- Solution

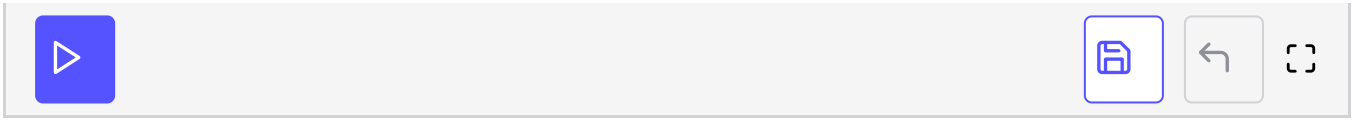
The solution to the task of the previous exercise will be explained in this lesson

WE'LL COVER THE FOLLOWING

Solution

```
// promiseFutureException.cpp

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

struct Div{
  void operator()(std::promise<int>&& intPromise, int a, int b){
    try{
      if ( b==0 ) throw std::runtime_error("illegal division by zero");
      intPromise.set_value(a/b);
    }
    catch ( ...){
      intPromise.set_exception(std::current_exception());
    }
  }
};

int main(){

  std::cout << std::endl;

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

  // get the futures
  std::future<int> divResult= divPromise.get_future();

  // calculate the result in a separat thread
  Div div;
  std::thread divThread(div, std::move(divPromise), 20, 0);

  // get the result
```

```
  try{
    std::cout << "20/0= " << divResult.get() << std::endl;
  }
  catch (std::runtime_error& e){
    std::cout << e.what() << std::endl;
  }

  divThread.join();

  std::cout << std::endl;

}
```



Explanation

- The promise can send a value (line 13) or an exception (line 16). The exception in this case is the current exception `std::current_exception()`.
- When we divide by zero in line 33, it triggers the exception in line 12.
- Lines 12 throws an exception of kind `std::runtime_error`.
- The catch-all clause in line 15 handles the exceptions and propagates it to the associated future (line 16).
- The associated future runs in the main program. The main program uses the future to get the result (line 37).
- If the `divResult.get()` call in line 37 fails, the main program handles the exception of the catch clause in line 39 and displays the error message for the associated promise `e.what()`.
- The program also works with valid denominators. In this case, the line 37 displays the result of the division.

For further information, read:

- std::promise
- std::future
- std::shared_future

Now that we have gone over **Several Tasks Simultaneously** in Embedded

Now that we have gone over **Several Tasks Simultaneously** in Embedded Programming with Modern C++.

For those of you interested in investigating this topic further, go to Modern C++ Concurrency in Practice: Get the most out of any machine.

Happy Learning!