CN Assignment 2

Akshat

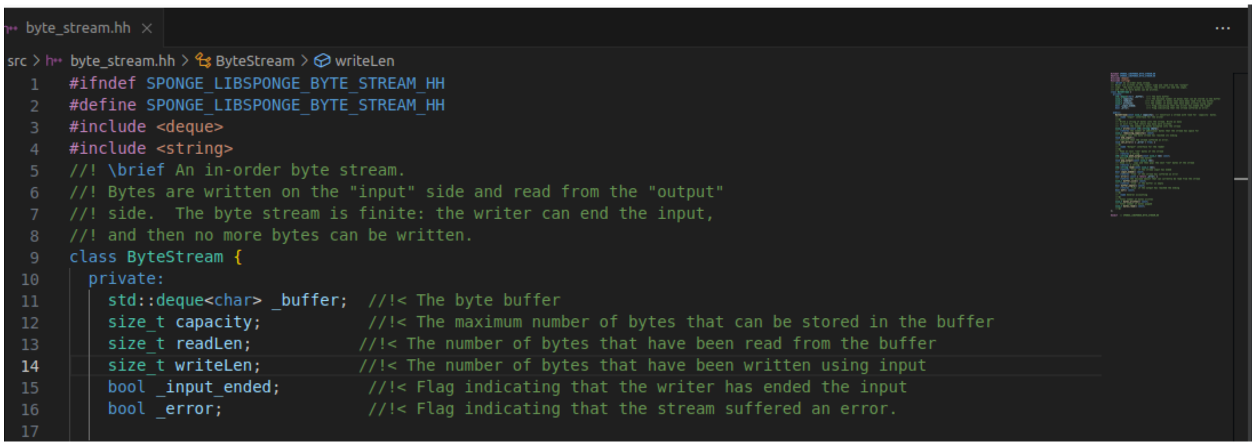
B.Tech ECE

2020172

Part I)

‘byte_stream.hh’:
private declarations:

1) A ‘deque’ (_buffer) which is our required data structure.
2) Three variables of type ‘Unsigned long size_t’ to store the ‘capacity’(capacity), total bytes ‘read’ (readLen), and ‘written’(writeLen).
3) Two boolean flags checking ‘End-of-file’(_input_ended) and ‘Error’(_error).



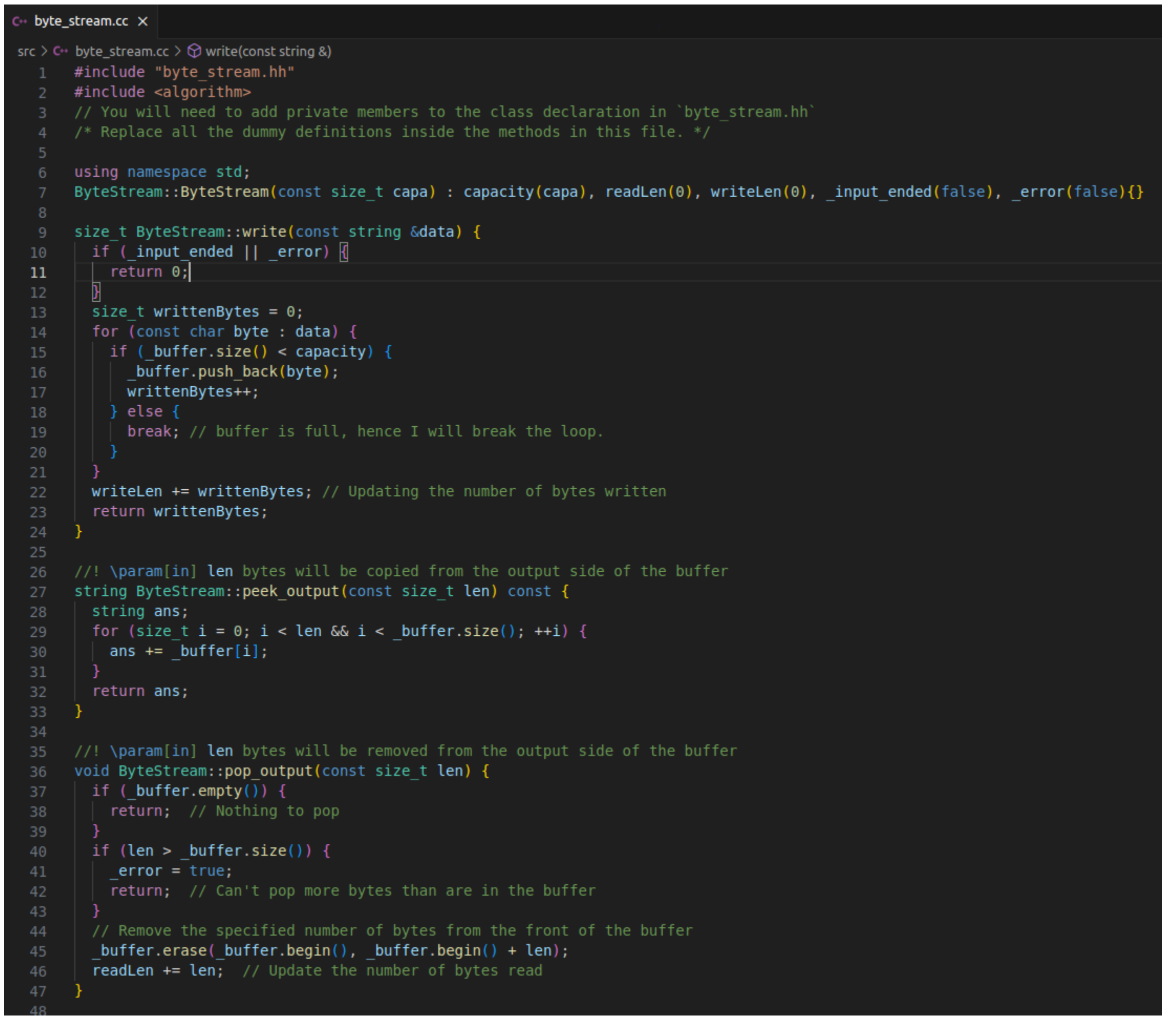
```
#ifndef SPONGE_LIBSPONGE_BYTE_STREAM_HH
#define SPONGE_LIBSPONGE_BYTE_STREAM_HH
#include <deque>
#include <string>
//! \brief An in-order byte stream.
//! Bytes are written on the "input" side and read from the "output"
//! side.  The byte stream is finite: the writer can end the input,
//! and then no more bytes can be written.
class ByteStream {
  private:
    std::deque<char> _buffer;  //!< The byte buffer
    size_t capacity;           //!< The maximum number of bytes that can be stored in the buffer
    size_t readLen;           //!< The number of bytes that have been read from the buffer
    size_t writeLen;          //!< The number of bytes that have been written using input
    bool _input_ended;         //!< Flag indicating that the writer has ended the input
    bool _error;               //!< Flag indicating that the stream suffered an error.

```

The remaining code is similar to the initial one.

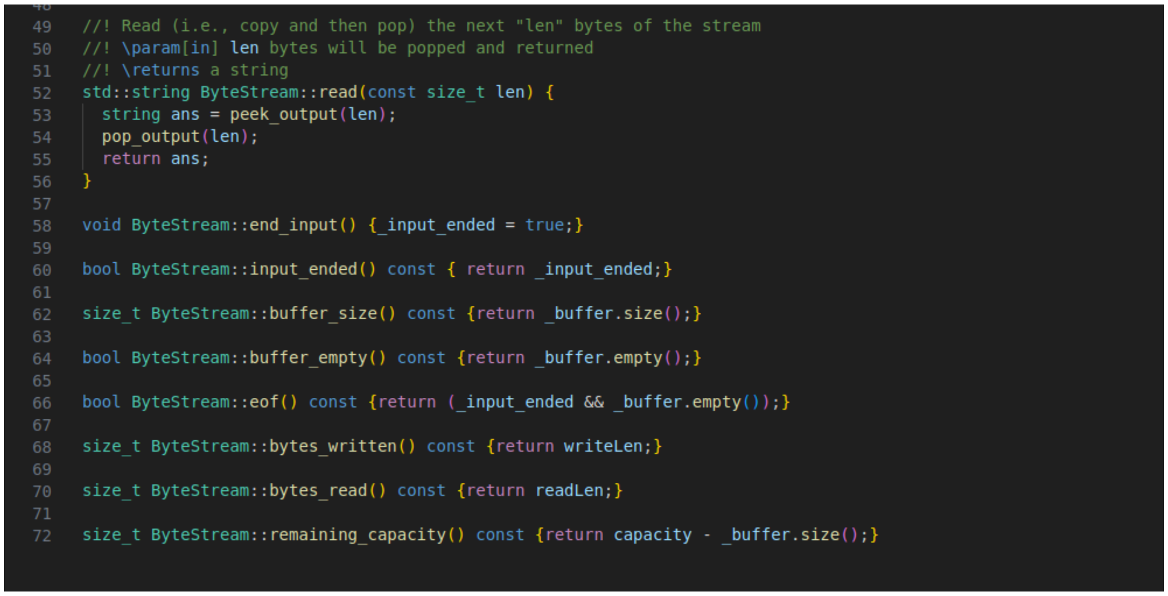
‘byte_stream.cc’:

1) Write: If the input has ended or there is an error, then no need to write, now start adding data content into the buffer (also checking if the buffer is filled or not).
2) Peek: Make a string variable, add the content to the required length, and then return the variable. (peeking as much as possible).
3) Pop: If the buffer is empty, then stop the function. Also, If the popping length is more than the current size of the buffer, then ‘set_error’ and return. Now pop the required length by simply erasing the content up to the required index.
4) Read: First peek and then pop.
5) The rest are simple functions.

Screenshot:

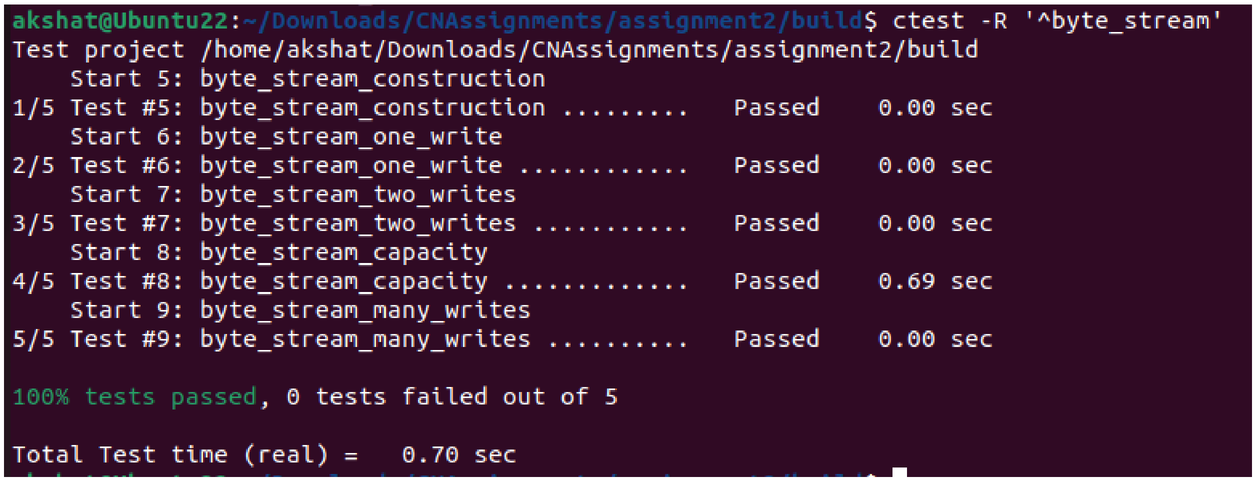


```cpp
#include "byte_stream.hh"
#include <algorithm>
// You will need to add private members to the class declaration in `byte_stream.hh`
/* Replace all the dummy definitions inside the methods in this file. */

using namespace std;
ByteStream::ByteStream(const size_t capa) : capacity(capa), readLen(0), writeLen(0), _input_ended(false), _error(false){}

size_t ByteStream::write(const string &data) {
  if (_input_ended || _error) {
    return 0;
  }
  size_t writtenBytes = 0;
  for (const char byte : data) {
    if (_buffer.size() < capacity) {
      _buffer.push_back(byte);
      writtenBytes++;
    } else {
      break; // buffer is full, hence I will break the loop.
    }
  }
  writeLen += writtenBytes; // Updating the number of bytes written
  return writtenBytes;
}

//! \param[in] len bytes will be copied from the output side of the buffer
string ByteStream::peek_output(const size_t len) const {
  string ans;
  for (size_t i = 0; i < len && i < _buffer.size(); ++i) {
    ans += _buffer[i];
  }
  return ans;
}

//! \param[in] len bytes will be removed from the output side of the buffer
void ByteStream::pop_output(const size_t len) {
  if (_buffer.empty()) {
    return;  // Nothing to pop
  }
  if (len > _buffer.size()) {
    _error = true;
    return;  // Can't pop more bytes than are in the buffer
  }
  // Remove the specified number of bytes from the front of the buffer
  _buffer.erase(_buffer.begin(), _buffer.begin() + len);
  readLen += len;  // Update the number of bytes read
}

```



```cpp
//! Read (i.e., copy and then pop) the next "len" bytes of the stream
//! \param[in] len bytes will be popped and returned
//! \returns a string
std::string ByteStream::read(const size_t len) {
  string ans = peek_output(len);
  pop_output(len);
  return ans;
}

void ByteStream::end_input() {_input_ended = true;}

bool ByteStream::input_ended() const { return _input_ended;}

size_t ByteStream::buffer_size() const {return _buffer.size();}

bool ByteStream::buffer_empty() const {return _buffer.empty();}

bool ByteStream::eof() const {return (_input_ended && _buffer.empty());}

size_t ByteStream::bytes_written() const {return writeLen;}

size_t ByteStream::bytes_read() const {return readLen;}

size_t ByteStream::remaining_capacity() const {return capacity - _buffer.size();}
```

Test results for byte_stream:



```
akshat@Ubuntu22:~/Downloads/CNAssignments/assignment2/build$ ctest -R '^byte_stream'
Test project /home/akshat/Downloads/CNAssignments/assignment2/build
    Start 5: byte_stream_construction
1/5 Test #5: byte_stream_construction .........   Passed    0.00 sec
    Start 6: byte_stream_one_write
2/5 Test #6: byte_stream_one_write ...........   Passed    0.00 sec
    Start 7: byte_stream_two_writes
3/5 Test #7: byte_stream_two_writes ..........   Passed    0.00 sec
    Start 8: byte_stream_capacity
4/5 Test #8: byte_stream_capacity ............   Passed    0.69 sec
    Start 9: byte_stream_many_writes
5/5 Test #9: byte_stream_many_writes .........   Passed    0.00 sec

100% tests passed, 0 tests failed out of 5

Total Test time (real) =   0.70 sec
```

All test cases have passed. Hence, the implementations are meeting the requirements for now.