**CSC2040 Data Structures and Algorithms, and Programming Languages**

**Practical 7**

Wednesday 29 November 2017

# Lists and Queues

The learning outcome of this practical is to be confident in writing your own class for a new data structure (a cyclic buffer for queues), and using this class for some test cases.

Create a new project called Practical7, with the file test.cpp for your main function.

**Part 1: A cyclic buffer implementation of a Queue of integers**

(1) Create a new class **CyclicQueue** (with files CyclicQueue.h and CyclicQueue.cpp) which provides a special implementation of a queue data structure for integers. The implementation should use a cyclic buffer (an array), as described in lectures.

The class header should contain the following data and function members (you don't need to use templates):

```
class CyclicQueue
{
public:
        CyclicQueue  (int maxSize)   // Constructor
        ~CyclicQueue  ()              // Destructor
        void addAtEnd(int i)        // Adds item to end of queue, if room
        int* removeFront()          // Returns pointer to front item, and removes it
        int* getAtFront()           // Returns pointer to front item, but doesn't remove it
        bool isEmpty();             // Returns true if the queue is currently empty
        int size();                 // Returns the number of items in the queue
private:
        int maxBufferSize;            // Size of the array
        int* buffer;                  // The array (buffer) itself
        int first, last;              // Positions of the first and last items in the buffer
        int numItems;                 // Number of items currently in the buffer
};
```

(2) Write a test function to test if your CyclicQueue class is working. Test it for the error cases – for example, try getting the front item when the queue is empty, and try adding when it is full.

(3) Recall that a cyclic buffer is useful for smoothing out the speeds of a producer process and a consumer process. Test your class and its implementation by a second test function which simulates the following sequence of events:

| | |
|---|---|
| Producer: | Adds 6 items (the integers 1..6) |
| Consumer: | Prints and removes the front item |
| Consumer: | Prints and removes the front 2 items |

| | |
|---|---|
| Producer: | Adds 6 items (the integers 7..12) |
| Consumer: | Prints and removes the front three items |
| Consumer: | Prints and removes the front item |
| Producer: | Adds 6 items (the integers 13..18) |
| Consumer: | Prints and removes the front four items |
| Consumer: | Flushes the buffer – prints and removes all remaining items |

This can be done just by a series of calls to addAtEnd and removeAtFront. Use loops rather than writing out each individual add and remove.

**Part 2: Extend the LinkedList class by adding a new member function**

Include the supplied template classes ListNode.h and LinkedList.h into your project Practical7.

Add a new public function to the LinkedList.h class:

int **countList** (T item)

This function should return the number of occurrences of ‘item’ in the list.

Write a test function in which define an integer-type LinkedList object *ilist*. Set *ilist* up by adding 100 random integer numbers into the list, each number with a value 0-9. Use this list to test if your revised class work properly, by counting the occurrences of 0-9 and 10-19, respectively, in the *ilist*. Note that the returned counts for 10-19 should all be zero, and the returned counts for 0-9 should total to 100.

**Part 3: Reading and building a linked list of words**

For this section, a sentence consists of a list of words separated by one or more spaces or a ‘,’, and terminated by a full stop. Write a function to read and create a linked list of words, each word being held in your list as a string. The code allows the user to type a sentence in a single line, e.g. *This is a pen, and that is a pencil.* The input is ended with a full stop, followed by hitting the Enter key.

Your function may take the following structure:

```
void testListOfWords()
{
        // a linked list of words
        LinkedList< (1) > words;

        // reading and set up the list terminated by a full stop
        char ch;
        string word = “”;
        while (ch = cin.get()) {
                if (ch == '.') break;
                /* (2) */
        }
}
```

The above function is incomplete:

- A data type is missing, at the part marked by (1).
- A section of code is missing, at the part marked by (2). This section of code will determine, for each input character *ch*, if it is a letter or a word separator. A letter will be added to the end of a string *word* for building up a word letter by letter, and a separator will be ignored. Finally, each complete *word* will be added to the list *words* to construct the word list.

Hints:

To add a character on to the end of a string word, just do “word = word + ch ; ”.

A character ch is a letter if ch >= ‘a’ && ch <= ‘z’.  And note upper case too!

Make sure you have created the correct word list by running appropriate tests.

**Part 4: Extend the LinkedList class by overloading operators**

This section gives you an opportunity to apply the technique you learned earlier to practice. In the LinkedList.h class, add two operator functions, one overloading the ‘+’ operator for the addAtEnd(T item) operation, and the other overloading the ‘/’ operator for the getAt(int p) operation.

Write a test function and run, for example, the following tests:

```
// an empty char list
LinkedList<char> clist;

// add 10 chars one after another
for (int i = 0; i < 10; i++)
      clist + ('a' + i);

// get chars at variable positions
char *f = clist / 3;
if(f)
      cout << "char at " << 3 << " is: " << *f << endl;
f = clist / 7;
if(f)
      cout << "char at " << 7 << " is: " << *f << endl;
```

Are your operator functions producing the correct results?