C++ now

2025

std::optional<T&> — Standardizing Optionals over References

A Case Study

Steve Downey

COLOPHON

- Slideware: reveal.js
- Slide Preperation: org-re-reveal
- Fonts: Atkinson Hyperlegible Next and Mono
- Color Themes: Modus Vivendi and Operandi Tinted

Intended to conform to Web Content Accessibility Guidelines Level AAA

W3C WAI-AAA WCAG 2.2

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 are on track for C++26.

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.

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 immensly 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`, but also for types such as `single`.

QUICK OVERVIEW OF OPTIONAL<T&>

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

INTRO

```cpp
template <class T> class optional<T &> {
      public:
        using value_type = T;
        using iterator = implementation_defined;

      public:
```

CONSTRUCTORS

```
        constexpr optional() noexcept = default;
        constexpr optional(nullopt_t) noexcept : optional() {}
        constexpr optional(const optional &rhs) nboexcept = default;
```

CONSTRUCTORS (CONTINUED)

```
template <class Arg>
constexpr explicit optional(in_place_t, Arg &&arg);
template <class U>
constexpr explicit(/*see below*/)
        optional(U &&u) noexcept(/*see below*/);

template <class U>
constexpr explicit(/*see below*/)
        optional(optional<U> &rhs) noexcept(/*see below*/);

template <class U>
constexpr explicit(/*see below*/)
        optional(const optional<U> &rhs) noexcept(/*see below*/);

template <class U>
constexpr explicit(/*see below*/)
        optional(optional<U> &&rhs) noexcept(/*see below*/);

template <class U>
constexpr explicit(/*see below*/)
        optional(const optional<U> &&rhs) noexcept(/*see below*/);
```

DESTRUCTOR

```cpp
constexpr ~optional() = default;
```

ASSIGNMENT

```
constexpr optional &operator=(nullopt_t) noexcept;
constexpr optional &operator=(const optional &rhs) noexcept = default;
template <class U> constexpr T &emplace(U &&u) noexcept(/*see below*/);
```

SWAP

```
    constexpr void swap(optional &rhs) noexcept;
```

ITERATOR

```
        constexpr iterator begin() const noexcept;
        constexpr iterator end() const noexcept;
```

OBSERVERS

```cpp
constexpr T *operator->() const noexcept;
constexpr T &operator*() const noexcept;
constexpr explicit operator bool() const noexcept;
constexpr bool has_value() const noexcept;
constexpr T &value() const; // freestanding-deleted
template <class U = remove_cv_t<T>>
constexpr remove_cv_t<T> value_or(U &&u) const;
```

MONADIC OPERATIONS

```
template <class F> constexpr auto and_then(F &&f) const;

template <class F>
constexpr optional<invoke_result_t<F, T &>> transform(F &&f) const;

template <class F> constexpr optional or_else(F &&f) const;
```

MODIFIERS

```cpp
constexpr void reset() noexcept;
```

EXPOSITION-ONLY DETAILS

```
      private:
        T *val = nullptr; // exposition only

        template <class U>
        constexpr void convert_ref_init_val(U &&u); // exposition only
};
```

THE PAPERS

- 2005: N1878: A Proposal to Add an Utility Class to Represent Optional Objects (Revision 1)
- 2012: N3406: A Proposal to Add a Utility Class to Represent Optional Objects (Revision 2)
- 2013: N3527: A Proposal to Add a Utility Class to Represent Optional Objects (Revision 3)
- 2013: N3672: A Proposal to Add a Utility Class to Represent Optional Objects (Revision 4)
- 2015: N4529: Working Draft, C++ Extensions for Library Fundamentals, Version 2
- 2016: P0220R0: Adopt Library Fundamentals TS for c++17
- 2018: P1175R0: A Simple and Practical Optional Reference for C++
- 2020: P1683R0: References for Standard Library Vocabulary Types - An Optional Case Study
- 2023: P2988R0: `std:optional<T&>`

Speaker notes

Optional was pulled at the last moment of 14 because of UB in the implementation technique of placement new with a storage buffer. Library TSs hadn't fully failed at that point.

THE PROBLEMS

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.

Speaker notes

NON-GENERIC TEMPLATE

`optional<T&>` violates genericity.

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

Reference categories are weird and non-generic.

CONSTEXPR AND UB ISSUES

At the time of C++14 they couldn't quite be constexpr.

Placement new had issues as did `union` techniques.

We taught the compiler to `constexpr` more things.

DESIGN CHOICES

`make_optional()`

`make_optional()` was largely supplanted by CTAD.

`make_optional<T&>()` creates an `optional<T>`. Doing otherwise would have been worse.

TRIVIAL CONSTRUCTION

`is_trivial` is deprecated in 26.

No worse than they have to be.

VALUE CATEGORY AFFECT ON

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

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

Choose to model pointers, a reference semantic value type.

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 langauge references are `const`. An `optional<T&>` is a reference semantic type.

Not a reference.

CONDITIONAL EXPLICIT

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

I would have preferred to, but it was too painful.

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.

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.

THE T& PROBLEM

OVERLOADED SYNTAX

Used for:

OVERLOADED SYNTAX

Used for:

- Parameter Passing

OVERLOADED SYNTAX

Used for:

- Parameter Passing
- Named alias

OVERLOADED SYNTAX

Used for:

- Parameter Passing
- Named alias
- Non-null const pointer in a struct

REFERENCES ARE NOT DATA

They are CoData.

Much more about this in my Streams talk.

T& IN AN GENERIC ALGEBRAIC TYPE

T& IN AN GENERIC ALGEBRAIC TYPE

- Request for reference semantics.

T& IN AN GENERIC ALGEBRAIC TYPE

- Request for reference semantics.
- Not a request for T& weirdness.

T& IN AN GENERIC ALGEBRAIC TYPE

- Request for reference semantics.
- Not a request for T& weirdness.
- Biggest problem for Union-like types – Sum Types.

PROJECT BEMAN

BEGAN LAST YEAR AT C++NOW 2024

Not a requirement for Standardization.

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

Details matter.

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

Actually doesn't exist.

FUTURE STANDARDS WORK

`std::expected`

`std::variant`

`std::views::single`

```
rebindable_reference
```

EXPOSITION-ONLY `movable_box<T>`

QUESTIONS?

Remember a question starts with:

QUESTIONS?

Remember a question starts with:

- who

QUESTIONS?

Remember a question starts with:

- what

QUESTIONS?

Remember a question starts with:

- when

QUESTIONS?

Remember a question starts with:

- where

QUESTIONS?

Remember a question starts with:

- how

QUESTIONS?

Remember a question starts with:

- why

or

A propositional statement

a statement that has a truth value, either true or false, but not both.

and goes up at the end.

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

Is a propositional statement, but hold them for a moment.

COMMENTS?

THANK YOU!