Document number: PXXXXR0
Date: 2016−07−26
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: Library Evolution Working Group, Library Working Group
Reply to: Vincent Reverdy (vince.rev@gmail.com)

What should the semantic of wrapper classes be?

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 semantic of templated wrapper classes in the C++ language, including value, reference and pointer wrappers. Although these questions have arisen in the context of the bit utility proposal P0237, we think that they deserve a specific paper since they are likely to be common to a wide range of proxy classes. We are asking for general guidelines from LEWG and LWG about the semantic of such classes. In particular, cv-qualifiers and implicit constructors happen to raise non-trivial questions. In this paper, we present these questions, explore several options, and ask for guidelines.

Contents

1 Introduction **2**

2 The problem **2**
2.1 Values, references and pointers 2
2.2 Semantics of the non-wrapped cases 3
2.3 Semantics of the wrapped cases 3
2.4 Questions to be answered 4

3 Some possible options **5**
3.1 Preliminary note 5
3.2 Exact mapping and undefined behavior 5
3.3 Mapping with equivalences 5

4 Conclusion **6**

5 Acknowledgements **6**

6 References **7**

1 Introduction

In the bit utility proposal P0237[R0, R1, R2] [Reverdy and Brunner, 2016a, Reverdy and Brunner, 2016b, Reverdy et al., 2016], we explore the design space around bit abstractions to provide a way to manipulate bits efficiently in the standard library, leading to a common interface to build bit-oriented data structures and algorithms, including arbitrary precision arithmetic such as described in N4038 [Becker, 2014]. The proposal include four main classes: `std::bit_value`, `std::bit_reference`, `std::bit_pointer` and `std::bit_iterator` of which the last three are templated. `std::bit_reference` and `std::bit_pointer` take the underlying unsigned integral type in which bits are referenced as a template parameter, while `std::bit_iterator` is an iterator adaptor around an iterator with an unsigned integral `value_type`. In terms of design, `std::bit_value`, `std::bit_reference` and `std::bit_pointer` act as the `value_type`, `reference` and `pointer` typedefs of `std::bit_iterator`. A full detailed motivation of the class `std::bit_value` was included in P0237R0 and has been validated within LEWG by informal polls as reported in P0237R1.

Technical discussions at the Oulu 2016 ISO C++ meeting raised the question of making the class `std::bit_value` templated as `std::bit_reference` and `std::bit_pointer` so that a reference to a bit value `std::bit_value<UIntType>&` could be implicitly converted to a bit reference `std::bit_reference<UIntType>`, and a pointer to a bit value `std::bit_value<UIntType>*` could be implicitly converted to a bit pointer `std::bit_pointer<UIntType>`. This could be particularly useful to ease the writing of standard algorithms for proxy iterators, as explored in P0022R1 [Niebler, 2015].

We end up having four templated wrapper classes: a value, a reference, a pointer and an iterator. In the standard library, the difference between a `const_iterator` and a `const iterator` is not always obvious for new users, but after a while they get it. But as we will see, with wrappers around values, references and pointers the problem rapidly becomes a nightmare. Also, we would like having design guidelines on this particular problem, that could be directly applied within the context of our bit utilities proposal P0237, but which could also serve as a reference for library designers who are working on wrappers and on proxy iterators.

2 The problem

2.1 Values, references and pointers

In all the following we consider three templated wrapper classes:

- `value<T>`: a wrapper around a value that should mimic the behaviour of a value (e.g. `std::bit_value<UIntType>`)
- `reference<T>`: a wrapper around a reference that should mimic the behaviour of a reference (e.g. `std::bit_reference<UIntType>`)
- `pointer<T>`: a wrapper around a pointer that should mimic the behaviour of a pointer (e.g. `std::bit_pointer<UIntType>`)

As iterators are built on the top of values, references and pointers, we don't include them in the problem: they can be treated separately, and most of the conclusions drawn for pointer<T> can be applied to them. Moreover, at first, and to present the problem, we ignore the volatile qualifier and rvalue references.

2.2 Semantics of the non-wrapped cases

In the non-wrapped case, the const qualifier leads to 8 cases:

```
/*A*/ T
/*B*/ const T
/*C*/ T&
/*D*/ const T&
/*E*/ T*
/*F*/ const T*
/*G*/ T* const
/*H*/ const T* const
```

with implicit conversions between values and references leading to:

```
int val = 42;
const int cval = 42;

int& refval = val;           // Compiles
int& refcval = cval;         // Does not compile
const int& crefval = val;    // Compiles
const int& crefcval = cval;  // Compiles
```

2.3 Semantics of the wrapped cases

However, with wrappers, a total of 12 cases are introduced:

```
/*01*/ value<T>
/*02*/ value<const T>
/*03*/ const value<T>
/*04*/ const value<const T>
/*05*/ reference<T>
/*06*/ reference<const T>
/*07*/ const reference<T>
/*08*/ const reference<const T>
/*09*/ pointer<T>
/*10*/ pointer<const T>
/*11*/ const pointer<T>
/*12*/ const pointer<const T>
```

leading to a far more complicated situation regarding to implicit conversions between values and references:

```
value<int> val = 42;
value<const int> valc = 42;
const value<int> cval = 42;
const value<const int> cvalc = 42;

reference<int> ref_val = val;                    // ?
reference<int> ref_valc = valc;                  // ?
```

```cpp
reference<int> ref_cval = cval;                 // ?
reference<int> ref_cvalc = cvalc;               // ?

reference<const int> refc_val = val;            // ?
reference<const int> refc_valc = valc;          // ?
reference<const int> refc_cval = cval;          // ?
reference<const int> refc_cvalc = cvalc;        // ?

const reference<int> cref_val = val;            // ?
const reference<int> cref_valc = valc;          // ?
const reference<int> cref_cval = cval;          // ?
const reference<int> cref_cvalc = cvalc;        // ?

const reference<const int> crefc_val = val;     // ?
const reference<const int> crefc_valc = valc;   // ?
const reference<const int> crefc_cval = cval;   // ?
const reference<const int> crefc_cvalc = cvalc; // ?
```

where `// ?` means "Should it be considered as valid or not?".

With `volatile`, the situation becomes even worse, with a total of 48 cases, including things such a `volatile reference<const volatile T>`, and 256 possible conversions, compiling or not, between values and references.

2.4 Questions to be answered

All these possibilities of conversion raise important design questions, since finding the right set of constructors required to perform these conversions is not trivial. Consequently, we would like guidance from LEWG and LWG on the following:

What should the semantics of a set of value, reference and pointer wrappers around a type `T` be, especially regarding to cv-qualifiers?

In particular:

- What should the semantic of operators `*` and `&` be? Is it ok to leave the default operator `&` for `value<T>`, and applies a symmetry for `reference<T>` and `pointer<T>` with `reference<T>::operator&` returning a `pointer<T>` and `pointer<T>::operator*` returning a `reference<T>`, plus implicit conversions from `value<T>&` to `reference<T>` and from `value<T>*` to `pointer<T>`?
- How should the 12 cases mentioned in part 2.3 be mapped to the original 8 cases mentioned in part 2.2? Or in other words, what numbers are associated to what letters? For example: should the semantics of `value<const T>` and `const value<T>` be the same?
- What is the minimal set of constructors required to perform all the implicit conversions required by the answer to the preceding question?

3 Some possible options

3.1 Preliminary note

In the following, we explore several possibilities, mainly to illustrate the extent of the consequences of some choices. Other possibilities exist, and are very welcome by the authors of this paper.

3.2 Exact mapping and undefined behavior

The first option, which seems to us the easiest, the most conservative and the most practical, is to consider the following mapping:

Exact mapping: 8 non-wrapped cases ↔ 8 wrapped cases

Description	Non-wrapped version	Wrapped version
mutable value	`T`	`value<T>`
constant value	`const T`	`value<const T>`
reference to mutable	`T&`	`reference<T>`
reference to constant	`const T&`	`reference<const T>`
mutable pointer to mutable	`T*`	`pointer<T>`
mutable pointer to constant	`const T*`	`pointer<const T>`
constant pointer to mutable	`T* const`	`const pointer<T>`
constant pointer to constant	`const T* const`	`const pointer<const T>`

This means that the remaining 4 cv-qualified wrappers, namely `const value<T>`, `const value<const T>`, `const reference<T>` and `const reference<const T>` are considered as irrelevant, and their behavior may be considered as undefined. Providing a set of constructors to perform the implicit conversion between the 8 well-defined cases mimicking the semantics of the non-wrapped versions is relatively easy. This option is easy to extend to the `volatile` and `const volatile` versions. A user could still declare a `const value<const T>`, but the implicit conversions would not be taken care of automatically since the semantics of the wrapper would be considered as undefined.

3.3 Mapping with equivalences

A second possible option is to consider that some of the wrapped cases are equivalent:

Mapping with equivalences: 8 non-wrapped cases ↔ 12 wrapped cases

Description	Non-wrapped version	Wrapped version
mutable value	`T`	`value<T>`

constant value	`const T`	`value<const T>`
		`const value<T>`
		`const value<const T>`

reference to mutable	`T&`	`reference<T>`

reference to constant	`const T&`	`reference<const T>`
		`const reference<T>`
		`const reference<const T>`

mutable pointer to mutable	`T*`	`pointer<T>`
mutable pointer to constant	`const T*`	`pointer<const T>`
constant pointer to mutable	`T* const`	`const pointer<T>`
constant pointer to constant	`const T* const`	`const pointer<const T>`

The advantage of this option is that every cv-qualified wrapper has a well-defined semantic, leaving no room for undefined behavior. However, taking care of implicit conversions becomes very tricky: since a `value<const T>` is considered equivalent to a `const value<T>` and `reference<const T>` is considered equivalent to a `const reference<T>`, `value<const T>&` and `const value<T>` should be both implicitly convertible to `reference<const T>` and also `const reference<T>`. With the `volatile` qualifier, ensuring the right implicit conversions becomes even more tricky. If this solution is chosen, the authors of this paper are asking for guidance on the minimal set of implicit constructors required to perform the conversions to ensure the equivalences.

4 Conclusion

In this short paper, we raised questions about the semantics of templated wrapper classes. These questions originally came from research on bit utilities during the standardization process of P0237. As these questions are likely to arise for other proxy iterators and wrappers classes we are asking for more general guidance on this problem, thinking that this guidance would also ensure good practices outside of the standard library.

5 Acknowledgements

The authors would like to thank Tomasz Kaminski who raised the questions on making `bit_value` a template class. The authors would also like to thank the contributors of the ISO C++ Standard - Discussion and of the ISO C++ Standard - Future Proposals groups for their 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.

6 References

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

[Niebler, 2015] Niebler, E. (2015). Proxy iterators for the ranges extensions. Technical Report P0022R1, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Reverdy and Brunner, 2016a] Reverdy, V. and Brunner, R. J. (2016a). On the standardization of fundamental bit manipulation utilities. Technical Report P0237R0, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Reverdy and Brunner, 2016b] Reverdy, V. and Brunner, R. J. (2016b). Wording for fundamental bit manipulation utilities. Technical Report P0237R1, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.

[Reverdy et al., 2016] Reverdy, V., Brunner, R. J., and Myers, N. (2016). Wording for fundamental bit manipulation utilities. Technical Report P0237R2, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee.