- Examples

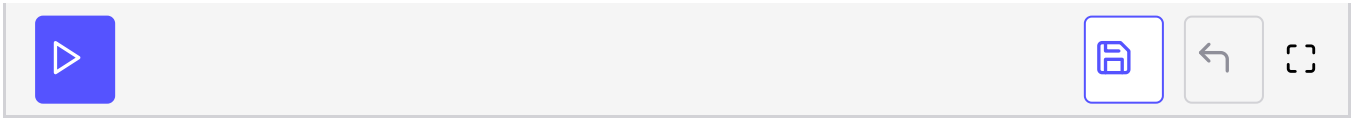
In this lesson, we will look into the use of std::promise and std::future in the scope of concurrency in C++.

WE'LL COVER THE FOLLOWING

Example 1

```
// promiseFuture.cpp

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

void product(std::promise<int>&& intPromise, int a, int b){
  intPromise.set_value(a*b);
}

struct Div{

  void operator() (std::promise<int>&& intPromise, int a, int b) const {
    intPromise.set_value(a/b);
  }

};

int main(){

  int a= 20;
  int b= 10;

  std::cout << std::endl;

  // define the promises
  std::promise<int> prodPromise;
  std::promise<int> divPromise;

  // get the futures
```

```
  std::future<int> prodResult= prodPromise.get_future();
  std::future<int> divResult= divPromise.get_future();

  // calculate the result in a separat thread
  std::thread prodThread(product,std::move(prodPromise),a,b);
  Div div;
  std::thread divThread(div,std::move(divPromise),a,b);

  // get the result
  std::cout << "20*10= " << prodResult.get() << std::endl;
  std::cout << "20/10= " << divResult.get() << std::endl;

  prodThread.join();

  divThread.join();

  std::cout << std::endl;

}
```

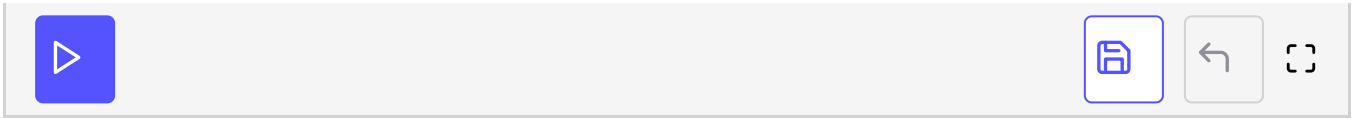


Explanation

- Thread `prodThread` (line 36) gets the function `product` (lines 8 -10), the `prodPromise` (line 32,) and the numbers `a` and `b`.
- To understand the arguments of `prodThread`, we must look at the signature of the function. `prodThread` needs a *callable* as its first argument. This is the previously mentioned function `product`.
- The function `product` requires a promise of the kind rvalue reference (`std::promise<int>&& intPromise`) and two numbers. Instead of an rvalue reference, the function `product` can also take the promise by value. These are the last three arguments of `prodThread`. `std::move` in line 36 creates an rvalue reference since a promise cannot be copied but moved.
- The rest of the process is simple. `divThread` (line 38) divides the two numbers `a` and `b`, and it uses the instance `div` of the class `Div` (lines 12 - 18). `div` is an instance of a function object.
- The future picks up the results by calling `prodResult.get()` and `divResult.get()`.

Example 2

```
// promiseFutureSynchronise.cpp


#include <future>
#include <iostream>
#include <utility>


void doTheWork(){
  std::cout << "Processing shared data." << std::endl;
}

void waitingForWork(std::future<void>&& fut){

    std::cout << "Worker: Waiting for work." << std::endl;
    fut.wait();
    doTheWork();
    std::cout << "Work done." << std::endl;

}

void setDataReady(std::promise<void>&& prom){

    std::cout << "Sender: Data is ready."  << std::endl;
    prom.set_value();

}

int main(){

  std::cout << std::endl;

  std::promise<void> sendReady;
  auto fut = sendReady.get_future();

  std::thread t1(waitingForWork, std::move(fut));
  std::thread t2(setDataReady, std::move(sendReady));

  t1.join();
  t2.join();

  std::cout << std::endl;

}
```



Explanation

That was easier than expected.

- Due to `sendReady` (line 32), we get a future `fut` (line 33).
- The promise communicates using its return value `void` (`std::promise<void> sendReady`) that it is only capable of sending

notifications. Both communication endpoints are moved into threads `t1` and `t2` (lines 35 and 36).

- The future waits using the call `fut.wait()` (line 15) for the notification of the promise: `prom.set_value()` (line 24).

The structure and the output of the program match the corresponding program in the section condition variable.

Test your knowledge on this topic with an exercise in the next lesson.