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

1. Introduction
2. Motivation
3. Proposal
4. Design rationale
5. Proposed wording
6. Implementability
7. Open points
8. Acknowledgements
9. References

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 could have for any and optional if P0032R2 is accepted.

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_exception(E) member function that sets a promise exception_ptr from an exception.

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

- Add future<T> emplace factory
  emplace_ready_future<T>(Args...) / make_ready_future<T>(Args...).

- Add future<T> emplace factory
  emplace_exceptional_future<T,E>(Args...)/make_exceptional_future<T,E>(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

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

Note that we need to repeat X.

With this proposal we can just emplace either the value or the exception.
void producer(promise<int> & p) {
    if (cnd) p.set_value(a, b, c);
    else p.set_exception(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

**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**

```cpp
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();
}
```

The same reasoning than the previous section applies here. With this proposal we can just write less code
and have more (as possible more efficient).

```cpp
future<int> futureProducer(bool cnd1, bool cnd2) {
    if (cnd1)
        return make_ready_future<int>(a, b, c);
    if (cnd2)
        return make_exceptional_future<int>(MyException(__FILE__, __LINE__));
    else
        return somethingElse();
}
```

**Design rationale**

**Why should we provide some kind of emplacement for**
**Why emplace factories instead of in_place constructors?**

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

std::experimental::optional provides in place constructors and it could provide emplace factory if P0032R0 is adopted.

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::experimental::optional provides emplace assignments via \( \text{optional}::\text{emplace()} \) and it could provide emplace factory if P0032R0 is accepted.

We believe that \( \text{promise}<T> \) 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.

**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 \( \text{decay\_unwrap\_t} \) as proposed in P0318R0, but if this is not accepted the wording can be changed without too much troubles.

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

```cpp
namespace std {
namespace experimental {
inline namespace concurrency_v2 {

    template <int=0, int ..., class T>
    future<decay_unwrap_t<T>> make_ready_future(T&& x) noexcept;

template <class T>
    future<T> make_ready_future(remove_reference<T> const& x) noexcept;

template <class T>
    future<T> make_ready_future(remove_reference<T> && x) noexcept;

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

X.Y Class template  promise

Add [futures.promise] the following in the synopsis

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

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

Add the following

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

Requires: is_constructible<R, 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.

Postconditions: this contains a value.

[NDLR] Throws and Error conditions as before
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 \( \text{il, forward<Args>(args)...} \) in the shared state and makes that state ready.

Postconditions: this contains a value.

[NDLR] Throws and Error conditions as before

Function template make ready future

[NDLR] Add to [futures.make_ready_future] the following

template <class T>
future<T> make_ready_future(remove_reference<T> const& v) noexcept;
template <class T>
future<T> make_ready_future(remove_reference<T> && r) noexcept;
template <class T, class ...Args>
future<T> make_ready_future(Args&&... args) noexcept;

Effects: The function creates a shared state immediately ready emplacing the \( T \) with \( x \) for the first overload, \( \text{forward<T>(r)} \) for the second and \( T\{\text{args...}\} \) for the third.

Returns: A future associated with that shared state.

Postconditions: The returned future contains a value.

Implementability

Boost.Thread contains an implementation of the emplace value functions.

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 a

```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

```cpp
template <typename ...Args>
void promise::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`?

**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 Jonathan Wakely for his suggestion to limit the proposal to the emplace value cases which should be more consensual.

**References**

- [P0032R0](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html) P0032 - Homogeneous interface for variant, any and optional
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

P0318R0 decay_unwrap and unwrap_reference
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0318r0.pdf

P0338R0 - C++ generic factories
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0338r0.pdf

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