A Guide to Undefined Behavior in C and C++, Part 1

Also see Part 2 and Part 3.

Programming languages typically make a distinction between normal program actions and erroneous actions. For Turing-complete languages we cannot reliably decide offline whether a program has the potential to execute an error; we have to just run it and see.

In a *safe* programming language, errors are trapped as they happen. Java, for example, is largely safe via its exception system. In an *unsafe* programming language, errors are not trapped. Rather, after executing an erroneous operation the program keeps going, but in a silently faulty way that may have observable consequences later on. Luca Cardelli's article on type systems has a nice clear introduction to these issues. C and C++ are unsafe in a strong sense: executing an erroneous operation causes the entire program to be meaningless, as opposed to just the erroneous operation having an unpredictable result. In these languages erroneous operations are said to have *undefined behavior*.

The C FAQ defines "undefined behavior" like this:

Anything at all can happen; the Standard imposes no requirements. The program may fail to compile, or it may execute incorrectly (either crashing or silently generating incorrect results), or it may fortuitously do exactly what the programmer intended.

This is a good summary. Pretty much every C and C++ programmer understands that accessing a null pointer and dividing by zero are erroneous actions. On the other hand, the full implications of undefined behavior and its interactions with aggressive compilers are not well-appreciated. This post explores these topics.

A Model for Undefined Behavior

For now, we can ignore the existence of compilers. There is only the "C implementation" which — if the implementation conforms to the C standard — acts the same as the "C abstract machine" when executing a conforming program. The C abstract machine is a simple interpreter for C that is described in the C standard. We can use it to determine the meaning of any C program.

The execution of a program consists of simple steps such as adding two numbers or jumping to a label. If every step in the execution of a program has defined behavior, then the entire execution is well-defined. Note that even well-defined executions may not have a unique result due to unspecified and implementation-defined behavior; we'll ignore both of these here.

If any step in a program's execution has undefined behavior, then the entire execution is without meaning. This is important: it's not that evaluating (1<<32)

has an unpredictable result, but rather that the entire execution of a program that evaluates this expression is meaningless. Also, it's not that the execution is meaningful up to the point where undefined behavior happens: the bad effects can actually precede the undefined operation.

As a quick example let's take this program:

```
#include <limits.h>
#include <stdio.h>

int main (void)
{
  printf ("%d\n", (INT_MAX+1) < 0);
  return 0;
}
```

The program is asking the C implementation to answer a simple question: if we add one to the largest representable integer, is the result negative? This is perfectly legal behavior for a C implementation:

```
$ cc test.c -o test
$ ./test
1
```

So is this:

```
$ cc test.c -o test
$ ./test
0
```

And this:

```
$ cc test.c -o test
$ ./test
42
```

And this:

```
$ cc test.c -o test
$ ./test
Formatting root partition, chomp chomp
```

One might say: Some of these compilers are behaving improperly because the C standard says a relational operator must return 0 or 1. But since the program has no meaning at all, the implementation can do whatever it likes. Undefined behavior trumps all other behaviors of the C abstract machine.

Will a real compiler emit code to chomp your disk? Of course not, but keep in mind that practically speaking, undefined behavior often does lead to Bad Things because many security vulnerabilities start out as memory or integer operations that have undefined behavior. For example, accessing an out of bounds array

element is a key part of the canonical stack smashing attack. In summary: the compiler does not need to emit code to format your disk. Rather, following the OOB array access your computer will begin executing exploit code, and that code is what will format your disk.

No Traveling

It is very common for people to say — or at least think — something like this:

The x86 ADD instruction is used to implement C's signed add operation, and it has two's complement behavior when the result overflows. I'm developing for an x86 platform, so I should be able to expect two's complement semantics when 32-bit signed integers overflow.

THIS IS WRONG. You are saying something like this:

Somebody once told me that in basketball you can't hold the ball and run. I got a basketball and tried it and it worked just fine. He obviously didn't understand basketball.

(This explanation is due to Roger Miller via Steve Summit.)

Of course it is physically possible to pick up a basketball and run with it. It is also possible you will get away with it during a game. However, it is against the rules; good players won't do it and bad players won't get away with it for long. Evaluating (INT_MAX+1) in C or C++ is exactly the same: it may work sometimes, but don't expect to keep getting away with it. The situation is actually a bit subtle so let's look in more detail.

First, are there C implementations that guarantee two's complement behavior when a signed integer overflows? Of course there are. Many compilers will have this behavior when optimizations are turned off, for example, and GCC has an option (-fwrapv) for enforcing this behavior at all optimization levels. Other compilers will have this behavior at all optimization levels by default.

There are also, it should go without saying, compilers that do not have two's complement behavior for signed overflows. Moreover, there are compilers (like GCC) where integer overflow behaved a certain way for many years and then at some point the optimizer got just a little bit smarter and integer overflows suddenly and silently stopped working as expected. This is perfectly OK as far as the standard goes. While it may be unfriendly to developers, it would be considered a win by the compiler team because it will increase benchmark scores.

In summary: There's nothing inherently bad about running with a ball in your hands and also there's nothing inherently bad about shifting a 32-bit number by 33 bit positions. But one is against the rules of basketball and the other is against the rules of C and C++. In both cases, the people designing the game have

created arbitrary rules and we either have to play by them or else find a game we like better.

Why Is Undefined Behavior Good?

The good thing — the only good thing! — about undefined behavior in C/C++ is that it simplifies the compiler's job, making it possible to generate very efficient code in certain situations. Usually these situations involve tight loops. For example, high-performance array code doesn't need to perform bounds checks, avoiding the need for tricky optimization passes to hoist these checks outside of loops. Similarly, when compiling a loop that increments a signed integer, the C compiler does not need to worry about the case where the variable overflows and becomes negative: this facilitates several loop optimizations. I've heard that certain tight loops speed up by 30%-50% when the compiler is permitted to take advantage of the undefined nature of signed overflow. Similarly, there have been C compilers that optionally give undefined semantics to unsigned overflow to speed up other loops.

Why Is Undefined Behavior Bad?

When programmers cannot be trusted to reliably avoid undefined behavior, we end up with programs that silently misbehave. This has turned out to be a really bad problem for codes like web servers and web browsers that deal with hostile data because these programs end up being compromised and running code that arrived over the wire. In many cases, we don't actually need the performance gained by exploitation of undefined behavior, but due to legacy code and legacy toolchains, we're stuck with the nasty consequences.

A less serious problem, more of an annoyance, is where behavior is undefined in cases where all it does is make the compiler writer's job a bit easier, and no performance is gained. For example a C implementation has undefined behavior when:

An unmatched ' or " character is encountered on a logical source line during tokenization.

With all due respect to the C standard committee, this is just lazy. Would it really impose an undue burden on C implementors to require that they emit a compile-time error message when quote marks are unmatched? Even a 30 year-old (at the time C99 was standardized) systems programming language can do better than this. One suspects that the C standard body simply got used to throwing behaviors into the "undefined" bucket and got a little carried away. Actually, since the C99 standard lists 191 different kinds of undefined behavior, it's fair to say they got a lot carried away.

Understanding the Compiler's View of Undefined Behavior

The key insight behind designing a programming language with undefined behavior is that the compiler is only obligated to consider cases where the behavior is defined. We'll now explore the implications of this.

If we imagine a C program being executed by the C abstract machine, undefined behavior is very easy to understand: each operation performed by the program is either defined or undefined, and usually it's pretty clear which is which. Undefined behavior becomes difficult to deal with when we start being concerned with all possible executions of a program. Application developers, who need code to be correct in every situation, care about this, and so do compiler developers, who need to emit machine code that is correct over all possible executions.

Talking about all possible executions of a program is a little tricky, so let's make a few simplifying assumptions. First, we'll discuss a single C/C++ function instead of an entire program. Second, we'll assume that the function terminates for every input. Third, we'll assume the function's execution is deterministic; for example, it's not cooperating with other threads via shared memory. Finally, we'll pretend that we have infinite computing resources, making it possible to exhaustively test the function. Exhaustive testing means that all possible inputs are tried, whether they come from arguments, global variables, file I/O, or whatever.

The exhaustive testing algorithm is simple:

1. Compute next input, terminating if we've tried them all

2. Using this input, run the function in the C abstract machine, keeping track of whether any operation with undefined behavior was executed
3. Go to step 1

Enumerating all inputs is not too difficult. Starting with the smallest input (measured in bits) that the function accepts, try all possible bit patterns of that size. Then move to the next size. This process may or may not terminate but it doesn't matter since we have infinite computing resources.

For programs that contain unspecified and implementation-defined behaviors, each input may result in several or many possible executions. This doesn't fundamentally complicate the situation.

OK, what has our thought experiment accomplished? We now know, for our function, which of these categories it falls into:

- Type 1: Behavior is defined for all inputs
- Type 2: Behavior is defined for some inputs and undefined for others
- Type 3: Behavior is undefined for all inputs

Type 1 Functions

These have no restrictions on their inputs: they behave well for all possible inputs (of course, "behaving well" may include returning an error code). Generally, API-level functions and functions that deal with unsanitized data should be Type 1. For example, here's a utility function for performing integer division without executing undefined behaviors:

```
int32_t safe_div_int32_t (int32_t a, int32_t b) {
  if ((b == 0) || ((a == INT32_MIN) && (b == -1))) {
    report_integer_math_error();
    return 0;
  } else {
    return a / b;
  }
}
```

Since Type 1 functions never execute operations with undefined behavior, the compiler is obligated to generate code that does something sensible regardless of the function's inputs. We don't need to consider these functions any further.

Type 3 Functions

These functions admit no well-defined executions. They are, strictly speaking, completely meaningless: the compiler is not even obligated to generate even a

return instruction. Do Type 3 functions really exist? Yes, and in fact they are common. For example, a function that — regardless of input — uses a variable without initializing it is easy to unintentionally write. Compilers are getting smarter and smarter about recognizing and exploiting this kind of code. Here's a great example from the Google Native Client project:

```
When returning from trusted to untrusted code, we must sanitize the
return address before taking it. This ensures that untrusted code cannot
use the syscall interface to vector execution to an arbitrary address.
This role is entrusted to the function NaClSandboxAddr, in sel_ldr.h.
Unfortunately, since r572, this function has been a no-op on x86.

-- What happened?

During a routine refactoring, code that once read

aligned_tramp_ret = tramp_ret & ~(nap->align_boundary - 1);

was changed to read

return addr & ~(uintptr_t)((1 << nap->align_boundary) - 1);

Besides the variable renames (which were intentional and correct), a
shift was introduced, treating nap->align_boundary as the log2 of bundle
size.
```

We didn't notice this because NaCl on x86 uses a 32-byte bundle size. On x86 with gcc, (1 << 32) == 1. (I believe the standard leaves this behavior undefined, but I'm rusty.) Thus, the entire sandboxing sequence became a no-op.

This change had four listed reviewers and was explicitly LGTM'd by two. Nobody appears to have noticed the change.

-- Impact

There is a potential for untrusted code on 32-bit x86 to unalign its instruction stream by constructing a return address and making a syscall. This could subvert the validator. A similar vulnerability may affect x86- 64.

ARM is not affected for historical reasons: the ARM implementation masks the untrusted return address using a different method.

What happened? A simple refactoring put the function containing this code into Type 3. The person who sent this message believes that x86-gcc evaluates (1<<32) to 1, but there's no reason to expect this behavior to be reliable (in fact it is not on a few versions of x86-gcc that I tried). This construct is definitely undefined and of course the compiler can do done anything it wants. As is typical for a C compiler, it chose to simply not emit the instructions corresponding to the undefined operation. (A C compiler's #1 goal is to emit efficient code.) Once the Google programmers gave the compiler the license to kill, it went ahead and

killed. One might ask: Wouldn't it be great if the compiler provided a warning or something when it detected a Type 3 function? Sure! But that is not the compiler's priority.

The Native Client example is a good one because it illustrates how competent programmers can be suckered in by an optimizing compiler's underhanded way of exploiting undefined behavior. A compiler that is very smart at recognizing and silently destroying Type 3 functions becomes effectively evil, from the developer's point of view.

Type 2 Functions

These have behavior that is defined for some inputs and undefined for others. This is the most interesting case for our purposes. Signed integer divide makes a good example:

```
int32_t unsafe_div_int32_t (int32_t a, int32_t b) {
  return a / b;
}
```

This function has a precondition; it should only be called with arguments that satisfy this predicate:

`(b != 0) && (!((a == INT32_MIN) && (b == -1)))`

Of course it's no coincidence that this predicate looks a lot like the test in the Type 1 version of this function. If you, the caller, violate this precondition, your program's meaning will be destroyed. Is it OK to write functions like this, that have non-trivial preconditions? In general, for internal utility functions this is perfectly OK as long as the precondition is clearly documented.

Now let's look at the compiler's job when translating this function into object code. The compiler performs a case analysis:

- Case 1: `(b != 0) && (!((a == INT32_MIN) && (b == -1)))`
  Behavior of / operator is defined â†' Compiler is obligated to emit code computing a / b
- Case 2: `(b == 0) || ((a == INT32_MIN) && (b == -1))`
  Behavior of / operator is undefined â†' Compiler has no particular obligations

Now the compiler writers ask themselves the question: What is the most efficient implementation of these two cases? Since Case 2 incurs no obligations, the simplest thing is to simply not consider it. The compiler can emit code only for Case 1.

A Java compiler, in contrast, has obligations in Case 2 and must deal with it (though in this particular case, it is likely that there won't be runtime overhead since processors can usually provide trapping behavior for integer divide by zero).

Let's look at another Type 2 function:

```
int stupid (int a) {
  return (a+1) > a;
}
```

The precondition for avoiding undefined behavior is:

```
(a != INT_MAX)
```

Here the case analysis done by an optimizing C or C++ compiler is:

- Case 1: `a != INT_MAX`
  Behavior of + is defined: Computer is obligated to return 1
- Case 2: `a == INT_MAX`
  Behavior of + is undefined: Compiler has no particular obligations

Again, Case 2 is degenerate and disappears from the compiler's reasoning. Case 1 is all that matters. Thus, a good x86-64 compiler will emit:

```
stupid:
  movl $1, %eax
  ret
```

If we use the -fwrapv flag to tell GCC that integer overflow has two's complement behavior, we get a different case analysis:

- Case 1: `a != INT_MAX`
  Behavior is defined: Computer is obligated to return 1
- Case 2: `a == INT_MAX`
  Behavior is defined: Compiler is obligated to return 0

Here the cases cannot be collapsed and the compiler is obligated to actually perform the addition and check its result:

```
stupid:
  leal 1(%rdi), %eax
  cmpl %edi, %eax
  setg %al
  movzbl %al, %eax
  ret
```

Similarly, an ahead-of-time Java compiler also has to perform the addition because Java mandates two's complement behavior when a signed integer overflows (I'm using GCJ for x86-64):

```
_ZN13HelloWorldApp6stupidEJbii:
  leal 1(%rsi), %eax
  cmpl %eax, %esi
  setl %al
  ret
```

This case-collapsing view of undefined behavior provides a powerful way to explain how compilers really work. Remember, their main goal is to give you fast code that obeys the letter of the law, so they will attempt to forget about undefined behavior as fast as possible, without telling you that this happened.

A Fun Case Analysis

About a year ago, the Linux kernel started using a special GCC flag to tell the compiler to avoid optimizing away useless null-pointer checks. The code that caused developers to add this flag looks like this (I've cleaned up the example just a bit):

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;

  if (!dev) return;
```

```
  ... do stuff using dev ...
}
```

The idiom here is to get a pointer to a device struct, test it for null, and then use it. But there's a problem! In this function, the pointer is dereferenced before the null check. This leads an optimizing compiler (for example, gcc at -O2 or higher) to perform the following case analysis:

- Case 1: `dev == NULL`
  "dev->priv" has undefined behavior: Compiler has no particular obligations
- Case 2: `dev != NULL`
  Null pointer check won't fail: Null pointer check is dead code and may be deleted

As we can now easily see, neither case necessitates a null pointer check. The check is removed, potentially creating an exploitable security vulnerability.

Of course the problem is the use-before-check of pci_get_drvdata()'s return value, and this has to be fixed by moving the use after the check. But until all such code can be inspected (manually or by a tool), it was deemed safer to just tell the compiler to be a bit conservative. The loss of efficiency due to a predictable branch like this is totally negligible. Similar code has been found in other parts of the kernel.

Living with Undefined Behavior

In the long run, unsafe programming languages will not be used by mainstream developers, but rather reserved for situations where high performance and a low resource footprint are critical. In the meantime, dealing with undefined behavior is not totally straightforward and a patchwork approach seems to be best:

- Enable and heed compiler warnings, preferably using multiple compilers
- Use static analyzers (like Clang's, Coverity, etc.) to get even more warnings
- Use compiler-supported dynamic checks; for example, gcc's -ftrapv flag generates code to trap signed integer overflows
- Use tools like Valgrind to get additional dynamic checks
- When functions are "type 2" as categorized above, document their preconditions and postconditions
- Use assertions to verify that functions' preconditions are postconditions actually hold

- Particularly in C++, use high-quality data structure libraries

Basically: be very careful, use good tools, and hope for the best.

July 9, 2010 regehr Computer Science, Software Correctness

---

29 responses to "A Guide to Undefined Behavior in C and C++, Part 1"

1. **Michael Norrish** says:
   July 9, 2010 at 6:13 am

   Nice write-up!

2. **Eric LaForest** says:
   July 9, 2010 at 6:56 am

Thought-provoking post, thank you.

But I'm puzzled: what type of optimization can infer the indeterminacy of stupid() or the null-pointer check? I've not heard of it.

Wouldn't it be better for the compiler to decide what to do based on what knowledge it has at compile-time?

For example: stupid(foo) would compile the full code, while stupid(5) would compile to a literal, as per constant propagation and expression simplification.

Similarly, in the case of the NULL-check, why bother at all? The value of dev is not known at compile time, and the first dereference would trigger a segfault in the case of a NULL anyway.

3. **regehr** says:
   July 9, 2010 at 10:13 am

   Hi Eric- I may need to update this post to be a bit more clear about these things!

   I don' t think these optimizations have any specific names, nor do I think they're written up in the textbooks. But basically they all fall under the umbrella of "standard dataflow optimizations." In other words, the compiler learns some facts and propagates them around in order to make changes elsewhere. The only difference is in the details of what facts are learned.

Just to be clear, these optimizations are absolutely based on knowledge the compiler has at compile time. Everything I described in this post is just a regular old compile-time optimization.

stupid(foo) — where foo is a free variable — compiles to “return 1” using the case analysis.

Re. the null check example, remember this is in the Linux kernel. In the best possible case, accessing a null pointer crashes the machine. In the worst case there is no crash: exploit code is waiting to take over the machine when you access the null pointer. This is precisely the case that the kernel programmers are worried about. This is not theoretical: if your Linux kernel accesses a null pointer, I can probably own your machine.

4. **Matthias Felleisen** says:
   July 9, 2010 at 10:44 am

   Thank you.

   What’s really sad is that some so-called high-level languages like Scheme intentionally include undefined behavior, too. This may make Scheme look like a real systems language after all.

   I have been preaching the ‘unsafe and undefined gospel’ for a long time in PL courses. Indeed, I have been preaching it for such a long time that former students still have T shirts with my quote “Scheme is just as bad as

FORTRAN" for the same issue. Indeed, some HotSpot compiler authors may still own this T shirt.

Sadly, I never got support from 'real' compiler colleagues at Rice nor from the systems people. Real man just cope. Shut up and work.

So thanks for speaking up as a "systems" person.

— Matthias

5. **regehr** says:
   July 9, 2010 at 11:15 am

   Hi Matthias- Thanks for the note!

   I feel like certain languages were designed by and for compiler people. Optimizations good, everything else: irrelevant. Hopefully these languages will lose (or be revised) to cope with the modern situation where machine resources are relatively cheap and program errors are relatively costly. Of course multicores will probably set us back a couple decades in terms of program correctness…

   I was very surprised to learn from Matthew about Scheme's undefined behaviors.

6. **Matthias Felleisen** says:
   July 9, 2010 at 12:23 pm

You write “Of course multicores will probably set us back a couple decades in terms of program correctnessâ€¦” How sad and how true!

In Scheme’s case, it is fortunately acceptable for a compiler to implement a safe and determinate language, which is what Matthew’s PLT Scheme did and what Racket does now. Sadly, compiler writers love indeterminate specs and they love writing language specifications even more. Perhaps the latter is the real reason that we will not get away from such bad languages for ever.

7. **Adam Morrison** says:
   July 10, 2010 at 12:20 pm

   I’m not sure about the Google Native client example. Looks to me like the compiler emitted x86 code that, when passed 32 as input, would calculate (1<<32) == 1.

   It just so happened that this function was always called with the "bundle_size" being 32 and as a result it became a no-op semantically. It didn't mask out the low bits of the address like it was supposed to.

8. **Ben L. Titzer** says:
   July 12, 2010 at 5:43 pm

   Though I completely agree with the idea of making programs’ semantics as well defined as possible (independent of implementation and target machine of course), there are always cracks to slip through. E.g. most

semantics assume that the target machine has enough resources to actually run the program. Of course then they define what should happen if the machine does not–OutOfMemoryError, StackOverFlowError, and the like. But what about the case when the machine *almost* has enough resources. Then the impact of optimizations can be felt. E.g. how much of the heap is occupied by class metadata? Even with the same heap size, the same program might run to completion (or continue working) or throw OutOfMemory on different VMs. Similarly with stack size–if the compiler or VM performs tail recursion elimination, the program may work fine, but fail immediately without (note that no compliant JVM does).

Although one can define semantics to be nondeterministic, it is unusual to go to all the trouble of formally defining them in order to leave some part nondeterministic. We try to eradicate it but nondeterminism keeps creeping in, like these choice examples in Java:

* result of java.lang.System.identityHashCode()
* order of finalizers being run, and on which thread
* policy for clearing of SoftReferences

9. **Ben L. Titzer** says:
   July 12, 2010 at 5:46 pm

   Just to be clear though: nondeterministic behavior is far preferable to undefined behavior 🙂

10. **regehr** says:
    July 12, 2010 at 7:30 pm

    Hey Ben- Let's be clear that there are two separate questions here. First, whether or not an error occurs. Second, what should the language do when an error occurs.

    People developing security-, safety-, and mission-critical software care a lot about the first one and it's a really hard problem, optimizations matter, etc.

    This post was only about the second one: does the program halt with a nice error or does it keep going in some screwed-up state. Nailing this down seems like the first order of business, then later on we can worry about making offline guarantees about error-freedom and all that.

11. **Eugene Toder** says:
    July 30, 2010 at 4:32 pm

    GCC actually tends to evaluate (1 <= 32 to 0. This is even more optimal than evaluating it to x, as there's no need to evaluate x, literal 0 can trigger further optimizations and literal 0 is very cheap on most platforms. In this case most likely GCC was not able to find undefined behaviour at compile-time and generated x86 shl instruction. 32-bit shl on x86 only looks at the lowest 5 bits of it's RHS, thus (1 << 32) == (1 << 0) == 1. However non-intuitive this is, this is a result of CPU optimization, not compiler optimization.

12. **Nadav** says:

August 17, 2010 at 11:30 pm

This is a good article!

I wanted to point out that in Verilog and VHDL (hardware description languages) you have syntax that is undefined. It is a part of the standard of the language but it is unsynthesizable to hardware circuits.

13. **Neil Harding** says:
August 19, 2010 at 10:18 am

The reason that arithmetic operations are undefined is due to not requiring a particular implementation, so if you used a processor with BCD (binary coded decimal) values for integer operations, or 1's complement format then INT_MAX + 1 would return different values than a 2's complement format architecture.

ASSERT can be used to check for preconditions, but since you are running in debug mode when this is enabled, some optimizations are not enabled and so the preconditions may hold true in debug mode, but not in release mode.

I actually prefer coding in 68000 (6502 & Z80 required too much work for the simple operations) to programming in C, or Java. I found I could do optimal code, and use the condition code flags to perform multiple checks at one time. So a = a + 1, would set Z flag if a was = -1, V flag (overflow) if a = MAX_INT, and N flag (if result < 0, which include MAX_INT). I've done

some x86 assembly but since it is such a horrible mess, then C/C++ is preferable.

14. **Peter da Silva** says:
    August 19, 2010 at 4:23 pm

    The reason for things like “An unmatched â€˜ or ” character is encountered on a logical source line during tokenization” being undefined is not to make the compiler’s job easier, it’s to make the standards body’s job possible. Many of these kinds of undefined behavior are cases where:

    * Important compilers did it differently.
    * Important code depended on what their particular compiler did.

    I am 99.44% positive that there are a number of cases where it would make a lot of sense to define certain behavior as an error, but if you did that you’d have to rewrite parts of the Linux kernel or the NT kernel because GCC and Microsoft C did things different ways… and since you’re never going to compile the Linux kernel with anything but GCC (try it some time) or the NT kernel with Microsoft C, neither side has a good reason to back down.

15. **regehr** says:
    August 19, 2010 at 9:59 pm

    Neil, how many non-2’s complement targets are out there? I know this rationale was used historically but it hardly seems relevant to C99 and C++0x.

I agree with you that the lack of access to overflow flags in C and C++ is really annoying. It makes certain kinds of code hard or impossible to express efficiently.

16. **regehr** says:
    August 19, 2010 at 10:02 pm

    Peter of course you're right, thanks for pointing this out. The standards body certainly had an unenviable task.

17. **Kumari Swarnim** says:
    August 20, 2010 at 4:50 am

    It is nice.

18. **Steve** says:
    August 20, 2010 at 8:53 am

    You go too far when you say that anyone relying on twos-complement overflow behaviour will one day be proven wrong. Of course pedantically it's true, but in reality, people know broadly what kind of platform they're developing for even if not precisely which platform. Twos-complement is pretty much universal now, and C/C++ compiler developers would be insane to not ensure it just worked as expected irrespective of the letter of the standards. Anyone developing for an obscure sign-magnitude platform or whatever will know about it.

19. **regehr** says:
    August 20, 2010 at 11:18 am

    Hi Steve- I must have failed to explain things clearly. This is the situation: today's C and C++ compilers do not have 2's complement semantics for signed overflow. Did you read the example in the post where real C compilers evaluate (x+1)>x as "true"? Do I need to point out that this result is not correct under a 2's complement interpretation of integers?

20. **Steve** says:
    August 20, 2010 at 3:20 pm

    Hi – no you didn't explain badly, but my point still stands. When dealing with the boundary between what is formally-undefined-but-expected and what is just plain undefined, there's always going to be a degree of subjectivity, but here's the thing – it is (for example) impractical if not impossible to implement a big-integer library without making (reasonable but standards-undefined) assumptions about integer representation and overflow behaviour. Big integer libraries exist. They won't run on every platform everywhere, but they still manage a fair bit of portability.

    If you write "(x+1)>x", the real question is "why are you doing this?". Of course the optimisation makes sense, just as the 2s complement assumption would make sense. But equally, this is an artificial example. For real world code, you can rely on the fact that compiler writers actually want scripting languages and other big-integer clients to carry on working too. I believe

GCC even *uses* GNUs big integer library to do it's compile-time calculations these days.

I repeat – pedantically, yes, you are going to encounter problems and odd corner cases – but stick within the kinds of coding patterns that are widely used and your code will work irrespective of "undefined behaviour".

The purpose of a real-world compiler is to compile real-world code and, while optimisation can sometimes get overzealous, the integer overflow issue isn't as bad as you make out.

OTOH – pointer alias analysis (or rather the failure to detect an alias due to pointer casts and arithmetic) is a real expletive-causing issue. I've had that with GCC recently, and I couldn't really figure out a resolution other than (1) have tons of template bloat to avoid having a type-unsafe implementation, or (2) dial down the optimisations. Yet no-one can seriously claim that there's no history of pointer casts and arithmetic in C and C++.

But IMO this ones a reason to complain to the compiler writers – as I said, the purpose of a real-world compiler is to compile real-world code. It's virtually impossible to write a real-world app that doesn't invoke some kind of undefined behaviour in C or C++, so compiler writers have more responsibilities than just complying with the standards. The end users, after all, are you and me – not just the standards people.

That said, so far I haven't looked that hard for a resolution, and that's the real issue here with the alias analysis. C and C++ are languages for people who are willing to patch up the problems from time to time (or stick to known versions of known compilers), and I just haven't checked how to fix this one low-priority library yet.

21. **regehr** says:
    August 20, 2010 at 6:42 pm

    Hi Steve- There is much merit to what you say. However, you are wrong about one fundamental point: when dealing with a programming language the real question is not "why are you doing this." I actually wrote a post about this exact topic a while ago:

    http://blog.regehr.org/archives/47

    I'll tell you what, let's run an experiment. If you've looked at part 2 of this series of posts, you'll see that my group has a tool for detecting integer undefined behaviors. I'll run this on GMP, which I suspect is the most popular bignum package (if you have other ideas, let me know).

    My expectation is that every single signed overflow in GMP will be considered a bug by the GMP developers and will be fixed after I report it. Your position, if I guess correctly, is that they are happy to leave these in there because the compiler somehow understands what the developer is

trying to do, and respects 2's complement behavior when it really matters. Does that sound right?

22. **Steve** says:
    August 21, 2010 at 1:59 pm

    I got your e-mails, and to be honest, I'm surprised this is going on so long. After all, we agree that C and C++ leave a lot undefined, and that means that those languages aren't as safe as e.g. Ada.

    As for what happens on GMP, my guess is that they'll look at the specific cases and see what they can do. Especially as, based on you finding a grand total of nine issues, it looks like I was wrong in saying that doing big integers is impractical without relying on integer representation and overflow behaviour.

    Since it's impossible to ever say "I'm wrong" without a "but" 😉

    Am I correct in saying that until C99 and stdint.h, there was no way in standard C to specify that you want a 32bit (or any other specific size) integer? GMP was first released in 1991, certainly. Evolving towards greater portability is hardly a surprise, but it *is* evolution, with problems fixed as and when they're found.

    Those undefined behaviour "bugs" are there now and have probably been there a while. I assume GMP have strong unit tests, so if a real problem arose, it would have been noticed.

So one relevant question is – is it actually productive to be spending time hunting down and fixing undefined behaviour bugs that don't cause anyone a problem, when they could be investing that time in something else?

As you said, one thing the developer has to do is to hope. I'd say it's more a matter of expecting to do maintenance as platforms and compilers evolve. And even if you code in Ada, you'd still need to do maintenance from time to time – e.g. you might find around now that you need 64 bit integers, where not only didn't you anticipate it years ago, but your compiler wouldn't allow it anyway.

Is the Ada way better that the C/C++ way? I think so, and apparently so do you. But reality is that very few people use Ada, and while C and C++ are less than ideal, they're only occasionally fatal. And lets face it – GNAT even has those mandated overflow checks disabled by default for performance reasons, so using Ada is no guarantee in itself. And I guess if your unit tests are strong enough, it doesn't matter – don't laugh, some people can manage to write unit tests, honest.

Actually, since you mentioned LLVM and Clang in an e-mail – bugpoint is something I must look at some time.

Moving on – if the compiler doesn't care about intent and only cares about the letter of the standard, how do you balance that against the fact that for half of its history C didn't have a standard – the ratio being somewhat worse for C++.

Obviously it's not the job of the compiler to guess, but there's no fundamental difference between the guys who write standards and the guys who write compilers – they're all people and all (hopefully) experienced developers. Someone somewhere figures out what is needed, documents it and develops it, not necessarily in that order. If they develop something that can't cope with real world code, thousands of other developers will shout foul. In the aftermath, either the offending compilers or real-world programming practice will adapt. In a world where perfection is rare, this mostly sort-of works.

For example, can you imagine what would happen if the Python devs suddenly decided (with no standard to say they can't) to change the semantics of the division operator? Errrm. Oh. Errr – actually, forget that bit 😉

Thinking about it, my whole argument requires people to shout foul from time to time, which you were doing. So maybe again I'm on the wrong side of things.

23. **regehr** says:
    August 21, 2010 at 2:21 pm

    Hi Steve- Yeah, we mostly agree. I think the point of disagreement is whether people should fix undefined behaviors if they're not currently causing problems. Of course this is an individual choice made by each developer. My position on the matter, for any software I cared about, would be to fix these

issues once I knew about them — it just saves time later. It's sort of like fixing compiling warnings that aren't pointing out major problems — often you just do it to get the tool to shut up, so that next time the tool says something you'll notice.

24. **Steve** says:
    August 21, 2010 at 7:16 pm

    On the warnings thing, I see the point.

    BTW – I just realised a minor misunderstanding, which kind of explains why you said about compilers guessing intent. When I said "If you write "(x+1)>x", the real question is "why are you doing this?".", my intent wasn't to suggest the compiler should work out your intent, but to point out that this isn't sane real-world code. Depending on your choice of common interpretation this either evaluates to (true) or to (x != INT_MAX) – and one of those is what you'd expect to see in real code.

    OTOH the issue can happen indirectly, and implicit inlines can lead to the optimisation happening where you wouldn't expect it, which is potentially a problem.

25. **regehr** says:
    August 22, 2010 at 8:56 pm

    Hi Steve- – Yes this is exactly right: machine generated code, macros, inlining, etc. can cause bizarre source code that a human would never write.

Also, most good compilers these days will perform inlining across compilation units, so it's not really easy to predict what the code that the compiler finally sees will look like.

26. **Steve** says:
    August 25, 2010 at 3:36 pm

    Cleared up my full misundertanding here…

    http://stackoverflow.com/questions/3569424/how-to-do-a-double-chunk-add-with-no-undefined-behaviour

    Basically, big integers without undefined behaviour have been perfectly possible for some time – the languages are (slightly, but in a very significant way) better defined than I realised, and have been for a little over ten years.

27. **Undefined behavior in C andÂ C++ « IPhVu::iLearn** says:
    October 27, 2010 at 1:17 am

    […] Undefined behavior in C andÂ C++ 19/08/2010 phvu Leave a comment Go to comments http://blog.regehr.org/archives/213 […]

28. **Undefined Behavior of C and C++ | Itsy Bitsy** says:
    November 2, 2010 at 4:27 pm

    […] A Guide to Undefined Behavior in C and C++, Part 1 A Guide to Undefined Behavior in C and C++, Part 2 A Guide to Undefined Behavior in C

and C++, Part 3 This entry was posted in Worth reading. Bookmark the permalink. ← Making a JUnit 4 Test Suite [...]

29. **outerplanet** says:
    January 22, 2011 at 6:56 pm

    [...] lecture, aboutÂ Undefined Behavior in c and c plus plus, at RegehrÂ´s blog Look at the piece of code: what is going to happen [...]