DigiPen
INSTITUTE OF TECHNOLOGY

Programming Evaluation

Module 2 – High-level Programming II

Purpose

This assignment gives you more practice with classes, constructors, overloaded operators, pointers, and dynamic memory allocation/deallocation and of course linked lists.

Information

Before implementing every method make sure you draw a diagram that shows what the method does and how your nodes need to be adjusted.

The goal is to implement a class called List which encapsulates a single-linked list structure. There are several methods that manipulate the nodes in the linked list. These methods include adding/removing items to and from either end, inserting new node, etc. Each node (Node) in a list contains an integer and a pointer to another Node. This node structure is intentionally kept simple so you can focus on the list aspect of this assignment. This is the interface to the class:

```
namespace CS175
{
    class List
    {
    public:
        // Default constructor
        List();

        // Copy contructor for constructing a list from an existing list
        List(const List &list);

        // Construct a list from an integer array
        List(const int *array, int size);

        // Destructor
        ~List();

        // adds the item to the front of the list
        void push_front(int value_);

        // adds the item to the end of the list
        void push_back(int value_);

        // removes the first item in the list
        int pop_front();

        // removes the last item in the list
        int pop_back();

        // removes the first node it finds with the user defined value
        void remove_node_by_value(int value_);

        // inserts a new node in the list. The node will be inserted
        // at the user defined location and will have the user defined value.
        void insert_node_at(int location_, int value_);
```

```
        // Overloaded assignment operator (=) for assigning one list to another
        List& operator=(const List &list_);

        // Overloaded addition operator (+) for adding two lists
        List operator+(const List &list_) const;

        // Overloaded addition/assignment (+=) for adding to a list "in place"
        List& operator+=(const List &list_);

        // returns the number of items in the list
        int list_size() const;

        // returns true if empty, else false
        bool empty() const;

        // clears the list from all nodes
        void clear();

        // Output operator for printing lists (<<)
        friend std::ostream & operator<<(std::ostream & os_, const List &list_);

        // returns the number of Lists that have been created
        static int created_list_count();

        // returns the number of Nodes that are still alive
        static int alive_node_count();

    private:

        // used to build a linked list of integers
        struct Node
        {
            Node(int);              // constructor
            ~Node();                // destructor
            Node *next;             // pointer to the next Node
            int data;               // the actual data in the node
            static int nodes_alive; // count of nodes still allocated
        };

        Node *head; // pointer to the head of the list
        Node *tail; // pointer to the last node
        int size;   // number of items in the list

        static int list_count;       // number of Lists created
        Node *new_node(int data_) const; // allocate node, initialize data/next
    };

} // namespace CS175
```

Many of the methods will call other methods you've implemented (code reuse), so the amount of code is not that great. The "worker" functions are **push_back**, **push_front**, **pop_front** and **pop_back**. Once these functions are implemented and thoroughly tested, the rest of the assignment is fairly straight-forward. The sample driver (***main.cpp***) shows many example function calls with the appropriate output. You should be able to get all of the information required from the driver.

Other criteria

- You must allocate the nodes using **new.** The only function that should use **new** is **new_node**. This means that the keyword **new** should be used exactly once in your program. (If it is used more than once, even in a comment, you will lose points.)
- Only the **push_front**, **push_back** and **insert_node_at** methods should call **new_node** (to create nodes).
- You must deallocate the nodes using **delete**. Make sure you are deallocating the memory anytime a node is removed otherwise you will have memory leaks.
- The **pop_front** and **pop_back** methods return the value of the node that got popped. If no nodes were popped, the methods will return the value **-1**.
- The **insert_node_at** method takes the location index as its first parameter. If a negative value is provided, the node should be inserted in the beginning of the list. If an index greater than the list size is provided, the node is added at the end of the list.
- Make sure that you use **const** where appropriate. Double-check your code. (And then check it again)
- Make sure you properly initialize the ***static*** properties.
- **THINK CODE REUSE.** In other words, can you call one of your functions to do something rather than write more of the same code?

Testing your code

A "Grading Scripts" folder is provided in order for you to check if your implementation is correct.

In order to run the tests follow these steps:

- grab the cpp file ***"List.cpp"*** and place them in the "**Grading Scripts**" folder.
- Double click on the **"RunAll.cmd"** file.
    - All tests will run automatically. For every test, you will know if you passed it or failed it. If you failed the test, the scripts will tell you why you failed it.

      NOTE: If you are working on a DigiPen lab/classroom machine you need to run the "DigiPen-RunAll.cmd" file instead of "RunAll.cmd"

What to submit

You must submit the ***List.cpp*** file in a single .zip file (go to the class page on moodle and you will find the assignment submit link).

Do not submit any other files than the ones listed.