std::promise and std::future

In this lesson, we will see how you have full control over the task with std::promise and std::future.

WE'LL COVER THE FOLLOWING

Promise and future are a mighty pair. A promise can put a value, an exception, or simply a notification into the shared data channel. One promise can serve many `std::shared_future` futures. With C++23, we may get ***extended futures*** that are compose-able.

std::promise

`std::promise` enables us to set a value, a notification, or an exception. In addition, the promise can provide its result in a delayed fashion.

Method	Description
`prom.swap(prom2)` and `std::swap(prom, prom2)`	Swaps the promises.
`prom.get_future()`	Returns the future.
`prom.set_value(val)`	Sets the value.
`prom.set_exception(ex)`	Sets the exception.

`prom.set_value_at_thread_exit(val)`	Stores the value and makes it ready if the promise exits.
`prom.set_exception_at_thread_exit(ex)`	Stores the exception and makes it ready if the promise exits.

If the value or the exception is set by the promise more then once, a `std::future_error` exception is thrown.

std::future

A `std::future` enables us to

- pick up the value from the promise.
- ask the promise if the value is available.
- wait for the notification of the promise. This waiting can be done either with a relative time duration or with an absolute time point.
- create a shared future (std::shared_future).

Method	Description
`fut.share()`	Returns a `std::shared_future`. Afterwards, the result is not available anymore.
`fut.get()`	Returns the result which can be a value or an exception.
`fut.valid()`	Checks if the result is available. After calling `fut.get()`, it returns false.
`fut.wait()`	Waits for the result.

`fut.wait_for(relTime)`	Waits for the result, but not longer than for a `relTime`.
`fut.wait_until(absTime)`	Waits for the result, but not longer than until `abstime`.

If a future `fut` asks for the result more than once, a `std::future_error` exception is thrown.

There is a one-to-one relation between the promise and the future. In contrast, `std::shared_future` supports one-to-many relations between a promise and many futures.

std::shared_future

The future creates a shared future by using `fut.share()`. Shared future is associated with its promise and can independently ask for the result. A `std::shared future` has the same interface as a `std::future`.

In addition to the `std::future`, a `std::shared_future` enables us to query the promise independently of the other associated futures.

There are two ways to create a `std::shared_future`:

1. Invoke `fut.share()` on a `std::future fut`. Afterwards, the state is transferred to the shared future. That means `fut.valid() == false`.
2. Initialize a `std::shared_future` from a `std::promise`: `std::shared_future<int> divResult= divPromise.get_future()`.

The handling of a `std::shared_future` is special.

Condition Variables

If we use promises and futures to synchronize threads, they have a lot in common with condition variables. Most of the time, promises and futures are the better choices.

Before we present an example, let's go over a chart that outlines all the details

you need to know.

Criteria	Condition Variables	Tasks
Multiple synchronizations	Yes	No
Critical section	Yes	No
Error handling in receiver	No	Yes
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

The condition variable has one main advantage to a promise and future: you can use condition variables to synchronize threads multiple times. In contrast, a promise can send its notification only once, so you must use more promise and future pairs to get the functionality of a condition variable. If you use the condition variable for only one synchronization, the condition variable will be much more difficult to use properly. A promise and future pair needs no shared variable and, therefore, no lock. These pairs are therefore not prone to spurious or lost wakeups. Additionally, tasks can handle exceptions. There are many reasons to favor tasks over condition variables.

Let’s take a look at a couple of examples of this topic in the next lesson.