# Into the depths of C: elaborating the de facto standards

**Kayvan Memarian** Justus Matthiesen James Lingard
Kyndylan Nienhuis David Chisnall Robert N. M. Watson
Peter Sewell

University of Cambridge

PLDI 2016, Santa Barbara

American National Standard

*INCITS/ISO/IEC 9899-2011[2012]*
*(ISO/IEC 9899-2011, IDT)*

*Information technology — Programming languages — C*

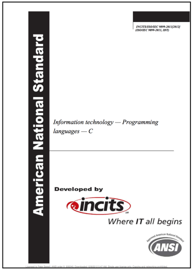
**Developed by**

incits

*Where IT all begins*

Approved American National Standard

ANSI



**American National Standard**

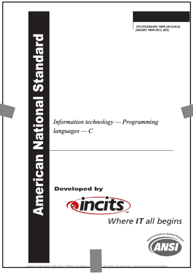
*INCITS/ISO/IEC 9899-2011[2012]*
*(ISO/IEC 9899-2011, IDT)*

*Information technology — Programming languages — C*

**Developed by**

incits
Where **IT** all begins

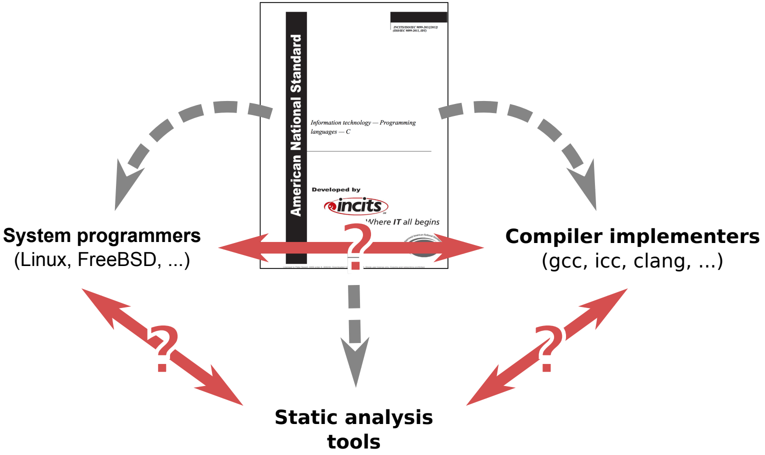
Approved American National Standard
ANSI



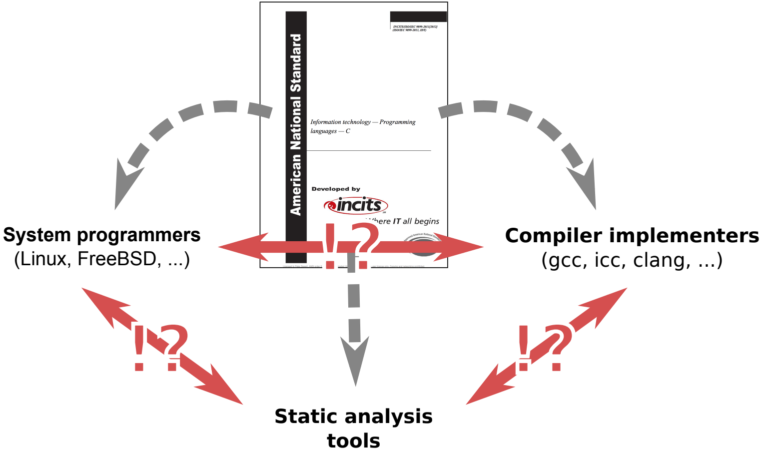

**System programmers**
(Linux, FreeBSD, ...)

**Compiler implementers**
(gcc, icc, clang, ...)

**Static analysis tools**



American National Standard
Information technology — Programming languages — C
Developed by
incits
Where IT all begins
System programmers
(Linux, FreeBSD, ...)
Compiler implementers
(gcc, icc, clang, ...)
Static analysis tools
?
?
?


American National Standard
Information technology — Programming languages — C
Developed by
incits
System programmers
(Linux, FreeBSD, ...)
!?
Compiler implementers
(gcc, icc, clang, ...)
!?
!?
Static analysis tools

# Why should we care?

## CVE-2009-1897 (Linux kernel 2.6.30 and 2.6.30.1)

"[...] when the **-fno-delete-null-pointer-checks** gcc option is omitted, allows local users to gain privileges via vectors involving a NULL pointer dereference [...]"

We present two contributions:

1. an in-depth analysis of the design space for the C memory object model
2. a formal model of a large fragment of C11 parametrised on the former

# Cerberus

De facto memory model(s)

# Cerberus project

**Cerberus** is a semantic model for a substantial fragment of C11

- **closely following** ISO C11

  when the standard is *clear* and *corresponds with practice*

- **parametric** on the *memory model*

  the main point of disagreement between the standard and practice

- **parametric** on implementation choices
- **executable** as a test oracle:

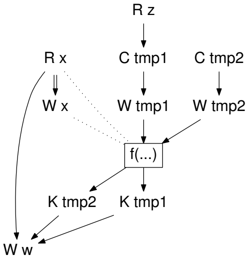
  can explore all behaviours or single executions of small programs

C11 expressions hide a lot of complexity:

C11 expressions hide a lot of complexity:

- loose and intricate ordering (sequence-before relation)

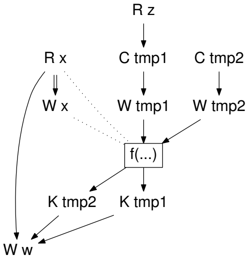
`w = x++ + f(z,2)`



C11 expressions hide a lot of complexity:

- loose and intricate ordering (sequence-before relation)

```
w = x++ + f(z,2)
```

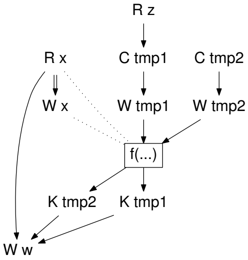



- hidden occurrence of memory operations (boundary of object lifetime)

C11 expressions hide a lot of complexity:

- loose and intricate ordering (sequence-before relation)

```
w = x++ + f(z,2)
```



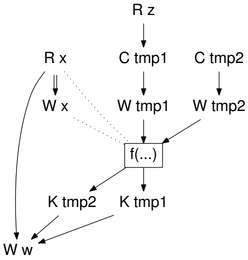

- hidden occurrence of memory operations (boundary of object lifetime)

- implicit type conversions (usual arithmetic conv; integer promotions)

C11 expressions hide a lot of complexity:

- loose and intricate ordering (sequence-before relation)

```
w = x++ + f(z,2)
```

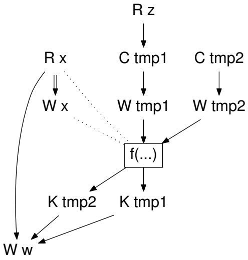



- hidden occurrence of memory operations (boundary of object lifetime)
- implicit type conversions (usual arithmetic conv; integer promotions)
- partiality (undefined behaviour)

C11 expressions hide a lot of complexity:

- loose and intricate ordering (sequence-before relation)

```
w = x++ + f(z,2)
```




- hidden occurrence of memory operations (boundary of object lifetime)
- implicit type conversions (usual arithmetic conv; integer promotions)
- partiality (undefined behaviour)
- parametricity (implementation-defined choices)

# Semantics by elaboration

$$\texttt{C11} \xrightarrow{\textit{elaboration}} \texttt{Core}$$

desugaring
statics

dynamics

We give the dynamics of C11 via elaboration (a compositional translation) to a purpose-built Core language:

- first-order functional
- typed (with pure/effectful separation)
- each language constructs have simple semantics
- with memory semantics factored out

$oTy$ ::= **types for C objects**

| | |
|---|---|
| `integer` | |
| `floating` | |
| `pointer` | |
| `cfunction` | |
| `array` ($oTy$) | |
| `struct` $tag$ | |
| `union` $tag$ | |

$bTy$ ::= **Core base types**

| | |
|---|---|
| `unit` | unit |
| `boolean` | boolean |
| `ctype` | Core type of C type exprs |
| `[`$bTy$`]` | list |
| ( $\overline{bTy_i}^{\,i}$ ) | tuple |
| $oTy$ | C object value |
| `loaded` $oTy$ | $oTy$ or unspecified value |

$coreTy$ ::= **Core types**

| | |
|---|---|
| $bTy$ | pure base type |
| `eff` $bTy$ | effectful base type |

$object\_value$ ::= **C object values**

| | |
|---|---|
| $intval$ | integer value |
| $floatval$ | floating-point value |
| $ptrval$ | pointer value |
| $name$ | C function pointer |
| `array` ( $\overline{object\_value_i}^{\,i}$ ) | C array value |
| ( `struct` $tag$ ){ $\overline{.member_i = memval_i}^{\,i}$ } | C struct value |
| ( `union` $tag$ ){ $.member = memval$ } | C union value |

$value$ ::= **Core values**

| | |
|---|---|
| $object\_value$ | C object value |
| `Specified` ($object\_value$) | non-unspecified loaded value |
| `Unspecified` ($ctype$) | unspecified loaded value |
| `Unit` | unit |
| `True` | true |
| `False` | false |
| $ctype$ | C type expr as value |
| $bTy[value_1, .., value_n]$ | list |
| $(value_1, .., value_n)$ | tuple |

$ptrop$ ::= **pointer operations involving the memory state**

| | |
|---|---|
| *pointer-equality-operator* | pointer equality comparison |
| *pointer-relational-operator* | pointer relational comparison |
| `ptrdiff` | pointer subtraction |
| `intFromPtr` | cast of pointer value to integer value |
| `ptrFromInt` | cast of integer value to pointer value |
| `ptrValidForDeref` | dereferencing validity predicate |

$a$ ::= **memory actions**

| | |
|---|---|
| `create` ($pe_1, pe_2$) | |
| `alloc` ($pe_1, pe_2$) | |
| `kill` ($pe$) | |
| `store` ($pe_1, pe_2, pe,$ *memory-order*) | |
| `load` ($pe_1, pe_2,$ *memory-order*) | |
| `rmw` ($pe_1, pe_2, pe_3, pe_4,$ *memory-order*$_1$, *memory-order*$_2$) | |

$pa$ ::= **memory actions with polarity**

| | |
|---|---|
| $a$ | positive, sequenced by both `let weak` and `let strong` |
| `neg` ($a$) | negative, only sequenced by `let strong` |

$pat$ ::=

| | |
|---|---|
| `_` | wildcard pattern |
| $ident$ | identifier pattern |
| $ctor(pat_1, .., pat_n)$ | constructor pattern |

$pe$ ::= **Core pure expressions**

| | |
|---|---|
| $ident$ | Core identifier |
| *<impl-const>* | implementation-defined constant |
| $value$ | value |
| `undef` (*ub-name*) | undefined behaviour |
| `error` ($string, pe$) | impl-defined static error |
| $ctor(pe_1, .., pe_n)$ | constructor application |
| `case` $pe$ `with` $\overline{\mid pat_i \Rightarrow pe_i}^{\,i}$ `end` | pattern matching |
| `array_shift`($pe_1, ctype, pe_2$) | pointer array shift |
| `member_shift`($pe, tag\,.\,member$) | pointer struct/union member shift |
| `not` ($pe$) | boolean not |
| $pe_1$ $binop$ $pe_2$ | binary operators |
| ( `struct` $tag$ ){ $\overline{.member_i = pe_i}^{\,i}$ } | C struct expression |
| ( `union` $tag$ ){ $.member = pe$ } | C union expression |
| $name(pe_1, .., pe_n)$ | pure Core function call |
| `let` $pat = pe_1$ `in` $pe_2$ | pure Core let |
| `if` $pe$ `then` $pe_1$ `else` $pe_2$ | pure Core if |
| `is_scalar`($pe$) | |
| `is_integer`($pe$) | |
| `is_signed`($pe$) | |
| `is_unsigned`($pe$) | |

$e$ ::= **Core expressions**

| | |
|---|---|
| `pure` ($pe$) | pure expression |
| `ptrop` ($ptrop, pe_1, .., pe_n$) | pointer op involving memory |
| $pa$ | memory action |
| `case` $pe$ `with` $\overline{\mid pat_i \Rightarrow e_i}^{\,i}$ `end` | pattern matching |
| `let` $pat = pe$ `in` $e$ | Core let |
| `if` $pe$ `then` $e_1$ `else` $e_2$ | Core if |
| `skip` | skip |
| `pcall` ( $pe, pe_1, .., pe_n$) | Core procedure call |
| `return` ($pe$) | Core procedure return |
| `unseq` ($e_1, .., e_n$) | unsequenced expressions |
| `let weak` $pat = e_1$ `in` $e_2$ | weak sequencing |
| `let strong` $pat = e_1$ `in` $e_2$ | strong sequencing |
| `let atomic` ($sym : oTy$) $= a_1$ `in` $pa_2$ | atomic sequencing |
| `indet` $[n]$ ($e$) | indeterminately sequenced expr |
| `bound` $[n]$ ($e$) | ...and boundary |
| `nd` ($e_1, .., e_n$) | nondeterministic sequencing |
| `save` $label$( $\overline{ident_i : ctype_i}^{\,i}$ ) `in` $e$ | save label |
| `run` $label$( $\overline{ident_i := pe_i}^{\,i}$ ) | run from label |
| `par` ($e_1, .., e_n$) | cppmem thread creation |
| `wait` (*thread-id*) | wait for thread termination |

$definition$ ::= **Core definitions**

| | |
|---|---|
| `fun` $name$( $\overline{ident_i : bTy_i}^{\,i}$ ) : $bTy := pe$ | Core function definition |
| `proc` $name$( $\overline{ident_i : bTy_i}^{\,i}$ ) : `eff` $bTy := e$ | Core procedure definition |

*oTy* ::= **types for C objects**
| `integer`
| `floating`
| `pointer`
| `cfunction`
| `array` (*oTy*)
| `struct` *tag*
| `union` *tag*

*bTy* ::= **Core base types**

| | |
|---|---|
| \| `unit` | unit |
| \| `boolean` | boolean |
| \| `ctype` | Core type of C type exprs |
| \| [*bTy*] | list |
| \| ( $\overline{bTy_i}^{\,i}$ ) | tuple |
| \| *oTy* | C object value |
| \| `loaded` *oTy* | *oTy* or unspecified value |

*coreTy* ::= **Core types**
| *bTy* pure base type
| `eff` *bTy* effectful base type

*pat* ::=

| | |
|---|---|
| \| _ | wildcard pattern |
| \| *ident* | identifier pattern |
| \| *ctor* ($pat_1$ , .. , $pat_n$) | constructor pattern |

*pe* ::= **Core pure expressions**

| | |
|---|---|
| \| *ident* | Core identifier |
| \| <*impl-const*> | implementation-defined constant |
| \| *value* | value |
| \| `undef` (*ub-name*) | undefined behaviour |
| \| `error` (*string* , *pe*) | impl-defined static error |
| \| *ctor* ($pe_1$ , .. , $pe_n$) | constructor application |
| \| `case` *pe* `with` $\overline{\mid pat_i \Rightarrow pe_i}^{\,i}$ `end` | pattern matching |
| \| `array_shift` ($pe_1$ , ctype , $pe_2$) | pointer array shift |


*oTy* ::= **types for C objects**
| `integer`
| `floating`
| `pointer`
| `cfunction`
| `array` (*oTy*)
| `struct` *tag*
| `union` *tag*

*bTy* ::= **Core base types**
| `unit` unit
| `boolean` boolean
| `ctype` Core type of C type exprs
| [*bTy*] list
| ( $\overline{bTy_i}^{\,i}$ ) tuple
| *oTy* C object value
| `loaded` *oTy* *oTy* or unspecified value


| | |
|---|---|
| \| `ptrdiff` | pointer subtraction |
| \| `intFromPtr` | cast of pointer value to integer value |
| \| `ptrFromInt` | cast of integer value to pointer value |
| \| `ptrValidForDeref` | dereferencing validity predicate |

*a* ::= **memory actions**
| `create` ($pe_1$ , $pe_2$ )
| `alloc` ($pe_1$ , $pe_2$ )
| `kill` (*pe*)
| `store` ($pe_1$ , $pe_2$ , *pe* , *memory-order*)
| `load` ($pe_1$ , $pe_2$ , *memory-order*)
| `rmw` ($pe_1$ , $pe_2$ , $pe_3$ , $pe_4$ , $memory\text{-}order_1$ , $memory\text{-}order_2$)

*pa* ::= **memory actions with polarity**
| *a* positive, sequenced by both `let weak` and `let strong`
| `neg` (*a*) negative, only sequenced by `let strong`

| | |
|---|---|
| \| `return` (*pe*) | Core procedure return |
| \| `unseq` ($e_1$ , .. , $e_n$) | unsequenced expressions |
| \| `let weak` *pat* = $e_1$ `in` $e_2$ | weak sequencing |
| \| `let strong` *pat* = $e_1$ `in` $e_2$ | strong sequencing |
| \| `let atomic` (*sym* : *oTy*) = $a_1$ `in` $pa_2$ | atomic sequencing |
| \| `indet` [*n*] (*e*) | indeterminately sequenced expr |
| \| `bound` [*n*] (*e*) | ...and boundary |
| \| `nd` ($e_1$ , .. , $e_n$) | nondeterministic sequencing |
| \| `save` *label* ( $\overline{ident_i : ctype_i}^{\,i}$ ) `in` *e* | save label |
| \| `run` *label* ( $\overline{ident_i := pe_i}^{\,i}$ ) | run from label |
| \| `par` ($e_1$ , .. , $e_n$) | cppmem thread creation |
| \| `wait` (*thread-id*) | wait for thread termination |

*definition* ::= **Core definitions**

| | |
|---|---|
| \| `fun` *name* ( $\overline{ident_i : bTy_i}^{\,i}$ ) : *bTy* := *pe* | Core function definition |
| \| `proc` *name* ( $\overline{ident_i : bTy_i}^{\,i}$ ) : `eff` *bTy* := *e* | Core procedure definition |

*oTy* ::= **types for C objects**
| `integer`
| `floating`
| `pointer`
| `cfunction`

*bTy* ::= **Core base types**
| `unit` unit
| `boolean` boolean
| `ctype` Core type of C type exprs
| [*bTy*] list

*pat* ::=
| _ wildcard pattern
| *ident* identifier pattern
| $ctor\,(pat_1, .., pat_n)$ constructor pattern

*pe* ::= **Core pure expressions**

*ptrop* ::= **pointer operations involving the memory state**

| | |
|---|---|
| \| *pointer-equality-operator* | pointer equality comparison |
| \| *pointer-relational-operator* | pointer relational comparison |
| \| `ptrdiff` | pointer subtraction |
| \| `intFromPtr` | cast of pointer value to integer value |
| \| `ptrFromInt` | cast of integer value to pointer value |
| \| `ptrValidForDeref` | dereferencing validity predicate |

*a* ::= **memory actions**
| `create` $(pe_1, pe_2)$
| `alloc` $(pe_1, pe_2)$
| `kill` $(pe)$
| `store` $(pe_1, pe_2, pe, \textit{memory-order})$
| `load` $(pe_1, pe_2, \textit{memory-order})$
| `rmw` $(pe_1, pe_2, pe_3, pe_4, \textit{memory-order}_1, \textit{memory-order}_2)$

| `store` $(pe_1, pe_2, pe, \textit{memory-order})$
| `load` $(pe_1, pe_2, \textit{memory-order})$
| `rmw` $(pe_1, pe_2, pe_3, pe_4, \textit{memory-order}_1, \textit{memory-order}_2)$

*pa* ::= **memory actions with polarity**
| *a* positive, sequenced by both `let weak` and `let strong`
| `neg` $(a)$ negative, only sequenced by `let strong`

| `save` $label\,(\overline{ident_i : ctype_i}^{\,i})$ `in` $e$ save label
| `run` $label\,(\overline{ident_i := pe_i}^{\,i})$ run from label
| `par` $(e_1, .., e_n)$ cppmem thread creation
| `wait` $(\textit{thread-id})$ wait for thread termination

*definition* ::= **Core definitions**
| `fun` $name\,(\overline{ident_i : bTy_i}^{\,i}) : bTy := pe$ Core function definition
| `proc` $name\,(\overline{ident_i : bTy_i}^{\,i}) :$ `eff` $bTy := e$ Core procedure definition

| $pe$ ::= | **Core pure expressions** |
|---|---|
| \| *ident* | Core identifier |
| \| <*impl-const*> | implementation-defined constant |
| \| *value* | value |
| \| `undef` (*ub-name*) | undefined behaviour |
| \| `error` (*string* , *pe*) | impl-defined static error |
| \| *ctor* ($pe_1$ , .. , $pe_n$) | constructor application |
| \| `case` *pe* `with` $\overline{\mid pat_i \texttt{=>} pe_i}^{\,i}$ `end` | pattern matching |
| \| `array_shift`($pe_1$ , *ctype* , $pe_2$) | pointer array shift |
| \| `member_shift`(*pe* , *tag* . *member*) | pointer struct/union member shift |
| \| `not` (*pe*) | boolean not |
| \| $pe_1$ *binop* $pe_2$ | binary operators |
| \| ( `struct` *tag*){ $\overline{.\,member_i = pe_i}^{\,i}$ } | C struct expression |
| \| ( `union` *tag*){. *member* = *pe*} | C union expression |
| \| *name*($pe_1$ , .. , $pe_n$) | pure Core function call |
| \| `let` *pat* = $pe_1$ `in` $pe_2$ | pure Core let |
| \| `if` *pe* `then` $pe_1$ `else` $pe_2$ | pure Core if |
| \| `is_scalar`(*pe*) | |
| \| `is_integer`(*pe*) | |
| \| `is_signed`(*pe*) | |
| \| `is_unsigned`(*pe*) | |

| $e ::=$ | **Core expressions** |
|---|---|
| $\mid\ \texttt{pure}\,(pe)$ | pure expression |
| $\mid\ \texttt{ptrop}\,(ptrop\,,pe_1\,,\,..\,,pe_n)$ | pointer op involving memory |
| $\mid\ pa$ | memory action |
| $\mid\ \texttt{case}\ pe\ \texttt{with}\ \overline{\mid pat_i \texttt{=>} e_i}^{\,i}\ \texttt{end}$ | pattern matching |
| $\mid\ \texttt{let}\ pat = pe\ \texttt{in}\ e$ | Core let |
| $\mid\ \texttt{if}\ pe\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ | Core if |
| $\mid\ \texttt{skip}$ | skip |
| $\mid\ \texttt{pcall}\,(\,pe\,,pe_1\,,\,..\,,pe_n)$ | Core procedure call |
| $\mid\ \texttt{return}\,(pe)$ | Core procedure return |
| $\mid\ \texttt{unseq}\,(e_1\,,\,..\,,e_n)$ | unsequenced expressions |
| $\mid\ \texttt{let}\ \texttt{weak}\ pat = e_1\ \texttt{in}\ e_2$ | weak sequencing |
| $\mid\ \texttt{let}\ \texttt{strong}\ pat = e_1\ \texttt{in}\ e_2$ | strong sequencing |
| $\mid\ \texttt{let}\ \texttt{atomic}\left(sym : oTy\right) = a_1\ \texttt{in}\ pa_2$ | atomic sequencing |
| $\mid\ \texttt{indet}\ [n]\,(e)$ | indeterminately sequenced expr |
| $\mid\ \texttt{bound}\ [n]\,(e)$ | ...and boundary |
| $\mid\ \texttt{nd}\,(e_1\,,\,..\,,e_n)$ | nondeterministic sequencing |
| $\mid\ \texttt{save}\ label\,(\,\overline{ident_i : ctype_i}^{\,i}\,)\ \texttt{in}\ e$ | save label |
| $\mid\ \texttt{run}\ label\,(\,\overline{ident_i := pe_i}^{\,i}\,)$ | run from label |
| $\mid\ \texttt{par}\,(e_1\,,\,..\,,e_n)$ | cppmem thread creation |
| $\mid\ \texttt{wait}\,(thread\text{-}id)$ | wait for thread termination |

```
int f(int n) {
  int x = 10;
  n+x;
}


int main(void) {
  return f(3);
}
```

```
proc f(n: pointer): eff loaded integer :=
  let strong x: pointer = create(Ivalignof("signed int"), "signed int") in
  store("signed int", x, conv_loaded_int("signed int", Specified(10))) ;

  let weak (a1_: loaded integer, a2_: loaded integer) =
    unseq(load("signed int", n), load("signed int", x)) in

  pure(case (a1_, a2_) of
    | (Specified(a1: integer), Specified(a2: integer)) =>
        Specified(catch_exceptional_condition("signed int",
                    conv_int("signed int", a1) + conv_int("signed int", a2)))
    | _ =>
        undef(<<UB036_exceptional_condition>>)
  end) ;

  kill(x) ;
  pure(undef(<<UB088_reached_end_of_function>>)) ;
  save ret (z: loaded integer) in
    pure(z)
```

### 6.5.7 Bitwise shift operators

**Syntax**

1 *shift-expression:*

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

**Constraints**

2 Each of the operands shall have integer type.

**Semantics**

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

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

5 ... *similarly for* **E1 >> E2** ...

```
⟦e1 << e2⟧ =
  sym_e1   := E.fresh_symbol; sym_e2   := E.fresh_symbol;
  sym_obj1 := E.fresh_symbol; sym_obj2 := E.fresh_symbol;
  sym_prm1 := E.fresh_symbol; sym_prm2 := E.fresh_symbol;
  sym_res  := E.fresh_symbol;
  core_e1 := ⟦e1⟧; core_e2 := ⟦e2⟧;
  E.return(
    let weak (sym_e1,sym_e2) = unseq(core_e1,core_e2) in
    pure(
      case (sym_e1, sym_e2) with
      | (_, Unspecified(_)) =>
          undef(Exceptional_condition)
      | (Unspecified(_), _) =>
          (IF is_unsigned_integer_type(ctype_of e1) THEN
          Unspecified(result_ty)
          ELSE
          undef(Exceptional_condition))
      | (Specified(sym_obj1), Specified(sym_obj2)) =>
          let sym_prm1 =
            integer_promotion (ctype_of e1) sym_obj1 in
          let sym_prm2 =
            integer_promotion (ctype_of e2) sym_obj2 in
          if sym_prm2 < 0 then
            undef(Negative_shift)
          else if ctype_width(result_ty) <= sym_prm2 then
            undef(Shift_too_large)
          else
          (IF is_unsigned_integer_type(ctype_of e1) THEN
          Specified(sym_prm1*(2^sym_prm2)
                    rem_t (Ivmax(result_ty)+1))
          ELSE
          if sym_prm1 < 0 then
            undef(Exceptional_condition)
          else
            let sym_res = sym_prm1*(2^sym_prm2) in
            if is_representable(sym_res,result_ty) then
              Specified(sym_res)
            else
              undef(Exceptional_condition) )))
```

### 6.5.7 Bitwise shift operators

**Syntax**

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

**Constraints**

Each of the operands shall have integer type.

**Semantics**

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

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

*. . . similarly for* **E1 >> E2** *. . .*

```
[e1 << e2] =
  sym_e1   := E.fresh_symbol; sym_e2   := E.fresh_symbol;
  sym_obj1 := E.fresh_symbol; sym_obj2 := E.fresh_symbol;
  sym_prm1 := E.fresh_symbol; sym_prm2 := E.fresh_symbol;
  sym_res  := E.fresh_symbol;
  core_e1 := [e1]; core_e2 := [e2];
  E.return(
   let weak (sym_e1,sym_e2) = unseq(core_e1,core_e2) in
   pure(
    case (sym_e1, sym_e2) with
    | (_, Unspecified(_)) =>
        undef(Exceptional_condition)
    | (Unspecified(_), _) =>
        (IF is_unsigned_integer_type(ctype_of e1) THEN
        Unspecified(result_ty)
        ELSE
        undef(Exceptional_condition))
    | (Specified(sym_obj1), Specified(sym_obj2)) =>
        let sym_prm1 =
          integer_promotion (ctype_of e1) sym_obj1 in
        let sym_prm2 =
          integer_promotion (ctype_of e2) sym_obj2 in
        if sym_prm2 < 0 then
          undef(Negative_shift)
        else if ctype_width(result_ty) <= sym_prm2 then
          undef(Shift_too_large)
        else
          (IF is_unsigned_integer_type(ctype_of e1) THEN
          Specified(sym_prm1*(2^sym_prm2)
                rem_t (Ivmax(result_ty)+1))
          ELSE
          if sym_prm1 < 0 then
            undef(Exceptional_condition)
```

```
[[e1 << e2]] =
  sym_e1   := E.fresh_symbol; sym_e2   := E.fresh_symbol;
  sym_obj1 := E.fresh_symbol; sym_obj2 := E.fresh_symbol;
  sym_prm1 := E.fresh_symbol; sym_prm2 := E.fresh_symbol;
  sym_res  := E.fresh_symbol;
  core_e1 := [[e1]]; core_e2 := [[e2]];
  E.return(
    let weak (sym_e1,sym_e2) = unseq(core_e1,core_e2) in
    pure(
      case (sym_e1, sym_e2) with
      | (_, Unspecified(_)) =>
          undef(Exceptional_condition)
      | (Unspecified(_), _) =>
          (IF is_unsigned_integer_type(ctype_of e1) THEN
          Unspecified(result_ty)
          ELSE
          undef(Exceptional_condition))
      | (Specified(sym_obj1), Specified(sym_obj2)) =>
          let sym_prm1 =
            integer_promotion (ctype_of e1) sym_obj1 in
          let sym_prm2 =
            integer_promotion (ctype_of e2) sym_obj2 in
          if sym_prm2 < 0 then
            undef(Negative_shift)
          else if ctype_width(result_ty) <= sym_prm2 then
            undef(Shift_too_large)
          else
            (IF is_unsigned_integer_type(ctype_of e1) THEN
            Specified(sym_prm1*(2^sym_prm2)
                 rem_t (Ivmax(result_ty)+1))
            ELSE
            if sym_prm1 < 0 then
              undef(Exceptional_condition)
```

### 6.5.7 Bitwise shift operators

**Syntax**

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

**Constraints**

Each of the operands shall have integer type.

**Semantics**

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

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

... *similarly for* `E1 >> E2` ...

### 6.5.7 Bitwise shift operators

**Syntax**

1 *shift-expression:*

*additive-expression*

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

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

**Constraints**

2 Each of the operands shall have integer type.

**Semantics**

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

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

5 ... *similarly for* **E1 >> E2** ...

```
[[e1 << e2]] =
  sym_e1   := E.fresh_symbol; sym_e2   := E.fresh_symbol;
  sym_obj1 := E.fresh_symbol; sym_obj2 := E.fresh_symbol;
  sym_prm1 := E.fresh_symbol; sym_prm2 := E.fresh_symbol;
  sym_res  := E.fresh_symbol;
  core_e1 := [[e1]]; core_e2 := [[e2]];
  E.return(
    let weak (sym_e1,sym_e2) = unseq(core_e1,core_e2) in
    pure(
      case (sym_e1, sym_e2) with
      | (_, Unspecified(_)) =>
          undef(Exceptional_condition)
      | (Unspecified(_), _) =>
          (IF is_unsigned_integer_type(ctype_of e1) THEN
          Unspecified(result_ty)
          ELSE
          undef(Exceptional_condition))
      | (Specified(sym_obj1), Specified(sym_obj2)) =>
          let sym_prm1 =
            integer_promotion (ctype_of e1) sym_obj1 in
          let sym_prm2 =
            integer_promotion (ctype_of e2) sym_obj2 in
          if sym_prm2 < 0 then
            undef(Negative_shift)
          else if ctype_width(result_ty) <= sym_prm2 then
            undef(Shift_too_large)
          else
            (IF is_unsigned_integer_type(ctype_of e1) THEN
            Specified(sym_prm1*(2^sym_prm2)
                      rem_t (Ivmax(result_ty)+1))
            ELSE
            if sym_prm1 < 0 then
              undef(Exceptional_condition)
            else
              let sym_res = sym_prm1*(2^sym_prm2) in
              if is_representable(sym_res,result_ty) then
                Specified(sym_res)
              else
                undef(Exceptional_condition) )))
```

## Validation

executability helped with the validation by testing against compilers and existing semantics:

- one of Ellison et al. testsuites
- 400 larger Csmith generated tests (40-600 lines long)

## Use as an oracle

exhaustive exploration allows for:

- detection of undefine behaviours
- with parametricity allow the emulation of specific implementations

# Integration with C/C++11 concurrency model (Nienhuis)

thanks to parameterisation on the memory object model:

- **lightweight integration**: no modification required to the concurrency model
- **improving over cppmem**: allows the simulation of richer concurrent programs

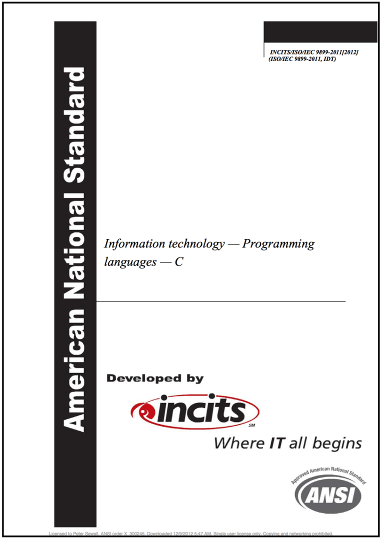
Caveats:

- only for a restricted object model
- meant to be used for pseudo-random explorations
- more engineering required

Cerberus

# De facto memory model(s)

# ISO C




Target of past formalisations:

- Gurevich and Higgens (1993)
- Cook and Subramanian (1994)
- Norrish (1998)
- Papaspyrou (1998)
- Ellison et al. (2012)
- Krebbers (2014)

and work on C-like languages

- CompCert, seL4, VCC
- Besson et al; Kang et al

So do we for the expression and statement dynamics (mostly §6).

But for the memory model we need more.

# ISO C vs practice

Sometimes unclear or ambiguous $\Rightarrow$ allowing conflicting interpretations:

- notion of **subobject** left undefined
- ambiguity regarding unspecified values
- allocated memory **regions**

Doesn't always match practice anymore – "de facto" interpretations emerged in the assumptions:

- relied upon by the corpus of C code for its "correct" execution
- necessary for the soundness of compiler optimisations.

# Investigating “de facto” C(s)

Needed a more empirical approach:

1. detailed analysis of the design space (85 questions), resulting in a “semantic” testsuite
2. surveying the belief and practices of programmers and compiler writers
3. testing compilers (and other semantics) on our testsuite

(see also Chisnall et al. in ASPLOS, 2015)

# Analysing the design space

| | |
|---|---|
| 20 | Pointer provenance |
| 18 | Other questions about pointers |
| 4 | Accesses to related structure and union types |
| 2 | Pointer lifetime end |
| 2 | Invalid accesses |
| 2 | Trap representations |
| 11 | Unspecified values |
| 13 | Structure and union padding |
| 9 | Effective types |
| 5 | Other questions |

- for 39 the ISO standard is unclear
- for 27 the de facto standards are unclear, in some cases with significant differences between usage and implementation
- for 27 there are significant differences between the ISO and the de facto standards

# Analysing the design space

| | |
|---|---|
| 20 | **Pointer provenance** |
| 18 | Other questions about pointers |
| 4 | Accesses to related structure and union types |
| 2 | Pointer lifetime end |
| 2 | Invalid accesses |
| 2 | Trap representations |
| 11 | **Unspecified values** |
| 13 | Structure and union padding |
| 9 | Effective types |
| 5 | Other questions |

- for 39 the ISO standard is unclear
- for 27 the de facto standards are unclear, in some cases with significant differences between usage and implementation
- for 27 there are significant differences between the ISO and the de facto standards

# Abstract concrete memory?

Originally one could think of C as manipulating:

"*the same sort of objects that most computers do, namely characters, numbers, and addresses*", Kernigan and Ritchie [24, p.2].

While still true at runtime, the ISO standard involve more abstract values:

- pointers with provenance
- unspecified values
- type regions of memory

Compiler optimisations do rely on these abstractions

# Abstract or concrete memory?

(from R. Krebbers)

```
int y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

# Abstract or concrete memory?

(from R. Krebbers)

```c
int y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

concrete memory would give:

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

# Abstract or concrete memory?

(from R. Krebbers)

```
int y=2, x=1;
int main() {
  int *p = &x + 1;
  int *q = &y;
  if (memcmp(&p, &q, sizeof(p)) == 0) {
    *p = 11;  // does this have undefined behaviour?
    printf("x=%d y=%d *p=%d *q=%d\n",x,y,*p,*q);
  }
}
```

concrete memory would give:

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

but we observe:

gcc x=1 y=2 *p=11 *q=2
icc x=1 y=2 *p=11 *q=11

**Q25. Can one do relational comparison (with <, >, <=, or >=) of two pointers to separately allocated objects?**

```
int  y = 2, x=1;
int main() {
  int *p = &x, *q = &y;
  _Bool b1 = (p < q); // defined behaviour?
  _Bool b2 = (p > q); // defined behaviour?
  printf("(p<q) = %s  (p>q) = %s\n",
         b1?"true":"false", b2?"true":"false");
}
```

**Q25. Can one do relational comparison (with <, >, <=, or >=) of two pointers to separately allocated objects?**

```
int  y = 2, x=1;
int main() {
  int *p = &x, *q = &y;
  _Bool b1 = (p < q); // defined behaviour?
  _Bool b2 = (p > q); // defined behaviour?
  printf("(p<q) = %s  (p>q) = %s\n",
         b1?"true":"false", b2?"true":"false");
}
```

Forbidden by ISO (would fail on segmented memory) ...

... but common practice (e.g. memory allocator, lock order).

Outside the scope of block-ID/offset semantics.

# Unspecified values?

The ISO standard defines a notion of **unspecified values**:

**3.19.3**

1 **unspecified value**
valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance

2 NOTE An unspecified value cannot be a trap representation.

and refers to them in (mostly) two contexts:

- for otherwise-uninitialized objects with automatic storage duration;
- for the values of padding bytes on writes to structs/unions.

# Unspecified values?

However, the ISO text leaves room for several rather different semantic interpretations:

1. stable concrete value, choosen nondeterministically;
2. abstract value, on which the language operators are defined somehow;
3. a fresh symbolic value (per bit, byte, or value) and allow computation on that.

# Unspecified values?

**Q61. After an explicit write of a padding byte, does that byte hold a well-defined value? (not an unspecified value)**

```
typedef struct { char c; float f; int i; } st;
int main() {
  // check there is a padding byte between c and f
  size_t offset_padding = offsetof(st,c)+sizeof(char);
  if (offsetof(st,f)>offset_padding) {
      st s;
      unsigned char *p = ((unsigned char*)(&s))
        + offset_padding;
      *p = 'A';
      unsigned char c1 = *p;
      // does c1 hold 'A', not an unspecified value?
      printf("c1=%c\n",c1);
  }
}
```

# Unspecified values?

**Q52. Do operations on unspecified values result in unspecified values?**

```
int main() {
  int i;
  int *p = &i;
  int j = (i-i);    // is this an unspecified value?
  _Bool b = (j==j); // can this be false?
  printf("b=%s\n",b?"true":"false");
}
```

# Many more interesting questions...

| | |
|---|---|
| Pointer provenance basics | 3 |
| Pointer provenance via integer types | 5 |
| Pointers involving multiple provenances | 5 |
| Pointer provenance via pointer representation copying | 4 |
| Pointer provenance and union type punning | 2 |
| Pointer provenance via IO | 1 |
| Stability of pointer values | 1 |
| Pointer equality comparison (with == or !=) | 3 |
| Pointer relational comparison (with <, >, <=, or >=) | 3 |
| Null pointers | 3 |
| Pointer arithmetic | 6 |
| Casts between pointer types | 2 |
| Accesses to related structure and union types | 4 |
| Pointer lifetime end | 2 |
| Invalid accesses | 2 |
| Trap representations | 2 |
| Unspecified values | 11 |
| Structure and union padding | 13 |
| Basic effective types | 2 |
| Effective types and character arrays | 1 |
| Effective types and subobjects | 6 |
| Other questions | 5 |

# Two Surveys

1. early 2013, 42 questions given to a small number of:
    - ISO C or C++ standards committee members
    - C analysis tool developers
    - experts in C formal semantics
    - compiler writers, and systems programmers
2. early 2015, selected 15 questions:
    - only asked about "de facto" C
    - larger audience (323 responses)
    - posted on technical mailing lists: gcc, llvmdev, cfe-dev, libc-alpha, xorg, freebsd-developers, xen-devel, ...

# Experimental data

We ran our testsuite on various compilers and static analysis tools:

- gcc: 4.8.x, 4.9.4, 5.3.0 (on x86_64)
- clang: 3.0, 3.3, 3.5.2, 3.6.2, 3.7.0, 3.8.0 (on x86_64),
  3.4.1, 3.7.0, 3.8.0 (on Cheri)
- CompCert 2.6
- clang's MSan, ASan UBSan
- TrustInSoft's tis-interpreter, kcc, ch2o

# Back to Cerberus

Based on this study:

- ongoing work on formalising a candidate model
- to be plugged to Cerberus
- aim to provide a test oracle for small-scale programs
- engaging with the ISO C committee (WG14)

# Conclusion

The C used in practice has diverged from ISO C in some ways.
$\Rightarrow$ tension between programmers and compilers

This works aims at capturing and formalising these “de facto” C(s), to clarify what C is in reality.

Analysis document, survey results and some WG14 N documents at:

`www.cl.cam.ac.uk/~km569/cerberus/`