A Provenance-aware memory object model for C (slides)

Jens Gustedt

A Provenance-aware memory object model for C

ISO/IEC TS 6010:2023

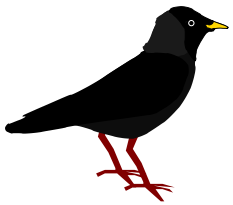
Jens Gustedt

INRIA – Camus

ICube – ICPS
Université de Strasbourg

joint work with
Peter Sewell, Kayvan Memarian, Victor B. F. Gomes, Martin Uecker

ISO/IEC JTC 1/SC 22/WG14 N3057 https://open-std.org/JTC1/SC22/WG14/www/docs/3057.pdf



https://modernc.gforge.inria.fr/

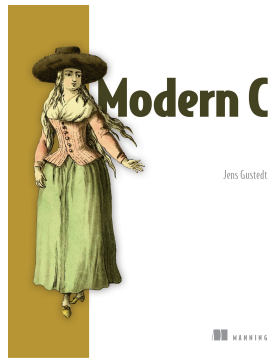


Table of Contents

The C programming language

ISO/IEC IS 9899:2018

Features

good: standardized, portable, stable, simple, efficient, extensible

bad: safety, out-of-bounds access, lack of encapsulation, pointer casts, lack of composability, side channels

One of the major programming languages

C is the base and interface for

- all modern processor architectures
  - 16 bit microprocessor (10 ¢)
  - ...
  - 128 bit multi-core high performance processor
- many other programming languages

C is highly optimizable

"As-if" rule

- The compiled code only performs visible effects.
- How the compiler achieves that is their business/secret.
- Internal rewriting into more efficient code with equivalent semantics.

Storage-backed objects

- Pointers allow access to objects from different angles/views.
- Many optimizations can only be applied if access is unique.
- Possible aliasing inhibits optimizations.

Aliasing techniques in C

Type-based aliasing analysis

```
void f(int* a, float* b) {
  // *a and *b will never alias
}
```

restrict-based aliasing analysis

```
void g(float*restrict a, float*restrict b) {
  // *a and *b will never alias
}
```

interdiction based aliasing prevention

```
// address of toto may never be taken
register double toto = 35.4;
```

How to avoid aliasing for out-of-bounds accesses?

Basic example

```
signed int y = 2, x = 1;
int main() {
    signed int* p = &x + 1;   // one beyond storage
    signed int* q = &y;       // pointers ok
    if (!memcmp(&p, &q, sizeof(p))) {       // exposure!
        *p = 11; // formed from x, accessing y?
    }
}
```

Question

- How to complement C's memory model such that the access to `*p` is forbidden?
- How to build tools that analyze such situations?

Table of Contents

Definitions

Storage instance

A storage instance is a continuous chunk of storage with a unique ID

- When a declared object or compound literal is instantiated
  - Static storage duration: program startup → program termination
  - Thread storage duration: thread startup → thread termination
  - Automatic storage duration: scope entrance → scope exit
- Allocated storage duration:
  - { `malloc`, `calloc`, `realloc` } → { `free`, `realloc` }
- Recycled storage (stack or heap) → new ID

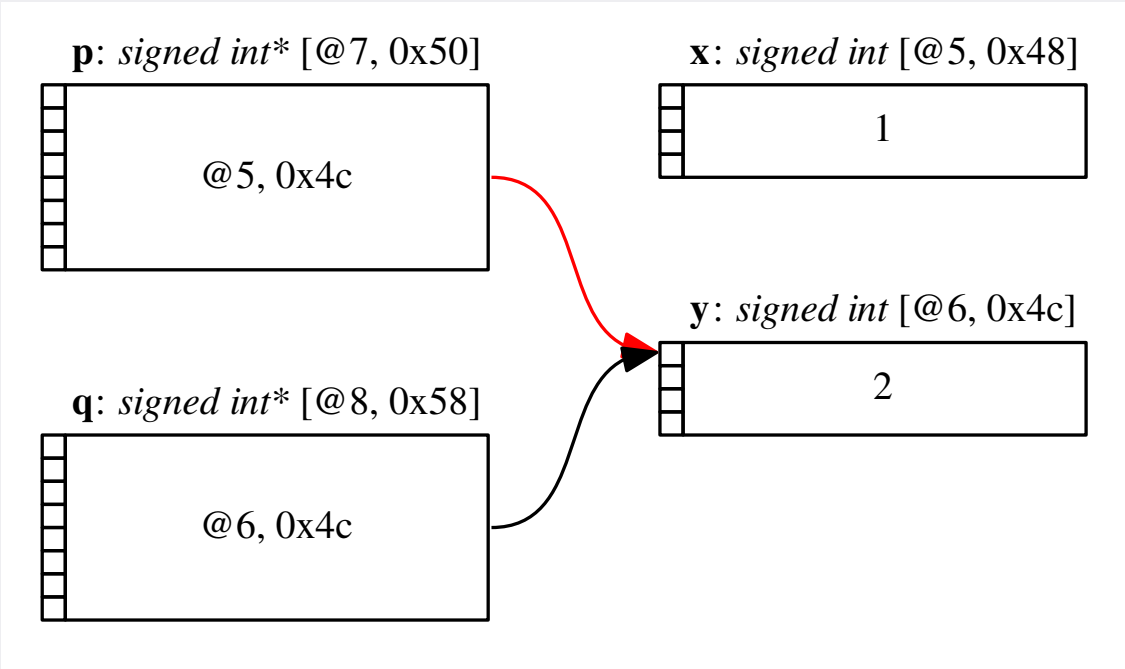
Provenance

Each valid pointer holds

- an ID of the corresponding storage instance
- an address

Detection of conflicts

A view from Cerberus



http://cerberus.cl.cam.ac.uk/cerberus

Exposure and synthesis

Conversion between pointers and integers

Bad, bad, bad!

All type information is lost!

All provenance information is lost!

Track provenance?

- through integers?
- through representation bytes?
- through control flow?

Exposure and synthesis

Terminology

- pointer to integer conversion: the whole storage instance is exposed
- integer to pointer conversion: the pointer is synthesized

Similar features

- `printf` or `scanf` with %p
- byte fiddling with in-storage representation of pointers
- `fwrite` or `fread` of a pointer representation

Exposure and synthesis

Rule

J has been exposed $\rightarrow$ A pointer to J may be synthesized

Interdiction

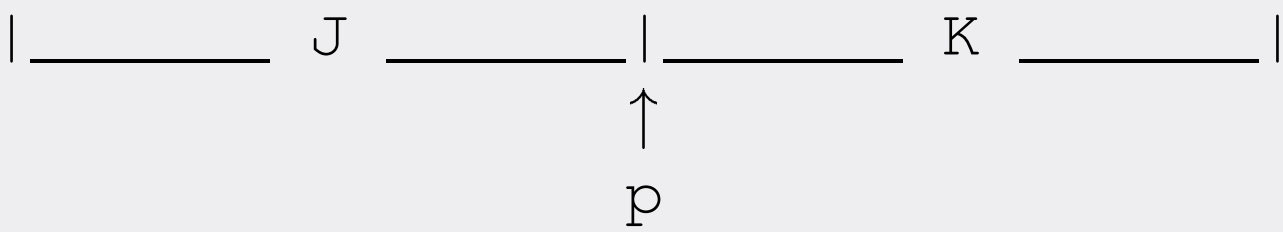
If a pointer to J is synthesized and J has not been exposed:

undefined behavior

Disambiguation

Two provenances for a synthesized pointer

A synthesized pointer p may point simultaneously to the end of exposed J and the start of exposed K



Rule

Only a use with either provenance J or K is allowed.

Table of Contents

The full technical specification

ISO/IEC TS 6010:2023

https://open-std.org/JTC1/SC22/WG14/www/docs/3057.pdf

A first step: know your constituency

What do programmers think?

What do compiler implementors think?

What do experts in WG14 think?

All depends!

- there are more diverging opinions than people
- opinions can be shifted with good arguments and a lot of patience
- everything flows, but very, very slowly

Tedious community work by the Cambridge group around Peter Sewell

A second step:
condense different strategies into sound models

Questions?

- Which granularity for the model? (allocation, object, member?)
- How to deal with reallocations?
- How to deal with type changes?
- How to deal with information leakage?

Method

- Yearlong discussions in WG14
- Liaison activities with WG21 (C++)
- A new Memory Model Study Group (head Peter Sewell)
- Mathematical specifications of several models
- An online test tool: Cerberus

A third step: integration and acceptance

The C standard itself

- Some new concepts and definitions
- Detailed changes all over
- More than 100 pages impacted

The community

- Convince implementors to obey to the "new" rules
- Tag behavior that is not conforming in open implementations as bugs
- Have these bugs accepted by the community

WG14

- Convince them of the proposed changes
- Partial success $\rightarrow$ propose a technical specification (TS 6010)

A fourth step: write an ISO norm

Collect all the material

(weird copyright rules)

- A paper on executable examples
- A paper on the semantic specification
- The "diff" to the C standard

$\rightarrow$ three annexes to the TS

Write the specification itself

- ISO language is quite particular
  - can, may, shall, should ... all have very restricted use
  - terminology sometimes fixed in other ISO documents
- ISO has no modern collaborative tools
- ISO is quite restrictive for the editorial "style"

A fifth step: publish an ISO norm

Navigate it through the voting process

- WG14
- National Bodies (AFNOR, DIN, INCITS, . . .)

publish

- ISO