CS 6650 – Programming Assignment #1
Getting your hands on C++11
The basics, multi-threading, and socket programming (10 pts)

Throughout this assignment you will be writing multiple simple C++11 programs to get familiar with C++11 to be prepared for the upcoming assignments. You are expected to learn C++11 by yourself, but this assignment will help you get on track. You will be practicing how to use simple to advanced C++11 features by reading and writing multiple short programs.

1. **Read through this assignment specification and understand the entire task first.**
2. **Before jumping into writing code, you must read and understand all the code and comments relevant to each part.**
3. **You only need to write code in part_1_your_task.cpp, part_2_your_task.cpp, and part_3_your_task.cpp and submit these three files.**
4. **You will find relevant code to each part in part_1_main.cpp part_2_main.cpp, part_3_client.cpp and part_3_server.cpp so read them side by side with part_X_your_task.cpp files. There are other class files relevant to each part so you must study them as well.**
5. **You can compile all three parts with a "make" command. Executables part_1, part_2, part_3_client, and part_3_server will be created.**
6. **The part_x_main.cpp functions already include test cases where you can check the correctness of your code to a certain degree. Since the grading will be done with different test cases, feel free to tweak the main files and test your code differently.**

1. Development Environment

Servers and VPN requirements. You should use Khoury Linux servers (i.e., linux-[071-085].khoury.northeastern.edu). If you are accessing the machines from an off-campus location, you should first log on to the Northeastern VPN (search for VPN on Northeastern ITS webpage for detailed information). If you are on campus you can directly access the servers without the VPN.

ID/Password for server access. These servers are accessible through `ssh` using the **Khoury user id and password** (NOT NORTHEASTERN ID/PASSWORD). If you do not have the account, apply here: https://my.khoury.northeastern.edu/account/apply. If you have trouble getting the account, you can contact Khoury-systems@northeastern.edu for help. For this assignment, you will need to use at most two servers.

Access the server using the following command (you can replace 075 with numbers ranging from 071 to 085 to access different servers):

```
ssh [YOUR KHOURY ID]@linux-075.khoury.northeastern.edu
```

To copy files to and from the servers, you can use the `scp` command: https://snapshooter.com/learn/linux/copy-files-scp.

Using the servers. The recommended way is to use command line tools and vim/emacs/nano like text editors for programming, but there are ways to connect to the server through VS-code-like tools (https://code.visualstudio.com/docs/remote/ssh). You are free to choose your own tools, but **all code must compile in the Khoury Linux servers using gcc/g++ and GNU Make (Makefile) and run in the Khoury Linux servers. Grading will be done on these servers.** There have been reports over the past few years that C++ programs written in Apple/Mac machines do not run properly, so develop and test your program in the Khoury Linux servers.

If you are not familiar with the Linux command line environment here is a helpful online resource:

- "The Linux Command Line," 5th edition, William Shotts (http://linuxcommand.org/tlcl.php).

2. Part 0: Basics (study it yourself; no submission required)

2.1. Basic C++

Here are some relevant references you can study:

- Learn C++: https://www.w3schools.com/cpp/default.asp
- Book accessible online through Northeastern library: "The C++ Programming Language," 4th Edition, Bjarne Stroustrup, Addison-Wesley.

Write a hello world program and test basic features of C++. Try out statements including if-else, for, while, and switch. Write a class and member functions.

Learn how to organize files when using classes: https://www.learncpp.com/cpp-tutorial/classes-and-header-files/

- Usually, each class definition (e.g, class Example) goes into a corresponding header file (e.g., Example.h) and the class member functions go into the source file (e.g., Example.cpp).
- The main function goes into the "main.cpp" file.

2.2. Compilation and GNU Make

Throughout this course, our default compiler will be g++ which is pre-installed in the server and GNU Make in the command line environment. Familiarize yourself with them. You are given a Makefile for this assignment and you will have a chance to learn and use them.

- How to use g++ compiler (our default): https://www.geeksforgeeks.org/linux-unix/compiling-with-g-plus-plus/
- Learn and understand how to compile multiple C++ files into a program using Makefile: https://www.geeksforgeeks.org/cpp/makefile-in-c-and-its-applications/
- GNU Make tutorial: https://makefiletutorial.com/

3. Part 1: Pointers and standard template library (STL) (2 pt)

From this part you will start to practice the C++11 features. C++11 is one of the modern C++ standards and there exist later versions, such as C++14, C++17, and C++20. However, this course will focus on C++11 as this is the most widely used version for system developments in the industry.

Here are some relevant references you can study about C++11:

- About C++11: https://www.geeksforgeeks.org/cpp/cpp-11-standard/
- Book accessible online through Northeastern library: "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14," 1st Edition, Scott Meyers, O'Reilly.

Note that to use C++11, you should use "-std=c++11" option for compilations:
e.g., `g++ -std=c++11 main.cpp -o main`

When you build the code using "`make`", an executable binary "part_1" will be created. Read through the Makefile and understand what it does. You can execute the compiled code by typing "`./part_1`". In this part, you will write code only in part_1_your_task.cpp file.

3.1. Part 1-1 Unique pointers and Part 1-2 shared pointers

Here are some relevant references you can study for this subparts:

- Unique pointers: https://en.cppreference.com/w/cpp/memory/unique_ptr.html
- Shared pointers: https://en.cppreference.com/w/cpp/memory/shared_ptr.html
- Move: https://en.cppreference.com/w/cpp/utility/move.html

Part 1-1 and part 1-2 involve `Book.h`, `Book.cpp`, `part_1_main.cpp`, and `part_1_your_task.cpp` files. Read through the code and understand what they do.

Part 1-1 (0.5 pt: no partial points)

Goal: You will compare and understand how regular C++ pointers and unique_ptrs work.

- Part_1_main.cpp includes part_1_1_unique_ptr_base(). Read through what it does.
- Part_1_your_task.cpp includes part_1_1_unique_ptr(). Add a single line, or just one statement ending with ";", which makes **part_1_1_unique_ptr() to output exactly the same as part_1_1_unique_ptr_base()**.

Part 1-2 (0.5 pt: no partial points)

Goal: You will compare and understand how regular C++ pointers and shared_ptrs work.

- Part_1_main.cpp includes part_1_1_shared_ptr_base(). Read through what it does.
- Part_1_your_task.cpp includes part_1_1_ shared_ptr(). Add a single line, or just one statement ending with ";", which makes **part_1_1_ shared_ptr() to output exactly the same as part_1_1_ shared_ptr_base()**.

3.2. Part 1-3 std::list in Standard Template Library (1 pt: no partial points)

Goal: In this subpart, you will practice how to use STL (e.g., list) with unique_ptr.

Here, `Library.h` and `Library.cpp` will be used in addition to the files used in the previous subparts. The Library class consists of a list of Books. Books can be added to the library, listed, and checked out. Library.cpp implements the Size and ListBooks functions. Read and understand the `Library.h/cpp` files.

You should implementation AddBook (0.5 pt: no partial points) and CheckoutBook (0.5 pt: no partial points) functions of the Library class in part_1_your_task.cpp. Read and understand corresponding skeleton functions. Implement the body of these functions following the description in the comments.

Here are some references for the list:

- std::list: https://en.cppreference.com/w/cpp/container/list.html

When properly implemented, the output for Part 1-3 should be:

```
[Book created]: "C++ Book" (id: 0)
Book added to library
[Book created]: "C++ Book" (id: 1)
Book added to library
[Book created]: "C++ Book" (id: 2)
Book added to library

Library contains 3 books:
"C++ Book" (id: 0)
"C++ Book" (id: 1)
"C++ Book" (id: 2)

Book checked out: "C++ Book" (id: 0)

Library contains 2 books:
"C++ Book" (id: 1)
"C++ Book" (id: 2)

[Book destroyed]: "C++ Book" (id: 0)
[Book destroyed]: "C++ Book" (id: 1)
[Book destroyed]: "C++ Book" (id: 2)
```

4. Part 2: Multi-threading (3 pt)

Here is a relevant reference you can study:

- Book accessible online through Northeastern library: “C++ Concurrency in Action,” 2nd Edition, Williams Anthony, Manning Publications.

4.1. Part 2-1 Running threads concurrently, passing an argument to threads, and getting execution results from threads (1 pt: no partial points)

Here are some relevant references you can study:

- Thread constructor: https://en.cppreference.com/w/cpp/thread/thread/thread.html
- Thread join: https://en.cppreference.com/w/cpp/thread/thread/join.html
- std::vector: https://en.cppreference.com/w/cpp/container/vector.html

Goal: In this subpart, you will learn how to run arbitrary number of threads concurrently.

If you use a loop to generate an arbitrary number of threads, it is important to keep relevant instances (e.g., std::thread and class object instance if any) to the running threads alive by storing them in a known place (e.g., in a list or a vector).

ThreadSleeper.cpp/h defines a class which represents a thread. The ThreadBody function of this class should be given as a parameter to the std::thread to create a thread. This function simply takes an id, stores it as the thread instance ID, records the start time of the thread, sleeps for 3 seconds and records the end time of the thread.

Your task is to complete part_2_1_work_with_threads function in part_2_your_task.cpp. This function should create the ThreadSleeper instances as many as the parameter "count" and store them into the parameter "sleeperVector". Using these instances, you should create a thread per each ThreadSleeper instance and make the threads run concurrently. The function should then call join() in a loop to wait for every thread to terminate.

When you implement your part and execute ./part_2 binary, you will see the start and end time of each thread; **all threads should start around 0 second mark and end around 3 second marks**. Retrieving results from each thread execution is handled by the given code in part 2-2 of `part_2_main.cpp`.

4.2. Part 2-2 Locks, thread pool and condition variables (1 pt: no partial points)

Here are some relevant references you can study:

- Mutex: https://en.cppreference.com/w/cpp/thread/unique_lock/mutex.html
- Unique lock: https://en.cppreference.com/w/cpp/thread/unique_lock.html
- Condition variables: https://en.cppreference.com/w/cpp/thread/condition_variable.html
- std::queue: https://en.cppreference.com/w/cpp/container/queue.html

Goal: In this subpart, you will learn how to create a thread pool and send jobs to the pool. You will practice how to use locks and condition variables as well.

Part_2_2_locks_and_thread_pool in Part_2_main.cpp creates four threads which will be part of the thread pool. The thread takes an id, reference to the condition variable, reference to the mutex, and pointer to the job queue. Then it enqueues a few work requests to the job queue and sends signals to the thread pool. Next, it enqueues as many quit requests as the number of threads to terminate all of them. Finally, it prints the entire duration of the execution.

Your task is to create the body of this worker thread in the pool in part_2_2_thread_in_pool function in part_2_your_task.cpp. Thread in the thread pool should run an infinite loop---wait for incoming work requests ("REQ_WORK"), retrieve one request from the queue upon receiving the wake up call, process the request (i.e., call part_2_2_process_work), and repeat---and should not terminate unless it is given a quit ("REQ_QUIT") command. You should properly use the lock, condition variable, and job queue to retrieve one incoming request and process the request concurrently with other threads if there are many queued requests. When you receive REQ_WORK you should call part_2_2_process_work function with the given id which will make the thread to work (simulated with a sleep) for 1 second. When you receive REQ_QUIT you can simply break out of the loop and end the thread.

Given 4 threads in the pool and 12 work items, the entire execution time should be roughly 3 seconds, which will be printed when you run ./part_2 binary.

4.3. Part 2-3 Making threads synchronously send and receive data with future and promise (1 pt: no partial points)

Here are some relevant references you can study:

- Future: https://en.cppreference.com/w/cpp/thread/future.html
- Promise: https://en.cppreference.com/w/cpp/thread/promise.html

Goal: Learn how to use promise and future to synchronously send and receive data.

Promise and future allows threads to wait for specific values to be given by another thread which can be useful for two threads to send and receive data synchronously. You will practice using them.

In this subpart, you will implement a simple thread that waits to receive an integer value using future, multiplies it by the multiplier given as an argument when the thread was initialized, and sends the product through the promise to another thread that holds the corresponding future. Part_2_3_promise_and_future_thread function in part_2_your_task.cpp should implement these features.

Part_2_main.cpp chains multiple threads with three sets of promise and future to multiply three values: 2 x 4 x 8. Your thread implementation should yield the final outcome of 64 for the execution of ./part_2 binary.

5. Part 3: Networking (5pt)

In this part, you will write client and server programs that communicate over the network. You will use two machines to make them send and receive messages. You can find out the ip addresses of the machines using the following command: `hostname -i`

You cannot use an arbitrary port number for communication, and you are limited to the range 10000-65535.

5.1. Instructions for running the client and server programs

For this part part_3_client.cpp implements a TCP client program and part_3_server.cpp implements a TCP server program. Your task is to use the send and recv functions to exchange messages between them. You will need to use two terminals. You can test the program by

1. First running the server on one machine (say its ip is 123.456.789.123): e.g., `./part_3_server 11111`
2. And then running the client program with the ip and port information of the server program on another machine: e.g., `./part_3_client 123.456.789.123 11111`

For convenience, you can also test the client and server programs locally on a single machine. In this case, you can use the ip, "127.0.0.1" which is the address for the localhost.

E.g., in one terminal run,

```
./part_3_server 11111
```

On another terminal run,

```
./part_3_client 127.0.0.1 11111
```

5.2. What the client and server programs do

Read the server and client code to see how network connections are initialized and how the initial "Hi server!" and "Hi client!" messages are exchanged. When you run the server and then the client, these messages will be exchanged and displayed.

The code section Part 3 is incomplete and you will have to complete it. The code as is will print three sets of sent and received messages. However, the byte stream to be sent and received are not properly generated and the send and receive over the network for these messages are not implemented (see 5.4. for details).

The messages to be exchanged are class Request and class Response (in Message.h/cpp). Your task for this part is to implement the Marshal and Unmarshal functions for these classes following the message formats in the comment in part_3_your_task.cpp. Then implement the missing three functions implementing send/recv/marshal/unmarshal features that will be called by the client and server programs.

5.3. Part 3-1: Marshalling and unmarshalling messages (2 pt: 0.5 pt for each marshal/unmarshal function; no partial points)

Here are some relevant references you can study:

- C-string: https://en.cppreference.com/w/cpp/header/cstring.html
- Memcpy: https://en.cppreference.com/w/cpp/string/byte/memcpy
- Htonl and ntohl: https://linux.die.net/man/3/htonl
- Endianness: https://en.wikipedia.org/wiki/Endianness

Goal: Learn how to marshal and unmarshal an object.

When sending a message over the network, you are sending a stream of bytes. Thus, you should encode your data into a byte array (i.e., marshalling). When receiving a message, you should do the inverse and decode the data from a byte array (i.e., unmarshalling).

C-strings (e.g., char string[32], "this is a string", char *) are character arrays which are the same as byte arrays. Send and recv APIs for sockets use these byte arrays for communications.

When sending integer-like primitive numeric data types, you will need to marshal the data into network format (Big-endian) before sending over the network and unmarshal the data into host format (Little Endian for x86 architecture) after receiving it. This is to allow programs running in different architectures to work compatibly with the networked programs.

Your task is to complete the marshal and unmarshal functions for class Request and class Response in part_3_your_task.cpp. **Carefully read the Message.h/cpp files and then read through the comments in part_3_your_task.cpp to implement two pairs of marshal/unmarshal functions.**

- The marshal function takes a buffer and fills in the buffer with network formatted values of the class Request/Response.
- Unmarshal function takes a buffer prefilled with network formatted data and converts the data to member variables of class Request/Response.

For testing, you can create a marshalled byte stream using the marshal function from one instance (e.g., of class Request), and unmarshal the buffer in the same type of different instance (i.e., in a different class Request instance). Then print the values of these two instances using the Print (e.g,. PrintRequest) functions and compare if the outputs are the same.

5.4. Part 3-2: Implementing missing communications (3 pt: 1 pt for each function; no partial points)

Here is a relevant reference you can study:

- Beej's guide to networking (socket and relevant APIs): https://beej.us/guide/bgnet/html/split/

Goal: Use marshal and unmarshal functions with send and recv to implement communications.

Once marshalling and unmarshalling implementations are complete you need to properly use them with send and recv functions to communicate over the network.

Part_3_main.cpp contains skeleton functions that are called in the part_3_client/server.cpp files (i.e., one function for client and two functions for server). They are simple functions that marshal/unmarshal/send/receive data. Understand how they fit into part_3_client/server.cpp to enable clients and servers to send and receive requests and responses.

In this part, the client sends two numbers to the server with some other meta information (i.e., request id, user id, requester name). The server computes the sum of two numbers and sends it back as a response with corresponding meta information (i.e, original request id, original user id, and a random response id). Filling

in the data happens in the given code, but you should handle the rest. Read through the comments in part_3_your_task.cpp to implement these functions.

The client and server use TCP, a stream-based protocol. What this means is, when you send N bytes you must receive all N bytes. If this alignment is broken, bad things will happen (e.g., get garbage data). So make sure that if one side sends N bytes the other side receives the same N bytes.

The code corresponding to part 3-2 runs a loop that sends and receives or receives and sends requests and responses. The request sent from the client should be received from the server (i.e., print the same contents in the client and the server) and the response sent from the server should be received by the client (i.e., print the same contents in the client and the server). To test if your code is correct, compare the outputs of the server program and the client program.

6. Due date and what to submit

The assignment is due at 11:59PM on Monday, September 22, 2025. See the syllabus for late policy.

You should submit your assignment through Canvas where you downloaded this file. Package part_1_your_task.cpp, part_2_your_task.cpp, and part_3_your_task.cpp, in a single zip file and upload it to the Canvas page. Do not include any other given files.