Assignment 1: Queues and Pointers

This lecture assignment, worth 20 marks and 5% of your final grade, is due by 4:00 PM on Friday, Sep. 19th. Submit your solution (a printed and stapled/bound copy of all files you write/edit) to the ECE 312 LEC A1 assignment box outside the ECERF reception (2nd floor). Include your name, your student ID, the percentage of the submission (excluding code supplied with the assignment) that is original by you, and the names of all other contributors to your submission (indicate the percent contributed by each).

Electronic and wrong-box submissions will not be accepted. If any part of the assignment is submitted late, the whole assignment will be penalized 2 marks, i.e., 10% of the maximum mark, per business day. Submissions received, in whole or in part, after 4:00 PM on Tuesday, Sep. 23rd, will receive 0 marks. Do not cheat, do not plagiarize, do not misrepresent facts, and do not participate in an offence!

Objectives

Producer-consumer problems often employ queues, or first-in first-out (FIFO) buffers. The producer and consumer tasks, running concurrently, put or get messages into or from a shared queue, respectively. Trying to put data into a full queue or to get data from an empty queue may either return an error or block the task. In this assignment, you will program, document, and test two subroutines of a queue with a special property inspired by online video streaming. State-driven producer and consumer tasks are given to illustrate the use of the queue in a simulated cooperative multi-tasking application.

Getting Started

In Visual Studio, create a new one-project solution with `Queue.h` as a header file and `Queue.cpp` and `Main.cpp` as source files. Put `Input.txt` in the project's working directory. Set it and `Output.txt` as input and output files, respectively, as follows. Right click on your project in the Solution Explorer and select Properties. Choose Configuration Properties ≫ Debugging in the left pane. In the right pane, by Command Arguments, enter "`Input.txt Output.txt`" without quotes and click on OK.

Build and run the project. Verify that it creates an `Output.txt` file successfully. If you cannot build and run the code, please try the ETL E5-012 lab, where Visual Studio is available via VMware Player.

In addition to this document, please read: the Wikipedia page on queues; the Embedded pages on ring or circular buffers (ignore the code); and all code and comments in the given `Queue.h`, `Queue.cpp`, and `Main.txt`. You may also want to review C/C++ pointers from online sources, a reference textbook, or your ECE 220 notes. The hyperlinks given below may be considered recommended reading.

Technical Specifications

Write/edit the `qPutMsg` and `qGetMsg` functions (`Queue.cpp`). Although some code is given there for Getting Started, you may delete it. Do not change any other code, including the interfaces of the `qPutMsg` and `qGetMsg` functions. Do not add `include` statements or modify existing ones. Your entire solution should be programmed into the `cpp` file. You may add other (helper) functions if that will help you develop a solution. Although used in `Main.cpp`, do not use static local or global variables.

Study the `QUEUE` structure, `qInit`, and `qFree` functions, in both the `Queue.h` and `Queue.cpp` files. Another programmer has implemented part of the queue, and your part must conform to his part. Although each message in the queue has the same size in bytes, note that this size is arbitrary and it is specified upon initialization. Heap memory is allocated to store a maximum number of messages, and the memory is configured as type `char`. Essentially, `memPtr` refers to an array of bytes.

The `qPutMsg` and `qGetMsg` functions each have two arguments. One, `q`, is a `QUEUE`. The other, `msgPtr`, is a void pointer to an input/output message. As with a snake, where food enters at the head and leaves at the tail, a new message should be put (copied) into the queue memory at the `headPtr` position. One gets (copies) an old message from the queue memory at the `tailPtr` position. Increment the pointers appropriately, upon each put or get. Wrap around to the beginning of the queue memory when the pointers reach the end. Avoid buffer overruns and memory leaks in your solution.

Keep track of the number of valid messages in the queue, a count that depends on puts and gets. The `qPutMsg` function should return `false`, without putting any message, if it is called on a full queue. The `qGetMsg` function should return `false`, without getting any message, if it is called on a queue where the number of messages is less than or equal to a threshold, set during initialization (default value is zero). This feature is needed to simulate the buffering typically used with video streaming.

Finally, the `qPutMsg` and `qGetMsg` functions should also return `false`, without putting/getting any messages, if they are called on an invalid queue, i.e., a queue where `memPtr` is zero (NULL).

Presentation Specifications

Format your code professionally. Compare your placement of curly braces, i.e., `{` and `}`, to that of the given code. Use a similar approach or another professional approach. Code of the same code block should be left aligned and the indentation should consistently match the nesting level. In Visual Studio, selecting EDIT ≫ Advanced ≫ Format Document from the menu may correct alignment errors. Watch the length of lines, comments included, so that lines do not overflow when printed. Either break long lines into multiple indented lines (the compiler ignores line breaks) or print in landscape mode.

Write suitable comment headers for all functions you write/modify, which clearly describe the purpose, inputs, and outputs of each function. Functions that use static local and/or global variables are not fully defined this way. Describe, in the comment header, the state-driven behaviour of functions that use such variables. This is one good reason to minimize/simplify the use of state variables.

Finally, because the queue-related code has been encapsulated in its own `cpp` file, a corresponding `h` file is needed to use the queue in an application (having a `main` function). Although implementation-related comments may be placed in the `cpp` file, interface-related comments are best placed in the `h` file. Therefore, for each function you write/edit, provide a comment header in the `h` file only.

Review your working solution and look for ways to simplify it so that it would be easier to follow. For example, use refactoring to reduce code smell. This is another good reason to minimize/simplify the use of state variables (note that the Technical Specifications for *this* assignment tell you *not* to use such variables). Furthermore, give helpful comments beyond those in the comment header.

Hints

You can find plenty of information online, e.g., an explanation of the `const` keyword used in `qPutMsg`. Read about and use `memcpy`, available through the included string library, to copy memory. Although you can correctly write your own code instead, such a solution may not qualify as tutorial quality.

Always run the code with debugging on to protect your computer from pointer-related bugs that you may introduce. To simplify debugging, students are advised to develop their own test code, not for submission, independent of `Main.cpp`. Queue code is encapsulated in `Queue.h` and `Queue.cpp`. Perform your own unit testing before employing the completed queue in the given simulation.

Do *not* submit the output file, `Output.txt`, generated by your code (corresponding to the given input file, `Input.txt`). The correct output is provided to you (`Output2.txt`) to help with your testing. See the `producer`, `consumer`, and `testSolution` functions (`Main.cpp`) for details on how all files are processed. Comparing your output to the correct output may help you meet most of the Technical Specifications. However, your code is marked directly, irrespective of any submitted output.

Each input line and the corresponding output line represent the concurrent downloading and playback of streaming video, which are done by state-driven `producer` and `consumer` tasks, respectively, that communicate via a shared queue. Cooperative multi-tasking is simulated where the two tasks are called in sequence at regular intervals. The first two numbers on each input line, echoed to the corresponding output line, define the maximum and threshold queue size (see Technical Specifications), respectively. Video packets are represented by letters, which are the messages put into and got from the queue. Asterisks denote waiting. Packets take four times longer to play back than to download.