PONDER 03 : STOCKS

Due Saturday at 11:59 PM MST

The third programming assignment will be to implement the queue data structure and use it to keep track of the buying and selling of stock.

Queue

Create a class encapsulating the notion of a array-based circular queue as described in the textbook. This will work exactly like the [std::queue](http://www.cplusplus.com/reference/queue/queue/) class. Of course, any data-type will need to be supported, so your class will be a template class. It will also need to grow to accommodate any number of elements. It will need to be defined in its own header file (queue.h). The class name must be Queue and will need to support the following operations:

* **Constructors**: Default constructor (create a queue with zero items in it), a non-default constructor (taking a capacity value as a parameter), and the copy constructor. If allocation is not possible, the following error will be thrown:  
  ERROR: Unable to allocate a new buffer for queue
* **Destructor**: When finished, the class should delete all the allocated memory.
* **operator=**: Assignment operator. This method takes a Queue as a parameter and copies all the elements to this. If the current buffer size is sufficient, not allocation is made. If the current buffer size is not sufficient, enough space is allocated to accomodate the new data. If there is insufficient memory to allocate a new buffer, then the following exception is thrown:  
  ERROR: Unable to allocate a new buffer for Queue. It also returns \*this by-reference as all assignment operators should.
* **empty()**: Test whether the queue is empty. This method takes no parameters and returns a Boolean value.
* **size()**: Return the queue size. This method takes no parameters and returns an integer value.
* **capacity()**: Return the current capacity of the queue. That is, the number of elements the queue is able to store without reallocating.
* **clear()**: Empties the queue of all items. There are no parameters and no return value.
* **push()**: Adds an item to the queue. This method takes a single parameter (the item to be added to the end of the queue) and has no return value. Note that if the queue is full, then the capacity will be doubled. In the case of an allocation error, the following c-string exception will be throw:  
  ERROR: Unable to allocate a new buffer for queue
* **pop()**: Removes an item from the head of the queue, serving to reduce the size by one. Note that if the queue is already empty, the following c-string exception will be thrown:  
  ERROR: attempting to pop from an empty queue
* **front()**: Returns the item currently at the front of the queue. This item is returned by-reference, so the last item can be changed through the front() method. If the queue is currently empty, the following exception will be thrown:  
  ERROR: attempting to access an item in an empty queue
* **back()**: Returns the item currently at the back of the queue. This item is returned by-reference so, the last item can be changed through the back() method. If the queue is currently empty, the following exception will be thrown:  
  ERROR: attempting to access an item in an empty queue

Note that the only way to access elements in a queue is through the front() and back() method. This means that there is no iterator for Queue.

Driver Program

A driver program is provided. This file (/home/cs235/week03/week03.cpp) will pound-include your header file (queue.h) and expect a template class Queue to be defined therein. This program will exercise your class, filling the container with user input and displaying the results. As with previous assignments, a makefile will be provided (/home/cs235/week03/makefile). There will also be provided a header file (stock.h), and an implementation file (stock.cpp). You will need to create the queue header file (queue.h).

Stocks

In addition to passing the four test functions for the Queue class, you will also need to use theQueue class to implement a program to keep track of stock transactions. An individual can buy stocks any number of stocks at the current market price. These stocks are then placed in the individual's portfolio. The person can also sell as many of the stocks in his portfolio as he likes at the current market price. When this is done, a record is made of which stocks were sold and for what price. To understand how this is done, a few concepts need to be explained:

When a share of common stock of a company is sold, the profit (or loss) is the difference between the share's selling price and the share's purchase price. For example, if 10 shares are bought at $1.50 and then sold at $2.00, the profits are ($2.00 - $1.50) x 10 = $5.00.

When we buy shares of stock over a long period of time and then decide to sell some (but not all) of them, we must identify the shares actually being sold. The accepted method for doing this is to sell the oldest ones first. For example, if 10 shares are bought at $1.50 and then 20 shares are bought for $1.75, then I will have 30 shares in my portfolio. If I then sell two shares from my portfolio for $2.50, then two shares are deduced from the oldest transaction: ($2.50 - $1.50) x 2 = $2.00. I will then have 8 shares remaining from the original batch and the full 20 from the second batch.

Note that one sell order can take more than one purchase batch to satisfy. Consider the case when I have made three purchases: 10 shares for $7.50, 12 shares for $7.80, and 4 shares for $9.00. All told, I have 26 shares. Now I wish to sell 25 shares for $10. My proceeds will be: ($10.00 - $7.50) x 10 + ($10.00 - $7.80) x 12 + ($10.00 - 9.00) x 3 = $44.40. I will also have one share left which was purchased for $9.00.

Our stocks program will allow the user to enter buy and sell records into his portfolio. When stocks are bought, then a record is made of the transaction and added to the portfolio. When stocks are sold, then the appropriate amount is deduced from the oldest transactions. The user can also display the contents of the portfolio at a given moment in time, including the total proceeds gained or lost from the transactions. To see how this works, consider the following execution:

This program will allow you to buy and sell stocks. The actions are:

buy 200 $1.57 - Buy 200 shares at $1.57

sell 150 $2.15 - Sell 150 shares at $2.15

display - Display your current stock portfolio

quit - Display a final report and quit the program

> buy 10 $1.50

> buy 20 $1.75

> display

Currently held:

Bought 10 shares at $1.50

Bought 20 shares at $1.75

Proceeds: $0.00

> sell 2 $2.00

> display

Currently held:

Bought 8 shares at $1.50

Bought 20 shares at $1.75

Sell History:

Sold 2 shares at $2.00 for a profit of $1.00

Proceeds: $1.00

A few hints that may come in handy when implementing this part of the assignment:

* When the display command is executed, there is a tab character immediately beforeBought and Sold.
* Only display "Currently held:" and "Sell History:" if there are transactions to display.
* When selling shares, use the oldest transactions first. This may mean one of three things: 1) only part of a buy-batch is used, 2) a complete buy-batch is used, or 3) multiple buy-batches are required to cover a given stock sell. Your program should handle all three cases.
* Compute the profit from a given sell buy subtracting the sell price from the buy price, multiplying the result by the number of shares sold. The proceeds are the sum of all the stock sells in the history.
* Use the Dollars class provided in /home/cs235/week03/dollars.cpp and/home/cs235/week03/dollars.h to handle currency. This will take care of the dollar sign that is often in front of currency on input.
* You may wish to create a class to handle a single transaction and another to handle a portfolio.
* With no iterator and no random access, it is difficult to view all the items in a queue. Nevertheless, this will be required when displaying the contents of the portfolio. You may need to create a copy of a queue which is destroyed through pops to view all the items in the portfolio.

As with the previous lessons, you must use your own Queue class to get full credit. If your class does not work, use the standard template library std::queue from #include <queue>. If you do this, you will loose points for the first half of the assignment, but not the second.

Common Mistakes

The most common mistakes students make with this assignment include the following:

* **Over simplified sell function**. Make sure you handle all the sell cases: all the stocks are available in a single transaction, only part of a single transaction is used, and more than one transaction is required to cover a given sell.
* **Incorrectly overloaded copy constructor and assignment operator.** The circular queue presents a unique problem with these overloaded operators and your reallocation function.

Test Bed

The testBed for this assignment is:

testBed cs235/week03 week03.tar

You can also run testBed on the executable:

testBed cs235/week03 a.out

Of course, you will need to pass testBed to get full credit on the assignment.

Submitting

You will submit this assignment as a pair using the Linux submit command. Please:

1. Create a TAR file built from the makefile, which will contain seven files:
   * makefile: Directly from /home/cs235/week03/makefile except with your edits on the comment block.
   * dollars.h: A class definition for Dollars, useful for keeping track of money.
   * dollars.cpp: The implementation for the Dollars class.
   * queue.h: Your class definition for Queue.
   * stock.h: Containing the prototype for stocksToBySell() and any other functions or classes you may need.
   * stock.cpp: Implementation for all the functions and classes necessary for the stocks program.
   * week03.cpp: Unmodified from /home/cs235/week03/week03.cpp.
2. Run the program by hand a few times through all four test cases as well as the stock program.
3. Verify your solution with testBed.
4. Submit your file using the submit command. The submit command will prompt you for your instructor, the class (cs235), and the assignment (week03). You submit your file with:

submit week03.tar

Your program will be graded according to the following rubric:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Exceptional 100% | Good 90% | Acceptable 70% | Developing 50% | Missing 0% |
| Queue Interface  20% | The interfaces are perfectly specified with respect to const, pass-by-reference, etc. | week03.cppcompiles without modification | All of the methods in Queue match the problem definition | Queue has many of the same interfaces as the problem definition | The public methods in theQueue class do not resemble the problem definition |
| Queue Implementation  20% | Passes all fourQueue testBed tests | Passes three testBed tests | Passes two testBed tests | Passes one testBed test | Program fails to compile or does not pass any testBed tests |
| Stocks  30% | The code demonstrates Object-Oriented design principles | Passes the stock testBed test | The code essentially works but with minor defects | Elements of the solution are present | The stock code was not attempted |
| Code Quality  20% | There is no obvious room for improvement | All the principles of encapsulation and modularization are honored | One function is written in a "backwards" way or could be improved | Two or more functions appears "thrown together." | The code appears to be written without any obvious forethought |
| Style  10% | Great variable names, no errors, great comments | No obvious style errors | A few minor style errors: non-standard spacing, poor variable names, missing comments, etc. | Overly generic variable names, misleading comments, or other gross style errors | No knowledge of the BYU-I code style guidelines were demonstrated |

Please make sure to fill out the program header in the makefile.