N3231

DRAFT TECHNICAL SPECIFICATION

ISO/IEC TS 6010

First edition
Working Draft
2024-03-18

Draft Technical Specification

Information Technology — Programming Languages — C — A provenance-aware memory object model for C—

Reference Number
ISO/IEC TC 6010:Working Draft

COPYRIGHT PROTECTED DOCUMENT

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members 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.

2 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).

3 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).

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

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

5 The committee responsible for this document is ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments, and system software interfaces.

6 ISO/IEC TS 6010 updates ISO/IEC 9899:2018, *Information technology, Programming Language C*, to support a provenance aware memory model for C.

1. Introduction

1 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. 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 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

2 In the first draft of this Technical Specification, the latter two parts have identical text to those earlier N-papers. Later, we integrated the following papers into the specification of this Technical Specification:

- N2861 *Indeterminate Values and Trap Representations. Martin Uecker, Jens Gustedt.* This paper has already been accepted by WG14 for ISO/IEC 9899:2023. It clarifies the previous contradictory terminology for what was then called "indeterminate values", but that described a property of an object representation.
- N2888 *Exact-width Integer Type Interfaces. Jens Gustedt.* This paper has also been accepted by WG14 for ISO/IEC 9899:2023. It clarifies some issues about integer types and is the basis for the integration of the following paper.
- N2889 *Pointers and integer types. Jens Gustedt.* Although this paper has not been accepted for ISO/IEC 9899:2023, WG14 voted in favor to integrate it in this TS. It makes the type **uintptr_t** mandatory and thereby eases the specifications that are proposed here.

3 In addition:

- At `https://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.
- N3005 *A Provenance-aware Memory Object Model for C. Jens Gustedt, Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Martin Uecker* This is a draft version of this TS which provides a visual difference to ISO 9899:2018 as well as a more in-depth discussion on the topic.

4 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.

5 To the best of our knowledge and ability, the proposal reconciles the various demands of existing implementations and the corpus of existing C code.

6 This Technical Specification does not address subobject provenance.

2. Scope

1 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.

Implementations that conform to this document shall behave as if these indicated differences to ISO/IEC 9899:2018 had been integrated into ISO/IEC 9899.

3. Normative References

1 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, Information technology – Vocabulary

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, definitions, and symbols

1 For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2018, and the following apply.

2 **Change to C17:**

After **3.16 parameter**, insert the following, renumbering subsequent items:

3 **3.17**
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

After **3.19 runtime-constraint**, insert the following, renumbering all subsequent items:

4 **3.20**
storage instance

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 **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.

6 **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.

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

8 **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.

9 **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.

10 Replace **3.21.2 indeterminate value** with the following:

11 **3.21.2**
indeterminate representation

object representation that either represents an unspecified value or is a non-value representation

12 Remove **Note 1** from entry **3.21.3 unspecified value**

13 Replace **3.21.4 trap representation** with the following:

14 **3.21.4**
non-value representation

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

5. Environment

1 This section describes changes to ISO/IEC 9899:9899, **5 Environment**, covering the *translation environment* and *execution environment*.

Change to C17:

2 In item **5.1.2.3 Program execution** paragraph 5, second sentence, change the word *value* to *representation*.

3 In item **5.2.4.1 Sizes of integer types <limits.h>**, replace the first paragraph with:

4 The values given below shall be replaced by constant expressions. If the value and promoted type is in the range of the type **`intmax_t`** (for a signed type) or **`uintmax_t`** (for an unsigned type), see 7.20.1.5, the expression shall be suitable for use in **`#if`** preprocessing directives.

6. Language

1 This section specifies changes to the C language specified in C17, ISO/IEC 9899:2018 in order to support a provenance-aware memory model.

6.1 Concepts

6.1.1 Storage duration of objects

Change to C17:

1 Replace section **6.2.4 Storage durations of objects** with the following, taking care to update the relevant footnotes:

6.2.4 Storage duration and object lifetimes

2 The *lifetime* of an object has a start and an end, which both constitute side effects in the abstract 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,[1] has a constant address,[2] if any, and retains its last-stored value throughout its lifetime.[3]

3 The lifetime of an object is 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**.

4 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*, as do storage instances for string literals and some compound literals.[4] The object's lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

5 The storage instance of an object whose identifier is declared with the storage-class specifier **_Thread_local** has *thread storage duration*. 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 instance of the object and distinct associated storage instance 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.

6 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 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.

7 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 representation of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the representation of the object becomes indeterminate each time the declaration is reached.

8 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.[5] If the scope is entered recursively, a new instance of the object and associated storage is created each time. The initial representation of the object is indeterminate.

9 A non-lvalue expression with with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to a temporary object with automatic storage duration and *temporary lifetime*[6] Its lifetime begins when the expression is evaluated and and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime results in undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. Such an object need not have a unique address.

Forward References: array declarations(**6.7.6.2**), compound literals(**6.5.2.5**), declara-

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

[2] 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.

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

[4] Such are for example compound literals that are evaluated in file scope or that are **const** qualified and have only constant expressions as initializers.

[5] 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.

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

tors(**6.7.6**), function calls(**6.5.2.2**), initialization(**6.7.9**), statements(**6.8**), effective type(**6.5**).

10 Replace the fifth dashed bullet point of **6.2.5 Types** paragraph 20 with:

– A *pointer type* may be derived from a function 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 type, if any; its value is *valid* if and only if it has 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. A pointer-to-function is valid if it refers to a valid function definition of the program. Pointers may additionally have a special value *null* that is different from the address of any storage instance and has no provenance (for object pointers),[7)] or from the address of any function of the program (for function pointers). If a pointer value is neither valid, nor null, it is *invalid*. A pointer type derived from the referenced type T is sometimes called a "pointer to T". The construction of a pointer type from a reference type is called "pointer type derivation". A pointer type is a complex object type.[8)] 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 not be used in ways that require (in different usages) more than one of these provenances.

11 Replace the last sentence of paragraph 28 with:

It is implementation-defined whether other groups of pointer types have the same representation or alignment requirements. (54)

6.1.2 Representation of types

6.1.2.1 General

Change to C17:

1 Replace the content of section **6.2.6.1 General** with the following:

2 The representation 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 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).

3 An addressable storage instance[9)] of size m provides access to a byte array of length m. Each byte of the array has an *abstract address*, which is a value of type **`uintptr_t`** that is determined in an implementation-defined manner by pointer-to-integer conversion. 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 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*

[7)] 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.

[8)] 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). Sometimes the object representation may even be indeterminate, namely when the lifetime of the storage instance has ended and no new storage instance uses the same address. Yet, this information is part of the abstract machine and may restrict the set of operations that can be performed on the pointer.

[9)] All storage instances that do not originate from an object definition with **`register`** storage class are addressable.

immediately if they are equal. If the lifetimes of any two distinct addressable storage instances X and Y overlap, 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.[10)]

4 The object representation of a pointer object does not necessarily determine provenance of a pointer value; at different points of the program execution, identical values may refer to distinct storage instances. Unless stated otherwise, a storage instance becomes *exposed* when a pointer value `p` of effective type `T*` with this provenance is used in the following contexts:[11)]

- Any byte of the object representation of `p` is used in an expression.[12)]
- The byte array pointed-to by the first argument of a call to the **`fwrite`** library function intersects with an object representation `p`.
- `p` is converted to an integer.
- `&p` is used as an argument to a `%p` conversion specifier of the **`printf`** family of library functions.[13)]

5 Nevertheless, 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 does not thereby become exposed.[14)] Exposure of a storage instance is irreversible and constitutes a side effect in the abstract machine.

6 Unless stated otherwise, pointer value `p` is *synthesized* if it is constructed by one of the following:[15)]

- 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 or representation where the source object does not have an effective pointer type.
- The object representation of `p` intersects with a byte array pointed-to by the first argument of a call to the **`fread`** library function.
- `p` is converted from an integer value.
- `p` is used as an argument to `%p` conversion specifier of the **`scanf`** family of library functions.

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

[10)] 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).

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

[12)] 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 reading the representation of a pointer value `p` through a **`union`** with a type that is not a pointer type (for example an integer type) or with a pointer type that has a different object representation than the original pointer.

[13)] Passing a pointer value to a `%s` conversion does not expose the storage instance.

[14)] 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 different qualified target types.

[15)] Synthesized pointer values may alias in ways that cannot be predicted by simple data flow analysis.

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

9 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[16)]

10 Value stored in non-bit-field objects of any other object type are represented using `n x CHAR_BIT`, where n is the size of that object type, in bytes. 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 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 range of bytes within the byte array 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 in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the range 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.

11 Certain object representations need not represent a value of the object type. 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.[17)] Such a representation is called a non-value representation.

12 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 corresponds to any padding bytes take unspecified values.[18)] The object representation of a structure or union object is never a non-value representation, even though the byte range corresponding to a member of the structure or union object may be a non-value representation for that member.

13 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.

14 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.[19)] Where the 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 non-value representation shall not be generated.

15 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**), integer types capable of holding object pointers (**7.20.1.4**), input/output (**7.21**)

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

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

[18)] Thus, for example, structure assignment need not copy the padding bits.

[19)] 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.

6.1.2.2 Integer Types

Change to C17:

1 In section **6.2.6.2 Integer Types**:

- paragraph 2, second last sentence, replace the word *trap* with *non-value*.
- paragraph 5, replace the second sentence with:

> A valid object representation of a signed integer type that represents a value where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value.

6.2 Conversion

6.2.1 Other operands

6.2.1.1 Lvalues, arrays and function designators

Change to C17:

1 In section **6.3.2.1 Lvalues, arrays and function designators** replace the text paragraph 2 and 3 with:

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. The behavior is undefined if the lvalue has an incomplete type, if the object representation is a non-value representation for the type,[20] 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 the object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use).

3 Additionally, if the type is a pointer type `T*`, a pointer value and 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 a byte address within (or one-past) the object representation of an exposed storage instance, such that the exposure happened before this lvalue conversion, and the result has that address and provenance.[21]

4 The behavior is undefined if the pointer object has an indeterminate representation, in particular if the lvalue conversion does not happen during the lifetime of the provenance that was associated to the stored pointer value, the represented address is not a valid address (or one-past) for the associated provenance, or the represented address is not correctly aligned for the type.

[20] Character types have no non-value representation, thus reading representation bytes of an addressable live storage instance is always defined.

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

6.2.1.2 Pointers

Change to C17:

1 Replace **6.3.2.3 Pointers** paragraph 5, 6 and 7 with the following:

2 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 value could have been the result of the conversion of a null pointer value. The result is a null pointer.
- The operand value is an abstract address within or one past a live or 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.[22)]
- The pointer value is invalid.

3 Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, might be invalid, and might produce an indeterminate representation when stored into an object.

4 Any pointer type may be converted to an integer type. For a null pointer, the result is chosen from a non-empty set of implementation-defined values.[23)] If the pointer value is valid, its provenance is henceforth exposed. Except as previously specified, the result is the abstract address (which has type **uintptr_t**) converted to the target type. If the target type has a width that is less than the width of **uintptr_t**, the behavior is undefined. If the target type is a signed type and the abstract address is larger than the maximum value of of that type the implementation-defined conversion from **uintptr_t** to the target type as specified in **6.3.1.3** is applied.[24)] If the pointer is null or valid, the integer result converted back to the pointer type shall compare equal to the original pointer.[25)] For two valid pointer values that compare equal, conversion to the same integer type yields identical values.

5 A pointer to an object type may be converted to a pointer to a different object type, retaining its provenance. If the resulting pointer is not correctly aligned[26)] for the reference 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 is the byte address of the object.

6 Insert the following **NOTE** at the end of the section as as paragraph 9:

7 **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 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 are derived from `p`, see **6.2.5**.

The following three cases determine if `p` is used with either 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:

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

[23)] It is recommended that `0` is a member of that set.

[24)] Thus, the result is an implementation-defined signal is raised.

[25)] 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.

[26)] In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to 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.

- 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 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 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 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 strictly greater than `0`;
- `p - n`, where `n` is 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`).

6.3 Lexical elements

6.3.1 String literals

Change to C17:

1 Insert the following footnote into paragraph 7 of **6.4.5 String literals**:

2 It is unspecified whether these arrays are distinct provided their elements have the appropriate values.[27)] If the program attempts to modify such an array, the behavior is undefined.

6.4 Expressions

6.4.1 Postfix operators

6.4.1.1 Structure and union members

1 Apply the following changes to **6.5.2.3 Structure and union members**:

– In paragraph 3, in the footnote attached to the second sentence, replace the word *trap* with *non-value*.

– In paragraph 4, insert the following sentence after the first sentence:

The pointer value shall be valid, not be the end address of its provenance, and be correctly aligned for the structure or union type.

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

6.4.1.2 Compound literals

1 In section **6.5.2.5 Compound literals** make the following changes:

– Change the footnote attached to the end of paragraph 7 to read:

> This allows implementations to storage instances for string literals and constant compound literals with the same or overlapping representations.

– In paragraph 13, **EXAMPLE 6**, insert the word *instance* after the word *storage* so that the end of the sentence reads: if the literals' storage instance is shared.

– Replace paragraph 16 with:

> 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 representation. The behavior of the lvalue conversion of `p` in the assignment to `q` would then be undefined.

6.4.2 Unary operators

6.4.2.1 Unary arithmetic operators

1 In section **6.5.3.3 Unary arithmetic operators** replace the last sentence of paragraph 4 with the following, retaining the existing footnote:

2 The pointer value shall be valid, not be the end address of its provenance, and be correctly aligned for *"type"*.

6.4.3 Additive operators

1 In section **6.5.6 Additive operators** replace paragraphs 8 thru 11 with the following, adding the additional paragraph:

2 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. 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.[28)]

3 When two pointers are subtracted, both shall be valid and point to elements of the same array object, or one past the last element of the array objects;[29)] 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.

4 **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

[28)] 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).

[29)] This implies that they also have the same provenance.

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.

5 **NOTE 2** If the expressions `P` and `Q` points 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 values as `((Q)-(P))+ 1` and as `-((P)-((Q)+1))` , and has the value zero if the expression `P` points one past the the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.

6 **NOTE 3** Another way to approach the 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.

7 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.

6.4.4 Relational operators

1 In section **6.5.8 Relational operators**, replace paragraph 5 with the following:

2 When two pointers are compared, they shall both be valid and have the same provenance. The result depends on the relative ordering of their abstract addresses.

6.4.5 Equality operators

1 In section **6.5.9 Equality operators**, add the following after the first sentence of paragraph 3:

2 None of the operands shall be an invalid pointer value.

3 Replace paragraph 6 with the following:

4 If one operand is null they compare equal if and only if the other operand is null. Otherwise, if both operands are pointers to function type they compare equal if and only if they refer to the same function. Otherwise, they are pointers to objects and compare equal if and only if they have the same abstract address.

6.4.6 Assignment operators

1 In **6.5.16 Assignment operators** insert the following after the first sentence in paragraph 3:

2 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.

6.4.7 Declarations

1 In section **6.7 Declarations** replace the first item of the itemized list under paragram 5 with:

– for an object, causes a unique storage instance to be resolved for that object;

6.4.7.1 Structure and union specifiers

1 In section **6.7.2.1 Structure and union specifiers**, paragraph 18, replace the word *object* with *storage instance*.

2 Replace paragraph 25 with the following:

> 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 are set to an indeterminate representation, that may or may not coincide with a copy of the representation of the elements of the source array.

6.4.8 Declarators

6.4.8.1 Array declarations

1 In section **6.7.6.2 Array declarations**, paragraph 8, insert the word *instance* after the word *storage*.

6.4.9 Initialization

1 In section **6.7.9 Initialization**, replace the following:

- In paragraph 9, replace *indeterminate value* with *indeterminate representation*.
- In paragraph 10, replace *its value is indeterminate* with *its representation is indeterminate*.

6.5 Statements and blocks

1 In section **6.8 Statements and block**, paragraph 3, replace the text between the parenthesis with:

> the representation of objects without an initializer becomes indeterminate

6.5.1 Selection statements

6.5.1.1 The `switch` statement

1 In section **6.8.4.2 The `switch` statement**, paragraph 7, replace *indeterminate value* with *object with an indeterminate representation*.

6.6 External definitions

1 In section **6.9 External definitions**, replace the last sentence of paragraph 4 with:

> As discussed in **5.1.1.1**, a declaration that also causes a storage instance to be reserved for an object or provides the body of a function named by the identifier is a definition.

6.6.1 Function definitions

In section **6.9.1 Function definitions**, replace paragraph 9 with

> Each parameter has automatic storage duration; its identifier is an lvalue.[30)]

7. Library

1 This section specifies changes to the standard library specified in C17, ISO/IEC 9899:2018 in order to support a provenance-aware memory model.

[30)] 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.

7.1 Introduction

7.1.1 Use of library functions

1 In section **7.1.4 Use of library functions**, paragraph 1, first bullet point, replace the text in the parenthesis with:

> 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

2 After paragraph 6, insert the following as paragraph 7 with the relevant footnote:

3 > Unless otherwise specified, library functions by themselves do not expose storage instances, but library functions that execute application-specific callbacks[31)] may expose storage instances through calls into these callbacks.

7.2 Errors <errno.h>

1 In section **7.5 Errors <errno.h>**, paragraph 3, replace the text in the parenthesis with:

2 > initially representation of the object corresponding to **errno** in any other thread is indeterminate

7.3 Nonlocal jumps <setjmp.h>

7.3.1 Restore calling environment

7.3.1.1 The longjmp function

1 In section **7.13.2.1 The longjmp function**, paragraph 3, replace:

- the word *values* with *representation*.
- the second last word, *are*, with *is*.

2 In paragraph 5, replace the word *memory* with *the storage instance*.

7.4 Signal handling <signal.h>

7.4.1 Specify signal handling

7.4.1.1 The signal function

1 In section **7.14.1.1 The signal function**, paragraph 5, bullet point 6, replace the text segment:

> the value of **errno** is indeterminate.

with:

> the object designated by **errno** has an indeterminate representation.

whilst retaining the attached footnote.

7.5 Variable arguments <stdarg.h>

1 In section **7.16 Variable arguments <stdarg.h>**, paragraph 3, replace the word *value* with the word *representation*.

[31)] 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).

7.6 Atomics <stdatomic.h>

7.6.1 Initialization

7.6.1.1 The ATOMIC_VAR_INIT macro

1 In **17.7.2.1 The ATOMIC_VAR_INIT macro**, paragraph 2, replace the section *is initially in an indeterminate state* with *has initially an indeterminate representation.*

7.6.2 Atomic flag type and operations

1 In **17.7.8 Atomic flag type and operations**, paragraph 4, replace the second sentence with:

2 An **atomic_flag** that is not explicitly initialized with **ATOMIC_FLAG_INIT** has initially an indeterminate representation.

7.7 Integer types <stdint.h>

7.7.1 Integer types

7.7.1.1 Exact-width integer types

1 In section **7.20.1.1 Exact-width integer types**, replace paragraph 3 with:

2 If an implementation provides standard or extended integer types with a particular width, no padding bits, and (for the signed types) that have a two's complement representation, it shall define the corresponding typedef names.

7.7.1.2 Integer types capable of holding object pointers

1 Replace **7.20.1.4 Integer types capable of holding object pointers** with the following:

2 The following type designates a signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:

```
intptr_t
```

3 The following type designates the corresponding unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are required.

4 **NOTE 1** The types **intptr_t** and **uintptr_t** are possibly wider than the types **intmax_t** and **uintmax_t** (7.20.1.5). This exception is intended to accommodate implementations that otherwise would not be able to specify **intptr_t** and **uintptr_t** consistent with the rules for these types.

5 **NOTE 2** Although these integer types allow roundtrip conversions of values of type pointer to **void** and therefore guarantee that such conversions do not lose information, arithmetic on these types is not necessarily consistent with arithmetic on pointer to character types, nor can properties of pointer values such as alignment be portably deduced from the bit pattern of the integer result of a conversion.

6 On the other hand, the rules for abstract addresses in **6.2.6.1**, **6.5.8** and **6.5.9** impose that two values of type **uintptr_t** that originate from conversions of two pointers to the same storage instance compare the same for relational and equality operators as the original pointer values. Also, the reconstruction of all the bits of a valid abstract address that has previously been exposed gives rise to an integer value that converts back to the corresponding byte address.

7.7.1.3 Greatest-width integer types

1 Replace **7.20.1.5 Greatest-width integer types** with the following:

2 The following type designates a signed integer type capable of representing any value of any signed integer type with the possible exception of signed extended integer types that are wider than **long long** and that are referred by the type definition for an exact width integer type or for **intptr_t**:

```
intmax_t
```

3 The following type designates the unsigned integer type that corresponds to **intmax_t**:[32)]

```
uintmax_t
```

7.7.2 Macros for integer constants

1 In **7.20.4 Macros for integer constants** replace paragraph 3 with:

2 Each invocation of one of these macros shall expand to an integer constant expression. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument. If the value and promoted type is in the range of the type **intmax_t** (for a signed type) or **uintmax_t** (for an unsigned type), see **7.20.1.5**, the expression is suitable for use in **#if** preprocessing directives.

7.8 Input/output <stdio.h>

7.8.1 Streams

1 In **7.21.2 Streams**, paragraph 5, replace the text in the second bullet point with:

2 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 may henceforth not consist of valid multibyte characters.

7.8.2 Files

1 In **7.21.3 Files** replace paragraph 4 with:

2 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 lifetime of a **FILE** object ends when 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.

7.8.3 File access functions

7.8.3.1 The setvbuf function

1 In section **7.21.5.6 The setvbuf function**, replace the last sentence of paragraph 2 with:

2 The members of the array at any time have unspecified values.

[32)] Thus this type is capable of representing any value of any unsigned integer type with the possible exception of particular extended integer types that are wider than **unsigned long long**.

7.8.3.2 The `fprintf` function

1 In section **7.21.6.1 The fprintf function**, paragraph 8, replace the text for the `p` specifier with:

2 > 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 implementation-defined manner. If the value of the pointer is valid its provenance is henceforth exposed.

7.8.3.3 The `fscanf` function

1 In section **7.21.6.2 The fscanf function**, paragraph 12, replace the text for the `p` specifier with:

2 > Matches the same implementation-defined 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`**.
>
> - If the input sequence could have been printed from a null pointer value, a null pointer value is stored in `*ptr`.
> - 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 representation of a valid pointer address x and the provenance of that storage instance is synthesized in `*ptr`.[33)]
> - Otherwise the representation of `*ptr` becomes indeterminate.

7.8.3.4 The `fgets` function

1 In section **7.21.7.2 The fgets function**, paragraph 3, replace the last sentence with:

2 > If a read error occurs during the operation, the members of the array have unspecified values and a null pointer is returned.

7.8.3.5 The `ungetc` function

1 In section **7.21.7.10 The ungetc function**, paragraph 5, replace the last sentence with:

2 > For a binary stream, its file position indicator is decremented by each successful call to the **`ungetc`** function if its value was zero before a call, it has an indeterminate representation after the call.

7.8.4 Direction input/output functions

7.8.4.1 The `fread` function

1 In section **7.21.8.1 The fread function**, paragraph 2, replace the last 2 sentences with:

2 > If an error occurs, the resulting representation of the file position indicator for the stream is indeterminate. If a partial element is read, its representation is indeterminate.

7.8.4.2 The `fwrite` function

1 In section **7.21.8.2 The fwrite function**, paragraph 2, replace the last sentence with:

2 > If an error occurs, the representation of the file position indicator for the stream is indeterminate.

3 And insert paragraph 3 after paragraph 2 as:

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

[33)] 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, the provenance can be different from the provenance that gave rise to the printf 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**.

7.9 General utilities <stdlib.h>

7.9.1 Storage management functions

1 In section **7.22.3 Memory management functions**, replace the header with **Storage management functions**. Then, replace the first paragraph with:

2 > If the allocation succeeds, the pointer to a storage instance returned by a call to **aligned_alloc**, **calloc**, **malloc**, or **realloc** is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested. It may then be used to access such asn object or array of such objects in the storage instance allocated (until the storage instance is explicitly deallocated). The lifetime of an allocated storage instance extends from the allocation until the deallocation. Each such allocation shall yield a pointer to a storage instance that is disjoint from any other storage instance. The pointer returned points to the start address of the allocated storage instance. If the storage instance cannot be allocated, a null pointer is returned. If the size of the storage instance required is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the address of a storage instance of size zero is returned. For the latter, the returned pointer shall not be used to access an object.

3 > For the purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only storage instances accessible through their arguments and not other static duration storage instances. These functions may, however, visibly modify the storage instance that they allocate or deallocate. Calls to these functions that allocate or deallocate storage instances in a particular region of the address space shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.[34)]

7.9.1.1 The aligned_alloc function

1 In **7.22.3.1 The aligned_alloc function**, replace the first sentence of paragraph 2 with:

2 > The **aligned_alloc** function allocates a storage instance whose alignment is specified by `alignment`, whose size is specified by `size`, and whose representation is indeterminate.

3 Replace paragraph 3 with:

4 > The **aligned_alloc** functions returns either a null pointer or a pointer to the allocated storage instance.

7.9.1.2 The calloc function

1 In **7.22.3.2 The calloc function**, replace paragraph 2, while retaining the existing footnote, with:

2 > The **calloc** function allocates a storage instance for an array of `nmemb` objects, each of whose size is `size`. The storage instance is initialize to all bits zero.

3 Replace paragraph 3 with:

4 > The **calloc** function returns either a null pointer or a pointer to the allocated storage instance.

7.9.1.3 The free function

1 In **7.22.3.3 The free function**, replace paragraph 2 with:

2 > The **free** function causes the storage instance pointed to by `ptr` to be deallocated, that is, made available for further use.[35)] If `ptr` is a null pointer, no action occurs. Otherwise, if

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

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

the argument does not match a pointer earlier returned by a storage management function, or if the storage instance has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

7.9.1.4 The malloc function

1 In **7.22.3.4 The malloc function**, replace paragraphs 2 and 3 with:

Description

2 The **malloc** function allocates a storage instance whose size is specified by `size` and whose representation is indeterminate.

Returns

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

7.9.1.5 The realloc function

1 Replace **7.22.3.5 The realloc function** with:

Synopsis

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

Description

4 The **realloc** function deallocates the old storage instance pointed to by `ptr` and returns a pointer to a new storage instance that has the size specified by `size`. The bytes of the old storage instance up to the lesser of the new and old sizes are copied as if by **memcpy** to the initial bytes of the new storage instance. Any bytes in the new storage instance beyond the size of the old object have unspecified values.

5 If `ptr` is a null pointer, then the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by a storage management function, or if the storage instance has been deallocated by a call to the **free** or **realloc** function, the behavior is unspecified. If `size` is nonzero and no storage instance is allocated, the old storage instance is not deallocated. If `size` is zero and no storage instance is allocated, it is implementation-defined whether the old storage instance is deallocated. If the old storage instance is not deallocated, it shall be unchanged.

Returns

6 The **realloc** function returns a pointer to the new storage instance (which may have the same value as a pointer to the old storage instance), or a null pointer if no new storage instance has been allocated.

7 **NOTE** If a call to **realloc** is successful, the initial part of the new storage instance represents objects with the same value and effective type as the initial part of the old storage instance, if any. Nevertheless, the new storage instance has to be considered to be different from the old one:

- Even if both storage instances have the same address, all pointers to the old storage instance (stored within or outside the storage instance) are invalid because that storage instance ceases to exist.
- Copies of objects in the new storage instance that have hidden state and need explicit initialization (such as variable argument lists, atomic objects, mutexes, or condition variables) may have indeterminate representation.
- Resources reserved for the original objects in the old storage instance that have hidden state and need destruction (such as variable argument lists, mutexes or condition variables) may be squandered.

7.9.2 Multibyte/wide character conversion functions

1 In section **7.22.7 Multibyte/wide character conversion functions**, paragraph 1, replace the last sentence with:

2 Changing the **LC_CTYPE** category causes the internal object describing the conversion state of these functions to have an indeterminate representation.

7.10 String handling <string.h>

7.10.1 Copying functions

1 In **7.24.2 Copying functions**, insert the following as paragraph 1:

2 If the representation of a pointer object is copied by a copying function, either directly or within an aggregate or union object, the pointer copy has the same provenance as the original.

7.10.2 Comparison functions

7.10.2.1 The strxfrm function

1 In **7.24.4.5 The strxfrm function**, paragraph 3 (**Returns**), replace the second sentence with:

2 If the value returned is `n` or more, the contents of the members of the array pointed to by `s1` have an indeterminate representation.

7.11 Threads <threads.h>

7.11.1 Thread-specific storage functions

7.11.1.1 The tss_create function

1 In section **7.26.6.1 The tss_create function**, paragraph 6 (**Returns**), replace the last word *value*, with *representation*.

7.11.1.2 The tss_set function

1 In section **7.26.6.4 The tss_set function**, add the following paragraph after paragraph 3 in the **Description** section:

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

7.12 Date and time <time.h>

7.12.1 Time conversion functions

7.12.1.1 The strftime function

1 In section **7.27.3.5 The strftime function**, paragraph 8, replace the last sentence with:

2 Otherwise, zero is returned and the members of the array have an indeterminate representation.

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

7.13.1 Formatted wide character input/output functions

7.13.1.1 The fwprintf function

1 In section **7.29.2.1 The fwprintf function**, paragraph 8, replace the description of the conversion specifier `p` with:

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

7.13.1.2 The `fwscanf` function

1 In section **7.29.2.2 The fwscanf function**, paragraph 12, replace the description of the conversion specifier `p` with the following, including the footnote:

2 Matches the same implementation-defined set of sequences of wide characters that may be produced by the `%p` conversion of the **`fwprintf`** function. The corresponding argument `ptr` shall be a pointer to a pointer to **`void`**.

- If the input sequence could have been printed from a null pointer value, a null pointer value is stored in `*ptr`.
- 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 representation of a valid pointer with address x and the provenance of that storage instance is synthesized in `*ptr`.[36)]
- Otherwise the representation of `*ptr` becomes indeterminate.

7.13.2 Wide character input/output functions

7.13.2.1 The `fgetws` function

1 In **Section 7.29.3.2 The fgetws function**, paragraph 3, replace the last sentence with:

2 If a read or encoding error occurs during the operation, the array members have an indeterminate representation and a null pointer is returned.

7.13.3 General wide string utilities

7.13.3.1 The `wscxfrm` function

1 In section **7.29.4.4 The wscxfrm function**, paragraph 3, replace the last sentence with:

2 If the value returned is `n` or greater, the array elements pointed to by `s1` have an indeterminate representation.

7.13.4 Wide character time conversion functions

7.13.4.1 The `wscftime` function

1 In section **7.29.5.1 The wcsftime function**, paragraph 3, replace the last sentence with:

2 Otherwise, zero is returned and the members of the array have an indeterminate representation.

[36)] 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**.

Annex A
(informative)
Language syntax summary

1 There are no changes to the **Language syntax summary** annex.

Annex B
(informative)
Library summary

1 There are no changes to the **Library summary** annex.

Annex C
(informative)
Sequence points

1 There are no changes to the **Sequence points** annex.

Annex D
(normative)
Universal character names for identifiers

1 There are no changes to the **Universal character names for identifiers** annex.

Annex E
(informative)
Implementation limits

1 There are no changes to the **Implementation limits** annex.

Annex F
(normative)
IEC 60559 floating-point arithmetic

1 There are no changes to the **IEC 60559 floating-point arithmetic** annex.

Annex G
(normative)
IEC 60559-compatible complex arithmetic

1 There are no changes to the **IEC 60559-compatible complex arithmetic** annex.

Annex H
(informative)
Language independent arithmetic

1 There are no changes to the **Language independent arithmetic** annex.

Annex I
(informative)
Common warnings

1 There are no changes to the **Common warnings** annex.

Annex J
(normative)
Portability issues

J.1 Unspecified behavior

1 In section **J.1 Unspecified behavior**, paragraph 1, insert an item after the item

- Many aspects of the representation of types (**6.2.6**)

that reads:

- The relative order of any two storage instances in the address space (**6.2.6.1**).

2 Remove the following items:

- The layout of storage for function parameters (**6.9.1**)
- The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** functions (**7.22.3**).

3 Modify the item:

- The amount of storage allocated by a successful call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function when 0 bytes was requested.

to read:

- If a call to the **calloc**, **malloc**, **realloc**, or **aligned_alloc** function requesting 0 bytes fails or returns a storage instance of size zero (**7.22.3**).

J.2 Undefined behavior

1 In section **J.2 Undefined Behavior**, paragraph 1, change the following three adjacent items:

- The value of an object with automatic storage duration is used while it is indeterminate (**(6.2.4, 6.7.9, 6.8)**)
- A trap value representation is read by an lvalue expression that does not have character type (**6.2.6.1**).
- A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (**6.2.6.1**).

to read:

- The value of an object with automatic storage duration is used while the object has an indeterminate representation (**(6.2.4, 6.7.9, 6.8)**)
- A non-value value representation is read by an lvalue expression that does not have character type (**6.2.6.1**).
- A non-value representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (**6.2.6.1**).

2 Modify the following adjacent items:

- The value of a pointer that refers to space deallocated by a call to **free** or **realloc** function is used (**7.22.3**).
- The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to **free** or **realloc** (**7.22.3.3**, **7.22.3.5**).

to read:

- The value of a pointer that refers to a storage instance deallocated by a call to **free** or **realloc** function is used (**7.22.3**).
- The pointer argument to the **free** or **realloc** function does not match a pointer earlier returned by a storage management function, or the storage instance has been deallocated by a call to **free** or **realloc** (**7.22.3**, **7.22.5**).

J.3 Implementation-defined behavior

J.3.1 Integers

In section **J.3.5 Integers**, paragraph 1, second item, replace *trap* with *non-value*.

J.3.2 Library functions

1 In section **J.3.12 Library functions**, paragraph 1, replace the following item:

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

with:

- Whether the **calloc**, **malloc**, **realloc**, and **aligned_alloc** functions return a null pointer or a pointer to a storage instance when the size requested is zero (**7.22.3**).

Annex K
(informative)
Bounds checking interfaces

K.1 Library

K.1.1 General utilities stdlib.h

K.1.1.1 Multibyte/wide character conversion functions

1 In section **K.3.6.4 Multibyte/wide character conversion functions**, paragraph 1, replace the last sentence with:

2 Changing the **LC_CTYPE** category causes the internal object describing the conversion state of these functions to have an indeterminate representation.

Annex L
(informative)
Analyzability

L.1 Definitions

L.1.1

1 In section **L.2.2**, replace paragraph 3 with:

2 **Note 2 to entry:** Any values produced or stored might be unspecified values, and the representation of objects that are written to might become indeterminate.