A Unit Type for C++

Document number:	P1014R0
Date:	April 1, 2018
Audience:	EWG
Reply-to:	Andrew Sutton <asutton@uakron.edu> Nevin Liber <nevin@eviloverlord.com>

Introduction

Many languages, especially those of the functional variety, include a univalent (having just one value) type called `unit`. This is typically used to represent the return value of functions or expressions that do not compute a result, or whose result is uninteresting.

Matt Calabrese proposes a unit type by making `void` regular (i.e., an actual value type like `int` or `bool`) in P0146R1. There are a number of reasons why this doesn't quite work. For example, the proposal breaks the equivalence of the parameter list `(void)` with `()`, which may be a breaking change. While that approach would likely not have worked due to the legacy concerns of C++, the availability of a unit type solves a number of real problems in C++.

One place where a unit type is particularly useful is generic facilities that wrap functions that can return either a value type or `void`. In order to implement these facilities, we need extra specializations to handle the `void` cases separately from the value cases. When coupled with non-forwarded const/non-const overloads, we end up requiring a combinatorial explosion of specializations.

This paper proposes the addition of a new fundamental type which represents a single value. We propose to call that type **`short bool`**. We also introduce a unit literal, spelled **`()`**.

We note that this approach does not solve the problems with `void`. We will need to educate users to use `short bool` in place of `void` in order to avoid the problems that Calabrese highlights in his paper.

This paper is organized as follows: we discuss a handful of related proposal, beyond P0146R1, the provide core wording to add that type to the language and language to update a number of library facilities where it `short bool` could be used. Finally, we address open issues and thoughts for future work.

Related proposals

Tony Van Eerd proposes describes a more ambitious reimagining of the C++ integer system, allowing `long` and `short` modifiers to be repeated and combined in the obvious ways. In this approach an integral-based unit type would seem to be impossible since the type `short bool` would have ½ bits in its representation. Some other formulation of the entity would need to be sought.

The `elastic_integer` proposal, P0828r0, also includes the ability to represent single-valued types using specializations of the form `elastic_integer<0, T>` where T is a `signed` or `unsigned` integer type. However, we prefer for this to be a fundamental type so that we can take advantage of core language rules to formulate its behavior and modulate its usage.

Proposed changes

5.3 Kinds of literals [lex.literal]

Add the unit literal to the list of literals.

literal:

integer-literal
character-literal
floating-literal
string-literal
boolean-literal
unit-literal
pointer-literal
user-defined-literal

5.13.7 Unit literals [lex.unit]

Insert this section with the given paragraph number. It should follow [lex.bool]. The text of this section follows:

unit-literal:

()

The unit literal is () and represents the *unit value*. This literal has type `short bool`.

6.7.1 Fundamental types

Modify paragraph 6.

Values of type `bool` are either `true` or `false`. [*Note:* There are no signed, unsigned, ~~short,~~ or long bool types or values. — *end note*] Values of type bool participate in integral promotions (7.6).

Add a new paragraph after paragraph 6 with new footnote.

The only value of `short bool` is `()`.[footnote] Values of type `short bool` participate in integral promotions (7.6).

With the footnote reading:

There is no way to construct an invalid `short bool` object. Every possible permutation of bits in the value representation is a valid representation of the value.

Even though there is no fixed bit pattern for the unit value, its use in arithmetic expressions is well-defined because it is always promoted to the value zero before the operator is applied (7.6 in this proposal). This is true for relational and equality operators as well, and with unsurprising results (e.g., `() < ()` is false, `() == ()` is true).

Modify paragraph 7 to read:

Types `short bool`, `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types.

7.6 Integral promotions

Add a new paragraph after paragraph 6.

A prvalue of type `short bool` can be converted to a prvalue of type `int`, with the value becoming zero.

7.8 Integral conversions

Add a new paragraph after paragraph 4.

If the destination type is `short bool`, see 7.15. If the source type is `short bool` the value is converted to zero.

7.15 Pointer conversions

Modify the definition of null pointer constant to support unit values.

A *null pointer constant* is an integer literal (5.13.2) with value zero or a prvalue of type `std::nullptr_t` or `short bool`.

7.15 Unit conversions

Add this section.

A prvalue of arithmetic, unscoped enumeration, pointer, or pointer-to-member type can be converted to a prvalue of type `short bool`. All values of the source type are converted to the unit value.

10.1.7 Type specifiers

Modify paragraph 2 as follows:

As a general rule, at most one *defining-type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration* or in a *defining-type-specifier-seq*, and at most one *type-specifier* is allowed in a *type-specifier-seq*. The only exceptions to this rule are the following:

- `const` can be combined with any type specifier except itself.
- `volatile` can be combined with any type specifier except itself.
- `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.
- `short` or `long` can be combined with `int`.
- `short` can be combined with `bool`.
- `long` can be combined with `double`.
- `long` can be combined with `long`.

10.1.7.2 Simple type specifiers

Add a new row to table 11.

Specifier(s)	Type
`short bool`	"`short bool`"

21.3.2 Header <limits> synopsis

Add the following specialization to the synopsis of `<limits>`.

```
template<> class numeric_limits<short bool>;
template<> class numeric_limits<bool>;
```

21.3.4.2 `numeric_limits` specializations

Add a specialization of `numeric_limits` for bool as a signed quantity.

The specialization for `short bool` shall be provided as follows:

```
namespace std {
  template<> class numeric_limits<short bool> {
  public:
    static constexpr bool is_specialized = true;
    static constexpr short bool min() noexcept { return (); }
    static constexpr short bool max() noexcept { return (); }
    static constexpr short bool lowest() noexcept { return (); }

    static constexpr int digits = 0;
```

```
    static constexpr int digits10 = 0;
    static constexpr int max_digits10 = 0;

    static constexpr bool is_signed = true;
    static constexpr bool is_integer = true;
    static constexpr bool is_exact = true;
    static constexpr int radix = 1;
    static constexpr bool epsilon() noexcept { return 0; }
    static constexpr bool round_error() noexcept { return 0; }

    static constexpr int min_exponent = 0;
    static constexpr int min_exponent10 = 0;
    static constexpr int max_exponent = 0;
    static constexpr int max_exponent10 = 0;

    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    static constexpr bool infinity() noexcept { return 0; }
    static constexpr bool quiet_NaN() noexcept { return 0; }
    static constexpr bool signaling_NaN() noexcept { return 0; }
    static constexpr bool denorm_min() noexcept { return 0; }

    static constexpr bool is_iec559 = false;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo = false;

    static constexpr bool traps = false;
    static constexpr bool tinyness_before = false;
    static constexpr float_round_style round_style = round_toward_zero;
  };
}
```

23.7.8 Class `monostate`

Replace the definition of `monostate`.

~~`struct monostate {};`~~

```
enum class monostate : short bool {};
```

~~23.7.8 `monostate` relational operators~~

Delete this section, since the new definition of `monostate` already provides the behaviors.

~~`constexpr bool operator<(monostate, monostate) noexcept { return false; }`~~

~~`constexpr bool operator>(monostate, monostate) noexcept { return false; }`~~

~~`constexpr bool operator<=(monostate, monostate) noexcept { return true; }`~~

~~`constexpr bool operator>=(monostate, monostate) noexcept { return true; }`~~

~~`constexpr bool operator==(monostate, monostate) noexcept { return true; }`~~

~~`constexpr bool operator!=(monostate, monostate) noexcept { return false; }`~~

~~[Note: monostate objects have only a single state; they thus always compare equal. — end note]~~

23.15.2 Header <type_traits> synopsis

Add the following declarations to the synopsis.

```
template<short bool U>
  using unit_constant = integral_constant<short bool, U>;
using unit_type = unit_constant<()>;
```

Open issues

Should we allow a declaration of `main` that returns `short bool`?

We probably want some `is_single_valued<T>` trait that is true when T is `short bool`, an empty class, or an enum with `short bool` as an underlying type. That would allow the specialization to be broadly defined over all single-valued types.

If a class has no data members, then we should synthesize an implicit constructor for values of type short bool. This would unit values to be implicitly converted into objects of empty class type. This would appear to add a new category of constructors (e.g., `is_unit_constructible`, `is_trivially_unit_constructible`, etc.).

We could add a specialization of `std::function<R(ArgTypes...)>` such that when R is single-valued, the function could be constructed over practically any operand, and we simply return the literal ().

Richard Smith notes that "the library wording for `operator<=>` specifies that the comparison category types can be compared with a literal 0 (but need not support comparison with any other value), and we do not have a good way to write a function that only accepts a literal 0 today." Our `short bool` type does not solve this problem as is because values of that type participate in conversions: any integer value can be converted to `short bool`.

An interesting direction to consider would be to allow the *function-specifier* `explicit` to also modify the type `short bool`. We could then prevent integer conversions to type `explicit short bool`, which would seem to satisfy the requirement above.

The addition of an explicit short bool type might also solve syntactic quirks in the definition of tag types throughout the library. Today, most tag types are empty classes with explicit default constructors. We could replace these with enums. For example, the nothrow facility might be written as:

```
enum nothrow_t : explicit short bool { nothrow };
```

Presumably, this could be made to require explicit construction of the enclosing enum.

References

Calabrese, Matt. “Regular void”. ISO/IEC JTC1 SC22 WG21. No. P0146R1.

Tony Van Eerd. “Standardizing Extended Integers”. ISO/IEC JTC1 SC22 WG21. No. P0989R0.

John McFarlane, “Elastic Integers”, ISO/IEC JTC1 SC22 WG21. No. P0828R0.