

std::promise

`std::promise` and `std::future` gives you the full control.

Method	Description
<code>prom.swap(prom2)</code>	Swaps the promise objects.
<code>prom.get_future()</code>	Returns a <code>std::future</code> .
<code>prom.set_value(val)</code>	Sets the value.
<code>prom.set_exception(ex)</code>	Sets the exception.
<code>prom.set_value_at_thread_exit(val)</code>	Stores the value and sets it to be valid if the thread is done.
<code>prom.set_exception_at_thread_exit(ex)</code>	Stores the exception and sets it to be valid if the thread is done.

std::future

Method	Description
<code>fut.share()</code>	Returns a <code>std::shared_future</code> . Afterwards, <code>fut</code> is not the owner of the shared state anymore.
<code>fut.get()</code>	Returns the value. This can be a value, a notification, or an exception.
<code>fut.valid()</code>	Checks if the shared state is available.
<code>fut.wait()</code>	Waits until the shared state is available.
<code>fut.wait_for(rel_time)</code>	Waits at most for the period of time.
<code>fut.wait_until(abs_time)</code>	Waits at most until the time point.



`fut.wait()` enables it to synchronize the future with the promise.

➔ Most of the time you should prefer to use a promise and future pair instead of a condition variable.

`promiseFuture.cpp`

std::shared_future

The `fut.share()` call creates a `std::shared_future`.

- `std::shared_future`
 - Can independently ask the associated `std::promise` for the value.
 - Has the same interface as `std::future`.
 - Can directly be created.

```
std::shared_future<int> divResult = divPromise.get_future();
```

Condition Variables versus Tasks

Criteria	Condition variable	Task
Multiple synchronizations possible	Yes	No
Critical region	Yes	No
Exception handling in the receiver	No	Yes
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

`promiseFutureSynchronize.cpp`