Page1:

C3.2 : Promises and Futures¶
The promise - future communication channel¶

The methods for passing data to a thread we have discussed so far are both useful during thread construction: We can either pass arguments to the thread function using variadic templates or we can use a Lambda to capture arguments by value or by reference. The following example illustrates the use of these methods again:

```cpp
#include <iostream>
#include <thread>

void printMessage(std::string message)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(10)); // simulate work
    std::cout << "Thread 1: " << message << std::endl;
}

int main()
{
    // define message
    std::string message = "My Message";

    // start thread using variadic templates
    std::thread t1(printMessage, message);

    // start thread using a Lambda
    std::thread t2([message] {
        std::this_thread::sleep_for(std::chrono::milliseconds(10)); // simulate work
        std::cout << "Thread 2: " << message << std::endl;
    });

    // thread barrier
    t1.join();
    t2.join();

    return 0;
}
```

A drawback of these two approaches is that the information flows from the parent thread (`main`) to the worker threads (`t1` and `t2`). In

this section, we want to look at a way to pass data in the opposite direction - that is from the worker threads back to the parent thread. In order to achieve this, the threads need to adhere to a strict synchronization protocol. There is a such a mechanism available in the C++ standard that we can use for this purpose. This mechanism acts as a single-use channel between the threads. The sending end of the channel is called "promise" while the receiving end is called "future".
In the C++ standard, the class template std::promise provides a convenient way to store a value or an exception that will acquired asynchronously at a later time via a `std::future` object. Each `std::promise` object is meant to be used only a single time.

```
#include <iostream>
#include <thread>

void printMessage(std::string message)
{
   std::this_thread::sleep_for(std::chrono::milliseconds(10)); // simulate work
   std::cout << "Thread 1: " << message << std::endl;
}

int main()
{
   // define message
   std::string message = "My Message";

   // start thread using variadic templates
   std::thread t1(printMessage, message);

   // start thread using a Lambda
   std::thread t2([message] {
      std::this_thread::sleep_for(std::chrono::milliseconds(10)); // simulate work
      std::cout << "Thread 2: " << message << std::endl;
   });
```

```
    // thread barrier
    t1.join();
    t2.join();

    return 0;
}
```

Page 2:

In the following example, we want to declare a promise which allows for transmitting a string between two threads and modifying it in the process.

```cpp
#include <iostream>
#include <thread>
#include <future>

void modifyMessage(std::promise<std::string> && prms, std::string message)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(4000)); // simulate work
    std::string modifiedMessage = message + " has been modified";
    prms.set_value(modifiedMessage);
}

int main()
{
    // define message
    std::string messageToThread = "My Message";

    // create promise and future
    std::promise<std::string> prms;
    std::future<std::string> ftr = prms.get_future();

    // start thread and pass promise as argument
    std::thread t(modifyMessage, std::move(prms), messageToThread);

    // print original message to console
    std::cout << "Original message from main(): " << messageToThread << std::endl;

    // retrieve modified message via future and print to console
    std::string messageFromThread = ftr.get();
    std::cout << "Modified message from thread(): " << messageFromThread << std::endl;

    // thread barrier
    t.join();

    return 0;
}
```

After defining a message, we have to create a suitable promise that can take a string object. To obtain the corresponding future, we need to call the method `get_future()` on the promise. Promise and future are the two types of the communication channel we want to use to pass a string between threads. The communication channel set up in this manner can only pass a string.
We can now create a thread that takes a function and we will pass it the promise as an argument as well as the message to be modified. Promises can not be copied, because the promise-future concept is a two-point communication channel for one-time use. Therefore, we must pass the promise to the thread function using `std::move`. The thread will then, during its execution, use the promise to pass back the modified message.
The thread function takes the promise as an rvalue reference in accordance with move semantics. After waiting for several seconds, the message is modified and the method `set_value()` is called on the promise.
Back in the main thread, after starting the thread, the original message is printed to the console. Then, we start listening on the other end of the communication channel by calling the function `get()` on the future. This method will block until data is available - which happens as soon as set_value has been called on the promise (from the thread). If the result is movable (which is the case for `std::string`), it will be moved - otherwise it will be copied instead. After the data has been received (with a considerable delay), the modified message is printed to the console.

```
Original message from main(): My Message
Modified message from thread(): My Message has been modified
```

It is also possible that the worker value calls set_value on the promise before `get()` is called on the future. In this case, `get()` returns immediately without any delay. After `get()` has been called once, the future is no longer usable. This makes sense as the normal mode of data exchange between promise and future works with `std::move` - and in this case, the data is no longer available in the channel after the first call to `get()`. If `get()` is called a second time, an exception is thrown.

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

void modifyMessage(std::promise<std::string> && prms,
std::string message)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(4000));
// simulate work
    std::string modifiedMessage = message + " has been
modified";
    prms.set_value(modifiedMessage);
}

int main()
{
    // define message
    std::string messageToThread = "My Message";

    // create promise and future
    std::promise<std::string> prms;
    std::future<std::string> ftr = prms.get_future();

    // start thread and pass promise as argument
    std::thread t(modifyMessage, std::move(prms),
messageToThread);

    // print original message to console
    std::cout << "Original message from main(): " <<
messageToThread << std::endl;

    // retrieve modified message via future and print to console
    std::string messageFromThread = ftr.get();
    std::cout << "Modified message from thread(): " <<
messageFromThread << std::endl;

    // thread barrier
    t.join();
```

```
    return 0;
}
```

Page 3:

Quiz: get() vs. wait()¶

There are some situations where it might be interesting to separate the waiting for the content from the actual retrieving. Futures allow us to do that using the `wait()` function. This method will block until the future is ready. Once it returns, it is guaranteed that data is available and we can use `get()` to retrieve it without delay.
In addition to wait, the C++ standard also offers the method `wait_for`, which takes a time duration as an input and also waits for a result to become available. The method `wait_for()` will block either until the specified timeout duration has elapsed or the result becomes available - whichever comes first. The return value identifies the state of the result.
In the following example, please use the wait_for method to wait for the availability of a result for one second. After the time has passed (or the result is available) print the result to the console. Should the time be up without the result being available, print an error message to the console instead.

```
#include <iostream>
#include <thread>
#include <future>
#include <cmath>

void computeSqrt(std::promise<double> &&prms, int input)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(2000)); // simulate work
    double output = sqrt(input);
    prms.set_value(output);
}

int main()
{
    // define input data
    double inputData = 42.0;

    // create promise and future
    std::promise<double> prms;
    std::future<double> ftr = prms.get_future();

    // start thread and pass promise as argument
    std::thread t(computeSqrt, std::move(prms), inputData);

// Student task STARTS here
    // wait for result to become available
    auto status = ftr.wait_for(std::chrono::milliseconds(1000));
    if (status == std::future_status::ready) // result is ready
    {
        std::cout << "Result = " << ftr.get() << std::endl;
    }

    //  timeout has expired or function has not yet been started
    else if (status == std::future_status::timeout || status == std::future_status::deferred)
    {
        std::cout << "Result unavailable" << std::endl;
    }
// Student task ENDS here

    // thread barrier
    t.join();

    return 0;
}
```

```
#include <iostream>
#include <thread>
#include <future>
#include <cmath>

void computeSqrt(std::promise<double> &&prms, double
input)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(2000));
// simulate work
    double output = sqrt(input);
    prms.set_value(output);
```

```
}

int main()
{
    // define input data
    double inputData = 42.0;

    // create promise and future
    std::promise<double> prms;
    std::future<double> ftr = prms.get_future();

    // start thread and pass promise as argument
    std::thread t(computeSqrt, std::move(prms), inputData);

// Student task STARTS here
    // wait for result to become available
    auto status = ftr.wait_for(std::chrono::milliseconds(1000));
    if (status == std::future_status::ready) // result is ready
    {
        std::cout << "Result = " << ftr.get() << std::endl;
    }

    //  timeout has expired or function has not yet been started
    else if (status == std::future_status::timeout || status ==
std::future_status::deferred)
    {
        std::cout << "Result unavailable" << std::endl;
    }
// Student task ENDS here

    // thread barrier
    t.join();

    return 0;
}
```

Page 4:

Passing exceptions¶

The future-promise communication channel may also be used for passing exceptions. To do this, the worker thread simply sets an exception rather than a value in the promise. In the parent thread, the exception is then re-thrown once `get()` is called on the future. Let us take a look at the following example to see how this mechanism works:

```
#include <iostream>
#include <thread>
#include <future>
#include <cmath>
#include <memory>

void divideByNumber(std::promise<double> &&prms, double num, double denom)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(500)); // simulate work
    try
    {
        if (denom == 0)
            throw std::runtime_error("Exception from thread: Division by zero!");
        else
            prms.set_value(num / denom);
    }
    catch (...)
    {
        prms.set_exception(std::current_exception());
    }
}

int main()
{
    // create promise and future
    std::promise<double> prms;
    std::future<double> ftr = prms.get_future();

    // start thread and pass promise as argument
    double num = 42.0, denom = 0.0;
    std::thread t(divideByNumber, std::move(prms), num, denom);

    // retrieve result within try-catch-block
    try
    {
        double result = ftr.get();
        std::cout << "Result = " << result << std::endl;
    }
    catch (std::runtime_error e)
    {
        std::cout << e.what() << std::endl;
    }

    // thread barrier
    t.join();

    return 0;
}
```

In the thread function, we need to implement a try-catch block which can be set to catch a particular exception or - as in our case - to catch all exceptions. Instead of setting a value, we now want to throw a `std::exception` along with a customized error message. In the catch-block, we catch this exception and throw it to the parent thread using the promise with set_exception. The function `std::current_exception` allows us to easily retrieve the exception which has been thrown.
On the parent side, we now need to catch this exception. In order to do this, we can use a try-block around the call to `get()`. We can set the catch-block to catch all exceptions or - as in this example - we could also catch a particular one such as the standard exception. Calling the method `what()` on the exception allows us to retrieve the message from the exception - which is the one defined on the promise side of the communication channel.
When we run the program, we can see that the exception is being thrown in the worker thread with the main thread printing the corresponding error message to the console.
So a promise future pair can be used to pass either values or exceptions between threads.

```
#include <iostream>
#include <thread>
#include <future>
#include <cmath>
#include <memory>

void divideByNumber(std::promise<double> &&prms, double
num, double denom)
{

std::this_thread::sleep_for(std::chrono::milliseconds(500)); //
simulate work
    try
    {
        if (denom == 0)
```

```
            throw std::runtime_error("Exception from thread:
Division by zero!");
        else
            prms.set_value(num / denom);
    }
    catch (...)
    {
        prms.set_exception(std::current_exception());
    }
}

int main()
{
    // create promise and future
    std::promise<double> prms;
    std::future<double> ftr = prms.get_future();

    // start thread and pass promise as argument
    double num = 42.0, denom = 0.0;
    std::thread t(divideByNumber, std::move(prms), num,
denom);

    // retrieve result within try-catch-block
    try
    {
        double result = ftr.get();
        std::cout << "Result = " << result << std::endl;
    }
    catch (std::runtime_error e)
    {
        std::cout << e.what() << std::endl;
    }

    // thread barrier
    t.join();

    return 0;
}
```