Lab2: the TCP receiver

0 Collaboration Policy

The programming assignments must be your own work: You must write all the code you hand in for the programming assignments, except for the code that we give you as part of the assignment. Please do not copy-and-paste code from Stack Overflow, GitHub, or other sources. If you base your own code on examples you find on the Web or elsewhere, cite the URL in a comment in your submitted source code.

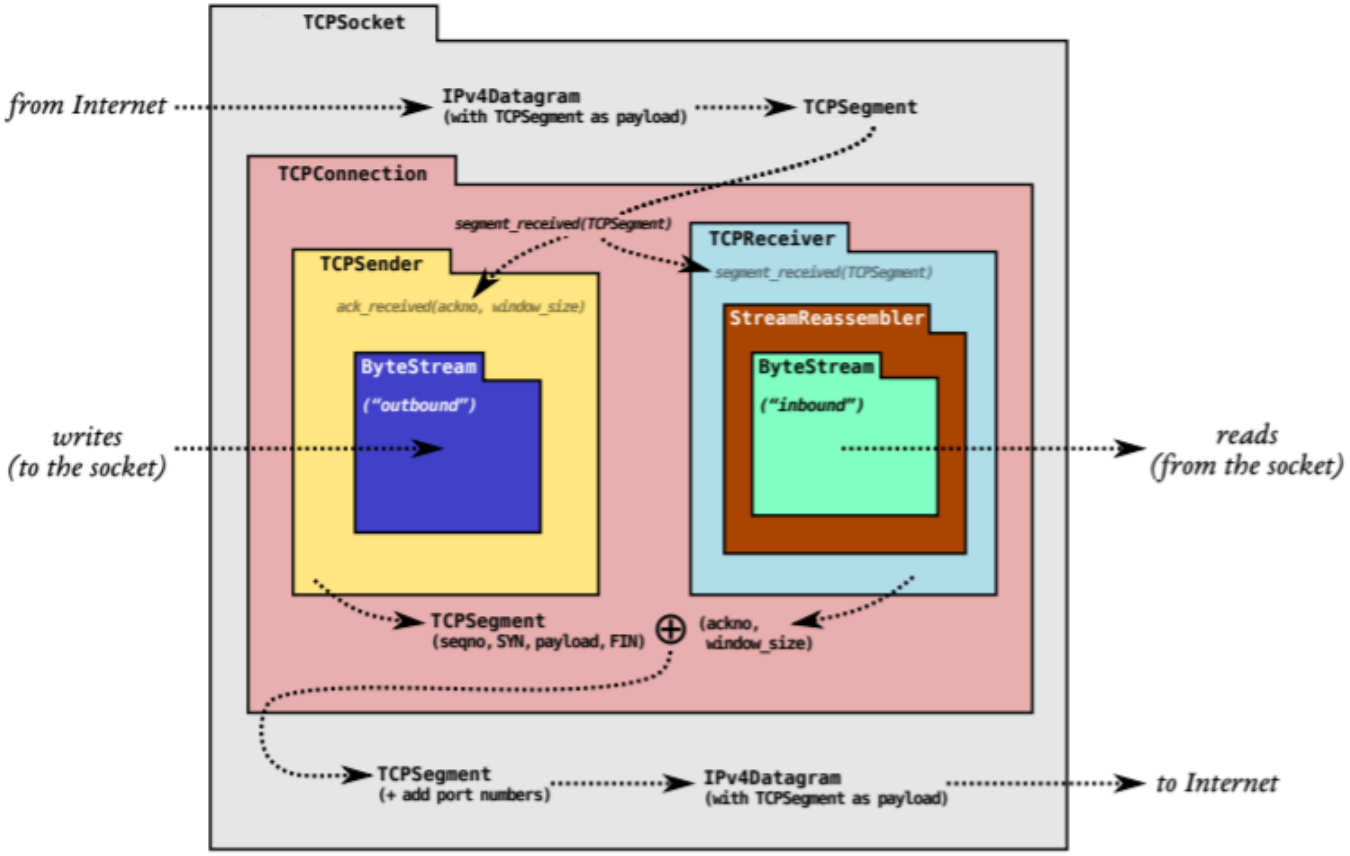
Working with others: You may not show your code to anyone else, look at anyone else's code, or look at solutions from previous years. You may discuss the assignments with other students, but do not copy anybody's code. If you discuss an assignment with another student, including at the lab session, please name them in your writeup. Please refer to the course administrative handout for more details, and ask on QQ group if anything is unclear.

1 Overview

In Lab1, you implemented the abstraction of a *flow-controlled byte stream* (**ByteStream**), and created a **StreamReassembler** that accepts a sequence of substrings, all excerpted from the same byte stream, and reassembles them back into the original stream.

These modules will prove useful in your TCP implementation, but nothing in them was specific to the details of the Transmission Control Protocol. That changes now. In Lab2, you will implement the TCPReceiver, the part of a TCP implementation that handles the incoming byte stream. The **TCPReceiver** translates between incoming **TCP segments** (the payloads of datagrams carried over the Internet) and the incoming **byte stream**.

Here's the diagram again from the last lab. The **TCPReceiver** receives segments from the Internet (via the **segment received**() method) and turns them into calls to your **StreamReassembler**, which eventually writes to the incoming **ByteStream**. Applications read from this **ByteStream**, just as you did in Lab0 by reading from the **TCPSocket**.



In addition to writing to the incoming stream, the TCPReceiver is responsible for telling the sender two things:

1. the index of the "first unassembled" byte, which is called the "acknowledgment number" or "**ackno**." This is the first byte that the receiver needs from the sender.
2. the distance between the "first unassembled" index and the "first unacceptable" index. This is called the "**window size**".

Together, the **ackno** and **window size** describe describes the receiver's window: a **range of indexes** that the TCP sender is allowed to send. Using the window, the receiver can control the flow of incoming data, making the sender limit how much it sends until the receiver is ready for more. We sometimes refer to the ackno as the "left edge" of the window (smallest index the **TCPReceiver** is interested in), and the ackno + window size as the "right edge" (just beyond the largest index the **TCPReceiver** is interested in).

You've already done most of the algorithmic work involved in implementing the **TCPReceiver** when you wrote the StreamReassembler and ByteStream; this lab is about wiring those general classes up to the details of TCP. The hardest part will involve thinking about how TCP will represent each byte's place in the stream—known as a "sequence number."

2 Getting started

Your implementation of a **TCPReceiver** will use the same Sponge library that you used in Lab0 and Lab1, with additional classes and tests. To get started:

1. Make sure you have committed all your solutions to Lab0 and Lab1. Now clone the Lab2 repository to your local workspace. Then make your changes to files in Lab0 and Lab1 be synchronized to Lab2 project. Specifically, you modified `apps/webget.cc` in Lab0 and `libsponge/byte_stream.cc`,

`libsponge/byte_stream.hh`, `libsponge/stream_reassembler.cc` ,`libsponge/stream_reassembler.hh` in Lab1.

2. Just as Lab1, run `mkdir build`, `cd build` ,`cmake ..` and `make` (you can run, e.g., `make -j4` to use four processors when compiling) one by one.
3. Outside the `build` directory, open and start editing the `writeups/lab2.md` file. This is the template for your lab writeup and will be included in your submission.

3 Lab 2: The TCP Receiver

TCP is a protocol that reliably conveys a pair of flow-controlled byte streams (one in each direction) over unreliable datagrams. Two parties participate in the TCP connection, and each party acts as both "sender" (of its own outgoing byte-stream) and "receiver" (of an incoming byte-stream) at the same time. The two parties are called the "endpoints" of the connection, or the "peers."

This week, you'll implement the "receiver" part of TCP, responsible for receiving TCP segments (the actual datagram payloads), reassembling the byte stream (including its ending, when that occurs), and determining that signals that should be sent back to the sender for acknowledgment and flow control.

> Why am I doing this? These signals are crucial to TCP's ability to provide the service of a flow-controlled, reliable byte stream over an unreliable datagram network. In TCP, **acknowledgment** means, "What's the index of the next byte that the receiver needs so it can reassemble more of the ByteStream?" This tells the sender what bytes it needs to send or resend. **Flow control** means, "What range of indices is the receiver interested and willing to receive?" (usually as a function of its remaining capacity). This tells the sender how much it's allowed to send.

3.1 Translating between 64-bit indexes and 32-bit seqnos

As a warmup, we'll need to implement TCP's way of representing indexes. Last week you created a StreamReassembler that reassembles substrings where each individual byte has a 64-bit **stream index**, with the first byte in the stream always having index zero. A 64-bit index is big enough that we can treat it as never overflowing.1 In the TCP headers, however, space is precious, and each byte's index in the stream is represented not with a 64-bit index but with a 32-bit "sequence number," or "seqno." This adds three complexities:

1. **Your implementation needs to plan for 32-bit integers to wrap around**. Streams in TCP can be arbitrarily long—there's no limit to the length of a ByteStream that can be sent over TCP. But 2^{32} bytes is only 4 GiB, which is not so big. Once a 32-bit sequence number counts up to $2^{32}-1$, the next byte in the stream will have the sequence number zero.
2. **TCP sequence numbers start at a random value**: To improve security and avoid getting confused by old segments belonging to earlier connections between the same endpoints, TCP tries to make sure sequence numbers can't be guessed and are unlikely to repeat. So the sequence numbers for a stream don't start at zero. The first sequence number in the stream is a random 32-bit number called the Initial Sequence Number (ISN). This is the sequence number that represents the SYN (beginning of stream). The rest of the sequence numbers behave normally after that: the first byte of data will have the sequence number of the ISN+1 (mod 2^{32}), the second byte will have the ISN+2 (mod 2^{32}), etc.
3. **The logical beginning and ending each occupy one sequence number**: In addition to ensuring the receipt of all bytes of data, TCP makes sure that the beginning and ending of the stream are received reliably. Thus, in TCP the SYN (beginning-ofstream) and FIN (end-of-stream) control flags are assigned

sequence numbers. Each of these occupies *one* sequence number. (The sequence number occupied by the SYN flag is the ISN.) Each byte of data in the stream also occupies one sequence number. Keep in mind that SYN and FIN aren't part of the stream itself and aren't "bytes"—they represent the beginning and ending of the byte stream itself.

These sequence numbers (**seqnos**) are transmitted in the header of each TCP segment. (And, again, there are two streams—one in each direction. Each stream has separate sequence numbers and a different random ISN.) It's also sometimes helpful to talk about the concept of an "**absolute sequence number**" (which always starts at zero and doesn't wrap), and about a "**stream index**" (what you've already been using with your StreamReassembler: an index for each byte in the stream, starting at zero).

To make these distinctions concrete, consider the byte stream containing just the three-letter string 'cat'. If the SYN happened to have seqno $2^{32} - 2$, then the seqnos, absolute seqnos, and stream indices of each byte are:

element	SYN	c	a	t	FIN
seqno	$2^{32} - 2$	$2^{32} - 1$	0	1	2
absolute seqno	0	1	2	3	4
stream index		0	1	2	

The figure shows the three different types of indexing involved in TCP:

Sequence Numbers	Absolute Sequence Numbers	Stream Indices
Start at the ISN	Start at 0	Start at 0
Include SYN/FIN	Include SYN/FIN	Omit SYN/FIN
32 bits, wrapping	64 bits, non-wrapping	64 bits, non-wrapping
"seqno"	"absolute seqno"	"stream index"s

Converting between absolute sequence numbers and stream indices is easy enough—just add or subtract one. Unfortunately, converting between sequence numbers and absolute sequence numbers is a bit harder, and confusing the two can produce tricky bugs. To prevent these bugs systematically, we'll represent sequence numbers with a custom type: **WrappingInt32**, and write the conversions between it and absolute sequence numbers (represented with **uint64_t**). WrappingInt32 is an example of a *wrapper type*: a type that contains an inner type (in this case uint32_t) but provides a different set of functions/operators.

We've defined the type for you and provided some helper functions (see wrapping_integers.hh), but you'll implement the conversions in wrapping_integers.cc:

1. ```
   WrappingInt32 wrap(uint64_t n, WrappingInt32 isn)
   ```

   Convert absolute seqno → seqno. Given an absolute sequence number (*n*) and an Initial Sequence Number (*isn*), produce the (relative) sequence number for n.
   ```

2. `uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint)`

   Convert seqno → absolute seqno. Given a sequence number (*n*), the Initial Sequence Number (*isn*), and an absolute *checkpoint* sequence number, compute the absolute sequence number that corresponds to n that is **closest to the checkpoint**.

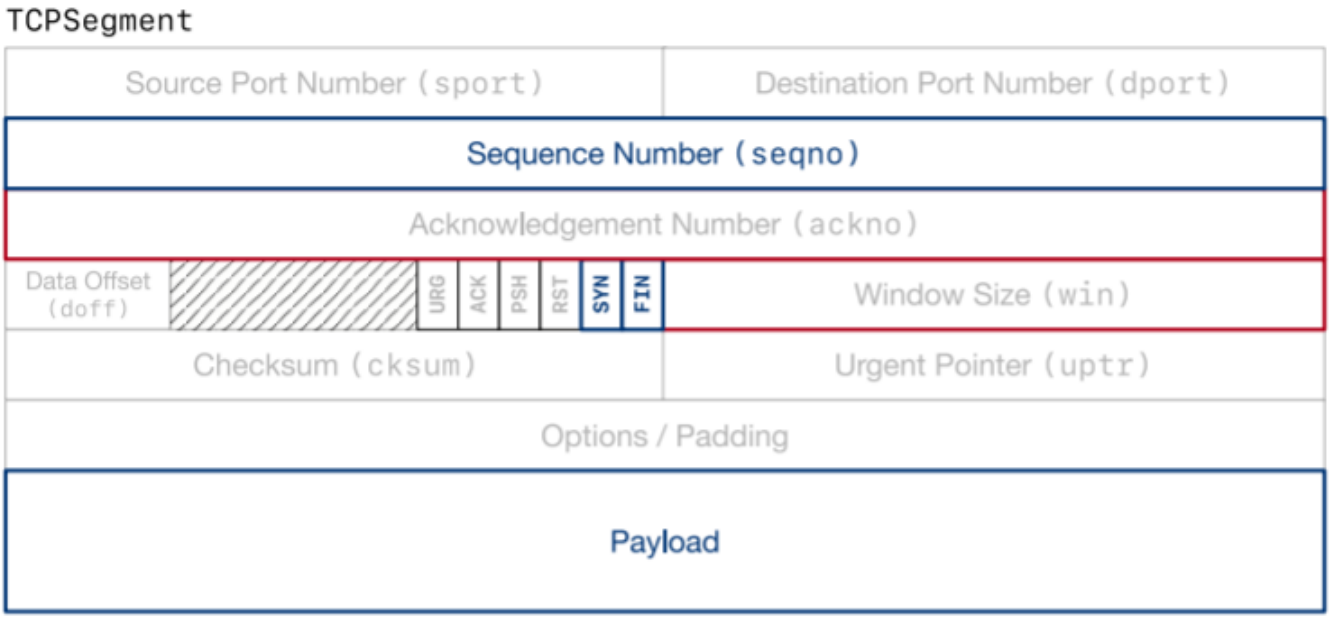
   Note: A **checkpoint** is required because any given seqno corresponds to *many* absolute seqnos. E.g. with an ISN of zero, the seqno "17" corresponds to the absolute seqno of 17, but also 2^{32} + 17, or 2^{33} + 17, or 2^{34} + 17, etc. The checkpoint helps resolve the ambiguity: it's an absolute seqno that the user of this class knows is "in the ballpark" of the correct answer. Here, "in the ballpark" can mean any 64-bit number that's within 2^{31} of the right answer. In your TCP implementation, you'll use the index of the last reassembled byte as the checkpoint.

   > Hint: The cleanest/easiest implementation will use the helper functions provided in wrapping integers.hh. The wrap/unwrap operations should preserve offsets—two seqnos that differ by 17 will correspond to two absolute seqnos that also differ by 17.

You can test your implementation by running the **WrappingInt32** tests. From the build directory, run `ctest -R wrap` .

3.2 Implementing the TCP receiver

Congratulations on getting the wrapping and unwrapping logic right! We'd shake your hand if we could. In the rest of this lab, you'll be implementing the **TCPReceiver**. It will (1) receive segments from its peer, (2) reassemble the **ByteStream** using your **StreamReassembler**, and (3) calculate the acknowledgment number (ackno) and the window size. The ackno and window size will eventually be transmitted back to the peer in an outgoing segment. First, please review the format of a TCP segment. This is the message that the two endpoints send each other; it is the payload of the lower-level datagrams. The non-grayed-out fields represent the information that's of interest in this lab: the sequence number, the payload, and the SYN and FIN flags. These are the fields that are written by the sender, and read and acted on by the receiver.



The **TCPSegment** class represents this message in C++. Please review the documentation for **TCPSegment** and **TCPHeader**. (Please refer to https://cs144.github.io/doc/lab2/annotated.html for details.) You may be interested in the length in sequence space() method, which calculates how many sequence numbers a segment occupies (including the fact that the SYN and FIN flags each occupy one sequence number, along with each byte of the payload).

Next, let's talk about the interface that your **TCPReceiver** will provide:

```
// Construct a `TCPReceiver` that will store up to `capacity` bytes
TCPReceiver(const size_t capacity); // implemented for you in .hh file
// Handle an inbound TCP segment
void segment_received(const TCPSegment &seg);
// The ackno that should be sent to the peer
//
// returns empty if no SYN has been received
//
// This is the beginning of the receiver's window, or in other words,
// the sequence number of the first byte in the stream
// that the receiver hasn't received.
std::optional<WrappingInt32> ackno() const;
// The window size that should be sent to the peer
//
// Formally: this is the size of the window of acceptable indices
// that the receiver is willing to accept. It's the distance between
// the ``first unassembled'' and the ``first unacceptable'' index.
//
// In other words: it's the capacity minus the number of bytes that the
// TCPReceiver is holding in the byte stream.
size_t window_size() const;
// number of bytes stored but not yet reassembled
size_t unassembled_bytes() const; // implemented for you in .hh file
// Access the reassembled byte stream
ByteStream &stream_out(); // implemented for you in .hh file
```

The **TCPReceiver** is built around your **StreamReassembler**. We've implemented the constructor and the **unassembled_bytes** and **stream_out** methods for you in the **.hh** file. Here's what you'll have to do for the others:

3.2.1 segment_received()

This is the main workhorse method **TCPReceiver::segment_received()** will be called each time a new segment is received from the peer.

This method needs to:

- **Set the Initial Sequence Number if necessary**. The sequence number of the firstarriving segment that has the SYN flag set is the initial sequence number. You'll want to keep track of that in order to keep converting between 32-bit wrapped seqnos/acknos and their absolute equivalents. (Note that the SYN

flag is just one flag in the header. The same segment could also carry data and could even have the FIN flag set.)

- **Push any data, or end-of-stream marker, to the StreamReassembler**. If the FIN flag is set in a **TCPSegment**'s header, that means that the last byte of the payload is the last byte of the entire stream. Remember that the **StreamReassembler** expects stream indexes starting at zero; you will have to unwrap the seqnos to produce these.

3.2.2 ackno()

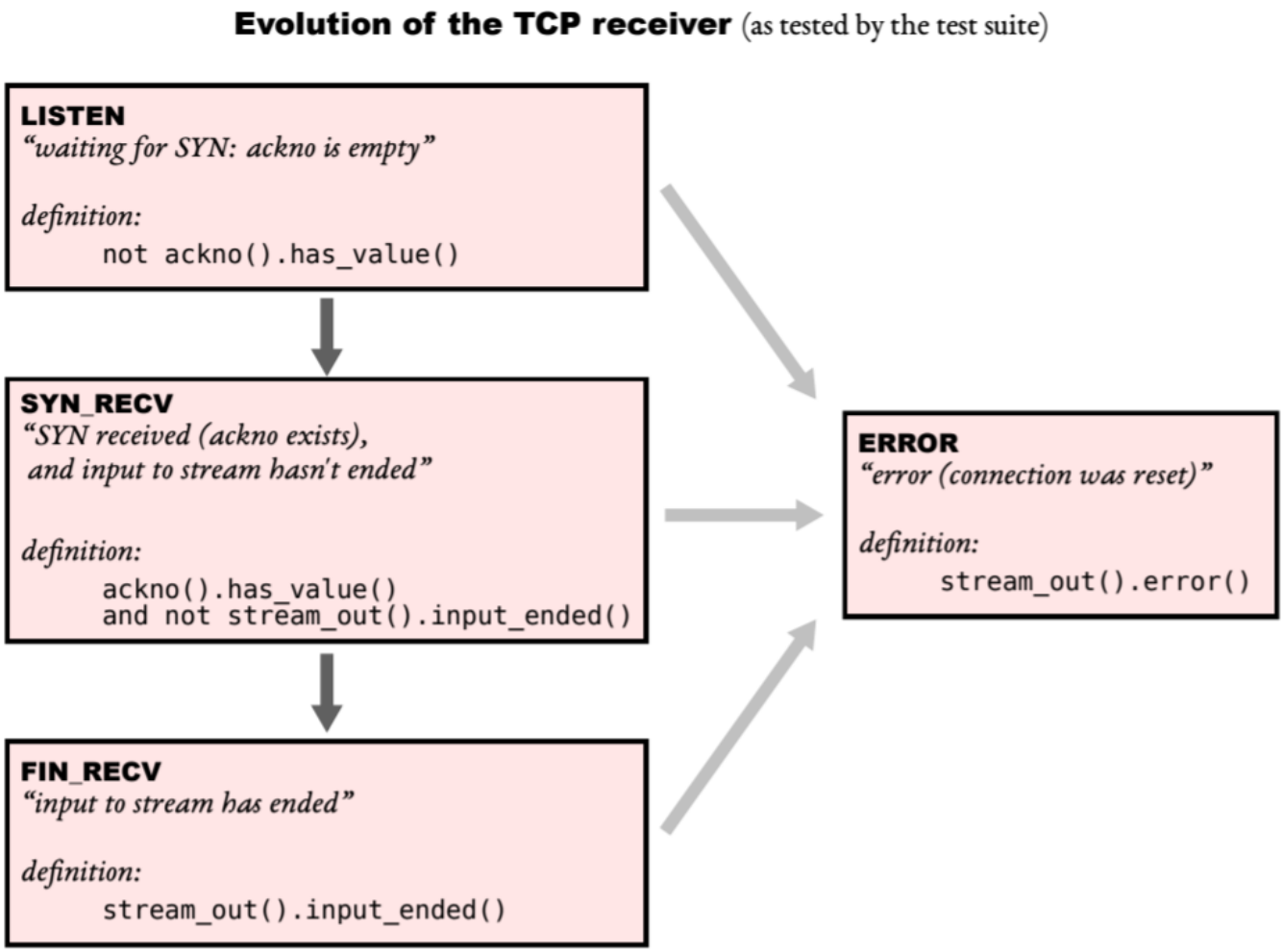
Returns an optional <WrappingInt32> containing the sequence number of the first byte that the receiver doesn't already know. This is the windows's left edge: the first byte the receiver is interested in receiving. If the ISN hasn't been set yet, return an empty optional.

3.2.3 window_size()

Returns the distance between the "first unassembled" index (the index corresponding to the ackno) and the "first unacceptable" index.

3.3 Evolution of the TCPReceiver over the life of the connection

Over the course of a TCP connection, your **TCPReceiver** will evolve through a sequence of states: from waiting for a SYN (with empty ackno), to an in-progress stream, to a stream that's finished, meaning input has ended on the ByteStream. The test suite will check that your **TCPReceiver** correctly handles incoming TCPSegments and evolves through these states, as shown below. (You don't have to worry about the error state or the RST flag until Lab4.)



4 Development and debugging advice

1. Implement the **TCPReceiver**'s public interface (and any private methods or functions you'd like) in the file **tcp_receiver.cc**. You may add any private members you like to the **TCPReceiver** class in **tcp_receiver.hh.**
2. After compiling, you can test your code with `make check_lab2`
3. Please work to make your code readable to the CA who will be grading it for style. Use reasonable and clear naming conventions for variables. Use comments to explain complex or subtle pieces of code. Use "defensive programming"—explicitly check preconditions of functions or invariants, and throw an exception if anything is ever wrong. Use modularity in your design—identify common abstractions and behaviors and factor them out when possible. Blocks of repeated code and enormous functions will make your code harder to follow.
4. Please also keep to the "Modern C++" style described in the Lab0 document. The cppreference website (https://en.cppreference.com) is a great resource, although you won't need any sophisticated features of C++ to do these labs.
5. If you get a segmentation fault, something is really wrong! We would like you to be writing in a style where you use safe programming practices to make segfaults extremely unusual (no malloc(), no new, no pointers, safety checks that throw exceptions where you are uncertain, etc.). That said, to debug you can configure your build directory with ~ to enable the compiler's "sanitizers" to detect memory errors and undefined behavior and give you a nice diagnostic about when they occur. You can also use the valgrind tool. You can also configure with `cmake .. -DCMAKE BUILD TYPE=Debug` and use the GNU debugger (**gdb**). Both of these will slow down your code—don't forget to return to a "Release" build when done.

5 Submit

1. In your submission, please only make changes to the **.hh** and **.cc** files in the top level of **libsponge**. Within these files, please feel free to add private members as necessary, but please don't change the public interface of any of the classes.
2. **Do not change the the file structure of original repository!** That means that your should not move files from `sponge` to root of your repository, or should not perform any other not required operations, which may cause your code is not in `sponge/libsponge`.
3. Before handing in any assignment, please run these in order:
    - `make` (to make sure the code compiles)
    - `make check_lab2`(to make sure the automated tests pass)
4. Write a report in `sponge/writeups/lab2.md`. **If you are not sure whether pictures in md are visible in your report, you can export md into pdf**. The report should contain the following sections:
    - **Program Structure and Design**: Describe the high-level structure and design choices embodied in your code. You do not need to discuss in detail what you inherited from the starter code. Use this as an opportunity to highlight important design aspects and provide greater detail on those areas for your grading TA to understand. You are strongly encouraged to make this writeup as readable as possible by using subheadings and outlines.
    - **Implementation Challenges**: Describe the parts of code that you found most troublesome and explain why. Reflect on how you overcame those challenges and what helped you finally understand the concept that was giving you trouble. How did you attempt to ensure that your code maintained your assumptions, invariants, and preconditions, and in what ways did you find this easy or difficult? How did you debug and test your code?
    - **Remaining Bugs**: **Submit your test screenshots** from `make check_lab2`. Point out and explain as best you can any bugs (or unhandled edge cases) that remain in the code.
5. Please also fill in the number of hours the assignment took you and any other comments.
6. When ready to submit, please follow the instructions in Our QQ group. Please make sure you have committed everything you intend before submitting. We can only grade your code if it has been committed.
7. Please let the course staff know ASAP of any problems at the Friday-afternoon lab session, or by posting a question on QQ group. Good luck and welcome to NJU networking lab!