#2: Containers

Одной из важнейших задач при разработке программ является задача организации хранения и предоставления доступа к данным. Эта задача решается при помощи различных контейнеров данных, обеспечивающих управление памятью, необходимой для хранения данных и предоставляющих интерфейсы для доступа к хранимым данным.

В вашу задачу входит реализация двух АТД: динамического массива и списка.

§1: Динамический массив

```
template<typename T>
class Array final
```

должен поддерживать следующие операции:

- ```
  Array();
  Array(int capacity);
  ```
  Конструктор распределяет память, необходимую для хранения некоторого количества элементов. Конструктор без параметров использует значение емкости по умолчанию (например, 8 или 16), конструктор с параметром capacity – явно переданное значение.

- ```
  ~Array();
  ```
  Деструктор освобождает память, выделенную под хранение элементов. При необходимости, при освобождении памяти вызываются деструкторы хранимых элементов.

- ```
  void insert(const T& value);
  void insert(int index, const T& value);
  ```
  Вставляет переданное значение в конец массива, или в указанную позицию, увеличивая размер массива на 1 и, при необходимости, сдвигая существующие элементы вправо. Если памяти для добавления элемента недостаточно, то перераспределяет память, копируя уже существующие элементы в новую область. Затем старый блок освобождается. Для копирования и сдвигания элементов желательно использовать move-семантику. (Также можно использовать устаревшее и менее эффективное копирование с последующим вызовом деструктора).
  ```

Выделение памяти происходит каждый раз с увеличением в 1.6..2 раза относительно текущего размера.

- `void remove(int index);`
  Удаляет элемент из указанной позиции массива, сдвигая остальные элементы влево (сдвиги выполняются аналогично insert(), но память при этом не освобождается).

- `const T& operator[](int index) const;`
  `T& operator[](int index);`
  Оператор индексирования позволяет обратиться к элементу массива по индексу для чтения и для записи.

- `int size() const;`
  Возвращает текущий размер (количество реально существующих в массиве) элементов.

- `Iterator iterator();`
  `const Iterator iterator() const;`
  Возвращает итератор, указывающий на первый элемент массива.

Обратите внимание, что по правилам C++ такой массив может копироваться и присваиваться, однако поведение копирования и присваивания по умолчанию приведет к проблемам, связанным с тем, что оба экземпляра будут использовать один и тот же блок памяти, и пытаться освобождать его в деструкторе. Поэтому вы должны или предоставить правильный механизм копирования и присваивания или запретить его.

Итератор должен быть реализован как внутренний для `Array<T>` класс, и предоставлять следующий интерфейс:

- `const T& get() const;`
  Получает значение массива в текущей позиции итератора.

- `void set(const T& value);`
  Устанавливает значение в текущей позиции итератора.

- `void insert(const T& value);`
  Вставляет значение в текущую позицию итератора, сдвигая соответствующие элементы (начиная с текущего) вправо. Итератор после вставки остается позиционированным на новый элемент.

- `void remove();`
  Удаляет значение из текущей позиции итератора, сдвигая соответствующие элементы (начиная с элементом, следующего за текущим) влево. Итератор после удаления остается позиционированным на элемент, оказавшийся на позиции удаленного.
- `void next();`
  Перемещает текущую позицию итератора на 1 вправо, переходя к следующему элементу.
- `void prev();`
  Перемещает текущую позицию итератора на 1 влево, переходя к предыдущему элементу.
- `void toIndex(int index);`
  Устанавливает позицию итератора на элемент с указанным индексом.
- `bool hasNext() const;`
  Возвращает `true`, если итератор может перейти к следующему элементу, или `false`, если итератор позиционирован на последний элемент.
- `bool hasPrev() const;`
  Возвращает `true`, если итератор может перейти к предыдущему элементу, или `false`, если итератор позиционирован на первый элемент.

Функции работы с индексом не должны выполнять проверку границ. Но в отладочной версии такие проверки можно добавить используя механизм утверждений.

Пример использования: поместим в массив числа от 1 до 10, умножим каждое на 2, затем выведем их на экран.

```
Array<int> a;
for (int i = 0; i < 10; ++i)
  a.insert(i + 1);

for (int i = 0; i < a.size(); ++i)
  a[i] *= 2;

for (auto it = a.iterator(); it.hasNext(); it.next())
  std::cout << it.get() << std::endl;
```

§2: Связный список

```
template<typename T>
class List final
```

должен реализовывать двусвязный список с чанками и поддерживать следующие операции:

- `List();`
  Конструктор распределяет память, необходимую для хранения одного чанка списка.

- `~List();`
  Деструктор освобождает память, выделенную под хранение элементов. При необходимости, при освобождении памяти вызываются деструкторы хранимых элементов.

- `void insertHead(const T& value);`
  `void insertTail(const T& value);`
  Вставляет переданное значение в голову или хвост списка, помещая значение в соответствующий чанк. При необходимости, выделяется новый чанк и добавляется к списку в начало или конец.

- `void removeHead();`
  `void removeTail();`
  Удаляет элемент из головы / хвоста списка, при необходимости удаляя освободившиеся чанки (за исключением последнего, который должен присутствовать даже для пустого списка).

- `const T& head() const;`
  `const T& tail() const;`
  Возвращает текущее значение находящееся в голове / хвосте списка.

- `int size() const;`
  Возвращает текущий размер (количество реально существующих в списке) элементов. Это значение лучше хранить в поле объекта `List`, изменять при вставке удалении и не пересчитывать каждый раз.

- `Iterator iterator();`
  `const Iterator iterator() const;`
  Возвращает итератор, указывающий на первый элемент списка.

Размер чанка подберите самостоятельно. В комментариях поясните, почему выбрали именно такой размер.

Для списка справедливы те же рассуждения о присваивании и копировании, что и для массива.

Итератор должен быть реализован как внутренний для `List<T>` класс, и предоставлять интерфейс, аналогичный `Array<T>::Iterator`, за исключением `toIndex()`, так как список не подразумевает произвольного доступа.

Для всех реализованных методов динамического массива и списка должны быть созданы необходимые модульные тесты.