More flexible `optional::value_or()`

Marc Mutz

Document #:	N3887
Date:	September 2, 2020
Project:	Programming Language C++ Library Evolution Group Incubator
Reply-to:	Marc Mutz <marc.mutz@kdab.com>

Abstract

We propose to extend the `value_or()` member function template in `optional`, so as to make requesting default-constructed values easier:

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

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

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_make(auto&&...)`, so that `value_or_make()` works.

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_v<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.

1.3 Adding emplace-like `value_or_make()`

The second change was suggested to the author in very early discussions on the LEWGI 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.

Taking a cue from existing factory functions in the standard, this author ended up with `value_or_make()` as the suggested name for the variadic function.

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. At the same time, no existing code is broken.

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

3 Proposed Wording

All wording is relative to [N4861]:

- In [**version.syn**], add a feature macro `__cpp_lib_optional_value_or` with the usual value 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_make(Args&&... args) const&;
+   template<class... Args> constexpr T value_or_make(Args&&... args) &&;

    // [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_make(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_make(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)...);
```

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.

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.

6 References

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