FINAL DRAFT
Technical Specification

ISO/IEC DTS 6010

Programming languages — C — A provenance-aware memory object model for C

ISO/IEC JTC 1/SC 22

Secretariat: **ANSI**

Voting begins on:
2024-12-16

Voting terminates on:
2025-02-10

Reference number
ISO/IEC DTS 6010:2024(en)

Contents

1 Scope **3**

2 Normative references **3**

3 Terms and definitions **3**

4 Environment **4**
- 4.1 Execution environments 4
- 4.2 Sizes of integer types `<limits.h>` 4

5 Language **5**
- 5.1 Concepts 5
  - 5.1.1 Storage durations and object lifetimes 5
  - 5.1.2 Types 5
  - 5.1.3 Representation of types 6
- 5.2 Conversion 8
  - 5.2.1 Lvalues, arrays and function designators 8
  - 5.2.2 Pointers 9
  - 5.2.3 String literals 11
- 5.3 Expressions 11
  - 5.3.1 General 11
  - 5.3.2 Postfix operators 11
  - 5.3.3 Address and indirection operators 11
  - 5.3.4 Additive operators 12
  - 5.3.5 Relational operators 12
  - 5.3.6 Equality operators 13
  - 5.3.7 Assignment operators 13
  - 5.3.8 Declarations 13
  - 5.3.9 Structure and union specifiers 13
  - 5.3.10 Array declarations 13
  - 5.3.11 Initialization 13
- 5.4 Statements and blocks 14
  - 5.4.1 General 14
  - 5.4.2 The `switch` statement 14
- 5.5 External definitions 14
  - 5.5.1 General 14
  - 5.5.2 Function definitions 14

6 Library **14**

6.1 Use of library functions 14
6.2 Errors <errno.h> 15
6.3 The longjmp function 16
6.4 The signal function 16
6.5 Variable arguments <stdarg.h> 16
6.6 Atomics <stdatomic.h> 16
6.6.1 The ATOMIC_VAR_INIT macro 16
6.6.2 Atomic flag type and operations 16
6.7 Integer types <stdint.h> 16
6.7.1 Integer types 16
6.7.2 Macros for integer constants 17
6.8 Input/output <stdio.h> 17
6.8.1 Streams 17
6.8.2 Files 17
6.8.3 File access functions 18
6.8.4 Direction input/output functions 18
6.9 General utilities <stdlib.h> 19
6.9.1 Storage management functions 19
6.9.2 Multibyte/wide character conversion functions 20
6.10 String handling <string.h> 20
6.10.1 Copying functions 20
6.10.2 The strxfrm function 20
6.11 Threads <threads.h> 20
6.11.1 The tss_create function 20
6.11.2 The tss_set function 21
6.12 The strftime function, Date and time <time.h> 21
6.13 Extended multibyte and wide character utilities <wchar.h> 21
6.13.1 The fwprintf function 21
6.13.2 The fwscanf function 21
6.13.3 The fgetws function 21
6.13.4 The wcsxfrm function 22
6.13.5 The wcsftime function 22

Annex A (informative) Portability issues 23

Annex B (informative) Bounds checking interfaces 24

Annex C (informative) Analyzability 25

Index 26

Foreword

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.

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 or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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 www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

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

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

The resolution of DR260 confirmed the concept of provenance of pointers, introduced as means to track and distinguish pointer values that represent storage instances with the 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 document 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.

In addition to this document, https://cerberus.cl.cam.ac.uk/cerberus provides an executable version of the semantics, with a web interface that allows one to explore and visualise the behaviour of small test programs.

This document does not address subobject provenance.

1 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 builds upon ISO/IEC 9899:2018 by constraining and clarifying the Memory Object Model.

2 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 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.

3 Terms and definitions

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

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

– ISO Online browsing platform: available at https://www.iso.org/obp/ui

– IEC Electropedia: available at https://www.electropedia.org/

3.1

pointer provenance

provenance

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

3.2

storage instance

storage instance

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

Note 1 to entry: Storage instances are created and destroyed when specific language constructs (ISO/IEC 9899:2018, 6.2.4) are met during program execution, including program startup, or when specific library functions (ISO/IEC 9899:2018, 7.22.3) are called.

Note 2 to entry: It is possible that a storage instance does not have a memory address and is not accessible from all threads of execution.

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

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.

Note 5 to entry: One or more objects can 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.

3.3
indeterminate representation

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

Note 1 to entry: This item is adapted from the term "indeterminate value" (ISO/IEC 9899:2018, 3.19.2)

3.4
unspecified value

valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance

[SOURCE:ISO/IEC 9899:2018, 3.19.3,modified - Note 1 to entry has been removed.]

3.5
non-value representation

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

Note 1 to entry: This term was adapted from the term "trap representation" (ISO/IEC 9899:2018, 3.19.4)

4 Environment

4.1 Execution environments

The requirements in ISO/IEC 9899:2818, 5.1.2.3 shall apply in addition to the following. For the purposes of this document, when processing of the abstract machine is interrupted by the receipt of a signal, the representation 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.

4.2 Sizes of integer types `<limits.h>`

The requirements in ISO/IEC 9899:2018, 5.2.4.2.1 shall apply. In addition 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 ISO/IEC 9899:2018, 7.20.1.5, the expression shall be suitable for use in `#if` preprocessing directives.

5 Language

5.1 Concepts

5.1.1 Storage durations and object lifetimes

For the purposes of this document the requirements from ISO/IEC 9899:2018, 6.2.4 shall apply in addition to the following. 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 occur during execution. 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)]

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 ISO/IEC 9899:2018, 7.22.3.

For the purposes of this document storage duration applies to an object's storage instance. Storage instances for string literals and some compound literals has static storage duration.[4)] There is a distinct instance of the object and distinct associated storage instance per thread for the storage instance of an object with thread storage duration. Storage instances of temporary objects has automatic storage duration.

5.1.2 Types

This document builds on the requirements of ISO/IEC 9899:2018, 6.2.5 regarding how a pointer type can be derived from a function type or an object type as follows.

A pointer type can 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; its 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. A pointer-to-function is valid if it refers to a valid function definition of the program. Pointers additionally can have a special value null that is different from the address of any storage instance and has no provenance (for object pointers),[5)] or from the address of any function of the program (for function

[1)]String literals, compound literals or certain objects with temporary lifetime can 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 is not required to 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)]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 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 complete object type.[6)] 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.

In addition to the requirements on the representation and alignment of pointers in ISO/IEC 9899:2018, it is implementation-defined whether other groups of pointer types have the same representation or alignment requirements.[7)]

5.1.3 Representation of types

5.1.3.1 General

For the purposes of this document, the requirements of ISO/IEC 9899:2018, 6.2.6.1 shall apply in addition to the following. 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).

An addressable storage instance[8)] 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 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. If the lifetimes of any two distinct addressable storage instances *X* and *Y* overlaps, 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.[9)]

of an arithmetic operation.

6)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 (ISO/IEC 9899:2018, 6.2.6) can sometimes represent valid values but with different provenance (and thus refer to different objects). Sometimes the object representation can 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 can restrict the set of operations that can be performed on the pointer.

7)An implementation can represent all pointers the same and with the same alignment requirements.

8)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 `&` (ISO/IEC 9899:2018, 6.5.3.2) to the object that gave rise to their definition (for other storage durations).

9)This means that no relative ordering between storage instances and the objects they represent

The object representation of a pointer object does not necessarily determine provenance of a pointer value; at different points of the program execution, identical object representations of pointer values can 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:[10)]

- Any byte of the object representation of `p` is used in an expression.[11)]
- The byte array pointed-to by the first argument of a call to the `fwrite` library function intersects with an object representation of `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.[12)]

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.[13)] Exposure of a storage instance is irreversible and constitutes a side effect in the abstract machine.

Unless stated otherwise, pointer value `p` is synthesized if it is constructed by one of the following:[14)]

- 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.

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).

10) Pointer values with exposed provenance can alias in ways that cannot be predicted by simple data flow analysis.

11) 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.

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

13) 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.

14) Synthesized pointer values can alias in ways that cannot be predicted by simple data flow analysis.