Document number: N4048
Date: 2014-05-26
Revises: N3865
Project: JTC1.22.32 Programming Language C++
Reply to: Vicente J. Botet Escribá <vicente.botet@wanadoo.fr>

More Improvements to std::future<T> - Revision 1

This paper complements "N3784 Improvements to `std::future<T>` and Related APIs" [4] with more future observers and factories. It is a revision of N3865 [3] making coherent the expected proposal [6] and this one.

The background of the additions of this paper respect to the previous revision is green.

The background of the suppression is magenta.

Contents

1 History **1**

2 Introduction **2**

3 Motivation and Scope **2**

4 Impacts on the Standard **3**

5 Design rationale **3**

6 Proposed Wording **4**

7 Implementability **10**

8 Acknowledgement **10**

1 History

- 1st revision:
  - Rename `next` by `bind`, `recover` by `catch_error`, and added `map` to be in line with the Expected proposal.
  - Make use of the `unexpected_type<E>` template class to be in line with the Expected proposal.
  - Rename the free function `make_exceptional_ptr()` by `make_unexpected<>()`.
  - Rename `exceptional_ptr` by `unexpected_type<exception_ptr>`.
  - Change the `get_exception_ptr()` member function by a free function `exception_ptr_cast()`.
- This revision was presented during the Issaquah meeting in revision to N3865 [3].
  - Removed `get_value()` as the exception guaranties can not be improved respect to `get()`.
  - Replaced `make_exceptional_future(Exception)` by `make_exceptional(Exception)`.
  - Make `future`/`shared_future` implicitly convertible from the result of `make_exceptional`.
  - Added `when_swapped_any(Range)`.
  - Associated a wide contract to `future<T>::get_exception_ptr()`.

2 Introduction

This proposal is an evolution of the functionality of `std::future`/`std::shared_future` that complements the proposal "N3784 Improvements to `std::future<T>` and Related APIs" [4] with more future observers and factories mainly based on the split between futures having a value or a exception.

3 Motivation and Scope

This proposal introduces the following asynchronous observer operations:

`.has_value()`:

This observer has the same "raison d'etre" than the `ready()` function proposed in [4] respect to the function `.then()`, but in this case respect to the functions `.map()`, `.bind()` and `.catch_error()` respectively.

.get_exception_ptr `exception_ptr_cast()`:

In some situations it is needed to store the exception stored in a `future` on another data structure.

With the current interface we need to try to get the value on a try-catch block and store the current exception on the data structure.

```
try { f.get(); } catch(...) { ex = std::current_exception;() };
```

Been able to retrieve the stored exception pointer would not only simplify the user code but also improve the efficiency of this code. The .get_exception_ptr `exception_ptr_cast()` function is there for this purpose.

```
ex = f.get_exception_ptr();
```

```
ex = exception_ptr_cast(f);
```

`.value_or(v)`:

Quite often the user has a fallback value that should be used when the future has an exception.

```
x = f.value_or(v);
```

is a shortcut for

```
try { x = f.get(); } catch(...) { x = v };
```

And the following future factories

`.next(f)` and `.recover(r)`:

`.map(f)`, `.bind(f)` and `.catch_error(r)`:

These functions behave like `.then()` but will call the continuation only when the future is ready with a value or an exception respectively. The continuations takes the future value type or an `exception_ptr` as parameter respectively. This has the advantage to make easier the continuation definition as is a lot of cases there is no need to protect the `future<T>::get()` operation against an exception thrown.

`.fallback_to(v)`:

Sometimes the user has a fallback value that should be used when the future has an exception. This factory creates a new future that will fallback to the parameter if the source future will be ready with an exception. The following

```
f.fallback_to(v);
```

is a shortcut for

```
f.then([](future<T> f) {
  return f.value_or(v);
})
```

`make_exceptional(e)`:

`make_unexpected(e)` and `future<T>(unexpected_type<E>)`:

We think that the case for functions that know that an exception must be thrown at the point of construction are as often than the ones that know the value. In both cases the result is know immediately but must be returned as future. By using make_exceptional `make_unexpected` a future can be created implicitly which hold a precomputed exception on its shared state.

`when_all`/`when_any`/`when_swapped_any`:

New overloads of the `when_all()`/`when_any`/`when_swapped_any` factories that take a range of futures as argument. And return a future container of the future values.

4 Impacts on the Standard

These changes are entirely based on library extensions and do not require any language features beyond what is available in C++ 11/14. The definition of a standard representation of asynchronous operations described in this document will have very limited impact on existing libraries, largely due to the fact that it is being proposed exactly to enable the development of a new class of libraries and APIs with a common model for functional composition.

5 Design rationale

5.1 `.next` / `recover` `.map` / `.bind` / `catch_error`

The proposal to include `.map` / `.bind` / `.catch_error` to the standard provides the ability to sequentially compose two futures by declaring one to be the continuation of another. With `.map`/`.bind` the antecedent future has a value before the continuation starts as instructed by the lambda function. The single difference is that `.map` wraps always the result and `.bind` does it only if the result is not a future. With `.catch_error` the antecedent future has an exception before the continuation starts as instructed by the lambda function.

In the example below the `future<int> f2` is registered to be a continuation of `future<int> f1` using the `.map` member function. This operation takes a lambda function which describes how `f2` should proceed with the future value. If the future is ready having an exception this functions returns the future itself.

```
#include <future>
using namespace std;
int main() {
  future<int> f1 = async([]() { throw "foo"; });

  future<string> f2 = f1
    .map([](int v) {
      return v.to_string();
    })
    .catch_error([](exception_ptr ex) {
        return "nan";
    });
}
```

As `.then()` these functions allows to chain multiple asynchronous operations. By using `.map` / `.bind` / `.catch_error`, creating a chain of continuations becomes straightforward and intuitive:

```
myFuture.map(...).bind (...).bind (...).catch_error(...).
```

Some points to note are:

- Each continuation will not begin until the preceding has completed.
- The exception thrown by a continuation are caught by the respective functions and the exception is propagated to the resulting future type.

Input Parameters:

- Lambda function: The lambda function on `map()`/`bind()` takes a `future<t>::value_type`. The lambda function on `catch_error()` takes an `exception_ptr`. Both could return whatever type. This makes propagating exceptions straightforward. This approach also simplifies the chaining of continuations.
- Executor: As `future<T>::then()`, an overloaded version on `.map`/`.bind`/`.catch_error` takes a reference to an executor object as an additional parameter. See there for more details.
- Launch policy: As `future<T>::then()`.

Return values: a `future` as it does `future<t>::then()`.

5.2 `has_value()` and .get_exception_ptr `exception_ptr_cast()`

The concept of checking if the shared state has a value or an exception already exists in the standard today. For example, calling `.get()` on a function internally checks if the shared state has a value, and if it isn't it throws an exception. These functions expose this ability to check the shared state to the programmer, and allows them to bypass the act of using a try-catch block to catch the stored exception. The example below illustrates using the ready member function to avoid using a try-catch block to manage with exceptions.

```
#include <future>
using namespace std;

int main() {

  future<int> f1 = async([]()  return possibly_long_computation(); );
  // later on when needed
  if(!f1.ready()) {
    //if not ready, attach a continuation and avoid a blocking wait
    f1.map([] (int v) {
      process_value(v);
    });
  } else if (f.has_value()) {
  //if ready and has_value, then no need to add continuation,
  // process value right away
      process_value(f1.get());
  } else
      process_exception(f1.get_exception_ptr());
      process_exception(exception_ptr_cast(f1));
}
```

The decision to add these functions as a member of the `future` and `shared_future` classes was straightforward, as this concept already implicitly exists in the standard (In particular Boost.Thread provides them since the beginning). Note that this functionality can not be obtained by the user directly. By explicitly allowing the programmer to check the shared state of a `future`, improvements on performance can be made.

5.3 `make_unexpected`

This function creates an exceptional instance implicitly convertible to a `future<T>` for a given exception. If no value is given then a `future<T>` is returned with the current exception stored. These functions are primarily useful in cases where sometimes, the exceptional case is immediately available, but sometimes it is not. The example below illustrates, that in an error path the value is known immediately, however in other usual path needing a short computation there is no need to do this task asynchronously. Last in the less usual path the function must return an eventual value represented as a `future` as it could take long time.

```
future<int> compute(int x)

  if (x < 0) return make_unexpected(invalid_argument());
  if (x == 0)
    try  do_some_short_work();
    catch (...)  make_unexpected();

  future<int> f1 = async([]()  return do_some_long_work(x); );
  return f1;
```

6 Proposed Wording

The proposed changes are expressed as edits to N3797, the C++ Draft Standard [1]. The wording has been adapted from N3857 [5].

Insert a new section (Shared with
cppexpected.

X.Y Unexpected objects [unexpected]

X.Y.1 In general [unexpected.general]

This subclause describes class template `unexpected_type` that wraps objects intended as unexpected. This wrapped unexpected object is used to be implicitly convertible to other object.

X.Y.2 Header <experimental/unexpected> synopsis [unexpected.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
  // X.Y.3, Unexpected object type
  template <class E>
  struct unexpected_type;
  // X.Y.4, Unexpected exception_ptr specialization
  template <>
  struct unexpected_type<exception_ptr>;

  // X.Y.5, Unexpected factories
  template <class E>
  constexpr unexpected_type<decay_t<E>> make_unexpected(E&& v);
  unexpected_type<std::exception_ptr> make_unexpected_from_current_exception();
}}}
```

A program that necessitates the instantiation of template `unexpected` for a reference type or `void` is ill-formed.

X.Y.3 Unexpected object type [unexpected.object]

```
template <class E=std::exception_ptr>
class unexpected_type {
public:
    unexpected_type() = delete;
    constexpr explicit unexpected_type(E const&);
    constexpr explicit unexpected_type(E&&);
    constexpr E const& value() const;
};
```

```
    constexpr explicit unexpected_type(E const&);
```

Effects:

Build an unexpected by copying the parameter to the internal storage.

```
    constexpr explicit unexpected_type(E &&);
```

Effects:

Build an unexpected by moving the parameter to the internal storage.

```
    constexpr E const& value() const;
```

Returns:

A const reference to the stored error.

X.Y.4 Unexpected exception_ptr specialization [unexpected.exception_ptr]

```
template <>
class unexpected_type<std::exception_ptr> {
public:
    unexpected_type() = delete;
    explicit unexpected_type(std::exception_ptr const&);
    explicit unexpected_type(std::exception_ptr&&);
```

```
    template <class E>
      explicit unexpected_type(E);
    std::exception_ptr const &value() const;
};
```

```
constexpr explicit unexpected_type(exception_ptr const&);
```

Effects:

Build an unexpected by copying the parameter to the internal storage.

```
constexpr explicit unexpected_type(exception_ptr &&);
```

Effects:

Build an unexpected by moving the parameter to the internal storage.

```
constexpr explicit unexpected_type(E e);
```

Effects:

Build an unexpected storing the result of make_exception_ptr(e).

```
constexpr exception_ptr const& value() const;
```

Returns:

A const reference to the stored `exception_ptr`.

X.Y.5 Factories [unexpected.factories]

```
template <class E>
constexpr unexpected_type<decay_t<E>> make_unexpected(E&& v);
```

Returns:

`unexpected<decay_t<E>>(v).`

```
constexpr unexpected_type<std::exception_ptr> make_unexpected_from_current_exception();
```

Returns:

`unexpected<std::exception_ptr>(std::current_exception()).`

Update section

30.6.1 Overview [futures.overview]

Header <future> synopsis

```
namespace std {
  ...


  // 30.6.x, Algebraic factories
  template <class Range>
  future<see below> when_any(Range rng);
  template <class Range>
  future<see below> when_all(Range rng);
  template <class Range>
  future<see below> when_swapped_any(Range rng);

  template <class R>
  exception_ptr exception_ptr_cast(future<R>&&);
```

```
  template <class R>
  exception_ptr exception_ptr_cast(shared_future<R>&&);
  template <class R>
  exception_ptr exception_ptr_cast(shared_future<R> const&);


}
```

Update section

30.6.6 Class template future [futures.unique_future]

3 The effect of calling any member function other than the destructor, the move-assignment operator, or any of the observers `valid`, `is_ready` or `has_value` on a future object for which `valid() == false` is undefined.

```
namespace std {
  template <class R>
  class future {
  public:

    // parameter typedef
    typedef R value_type;
    // Constructor from unexpected_type<exception_ptr>
    future(unexpected_type<exception_ptr>&&) noexcept;

  ...
    // functions to check state
    ...
    bool has_value() const noexcept;

    'see below' value_or('see below');
  ...
    // factories
    template <class S>
      future<result_of<S(value_type)>> map(S&& cont);
    template <class S>
      'see below' bind(S&& cont);
    template <class R>
      future<T> catch_error(R&& rec);
    future<T> fallback_to(T&& v);

  };
}
```

Adding

```
future(unexpected_type<exception_ptr>&&) noexcept;
```

Effects:

The `exception_ptr` contained in passed parameter is moved to the shared state of the constructed future.

```
bool has_value() const noexcept;
```

Returns:

`true` if `*this` is associated with a shared state, that result is ready for retrieval, and the result is a stored value, `false` otherwise.

```
T future<T>::value_or(T&& v) noexcept(see below);
T future<T>::value_or(T const& v) noexcept(see below);
T& future<T&>::value_or(T& v) noexcept;
```

Remark(s):

The expression inside `noexcept` is equivalent to

- `value_or(T&& v)`: `is_nothrow_move_constructible<T>::value`
- `value_or(T const& v)`: `is_nothrow_copy_constructible<T>::value`

Effects:

blocks until the future is ready.

Returns:

the stored value if `has_value()` or `v` otherwise.

Throws:

Any exception throw by the move constructor.

Note(s):

The authors have not found a use case for `void future<void>::value_or();`

```
template<class F>
future<result_type_t<decay<F>(value_type)>> map(F&& func);
template<class Executor, class F>
future<result_type_t<decay<F>(value_type)>> map(Executor &ex, F&& func);
template<class F>
future<result_type_t<decay<F>(value_type)>> map(launch policy, F&& func);
```

Note(s):

The three functions differ only by input parameters. The first only takes a callable object which accepts a `value_type` object as a parameter. The second function takes an executor as the first parameter and a callable object which accepts a `value_type` object as a parameter as the second parameter. The third function takes a launch policy as the first parameter and a callable object which accepts a `value_type` object as a parameter as the second parameter.

Effects:

- The continuation is called when the object's shared state value is ready and has a value.
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- Any value returned from the continuation is stored as the result in the shared state of the resulting `future`. Any exception propagated from the execution of the continuation is stored as the exceptional result in the shared state of the resulting `future`
- If the parent was created with `promise` or with a `packaged_task` (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of `launch::async` |`launch::deferred` and the same argument for `func`.
- If the parent has a policy of `launch::deferred` then the parent is filled by immediately calling `.wait()` or `get()` on the resulting future.

Returns:

Returns an object of preceding type that refers to valid created shared state. When the dependent future is ready by the continuation if the shared state has a value or the future itself if it has an exceptions stored.

Postconstion(s):

- The future object value is moved to the parameter of the continuation function if it was present.
- `valid() == false` on original future object immediately after it returns.
- `valid() == true` for the returned future.

```
template<class F>
'see below' bind(F&& func);
template<class Executor, class F>
'see below' bind(Executor &ex, F&& func);
template<class F>
'see below' bind(launch policy, F&& func);
```

Note(s):

The three functions differ only by input parameters. The first only takes a callable object which accepts a `value_type` object as a parameter. The second function takes an executor as the first parameter and a callable object which accepts a `value_type` object as a parameter as the second parameter. The third function takes a launch policy as the first parameter and a callable object which accepts a `value_type` object as a parameter as the second parameter.

Effects:

- The continuation is called when the object's shared state value is ready and has a value.
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.
- Any value returned from the continuation is stored as the result in the shared state of the resulting `future`. Any exception propagated from the execution of the continuation is stored as the exceptional result in the shared state of the resulting `future`
- If the parent was created with `promise` or with a `packaged_task` (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of `launch::async` | `launch::deferred` and the same argument for `func`.
- If the parent has a policy of `launch::deferred` then the parent is filled by immediately calling `.wait()` or `get()` on the resulting future.

Returns:

The return type depends on the return type of the closure func as defined below:

- When `result_of_t<decay_t<F>(value_type)>` is `future<R2>`, the function returns `future<R2>`.
- Otherwise, the function returns `future<result_of_t<decay_t<F>(value_type)>>`.

Returns an object of preceding type that refers to valid created shared state. When the dependent future is ready by the continuation if the shared state has a value or the future itself if it has an exceptions stored.

Postconstion(s):

- The future object value is moved to the parameter of the continuation function if present.
- `valid() == false` on original future object immediately after it returns.
- `valid() == true` for the returned future.

```
template<class F>
future<T> catch_error(F&& func);
template<class Executor, class F>
future<T> catch_error(Executor &ex, F&& func);
template<class F>
future<T> catch_error(launch policy, F&& func);
```

Effects:

- The continuation is called when the object's shared state is ready and has an exception with an `exception_ptr` containing the stored exception.
- The continuation launches according to the specified launch policy or executor.
- When the executor or launch policy is not provided the continuation inherits the parent's launch policy or executor.

- If the parent was created with `promise` or with a `packaged_task` (has no associated launch policy), the continuation behaves the same as the third overload with a policy argument of `launch::async | launch::deferred` and the same argument for `func`.
- If the parent has a policy of `launch::deferred` and the continuation does not have a specified launch policy or scheduler, then the parent is filled by immediately calling `.wait()`, and the policy of the antecedent is `launch::deferred`

Returns:

An object of type `future<T>` that refers to the shared state created by the continuation if the shared state has an exception or the future itself if it has a value.

Postconstion(s):

- The future object is moved to the parameter of the continuation function
- `valid() == false` on original future object immediately after it returns

```
future<T> future<T>::fallback_to(T v);
```

Returns:

a `future<T>` that would return `v` when the source future has an exception.

Update section

30.6.7 Class template shared_future **[futures.shared_future]**

To be completed once the wording for future<T> is correct.

Add new section

30.6.x Function template `exception_ptr_cast` **[futures.exception_ptr_cast]**

```
\update{template <class R>}
\update{exception_ptr exception_ptr_cast(future<R>&&);}
\update{template <class R>}
\update{exception_ptr exception_ptr_cast(shared_future<R>&&);}
\update{template <class R>}
\update{exception_ptr exception_ptr_cast(shared_future<R> const&);}
```

Effects:

blocks until the future/shared_future is ready.

Returns:

the stored exception on `exception_ptr` if any, otherwise `exception_ptr()`.

Throws:

Nothing

7 Implementability

Boost.Thread [2] provides already the typedef `value_type`, the observers `has_value`, `get_exception_ptr`, `value_or`, `get_or`, and the factories `fallback_to`, `make_unexpected`. Not yet implemented, `bind` and `catch_error`.

8 Acknowledgement

I'm very grateful to Niklas Gustafsson, Artur Laksberg, Herb Suttev, Sana Mithani as this proposal would not exists without their proposal [4].

Thanks to Agustín K-ballo Bergé for its critical comments which have been the motivation of the first revision.

Thanks to Peter Sommerlad for questioning the the intrusiveness of `get_exception_ptr` which have been the motivation of the second revision.

References

[1] N3797 - Working Draft, Standard for Programming Language C++, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[2] Vicente J. Botet Escriba Anthony Williams. Boost.Thread, 2014. http://www.boost.org/doc/libs/1_55_0/doc/html/thread.html.

[3] Vicente J. Botet Escriba. N3865,more improvements to std::future<t>, 2014. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3865.pdf.

[4] H. Sutter S. Mithani N. Gustafsson, A. Laksberg. N3784,improvements to std::future<t> and related apis, 2014. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3784.pdf.

[5] H. Sutter S. Mithani N. Gustafsson, A. Laksberg. N3857,improvements to std::future<t> and related apis, 2014. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3784.pdf.

[6] Pierre Talbot Vicente J. Botet Escriba. N4015 - a proposal to add a utility class to represent expected monad, 2014. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4015.pdf.