Project 8: Queue

A generic Queue data structure

Revision dated 01/04/18

Educational Objectives: After completing this assignment the student should have the following knowledge, ability, and skills:

- Define the concept of Generic Container
- Define the ADT Queue with elements of type T
- Give examples of the use of Queue in computing
- Describe an implementation plan for generic Queue as a class template `Queue<T>` based on dynamically allocated links containing `T` objects
- Code the implementation for `Queue<T>` using the plan
- Describe an implementation plan for `Stack<T,N>` based on dynamically allocated links containing `T` objects
- Explain why it is impractical to implement `Queue<T>` using an array

Operational Objectives: Create the generic container class `fsu::Queue<T>` that satisfies the interface requirements given below, along with an appropriate test harness for the class.

Deliverables: Four files `tqueue.h`, `fqueue.cpp`, `makefile`, and `log.txt`.

Assessment Rubric

```
    =================================================
    Rubric to be used in Assessment           P7  P8
    -------------------------------------------------
    test builds:    [0..2 pts each]
      fstack.x [student harness]               x
      fstack_char.x                            x
      fstack_int.x                             x
      fstack_String.x                          x
      fstack_T.x                               x
      fqueue.x [student harness]                   x
      fqueue_char.x                                x
      fqueue_int.x                                 x
      fqueue_String.x                              x
      fqueue_T.x                                   x
    build in2post.x [0..5 pts each]            x   x
    tests:          [0..5 pts each]
      fstack_char.x                            x
      fstack_int.x                             x
      fstack_String.x                          x
      fstack_T.x                               x
      fqueue_char.x                                x
      fqueue_int.x                                 x
      fqueue_String.x                              x
      fqueue_T.x                                   x
      in2post.x                                x   x
    log.txt                   [-25..5] each:   x   x
    code quality              [-25..5] each:   x   x
    dated submission deduction [2 pts each]: ( 0)( x)
                                              --  --
    total P7                        [0..50]:  xx      <-- cannot excede 40 without proj8 and in2post
    total P8                        [0..50]:      xx
    =================================================
```

Note 1: Projects 7 and 8 will be assessed together for a total of 100 points for both projects. The above shows the weighting of the two parts of the coupled assignment. Project 7 alone can only achieve 40 points, because in2post requires interaction between stacks and queues.

Note 2: Code Quality and Log entries will be assessed for both Stack and Queue simultaneously. Be sure that your log.txt covers both projects in the same log.

Abstract Data Types

An *abstract data type*, abbreviated ADT, consists of three things:

1. A set of elements of some type `T`
2. Operations that may modify the set or return values associated with the set
3. Rules of behavior, or axioms, that determine how the operations interact

The operations and axioms together should determine a unique character for the ADT, so that any two implementations should be essentially equivalent. (The word *isomorphic* is used to give precision to "essentially equivalent". We'll look at this in the next course.)

Queues

The *queue* ADT is used in many applications and has roots that pre-date the invention of high-level languages. Conceptually, queue is a set of data that can be expanded, contracted, and accessed using very specific operations. The queue ADT models the "FIFO", or first-in, first-out rule. The actual names for the queue operations may vary somewhat from one description to another, but the behavior of the abstract queue operations is well known and unambiguously understood throughout computer science. Queues are important in many aspects of computing, ranging from hardware design and I/O to inter-machine communication and algorithm control structures.

Typical uses of ADT Queue are (1) buffers, without which computer communication would be impossible, (2) control of algorithms such as breadth first search, and (3) simulation modelling of systems as diverse as manufacturing facilities, customer service, and computer operating systems.

Abstract Queue Interface

The queue abstraction has the following operations and behavior:

- **`Push(t)`** Inserts the element `t` into the queue
- **`Pop()`** Removes the first-inserted element; undefined when queue is empty
- **`Front()`** Returns the first-inserted element; undefined when queue is empty
- **`Empty()`** Returns true iff the queue has no elements
- **`Size()`** Returns the number of elements in the queue

Application: Converting Infix to Postfix Notation

As one example of the use of ADTs in computing, consider the following function that illustrates an algorithm for converting arithmetic expressions from infix to postfix notation:

```
...
#include <tqueue.h>
#include <tstack.h>
...
typedef fsu::Queue < Token > TokenQueue;
typedef fsu::Stack < Token > TokenStack;
// a Token is either an operand, an operator, or a left or right parenthesis
...
bool i2p (TokenQueue & Q1, TokenQueue & Q2)
// converts infix expression in Q1 to postfix expression in Q2
// returns true on success, false if syntax error is encountered
{
   ...
   TokenStack S;               // algorithm control stack
   Q2.Clear();                 // make sure ouput queue is empty
   while (!Q1.Empty())
   {
     // take tokens off Q1 and use them to build Q2
     // if syntax error is detected return false
  }
   return true;
}  // end i2p()
```

This is a complex algorithm, but not beyond your capability to understand. The main points to displaying it here are: (1) to illustrate how stacks and queues may be used in implementing applications, and (2) to let you know that your code must be compatible with such apps. in2post.cpp is one of our tests!

We will implement the queue abstraction as a C++ class template `Queue` that contains the Queue ADT operations in its public interface.

Procedural Requirements

1. Continue your `log.txt` from project 7 (Stack). This log should serve for both projects and will be assessed when Stack and Queue are assessed together.

2. Create and work within a separate subdirectory `cop3330/proj8`. Review the COP 3330 rules found in Introduction/Work Rules.

3. After starting your log, copy the following files from the course directory `[LIB]` into your `proj8` directory:

   ```
   proj8/in2post.cpp
   proj8/constTest.cpp
   proj8/deliverables.sh
   area51/in2post*.x
   area51/fqueue*.x
   ```

4. Define and implement the class template `fsu::Queue<T>` in the file `tqueue.h` according to the RingBuffer implementation plan. Be sure to make log entries for your work sessions.

5. Devise a test client for `Queue<T>` that exercises the Queue interface (as well as the RingBuffer enhancements) for at least one native type and one user-defined type `T`. Put this test client in the file `fqueue.cpp`. Be sure to make log entries for your work

sessions.

6. Test `Stack` and `Queue` using the supplied application `in2post.cpp`. Again, make sure behavior is appropriate and make corrections if necessary. Log your activities.
7. Turn in `tqueue.h`, `fqueue.cpp`, `makefile`, and `log.txt` using the `submit.sh` submit script.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

RingBuffers

The term *ringbuffer* describes a specific type of implementation of the ADT **queue**.

ringbuffers are used in a number of applications, typically to store keyboard input temporarily "at" a terminal before a "send" command is issued. (The quotes are intended to convey that there is some ambiguity to the words inside them.) For example, if you establish a command prompt terminal to a Unix login using ssh, the operating system will need a way to store the characters you type as you type them prior to hitting the ENTER key. When ENTER is entered, this signals the OS to send the accumulated input to whatever application is expecting it.

Terminal support is just one of many applications of ringbuffers. They are useful just about anywhere an ADT **queue** is called for.

RingBuffer Interface

The ringbuffer interface is an ADT **queue** API, with some enhancements, as shown in the public sector of the class definition:

```
template < typename T >
class Queue // as RingBuffer
{
  public:
    // Queue API
    void     Push     (const T& t);
    T        Pop      ();
    T&       Front    ();
    const T& Front    () const;
    size_t   Size     () const;
    bool     Empty    () const;
    void     Clear    ();

    // extra goodies
    void     Append   (const Queue& q);           // append entire contents of q
    size_t   Capacity () const;                   // how many items can be stored using existing allocation
    void     Release  ();                         // de-allocate all links

    // proper type
    Queue   ();
    Queue   (const Queue&);
    ~Queue  ();
    Queue& operator = (const Queue&);

  public:
    void     Display    (std::ostream& os) const; // output queue contents through os - depends on ofc_
    void     SetOFC     (char ofc);
    void     Dump       (std::ostream& os) const; // output all private data (for development use only)

  private:
    char    ofc_;

  private:
    class Link
    {
      Link (const T& t) : element_(t), next_(nullptr){}
      T      element_;
      Link * next_;
      friend class Queue;
    };

    Link * firstUsed_; // head of queue
    Link * firstFree_; // one past last item in queue

    static Link* NewLink (const T& t);
};

template < typename T >
std::ostream& operator << (std::ostream& os, const Queue<T>& q)
{
  q.Display(os);
  return os;
}
```

Note that we have implemented the output operator in terms of the Display method. The private sector of the class defined above gives hints toward an implementation.

RingBuffer Implementation Plan

The RingBuffer implementation plan uses a collection of links created dynamically to store data, one data item per link, each link pointing to a following link. This aspect of the implementation is analagous to that of a linked list such as `fsu::List<>`. However, there is no "first" or "last" link - the links form a cycle, or ring, hence the name "ringbuffer".

The RingBuffer implementation maintains two pointers into this ring structure, "`firstUsed_`", which points to the link containing the first item in the queue, and "`firstFree_`" which points to the first link that is not currently used to store queue data. Items are removed from the "front" of the queue by simply advancing the `firstUsed_` pointer. Space is created for new items at the "back" of the queue by simply advancing the `firstFree_` pointer. (There's an exception to this - see below.)

We refer to the underlying ring of links as the *carrier ring*. At any given time, the carrier ring must have at least one link more than the size of the queue in order to distinguish between the "empty queue" and "full carrier" states. The "queue support" consists of the links from firstUsed_ to (but not including) firstFree_.

States and values that need special treatment are:

```
null carrier:  either or both queue pointers are nullptr
empty queue:   the carrier is null, or the first used link is the same as the first free link
full carrier:  the first free link is the only free link
size:          number of elements currently stored in the queue
capacity:      zero, or one less than the number of links in the carrier ring
```

Whenever the carrier is full, a `Push(t)` operation must create a new link for the new data item. However, `Pop()` operations do not release memory but just change the `firstUsed_` pointer. In this way, the carrier maintains capacity equal to the largest size the queue has attained since it was created (or since the last call to `Release()`). Similarly, `Clear()` does not de-allocate memory but just resets queue pointers to the empty queue state. Only the `Release()` method actually reduces the size of the carrier. This conservation of created links during operations that decrease the size of the queue leads to runtime efficiency in situations where a queue may exist for a long time but always maintain a modest size.

Implementation Notes

1. A null carrier ring is detected by `firstUsed_ == nullptr`.
2. A full carrier ring is detected by `firstFree_->next_ == firstUsed_` (i.e., there is only one free link).
3. Maintain the (non-null) carrier ring with at least one "unused" link.
4. There are three cases in implementing `Push(t)`. If the carrier ring is null, two links must be created, one to hold the item and one to be free. If the carrier ring is full, one new (free) link must be created. Otherwise, all that is needed is to "advance" the `firstFree_` pointer to the next link after assigning the data appropriately. To *advance* a link pointer `p` means going to the next link:

   ```
   p = p->next_
   ```
5. An empty queue is detected by either a null carrier ring or `firstFree_ == firstUsed_`.
6. If the queue is not empty, `Pop()` should "advance" the `firstUsed_` pointer, otherwise do nothing. Return the value that is "removed" from the queue.
7. `Clear()` does not change the carrier ring. Only the pointer member variables are changed to make the queue empty.
8. `Release()` actually de-allocates all of the dynamically allocated memory of the carrier ring and makes the carrier ring null. `Release()` is called by the destructor. It may also be called by client programs.
9. The assignment operator may be implemented with a call to either Release or Clear, followed by a call to Append. Using Release ensures that memory for the destination queue is just enough to hold a copy, whereas using Clear ensures that previously allocated memory is re-used.
10. Note that `Link` is defined only in the scope `Queue<T>::`. Also note that all members of class `Link` are private, which means a `Link` object can be created or accessed only inside an implementation of its friend class `Queue<T>`. (This prevents any client program from having access to a `Link` object.) The only method for class `Link` is its constructor, whose implementation should just initialze the two variables.
11. `Append(q)` just pushes the contents of `q`. It is useful in implementing the copy constructor and assignment operator.
12. The only data stored statically in a `Queue` object are the values of the two pointers `firstUsed_` and `firstFree_`. These in turn provide access to the carrier ring, which consists of a collection of `Link` objects (aka "links") arranged in a circular manner using the "next" pointers in the links.

Code Requirements and Specifications

1. Your `makefile` should build the target `fqueue.x`. If you have other targets to build, that is not a problem as long as this target is there.

2. `Queue` should implement the class template Queue as defined under RingBuffer Interface and Ringbuffer Implementation Plan above.

3. The `Queue` constructor should create an empty queue with no dynamic memory allocation.

4. The `Queue<T>::Push(t)` operation operates in three modes, depending on the state of the carrier ring:

   i. When the carrier ring is null, two links are created with `t` in the link pointed to by `firstUsed_`.
   ii. When the carrier ring is full, one new link is created, `t` is added to the queue by copying `t` into `firstFree_` before advancing `firstFree_`.
   iii. When the carrier ring has extra capacity, no new link is created and `t` is added to the queue by by copying `t` into `firstFree_` before advancing `firstFree_`.

5. The `Queue<T>::Pop()` operation does not de-allocate memory, it just removes the front of the queue from consideration by advancing `firstUsed_`.

6. As always, the class destructors should de-allocate all dynamic memory still owned by the object. The stack and queue implementations will be very different.

7. Use the RingBuffer implementation plan discussed above, and implement the complete RingBuffer API.

8. The `Display(os)` method is intended to regurgitate the contents out through the `std::ostream` object `os`. The second parameter `ofc` is a single *output formatting character* that has the default value `'\0'`. (The other three popular choices for `ofc` are `' '`, `'\t'` and `'\n'`.) The implementation of `Display` must recognize two cases:

   i. `ofc_ == '\0'`: send the contents to output with nothing between them
   ii. `ofc_ != '\0'`: send the contents to output with `ofc_` separating them

   Thus, for example, `q.Display(std::cout)` would send the contents of `q` to standard output.

   NOTE: '\0' is NOT nothing!

9. The output operator should be overloaded as follows:

   ```
   template < typename T >
   std::ostream& operator << (std::ostream& os, const Queue<T>& q)
   {
     q.Display (os);
     return os;
   }
   ```

   The overload of `operator <<()` should be placed in your queue header file immediately following the class definition.

10. The class `Queue` should be in the `fsu` namespace.

11. The file `tqueue.h` should be protected against multiple reads using the `#ifndef ... #define ... #endif` mechanism.

12. The test client program `fqueue.cpp` should adequately test the functionality of queue, including the output operator. It is your responsibility to create this test program and to use it for actual testing of your queue data structure.

Hints

- Your test client can be created by making very small changes in a copy of `fstack.cpp` you created as part of the preceding assignment.

- Keep in mind that the implementations of class template methods are in themselves template functions. For example, the implementation of the `Queue` method `Pop()` would look something like this:

  ```
  template < typename T >
  T Queue<T>::Pop()
  {
    // yada dada
    return ??;
  }
  ```

- We will test *your* implementation `tqueue.h` using (1) *our* test client `fqueue.cpp` and (2) `in2post` (with *your* stack implementation as submitted for Project 7).

- There are two versions of `Queue::Front()`. These are distunguished by "`const`" modifiers for one of the versions. The implementation code is identical for each version. The main point is that "`Front()`" can be called on a constant queue, but the returned reference may not be used to modify the front element. This nuance will be tested in our assessment. You can test it with two functions such as:

  ```
  char ShowFront(const fsu::Queue<char>& q)
  {
  ```

```
    return q.Front();
  }

  void ChangeFront(fsu::Queue<char>& q, char newFront)
  {
    q.Front() = newFront;
  }
```

Note that `ShowFront` has a `const` reference to a queue, so would be able to call the const version of `Front()` but not the non-const version, but that suffices. `ChangeFront` would need to call the non-const version in order to change the value at the front of the queue. A simple test named "`constTest.cpp`" is posted in the distribution directory.