Tasks

In this lesson, we will see how tasks should be your first choice in the general case of multithreading.

WE'LL COVER THE FOLLOWING

Introduction

Tasks were one of the latest additions to the C++11 standard, and they offer a better abstraction than threads.

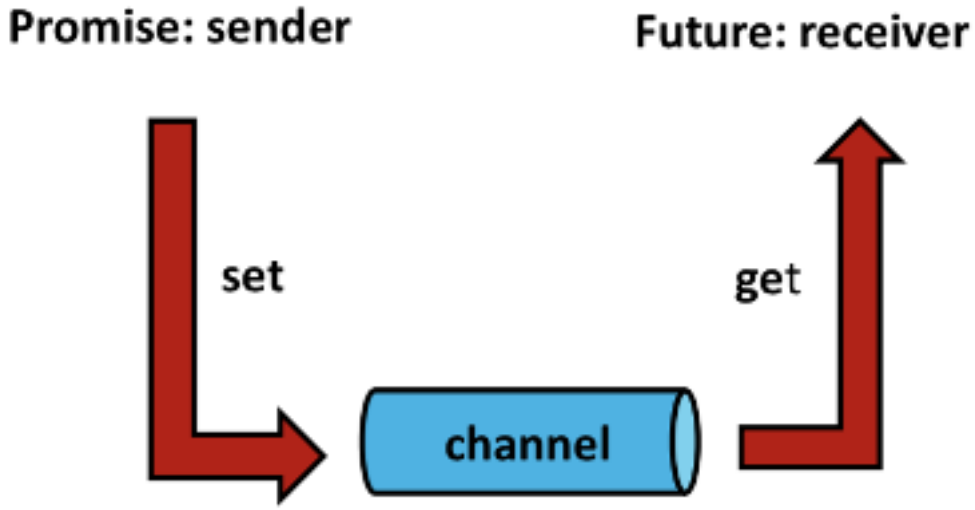
In addition to threads, C++ has tasks to perform work asynchronously. Tasks need the `<future>` header. A task will be parameterized with a work package, and it consists of the two associated components: a `promise` and a `future`. Both are connected via data channel.

The `promise` executes the work packages and puts the result in the data channel. The associated `future` picks up the result. Both communication endpoints can run in separate threads. What is special is that the `future` can pick up the result at a later time, meaning that the resulting calculation by the `promise` is independent of the query of the result by the associated `future`. .

Regard tasks as data channels between communication endpoints

Tasks behave like data channels between communication endpoints. One endpoint of the data channel is called the `promise`. The other endpoint of the data channel is called the `future`. These endpoints can exist in the same or in different threads. The `promise` puts its result in the data channel. The `future` waits for it and picks it up.

channel. The `future` waits for it and picks it up.



Threads vs Tasks

Threads are very different from tasks.

```
// asyncVersusThread.cpp

#include <future>
#include <thread>
#include <iostream>

int main(){

  std::cout << std::endl;

  int res;
  std::thread t([&]{ res = 2000 + 11; });
  t.join();
  std::cout << "res: " << res << std::endl;

  auto fut= std::async([]{ return 2000 + 11; });
  std::cout << "fut.get(): " << fut.get() << std::endl;

  std::cout << std::endl;

}
```

The child thread `t` and the asynchronous function call `std::async` both calculate the sum of `2000` and `11`. The creator thread gets the result from its child thread `t` via the shared variable `res`. This is then displayed in line 14. The call `std::async` in line 16 creates the data channel between the sender

(`promise`) and the receiver (`future`). The `future` asks the data channel with

`fut.get()` (line 17) for the result of the calculation. This `fut.get` call is blocking.

Based on this program, we will explicitly emphasize on the differences between threads and tasks in the chart below:

Criteria	Threads	Tasks
Participants	creator and child thread	`promise` and `future`
Communication	shared variable	communication channel
Thread creation	obligatory	optional
Synchronisation	via `join()` (waits)	`get` call blocks
Exception in child thread	child and creator threads terminates	return value of the `promise`
Kinds of communication	values	values, notifications, and exceptions

Threads need the `<thread>` header. Tasks need the `<future>` header.

Communication between the creator thread and the created thread requires the use of a shared variable. The task communicates via its data channel, which is implicitly protected. Therefore, a task must not use a protection mechanism such as a ***mutex***.

While you can *misuse* a global mutable variable to communicate between the child and its creator, the communication of a task is more explicit. The `future` can request the result of the task only once (by calling `fut.get()`). Calling it more than once results in undefined behavior. This is not true for a

`std::shared_future`, which can be queried multiple times.

The creator thread waits for its child with the call to `join`. The `future` `fut` uses the `fut.get()` call which blocks until the result is available.

If an exception is thrown in the created thread, the created thread will terminate meaning that the whole process is terminated as well. In contrast, the `promise` can send the exception to the `future`, which must handle the exception.

A `promise` can serve one or many `future`s. It can send a value, an exception, or a notification. You can use a `promise` as a safe replacement for a condition variable.

`std::async` is the easiest way to create a `future`, and we'll see why in the next lesson.