Introduction to Extended Futures

This lesson gives an overview of extended futures, predicted to be introduced in C++20.

WE'LL COVER THE FOLLOWING

- `std::future`
  - valid vs ready

Tasks in the form of promises and futures have an ambivalent reputation in C++11. On the one hand, they are a lot easier to use than threads or condition variables; on the other hand, they have a great deficiency. They cannot be composed. C++20 will overcome this deficiency.

I have written about tasks in the form of `std::async`, `std::packaged_task`, or `std::promise` and `std::future`. The details are here: *tasks*. With C++20 we may get extended futures.

`std::future`

The name extended futures is quite easy to explain. First, the interface of the C++11 `std::future` was extended; second, there are new functions for creating special futures that are compostable. I will start with my first point.

The extended future has three new methods:

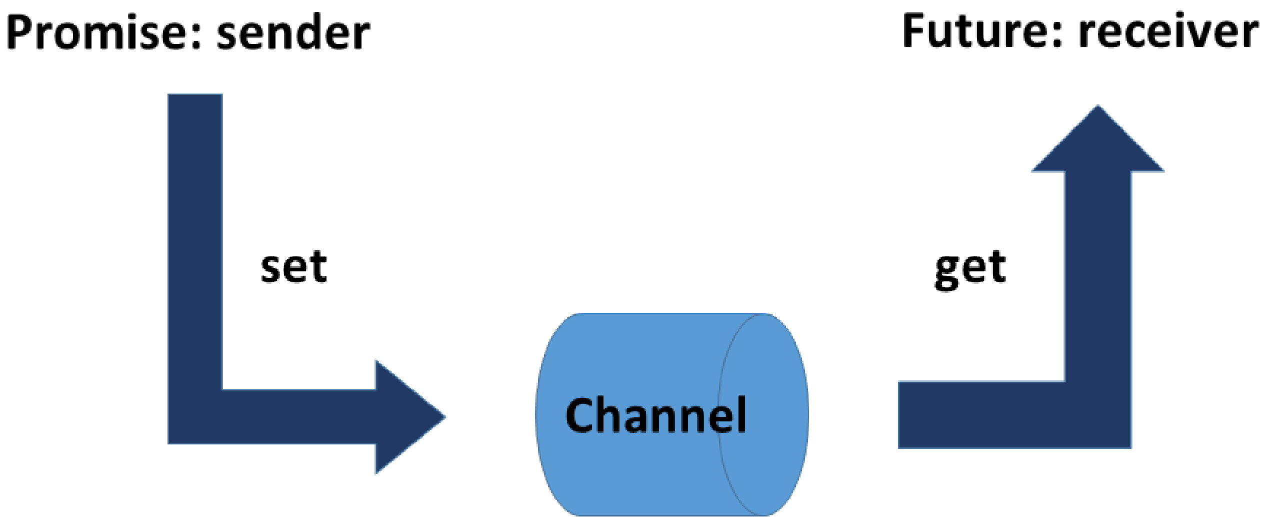
- The unwrapping constructor that unwraps the outer future of a wrapped future (`future<future<T>>`).
- The *predicate* `is_ready` that returns if a shared state is available.
- The method `then` that attaches a continuation to a future.

At first, the state of a future can be `valid` or `ready`.

valid vs ready

- **valid**: a future is valid if it has a shared state (with a promise). This does not have to be the case because you can default-construct an `std::future` without a promise
- **ready**: a future is ready if the shared state is available, i.e. the promise has already produced its value

Therefore, (`valid == true`) is a requirement for (`ready == true`). My mental model of promise and future is that they are the endpoints of a data channel.



Now the difference between valid and ready becomes quite natural. The future is valid if there is a data channel to a promise. The future is ready if the promise has already put its value into the data channel. It is possible to attach one future to another; I will discuss this in the next lesson.