Integer Comparison Macros

WG14 N3776

Author, affiliation: Robert C. Seacord,
Author, Effective C
rcseacord@gmail.com

Aaron Ballman
Intel
aaron@aaronballman.com

Date: 2026-1-5

Proposal category: Feature

Target audience: Implementers, users

Abstract: C implementation of C++ Utility functions

Prior art: C++, https://github.com/rcseacord/cmp_int

Integer Comparison Macros

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

Document No: **N3776**

Reference Document: **N3467, P0586R2**

Date: 2026-1-5

Change Log

2026-1-5:

- Initial version 1.0.0

Table of Contents

1 Problem Description

Signed / unsigned comparison is a major source of defects and software vulnerabilities in C and C++ programming.

For example, the following expression:

```
 -1 > 0u
```

evaluates to 1 which can be surprising to inexperienced programmers.

The following example is similar to the previous one, but uses objects with defined, but different integer types:

```
signed int x = -1;
unsigned int y = 1;
printf("beware the output of x < y is actually %d\n", x < y);
```

This paper proposes functions that provide integral comparisons that produce the mathematical result even when the operands have different types. For example, `stdc_less(-1, 0U)` is `true`, whereas `-1 < 0U` is `false` (due to arithmetic conversions).

This proposal is based on Integer Comparison Macros first released on Nov 9, 2021.

2 Proposal

C implementation of C++ Utility functions `cmp_equal, cmp_not_equal, cmp_less, cmp_greater, cmp_less_equal`, and `cmp_greater_equal`.

Initially, we are proposing that the following functions operate only on signed or unsigned integer type. The use of other types should be treated as a constraint violation and diagnosed.

```
bool stdc_equal(A a, B b);
bool stdc_not_equal(A a, B b);
bool stdc_less(A a, B b);
bool stdc_greater(A a, B b);
bool stdc_less_equal(A a, B b);
bool stdc_greater_equal(A a, B b);
```

These type generic macros compare the values of two integers `a` and `b`. Unlike comparison operators, negative signed integers always compare "less than" (and "not equal to") unsigned integers: the comparison is safe against lossy integer conversion.

```
-1 > 0u; // true
std::stdc_greater(-1, 0u); // false
```

It is a compile-time error if either `A` or `B` is not a signed or unsigned integer type.

```
stdc_greater_equal(1.0, -1.0) ? puts("true") : puts("false"); // type double, doesn't compile
```

C++ introduced integral comparisons that produce the mathematical result even when the operands have different types in P0586R2 to help address this problem.

The C++ functions on which this proposal is based compare standard integer types and extended integer types but they do not compare `bool`, character types, or enumerated types. This paper proposes that these functions support signed and unsigned integer types which include standard integer types, extended integer types, and bit-precise unsigned integer types. Bit-precise types are not currently supported in C++.

This paper is related to N3762 *Add type-safe minimum and maximum type-generic macros v3* by Gustedt and should be viewed as being in the same family of functions.

2.1 Supported Types

There is considerable debate about which types should be supported by the “integer” comparison macros.

The Integer Comparison Macros on which this proposal is based is restricted to signed and unsigned integer types.

This excludes enumeration types, plain `char`, and the `bool` type.

Ordered comparison over bool makes sense. equal and not equal make sense, but the comparisons seem problematic. Rather than support the type in some but not all operations, I think it's better to follow C++ and disallow it until there's sufficient justification to support it.

```
// type _Bool, doesn't compile
stdc_greater((_Bool)0, (_Bool)1) ? puts("true") : puts("false");
```

Supporting only explicitly signed or unsigned `char` makes for a better feature. `char` is always going to be one of the two, so I can see the appeal to wanting to support the type, but it’s still problematic. `stdc_less(ch, value)` returning `true` on one platform and `false` on another may meet some developers expectations, but I think in general, `char` is a weird beast and it's better to not support it, at least initially.

```
stdc_greater((char)-1, (char)1) ? puts("true") : puts("false"); // type char, doesn't
compile
```

Enumeration types in C are considered compatible with one of the integer types, but they are distinct types in themselves and not inherently signed or unsigned at the type level. C++ disallows enumeration types because enums provide extra semantic information to an integer type and there's no way (in general) to know whether two enumeration types are comparable in any way. e.g., `enum Category { Fruit, Meat, Vegetable }` is not comparable with `enum Color { Red, Green, Blue }` even if the integer values are comparable.

Using an enumeration type might initially be a constraint violation. Users who really want to do this can cast to an integer type. If this is happening a lot in practice, we can relax the restriction later. But if we allow the comparison initially, we're stuck supporting it forever.

Looking at existing practice is not particularly useful, as existing type-generic functions vary widely.

Because this proposal is related to N3762 it should support the same types. N3762 currently differs from this proposal in that it proposes that the type-safe minimum and maximum type-generic macros operate on both the `bool` and `char` types.

Other type-generic macros such as those in subclause 7.20.2 Checked Integer Operation Type-generic Macros that supports checked integer arithmetic on integer types other than "plain" char, bool, bit-precise integer types, or an enumerated types. This differs from this proposal in the lack of support for bit-precise integer types.

3 Proposed Text

Add the following subclauses to 7.25 in the C2Y working draft n3467:

7.25.7.4 Signed and unsigned integer comparison type-generic macros

Synopsis

```
1 #include <stdlib.h>
  bool stdc_equal(A a, B b) [[unsequenced]];
  bool stdc_not_equal(A a, B b) [[unsequenced]];
  bool stdc_less(A a, B b) [[unsequenced]];
  bool stdc_greater(A a, B b) [[unsequenced]];
  bool stdc_less_equal(A a, B b) [[unsequenced]];
  bool stdc_greater_equal(A a, B b) [[unsequenced]];
```

Description

2 These type-generic macros determine whether the mathematical value of `a` is equal, not equal, strictly less, strictly greater, less than or equal, or greater than or equal to `b`. These operations are applicable to pairs of values independent of their type or representation.

3 Both `A` and `B` shall be a signed or unsigned integer type. If both arguments are integer constant expressions, the macro invocation is also an integer constant expression.

Returns

4 The integer comparison type-generic macros return `true` if the corresponding mathematical relationship between the values holds, `false` if it does not.

4 Polling Questions

1. Should these type generic macros also apply to `char`, `bool`, and `enum` types?
2. Should these type generic macros also apply to floating types?
3. Should these be ICE if the arguments are ICE?
4. Should these macros be applicable within `#if` evaluation?
5. Should we add macros to return function pointers?

5 Acknowledgements

We would like to recognize the following people for their help with this work: Jens Gusted and Anton Gerasimov.