More flexible `value_or()`

Marc Mutz

Document #:	D2218R1
Date:	February 14, 2023
Audience:	LEWGI
Target:	C++26

Reply-to:	Marc Mutz <marc.mutz@hotmail.com>

Abstract

We propose to extend the `value_or()` member function templates in `optional` and `expected` by 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()`).

Contents

0 Change History

0.1 R0 → R1

- Dropped `value_or_construct()` and `value_or_else()`. The former is too ambitious, the latter superseded by C++23's adoption of `or_else()`.

- Extended to `expected::value_or()`, added to the IS after R0 was published.
- Rebased onto [N4928].

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.value_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.

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. This is unaffected by the proposed resolution to [LWG3424], btw., see Section 4. We don't need to remove references, as `optional<T&>` and `expected<T&,E>` are ill-formed. If and when `optional` and/or `expected` start to support references, 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()`.

2 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()`.

See P2218R0 for rationale on not making `value_or()` variadic instead.

3 Impact on the Standard

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

4 Relation to Other Proposals

- [P2248R7] is a wider-scope version of this paper, addressing a similar issue across `<algorithm>`.
- [LWG3424] addresses the same functions, but deals with the return value instead of the argument. As it's orthogonal to this proposal; neither subsumes the other. We note that the proposed wording at the time of writing omits `exchange::value_or()`, which, however, seems to have the same issue as `optional::value_or()`.

5 Proposed Wording

All wording is relative to [N4928]:

Conflict resolution note: If the proposed resolution to [LWG3424] is accepted, the intent of this paper is to change the template-initializer-list, and adopt [LWG3424]'s return value, see Section 4.

- In [**version.syn**], add a new row

```
#define __cpp_lib_value_or      YYYYMML // also in <expected>, <optional>
```

- Change [**optional.optional.general**] 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&&) &&;

  // [optional.monadic], monadic operations
```

- Apply the above `remove_cv_t<T>` default argument also to the declarations of `value_or()` just above [**optional.observe**]/**15** and [**optional.observe**]/**17**.

- Change [**expected.object.general**] as indicated:

```
    constexpr E&& error() && noexcept;
-   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 G = E> constexpr E error_or(G&&) const &;
```

  [Note: [**expected.void**] does not offer `value_or()`, so needs no changes]

- Apply the above `remove_cv_t<T>` default argument also to the declarations of `value_or()` just above [**expected.object.obs**]/**16** and [**expected.object.obs**]/**18**.

6 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 `optional::value_or()`'s non-defaulted template parameter was not a conscious omission, and Jonathan Wakely and Tomasz Kamiński for dragging the paper out of its two-year hiatus.

7 References

[LWG3424] Casey Carter (reporter)
optional::value_or should never return a cv-qualified type
https://wg21.link/LWG3424

[N4928] Thomas Köppe (editor)
Working Draft: Standard for Programming Language C++
https://wg21.link/N4928

[P2248R7] Giuseppe D'Angelo
Enabling list-initialization for algorithms
https://wg21.link/P2248R7