C and C++ Prioritize Performance over Correctness

Russ Cox
August 18, 2023

research.swtch.com/ub

The original ANSI C standard, C89, introduced the concept of "undefined behavior," which was used both to describe the effect of outright bugs like accessing memory in a freed object and also to capture the fact that existing implementations differed about handling certain aspects of the language, including use of uninitialized values, signed integer overflow, and null pointer handling.

The C89 spec defined undefined behavior (in section 1.6) as:

> Undefined behavior—behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately-valued objects, for which the Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

Lumping both non-portable and buggy code into the same category was a mistake. As time has gone on, the way compilers treat undefined behavior has led to more and more unexpectedly broken programs, to the point where it is becoming difficult to tell whether any program will compile to the meaning in the original source. This post looks at a few examples and then tries to make some general observations. In particular, today's C and C++ prioritize performance to the clear detriment of correctness.

Uninitialized variables

C and C++ do not require variables to be initialized on declaration (explicitly or implicitly) like Go and Java. Reading from an uninitialized variable is undefined behavior.

In a blog post*, Chris Lattner (creator of LLVM and Clang) explains the rationale:

> **Use of an uninitialized variable**: This is commonly known as source of problems in C programs and there are many tools to catch these: from compiler warnings to static and dynamic analyzers. This improves performance by not requiring that all variables be zero initialized when they come into scope (as Java does). For most scalar variables, this would cause little overhead, but stack arrays and malloc'd memory would incur a memset of the storage, which could be quite costly, particularly since the storage is usually completely overwritten.

Early C compilers were too crude to detect use of uninitialized basic variables like integers and pointers, but modern compilers are dramatically more sophisticated. They could absolutely react in these cases by "terminating a translation or execution (with the issuance of a diagnostic message)," which is to say reporting a compile error. Or, if they were worried about not rejecting old programs, they could insert a zero initialization with, as Lattner admits, little overhead. But they don't do either of these. Instead, they just do whatever they feel like during code generation.

For example, here's a simple C++ program with an uninitialized variable (a bug):

```
#include <stdio.h>

int main() {
    for(int i; i < 10; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

If you compile this with `clang++ -O1`, it deletes the loop entirely: `main` contains only the `return 0`. In effect, Clang has noticed the uninitialized variable and chosen not to report the error to the user but instead to pretend `i` is always initialized above 10, making the loop disappear.

It is true that if you compile with `-Wall`, then Clang does report the use of the uninitialized variable as a warning. This is why you should always build with and fix warnings in C and C++ programs. But not all compiler-optimized undefined behaviors are reliably reported as warnings.

Arithmetic overflow

At the time C89 was standardized, there were still legacy ones'-complement computers*, so ANSI C could not assume the now-standard two's-complement representation for negative numbers. In two's complement, an `int8` −1 is 0b11111111; in ones' complement that's −0, while −1 is 0b11111110. This meant that operations like signed integer overflow could not be defined, because

`int8` 127+1 = 0b01111111+1 = 0b10000000

is −127 in ones' complement but −128 in two's complement. That is, signed integer overflow was non-portable. Declaring it undefined behavior let compilers escalate the behavior from "non-portable", with one of two clear meanings, to whatever they feel like doing. For example, a common thing programmers expect is that you can test for signed integer overflow by checking whether the result is less than one of the operands, as in this program:

```
#include <stdio.h>

int f(int x) {
    if(x+100 < x)
        printf("overflow\n");
    return x+100;
}
```

Clang optimizes away the `if` statement. The justification is that since signed integer overflow is undefined behavior, the compiler can assume it never happens, so `x+100` must never be less than `x`. Ironically, this program would correctly detect overflow on both ones'-complement and two's-complement machines if the compiler would actually emit the check.

In this case, `clang++ -O1 -Wall` prints no warning while it deletes the `if` statement, and neither does `g++`, although I seem to remember it used to, perhaps in subtly different situations or with different flags.

For C++20, the first version of proposal P0907* suggested standardizing that signed integer overflow wraps in two's complement. The original draft gave a very clear statement of the history of the undefined behavior and the motivation for making a change:

[C11] Integer types allows three representations for signed integral types:

- Signed magnitude
- Ones' complement
- Two's complement

See §4 C Signed Integer Wording for full wording.

C++ inherits these three signed integer representations from C. To the author's knowledge no modern machine uses both C++ and a signed integer representation other than two's complement (see §5 Survey of Signed Integer Representations). None of [MSVC], [GCC], and [LLVM] support other representations. This means that the C++ that is taught is effectively two's complement, and the C++ that is written is two's complement. It is extremely unlikely that there exist any significant code base developed for two's complement machines that would actually work when run on a non-two's complement machine.

The C++ that is spec'd, however, is not two's complement. Signed integers currently allow for trap representations, extra padding bits, integral negative zero, and introduce undefined behavior and implementation-defined behavior for the sake of this extremely abstract machine.

Specifically, the current wording has the following effects:

- Associativity and commutativity of integers is needlessly obtuse.
- Naïve overflow checks, which are often security-critical, often get eliminated by compilers. This leads to exploitable code when the intent was clearly not to and the code, while naïve, was correctly performing security checks for two's complement integers. Correct overflow checks are difficult to write and equally difficult to read, exponentially so in generic code.
- Conversion between signed and unsigned are implementation-defined.
- There is no portable way to generate an arithmetic right-shift, or to sign-extend an integer, which every modern CPU supports.
- constexpr is further restrained by this extraneous undefined behavior.
- Atomic integral are already two's complement and have no undefined results, therefore even freestanding implementations already support two's complement in C++.

Let's stop pretending that the C++ abstract machine should represent integers as signed magnitude or ones' complement. These theoretical implementations are a different programming language, not our real-world C++. Users of C++ who require signed magnitude or ones' complement integers would be better served by a pure-library solution, and so would the rest of us.

In the end, the C++ standards committee put up "strong resistance against" the idea of defining signed integer overflow the way every programmer expects; the undefined behavior remains.

Infinite loops

A programmer would never accidentally cause a program to execute an infinite loop, would they? Consider this program:

```
#include <stdio.h>

int stop = 1;

void maybeStop() {
    if(stop)
        for(;;);
}

int main() {
    printf("hello, ");
    maybeStop();
    printf("world\n");
}
```

This seems like a completely reasonable program to write. Perhaps you are debugging and want the program to stop so you can attach a debugger. Changing the initializer for `stop` to `0` lets the program run to completion. But it turns out that, at least with the latest Clang, the program runs to completion anyway: the call to `maybeStop` is optimized away entirely, even when `stop` is `1`.

The problem is that C++ defines that every side-effect-free loop may be assumed by the compiler to terminate. That is, a loop that does not terminate is therefore undefined behavior. This is purely for compiler optimizations, once again treated as more important than correctness. The rationale for this decision played out in the C standard and was more or less adopted in the C++ standard as well.

John Regehr pointed out this problem in his post "C Compilers Disprove Fermat's Last Theorem*," which included this entry in a FAQ:

> Q: Does the C standard permit/forbid the compiler to terminate infinite loops?
>
> A: The compiler is given considerable freedom in how it implements the C program, but its output must have the same externally visible behavior that the program would have when interpreted by the "C abstract machine" that is described in the standard. Many knowledgeable people (including me) read this as saying that the termination behavior of a program must not be changed. Obviously some compiler writers disagree, or else don't believe that it matters. The fact that reasonable people disagree on the interpretation would seem to indicate that the C standard is flawed.

A few months later, Douglas Walls wrote WG14/N1509: Optimizing away infinite loops*, making the case that the standard should *not* allow this optimization. In response, Hans-J. Boehm wrote WG14/N1528: Why undefined behavior for infinite loops?*, arguing for allowing the optimization.

Consider the potential optimization of this code:

```
for (p = q; p != 0; p = p->next)
    ++count;
for (p = q; p != 0; p = p->next)
    ++count2;
```

A sufficiently smart compiler might reduce it to this code:

```
for (p = q; p != 0; p = p->next) {
        ++count;
        ++count2;
}
```

Is that safe? Not if the first loop is an infinite loop. If the list at `p` is cyclic and another thread is modifying `count2`, then the first program has no race, while the second program does. Compilers clearly can't turn correct, race-free programs into racy programs. But what if we declare that infinite loops are not correct programs? That is, what if infinite loops were undefined behavior? Then the compiler could optimize to its robotic heart's content. This is exactly what the C standards committee decided to do.

The rationale, paraphrased, was:

- It is very difficult to tell if a given loop is infinite.
- Infinite loops are rare and typically unintentional.
- There are many loop optimizations that are only valid for non-infinite loops.
- The performance wins of these optimizations are deemed important.
- Some compilers already apply these optimizations, making infinite loops non-portable too.
- Therefore, we should declare programs with infinite loops undefined behavior, enabling the optimizations.

Null pointer usage

We've all seen how dereferencing a null pointer causes a crash on modern operating systems: they leave page zero unmapped by default precisely for this purpose. But not all systems where C and C++ run have hardware memory protection. For example, I wrote my first C and C++ programs using Turbo C on an MS-DOS system. Reading or writing a null pointer did not cause any kind of fault: the program just touched the memory at location zero and kept running. The correctness of my code improved dramatically when I moved to a Unix system that made those programs crash at the moment of the mistake. Because the behavior is non-portable, though, dereferencing a null pointer is undefined behavior.

At some point, the justification for keeping the undefined behavior became performance. Chris Lattner explains*:

> In C-based languages, NULL being undefined enables a large number of simple scalar optimizations that are exposed as a result of macro expansion and inlining.

In an earlier post*, I showed this example, lifted from Twitter in 2017*:

```
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
    return system("rm -rf slash");
}
```

```
void NeverCalled() {
    Do = EraseAll;
}

int main() {
    return Do();
}
```

Because calling `Do()` is undefined behavior when `Do` is null, a modern C++ compiler like Clang simply assumes that can't possibly be what's happening in `main`. Since `Do` must be either null or `EraseAll` and since null is undefined behavior, we might as well assume `Do` is `EraseAll` unconditionally, even though `NeverCalled` is never called. So this program can be (and is) optimized to:

```
int main() {
    return system("rm -rf slash");
}
```

Lattner gives an equivalent example* (search for `FP()`) and then this advice:

> The upshot is that it is a fixable issue: if you suspect something weird is going on like this, try building at -O0, where the compiler is much less likely to be doing any optimizations at all.

This advice is not uncommon: if you cannot debug the correctness problems in your C++ program, disable optimizations.

Crashes out of sorts

C++'s `std::sort` sorts a collection of values (abstracted as a random access iterator, but almost always an array) according to a user-specified comparison function. The default function is `operator<`, but you can write any function. For example if you were sorting instances of class `Person` your comparison function might sort by the `LastName` field, breaking ties with the `FirstName` field. These comparison functions end up being subtle yet boring to write, and it's easy to make a mistake. If you do make a mistake and pass in a comparison function that returns inconsistent results or accidentally reports that any value is less than itself, that's undefined behavior: `std::sort` is now allowed to do whatever it likes, including walking off either end of the array and corrupting other memory. If you're lucky, it will pass some of this memory to your comparison function, and since it won't have pointers in the right places, your comparison function will crash. Then at least you have a chance of guessing the comparison function is at fault. In the worst case, memory is silently corrupted and the crash happens much later, with `std::sort` is nowhere to be found.

Programmers make mistakes, and when they do, `std::sort` corupts memory. This is not hypothetical. It happens enough in practice to be a popular question on StackOverflow*.

As a final note, it turns out that `operator<` is not a valid comparison function on floating-point numbers if NaNs are involved, because:

- 1 < NaN and NaN < 1 are both false, implying NaN == 1.
- 2 < NaN and NaN < 2 are both false, implying NaN == 2.
- Since NaN == 1 and NaN == 2, 1 == 2, yet 1 < 2 is true.

Programming with NaNs is never pleasant, but it seems particularly extreme to allow `std::sort` to crash when handed one.

Reflections and revealed preferences

Looking over these examples, it could not be more obvious that in modern C and C++, performance is job one and correctness is job two. To a C/C++ compiler, a programmer making a mistake and (gasp!) compiling a program containing a bug is just not a concern. Rather than have the compiler point out the bug or at least compile the code in a clear, understandable, debuggable manner, the approach over and over again is to let the compiler do whatever it likes, in the name of performance.

This may not be the wrong decision for these languages. There are undeniably power users for whom every last bit of performance translates to very large sums of money, and I don't claim to know how to satisfy them otherwise. On the other hand, this performance comes at a significant development cost, and there are probably plenty of people and companies who spend more than their performance savings on unnecessarily difficult debugging sessions and additional testing and sanitizing. It also seems like there must be a middle ground where programmers retain most of the control they have in C and C++ but the program doesn't crash when sorting NaNs or behave arbitrarily badly if you accidentally dereference a null pointer. Whatever the merits, it is important to see clearly the choice that C and C++ are making.

In the case of arithmetic overflow, later drafts of the proposal removed the defined behavior for wrapping, explaining:

> The main change between [P0907r0] and the subsequent revision is to maintain undefined behavior when signed integer overflow occurs, instead of defining wrapping behavior. This direction was motivated by:
>
> – Performance concerns, whereby defining the behavior prevents optimizers from assuming that overflow never occurs;
>
> – Implementation leeway for tools such as sanitizers;
>
> – Data from Google suggesting that over 90% of all overflow is a bug, and defining wrapping behavior would not have solved the bug.

Again, performance concerns rank first. I find the third item in the list particularly telling. I've known C/C++ compiler authors who got excited about a 0.1% performance improvement, and incredibly excited about 1%. Yet here we have an idea that would change 10% of affected programs from incorrect to correct, and it is rejected, because performance is more important.

The argument about sanitizers is more nuanced. Leaving a behavior undefined allows any implementation at all, including reporting the behavior at runtime and stopping the program. True, the widespread use of undefined behavior enables sanitizers like ThreadSanitizer, MemorySanitizer, and UBSan, but so would defining the behavior as "either this specific behavior, or a sanitizer report." If you believed correctness was job one, you could define overflow to wrap, fixing the 10% of programs outright and making the 90% behave at least more predictably, and then at the same time define that overflow is still a bug that can be reported by sanitizers. You might object that requiring wrapping in the absence of a sanitizer would hurt performance, and that's fine: it's just more evidence that performance trumps correctness.

One thing I find surprising, though, is that correctness gets ignored even when it clearly doesn't hurt performance. It would certainly not hurt performance to emit a compiler warning about deleting the `if` statement testing for signed overflow, or about optimizing away the possible null pointer dereference in `Do()`. Yet I could find no way to make compilers report either one; certainly

not `-Wall`.

The explanatory shift from non-portable to optimizable also seems revealing. As far as I can tell, C89 did not use performance as a justification for any of its undefined behaviors. They were non-portabilities, like signed overflow and null pointer dereferences, or they were outright bugs, like use-after-free. But now experts like Chris Lattner and Hans Boehm point to optimization potential, not portability, as justification for undefined behaviors. I conclude that the rationales really have shifted from the mid-1980s to today: an idea that meant to capture non-portability has been preserved for performance, trumping concerns like correctness and debuggability.

Occasionally in Go we have changed library functions to remove surprising behavior*, It's always a difficult decision, but we are willing to break existing programs depending on a mistake if correcting the mistake fixes a much larger number of programs. I find it striking that the C and C++ standards committees are willing in some cases to break existing programs if doing so merely *speeds up* a large number of programs. This is exactly what happened with the infinite loops.

I find the infinite loop example telling for a second reason: it shows clearly the escalation from non-portable to optimizable. In fact, it would appear that if you want to break C++ programs in service of optimization, one possible approach is to just do that in a compiler and wait for the standards committee to notice. The de facto non-portability of whatever programs you have broken can then serve as justification for undefining their behavior, leading to a future version of the standard in which your optimization is legal. In the process, programmers have been handed yet another footgun to try to avoid setting off.

(A common counterargument is that the standards committee cannot force existing implementations to change their compilers. This doesn't hold up to scrutiny: every new feature that gets added is the standards committee forcing existing implementations to change their compilers.)

I am not claiming that anything should change about C and C++. I just want people to recognize that the current versions of these sacrifice correctness for performance. To some extent, all languages do this: there is almost always a tradeoff between performance and slower, safer implementations. Go has data races in part for performance reasons: we could have done everything by message copying or with a single global lock instead, but the performance wins of shared memory were too large to pass up. For C and C++, though, it seems no performance win is too small to trade against correctness.

As a programmer, you have a tradeoff to make too, and the language standards make it clear which side they are on. In some contexts, performance is the dominant priority and nothing else matters quite as much. If so, C or C++ may be the right tool for you. But in most contexts, the balance flips the other way. If programmer productivity, debuggability, reproducible bugs, and overall correctness and understandability are more important than squeezing every last little bit of performance, then C and C++ are not the right tools for you. I say this with some regret, as I spent many years happily writing C programs.

I have tried to avoid exaggerated, hyperbolic language in this post, instead laying out the tradeoff and the preferences revealed by the decisions being made. John Regehr wrote a less restrained series of posts about undefined behavior a decade ago, and in one of them* he concluded:

> It is basically evil to make certain program actions wrong, but to not give developers any way to tell whether or not their code performs these actions and, if so, where. One of C's design points was "trust the programmer." This is fine, but there's trust and then there's trust. I mean, I trust my 5 year old but I still don't let him cross a busy street

> by himself. Creating a large piece of safety-critical or security-critical code in C or C++ is the programming equivalent of crossing an 8-lane freeway blindfolded.

To be fair to C and C++, if you set yourself the goal of crossing an 8-lane freeway blindfolded, it does make sense to focus on doing it as fast as you possibly can.

* Asterisks mark hyperlinked text.