## **Objectives HW01**

Задачи этой задачи - оценить ваши знания и навыки в создании объектов и их инициализация, перегрузка операторов для пользовательских объектов, исправление динамической памяти обработки, использования указателей и ссылок, а также указателей на функции.

## **Task description**

Ваша задача - создать класс с именем **MyLinkedList**, представляющий **linked list** ,способный сохранение целых чисел и поддержка определенных методов, как описано ниже. **Linked list** - это последовательность элементов, где каждый элемент имеет знания о следующем элементе в последовательность. По этой причине мы можем пройти такой список в одном направлении - с самого начала до конца, но не в противоположном направлении.

Технически каждый элемент списка является объектом некоторого произвольного типа, который сохраняет значение элемент (**Integer**), указатель на следующий объект в последовательности. Тип **MyLinkedList** - это другой тип объекта, в котором хранится указатель на первый элементи содержит методы, которые необходимо реализовать (описано ниже). Нужно создавать свой собственный тип объекта для представления элемента списка, не стесняйтесь добавлять методы как вы считаете нужным. Однако вам не разрешено изменять общедоступные методы **MyLinkedList**, как требуется задаче.

**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. Должно быть возможно создать пустой список,вызвав конструктор по умолчанию, то есть: **MyLinkedList list**;

2. Должно быть возможно создать список из **n** элементов, инициализированных значением **w** путем вызова пользовательский конструктор, принимающий **(n, v)** в качестве списка аргументов, т. е. для создания списка из десяти номера «**5**» **MyLinkedList list(10,5);**

3. Необходимо получить количество элементов в списке, вызвав метод **getLength().** **Примечание**: количество элементов всегда неотрицательно, поэтому ожидается, что метод возвращает значение типа **size\_t.** Этот метод должен вернуть **0** для пустой список. Должна быть задана длина списка, обозначенного как **const**.

4. Должна быть возможность **copy-construct a list**. **Сopy-construct a list** должен быть полностью независимый от источника, из которого он был скопирован. Пример вызов конструктора копирования: **MyLinkedList list, anotherlist(list);**

5. Должна быть возможность **copy-assign a list**. **The copy-assign a list**, назначенный для копирования, должен быть полностью независимый от источника, из которого он был скопирован. **Пример** вызов **copy-assignment operator**: **MyLinkedList list = MyLinkedList(5,10);**

6. Доступ к случайным элементам в списке возможен, используя случайные **access operator []**. Этот оператор должен предоставлять **to elements stored in a non-const list, and read-only access to elements stored in the list is marked as const.** Это означает, что если у нас есть **non-const list**, скажем, **MyLinkedList1;** Это должны быть доступны для чтения значений элементов как **int k = list1[0];** а также значения записи **list1 [1] = 5;** Однако в случае списка **const** можно читать элементы как **int k = list1 [0];** но любые модификации, такие как list1 [1] = 5; не должны влияют на элементы списка.

7. Должна быть возможность вставлять объекты типа **MyLinkedList** в выходной поток, i.e: список **MyLinkedList; std :: cout << list << std :: endl;** В результате последовательность числа, хранящиеся в списке, должны быть напечатаны с пробелами в качестве разделителей, то есть: 1 2 3 ...

8. Должна быть доступна возможность проверки, совпадают ли 2 объекта **MyLinkedList**, используя

**operator ==,** т. е .: **list1 == list2;** Наше определение равенства для списков – это следующие: два списка равны, если оба содержат одинаковое количество элементов, и элементы в соответствующих положениях попарно равны.

9. Должна быть предусмотрена возможность добавления новых элементов в начало списка путем вызова **push\_front (element)** и добавлять новые элементы в конец списка, вызывая **push\_back (element)**. I.e., рассмотрим список **list**, содержащий один единственный элемент **3**. Вызов **list.push\_front (1)** добавит элемент **1** в список, давая список содержащие **1** и **3**. Вызов **list.push\_back(5)** добавит элемент **5** в конец список, содержащий список, содержащий элементы **1, 3** и **5**.

10. Необходимо вставлять элементы в случайные позиции в списке, используя метод **insert(position,element)**. I.e., если список содержит элементы **1,3,5**, вызывающие **list.insert(0,10)** приведет к составлению списка, содержащего **10,1,3,5.** Вызов **list.insert(1,10)** будет составлять список, содержащий **1, 10, 3, 5**. Вызов **list.insert(2,10)** будет создать список, содержащий элементы **1,3,10,5** и т. д.

**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. Необходимо удалить элементы в произвольных позициях в списке, позвонив **remove(position)**. Рассмотрим список, содержащий элементы **1,2,3,4,5**. Вызвав **list.remove(0)** создаст список, содержащий **2,3,4,5.** Вызов **list.remove(2)** ожидается, что в результате будет составлен список, содержащий **2,3,5** и т. д.

**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. Должна быть предоставлена ​​унарная функция с сигнатурой **int (int) elementwise** в список, вызвав **static** метод **applyUnaryOperator**, который принимает список объект как первый аргумент, а указатель на функцию как второй аргумент. Затем этот метод вызывает предоставленную функцию для каждого элемента в **list** и перезаписывает значение элемента с результатом этой функции.

**Пример**, если у нас есть список элементов **1,2,3**, применяющий функцию **int negate(int x)** **{return -x; }** каждый элемент меняет значения в списке на **-1, -2, -3**. Применение **int pow2 (int x) {return x \* x; }** изменяет значения в списке до **1,4,9**.

13. Наконец, поскольку вы работаете с динамической памятью, необходимо обрабатывать памяти. Не забывайте о **деструкторах** и **деинициализируйте** свои объекты должным образом.

## **Assessment Criteria**

Ваши материалы будут оцениваться на соответствие следующим требованиям:

1. Решение компилируемо.

2. Можно построить пустой список.

3. Можно создать список, используя пользовательский конструктор.

4. Можно скопировать-построить список (не пустой, а также пустой).

5. Скопированный список полностью не зависит от его источника.

6. Можно скопировать-присвойте список (не пустой, а также пустой).

7. Список, назначенный копией, полностью не зависит от его источника.

8. Метод **getLength()** возвращает правильные длины (0 для пустого списка).

9. Метод **getLength()** доступен для объектов, помеченных как **const**.

10. Метод **getLength()** возвращает значение типа **size\_t.**

11. Любой **list (including empty)** может быть перенесен в выходной поток.

12. Элементы **неконстантного** списка могут быть случайно прочитаны с использованием **operator[]**.

13. Элементы **неконстантного** списка могут быть случайным образом назначены с помощью **operator[]**.

14. Элементы списка **const** могут быть случайно прочитаны с использованием **operator[]**.

15. Присвоение значения с помощью **operator[]** любому элементу списка const не влияет

этот элемент.

16. Эти два списка сопоставимы с использованием **operator==** в соответствии с предоставленным

определение равенства.

17. Метод **push\_front ()** работает по мере необходимости.

18. Метод **push\_back ()** работает по мере необходимости.

19. Метод **insert ()** позволяет вставлять элементы в произвольные позиции в списке.

20. Метод **remove ()** позволяет удалить элементы из произвольных позиций в списке.

21. **applyUnaryOperator ()** работает по мере необходимости.

22. Соответствие динамической памяти.

Точки, которые вы получаете для задачи, - это процент требований вашего решения в соответствии.