one might want the two to have exactly the same semantics. Though strictly speaking that is a question about the compiler intermediate language semantics, not C source semantics.

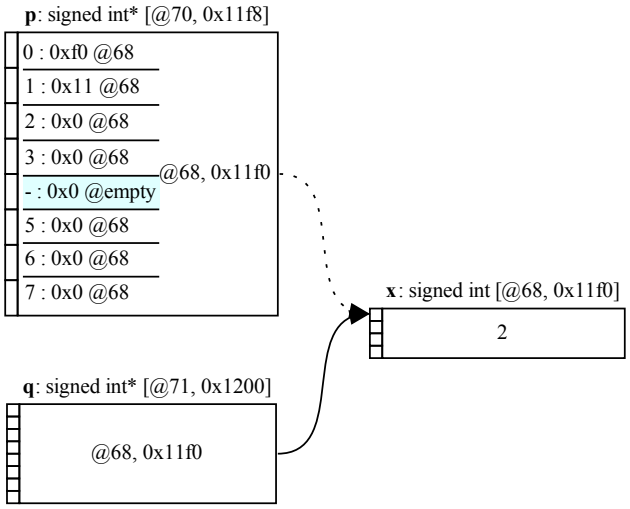
PVI makes user-memcpy legal by regarding each byte (as an integer value) as having the provenance of the original pointer, and the result pointer, being composed of representation bytes of which at least one has that provenance and none have a conflicting provenance, as having the same.

Real `memcpy()` implementations are more complex. The glibc `memcpy()`[gli18] involves copying byte-by-byte, as above, and also word-by-word and, using virtual memory manipulation, page-by-page. Word-by-word copying is not permitted by the ISO standard, as it violates the effective type rules, but we believe C2x should support it for suitably annotated code. Virtual memory manipulation is outside our scope at present.

Reading pointer values from byte writes In all these provenance semantics, pointer values carry their provenance unchanged, both while manipulated in expressions (e.g. with pointer arithmetic) and when stored or loaded as values of pointer type. In the detailed semantics, memory contains abstract bytes rather than general C language values, and so we record provenance in memory by attaching a provenance to each abstract byte. For pointer values stored by single writes, this will usually be identical in each abstract byte of the value.

However, we also have to define the result of reading a pointer value that has been partially or completely written by (integer) representation-byte writes. In PNVI-*, we use the same semantics as for integer-to-pointer casts, reading the numeric address and reconstructing the associated provenance iff a live storage instance covering that address exists (and, for PNVI-ae and PNVI-ae-udi, if that instance has been exposed). To determine whether a pointer value read is from a single pointer value write (and thus should retain its original provenance when read), or from a combination of representation byte writes and perhaps also a pointer value write (and thus should use the integer-to-pointer cast semantics when read), we also record, in each abstract byte, an optional pointer-byte index (e.g. in 0..7 on an implementation with 8-byte pointer values). Pointer value writes will set these to the consecutive sequence $0, 1, .., 7$, while other writes will clear them. For example, the code on the left below sets the fourth byte of `p` to `0`. The memory state on the right, just after the `*q=2`, shows the pointer-byte indices of `p`, one of which has been cleared (shown as -). When the value of `p` is read (e.g. in the `q=p`), the fact that there is not a consecutive sequence $0, 1, .., 7$ means that PNVI-* will apply the integer-to-pointer cast semantics, here successfully recovering the provenance `@68` of the storage instance `x`. Then the write of `q` will itself have a consecutive sequence (its pointer-byte indices are therefore suppressed in the diagram). Any non-pointer write overlapping the footprint of `p`, or any pointer write that overlaps that footprint but does not cover it all, would interrupt the consecutive sequence of indices.

```
1  int x=1;
2  int main() {
3    int *p = &x;
4    if (*((unsigned char*)&p+4)==0)
5      *((unsigned char*)&p+4)=0;
6    int *q = p;
7    *q=2;
8  }
```



In PNVI-plain a representation-byte copy of a pointer value thus is subtly different from a copy done at pointer type: the latter retains the original provenance, while the former, when it is loaded, will take on the provenance of whatever storage instance is live (and covers its address) *at load time*.

The conditional in the example is needed to avoid UB: the semantics does not constrain the allocation address of `x`, so there are executions in which byte 4 is not `0`, in which case the read of `p` would have a wild address and the empty provenance, and the write `*q=2` would flag UB.

Pointer provenance for bytewise pointer representation manipulations To examine the possible semantics for pointer representation bytes more closely, especially for PNVI-ae and PNVI-ae-udi, consider the following. As in provenance_tag_bits_via_uintptr_t_1.c, it manipulates the low-order bits of a pointer value, but now it does so by manipulating one of its representation bytes (as in pointer_copy_user_dataflow_direct_bytewise.c) instead of by casting to `uintptr_t` and back. In PNVI-plain and PVI this will just work, respectively reconstructing the original provenance and tracking it through the (changed and unchanged) integer bytes.

In PNVI-ae and PNVI-ae-udi, we regard the storage instance of `x` as having been exposed by the read of a pointer value (with non-empty provenance in its abstract bytes in memory) at an integer (really, non-pointer) type. Then the last reads of the value of `p`, from a combination of the original `p=&x` write and later integer byte writes, use the same semantics as integer-to-pointer casts, and thus recreate the original provenance.

// provenance_tag_bits_via_repr_byte_1.c

```c
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
int x=1;
int main() {
  int *p=&x, *q=&x;
  // read low-order (little endian) representation byte of p
  unsigned char i = *(unsigned char*)&p;
  // check the bottom two bits of an int* are not used
  assert(_Alignof(int) >= 4);
  assert((i & 3u) == 0u);
  // set the low-order bit of the byte
  i = i | 1u;
  // write the representation byte back
  *(unsigned char*)&p = i;
  // [p might be passed around or copied here]
  // clear the low-order bits again
  *(unsigned char*)&p = (*(unsigned char*)&p) & ~((unsigned char)3
      u);
  // are p and q now equivalent?
  *p = 11;          // does this have defined behaviour?
  _Bool b = (p==q); // is this true?
  printf("x=%i *p=%i (p==q)=%s\n",x,*p,b?"true":"false");
}
```

Copying pointer values via encryption To more clearly delimit what idioms our proposals do and do not allow, consider copying pointers via code that encrypts or compresses a block of multiple pointers together, decrypting or uncompressing later.

In PNVI-plain, it would just work, in the same way as `user_memcpy()`. In PNVI-ae and PNVI-ae-udi, it would work but leave storage instances pointed to by those pointers exposed (irrespective of whether the encryption is done via casts to integer types or by reads of representation bytes), similar to `user_memcpy` and provenance_tag_bits_via_repr_byte_1.c.

One might argue that pointer construction via `intptr_t` and back via any value-dependent identity function should be required to work. That would admit these, but defining that notion of "value-dependent" is exactly what is hard in the concurrency thin-air problem [BMN+15], and we do not believe that it is practical to make compilers respect dependencies in general.

In PVI, this case involves exactly the same combination of distinct-provenance values that (to prohibit inter-object arithmetic, and thereby enable alias analysis) we above regard as having empty-provenance results. As copying pointers in this way is a very rare idiom, one can argue that it is reasonable to require such code to have additional annotations.

Copying pointer values via control flow We also have to ask whether a usable pointer can be constructed via non-dataflow control-flow paths, e.g. if testing equality of an unprovenanced integer value against a valid pointer permits the integer to be used as if it had the same provenance as the pointer. We do not believe that this is relied on in practice. For example, consider exotic versions of `memcpy` that make a control-flow choice on the value of each bit or each byte, reconstructing each with constants in each control-flow branch

// pointer_copy_user_ctrlflow_bytewise_abbrev.c

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <limits.h>
int  x=1;
unsigned char control_flow_copy(unsigned char c) {
  assert(UCHAR_MAX==255);
  switch (c) {
  case 0: return(0);
  case 1: return(1);
  case 2: return(2);
  ...
  case 255: return(255);
  }
}
void user_memcpy2(unsigned char* dest,
                  unsigned char *src, size_t n) {
  while (n > 0)  {
    *dest = control_flow_copy(*src);
    src += 1;
    dest += 1;
    n -= 1;
  }
}
int main() {
  int *p = &x;
  int *q;
  user_memcpy2((unsigned char*)&q, (unsigned char*)&p,
               sizeof(p));
  *q = 11; // does this have undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

// pointer_copy_user_ctrlflow_bitwise.c

```c
#include <stdio.h>
#include <inttypes.h>
#include <limits.h>
int x=1;
int main() {
  int *p = &x;
  uintptr_t i = (uintptr_t)p;
  int uintptr_t_width = sizeof(uintptr_t) * CHAR_BIT;
  uintptr_t bit, j;
  int k;
  j=0;
  for (k=0; k<uintptr_t_width; k++) {
    bit = (i & (((uintptr_t)1) << k)) >> k;
    if (bit == 1)
      j = j | ((uintptr_t)1 << k);
    else
      j = j;
  }
  int *q = (int *)j;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

In PNVI-plain these would both work. In PNVI-ae and PNVI-ae-udi they would also work, as the first exposes the storage instance of the copied pointer value by representation-byte reads and the second by a pointer-to-integer cast. In PVI they would give empty-provenance pointer values and hence UB.

Integer comparison and provenance If integer values have associated provenance, as in PVI, one has to ask whether the result of an integer comparison should also be allowed to be provenance dependent (provenance_equality_uintptr_t_global_xy.c). GCC did do so at one point, but it was regarded as a bug and fixed (from 4.7.1 to 4.8). We propose that the numeric results of all operations on integers should be unaffected by the provenances of their arguments. For PNVI-*, this question is moot, as there integer values have no provenance.

Pointer provenance and union type punning Pointer values can also be constructed in C by type punning, e.g. writing a pointer-type union member, reading it as a `uintptr_t` union member, and then casting back to a pointer type. (The example assumes that the object representation of the pointer and the object representation of the result of the cast to integer are identical. This property is not guaranteed by the C standard, but holds for many implementations.)

The ISO standard says *"the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type"*, but says little about that reinterpretation. We propose that these reinterpretations be required to be implementation-defined, and, in PNVI-plain, that the usual integer-to-pointer cast semantics be used at such reads.

For PNVI-ae and PNVI-ae-udi, the same semantics as for representation-byte reads also permits this case: the storage instance is deemed to be exposed by the read of the provenanced representation bytes by the non-pointer-type read. The integer-to-pointer cast then recreates the provenance of `x`.

For PVI, we propose that it be implementation-defined whether the result preserves the original provenance (e.g. where they are the identity).

`// provenance_union_punning_3_global.c`

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <inttypes.h>
4  int x=1;
5  typedef union { uintptr_t ui; int *up; } un
       ;
6  int main() {
7    un u;
8    int *p = &x;
9    u.up = p;
10   uintptr_t i = u.ui;
11   int *q = (int*)i;
12   *q = 11;  // does this have UB?
13   printf("x=%d *p=%d *q=%d\n",x,*p,*q);
14   return 0;
15 }
```

Pointer provenance via IO Consider now pointer provenance flowing via IO, e.g. writing the address of an object to a string, pipe or file and reading it back in. We have three versions: one using `fprintf`/`fscanf` and the `%p` format, one using `fwrite`/`fread` on the pointer representation bytes, and one converting the pointer to and from `uintptr_t` and using `fprintf`/`fscanf` on that value with the `PRIuPTR`/`SCNuPTR` formats (`provenance_via_io_percentp_global.c`, `provenance_via_io_bytewise_global.c`, and `provenance_via_io_uintptr_t_global.c`) The first gives a syntactic indication of a potentially escaping pointer value, while the others (after preprocessing) do not. Somewhat exotic though they are, these idioms are used in practice: in graphics code for serialisation/deserialisation (using `%p`), in xlib (using `SCNuPTR`), and in debuggers.

In the ISO standard, the text for `fprintf` and `scanf` for `%p` says that this should work: *"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* `%p` *conversion is undefined"* (again construing the pre-DR260 "compare equal" as implying the result should be usable for access), and the text for `uintptr_t` and the presence of `SCNuPTR` in `inttypes.h` weakly implies the same there.

But then what can compiler alias analyses assume about such a pointer read? In PNVI-plain, this is simple: at `scanf`-time, for the `%p` version, or when a pointer is read from memory written by the other two, we can do a runtime check and potential acquisition of provenance exactly like an integer-to-pointer cast.

In PNVI-ae and PNVI-ae-udi, for the `%p` case we mark the associated storage instance as exposed by the output, and use the same semantics as integer-to-pointer casts on the input. The `uintptr_t` case and representation-byte case also mark the storage instance as exposed, in the normal way for these models.

For PVI, there are several options, none of which seem ideal: we could use a PNVI-like semantics, but that would be stylistically inconsistent with the rest of PVI; or (only for the first) we could restrict that to provenances that have been output via `%p`), or we could require new programmer annotation, at output and/or input points, to constrain alias analysis.

Pointers from device memory and linking In practice, concrete memory addresses or relationships between them sometimes are determined and relied on by programmers, in implementation-specific ways. Sometimes these are simply concrete absolute addresses which will never alias C stack, heap, or program memory, e.g. those of particular memory-mapped devices in an embedded system. Others are absolute addresses and relative layout of program code and data, usually involving one or more *linking* steps. For example, platforms may lay out certain regions of memory so as to obey particular relationships, e.g. in a commodity operating system where high addresses are used for kernel mappings, initial stack lives immediately below the arguments passed from the operating system, and so on. The details of linking and of platform memory maps are outside the scope of ISO C, but real C code may embody knowledge of them. Such code might be as simple as casting a platform-specified address, represented as an integer literal, to a pointer. It might be more subtle, such as assuming that one object directly follows another in memory—the programmer having established this property at link time (perhaps by a custom linker script). It is necessary to preserve the legitimacy of such C code, so that compilers may not view such memory accesses as undefined behaviour, even with increasing link-time optimisation.

We leave the design of exactly what escape-hatch mechanisms are needed here as an open problem. For memory-mapped devices, one could simply posit implementation-defined ranges of such memory which are guaranteed not to alias C objects. The more general linkage case is more interesting, but well outside current ISO C. The tracking of provenance through embedded assembly is similar.

Pointers from allocator libraries Our semantics special-cases `malloc` and the related functions, by giving their results fresh provenances. This is stylistically consistent with the ISO text, which also special-cases them, but it would be better for C to support a general-purpose annotation, to let both `stdlib` implementations and other libraries return pointers that are treated as having fresh provenance outside (but not inside) their abstraction boundaries.

Compilers already have related annotations, e.g. GCC's `malloc` attribute *"tells the compiler that a function is malloc-like, i.e., that the pointer P returned by the function cannot alias any other pointer valid when the function returns, and moreover no pointers to valid objects occur in any storage addressed by P"* (https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes).

A.5 Implications of provenance semantics for optimisations

In an ideal world, a memory object semantics for C would be consistent with all existing mainstream code usage and compiler behaviour. In practice, we suspect that (absent a precise standard) these have diverged too much for that, making some compromise required. As we have already seen, the PNVI semantics would make some currently observed GCC and ICC behaviour unsound, though at least some key GCC developers already regard that behaviour as a longstanding unfixed bug, due to the lack of integer/pointer type distinctions in RTL. We now consider some other important cases, by example.

Optimisation based on equality tests Both PNVI-* and PVI let `p==q` hold in some cases where `p` and `q` are not interchangeable. As the authors of [LHJ+18] observe in the LLVM IR context, that may limit optimisations such as GVN (global value numbering) based on pointer equality tests. PVI suffers from the same problem also for integer comparisons, wherever the integers might have been cast from pointers and eventually be cast back. This may be more serious.

Can a function argument alias local variables of the function? In general one would like this to be forbidden, to let optimisation assume its absence. Consider first the example below, where `main()` guesses the address of `f()`'s local variable, passing it in as a pointer, and `f()` checks it before using it for an access. Here we see, for example, GCC -O0 optimising away the `if` and the write `*p=7`, even in executions where the `ADDRESS_PFI_1PG` constant is the same as the `printf`'d address of `j`. We believe that compiler behaviour should be permitted, and hence that this program should be deemed to have UB — or, in other words, that code should not normally be allowed to rely on implementation facts about the allocation addresses of C variables.

```
// pointer_from_integer_1pg.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1PG;
  int *p = (int*)i;
  f(p);
}
```

The PNVI-* semantics deems this to be UB, because at the point of the `(int*)i` cast the `j` storage instance does not yet exist (let alone, for PNVI-ae and PNVI-ae-udi, having been exposed by having one of its addresses taken and cast to integer), so the cast gives a pointer with empty provenance; any execution that goes into the `if` would thus flag UB. The PVI semantics flags UB for the simple reason that `j` is created with the empty provenance, and hence `p` inherits that.

Varying to do the cast to `int*` in `f()` instead of `main()`, passing in an integer `i` instead of a pointer, this becomes defined in PNVI-plain, as `j` exists at the point when the abstract machine does the `(int*)i` cast. But in PNVI-ae and PNVI-ae-udi, the storage instance of `j` is not exposed, so the cast to `int*` gives a pointer with empty provenance and the access via it is UB. This example is also UB in PVI.

```
// pointer_from_integer_1ig.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  if (p==&j)
    *p=7;
  printf("j=%d &j=%p\n",j,(void*)&j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1IG;
  f(j);
}
```

At present we do not see any strong reason why making this defined would not be acceptable — it amounts to requiring compilers to be conservative for the results of integer-to-pointer casts where they cannot see the source of the integer, which we imagine to be a rare case — but this does not match current O2 or O3 compilation for GCC, Clang, or ICC.

Allocation-address nondeterminism Note that both of the previous examples take the address of `j` to guard their `*p=7` accesses. Removing the conditional guards gives the left and middle tests below, that one would surely like to forbid:

`// pointer_from_integer_1p.c`

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(int *p) {
  int j=5;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t i = ADDRESS_PFI_1P;
  int *p = (int*)i;
  f(p);
}
```

`// pointer_from_integer_1i.c`

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

`// pointer_from_integer_1ie.c`

```
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f(uintptr_t i) {
  int j=5;
  uintptr_t k = (uintptr_t)&j;
  int *p = (int*)i;
  *p=7;
  printf("j=%d\n",j);
}
int main() {
  uintptr_t j = ADDRESS_PFI_1I;
  f(j);
}
```

Both are forbidden in PVI for the same reason as before, and the first is forbidden in PNVI-*, again because `j` does not exist at the cast point.

But the second forces us to think about how much allocation-address nondeterminism should be quantified over in the basic definition of undefined behaviour. For evaluation-order and concurrency nondeterminism, one would normally say that if there exists any execution that flags UB, then the program as a whole has UB (for the moment ignoring UB that occurs only on some paths following I/O input, which is another important question that the current ISO text does not address).

This view of UB seems to be unfortunate but inescapable. If one looks just at a single execution, then (at least between input points) we cannot temporally bound the effects of an UB, because compilers can and do re-order code w.r.t. the C abstract machine's sequencing of computation. In other words, UB may be flagged at some specific point in an abstract-machine trace, but its consequences on the observed implementation behaviour might happen much earlier (in practice, perhaps not very much earlier, but we do not have any good way of bounding how much). But then if one execution might have UB, and hence exhibit (in an implementation) arbitrary observable behaviour, then anything the standard might say about any other execution is irrelevant, because it can always be masked by that arbitrary observable behaviour.

Accordingly, our semantics nondeterministically chooses an arbitrary address for each storage instance, subject only to alignment and no-overlap constraints (ultimately one would also need to build in constraints from programmer linking commands). This is equivalent to noting that the ISO standard does not constrain how implementations choose storage instance addresses in any way (subject to alignment and no-overlap), and hence that programmers of standard-conforming code cannot assume anything about those choices. Then in PNVI-plain, the `..._1i.c` example is UB because, even though there is one execution in which the guess is correct, there is another (in fact many others) in which it is not. In those, the cast to `int*` gives a pointer with empty provenance, so the access flags UB — hence the whole program is UB, as desired. In PNVI-ae and PNVI-ae-udi, the `..._1i.c` example is UB for a different reason: the storage instance of `j` is not exposed before the cast `(int*)i`, and so the result of that cast has empty provenance and the access `*p=7` flags UB, in every execution. However, if `j` is exposed, as in the example on the right, these models still make it UB, now for the same reason as PNVI-plain.

Can a function access local variables of its parent? This too should be forbidden in general. The example on the left below is forbidden by PVI, again for the simple reason that `p` has the empty provenance, and by

```c
// pointer_from_integer_2.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  f();
  printf("j=%d\n",j);
}
```

```c
// pointer_from_integer_2g.c
#include <stdio.h>
#include <stdint.h>
#include "charon_address_guesses.h"
void f() {
  uintptr_t i=ADDRESS_PFI_2G;
  int *p = (int*)i;
  *p=7;
}
int main() {
  int j=5;
  if ((uintptr_t)&j == ADDRESS_PFI_2G)
    f();
  printf("j=%d &j=%p\n",j,(void*)&j);
}
```

PNVI-plain by allocation-address nondeterminism, as there exist abstract-machine executions in which the guessed address is wrong. One cannot guard the access within `f()`, as the address of `j` is not available there. Guarding the call to `f()` with `if ((uintptr_t)&j == ADDRESS_PFI_2)` (`pointer_from_integer_2g.c` on the right above) again makes the example well-defined in PNVI-plain, as the address is correct and `j` exists at the `int*` cast point, but notice again that the guard necessarily involves `&j`. This does not match current Clang at O2 or O3, which print `j=5`.

In PNVI-ae and PNVI-ae-udi, `pointer_from_integer_2.c` is forbidden simply because `j` is never exposed (and if it were, it would be forbidden for the same reason as in PNVI-plain). PNVI-ae and PNVI-ae-udi allow `pointer_from_integer_2g.c`, because the `j` storage instance is exposed by the `(uinptr_t)&j` cast.

The PNVI-address-taken and PNVI-wildcard alternatives A different obvious refinement to PNVI would be to restrict integer-to-pointer casts to recover the provenance only of objects that have had their address taken, recording that in the memory state. PNVI-address-exposed is based on PNVI-address-taken but with the tighter condition that the address must also have been cast to integer.

A rather different model is to make the results of integer-to-pointer casts have a "wildcard" provenance, deferring the check that the address matches a live object from cast-time to access-time. This would make `pointer_from_integer_1pg.c` defined, which is surely not desirable.

Perhaps surprisingly, the PNVI-ae and PNVI-ae-udi variants seem not to make much difference to the allowed tests, because the tests one might write tend to already be UB due to allocation-address nondeterminism, or to already take the address of an object to use it in a guard. These variants do have the conceptual advantage of identifying these UBs without requiring examination of multiple executions, but the disadvantage that whether an address has been taken is a fragile syntactic property, e.g. not preserved by dead code elimination.

The problem with lost address-takens and escapes Our PVI proposal allows computations that erase the numeric value (and hence a concrete view of the "semantic dependencies") of a pointer, but retain provenance. This makes examples like that below [Richard Smith, personal communication], in which the code correctly guesses a storage instance address (which has the empty provenance) and adds that to a zero-valued quantity (with the correct provenance), allowed in PVI. We emphasise that we do not think it especially desirable to allow such examples; this is just a consequence of choosing a straightforward provenance-via-integer semantics that allows the bytewise copying and the bitwise manipulation of pointers above. In other words, it is not clear how it could be forbidden simply in PVI.

However, in implementations some algebraic optimisations may be done before alias analysis, and those optimisations might erase the `&x`, replacing it and all the calculation of `i3` by `0x0` (a similar example would have `i3 = i1-i1`). But then alias analysis would be unable to see that `*q` could access `x`, and so report that it could not, and hence enable subsequent optimisations that are unsound w.r.t. PVI for this case. The basic point is that whether a variable has its address taken or escaped in the source language is not preserved by optimisation. A possible solution, which would need some implementation work for

```c
// provenance_lost_escape_1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include "charon_address_guesses.h"
int x=1; // assume allocation ID @1, at ADDR_PLE_1
int main() {
  int *p = &x;
  uintptr_t i1 = (intptr_t)p;            // (@1,ADDR_PLE_1)
  uintptr_t i2 = i1 & 0x00000000FFFFFFFF;//
  uintptr_t i3 = i2 & 0xFFFFFFFF00000000;// (@1,0x0)
  uintptr_t i4 = i3 + ADDR_PLE_1;        // (@1,ADDR_PLE_1)
  int *q = (int *)i4;
  printf("Addresses: p=%p\n",(void*)p);
  if (memcmp(&i1, &i4, sizeof(i1)) == 0) {
    *q = 11;  // does this have defined behaviour?
    printf("x=%d *p=%d *q=%d\n",x,*p,*q);
  }
}
```

implementations that do track provenance through integers, but perhaps acceptably so, would be to require those initial optimisation passes to record the address-takens involved in computations they erase, so that that could be passed in explicitly to alias analysis. In contrast to the difficulties of preserving dependencies to avoid thin-air concurrency, this does not forbid optimisations that remove dependencies; it merely requires them to describe what they do.

In PNVI-plain, the example is also allowed, but for a simpler reason that is not affected by such integer optimisation: the object exists at the `int*` cast. Implementations that take a conservative view of all pointers formed from integers would automatically be sound w.r.t. this. At present ICC is not, at O2 or O3.

PNVI-ae and PNVI-ae-udi are more like PVI here: they allow the example, but only because the address of `p` is both taken and cast to an integer type. If these semantics were used for alias analysis in an intermediate language after such optimisation, this would likewise require the optimsation passes to record which addresses have been taken and cast to integer (or otherwise exposed) in eliminated code, to be explicitly passed in to alias analysis.

Should PNVI allow one-past integer-to-pointer casts? For PNVI*, one has to choose whether an integer that is one-past a live object (and not strictly within another) can be cast to a pointer with valid provenance, or whether this should give an empty-provenance pointer value. Lee observes that the latter may be necessary to make some optimisation sound [personal communication], and we imagine that this is not a common idiom in practice, so for PNVI-plain and PNVI-ae we follow the stricter semantics.

PNVI-ae-udi, however, is designed to permit a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer semantic check that x is properly within the footprint of the storage instance by a check that it is properly within or one-past. That makes the following example allowed in PNVI-ae-udi, while it is forbidden in PNVI-ae and PNVI-plain.

```
// provenance_roundtrip_via_intptr_t_onepast.c
#include <stdio.h>
#include <inttypes.h>
int x=1;
int main() {
  int *p = &x;
  p=p+1;
  intptr_t i = (intptr_t)p;
  int *q = (int *)i;
  q=q-1;
  *q = 11; // is this free of undefined behaviour?
  printf("*p=%d  *q=%d\n",*p,*q);
}
```

The downside of this is that one has to handle pointer-to-integer casts for integer values that are ambiguously both one-past one storage instance and at the start of the next. The PNVI-ae-udi approach to that is to leave the provenance of pointer values resulting from such casts unknown until the first operation (e.g. an access, pointer arithmetic, or pointer relational comparison) that disambiguates them. This makes the following two, each of which uses the result of the cast in one consistent way, well defined:

```
// pointer_from_int_disambiguation_1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    int *r = (int *)i;
    *r=11;  // is this free of UB?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y
        ,*p,*q,*r);
  }
}
```

```
// pointer_from_int_disambiguation_2.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    int *r = (int *)i;
    r=r-1;  // is this free of UB?
    *r=11;  // and this?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n"
        ,x,y,*p,*q,*r);
  }
}
```

while making the following, which tries to use the result of the cast to access both objects, UB.

```c
// pointer_from_int_disambiguation_3.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
int y=2, x=1;
int main() {
  int *p = &x+1;
  int *q = &y;
  uintptr_t i = (uintptr_t)p;
  uintptr_t j = (uintptr_t)q;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    int *r = (int *)i;
    *r=11;
    r=r-1;  // is this free of UB?
    *r=12;  // and this?
    printf("x=%d y=%d *p=%d *q=%d *r=%d\n",x,y,*p
        ,*q,*r);
  }
}
```

In this, the `*r=11` will resolve the provenance of the value in one way, making the `r-1` UB.

A.6 Testing the example behaviour in Cerberus

We have implemented executable versions of the PNVI-plain, PNVI-ae, and PNVI-ae-udi models in Cerberus [MGD+19, MML+16], closely following the detailed semantics of the accompanying note. This makes it possible to interactively or exhaustively explore the behaviour of the examples, confirming that they are allowed or not as intended.

		intended behaviour			**observed behaviour** Cerberus (decreasing allocator)		
test family	**test**	**PNVI-plain**	**PNVI-ae**	**PNVI-ae-udi**	**PNVI-plain**	**PNVI-ae**	**PNVI-ae-udi**
1	provenance_basic_global_xy.c	UB			not triggered		
	provenance_basic_global_yx.c				UB (line 9)		
	provenance_basic_auto_xy.c				not triggered		
	provenance_basic_auto_yx.c				UB (line 9)		
2	**cheri_03_ii.c**	UB			UB (except with *permissive_pointer_arith* switch)		
3	**pointer_offset_from_ptr_subtraction_global_xy.c**	UB (pointer subtraction)			UB (pointer subtraction) Or UB (out-of-bound store with *permissive_pointer_arith* switch)		
	pointer_offset_from_ptr_subtraction_global_yx.c						
	pointer_offset_from_ptr_subtraction_auto_xy.c						
	pointer_offset_from_ptr_subtraction_auto_yx.c						
4	**provenance_equality_global_xy.c**	defined, nondet			not triggered		
	provenance_equality_global_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_auto_xy.c				not triggered		
	provenance_equality_auto_yx.c				defined (ND except with *strict pointer equality* switch)		
	provenance_equality_global_fn_xy.c				not triggered		
	provenance_equality_global_fn_yx.c				defined (ND except with *strict pointer equality* switch)		
5	**provenance_roundtrip_via_intptr_t.c**	defined			defined		
6	**provenance_basic_using_uintptr_t_global_xy.c**	defined			not triggered		
	provenance_basic_using_uintptr_t_global_yx.c				defined		
	provenance_basic_using_uintptr_t_auto_xy.c				not triggered		
	provenance_basic_using_uintptr_t_auto_yx.c				defined		
7	**pointer_offset_from_int_subtraction_global_xy.c**	defined			defined		
	pointer_offset_from_int_subtraction_global_yx.c				defined		
	pointer_offset_from_int_subtraction_auto_xy.c				defined		
	pointer_offset_from_int_subtraction_auto_yx.c				defined		
8	**pointer_offset_xor_global.c**	defined			defined		
	pointer_offset_xor_auto.c				defined		
9	provenance_tag_bits_via_uintptr_t_1.c	defined			defined		
10	**pointer_arith_algebraic_properties_2_global.c**	defined			defined		
11	**pointer_arith_algebraic_properties_3_global.c**	defined			defined		
12	**pointer_copy_memcpy.c**	defined			defined		
13	**pointer_copy_user_dataflow_direct_bytewise.c**	defined			defined		
13	**provenance_tag_bits_via_repr_byte_1.c**	defined			defined		
15	pointer_copy_user_ctrlflow_bytewise.c	defined			defined		
16	**pointer_copy_user_ctrlflow_bitwise.c**	defined			defined		
17	provenance_equality_uintptr_t_global_xy.c	defined			not triggered		
	provenance_equality_uintptr_t_global_yx.c				defined (true)		
	provenance_equality_uintptr_t_auto_xy.c				not triggered		
	provenance_equality_uintptr_t_auto_yx.c				defined (true)		
18	provenance_union_punning_2_global_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_global_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
	provenance_union_punning_2_auto_xy.c	defined	UB (line 16, deref)	UB (line 16, store)	not triggered		
	provenance_union_punning_2_auto_yx.c	defined	UB (line 16, deref)	UB (line 16, store)	defined	UB (line 16, deref)	UB (line 16, store)
19	**provenance_union_punning_3_global.c**	defined			defined		
20	provenance_via_io_percentp_global.c	**filesystem and scanf() are not currently supported by Cerberus**					
	provenance_via_io_bytewise_global.c						
	provenance_via_io_uintptr_t_global.c						
21	**pointer_from_integer_1pg.c**	UB (line 7)			UB in one exec (line 7)		
	pointer_from_integer_1ig.c	defined (j = 7)	UB (line 8)		defined (j = 7)	UB (line 8)	
	pointer_from_integer_1p.c	UB (line 6)			UB (line 6)		
	pointer_from_integer_1i.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_1ie.c	defined (j = 7)			defined (j = 7)		
	pointer_from_integer_2.c	defined (j = 7)	UB (line 7)		defined (j = 7)	UB (line 7)	
	pointer_from_integer_2g.c	defined (j = 7)			defined (j = 7)		
	provenance_lost_escape_1.c	defined			defined		
22	**provenance_roundtrip_via_intptr_t_onepast.c**	UB (line 10)		defined	UB (line 10)		defined
23	**pointer_from_int_disambiguation_1.c**	defined (y = 11)			defined (y = 11)		
	pointer_from_int_disambiguation_1_xy.c				not triggered		
	pointer_from_int_disambiguation_2.c	UB (line 14)		defined	UB (line 14)		defined (x = 11)
	pointer_from_int_disambiguation_2_xy.c				not triggered		
	pointer_from_int_disambiguation_3.c	UB (line 15)		UB (line 15)	UB (line 15)		
	pointer_from_int_disambiguation_3_xy.c				not triggered		

(**bold** = tests mentioned in the document)

green = Cerberus behaviour matches intent
blue = Cerberus behaviour matches intent (witch *permissive_pointer_arith* switch)
grey = Cerberus' allocator doesn't trigger the interesting behaviour

A.7 Testing the example behaviour in mainstream C implementations

We have also run the examples in various existing C implementations, including GCC and Clang at various optimisation levels.

Our test cases are typically written to illustrate a particular semantic question as concisely as possible. Some are "natural" examples, of desirable C code that one might find in the wild, but many are intentionally pathological or are corner cases, to explore just where the defined/undefined-behaviour boundary is; we are not suggesting that all these should be supported.

Making the tests concise to illustrate semantic questions also means that most are not written to trigger interesting compiler behaviour, which might only occur in a larger context that permits some analysis or optimisation pass to take effect. Moreover, following the spirit of C, conventional implementations cannot and do not report all instances of undefined behaviour. Hence, only in some cases is there anything to be learned from the experimental compiler behaviour. For any executable semantics or analysis tool, on the other hand, all the tests should have instructive outcomes.

Some tests rely on address coincidences for the interesting execution; for these we sometimes include multiple variants, tuned to the allocation behaviour in the implementations we consider. Where this has not been done, some of the experimental data is not meaningful.

The detailed data is available at https://www.cl.cam.ac.uk/~pes20/cerberus/supplementary-material-pnvi-star/generated_html_pnvi_star/, and summarised in the table below.

Compilers

		Observed behaviour (compilers), sound w.r.t PNVI-*? (relying on UB or ND?)								
			gcc-8.3			clang-7.0.1			icc-19	
test family	test	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi	PNVI-plain	PNVI-ae	PNVI-ae-udi
1	provenance_basic_global_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_global_yx.c		y (y for O2+)			not triggered			not triggered	
	provenance_basic_auto_xy.c		y (n)			y (n)			y (y for O2+)	
	provenance_basic_auto_yx.c		y (n)			y (n)			y (y for O2+)	
2	**cheri_03_ii.c**		y (n)			y (n)			y (n)	
3	**pointer_offset_from_ptr_subtraction_global_xy.c**		y (n)			y (n)			y (n)	
	pointer_offset_from_ptr_subtraction_global_yx.c								y (n)	
	pointer_offset_from_ptr_subtraction_auto_xy.c								y (y for O2+)	
	pointer_offset_from_ptr_subtraction_auto_yx.c								y (y for O2+)	
4	**provenance_equality_global_xy.c**		y (n)			y (n)			y (n)	
	provenance_equality_global_yx.c		y (y for O2+)							
	provenance_equality_auto_xy.c		y (y for O2+)							
	provenance_equality_auto_yx.c		y (n)							
	provenance_equality_global_fn_xy.c		y (n)							
	provenance_equality_global_fn_yx.c		y (y for O2+)							
5	**provenance_roundtrip_via_intptr_t.c**		y (n)			y (n)			y (n)	
6	**provenance_basic_using_uintptr_t_global_xy.c**		y (n)			y (n)			**n (y)**	
	provenance_basic_using_uintptr_t_global_yx.c		**n (y)**			not triggered			not triggered	
	provenance_basic_using_uintptr_t_auto_xy.c		y (n)			not triggered			**n (y)**	
	provenance_basic_using_uintptr_t_auto_yx.c		y (n)			y (n)			**n (y)**	
7	**pointer_offset_from_int_subtraction_global_xy.c**		y (n)			y (n)			y (n)	
	pointer_offset_from_int_subtraction_global_yx.c									
	pointer_offset_from_int_subtraction_auto_xy.c									
	pointer_offset_from_int_subtraction_auto_yx.c									
8	**pointer_offset_xor_global.c**		y (n)			y (n)			y (n)	
	pointer_offset_xor_auto.c									
9	provenance_tag_bits_via_uintptr_t_1.c		y (n)			y (n)			y (n)	
10	**pointer_arith_algebraic_properties_2_global.c**		y (n)			y (n)			y (n)	
11	**pointer_arith_algebraic_properties_3_global.c**		y (n)			y (n)			y (n)	
12	**pointer_copy_memcpy.c**		y (n)			y (n)			y (n)	
13	**pointer_copy_user_dataflow_direct_bytewise.c**		y (n)			y (n)			y (n)	
13	**provenance_tag_bits_via_repr_byte_1.c**		y (n)			y (n)			y (n)	
15	pointer_copy_user_ctrlflow_bytewise.c		y (n)			y (n)			y (n)	
16	**pointer_copy_user_ctrlflow_bitwise.c**		y (n)			y (n)			y (n)	
17	provenance_equality_uintptr_t_global_xy.c		y (n)			y (n)			y (n)	
	provenance_equality_uintptr_t_global_yx.c									
	provenance_equality_uintptr_t_auto_xy.c									
	provenance_equality_uintptr_t_auto_yx.c									
18	provenance_union_punning_2_global_xy.c		y (n)			y (n)		y (y for O2+)	**n (y)**	
	provenance_union_punning_2_global_yx.c	y (y for O2+)	**n (y)**			not triggered			not triggered	
	provenance_union_punning_2_auto_xy.c		y (n)			y (n)		y (y for O2+)	**n (y)**	
	provenance_union_punning_2_auto_yx.c		y (n)					y (y for O2+)	**n (y)**	
19	**provenance_union_punning_3_global.c**		y (n)			y (n)			y (n)	
20	provenance_via_io_percentp_global.c		NO OPT			NO OPT			NO OPT	
	provenance_via_io_bytewise_global.c									
	provenance_via_io_uintptr_t_global.c									
21	**pointer_from_integer_1pg.c**		y (y for O0+)			y (y for O2+)			y (y for O2+)	
	pointer_from_integer_1ig.c	**n (y)**	y (y for O2+)		**n (y)**	y (y for O2+)			**n (y for O2+)**	
	pointer_from_integer_1p.c	**can't test with charon**								
	pointer_from_integer_1i.c									
	pointer_from_integer_1ie.c									
	pointer_from_integer_2.c									
	pointer_from_integer_2g.c		y (n)			**n (y)**			y (n)	
	provenance_lost_escape_1.c		y (n)			y (n)			**n (y for O2+)**	
22	**provenance_roundtrip_via_intptr_t_onepast.c**		y (n)			y (n)			y (n)	
23	**pointer_from_int_disambiguation_1.c**		**n (y)**			not triggered			not triggered	
	pointer_from_int_disambiguation_1_xy.c		not triggered			y (n)			**n (y for O2+)**	
	pointer_from_int_disambiguation_2.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_2_xy.c		not triggered			y (n)			y (n)	
	pointer_from_int_disambiguation_3.c		y (n)			not triggered			not triggered	
	pointer_from_int_disambiguation_3_xy.c		not triggered			y (n)			y (y for O2+)	

(**bold** = tests mentioned in the document)

Page 2

B Detailed semantics (informative)

This annex gives detailed mathematical semantics for four variants of C provenance semantics:

- **PNVI-plain**: a semantics that does not track provenance via integers, but instead, at integer-to-pointer cast points, checks whether the given address points within a live object and, if so, recreates the corresponding provenance.
- **PNVI-ae (PNVI exposed-address)**: 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 cast 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)**: 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.
- **PVI**: a semantics that tracks provenance via integer computation, associating a provenance with all integer values (not just pointer values), preserving provenance through integer/ pointer casts, and making some particular choices for the provenance results of integer and pointer +/- integer operations; or

We write PNVI-* for PNVI-plain, PNVI-ae, and PNVI-ae-udi. The PNVI-plain and PVI semantics were described in the POPL 2019/N2311 paper [MGD+19]. PNVI-ae and PNVI-ae-udi have emerged from discussions in the C memory object model study group.

Changes for PNVI-ae from PNVI-plain are highlighted. Additional changes for PNVI-ae-udi are highlighted.

This should be read together with the two companion notes, one giving a series of examples (N2363), and another giving detailed diffs to the C standard text (N2362).

The PNVI-ae and PNVI-ae-udi variants of PNVI permit bytewise copy of a pointer to an initially unexposed object, but leaves it marked as exposed. Additional machinery may well be desirable for PNVI-ae and PNVI-ae-udi to give programmers more control of the provenance of the results of byte manipulations, and of what is left marked as exposed. The design of that machinery should ideally be based on the treatment of representation-byte-accessed pointer values by existing compiler alias analyses and optimisations.

B.1 The PNVI-ae-udi, PNVI-ae, PNVI-plain, and PVI semantics

These semantic definitions are manually typeset mathematics simplified from the executable-as-test-oracle Cerberus source (expressed in the pure-functional Lem [MOG+14] definition language). We have removed most subobject details, function pointers, and some options. Neither the typeset models or the Lem source consider linking, or pointers constructed via I/O (e.g. via `%p` or representation-byte I/O).

The memory object semantics can be combined with a semantics for the thread-local semantics of the rest of C (expressed in Cerberus as a translation from C source to the *Core* intermediate language, together with an operational semantics for Core) to give a complete semantics for a large fragment of sequential C.

For simplicity, we assume that pointer representations are the two's complement representation of their addresses (and identical to the two's complement representations of their conversions to sufficiently wide integer types), assume NULL pointers have address (and representation) 0, and allow NULL pointers to be constructed from any empty-provenance integer zero, not just integer constant expressions.

At present the model does not include the ISO semantics that makes all pointers to an object or region invalid at the end of its lifetime, and it permits equality comparison between pointers irrespective of whether the objects of their provenances are live, but it does permit pointer subtraction, relational comparison, array offset, member offset, and casts to integer only for pointers to live objects for which the address is within or one past the object footprint. These are all debatable choices. One could instead check only that the addresses are within or one past the original object footprint (and not check the object is live), or go further towards a concrete-address view of pointer values and not check that either. Sketching out some of the options:

- ZOMBIE-POINTERS-BECOME-INDETERMINATE For the current ISO semantics, at every storage instance lifetime end, the semantics should replace every pointer value with that provenance in the abstract-machine environment with the indeterminate value, and, for every memory footprint containing a pointer value with that provenance (that came from a single pointer value write), synthesise a a write of the indeterminate value to that footprint. With this, the live-object checks for equality, relational comparison, subtraction, array offset, member offset, and casts to integers all become moot.
- ZOMBIE-POINTERS-ALLOW-EQUALITY-ONLY This is what the maths below details.

- ZOMBIE-POINTERS-ALLOW-ALL-IN-BOUNDS-ARITHMETIC For this, we would retain metadata for the bounds of lifetime-ended pointers and check against that for non-load/store operations.
- ZOMBIE-POINTERS-ALLOW-ALL-ARITHMETIC For this, we would remove the lifetime and bounds checks for non-load/store operations.
- ALL-POINTERS-ALLOW-ALL-ARITHMETIC This would make all the non-load/store operations operate just on abstract addresses, ignoring provenance and storage instance metadata.

B.1.1 The memory object model interface

In Cerberus, the memory object model is factored out from Core with a clean interface, roughly as in [MML+16, Fig. 2]. This provides functions for memory operations:

- allocate_object (for objects with automatic or static storage duration, i.e. global and local variables),
- allocate_region (for the results of `malloc`, `calloc`, and `realloc`, i.e. heap-allocated regions),
- kill (for lifetime end of both kinds of allocation),
- load, and
- store,

and for pointer/integer operations: arithmetic, casts, comparisons, offseting pointers by struct-member offsets, etc. The interface involves types `pointer_value` (p), `integer_value` (x), `floating_value`, and `mem_value` (v), which are abstract as far as Core is concerned. Distinguishing pointer and integer values gives more precise internal types.

In PNVI-ae, PNVI, and PVI, a provenance π is either $@i$ where i is a storage-instance ID, or the *empty* provenance `@empty`. In PNVI-ae-udi a provenance can also be a symbolic storage instance ID ι (iota), initially associated to two storage instance IDs and later resolved to one or the other.

A pointer value can either be `null` or a pair (π, a) of a provenance π and address a. In PNVI*, an integer value is simply a mathematical integer (within the appropriate range for the relevant C type), while in PVI, an integer value is a pair (π, n) of a provenance π and a mathematical integer n.

Memory values are the storable entities, either a pointer, integer, floating-point, array, struct, or union value, or `unspec` for unspecified values, each together with their C type.

B.2 The memory object model state

In both PVI and PNVI*, a memory state is a pair (A, M). The A is a partial map from storage-instance IDs to either `killed` or storage-instance metadata $(n, \tau_{\mathrm{opt}}, a, f, k, t)$:

- size n,
- optional C type τ (or `none` for allocated regions),
- base address a,
- permission flag $f \in$ {`readWrite`, `readOnly`},
- kind $k \in$ {`object`, `region`}, and
- for PNVI-ae and PNVI-ae-udi, a taint flag $t \in$ {`unexposed`, `exposed`}.

In PNVI-ae-udi, A also maps all symbolic storage instance IDs ι, to sets of either one or two (non-symbolic) storage instance IDs. One might also need to record a partial equivalence relation over symbolic storage instance IDs, to cope with the pointer subtraction and relational comparison cases where one learns that two provenances are equal but both remain ambiguous, but that is debatable and not spelt out in this document.

The M is a partial map from addresses to abstract bytes, which are triples of a provenance π, either a byte b or `unspec`, and an optional integer pointer-byte index j (or `none`). The last is used in PNVI* to distinguish between loads of pointer values that were written as whole pointer writes vs those that were written byte-wise or in some other way.

B.2.1 Mappings between abstract values and representation abstract-byte sequences

The M models the memory state in terms of low-level abstract bytes, but `store` and `load` take and return the higher-level memory values. We relate the two with functions $\mathrm{repr}(v)$, mapping a memory value to a list of abstract bytes, and $\mathrm{abst}(\tau, bs)$, mapping a list of abstract bytes bs to its interpretation as a memory value with C type τ.

The $\mathrm{repr}(v)$ function is defined by induction over the structure of its memory value parameter and returns a list of $\mathrm{sizeof}(\tau)$ abstract bytes, where τ is the C type of the parameter. The base cases are values with scalar types (integer, floating and pointers) and unspecified values. For an unspecified value of type τ, it returns a list with abstract bytes of the form $(\texttt{@empty}, \texttt{unspec}, \texttt{none})$. Non-null pointer values are represented with lists of abstract bytes that each have the provenance of the pointer value, the appropriate part of the two's complement encoding of the address, and the $0..\mathrm{sizeof}(\tau)-1$ index of each byte. Null pointers are represented with lists of abstract bytes of the form $(\texttt{@empty}, 0, \texttt{none})$. In PVI, integer values are represented similarly to pointer values except that the third component of each abstract byte is `none`. In PNVI*, integer values are represented by lists of abstract bytes, with each of their first components always the empty provenance, and each of their third components again `none`. Floating-point values are similar, in all the models, except that the provenance of the abstract bytes is always empty. For array and struct/union values the function is inductively applied to each subvalue and the resulting byte-lists concatenated. The layout of structs and unions follow an implementation-defined ABI, with padding bytes like those of unspecified values.

The $\mathrm{abst}(\tau, bs)$ function is defined by induction over τ. The base cases are again the scalar types. For these, $\mathrm{sizeof}(\tau)$ abstract bytes are consumed from bs and a scalar memory value is constructed from their second components: if any abstract byte has an `unspec` value, an unspecified value is constructed; otherwise, depending on τ, a pointer, integer or floating-point value is constructed using the two's complement or floating-point encoding. For pointers with address 0, the provenance is empty. For non-0 pointer values and integer values, in PVI the provenance is constructed as follows: if at least one abstract byte has non-empty provenance and all others have either the same or empty provenance, that provenance is taken, otherwise the empty provenance is taken. In PNVI*, when constructing a pointer value, if the third components of the bytes all carry the appropriate index, and all have the same provenance (which will be guaranteed if pointer types all have the same size), the provenance of the result is that provenance. Otherwise, the A part of the memory state is examined to find whether a live storage instance exists with a footprint containing the pointer value that is being constructed. If so, in PNVI-plain, its storage instance ID is used for the provenance of the pointer value, otherwise the empty provenance is used. In PNVI-ae and PNVI-ae-udi, when constructing a pointer value, if A has to be examined then, matching the relevant integer-to-pointer cast semantics below, the storage instance must have been exposed, otherwise the result have the empty provenance. In PNVI-ae-udi, if there are two such live storage instances, with IDs i_1 and i_2, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This can only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second. For array/struct types, $abst()$ recurses on the progressively shrinking list of abstract bytes.

B.2.2 Memory operations

The successful semantics of memory operations is expressed as a transition relation between memory states, with transitions labelled by the operation (including its arguments) and return value:

$$(A, M) \xrightarrow{\text{LABEL}} (A', M')$$

For example, the transitions

$$(A, M) \xrightarrow{\mathrm{load}(\tau,p)=v} (A', M')$$

describe the semantics of a $\mathrm{load}(\tau, p)$ in memory state (A, M), returning value v and with resulting memory state (A', M'). The semantics also defines when each operation flags an out-of-memory (OOM) or undefined behaviour (UB) in a memory state (A, M).

Storage instance creation When a new storage instance is created, either with allocate_region (for the results of `malloc`, `calloc`, and `realloc`, i.e. heap-allocated regions), or with allocate_object (for objects with automatic or static storage duration, i.e. global and local variables), in non-`const` and `const` variants: a fresh storage-instance ID i is chosen; an address a is chosen from $\mathrm{newAlloc}(A, al, n)$, defined to be the set of addresses of blocks of size n aligned by al that do not overlap with 0 or any other allocation in A; and the pointer value $p = (@i, a)$ is returned. In all three cases the storage-instance metadata A is updated with a new record for i, and this is initially marked as `unexposed`. In the allocate_object case the size n of the allocation is the representation size of the C type τ. In the $\mathrm{allocate\_region}(al, \tau, \texttt{readOnly}(v))$ case, the last of the three rules, the memory M is updated to contain the

representation of v at the addresses $a..a + \text{sizeof}(\tau) - 1$.

$$\frac{[\text{LABEL: allocate\_region}(al, n) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad p = (@i, a)}{A, M \to A[i \mapsto (n, \texttt{none}, a, \texttt{readWrite}, \texttt{region}, \texttt{unexposed})], M}$$

$$\frac{[\text{LABEL: allocate\_object}(al, \tau, \texttt{readWrite}) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad n = \text{sizeof}(\tau) \quad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readWrite}, \texttt{object}, \texttt{unexposed})), M}$$

$$\frac{[\text{LABEL: allocate\_object}(al, \tau, \texttt{readOnly}(v)) = p] \quad i \notin \text{dom}(A) \quad a \in \text{newAlloc}(A, al, n) \quad n = \text{sizeof}(\tau) \quad p = (@i, a)}{A, M \to A(i \mapsto (n, \tau, a, \texttt{readOnly}, \texttt{object}, \texttt{unexposed})), M([a..a + n - 1] \mapsto \text{repr}(v))}$$

Storage instance lifetime end When the storage instance of a pointer value $(@i, a)$ is killed, either by a `free()` for a heap-allocated region or at the end of lifetime of an object with automatic storage duration, the storage-instance metadata A of storage instance i is updated to record that i has been killed.

$$\frac{[\text{LABEL: kill}(p, k)] \quad p = (@i, a) \quad k = k' \quad A(i) = (n, \_, a, f, k', \_)}{A, M \to A(i \mapsto \texttt{killed}), M}$$

Load To load a value v of type τ from a pointer value $p = (@i, a)$, there must be a live storage instance for i in A, the footprint of τ at a must be within the footprint of that allocation, and the value v must be the abstract value obtained from the appropriate memory bytes from M.

$$\frac{[\text{LABEL: load}(\tau, p) = v] \quad p = (@i, a) \quad A(i) = (n, \_, a', f, k, \_) \quad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1] \quad v = \text{abst}(\tau, M[a..a + \text{sizeof}(\tau) - 1])}{A, M \to A, M}$$

For PNVI-ae and PNVI-ae-udi, if the recursive-on-τ computation of $\text{abst}(\tau, M[a..a + \text{sizeof}(\tau) - 1])$ involves a call of abst at any non-pointer scalar type for a region of M including an abstract byte with non-empty provenance, and the corresponding storage instance is live, it is marked as exposed. This applies e.g. for reads of pointer values via `char*` pointers, and for union type punning reads at `uintptr_t` of pointer values.

Store To store a value v of type τ to a pointer value $p = (@i, a)$, there must be a live storage instance for i in A, which must be writable, and the footprint of τ at a must be within the footprint of that allocation. The memory M is updated with the representation bytes of the value v.

$$\frac{[\text{LABEL: store}(\tau, p, v)] \quad p = (@i, a) \quad A(i) = (n, \_, a', \texttt{readWrite}, k, \_) \quad [a..a + \text{sizeof}(\tau) - 1] \subseteq [a'..a' + n - 1]}{A, M \to A, M([a..a + \text{sizeof}(\tau) - 1] \mapsto \text{repr}(v))}$$

For PNVI-ae-udi, the kill, load, and store rules above must be adapted. If $p = (\iota, a)$ and $A(\iota) = \{i\}$, the other premises and conclusion of the appropriate above rule apply. If $A(\iota) = \{i_1, i_2\}$ and the premises are satisfied for one of the two, say i_j, the rest of the rule applies except that in the final state A is additionally updated to map ι to $\{i_j\}$.

The memory operations flag out-of-memory (OOM) and undefined behaviour (UB) as follows:

allocate_region(al, n) / allocate_object$(al, \tau,$ `readwrite`$)$ / allocate_object$(al, \tau,$ `readOnly`$(v))$:	
OOM out of memory	if newAlloc$(A, al, n) = \{\}$ or newAlloc$(A, al,$ sizeof$(\tau)) = \{\}$
load(τ, p) / store(τ, p, v) / kill(p):	
UB null pointer	if $p =$ `null`
UB empty provenance	if $p = ($@`empty`$, a)$
UB killed provenance	if $p = (@i, a)$ and $A(i) =$ `killed`
load(τ, p) / store(τ, p, v):	
UB out of bounds	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $[a..a + \text{sizeof}(\tau) - 1] \not\subseteq [a'..a' + n - 1]$
store(τ, p, v):	
UB read-only	if $p = (@i, a)$ and $A(i) = (n, \_, a',$ `readOnly`$, k, \_)$
kill(p):	
UB non-alloc-address	if $p = (@i, a)$, $A(i) = (n, \_, a', f, k, \_)$, and $a \neq a'$

For PNVI-ae-udi, the rules above must be adapted. In the case where $p = (\iota, a)$ and $A(\iota) = \{i\}$, the semantics is exactly as for $p = (i, a)$, while if $A(\iota) = \{i_1, i_2\}$, one has UB only if the conditions above apply to both i_1 and i_2.

B.2.3 Pointer / Integer operations

Pointer subtraction Pointers $p = (@i, a)$ and $p' = (@i', a')$ can be subtracted if they have the same provenance ($i = i'$), there is a live storage instance for i in A, and both a and a' are within or one-past the footprint of that allocation (in ISO C the last will always hold, otherwise UB would have been flagged in earlier pointer arithmetic). Otherwise UB. The result is the numerical difference $a - a'$ divided by sizeof(dearray(τ))), where dearray(τ) returns τ if it is not an array type, and otherwise returns its element type. Note that this disallows subtraction for which one or both arguments are null pointers, which is the ISO semantics but may be a debatable choice.

This rule is stated for PNVI and PNVI-ae, returning pure integer. For PVI, diff_ptrval constructs the same integer but with `@empty` provenance. For PNVI-ae-udi, because subtraction of pointers with different provenance should be UB:

- if both the two pointers have either a provenance $@i$ (resp. $@i'$) or a symbolic storage instance ID ι (resp. ι') mapped by A to a singleton $\{i\}$ (resp. $\{i'\}$), then $i = i'$, otherwise UB.
- if one of the two pointers has a symbolic storage instance ID ι, mapped by A to $\{i_1, i_2\}$, while the other either has a provenance $@i'$ or an ι' mapped to a singleton $\{i'\}$, then i' must be either i_1 or i_2, and ι is resolved to that in the A of the final state. Otherwise UB.
- If both pointers are ambiguous, say mapped to $\{i_1, i_2\}$ and $\{i'_1, i'_2\}$, then if those two sets share exactly one element which satisfies the other rule preconditions, both symbolic storage instance IDs are resolved to that. Otherwise UB.
- If both pointers are ambiguous and those sets share two elements that satisfy the other conditions (which we believe can only happen if the addresses are equal), then subtraction is permitted but the symbolic storage instance IDs are left unresolved. Otherwise UB.

  For example, suppose `p` and `q` have been produced by separate casts from an integer which is ambiguously one-past one allocation and at the start of another. Then after `p-q` or `p<q` we know they must have been the same provenance, but we still don't know which. (Alternatively, we could change the semantics to record an identity relation over symbolic storage instance IDs, and additional modifications to the rules below beyond what is in this draft, but that seems to be unwarranted complexity).

$$\frac{\begin{array}{l}[\text{LABEL: diff\_ptrval}(\tau, p, p') = x] \\ p = (@i, a) \quad p' = (@i', a') \quad i = i' \quad A(i) = (n, \_, \hat{a}, f, k, \_) \\ x = (a - a')/\text{sizeof}(\text{dearray}(\tau)) \quad a \in [\hat{a}..\hat{a} + n] \quad a' \in [\hat{a}..\hat{a} + n]\end{array}}{A, M \to A, M}$$

Pointer relational comparison Pointers $p = (@i, a)$ and $p' = (@i', a')$ can be compared with a relational operator (<, <=, etc.) if they have the same provenance ($i = i'$). The result is the boolean result of the mathematical comparison of a and a'. To make this analogous to pointer subtraction, we also require (though this is debatable) that there is a live storage instance for i in A, and both a and a' are within or one-past the footprint of that allocation. Otherwise UB. Note that this disallows relational comparison against null pointers; a debatable choice. For PNVI-ae-udi, this has to be adapted in much the same way as the pointer subtraction rule above.

$$\frac{\begin{array}{l}[\text{LABEL: rel\_op\_ptrval}(p, p', op) = b] \\ p = (@i, a) \quad p' = (@i', a') \quad i = i' \quad A(i) = (n, \_, \hat{a}, f, k, \_) \\ b = op(a, a') \quad a \in [\hat{a}..\hat{a} + n] \quad a' \in [\hat{a}..\hat{a} + n] \quad op \in \{\leq, <, >, \geq\}\end{array}}{A, M \to A, M}$$

Relational comparison is used in practice between pointers to different objects. A variant which would allow that, which we call ALLOW-INTER-OBJECT-RELATIONAL-OPERATORS TRUE, removes the $i = i'$ test above and (in the ZOMBIE-POINTERS-BECOME-INDETERMINATE and ZOMBIE-POINTERS-ALLOW-EQUALITY-ONLY variants) additionally checks that i' maps to a live object with in-range address.

Pointer equality comparison Pointers p and p' can always be compared with an equality operator (`=`, `!=`). The result is true if they are either both null or both non-null and have the same provenance and address; nondeterministically either $a = a'$ or `false` if they are both non-null and have different provenances; and false otherwise. For PNVI-ae-udi, because equality comparison is permitted (without UB) irrespective of the provenances of the pointers, if the two pointers both have determined single provenances after looking up any symbolic IDs in A, this should give `true`, otherwise the middle (nondeterministic) clause should apply. The final A should not resolve any symbolic IDs.

$$\frac{[\text{LABEL: eq\_op\_ptrval}(p, p') = b] \quad \begin{cases} b = \texttt{true} & \text{if } p = p' \\ b \in \{(a = a'), \texttt{false}\} & \text{if } p = (\pi, a),\ p' = (\pi', a'), \text{ and } \pi \neq \pi' \\ b = \texttt{false} & \text{otherwise} \end{cases}}{A, M \to A, M}$$

Note that the above nondeterminism appears to be necessary to admit the observable behaviour of current compilers, but a simpler provenance-oblivious semantics is arguably desirable:

$$\frac{[\text{LABEL: eq\_op\_ptrval}(p, p') = b] \quad \begin{cases} b = \texttt{true} & \text{if } p = p' = \texttt{null} \\ b = \texttt{true} & \text{if } p = (\pi, a),\ p' = (\pi', a'), \text{ and } a = a' \\ b = \texttt{false} & \text{otherwise} \end{cases}}{A, M \to A, M}$$

We call these two options POINTER-EQUALITY-PROVENANCE-NONDET TRUE and FALSE.

Pointer array offset Given a pointer p at C type τ, the result of offsetting p by integer x (either by array indexing or explicit pointer/integer addition) is as follows, where $x = n$ in PNVI*, or $x = (\pi', n)$ in PVI. For the operation to succeed, p must be some non-null $(@i, a)$. Then there must be a live storage instance for i, and the numeric result of the addition of $a + n * \text{sizeof}(\tau)$ must be within or one-past the footprint of that storage instance. Otherwise the operation flags UB. For PNVI-ae-udi, if p is ambiguous (i.e., $p = (\iota, a)$ and $A(\iota) = \{i_1, i_2\}$ then if x is non-zero this should only be defined behaviour for (at most) one of the two, and then ι should be resolved to that one in the final state. If $x = 0$ it does not resolve the ambiguity.

$$\text{iso\_array\_offset\_ptrval}(A, p, \tau, x) = \begin{cases} (@i, a') & \text{if } p = (@i, a) \text{ and } a' = a + n * \text{sizeof}(\tau) \text{ and } A(i) = (n'', \_, a'', \_, \_, \_) \text{ and } a' \in [a''..a'' + n''] \\ \text{UB: out of bounds} & \text{if all except the last conjunct above hold} \\ \text{UB: empty prov} & \text{if } p = (@\texttt{empty}, a) \\ \text{UB: killed prov} & \text{if } p = (@i, a) \text{ and } A(i) = \texttt{killed} \\ \text{UB: null pointer} & \text{if } p = \texttt{null} \end{cases}$$

Pointer member offset Given a non-null pointer p at C type τ, which points to the start of a struct or union type object (ISO C suggests this has to exist, writing *"The value is that of the named member of the object to which the first expression points"*) with a member m, if p is (π, a), the result of offsetting the pointer to member m has the same provenance π and the suitably offset a.

If p is null, the result is a pointer with empty provenance and the integer offset of m within τ's representation (this is de facto C behaviour, in the sense that the GCC torture tests rely on it; it does not exactly match ISO C).

For the first case, p should point to the start of an object of type τ, with UB otherwise, but without a subobject-aware effective-type semantics, we cannot check that here. Instead, we just check that there is a live storage instance of p's provenance such that the resulting address is within or one-past its a footprint. That makes

this analogous to pointer array offset.

$$\text{member\_offset\_ptrval}(p,\tau,m) = \begin{cases} (\pi, a'), & \text{if } p = (@i, a) \text{ and} \\ & a' = a + \text{offsetof\_ival}(\tau, m) \text{ and} \\ & A(i) = (n'', \_, a'', \_, \_, \_) \text{ and} \\ & a' \in [a''..a'' + n''] \\ (@\texttt{empty}, \text{offsetof\_ival}(\tau, m)), & \text{if } p = \texttt{null}. \end{cases}$$

Casts (PNVI-plain) In PNVI-plain, a cast of a pointer value p to an integer value (at type τ) just converts null pointers to zero and non-null pointer values to the address a of the pointer, if that is representable in τ, otherwise flagging UB. The provenance of the pointer is discarded. At present we require that the object is live and that its address is within bounds.

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} 0, & \text{if } p = \texttt{null}; \\ a, & \text{if } p = (@i, a) \text{ and} \\ & A(i) = (n'', \_, a'', \_, \_, \_) \text{ and} \\ & a \in [a''..a'' + n''] \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

In PNVI-plain, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, it returns a pointer $(@i, x)$ with the provenance of that storage instance. (The "properly within" prevents the one-past ambiguous case.) If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \_) \text{ and } x \in [a..a + n - 1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

Casts (PNVI-ae) In PNVI-ae, the result of a cast of a pointer value p to an integer value is exactly as in PNVI-plain. In addition, for a cast of pointer value $p = (@i, a)$ with provenance $@i$, where $A(i) = (n, \tau_{\text{opt}}, a, f, k, t)$ is the storage instance metadata for i, the memory state (A, M) is updated to $(A(i \mapsto (n, \tau_{\text{opt}}, a, f, k, \texttt{exposed})), M)$ to mark the that storage instance as exposed.

In PNVI-ae, an integer-to-pointer cast of 0 returns the null pointer. For a non-0 integer x, casting to a pointer to τ, if there is a storage instance i in the current memory model state (A, M) for which the address of the pointer would be properly within the footprint of the storage instance, and storage instance i is exposed, it returns a pointer $(@i, x)$ with the provenance of that storage instance. If there is no such storage instance, it returns a pointer with empty provenance.

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a + n - 1] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

Casts (PNVI-ae-udi) In PNVI-ae-udi, a cast of a pointer value p to an integer is just like PNVI-ae.

Unlike PNVI-ae, PNVI-ae-udi permits a cast of a one-past pointer to integer and back to recover the original provenance, replacing the integer-to-pointer check that x is properly within the footprint of the storage instance by a check that it is properly within or one-past:

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = 0 \\ (@i, x), & \text{if } A(i) = (n, \_, a, f, k, \texttt{exposed}) \text{ and } x \in [a..a + n] \\ (@\texttt{empty}, x), & \text{if there is no such } i \end{cases}$$

But then a PNVI-ae-udi cast of an integer value to a pointer can create a pointer with ambiguous provenance (as in the definition of repr) : if it could be within or one-past two live storage instances, with IDs i_1 and i_2, and both storage instances have been marked as exposed, the resulting pointer value is given a fresh symbolic storage instance ID ι, and A is updated to map ι to $\{i_1, i_2\}$. This can only happen if the two storage instances are adjacent and the address is one-past the first and at the start of the second.

Casts (PVI)

$$\text{cast\_ival\_to\_ptrval}(\tau, x) = \begin{cases} \texttt{null}, & \text{if } x = (\texttt{@empty}, 0) \\ (\pi, n), & \text{otherwise, where } x = (\pi, n) \end{cases}$$

$$\text{cast\_ptrval\_to\_ival}(\tau, p) = \begin{cases} (\texttt{@empty}, 0), & \text{if } p = \texttt{null}; \\ (\pi, a), & \text{if } p = (\pi, a) \text{ and } a \in \texttt{value\_range}(\tau) \\ \text{UB}, & \text{otherwise} \end{cases}$$

Integer operations (PVI) In PVI one also has to define the provenance results of all the other operations returning integer values. Below we do so for the basic operations, though this would also be needed for all the integer-returning library functions. Most would give integers with empty provenance. One might or might not also want to require that the objects of those provenances are live.

$$\pi \oplus \pi' = \begin{cases} \pi, & \text{if } \pi = \pi' \text{ or } \pi' = \texttt{@empty}; \\ \pi', & \text{if } \pi = \texttt{@empty}; \\ \texttt{@empty}, & \text{otherwise.} \end{cases}$$

$$\text{op\_ival}(op, (\pi, n), (\pi', m)) = (\pi \oplus \pi', op(n, m)), \text{ where } op \in \{+, *, /, \%, \&, |, \wedge\}$$

$$\text{op\_ival}(-, (\pi, n), (\pi', m)) = \begin{cases} (\texttt{@empty}, n - m), & \text{if } \pi = @i \text{ and } \pi' = @i', \text{ whether } i = i' \text{ or not;} \\ (@i, n - m), & \text{if } \pi = @i \text{ and } \pi' = \texttt{@empty}; \\ (\texttt{@empty}, n - m), & \text{if } \pi = \texttt{@empty}. \end{cases}$$

$$\text{eq\_ival}((\pi, n), (\pi', m)) = (n = m)$$
$$\text{lt\_ival}((\pi, n), (\pi', m)) = (n < m)$$
$$\text{le\_ival}((\pi, n), (\pi', m)) = (n \leq m)$$

B.2.4 No-expose annotation

For PNVI-ae and PNVI-ae-udi, to permit implementations, e.g. of `memcpy`-like functions, to operate on representation bytes but without needlessly leaving all the storage instances that were pointed to in those bytes exposed, we envisaged some "`no-expose`" annotation that users could apply to such code. But now it's not so clear how that could work. We can turn off exposure during execution of annotated code easily enough (though Jens points out that this might not be the right thing for code which is passed a function pointer). But if the user-memcpy code copies bytes via a `char *` pointer, then the resulting abstract types in memory still have empty provenance (because we're not tracking provenance via the intervening integer values), so when a pointer value is read (after the user-memcpy) from the copy, it will still get empty provenance.

B.2.5 Provenance of other operations

In addition to the operations defined above, some operations are desugared/elaborated to simpler expressions by the Cerberus pipeline. Their PVI results have provenance as follows; their PNVI* results are the same except that there integers have no provenance:

- the result of address-of (`&`) has the provenance of the object associated with the lvalue, for non-function-pointers, or empty for function pointers.
- prefix increment and decrement operators follow the corresponding pointer or integer arithmetic rules.
- the conditional operator has the provenance of the second or third operand as appropriate; simple assignment has the provenance of the expression; compound assignment follows the pointer or integer arithmetic rules; the comma operator has the provenance of the second operand.
- integer unary `+`, unary `-`, and `~` operators preserve the original provenance; logical negation `!` has a value with empty provenance.
- `sizeof` and `_Alignof` operators give values with empty provenance.
- bitwise shifts has the provenance of their first operand.
- Jens Gustedt highlights that atomic operations have their own specific provenance properties, not yet discussed here, as do some library functions.

C Modifications to ISO/IEC 9899:2018 (normative)

Implementations that conform to this technical specification, shall behave as if the modifications described in this annex were applied to ISO/IEC 9899:2018. This annex is organized as follows:

- If possible, numbers of clauses refer to the clauses of ISO/IEC 9899:2018.
- Two new sub-clauses are introduced in clause 3 with numbers 3.17 (“provenance”) and 3.20 (“storage instance”). The given context of ISO/IEC 9899:2018 and the numbering indicates the places of insertion.
- Clause 6.2.4 of ISO/IEC 9899:2018 is renamed from ”~~Storage durations of objects~~” to ”Storage durations and object lifetimes”.
- Clause 7.22.3 of ISO/IEC 9899:2018 is renamed from ”~~Memory management functions~~” to ”Storage management functions”.
- Page numbers in the footer correspond to an approximation of the page number in ISO/IEC 9899:2018.
- Page numbers in the top right corner correspond to the page numbering within this document, here.
- Additions to the text are marked as shown.
- Deletions of text are marked as ~~shown~~.

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

an entity that is associated to a pointer value in the abstract machine, which is either empty, or the identity of a storage instance

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**

the inclusion-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.

5 **Note 4 to entry:** A storage instance with a memory address occupies a region of zero or more bytes of contiguous data storage in the execution environment.

6 **Note 5 to entry:** One or more objects may be represented within the same storage instance, such as two subobjects within an object of structure type, two `const`-qualified compound literals with identical object representation, or two string literals where one is the terminal character sequence of the other.

3.21

1 **value**

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

3.21.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.21.2

1 **indeterminate state**

~~either~~ state of an object with an object representation that either represents an unspecified value or is a trap representation

3.21.3

1 **unspecified value**

valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance ~~An unspecified value cannot be a trap representation.~~

3.21.4

1 **trap representation**

an object representation that ~~need~~ does not represent a value of the object type

3.21.5

1 **perform a trap**

interrupt execution of the program such that no further operations are performed

2 **Note 1 to entry:** In this document, when the word "trap" is not immediately followed by "representation", this is the intended usage.[2)]

3 **Note 2 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.22

1 $\lceil x \rceil$

ceiling of x

the least integer greater than or equal to x

2 **EXAMPLE** $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2.

3.23

1 $\lfloor x \rfloor$

floor of x

the greatest integer less than or equal to x

2 **EXAMPLE** $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3.

[2)] For example, "Trapping or stopping (if supported) is disabled . . . " (F.8.2). Note that fetching a trap representation might perform a trap but is not required to (see 6.2.6.1).

of those operations are all *side effects*,[12] which are changes in the state of the execution environment. *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

3 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is *sequenced after* A.) If A is not sequenced before or after B, then A and B are *unsequenced*. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which.[13] The presence of a *sequence point* between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the sequence points is given in Annex C.)

4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **volatile sig_atomic_t** are unspecified, as is the state of the floating-point environment. The ~~value~~ state of any object modified by the handler that is neither a lock-free atomic object nor of type **volatile sig_atomic_t** becomes indeterminate when the handler exits, as does the state of the floating-point environment if it is modified by the handler and not restored to its original state.

6 The least requirements on a conforming implementation are:

- Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
- At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
- The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

This is the *observable behavior* of the program.

7 What constitutes an interactive device is implementation-defined.

8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.

9 **EXAMPLE 1** An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **volatile** would then be redundant.

10 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the **signal** function would require explicit specification of **volatile** storage, as well as other implementation-defined restrictions.

[12] The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see Annex F for details. The floating-point environment library `<fenv.h>` provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

[13] The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

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,~~[33] has a constant address,[34] if any, and retains its last-stored value throughout its lifetime.[35] ~~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~~ Initially the object has an indeterminate state. 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~~ state of the object 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.[36] 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~~ Initially the object has an indeterminate state.

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)

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

[34] 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.

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

[36] 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.

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

— A *pointer type* may be derived from a function type or an object type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced 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,[48)] and *invalid* 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.[49)] 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*.[50)]

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,[51)] 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.[52)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

[48)] 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.

[49)] The provenance of a pointer value and the property that such a pointer value is valid or not are generally not observable. In particular, in the course of the same program execution the same pointer object with the same representation bytes (6.2.6) may sometimes represent valid values but with different provenance (and thus refer to different objects) and sometimes the state of the object may even 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.

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

[51)] See 6.7.3 regarding qualified array and function types.

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

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 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.[52)] 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.[53)]

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[54)] of size m provides access to a byte array of length m. 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 such relative position of addressable storage instances whenever they are created.[55)]

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:[56)]

— Any byte of the object representation of `p` is used in an expression.[57)]

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

[54)] All storage instances that do not originate from an object definition with **register** storage class are addressable by using the pointer value that was returned by their allocation (for allocated storage duration) or by applying the address-of operator & (6.5.3.2) to the object that gave rise to their definition (for other storage durations).

[55)] This means that no relative ordering between storage instances and the objects they represent can be deduced from syntactic 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).

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

[57)] 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.

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

— p is converted to an integer.

— 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.[58] Exposure of a storage instance is irreversible and constitutes a side effect in the abstract state machine.

4 Unless stated otherwise, pointer value p is *synthesized* if it is constructed by one of the following:[59]

— Any byte of the object representation of p is changed

- by an explicit byte operation,
- by type punning with a non-pointer object or with a pointer object that only partially overlaps,
- or by a call to **memcpy** or similar function that does not write the entire pointer representation or where the source object does not have an effective pointer type.

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

— p is converted from an integer value.

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

Special provisions in the respective clauses clarify when such a synthesized pointer is a null, valid, or invalid.

5 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.

6 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[60]

7 Values stored in non-bit-field objects of any other object type ~~consist of~~ are represented using $n \times$ **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 a pointer to a character type or **void** 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 position 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.

[58] 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.

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

[60] 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 $2^{\text{CHAR\_BIT}} - 1$.

8 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.[61] Such a representation is called a trap representation.

9 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.[62] The ~~value~~ representation of a structure or union object is never a trap representation, even though the ~~value~~ representation of a member of the structure or union object may be a trap representation.

10 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.

11 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.[63] 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.

12 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), input/output (7.21).

6.2.6.2 Integer types

1 For unsigned integer types other than **`unsigned char`**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.[64]

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; **`signed char`** shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

[61] 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.

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

[63] 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.

[64] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

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 (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 pointer object is in an indetermined state, in particular if the lvalue conversion does not happen during the lifetime of the provenance that was associated to the stored pointer value, 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

[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.

[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.

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:

- 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 conversion synthesizes a pointer value with that address, provenance and target type.[81]
- The pointer value is invalid.

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~~might be invalid, and might leave a pointer object in an indetermined state after a store. ~~The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.~~

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

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

[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.

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`.

— 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`).

6.4 Lexical elements

Syntax

1 *token:*

keyword
identifier
constant
string-literal
punctuator

preprocessing-token:

header-name
identifier
pp-number
character-constant
string-literal

[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.

values.[95] If the program attempts to modify such an array, the behavior is undefined.

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

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are `'\x12'` and `'3'`, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 **EXAMPLE 2** Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions <stddef.h> (7.19), the **mbstowcs** function (7.22.8.1), Unicode utilities <uchar.h> (7.28).

6.4.6 Punctuators

Syntax

1 *punctuator:* one of

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

Semantics

2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

3 In all aspects of the language, the six tokens[96]

```
<:   :>   <%   %>   %:   %:%:
```

behave, respectively, the same as the six tokens

[95] This allows implementations to share storage instances for string literals and constant compound literals (6.5.2.5) with the same or overlapping representations.

[96] These tokens are sometimes called "digraphs".

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 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 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, 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,

— a qualified version of 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] ~~Allocated objects have~~ 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.

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

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

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

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

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

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

10 There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[113)]

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

12 **EXAMPLE** In the function call

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

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

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

6.5.2.3 Structure and union members

Constraints

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

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

Semantics

3 A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member,[114)] and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

4 A postfix expression followed by the-> operator and an identifier designates a member of a structure or union object. The pointer value shall be valid, not be the end address of its provenance and be

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

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

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`):

```
struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
      if (p1->m  <  0)
```

[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.

list.[118)]

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

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

6 All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.[119)]

7 String literals, and compound literals with const-qualified types, need not designate distinct objects. ~~This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.~~ [120)]

8 **EXAMPLE 1** The file scope definition

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

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

9 **EXAMPLE 2** In contrast, in

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

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

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

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

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

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

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

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

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

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

The first always has static storage duration and has type array of **char**, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

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

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

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

13 **EXAMPLE 6** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage instance is shared.

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

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

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

```
struct s { int i; };

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

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

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

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

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

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

6.5.3 Unary operators

Syntax

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

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

6.5.3.1 Prefix increment and decrement operators

Constraints

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

Semantics

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

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

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

6.5.3.2 Address and indirection operators

Constraints

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

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

Semantics

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

4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to *type*", the result has type "*type*". ~~If an invalid value has been assigned to the pointer , the behavior of the unary * operator is undefined~~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, ~~and~~ the address of an object after the end of its lifetime, or any other invalid 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~~

[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. ~~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.~~

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 $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. If `E1` has a signed type and nonnegative value, and $E1 \times 2^{E2}$ 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

`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;

[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 to an object type and the other is a pointer to a qualified or unqualified version of **void**; or

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

Semantics

3 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.[127] None of the operands shall be an invalid pointer value. 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. Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.~~ 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).

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

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

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

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

6.5.16 Assignment operators

Syntax

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

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;

[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**).

6.7 Declarations

Syntax

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

Constraints

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

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

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

— tags may be redeclared as specified in 6.7.2.3.

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

Semantics

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

— for an object, causes ~~storage~~ a unique storage instance to be reserved for that object;

— for a function, includes the function body;[136)]

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

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

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

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

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

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

6.7.1 Storage-class specifiers

Syntax

1 *storage-class-specifier:*

typedef
extern
static
_Thread_local
auto
register

Constraints

2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that **_Thread_local** may appear with **static** or **extern**.[137)]

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

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

Semantics

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

6 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[138)]

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

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

Forward references: type definitions (6.7.8).

6.7.2 Type specifiers

Syntax

1 *type-specifier:*

void
char
short
int
long
float
double
signed
unsigned
_Bool
_Complex
atomic-type-specifier

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

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

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. 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.

```
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

24 Following the further successful assignments:

```
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) +  6);
```

they then behave as if the declarations were:

```
struct { int n; double d[1]; } *s1, *s2;
```

and:

```
double *dp;
dp = &(s1->d[0]); // valid
*dp = 42;         // valid
dp = &(s2->d[0]); // valid
*dp = 42;         // undefined behavior
```

25 The assignment:

```
*s1 = *s2;
```

only copies the member `n`; if any of the array elements are within the first **`sizeof`** **`(struct`** `s`**`)`** bytes of the structure, they ~~might be copied or simply overwritten with indeterminate values~~are set to an indeterminate state, that may or may not coincide with a copy of the representation of the elements of the source array.

26 **EXAMPLE 3** Because members of anonymous structures and unions are considered to be members of the containing structure or union, **`struct`** `s` in the following example has more than one named member and thus the use of a flexible array member is valid:

```
struct s {
      struct { int i; };
      int a[];
};
```

Forward references: declarators (6.7.6), tags (6.7.2.3).

6.7.2.2 Enumeration specifiers

Syntax

1 *enum-specifier:*

enum *identifier*$_{\text{opt}}$ **{** *enumerator-list* **}**
enum *identifier*$_{\text{opt}}$ **{** *enumerator-list* **,** **}**
enum *identifier*

enumerator-list:

enumerator
enumerator-list **,** *enumerator*

enumerator:

enumeration-constant
enumeration-constant **=** *constant-expression*

Constraints

2 The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **`int`**.

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to *it*, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the declaration

```
struct s2;
```

can be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

Forward references: declarators (6.7.6), type definitions (6.7.8).

6.7.2.4 Atomic type specifiers

Syntax

1 *atomic-type-specifier:*

_Atomic (*type-name*)

Constraints

2 Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.8.3).

3 The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, or a qualified type.

Semantics

4 The properties associated with atomic types are meaningful only for expressions that are lvalues. If the **_Atomic** keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.

6.7.3 Type qualifiers

Syntax

1 *type-qualifier:*

const
restrict
volatile
_Atomic

Constraints

2 Types other than pointer types whose referenced type is an object type shall not be restrict-qualified.

3 The **_Atomic** qualifier shall not be used if the implementation does not support atomic types (see 6.10.8.3).

4 The type modified by the **_Atomic** qualifier shall not be an array type or a function type.

Semantics

5 The properties associated with qualified types are meaningful only for expressions that are lvalues.[149)]

6 If the same qualifier appears more than once in the same specifier-qualifier list or as declaration specifiers, either directly or via one or more **typedef**s, the behavior is the same as if it appeared only once. If other qualifiers appear along with the **_Atomic** qualifier the resulting type is the so-qualified atomic type.

7 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an

[149)] The implementation can place a **const** object that is not **volatile** in a read-only ~~region of~~ storage instance. Moreover, ~~the implementation need not allocate~~ a storage instance for such an object need not be addressable if its address is never used.

operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated.

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

7 **EXAMPLE 1**

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

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

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

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

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

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

```
extern int n;
extern int m;

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

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

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

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

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

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

3 The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type.

4 All the expressions in an initializer for an object that has static or thread storage duration shall be constant expressions or string literals.

5 If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

6 If a designator has the form

[*constant-expression* **]**

then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

7 If a designator has the form

. *identifier*

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

Semantics

8 An initializer specifies the initial value stored in an object.

9 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate ~~value~~ state even after initialization.

10 If an object that has automatic storage duration is not initialized explicitly, its ~~value~~ state is indeterminate. If an object that has static or thread storage duration is not initialized explicitly, then:

— if it has pointer type, it is initialized to a null pointer;

— if it has arithmetic type, it is initialized to (positive or unsigned) zero;

— if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

— if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

11 The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

12 The rest of this subclause deals with initializers for objects that have aggregate or union type.

13 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.

14 An array of character type may be initialized by a character string literal or UTF–8 string literal, optionally enclosed in braces. Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

15 An array with element type compatible with a qualified or unqualified version of **wchar_t**, **char16_t**, or **char32_t** may be initialized by a wide string literal with the corresponding encoding prefix (L, u, or U, respectively), optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

6.8 Statements and blocks

Syntax

1 *statement:*

labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Semantics

2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (~~including storing an indeterminate value in~~ objects without an initializer are in an indeterminate state) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A *full expression* is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 **NOTE** Each of the following is a full expression:

— a full declarator for a variably modified type,

— an initializer that is not part of a compound literal,

— the expression in an expression statement,

— the controlling expression of a selection statement (**`if`** or **`switch`**),

— the controlling expression of a **`while`** or **`do`** statement,

— each of the (optional) expressions of a **`for`** statement,

— the (optional) expression in a **`return`** statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **`return`** statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 *labeled-statement:*

identifier **`:`** *statement*
`case` *constant-expression* **`:`** *statement*
`default` **`:`** *statement*

Constraints

2 A **`case`** or **`default`** label shall appear only in a **`switch`** statement. Further constraints on such labels are discussed under the **`switch`** statement.

3 Label names shall be unique within a function.

7 **EXAMPLE** In the artificial program fragment

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

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

6.8.5 Iteration statements

Syntax

1 *iteration-statement:*

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

Constraints

2 The controlling expression of an iteration statement shall have scalar type.

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

Semantics

4 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.[172)]

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

6 An iteration statement may be assumed by the implementation to terminate if its controlling expression is not a constant expression,[173)] and none of the following operations are performed in its body, controlling expression or (in the case of a **`for`** statement) its *expression-3*:[174)]

— input/output operations

— accessing a volatile object

— synchronization or atomic operations.

6.8.5.1 The `while` statement

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

6.8.5.2 The `do` statement

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

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

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

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

6.9 External definitions

Syntax

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

external-declaration:
function-definition
declaration

Constraints

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

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

Semantics

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

5 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** or **_Alignof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.[178)]

6.9.1 Function definitions

Syntax

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

declaration-list:
declaration
declaration-list declaration

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[179)]

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

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

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

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

6 If the declarator includes an identifier list, each declaration in the declaration list shall have at least 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.~~A parameter identifier cannot be redeclared in the function body except in an enclosed block.~~ ~~The layout of the storage for parameters is unspecified.~~[181]

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; }
```

[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. As any object with automatic storage duration, each parameter gives rise to a unique storage instance representing it. Thus the relative layout of parameters in the address space is unspecified.

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to a non-modifiable storage instance when the corresponding parameter is not const-qualified) or a type (after default argument promotion) not expected by a function with a variable number of arguments, the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

— Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[204] The use of **#undef** to remove any macro definition will also ensure that an actual function is referred to.

— Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.[205]

— Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.[206]

— All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in **#if** preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

4 The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.[207]

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.[208] Implementations may share their own

[204] This means that an implementation is required to provide an actual function for each library function, even if it also provides a macro for that function.

[205] Such macros might not contain the sequence points that the corresponding function calls do.

[206] Because external identifiers and some macro names beginning with an underscore are reserved, implementations can provide special semantics for such names. For example, the identifier `_BUILTIN_abs` could be used to indicate generation of in-line code for the **abs** function. Thus, the appropriate header could specify

```
#define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as **abs** will be a genuine function can write

```
#undef abs
```

whether the implementation's header provides a macro implementation of **abs** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

[207] Thus, a signal handler cannot, in general, call standard library functions.

[208] 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.

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.[209)]

7 Unless otherwise specified, library functions by themselves do not expose storage instances, but library functions that execute application specific callbacks[210)] 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);
```

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.

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

[210)] 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).

7.5 Errors <errno.h>

1 The header <errno.h> defines several macros, all relating to the reporting of error conditions.

2 The macros are

```
EDOM
EILSEQ
ERANGE
```

which expand to integer constant expressions with type **int**, distinct positive values, and which are suitable for use in **#if** preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue[221] that has type **int** and thread local storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **errno**, the behavior is undefined.

3 The value of **errno** in the initial thread is zero at program startup (~~the initial value of **errno** in other threads is an indeterminate value~~initially the object corresponding to **errno** in any other thread is in an indeterminate state), but is never set to zero by any library function.[222] The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in this document.

4 Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter,[223] may also be specified by the implementation.

[221] The macro **errno** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, ∗**errno**()).

[222] Thus, a program that uses **errno** for error checking would set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **errno** on entry and then set it to zero, as long as the original value is restored if **errno**'s value is still zero just before the return.

[223] See "future library directions" (7.31.3).

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[268)] 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[269)] have state, as of the time the **longjmp** function was called, except that the ~~values~~ states 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
}
```

[268)] 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.

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

3 When a signal occurs and `func` points to a function, it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of **SIGILL**, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is **SIGFPE**, **SIGILL**, **SIGSEGV**, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

4 If the signal occurs as the result of calling the **abort** or **raise** function, the signal handler shall not call the **raise** function.

5 If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as **volatile sig_atomic_t**, or the signal handler calls any function in the standard library other than

— the **abort** function,

— the **_Exit** function,

— the **quick_exit** function,

— the functions in <stdatomic.h> (except where explicitly stated otherwise) when the atomic arguments are lock-free,

— the **atomic_is_lock_free** function with any atomic argument, or

— the **signal** function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the ~~value~~ state of **errno** is indeterminate.[272)]

6 At program startup, the equivalent of

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

7 Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the **signal** function.

Returns

8 If the request can be honored, the **signal** function returns the value of `func` for the most recent successful call to **signal** for the specified signal `sig`. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Forward references: the **abort** function (7.22.4.1), the **exit** function (7.22.4.4), the **_Exit** function (7.22.4.5), the **quick_exit** function (7.22.4.7).

7.14.2 Send signal

7.14.2.1 The raise function

Synopsis

1
```
#include <signal.h>
int raise(int sig);
```

[272)] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

7.16 Variable arguments <stdarg.h>

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

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

3 The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros **va_start**, **va_arg**, **va_end**, and **va_copy**. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type **va_list**. The object ap may be passed as an argument to another function; if that function invokes the **va_arg** macro with parameter ap, ~~the value of~~ ap in the calling function is ~~indeterminate~~ in an indeterminate state and shall be passed to the **va_end** macro prior to any further reference to ap.[273)]

7.16.1 Variable argument list access macros

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

7.16.1.1 The va_arg macro

Synopsis

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

Description

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

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

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

Returns

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

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

stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

4 Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to the **freopen** function or the **fwide** function can otherwise alter the orientation of a stream. (A successful call to **freopen** removes any orientation.)[288)]

5 Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

— Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

— For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written ~~are henceforth indeterminate~~may henceforth not consist of valid multibyte characters.

6 Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

7 Each stream has an associated lock that is used to prevent data races when multiple threads of execution access a stream, and to restrict the interleaving of stream operations performed by multiple threads. Only one thread may hold this lock at a time. The lock is reentrant: a single thread may hold the lock multiple times at a given time.

8 All functions that read, write, position, or query the position of a stream lock the stream before accessing it. They release the lock associated with the stream when the access is complete.

Environmental limits

9 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

Forward references: the **freopen** function (7.21.5.4), the **fwide** function (7.29.3.5), **mbstate_t** (7.29.1), the **fgetpos** function (7.21.9.1), the **fsetpos** function (7.21.9.3).

7.21.3 Files

1 A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (character number

[288)] The three predefined streams **stdin**, **stdout**, and **stderr** are unoriented at program startup.

zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.21.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **setbuf** and **setvbuf** functions.

4 A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is ~~indeterminate~~ invalid after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

5 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

6 The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not serve in place of the original.

7 At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

9 Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

— Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).

— A file need not begin nor end in the initial shift state.[289)]

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the **fgetwc** function. Each conversion occurs as if by a call to the **mbrtowc** function, with the conversion state described by the stream's

[289)] Setting the file position indicator to end-of-file, as with **fseek**(file, 0, **SEEK_END**), has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

Returns

5 The **freopen** function returns a null pointer if the open operation fails. Otherwise, **freopen** returns the value of stream.

7.21.5.5 The setbuf function

Synopsis

1
```
#include <stdio.h>
void setbuf(FILE * restrict stream,
      char * restrict buf);
```

Description

2 Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for mode and **BUFSIZ** for size, or (if buf is a null pointer), with the value **_IONBF** for mode.

Returns

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

Forward references: the **setvbuf** function (7.21.5.6).

7.21.5.6 The setvbuf function

Synopsis

1
```
#include <stdio.h>
int setvbuf(FILE * restrict stream,
      char * restrict buf,
      int mode, size_t size);
```

Description

2 The **setvbuf** function may be used only after the stream pointed to by stream has been associated with an open file and before any other operation (other than an unsuccessful call to **setvbuf**) is performed on the stream. The argument mode determines how stream will be buffered, as follows:

_IOFBF causes input/output to be fully buffered;

_IOLBF causes input/output to be line buffered;

_IONBF causes input/output to be unbuffered.

If buf is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function[294] and the argument size specifies the size of the array; otherwise, size may determine the size of a buffer allocated by the **setvbuf** function. The contents of the array is unspecified at any time~~are indeterminate~~.

Returns

3 The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honored.

7.21.6 Formatted input/output functions

1 The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[295]

[294] The buffer has to have a lifetime at least as great as the open stream, so not closing the stream before a buffer that has automatic storage duration is deallocated upon block exit results in undefined behavior.

[295] The **fprintf** functions perform writes to memory for the %n specifier.

of 2, then the precision is sufficient to distinguish[300] 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.[301] 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.[302]

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 %%.

9 If a conversion specification is invalid, the behavior is undefined.[303] 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.

[300] 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.

[301] No special provisions are made for multibyte characters.

[302] Redundant shift sequences can result if multibyte characters have a state-dependent encoding.

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

`c` Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).[307)]

If no `l` length modifier is present, thecorresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. Thecorresponding argument shall be a pointer to the initial element of an array of **`wchar_t`** large enough to accept the resulting sequence of wide characters.No null wide character is added.

`s` Matches a sequence of non-white-space characters.[307)]

If no `l` length modifier is present, thecorresponding 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 input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. Thecorresponding 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.

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[307)]

If no `l` length modifier is present, thecorresponding 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 input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **`mbrtowc`** function, with the conversion state described by an **`mbstate_t`** object initialized to zero before the first multibyte character is converted. Thecorresponding 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 characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) compose the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all 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 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.~~

[307)] No special provisions are made for multibyte characters in the matching rules used by the `c`, `s`, and `[` conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

– 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 synthesized in `*ptr`.[308)]
– Otherwise the state of `*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.[309)]

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 (including new-line characters) is 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
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and to `name` the sequence `thompson\0`.

18 **EXAMPLE 2** The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

[308)] 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.
[309)] See "future library directions" (7.31.11).

Returns

3 The **vfscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vfscanf** 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.

7.21.6.10 The vprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char * restrict format,
      va_list arg);
```

Description

2 The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[310)]

Returns

3 The **vprintf** function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.11 The vscanf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format,
      va_list arg);
```

Description

2 The **vscanf** function is equivalent to **scanf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vscanf** function does not invoke the **va_end** macro.[310)]

Returns

3 The **vscanf** function returns the value of the macro **EOF** if an input failure occurs before the first conversion (if any) has completed. Otherwise, the **vscanf** 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.

7.21.6.12 The vsnprintf function

Synopsis

1
```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * restrict s, size_t n,
      const char * restrict format,
      va_list arg);
```

Description

2 The **vsnprintf** function is equivalent to **snprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsnprintf** function does not invoke the **va_end** macro.[310)] If copying takes place between objects that overlap, the behavior is undefined.

Bibliography

[BMN+15] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015.*, pages 283–307, April 2015.

[c1818] *Programming languages – C*, ISO/IEC 9899:2018 edition, 2018.

[CMM+16] 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.

[Fea04] 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`.

[FSF18] FSF. Using the gnu compiler collection (gcc) / 4.7 arrays and pointers. `https://gcc.gnu.org/onlinedocs/gcc/Arrays-and-pointers-implementation.html`, 2018. Accessed 2018-10-22.

[gli18] glibc. `memcpy`, 2018.

[Kre15] Robbert Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, December 2015.

[KW12] 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`.

[LHJ+18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling high-level optimizations and low-level code with twin memory allocation. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, November 4-9, 2018.* ACM, 2018.

[MGD+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C semantics and pointer provenance. In ***POPL 2019****: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO WG14 N2311, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2311.pdf`.

[MGS18] Kayvan Memarian, Victor Gomes, and Peter Sewell. n2263: Clarifying pointer provenance v4. ISO WG14 `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2263.htm`, May 2018.

[MML+16] 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.

[MOG+14] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, 2014.

[MS16a] Kayvan Memarian and Peter Sewell. N2090: Clarifying pointer provenance (draft defect report or proposal for c2x). ISO WG14 `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2090.htm`, 2016.

[MS16b] Kayvan Memarian and Peter Sewell. What is C in practice? (Cerberus survey v2): Analysis of responses – with comments. ISO SC22 WG14 N2015, `http://www.cl.cam.ac.uk/~pes20/cerberus/analysis-2016-02-05-anon.txt`, March 2016.