Document Number: P1407R1
Date: 2019-03-08
Reply-to: Scott Schurr: s dot scott dot schurr at gmail dot com
Author: Scott Schurr
Audience: SG12

Tell Programmers About Signed Integer Overflow Behavior

Abstract: Every C++ implementation knows exactly what it does when signed integer overflow occurs. If signed integer overflow in C++ were implementation-defined, rather than undefined, then each implementation would be obliged to document what it does when a signed integer overflows. That, in the opinion of the author, would be a Good Thing ®.

Revision History

Revision 1

- Improve motivating example.
- Add Compiler Warnings section.
- Add saturation as a potential overflow behavior.
- Incorporate additional reactions and responses.

Revision 0

Initial revision.

A Motivating Example

Imagine you are a programmer with a background in electrical engineering. You have never worked on compiler internals or C++ standard libraries. You are developing an embedded product on a small team for a small company. You are coding a helper function that clips on integer overflow. Which of these two implementations do you choose?

```
int
add_100_without_wrap (int a)
{
  using namespace std;
  int const ret = a + 100;
  if (ret < a)
    return
      numeric_limits<int>::max();
  return ret;
}
```

```
unsigned int
add_100_without_wrap (unsigned int a)
{
  using namespace std;
  unsigned int const ret = a + 100u;
  if (ret < a)
    return
      numeric_limits<unsigned int>::max();
  return ret;
}
```

You, as a member of the C++ Standards Committee, of course start out by saying, "I would never write that code." Yes, but you were asked to have some imagination. If you had to pick the code that would behave in the expected fashion, you'd pick the code on the right. That's because you know that the code on the left contains undefined behavior.

Now, if you please, step back. How did you know that?

Sources of Information About Signed Integer Overflow

Reflect on how you, personally, learned that in C++ signed integer overflow is undefined behavior.

Colleagues and Friends

The author hazards to guess that most programmers who find out that signed integer overflow is undefined behavior in C++ find out from friends or colleagues. That is certainly how the author found out.

Internet Blogs, Posts, and Videos

Yes, there are quite a number of videos, blogs, and posts about the undefined behavior of signed integer overflow in C++. But the internet is a big place. Some dedicated video-watching C++ programmers might wander across those by accident. But, honestly, haven't most of those dedicated video-watching C++ programmers already heard from a coworker that signed integer overflow is undefined?

Case in point. There are reasons to believe there are about 4.4 million C++ programmers in the world today [1]. The most popular CppCon video on undefined behavior that this author has identified is by Chandler Carruth [2]. Here is a link to where Mr. Carruth talks about signed integer overflow: https://youtu.be/yG1OZ69H_-o?t=1994. At the time of this writing that video has 30,366 views.

So that video has informed approximately 0.7% of C++ programmers world wide about the undefined behavior of unsigned integer overflow. We know nothing about the remaining 99.3% of C++ programmers.

Similarly, the cppreference.com website explicitly gives an example of signed overflow being undefined behavior [3]. We have no idea how many (or few) of the 4.4 million C++ programmers have seen this example and understood its ramifications.

C++ Books

A common, if somewhat old fashioned, way to learn C++ is by reading C++ books. The author took an arbitrary survey of C++ books either in his possession, or at the local public library, or on the shelves at local book stores. The goal was to see how many of them talk about…

- Undefined behavior in general and
- Specifically discuss that signed integer overflow is undefined.

The results are below.

Author	**Title**	**Lists undefined behavior in the index**	**Mentions that signed integer overflow is undefined**
Davis, Stephen R.	C++ For Dummies, 7th Edition	No	No
McGrath, Mike	C++ Programming in Easy Steps	No	No
Meyers, Scott	Effective C++	No	No
Meyers, Scott	More Effective C++	Yes: pages 10, 21, 35, 163, 167, 173, 275, 281	No
Meyers, Scott	Effective C++ Third Edition	Yes: pages 6, 7, 26, 30, 41, 43, 45, 63, 73, 74, 91, 231, 247	No
Meyers, Scott	Effective Modern C++	Yes: page 6	No
Prata, Stephen	C++ Primer Plus Sixth Edition	No	No

Author	Title	Lists undefined behavior in the index	Mentions that signed integer overflow is undefined
Rao, Siddhartha	Sam's Teach Yourself C++ in One Hour a Day, 7th Edition	No	No
Schildt, Herbert	C/C++ Programmer's Reference 2nd Edition	No	No
Stroustrup, Bjarne	The C++ Programming Language Third Edition	Yes: page 828	No
Stroustrup, Bjarne	The C++ Programming Language Fourth Edition	Yes: page 136	No
Sutter, Herb	Exceptional C++	No	No
Sutter, Herb	More Exceptional C++	No	No
Sutter and Alexandrescu	C++ Coding Standards	Yes: pages 19, 25, 27, 36, 39, 61, 71, 88, 90, 91, 93, 173, 179, 181, 182, 183, 184, 185	No
Yaroshenko, Oleg	The Beginner's Guide to C++	No	No

Compiler Warnings

According to a report from Marc Glisse [4] the motivating example at the beginning of this paper, when compiled with g++-7 -O2 -Wall, produces the following diagnostic:

```
<source>: In function 'int add_100_without_wrap(int)'
<source>:8:3: warning: assuming signed integer overflow does not occur when
    assuming that (X + c) < X is always false [-W-strict-overflow]
       if (ret < a)
```

According to Mr. Glisse the warning was removed from gcc-8 because it was too noisy and impossible to work around when the optimization is actually what you want.

So compilers could warn users that their code was making assumptions about the results of signed integer wrapping. And one compiler did, at least at certain optimization levels. But, apparently, no current compilers do so.

Reading the C++ Standard

We don't realistically believe most C++ programmers read the standard, do we? But, if one did, how would they find out that signed integer overflow is undefined? In the C++17 standard Section 6.9.1 Fundamental types [basic.fundamental] paragraph 4 footnote 49 says, "This

implies that unsigned arithmetic does not overflow …". But that entire section does not say anything explicitly about whether signed integer overflow is defined or not. The naive reader might assume that signed integer overflow is not an issue.

There are *other* places in the C++17 standard that make it clear signed integer overflow is undefined. This hinges on Section 8 Expressions [expr] paragraph 4:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

Then there are a few places throughout the standard that mention signed integer overflow having undefined behavior. These places include:

- Section 8.20 Constant expressions [expr.const] paragraph 2.6
- Section 21.3.4.1 numeric_limits members [numeric.limits.members] paragraph 62.
- Possibly Section 23.16.5 Comparison of ratios [ratio.comparison] paragraph 1.
- Section 23.17.5.8 Suffixes for duration literals [time.duration.literals] paragraph 3.

So, yeah, the information is in the standard. But you need to know how the read the standard in order to find it. And, the author submits, most of the 4.4 million C++ programmers don't read the standard.

In Total

So, truly, the information is available. But the author's best guess is that the information is primarily passed through the rumor mill to those few (percentage-wise) who receive it. Once the information is provided it is easy to confirm using the web. But getting the initial message through is tenuous.

We, the members of the C++ Standards Committee, have the problem that we are living in our own echo chamber of C++ experts. The non-experts do not hear everything that we hear.

Signed Integer Overflow Behaviors

Even though signed integer overflow is undefined in the standard, and has been for decades, signed integer overflow actually occurs in real programs. In order to deterministically generate code a compiler vendor needs to have a policy for what should happen if signed integer overflow occurs. To the best of the author's understanding, there are three different behaviors that C++ compilers/optimizers implement today for signed integer overflow. And there's another behavior that might be desirable and is supported by certain hardware [5]. They are:

- **Modulus wrapping.** This is what two's complement hardware typically does (if one ignores the flags register, which C and C++ do). If one wants to leave no room between C++ and the hardware [6], then this would be the expected behavior. It is also the behavior that many people trained in electrical engineering or physics find minimally surprising. Both clang and gcc provide this behavior with the **`-fwrapv`** compiler flag. And, until 2007 or so, a programmer could generally expect this behavior from their compiler even though the standard did not guarantee it.

- **Trapping.** If a signed integer operation overflows, then the program traps, typically with a diagnostic that helps someone locate the fault. This model seems to be favored by many mathematicians and computer scientists. The **`-ftrapv`** compiler flag, supported by both clang and gcc [7], implements this model. Visual Studio also supports arithmetic overflow checking.

- **Can't happen.** Starting somewhere around 2007 [8] this model has been used by some optimizers that assume code is free of signed integer overflow, presumably due to extensive testing with the **`-ftrapv`** flag. In this mode the optimizer assumes that signed integer overflow can never happen. So if code, accidentally or intentionally, relies on signed integer overflow, that code may be elided by the optimizer. Both clang and gcc support this model today through various compiler optimizer flags.

- **Saturation.** Beyond the three behaviors that have known C++ implementations, another possibly desirable signed integer overflow behavior would be saturation. This is the effect that our troubled programmer in the motivating example was trying to accomplish. There is existing hardware with support for saturating signed integer computations[9]. For example the SSE2 instruction set has the PADDSW (add packed signed word integers with saturation) instruction [10]. The PowerISA has vector add, subtract, and multiply instructions that saturate [11]. And the ARM NEON instruction set includes vector integer addition and subtraction which saturates [12].

It is possible that there are other models for signed integer overflow, but it's unlikely. As noted in P0907R1 Section 6 [13], there is no known non-two's complement hardware with a modern C++ compiler.

But the important point here is not a complete list of all behaviors. It is to see that, in every case, each compiler has a well understood reaction when faced with signed integer overflow. That behavior may change based on compiler flags, but it remains well understood. In effect, in actual implementations, signed integer overflow *is* (in the English, non-C++ Standard, meaning) implementation-defined.

The Standard Can Encourage Communication

Now that the standard defines signed integers as two's complement, it would be entirely reasonable to give signed integer overflow well defined behavior. However such a proposal would be unlikely to achieve consensus. The standards committee contains many fans of all three of the behaviors that compilers currently provide for signed integer overflow.

In the author's opinion the next best thing is to communicate to C++ programmers what happens when signed integers overflow. By good fortune it turns out that implementation-defined behavior, as specified by the C++ Standard, provides exactly that. From Section 3.12 [defns.impl.defined] of the C++17 Standard:

> **implementation-defined behavior**
>
> behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

(Emphasis added by the author.)

Implementation-defined Requires Implementation-Documented

So if signed integer overflow became implementation-defined, rather than undefined, each compiler vendor would be required to document the behavior they exhibit when a signed integer overflows. Programmers are notorious for not reading documentation unless something goes wrong. But if something does go wrong the documentation would now be available to help them figure out what happened.

But Some Implementations Don't Document Implementation-defined Behavior

True enough. The poster child for this is Clang [14]. The Intel C++ Compiler doesn't provide such documentation either. Visual Studio documents implementation-defined behavior for C [15], but not for C++. However there are compilers that make an effort to be compliant by documenting C++ implementation-defined behavior. GCC makes a stab at it [16].

You don't have to have a big name to do the right thing. The Analog Devices C++ compiler provides documentation for implementation-defined behavior that puts the big name compilers to shame [17], albeit that is a 2003 compliant compiler. The Analog Devices documentation includes 15 pages that describe its C++ implementation-defined behavior. Surely the big-time compiler vendors can afford to do the same.

The author's position is that compiler vendors that don't document their implementation-defined behavior are both out of compliance and doing their users a disservice. The C++17 Standard (N4660) indexes 232 items that are implementation-defined. That index includes important but not always easy to discover items like:

- Alignment,
- Behavior of non-standard attributes, and
- Whether certain kinds of dynamic initialization occur before main or are deferred.

If those go undocumented then users can only discover them experimentally or through hearsay, either of which may lead to incorrect answers.

Regardless, the standard has already done what it can. It requires compiler vendors to document their implementation-defined behavior. The standard has no means to enforce compliance.

Implementation-defined May Be Undefined

There might be some concern that implementation-defined behavior only allows well defined behavior. If that were the case then the "signed integer overflow can't happen" model would not be supported. However there are currently existing examples where implementation-defined behavior is allowed to lead to undefined behavior.

Check out, for instance, C++17 Standard Section 20.5.5.8 Reentrancy [reentrancy] paragraph 1. "Except where it is explicitly specified in this International Standard, it is implementation-defined which functions in the C++ standard library may be recursively reentered." That means a recursively called standard library function, where the implementation does not support such reentrancy, results in undefined behavior.

On a similar note, from Section 29.6.9 Low-quality random noise generation [c.math.rand] paragraph 3: "It is implementation-defined whether the rand function may introduce data races (20.5.5.6)." Remember, of course, that data races may result in undefined behavior (see Section 4.7.1 Data races [intro.races]).

Summary

So it turns out that changing signed integer overflow to be implementation-defined behavior achieves two goals:

- It allows the implementation to provide whatever behavior is deemed appropriate, including behaving as though signed integer overflow cannot happen.
- It requires that the implementation document whichever behavior(s) it provides.

The documentation component is specifically what is missing today.

Reactions and Responses

Reaction: Don't take away my optimization!
Response: With this proposed change the optimization can remain but, if implemented, must be documented by the compiler vendor.

Reaction: Most signed integer overflow is a programming error.
Response: That's probably true, but not all of it is. This change simply requires the compiler vendor to identify the contract that they are supplying for signed integer overflow. Anything beyond that is between the programmer and their tool vendors.

Reaction: The change is not worth it. It doesn't fix anything.
Response: It's true that the improvement is minor; it's only an improvement in documentation. However the relative cost is low. No compiler changes are required, only documentation changes. Documentation is certainly not free. However the required delta imposed in an implementation's documentation by this change is probably relatively small.

Reaction: Shouldn't we just make a new signed integer type?
Response: Possibly, but we don't have that now and providing it would likely take several standards cycles. Let's tell programmers what we're up to *right now*. After that we can talk about a new integer type.

Reaction: But if we make this change UBSan won't find the error.
Response: UBSan, and similar tools, are not defined by the standard. The UBSan community will, of course, follow it's own path. However it is easily imaginable that UBSan and similar tools could be parameterized to optionally identify specific cases of implementation-defined behavior should their communities choose to do so. As noted earlier there are pre-existing instances of implementation-defined behavior, like recursively calling implementation-defined standard library functions, that lead to undefined behavior. So such a UBSan extension seems within the realm of possibility.

Reaction: What about constexpr evaluation of signed integer overflow?
Response: For perspective, it's worth taking a step back to consider preexisting expectations on constexpr evaluation of floating point. According to The C++17 Standard, N4660 Section 8.20 Constant expressions [expr.const] paragraph 6, in a non-normative note,

> … it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution.

What's proposed here is that signed integer overflow would be implementation-defined. The author's expectation is that most implementations would stop translation and issue a diagnostic if signed integer overflow were encountered during constexpr evaluation, even if the runtime implementation does modulus wrapping. But it would be up to the implementation to choose (and document) the behavior. At the end of the day the situation would be no worse than the standing situation with floating point.

Reaction: Why signed integer overflow? What about other forms of undefined behavior?
Response: Shall we guess what is the most important data type supplied by the C++ standard today? A good guess would be `std::string`. And possibly the second most important data type is integers, both signed and unsigned. This proposal suggests that we, the C++ Standards Committee, should make a concerted effort to inform our users about potentially surprising behavior of common operations (addition, subtraction, and multiplication) on a ubiquitous data type.

Reaction: Will this encourage breaking the language into dialects?
Response: To a certain extent, the dialects already exist. The existence of the **`-fwrapv`** and **`-ftrapv`** compiler flags testify to this. The author sees no reason to believe changing the behavior from undefined to implementation-defined would aggravate the situation.

Reaction: We should fix this for real.
Response: True, however the problem is determining the correct fix and arriving at consensus. That is a worthwhile goal. In the meantime, let's document what we've done so our users have a ghost of a chance of finding out.

Outline for Proposed Wording Changes

- P1236R0 [18] has proposed wording for Section 6.7.1 [basic.fundamental] that describes the representation of the signed integer types. In a non-normative note it says, "Overflow for signed arithmetic yields undefined behavior (7.1 [expr.pre])." This paragraph would be the right place to specify that signed arithmetic overflow is implementation-defined.
- Switch all examples of undefined behavior that reference signed integer overflow to instead reference out-of-range pointer arithmetic.
- Add signed integer arithmetic overflow to the Index of implementation-defined behavior.

Thanks and Gratitude

The author would like to offer thanks to the following people who contributed to (but may or may not endorse) this paper: Howard Hinnant, J. F. Bastien, Aaron Ballman, Erich Keane, Marc

Glisse, John McFarlane, Hubert Tong, Jens Maurer, Melissa Mears and Robert Ramey. All mistakes are the sole property of the author.

References

[1] Anastasia Kazakova. C/C++ facts we learned before going ahead with CLion. URL: https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/

[2] Chandler Carruth. Garbage In, Garbage Out: Arguing About Undefined Behavior With Nasal Demons. CppCon 2016. Discussion of signed integer overflow. URL: https://youtu.be/yG1OZ69H_-o?t=1994

[3] cppreference.com, Undefined behavior, observed January 28, 2018. URL: https://en.cppreference.com/w/cpp/language/ub

[4] Thanks to Marc Glisse, private communication, for this example.

[5] Thanks to Mellissa Mears, private communication, for pointing out that saturation might be a desirable model.

[6] Stroustrup et al. Direction for ISO C++. P0939R1. Page 4, "Technically, C++ rests on two pillars: A direct map to hardware (initially from C)…" URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0939r1.pdf

[7] According to Marc Glisse, private communication, even though gcc documents support for the **`-ftrapv`** flag, it does not work correctly. With **`-fsanitize=signed-integer-overflow -fsanitize-undefined-trap-on-error`** you would get something roughly equivalent to what **`-ftrapv`** is supposed to do.

[8] assert(int+100 > int) optimized away. Bug 30475, GCC, 2007. URL: http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475

[9] Thanks to Hubert Tong and Jens Maurer, private communications, for identifying hardware implementations that support saturating signed integer arithmetic.

[10] Wikipedia, X86 Instruction Listing, accessed January 31 2019. URL: https://en.wikipedia.org/wiki/X86_instruction_listings#SSE2_instructions

[11] PowerISA ™ Version 2.07, May 3 2017, page 176. URL: http://fileadmin.cs.lth.se/cs/education/EDAN25/PowerISA_V2.07_PUBLIC.pdf

[12] ARM Developer Information Center, general arithmetic instructions. Accessed January 31, 2019. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/CIHJCAAG.html

[13] JF Bastien. P0907R1 - Signed Integers are Two's Complement. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r1.html

[14] Clang bug 11272 - "document implementation-defined behavior" reported 2011-10-30 by Richard Smith. URL: https://bugs.llvm.org/show_bug.cgi?id=11272

[15] Microsoft C Implementation Defined Behavior. URL: https://docs.microsoft.com/en-us/cpp/c-language/implementation-defined-behavior?view=vs-2017

[16] GCC C++ Implementation Defined Behavior. URL: https://gcc.gnu.org/onlinedocs/gcc-6.4.0/gcc/C_002b_002b-Implementation.html

[17] Analog Devices CrossCore Embedded Studio 2.8.0 C/C++ Compiler Manual for SHARC Processors. URL: https://www.analog.com/media/en/dsp-documentation/software-manuals/cces-sharccompiler-manual.pdf pages 2-310 through 2-324

[18] Jens Maurer. P1236R0 - Alternative Wording for P0907R4 Signed Integers Are Two's Complement. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1236r0.html