Proposal for C23
WG14 N 2837

Title:	Clarifying integer terms v2
Author, affiliation:	Robert C. Seacord, NCC Group
Date:	2021-10-8
Proposal category:	Defect
Target audience:	Implementers
Abstract:	Clarify the use of integer terms in the C Standard
Prior art:	C

Clarifying integer terms

Reply-to: Robert C. Seacord (rcseacord@gmail.com)
Document No: **N 2837**
Reference Document: **N 2811**
Date: 2021-10-8

Change Log

2021-9-12:

- Initial version

2021-10-8:

- Changed definition of wraparound to also apply to signed values
- Changed the proposed language for checked integer operations

Introduction and Rationale

During a committee discussion of the behavior of calloc, it became clear that the committee lacks a consensus definition of terms such as "overflow" and "wraparound" that are commonly used to discuss integer arithmetic. The objective of this paper is to clarify the meaning of terms so that they can be used in the standard and communicating about C in a precise manner, and not to alter any existing normative behaviors.

The C99 Rationale, Subclause 6.2.5, "Types" states that:

> *The keyword* `unsigned` *is something of a misnomer, suggesting as it does in arithmetic that it is non-negative but capable of overflow. The semantics of the C type* `unsigned` *is that of modulus, or wrap-around, arithmetic for which overflow has no meaning. The result of an* `unsigned` *arithmetic operation is thus always defined, whereas the result of a signed operation may be undefined. In practice, on two's-complement machines, both types often give the same result for all operators except division, modulus, right shift, and comparisons. Hence there has been a lack of sensitivity in the C community to the differences between signed and unsigned arithmetic.*

The term overflow is a defined term of art in ISO/IEC 2382:2015, Information technology — Vocabulary which is included in Subclause 2, "Normative References", paragraph 2 as a normative reference:

2124111

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

Note 1 to entry: overflow; arithmetic overflow: terms and definition standardized by ISO/IEC [ISO 2382-2:1976].

Note 2 to entry: 02.07.03 (2382)

[SOURCE: ISO-2382-2 * 1976 * * *]

Overflow

The term overflow appears 49 times in the N 2596 working draft.

The following types of overflow in the standard:

1. Integer overflow

2. Pointer arithmetic overflow (Subclause 6.5.6, “Additive operators”, paragraph 9)

3. Floating-point overflow

4. Buffer overflow

We propose retaining the overflow term from ISO/IEC 2382 but preface it with “integer overflow” when specifically discussing integer overflow. Because most uses of the term “overflow” in the C Standard refer to floating-point overflow, we proposed continuing to use the unadorned term “overflow” to refer to floating-point overflow.

There are three separate behaviors that are considered integer overflow by the C Standard:

1a. Signed integer arithmetic that overflows.

1b. `INT_MIN % -1` which has undefined behavior (6.5.5#7) because `INT_MIN /-1` overflows.

1c. Out-of-range shift counts.

Some examples of the use of the term “overflow” from the C standard when specifically referring to integer overflow include:

“Provided the addition of two chars can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.”

“However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.”

“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.”

Of particular concern is the following text from subclause "6.2.5 Types" from paragraph 9:

> *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.*

The key phrase is "a result that cannot be represented by the resulting unsigned integer type". Because of this wording, the result has overflowed based on the ISO 2382 definition.

C++ states that "The range of representable values for the unsigned type is 0 to 2^N-1 (inclusive); arithmetic for the unsigned type is performed modulo 2^N (https://eel.is/c++draft/basic#fundamental-2).

C++ defines arithmetic for unsigned integers as never producing a value that cannot be represented. The C++ phrasing is consistent with ISO 2382 because it defines the arithmetic result of unsigned operations as not producing an overflow.

So our text is clearly wrong and contradictory. To maintain the intended meaning that unsigned integer arithmetic cannot overflow it can be changed to the following (consistent with C++):

> The range of representable values for the unsigned type is 0 to 2^N-1 (inclusive). A computation involving unsigned operands can never produce an overflow, because arithmetic for the unsigned type is performed modulo 2^N.

Having unsigned integers behave as a semiring preserves the notion that overflow is an error condition and that wraparound is well-defined behavior.

For checked integer operations [N2683], the operations are performed as if both operands were represented in a signed integer type with infinite range. This means that the operation cannot overflow an infinite number of bits.

Checked arithmetic produces a wrapped value. Normally conversion of out-of-range integers to a signed integer type is implementation-defined (6.3.1.3#3: "Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised."), so simply converting the value as if by assignment is not the intended behavior.

Pointer Arithmetic

Subclause 6.5.6, “Additive operators”, paragraph 9 uses the term “overflow” with respect to pointer arithmetic:

> *If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.*

It is unclear from this text if the resulting pointer was beyond the too-far element that overflow had occurred. You can also read this as saying that overflow can't happen if the resulting pointer points to the array. This reading would contradict the idea that if the pointer arithmetic produces overflow that this results in undefined behavior. The following formulation makes it clear how the overflow term is being used and exactly which behaviors are undefined:

> …If ~~both~~ the pointer operand and the result do not point to elements of the same array object~~,~~ or one past the last element of the array object, the ~~evaluation shall not produce an overflow; otherwise,~~ behavior is undefined. If the addition or subtraction produces an overflow, the behavior is undefined.

This change is severable from the remainder of this proposal.

Floating Point

Conversions from floating-point to unsigned integer can overflow.

For IEEE-754 floating-point, that out of range conversion raises `FE_INVALID`.

However, real implementations mostly do NOT detect that error.

See http://www.tybor.com/tflt2int.c

6.3.1.4 Real floating and integer

If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[65)]

[65)] The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. Thus, the range of portable real floating values is (`-1`, `Utype_MAX + 1`).

This paper does not propose any changes to the use of the term "overflow" with respect to floating point.

Buffer

The term buffer overflow is also used in Annex K of this paper to refer to write outside the bounds of an array. This used of the term overflow is not addressed by this paper.

Wraparound

ISO/IEC 2382 also contains two definitions of wraparound, neither of which are appropriate in the context of <arithmetic and logic operations>

2126020

wraparound

<computer graphics> displaying, at the opposite end of the display space, the part of a display image that would otherwise lie outside that display space

Note 1 to entry: wraparound: term and definition standardized by ISO/IEC [ISO/IEC 2382-13:1996].

Note 2 to entry: 13.03.29 (2382)

[SOURCE: ISO-IEC-2382-13 * 1996 * * *]

2126184

wraparound

<text processing> function that enables text entered after the last position on a line to be placed automatically at the beginning of the next line

Note 1 to entry: wraparound: term and definition standardized by ISO/IEC [ISO/IEC 2382-23:1994].

Note 2 to entry: 23.04.21 (2382)

[SOURCE: ISO-IEC-2382-23 * 1994 * * *]

The C Standard does not explicitly define any of these terms in Subclause 3, "Terms, definitions, and symbols". The definition of wraparound is implicit in the text.

We propose to add the following definition:

wraparound

the process by which a value is reduced modulo 2^N where N is the width of the resulting type

This is defined as a noun to be consistent with the ISO/IEC 2382 definition of overflow but can be used in various forms: wraparound (noun), wraps around (v, present tense), wrapped around (v, past tense), wrap-around arithmetic (adjective).

This change is severable from the remainder of this proposal.

Annex H

Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language independent arithmetic (LIA–1). The language independent (arithmetic) standards were developed by WG11 many years ago, and at the time was considered useful not only by WG11 members but were also supported by SC22 and several language developing WGs like WG14. There are three LIA standards, which are ISO/IEC 10967 Parts 1, 2, and 3 (nicknamed LIA-1, LIA-2, and LIA-3). None of them really fit C well, and after binding to LIA-1, WG 14 did not try to fit to the others.

Part of LIA-1 is an abstract version of IEEE floating point (without the bit representations). Because we now do a great job of binding directly to IEEE floating point, we do not need that part of LIA-1.

The integer part of LIA-1 only marginally fits C. It is consequently confusing for novice programmers who are trying to understand the C language. The confusion is compounded by the fact that LIA-1 overloads the term overflow, introducing a new additional meaning for it. It would also need to be maintained, which will cause extra work for the committee.

There are no known dependencies on Annex H. This proposal recommends removing Annex H from the standard. Removing Annex H will leave a gap in the Annex naming sequence. The resolution of this problem is an editorial matter, but could be addressed, for example, by moving Annex N (TS 18661-3 as Annex) so that all three floating-point Annexes are collocated.

This change is severable from the remainder of this proposal.

Proposed Wording

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

Add the following text as Subclause 3.22:

3.22

wraparound

the process by which a value is reduced modulo 2^N where *N* is the width of the resulting type

Subclause 5.1.2.3, "Program execution" paragraph 11:

… Provided the addition of two chars can be done without integer overflow, or with integer overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

Subclause 5.1.2.3, "Program execution" paragraph 15:

…On a machine in which integer overflows produce an explicit trap and in which the range of values representable by an int is [−32768, +32767], the implementation cannot rewrite this expression as… However, on a machine in which integer overflow silently generates some value and where positive and negative integer overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

Replace Subclause 6.2.5, "Types" paragraph 9 with the following paragraph:

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)] The range of representable values for the unsigned type is 0 to 2^N-1 (inclusive). A computation involving unsigned operands can never produce an 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.~~ arithmetic for the unsigned type is performed modulo 2^N.

Replace Subclause 6.2.6.2, "Integer types" paragraph 5 with the following paragraph:

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 integer 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.

Modify Subclause , "Additive operators", paragraph 9 as follows:

…If ~~both~~ the pointer operand and the result do not point to elements of the same array object~~,~~ or one past the last element of the array object, the ~~evaluation shall not produce an overflow; otherwise,~~ behavior is undefined. If the addition or subtraction produces an overflow, the behavior is undefined.

Modify Subclause 6.5.7, "Bitwise shift operators", paragraph 4 as follows:

The result of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are filled with zeros. If `E1` has an unsigned type, the value of the result is $E1 \times 2^{E2}$, wrapped around ~~reduced modulo one more than the maximum value representable in the result type~~. If `E1` has a signed type and nonnegative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

Modify Subclause 7.17.7.5, "The atomic_fetch and modify generic functions" paragraph 3 as follows:

…For signed integer types, arithmetic ~~is defined to use~~ performs silent ~~wrap around~~ wraparound on integer overflow; there are no undefined results…

Modify Subclause 7.24.1, "Checked Integer Operations" paragraph 5 as follows:

If these macros return `false`, the value assigned to `*result` correctly represents the mathematical result of the operation. Otherwise, these macros return `true`. In this case, the value assigned to `*result` is the mathematical result of the operation wrapped around to the width of `*result` ~~result reduced by modular arithmetic on two's complement representation with silent wrap around on overflow~~.

Remove Annex H (informative) Language independent arithmetic

4.0 Acknowledgements

I would like to recognize the following people for their help with this work: Aaron Ballman, David Keaton, David Svoboda, Joseph Myers, and Jens Gustedt.

5.0 References

[N2803] Draft Minutes for 27 and 30 – 31 August, 1 – 3 September, 2021. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2803.pdf

[N2683] N2683 2021/03/10 Svoboda, Towards Integer Safety (updates N 2681) URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2683.pdf