Generally speaking, you use a promise-future pair by first creating a `std::promise<T>`, where `T` is the type of data you're planning to send through it; then creating the wormhole's "future" end by calling `p.get_future()`. When you're ready to fulfill the promise, you call `p.set_value(v)`. Meanwhile, in some other thread, when you're ready to retrieve the value, you call `f.get()`. If a thread calls `f.get()` before the promise has been fulfilled, that thread will block until the promise is fulfilled and the value is ready to retrieve. On the other hand, when the promise-holding thread calls `p.set_value(v)`, if nobody's waiting, that's fine; `set_value` will just record the value `v` in memory so that it's ready and waiting whenever anyone *does* ask for it via `f.get()`.

Let's see `promise` and `future` in action!

```
std::promise<int> p1, p2;
std::future<int> f1 = p1.get_future();
std::future<int> f2 = p2.get_future();

  // If the promise is satisfied first,
  // then f.get() will not block.
p1.set_value(42);
assert(f1.get() == 42);

  // If f.get() is called first, then it
  // will block until set_value() is called
  // from some other thread.
std::thread t([&](){
  std::this_thread::sleep_for(100ms);
  p2.set_value(43);
});
auto start_time = std::chrono::system_clock::now();
assert(f2.get() == 43);
auto elapsed = std::chrono::system_clock::now() - start_time;
printf("f2.get() took %dms.\n", count_ms(elapsed));
t.join();
```

(For the definition of `count_ms`, see the previous section, *Special-purpose mutex types*.)

One nice detail about the standard library's `std::promise` is that it has a specialization for `void`. The idea of `std::future<void>` might seem a little silly at first--what good is a wormhole if the only data type you can shove through it is a type with no values? But in fact `future<void>` is extremely useful, whenever we don't care so much about the *value* that was received as about the fact that some signal was received at all. For example, we can use `std::future<void>` to implement yet a third version of our "wait for thread B to launch" code:

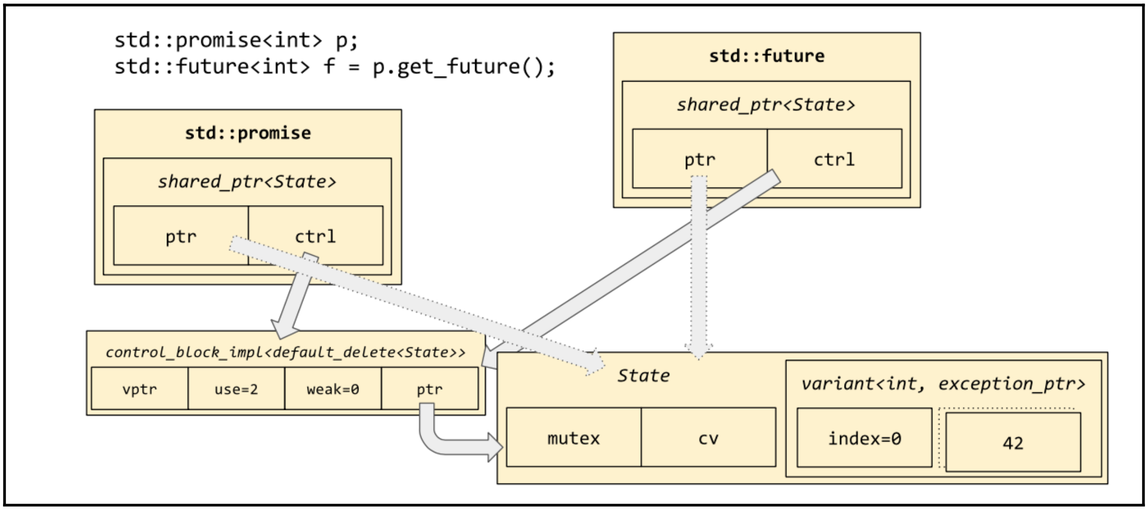
```
std::promise<void> ready_p;
std::future<void> ready_f = ready_p.get_future();

std::thread thread_b([&]() {
  prep_work();
  ready_p.set_value();
  main_work();
});

  // Wait for thread B to be ready.
ready_f.wait();
  // Now thread B has completed its prep work.
```

Compare this version to the code samples from the section titled "Waiting for a condition." This version is much cleaner! There's practically no cruft, no boilerplate at all. The "signal B's readiness" and "wait for B's readiness" operations both take only a single line of code. So this is definitely the preferred way to signal between a single pair of threads, as far as syntactic cleanliness is concerned. For yet a fourth way to signal from one thread to a group of threads, see this chapter's subsection titled "Identifying individual threads and the current thread."

There *is* a price to pay for `std::future`, though. The price is dynamic memory allocation. You see, `promise` and `future` both need access to a shared storage location, so that when you store `42` in the promise side, you'll be able to pull it out from the future side. (That shared storage location also holds the mutex and condition variable required for synchronizing between the threads. The mutex and condition variable haven't disappeared from our code; they've just moved down a layer of abstraction so that we don't have to worry about them.) So, `promise` and `future` both act as a sort of "handle" to this shared state; but they're both movable types, so neither of them can actually hold the shared state as a member. They need to allocate the shared state on the heap, and hold pointers to it; and since the shared state isn't supposed to be freed until *both* handles are destroyed, we're talking about shared ownership via something like `shared_ptr` (see `Chapter 6`, *Smart Pointers*). Schematically, `promise` and `future` look like this:



The shared state in this diagram will be allocated with `operator new`, unless you use a special "allocator-aware" version of the constructor `std::promise`. To use `std::promise` and `std::future` with an allocator of your choice, you'd write the following:

```
MyAllocator myalloc{};
std::promise<int> p(std::allocator_arg, myalloc);
std::future<int> f = p.get_future();
```

`std::allocator_arg` is defined in the `<memory>` header. See `Chapter 8`, *Allocators*, for the details of `MyAllocator`.

Packaging up tasks for later

Another thing to notice about the preceding diagram is that the shared state doesn't just contain an `optional<T>`; it actually contains a `variant<T, exception_ptr>` (for `variant` and `optional`, see `Chapter 5`, *Vocabulary Types*). This implies that not only can you shove data of type `T` through the wormhole; you can also shove *exceptions* through. This is particularly convenient and symmetrical because it allows `std::future<T>` to represent all the possible outcomes of calling a function with the signature `T()`. Maybe it returns a `T`; maybe it throws an exception; and of course maybe it never returns at all. Similarly, a call to `f.get()` may return a `T`; or throw an exception; or (if the promise-holding thread loops forever) might never return at all. In order to shove an exception through the wormhole, you'd use the method `p.set_exception(ex)`, where `ex` is an object of type `std::exception_ptr` such as might be returned from `std::current_exception()` inside a catch handler.

Let's take a function of signature `T()` and package it up in a future of type `std::future<T>`:

```
template<class T>
class simple_packaged_task {
  std::function<T()> m_func;
  std::promise<T> m_promise;
public:
  template<class F>
  simple_packaged_task(const F& f) : m_func(f) {}

  auto get_future() { return m_promise.get_future(); }

  void operator()() {
    try {
      T result = m_func();
      m_promise.set_value(result);
    } catch (...) {
      m_promise.set_exception(std::current_exception());
    }
  }
};
```

This class superficially resembles the standard library type `std::packaged_task<R(A...)>`; the difference is that the standard library type takes arguments, and uses an extra layer of indirection to make sure that it can hold even move-only functor types. Back in `Chapter 5`, *Vocabulary Types*, we showed you some workarounds for the fact that `std::function` can't hold move-only function types; fortunately those workarounds are not needed when dealing with `std::packaged_task`. On the other hand, you'll probably never have to deal with `std::packaged_task` in your life. It's interesting mainly as an example of how to compose promises, futures, and functions together into user-friendly class types with externally very simple interfaces. Consider for a moment: The `simple_packaged_task` class above uses type-erasure in `std::function`, and then has the `std::promise` member, which is implemented in terms of `std::shared_ptr`, which does reference counting; and the shared state pointed to by that reference-counted pointer holds a mutex and a condition variable. That's quite a lot of ideas and techniques packed into a very small volume! And yet the interface to `simple_packaged_task` is indeed simple: construct it with a function or lambda of some kind, then call `pt.get_future()` to get a future that you can `f.get()`; and meanwhile call `pt()` (probably from some other thread) to actually execute the stored function and shove the result through the wormhole into `f.get()`.

If the stored function throws an exception, then `packaged_task` will catch that exception (in the promise-holding thread) and shove it into the wormhole. Then, whenever the other thread calls `f.get()` (or maybe it already called it and it's blocked inside `f.get()` right now), `f.get()` will throw that exception out into the future-holding thread. In other words, by using promises and futures, we can actually "teleport" exceptions across threads. The exact mechanism of this teleportation, `std::exception_ptr`, is unfortunately outside the scope of this book. If you do library programming in a codebase that uses a lot of exceptions, it is definitely worth becoming familiar with `std::exception_ptr`.

The future of futures

As with `std::shared_mutex`, the standard library's own version of `std::future` is only half-baked. A much more complete and useful version of `future` is coming, perhaps in C++20, and there are very many third-party libraries that incorporate the best features of the upcoming version. The best of these libraries include `boost::future` and Facebook's `folly::Future`.

The major problem with `std::future` is that it requires "touching down" in a thread after each step of a potentially multi-step computation. Consider this pathological usage of `std::future`:

```
template<class T>
auto pf() {
  std::promise<T> p;
  std::future<T> f = p.get_future();
  return std::make_pair(std::move(p), std::move(f));
}

void test() {
  auto [p1, f1] = pf<Connection>();
  auto [p2, f2] = pf<Data>();
  auto [p3, f3] = pf<Data>();

  auto t1 = std::thread([p1 = std::move(p1)]() mutable {
    Connection conn = slowly_open_connection();
    p1.set_value(conn);
    // DANGER: what if slowly_open_connection throws?
  });
  auto t2 = std::thread([p2 = std::move(p2)]() mutable {
    Data data = slowly_get_data_from_disk();
    p2.set_value(data);
  });
  auto t3 = std::thread(
  [p3 = std::move(p3), f1 = std::move(f1)]() mutable {
    Data data = slowly_get_data_from_connection(f1.get());
    p3.set_value(data);
  });
  bool success = (f2.get() == f3.get());

  assert(success);
}
```

Notice the line marked `DANGER`: each of the three thread bodies has the same bug, which is that they fail to catch and `.set_exception()` when an exception is thrown. The solution is a `try...catch` block, just like we used in our `simple_packaged_task` in the preceding section; but since that would get tedious to write out every time, the standard library provides a neat wrapper function called `std::async()`, which takes care of creating a promise-future pair and spawning a new thread. Using `std::async()`, we have this much cleaner-looking code:

```
void test() {
  auto f1 = std::async(slowly_open_connection);
  auto f2 = std::async(slowly_get_data_from_disk);
```

```
  auto f3 = std::async([f1 = std::move(f1)]() mutable {
    return slowly_get_data_from_connection(f1.get());
    // No more danger.
  });
  bool success = (f2.get() == f3.get());

  assert(success);
}
```

However, this code is cleaner only in its aesthetics; it's equally horrifically bad for the performance and robustness of your codebase. This is *bad* code!

Every time you see a `.get()` in that code, you should think, "What a waste of a context switch!" And every time you see a thread being spawned (whether explicitly or via `async`), you should think, "What a possibility for the operating system to run out of kernel threads and for my program to start throwing unexpected exceptions from the constructor of `std::thread`!" Instead of either of the preceding codes, we'd prefer to write something like this, in a style that might look familiar to JavaScript programmers:

```
void test() {
  auto f1 = my::async(slowly_open_connection);
  auto f2 = my::async(slowly_get_data_from_disk);
  auto f3 = f1.then([](Connection conn) {
    return slowly_get_data_from_connection(conn);
  });
  bool success = f2.get() == f3.get();

  assert(success);
}
```

Here, there are no calls to `.get()` except at the very end, when we have nothing to do but wait for the final answer; and there is one fewer thread spawned. Instead, before `f1` finishes its task, we attach a "continuation" to it, so that when `f1` does finish, the promise-holding thread can immediately segue right into working on the continuation task (if original task of `f1` threw an exception, we won't enter this continuation at all. The library should provide a symmetrical method, `f1.on_error(continuation)`, to deal with the exceptional codepath).

Something like this is already available in Boost; and Facebook's Folly library contains a particularly robust and fully featured implementation even better than Boost's. While we wait for C++20 to improve the situation, my advice is to use Folly if you can afford the cognitive overhead of integrating it into your build system. The single advantage of `std::future` is that it's standard; you'll be able to use it on just about any platform without needing to worry about downloads, include paths, or licensing terms.