

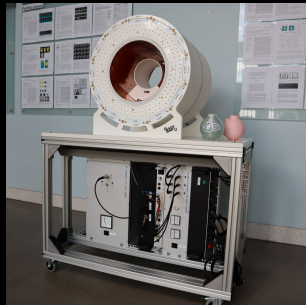
Undefined Behavior in the C Language - A Lost Cause?

Martin Uecker

Graz University of Technology

About Me

- ▶ Professor for Biomedical Imaging
- ▶ Linux user since 1995
- ▶ Open science and free software enthusiast
- ▶ GCC contributor (C front-end)
- ▶ Member of ISO JTC1 SC22 WG14 (C)



Open-Source Low-Field MRI

About Me

Open source work:

- ▶ Maintainer: BART Toolbox for Computational MRI
- ▶ Contributions to the GCC C frontend
- ▶ My Linux Kernel distribution:

```
include/linux/compiler.h:
```

```
/*  
 * This returns a constant expression while determining if an argument is  
 * a constant expression, most importantly without evaluating the argument.  
 * Glory to Martin Uecker <Martin.Uecker@med.uni-goettingen.de>  
 * ...  
 */  
#define __is_constexpr(x) \  
(sizeof(int) == sizeof(*(8 ? ((void *)((long)(x) * 0l)) : (int *)8)))
```

Why C?

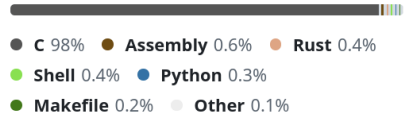
- ▶ Portability
- ▶ Long-term stability
- ▶ Fast compilation
- ▶ Efficient code
- ▶ Interoperability
- ▶ Relatively simple and elegant
- ▶ What you see is what you get
- ▶ Many tools, documentation, training materials, etc.
- ▶ An international standard
- ▶ Gets out of the way when necessary

Tiobe Index

Sep 2026	Sep 2025	Change	Programming Language	Ratings	Change
1	1		 Python	17.76%	-8.22%
2	3	▲	 C	10.28%	+1.63%
3	2	▼	 C++	8.67%	-0.13%
4	4		 Java	7.54%	-0.81%
5	5		 C#	4.22%	-2.16%
6	6		 JavaScript	2.76%	-0.46%
7	7		 Visual Basic	2.55%	-0.28%
11		▲	 SQL	2.16%	

Linux Kernel (43 million lines):

Languages



<https://www.tiobe.com/tiobe-index/>

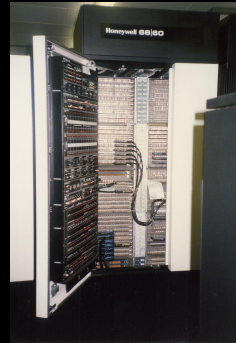
<https://ostechnix.com/linux-kernel-7-2-43-million-lines-of-code/>

- ▶ The Abstract Machine
- ▶ Undefined Behavior
- ▶ Time Travel
- ▶ Progress...
- ▶ Memory Safety?
- ▶ Standardization

C89 had to support: two's complement, ones' complement, sign-magnitude, weird pointer representations, segmented memory, address spaces, etc.

char	9
short	18
int	36
long	36
long long	72

Honeywell: Bitwidth of integer types



Honeywell
Mainframe, Wikipedia

<https://begriffs.com/posts/2018-11-15-c-portability.html>

The Abstract Machine

C23,

5.1.2.4 Program semantics

§1 The semantic descriptions in this document describe the behavior of an **abstract machine** in which issues of optimization are irrelevant.

§4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation is not required to ...

Note: This is the so-called as-if rule.

Observable Behavior

§6 The least requirements on a conforming implementation are:

- **Volatile** accesses to objects are evaluated strictly according to the rules of the abstract machine.
- At program termination, **all data written into files** shall be identical to the result that execution of the program according to the abstract semantics would have produced.
- The **input and output** dynamics of **interactive devices** shall take place as specified in 7.23.3.

The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages appear prior to a program waiting for input. This is the **observable behavior** of the program.

5.1.1.3 Diagnostics

§1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages are not required to be produced in other circumstances

Note: Compilers may need a flag (e.g. `-pedantic`) to emit all required diagnostics.

Maximally Portable Programs

C23, 4 Conformance

§5 A **strictly conforming program** shall use only those features of the language and library specified in this document. It shall not produce output dependent on any unspecified, undefined, or implementation- defined behavior, and shall not exceed any minimum implementation limit.

§9 A **conforming program** is one that is acceptable to a conforming implementation

C99 Rationale:

A strictly conforming program is another term for a maximally portable program. The goal is to give the programmer a **fighting chance** to make powerful C programs that are also highly portable, without seeming to demean **perfectly useful C programs that happen not to be portable**, thus the adverb strictly.

The C standard gives implementors a lot of freedom:

- ▶ The compiler only has to preserve **observable behavior**.
- ▶ Constraint violations require a diagnostic only.
- ▶ A compiler can translate an invalid program (but has to emit required diagnostics).
- ▶ There is unspecified, implementation-defined, and undefined behavior.

⇒ The C standard formulates **minimal requirements** for common subset of C dialects (such as GNU C) and is not a complete specification for a single language.

Additional specifications constrain a C compiler:
ABI, POSIX, backwards compatibility, own standards.

ABI requirements in combination with separate compilation implies many hard constraints on how individual functions are compiled.

Multithreading requires a memory model (added in C11) that also constrains the compiler (e.g. it can not invent writes to variables not written to in the source).

Undefined Behavior

C23, 3.5.3

§1 **undefined behavior**

behavior, upon use of a **nonportable or erroneous** program construct or of erroneous data, for which this document imposes no requirements.

What is undefined behavior

Undefined behavior is various use cases:

- ▶ Extensions
- ▶ Safety features (segmentations, capabilities, Itanium NoT, etc.)
- ▶ Allow difficult to detect errors to be ignored
- ▶ Optimization

C99Rationale: Undefined behavior gives the implementor license not to catch certain program errors that are difficult to diagnose. It also identifies areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior.

Nasal Demons

“When the compiler encounters it is legal for it to make demons fly out of your nose”
comp.std.c, 1992

“The C and C++ languages are notorious for the very large section of the map labeled here be dragons, or more formally, undefined behavior.” – Raymond Chen

“Undefined behavior exists in C-based languages because the designers of C wanted it to be an extremely efficient low-level programming language.” – Chris Lattner

What Every C Programmer Should Know About Undefined Behavior, Chris Lattner, May 13, 2011
<https://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>

No Semantics

People working on programming language semantics started to treat the standard as a specification and programs with UB are considered invalid and have "no semantics".

C23, 3.5.3

undefined behavior

behavior, upon use of a **nonportable or erroneous** program construct or of erroneous data, **for which** this document imposes no requirements.

C++23, 4.1.2

undefined behavior

behavior for which this document imposes no requirements ...

[Note 1 : This document imposes no requirements on the behavior of programs that contain undefined behavior. — end note]

Cerberus Survey

There is wide disagreement between programmers, compiler writers, and the standard experts about what is allowed in C.

How predictable are reads from padding bytes?

If you zero all bytes of a struct and then write some of its members, do reads of the padding return zero?
(e.g. for a bitwise CAS or hash of the struct, or to know that no security-relevant data has leaked into them.)

Will that work in normal C compilers?

yes : 116 (36%)
only sometimes : 95 (29%)
no : 21 (6%)
don't know : 82 (25%)
I don't know what the question is asking : 3 (1%)
no response : 6

Kayvan Memarian and Peter Sewell, What is C in practice? 2015.

<https://www.cl.cam.ac.uk/~pes20/cerberus/notes50-survey-discussion.html>

Time Travel

"Undefined behavior can result in time travel (among other things, but time travel is the funkiest)" – Raymond Chen

```
extern int x;  
  
int f(int a, int b)  
{  
    x = b ? 42 : 43;  
    return a / b;  
}
```



BBC, CC BY 3.0

<https://devblogs.microsoft.com/oldnewthing/20140627-00/?p=633>

Time Travel: Example 1

extern int x;	
int f(int a, int b)	mov eax, edi
{	mov rcx, qword ptr [rip + x]
x = b ? 42 : 43;	mov dword ptr [rcx], 42
return a / b;	cdq
}	idiv esi
	ret

Question: Dividing by zero has undefined behavior. Can the compiler assume $b \neq 0$ and directly store the value 42?

Answer:

Time Travel: Example 1

extern int x;	
int f(int a, int b)	mov eax, edi
{	mov rcx, qword ptr [rip + x]
x = b ? 42 : 43;	mov dword ptr [rcx], 42
return a / b;	cdq
}	idiv esi
	ret

Question: Dividing by zero has undefined behavior. Can the compiler assume $b \neq 0$ and directly store the value 42?

Answer:

Time Travel: Example 1

extern int x;	
int f(int a, int b)	mov eax, edi
{	mov rcx, qword ptr [rip + x]
x = b ? 42 : 43;	mov dword ptr [rcx], 42
return a / b;	cdq
}	idiv esi
	ret

Question: Dividing by zero has undefined behavior. Can the compiler assume $b \neq 0$ and directly store the value 42?

Answer: Yes. The store is not observable behavior.

Time Travel: Example 2

```
extern void g(int x);
```

```
int f(int a, int b)
{
    g(b ? 42 : 43);
    return a / b;
}
```

Question: Dividing by zero has undefined behavior. Can the compiler assume $b \neq 0$ and directly pass 42 to the function?

MSVC bug: <https://devblogs.microsoft.com/oldnewthing/20241104-00/?p=110466>

Old GCC bug: https://bugzilla.redhat.com/show_bug.cgi?id=520916

Time Travel: Example 2

```
extern void g(int x);
```

```
int f(int a, int b)
{
    g(b ? 42 : 43);
    return a / b;
}
```

Question: Dividing by zero has undefined behavior. Can the compiler assume $b \neq 0$ and directly pass 42 to the function?

```
void g(int x) { if (x == 43) exit(0); }
```

MSVC bug: <https://devblogs.microsoft.com/oldnewthing/20241104-00/?p=110466>

Old GCC bug: https://bugzilla.redhat.com/show_bug.cgi?id=520916

Time Travel: Example 3

```
volatile int x;  
  
int foo(int a, int b, bool store_to_x)  
{  
    if (!store_to_x) {  
        return a / b;  
    }  
    x = b;  
    return a / b;  
}
```

Question: Can the compiler hoist the division above the volatile store?

Time Travel

Sorry, time travel is **not** allowed! In C23, we added the following clarification:

3.5.3 §4 Note 3 to entry: Any other behavior during execution of a program is only affected as a direct consequence of the concrete behavior that occurs when encountering the erroneous or non-portable program construct or data. In particular, all observable behavior (5.1.2.4) appears as specified in this document when it happens before an operation with undefined behavior in the execution of the program.

[LangRef] Make volatile loads non-willreturn, Merged 2026-04-23.

<https://github.com/llvm/llvm-project/pull/192992>

Reodering of potentially trapping operations and volatile stores. Confirmed 2026-09-14.

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=104800

No Time Travel in C++

In C++ 26, you need to add an observable checkpoint.

```
volatile int x;  
int foo(int a, int b, bool store_to_x)  
{  
    if (!store_to_x) {  
        return a / b;  
    }  
    x = b;  
    std::observable_checkpoint();  
    return a / b;  
}
```

Where the standard seems clear, but compilers sometimes have a different opinion:

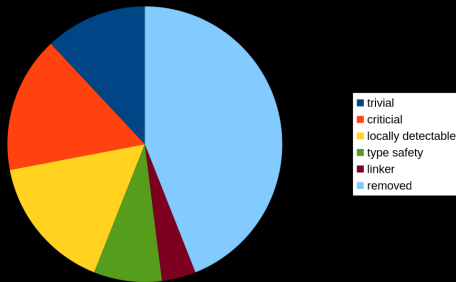
- ▶ Reading uninitialized values is defined. (except for automatic variables whose address is not taken and when reading non-value representations)
- ▶ Reading an uninitialized object produces the same result each time.
(an allocator using `MADV_FREE` is **not** conforming)
- ▶ Equality comparison of pointers is always defined.
(GCC / clang miscompile this in specific cases)
- ▶ Round-trips via integers are defined.
(GCC / clang miscompile this in specific cases)
- ▶ Type-changing writes to allocated storage are defined.
(clang is not conforming without `-fno-strict-aliasing`)

To address these issues, the C committee has three study groups.

- ▶ Memory Object Model Study Group
- ▶ Undefined Behavior Study Group
- ▶ Memory Safety Study Group

Demon Slaying

C23 has about 100 (presumed) UBs in the core language.
In the C2y draft, 45 of those were removed - so far.



<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3128.pdf>

Tools

Finding issues:

- ▶ Compilers warnings
- ▶ Static analyzers
- ▶ Sanitizers
- ▶ AI tools
- ▶ Formal verification

Hardening/mitigation techniques:

- ▶ Hardened booleans
- ▶ Control-flow integrity
- ▶ Type-aware allocators
- ▶ Stack canaries

Compiler Warnings

```
enum {N = 32};  
int a[N], pfx[N];  
void prefix_sum (void)  
{  
    int i, accum;  
    for (i = 0, accum = a[0]; i < N; i++, accum += a[i])  
        pfx[i] = accum;  
}
```

John Regehr, Undefined Behavior Consequences Contest Winners, 2012.
<https://blog.regehr.org/archives/767>

Martin Uecker, Revisiting Two Examples of Undefined Behavior, 2025.
<https://uecker.codeberg.page/2025-09-29.html>

Compiler Warnings

```
enum {N = 32};  
int a[N], pfx[N];  
void prefix_sum (void)  
{  
    int i, accum;  
    for (i = 0, accum = a[0]; i < N; i++, accum += a[i])  
        pfx[i] = accum;  
}
```

```
      : In function 'prefix_sum':  
:9:51: warning: iteration 31 invokes undefined behavior [-Waggressive-loop-ops-limit=]  
9 |     for (i = 0, accum = a[0]; i < N; i++, accum += a[i])
```

John Regehr, Undefined Behavior Consequences Contest Winners, 2012.
<https://blog.regehr.org/archives/767>

Martin Uecker, Revisiting Two Examples of Undefined Behavior, 2025.
<https://uecker.codeberg.page/2025-09-29.html>

```
int main() {  
    int *p = malloc(sizeof(int));  
    int *q = realloc(p, sizeof(int));  
    *p = 1;  
    *q = 2;  
    if (p == q)  
        printf("%d-%d\n", *p, *q);  
}
```

John Regehr, Undefined Behavior Consequences Contest Winners, 2012.
<https://blog.regehr.org/archives/767>

Martin Uecker, Revisiting Two Examples of Undefined Behavior, 2025.
<https://uecker.codeberg.page/2025-09-29.html>

```

int main() {
    int *p = malloc(sizeof(int));
    int *q = realloc(p, sizeof(int));
    *p = 1;
    *q = 2;
    if (p == q)
        printf("%d-%d\n", *p, *q);
}

```

```

:11:5: warning: pointer 'p' may be used after 'realloc' [-Wuse-after-free]

```

```

11 |         printf("%d-%d\n", *p, *q);
    |         ^~~~~~

```

```

:8:6: warning: pointer 'p' used after 'realloc' [-Wuse-after-free]

```

```

8 |     *p = 1;
  |     ~~~~

```

John Regehr, Undefined Behavior Consequences Contest Winners, 2012.
<https://blog.regehr.org/archives/767>

Martin Uecker, Revisiting Two Examples of Undefined Behavior, 2025.
<https://uecker.codeberg.page/2025-09-29.html>

Static Analyzers

External tools and now also part of GCC / Clang.

```
gcc -fanalyzer
```

```
void test(void)
{
    char buf[10];
    strcpy (buf, "hello");
    strcat (buf, "-world!");
}
```

```
<source>: In function 'test':
```

```
<source>:8:3: warning: stack-based buffer overflow [CWE-121] [-Wanalyzer-out-of-bounds]
```

```
8 |     strcat (buf, " world!");
  |     ~~~~~~
```

David Malcom, Improvements to static analysis in the GCC 14 compiler, 2024.

<https://developers.redhat.com/articles/2024/04/03/>

improvements-static-analysis-gcc-14-compiler

Sanitizers

Sanitizers add run-time checks.

```
volatile uint8_t start = -1;  
uint64_t result = start << 24; // undefined
```

```
runtime error: left shift of 255 by 24 places cannot be represented in type 'int'
```

Some sanitizers can also be used for hardening when used in trapping mode.

Memory Safety

Can we achieve memory safety?

- ▶ Type safety
- ▶ Spatial memory safety
- ▶ Temporal memory safety

Type Safety

C generally has a **strong type** system.

Remaining issues are fixable:

- ▶ Tagless unions $\Rightarrow$ new annotation
- ▶ It allows unsafe downcasts from void (or array types of unknown length)
 $\Rightarrow$ new diagnostics
- ▶ Cross translation-unit type checking
- ▶ String literals are not const qualified.

Spatial Memory Safety

- ▶ Array bounds
- ▶ Arbitrary pointers

Spatial Memory Safety: Array Bounds

For (pointers to) arrays, a compiler can create bounds-checked code today.

```
char a[10];

char *p = a;           // bound is lost in the type

int N = _Countof a;
char (*q)[N] = &a;     // bound is preserved in the type

p[10] = 0;             // ?
(*q)[10] = 0;          // array bound checked with -fsanitize=bounds
```

Note: Bound can be variable (variably modified types).

Spatial Memory Safety: Array Bounds

This also works for pointers passed via functions...

```
int foo(int n, char (*buf)[n], int i)
{
    return (*buf)[i];
}
```

One missing language feature: No way to store vm-type in structures. (can be worked around by macros)

Spatial Memory Safety: Array Bounds

GCC's dynamic object size pass or Clang's bounds safety extension for regular pointers.

```
int a[10];

int *p = a;
p[10] = a;           // object size checked with -fsanitize=object-size

int foo(int n, char buf[n])
{
    buf[10] = 1;      // object size checked with -fsanitize=object-size
}
```

Spatial Memory Safety: Attributes

Annotations (gnu::access, clang,gnu::counted_by, ...) bridge API boundaries.

```
int foo(int n, char *buf)
    [[gnu::access(read_write, 1, 2)]]
{
    buf[10] = 1;    // object size checked with -fsanitize=object-size
}

struct bar { int n; char *buf [[gnu::counted_by(n)]]; };

int foo(struct bar *b)
{
    b->buf[10] = 1; // object size checked with -fsanitize=object-size
}
```

Temporal Memory Safety

Run-time checking of largely unmodified C programs.

Capability Hardware Enhanced RISC Instructions (CHERI)

Fil-C uses capabilities in shadow memory and concurrent garbage collection. It can run existing code with zero or minimal modifications.

(possibly because of undefined behavior related to pointer provenance!)

<https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/> <https://fil-c.org/>

Formal Verification

Formal verification can ensure memory safety statically.

```
> cbmc sudoku.c
...
[sudoku_r.undefined-shift.13] line 77 shift distance is negative in 1 << n: SUCCESS
[sudoku_r.undefined-shift.14] line 77 shift distance too large in 1 << n: SUCCESS
[sudoku_r.array_bounds.3] line 79 array dynamic object upper bound in board[(signed long int)i][(signed long int)j]: SUCCESS
[sudoku_r.pointer_dereference.67] line 79 dereference failure: pointer NULL in board[(signed long int)i]: SUCCESS
[sudoku_r.pointer_dereference.68] line 79 dereference failure: pointer invalid in board[(signed long int)i]: SUCCESS
[sudoku_r.pointer_dereference.69] line 79 dereference failure: deallocated dynamic object in board[(signed long int)i]: SUCCESS
[sudoku_r.pointer_dereference.70] line 79 dereference failure: dead object in board[(signed long int)i]: SUCCESS
[sudoku_r.pointer_dereference.71] line 79 dereference failure: pointer outside object bounds in board[(signed long int)i]: SUCCESS
[sudoku_r.pointer_dereference.72] line 79 dereference failure: invalid integer address in board[(signed long int)i]: SUCCESS

** 0 of 134 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

Advances in SAT/SMT solver have made such techniques more powerful, but still limited when applied to existing programs.

<https://www.cprover.org/cbmc/>

Formal Verification - Light

A hack to detect undefined behavior statically in small examples: Use the sanitizer to insert traps and check which traps remain after optimization.

```
gcc -fsanitize=shift -fsanitize-trap -O2
```

```
[[gnu::error("UNSAFE")]] void __builtin_trap();
```

```
volatile uint8_t start = -1;
```

```
uint64_t result = start << 24; // undefined behavior
```

```
<source>:15:29: error: call to '__builtin_trap' declared with attribute error: UNSAFE
    15 |         uint64_t result = start << 24;
```

(Adding the right cast fixes this issue and the error disappears.)

Can we achieve full memory safety?

Today:

- ▶ Expensive run-time checking
- ▶ Formal verification

Outlook:

Static verification for a restrict language in combination with new annotations.

The Committee

ISO/IEC JTC1/SC22/WG14

Meetings 2-3 times a year for a week (hybrid).

One can join via a national body (rules vary).

Proposal driven. Non-members can submit proposals!

Feedback is appreciated.

Homepage: <https://www.open-std.org/jtc1/sc22/wg14/>



C - Contributing

2024-06-14: [home](#) | [projects](#) | [documents](#) | [issues](#) | [contributing](#) | [meetings](#) | [contacts](#)

Contributing to WG14

WG14 serves a broad community of programmers, tooling and hardware vendors, book authors, and focuses on how best to interact with the committee for some common activities, such as requesting a new standard.

Submitting a Clarification Request

Ensuring a technical document is precise and readable is a difficult job, and we don't always

- ▶ C2y ongoing work...
- ▶ C23 ISO/IEC 9899:2024
- ▶ C17 ISO/IEC 9899:2018
- ▶ C11 ISO/IEC 9899:2011
- ▶ C99 ISO/IEC 9899:1999
- ▶ C95 ISO/IEC 9899:1990/Amd 1:1995
- ▶ C90 ISO/IEC 9899:1990
- ▶ C89 ANSI X3.159-1989

The current C standard is C23. Older standards are withdrawn.

- ▶ K&R function definitions **removed**
- ▶ one's complement or signed-magnitude **removed**
- ▶ trigraphs **removed**
- ▶ variably modified types **required**
- ▶ —typeof, auto, true, false, constexpr, nullptr—
- ▶ empty initialization: —struct foo f = { };—
- ▶ checked integer operations: —if (ckd_mul(&result, a, b)) ...—
- ▶ if declarations: —if (int i = ...) ...—
- ▶ tag compatibility
- ▶ unnamed function arguments: —int callback(int a, void *);—
- ▶ labels before declaration / end of function
- ▶ bit-precise integer types: —_BitInt(N)—

Outlook C2y

- ▶ case ranges: `case 1 ... 5:`

- ▶ named loops:

```
foo: for (int i = 0; i < 10; i++) {  
    ...  
    break foo;  
    ...  
}
```

- ▶ array length operator: `int arr[10]; size_t len = _Countof(arr);`

- ▶ `static_assert` in expressions

- ▶ bit operations

- ▶ **demon slaying**

Conclusion

- ▶ C is still evolving and tools are improving.
- ▶ Full memory safety is possible and will become practical.
- ▶ You can help!