Document number: DXXXX
Date: 2015–10-31
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>` and `exception_ptr` and emplace assignment for `promise<T>` as we have proposed for `any` and `optional` in [P0032R0].

Contents

Introduction

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

In addition this paper suggest some type traits as `decay_unwrap` and why not `unwrap_reference`. Now that we have replaced INVOKE by std::invoke, maybe `decay_copy` could be standardized also. These last propositions are completely independent and could be the subject of another paper.

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`, `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 even if only a little bit.

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.

We propose also to:

- Add an `unwrap_reference` type trait
- Add a `decay_unwrap` type trait
- Add `decay_copy` function

Tutorial

Emplace factory for exception_ptr

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

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

Note that we need to repeat X and MyException.

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 neither X neither MyException.

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 be built.

Before

```
future<int> futureProducer(bool cnd1, bool cnd2) {
  if (cnd1)
    return make_ready_future(X(a, b, c));
  if (cnd2)
    return make_exceptional_future<MyException>(
      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)

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

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::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 purpose and in order to be generic, yes, we should. However we should also provide a constructor from a `T` which doesn't exists neither.

Promise emplace assignments

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

`decay_unwrap` type trait

`decay_unwrap` is an implementation detail and doesn't impact the other features. However, the author find that it makes the wording simpler.

Compare

Returns: `pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));` where `V1` and `V2` are determined as follows: Let `Ui` be `decay_t<Ti>` for each `Ti`. Then each `Vi` is `X&`

if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

With

Returns: `pair<decay_unwrap_t<T1>, decay_unwrap_t<T2>>(std::forward<T1>(x), std::forward<T2>(y));`

If the trait is not adopted, the author suggest to use DECAY_UNWRAP(T) and define it only once on the standard.

This trait can already be used in the following cases

- [pair.spec] p8
- [tuple.creation] p2,3

To the knowledge of the author `decay_unwrap` is used already in HPX, and in Boost.Thread as `deduce_type`.

`unwrap_reference` type trait

`decay_unwrap` can be defined n function of `decay` and a more specific `unwrap_reference` type trait.

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

```
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`. Why do we prefer?

Do we want a decay_unwrap type trait?

If the traits is not adopted, the author suggest to use `DECAY_UNWRAP(T)`, define it only once on the standard and adapt [pair.spec] p8 and [tuple.creation] p2,3.

Do we want `DECAY_UNWRAP` instead?

Should it be named unwrap_decay instead?

As what I really done is to first decay and then unwrap reversing would swapping the two words be better in English?

A better name for decay_unwrap?

Do we want a unwrap_reference ?

Do we want to adopt the decay_copy function?

If `decay_unwrap` is adopted, do we want to adopt the `decay_copy` function that would replace `DECAY_COPY`?

Technical Specification

The wording is relative to [N4538].

General utilities library

Type Traits

Metaprogramming and type traits

Add the following declarations in [type_traits.synop]

```
template <class T>
struct decay_unwrap;

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

Let U be decay_t<T>. Then decay_unwrap<T>::type is
X& if U equals reference_wrapper<X>,
U otherwise.

Language support library

Exception handling

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;
```

Thread support library

Futures

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;
```

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

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

Class template promise

Add [futures.promise] the following in the synopsis

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

template <class E, class ...Args>
void set_exception(Args&& ...args);
template <class E, class U, class... Args>
void set_exception(initializer_list<U> il, Args&&... args);
```

Add the following

```
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 `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

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

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

Effects: atomically initializes the the exception pointer 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 an exception.

[NDLR]Throws and *Error conditions* as before

```
template <class E, class U, class... Args>
void set_exception(initializer_list<U> il, Args&&... args);
```

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

Effects: atomically initializes the the exception pointer 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 an exception.

[NDLR]Throws and *Error conditions* as before

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.

Postconditions: The returned future contains a value.

Function template make_exceptional_future

Add to [futures.make_exceptional_future] the following

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

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

Returns: A future associated with that shared state.

Postconditions: The returned future contains a value.

Implementation

[Boost.Thread] contains an implementation of the `future` interface. However the `exception_ptr` emplace functions have not been implemented yet, and so `promise::set_exception<E>(a1, …, aN)` as it can not ensure a real emplace.

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]

[HPX]

Appendix

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?
}
```