Document number:
Date: 2016−01−20
Project: Programming Language C++
Reply to: Vincent Reverdy (vince.rev@gmail.com)

On the standardization of fundamental bit manipulation utilities

Vincent Reverdy[1] and Robert J. Brunner[1]

[1] *Department of Astronomy, University of Illinois at Urbana-Champaign, 1002 W. Green St., Urbana, IL 61801*

Abstract

We discuss the addition to the standard library of class templates to ease the manipulation of bits in C++. This includes a `bit_value` class emulating a single bit, a `bit_reference` emulating a reference to a bit, a `bit_pointer` emulating a pointer to a bit, and a `bit_iterator` to iterate on bits. These tools would provide a solid foundation of algorithms operating on bits and would facilitate the use of unsigned integers as bit containers.

Contents

1 Introduction

This proposal introduces a class template `std::bit_reference` that is designed to emulate a reference to a bit. It is inspired by the existing nested classes of the standard library: `std::bitset::reference` and `std::vector<bool>::reference`, but this new class is made available to C++ developers as a basic tool to construct their own bit containers and algorithms. It is supplemented by a `std::bit_value` class to deal with non-referenced and temporary bit values. To provide a complete and consistent set of tools, we also introduce a `std::bit_pointer` in order to emulate the behaviour of a pointer to a bit. Based upon these class templates, we design a `std::bit_iterator` that provides a foundation of bit manipulation algorithms. We discuss the API that is required to access the underlying representation of bits in order to make these algorithms faster. Although they will be given as illustrating examples, bit algorithms would need a separate proposal and are thus considered as out of the scope of this proposal that focuses on the fundamental tools.

2 Motivation

In *The C++ Programming Language* [Stroustrup, 2013], Bjarne Stroustrup highlights the fact that "unsigned integer types are ideal for uses that treat storage as a bit array." One of the most basic functionality that an array generally provides is a convenient way to access its elements. However, the C++ standard library is currently missing a tool to access single bits in a standardized way. Such tools already exist, but they are buried as internal helper classes with private constructors and thus they are kept away from C++ developers. Specific examples include `std::bitset::reference`, `std::vector<bool>::reference` and `boost::dynamic_bitset::reference` [Siek et al., 2015]. If unsigned integral types should be seen as bit containers, it would be convenient to have a standard utility to access and operate on single bits as if they were array elements.

In addition to this basic motivation, applications that could leverage bit utilities include, among others, performance oriented software development for portable devices, servers, data centers and supercomputers. Making the most of these architectures often involves low-level optimizations and cache-efficient data structures [Carruth, 2014]. In fact, these aspects are going to become more and more critical in a post-Moore era where energy efficiency is a primary concern. In that context, being able to act directly on bits, for example to design efficient data structures based on hash tables, is of primary importance. Moreover, the spread of arbitrary-precision integral arithmetic both at the hardware level [Ozturk et al., 2012] and at the software level, as proposed in N4038 [Becker, 2014], will require, once again, tools to efficiently access single bits.

For all of these reasons, and to prevent bit references to be repeatedly implemented, we propose to add a `std::bit_reference` class template to the C++ standard library. As a response to feedback gathered through the future proposal platform, we have complemented this class template with a `std:: bit_value`, a `std::bit_pointer` and a `std::bit_iterator` in order to have a complete set of bit utilities, and to serve as the basis of a future standardized library of bit algorithms based on an alternative and more generic approach than N3864 [Fioravante, 2014].

With bit iterators, some standard algorithms could benefit from substantial optimizations. For example, a specialization of `std::count` on `std::bit_iterator` should be able to call, when available, the assembly instruction `popcnt` on the underlying unsigned integers of the bit sequence. `std::sort` could also call `popcnt` to count the number of zeroes and ones, and then directly change the value of unsigned integers accordingly. In fact, most standard algorithms, such as `std::copy`, should be able to operate directly on integers instead of individual bits. These types of approaches have already been explored in `libc++` for `std::vector<bool>` with significant performance improvements [Hinnant, 2012]. Specialized bit algorithms could also be provided. As an example, a `parallel_bit_deposit` algorithm could be far more efficient than a `std::copy_if` by calling the assembly function `pdep` on integers. And, of course, a library of bit utilities as described here would allow users to write their own efficient bit algorithms using similar strategies.

3 Impact on the standard

This proposal is a pure library extension. It does not require changes to any standard classes or functions, and introduces a new header for bit utilities whose name is discussed in part 4. Section 6 discusses the nested classes `std::bitset::reference` and `std::vector<bool>::reference`.

4 Design decisions

Introduction

We propose a `<bit>` header providing a class `std::bit_value` and three class templates parameterized by a type: a `std::bit_reference`, with a design inspired by the existing nested bit reference classes [ISO, 2014], a `std::bit_pointer` and a `std::bit_iterator`.

Background

A clear definition of what a bit is, how it is related to bytes and to fundamental types, and what its behaviour should be like are prerequisites of well designed bit utility classes. The need of raising the question of the definition of a bit can be illustrated by the following problem, where `0` and `1` indicate the bit value obtained at the end of the line, and where `X` refers to non-compiling lines:

```cpp
struct field {unsigned int b : 1;};

bool b0 = false; b0 = ~b0; b0 = ~b0;                    // 1
auto x0 = std::bitset<1>{}[0]; x0 = ~x0; x0 = ~x0;      // 0
auto f0 = field{}; f0.b = ~f0.b; f0.b = ~f0.b;          // 0

bool b1 = false; b1 = ~~b1;                             // 0
auto x1 = std::bitset<1>{}[0]; x1 = ~~x1;               // 1
auto f1 = field{}; f1.b = ~~f1.b;                       // 0

bool b2 = false; b2 += 1; b2 += 1;                      // 1
auto x2 = std::bitset<1>{}[0]; x2 += 1; x2 += 1;        // X
```

```
auto f2 = field{}; f2.b += 1; f2.b += 1;                          // 0

bool b3 = false; b2 = b3 + 1; b3 = b3 + 1;                        // 1
auto x3 = std::bitset<1>{}[0]; x3 = x3 + 1; x3 = x3 + 1;          // 1
auto f3 = field{}; f3.b = f3.b + 1; f3.b = f3.b + 1;              // 0

bool b4 = false; b4 += 3;                                         // 1
auto x4 = std::bitset<1>{}[0]; x4 += 3;                           // X
auto f4 = field{}; f4.b += 3;                                     // 1
```

As shown in this example, three existing C++ bit-like entities exhibit three different behaviours. Given that `std::bit_value` and `std::bit_reference` will define an arithmetic behaviour for a bit, it is important to think carefully about what this behaviour should be. Also, before discussing in more details the chosen design and its alternatives, we summarize what the existing standards have to say on bits and bytes, as well as on integral types, using the C++ working draft N4567 [Smith, 2015] and the C working draft N1548 [Jones, 2011]. The purpose of the following paragraphs is to provide condensed background information from the standards related to this proposal before starting discussing the design decisions in the next subsection.

The C standard gives the following definition in its section 3.5: a *bit* is a unit of data storage in the execution environment large enough to hold an object that may have one of two values. The C++ standard defines a bit in [intro.memory] as an element of a contiguous sequence forming a byte, a *byte* being the fundamental storage unit in the C++ memory model. According to this model, the memory available to a C++ program consists of one or more sequences of contiguous bytes. A byte is required to have a unique address and to be at least large enough to contain any member of the basic execution character set and the eight-bit code units of the Unicode UTF-8 encoding form. An *object* is defined in [intro.object] as a region of storage. According to [intro.memory] the address of an object is the address of the first byte it occupies, unless this object is a bit-field or a base class subobject of zero size. The section [basic.types] defines two representations. The *object representation* of an object of type `T` is the sequence of N `unsigned char` objects taken up by the object of type `T`. The number N is given by `sizeof`, where the `sizeof` operator yields the number of bytes in the object representation of its operand according to [expr.sizeof]. In the C++ standard, [basic.types] states that 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 making up the object can be copied into an array of `char` or `unsigned char`. The relationship between bytes and characters is made clearer in [basic.fundamental] and [expr.sizeof], as well as in the section 6.2.6.1 of the C standard which establishes a direct link between bytes and `unsigned char`. The other representation defined by the C++ standard is the *value representation*. It corresponds to the set of bits that hold the value of type `T`.

Regarding integers, [fundamental.types] defines five *standard signed integer types*, five *standard unsigned integer types* and additional *extended integer types*. Figure 1 summarizes the properties of the C++ *integral types*, their representation and their conversion rules according to [fundamental.types] , [cstdint.syn], [numeric.limits.members], [conv.prom], [conv.integral] and [conv.rank]. As stated in this figure, the C++ standard require representation of integral types to define values by use of a pure binary numeration system. Such a system corresponds to a positional representation of integers that uses the binary digits `0` and `1`, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral

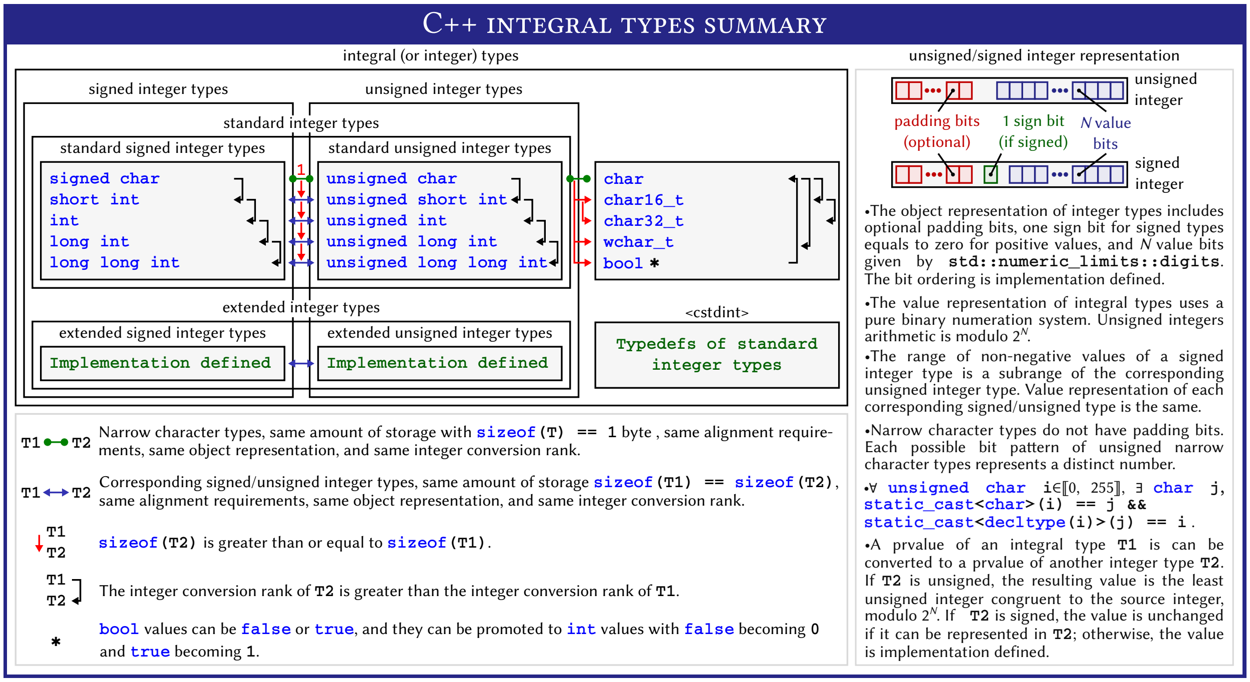

Figure 1: Integral types.

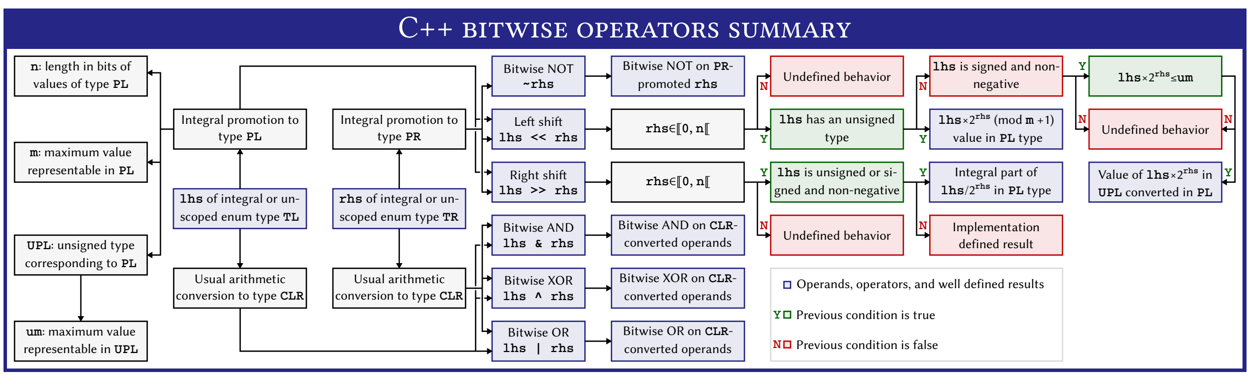

Figure 2: Bitwise operators.

power of 2, except perhaps for the bit with the highest position [ISO, 2014]. Integral types come with bitwise operators whose behaviour is presented on figure 2 and that are of primary interest regarding the topic of this proposal for the role they play in bit extraction. In top of integral types, booleans, and character types, fundamental types also include floating point types. A synoptic view of conversion rules involved in arithmetic operations for all these types is given on figure 3 for an implementation compliant with the C++ standard. In this figure, the type of `x` corresponds to rows while the type of `y` corresponds to columns. Each type is associated with a color that is used to indicate the decayed result type of an operation involving `x` and `y`. As an example, for `x` a `long long int` and for `y` an `unsigned long int`, the type of `x + y` is `unsigned long long int`. An interesting property to note is that integral types smaller than `int` are implicitly converted to `int` during arithmetic operations regardless of their signedness: as an example, a `bool`, a `char`, an `unsigned char` and an `unsigned short int` exhibit similar arithmetic behaviours for most operations.

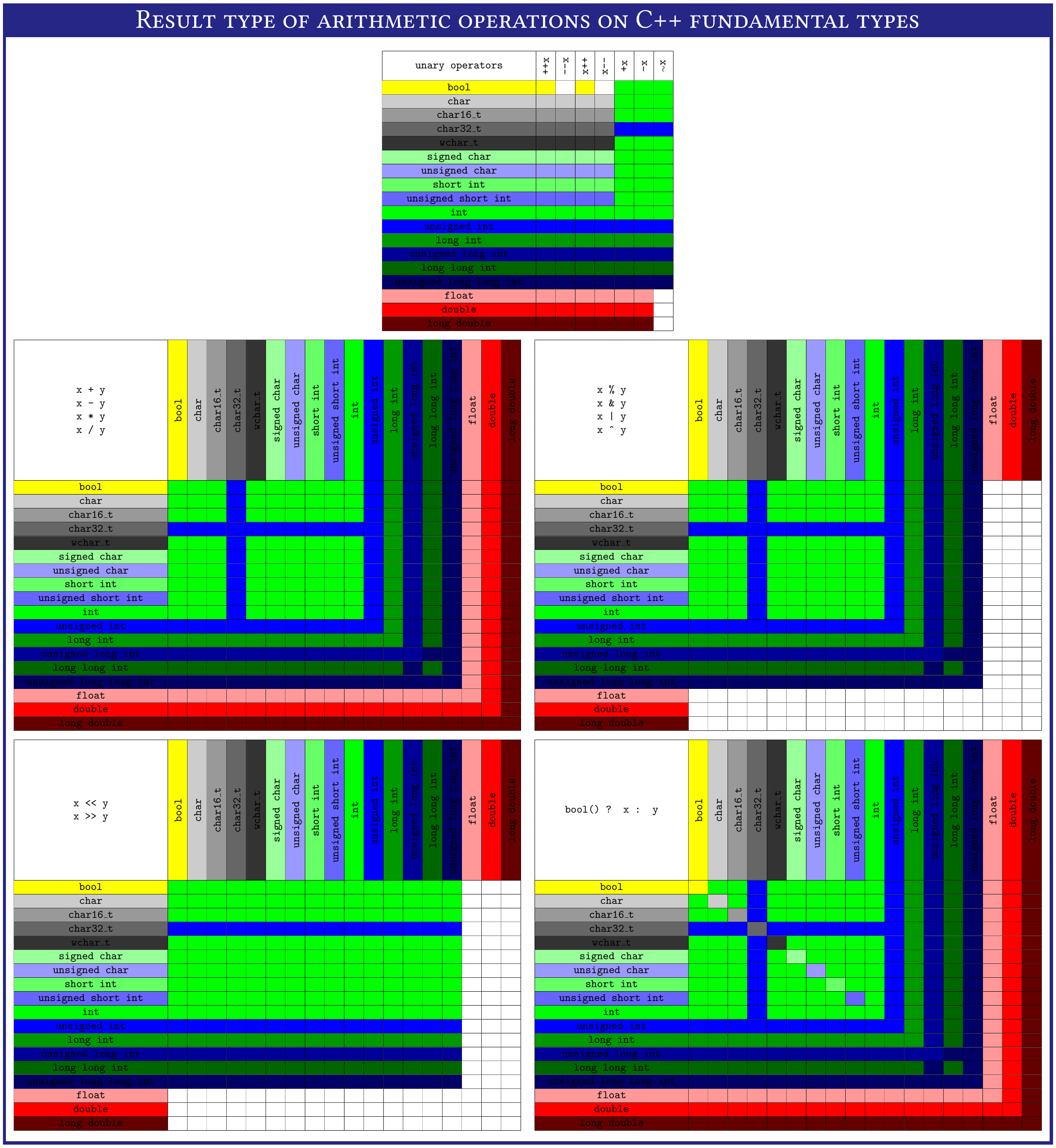

Figure 3: Arithmetic operations on fundamental types.

In this context, two questions regarding the definition and the behaviour of a bit appear, and are at the core of the design of the class templates we propose:

- How to define the position of a bit within an object?
- What is the arithmetic definition of a bit?

The consequences of the first question include the types on which `std::bit_reference` and `std::bit_pointer` will operate, and what bits will be accessible through iteration. The answer to the second question will determine the implicit conversions and the results of arithmetic operations on `std::bit_value` and `std::bit_reference`.

How to define the position of a bit within an object?

Bits are not directly addressable, but they are defined as binary elements of bytes which are the most fundamental addressable entities of a given system and are required to be made of at least 8 bits. Consequently, identifying a bit requires a byte *address* and a *position* within a byte. The problem is that the underlying ordering of bits within a byte is not specified by the standard. Therefore, according to the sole criterion of bits seen as elements of bytes, the mapping between a position and an actual bit is implementation-defined.

To make bit references, pointers, and iterators usable, the design needs to specify this mapping. As presented in the background subsection, the standard defines a clear connection between bytes and `unsigned char`s: an `unsigned char` have a size of exactly one byte, has no padding or sign bits, each possible bit pattern represents a distinct number, and its value rely on a pure binary numeration system. In other words, `unsigned char`s define an unambiguous bit mapping which corresponds to the definition of a bit seen as a *binary digit* of natural numbers. According to this mapping, the `n`-th bit of an `unsigned char uc` is obtained by the operation `uc >> n & 1`, when `n` $\in$ ⟦`0, std::numeric_limits<unsigned char>::digits`⟦. With this definition, every bit can be referenced in an univocal manner with a pair of byte address and position `std::pair<unsigned char*, std::size_t>`.

Although it is very well defined, this method is very limited in the sense that it only gives access to the object representation of types, and does not provide a direct implementation independent way of accessing the `n`-th bit of the value representation of integral types. In fact, as the object representation of integers other than the unsigned narrow character types is implementation-defined, the method described above gives access to all bits of the integers, but in an order that can depends on the architecture and on the compiler. Endianness and padding bits are, of course, a part of the problem. Defining a bit as the `n`-th binary digit of a natural number makes the design more generic and more usable. According to this definition, for any unsigned integer `ui` of type `UIntType`, we can obtain the `n`-th bit by the same formula we used for the `unsigned char` case: `ui >> n & 1` for `n` $\in$ ⟦`0, std::numeric_limits<UIntType>::digits`⟦. A design relying on this approach presents several advantages: it defines unambiguously the position of a bit for all unsigned integer types, it produces a platform-independent behaviour regardless of the underlying representation of these integers, their endianness and the number of padding bits they include, and it still provides an access to the object representation through a `reinterpret_cast` to `unsigned char`s. Additionally, and more importantly, the definition of the position of a bit matches its mathematical definition in a *positional numeration system*, making the use of the design intuitive.

At this point, the question of the generalization of this design arises. Should types other than unsigned integer types be allowed? For a complete arbitrary type `T`, the only relevant bit definition is the one based upon the object representation of `T`. The design proposed in the previous paragraph can already provides an easy access to the object representation of `T` through a `reinterpret_cast` to `unsigned char` pointers. However, the question remains open for the following types: non-integral arithmetic types, bit containers, unbounded-precision integer types as proposed in N4038 [Becker, 2014] and, of course, non-unsigned integral types. Concerning floating-point types, as their underlying representation is implementation-defined as specified in [basic.fundamental] and is left completely free by the standard, as this representation is not relying on a pure positional numeration system but generally includes a sign, a mantissa and an exponent, and as the shift operators does not apply to them, it does not make much sense to treat them differently than any other arbitrary type `T`. For the three remaining cases, namely bit containers, unbounded-precision integer types and non-unsigned integral types, the situation is different since one can define a bit position relying on the value representation of these objects.

The question of referencing a bit in bit containers, like `std::bitset` and the specialization `std::vector<bool>`, and in unbounded-precision integer types is very similar. Even if not required, the vast majority of implementations of these objects rely on contiguous arrays of *limbs* of unsigned integer types. For bit containers, the most natural definition of the bit position would be the same as the one entering in the declaration of the subscript operator `operator[]`. For unbounded-precision integers it can be trickier since they can be signed and include a sign bit. But even if we ignore, for the moment, the issue of the sign bit, other design questions exist. For example, it is unlikely that most unbounded-precision integers define a subscript operator. In this case, accessing a bit through the shift operator, as in the previous paragraphs, would make more sense. This technique could also apply to `std::bitset` but not to `std::vector<bool>` since the specialization does not provide an `operator>>`. Moreover if a maker helper function such as `make_bit_reference(T& object, std::size_t pos)` was provided for bit containers and unbounded-precision integers, should the bit reference behaviour rely on the object, or on its underlying representation in terms of limbs? Regarding to this question, it would make more sense to rely on the object and its `operator[]` or `operator>>` regardless of the underlying representation in the similar way a bit reference relying on unsigned integers would work regardless of the optional presence padding bits. An internal access to the underlying container of limbs could still be provided through member functions returning `std::bit_references` instead of `std::bit_references` taking bit containers as parameters. It also opens the question of whether or not `std::bitset::reference` and `std::vector<bool>::reference` should be replaced by a `std::bit_reference`, or at least adjusted to provide the same interface. As already noted, for unbounded-precision or non-unsigned integer types, the question of the sign bit and negative values also has to be solved. In fact, the mapping between the object representation of signed integers and their negative values is far less constrained than for unsigned integer types. Consequently, a design relying on `operator>>` would lead to implementation-defined results. Whether we should, or not, accept such a design is left as an open question. As a remark, the feedback gathered online from the C++ standard discussion board pointed out that bit manipulation on signed integers could be achieved with a design limited to unsigned integer types, through a `reinterpret_cast<typename std::make_unsigned<IntType>::type*>`.

In all the following, we restrain the design to unsigned integer types as defined on figure 1. We also define the bit position such as the expression `ui >> n & 1` is extracting the `n`-th binary digit for `n` ∈ ⟦0, `std::numeric_limits<T>::digits`⟦. This choice is motivated by the fact that:

- the bit position matches the mathematical definition of a binary digit position in a positional numeration system

- it leads to a platform-independent behaviour
- it provides an access to the underlying bits of any type through a `reinterpret_cast` to `unsigned char*`
- it provides an access to the bits of signed integer types through a `reinterpret_cast` to `typename std::make_unsigned<IntType>::type*`
- "unsigned integer types are ideal for uses that treat storage as a bit array" as highlighted in section 2
- it matches the requirements of most use cases including cryptographic operations, hash value calculations and computations on arrays of limbs
- classes such as `std::bitset` already set a preference of conversions from and to unsigned integer types over generic integer types

However, and as a final note, the proposed design could stay the same and still accept all integral types, including future unbounded-precision integral types with minor modifications, at the expense of implementation-defined results since its specification is relying on the use of bitwise operations to extract bits.

What is the arithmetic definition of a bit?

The second main question on which a significant part of the design of a bit reference relies concerns the arithmetic behaviour of a bit. As shown in the introductory listing of the background subsection, three bit-like objects already present in the standard exhibit three different behaviours. In this part, we discuss the different options, their advantages and their drawbacks.

The first option is the one followed by `std::bitset::reference` and `std::vector<bool>::reference`. These classes are nested classes, mostly intended to take care of the result of the subscript operator `operator[]` and implementing the behaviour of a boolean value from the user's point of view. As the goal of `std::bit_value`, `std::bit_reference`, `std::bit_pointer` and `std::bit_iterator` will be slightly different in the sense that they are specifically intended to provide users with the ability of writing their own bit manipulation algorithms, the choices made in terms of arithmetic can be different from the the ones of the nested classes, especially if it leads to a better interface for users. Many approaches tend to identify a bit with a boolean although the two are conceptually different: the first one is a digit whereas the second one is a logical data type. Both happen to have two possible values which generally leads to representation of the first one in terms of the second one. The arithmetic behaviour of `std::bitset::reference` and `std::vector<bool>::reference` mainly relies on the implicit conversion to `bool`. As a consequence, all binary operators applicable to `bool` also applies to `std::bitset::reference` through this implicit conversion. However, the reference does not exactly behave as a `bool` since it provides a `flip` member, it implements its own `operator~`, and it does not allow arithmetic assignment operations. In other words, if `ref` is of type `std::bitset::reference`, `~ref` can lead to different values than if it were a `bool`, `ref = ref + 3` gives the same result as if it were a `bool`, and `ref += 3` does not compile. For a nested class whose main role is to serve as proxy for the result of `operator[]`, this very specific behaviour may not be of primary concern. But for a `std::bit_reference` designed to provide a generic way to deal with bit operations, an implicit conversion to `bool` and its implicit integral promotion to `int` mixed with specifically designed operators such as `operator∼` could be very error-prone. Also, we investigate other alternatives to the original scenario which would consists in reproducing the exact same behaviour as `std::bitset::reference`.

The first alternative is to consider that a bit, as a pure binary digit, is not an arithmetic object and therefore should not implement any arithmetic behaviour. Instead, it would provide three member functions: `set`, `reset` and `flip`, these functions already being a part of the implementation details of some bit references such as `boost::dynamic_bitset::reference` [Siek et al., 2015]. Boolean conversions would be provided for arithmetic purpose through an `operator=` and through an *explicit* `operator bool`. The explicitness of the operator would prevent any undesired conversion and integral promotion, and would make clearer the conceptual difference between a binary digit and a boolean data type while still providing the desired casting functionality. This would lead to a minimal but very consistent design.

The second alternative is to consider that a bit and a `bool` have the exact same behaviour. In that case, the class would not provide special members like `set`, `reset` and `flip` and would stick to the arithmetic operators executable on booleans. The binary arithmetic operators could be provided either explicitly, or by an implicit cast to `operator bool`. `operator∼` and arithmetic assignment operators would be provided and would lead to the same results as for booleans. As in the case of the first alternative, this strategy would avoid unexpected arithmetic behaviours for users and would introduce an easily understandable interface.

The third alternative echoes the fact that the C++ standard identifies bytes with an unsigned integer type, namely `unsigned char`s. In the same manner, we can consider a bit as a binary digit with an arithmetic behaviour equivalent to a hypothetical `uint1_t`, an unsigned integer one-digit long that can be equal to either $0 \times 2^0 = 0$ or $1 \times 2^0 = 1$. For the binary digit side, `std::bit_reference` would get the member functions `set`, `reset` and `flip` and an explicit `operator bool`. For the arithmetic side, assignment operators and increment and decrement operators would implement a modulo $2^1 = 2$ arithmetic. Consequently, for a bit reference `bit` initially equals to `0`, `bit += 3` would lead to a value of `1` and `(++bit)++` would lead to a value of `0`. For binary arithmetic operators there are two options to consider: either implementing the overloads explicitly, or making the operators work through an implicit cast to an unsigned integral type. For this last option there are three possibilities: this type could be set to the smallest unsigned integral type, namely `unsigned char` or `uint_least8_t`, or it could be set to the type in which the bit is referenced, or it could be set through an additional template parameter of `std::bit_reference`. However, adding a template parameter to specify the arithmetic behaviour of a bit would made the bit classes more complex for no real benefit.

To summarize, the main possibilities in terms of the arithmetic behaviour of a bit are the following:

- the design of the nested classes `std::bitset::reference` and `std::vector<bool>::reference`, with a mix of behaviours, possibly error-prone
- the first alternative, consisting in considering a bit as a pure binary digit therefore stripped of an arithmetic behaviour, although still accessible through an explicit conversion to a `bool`

- the second alternative, consisting in considering a bit as a boolean an therefore providing the exact same functionalities as a `bool`
- the third alternative, consisting in considering the first alternative with additional arithmetic properties corresponding to a one-digit long unsigned integer

Earliest drafts of this proposal were limited to these four options and the chosen design was based on the first alternative to keep the technical specifications as simple as possible. However, this simplicity was coming with a minor open problem. Considering that `std::bit_iterator::reference` is a `std::bit_reference`, and that `std::bit_iterator::pointer` is a `std::bit_pointer`, then what should `std::bit_iterator::value` be? Defining it as a `bool` or as an `unsigned char` would not provide the arithmetic behaviour of a one-digit long unsigned integer, while defining it as a `std::bit_reference` could lead to errors, since a reference and a value are two different things. Moreover, if `std::bit_reference` implements a one-digit long unsigned integer arithmetic, then what should be returned by the postfix increment and decrement operators? For consistency it has to return a type with the same functionalities as `std::bit_reference`, including `set`, `reset` and `flip` functions, but it cannot be a referenced bit: it has to be an independent bit.

This is where the idea of `std::bit_value` comes into play, solving these problems, allowing a consistent arithmetic behaviour implementation, and simplifying the design of the class template `std::bit_reference`. The role of `std::bit_value` is to mimic the value of independent, non-referenced bits. As a class representing independent bits implicitly constructible from bit references, it has to provide the arithmetic behaviour of a one-digit long unsigned integer. But the question of how to implement these arithmetic operators in a lightweight manner still remains. The answer can be found by analyzing the content of figure 3. The important thing to notice is that, for most operations, `bool`, `unsigned char` and `unsigned short int` act in the same way: they are implicitly casted to `int`, and so should a one-digit long unsigned integer. Fitting `std::bit_value` with an implicit `operator bool` would enable this behaviour. However, making `std::bit_value` implicitly constructible from `bool` would not result in a one-digit long unsigned integer arithmetic, but making it implicitly constructible from `unsigned char` would do it. This strategy leads to a conversion and optionally a narrowing of any integer type to `unsigned char`, whose least significant bit could then be extracted to set the actual value of `std::bit_value`. In addition to these implicit conversions, `std::bit_value` and `std::bit_reference` would have to implement the arithmetic operators that mutate their states such as compound assignment operators and both increment and decrement operators.

This approach, namely:

- the fourth alternative, consisting in considering the third alternative implemented with an additional `std::bit_value` class

is the one that is followed in this proposal because of the solution it provides to the above-mentioned problems.

What design?

Based on the answers to the fundamental questions of the position of a bit and of its arithmetic behaviour, we can design a library solution to access bits. Accordingly to the previous subsections,

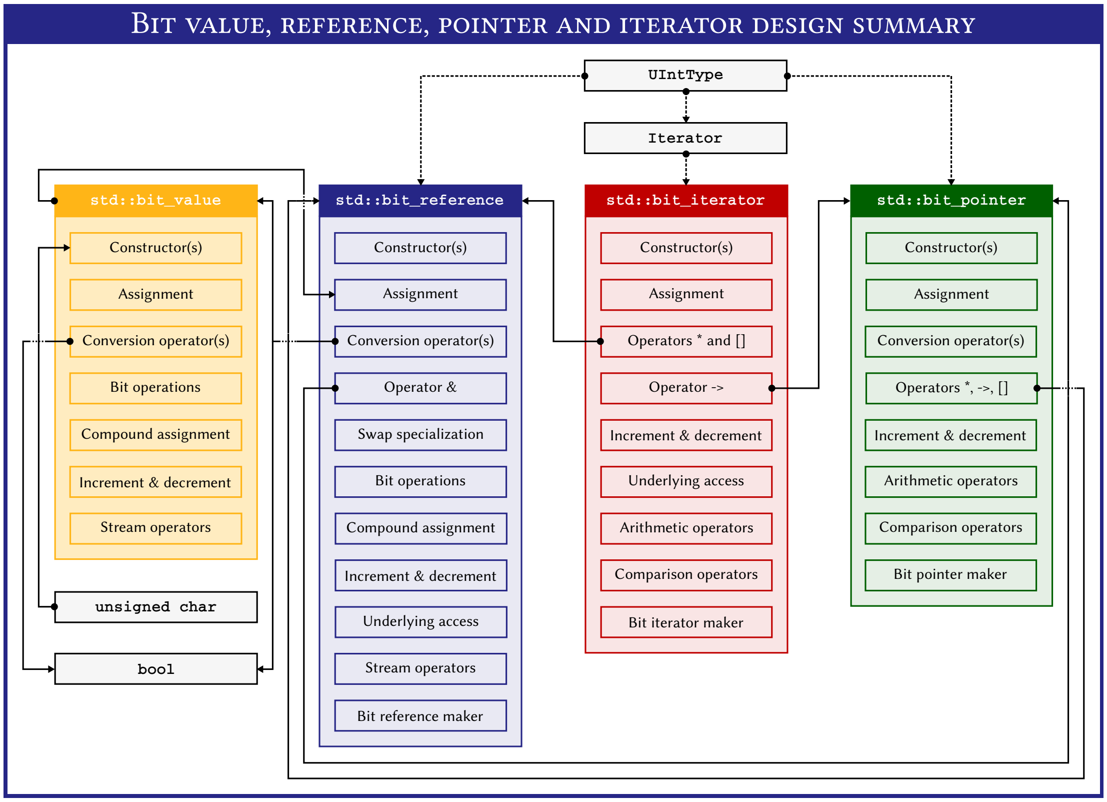

Figure 4: Design summary.

this design is built around four elements:

- `std::bit_value` emulating an independent, non-referenced bit
- `std::bit_reference` emulating a reference to a bit
- `std::bit_pointer` emulating a pointer to a bit
- `std::bit_iterator`, based on the preceding classes and emulating an iterator on bits

`std::bit_reference` and `std::bit_pointer` are parameterized by a template type indicating the underlying object type of which the bits belongs to. They both have a `constexpr` constructor taking either a reference or a pointer to a value of the underlying type, and a position indicating the position of the bit. `std::bit_reference` gives access to a `std::bit_pointer` through its member `operator&`, and, reciprocally, `std::bit_pointer` gives access to a bit reference through its operators `operator*`, `operator->` and `operator[]`. `std::bit_reference` implements the behaviour of a bit: it provides basic bit functionalities as well as a conversion operator to a `std::bit_value` and an assignment operator taking a `std::bit_value` as a parameter. `std::bit_value` implements an independent bit and provides the same functionalities as bit references. Stream operators are also overloaded for both bit values and references, to provide a display of `0` and `1` values. Finally, an interface to access the underlying information of `std::bit_reference`, namely the address of the referenced object and the bit position, is provided to allow the writing of faster bit manipulation algorithms. `std::bit_pointer` emulates the behaviour of a pointer to a bit, implementing all the classical functions operating on traditional pointers. A `std::bit_pointer` can be nullified, and in that case, the underlying pointer is set to nullptr and the position is set to 0. std::bit_iterator is built on the top of both std::bit_reference and std::bit_pointer. It takes an iterator Iterator on an underlying object type as a template parameter. std::bit_iterator::operator++ and other increment and decrement operators implement the following behaviour: they iterate through the binary digits of the underlying object, and execute the member Iterator::operator++ to go to the next object once the last binary digit of the current object has been reached. This strategy allows to iterate through contiguous, reversed, non-contiguous and virtually all possible iterable sequences of unsigned integers. A std::bit_iterator can be constructed from an Iterator value and a position, and it implements the traditional behaviour of a standard iterator, with its value type being a std::bit_value, its reference type being a std::bit_reference, its pointer type being a std::bit_pointer and its category being std::iterator_traits<Iterator>::iterator_category. Finally, and for conveniency, the classes come with non-member functions to make the right type of std::bit_reference, std::bit_pointer or std::bit_iterator based on a provided reference, pointer, or iterator. All the design decisions are summarized in figure 4.

Additional remarks: implicit conversions, swap operations, and cv-qualifiers

Additionally to the main design decisions listed in the previous subsection, some details deserve a particular attention. The first one concern the implicit conversions between bit values and bit references. A straightforward approach would be limited to the following:

- std::bit_value is implicitly constructible from unsigned char
- std::bit_value is implicitly convertible to bool
- std::bit_reference is assignable from std::bit_value
- std::bit_reference is implicitly convertible to std::bit_value

The problem with this strategy is that a bit reference would be two implicit conversions away from binary arithmetic operators: in other words, adding a bit reference to another arithmetic type would need a first conversion to std::bit_value and a second conversion to bool. But these two conversions are *user-defined* and the [class.conv] section of the standard specifies that, at most, one user-defined conversion can be implicitly applied to a single value. Consequently, and to avoid this problem, std::bit_reference should be made implicitly convertible to bool. However, whether or not bit references should remain implicitly convertible to std::bit_value too, as in the proposed design, is an open question.

The second remark concerns the std::swap function. Because the copy constructor and the copy assignment operator of std::bit_reference do not act in the same way, std::swap has to be overloaded. The proposed design includes only one overload with two std::bit_reference with the same underlying type. However, if we consider that bit values and bit references model the same fundamental concept of a bit, we should consider also adding the following overloads:

```
template <class UIntType>
void std::swap(
    std::bit_value& x,
    std::bit_reference<UIntType>& y
);
template <class UIntType>
void std::swap(
```

```
    std::bit_reference<UIntType>& x,
    std::bit_value& y
);
template <class UIntType>
void std::swap(
    std::bit_reference<UIntType>& x,
    std::bit_reference<UIntType>& y
);
```

Additionally, we should consider overloading the `std::exchange` function because the generic version will not lead to the expected result for bit references:

```
template <class UIntType, class U = std::bit_value>
std::bit_value std::exchange(
    std::bit_reference<UIntType>& obj,
    U&& new_val
)
```

These overloads of `std::swap` and `std::exchange` are not currently included in the design, but are left for discussion. Note that the same kind of questions arise for the comparison operators of bit pointers.

The third remark involves cv-qualified bit references and pointers. If we consider a hypothetical user-defined bit container, how should the typedefs `const_reference` and `const_pointer` be defined? For clarity, we list below all the possibilities regarding the constness of bit references and pointers, with `T` being a non cv-qualified unsigned integer type and with `bit` being a hypothetical fundamental arithmetic type representing a bit:

- `std::bit_reference<T>` models a standard non cv-qualified reference, which is equivalent to a `bit&`
- `std::bit_reference<const T>` models a reference to a constant and therefore mimics a `const bit&`
- `const std::bit_reference<T>` models a constant reference to a non-constant type and is the theoretical equivalent of a hypothetical `bit& const`, which does not compile
- `const std::bit_reference<const T>` models a constant reference to a constant type and is the theoretical equivalent of a hypothetical `const bit& const`, which does not compile
- `std::bit_pointer<T>` models a standard non cv-qualified pointer, which is equivalent to `bit*`
- `std::bit_pointer<const T>` models a pointer to a constant and mimics a `const bit*`
- `const std::bit_pointer<T>` models a constant pointer to a non-constant type and therefore mimics a `bit* const`
- `const std::bit_pointer<const T>` models a constant pointer to a constant type and therefore mimics a `const bit* const`

Consequently, even if both const-qualified types `const std::bit_reference<const T>` and `const std::bit_reference<T>` compile and can be useful as proxies to carry information about the location and the position of a referenced bit, they should be used with care as they do not have non-proxy equivalents. Moreover, given the listed definitions, it appears more clearly that the operator-> of const std::bit_pointer<T> should return a pointer to a non cv-qualified bit reference, or, in other words, a std::bit_reference<T>*, instead of a const std::bit_reference<T>*. And to answer the original question, a const_reference typedef should be defined as a std::bit_reference<const T> and a const_pointer typedef as a std::bit_pointer<const T>. The last remark concern the implicit cv conversions of bit references and bit pointers. In both cases, a default copy constructor and a constructor taking a reference to the provided template parameter type as an input already handle most cases. However, a std::bit_reference<const T> cannot be constructed from a std::bit_reference<T>, and a std::bit_pointer<const T> cannot be constructed from a std::bit_pointer<T>. To make it possible, we have to add generic conversion constructors of the form template <class T> bit_reference(const bit_reference<T>& other) and of the form template <class T> bit_pointer(const bit_pointer<T>& other). For bit pointers, an additional generic conversion assignment operator is also required. This last point conclude the remarks and allow us to detail the technical specifications.

5 Technical specifications

Introduction

The design decisions described in section 4, lead to the technical specifications presented in the following pages. A working C++14 implementation will be made available on a public GitHub repository [Reverdy, 2016].

Naming

Before discussing the definitions of the bit utility class templates, we list all the names related to this proposal, as well as possible alternatives. When these names already exist in the standard library, or are inspired by existing names, they appear in blue and we provide the link of their original source. Parentheses are used for optional prefixes and suffixes and to avoid listing all possible combinations. We start with the header name associated with the classes of this proposal and which could be extended through for future work on bits:

Naming summary: header		
Description	**Name**	**Alternatives**
Header (bit utilities, bit manipulation functions...)	<bit>	<bits> <bitwise> <bit_utility> <bitutils> <bit_tools>

Then, we list the main class names. We prefer std::bit_value over std::bit because the second one could be misleading, since the class it refers to does not correspond to a single bit in memory, but instead wraps the value of a bit and provides the desired functionalities.

Naming summary: classes		
Description	**Name**	**Alternatives**
Bit value class	bit_value	bit bitval bit_val
Bit reference class template	bit_reference	bitref bit_ref
Bit pointer class template	bit_pointer	bitptr bit_ptr
Bit iterator class template	bit_iterator	bititer bit_iter

Then, we list the names used for template parameters:

Naming summary: template parameters		
Description	**Name**	**Alternatives**
Generic type	T	Type
Other generic type	U	Other(Type)
Unsigned integer type	UIntType	UInt UnsignedInteger(Type)
Iterator type	Iterator	It
Character type	CharT	
Character traits type	Traits	

and the names of member typedefs:

Naming summary: member types		
Description	**Name**	**Alternatives**
Byte type from which a bit value is constructible	byte_type	byte byte_t
Type to which a bit belongs to	underlying_type	object_type element_type storage_type
Bit position type	size_type	position_type shift_type offset_type
Bit distance type	difference_type	
Base iterator type	iterator_type	underlying_iterator(_type)
Iterator traits member types	value_type difference_type pointer reference iterator_category	

Then, we list the names of function members:

Naming summary: function members		
Description	**Name**	**Alternatives**
Swap function member	swap	
Set bit function member	set	
Reset bit function member	reset	
Flip bit function member	flip	
Underlying iterator access function member	base	(get_)(underlying_)iterator
Bit memory address access function member	address	addressof (get_)(underlying_)address (get_)(underlying_)pointer (get_)(underlying_)ptr
Bit position access function member	position	(get_)(underlying_)position (get_)(underlying_)pos (get_)(underlying_)shift (get_)(underlying_)offset

as well as the names of non-member functions:

Naming summary: functions		
Description	**Name**	**Alternatives**
Non-member swap function	swap	
Bit reference creation function	make_bit_reference	make_bitref make_bit_ref
Bit pointer creation function	make_bit_pointer	make_bitptr make_bit_ptr
Bit iterator creation function	make_bit_iterator	make_bititer make_bit_iter

A finally, the following names are used for function parameters:

Naming summary: parameters		
Description	**Name**	**Alternatives**
Reference, pointer and iterator	ref ptr i	
Position	pos	
Value to be assigned	val	
Increment or decrement	n	
Object to be copied or assigned	other	
Left-hand and right-hand sides of an operator	lhs rhs	
Output and input streams	os is	
Bit reference, pointer or iterator in non-member functions	x	

Bit value specifications

The specifications of `std::bit_value` are given on figure 5.

Bit value synopsis

```cpp
// Bit value class
class bit_value
{
public:

  // Types
  using byte_type = unsigned char;

  // Lifecycle
  bit_value() noexcept = default;
  constexpr bit_value(byte_type val) noexcept;

  // Conversion
  constexpr operator bool() const noexcept;

  // Operations
  void set(bool val) noexcept;
  void set() noexcept;
  void reset() noexcept;
  void flip() noexcept;

  // Compound assignment operators
  template <class T> bit_value& operator+=(const T& val) noexcept;
  template <class T> bit_value& operator-=(const T& val) noexcept;
  template <class T> bit_value& operator*=(const T& val) noexcept;
  template <class T> bit_value& operator/=(const T& val) noexcept;
  template <class T> bit_value& operator%=(const T& val) noexcept;
  template <class T> bit_value& operator&=(const T& val) noexcept;
  template <class T> bit_value& operator|=(const T& val) noexcept;
  template <class T> bit_value& operator^=(const T& val) noexcept;
  template <class T> bit_value& operator<<=(const T& val) noexcept;
  template <class T> bit_value& operator>>=(const T& val) noexcept;

  // Increment and decrement operators
  bit_value& operator++() noexcept;
  bit_value& operator--() noexcept;
  bit_value operator++(int) noexcept;
  bit_value operator--(int) noexcept;
};

// Stream functions
template <class CharT, class Traits>
basic_ostream<CharT, Traits>& operator<<(
  basic_ostream<CharT, Traits>& os,
  const bit_value& x
);
template <class CharT, class Traits>
basic_istream<CharT, Traits>& operator>>(
  basic_istream<CharT, Traits>& is,
  bit_value& x
);
```

Figure 5: Bit value technical specifications

Bit reference specifications

The specifications of `std::bit_reference` are given on figures 6 and 7.

Bit reference synopsis

```
// Bit reference class template
template <class UIntType>
class bit_reference
{
public:

  // Types
  using underlying_type = UIntType;
  using size_type = size_t;

  // Lifecycle
  template <class T> constexpr bit_reference(const bit_reference<T>& other) noexcept;
  constexpr bit_reference(underlying_type& ref, size_type pos);

  // Assignment
  bit_reference& operator=(const bit_reference& other) noexcept;
  bit_reference& operator=(bit_value val) noexcept;

  // Conversion
  constexpr operator bool() const noexcept;
  constexpr operator bit_value() const noexcept;

  // Access
  constexpr bit_pointer<UIntType> operator&() const noexcept;

  // Operations
  void swap(bit_reference& other);
  void set(bool val) noexcept;
  void set() noexcept;
  void reset() noexcept;
  void flip() noexcept;

  // Compound assignment operators
  template <class T> bit_reference& operator+=(const T& val) noexcept;
  template <class T> bit_reference& operator-=(const T& val) noexcept;
  template <class T> bit_reference& operator*=(const T& val) noexcept;
  template <class T> bit_reference& operator/=(const T& val) noexcept;
  template <class T> bit_reference& operator%=(const T& val) noexcept;
  template <class T> bit_reference& operator&=(const T& val) noexcept;
  template <class T> bit_reference& operator|=(const T& val) noexcept;
  template <class T> bit_reference& operator^=(const T& val) noexcept;
  template <class T> bit_reference& operator<<=(const T& val) noexcept;
  template <class T> bit_reference& operator>>=(const T& val) noexcept;

  // Increment and decrement operators
  bit_reference& operator++() noexcept;
  bit_reference& operator--() noexcept;
  bit_value operator++(int) noexcept;
  bit_value operator--(int) noexcept;

  // Underlying details
  constexpr underlying_type* address() const noexcept;
  constexpr size_type position() const noexcept;
};
```

Figure 6: Bit reference technical specifications

Bit reference non-member functions

```cpp
// Swap specialization
template <class T>
void swap(
  bit_reference<T>& lhs,
  bit_reference<T>& rhs
) noexcept;

// Stream functions
template <class CharT, class Traits, class T>
basic_ostream<CharT, Traits>& operator<<(
  basic_ostream<CharT, Traits>& os,
  const bit_reference<T>& x
);
template <class CharT, class Traits, class T>
basic_istream<CharT, Traits>& operator>>(
  basic_istream<CharT, Traits>& is,
  const bit_reference<T>& x
);

// Make function
template <class T>
constexpr bit_reference<T> make_bit_reference(
  T& ref,
  typename bit_reference<T>::size_type pos
);
```

Figure 7: Bit reference non-member functions

Bit pointer specifications

The specifications of `std::bit_pointer` are given on figures 8 and 9.

Bit pointer synopsis

```cpp
// Bit pointer class template
template <class UIntType>
class bit_pointer
{
public:

  // Types
  using underlying_type = UIntType;
  using size_type = size_t;
  using difference_type = intmax_t;

  // Lifecycle
  template <class T> constexpr bit_pointer(const bit_pointer<T>& other) noexcept;
  bit_pointer() noexcept = default;
  constexpr bit_pointer(nullptr_t) noexcept;
  constexpr bit_pointer(underlying_type* ptr, size_type pos);

  // Assignment
  bit_pointer& operator=(const bit_pointer& ptr) noexcept;

  // Conversion
  explicit constexpr operator bool() const noexcept;

  // Access
  constexpr bit_reference<UIntType> operator*() const;
  constexpr bit_reference<UIntType>* operator->() const;
  constexpr bit_reference<UIntType> operator[](difference_type n) const;

  // Increment and decrement operators
  bit_pointer& operator++();
  bit_pointer& operator--();
  bit_pointer operator++(int);
  bit_pointer operator--(int);
  constexpr bit_pointer operator+(difference_type n) const;
  constexpr bit_pointer operator-(difference_type n) const;
  bit_pointer& operator+=(difference_type n);
  bit_pointer& operator-=(difference_type n);
};
```

Figure 8: Bit pointer technical specifications

Bit pointer non-member functions

```
// Non-member arithmetic operators
template <class T>
constexpr bit_pointer<T> operator+(
  typename bit_pointer<T>::difference_type n,
  const bit_pointer<T>& x
);
template <class T, class U>
typename common_type<
  typename bit_pointer<T>::difference_type,
  typename bit_pointer<U>::difference_type
>::type operator-(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;

// Comparison operators
template <class T, class U>
constexpr bool operator==(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;
template <class T, class U>
constexpr bool operator!=(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;
template <class T, class U>
constexpr bool operator<(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;
template <class T, class U>
constexpr bool operator<=(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;
template <class T, class U>
constexpr bool operator>(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;
template <class T, class U>
constexpr bool operator>=(
  const bit_pointer<T>& lhs,
  const bit_pointer<U>& rhs
) noexcept;

// Make function
template <class T>
constexpr bit_pointer<T> make_bit_pointer(
  T* ptr,
  typename bit_pointer<T>::size_type pos
);
```

Figure 9: Bit pointer non-member functions

Bit iterator specifications

The specifications of `std::bit_iterator` are given on figures 10 and 11.

Bit iterator synopsis

```
// Bit iterator class template
template <class Iterator>
class bit_iterator
{
public:

  // Types
  using iterator_type = Iterator;
  using underlying_type = typename iterator_traits<Iterator>::value_type;
  using iterator_category = typename iterator_traits<Iterator>::iterator_category;
  using value_type = bit_value;
  using difference_type = intmax_t;
  using pointer = bit_pointer<underlying_type>;
  using reference = bit_reference<underlying_type>;
  using size_type = size_t;

  // Lifecycle
  template <class T> bit_iterator(const bit_iterator<T>& other);
  bit_iterator();
  bit_iterator(const underlying_iterator& i, size_type pos);

  // Access
  reference operator*() const;
  pointer operator->() const;
  reference operator[](difference_type n) const;

  // Increment and decrement operators
  bit_iterator& operator++();
  bit_iterator& operator--();
  bit_iterator operator++(int);
  bit_iterator operator--(int);
  bit_iterator operator+(difference_type n) const;
  bit_iterator operator-(difference_type n) const;
  bit_iterator& operator+=(difference_type n);
  bit_iterator& operator-=(difference_type n);

  // Underlying details
  iterator_type base() const;
  constexpr size_type position() const noexcept;
};
```

Figure 10: Bit iterator technical specifications

Bit iterator: non-member functions

```
// Non-member arithmetic operators
template <class T>
bit_iterator<T> operator+(
  typename bit_iterator<T>::difference_type n,
  const bit_iterator<T>& i
);
template <class T, class U>
typename common_type<
  typename bit_iterator<T>::difference_type,
  typename bit_iterator<U>::difference_type
>::type operator-(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);

// Comparison operators
template <class T, class U>
bool operator==(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);
template <class T, class U>
bool operator!=(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);
template <class T, class U>
bool operator<(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);
template <class T, class U>
bool operator<=(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);
template <class T, class U>
bool operator>(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);
template <class T, class U>
bool operator>=(
  const bit_iterator<T>& lhs,
  const bit_iterator<U>& rhs
);

// Make function
template <class T>
bit_iterator<T> make_bit_iterator(
  const T& x,
  typename bit_iterator<T>::size_type pos
);
```

Figure 11: Bit iterator non-member functions

6 Discussion and open questions

As a first version, the intent of this proposal is to start a discussion about the introduction of basic bit utilities in the standard library. Several design options have been detailed in section 4, and the specification presented in part 5 represents only one option amongst multiple alternatives. Answering the following questions are of primary importance regarding design and specification choices:

- What types should be allowed as template parameters of `std::bit_reference` and `std::bit_pointer`? Only unsigned integers? All integral types? And what about bit containers?
- What functionalities and arithmetic should a bit implement? A design with `set`, `reset` and `flip` operators? Or one emulating a `bool` and nothing else? Or one adding the arithmetic behaviour of an unsigned integer of exactly one bit?
- Should `std::bit_value` be introduced to improve the global design? Should the naming `std::bit` be used instead of `std::bit_value`, even though the class is not a bit an only mimics the behaviour of a non-referenced bit value?
- Should bit references be both implicitly convertible to `bool` and `std::bit_value`?
- Should bit pointers be implicitly or explicitly convertible to `bool` to check their state?
- Should other overloads of `std::swap` and `std::exchange` be provided as described in the additional design remarks subsection?
- Should `const` versions of the class templates be provided separately in order to replace the solution consisting in passing `const T` as template parameters? Or should typedefs referring to `std::bit_reference<const T>` and `std::bit_pointer<const T>` be provided?
- How should the internal details, namely the address of the underlying value and the bit position, be accessed? Are `underlying_type`, `address` and `position` good names for these underlying details?
- What should happen when `pos >= std::numeric_limits<T>::digits`?
- Should a type traits helper structure such as `std::iterable_bits` be introduced to count the number of iterable bits of unsigned integral types in order to replace `std::numeric_limits<T>::digits`?
- What functions should be specified as `constexpr` and what members should be specified as `noexcept`? In particular, could the constructors of `std::bit_iterator` and its `base` member function be marked as `constexpr` to facilitate compile-time computation?
- Should any relation be introduced between `std::bit_reference` and `std::bitset::reference`? Or should they be kept as two completely independent entities in terms of design as in this version of the proposal?

Answering and achieving a consensus on these questions should lead to a minimalist but very versatile set of tools to manipulate unique bits.

Bit manipulation algorithms should be the subject of another proposal built on the top of the fundamental layer discussed here. Such a library could include a `std::bit_view`, as well as specializations of the standard algorithms. As already mentioned in section 2, thanks to the `address` and `position` members, the algorithms could operate on the `underlying_type` instead of operating on each bit, thus providing a significant speedup. For example, `std::count` could call the `popcnt` assembly function when operating on bit iterators. Moreover the set of standard algorithms could be extended with algorithms dedicated to bit operations. These extensions could include, amongst others, algorithms inspired by the very exhaustive proposal N3864 [Fioravante, 2014], algorithms implementing unsigned unbounded integer arithmetic, and algorithms based on the Bit Manipulation Instruction sets such as `parallel_bit_deposit` and `parallel_bit_extract`.

The resulting bit library could serve a wide range of purposes, from cryptography to video games, and from arbitrary-precision integral arithmetic to high performance computing. And, of course, it could finally offer a proper way to use unsigned integers as bit containers.

7 Acknowledgements

The authors would like to thank Howard Hinnant, Klemens Morgenstern, Tony Van Eerd, Vicente Botet Escriba, Jens Maurer, and the other contributors of the ISO C++ Standard - Discussion and of the ISO C++ Standard - Future Proposals groups for their initial reviews and comments.

Vincent Reverdy and Robert J. Brunner have been supported by the National Science Foundation Grant AST-1313415. Robert J. Brunner has been supported in part by the Center for Advanced Studies at the University of Illinois.

8 References

[ISO, 2014] (2014). Information technology – programming languages – c++. Technical Report 14882:2014, ISO/IEC.

[Becker, 2014] Becker, P. (2014). Proposal for unbounded-precision integer types. Technical Report N4038, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Carruth, 2014] Carruth, C. (2014). Efficiency with algorithms, performance with data structures. https://youtu.be/fHNmRkzxHWs.

[Fioravante, 2014] Fioravante, M. (2014). A constexpr bitwise operations library for c++. Technical Report N3864, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Hinnant, 2012] Hinnant, H. (2012). On vector<bool>. https://isocpp.org/blog/2012/11/on-vectorbool.

[Jones, 2011] Jones, D. (2011). Working draft, standard for programming language c. Technical Report N1570, ISO/IEC JTC1/SC22/WG14 - The C Standards Committee.

[Ozturk et al., 2012] Ozturk, E., Guilford, J., Gopal, V., and Feghali, W. (2012). New instructions supporting large integer arithmetic on intel architecture processors. Technical report, Intel.

[Reverdy, 2016] Reverdy, V. (2016). Implementation of bit utility class templates. https://github.com/vreverdy/bit_utilities.

[Siek et al., 2015] Siek, J. et al. (2015). Boost dynamic bitset. http://www.boost.org/doc/libs/1_59_0/libs/dynamic_bitset/dynamic_bitset.html.

[Smith, 2015] Smith, R. (2015). Working draft, standard for programming language c++. Technical Report N4567, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Stroustrup, 2013] Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edition.