Objectives

The objectives of this task is to assess your knowledge and skills in creating objects and initializing them, overloading operators for custom objects, correct dynamic memory handling, using pointers and references, as well as pointers to functions.

Task description

Your task is to create a class named **MyLinkedList** representing a **linked list** capable of storing integers and supporting certain methods as described below. A linked list is a sequence of elements, where every element has knowledge about the next element in the sequence. For this reason, we can traverse such a list in one direction – from beginning to the end, but not in the opposite direction.

Technically, every list element is an object of some arbitrary type, that stores a value of the element (an integer), an a pointer to the next object in the sequence. The **MyLinkedList** type is another object type, which stores a pointer to the first element and contains the methods that you need to implement (described further below). You are free to design your own object type to represent a list element, feel free to add methods as you see fit. However, you are not allowed to modify the public methods of the **MyLinkedList** class as required by the task.

Note: *the linked list shall be implemented "from scratch" using atomic C++ language constructions, using STL data structures is not permitted for the purposes of this task.*
Note: *no methods of the class specified below, nor their signatures, shall be altered in any way, these methods shall be implemented* **AS-IS**.

Requirements towards objects of MyLinkedList type

1. It shall be possible to create an empty list by calling the default constructor, i.e.: **MyLinkedList list**;
2. It shall be possible to create a list of **n** elements initialized with value **w** by calling a custom constructor taking **(n,v)** as argument list, i.e.: to create a list of ten numbers "5" call **MyLinkedList list(10,5);**
3. It shall be possible to obtain the number of elements in the list by calling the **getLength()** method. **Note:** the number of elements is always nonnegative, so this method is expected to return a value of type **size_t**. This method shall return 0 for an empty list. It shall be possible to ask for the length of a list marked as **const**.

4. It shall be possible to copy-construct a list. The copy-constructed list shall be completely independent from the source from which it was copied. An example of invoking a copy constructor: **MyLinkedList list, anotherlist(list);**
5. It shall be possible to copy-assign a list. The copy-assigned list shall be completely independent from the source from which it was copied. An example of invoking a copy-assignment operator:
   MyLinkedList list = MyLinkedList(5,10);
6. It shall be possible to access random elements in the list by using the random access operator **[]**. This operator shall provide read-write access to elements stored in a non-const list, and read-only access to elements stored in the list is marked as **const**. This means that if we've got a non-const list, say, **MyLinkedList list1;** it shall be possible to read element values as **int k = list1[0];** as well as write values there **list1[1] = 5;** In the case of a **const** list, however, it shall be possible to read elements as **int k = list1[0];** but any modifications like **list1[1] = 5;** shall not affect the list elements.
7. It shall be possible to push objects of type MyLinkedList into the output stream, i.e.: **MyLinkedList list; std::cout << list << std::endl;** In result, the sequence of numbers, stored in the list, shall be printed with spaces as separators, i.e.: 1 2 3 …
8. It shall be possible to check if the two MyLinkedList objects are equal using operator ==, i.e.: **list1 == list2;** Our definition of equality for lists is the following: the two lists are equal if both contain the same number of elements, and the elements in the corresponding positions are pairwise equal.
9. It shall be possible to prepend new elements to the beginning of the list by calling **push_front(element)**, and to append new elements to the end of the list by calling **push_back(element)**. I.e., consider the list **list** containing one single element 3. Calling **list.push_front(1)** would add element 1 to the list, giving is the list containing 1 and 3. Calling **list.push_back(5)** would add element 5 to the end of the list giving us the list containing elements 1, 3, and 5.
10. It shall be possible to insert elements into random positions in the list using method **insert(position,element)**. I.e., if the list contained elements 1,3,5, calling **list.insert(0,10)** would produce a list containing 10,1,3,5. Calling **list.insert(1,10)** would produce a list containing 1, 10, 3, 5. Calling **list.insert(2,10)** would produce a list containing elements 1,3,10,5, and so on.
    Note: You do not need to check for out-of-bounds condition, it is safe to assume that the method will always get nonnegative indices starting with 0 and not exceeding the length of the list.
    Note: the position of an element is nonnegative, and therefore is expected to be of unsigned type **size_t**.
11. It shall be possible to remove elements at arbitrary positions in the list by calling remove(position). Lets consider the list containing elements 1,2,3,4,5. Calling **list.remove(0)** would produce a list containing 2,3,4,5. Calling **list.remove(2)** on the result is expected to produce a list containing 2,3,5, etc.

Note: You do not need to check for out-of-bounds condition, it is safe to assume that the method will always get nonnegative indices starting with 0 and not exceeding the length of the list.
Note: the position of an element is nonnegative, and therefore is expected to be of unsigned type **size_t**.

12. It shall be possible to supply a unary function with signature **int(int)** elementwise to the list by calling a **static** method **applyUnaryOperator**, which takes the list object as the first argument, and the pointer to the function as the second argument. Then this method calls the supplied function for every element in the list, and overwrites the value of the element with the result of this function. In example, if we have a list of elements 1,2,3, applying a function **int negate(int x) { return -x; }** to every element changes the values in the list to -1,-2,-3. Applying int **pow2(int x){ return x*x; }** changes the values in the list to 1,4,9.
13. Finally, as you are working with dynamic memory, it is essential to handle memory properly. Do not forget about destructors and de-initialize your objects properly.

Project Template Setup

The project template you are supplied with contains two build targets:

- **Run Tests** – the target you may use to run unit tests from the **hw01test.cpp** file to get constant feedback on how you are progressing. This requires the **googletest** library, located in the **gtest** directory.
- **HW01** – the target containing more useful and human-readable feedback printed on the screen in the **main()** entry point function in **hw01.cpp** file.

You are free and welcome to create any number of extra files as you see fit for any particular purpose.

Assessment Criteria

Your submissions will be assessed for compliance to the following set of requirements:

1. The solution is compilable.
2. It is possible to construct an empty list.

3. It is possible to construct a list using the custom constructor.
4. It is possible to copy-construct a list (non-empty as well as empty).
5. The copy-constructed list is completely independent from its source.
6. It is possible to copy-assign a list (non-empty as well as empty).
7. The copy-assigned list is completely independent from its source.
8. The **getLength()** method returns correct lengths (0 for an empty list).
9. The **getLength()** method is callable on objects marked as **const**.
10. The **getLength()** method returns a value of type **size_t**.
11. Any list (including empty) can be pushed into the output stream.
12. Elements of a non-const list can be randomly read using **operator[]**.
13. Elements of a non-const list can be randomly assigned using **operator[]**.
14. Elements of a const list can be randomly read using **operator[]**.
15. Value assignment using **operator[]** to any element of a const list does not affect this element.
16. The two lists are comparable using **operator==** in accordance with the provided definition of equality.
17. **push_front()** method is working as required.
18. **push_back()** method is working as required.
19. **insert()** method allows to insert elements in the arbitrary positions in the list.
20. **remove()** method allows to remove elements from arbitrary positions in the list.
21. **applyUnaryOperator()** method is working as required.
22. Dynamic memory handling is appropriate.

The points you get for the task is the percentage of the requirements your solution complies with.

Plagiarism warning*: please be advised that plagiarism shall not be tolerated in any form. If you happen to exhibit statistically insignificant coincidences in your submittions, you do it at your own risk.*