`completion<T>`

Improving the `future<T>` with monads

Travis Gockel

C++ Pirate
SolidFire

22 September 2015

Outline

Outline

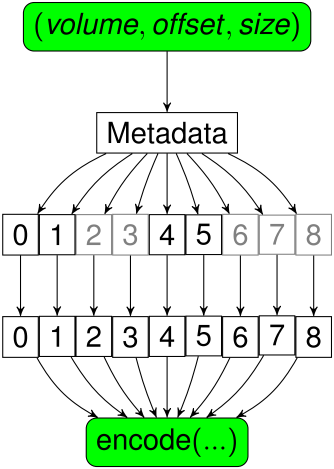
1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

Who is SolidFire?

- Distributed all-flash block storage vendor
  - Data loss or unavailability is unacceptable
  - ...even in the face of hardware failure
  - Deployed as an appliance – software must maintain itself
  - ...in environments we do not control
- Small engineering team ($\lessapprox$ 40 C++ programmers)
  - Team efficiency is critical
  - Growing – code can't look completely foreign

The Problem Space

- Request for data comes into the transport service over iSCSI
- Look up the internal IDs for the request
- Sometimes, the data is in the local cache
- For those that are not, we fetch associated chunks of data from other services
  $\longrightarrow$ User-visible latency is slowest-responding service



Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

What is the goal of `future<T>`?

- A mechanism for asynchronous communication between a provider (`promise<T>`) and a receiver (`future<T>`)
- Make concurrent programming easier

std::future<T>

```
template <typename T> class future {
public:
  T get();
  bool valid() const noexcept;
  void wait() const;
  template <typename TRep, typename TPeriod>
  future_status wait_for(const chrono::duration<TRep,
      TPeriod>& rel_time) const;
  template <typename TClock, typename TDuration>
  future_status wait_until(const
      chrono::time_point<TClock, TDuration>& abs_time)
      const;
  shared_future<T> share();
};
```

How do I use this?

```
buffer read_multi(location x, location y) {
  future<buffer> fut_z1 = async(&read, x);
  buffer z2 = read(y);
  return fut_z1.get() + z2;
}
```

How do I use this?

```
buffer read_multi(location x, location y) {
  future<buffer> fut_z1 = async(&read, x);
  buffer z2 = read(y);
  return fut_z1.get() + z2;
}
```

Only sort of asynchronous.

What is wrong with `future<T>`?

`get` and `wait`

What is wrong with `future<T>`?

`get` and `wait`

Should never be used.

What about then<F>?

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

Usage...

```
future<int>    x = async(&thing);
future<double> y = x.then([] (future<int> f) { return
    double(f.get()); });
```

Awkward for Aggregation

```
std::vector<std::future<int>> all_tasks = foo();
std::future<std::vector<std::future<int>>> all_done
    = when_all(make_move_iterator(begin(all_tasks)),
               make_move_iterator(end(all_tasks))
              );
```

- Which `future<T>::get` will block?
- Slow – `future<T>` is multithreaded, so it *must* deal with locking
  $\longrightarrow$ Adding the values in `std::vector<int>` is much faster than adding `std::vector<std::future<int>>`

```
try {
  try {
    try {
      foo();
    } catch (...) {
      throw;
    }
    bar();
  } catch (...) {
    throw;
  }
  baz();
} catch (const
   std::exception& ex) {
  report(ex);
}
```

```
try {
  foo();
  bar();
  baz();
} catch (const
   std::exception& ex) {
  report(ex);
}
```

```
try {
  try {
    try {
      foo();
    } catch (...) {
      throw;
    }
    bar();
  } catch (...) {
    throw;
  }
  baz();
} catch (const
   std::exception& ex) {
  report(ex);
}
```

```
foo_async()
.then([] (std::future<void> x)
    {
        x.get(); return bar();
      })
.then([] (std::future<void> y)
    {
        y.get(); return baz();
      })
.then([] (std::future<void> z)
    {
        try {
          z.get();
        } catch (const
           std::exception& ex) {
          report(ex);
        }
      });
```

Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

Error Reporting

- `future<T>` combines two ideas: asynchronous communication *and* error reporting.
- Can we split this?

expected<T, E>

“N4015: A proposal to add a utility class to represent expected monad.”

```
template <typename T, typename E> class expected {
public:
  auto status() const -> bool;
  auto get() -> T;
  template <typename F> auto map(F f) ->
     expected<decltype(f(std::declval<T>()))>;
  template <typename F> auto catch_error(F f)
    -> expected<std::common_type_t<T,
       decltype(f(std::declval<std::expection_ptr>()))>>;
  // ...
};
```

expected<T, E>

```
expected<int>    x = foo();
expected<double> y = x.map([] (int a) { return
    double(a); });
expected<double> z = y.catch_error([] (auto) { return
    0.0; });
expected<string> w = z.map([] (double a) { return
    to_string(a); });
cout << w.get() << endl;
```

Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

completion<T>

```
template <typename T> class completion {
public:
  auto status() const -> completion_status;
  auto on_complete(F f) -> void;
  template <typename F> auto then(F f) ->
      completion<decltype(f(std::declval<expected<T>>()))>;
  template <typename F> auto map(F f) ->
      completion<decltype(f(std::declval<T>()))>;
  template <typename F> auto catch_error(F f)
    -> completion<std::common_type_t<T,
         decltype(f(std::declval<std::exception_ptr>()))>>;
  auto disable() -> void;
};
```

Missing features?

`get` and `wait` Why bother?
But we can for compatibility with `std::future<T>`

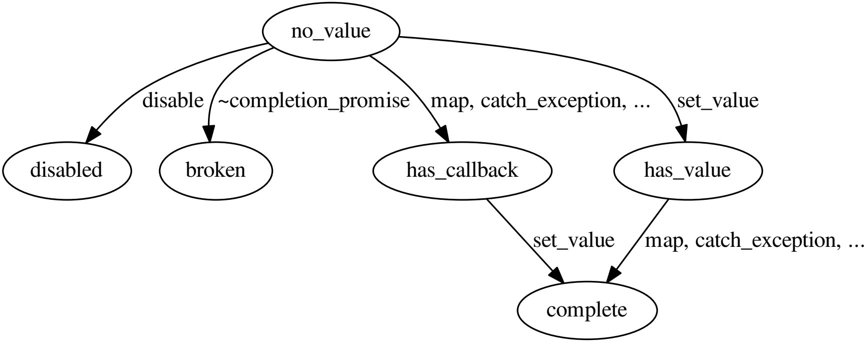
Overloads for `T&` and `void`? Not needed – let `expected<T>` deal with that

completion_data<T>

```
template <typename T> struct completion_data {
  mutable spin_mutex                        protect;
  completion_status                         status;
  optional<expected<T>>                     value;
  function<void (expected<T>&&)>  callback;

  reference_tracking_data<completion_data> tracker;
};
```

completion_status Transitions



completion<T>::then(Func&&)

```
template <typename T>
template <typename Func>
completion<T>::then(Func&& func)
  ->
     completion<decltype(func(std::declval<expected<T>>()))>
```

Purpose: Call `func` with an `expected<T>`

- If status is `has_value`, call `func` immediately
- If status is `no_value`, store `func` for later use
- All other cases ⟶ error

Arg `func`: `U (*)(expected<T>&&)`

completion<T>::on_complete(Func&&)

```
template <typename T>
template <typename Func>
void completion<T>::on_complete(Func&& func);
```

Purpose: Chaining the last step in the process

Arg `func`: `void (*)(expected<T>&&)`

completion<T>::map(Func&&)

```
template <typename T>
template <typename Func>
auto completion<T>::map(Func&& func);
  -> completion<decltype(func(std::declval<T>()))>
```

Purpose: Similar to `then(F&&)`, *but* checks the status of `expected<T>`

- If completed in success, call the function
- If completed in failure, forward the `exception_ptr` to the next step

  $\longrightarrow$ Not throwing is significantly faster!

Arg `func`: `U (*)(T&&)`

Note: `completion<T>::catch_error(Func&& func)` has the opposite rules

completion<T>::get()

```
template <typename T>
T
completion<T>::get() {
  std::promise<expected<T>> promise;
  on_complete([&promise] (expected<T>&& x) {
                promise.set_value(std::move(x));
              });
  return promise.get_future().get().get();
}
```

Only incur the slow `mutex` *if somebody asks for the slow call.*

⟶ The `wait` family requires internal tricks, but is still implementable.

Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

completion_promise<T>

- Same role as `std::promise<T>`
- Allows for the delivery of values to the `completion`
- Shares the `completion_data`

completion_promise<T>::complete(expected<T

```
template <typename T>
void completion_promise<T>::complete(expected<T> value);
```

Purpose: Deliver a value to the associated `completion<T>`

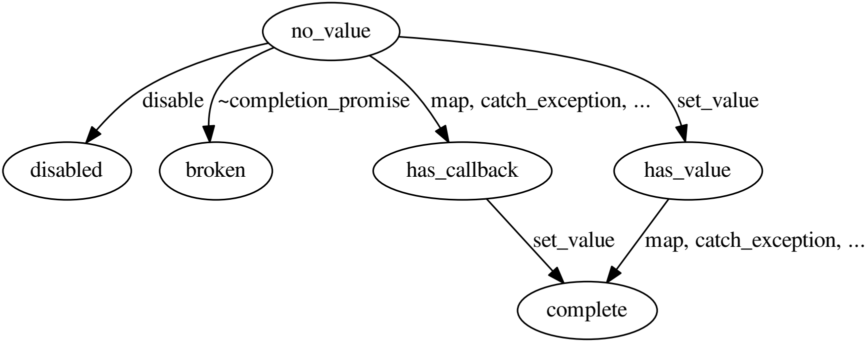
- If status is `has_callback`, call it immediately
- If status is `no_value`, store `value` for later retrieval
- If status is `disabled`, do nothing
- All other cases ⟶ error

std::promise<T> Compatibility Functions

```
template <typename... U>
void set_value(U&&... args) {
  complete(make_expected(std::forward<U>(args)...));
}

void set_exception(std::exception_ptr ex) {
  complete(make_unexpected(std::move(ex)));
}
```

completion_status Transitions



Outline

Dealing with Broken Promises

When the `completion_promise` is destroyed without having set the value, what should happen?

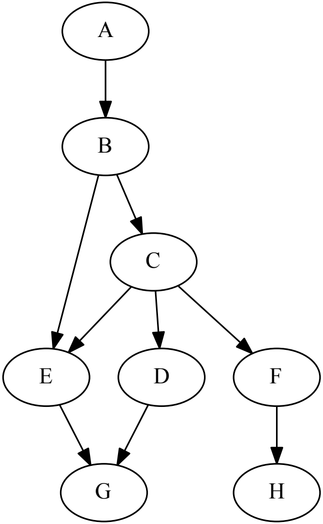
Do nothing? Potentially causes deadlock elsewhere.

Complete with a `broken_promise`? This can trigger a chain of actions, thus blocking in the destructor.

throw? Throwing in a destructor is bad.

`assert(false)`? Panic – something terrible has happened.

Disabling



- Disable *H* – should *F* be disabled?
  - Okay if *F* has no side-effects
  - But how can I know? $\longrightarrow$ Allow opt-in
- Disable *C* – what about *D*, *E* and *F*?
  - Only allow disabling leafs (do nothing)
- Is this useful?
  - Perhaps “disabling” should be handled by a different construct

Declarative “Pureness”

```
template <typename T>
template <typename Func>
auto completion<T>::then(Func&& func) -> ...
{
  bool is_pure = continuation_traits<Func>::is_pure;
  // ...
}
```

```
template <typename F> struct pure_func_wrapper { ... };
template <typename F> pure_func_wrapper<F>
    pure_func(F&&);
```

Better: Pick up `__attribute__((pure_function))` automatically?

Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

Introduction

- Algorithms are range-based where possible

concat

```
template <typename... T>
completion_concat_result_t<completion<T>...>
concat(completion<T>&&...);
```

Purpose: Combines heterogeneous `completion`s and notifies you when all of them are completed.

Complexity: $O(n)$

Return Type: Similar rules to `tuple_cat`

concat usage

```
template <typename... T>
using expected_tuple = tuple<expected<T>...>;

completion<int>                         a = foo();
completion<void>                        b = bar();
completion<tuple<double, string>> c = baz();

completion<expected_tuple<int, void, double, string>> d =
    concat(move(a), move(b), move(c));
// Do things with d
```

for_each

```
template <typename TInputRange, typename FUnitFunc>
completion<void>
for_each(TInputRange&&     input,
         const FUnitFunc& func
        );
```

Purpose: Call a callback for some `completion`s and notifies you when all callbacks have completed running.

Complexity: $O(n)$

`for_each` usage

```
std::vector<completion<int>> input = foo();
completion<void> all_done = for_each(std::move(input),
    bar);
```

collect

```
template <template <typename...> class TTOutput,
          template <typename...> class TTInput,
          typename                       T
         >
completion<TTOutput<expected<T>>>
collect(TTInput<completion<T>>& input);

template <typename                       T,
          template <typename...> class TTInput
         >
completion<TTInput<expected<T>>>
collect(TTInput<completion<T>>& input);
```

Purpose: Transform a `collection<completion<T>>` into a `completion<collection<T>>`

Complexity: $O(n)$

collect

```
template <typename                            T,
          template <typename...> class TTInput
         >
completion<TTInput<expected<T>>>
collect(TTInput<completion<T>>& input) {
  return collect<TTInput>(input);
}
```

collect_n

```
template <template <typename...> class TTOutput,
          template <typename...> class TTInput,
          typename                     T
         >
completion<pair<TTOutput<std::pair<size_t, expected<T>>>>
collect_n(TTInput<completion<T>>& input, size_t min);

template <typename                     T,
          template <typename...> class TTInput
         >
completion<pair<TTOutput<std::pair<size_t, expected<T>>>>
collect_n(TTInput<completion<T>>& input, size_t min);
```

Purpose: Transform a `collection<completion<T>>` into a `completion<collection<T>>`

Complexity: $O(n)$

Arg `min`: Minimum amount of items to wait for

sequence

```
template <template <typename...> class TTOutput,
          template <typename...> class TTInput,
          typename                         T
         >
completion<TTOutput<expected<T>>>
sequence(TTInput<completion<T>>& input);

template <typename                         T,
          template <typename...> class TTInput
         >
completion<TTInput<expected<T>>>
sequence(TTInput<completion<T>>& input);
```

Purpose: Transform a `collection<completion<T>>` into a `completion<collection<T>>` *in the same order*.

Complexity: $O(n \log n)$ (data structure dependent)

Outline

Who performs the next step?

- When binding a continuation, who should perform the action?
- It depends on `completion::status`...

  `has_value`: The binder invokes immediately

  `no_value`: The deliverer will invoke (later)
- This can be problematic (sometimes)
  - If the deliverer is on a “never block” thread (example: RPC message delivery), we can quickly lose control
  - If the receiver is client-facing, immediately invoking is “unfair” – the result is usually strange-looking latency in unrelated activities
- Enter the `executor`!

What is an Executor?

- Decides when and where to perform some action
- Launch policy describes if we should execute now or later

  `immediate`: Always execute the next step inline

  `async`: Always post the task to work later (potentially on a different thread)

  `sharing`: If the executor has queued work, post to work later; otherwise, execute immediately
- Executors use various methods of determining what order to run queued tasks

  `fifo`: Run tasks in almost the same order they were queued

  `fifo_strict`: Run tasks in exactly the same order they were queued

  `qos`: Use information about the task to prioritize (requires task introspection)

  `random`: Run in a random order (useful for testing)

Informing the next step

```
template <typename Func>
auto then_with(execute_token exec, Func&& continuation)
    -> ...;
template <typename Func>
auto map_with(execute_token exec, Func&& continuation)
    -> ...;
// etc.
```

What is this `execute_token`?

- `executor`-defined data structure for informing it how to run
  $\longrightarrow$ Often just a `shared_ptr<executor>`
- `execute_token`s are created through the base `executor::get_token()` method *or* through implementation-specific named functions
  Example: `priority_worker::get_priority_token(int)`

QoS-based Execution

```
execute_token token = worker->create_job_token(volume);
read_metadata(volume, lba, length)
  .map_with(token, check_cache, _1)
  .map_with(token, remote_data_fetch, m_rpc, _1)
  ...
```

- Create a new token at the beginning of each user request (really: draw them from a pool)
- Prioritization happens at the beginning of each “next step” – if work units are small enough, we do not need to support reprioritization
  $\longrightarrow$ Reprioritization isn’t beneficial in our use, anyway

Outline

1. Introduction
   - SolidFire
   - Why a different `future<T>`?
2. `completion<T>`
   - `expected<T, E>`
   - `completion<T>`
   - `completion_promise<T>`
   - Edge Cases
3. Extra
   - Aggregation functions
   - Workers/Executors
   - Implementation

Note

- The remainder of this presentation
- These are meant only as *examples* to demonstrate the idea – they are not “production-ready”

completion<T>::on_complete(F&& func)

```
template <typename F> void on_complete(F&& func) {
  unique_lock lock(impl->protect);
  if (impl->status == completion_status::no_value) {
    impl->callback = std::forward<F>(func);
    impl->status   = completion_status::has_callback;
  } else if (impl->status ==
     completion_status::has_value) {
    auto completer = on_scope_exit([this] { impl->status
        = completion_status::complete; });
    std::forward<F>(func)(std::move(impl->value));
  } else {
    throw std::logic_error("invalid␣status");
  }
}
```

completion<T>::then(F&& func)

```
template <typename F>
then_result_t<F> then(F&& func) {
  unique_lock lock(impl->protect);
  if (impl->status == completion_status::no_value) {
    result_promise promise;
    impl->callback = [promise, func = std::forward<F>(func)] (expected<T>&& x) {
                         promise.complete(std::move(func), std::move(x));
                       };
    impl->status = completion_status::has_callback;
    return promise.get_completion();
  } else if (impl->status == completion_status::has_value) {
    auto completer = on_scope_exit([this] { impl->status = completion_status::complete; });
    return result_promise::create_completed(try_to(std::move(func), std::move(impl->value)))
           .get_completion();
  } else {
    throw std::logic_error("invalid status");
  }
}
```

completion<T>::map(F&& func)

```
template <typename F>
completion_map_result_t<completion, F>
map(F&& func) {
  return then([func = std::forward<F>(func)]
      (expected<T> x) {
                 return std::move(x).map(func).get();
             });
}
```

Yuck! Should set the result directly instead of relying on `get`.

for_each

```
template <typename TInputRange, typename FUnitFunc>
completion<void>
for_each(TInputRange&& input, const FUnitFunc& func) {
  completion_promise<void> promise;
  auto remaining =
      make_shared<atomic<size_t>>(distance(input));
  for (auto&& step : input)
    step.then(func)
        .on_complete([promise, remaining]
            (expected<void>&&) mutable {
                      if (remaining->fetch_sub(1,
                          std::memory_order_relaxed) ==
                          1)
                        promise.set_result();
                    });
  return promise.get_completion();
}
```

for_each

```
template <typename TInputRange, typename FUnitFunc>
completion<void>
for_each(TInputRange&& input, const FUnitFunc& func) {
  completion_promise<void> promise;
  auto remaining =
     make_shared<atomic<size_t>>(distance(input));
  for (auto&& step : input)
    step.then(func)
        .on_complete([promise, remaining]
           (expected<void>&&) mutable {
                      if (remaining->fetch_sub(1,
                         std::memory_order_relaxed) ==
                         1)
                      promise.set_result();
```

Why `std::memory_order_relaxed`?

The user-defined `func` is called under a different `completion` action, which happens under a lock, so the action is sequentially-consistent.

collect

```
template <template <typename...> class TTOutput,
          template <typename...> class TTInput,
          typename                       T
         >
completion<TTOutput<expected<T>>>
collect(TTInput<completion<T>>& input)
{
  auto collected = make_shared<pair<spin_mutex, TTOutput<expected<T>>>();
  auto done = for_each(input,
                       [collected] (expected<T>&& x) {
                         spin_lock lock(collected->first);
                         collected->second.insert(end(collected->second), std::move(x))
                       })
              .then([collected] (expected<void>) { return std::move(collected->second); };

  #if COMPLETION_COLLECT_SHUFFLE
  done = done.map([] (auto x) {
                    random_shuffle_if_possible(x);
                    return x;
                 });
  #endif
  return done;
}
```

completion_promise<T>::complete(expected<

```
template <typename U> void complete(expected<U> value) {
  if (impl->status == completion_status::no_value) {
    impl->value  = std::move(value);
    impl->status = completion_status::has_value;
  } else if (impl->status ==
      completion_status::has_callback) {
    auto completer = on_scope_exit([this] { impl->status
        = completion_status::complete; });
    impl->callback(std::move(value));
  } else if (impl->status ==
      completion_status::disabled) {
    // do nothing?
  } else {
    throw std::logic_error("invalid␣status");
  }
}
```

concat

```
template <size_t KInputCount, typename... TResultArgs>
struct concat_completer {
  using result_type = tuple<TResultArgs...>;

  atomic<size_t>                      remaining{KInputCount};
  result_type                         values;
  completion_promise<result_type> promise;

  template <size_t KStartIdx, typename... TValues>
  void complete_part(integral_constant<size_t,
      KStartIdx>, TValues&&... values);

  template <size_t... DestIdxs, typename... TValues>
  void set_data(index_tuple<DestIdxs...>, TValues&&...
      values);
};
```

concat

```
template <size_t... DestIdxs, typename... TValues>
void set_data(index_tuple<DestIdxs...>, TValues&&...
    values) {
  std::tie(std::get<DestIdxs>(values)...) =
      std::make_tuple<TValues...>(values...);
}
```

concat

```
template <size_t KStartIdx, typename... TValues>
void complete_part(integral_constant<size_t, KStartIdx>,
    TValues&&... values) {
  set_data(make_index_tuple_range<KStartIdx,
       sizeof...(TValues)>(),
            std::forward<TValues>(values)...);

  if (remaining.fetch_sub(1, std::memory_order_seq_cst)
       == 1)
    deliver_promise();
}
```

concat

```
template <size_t KStartIdx, typename... TValues>
void complete_part(integral_constant<size_t, KStartIdx>,
     TValues&&... values) {
   set_data(make_index_tuple_range<KStartIdx,
        sizeof...(TValues)>(),
              std::forward<TValues>(values)...);

   if (remaining.fetch_sub(1, std::memory_order_seq_cst)
        == 1)
      deliver_promise();
}
```

Why `std::memory_order_seq_cst`?

Before delivering the promise, we must make sure that other threads which are *not* calling `deliver_promise` have their memory modifications from `set_data` visible to all other threads.