Abstract

This paper proposes the addition of emplace factories for `future<T>` and emplace functions for `promise<T>` as we have proposed for `any` and `optional` in P0032R2.

Table of Contents

- History
- Introduction
- Motivation
- Proposal
- Design rationale
- Proposed wording
- Implementability
- Open points
- Acknowledgements
- References

History

Revision 2

Take in account the feedback from the LEWG weekly reflector:

- Fix a lot of typos

Revision 1
Take in account the feedback from Kona:

- Clean up the proposal a bit.
- Remove the `make_ready_future` overloads taking a `remove_reference_t<T>`.
- Explain why there were integer template parameters.
- Remove noexcept from the `make_ready_future()` factory functions.
- Added a comparison table for `make_ready_future`.

In addition:

- Any references to `std::experimental::optional` have been replaced by `std::optional`.

### Introduction

This paper proposes the addition of emplace factories for `future<T>` and emplace functions for `promise<T>` as we have proposed for `any` and `optional` in [P0032R2](#).

### Motivation

While we have added the `future<T>` factories `make_ready_future` and `make_exceptional_future` into [P0159R0](#), we don't have emplace factories as we have for `shared_ptr` and `unique_ptr` and we have for `any` and `optional`.

The C++ standard should be coherent for features that behave the same way on different types and complete, that is, don't miss features that could make the user code more efficient.

### Proposal

We propose to:

- Add `promise<T>::set_value(Args...)` member function that emplaces the value instead of setting it.
- Add `future<T>` emplace factory `make_ready_future<T>(Args...)`.

### Emplace assignment for promises

Some times a promise setter function must construct the promise value type and possibly the exception, that is the value or the exceptions are not yet built.

Before
Note that we need to repeat `X`.

With this proposal we can just emplace either the value or the exception.

```
void promiseSetter(std::promise<X>& p, bool cnf) {
    if (cnf)
        p.set_value(X(a, b, c));
    else
        p.set_exception(std::make_exception_ptr(MyException(__FILE__, __LINE__)));
}
```

Note that not only the code can be more efficient, it is also clearer and more robust as we don't repeat `X`.

**Emplace factory for futures**

Some `future` producer functions may know how to build the value at the point of construction and possibly the exception. However, when the value type is not available it must be constructed explicitly before making a ready future. The same applies for a possible exception that must be built.

Before

```
future<X> futureProducer(bool cnf1, bool cnf2) {
    if (cnf1)
        return make_ready_future(X(a, b, c));
    if (cnf2)
        return make_exceptional_future<X>(MyException(__FILE__, __LINE__));
    else
        return somethingElse();
}
```

With this proposal we can just write as much code and have possibly a more efficient implementation.
```cpp
def future<X> futureProducer(bool cnd1, bool cnd2) {
    if (cnd1)
        return make_ready_future<X>(a, b, c);
    if (cnd2)
        return make_exceptional_future<X>(MyException(__FILE__, __LINE__));
    else
        return somethingElse();
}
```

**Building a future**

In order to deduce a reference we need to use `std::ref`:

```cpp
int v=0;
std::future<int&> x = std::experimental::make_ready_future(std::ref(v));
```

However we want also to be able to force the future value as a template parameter:

```cpp
int v=0;
std::future<int&> x = std::experimental::make_ready_future<int&>(v);
```

We believe this usage would appear in generic contexts and is for this reason desirable.

**Comparison of `make_ready_future` factory**

In this table we use `mrf` instead of `make_ready_future` for layout concerns.
Design rationale

Why should we provide some kind of emplacement for `future` / `promise`?

Wrapping and type-erasure classes should all provide some kind of emplacement as it is more efficient to emplace than to construct the wrapped/type-erased type and then copy or assign it.

The current standard and the TS provide already a lot of such emplace operations, either in place constructors, emplace factories, emplace assignments.

Why emplace factories instead of `in_place` constructors?

`std::optional` provides in place constructors and emplace factory.

This proposal just extends the current future factories to emplace factories.

Should we provide a future `in_place` constructor? For coherency purposes and in order to be generic, yes, we should. However we should also provide a constructor from a `T` which doesn't exists neither. This paper doesn't proposes this yet.
Promise emplace assignments

`std::optional` provides emplace assignments via `optional::emplace()` and provides emplace factory.

We believe `promise<T>` should provide and similar interface. However, a promise accepts to be set only once, and so the function name should be different for the authors.

`reference_wrapper<T> overload to deduce T&`

As it is the case for `make_pair` when the parameter is `reference_wrapper<T>`, the type deduced for the underlying type is `T&`.

How to ensure that the template parameter `T` is not deduced?

If we had the following overload

```cpp
template <class T>
future<experimental::unwrap_ref_decay_t<T>> make_ready_future(T& x); // (1)
```

the following call will be accepted by (1) resulting in a `future<int>`, as the type is decayed.

```cpp
int v=0;
auto x = std::experimental::make_ready_future<int&>(v);
```

Clearly, we don't want the result to be `future<int>` but `future<int&>`.

Adding at least a default `int` template parameter as follows

```cpp
template <int=0, ...int, class T>
future<experimental::unwrap_ref_decay_t<T>> make_ready_future(T& x); // (1)
template <class T, class ...Args>
future<T> make_ready_future(Args&&... args); // (2)
```

avoids the selection of overload (1) and selects (2).

Impact on the standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++ 14.

Proposed wording
The wording is relative to P0159R0.

The current wording make use of `unwrap_ref_decay_t` as proposed in P0318R1, but if this is not accepted the wording can be changed without too much troubles.

2 Improvements to std::future and Related APIs

2.2 Header `<experimental/future>` synopsis

Replace the make_ready_future declaration in [header.future.synop] by

```cpp
namespace std {
    namespace experimental {
        inline namespace concurrency_v2 {
            future<void> make_ready_future();
            template <class T>
            future<void> make_ready_future();
            template <class T>
            future<unwrap_ref_decay_t<T>> make_ready_future(T&& x);
            template <class T, class ...Args>
            future<T> make_ready_future(Args&& ...args);
            template <class T, class U, class ...Args>
            future<T> make_ready_future(initializer_list<U> il, Args&& ...args);
        }
    }
}
```

2.5 Class template promise

Add [futures.promise] the following in the synopsis

```cpp
template <class ...Args>
void promise<R>::set_value(Args&& ...args);
template <class U, class... Args>
void promise<R>::set_value(initializer_list<U> il, Args&&... args);
```

Add the following

```cpp
template <class ...Args>
void promise<R>::set_value(Args&& ...args);
```

**Effects:** Atomically initializes the stored value as if direct-non-list-initializing an object of type `R` with the arguments `forward<Args>(args)...)` in the shared state and makes that state ready. These function shall not participate in overload resolution unless `is_constructible<R, Args&&...>` and `R` is not `void` and not a reference `S&`. 
Postconditions: this contains a value.

Throws and Error conditions as before

```cpp
template <class U, class... Args>
void promise::set_value(initializer_list<U> il, Args&&... args);
```

Requires: is_constructible<R, initializer_list<U>&, Args&&...>

Effects: atomically initializes the stored value as if direct-non-list-initializing an object of type `R` with the arguments `il, forward<Args>(args)...` in the shared state and makes that state ready.

Postconditions: this contains a value.

Throws and Error conditions as before

2.10 Function template `make_ready_future` [futures. make_ready_future ]

Replace in `[futures. make_ready_future ]` by the following.

```cpp
future<void> make_ready_future();
template <class T>
future<void> make_ready_future();
```

Effects: The function creates a shared state that is immediately ready and returns a future associated with that shared state. The type of the shared state is `void`.

Returns: A future associated with that shared state.

Postconditions: For the returned future, `valid() == true` and `is_ready() == true`.

Throws: Any exception thrown by the construction.

Remark: The second overload shall not participate in overload resolution unless `is_void_v<T>`.

```cpp
template <class T>
future<unwrap_ref_decay_t<T>> make_ready_future(T&& value);
```

Let `V` be `unwrap_ref_decay_t<T>`

Effects: The function creates a shared state that is immediately ready and returns a future associated with that shared state. The type of the shared state is `V` and the result is constructed from `std::forward<T>(value)`.

For the second overload, the type of the shared state is void.

Returns: A future associated with that shared state.

Postconditions: For the returned future, `valid() == true` and `is_ready() == true`.
Throws: Any exception thrown by the construction.

Remark: This function shall not participate in overload resolution unless the template argument \( T \) is deduced.

```cpp
template <class T, class ...Args>
future<T> make_ready_future(Args&& ...args);
```

Effects: The function creates a shared state immediately ready and initializes the stored value as if direct-non-list-initializing an object of type \( R \) with the arguments \( \text{forward<Args>(args)...} \) in the shared state.

Returns: A future associated with that shared state.

Postconditions: For the returned future, \( \text{valid() == true} \) and \( \text{is_ready() == true} \).

Throws: Any exception thrown by the construction.

Remark: These functions shall not participate in overload resolution unless the \( \text{is_constructible_v<T, Args&&...>} \).

```cpp
template <class T, class U, class ...Args>
future<T> make_ready_future(initializer_list<U> il, Args&& ...args);
```

Effects: The function creates a shared state immediately ready and initializes the stored value as if direct-non-list-initializing an object of type \( R \) with the arguments \( \text{il, forward<Args>(args)...} \) in the shared state.

Returns: A future associated with that shared state.

Postconditions: For the returned future, \( \text{valid() == true} \) and \( \text{is_ready() == true} \).

throws: Any exception thrown by the construction.

Remark: These functions shall not participate in overload resolution unless the \( \text{is_constructible_v<T, initializer_list<U>, Args&&...>} \).

**Implementability**

Boost.Thread contains an implementation of the emplace value functions. make.impl contains the implementation of the factories.

**Open Points**

The authors would like to have an answer to the following points if there is at all an interest in this proposal. Most of them are bike-shedding about the name of the proposed functions:

emplace_ versus make_ factories
shared_ptr and unique_ptr factories make_shared and make_unique emplace already the underlying type and are prefixed by make_. For coherency purposes the function emplacing future should use also make_ prefix.

**promise::emplace** versus **promise::set_value**

promise<R> has a set_value member function that accepts the value.

```cpp
void promise::set_value(const R& r);
void promise::set_value(R&& r);
void promise<R>::set_value(R& r);
void promise<void>::set_value();
```

There is no reason for constructing an additional R to set the value, we can emplace it it

```cpp
template<typename ...Args>
void promise<R>::set_value(Args&&... as);
```

optional names this member function emplace. However, a promise accepts to be set only once, and so the function name should be different. Should we add a new member emplace function to promise<T> or overload set_value?

**If promise::set_value is retained, do we want to add in_place_t?**

Aaryaman Sagar has proposed to add the 'inplace' parameter

```cpp
template<typename... Args>
void set_value(std::in_place_t, Args&&... args);
```

```cpp
template<typename U, typename... Args>
void set_value(std::in_place_t, std::initializer_list<U> ilist, Args&&...)
```

Do we want to be so explicit?

**Future work**

In addition to emplace value functions we could also have emplace exceptions functions. This would need to update also exception_ptr emplace factories. While this cases can perform better, the exceptional case need less optimizations.
Acknowledgements

Thanks to all those that gave some feedback during the LEWG Weekly review. Thanks to Jonathan Wakely for his suggestion to limit the proposal to the emplace value cases which should be more consensual. Many thanks to Agustin K-ballo Bergé from which I learn the trick to implement the different overloads. Many thanks to Patrice Roy for presenting the P0319R0. Thanks to Aaryaman Sagar for the `inplace` suggestion.

References

- **N4480** N4480 - Working Draft, C++ Extensions for Library Fundamentals
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html

- **P0032R0** P0032 - Homogeneous interface for variant, any and optional
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.pdf

- **P0032R2** P0032 - Homogeneous interface for variant, any and optional - Revision 1
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0032r2.pdf

- **P0159R0** P0159 - Draft of Technical Specification for C++ Extensions for Concurrency
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0159r0.html

- **P0318R1** unwrap_ref_decay and unwrap_reference

- **P0338R3** - C++ generic factories

- **make.impl** C++ generic factory - Implementation
  https://github.com/viboes/std-make/blob/master/include/experimental/stdmakev1/make.hpp