SET07109 Programming Fundamentals
Coursework 2

1 Overview

This is the second coursework for SET07109 Programming Fundamentals. This coursework is worth **60%** of the coursework total. The coursework is worth **60%** of the module total. Therefore this coursework is worth **36%** of the total marks available for this module.

In this coursework you are to build a class that supports a linked list data structure for storing `int` data. You will be provided with three files – a header file and supporting C++ file for the linked list, and a test file to test the code. Your task is to complete the method bodies in the C++ linked list file, so that it implements every method defined in the header file and passes all of the tests in the test file. **You must not modify the header file**.

You will need to ensure that your compiler is operating to the C++ 2011 standard for the coursework files to compile. On the Microsoft compiler this should be automatic. With other compilers you may need to pass a command line argument. For example, in the GNU C++ compiler you would use `g++ -std=c++11 filename.cpp`.

The linked list class will require a number of operator overloads to function. For example, to add a number to the linked list we would use the following line of code:

```
list = list + 5;
```

The specification will go into detail about how the application should behave and the requirements.

2 Background

The `LinkedList` class will implement a dynamically sized list of integers. This has the advantage for data storage that, in contrast to arrays, you

do not have to predeclare the size of the list. Memory is allocated for each element in the list at the time that the element is added. The element is then connected to the rest of the list by means of pointers: an element contains a pointer `next`, which points to the next element in the list after itself.

Each element of the list is called a node and is defined by the following structure:

```
struct Node
{
    int data;
    Node *next;
};
```

The last node in the list has its `next` pointer set to `nullptr`. We can visualise the list as shown in Figure 1. Each node is allocated on the heap with a call to `new`. When a node is removed from the list, the memory occupied by that node should be released with a call to `delete`. Access to the list can be gained by two pointers in the `LinkedList` class, `head`, which points to the first node, and `tail`, which points to the last node. These are both private instance variables of the `LinkedList` class. An empty list is represented by `head==nullptr` and `tail==nullptr`.

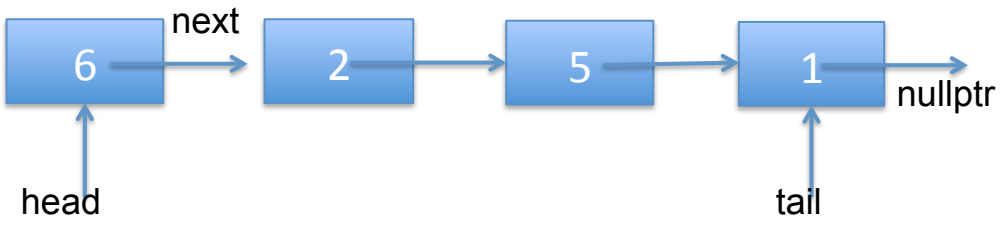

Figure 1: A linked list showing the `head`, `next`, and `tail` pointers.

To create a list with a single element, `value`, the constructor of `LinkedList` would look like this:

```
LinkedList (int value)
{
    head = new Node;
    head->data = value;
    tail = head;
}
```

Accessing nodes requires starting at the `head` and iterating over the list by following `next` pointers until the desired node is reached. This will typically

involve using a `while` loop. The following is an example of a method that counts the number of nodes in the list by following `next` pointers starting from the `head`:

```
int getCount ()
{
    Node *currentNode = head;
    int count = 0;
    while(currentNode != nullptr)
    {
        count++;
        currentNode = currentNode->next;
    }
    return count;
}
```

Adding an element into the middle of the list requires finding the preceding node in the list, and changing its `next` pointer to point to the new node. The new node should then have its `next` pointer set to point to the following element in the list. This is illustrated in Figure 2. Remember to update the `head` and/or `tail` when you add or remove from the start or end of the list.

For further practise with nodes and pointers, you are advised to work through the binary search tree case study at the end of Chapter 6 in the Workbook.

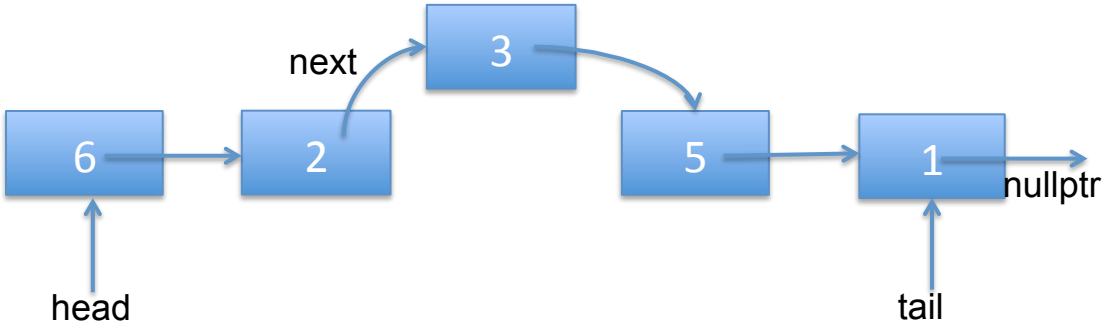

Figure 2: Inserting a node into the linked list after the value 2.

3 Specification

The `LinkedList` class requires a number of methods to be able to operate as illustrated in Table 1. The list can store duplicate integers.

The `LinkedList` class also requires a number of operator overloads. These operators are specified in Table 2.

4 Restrictions and Resources

Your application must be based purely in C++. Further, your code must not make use of functions declared in any other header files apart from the headers already included in the `linked_list.h` and `LinkedList.cpp` files. This means that functions in other headers from the standard library cannot be used. Your application must build in a standard compiler such as Microsoft's `cl`, `g++` or `clang++`.

5 Submission

Your work must be submitted to the Moodle Assignment Dropbox by 23:55 on Sunday 2nd April. If your submit late without prior authorisation from your personal tutor your grade will be capped at 40%.

The submission must be your own work. **If it is suspected that your submission is not your own work then you will be referred to the Academic Conduct Officer for investigation**.

All coursework must be demoed during Week 13. If the coursework is not demoed to a member of the teaching team this will result in a grade of 0. The demo session is the point where the teaching team will give you initial feedback on your work. Final marks, and additional feedback, will be delivered via Moodle after the demonstration.

The submission to Moodle must take the form of the following:

- The code file(s) required to build your application.
- The supplied test file.
- A `makefile` to allow building of your application. The `makefile` should build the linked list as a separate library, and also allow building of the supplied test application against this library. Your `makefile` must also include a clean section.
- A *read me* file indicating how the `makefile` is used and the toolchain used to build it (i.e. Microsoft Compiler, clang, etc.)

These files must be bundled together into a single archive (a zip file) using your matriculation number as a filename. For example, if your matriculation

number is *1234* then your file should be called *1234.zip*. All submissions must be uploaded to Moodle by the time indicated.

6 Marking Scheme

The coursework marks will be divided as follows:

Description	**Marks**
Basic constructors	2
Copy constructor	2
Push and append single value methods	2
Pop and peekTail methods	2
Append another list method	2
Remove all occurrences method	4
Reverse method	4
toString method	2
+ operator	2
Assignment operator (=)	2
Equality operators	2
Input / output operators	2
Resources free and cleared up correctly	2
Code quality	2
Makefile works correctly	2
Submission zip folder complete and correctly collated	2
Total	36

To pass this coursework you will require a minimum of 15 marks, although you should be aiming to achieve a much higher mark to illustrate your understanding of the material.

BE WARNED! This coursework requires an understanding of the material covered in units 5 to 11 of the module. If you do not complete these you will struggle. DO NOT LEAVE THIS WORK TO THE LAST MINUTE!

If you have any queries please contact a member of the teaching team as a matter of urgency. This coursework is designed to be challenging and will require you to spend time developing the solution.

Method	Description
`LinkedList()`	Creates an empty linked list.
`LinkedList(int value)`	Creates a linked list with an initial value to store.
`LinkedList(const LinkedList &rhs)`	Creates a linked list by copying an existing linked list. This must be a deep copy of every node, not a pointer to the existing nodes.
`~LinkedList()`	Destroys the linked list, cleaning up used resources by deleting nodes.
`void push(int value)`	Adds a value to the front of the list.
`void append(int value)`	Adds a value to the end of the list.
`void appendList(const LinkedList &rhs)`	Adds deep copies of the nodes of `rhs` on to the end of this list.
`void insertAfter(int value, int afterValue)`	Inserts a value immediately after the first occurrence of `afterValue`, if `afterValue` is not in the list then inserts at the end of the list.
`int pop()`	Removes and returns the value at the front of the list. It should use an assertion to check that the list is not empty.
`int peekTail()`	Returns the value at the end of the list, without removing it. It should use an assertion to check that the list is not empty.
`void removeAllOccurences(int value)`	Removes all occurrences of the value from the list.
`string toString()`	Generates a string representing the integers in the list in order. The integers must be separated by a single space, with no space at the end.
`void reverse()`	Reverses the order of nodes in the list.

Table 1: List of methods for the `LinkedList` class.

Operator	Example	Description
`=`	`list1 = list2`	Copy assignment operator. Deletes the existing nodes in the left-hand list, frees the memory they occupied, and then deep copies the nodes of the right-hand list into the left-hand list. It should return a reference to the left-hand list.
`==`	`bool res = list1 == list2`	Checks if two lists are equal. Two lists are equal when they both contain the exact same values in the same order.
`!=`	`bool res = list1 != list2`	Checks if two lists are not equal. Two lists are not equal if they do not both contain the exact same values in the same order.
`+`	`list = list + 5`	Inserts a new value into the front of the list. It should return a reference to the list that has just been edited.
`>>`	`cin >> list`	Reads in values from an input stream into the list, adding each value to the front of the list. It should return the input stream.
`<<`	`cout << list`	Writes the values, in order, to an output stream. It should return the output stream.

Table 2: List of operator overloads for the `LinkedList` class.