Document number: DXXXX
Date: 2015–10-20
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Emplacing promise<T>, future<T> and exception_ptr

This paper proposes the addition of emplace factories for `future<T>` as we have proposed for `any` and `optional` in [P0032R0].

Contents

Introduction

This paper proposes the addition of emplace factories for future<T> as we have proposed for any and optional in [P0032R0].

Motivation and Scope

While we have added the future<T> factories make_ready_future and make_exceptional_future into [N4256], we don't have emplace factories as we have for shared_ptr and unique_ptr and we could have for any and optional if [P0032R0] is adopted. The same rationale that motivated the emplace factories make_shared and make_unique applies to future<T>, that is performances.

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

Proposal

We propose to:

- Add exception_ptr emplace factory make_exception_ptr<E>(Args...) that emplaces any exception on an exception_ptr instead of moving it.
- Add future<T> emplace factory make_ready_future<T>(Args...).
- Add future<T> emplace factory make_exceptional_future<T,E>(Args...).
- Add promise<T>::set_value(Args...) member function that emplaces the value instead of setting it.
- Add promise<T>::set_exception<E>(Args...) member function that emplaces the exception E instead of setting it.

Tutorial

Emplace factory for futures

Emplace assignment for promises

Design rationale

Why should we provide some kind of emplacement?

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 emplace constructors?

`std::experimental::optional` provides in place constructors and it could provide emplace factory if [P0032R0] is accepted.

Should we provide a future in place constructor?

Promise emplace assignments

`std::experimental::optional` provides emplace assignments vie ::emplace(). constructors and it could provide emplace factory if [P0032R0] is accepted.

Is there something missing in the language?

It is cumbersome and artificial to create all these “make_ functions”. Why the C++ language don't help us here?

P0091R0 proposes extending template parameter deduction for functions to constructors of template classes. This is a 1st step that avoid writing such factories when the template are deduced from the parameters.

```
return shared_ptr(a);
```

However, this is not the case for the emplace factories. We need to state explicitly the type to be wrapped.

```
return make_shared<T>(a1, a2);
```

The following doesn't do anymore emplacement and in addition `T` is duplicated

```
return shared_ptr<T>(T(a1, a2));
```

Using `in_place` constructors if we had them for `shared_ptr, results in`

```
return shared_ptr<T>(in_place, a1, a2);
```

P0091R0 combined with in_place as proposed in P0032R0 allows us to have a in place factory

```
return shared_ptr(in_place<T>, a1, a2);
```

This is yet more verbose than the original make_ emplace factory and doesn't avoid the definition of some kind of in place factory throgh an specific constructor

```
return make_shared<T>(a1, a2);
```

I'm not sure if P0091R0 has another limitation, as I don't know if the following is the correct P0091R0 idiom when we are writing generic code

```
template <template <class> class TC, class T>
TC<T> f() {
  T a;
  ...
  return TC(a); // would this be correct?
}
```

Open points

The authors would like to have an answer to the following points if there is at all an interest in this proposal:

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

```
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.

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

However optional names this member function emplace. Should we add a new member emplace function or overload set_value?

promise::emplace_exception<E> versus promise<T>::set_exception<E>

The same applies to promise<R>::set_exception member function that could accept

```
template < typename E, typename ...Args>
void promise<R>::set_exception(Args&& as);
```

Alternatively we could name this function emplace_exception.

Technical Specification

The wording is relative to [N4538].

Header <experimental/type_traits> synopsis

Add the following declarations in []

```
template <class T>
struct decay_unwrap;

template <class T>
using decay_unwrap_t = typename decay_unwrap<T>::type;
```

Header <experimental/exception> synopsis

Replace the make_ready_future declaration in [support.exception] by

```
template <class E>
exception_ptr make_exception_ptr(E e) noexcept;
template <class E, class ...Args>
exception_ptr make_exception_ptr(Args&& ...args) noexcept;
```

Header <experimental/future> synopsis

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

```
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;
```

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

```
template <class T>
future<T> make_exceptional_future(exception_ptr p) noexcept;
template <class T, class E>
future<T> make_exceptional_future(E e) noexcept;
template <class T, class E, class ...Args>
future<T> make_exceptional_future(Args&& ...args) noexcept;
```

Class template promise

Replace [futures.promise] the following declaration

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

Effects: atomically stores the value `r`, `r`, `R{forward<Args>(args)...)`, `r` and nothing respectively in the shared state and makes that state ready.

Throws and *Error conditions* as before

```
void set_exception(exception_ptr p);
template <class E>
void set_exception(E e);
template <class E, class ...Args>
void set_exception(Args&& ...args);
```

Effects: atomically stores the exception pointer `p` , `e`, `E{args}` respectively in the shared state and makes that state ready.

Function template make_ready_future

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, `forward<T>(r)` for the second and `T{args...}` for the third.

Returns: A future associated with that shared state.

Function template make_exceptional_future

Add to [futures.make_exceptional_future] the following

```
template <class T>
future<T> make_exceptional_future(exception_ptr ex);
template <class T, class E>
future<T> make_exceptional_future(E e);
template <class T, class E, class ...Args>
future<T> make_exceptional_future(Args&& ...args);
```

Effects : The function creates a shared state immediately ready emplacing the `exception_ptr` with `p` for the first overload, `e` for the second and `E{args...}` for the third.

Returns: A future associated with that shared state.

Implementation

[Boost.Thread] contains an implementation of the future interface. However the exception_ptr emplace factory has not been implemented yet.

Acknowledgements

Many thanks to Agustín K-ballo Bergé from which I learnt the trick to implement the different overloads.

References

[N4480] N4480 - Working Draft, C++ Extensions for Library Fundamentals

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html

[N4480] Technical Specification for C++ Extensions for Concurrency

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4538.pdf

[P0032R0] P0050 – Homogeneous interface for variant, any and optional

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.pdf

[Boost.Thread]