Project: Programming Language C++, Core Working Group, Library Working Group
Document number: P0298R3
Date: 2017-03-03
Reply-to: Neil MacIntosh neilmac@microsoft.com

# A byte type definition

## Contents

# Changelog

## Changes from R0

- Following advice from LEWG, expanded range of bitwise operations to include compound operations (|=, &= etc) and to appropriately qualify all operations as constexpr and noexcept.
- Added wording for operations on byte.
- Corrected minor wording errors and removed use of non-standard extensions in wording for byte operations.
- Simplified wording describing type in Alternative B following feedback from CWG.
- Merged in wording content from P0257 as directed by CWG.
- Increased wording changes to include `std::byte` in 8.5.12 where appropriate, as suggested by CWG.
- Added editing instruction to update P0137R1 to include std::byte should both that paper and this paper end up being accepted for inclusion in the standard.
- Remove Alternative B (implementation-defined type) after straw polling in LWG
- Corrected wording based on review by LWG
- Added note about lack of implicit conversion in boolean contexts.

## Changes from R1

- Updated wording to include 1.8p3 in response to NB comments CA 11 and US 72.
- Updated reference to the Working Draft of the standard to be N4606.
- Removed reference to P0137R1 as it has been incorporated into N4606.

## Changes from R2

- Updated reference to the Working Draft of the standard to be N4618 and corrected wording to be changes against that document.
- Added Appendix D wording to ensure `::byte` is not declared when a user includes <stddef.h>.

# Introduction

The core of this proposal is to introduce a distinct type implementing the concept of byte as specified in the C++ language definition. The proposal suggests a very simple definition of that type, named `std::byte`. It is argued that C++17 is expressive enough for a simple library definition, as opposed to a keyword (however uglified). An additional small fix is suggested to allow `std::byte` to embody an aliasing property for access to arbitrary object representations. Taken altogether, `std::byte` is just as “builtin” as if it was designated by dedicated keyword (which would have to be ugly or incur source breaking changes.); an illustration of C++’s expressive power.

# Motivation and Scope

Many programs require byte-oriented access to memory. Today, such programs must use either the `char`, `signed char`, or `unsigned char` types for this purpose. However, these types perform a "triple duty". Not only are they used for byte addressing, but also as arithmetic types, and as character types. This multiplicity of roles opens the door for programmer error – such as accidentally performing arithmetic on memory that should be treated as a byte value – and confusion for both programmers and tools.

Having a distinct byte type improves type-safety, by distinguishing byte-oriented access to memory from accessing memory as a character or integral value. It improves readability. Having the type would also make the intent of code clearer to readers (as well as tooling for understanding and transforming programs). It increases type-safety by removing ambiguities in expression of programmer's intent, thereby increasing the accuracy of analysis tools.

# Design Decisions

## std::byte is not an integer and not a character

The key motivation here is to make byte a distinct type – to improve program safety by leveraging the type system. This leads to the design that `std::byte` is not an integer type, nor a character type. It is a distinct type for accessing the bits that ultimately make up object storage.

As its underlying type is `unsigned char`, to facilitate bit twiddling operations, convenience conversion operations are provided for mapping a byte to an unsigned integer type value. They are provided through the function template:

```
namespace std {

template <class IntegerType> // constrained appropriately
  IntegerType to_integer(byte b);

}
```

This supports code that, for example, wants to produce a integer hash of an object to take a sequence of `std::byte`, convert them to integers and then perform arithmetic operations on them as needed.

Conversely, conversions in the other direction, e.g. from an integer value to `std::byte` type is now made simpler, and safer with the relaxation of enum value construction rule in C++17. Now, you can just write `std::byte{i}` when `i` is a non-negative integer value no more than `std::numeric_limits<unsigned char>::max()`.

During a previous review by CWG at the Spring 2015 meeting in Jacksonville, FL, John Spicer suggested an alternative definition that would leave much to "implementation defined" to avoid dependency on `<cstddef>`. That dependency is not a substantial issue in practice, for any interesting program. So, we do not recommend that path as it would add complexity to the wording and definition of `std::byte`.

## std::byte is storage of bits

A byte is a collection of bits, as described in 1.7/1. std::byte would provide operations that allow manipulation of the bits that it contains. The definition suggested below is kept deliberately simple with a minimum interface.

As illustration, these operations would be declared as follows:

```
namespace std {

// IntType would be constrained to be true for is_integral_v<IntType>
template <class IntType>
  constexpr byte& operator<<=(byte& b, IntType shift) noexcept; template
<class IntType>
  constexpr byte operator<<(byte b, IntType shift) noexcept; template
<class IntType>
  constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
template <class IntType>
  constexpr byte operator>>(byte b, IntType shift) noexcept;

constexpr byte operator|(byte l, byte r) noexcept;
constexpr byte& operator|=(byte& l, byte r) noexcept;

constexpr byte operator&(byte l, byte r) noexcept;
constexpr byte& operator&=(byte& l, byte r) noexcept;

constexpr byte operator~(byte b) noexcept;

constexpr byte operator^(byte l, byte r) noexcept;
constexpr byte& operator^=(byte& l, byte r) noexcept;
}
```

Their semantics would be the simple ones you might expect for a non-numeric type that is a collection of bits. Right-shift will shift bits rightwards, filling trailing places with zero bits. Left-shift will shift bits leftwards, filling vacated places with zero bits.

Similarly, std::byte can be compared, as comparing and ordering instances is a sensible and useful operation. Given its underlying storage type, the comparison operators would give the same results as if performed on the underlying type.

Note that because std::byte is a scoped enumeration there is no way to overload an operator to provide implicit conversions in boolean contexts. It would be possible to add implicit conversion in boolean contexts, by further modifying core language wording but that is left beyond the scope of the present paper. Users can still test variables of std::byte type in boolean contexts, but must use explicit comparisons to do so:

```
constexpr std::byte zero_byte{0x00};
```

```
if (b != zero_byte)   do_something();
```

Because `std::byte` uses `unsigned char` as its underlying type, initialization from an integer literal is possible, which is a convenience feature.

```
// example of initializing a byte, relying on syntax proposed for C++17 byte
b { 0x01 };
```

## Implementation Alternatives

Two possible definitions of `std::byte` were originally considered:

```
// Alternative A:
namespace std {
    enum class byte : unsigned char {};
}
```

and

```
// Alternatve B:
namespace std {
    using byte = /* implementation defined */;
}
```

We find alternative A easier to present and explain to ordinary programmers. For all practical purposes, it has the right semantics. After presenting both alternatives to the Library Working Group, straw polling showed unanimous support for the Alternative A, which is the form presented in this paper.

## Proposed Wording Changes

The following proposed changes are relative to N4618 [1].

1.8 The C++ object model [intro.object]

3 If a complete object is created (5.3.4) in storage associated with another object e of type "array of N `unsigned char`" or "array of N `std::byte`", that array provides storage for the created object if:

—(3.1) the lifetime of e has begun and not ended, and

—(3.2) the storage for the new object fits entirely within e, and

—(3.3) there is no smaller array object that satisfies these constraints.

### 3.8 Object lifetime [basic.life]

6 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. For an object under construction or destruction, see 12.7. Otherwise, such a pointer refers to allocated storage (3.7.4.2), and using the pointer as if the pointer were of type `void*`, is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

—(6.1) the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*,

—(6.2) the pointer is used to access a non-static data member or call a non-static member function of the object, or

—(6.3) the pointer is implicitly converted (4.10) to a pointer to a virtual base class, or

—(6.5) the pointer is used as the operand of a `static_cast` (5.2.9), except when the conversion is to pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to ~~either~~ *cv* `char`, ~~or~~ *cv* `unsigned char`, ~~or~~ *cv* `std::byte` (18.2.10), or

### 3.9 Types [basic.types]

2 For any object (other than a base-class subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (1.7) making up the object can be copied into an array of `char`, ~~or~~ `unsigned char`, or `std::byte` (18.2.10). If the content of that ~~the~~ array ~~of `char` or `unsigned char`~~ is copied back into the object, the object shall subsequently hold its original value.

### 3.10 Lvalues and rvalues [basic.lval]

8 If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:

—(8.1) the dynamic type of the object,

—(8.2) a cv-qualified version of the dynamic type of the object,

—(8.3) a type similar (as defined in 4.4) to the dynamic type of the object,

—(8.4) a type that is the signed or unsigned type corresponding to the dynamic type of the object,

—(8.5) a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

—(8.6) an aggregate or union type that includes one of the aforementioned types among its elements or nonstatic data members (including, recursively, an element or non-static data member of a subaggregate or contained union),

—(8.7) a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

—(8.8) a `char`, ~~or~~ `unsigned char`, ~~type~~ or `std::byte` type.

5.3.4 New [expr.new]

11 When a new-expression calls an allocation function and that allocation has not been extended, the new-expression passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of `char`, ~~and~~ `unsigned char`, and `std::byte`, the difference between the result of the *new-expression* and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (3.11) of any object type whose size is no greater than the size of the array being created. [Note: Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. —end note]

8.6 Initializers [dcl.init]

12 If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an *indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced (5.18). [Note: Objects with static or thread storage duration are zero-initialized, see 3.6.2. –end note] If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:

(12.1) – If an indeterminate value of unsigned narrow character type (3.9.1), or `std::byte` type (18.2.10) is produced by the evaluation of:

(12.1.1) – the second or third operand of a conditional expression (5.16),

(12.1.2) – the right operand of a comma expression (5.19),

(12.1.3) – the operand of a cast or conversion to an unsigned narrow character type (4.7, 5.2.3, 5.2.9, 5.4), or `std::byte` type (18.2.10), or
(12.1.4) – a discarded-value expression (Clause 5),
then the result of the operation is an indeterminate value.

(12.2) – If an indeterminate value of unsigned narrow character type, or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator (5.18) whose first operand is an lvalue of unsigned narrow character type, or `std::byte` type, an indeterminate value replaces the value of the object referred to by the left operand.

(12.3) – If an indeterminate value of unsigned narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.

(12.4) – If an indeterminate value of unsigned narrow character type, or `std::byte` type is produced by the evaluation of the initialization expression when initializing an object of `std::byte` type, that object is initialized to an indeterminate value.

18.2 Common definitions [support.types]
18.2.1 Header <cstddef> synopsis [cstddef.syn]

```
namespace std {
     using ptrdiff_t = see below ;
     using size_t = see below ;
     using max_align_t = see below ;
     using nullptr_t = decltype(nullptr);
     enum class byte : unsigned char {} ;
     template <class IntegerType>
       constexpr byte& operator<<=(byte& b, IntegerType shift) noexcept;
     template <class IntegerType>
       constexpr byte operator<<(byte b, IntegerType shift) noexcept;

      template <class IntegerType>
        constexpr byte& operator>>=(byte& b, IntegerType shift) noexcept;
      template <class IntegerType>
        constexpr byte operator>>(byte b, IntegerType shift) noexcept;

      constexpr byte& operator|=(byte& l, byte r) noexcept;
      constexpr byte operator|(byte l, byte r) noexcept;
      constexpr byte& operator&=(byte& l, byte r) noexcept;
      constexpr byte operator&(byte l, byte r) noexcept;
      constexpr byte& operator^=(byte& l, byte r) noexcept;
      constexpr byte operator^(byte l, byte r) noexcept;
      constexpr byte operator~(byte b) noexcept;

      template <class IntegerType>
         constexpr IntegerType to_integer(byte b) noexcept;

}
     #define NULL see below
     #define offsetof(P, D) see below
```

18.2.5 `byte` type operations [support.types.byteops]

```
template <class IntegerType>
  constexpr byte& operator<<=(byte& b, IntegerType shift) noexcept;
```

Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntegerType>` is `true`.

Effects: Equivalent to: `return b = byte(static_cast<unsigned char>(b) << shift);`

```
template <class IntegerType>
  constexpr byte& operator>>=(byte& b, IntegerType shift) noexcept;
```

Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntegerType>` is `true`.

Effects: Equivalent to: `return b = byte(static_cast<unsigned char>(b) >> shift);`

```
template <class IntegerType>
  constexpr byte operator <<(byte b, IntegerType shift) noexcept;
```

Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntegerType>` is `true`.

Effects: Equivalent to: `return byte(static_cast<unsigned char>(b) << shift);`

```
template <class IntegerType>
  constexpr byte operator >>(byte b, IntegerType shift) noexcept;
```

Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntegerType>` is true.

Effects: Equivalent to: `return byte(static_cast<unsigned char>(b) >> shift);`

```
constexpr byte& operator|=(byte& l, byte r) noexcept;
```

Effects: Equivalent to: `return l = byte(static_cast<unsigned char>(l) | static_cast<unsigned char>(r));`

```
constexpr byte operator|(byte l, byte r) noexcept;
```

Effects: Equivalent to: `return byte(static_cast<unsigned char>(l) | static_cast<unsigned char>(r));`

```
constexpr byte& operator&=(byte& l, byte r) noexcept;
```

Effects: Equivalent to: `return l = byte(static_cast<unsigned char>(l) & static_cast<unsigned char>(r));`

```
constexpr byte operator&(byte l, byte r) noexcept;
```

Effects: Equivalent to: `return byte(static_cast<unsigned char>(l) & static_cast<unsigned char>(r));`

```
constexpr byte& operator^=(byte& l, byte r) noexcept;
```

Effects: Equivalent to: `return l = byte(static_cast<unsigned char>(l) ^ static_cast<unsigned char>(r));`

```
constexpr byte operator^(byte l, byte r) noexcept;
```

Effects: Equivalent to: `return byte(static_cast<unsigned char>(l) ^ static_cast<unsigned char>(r));`

```
constexpr byte operator~(byte b) noexcept;
```

Effects: Equivalent to: `return byte(~static_cast<unsigned char>(b));`

```
template <class IntegerType>
  constexpr IntegerType to_integer(byte b) noexcept;
```

Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntegerType>` is `true`.

Effects: Equivalent to: `return IntegerType(b);`

D.4 C standard library headers [depr.c.headers]

4 Every other C header, each of which has a name of the form *name.h*, behaves as if each name placed in the standard library namespace by the corresponding *cname* header is placed within the global namespace scope, except for the functions described in 26.9.5, the declaration of `std::byte`, the functions and function templates described in [support.types.byteops]. It is unspecified whether these names are first declared or defined within namespace scope (3.3.6) of the namespace std and are then injected into the global namespace scope by explicit using-declarations (7.3.3).

## Acknowledgements

Gabriel Dos Reis originally suggested the definition of byte as a library type by using a scoped enumeration and also provided valuable review of this proposal.

## References

[1] Richard Smith, "Working Draft: Standard For Programming Language C++", N4618, 2016, [Online], Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4618.pdf

[2] Neil MacIntosh, "A byte type for increased type safety", P0257, 2016, [Online].