

+ 25

std::optional <T&>

Optional Over References

STEVE DOWNEY



Cppcon
The C++ Conference

20
25



September 13 - 19

std::optional<T&>

OPTIONAL OVER REFERENCES

Steve Downey

Bloomberg

© 2025 Bloomberg Finance L.P. All rights reserved.

COLOPHON

- Slideware: [reveal.js](#)
- Slide Preparation: [org-reveal](#)
- Fonts: [Atkinson Hyperlegible](#) Next and Mono
- Color Themes: [Modus Vivendi](#) and [Operandi Tinted](#)

Intended to conform to [Web Content Accessibility Guidelines Level AAA](#)



Speaker notes

Please try to be considerate when making presentations. Accessibility helps everyone.

Try to present working code, even in slideware.

STANDARDISING OPTIONALS OVER REFERENCES

Optionals were first proposed for C++ in 2005.

Optional<T>, where T is constrained not to be a reference, was added in 2017.

Optionals for lvalue references were voted in at the Sofia meeting in June, 2025

Speaker notes

This talk will discuss the early history, starting with Boost.Optional and “[N1878: A Proposal to Add an Utility Class to Represent Optional Objects \(Revision 1\)](#)”, and what the early concerns were for the reference specialization. “[P1175R0: A Simple and Practical Optional Reference for c++](#)”, reproposed reference support for C++20, which was not adopted. “[P1683R0: References for Standard Library Vocabulary Types - an Optional Case Study](#)”, in 2020 surveyed existing behavior of optional references in the wild, and pointed out the trap of assingment behaviour being state dependent. “[P2988R0: Std:Optional](#)” picked up the torch again in 2023, of which revision 9 is the proposal which is design approved by the Library Evolution Working Group, and R13 was adopted.

In 2024, the proposal to make optional a range, “[P3168R0: Give Std:Optional Range Support](#)”, as opposed to having a separate range of zero or one, was adopted. The reference implementation for `optional<T&>` and the test cases for `views::maybe` were used to vet the additional interfaces for optional range support. This merged implementation became one of the first Beman libraries, where the library and the optional reference proposal benefited immensely from the visibility and feedback.

WHY SO LONG?

- What were the concerns that made the process take so long?
- How were concerns addressed?
- What did we end up with?
- What remains to be done?

Speaker notes

The core of the difficulty has been that references are not values and types containing a reference do not have value semantics. References do not fit comfortably in the C++ type system. The core value semantic type that also has reference semantics is a pointer, but pointers have underconstrained and unsafe semantics. The long discussion has been a proxy for what reference semantic types should look like in value semantic types in the standard library, particularly for "sum" types, like `expected` and `variant`.

QUICK OVERVIEW OF OPTIONAL<T>

- An owning type,
- with value semantics,
- with one additional out-of-band value.

ALGEBRAICALLY

T + 1

std::variant<T, std::monostate>

CORE USE CASE

```
int size;
if (optional<int> s = readConfigValue("Size")) {
    size = *s;
} else {
    size = 0;
}
return size;
```

C++26 RANGE VERSION

```
int size = 0;
for (int s : readConfigValue("Size")) {
    size = s;
}
return size;
```

DEFAULT OPTIONAL PARAMETER

```
constexpr int optParam(int a, optional<int> b = {}) {
    if (b) {
        return a + *b;
    }
    return a;
}
```

```
const auto t1 = optParam(3);
const auto t2 = optParam(3, 4);
static_assert(t1 != t2);
```

QUICK OVERVIEW OF OPTIONAL<T&>

- A non-owning type,
- with reference and value semantics,
- with one additional value representing the empty state.

CORE USE CASE

Looking up something for modification.

TODAY

```
auto i = map.find("one");
auto j = map.find("two");
auto k = map.find("three");
assert(i->second == 1);
assert(j->second == 2);
assert(k == map.end());
```

WITH OPTIONAL<INT&>

```
optional<int&> i = findInMap("one");
optional<int&> j = findInMap("two");
optional<int&> k = findInMap("three");
EXPECT_TRUE(*i == 1);
EXPECT_TRUE(*j == 2);
EXPECT_TRUE(!k);
*(findInMap("one")) = 3;
EXPECT_TRUE(*i == 3);
```

OPTIONAL REFERENCE PARAMETER

Instead of a pointer:

```
void doSomething(std::string const& data,
                 optional<Logger&> logger = {}) {
    for (auto l : logger) {
        l.log(data);
    }
    return;
}
```

EXISTING PROBLEMS WITH OPTIONAL

Construction and assignment from the underlying T or a type convertible to a T produces inevitable surprises and deep complication in the implementation.

Reasoning about overload sets is difficult for humans.

THE DESIGN PROBLEMS FOR REFERENCES

ASSIGN OR REBIND?

```
Cat fynn;
Cat loki;
optional<Cat&> maybeCat1;
optional<Cat&> maybeCat2{fynn};
maybeCat1 = fynn;
maybeCat2 = loki;
```

What do those assignments do?

Ought they be allowed?

State independence won out, eventually.

ALWAYS rebind.

Speaker notes

What assignment does is not dependent on the state of the optional.

It always rebinds the "reference", which is not possible with a C++ reference in a struct.

NON-GENERIC TEMPLATE

optional<T&> violates genericity.

The "vector<bool>" problem only for an entire value category.

Reference categories are weird and non-generic.

DESIGN CHOICES

`make_optional()`

`make_optional()` was largely supplanted by CTAD.

`make_optional<T&>()` creates an `optional<T>`.

Doing otherwise would have been worse.

VALUE CATEGORY'S AFFECT ON

`optional<T&>::value() &&`

What should `optional<T&>::value() &&` return?

Choose to model pointers, a reference semantic value type.

The value category of the object does not affect value category of the referent.

Otherwise, an rvalue `optional<T&>` could enable moves from the referent.

SHALLOW VS. DEEP const

What should `optional<T&>::value() const;` return?

Choose to model pointers, a reference semantic value type.

A `const` pointer is not a pointer to `const`.

All language references are `const`.

An `optional<T&>` is a reference semantic type.

Not a reference.

CONDITIONAL EXPLICIT

Is a spelled out `optional<T&>(x)` required to construct an `optional<T&>?`

Or can an `optional<T&>` be constructed implicitly from a `t`?

I would have preferred requiring explicit, but it was too painful in practice.

Lack of explicit makes the type exponentially more complex.

There are more interactions between member functions.

Speaker notes

However, lack of explicit makes the type exponentially more complex, as there are more interactions between member functions.

value_or()

What should `optional<T&>::value_or(U &&u);` return?

What is the "value type" for an optional?

All choices are surprising to someone.

Chose to return T, as that seems least dangerous.

It is what `optional<T>` does.

Speaker notes

Future work: generic **nullable** functions.

in_place_t CONSTRUCTION

There is no "place" to construct in to.

CONVERTING ASSIGNMENT

Avoid conversions that produce temporaries.

Avoid confusion with `optional<U&>` or `optional<T>` constructors.

Large *overload sets* are difficult to reason about.

REIFICATION PRINCIPLES

CONSTRUCTION FROM TEMPORARY

Avoid taking references to temporaries.

Rules out some safe cases, disallows many dangerous cases.

DELETING DANGLING OVERLOADS

Delete, rather than remove via concept, function overloads that produce dangling references.

ASSIGNMENT OF optional<T&>

Assignment of an optional<T&> is equivalent to a pointer copy.

All assignments are through the single function.

PROJECT BEMAN

BEGAN LAST YEAR AT C++NOW 2024

Not a requirement for Standardization.

Details matter.

Speaker notes

LEWG is getting better at asking for implementation of exact proposal.

PRE-EXISTING SMD::OPTIONAL

Confirmed at Tokyo, live, that the range-ification would work for my test cases for
views::maybe.

Unfortunately `smd::optional` used early-Modern CMake.
This meant rework to bring it to current standards.

THE REF-STEALING BUG FOUND

```
Cat fynn;
std::optional<Cat&> maybeCatRef{fynn};
std::optional<Cat> maybeCat;
maybeCat = std::move(maybeCatRef);
// fynn is moved from
```

Now fixed.

THE FIX

Don't move the result of operator*, move the rhs and apply operator*().

```
//instead of  
*std::move(rhs)  
// use  
std::move(*rhs)
```

Because

```
std::optional<T&>::operator*() && -> T&; // overload not actually present
```

does not return an rvalue reference.

Speaker notes

THE MODAL USE-CASE

The expected most common use is for looking up something and failure is not exceptional.

```
constexpr optional<mapped_type&> get(const key_type& k);
```

We plan to add this to associative containers for C++29.

P3091 BY PABLO HALPERN

Better Lookups for `map`, `unordered_map`, and `flat_map`

QUESTIONS?

Remember a question starts with:

- who
- what
- when
- where
- how
- why

and goes up at the end.

"More of a comment than a question ..."

hold them for a moment.

QUESTIONS?

COMMENTS?

THANK YOU!