WG 14, N2817

Can Signed Integers Overflow?

David Svoboda
svoboda@cert.org

Date: 2021-09-27

The Problem

Traditionally, the C standard has two possible consequences when the result of a binary operation on two integers cannot be represented in the type of integer mandated by the usual arithmetic conversions (henceforth known as the result type). If the result type is unsigned, the result wraps (that is, its most significant bits are truncated, and the result indicates the low-order bits of the mathematical result). If the result type is signed, then the behavior is undefined (although traditionally the result wraps on twos-complement platforms).

Because of this, the question arose of how the word "overflow" should be used in relation to integer arithmetic in C. Note that this paper does not advocate changing the behavior of any C platforms; it is strictly concerned with terminology in general, and in particular, the term "overflow".

Section 6.2.5, paragraph 9 of the draft C23 standard (n2596) says:

> The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[44)] A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Section 6.2.6.2, paragraph 5 says:

> **NOTE 1** Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

These two paragraphs have caused controversy in WG14, because some members still speak of arithmetic operations on unsigned integers as "overflowing", in addition to the fact that unsigned arithmetic is known to wrap.

However, several papers and much discussion still apply the word "overflow" to unsigned integers. For example, N2629 "Towards Integer Safety", part of which was recently adopted into the forthcoming C23 standard. This document tries to apply the term "overflow" strictly to signed integers in the normative text. However, the informal text uses "overflow" to apply to signed and unsigned integers. It also makes use of an "overflow" flag (in the normative text) when trying to describe overflow on both

signed and unsigned integer arithmetic. This term underwent some evolution, having started off as "overflow", then changed to "exact", "inexact", and finally "overflow" again. It was argued that "overflow" was the most precise and best term to describe the condition of a result type being unable to represent the result of a binary mathematical operation, regardless of signedness. The portion of the proposal that was accepted was based on the Built-in Functions to Perform Arithmetic with Overflow Checking functions provided by Glibc and made available to GCC and Clang users. These functions, and the Glibc documentation consistently uses the term "overflow" for integer arithmetic, both signed and unsigned.

A straw poll was conducted at the August 2021 meeting. The question was:

> If the mathematical result of an operation on two unsigned integers is outside the range of the resulting type, should that constitute "overflow" as the term should be used in the standard?

The poll results were: 6 yes, 6 no, and 10 abstaining, which was ruled as no sentiment to change the definition of "overflow". At least one member of the committee suggested that they abstained because they wanted to be convinced by a future paper as to how to vote.

This paper hopes to persuade the committee how to resolve this issue.

Current Usage

The draft C standard (n2596) does not define the term 'overflow" for integers, but does define it for floating-point types, in section 7.12.1, paragraph 5:

> A floating result overflows if the magnitude (absolute value) of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type.

(N2805 touches on this definition but does not change it.)

The C standard does use the term "overflow" in not only integers and floating-point numbers, but also in the context of pointer arithmetic, including the notorious buffer overflow.

ISO 2382:2015 provides a general definition for arithmetic overflow, which clearly applies to fixed-size integers (as opposed to variable-sized integers like bignums):

> **arithmetic overflow**
> overflow
> <arithmetic and logic operations> portion of a numeric word expressing the result of an arithmetic operation by which its word length exceeds the word length provided for the number representation

The Argument: Overflow is an Error Condition

When we suggest that overflow is possible, that implies an error condition without specifying how the error is handled. For example, the phrase "buffer overflow" has become common parlance among C developers and cybersecurity professionals as an error condition. Yet the C standard leaves undefined the definition of what happens when a buffer overflows. Some platforms, such as Valgrind, do reliably detect buffer overflows and warn the user. Glibc also has some limited functionality to detect and warn about buffer overflows.

Typically, when cybersecurity professionals say that an overflow cannot happen, they are speaking about the circumstances surrounding a set of operations that could be expected to cause overflow. For example, the addition of two uint8_t values cannot overflow on a 32-bit platform, because they will first be promoted to uint32_t values, and the resulting 32-bit value can hold the sum (or difference or product) of any pair of 8-bit integer values.

There are many ways to handle overflow. For example, floating-point arithmetic has the values Infinity, -Infinity, and NaN. As noted earlier, buffer overflows, being undefined behavior, are not handled by the ISO C standard, and is therefore implicitly "handled" by each platform. And unsigned integer overflow in C is handled by wrapping, as is mandated by the standard.

To summarize, the concept of overflow is an error condition, and can happen in many contexts, including unsigned integer arithmetic. Each context that allows overflow must therefore address how that overflow is to be handled (even if it merely claims that it is undefined behavior).

We would therefore claim that wording that says "A computation involving unsigned operands can never overflow" is misleading using any definition of "overflow" in common usage. Unsigned operands can overflow, even though C defines precisely how this kind of overflow is handled (by wrapping).

The prevailing counterargument that unsigned operands cannot overflow arises from the notion that unsigned integers should not be thought of as analogous to the set of positive integers, but rather as a mathematical ring (the set of integers form 0 to 2^N-1, with wraparound behavior). This counterargument has two rebuttals: First, relatively few people are familiar with rings, while everyone is familiar with the infinite set of positive integers. And secondly, mathematical rings typically support multiplication but not division, whereas C unsigned integers do support both division and modulo.

Normative Text

We considered adding a standard definition for integer overflow to Chapter 3. However, that would ignore both floating-point overflows (and underflows) and pointer-arithmetic overflows. We could add a definition for "overflow" that encompasses all three, but it would still not handle floating-point underflows. While a solution could be achieved here, we did not deem it worth the effort.

There are 39 instances of the word "overflow" in the current draft standard. However, most of these address floating-point overflows, with relatively few addressing integer overflows. We judge that the only changes needed are the following:

The following wording proposed is a diff from WG14 N2596. Green text is new text, while ~~red text~~ is deleted text.

Change Section 6.2.5, paragraph 9 from:

> The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[44)] A computation involving unsigned operands ~~can never overflow,~~ because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

to:

> The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.[44)] A computation involving unsigned operands that overflows, because it produces a result that cannot be represented by the resulting unsigned integer type, exhibits wrap-around behavior. That is, the result is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Change Section 6.2.6.2, paragraph 5 from:

> **NOTE 1** Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

to:

> **NOTE 1** Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types because they exhibit wrap-around behavior when overflow occurs. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

Change Section 6.3.1.3, paragraphs 1 and 2 from:

1. When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged.

2. ~~Otherwise, if~~ the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

to:

1. When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged. Otherwise, an overflow has occurred.

2. If the value cannot be represented by the new type, and the new type is unsigned, then wrap-around occurs. That is, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

Acknowledgements

This proposal was suggested by Robert Seacord.

The following people reviewed this document and suggested improvements: David Goldblatt, Will Klieber, Robert Seacord.

This material is based upon work funded and supported by the U.S. Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

DM20-0381