Cpt S 422 Homework Assignment #2 and #3
Streams
by Evan Olds, Fall 2014

HW assignments 2 and 3 are worth 20 points total. HW2 must be submitted within the first week as a progress report and then the final implementation is delivered as homework #3 one week after that.

This is a coding assignment. The overwhelming majority of the class chose Visual Studio as the preferred development environment (more than all the other categories combined) so Visual Studio projects will be required for the remainder of the coding assignments (homework assignments only, for the final project you can use whatever).

You may work in groups of up to 5 people on this assignment. Groups of more than 5 are not allowed. The number of people in the group affects how many items you have to implement. Submit 1 copy of the homework per group. Make sure names of all the group members are on the assignment. Submit a .zip to Angel with project and code files containing implementations for the requirements listed below.

Open the Visual Studio 2013 project included in the assignment .zip file. There is a Stream base class declared in the Stream.hpp file. There is also a Source.cpp that contains the main function and nothing else.

Part 1 (10 points) – Stream Subclasses

Write the following classes that inherit from Stream. Make sure each matches the specification listed in the comments for the base class (matching return value conventions, file position adjustment behavior, and so on).

FileStream (must be implemented regardless of group size)

- This class opens a file on disk for reading and/or writing. Have a constructor that takes a value indicating the open mode. Support opening for read only, write only and read/write.

- Implement the file I/O operations using either standard I/O (fopen, fread, fwrite…) or stream I/O (ifstream, ofstream).

<u>IndexedNumbersStream</u> (must be implemented regardless of group size)

- The class procedurally generates byte data in the stream. There is no file or memory buffer behind the stream implementation.
- The stream provides read-only support. It does not support writing.
- The stream constructor takes an integer that represents the size, in bytes, of the stream. Store this value in a member variable.
- All reads populate data in the output buffer such that:
    - byte value at stream position i = i % 256
- Because of the consistent byte value scheme, data from the stream is consistent. If you read 13 bytes at position 23 at some particular time, doing a bunch of operations and then coming back later to read 13 bytes at position 23 again will yield the same content.
- Conceptually the stream is representing data as shown in the table below. Remember that it doesn't need to create that array in memory, it generates the necessary data each time the read function is called.

Stream Position:	0	1	2	…	255	256	257	…	n
Byte Value:	0	1	2	…	255	0	1	…	n % 256

<u>MemoryStream</u> (must be implemented by all groups with 3 or more members)

- The memory stream represents in-memory content and supports reading and writing.
- Reads cannot go beyond the end of the stream.
- The stream size is dynamic, and determined by writes.
- Writes can extend the size/length of the stream, which is 0 initially. As an example, if you make 3 write calls sequentially (without any seeking in between) of x bytes, then y bytes, then z bytes, then the stream's GetLength() function should return x + y + z.

- Setting the stream position to any location in the range [0, stream_length] is valid. Note that setting the position to the length of the stream makes the position right *after* the last byte in the stream, which is a valid position.

Part 2 (10 points) – Unit tests

Write the following unit tests for the streams:

Write Validation Tests (must be implemented regardless of group size)

- Write a unit test that takes a stream and writes data to it in various ways, then reads it back in various ways to verify contents, byte-by-byte.
- The test must check to see that the stream supports writing. If it doesn't then the test is considered successful and just returns after asserting success.
- If the stream does support writing, then first generate contents to write to the file. It can be totally random data generated in memory, hard-coded data, data from a test file, or anything else where you can go back and verify the stream contents against it.
- Have tests that write the data to the stream in the ways listed below. Validate the stream position after each write. At least 3 of these tests must be implemented regardless of group size. All of them must be implemented if the group size is 3 or more people.
    - All at once with one Write call
    - Sequentially with a fixed and common buffer size (like 1024)
    - Sequentially with a fixed and uncommon buffer size (like 1801)
    - In backwards order[1] with a random buffer sizes. You'll have to do an initial write to pad the stream size out to the correct length. Then you can write chunks with actual data in backwards order.
- After the writes complete, you must have tests to read and verify stream contents. This implies that you'll need to have the original written data around in some other form (array in memory is the easiest) to verify against.

[1] To read stream of size n bytes backwards: seek to position n – buffer_size and read buffer_size bytes of data. Seek to n – (buffer_size * 2) and reads buffer_size bytes of data. Carry on seeking with the pattern n – (buffer_size * i) until i is 0, which will be the beginning of the stream and will be the last read.

At least 3 of these tests must be implemented regardless of group size; all 5 of them must be implemented if the group size is 3 or more people.

- o All at once with one Read call
- o Sequentially with a fixed and common buffer size (like 1024)
- o Sequentially with a fixed and uncommon buffer size (like 2713)
- o In backwards order with a random buffer sizes.
- o In backwards order 1 byte at a time.

<u>Data consistency tests</u> (must be implemented regardless of group size)

- These tests aim to make sure that the data is consistent within the stream. While many of the tests above probably just read a chunk once and compared it to the original source for validation, they aren't checking to see if multiple overlapping reads are returning the same and correct data each time.
- Write a series of unit test that read from the same region in the stream multiple times in random, sequential, and backwards orders. Pass streams into these test functions that have non-zero sizes. This means you need to do something along these lines for each stream type:
    1. Create an instance of the stream, passing whatever information is needed (file names, size values, etc.) to the constructor so that it has a non-zero size and has data to read.
    2. Pass the stream to the unit test function that reads in sequential order
    3. Pass the stream to the unit test function that reads in backwards order
    4. Pass the stream to the unit test function that reads in random order
- You will test each stream type that you implemented with your data consistency tests. This means that:
    1. For groups of 1 or 2 people you'll be testing the file and indexed number streams.
    2. For groups of 3 or more people you'll be testing the file, indexed number, and memory streams.
- Describe what your tests are achieving in your code comments.

<u>Other requirements / notes:</u>

- Make sure that the results of these tests are listed in the console/terminal window.
- Do not use debug asserts or anything else that kills your app because a test fails. If a test fails indicate so in your results output and move on to the next test in the sequence.
  - The final result output from your code should always be giving only success messages. The tests finding and giving info about errors are really more for your own benefit to ensure that your stream implementations are correct.
- Comment your code and make it clear where the different parts are implemented. Make sure each of your stream class implementations is in its own code file. Do not declare everything in one giant .cpp file.
- Make sure the names of all group participants are clearly listed so the TA gives everyone points.
- Code that does not compile in Visual Studio 2013 is worth 0.
- Code that compiles and runs but crashes is going to be worth very few points. At 400-level it is expected that you can write code that works without crashing, especially for assignments like this where there are not a lot of complex algorithms or data structures.