Document: P1619R1
Date: 2019-10-04
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: SG6 (Numerics), LEWG

Functions for Testing Boundary Conditions on Integer Operations

Lisa Lippincott

Abstract

The integer operations in C++ have boundary conditions that may readily be encountered by novices. Unfortunately for those novices, expressing these conditions in the language requires detailed knowledge of the language, a degree of mathematical subtlety, and considerable care. I propose that we add library functions that name and express these conditions more simply and directly, in a form conducive to use in assertions.

For SG6: Since Cologne, the functions `can_bitwise_{not, and, or, xor}[_modular]` have been added. Their descriptions are somewhat more complex than the others; please check my math.

For LEWG: This is the first time the paper has been presented to LEWG.

Changes from r0 to r1: In Cologne, SG6 voted to pursue this for C++23, with parallel inclusion into a future Numerics TS (which will be formally based on C++20). SG6 requested that functions corresponding to bitwise operations be included, noting that they may be non-trivial for mixed-sign or in-place operations.

A question was also raised as to the interpretation of 7.6.18[expr.ass]¶3, describing assignments:

> The expression is implicitly converted to the cv-unqualified type of the left operand.

I have taken this to apply to both compound and simple assignments, with "the expression" referring to the right-hand operand. Others have suggested that it applies only to simple assignments, or that "the expression" refers to the whole assignment expression. This is being addressed in CWG issue 2399. Only the motivating text, figures, and examples are affected by the interpretation of this paragraph; the proposed wording is unaffected.

1 Introduction

Integer arithmetic is one of the most elementary aspects of C++ programming, but the rules governing integer arithmetic are arcane. Both experts and novices would be helped by a simple mechanism that predicts whether an arithmetic operation will produce the expected result.

Keeping the mechanism simple presents a challenge. There are many integer operations, and these may be applied to many integral types. In addition, the binary operations come with an additional in-place form, which has different boundary conditions. And finally, the operations are used for both modular and non-modular arithmetic, without making the intent of the programmer apparent in the syntax.

The C++ standard describes the behavior of arithmetic operations as a composition of promotions, conversions, and arithmetic. But a parallel compositional approach fails to simplify user-facing tests, because the list of operations to be composed is itself rather arcane. As witness to the complexity, I illustrate in the diagram below my understanding of the steps involved in a binary expression; each step shown may affect the value of the result. As witness to the arcane nature, I note that I have already had to correct the diagram, and I think it's likely that I will have to correct it again as people point out its flaws.

To reduce the complexity of performing these tests, I instead focus on connecting the action of whole expressions to the intent of the programmer. This approach requires a large number of functions, but I attempt to reduce cognitive load through uniform naming and semantics.

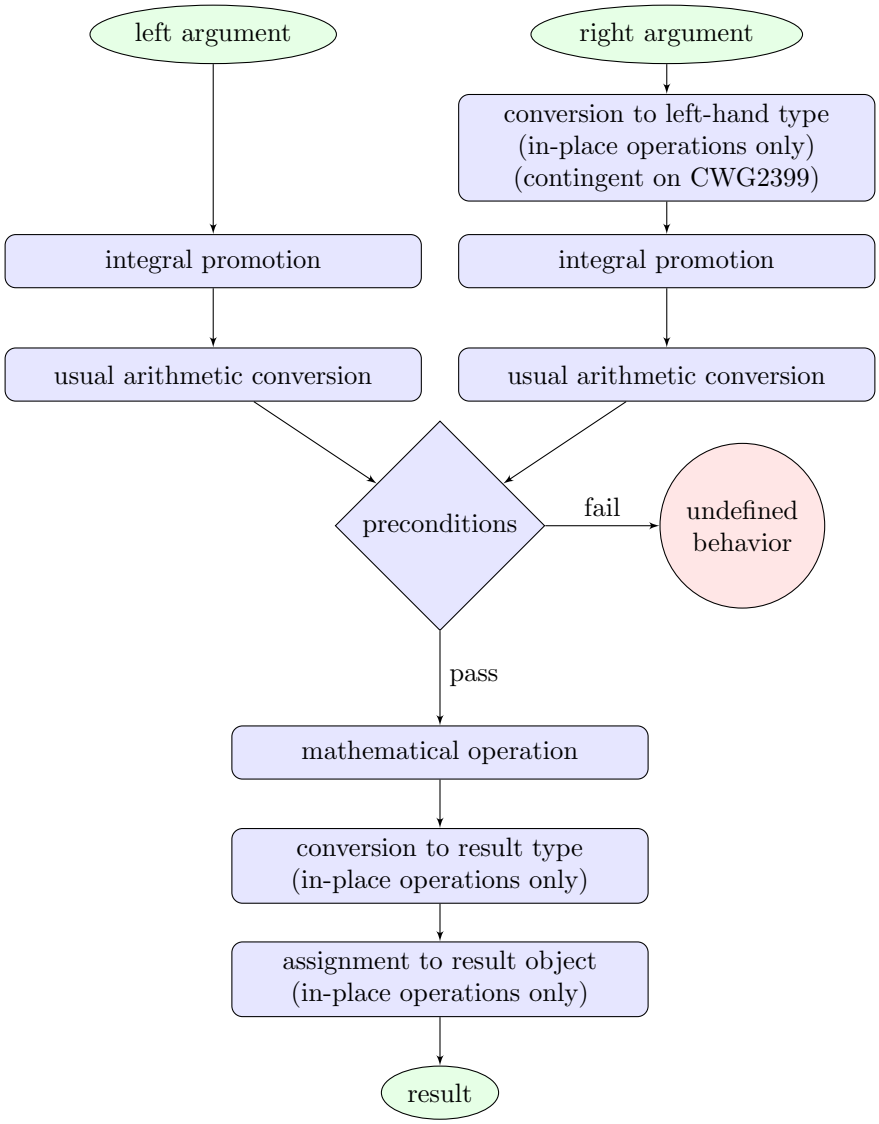

Figure 1: Steps in a binary expression. Steps in rectangles may affect the result.

2 Wording

```
template <class R, class A> constexpr bool can_convert( A a ) noexcept;
template <class R, class A> constexpr bool can_convert_modular( A a ) noexcept;

template <class A> constexpr bool can_increment( A a ) noexcept;
template <class A> constexpr bool can_decrement( A a ) noexcept;
template <class A> constexpr bool can_promote( A a ) noexcept;
template <class A> constexpr bool can_negate( A a ) noexcept;
template <class A> constexpr bool can_bitwise_not( A a ) noexcept;

template <class A> constexpr bool can_increment_modular( A a ) noexcept;
template <class A> constexpr bool can_decrement_modular( A a ) noexcept;
template <class A> constexpr bool can_promote_modular( A a ) noexcept;
template <class A> constexpr bool can_negate_modular( A a ) noexcept;
template <class A> constexpr bool can_bitwise_not_modular( A a ) noexcept;

template <class A, class B> constexpr bool can_add(A a, B b) noexcept;
template <class A, class B> constexpr bool can_subtract(A a, B b) noexcept;
template <class A, class B> constexpr bool can_multiply(A a, B b) noexcept;
template <class A, class B> constexpr bool can_divide(A a, B b) noexcept;
template <class A, class B> constexpr bool can_remainder(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_left(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_right(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_and(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_xor(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_or(A a, B b) noexcept;

template <class A, class B> constexpr bool can_compare(A a, B b) noexcept;

template <class A, class B> constexpr bool can_add_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_subtract_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_multiply_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_left_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_right_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_and_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_xor_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_or_modular(A a, B b) noexcept;

template <class A, class B> constexpr bool can_add_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_subtract_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_multiply_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_divide_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_remainder_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_left_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_right_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_and_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_xor_in_place(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_or_in_place(A a, B b) noexcept;

template <class A, class B> constexpr bool can_add_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_subtract_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_multiply_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_shift_left_in_place_modular(A a, B b) noexcept;
```

```
template <class A, class B> constexpr bool can_shift_right_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_and_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_xor_in_place_modular(A a, B b) noexcept;
template <class A, class B> constexpr bool can_bitwise_or_in_place_modular(A a, B b) noexcept;
```

These functions shall not participate in overload resolution unless all template arguments are integral types.

These functions express the conditions under which C++ integer arithmetic expressions model mathematical operations in the integers. The correspondence between functions, expressions, and mathematical operations is given by the table below. Functions ending in `_modular` correspond to the same expressions and operations as their non-modular counterparts, but relate C++ arithmetic to arithmetic in the integers modulo 2^N. The notations $a \otimes b$ and $a \oplus b$ here denote the bitwise conjunction and bitwise exclusive disjunction of infinitely sign-extended two's complement integers; that is, the unique integers such that, for each nonnegative integer i,

$$\left\lfloor \frac{a \otimes b}{2^i} \right\rfloor \cong \left\lfloor \frac{a}{2^i} \right\rfloor \left\lfloor \frac{b}{2^i} \right\rfloor \text{ modulo } 2 \qquad \left\lfloor \frac{a \oplus b}{2^i} \right\rfloor \cong \left\lfloor \frac{a}{2^i} \right\rfloor + \left\lfloor \frac{b}{2^i} \right\rfloor \text{ modulo } 2$$

Function	**Expression**	**Mathematical Operation**
`can_convert<R>`	`static_cast<R>(a)`	a
`can_increment`	`++a`	$a + 1$
`can_decrement`	`--a`	$a - 1$
`can_promote`	`+a`	a
`can_negate`	`-a`	$-a$
`can_bitwise_not`	`~a`	$-1 - a$
`can_add`	`a+b`	$a + b$
`can_subtract`	`a-b`	$a - b$
`can_multiply`	`a*b`	$a \cdot b$
`can_divide`	`a/b`	a/b, truncated toward zero
`can_remainder`	`a%b`	remainder from truncation of a/b toward zero
`can_shift_left`	`a<<b`	$a \cdot 2^b$
`can_shift_right`	`a>>b`	$a/2^b$, truncated downward
`can_bitwise_and`	`a&b`	$a \otimes b$
`can_bitwise_xor`	`a^b`	$a \oplus b$
`can_bitwise_or`	`a|b`	$(a \otimes b) + (a \oplus b)$
`can_add_in_place`	`a+=b`	$a + b$
`can_subtract_in_place`	`a-=b`	$a - b$
`can_multiply_in_place`	`a*=b`	$a \cdot b$
`can_divide_in_place`	`a/=b`	a/b, truncated toward zero
`can_remainder_in_place`	`a%=b`	remainder from truncation of a/b toward zero
`can_shift_left_in_place`	`a<<=b`	$a \cdot 2^b$
`can_shift_right_in_place`	`a>>=b`	$a/2^b$, truncated downward
`can_bitwise_and_in_place`	`a&=b`	$a \otimes b$
`can_bitwise_xor_in_place`	`a^=b`	$a \oplus b$
`can_bitwise_or_in_place`	`a|=b`	$(a \otimes b) + (a \oplus b)$
`can_compare`	`a==b`, `a<b`, `a<=b`,	$a = b$, $a < b$, $a \le b$
	`a!=b`, `a>b`, `a>=b`,	$a \ne b$, $a > b$, $a \ge b$, respectively

Each of these functions compares the hypothetical evaluation of the expression listed in the table above to evaluation of the corresponding mathematical operation as performed in the integers. Letting N be the range exponent of the result type of the expression, the functions return `true` when all of the following conditions hold, and `false` otherwise.

- If evaluated, the expression would have defined behavior according to this standard.

- For each operand, the values before and after integral promotions and conversions are applied would be congruent modulo 2^N. Further, for each operand of a comparison, `/`, `/=`, `%`, `%=`, `>>`, or `>>=`, and for each right-hand operand of `<<` or `<<=`, the values before and after integral promotions and conversions are applied would be equal.

- The result of the expression and the result of the mathematical operation would be congruent modulo 2^N. Further, for functions not ending in "`_modular`", the result of the expression and the result of the mathematical operation would be equal.

[*Note:* The arguments to the hypothetical expression are the parameters of the function, not the arguments used to initialize those parameters. This means that the widths of bit-field arguments are not considered:

```
struct S { int b: 3 };
S s = { 7 };
if ( can_increment( s.b ) )   // true, but the bit-field can't represent 8
   ++s.b;                     // s.b has an implementation-defined value
```

—*end note*]

3 Design Questions and Answers

Why not directly express the preconditions for the expressions? There is no point in meeting the precondition of a function if it will nevertheless produce a useless answer. Therefore, these functions ensure that the corresponding expression will produce a result fit for a particular purpose.

Why are there separate functions for modular and non-modular arithmetic? In C++, the same expressions are used for both modular and non-modular arithmetic. The intended form of arithmetic can only be determined by the programmer. Separating modular from non-modular arithmetic allows the programmer to express that intent.

Why do these functions all consider promotions and conversions? To consider conversions and promotions separately, a programmer would have to have a full understanding of the promotions and conversions that occur in the evaluation of an expression. This would create a trap for less expert (or less attentive) programmers, who could easily perform tests that do not model the C++ expressions they intend to use.

Why are there separate functions for in-place arithmetic? In an in-place operation, the result of the operation is converted to the type of the left-hand operand. The additional conversion may affect the boundary conditions.

Aren't `can_divide` and `can_remainder` the same function? They always compute the same value. But having separate functions is less trouble than teaching people to use `can_divide` as the check for remainder. Also, they need not be the same function in all future versions of C++; we could reasonably extend the remainder operation to provide the mathematically correct remainder even when the quotient would be out of range.

Why aren't there functions `can_divide_modular` or `can_remainder_modular`? Unfortunately, there are two possible meanings for those names. A programmer might reasonably expect them to refer to integer division and remainder with the result reduced modulo 2^N, so that 23 divided by 12 is 1 with a remainder of 11. A mathematician might reasonably expect division and remainder in $\mathbb{Z}/2^N$, so that (for $N = 32$) 23 divided by 12 is 0x55555557 with a remainder of 3. As the operators `/` and `%` are described perfectly well by `can_divide` and `can_remainder`, it seems like the best course is to avoid these confusing names.

Are shift operations really arithmetic? Arguably. But more importantly, left and right shift have preconditions on the right operand that are easily violated, and we should have functions to express the

preconditions. The precondition for right shift is `can_shift_right`, and the precondition for left shift is `can_shift_left_modular`. (These preconditions apply to the promoted and converted operands.)

Why isn't `can_shift_left` the precondition for `<<` for signed integers? Don't ask me. I'm not the one who decided to allow one modular operation on signed integers. But `can_shift_left_modular` expresses the precondition for `<<` on all integer types, while `can_shift_left` is more strict.

Isn't `can_shift_right_modular` the precondition for `>>`? Yes, it is. The functions `can_shift_right` and `can_shift_right_modular` always produce the same result. It seems easier to use a consistent naming scheme than to have everyone remember that some names are redundant.

Why are there functions `can_convert_modular` and `can_promote_modular`? By my understanding, these functions will always return true, so they have minimal utility. But it seems easier to include the functions than to have everyone remember that the functions do not exist.

What's so special about the operands of `/`, `%`, `>>`, and the right-hand operand of `<<`? A difference of 2^N in these operands may change the result by an amount that is not a multiple of 2^N. This leads to situations where code produces a mathematically correct result for obscure reasons. Calculating whether such obscure reasons pertain may be difficult, and does not seem helpful.

```
short s = 6;      // Assuming 16-bit shorts
s %= 0x12340005;  // C++ result: 1. Integer remainder: 6.
s = 6;
s %= 0x12340006;  // C++ result: 0. Integer remainder: 6.
s = 6;
s %= 0x12340007;  // C++ result: 6. Integer remainder: 6.
s = 6;
s >>= 0x12340008; // Defined behavior, but please don't do this!
```

Why not test for all operands being unaltered by promotions and conversions? For most operands, allowing differences of 2^N does not complicate boundary conditions, and may be a reliable intermediate step toward a correct result. In particular, negative integers may reliably be added to unsigned integers of greater or equal magnitude (*e.g.* `-5 + 15u` is `10u`).

Don't all promotions and conversions leave operands unaltered modulo 2^N? For some value of N, yes. But not necessarily for the value of N relevant to the calculation at hand. In a perverse implementation, integral promotion to `unsigned` may alter the value of an operand. If the result type of the expression has range exponent larger than that of `unsigned`, the promotion may alter the result of the calculation is bizarre ways. Consider an implementation with these types:

Type	Size	Range exponent	Padding bits
short	8	48	16
int	8	32	32
unsigned	8	32	32
long	8	64	none

In such an implementation, `short` promotes to `unsigned`, making negative values of `short` quite difficult to use:

```
short a = -1;
a += 1;      // 0x100000000s, surprisingly positive
```

Can this scheme be made to work better with bit-fields?

I don't see how any purely library solution can account for bit-fields. The type system effectively lies about the types of bit-fields, giving the library no way to account for their limited width.

Can this scheme be extended to enumerated types? Yes, in at least four ways, each with its own problems. The fourth choice requires the least effort.

- Ignore the possibility of operators overloaded for the enumerated type, and always convert the enumerated type to its underlying type.
- Ignore the possibility of operators overloaded for the enumerated type, but consider only the usual promotions and conversions.
- Provide a templated extension mechanism that authors can use to match the behavior of their overloaded operators.
- Just let people overload these names in their own namespaces, and rely on dependent lookup.

Can this scheme be extended to floating point types? Perhaps not in a very useful manner. On implementations that support them, infinities, signed zeroes, and NaNs may better express the boundary conditions of floating point operations.

Can this scheme be extended to check full expressions? I expect these functions could be building blocks for such a system. One approach would be to use expression templates, and a second approach would be to create a set of types which can separately represent exact results, results reduced modulo 2^N for various N, and NaNs.

Can these functions be used to implement a natural number type? Yes, I expect they can.