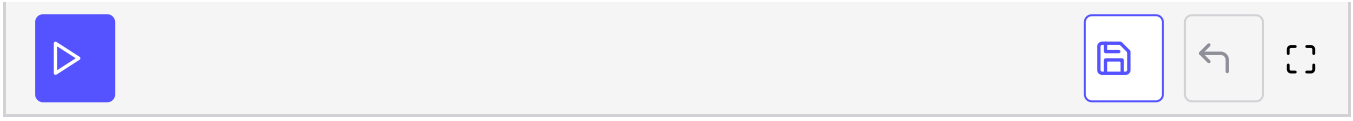
Threads vs Tasks

This lesson highlights the differences between threads and tasks used in C++ for multithreading.

Threads are very different from tasks. Let's see how by looking at this piece of code first:

```
// 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` to calculate both the sum of `2000` and `11`. The creator thread gets the result from its child thread `t` via the shared variable `res` and displays it in line 14. The call `std::async` in line 16 creates the data channel between the sender (promise) and the receiver (future). Following that, 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, I want to explicitly emphasize the differences between threads and tasks.

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 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 like 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 and so will the creator and the whole process. In contrast, the promise can

send the exception to the future, which has to handle the exception.

A promise can serve one or many futures, and it can send a value, an exception, or just a notification. In addition, you can use 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.