A more flexible `optional::value_or` (else!)

Document #: P3413R0
Date: 2024-10-15
Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We introduce `value_or_construct` and `value_or_else` to complement `std::optional::value_or` and `std::expected::value_or`.

Motivation

This is a follow up to P2218R0 [3] (which is no longer pursed by its author). With the design mostly unchanged, except that we apply the changes to `std::expected` as well. The changes mades by LWG3886 [1] are left out of this paper (as the issue is currently tentatively ready).

We proposed to add, in addition to the existing `value_or`

- A `value_or_construct` member function for `optional` and `expected` constructing an object lazily from its argument. This improve both the performance and ergonomics of `value_or`.

  We choose a to give this facility a new name (`value_or_construct`) rather than adding overloads to (`value_or` - which would be technically possible) - because feedback on the original paper that in the empty case, the meaning of (`opt.value_or()`) would be cryptic and confusing.

- A `value_or_else` member function for `optional` and `expected` taking a lazily evaluated callable, to alleviate the cost of constructing expensive object.

`error_or` ?

We do do not propose similar functions for `expected:::error_or` as this seems less motivated.

Impact on optional<T&>

P2988R7 [2] might go in the direction to support `value_or` and to, in that case, return a non-reference type. If that direction comes to pass, the proposed `value_or_construct` and `value_or_else` can be directly applied to the `optional<T&>`. Beyond that, the discussions about the interaction of `value_or` and `optional<T&>` are best left to the `optional<T&>` paper. What we are proping are merely more convenient ways to spell the behavior of `value_or`.

Wording

� Class template `optional` [optional.optional]

� General [optional.optional.general]

```
namespace std {
template<class T>
class optional {
    public:
    constexpr bool has_value() const noexcept;
    constexpr const T& value() const &;     // freestanding-deleted
    constexpr T& value() &;                 // freestanding-deleted
    constexpr T&& value() &&;               // freestanding-deleted
    constexpr const T&& value() const &&;   // freestanding-deleted
    template<class U> constexpr T value_or(U&&) const &;
    template<class U> 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, Args&&... args>) const &;

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

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

    // ??, monadic operations
    template<class F> constexpr auto and_then(F&& f) &;
    template<class F> constexpr auto and_then(F&& f) &&;
    template<class F> constexpr auto and_then(F&& f) const &;
    template<class F> constexpr auto and_then(F&& f) const &&;
    template<class F> constexpr auto transform(F&& f) &;
    template<class F> constexpr auto transform(F&& f) &&;
    template<class F> constexpr auto transform(F&& f) const &;
    template<class F> constexpr auto transform(F&& f) const &&;
    template<class F> constexpr optional or_else(F&& f) &&;
    template<class F> constexpr optional or_else(F&& f) const &;
};

}
```

� Observers [optional.observe]

```
template<class U> constexpr T value_or(U&& v) const &;
```

Mandates: `is_copy_constructible_v<T> && is_convertible_v<U&&, T>` is `true`.

Effects: Equivalent to:

```
    return has_value() ? **this : static_cast<T>(std::forward<U>(v));
```

```
template<class U> constexpr T value_or(U&& v) &&;
```

Mandates: `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is `true`.

Effects: Equivalent to:

```
    return has_value() ? std::move(**this) : static_cast<T>(std::forward<U>(v));
```

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

```
template <invocable 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 has_value() ? **this : std::forward<F>(f)();
```

```
template <invocable 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 has_value() ? std::move(**this) : std::forward<F>(f)();
```

� Class template `expected` [expected.expected]

� General [expected.object.general]

```
namespace std {
template<class T, class E>
class expected {
    public:
    // ??, observers
    constexpr const T* operator->() const noexcept;
    constexpr T* operator->() noexcept;
    constexpr const T& operator*() const & noexcept;
    constexpr T& operator*() & noexcept;
    constexpr const T&& operator*() const && noexcept;
    constexpr T&& operator*() && noexcept;
    constexpr explicit operator bool() const noexcept;
    constexpr bool has_value() const noexcept;
    constexpr const T& value() const &;                                    //
    freestanding-deleted
    constexpr T& value() &;                                                //
    freestanding-deleted
    constexpr const T&& value() const &&;                                  //
    freestanding-deleted
    constexpr T&& value() &&;                                              //
    freestanding-deleted
    constexpr const E& error() const & noexcept;
    constexpr E& error() & noexcept;
    constexpr const E&& error() const && noexcept;
    constexpr E&& error() && noexcept;
    template<class U> constexpr T value_or(U&&) const &;
    template<class U> 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, Args&&... args>) const &;

    template<class U, class ... Args>
    constexpr T value_or_construct (initializer_list <U> il, Args&&... args>) &&;
    template <class F> constexpr T value_or_else (F&& f) const &;
    template <class F> constexpr T value_or_else (F&& f) &&;

    template<class G = E> constexpr E error_or(G&&) const &;
    template<class G = E> constexpr E error_or(G&&) &&;
};
}
```

� Observers [expected.object.obs]

```
template<class U> constexpr T value_or(U&& v) const &;
```

Mandates: `is_copy_constructible_v<T>` is `true` and `is_convertible_v<U, T>` is `true`.

Returns: `has_value() ? **this : static_cast<T>(std::forward<U>(v))`.

```
template<class U> constexpr T value_or(U&& v) &&;
```

Mandates: `is_move_constructible_v<T>` is `true` and `is_convertible_v<U, T>` is `true`.

Returns: `has_value() ? std::move(**this) : static_cast<T>(std::forward<U>(v))`.

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

```
    template <invocable 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 has_value() ? **this : T(std::forward<F>(f)());
```

```
template <invocable 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 has_value() ? std::move(**this) : T(std::forward<F>(f)());
```

Feature test macros

Bump `__cpp_lib_optional` and `__cpp_lib_expected` to the date of adoption.

[*Editor's note:* This does not conflicts with `optional<T&>` as the paper chooses to introduce a new macro] .

Acknowledgments

We would like to thanks Marc Mutz for the original paper (P2218R0 [3]).
Thanks to Barry Revzin for providing wording feedback.

References

[1] Casey Carter. LWG3886: Monad mo' problems. https://wg21.link/lwg3886.

[2] Steve Downey and Peter Sommerlad. P2988R7: std::optional<t&>. https://wg21.link/p2988r7, 9 2024.

[3] Marc Mutz. P2218R0: More flexible optional::value_or(). https://wg21.link/p2218r0, 9 2020.