Project: Programming Language C++, Evolution Working Group
Document number: P0257R1
Date: 2016-05-27
Reply-to: Neil MacIntosh neilmac@microsoft.com

A byte type for increased type safety

Contents

Changelog 2
Changes from R0 2
Introduction 2
Motivation and Scope 2
Proposed Wording Changes 3
Acknowledgements 4
References 4

Changelog

Changes from R0

- After polling in Evolution Working Group, removed the alternative wording that allowed access to object representation through any enum that met the same definition as the proposed `std::byte` type.
- Added a note to explain the wording for the definition of `std::byte` itself will come via a separate Library Evolution Working Group proposal.

Introduction

This proposal defines a simple distinct standard library type `std::byte` specifically for representing a byte of storage, with adjustment to the current language rules around type aliasing for this type to fully serve its purpose.

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.

However, adding a new fundamental type to the language is unnecessary because it can be expressed in library almost entirely. Adding a new keyword to denote bytes that is definable (for the large part) also increases the risks of breaking existing working code. This paper proposes instead adding a `byte` type as a definition in the standard library. This approach is both backward-compatible and forward looking, efficient – and perhaps a testimony to C++'s ability to express abstractions with zero overhead.

The definition of `std::byte` is:

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

Note that this definition (with the adjustment to the language rules below) makes `byte` just as a "first-class type" (whatever that means) as if it had been designated by a keyword.

This proposal suggests clarifying a number of existing paragraphs of the standard text to allow use of pointers of type `std::byte*` to access object representation of an object. That can be achieved in at least two ways: (a) allow any scoped enumeration with a character type as its underlying type to alias any

other storage; or (b) specifically singling out `std::byte` as a permitted type for object representation access.

Proposed Wording Changes

The following proposed changes are relative to N4567 [1]. Additions are highlighted here in green and deletions in red (the deletions are merely for grammatical purposes).

3.8 Object lifetime [basic.life]

5 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:

—(5.1) 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*,

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

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

—(5.4) 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`, 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`. 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]

10 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:

—(10.1) the dynamic type of the object,

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

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

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

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

—(10.6) 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),

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

—(10.8) 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*]

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++”, N4567, 2015, [Online], Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4567.pdf

[2] Neil MacIntosh, “A byte type definition”, P0298, 2016 [Online].