Document number: P1856R0
Revises:
Date: 2019-10-07
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: LEWG, LWG
Reply to: Vincent Reverdy
University of Illinois at Urbana-Champaign
vince.rev@gmail.com

Bit operations do not work on bytes: a generic fix

Abstract

The current wording of low level bit manipulation functions specified by P0553R4: Bit operations and by P0556R3: Integral power-of-2 operations make these functions unusable with `std::byte`. We suggest a generic and extensible mechanism to fix this limitation inspired by P0237R10: Wording for fundamental bit manipulation utilities. Instead of limiting the functions to unsigned integer types, we suggest to introduce: (1) a type trait that acts as a customization point for user-defined types behaving like machine words and (2) a type trait to check whether a type is a machine word. This removes the current limitation and allows advanced users to provide their own word types. Both of these traits have been used for years as part of the bit library that serves as a basis of P0237 which is currently under review by LWG for a later revision of the C++ standard.

Contents

1 Proposal

1.1 Tony tables

Before

```
// Initialization
unsigned char c = 42;
unsigned int u = 42;
std::byte b{42};
int result = 0;

// Ispow2
result = std::ispow2(c);
result = std::ispow2(u);
//result = std::ispow2(b);        // Does not compile

// Ceil2
result = std::ceil2(c);
result = std::ceil2(u);
//result = std::ceil2(b);         // Does not compile

// Floor2
result = std::floor2(c);
result = std::floor2(u);
//result = std::floor2(b);        // Does not compile

// Log2p1
result = std::log2p1(c);
result = std::log2p1(u);
//result = std::log2p1(b);        // Does not compile

// Rotl
result = std::rotl(c);
result = std::rotl(u);
//result = std::rotl(b);          // Does not compile

// Rotr
result = std::rotr(c);
result = std::rotr(u);
//result = std::rotr(b);          // Does not compile

// Countl0
result = std::countl_zero(c);
result = std::countl_zero(u);
//result = std::countl_zero(b);   // Does not compile

// Countl1
result = std::countl_one(c);
result = std::countl_one(u);
//result = std::countl_one(b);    // Does not compile

// Countr0
result = std::countr_zero(c);
result = std::countr_zero(u);
//result = std::countr_zero(b);   // Does not compile

// Countr1
result = std::countr_one(c);
result = std::countr_one(u);
//result = std::countr_one(b);    // Does not compile

// Popcount
result = std::popcount(c);
result = std::popcount(u);
//result = std::popcount(b);      // Does not compile
```

After

```
// Initialization
unsigned char c = 42;
unsigned int u = 42;
std::byte b{42};
int result = 0;

// Ispow2
result = std::ispow2(c);
result = std::ispow2(u);
result = std::ispow2(b);

// Ceil2
result = std::ceil2(c);
result = std::ceil2(u);
result = std::ceil2(b);

// Floor2
result = std::floor2(c);
result = std::floor2(u);
result = std::floor2(b);

// Log2p1
result = std::log2p1(c);
result = std::log2p1(u);
result = std::log2p1(b);

// Rotl
result = std::rotl(c);
result = std::rotl(u);
result = std::rotl(b);

// Rotr
result = std::rotr(c);
result = std::rotr(u);
result = std::rotr(b);

// Countl0
result = std::countl_zero(c);
result = std::countl_zero(u);
result = std::countl_zero(b);

// Countl1
result = std::countl_one(c);
result = std::countl_one(u);
result = std::countl_one(b);

// Countr0
result = std::countr_zero(c);
result = std::countr_zero(u);
result = std::countr_zero(b);

// Countr1
result = std::countr_one(c);
result = std::countr_one(u);
result = std::countr_one(b);

// Popcount
result = std::popcount(c);
result = std::popcount(u);
result = std::popcount(b);
```

1.2 Problem description

C++17 introduced `std::byte` as proposed by P0298: A byte type definition as a vocabulary type for the smallest adressable entity that can be used to store bits. As suggested by P0237R0: On the standardization of fundamental bit manipulation utilities for such types, `std::byte` is only equiped with bitwise operations, unlike `unsigned char` which was previously used to represent at the same time a storage of bits, a character, and an unsigned integer. In this context, C++20 is expected to introduce a new `<bit>` header as originally proposed by P0237. The current draft of the C++ standard N4830 provides the following functionalities as part of this header: `bit_cast` as proposed by P0476: Bit-casting object representations, `ispow2`, `ceil2`, `floor2`, `log2p1` as proposed by P0556: Integral power-of-2 operations, and `rotl`, `rotr`, `countl_zero`, `countl_one`, `countr_zero`, `countr_one`, `popcount` as proposed by P0553: Bit operations. In all the following we focus on the last two groups of functionalities.

Currently, the standard draft specifies that the functions of the last two groups shall not participate in overload resolution unless the type of the argument is an unsigned integer type. In the standard, an *unsigned integer type* is, according to [basic.fundamental], either a *standard unsigned integer type* (unsigned char, unsigned short int, unsigned int, unsigned long int, unsigned long long int), or an implementation-specific *extended unsigned integer type*. The good thing is that bool and char are not included in this category even if the two standard type traits std::is_integral_v and std::is_unsigned_v may return true for these types. The bad thing, however, is that std::byte, which was purposefully introduced to model a storage of bits, is not included either. And on top of that, specifying these functions in terms of unsigned integer types brings us back to the pre-std::byte world, where unsigned integer types were used to mean several things at the same time. The whole motivation behind the standardization of std::byte was to break the ambiguity and provide a way to express whether a type is just a collection of bits or whether it should be seen as an integer. Typically, in the context of std::memcpy, the integral interpretation of the collection of bits is irrelevant: it's a pure memory operation. In generic code, it can be important for users to convey this information through types. It makes the code clearer, and far easier to debug. Again, this was one of the main motivation when we voted std::byte in the C++17 standard.

1.3 Proposed solution

So if low level bit operations should not operate on unsigned integer types, what should they operate on? We argue that the whole problem arises from the fact that the standard library is currently missing a simple but essential concept for bit operations: *words* (as in *machine words*). The whole machinery of bit proxy types and bit iterators introduced by P0237, currently under review by LWG and probably targeting C++23, relies around this notion. Also, to solve the problem described here for low level bit operations, we propose to introduce two simple type traits to make machine words a part of the standard library. This, we feel, represents, a very minor change that can provide far cleaner and more robust foundations to the <bit> header, as well as a clean evolution path for the next revision of the standard.

For the exact same reason that integer values are irrelevant to std::memcpy, integer values are irrelevant to operations like counting bits or rotating bits. For counting or rotating bits, as well as for all low level bit operations, the only two things one needs in order to provide an implementation are (1) the size of the collection of bits and (2) bitwise operators: left shift << and <<=, right shift >> and >>=, bitwise and & and &=, bitwise or | and |=, bitwise xor ^ and ^=, and bitwise not ∼, which happen to be the set of operators provided for std::byte. As a consequence we propose the two following traits:

- template <class T> struct binary_digits that acts as a customization point to specify that a type should be treated as a machine word by specifying its size in bits. The standard would provide specializations for (optionally cv-qualified) standard unsigned integer types and std::byte. Implementations could provide specializations for extended unsigned integer types. Advanced users could provide their own specializations. Note that the trait std::binary_digits_v<std::byte> would finally provide a modern C++ replacement of CHAR_BIT. As a remark, the name of the trait here is the one provided in P0237 and approved by LEWG, but this can be changed. Alternative names may include (machine_)word_(digits, size, bits, bitsize, bit_size, bitcount, bit_count, width, length).
- template <class T> struct is_word that returns whether a type should be treated as a machine word, that is: (1) binary_digits<T>::value is defined, (2) T is equiped with bitwise operators, and (3) can be converted to standard unsigned integer types. Alternative names may include is_machine_word. Whether or not the standard library should specify a corresponding concept is left as an open question. To provide a unique interface for integer conversion in bit manipulation context, we suggest to overload the existing std::to_integer

template function that exists for bytes.

And that is all. These two type traits would be sufficient to solve the current problem in a generic, robust, and extensible way, as well as to provide a clean and consistent evolution path to <bit>.

1.4 Design options

In order to be exhaustive, we list below the potential options available regarding the design of the two type traits:

- Should std::binary_digits and std::is_word be part of:
  - <bit> because it's directly related to bit manipulation
  - <type_traits> because it's where other type traits are
  - <limits> because it's where std::numeric_limits<T>::digits is
  - <memory> because words are related to memory manipulation
- Should the dependency of std::is_word on std::binary_digits:
  - require only std::binary_digits<T>::value to be defined
  - require std::binary_digits<T>::value to be defined and be such that the condition std::binary_digits_v<T> > 0 is true
- What should be the type of std::binary_digits_v<T>?
  - std::size_t as approved by LEWG in P0237R10
  - int as the current return type of the low level bit operation functions specified by P0556 and P0553 (std::popcount and the other functions listed in this paper)
  - std::ptrdiff_t as the type that is most likely to be returned by standard algorithms like std::count when executed on sequence of bits through bit iterators
  - std::intmax_t as an alternative to std::ptrdiff_t since a sequence of bits can contain more bits than the maximum number of addressable bytes
  - std::uintmax_t as an unsigned alternative to std::intmax_t
  - a standard integer type left to the implementation
  - a standard signed integer type left to the implementation
  - a standard unsigned integer type left to the implementation
  - other alternatives not listed in this proposal
- Should std::to_integer be extended with a generic overload that can take any type T as an input such that
  - std::is_integral_v<T> is true as an input?
  - static_cast<IntType>() is true as an input?

  that can take any type T such that std::is_integral_v<T> is true as an input?
- What should be the name of std::binary_digits?
  - std::binary_digits, originally based on std::numeric_limits<T>::digits and approved by LEWG in P0237R10
  - std::word_digits to keep the pattern of std::numeric_limits<T>::digits but highlight the fact that it is a customization point for word types
  - std::machine_word_digits to highlight the fact that we are speaking about machine words and not words made of characters
  - std::word_width or std::machine_word_width as *width* is often used in computer science and engineering to refer to the size of a register in bits
  - std::word_size or std::machine_word_size to highlight that it returns the size of a word
  - std::binary_word_size, std::word_bit_size, or std::machine_word_bit_size to make it clear that the size is expressed in bits
  - std::word_bit_count, or std::machine_word_bit_count to avoid using the term

    size as it means something specific in the standard library related to containers and ranges
  - other alternatives not listed in this proposal
- What should be the name of `std::is_word`?
  - `std::is_word` as suggested here
  - `std::is_machine_word` to highlight the fact that we are speaking about machine words and not words made of characters
  - `std::is_binary_word` to highlight the fact that we are speaking about words for bits and binary manipulation and not words made of characters
  - other alternatives not listed in this proposal
- Should the standard library provide a `concept` corresponding to the `std::is_word` type trait?
- If the standard library provides a machine word `concept`, should the low level bit operation functions specified by P0556 and P0553 (`std::popcount` and the other functions listed in this paper) be formally constrained by it?
- If the standard library provides a machine word `concept`, what should be its name?
  - `std::word` to correspond to a `std::is_word` type trait
  - `std::machine_word` to correspond to a `std::is_machine_word` type trait
  - `std::binary_word` to correspond to a `std::is_binary_word` type trait
  - other alternatives not listed in this proposal

2 Acknowledgements

This work has been made possible thanks to the National Science Foundation through the awards CCF-1647432 and SI2-SSE-1642411.