Document number: N3857
Supersedes: N3784
Date: 2014-01-16
Reply-to: Niklas Gustafsson < niklas.gustafsson@microsoft.com>
Artur Laksberg < arturl@microsoft.com>
Herb Sutter < hsutter@microsoft.com>
Sana Mithani <sanam@microsoft.com>

Improvements to std::future<T> and Related APIs

I. Table of Contents

II. Introduction

This proposal is an evolution of the functionality of **`std::future/std::shared_future`**. It details additions which can enable wait free compositions of asynchronous operations.

This document supersedes N3784. Several typos in the code samples have been fixed, and a small editorial change made in the Technical Specification section..

III. Motivation and Scope

There has been a recent increase in the prevalence of I/O heavy applications which are coupled with compute heavy operations. As the industry trends towards connected and multicore programs, the importance of managing the latency and unpredictability of I/O operations becomes ever more significant. This has mandated the necessity of responsive asynchronous operations. Concurrency is about both decomposing and composing the program from the parts that work well individually and together. It is in the composition of connected and multicore components where today's C++ libraries are still lacking.

The functionality of **`std::future`** offers a partial solution. It allows for the separation of the initiation of an operation and the act of waiting for its result; however the act of waiting is synchronous. In communication-intensive code this act of waiting can be unpredictable, inefficient and simply frustrating. The example below illustrates a possible synchronous wait using futures.

```
#include <future>
using namespace std;
int main() {
    future<int> f = async([]() { return 123; });

    int result = f.get(); // might block
}
```

C++ suffers an evident deficit of asynchronous operations compared to other languages, thereby hindering programmer productivity. JavaScript on Windows 8 has promises (then, join and any), .NET has the Task Parallel Library (ContinueWith, WhenAll, WhenAny), C#/VB has the await keyword (asynchronous continuations), and F# has asynchronous workflows. When compared to these languages, C++ is less productive for writing I/O-intensive applications and system software. In particular writing highly scalable services becomes significantly more difficult.

This proposal introduces the following key asynchronous operations to **`std::future`**, **`std::shared_future`**, and **`std::async`**, which will enhance and enrich these libraries.

then:
In asynchronous programming, it is very common for one asynchronous operation, on completion, to invoke a second operation and pass data to it. The current C++ standard does not allow one to register a continuation to a future. With **`then`**, instead of waiting for the result, a continuation is "attached" to the asynchronous operation, which is invoked when the result is ready. Continuations registered using the **`then`** function will help to avoid blocking waits or wasting threads on polling, greatly improving the responsiveness and scalability of an application.

unwrap:

In some scenarios, you might want to create a **`future`** that returns another **`future`**, resulting in nested futures. Although it is possible to write code to unwrap the outer future and retrieve the nested future and its result, such code is not easy to write because you must handle exceptions and it may cause a blocking call. **`unwrap`** can allow us to mitigate this problem by doing an asynchronous call to unwrap the outermost future.

is_ready:

There are often situations where a **`get()`** call on a future may not be a blocking call, or is only a blocking call under certain circumstances. This function gives the ability to test for early completion and allows us to avoid associating a continuation, which needs to be scheduled with some non-trivial overhead and near-certain loss of cache efficiency.

when_any / when_any_swapped / when_all:

The standard also omits the ability to compose multiple futures. This is a common pattern that is ubiquitous in other asynchronous frameworks and is absolutely necessary in order to make C++ a powerful asynchronous programming language. Not including these functions is synonymous to Boolean algebra without AND/OR. **`when_any`** and **`when_any_swapped`** asynchronously wait for at least one of multiple **`future`** or **`shared_future`** objects to finish. **`when_all`** asynchronously waits for multiple **`future`** and **`shared_future`** objects to finish.

make_ready_future

Some functions may know the value at the point of construction. In these cases the value is immediately available, but needs to be returned as a **`future`**. By using **`make_ready_future`** a **`future`** can be created which holds a pre-computed result in its shared state. In the current standard it is non-trivial to create a future directly from a value. First a **`promise`** must be created, then the **`promise`** is set, and lastly the **`future`** is retrieved from the **`promise`**. This can now be done with one operation.

Target Audience

- Programmers wanting to write I/O and compute heavy applications in C++
- Programmers who demand server side scalability, client side responsiveness, and non-blocking UI threads from applications.
- Programmers who rely on multiple asynchronous operations, each with its own completion event

IV. 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++ 11. 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.

V. Design Decisions

Overview

The proposal introduces new features to the C++ standard as a library based proposal. Many of the design decisions were based primarily on Microsoft's successful Parallel Programming Libraries (PPL). PPL is widely adopted throughout the organization and has become the default model for asynchrony. Furthermore, the library based approach creates a basis for the natural evolution of future language based changes.

then

The proposal to include **`future::then`** to the standard provides the ability to sequentially compose two futures by declaring one to be the continuation of another. With **`then`** the antecedent **`future`** is ready (has a value or exception stored in the shared state) 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 **`then`** member function. This operation takes a lambda function which describes how **`f2`** should proceed after **`f1`** is ready.

```
#include <future>
using namespace std;
int main() {
   future<int> f1 = async([]() { return 123; });

   future<string> f2 = f1.then([](future<int> f) {
        return to_string(f.get()); // here .get() won't block
   });
}
```

One key feature of this function is the ability to chain multiple asynchronous operations. In asynchronous programming, it's common to define a sequence of operations, in which each continuation executes only when the previous one completes. In some cases, the *antecedent* future produces a value that the continuation

accepts as input. By using **`future.then`**, creating a chain of continuations becomes straightforward and intuitive: **`myFuture.then(…).then(…).then(…).`** 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: One option which was considered was to follow JavaScript's approach and take two functions, one for success and one for error handling. However this option is not viable in C++ as there is no single base type for exceptions as there is in JavaScript. The lambda function takes a **`future`** as its input which carries the exception through. This makes propagating exceptions straightforward. This approach also simplifies the chaining of continuations.

- Executor: An overloaded version on **`then`** takes a reference to an **`executor`** object as an additional parameter. The details of the executor are described in document N3562, and won't be repeated here. It is recommended to read N3562 to get a better understanding of what an executor is and motivation behind standardizing it. This variant of **`then`** is useful in cases where there is a desire to control which threads are being executed. Some circumstances where the executor is necessary include:
  - The continuation of a future requires significant time to complete and therefore cannot execute in the context of the completing thread (inline). Association of an **`executor`** with the future causes the future's continuation to be scheduled using the **`executor`**.
  - A continuation needs to execute on the thread owning a particular hardware or software resource, such as the main graphical user interface thread.
  - The application needs to throttle its work into a thread pool of a limited size.

- Launch policy: if the flexibility that the executor provides is not required.

Return values: The decision to return a **`future`** was based primarily on the ability to chain multiple continuations using **`then`**. This benefit of composability gives the programmer incredible control and flexibility over their code. Returning a **`future`** object rather than a **`shared_future`** is also a much cheaper operation thereby improving performance. A **`shared_future`** object is not necessary to take advantage of the chaining feature. It is also easy to go from a **`future`** to a **`shared_future`** when needed using **`future::share().`**

It is a common situation that the body of a then function object will itself be a future-based operation, which leads to the then() returning a future<future<T>>. In this case, it is almost always the case that what you really care about is the inner future, so then() performs an implicit unwrap() (see below) before returning. Only one level of future nesting is unwrapped.

unwrap

Calling **unwrap()** on a **future<future>** returns a proxy to the inner **future**. Often unwrapping is required before attaching a continuation using **then**. In the example below the **outer_future** is of type **future<future<int>>.** Calling unwrap() is unlike get() in that it does not wait for the outer future to be ready before returning. Instead, it returns a proxy for the inner future.

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

int main() {

    future<future<int>> outer_future = async([]{
        future<int> inner_future =  async([] {
                return 1;
        });
        return inner_future;
    });

    future<int> inner_future = outer_future.unwrap();

     inner_future.then([](future<int> f) {
        do_work(f);
     });
}
```

Explicit unwrapping: During the design phase the option of doing automatic unwrapping was considered. Microsoft's PPL task does automatic asynchronous unwrapping in several operations when returning a nested future. Based on experience with PPL, and feedback at meetings in Portland and Bristol, it was decided that explicit unwrapping will be available via the unwrap() function.

Implicit unwrapping: Automatically unwrapping nested futures, when the context calls for it, is also very valuable as it makes a lot of code simpler and thus easier to read.

Implicit unwrapping shall be implemented by the implementation of then(), which will automatically unwrap one level only. In other words, any then() for which the resulting type would be future<future<T>> without unwrapping will return future<T> as if .unwrap() were called on the result. Automatic unwrapping shall also be implemented throug construction, so that the constructor for future<T> will accept values of type future<future<T>> as if unwrap() were called on the value.

is_ready

The concept of checking if the shared state is ready already exists in the standard today. For example, calling **get()** on a function internally checks if the shared state is ready, and if it isn't it **wait()s**. This function exposes this ability to check the shared state to the programmer, and allows them to bypass the act of waiting by

attaching a continuation if necessary. The example below illustrates using the **is_ready** member function to determine whether using **then** to attach a continuation is needed.

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

int main() {

   future<int> f1 = async([]() { return possibly_long_computation(); });

   if(!f1.is_ready()) {
      //if not ready, attach a continuation and avoid a blocking wait
      f1.then([] (future<int> f2) {
            int v = f2.get();
            process_value(v);
      });
   }
   //if ready, then no need to add continuation, process value right away
   else {
      int v = f1.get();
      process_value(v);
   }
}
```

The decision to add this function as a member of the **future** and **shared_future** classes was straightforward, as this concept already implicitly exists in the standard. This functionality can also explicitly be called by using **f1.wait_for(chrono::seconds(0))**. However **is_ready** is less verbose and much easier to discover by the programmer. By explicitly allowing the programmer to check the shared state of a **future**, improvements on performance can be made. Below are some examples of when this functionality becomes especially useful:

- A library may be buffering I/O requests so that only the initial request takes a significant amount of time (can cause a blocking wait), while subsequent requests find that data already available.
- A function which produces a constant value 90% of the time, but has to perform a long running computation 10% of the time.
- A virtual function that in some derived implementations may require long-running computations, but on some implementations never block.

when_any

The choice operation is implemented by **when_any**. This operation produces a **future** object that completes after one of multiple input futures complete. The **future** that is returned holds a **vector** or **tuple** object with the input futures as elements, in the same order. There are two variations of this function which differ by their input parameters. The first, producing a **vector**, takes a pair of iterators, and the second, producing a **tuple**, takes any arbitrary number of **future** and **shared_future** objects.

when_any is useful in scenarios where there are redundant or speculative executions; you launch several tasks and the first one to complete delivers the required result. You could also add a timeout to an operation—start with an operation that returns a task and combine it with a task that sleeps for a given amount of time. If the sleeping task completes first, your operation has timed out and can therefore be discarded or canceled. Another useful scenario is a parallel search. As soon as the value being searched for is found **when_any** returns the **vector/tuple** with the input futures. The collection can then be parsed looking at the state of each **future** using the **ready()** operator. The first future with a ready state contains the value that was being searched for.

An issue with this design is that figuring out which future is ready requires a linear scan of the output collection. In the case of the iterator-based input, the size of the output vector can be fairly large. In order to mitigate this cost in the common scenario where the calling code only cares about getting the result of the first ready future, a variant of **when_any**, named **when_any_swapped**, swaps the first ready future (the one that triggered completion of the composed future) with the last future of the vector. Thus, the ready future is available for being popped off the vector. This version of the functionality will not maintain the order-based correlation between input and output, and is therefore not the default behavior of **when_any**.

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

int main() {

   future<int> futures[] = {async([]() { return intResult(125); }),
                            async([]() { return intResult(456); })};


   future<vector<future<int>>> any_f = when_any(begin(futures), end(futures));

   future<int> result = any_f.then([](future<vector<future<int>> f) {
          for(future<int> i : f.get()) {
             if(i.is_ready())
                    return i.get(); //get() will not block
           }

   });
}
```

Alternatives: The only other alternative that was considered was to not include this function and to let the user build their own version using promises. However, it is our assessment that this operator is so essential that without it this proposal would be incomplete and C++ would not have a comprehensive set of asynchronous composition primitives.

Input Parameters: There are two variations in this implementation of **when_any**. The first being a function which takes a pair of iterators and the second variation takes any number (except zero) of **future** and **shared_future** objects. The reason to have two versions was to provide convenience and flexibility to the programmer. It is

often the case when there is a collection of futures which has an unknown size that needs to be operated on. By using iterators the number of operands does not need to be known statically. The second variant provides additional convenience by allowing mixing of futures and shared futures of different types. **when_any** also accepts zero arguments and returns **future<tuple<>>**.

Return values: The function always returns a **future** object; the type of the **future** is dependent on the inputs.

- **future<vector<future<R>>>**: If the input cardinality is unknown at compile time and the futures are all of the same type, and the iterator pair yields **future<R>**. **R** may be void.
- **future<vector<shared_future<R>>>**: If the input cardinality is unknown at compile time and the futures are all of the same type, and the iterator pair yields **shared_future<R>**. **R** may be void.
- **future<tuple<future<R0>, future<R1>, future<R2>…>>**: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of **future** and **shared_future** objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of **R0**, **R1**, **R2**, etc. may be void.

when_all

The join operator is implemented by **when_all**. This operation asynchronously waits for all of multiple **future** and **shared_future** objects to finish. The **future** that is returned holds a **tuple** or a **vector** with copies of all of the input futures. Like **when_any** there are also two variations. The first taking an iterator pair and the second taking a series of **future** or **shared_future** objects as shown below.

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

int main() {

    shared_future<int> shared_future1 = async([] { return intResult(125); });
    future<string> future2 = async([]() { return stringResult("hi"); });

    future<tuple<shared_future<int>, future<string>>> all_f =
            when_all(shared_future1, future2);

    future<int> result = all_f.then([](future<tuple<shared_future<int>,
    future<string>>> f) {
            return doWork(f.get());
    });
}
```

Exception handling: The resulting **future** will not throw an exception when its value is retrieved. Any exceptions arising from the operations represented by the input futures are raised by the individual futures in the output collection.

Input Parameters: Again, as the case with **when_any**, there are two variations. The first which takes an iterator pair, and the second which takes a sequence of **future** and **shared_future** objects. One key difference with this

operation is that the **future** and **shared_future** objects do not have to be of the same type. **when_all** also accepts zero arguments and returns **future<tuple<>>**.

Return values: The function always returns a **future** object, however the type of the **future** is dependent on the inputs.

- **future<vector<future<R>>>**: If the input cardinality is unknown at compile time and the futures are all of the same type, and the iterator pair yields future<R>. **R** may be void.
- **future<vector<shared_future<R>>>**: If the input cardinality is unknown at compile time and the futures are all of the same type, and the iterator pair yields shared_future<R>. **R** may be void.
- **future<tuple<future<R0>, future<R1>, future<R2>…>>**: If inputs are fixed in number and are of heterogeneous types. The inputs can be any arbitrary number of **future** and **shared_future** objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of **R0**, **R1**, **R2**, etc. may be void.

make_ready_future

This function creates a ready future<T> for a given value of type T. If no value is given then a **future<void>** is returned. This function is primarily useful in cases where sometimes, the return value 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 paths the function must return an eventual value represented as a **future**.

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

       if (x < 0) return make_ready_future<int>(-1);
       if (x == 0) return make_ready_future<int>(0);

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

There are two variants of this function. The first takes a value of any type, and returns a **future** of that type. The input value is passed to the shared state of the returned **future**. The second version takes no input and returns a **future<void>.**

If the programmer wants to create a ready **shared_future**, they must first use **make_ready_future** to create a future, then call share on that **future** to get a **shared_future**.

VI. Technical Specification

This proposal presupposes at least partial approval of the proposal contained in document N3562, specifically, the section identified as “III.1.1 Class executor [executors.base.executor]” in that document.

30.6.6 Class template **future** [futures.unique_future]

To the class declaration found in 30.6.6/3, add the following to the public functions:

```
bool ready() const;

future(future<future<R>>&& rhs) noexcept;

template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;

see below unwrap();
```

Between 30.6.6/8 & 30.6.6/9, add the following:

```
future(future<future<R>>&& rhs) noexcept;
```

Effects: constructs a future object by moving the instance referred to by rhs and unwrapping the inner future (see unwrap()).

Postconditions:

- valid() returns the same value as rhs.valid() prior to the constructor invocation.
- rhs.valid() == false.

After 30.6.6/24, add the following:

```
template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

Notes: The three functions differ only by input parameters. The first only takes a callable object which accepts a **future** object as a parameter. The second function takes an **executor** as the first parameter and a callable object as the second parameter. The third function takes a launch policy as the first parameter and a callable object as the second parameter.
In cases where 'decltype(func(*this))' is future<R>, the resulting type is future<R> instead of future<future<R>>.

Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- 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 **std::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(*this))>** that refers to the shared state created by the continuation.

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

```
template<typename R2>
future<R2> future<R>::unwrap()
```

Notes:

- **R** is a **future<R2>** or **shared_future<R2>**
- Removes the outer-most future and returns a proxy to the inner future. The proxy is a representation of the inner future and it holds the same value (or exception) as the inner future.

Effects:
- **future<R2> X = future<future<R2>>.unwrap()**, returns a **future<R2>** that becomes ready when the shared state of the inner future is ready. When the inner future is ready, its value (or exception) is *moved* to the shared state of the returned **future**.
- **future<R2> Y = future<shared_future<R2>>.unwrap(),** returns a **future<R2>** that becomes ready when the shared state of the inner future is ready. When the inner **shared_future** is ready, its value (or exception) is *copied* to the shared state of the returned **future**.
- If the outer **f**uture throws an exception, and **.get()** is called on the returned **future**, the returned **future** throws the same exception as the outer future. This is the case because the inner future didn't exit

Returns: a **future** of type **R2**. The result of the inner **future** is moved out (**shared_future** is copied out) and stored in the shared state of the returned future when it is ready or the result of the inner future throws an exception.

Postcondition:
- The returned **future** has **valid() == true**, regardless of the validity of the inner **future**.

[*Example:*
```
future<int> work1(int value);
int work(int value) {
	future<future<int>> f1 = std::async([=] {return work1(value); });
```

```
        future<int> f2 = f1.unwrap();
        return f2.get();
}
```
-end example]

```
bool is_ready() const;
```

Returns: **true** if the shared state is ready, **false** if it isn't.

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

To the class declaration found in 30.6.7/3, add the following to the public functions:

```
bool is_ready() const;

template<typename F>
auto then(F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(executor &ex, F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto then(launch policy, F&& func) -> future<decltype(func(*this))>;

see below unwrap();
```

After 30.6.7/26, add the following:

```
template<typename F>
auto shared_future::then(F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto shared_future::then(executor &ex, F&& func) -> future<decltype(func(*this))>;
template<typename F>
auto shared_future::then(launch policy, F&& func) -> future<decltype(func(*this))>;
```

Notes: The three functions differ only by input parameters. The first only takes a callable object which accepts a **shared_future** object as a parameter. The second function takes an **executor** as the first parameter and a callable object as the second parameter. The third function takes a launch policy as the first parameter and a callable object as the second parameter.
In cases where 'decltype(func(*this))' is future<R>, the resulting type is future<R> instead of future<future<R>>.

Effects:

- The continuation is called when the object's shared state is ready (has a value or exception stored).
- The continuation launches according to the specified policy or **executor**.
- When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or **executor**.

- If the parent was created with **std::promise** (has no associated launch policy), the continuation behaves the same as the third function 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(*this))>** that refers to the shared state created by the continuation.

Postcondition:
- The **shared_future** passed to the continuation function is a copy of the original **shared_future**
- **valid() == true** on the original **shared_future** object

```
template<typename R2>
future<R2> shared_future<R>::unwrap();
```

Requires: **R** is a **future<R2>** or **shared_future<R2>**

Notes: Removes the outer-most shared_future and returns a proxy to the inner future. The proxy is a representation of the inner future and it holds the same value (or exception) as the inner future.

Effects:
- **future<R2> X = shared_future<future<R2>>.unwrap()**, returns a **future<R2>** that becomes ready when the shared state of the inner future is ready. When the inner future is ready, its value (or exception) is *moved* to the shared state of the returned **future**.
- **future<R2> Y = shared_future<shared_future<R2>>.unwrap()**, returns a **future<R2>** that becomes ready when the shared state of the inner future is ready. When the inner **shared_future** is ready, its value (or exception) is *copied* to the shared state of the returned **future**.
- If the outer future throws an exception, and **.get()** is called on the returned **future**, the returned **future** throws the same exception as the outer future. This is the case because the inner future didn't exit

Returns: a **future** of type **R2**. The result of the inner **future** is moved out (**shared_future** is copied out) and stored in the shared state of the returned future when it is ready or the result of the inner future throws an exception.

Postcondition:
- The returned **future** has **valid() == true**, regardless of the validity of the inner **future**.

```
bool is_ready() const;
```

Returns: **true** if the shared state is ready, **false** if it isn't.

30.6.X Function template **when_all** [futures.when_all]

```
template <class InputIterator>
see below when_all(InputIterator first, InputIterator last);

template <typename... T>
see below when_all(T&&... futures);
```

Requires: **T** is of type **future<R>** or **shared_future<R>**.

Notes:

- There are two variations of **when_all**. The first version takes a pair of **InputIterators**. The second takes any arbitrary number of **future<R0>** and **shared_future<R1>** objects, where **R0** and **R1** need not be the same type.
- Calling the first signature of **when_all** where **InputIterator** index first equals index last, returns a **future** with an empty vector that is immediately ready.
- Calling the second signature of **when_any** with no arguments returns a **future<tuple<>>** that is immediately ready.

Effects:

- Each **future** and **shared_future** is waited upon and then copied into the collection of the output (returned) future, maintaining the order of the futures in the input collection.
- The future returned by **when_all** will not throw an exception, but the futures held in the output collection may.

Returns:

- **future<tuple<>>**: if **when_all** is called with zero arguments.
- **future<vector<future<R>>>**: If the input cardinality is unknown at compile and the iterator pair yields **future<R>**. **R** may be void. The order of the futures in the output vector will be the same as given by the input iterator.
- **future<vector<shared_future<R>>>**: If the input cardinality is unknown at compile time and the iterator pair yields **shared_future<R>. R** may be void. The order of the futures in the output vector will be the same as given by the input iterator.
- **future<tuple<future<R0>, future<R1>, future<R2>…>>**: If inputs are fixed in number. The inputs can be any arbitrary number of **future** and **shared_future** objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of **R0**, **R1**, **R2**, etc. may be void.

Postcondition:

- All input future<T>s **valid() == false**
- All output shared_future<T> **valid() == true**

30.6.X Function template **when_any** [futures.when_any]

```
template <class InputIterator>
see below when_any(InputIterator first, InputIterator last);

template <typename... T>
see below when_any(T&&... futures);
```

Requires: **T** is of type **future<R>** or **shared_future<R>**. All **R** types must be the same.

Notes:

- There are two variations of **when_any**. The first version takes a pair of **InputIterators**. The second takes any arbitrary number of **future<R>** and **shared_future<R>** objects, where **R** need not be the same type.
- Calling the first signature of **when_any** where **InputIterator** index first equals index last, returns a **future** with an empty vector that is immediately ready.
- Calling the second signature of **when_any** with no arguments returns a **future<tuple<>>** that is immediately ready.

Effects:

- Each **future** and **shared_future** is waited upon. When at least one is ready, all the futures are copied into the collection of the output (returned) **future**, maintaining the order of the futures in the input collection.
- The **future** returned by **when_any** will not throw an exception, but the futures held in the output collection may.

Returns:

- **future<tuple<>>**: if **when_any** is called with zero arguments.
- **future<vector<future<R>>>**: If the input cardinality is unknown at compile time and the iterator pair yields future<R>. **R** may be void. The order of the futures in the output vector will be the same as given by the input iterator.
- **future<vector<shared_future<R>>>**: If the input cardinality is unknown at compile time and the iterator pair yields **shared_future<R>. R** may be void. The order of the futures in the output vector will be the same as given by the input iterator.
- **future<tuple<future<R0>, future<R1>, future<R2>…>>**: If inputs are fixed in number. The inputs can be any arbitrary number of **future** and **shared_future** objects. The type of the element at each position of the tuple corresponds to the type of the argument at the same position. Any of **R0**, **R1**, **R2**, etc. maybe void.

Postcondition:

- All input future<T>s **valid() == false**
- All input shared_future<T> **valid() == true**

30.6.X Function template **when_any_swapped** [futures. when_any_ swapped]

```
template <class InputIterator>
see below when_any_swapped(InputIterator first, InputIterator last);
```

Requires: **InputIterator**'s value type shall be convertible to **future<R>** or **shared_future<R>**. All **R** types must be the same.

Notes:

- The function **when_any_swapped** takes a pair of **InputIterators**.
- Calling **when_any_swapped** where **InputIterator** index first equals index last, returns a **future** with an empty vector that is immediately ready.

Effects:

- Each **future** and **shared_future** is waited upon. When at least one is ready, all the futures are copied into the collection of the output (returned) **future**.
- After the copy, the **future** or **shared_future** that was first detected as being ready swaps its position with that of the last element of the result collection, so that the ready **future** or **shared_future** may be identified in constant time. Only one **future** or **shared_future** is thus moved.
- The **future** returned by **when_any_swapped** will not throw an exception, but the futures held in the output collection may.

Returns:

- **future<vector<future<R>>>**: If the input cardinality is unknown at compile time and the iterator pair yields future<R>. **R** may be void.
- **future<vector<shared_future<R>>>**: If the input cardinality is unknown at compile time and the iterator pair yields **shared_future<R>. R** may be void.

Postcondition:

- All input future<T>s **valid() == false**
- All input shared_future<T> **valid() == true**

30.6.X Function template **make_ready_future** [futures.make_future]

```
template <typename T>
future<typename decay<T>::type> make_ready_future(T&& value);
future<void> make_ready_future();
```

Effects: The value that is passed in to the function is *moved* to the shared state of the returned function if it is an rvalue. Otherwise the value is *copied* to the shared state of the returned function.

Returns:

- future<T>, if function is given a value of type T
- future<void>, if the function is not given any inputs.

Postcondition:

- Returned future<T>, **valid() == true**

- Returned future<T>, **is_ready() = true**

30.6.8 Function template **async** [futures.async]

Change 30.6.8/1 as follows:

> The function template async provides a mechanism to launch a function potentially in a new thread and provides the result of the function in a future object with which it shares a shared state.
>
> ```
> template <class F, class... Args>
> future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
> async(F&& f, Args&&... args);
> template <class F, class... Args>
> future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
> async(launch policy, F&& f, Args&&... args);
> template<class F, class... Args>
> future<typename result_of<typename decay<F>::type(typename decay<Args>::type...)>::type>
> async(executor &ex, F&& f, Args&&... args);
> ```

Change 30.6.8/3 as follows:

> *Effects:* The first function behaves the same as a call to the second function with a policy argument of launch::async | launch::deferred and the same arguments for F and Args. The second and third functions create~~s~~ a shared state that is associated with the returned future object. The further behavior of the second function depends on the policy argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):
>
> - if policy & launch::async is non-zero — calls *INVOKE* (*DECAY_COPY* (std::forward<F>(f)), *DECAY_COPY* (std::forward<Args>(args))...) (20.8.2, 30.3.1.2) as if in a new thread of execution represented by a thread object with the calls to *DECAY_COPY* () being evaluated in the thread that called async. Any return value is stored as the result in the shared state. Any exception propagated from the execution of *INVOKE* (*DECAY_COPY* (std::forward<F>(f)), *DECAY_COPY* (std::forward<Args>(args))...) is stored as the exceptional result in the shared state. The thread object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.
>
> - if policy & launch::deferred is non-zero — Stores DECAY_COPY (std::forward<F>(f)) and DECAY_COPY (std::forward<Args>(args))... in the shared state. These copies of f and args constitute a deferred function. Invocation of the deferred function evaluates INVOKE std::move(g), std::move(xyz)) where g is the stored value of DECAY_COPY (std::forward<F>(f)) and xyz is the stored copy of DECAY_COPY (std::forward<Args>(args))... The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (30.6.4) on an asynchronous return object referring to this shared state shall invoke the deferred function in the thread that called the waiting function. Once evaluation of INVOKE (std::move(g), std::move(xyz)) begins, the function is no longer considered deferred. [Note: If this policy isspecified together with other policies, such as when using a policy value

of launch::async | launch::deferred, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited. —*end note*]

The further behavior of the third function is as follows:

The **executor::add() function** is given a **function<void ()>** which calls **INVOKE (DECAY_COPY (std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)**. The implementation of the executor is decided by the programmer.

Change 30.6.8/8 as follows:

Remarks: The first signature shall not participate in overload resolution if decay<F>::type is std::launch or std::executor.

VII. Acknowledgements

- Niklas Gustafsson
- Artur Laksberg
- Herb Sutter
- Sana Mithani
- Krishnan Varadarajan
- Shuai Wang
- Stephan T. Lavavej
- Rahul V. Patil
- Jeffrey Yasskin
- Matt Austern

VIII. References

Josuttis, N. M. (2012). *The C++ standard library: a tutorial and reference.*

Laksberg, A. (2012, February). *Asynchronous Programming in C++ Using PPL*. Retrieved from MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/hh781020.aspx

Laksberg, A. (2012, July). *Improving Futures and Callbacks in C++ To Avoid Synching by Waiting*. Retrieved from Dr.Dobb's: http://www.drdobbs.com/parallel/improving-futures-and-callbacks-in-c-to/240004255

Microsoft. (2011, November). *Asynchronous Programming for C++ Developers: PPL Tasks and Windows 8*. Retrieved from Channel9: http://channel9.msdn.com/Blogs/Charles/Asynchronous-Programming-for-C-Developers-PPL-Tasks-and-Windows-8

Microsoft. (2011, March). *Tasks and Continuations: Available for Download Today!* Retrieved from MSDN Blogs: http://blogs.msdn.com/b/nativeconcurrency/archive/2011/03/09/tasks-and-continuations-available-for-download-today.aspx

Microsoft. (n.d.). *Asynchronous Workflows (F#)*. Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/dd233250.aspx