Part I

1. **ByteStream::ByteStream(const size_t capa):**
    - This is the constructor for the ByteStream class.
    - It takes a size_t parameter capa, representing the capacity of the byte stream.
    - Initializes several member variables:
        - capacity: Set to the provided capacity value.
        - bytesWritten: Initialized to 0, representing the number of bytes written.
        - bytesRead: Initialized to 0, representing the number of bytes read.
        - inputEnded: Initialized to false, indicating that the input has not ended.
        - _error: Initialized to false, indicating no error has occurred.
2. **size_t ByteStream::write(const string& data):**
    - This method is responsible for writing data into the ByteStream.
    - It takes a const string& parameter data representing the data to be written.
    - Checks whether the input has already ended or an error has occurred. If either is true, it sets _error to true and returns 0.
    - Calculates bytesToWrite as the minimum between the length of the provided data and the remaining capacity of the stream.
    - Iterates through the data and appends bytes to the internal buffer while updating the bytesWritten count.
    - Finally, it returns the number of bytes actually written.
3. **string ByteStream::peek_output(const size_t len) const:**
    - This method allows you to examine a specified number of bytes from the beginning of the internal buffer without removing them.
    - It takes a size_t parameter len indicating the maximum number of bytes to peek.
    - Calculates outputLen as the minimum between len and the current size of the internal buffer.
    - Returns a string containing the peeked bytes.
4. **void ByteStream::pop_output(const size_t len):**
    - This method is responsible for removing a specified number of bytes from the beginning of the internal buffer.
    - Takes a size_t parameter len representing the number of bytes to remove.
    - Calculates bytesToPop as the minimum between len and the current size of the internal buffer.
    - Iterates through the internal buffer and removes bytes from the front while updating the bytesRead count.
5. **string ByteStream::read(const size_t len):**
    - This method combines peeking and popping operations.
    - It calls peek_output() to get a string of the specified length and then calls pop_output() to remove those bytes from the buffer.
    - Finally, it returns the string of bytes that were read.
6. **void ByteStream::end_input():**
    - This function marks the end of input by setting the inputEnded flag to true.
    - After this call, no more data can be written to the stream.

7. **bool ByteStream::input_ended() const:**
    - Returns true if the input has ended (i.e., inputEnded is true), indicating that no more data can be written.
8. **size_t ByteStream::buffer_size() const:**
    - Returns the current size (number of bytes) in the internal buffer.
9. **bool ByteStream::buffer_empty() const:**
    - Returns true if the internal buffer is empty, indicating that there is no data in the buffer.
10. **bool ByteStream::eof() const:**
    - Returns true if both the input has ended (via inputEnded) and the internal buffer is empty, indicating that there is no more data to read.
11. **size_t ByteStream::bytes_written() const:**
    - Returns the total number of bytes that have been written to the stream.
12. **size_t ByteStream::bytes_read() const:**
    - Returns the total number of bytes that have been read from the stream.
13. **size_t ByteStream::remaining_capacity() const:**
    - Calculates and returns the remaining capacity of the stream, which is the difference between the specified capacity and the current buffer size, ensuring it doesn't go negative.

Part II

1. **StreamReassembler::StreamReassembler(const size_t capacity)**:
    - Constructor for the StreamReassembler class. It initializes various member variables, including the capacity, unassembled data storage, and tracking of the assembled data. It also initializes the capacity for the reassembled stream.
2. **StreamReassembler::push_substring(const string &data, const size_t index, const bool eof)**:
    - This function is used to push a substring of data into the reassembler.
    - It takes as input the data substring, its starting index, and a boolean indicating whether this is the end of the stream (eof).
    - It first finds the appropriate index to insert the data, considering overlapping data and the next expected assembled index.
    - It processes overlapping data and ensures that data is not pushed if there's not enough space in the output buffer.
    - If there is enough space, it stores the data and processes fully assembled data.
    - If eof is true, it marks the end of the stream and ends input if the end index is less than or equal to the next expected assembled index.
3. **StreamReassembler::unassembled_bytes() const**:
    - This function returns the number of unassembled bytes, which are bytes that have been received but are not yet part of the contiguous assembled stream.
4. **StreamReassembler::empty() const**:
    - This function checks if there are any unassembled bytes. It returns true if there are none, indicating that all received data has been successfully reassembled.

5. **StreamReassembler::ack_index() const**:
    - This function returns the acknowledgment index, which is the next expected byte index that should be assembled.
6. **StreamReassembler::findNewIndex(size_t index)**:
    - A private function used to find the new index at which to insert incoming data.
    - It considers overlapping data and the next expected assembled index to determine the appropriate insertion point.
7. **StreamReassembler::calculateDataSize(const std::string &data, size_t newIndex, size_t index)**:
    - A private function used to calculate the size of the data to store.
    - It considers the new index and the original index to determine how much data should be stored.
8. **StreamReassembler::processOverlappingData(size_t newIndex, ssize_t &dataSize)**:
    - A private function that processes overlapping data.
    - It checks for data fragments that overlap with the new data and handles them by updating the dataSize and removing overlapping data from storage.
9. **StreamReassembler::hasEnoughSpace(size_t newIndex)**:
    - A private function that checks if there is enough space in the output buffer for new data.
10. **StreamReassembler::storeData(size_t newIndex, const std::string &data, ssize_t dataSize, size_t index)**:
    - A private function that stores incoming data based on the new index.
    - It manages data storage, including partially assembled data when necessary.
11. **StreamReassembler::processFullyAssembledData()**:
    - A private function that processes fully assembled data.
    - It checks if there is contiguous data that can be added to the output stream and performs the necessary updates.

Part III

tcp_receiver.cc: -

1. **TCPReceiver::segment_received(const TCPSegment &seg)**
    - This method is responsible for processing incoming TCP segments.
    - It first checks if it should process the segment based on the SYN flag (TCP connection establishment).
    - If the SYN flag is set, it calls handleSynReceived() to handle the initial SYN packet.
    - It then extracts the payload data from the segment, calls processPayloadData() to process it, and checks for the FIN flag to potentially close the stream.
2. **optional<WrappingInt32> TCPReceiver::ackno() const**

- This function calculates and returns the acknowledgment number (ACK) for the received TCP segments.
- If the initial SYN packet has not been received (_synReceived is false), it returns nullopt to indicate that the ACK cannot be determined yet.
- Otherwise, it calculates the ACK number as the next expected sequence number (wait_index() + 1), possibly incrementing it by 1 if the input has ended (stream_out().input_ended()).

3. **size_t TCPReceiver::window_size() const**
   - This method calculates and returns the available window size for the sender based on the capacity of the receiver's reassembler and the current buffer size of the `stream_out()` object.
4. **bool TCPReceiver::shouldProcessSegment(bool syn) const**
   - A helper function that determines whether a received segment should be processed based on whether it has the SYN flag set or if the SYN has already been received.
5. **void TCPReceiver::handleSynReceived(const TCPHeader &header)**
   - This method handles the reception of the initial SYN packet.
   - It marks the SYN as received (`_synReceived = true`) and initializes the initial sequence number (`_isn`) with the received sequence number (`header.seqno`).
6. **void TCPReceiver::processPayloadData(const string &data, const TCPHeader &header, bool syn, bool fin)**
   - This method processes the payload data received in TCP segments.
   - It checks if the data is not empty and whether it starts with a sequence number different from the initial sequence number (`_isn`), indicating out-of-order data.
   - It calculates the correct index for this data, based on the expected sequence number and the current waiting index (`unwrap()`).
   - It then pushes the data into the reassembler, potentially marking the end of the stream if the FIN flag is set.
7. **void TCPReceiver::handleFinReceived(bool fin)**
   - This method handles the reception of FIN packets.
   - If the FIN flag is set, it marks the FIN as received (`_finReceived = true`) and checks if there are no unassembled bytes left in the reassembler. If so, it ends input for the `stream_out()`.
   - If the FIN flag is not set, it does nothing.
8. **uint64_t TCPReceiver::calculateAckIndex() const**
   - This function calculates the acknowledgment index by adding 1 to the current wait index in the reassembler.

wrapping_integers.cc: -

1. **WrappingInt32 wrap(uint64_t n, WrappingInt32 isn)**
   - This function is responsible for "wrapping" a 64-bit unsigned integer (n) into a 32-bit wrapping integer space, represented by the WrappingInt32 type.

- It takes two arguments: n, the input 64-bit integer to wrap, and isn, an initial value within the wrapping integer space.
- It adds the 64-bit integer n to the 32-bit isn by first casting n to a uint32_t to ensure it fits within the 32-bit space.
- The result represents the wrapped value in the 32-bit space and is returned.

2. **uint64_t unwrap(WrappingInt32 n, WrappingInt32 isn, uint64_t checkpoint)**
   - This function performs the reverse operation of "unwrapping" a 32-bit wrapping integer back to a 64-bit unsigned integer.
   - It takes three parameters: n, the 32-bit wrapping integer to unwrap, isn, the initial value within the wrapping integer space, and checkpoint, a reference point for unwrapping.
   - The function starts by calculating the offset, which is the difference between n and isn. This offset represents how much n has wrapped around the 32-bit space.
   - It checks whether checkpoint is greater than the offset. If it is, it means that n has wrapped around at least once, and additional steps are needed to correctly unwrap it.
   - Inside the conditional branch, it calculates the real_checkpoint by subtracting the offset from checkpoint and adding half of the 32-bit wrapping range (INT32_RANGE / 2). This adjustment ensures that the result is within the correct wrapping range.
   - It then calculates how many times real_checkpoint has wrapped around the 32-bit space, storing this value in wrap_num. The final result is computed by multiplying wrap_num with the wrapping range and adding the offset.
   - If checkpoint is not greater than the offset, it means that n has not wrapped around, so the offset is directly returned as the unwrapped value.

```
abhijeetanand@abhijeetanand-VirtualBox:~/Desktop/CN Assignment 2/build$ ctest
Test project /home/abhijeetanand/Desktop/CN Assignment 2/build
      Start  1: wrapping_integers_cmp
 1/23 Test  #1: wrapping_integers_cmp ..............   Passed    0.00 sec
      Start  2: wrapping_integers_unwrap
 2/23 Test  #2: wrapping_integers_unwrap ............   Passed    0.00 sec
      Start  3: wrapping_integers_wrap
 3/23 Test  #3: wrapping_integers_wrap ..............   Passed    0.00 sec
      Start  4: wrapping_integers_roundtrip
 4/23 Test  #4: wrapping_integers_roundtrip .........   Passed    1.90 sec
      Start  5: byte_stream_construction
 5/23 Test  #5: byte_stream_construction ............   Passed    0.00 sec
      Start  6: byte_stream_one_write
 6/23 Test  #6: byte_stream_one_write ...............   Passed    0.00 sec
      Start  7: byte_stream_two_writes
 7/23 Test  #7: byte_stream_two_writes ..............   Passed    0.00 sec
      Start  8: byte_stream_capacity
 8/23 Test  #8: byte_stream_capacity ................   Passed    0.86 sec
      Start  9: byte_stream_many_writes
 9/23 Test  #9: byte_stream_many_writes .............   Passed    0.00 sec
      Start 10: recv_connect
10/23 Test #10: recv_connect ........................   Passed    0.00 sec
      Start 11: recv_transmit
11/23 Test #11: recv_transmit .......................   Passed    0.08 sec
      Start 12: recv_window
12/23 Test #12: recv_window .........................   Passed    0.00 sec
      Start 13: recv_reorder
13/23 Test #13: recv_reorder ........................   Passed    0.00 sec
      Start 14: recv_close
14/23 Test #14: recv_close ..........................   Passed    0.00 sec
      Start 15: recv_special
15/23 Test #15: recv_special ........................   Passed    0.00 sec
      Start 16: fsm_stream_reassembler_cap
16/23 Test #16: fsm_stream_reassembler_cap ...........   Passed    0.17 sec
      Start 17: fsm_stream_reassembler_single
17/23 Test #17: fsm_stream_reassembler_single ........   Passed    0.00 sec
      Start 18: fsm_stream_reassembler_seq
18/23 Test #18: fsm_stream_reassembler_seq ...........   Passed    0.00 sec
      Start 19: fsm_stream_reassembler_dup
19/23 Test #19: fsm_stream_reassembler_dup ...........   Passed    0.01 sec
      Start 20: fsm_stream_reassembler_holes
20/23 Test #20: fsm_stream_reassembler_holes .........   Passed    0.00 sec
      Start 21: fsm_stream_reassembler_many
21/23 Test #21: fsm_stream_reassembler_many ..........   Passed    0.32 sec
      Start 22: fsm_stream_reassembler_overlapping
22/23 Test #22: fsm_stream_reassembler_overlapping ...   Passed    0.00 sec
      Start 23: fsm_stream_reassembler_win
23/23 Test #23: fsm_stream_reassembler_win ...........   Passed    0.28 sec

100% tests passed, 0 tests failed out of 23

Total Test time (real) =   3.68 sec
```