More flexible `optional::value_or()`

Marc Mutz

Document #: P2218R0
Date: September 15, 2020
Audience: LEWGI
Reply-to: Marc Mutz <marc.mutz@kdab.com>

Abstract

We propose to extend the `value_or()` member function template in `optional` in three ways:

1. Adding a default template argument to make requesting default-constructed values simpler:

```
// now                          // proposed:
opt.value_or(Type{});           opt.value_or({});
```

   This brings `value_or()` in line with other functions (most prominently `exchange()`).

2. Adding a new emplace-like overload:

```
// now                          // proposed:
opt.value_or(Type{});           opt.value_or_construct();
```

   This optimizes `value_or()` for types that are expensive to construct.

3. Adding a lazy version of the latter:

```
// proposed:
opt.value_or_else([] -> Type { return {}; });
```

   further optimizing `value_or_construct()` at the cost of more verbosity.

Contents

1 Motivation and Scope

When using `optional::value_or()`, more often than not, the fall-back value passed is some form of default-constructed value:

```
optional<int> oi = ~~~;          // (1)
use(oi.value_or(0));
optional<bool> ob = ~~~;         // (2)
use(ob.value_or(false));
optional<string> os = ~~~;       // (3)
use(os.value_or(nullptr));       // (a)
use(os.value_or(""));            // (b)
use(os.value_or({}));            // (c)
use(os.vlue_or(string{});        // (d)
optional<vector<string>> ov = ~~~;
use(ov.value_or(~~~???~~~));     // (4)
```

While this works fine in case of built-in types (1, 2), it already fails to be convenient when the payload type is a user-defined type without literals.

1.1 How the C++ Developer Became a Gardener

Here's the tale of a C++ developer trying to use `value_or()` in the `string` case (3): The developer first tries to use `nullptr` (a), which crashes on him at runtime due to [**char.traits.require**]**/1** in conjunction with [**string.cons**]**/13**. The next try (b) succeeds, but may invoke an unnecessary "`strlen`", so he's told in review to use the `string` default constructor instead. So the developer tries (c) which fails to compile because {} fails to deduce the template argument of `value_or()`, which is not defaulted, as e.g. the second argument of `exchange()` is. Grumpily, the developer caves in and repeats the type name of the `optional`'s `value_type` (d).

The next day, he's asked to use a `optional<vector<string>>` (4) and decides to quit and become a gardener instead.

We propose two different, orthogonal, solutions to the problem:

- Default the `value_or` template argument, so `value_or({})` works, and/or
- Add an emplacement-like function `value_or_construct(auto&&...)`, so that `value_or_construct()` works.

The latter addition gives rise to:

- Add a lazy version, `value_or_else(Func&&)`.

1.2 Defaulting `value_or()`'s template argument

With this change, we'd like to ensure that `value_or({})` works, like `exchange(var, {})` does.

We can't just default like this:

```
template <typename T>
class optional {
public:
    ~~~~
    template <typename U = T>
    T value_or(U&&) const;
};
```

as that would prevent moving the argument into the return value when `T` is cv-qualified (as in `optional<const string>`). It follows that we need to remove cv-qualifiers. We don't need to remove references, as `optional<T&>` is ill-formed. If and when optional references become supported, this needs to be rethought.

```
template <typename T>
class optional {
public:
    ~~~~
    template <typename U = remove_cv_t<T>>
    T value_or(U&&) const;
};
```

This enables developers to write `value_or({})`, which is self-explanatory, as long as you know `value_or()` as currently specified.

It also enables all other braced initializers, not just {}, to be passed to `value_or()`.

1.3 Adding emplace-like `value_or_construct()`

The second change was suggested to the author in very early discussions on the LEWG(I) reflector: If `value_or()` was a variadic emplace-like function, then `opt.value_or()` would return a default-constructed value if `opt` is not engaged.

While this extension would be SC and BC[1], this author does not believe that `value_or()` is a good name for such a function: What does `opt.value_or()` *look like*? Can a developer that knows `value_or()` as currently specified make sense of this expression? This author doubts that very much. To him, this looks like "value or nothing". Then what's the "nothing" that's being returned? Another `optional` specialisation?

So, it seems to this author that just making `value_or()` emplace-like would be counter-intuitive, but at the same time such functionality could be useful. E.g., even if `value_or({})` was enabled (as per Section 1.2), that call would create a default-constructed `T` which is then moved into the return value, instead of default-constructing the return value directly. Adding a new function for this purpose seems the best way forward.

Taking a cue from existing factory functions in the standard (`Allocator::construct()`), this author ended up with `value_or_construct()` as the suggested name for the variadic function. See Section 4.1 for alternative names.

[1] The variadic version could overload the existing unary version by constraining the variadic version to `sizeof...(Args) != 1`

1.4 Adding lazy `value_or_else()`

The third change was also suggested in the initial discussion on the LEWG(I) mailing list. While `value_or_construct()` already defers contruction of the `T` to when it is actually needed, it still requires construction of the *arguments* of construction. For cases where even that is too much, this author suggests to add a lazy version, `value_or_else()`, too:

```
optional<vector<string>> opt = ~~~;
// this works today, with optimal efficiency, but only for lvalues:
auto v0 = opt ? *opt : vector{"Hello"s, "world"s} ;
// value_or constructs a full vector even when not needed:
auto v1 = opt.value_or({"Hello"s, "World"s});
// value_or_construct() still constructs an initializer_list<string>:
auto v2 = opt.value_or_construct({"Hello"s, "World"s});
// value_or_else() would construct nothing:
auto v3 = opt.value_or_else([] { return vector{"Hello"s, "World"s}; });
```

While `value_or_construct()` and `value_or_else()` solve the same problem, this author thinks that they have sufficient drawbacks each to warrant adding both, to wit:

- `value_or_construct()` may be very inefficient, asking to construct possibly-expensive constructor arguments before we know they're needed. Without `value_or_else()`, the developer is required to perform a manual check (cf. `v0` above), which only works for lvalues.
- `value_or_else()` may be too complex and/or verbose, with no efficiency gains compared to `value_or_construct()` when passing cheap contructor arguments:

  ```
  optional<QPen> opt = ~~~;
  auto c1 = opt.value_or(Qt::NoPen); // passing an enum value is cheap
  auto c2 = opt.value_or_construct(Qt::NoPen); // ditto
  auto c3 = opt.value_or_else([]{ return Qt::NoPen; }); // needlessly verbose
  ```

2 Impact on the Standard

Only positive. Expressions enabled by this proposal make the use of `optional::value_or()` easier and more consistent with the rest of the standard library, in particular, `std::exchange()`. At the same time, no existing code is broken, because the status quo cannot accept braced intializers as `value_or()` arguments.

3 Proposed Wording

All wording is relative to [N4861]:

- In [**version.syn**], add a feature macro `__cpp_lib_optional_value_or` with the value calculated as usual and comment "`// also in <optional>`".
- Change [**optional.optional**] as indicated:

```
    constexpr const T&& value() const&&;
-   template<class U> constexpr T value_or(U&&) const&;
-   template<class U> constexpr T value_or(U&&) &&;
+   template<class U=remove_cv_t<T>> constexpr T value_or(U&&) const&;
+   template<class U=remove_cv_t<T>> constexpr T value_or(U&&) &&;
+   template<class... Args> constexpr T value_or_construct(Args&&... args) const&;
+   template<class... Args> constexpr T value_or_construct(Args&&... args) &&;
+   template<class U, class... Args> constexpr T value_or_construct(initializer_list<U> il,
+   template<class U, class... Args> constexpr T value_or_construct(initializer_list<U> il,
+   template<class F> constexpr T value_or_else(F&& f) const&;
+   template<class F> constexpr T value_or_else(F&& f) &&;

    // [optional.mod], modifiers
```

- Apply the above `remove_cv_t<T>` default argument also to the declarations of `value_or()` just above [**optional.observe**]/**17** and [**optional.observe**]/**19**.
- At the end of [**optional.observe**], add:

```
template<class... Args> constexpr T value_or_construct(Args&&... args) const&
```

Mandates: `is_copy_constructible_v<T> && is_constructible_v<T, Args...>` is `true`.
Effects: Equivalent to:

```
  return bool(*this) ? **this : T(std::forward<Args>(args)...);
```

```
template<class... Args> constexpr T value_or_construct(Args&&... args) &&
```

Mandates: `is_move_constructible_v<T> && is_constructible_v<T, Args...>` is `true`.
Effects: Equivalent to:

```
  return bool(*this) ? std::move(**this) : T(std::forward<Args>(args)...);
```

```
template<class U, class... Args>
  constexpr T value_or_construct(initializer_list<U> il, Args&&... args) const&
```

Mandates: `is_copy_constructible_v<T> &&`
`is_constructible_v<T, initializer_list<U>&, Args...>` is `true`.
Effects: Equivalent to:

```
  return bool(*this) ? **this : T(il, std::forward<Args>(args)...);
```

```
template<class U, class... Args>
  constexpr T value_or_construct(initializer_list<U> il, Args&&... args) &&
```

Mandates: `is_move_constructible_v<T> &&`
`is_constructible_v<T, initializer_list<U>&, Args...>` is `true`.
Effects: Equivalent to:

```
  return bool(*this) ? std::move(**this) : T(il, std::forward<Args>(args)...);
```

`template<class F> constexpr T value_or_else(F&& f) const&`

Let `U` be `invoke_result_t<F>`.
Mandates: `is_copy_constructible_v<T> && is_convertible_v<U, T>` is `true`.
Effects: Equivalent to:

```
return bool(*this) ? **this : std::forward<F>(f)();
```

`template<class F> constexpr T value_or_else(F&& f) &&`

Let `U` be `invoke_result_t<F>`.
Mandates: `is_move_constructible_v<T> && is_convertible_v<U, T>` is `true`.
Effects: Equivalent to:

```
return bool(*this) ? std::move(**this) : std::forward<F>(f)();
```

4 Design Decisions

If all we wanted was to make it easier to return a default-constructed `T`, we could just add a new function `value_or_default_initialized()`. This is not proposed, because it does not address the consistency concern with `exchange()`.

As mentioned in Section 1.3, just making `value_or()` variadic leaves a lot to be desired: while `opt.value_or(0xff, 0xff, 0xff)` works reasonably well for a `optional<color>`, it doesn't really work for default construction, which is the driver behind this proposal. So this author does not propose to make `value_or()` variadic, but suggests to choose a different name for this functionality.

This author chose to make `value_or_else()` take just a single invokable, not a `bind`- or `thread`-style N-ary argument list. The reason was twofold: First, the single-argument version is consistent with the P0798-proposed `or_else()`. Second, this author considers the `thread` constructor and `bind` functions to be old-fashioned APIs that predate the introduction of lambdas, requiring use of `reference_wrapper`, which makes such APIs hard to use.

4.1 Naming

The `value_or()` function is pre-existing, so the name is fixed.

For the emplacement-style function, the following names were considered by this author:

- `value_or()` works well for N-ary arguments, $N > 0$, but not ery well for $N = 0$, which is the major motivation for this proposal in the first place.
- `value_or_make()` emplacement-style factory functions have traditionally been called `make_xxx`, but those are free functions, not class member functions. Members, indeed, tend to be called `construct()` (example: `Allocator`).
- `value_or_constructed()` (using the past participle form of *construct* instead) arguably more correct form, gammatically, but unknown in the case of the standard API, so not proposed.

For the lazy version, no other names but `value_or_else()` come to mind, so no alternatives were considered.

5 Acknowledgements

The author would like to thank all participants of the LEWG(I) reflector discussion that led to this proposal, esp. Andrzej Krzemienski for confirming that `value_or()`'s non-defaulted template parameter was not a conscious omission. Barry Revzin suggested `value_or_else()` and mentioned the alternative name `value_or_construct()` and this author never looked back.

6 References

[N4861] Richard Smith (editor)
Working Draft: Standard for Programming Language C++
http://wg21.link/N4861