makefile

```
###################################################################
# Program:
#     Week 03, QUEUE
#     Brother Ercanbrack, CS235
# Author:
#     Yurii Vasiuk
# Summary:
#     Custome made queue and its application in the stock program
# Time:
#     15 hours
###################################################################

###################################################################
# The main rule
###################################################################
a.out: queue.h week03.o dollars.o stock.o
       g++ -o a.out week03.o dollars.o stock.o
       tar -cf week03.tar *.h *.cpp makefile

dollarsTest: dollars.o dollarsTest.cpp
       g++ -o dollarsTest dollars.o dollarsTest.cpp

###################################################################
# The individual components
#      week03.o       : the driver program
#      dollars.o      : the Dollars class
#      stock.o        : the logic for the stock program
###################################################################
week03.o: queue.h week03.cpp
       g++ -c week03.cpp

dollars.o: dollars.h dollars.cpp
       g++ -c dollars.cpp

stock.o: stock.h stock.cpp queue.h transaction.h
       g++ -c stock.cpp
```

dollars.h

```
/***********************************************************************
 * Header:
 *    DOLLARS
 * Summary:
 *    This file contains the notion of money
 * Author
 *    Br. Helfrich
 ************************************************************************/

#ifndef DOLLARS_H
#define DOLLARS_H

#include <iostream>  // for OSTREAM and ISTREAM

/****************************************
 * DOLLARS
 * This class behaves like a number except it handles
 * input and output for money different.
 * The following inputs are the same for example:
 *   -4 -4.0 (4.00) -$4.00 $-4
 ****************************************/
class Dollars
{
  public:
   // constructors
  Dollars()                       : cents(0)     {                       }
  Dollars(int cents)              : cents(cents) {                       }
  Dollars(double dollars)         : cents(0)     { *this = dollars;      }
  Dollars(const Dollars & dollars) : cents(0)    { *this = dollars;      }
```

```
   // operators
   Dollars & operator = (double dollars)
   {
      cents = (int)(dollars * 100.0);
      return *this;
   }
   Dollars & operator = (int dollars)
   {
      *this = (double)dollars;
      return *this;
   }
   Dollars & operator = (const Dollars & dollars)
   {
      cents = dollars.cents;
      return *this;
   }
   Dollars operator - (const Dollars & rhs) const
   {
      return Dollars(cents - rhs.cents);
   }
   Dollars operator * (int value) const
   {
      return Dollars(cents * value);
   }
   Dollars operator * (double value) const
   {
      return Dollars((int)((double)cents * value));
   }
   Dollars operator + (const Dollars & rhs) const
   {
      return Dollars(cents + rhs.cents);
   }
   Dollars & operator += (const Dollars & rhs)
   {
      return *this = *this + rhs;
   }
   bool operator == (const Dollars & rhs) const
   {
      return this->cents == rhs.cents;
   }
   bool operator != (const Dollars & rhs) const
   {
      return !(*this == rhs);
   }
   bool operator > (const Dollars & rhs) const
   {
      return this->cents > rhs.cents;
   }
   bool operator >= (const Dollars & rhs) const
   {
      return *this > rhs || *this == rhs;
   }
   bool operator < (const Dollars & rhs) const
   {
      return !(*this >= rhs);
   }
   bool operator <= (const Dollars & rhs) const
   {
      return !(*this > rhs);
   }

   // input and output
   friend std::ostream & operator << (std::ostream & out, const Dollars & rhs);
   friend std::istream & operator >> (std::istream & in,        Dollars & rhs);

  private:
    int cents;  // more accurate than floating point numbers; no errors!
};

#endif // DOLLARS_H
```

queue.h

```
/**********************************************************************
* Header:
*    Queue
* Summary:
```

```
*    Custome made queue, analogous to the STD queue.
*    (the class is implemented with size and capacity,
*     and with back pointing at the next element after the last one)
* Author
*    Yura Vasiuk
************************************************************************/

#ifndef QUEUE_H
#define QUEUE_H

#include <cassert>

/*********************************************
* QUEUE
* Custom made queue
*********************************************/
template <typename T>
class Queue
{
public:
   // default constructor : empty and kinda useless
   Queue() : _numItems(0), _capacity(0), _front(0), _back(0), _data(0x00000000) {} // done

   // copy constructor : copy it
   Queue(const Queue & rhs) throw (const char *);

   // non-default constructor : pre-allocate
   Queue(int _capacity) throw (const char *);

   // destructor : free everything
   ~Queue()        { if (_capacity) delete[] _data; }                // done

   // assignment operator
   Queue <T> & operator=(const Queue <T> & rhs) throw (const char *);

   // is the queue currently empty
   bool empty() const  { return _numItems == 0; }                   // done

   // remove all the items from the container
   void clear()        { _numItems = 0; _front = 0; _back = 0; }    // done

   // how many items are currently in the queue?
   int size() const    { return _numItems; }                        // done

   // what is the capasity?
   int capacity() const { return _capacity; }                       // done

   // add an item to the end of the queue
   void push(const T & t) throw (const char *);

   // remove an element from the beginning of the queue
   void pop() throw (const char *);

   // access to the first element
   T & front() throw (const char *);

   // access to the last element
   T & back() throw (const char *);


private:
   T * _data;          // dynamically allocated array of T
   int _front;         // the index of the first element
   int _back;          // the index of the next after the last element
   int _numItems;      // how many items are currently in the Container?
   int _capacity;      // how many items can I put on the Container before full?

   // reallocate (this work on the current object)
   void realloc();
};

/*********************************************
* QUEUE : NON-DEFAULT CONSTRUCTOR
* Preallocate the queue to "capacity"
*********************************************/
template <typename T>
Queue <T>::Queue(int capacity) throw (const char *)
{
   assert(capacity >= 0);
```

```
   // front and back do not depend on the capacity
   _front = _back = 0;

   // do nothing if there is nothing to do
   if (capacity == 0)
   {
      _capacity = _numItems = 0;
      _data = 0x00000000;
      return;
   }

   // attempt to allocate
   try
   {
      _data = new T[capacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }

   // copy over the stuff
   _capacity = capacity;
   _numItems = 0;
}

/*****************************************
* QUEUE :: COPY CONSTRUCTOR
*****************************************/
template <typename T>
Queue <T>::Queue(const Queue <T> & rhs) throw (const char *)
{
   assert(rhs._capacity >= 0);

   // do nothing if there is nothing to do
   if (rhs._capacity == 0)
   {
      _capacity = _numItems = _front = _back = 0;
      _data = 0x00000000;
      return;
   }

   // attempt to allocate
   try
   {
      _data = new T[rhs._capacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer";
   }

   // copy over the stuff (with shifting the data down to front = 0)
   assert(rhs._numItems >= 0 && rhs._numItems <= rhs._capacity);
   _capacity = rhs._capacity;
   _numItems = rhs._numItems;
   // copy the elements
   int j = rhs._front;
   for (int i = 0; i < _numItems; i++, j = (j + 1) % _capacity)
   {
      _data[i] = rhs._data[j];
   }

   // assign front and back of the new queue
   _front = 0;
   _back = _numItems;
}

/*****************************************
* QUEUE :: ASSIGNMENT OPERATOR
* Copy items from one queue to another.  Return the new queue by reference!!
*****************************************/
template <typename T>
Queue <T> & Queue <T>::operator=(const Queue <T> & rhs) throw (const char *)
{
   assert(rhs._capacity >= 0);

   // attempt to allocate in case the new object is smaller than the old one
```

```
   if (_capacity < rhs._capacity)
   {
      try
      {
         delete[] _data;  // prevents memory leak!!!
         _data = new T[rhs._capacity];
      }
      catch (std::bad_alloc)
      {
         throw "ERROR: Unable to allocate buffer";
      }
   }

   // copy over the stuff (with shifting the data down to front = 0)
   assert(rhs._numItems >= 0 && rhs._numItems <= rhs._capacity);
   _capacity = rhs._capacity;
   _numItems = rhs._numItems;
   // copy the elements
   int j = rhs._front;
   for (int i = 0; i < _numItems; i++, j = (j + 1) % _capacity)
   {
      _data[i] = rhs._data[j];
   }

   // assign front and back of the new queue
   _front = 0;
   _back = _numItems;

   return *this;
}

/*****************************************
* QUEUE :: REALLOC
* This reallocates the current object to a twice bigger (tor using in the push())
*****************************************/
template <typename T>
void Queue <T>::realloc()
{
   // temporary holders
   T* temp = _data;
   int tempCapacity = _capacity;

   // allocate a new queue
   _capacity *= 2;
   try
   {
      _data = new T[_capacity];
   }
   catch (std::bad_alloc)
   {
      throw "ERROR: Unable to allocate buffer for queue";
   }

   // copy the elements
   int j = _front;
   for (int i = 0; i < _numItems; i++, j = (j + 1) % tempCapacity)
   {
      _data[i] = temp[j];
   }

   // free the temp memory
   delete[] temp;

   // assign front and back of the new queue
   _front = 0;
   _back = _numItems;
}

/*************************************************
* QUEUE :: PUSH
* Push an item on the end of the queue
*************************************************/
template <typename T>
void Queue <T>::push(const T & t) throw (const char *)
{
   if (_capacity == 0) // case 1 (0 capacity)
   {
      _capacity = 2;
      //_front = 0;
```

```
      //_back = 0;
      _data = new T[_capacity];
      _data[_back++] = t;
      _numItems++;
   }
   else                // case 2
   {
      if (_capacity == _numItems)
         realloc();
      _data[_back] = t;
      _back = (_back + 1) % _capacity;
      _numItems++;
   }
}

/*************************************************
* QUEUE :: POP
* Pop an item from the beginning of the queue
*************************************************/
template <typename T>
void Queue <T>::pop() throw (const char *)
{
   if (empty())
      throw "ERROR: attempting to pop from an empty queue";
   else
   {
      _front = (_front + 1) % _capacity;
      _numItems--;
   }
}

/*************************************************
* QUEUE :: FRONT
* Return an item from the beginning of the queue
*************************************************/
template <typename T>
T & Queue <T>::front() throw (const char *)
{
   if (empty())
      throw "ERROR: attempting to access an item in an empty queue";
   else
      return _data[_front];
}

/*************************************************
* QUEUE :: BACK
* Return an item from the end of the queue
*************************************************/
template <typename T>
T & Queue <T>::back() throw (const char *)
{
   if (empty())
      throw "ERROR: attempting to access an item in an empty queue";
   else
      return _data[_back];
}

#endif // QUEUE_H
```

stock.h

```
/***********************************************************************
 * Header:
 *    STOCK
 * Summary:
 *    This will contain just the prototype for stocksBuySell(). You may
 *    want to put other class definitions here as well.
 * Author
 *    Yurii Vasiuk
 ************************************************************************/

#ifndef STOCK_H
#define STOCK_H

#include "dollars.h"   // for Dollars defined in StockTransaction
#include "queue.h"     // for QUEUE
#include "transaction.h"
#include <iostream>    // for ISTREAM and OSTREAM
```

```
using namespace std;

// the interactive stock buy/sell function
void stocksBuySell();

// my code
class Stock
{
public:
   Stock(){}
   void buyStock(Transaction theTransaction);
   void sellStock(Transaction theTransaction);
   void displayStock();
private:
   // containers for the stock data
   Queue <Transaction> _boughtStock;
   Queue <Transaction> _soldStock;
   Dollars _proceeds;
};
#endif // STOCK_H
```

transaction.h

```
/***********************************************************************
* Header:
*    TRANSACTION
* Summary:
*    This will represent the data and operations of a transaction
* Author
*    Yurii Vasiuk
************************************************************************/
#ifndef TRANSACTION_H
#define TRANSACTION_H

#include "dollars.h"   // for Dollars defined in StockTransaction
#include <iostream>    // for ISTREAM and OSTREAM
using namespace std;

class Transaction
{
public:
   Transaction() {}
   //Transaction(int shares, Dollars price) : _shares(shares), _price(price) {}
   void display() { cout << _shares << " shares at " << _price << endl; }
   int & getShares() { return _shares; }
   Dollars & getPrice() { return _price; }
   void setShares(int shares) { _shares = shares; }
   void setPrice(Dollars price) { _price = price; }
private:
   int _shares;
   Dollars _price;
};

#endif // TRANSACTION_H
```

dollars.cpp

```
/***********************************************************************
 * Program:
 *    DOLLARS
 * Summary:
 *    This file contains the notion of money
 * Author
 *    Br. Helfrich
 ************************************************************************/

#include <iostream>  // for OSTREAM and ISTREAM
#include <cassert>   // for ASSERT
#include "dollars.h" // for the class definition
using namespace std;

/*****************************************
 * DOLLARS READ
 * This function reads dollars from the input stream:
 *     - skips leading white spaces
 *     - skips leading $ signs
 *     - only consumes two decimal places
```

```
 *     - negative values work with () or -
 * For example:
 *   $1.34     -->  134 cents
 *   -1.2      --> -120 cents
 *  $(4.211)   --> -421 cents
 *   -6        --> -600 cents
 ****************************************/
istream & operator >> (istream & in, Dollars & rhs)
{
   // initially zero
   rhs.cents = 0;
   if (in.fail())
      return in;

   // skip leading spaces and dollar signs;
   while (isspace(in.peek()) || in.peek() == '$')
      in.get();

   // is the next character a negative?
   bool negative = false;
   while ('-' == in.peek() || '(' == in.peek())
   {
      negative = true;
      in.get();
   }

   // consume digits, assuming they are dollars
   while (isdigit(in.peek()))
      rhs.cents = rhs.cents * 10 + (in.get() - '0');

   // everything up to here was dollars so multiply by 100
   rhs.cents *= 100;

   // did we get a decimal
   if ('.' == in.peek())
   {
      // consume the decimal
      in.get();

      // next digit is in the 10cent place if it exists
      if (isdigit(in.peek()))
         rhs.cents += (in.get() - '0') * 10;
      // the final digit is the 1cent place if it exists
      if (isdigit(in.peek()))
         rhs.cents += (in.get() - '0');
   }

   // take care of the negative stuff
   rhs.cents *= (negative ? -1 : 1);

   // see if there is a trailing )
   if (')' == in.peek())
      in.get();

   return in;
}

/****************************************
 * DOLLARS DISPLAY
 * This function displays dollars on the screen
 *     - All dollars are preceeded with $
 *     - Negative amounts have () rather than -
 *     - Exactly two decimal places are always shown
 * For example:
 *   124 cents   --> $1.24
 *   300 cents   --> $3.00
 *  -498 cents   --> $(4.98)
 ****************************************/
ostream & operator << (ostream & out, const Dollars & rhs)
{
   // units
   out << '$';
   int cents = rhs.cents;

   // negative?
   if (rhs.cents < 0)
   {
      out << '(';
      cents *= -1;
```

```
   }
   assert(cents >= 0);

   // dollars
   out << cents / 100;

   // cents
   out << '.'
       << (cents % 100 < 10 ? "0" : "")
       << cents % 100;

   // negative?
   if (rhs.cents < 0)
      out << ')';

   return out;
}
```

dollarsTest.cpp

```
/**********************************************************************
 * Program:
 *    DOLLARS TEST
 * Summary:
 *    This file will test the Dollars class
 * Author
 *    Br. Helfrich
 **********************************************************************/

#include <iostream>
#include "dollars.h"
using namespace std;

/*************************************
 * MAIN - simple driver program
 *************************************/
/*
int main()
{
   // for ever
   while (true)
   {
      Dollars d;
      cout << "> ";
      cin  >> d;
      cout << '\t' << d << endl;
   }

   return 0;
}
*/
```

stock.cpp

```
/**********************************************************************
 * Implementation:
 *    STOCK
 * Summary:
 *    This will contain the implementation for stocksBuySell() as well
 *    as any other function or class implementation you need
 * Author
 *    Yurii Vasiuk
 **********************************************************************/

#include <iostream>    // for ISTREAM, OSTREAM, CIN, and COUT
#include <string>      // for STRING
#include <cassert>     // for ASSERT
#include "stock.h"     // for STOCK_TRANSACTION
#include "queue.h"     // for QUEUE
using namespace std;
#include <sstream>
/*********************************************
 * STOCKS BUY SELL
 * The interactive function allowing the user to
 * buy and sell stocks
 *********************************************/
void stocksBuySell()
```

Commented [ES1]: You overcomplicated this application.
Especially the read process.
The dollars object had the >> and << overloaded for easy reading and writing of the price. It handles the $ being present or not and handles decimal points.

You really didn't need to do the substring stuff and in the process the data was not read in properly.

```
{
   // instructions
   cout << "This program will allow you to buy and sell stocks. "
      << "The actions are:\n";
   cout << "  buy 200 $1.57   - Buy 200 shares at $1.57\n";
   cout << "  sell 150 $2.15  - Sell 150 shares at $2.15\n";
   cout << "  display         - Display your current stock portfolio\n";
   cout << "  quit            - Display a final report and quit the program\n";

   // your code here...
   cin.clear();
   cin.ignore(256, '\n');

   // make working objects
   Dollars price;
   Transaction myTransaction;
   Stock myStock;

   // get the user input and put it into the right container
   string input = " ";              // for getting the input
   string command, sharesS, priceS; // temporary holders
   int shares;                      // converted number of shares
   double priceD;                   // temporary holder

   while (input != "quit")
   {
      // get the user input
      cout << "> ";
      getline(cin, input);

      // parse the input
      command = input.substr(0, input.find(" "));
      if (command == "buy" || command == "sell")
      {
         sharesS = input.substr(input.find(" "), input.find(" "));
         priceS = input.substr(input.find("$") + 1);
      }
    /*
      shares = atoi(sharesS.c_str());
      priceD = atod(priceS.c_str());
    */
     istringstream buffer(sharesS);
     buffer >> shares;
     istringstream buffer1(priceS);
     buffer1 >> priceD;

      price = priceD;
      myTransaction.setShares(shares);
      myTransaction.setPrice(price);
      // myTransaction.display();  used for debugging

      if (command == "buy")
         myStock.buyStock(myTransaction);
      if (command == "sell")
         myStock.sellStock(myTransaction);
      if (command == "display")
         myStock.displayStock();
   }
}

/************************************************
* STOCK :: BUYSTOCK
* Add the transation to the transaction queue
************************************************/
void Stock::buyStock(Transaction theTransaction)
{
   _boughtStock.push(theTransaction);
}

/************************************************
* STOCK :: SELLSTOCK
* Subtract the transaction while adding the procceds
************************************************/
void Stock::sellStock(Transaction theTransaction)
{
   int trShares = theTransaction.getShares();
   Dollars trPrice = theTransaction.getPrice();
   int stShares = _boughtStock.front().getShares();
   Dollars stPrice = _boughtStock.front().getPrice();
```

Commented [ES2]: Instead of reading in an entire line.
Why not:

```
cin >> input; // buy,sell,display,quit

if (input == “buy” || input == “sell”)
{
  cin >> shares;
  cin >> price;  // >> is overloaded
                 // for dollars
}
```

```
   Transaction transactionSold;

   // case 1 and 2 -- less than or equal
   if (trShares < stShares || trShares == stShares)
   {
      // calculate the transaction profit
      Dollars profit = 0;
      Dollars oneShareProfit = stPrice - trPrice;
      for (int i = 0; i < trShares; i++)
         profit += oneShareProfit;
      // add to the proceeds
      _proceeds += profit;
      // case 1 -- less
      if (trShares < stShares)
      {
         // subtract the sold shares
         int newShares = stShares - trShares;
         // set the bought and sold queues
         _boughtStock.front().setShares(newShares);
         transactionSold.setShares(trShares);
         transactionSold.setPrice(trPrice);
         _soldStock.push(transactionSold);
      }
      // case 2 -- equal
      if (trShares == stShares)
      {
         _boughtStock.pop();
         _soldStock.push(transactionSold);
      }
   }

   // case 3 -- more than
   // !!! to finish
}

/************************************************
* STOCK :: DISPLAYSTOCK
* Add a transation to the transaction queue
************************************************/
void Stock::displayStock()
{
   Transaction displayTransaction;
   cout << "Currently held:\n";
   Queue <Transaction> displayQueue(_boughtStock);  // make a copy
   // use the copy -- display and pop
   while (!displayQueue.empty())
   {
      cout << "\tBought ";
      displayTransaction = _boughtStock.front();
      displayTransaction.display();
      _boughtStock.pop();  // !!! bug in looping and wrong number of shares displayed
   }

   cout << "Sell history:\n";
   // !!! to finish
   cout << "Proceeds: " << _proceeds << endl;
   // !!! to finish
}
```

week03.cpp

```
/**********************************************************************
* Program:
*    Week 03, Queue
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise the Queue class.  When you
*    submit your program, this should not be changed in any way.  That being
*    said, you may need to modify this once or twice to get it to work.
**********************************************************************/

#include <iostream>    // for CIN and COUT
#include <string>      //
```

```
#include "queue.h"     // your Queue class should be in queue.h
#include "stock.h"     // your stocksBuySell() function
#include "dollars.h"   // for the Dollars class
using namespace std;


// prototypes for our four test functions
void testSimple();
void testPushPopTop();
void testCircular();
void testErrors();

// To get your program to compile, you might need to comment out a few
// of these. The idea is to help you avoid too many compile errors at once.
// I suggest first commenting out all of these tests, then try to use only
// TEST1.  Then, when TEST1 works, try TEST2 and so on.
#define TEST1   // for testSimple()
#define TEST2   // for testPushPopTop()
#define TEST3   // for testCircular()
#define TEST4   // for testErrors()

/*********************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 *********************************************************************/
int main()
{
   // menu
   cout << "Select the test you want to run:\n";
   cout << "\t1. Just create and destroy a Queue\n";
   cout << "\t2. The above plus push, pop, and top\n";
   cout << "\t3. The above plus test implementation of the circular Queue\n";
   cout << "\t4. Exercise the error handling\n";
   cout << "\ta. Selling Stock\n";

   // select
   char choice;
   cout << "> ";
   cin  >> choice;
   switch (choice)
   {
      case 'a':
         stocksBuySell();
         break;
      case '1':
         testSimple();
         cout << "Test 1 complete\n";
         break;
      case '2':
         testPushPopTop();
         cout << "Test 2 complete\n";
         break;
      case '3':
         testCircular();
         cout << "Test 3 complete\n";
         break;
      case '4':
         testErrors();
         cout << "Test 4 complete\n";
         break;
      default:
         cout << "Unrecognized command, exiting...\n";
   }

   return 0;
}

/****************************************
 * TEST SIMPLE
 * Very simple test for a Queue: create and destroy
 ****************************************/
void testSimple()
{
#ifdef TEST1
   try
   {
      // Test 1.a: bool Queue with default constructor
      cout << "Create a bool Queue using default constructor\n";
      Queue <bool> q1;
```

```
      cout << "\tSize:     " << q1.size()                  << endl;
      cout << "\tCapacity: " << q1.capacity()              << endl;
      cout << "\tEmpty?    " << (q1.empty() ? "Yes" : "No") << endl;

      // Test 1.b: double Queue with non-default constructor
      cout << "Create a double Queue using the non-default constructor\n";
      Queue <double> q2(10 /*capacity*/);
      cout << "\tSize:     " << q2.size()                  << endl;
      cout << "\tCapacity: " << q2.capacity()              << endl;
      cout << "\tEmpty?    " << (q2.empty() ? "Yes" : "No") << endl;

      // Test 1.c: copy the Queue using the copy constructor
      {
         cout << "Create a double Queue using the copy constructor\n";
         Queue <double> q3(q2);
         cout << "\tSize:     " << q3.size()                  << endl;
         cout << "\tCapacity: " << q3.capacity()              << endl;
         cout << "\tEmpty?    " << (q3.empty() ? "Yes" : "No") << endl;
      }

      // Test 1.d: copy the Queue using the assignment operator
      cout << "Copy a double Queue using the assignment operator\n";
      Queue <double> q4(2);
      q4 = q2;
      cout << "\tSize:     " << q4.size()                  << endl;
      cout << "\tCapacity: " << q4.capacity()              << endl;
      cout << "\tEmpty?    " << (q4.empty() ? "Yes" : "No") << endl;
   }
   catch (const char * sError)
   {
      cout << sError << endl;
   }
#endif //TEST1
}

#ifdef TEST2
/*****************************************
 * DISPLAY
 * Display the contents of the queue
 *****************************************/
template <class T>
ostream & operator << (ostream & out, Queue <T> q)
{

   out << "{ ";
   while (!q.empty())
   {
      out << q.front() << ' ';
      q.pop();
   }
   out << '}';

   return out;
}
#endif // TEST2

/*****************************************
 * TEST PUSH POP TOP
 * Add a whole bunch of items to the Queue.  This will
 * test the Queue growing algorithm
 *****************************************/
void testPushPopTop()
{
#ifdef TEST2
   try
   {
      // create
      Queue <Dollars> q1;
      Dollars noMoney;

      // fill
      cout << "Enter money amounts, type $0 when done\n";
      Dollars money;
      do
      {
         cout << "\t" << q1 << " > ";
         cin  >> money;
         if (money != noMoney)
            q1.push(money);
```

```
      }
      while (money != noMoney);

      // display how big it is
      cout << "\tSize:     " << q1.size()                  << endl;
      cout << "\tEmpty?    " << (q1.empty() ? "Yes" : "No") << endl;
      cout << "\tCapacity: " << q1.capacity()              << endl;

      // make a copy of it using the assignment operator and copy constructor
      Queue <Dollars> q2(2);
      q2 = q1;
      Queue <Dollars> q3(q1);

      // destroy the old copy
      q1.clear();

      // display the two copies
      cout << "\tq1 = " << q1 << endl;
      cout << "\tq2 = " << q2 << endl;
      cout << "\tq3 = " << q3 << endl;
   }
   catch (const char * sError)
   {
      cout << sError << endl;
   }

#endif // TEST2
}

/*****************************************
 * TEST CIRCULAR
 * This will test whether the circular aspect
 * of the Queue is working correctly
 *****************************************/
void testCircular()
{
#ifdef TEST3
   // create
   cout << "Create a string Queue with the default constructor\n";
   Queue <string> q(4);

   // instructions
   cout << "\tTo add the word \"dog\", type +dog\n";
   cout << "\tTo pop the word off the queue, type -\n";
   cout << "\tTo display the state of the queue, type *\n";
   cout << "\tTo quit, type !\n";

   // interact
   char instruction;
   string word;
   try
   {
      do
      {
         cout << "\t" << q << " > ";
         cin  >> instruction;
         switch (instruction)
         {
            case '+':
               cin >> word;
               q.push(word);
               break;
            case '-':
               q.pop();
               break;
            case '*':
               cout << "Size:     " << q.size()                  << endl;
               cout << "Empty?    " << (q.empty() ? "Yes" : "No") << endl;
               cout << "Capacity: " << q.capacity()              << endl;
               break;
            case '!':
               break;
            default:
               cout << "Invalid command\n";
         }
      }
      while (instruction != '!');
   }
   catch (const char * error)
```

```
   {
      cout << error << endl;
   }

   // verify that copy works as we expect
   Queue <string> qCopy(q);
   assert(q.size()     == qCopy.size()    );
   assert(q.empty()    == qCopy.empty()   );
   while (!q.empty())
   {
      assert(q.front() == qCopy.front());
      assert(q.back()  == qCopy.back() );
      assert(q.size()  == qCopy.size() );
      q.pop();
      qCopy.pop();
   }
#endif // TEST3
}

/*****************************************
 * TEST ERRORS
 * Numerous error conditions will be tested
 * here, including bogus popping and the such
 *****************************************/
void testErrors()
{
#ifdef TEST4
   // create
   Queue <char> q;

   // test using front() with an empty queue
   try
   {
      q.front();
      cout << "BUG! We should not be able to front() with an empty queue!\n";
   }
   catch (const char * error)
   {
      cout << "\tQueue::front() error message correctly caught.\n"
           << "\t\"" << error << "\"\n";
   }

   // test using back() with an empty queue
   try
   {
      q.back();
      cout << "BUG! We should not be able to back() with an empty queue!\n";
   }
   catch (const char * error)
   {
      cout << "\tQueue::back() error message correctly caught.\n"
           << "\t\"" << error << "\"\n";
   }

   // test using pop() with an empty queue
   try
   {
      q.pop();
      cout << "BUG! We should not be able to pop() with an empty queue!\n";
   }
   catch (const char * error)
   {
      cout << "\tQueue::pop() error message correctly caught.\n"
           << "\t\"" << error << "\"\n";
   }
#endif // TEST4
}
```

Test Bed Results

```
terminate called after throwing an instance of 'char const*'
---------------
Starting Test 1

   > Select the test you want to run:
   >    1. Just create and destroy a Queue
   >    2. The above plus push, pop, and top
   >    3. The above plus test implementation of the circular Queue
```

```
   >    4. Exercise the error handling
   >    a. Selling Stock
   > > 1
Create, destory, and copy a Queue
   > Create a bool Queue using default constructor
   >    Size:     0
   >    Capacity: 0
   >    Empty?    Yes
   > Create a double Queue using the non-default constructor
   >    Size:     0
   >    Capacity: 10
   >    Empty?    Yes
   > Create a double Queue using the copy constructor
   >    Size:     0
   >    Capacity: 10
   >    Empty?    Yes
   > Copy a double Queue using the assignment operator
   >    Size:     0
   >    Capacity: 10
   >    Empty?    Yes
   > Test 1 complete

Test 1 passed.
----------------------------------------------------------------

----------------------------------------------------------------
Starting Test 2

   > Select the test you want to run:
   >    1. Just create and destroy a Queue
   >    2. The above plus push, pop, and top
   >    3. The above plus test implementation of the circular Queue
   >    4. Exercise the error handling
   >    a. Selling Stock
   > > 2
Create a Dollars Queue with the default constructor\n
   > Enter money amounts, type $0 when done
Initially an empty queue
   >    { } > $1
First resize should set the capacity to one
   >    { $1.00 } > -2
Now the capacity should be two
   >    { $1.00 $(2.00) } > 3.0
Double again to four
   >    { $1.00 $(2.00) $3.00 } > (4.1)
   >    { $1.00 $(2.00) $3.00 $(4.10) } > $(-5.21)
Double again to 8
   >    { $1.00 $(2.00) $3.00 $(4.10) $(5.21) } > 0
   >    Size:     5
   >    Empty?    No
   >    Capacity: 8
When we clear q1, then q2 and q3 should keep their data
   >    q1 = { }
   >    q2 = { $1.00 $(2.00) $3.00 $(4.10) $(5.21) }
   >    q3 = { $1.00 $(2.00) $3.00 $(4.10) $(5.21) }
   > Test 2 complete

Test 2 passed.
----------------------------------------------------------------

----------------------------------------------------------------
Starting Test 3

   > Select the test you want to run:
   >    1. Just create and destroy a Queue
   >    2. The above plus push, pop, and top
   >    3. The above plus test implementation of the circular Queue
   >    4. Exercise the error handling
   >    a. Selling Stock
   > > 3
   > Create a string Queue with the default constructor
   >    To add the word "dog", type +dog
   >    To pop the word off the queue, type -
   >    To display the state of the queue, type *
   >    To quit, type !
Initially the capacity should be 4 so no resizing is needed here
   >    { } > +alfa
   >    { alfa } > +bravo
   >    { alfa bravo } > +charlie
```

```
   >    { alfa bravo charlie } > +delta
   >    { alfa bravo charlie delta } > *
   > Size:     4
   > Empty?    No
   > Capacity: 4
Next we will make room for two items on the front
   >    { alfa bravo charlie delta } > -
   >    { bravo charlie delta } > -
   >    { charlie delta } > *
   > Size:     2
   > Empty?    No
   > Capacity: 4
This tests the circular queue. We will add two items which will wrap around
   >    { charlie delta } > +echo
   >    { charlie delta echo } > +foxtrot
   >    { charlie delta echo foxtrot } > *
Now our capacity should be unchagned (there are only 3 items on the queue),
but we are wrapping around using the circular queue
   > Size:     4
   > Empty?    No
   > Capacity: 4
Next we will empty the queue
   >    { charlie delta echo foxtrot } > -
   >    { delta echo foxtrot } > -
   >    { echo foxtrot } > -
   >    { foxtrot } > -
   >    { } > *
The capacity should remain at 4 even though the size is zero
   > Size:     0
   > Empty?    Yes
   > Capacity: 4
From here we will fill the queue with four items
   >    { } > +hotel
   >    { hotel } > +india
   >    { hotel india } > +juliett
   >    { hotel india juliett } > +kilo
   >    { hotel india juliett kilo } > -
Now we will get into the wrapped condition of the circular queue
   >    { india juliett kilo } > +lima
   >    { india juliett kilo lima } > *
   > Size:     4
   > Empty?    No
   > Capacity: 4
Now we will add a fifth item. This requires a resize.
Our resize code must know how to deal with a wrapping condition
   >    { india juliett kilo lima } > +mike
   >    { india juliett kilo lima mike } > *
   > Size:     5
   > Empty?    No
   > Capacity: 8
   >    { india juliett kilo lima mike } > -
   >    { juliett kilo lima mike } > -
   >    { kilo lima mike } > *
   > Size:     3
   > Empty?    No
   > Capacity: 8
   >    { kilo lima mike } > !
   > Test 3 complete

Test 3 passed.
---------------------------------------------------------------


---------------------------------------------------------------
Starting Test 4

   > Select the test you want to run:
   >    1. Just create and destroy a Queue
   >    2. The above plus push, pop, and top
   >    3. The above plus test implementation of the circular Queue
   >    4. Exercise the error handling
   >    a. Selling Stock
   > > 4

Test front() from an empty queue
   >    Queue::front() error message correctly caught.
   >    "ERROR: attempting to access an item in an empty queue"

Test back() from an empty queue
   >    Queue::back() error message correctly caught.
```

```
   >    "ERROR: attempting to access an item in an empty queue"

Test pop() from an empty queue
   >    Queue::pop() error message correctly caught.
   >    "ERROR: attempting to pop from an empty queue"
   > Test 4 complete

Test 4 passed.
------------------------------------------------------------

------------------------------------------------------------
Starting Test 5

   > Select the test you want to run:
   >    1. Just create and destroy a Queue
   >    2. The above plus push, pop, and top
   >    3. The above plus test implementation of the circular Queue
   >    4. Exercise the error handling
   >    a. Selling Stock
   > > a
   > 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 two batches
   > > buy 100 $2.00
   > > buy 400 $3.00
   > > display
   > Currently held:
   > \tBought 100 shares at $2.00\n
Exp: \tBought 100 shares at $2.00\n
   > \tBought 400 shares at $3.00\n
Exp: \tBought 400 shares at $3.00\n
   > \tBought
Exp: Proceeds: $0.00\n

Sell 150 which will require taking 100 from the first batch
and 50 from the second
   >
Exp: >
sell 150 5.5
```

Grading Criteria

Criteria	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%	Weight	Score
Queue interface	The interfaces are perfectly specified with respect to const, pass-by-reference, etc.	week03.cpp compiles 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 the Queue class do not resemble the problem definition	20	
Queue Implementation	Passes all four Queue 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	20	
Stock	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 problem was not attempted	30	
Code Quality	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	20	
Style	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	10	
Total	Test bed errors on Stocks, threw an exception when executing display() function. The # of shares are wrong and price was zero.						84/100

vas14001@byui.edu