Integer Sets, v3

WG14 N 3699

Title:

Author, affiliation: Robert C. Seacord,
Woven by Toyota,
rcseacord@gmail.com

Date: 2025-9-2

Proposal category: Defect

Target audience: Implementers, users

Abstract: Reorganize integer sets

Prior art: C++

Integer Sets, v3

Reply-to: Robert C. Seacord (rcseacord@gmail.com)

Document No: **N 3699**

Reference Document: **N3550**

Date: 2025-6-16

This proposal aligns the definition of integer sets with C++.

Change Log

2025-6-16:

- Initial version 1.0.0

2025-7-5:

- Fixed intro text
- Allow for `_BitInt(1)`
- Clarified meaning of blue arrows
- Clarified underlying type for enumerations
- Fixed descriptions of basic type

2025-9-2:

- Removed new paragraph 20 after subclause “6.2.5 Types”
- Deleted “(the least significant bit)” and split paragraph into two paragraphs
- Simplified “6.7.3.2 Structure and union specifiers”, paragraph 12
- Eliminated UB for Rotate Left and Rotate Right type-generic functions WRT to negative signed integer values
- Repaired language for bit manipulation functions

Table of Contents

1 Problem Description

C++ groups integers into sets as follows:

> There are five *standard signed integer types*: “signed char”, “short int”, “int”, “long int”, and “long long int”.
>
> For each of the *standard signed integer types*, there exists a corresponding (but different) *standard unsigned integer type* : “unsigned char”, “unsigned short int”, “unsigned int”, “unsigned long int”, and “unsigned long long int”.
>
> Type `bool` is a distinct type that has the same object representation, value representation, and alignment requirements as an implementation-defined unsigned integer type.

C on the other hand groups integers into sets in “Subclause 6.2.5 Types” as:

> There are five *standard signed integer types*, designated as signed char, short int, int, long int, and long long int.
>
> There may also be implementation-defined extended signed integer types.31) The standard signed integer types, bit-precise signed integer types, and extended signed integer types are collectively called *signed integer types*.32)
>
> For each of the *signed integer types*, there is a corresponding (but different) *unsigned integer type*
>
> The type bool and the *unsigned integer types* that correspond to the *standard signed integer types* are the *standard unsigned integer types*.

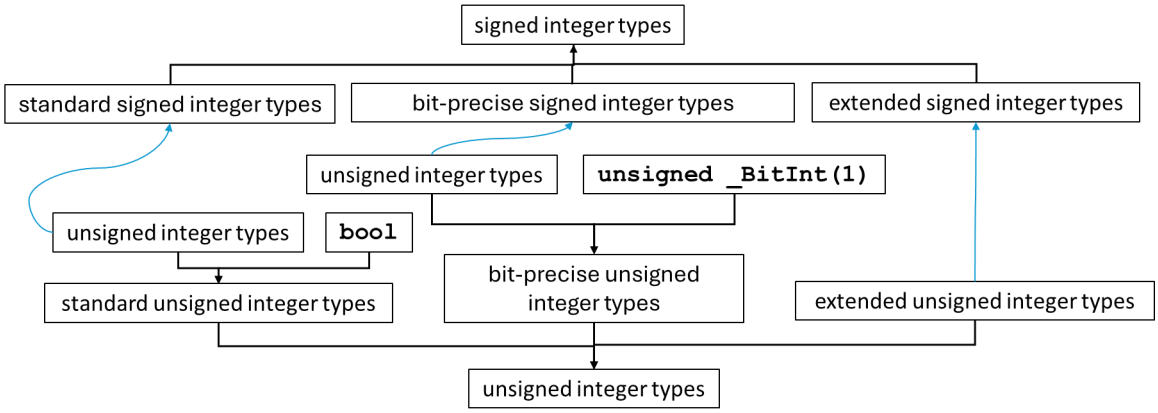
This means that in C `bool` is a standard unsigned integer type while in C++ it is not. This is very confusing for anyone including implementers, programmers, or educators when dealing with integer behavior where people assume similar if not identical behaviors. Furthermore, calling `bool` an unsigned type is misleading because it doesn't wrap around and has special conversion rules. For example, as an unsigned type you would expect `(bool)4` to be 0 not 1.

Another problem with the existing text is that the term “unsigned integer type(s)” is used before it is defined, and when it is defined it has a different meaning.

Because this proposal addresses many defects where the term unsigned integer type was incorrectly used to apply to the `bool` type, this proposal changed the semantics from C23.

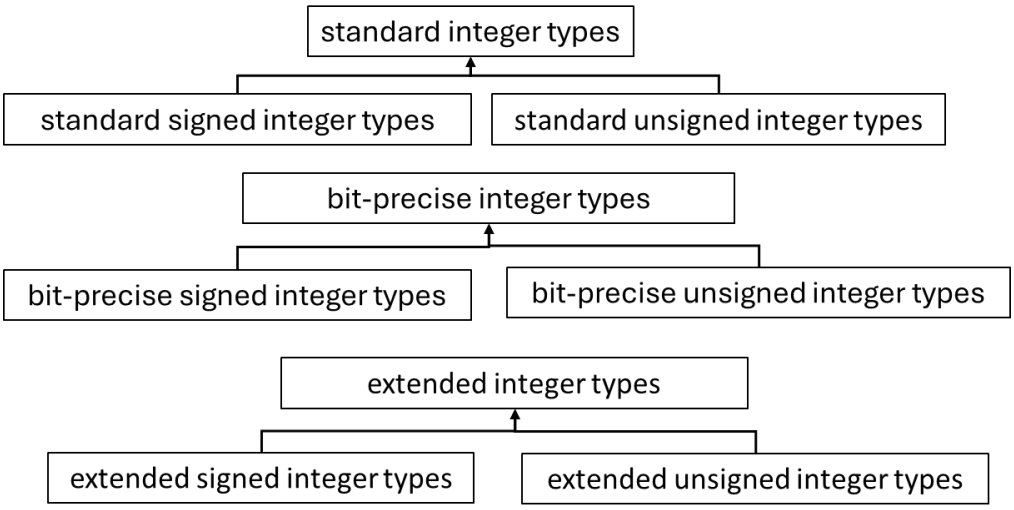
1.1 C Integer Types

N3550 working draft subclause 6.2.5 Types paragraph 4 through 8 define the following terms and relationships:

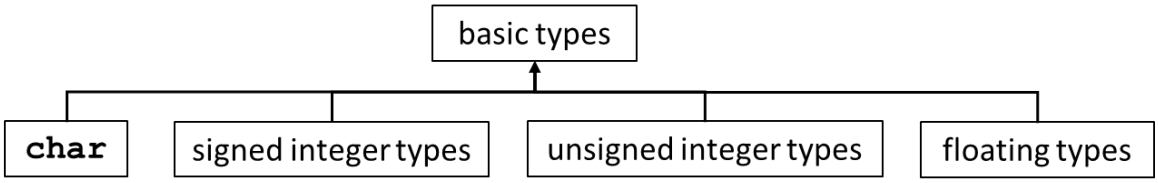


The blue arrows in the diagram mean “that correspond to”.

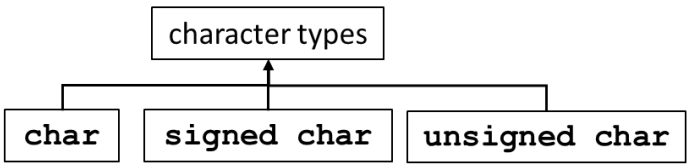
Paragraph 9 adds the following terms and relationships:



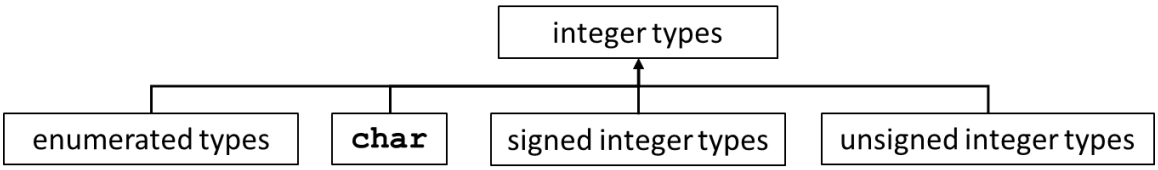
Paragraph 18 adds the following terms and relationships:



Paragraph 20 adds the following terms and relationships:



Paragraph 22 adds the following terms and relationships:



1.2 Bit-precise Integers

The user community has provided feedback that they would like `signed _BitInt(1)` to behave the same way as `struct S { signed int i : 1; };` works -- so it can hold the values 0 and -1. This is a more feasible approach now that C23 only supports two's complement. In two's complement, the most-significant bit (MSB) is a value bit with a negative weight, so an `_BitInt(1)` can have two values: -1 and 0.

One example of an application is splitting a wide product as a sum of multiple subproducts.

If we take the example of a product between two 33 bits signed integers a and b, on an FPGA that has a digital signal processor (DSP) able to compute 32x32 products, the product can be computed as follows:

- Each input is split between its MSB (`a_h` and `b_h`), and the remaining 32 low bits (`a_l` and `b_l`).
- The result is `((a_h x b_h) << 64) + ((a_h x b_l + b_h x a_l) << 32) + a_l x b_l`.
- This computation uses only one DSP for `a_l x b_l`, the three other products are done inexpensively on FPGA logic.

Both `a_h` and `b_h` are conceptually `_BitInt(1)` (one-bit integers that can be either 0 or -1). There is no difference with the unsigned case for the 1x1 product, but in the case of non-square products such as `a_h x b_l`, it is important that the product gives `-b_l` when the `a_h` bit is set, otherwise the result is false.

This example is in the case of a full square product and can look a bit artificial, but when computed truncated product (which makes sense on FPGA, as it is less expensive than computing the full product) the case of subproducts involving a 1-bit signed integer can also appear.

1.3 Preserve Existing Language with New Meaning

Generally speaking, the use of the term *unsigned integer type* used throughout the standard predates the introduction of the `bool` type and is not meant to apply to the `bool` type.

In subclause “5.3.5.3.2 Characteristics of integer types `<limits.h>` and `<stdint.h>`” paragraph 2, the `bool` type is included:

> For all unsigned integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix **_WIDTH** holding its width N, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that

> is representable by the type and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

Subclause 6.2.6.2 paragraph 1 states that "The type `bool` has one value bit and `(sizeof(bool)*CHAR_BIT) - 1` padding bits". The width of an unsigned type is the number of value bits. For the `bool` type, this is always one. Consequently, the specification of a **_WIDTH** for the `bool` type has dubious value, but is not wrong. Conceptually, a Boolean does not have a maximal value as it only stores the values false and true. The specification of a **_MAX** macro for the `bool` type is conceptually incorrect.

Removing `bool` from the set of *unsigned integer types* eliminates the requirement to provide the **BOOL_MAX** macro without changing subclause 5.3.5.3.2. To retain this requirement, `bool` will need to be explicitly added.

In C++, there is no concept of a **BOOL_MAX** constant for the bool data type, unlike integer types which have **INT_MAX**, **LONG_MAX**, etc., defined in `<climits>`.

While true and false can be implicitly converted to integer values (where true becomes 1 and false becomes 0), bool does not have a maximum numerical value in the same sense that an integer type does. Its range is simply limited to these two distinct logical states.

Removing **BOOL_MAX** from C2Y doesn't mean that implementations could no longer define this macro. According to subclase "7.35.15 Integer types <stdint.h>":

Macro names beginning with **INT** or **UINT** and ending with **_MAX**, **_MIN**, **_WIDTH**, or **_C** are potentially reserved identifiers and may be added to the macros defined in the `<stdint.h>` header.

This paper introduces the necessary changes to preserve the **BOOL_MAX** macro, largely for backwards compatibility.

Subclause 6.7.3.3 "Enumeration specifiers" paragraph 2 states:

> If it is not explicitly specified, the underlying type is the enumeration's compatible type, which is either `char` or a standard or extended signed or unsigned integer type.

However, **bool** is unnecessarily prohibited by paragraph 13 because it is no longer a member of the set of unsigned types:

> For all enumerations without a fixed underlying type, each enumerated type shall be compatible with **char** or a signed or an unsigned integer type that is not **bool** or a bit-precise integer type.

So this paragraph is modified by the proposed wording.

In subclause "7.18.3 Count Leading Zeros" paragraph 2, the `bool` type is included:

> The ***generic_return_type*** type shall be a suitably large unsigned integer type capable of representing the computed result.

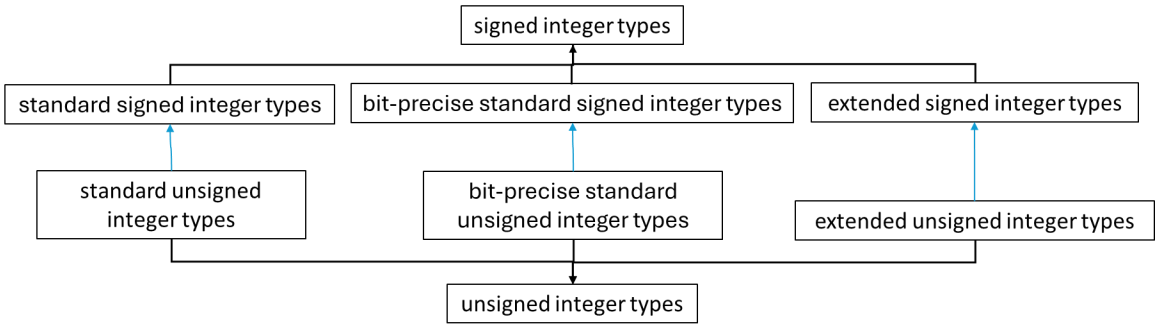
However, because the input type has at least width 8, the return type can never be `bool`. Consequently, removing `bool` from the set of unsigned integer types does not alter the semantics and the text can remain unchanged.

2 Proposal

This paper proposes removing the type `bool` from the set of *unsigned integer types* and from the set of *standard unsigned integer types* and adding the type `bool` to the set of *basic types* and the set of *integer types*.

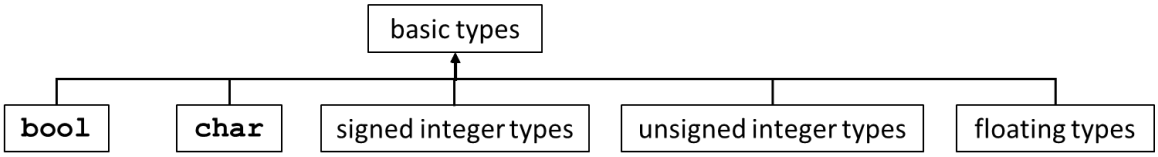
The type `unsigned _BitInt(1)` is eliminated as a separate type and is now simply treated as one of the *bit-precise unsigned integer types*.

The relationship between signed and unsigned integer types is therefore greatly simplified:

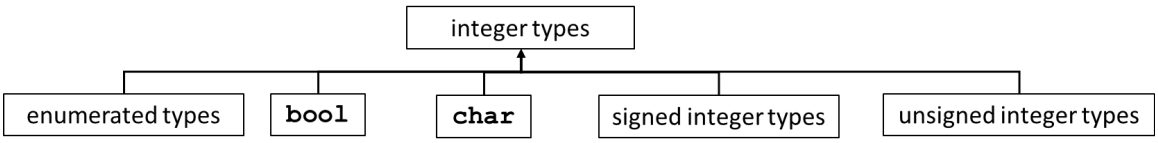


The blue arrows in the diagram mean “that correspond to”.

The type `bool` is added to the set of *basic types*.



The type `bool` is added to the set of *integer types*.



3 Proposed Text

Text in green is added to the C2Y working draft n3467. ~~Text in red~~ that has been struck through is removed from the C2Y working draft n3467.

Modify subclause “5.3.5.3.2 Characteristics of integer types <limits.h> and <stdint.h>”, paragraph 2:

For the type **bool** and all unsigned integer types for which `<limits.h>` or `<stdint.h>` define a macro with suffix **_WIDTH** holding its width *N*, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that is representable by the type and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. If the value is in the range of the type **uintmax_t** (7.23.2.6) the macro is suitable for use in conditional expression inclusion preprocessing directives.

Modify subclause “6.2.5 Types”, paragraph 5:

A bit-precise signed integer type is designated as **_BitInt(N)** where *N* is an integer constant expression that specifies the number of bits that are used to represent the type, ~~including the sign bit~~ which has the same representation as a signed integer type of width *N*. Each value of *N* designates a distinct type.

Modify subclause "6.2.5 Types", paragraph 8:

For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword unsigned) that uses the same amount of storage (including sign information) and has the same alignment requirements. The ~~type bool and the~~ unsigned ~~integer~~ types that correspond to the standard signed integer types are the standard unsigned integer types. The unsigned ~~integer~~ types that correspond to the extended signed integer types are the extended unsigned integer types. ~~In addition to the unsigned integer types that correspond to the bit-precise signed integer types there is the type unsigned _BitInt(1), which uses one bit to represent the type. Collectively, unsigned _BitInt(1) and t~~ The unsigned ~~integer~~ types that correspond to the bit-precise signed integer types are the bit-precise unsigned integer types. The standard unsigned integer types, bit-precise unsigned integer types, and extended unsigned integer types are collectively called unsigned integer types.[34)]

The ~~type bool and the~~ unsigned integer types that correspond to the standard signed integer types are the standard unsigned integer types.

Modify subclause "6.2.5 Types", paragraph 11:

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.[35)] The range of representable values for ~~the~~ unsigned integer types is 0 to $2^N - 1$ (inclusive). A computation involving unsigned operands can never produce an overflow, because arithmetic for ~~the~~ unsigned integer types is performed modulo 2^N.

Modify subclause "6.2.5 Types", paragraph 18:

The type bool, the type char, the signed and unsigned integer types, and the floating types are collectively called the basic types. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless distinct types.

Modify subclause “6.2.5 Types”, paragraph 22:

The type **bool**, the type **char**, the signed and unsigned integer types, and the enumerated types are collectively called integer types. The integer and real floating types are collectively called real types.

Split subclause “6.2.6.2 Integer types”, paragraph 1 into two paragraphs:

1 For unsigned integer types the bits of the object representation shall be divided into two groups: value bits and padding bits. If there are N value bits, each bit shall represent a different power of 2 between 1 and $2^{N}-1$, so that objects of that type shall be capable of representing values from 0 to $2^{N} - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified. The number of value bits N is called the width of the unsigned integer type.

2 The type **bool** shall have one value bit and **(sizeof(bool)*CHAR_BIT) - 1** padding bits. Otherwise, there is no requirement to have any padding bits; **unsigned char** shall not have any padding bits.

Modify subclause “6.3.2.1 Boolean, characters, and integers”, paragraph 1:

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type~~, if any~~.

— The rank of type **bool** shall be less than the rank of all other ~~standard~~ integer types.

Modify subclause “6.4.5.2 Integer literals”, paragraph 7:

wb or **WB**	**_BitInt**(N) where ~~the width~~ N is the smallest width ~~N greater than 1~~ which can accommodate the signed value ~~and the sign bit~~.	**_BitInt**(N) where ~~the width~~ N is the smallest width ~~N greater than 1~~ which can accommodate the signed value ~~and the sign bit~~.
Both u or **U** and **wb** or **WB**	**unsigned _BitInt(*N*)** where ~~the width~~ *N* is the smallest width ~~N greater than 0~~ which can accommodate the unsigned value.	**unsigned _BitInt(*N*)** where ~~the width~~ *N* is the smallest width ~~N greater than 0~~ which can accommodate the unsigned value.

Replace subclause “6.4.5.2 Integer literals”, paragraph 9 with:

EXAMPLE 2 The **wb** suffix results in an **_BitInt** which can accommodate the signed value of binary, octal, decimal, or hexadecimal literals.

```
0wb /* Yields a _BitInt(1) with the value 0 */

1wb /* Yields a _BitInt(2) */

-1wb /* Yields a _BitInt(2) that is arithmetically negated */
```

```
~0wb /* Yields a _BitInt(1) that has the value -1 */

-3wb /* Yields a _BitInt(3) that is arithmetically negated */

-0x3wb /* Yields a _BitInt(3) that is arithmetically negated */

3wb /* Yields a _BitInt(3) */

3uwb /* Yields an unsigned _BitInt(2) */

-3uwb /* Yields an unsigned _BitInt(2) that is arithmetically

negated, resulting in wraparound */

-4wb /* Yields an _BitInt(4) that is arithmetically negated */
```

Modify subclause “6.7.3.1 General”, paragraph 4:

The parenthesized constant expression that follows the **_BitInt** keyword shall be an integer constant expression N that specifies the width (6.2.6.2) of the type. The value of N for **_BitInt** and **unsigned _BitInt** shall be greater than or equal to 1~~. The value of N for _BitInt shall be greater than or equal to 2. The value of N shall be~~ and less than or equal to the value of **BITINT_MAXWIDTH** (see 5.3.5.3.2).

Modify subclause “6.7.3.2 Structure and union specifiers”, paragraph 12:

A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.[137] ~~If the value 0 or 1 is stored into a nonzero-width bit-field of type bool, the value of the bit-field shall compare equal to the value stored; a~~ A nonzero width **bool** bit-field has the semantics of a **bool**. A **bool** bit-field occupies the given number of bits.

Modify subclause “6.7.3.3 Enumeration specifiers”, paragraph 2:

If it is not explicitly specified, the underlying type is the enumeration’s compatible type~~, which is either char or a standard or extended signed or unsigned integer type~~.

Modify subclause “6.7.3.3 Enumeration specifiers”, paragraph 13:

For all enumerations without a fixed underlying type, each enumerated type shall be compatible with **char**, ~~or~~ a signed integer type, or an unsigned integer type ~~that is~~ but is not compatible with ~~bool or~~ a bit-precise integer type. The choice of type is implementation-defined,[142] but shall be capable of representing the values of all the members of the enumeration.[143]

Modify subclause “7.18.3 Count Leading Zeros”, paragraph 2:

The type-generic function *stdc_leading_zeros* (marked by its ***generic_value_type*** argument) returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.4 Count Leading Ones”, paragraph 2:

The type-generic function *stdc_leading_ones* (marked by its ***generic_value_type*** argument) returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.5 Count Trailing Zeros”, paragraph 2:

The type-generic function *stdc_trailing_zeros* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.6 Count Trailing Ones”, paragraph 2:

The type-generic function *stdc_trailing_ones* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.7 First Leading Zero”, paragraph 2:

The type-generic function *stdc_first_leading_zero* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.8 First Leading One", paragraph 2:

The type-generic function *stdc_first_leading_one* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.9 First Trailing Zero", paragraph 2:

The type-generic function *stdc_first_trailing_zero* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.10 First Trailing One", paragraph 2:

The type-generic function *stdc_first_trailing_one* ~~(marked by its generic_value_type argument)~~ returns the appropriate value based on the type of the input value, ~~so long as~~ provided that it is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.11 Count Zeros", paragraph 2:

The type-generic function *stdc_count_zeros* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.12 Count Ones", paragraph 2:

The type-generic function *stdc_count_ones* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.13 Single-bit Check", paragraph 2:

The type-generic function *stdc_has_single_bit* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause "7.18.14 Bit Width", paragraph 2:

The type-generic function *stdc_bit_width* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.15 Bit Floor”, paragraph 2:

The type-generic function *stdc_bit_floor* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.16 Bit Ceiling”, paragraph 2:

The type-generic function *stdc_bit_ceil* ~~(marked by its generic_value_type argument)~~ returns the previously described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is ~~a~~:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches a standard or extended integer type~~, excluding bool~~.

Modify subclause “7.18.17 Rotate Left”, paragraph 4:

The type-generic function `stdc_rotate_left` ~~(marked by its generic_value_type argument)~~ returns the previously ~~above~~ described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches any standard or extended integer type~~, excluding bool~~.

The ~~**generic_count_type**~~ `count` argument to the type-generic function `stdc_rotate_left` function shall be a ~~non-negative~~ value of signed or unsigned integer type, or **char**.

Modify subclause “7.18.18 Rotate Right”, paragraph 4:

The type-generic function stdc_rotate_right ~~(marked by its generic_value_type argument)~~ returns the previously ~~above~~ described result for a given input value ~~so long as~~ provided that the type of the ***generic_value_type*** argument is:

— a standard unsigned integer type~~, excluding bool~~;

— an extended unsigned integer type; or

— ~~or,~~ a bit-precise unsigned integer type whose width matches any standard or extended integer type~~, excluding bool~~.

The ~~**generic_count_type**~~ `count` argument to the type-generic function `stdc_rotate_left` shall be a ~~non-negative~~ value of signed or unsigned integer type, or **char**.

Modify subclause "7.25.2.8 The strtol, strtoll, strtoul, and strtoull functions", paragraph 5:

If the subject sequence begins with a minus sign, the resulting value is the negative of the converted value; for the strtoul and strtoull functions ~~whose return type is an unsigned integer type~~ this action is performed in the return type.

Modify subclause "7.33.1 Introduction", paragraph 4:

> The macros defined are **NULL** (described in 7.22); **WCHAR_MIN, WCHAR_MAX**, and **WCHAR_WIDTH** (described in 7.23);
>
> **WCHAR_UTF8**
>
> **WCHAR_UTF16**
>
> **WCHAR_UTF32**
>
> which expand to an expression of ~~signed or unsigned~~ integer type (that is potentially not an integer constant expression) whose value is nonzero if:
>
> — the wide execution encoding (6.2.9) is capable of representing every character in the required Unicode set;
>
> — the width of `wchar_t` is at least 8, 16, or 32 for UTF-8, UTF-16, or UTF-32, respectively;
>
> — and, the values of a sequence of `wchar_t` objects consumed and produced by related character functions have a values consistent with a sequence of code units of the UTF-8, UTF-16, or UTF-32 encodings, respectively;
>
> **MB_UTF8**
>
> **MB_UTF16**
>
> **MB_UTF32**
>
> which expand to an expression of ~~signed or unsigned~~ integer type (that is potentially not an integer constant expression) whose value is nonzero if:

Modify subclause "7.33.4.2.4 The wcstol, wcstoll, wcstoul, and wcstoull functions", para. 5:

If the subject sequence begins with a minus sign, the resulting value is the negative of the converted value; for the wcstoul and wcstoull functions ~~whose return type is an unsigned integer type~~ this action is performed in the return type.

4 Prior Art

These definitions more closely align the integer type system in C with C++.

5 Acknowledgements

I would like to recognize the following people for their help with this work: Aaron Ballman, Joseph Myers, JeanHeyd Meneide, Carlos Ramirez, Charles Hussong, Karsten Fischer, Vincent Mailhol, and Jens Gustedt.