Document number: *N3428=12-0118.*
Date: *2012-09-21*
Reply-to: Niklas Gustafsson < niklas.gustafsson@microsoft.com>
Artur Laksberg < arturl@microsoft.com>
Herb Sutter < hsutter@microsoft.com>
Sana Mithani <sanam@microsoft.com>

A Standardized Representation of Asynchronous Operations

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.

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:[1]

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. **`future.unwrap`** can allow us to mitigate this problem by doing an asynchronous call to unwrap the outermost future.

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_all:

The standard also omits the ability to compose multiple futures. This is a common pattern that is ubiquitous in other asynchronous languages and is absolutely necessary in order to make C++ a powerful asynchronous language. Not including these functions is synonymous to Boolean operations missing AND and OR. **`when_any`** asynchronously waits for one of multiple **`future`** or **`shared_future`** objects to finish. **`when_all`** asynchronously waits for multiple **`future`** and **`shared_future`** objects to finish.

make_future / make_shared_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`** or **`shared_future`**. By using **`make_future`** **`(make_shared_future)`** a **`future`** (**`shared_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.

[1] One suggestion is for the constructor of future to implicitly collapse future<future<T>> to future<T>, such that the case of nested futures is not even possible. This suggestion eliminates the need to explicitly unwrap nested futures. We have yet to come across any use cases where nested futures are needed, but before making such a proposal we want to ensure to not break any existing code.

scheduler:

Under some circumstances more precise control over resources, whether hardware or software is required. Unfortunately, `std::thread` does not provide the abstractions necessary for such control, and the launch policy in `std::async` is not a sufficient abstraction of scheduling. To mitigate this issue an overloaded `.then` function is being proposed which takes a `scheduler` reference as an additional parameter. The `scheduler` will decide how to allocate machine resources to the work associated with an event. Scheduling may happen in the operating system, in supporting libraries, or (generally) both. The capabilities of the `scheduler` surpass those of a launch policy and place full control of how futures are executed in the hands of the programmer.

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

Implementation

There is a current reference implementation which redesigns `std:futures` to include the above functionality. A test suite associated with this implementation is also available.

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.

scheduler

The decision to implement the Scheduler as virtual functions was discussed at the standards meeting in May, and was the basis for our implementation. We are aware that this approach is not conventional and potentially controversial, and therefore we are open to other options.

This **`scheduler`** is an interface class which is implemented by the programmer. The decision to propose the scheduler class as an interface is based on the ability to give the user full and complete control over its implementation. The user can specify the execution of the future object based on what is most appropriate for that context. In order to maintain consistency extending the functionality of the **`scheduler`** to **`std::async`** is necessary.

Some circumstances where this **`scheduler`** 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 a scheduler with the future causes the future's continuation to be scheduled using the scheduler
- 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.

To extend this flexibility of the scheduler an overload to **`std::async`**, is also being proposed. This overload will take a reference of a **`scheduler`** object (instead of a launch policy) which is implemented by the programmer. The following example shows the implementation of a scheduler. The scheduler is passed to **`std:async`**, which in turn directs the future where to run.

```
class threadpool_scheduler : public std::scheduler
{
public:
      virtual void schedule(const std::function<void()>& work_item) override {
             mythreadpool_submit_work(work_item);
       }
};


void demoScheduler()
{
        threadpool_scheduler scheduler;
        std::future<int> f = std::async(scheduler, []() { return 123; });
        std::cout << f.get() << std::endl;
}
```

.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 f.get().to_string(); // 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[2]: 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.
- Scheduler: Providing an overload to **.then**, to take a **scheduler** reference places great flexibility over the execution of the future in the programmer’s hand. As described in the

[2] One implementation which was suggested is to require the lambda function to return a future<T>. Now .then doesn’t need to wrap the return value in a future. This implementation will remove the need to call unwrap

previous section, often taking a launch policy is not sufficient for powerful asynchronous operations. The lifetime of the scheduler must outlive the continuation.

- Launch policy: if the additional flexibility that the scheduler 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().`**

Naming: Alternatives which were considered include **`continue_with`** (TPL) and **`and_then`**. However JavaScript and PPL use **`.then`**, which is more concise and intuitive and therefore was chosen.

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>>.`**

```
#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 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. The decision to not follow this model was based on a few key points. First, if automatic unwrapping is introduced for **`future.then`**, then for consistency purposes **`std::async`** must also be changed for automatic unwraps. This however is not desirable as it would be a breaking change. This approach is also easier for the programmer to understand conceptually and is easy to discover.

Therefore based on the precedent set by C#'s Task Parallel Library, **`future.unwrap()`** is introduced to perform explicit unwrapping.

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 **`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.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 **`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 futures complete. The **future** that is returned holds a **pair** object with the first element being the index of the completed future and the second element, its value. There are two variations of this function which differ by their input parameters. The first takes a pair of iterators, and the second takes any arbitrary number of **future** and **shared_future** objects. **when_any** is handy 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 a future containing the value and its index in the collection is returned.

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

int main() {

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


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

   future<int> result = any_f.then([](future<pair<size_t,int>> f) {
          return doWork(f.get());
   });
}
```

Naming: “choice” is a common name for this function. However in order to be as clear as possible and follows the precedent set by the function **whenAny** in TPL, **when_any** is used.

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 this operator is so essential that without it this proposal would be incomplete and C++ would not have a comprehensive set of asynchronous operations.

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 size does not need to be known from before. Additionally, the second variation provides additional convenience by allowing mixing of futures and shared futures. One option which was considered was to not support this feature, however the benefits to the programmer greatly outweighed any implementation costs. One restriction which has been placed however is that all of the **`future`** and **`shared_future`** objects in the sequence must be of the same type. This is necessary in order to determine the return type.

Return values: For the same reasons as the **`.then`** operator, the return type is always a **`future`**. In most cases, the value of the future is a **`pair`** with the first element containing the index of the completed **`future`** or **`shared_future`**, and the second element is its value. The completed **`future`** is nondeterministically selected, and ideally is the first one which completed. There is a special case for when the inputs are **`future<void>`** and **`shared_future<void>`**. In this case a **`future<size_t>`** with just the index of the winner is returned, as there is no value associated with futures of type 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`** with all of the results from the 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<int, string>> all_f = when_all(shared_future1, future2);

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

Naming: Like **`when_any`**, the naming convention was adopted from TPL and the join keyword was not used.

Exception handling: The exception held by any input futures is transferred to the resulting **future**. If more than one input **future** completes with an exception, one of them is picked, non-deterministically. This alternative was preferred over providing an aggregate of all the exceptions.

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<R>>**: If the input cardinality is unknown at compile time and the futures are all of the same type (except **void**).
- **future<tuple<R0, R1, R2…>>**: If inputs are fixed in number and are of heterogeneous types (except **void**). The inputs can be any arbitrary number of **future** and **shared_future** objects.
- **future<void>**: If either the input is an iterator pair associated to a collection of **future<void>** objects or if the input is a sequence of **future<void>** and **shared_future<void>** objects. All of the inputs must be of type void.

make_future / make_shared_future

This function creates a future for a given value. 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**.

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

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

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

There are two variations 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>.**

make_shared_future has the same functionality as **make_future**, except has a return type of **shared_future**.

VI. Technical Specification

Class scheduler

1. The class **`scheduler`** defines an interface which can be implemented to invoke functions on a specified thread. The implementation is left up to the programmer.

```
namespace std{
  class scheduler {
    public:
       virtual void schedule(function<void()> work_item) = 0;
       virtual ~schedule();
       scheduler(const scheduler& rhs) = delete;
       scheduler& operator=(const scheduler& rhs) = delete;
  };
}
```

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

```
template<typename F>
auto future::then(F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto future::then(scheduler &s, F&& func) -> future<decltype(func(*this))>;

template<typename F>
auto 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 **`future`** object as a parameter. The second function takes a **`scheduler`** 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.

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 scheduler.
- When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or scheduler.
- 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

```
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 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`**.

[*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 future::ready() const;
```

Notes: queries the shared state to see if it is ready, returns true if it is and false if it isn't

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

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

template<typename F>
auto shared_future::then(scheduler &s, 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 **`future`** object as a parameter. The second function takes a **`scheduler`** 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.

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 scheduler.
- When the scheduler or launch policy is not provided the continuation inherits the parent's launch policy or scheduler.
- 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

`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 shared_future::ready() const;
```

Notes: queries the shared state to see if it is ready, returns true if it is and 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<R>** and **shared_future<R>** objects, where **R** need not be the same type (cannot be **void)**. There is also a special case for each that returns a **future<void>** object when the inputs are either **future<void>** or **shared_future<void>** (**R** is of type **void**).
- Calling the first signature of **when_all** with iterator pointing to **future<R&> /shared_future<R&>** is a compile time error, as the return type would be **future<vector<R&>>** which itself is invalid.
- Calling the first signature of **when_all** where **InputIterator** index first equals index last, returns a **future** with an empty vector as its result is ready by the time **when_all** returns

Effects:

- Each **`future`** and **`shared_future`** is run and the resulting value (if any) is stored (as a vector or tuple) in the shared state of returned future.
- If **`when_all`** is called with zero arguments, **`future<tuple<>>`** is returned.
- If any of the inputs throws an exception, then the exception held by any input future is transferred to the resulting future. If more than one input future completes with an exception, one of them is picked, non-deterministically. All other exceptions are swallowed.

Returns:

- **`future<vector<typename decay<R>::type>>`** if input is a pair of **`InputIterators`**, where **`R`** is the type of the **`future`** or **`shared_future`** objects that the iterators are associated with.
- **`future<tuple<typename decay<R>::type...>>`** if input is any arbitrary number of **`future<R>`** objects and **`shared_future<R>`** objects, where **`R`** is not void.
- **`future<void>`** if input is of type **`future<void>`** or of type **`shared_future<void>.`** These futures do not have a resulting value.

Postcondition:

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

Remarks: The first signature shall not participate in overload resolution if **`decay<InIt>::type`** is **`std::future`** or **`std::shared_future`**

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 2 versions of **`when_any`**. The first version takes a pair of **`InputIterators`**. The second takes an arbitrary number of **`future<R>`** and **`shared_future<R>`**, where R must be the same type and **`R`** is not void. There is also a special case for each when the inputs are of **`future<void>`** or **`shared_future<void>`** (**`R`** is of type **`void`**). This case only returns the index of the associated **`future`** or **`shared_future`** object which completed first, as there is no associated value.

Effects:

- Each **`future`** and **`shared_future`** is run and the resulting value (if any) and its corresponding index is stored as a **`pair`**, in the shared state of returned as a future. The "winner" is non-deterministically selected from the completed (shared state has a value or an exception) inputs. In the implementation this is ideally the one that completed first.
- **`when_any`** accepting zero argument does not exist (compile time error)

Returns:

- **`future<pair<size_t, typename decay<R>::type>>`** if inputs are a pair of **`InputIterators`** or any arbitrary number for **`future`** and **`shared_future`** objects. The returned **`future`** value contains a **`pair`** with the index of the completed **`future`** or **`shared_future`** object and its associated value.
- **`future<size_t>`** if input is of type **`future<void>`** and **`shared_future<void>.`** These futures do not have a resulting value.

Postcondition:

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

Remarks: The first signature shall not participate in overload resolution if **`decay<InIt>::type`** is **`std::future`** or **`std::shared_future`**

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

```
template <typename T>
future<typename decay<T>::type> make_future(T&& value);
future<void> make_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>, **`ready() = true`**

30.6.X Function template **`make_shared_future`** [futures.make_shared_future]

```
template <typename T>
```

```
shared_future<typename decay<T>::type> make_shared_future(T&& value);
shared_future<void> make_shared_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:

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

Postcondition:

- Returned shared_future<T>, `valid() == true`
- Returned shared_future<T>, `ready() = true`

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

```
template<class F, class... Args>
  future<typename result_of<F(Args...)>::type>
  async(scheduler &s, F&& f, Args&&... args);
```

Effects: The scheduler is given a `function<void ()>` which calls `INVOKE (DECAY_COPY (std::forward<F>(f)), DECAY_COPY (std::forward<Args>(args))...)` (20.8.2, 30.3.1.2. The implementation of the scheduler is decided by the programmer.

Remarks: Need to add a solution to disambiguate this signature from the other overloads. Need to ensure that policy and no-policy overloads do not participate in overload resolution if `decay<F>::type` is `std::scheduler` (and vice versa)

VII. Acknowledgements

- Niklas Gustafsson
- Artur Laksberg
- Herb Sutter
- Shuai Wang
- Stephan T. Lavavej
- Rahul V. Patil

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