Document number: N3865
Date: 2014/01/18
Revises: None
Project: JTC1.22.32 Programming Language C++
Reply to: Vicente J. Botet Escriba
<vicente.botet@wanadoo.fr>

More Improvements to std::future<T>

This paper complements "N3784 Improvements to `std::future<T>` and Related APIs" [3] with more future observers and factories.

Contents

1 Introduction **1**

2 Motivation and Scope **1**

3 Impacts on the Standard **2**

4 Design rationale **2**

5 Proposed Wording **4**

6 Implementability **7**

7 Acknowledgement **7**

1 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" [3] with more future observers and factories mainly based on the split between futures having a value or a exception.

2 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 [3] respect to the function `.then()`, but in this case respect to the functions `.next()` and `.recover()` respectively.

`.get_value()`, `.get_exception_ptr()`:

Been able to know if a future has a value or an exception is not too much useful if we can not retrieve the `exception_ptr` efficiently. Been able to know if a future has a value or an exception allows to get the value with better exception guaranties.

`.value_or(v)`:

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

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

is a shortcut for

```
x = (f.has_value()) ? f.get_value() : v;
```

And the following future factories

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

Quite often 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_future(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_future` a future can be created which hold a precomputed exception on its shared state.

`when_all`:

A new overload of the `when_all()` factory that takes a range of futures as argument. And return a future container of the future values.

3 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.

4 Design rationale

4.1 `.next /recover`

The proposal to include `future.next`/`future.recover` to the standard provides the ability to sequentially compose two futures by declaring one to be the continuation of another. With `.next` the antecedent future has a value before the continuation starts as instructed by the lambda function. With `.recover` 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 `.next` 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
    .next([](int v) {
      return v.to_string();
    })
    .recover([](exception_ptr ex) {
        return "nan";
    });
}
```

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

```
myFuture.next(...).next (...).next (...).recover(...).
```

Some points to note are:

- Each continuation will not begin until the preceding has completed.
- If an exception is thrown, the following continuation can handle it in a try-catch block

Input Parameters:

- Lambda function: The lambda function on `next()` takes a `future<t>::value_type`. The lambda function on `recover()` 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 `.next`/`.recover` 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()`.

4.2 `has_value()`, `get_value()` and `get_exception_ptr()`

The concept of checking if the shared state is 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 is 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(); });

if(!f1.ready()) {
    //if not ready, attach a continuation and avoid a blocking wait
    fl.next([] (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_value());
  }
}
```

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.

4.3 `make_exceptional_future`

This function creates an exceptional `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_exceptional_future<int>(invalid_argument());
  if (x == 0)
    try { do_some_short_work(); }
```

```
    catch (...) make_exceptional_future<int>();

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

5 Proposed Wording

The proposed changes are expressed as edits to N3797, the C++ Draft Standard [1]. The wording has been adapted from N3784 [3].
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);

  // 5,  Exceptional factories
  template <class T>
  future<T> make_exceptional_future(exception_ptr ex);
  template <class T, class E>
  future<T> make_exceptional_future(E ex);


}
```

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;

  ...
    // functions to check state
    ...
    bool has_value() const noexcept;
    see below get_value() const;
    exception_ptr get_exception_ptr() const;
    see below value_or(see below);
  ...
    // factories
    template <class S>
      future<T> next(S&& cont);
```

```
    template <class R>
      future<T> recover(R&& rec);
    future<T> fallback_to(T&& v);


  };
}
```

Adding

```
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.

```
exception_ptr get_exception_ptr() const;
```

Requires:

`this->has_value() == false`;.

Returns:

the stored `exception_ptr`

Throws:

Nothing

```
T get_value() const;
```

Requires:

`this->has_value() == true`;.

Returns:

the stored value.

Throws:

Nothing

```
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`

Returns:

blocks until the future is ready and returns the stored value if `has_value()` or `v` otherwise.

Note(s):

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

```
template<class F>
auto next(F&& func) -> future<decltype(func(T&))>;
template<class Executor, class F>
auto next(Executor &ex, F&& func) -> future<decltype(func(T&))>;
template<class F>
auto next(launch policy, F&& func) -> future<decltype(func(T&))>;
```

Effects:

- The continuation is called when the object's shared state is ready and has a value with the stored 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.
- 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<decltype(func(T&))>` that refers to the shared state created 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 is moved to the parameter of the continuation function
- `valid() == false` on original future object immediately after it returns

```
template<class F>
future<T> recover(F&& func);
template<class Executor, class F>
future<T> recover(Executor &ex, F&& func);
template<class F>
future<T> recover(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` storing the 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 make_exceptional_future [futures.make_exceptional_future]

```
template <class T>
  future<T> make_exceptional_future(exception_ptr ex);
template <class T, class E>
  future<T> make_exceptional_future(E ex);
template <class T>
  future<T> make_exceptional_future();
```

Effects:

The exception that is passed in to the function is moved to the shared state of the returned future if it is an rvalue. Otherwise the exception is copied to the shared state of the returned future.

Returns:

a `future<T>`

Postconstion(s):

- Returned `future<T>, valid() == true`
- Returned `future<T>`, `is_ready() = true`
- Returned `future<T>`, `has_value() = false`

6 Implementability

Boost.Thread [2] provides already the typedef `value_type`, the observers `has_value`, `get_value`, `get_exception_ptr`, `value_or`, and the factories `fallback_to`, `make_exceptional_future`. Not yet implemented, `next` and `recover`.

7 Acknowledgement

I'm very grateful to Niklas Gustafsson, Artur Laksberg, Herb Suttev, Sana Mithani as this proposal would not exists without their proposal [3].

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, 2013. http://www.boost.org/doc/libs/1_55_0/doc/html/thread.html.

[3] Vicente J. Botet Escriba. N3784,improvements to std::future<t> and related apis, 2013. http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3784.pdf.