More Natural Arithmetic in C++

Document number:	P0999R0
Date:	2018-04-01

Reply-to:	James Dennett <jdennett@google.com>

Audience:	SG6, SG12, EWG, CWG.

Synopsis

A minor change to the rules of arithmetic for unsigned types makes them behave as programmers expect and reduces the amount of unnecessarily undefined behavior in C++ code.

Guiding Principles

- Least astonishment: code should be unsurprising to reasonably well-informed readers.
- There should be no room for a lower-level language between C++ and assembler.
- Undefined behavior should be present only when justified (e.g., by performance, by practicality, for conceptual integrity, or by allowing better runtime diagnostics of unintended behaviors).

Background

A brief recap: C++ has modular arithmetic types (known as “unsigned”) which wrap around rather than overflowing, and has non-wrapping arithmetic types (known as “signed”) for which overflow is undefined. Arithmetic modulo n (for a positive integer n > 1) produces a well-known commutative ring (with identity), with multiplication distributive over addition as expected. C++ also has optional signed types that use two’s complement representations: [atomics.types.int] provides that for the signed integral specializations of `atomic<T>`, two’s complement shall be used (”For signed integer types, arithmetic is defined to use two’s complement representation.”).

At the Jacksonville 2018 meeting of WG21, EWG showed renewed openness to addressing some longstanding quirks in how signed and unsigned types work together, going so far as to take a direction poll in favor of moving towards a world in which `-1 < 0u`. Given this optimism that we are in a position to incrementally improve fundamental aspects of the language, the time may be right for the present paper to offer a path towards harmonizing one more aspect of the signed and unsigned types.

Other papers (such as p0907r0) have proposed changes to require changes to the signed types to require them to always use 2’s complement representations, and in some cases advocated

that they should have well-defined (wrapping) behavior instead of overflow. This paper does not directly address signed types, though if they were to adopt modular arithmetic in the future then it would apply to them just as it does for the existing (unsigned) modular types. The changes proposed by this paper would reduce the gap between unsigned types and signed types.

Unsigned types have an unambiguous expression in bits, one that's so obvious that mentioning it seems almost redundant: a number *x* in the range $0...2^n-1$ is represented by the bit pattern $<b_{n-1}, b_n, \ldots, b_0>$ where $x = \Sigma_i b_i 2^i$. This gives rise to a natural correspondence between certain operations on bit patterns and arithmetic on the numbers represented by those bit patterns. For example, [expr.shift]p3 says: "The value of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a non-negative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$." There is some elegance (if also some indirection) to defining the semantics of bit shifts via this correspondence. Bit shifts work (for unsigned types) work as they should, and give arithmetically unsurprising results (per the principle of least astonishment).

A Non-Motivating Example

We include this example (that we do not propose to change) for comparison to the motivating example presented below.

```
double d = 3 / 4;
```

This is a common error for novice programmers, who expect the resulting value for d to be 0.75 and are surprised when it is 0. For those who have not previously worked in a language where integral division truncates by default, it is easy forget that the `/` operator on the type `int` does so. While a not infrequent problem, this is viewed largely as an educational challenge rather than as a defect in the programming language (though some other languages with different tradeoffs make different choices here, such as using different operators for floating point division versus truncating integral division). Experienced C++ developers understand that floating point division behaves very differently than integral division, even while addition, subtraction and multiplication are largely the same between integral and floating point types (e.g., `3 + 4 == 3.0 + 4.0`, even though `3 / 4 != 3.0 / 4.0`).

A Motivating Example

In a context in which the semantics of an unsigned integer `i` are that of a number, rather than a bit pattern, a well-meaning programmer with experience in other programming languages might transform

```
  unsigned j = i >> 2;
```

into the apparently equivalent expression (that arguably expresses intent more directly)

```
  unsigned j = i / 4;
```

For a non-negative value `i` of a *signed* type, this transformation is (of course) valid.

Discussion

When we look at arithmetic on unsigned types, all is well for the additive operations and for multiplication (which distributes over addition and is completely defined by that property and the fact that 1 is the identity for the multiplicative group of non-zero elements).

Indeed from [expr.shift]p3 the definition of the right-shift operator `>>` on unsigned numbers ensures that the first form (`i >> 2`) yields "the integral part of the quotient of E1/2^{E2}"[1] where E2 is 2, i.e., the integral part of the number `i` divided by 4.

Recall that, building on a solid mathematical foundation, [basic.fundamental]p4 says "Unsigned integers shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer." The laws of modular arithmetic are familiar to children under the name "clock arithmetic", and are well understood and uncontroversial. At a more advanced level, modular arithmetic underlies much of modern cryptography, where the result that (for positive integers `a` and `b`) `a` is invertible modulo `b` iff they are coprime plays a key role.

The binary `/` operator is defined (for arithmetic types) by [expr.mul]p4: "The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. For integral operands the / operator yields the algebraic quotient with any fractional part discarded; if the quotient a/b is representable in the type of the result, (a/b)*b + a%b is equal to a; otherwise, the behavior of both a/b and a%b is undefined."

Herein lies the problem this paper aims to address: the rules of modular arithmetic imply that the transformation in our motivating example creates a use of the `/` operator with undefined behavior. Our well-meaning programmer has produced a simpler expression of the intent to divide by four, but forgetting that the operator has modular semantics they have assumed that it is real division, or at least not modular division.

More explicitly, given our unsigned integer `i`, `i / 4` is defined as the quotient, i.e., the result of dividing `i` by `4`. Division in the modular ring $\mathbb{Z}/2^n\mathbb{Z}$ is a partial operation defined by the identity $a/b = ab^{-1}$, where the notation b^{-1} refers to the multiplicative inverse of `b` and by convention juxtaposition indicates multiplication. Cleary this is not defined when `b` is zero, as we would expect. More precisely, it is undefined whenever `b` has a common factor with 2^n, or equivalently when `b` is even. Our divisor 4 is (manifestly) even, and hence the expression `i / 4` is undefined.

---

[1] In case the typesetting is unclear: this is E1/2^{E2} in followed by double quotes, i.e. it is E1/2^{E2}", not E1/$2^{E2"}$.

One obscure optimization is enabled by the use of modular division: If `a` and `b` have unsigned types, a conforming compiler can replace `a % b` by the constant 0 (using the fact that the expression has undefined behavior if the result of `a / b` is not representable, and is 0 when `a / b` is defined (because $(a / b)*b = (a * b^{-1})*b = a*(b^{-1} * b) = a*1 = 1$).

Proposed Fix

At the cost of some backwards incompatibility and the loss of a possible optimization, we can make unsigned division in C++ less surprising, more generic, and more useful:

- Change the semantics of division on unsigned types to give the same value as if the operation were performed in a sufficiently large signed integral type. (Note: this is similar to the resolution of CWG 1457 in which C++14 requires certain signed shifts to be evaluated as if they are performed in the corresponding unsigned type. Here we cannot use "the corresponding" signed type, as we need a signed type large enough to represent the maximum value of the unsigned type, and in general such a type might not be available on a given implementation.)
- Update Annex C to note that this is a breaking change for code that depends on unsigned division with divisors other than 1, as division will no longer be injective and hence not invertible.

This will also change the current behavior of the `%` operator on unsigned types, but the existing behavior where `a%b` is zero if b is odd and undefined otherwise appears not to have any applications, and needn't be preserved for generic code as it's not consistent with how `%` behaves for non-modular types: code using the C++17 rules already needs to know whether it's working with a modular type or not. The proposed behavior would make `%` more useful in code that handles unsigned types specifically (making it yield values consistent with how it behaves for signed types), and would also improve consistency for generic code.

Optionally, we might provide an operator that gives access to the legacy, modular behavior such as via a new function template in the standard library:

> `template <typename T> T std::modular_inverse<T>(T n);`
> Requires: `n` is odd.
> Returns: `m` such that `n*m == 1`

which allows modular division to be implemented as `a/b == a*std::modular_inverse(b)`. Alternatively the library could provide `std::modular_divide(a, b)`; given that `std::modular_inverse(n) == std::modular_divide(1, n)` and `std::modular_divide(a,b) == a*std::modular_inverse(b)`, either of these is trivially implementable in terms of the other, and the library could be specified to provide either or both.

Alternatives Considered

1. We could fix the motivating example by defining `>>` in terms of the modular `/` operator, such that `a >> b` is (by definition) `a / 2^b`. A limitation of this approach is that while it

ensures the desired equivalence, it leaves `a >> b` undefined unless b is zero, somewhat restricting the situations in which it is applicable (to those in which the shift is a no-op).

2. Instead of changing the definition of the `%` operator for unsigned types, we could restrict unsigned integral types to being represented as `p` bits such that 2^p-1 is (Mersenne) prime. Reserving one bit pattern `111…111` for a trap representation leaves 2^p-1 remaining values which form a division ring (in fact, a field) under (+,*), ensuring that the modular division required by C++17 is defined for all non-zero denominators. Compared to our proposed solution this has the advantage of not tampering with the definition of division and building C++'s unsigned types on the well-understood mathematical foundation of finite fields. It does, however, fail to address the divergence between signed and unsigned division, leaving unsigned operations surprising to non-expert users.
3. Following P0997 ("Retire Pernicious Language Constructs in Module Contexts"), we could change the behavior of unsigned integral division to truncation rather than modular semantics only in module contexts, assuming that a notion of module context is adopted into C++. That would be a somewhat radical difference between module and non-module contexts, and might violate this paper's guiding principle of "least astonishment."

Acknowledgements

The author would like to thank Richard "The Smith" Smith for review and for the observation that Mersenne primes are kind of cool.

References

P0907R0 "Signed Integers are Two's Complement", JF Bastien.
P0997R0 "Retire Pernicious Language Constructs in Module Contexts", Nathan Myers.