Линейные контейнеры

М. А. Ложников

4 марта 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Контейнер std::array (C++11)

Контейнер `std::array` представляет собой оболочку над обычным статическим массивом. Он определён в заголовочном файле `<array>` и реализован как шаблон с двумя параметрами: типом данных элемента и количеством элементов.

```cpp
#include <iostream>
#include <array>

int main() {
  std::array<double, 5> aDouble; // Массив из пяти элементов типа double

  /* Обратите внимание, что точно также как и в обычном статическом массиве, элементы
    контейнера array необходимо инициализировать при условии, что тип данных элемента
    является одним из стандартных типов. В противном случае в массиве будут лежать
    случайные значения. Если тип данных элемента массива является классом, у которого
    есть конструктор по-умолчанию, то элементы массива инициализировать не обязательно.
 */

  // Метод fill() позволяет проинициализировать элементы массива одним и тем же числом
  aDouble.fill(3.0);

  // Для шаблона array переопределена операция [] для доступа к элементу по индексу
  // как на чтение
  std::cout << aDouble[1] << std::endl; // выведет 3
  // так и на запись
  aDouble[0] = 4.3;

  // Метод size() позволяет узнать количество элементов в массиве
  std::cout << "Array size = " << aDouble.size() << std::endl; // выведет 5

  // Поддерживается следующий вариант инициализации элементов
  std::array<int, 3> aInt = { 0 }; // все элементы инициализируются нулями

  std::array<int, 3> aInt2 = {1, 2, 3}; // aInt2[0] = 1, aInt2[1] = 2, aInt[2] = 3

  std::array<int, 3> aInt3 = {1, 2}; // aInt2[0] = 1, aInt2[1] = 2, aInt[2] = 0
```

```
32
33   /* При инициализации массива списком элементов в фигурных скобках, элементы
34      проинициализируются по порядку. Если в скобках указано меньше элементов,
35      чем в массиве, то оставшиеся элементы проинициализируются значениями по умолчанию,
36      в данном случае нулями. */
37
38   // Метод empty() позволяет узнать, является ли массив пустым
39   std::array<int, 0> emptyArray;
40   std::array<int, 4> notEmptyArray;
41
42   std::cout << emptyArray.empty() << std::endl;
43   std::cout << notEmptyArray.empty() << std::endl;
44   return 0;
45 }
```

Задача 1 *Написать шаблон для подсчёта статистики использования сетевых портов маршрутизатора.*

```
template<unsigned numPorts>
class RouterStats {
 public:
  RouterStats();

  void Receive(unsigned port, std::size_t numBytes);

  std::array<std::pair<unsigned, std::size_t>, numPorts> GetSortedStats() const;
 private:
  std::array<std::size_t, numPorts> numReceivedBytes;
};
```

В массиве `numReceivedBytes` *содержится число принятых по сети байт, таким образом, порт с номером* `i` *принял* `numReceivedBytes[i]` *байт. Конструктор класса должен инициализировать массив нулевыми значениями. Метод* `Receive()` *должен увеличивать статистику порта* `port` *на* `numBytes` *байт. Если* `port >= numPorts`*, то следует выбросить исключение* `std::invalid_argument`*, определённое в заголовочном файле* `<stdexcept>`*. Метод* `GetSortedStats()` *возвращает массив пар (номер порта, число байт), отсортированных по числу принятых байт в порядке убывания. Шаблон* `std::pair` *определён в заголовочном файле* `<utility>` *и устроен аналогично рассмотренному ранее шаблону* `Pair`*, то есть имеет два поля* `first` *и* `second` *соответствующих типов данных.*

1.1 Итераторы контейнера array

У контейнера `array` есть набор итераторов, которые поддерживают доступ к произвольному элементу как на чтение, так и на запись, а также позволяют произвести обход массива как вперёд (прямые итераторы), так и назад (обратные итераторы). Поясним на примере.

```
// const_iterator предназначен для чтения элементов в прямом порядке
template<typename T, std::size_t N>
void PrintArray(const std::array<T, N>& a) {
```

```
  // Сделаем новое имя для типа данных, а то оно слишком длинное.
  using It = typename std::array<T, N>::const_iterator;

  for (It it = a.begin(); it != a.end(); ++it)
    std::cout << *it << std::endl;
}

// const_reverse_iterator предназначен для чтения элементов в обратном порядке
template<typename T, std::size_t N>
void PrintReversedArray(const std::array<T, N>& a) {
  // Сделаем новое имя для типа данных, а то оно слишком длинное.
  using It = typename std::array<T, N>::const_reverse_iterator;

  for (It it = a.rbegin(); it != a.rend(); ++it)
    std::cout << *it << std::endl;
}

// iterator предназначен для записи элементов в прямом порядке
template<typename T, std::size_t N>
void ReadArray(std::array<T, N>& a) {
  // Сделаем новое имя для типа данных, а то оно слишком длинное.
  using It = typename std::array<T, N>::iterator;

  for (It it = a.begin(); it != a.end(); ++it)
    std::cin >> *it;
}

// reverse_iterator предназначен для записи элементов в обратном порядке
template<typename T, std::size_t N>
void ReadReversedArray(std::array<T, N>& a) {
  // Сделаем новое имя для типа данных, а то оно слишком длинное.
  using It = typename std::array<T, N>::reverse_iterator;

  for (It it = a.rbegin(); it != a.rend(); ++it)
    std::cin >> *it;
}
```

Замечание 1.1 *В тех случаях, когда происходит обращение к типу данных, определённому внутри шаблона необходимо ставить слово* `typename` *при условии, что шаблон не инстанцирован конкретными значениями. В примере выше* `T` *и* `N` *являются параметрами, таким образом,* `std::array<T, N>` *является зависимым типом данных. Для того, чтобы компилятор понял, что* `iterator`*, определённый внутри* `std::array` *является типом данных, нужно перед всей конструкцией ставить слово* `typename`*. Таким образом, следует писать* `typename std::array<T, N>::iterator`*, однако нужно писать* `std::array<int, 5>::iterator`*, то есть опустить слово* `typename` *в том случае, если шаблон инстанцирован некими конкретными значениями. Если бы весь контейнер был шаблонным параметром, то также следовало бы ставить слово* `typename`*:*

```
template<typename Container>
void PrintContainer(const Container& c) {
  for (typename Container::const_iterator it = c.begin(); it != c.end(); ++it)
```

```
    std::cout << *it << std::endl;
}
```

Как обычно, метод `begin()` возвращает прямой итератор на первый элемент, метод `end()` возвращает прямой невалидный итератор на конец массива (на элемент “после последнего”, он будет концом прямого промежутка). Метод `rbegin()` возвращает обратный итератор на последний элемент массива, метод `rend()` возвращает обратный невалидный итератор на начало массива (на элемент “перед первым”, он будет концом обратного промежутка). Прямые и обратные итераторы при инкременте движутся по массиву в противоположенных направлениях. Таким образом, пара итераторов задаёт полуинтервал `[begin(), end())` (или `(rend(), rbegin()]`).

Отметим, что вместо методов `begin()`, `end()` класса можно использовать внешние функции `std::begin()` и `std::end()`. То есть записи `a.begin()` и `std::begin(a)`, а также `a.end()` и `std::end(a)` эквивалентны. То же самое верно и для обратных итераторов.

Итератор контейнера `array` можно рассматривать как оболочку над обычным указателем. Рассмотрим его чуть подробнее.

```
std::array<int, 5> arr = {1, 3, 5, 7, 9};

std::array<int, 5>::iterator it = arr.begin();
// Операция * возвращает ссылку на элемент контейнера, на который указывал итератор
int& a0 = *it;

a0 = -1; // arr[0] станет равным -1

// Для константного итератора операция * возвращает константную ссылку
std::array<int, 5>::const_iterator itConst = arr.begin();
const int& constA0 = *itConst;

++it; // теперь итератор указывет на arr[1]
--it; // теперь указываем снова на arr[0]
it += 3; // теперь указываем на arr[3]
it -= 2; // теперь указываем на arr[1]
std::array<int, 5>::iterator it2 = it + 3; // it2 указываем на arr[4]

/* Итераторы массива можно вычитать друг из друга. Для двух итераторов массива
it1 <= it2 разность it2 - it1 равна количеству элементов в полуинтервале [it1, it2).] */
std::ptrdiff_t difference = it2 - arr.begin(); // разность равна 4

// Если необходимо получить соседний итератор, не меняя текущий, то помимо операций
// сложения и вычитания можно использовать функции next() и prev().

it2 = std::next(it); // it2 указывает на arr[2] (следующий за it)
it2 = std::prev(it); // it2 указывает на arr[0] (перед it)

// Можно также передать на сколько элементов нужно сдвинуться.
it2 = std::next(it, 2); // it2 указывает на arr[3]
it = std::prev(it2, 3); // it указывает на arr[0]
```

Отметим, что сложность каждой из приведённых операций для итераторов контейнера `array` составляет $O(1)$.

Кроме того, итераторы как правило можно передавать по значению. Они довольно просто устроены, поэтому стоимость копирования обычно меньше стоимости обращения по ссылке.

Задача 2 *Напишите шаблонную функцию*

```
template<typename IteratorType>
void QuickSort(IteratorType rangeBegin, IteratorType rangeEnd);
```

сортирующую промежуток `[rangeBegin, rangeEnd)` *при помощи алгоритма быстрой сортировки. Гарантируется, что итераторы типа* `IteratorType` *предоставляют произвольный доступ на запись, а элементы, на которые указывают эти итераторы можно сравнивать при помощи операции* `<`.

Внимание. *В данной задаче нельзя пользоваться функциями сортировки из стандартной библиотеки.*

Возможно, при решении задачи потребуется завести временную переменную для хранения элементов, на которые указывает итератор. Тип данных элемента есть `typename IteratorType::value_type`. *В каждом контейнере стандартной библиотеки (а также в итераторах этих контейнеров) определён тип* `value_type`, *соответствующий типу данных элемента контейнера. Слово* `typename` *здесь нужно указывать для того, чтобы компилятор понял, что* `value_type` *есть тип данных, он этого не знает поскольку* `IteratorType` *— произвольный параметр (см. замечание 1.1).*

Есть способ, позволяющий избежать такой громоздкой конструкции, а также обеспечить работу функции `QuickSort` *для контейнеров, не определяющих тип* `value_type`. *Для этого нужно написать шаблонную функцию*

```
template<typename ElemType>
void Swap(ElemType& first, ElemType& second);
```

переставляющую свои аргументы местами, или воспользоваться шаблонной функцией `std::swap()` *из заголовочного файла* `<utility>`.

Замечание 1.2 *Если в функцию* `QuickSort()` *передать пару обратных итераторов* `rbegin()` *и* `rend()`, *то она отсортирует элементы контейнера в обратном порядке.*

2 Цикл range-based for и ключевое слово auto (C++11)

Если контейнер предоставляет методы `begin()` и `end()`, которые возвращают нечто, по чему можно проитерироваться, то для обхода элементов этого контейнера можно использовать цикл *range-based for*.

```
std::array<int, 10> arr;

/* Сначала указывается тип данных, потом имя переменной для элемента контейнера,
  потом через двоеточие сам контейнер. Итерироваться можно по значению,
  по ссылке и по константной ссылке. */

// Заполняем массив. Итерируемся по ссылке.
for (int& elem : arr)
  std::cin >> elem;
```

```
10
11 // Выводим массив. Итерируемся по значению.
12 for (int elem : arr)
13   std::cout << elem << std::endl;
14
15 /* Если необходимо избежать копирования элементов, то можно проитерироваться
16   по константной ссылке. Имеет смысл для тяжёлых типов, для типа int не актуально.*/
17 for (const int& elem : arr)
18   std::cout << elem << std::endl;
```

С помощью ключевого слова `auto` можно сделать цикл более универсальным, а именно писать его одинаково для любого типа элемента контейнера.

```
1 std::array<int, 10> arr;
2
3 // Выводим массив. Итерируемся по значению.
4 for (auto elem : arr) // elem имеет тип int
5   std::cout << elem << std::endl;
6
7 // Выводим массив. Итерируемся по ссылке.
8 for (auto& elem : arr) // elem имеет тип int&
9   std::cout << elem;
10
11 /* auto можно указывать вместо типа данных при инициализации при объявлении,
12    компилятор автоматически выведет тип данных переменной. */
13 const auto& constArr = arr; // constArr имеет тип данных const std::array<int, 10>&
14
15 auto i = 2; // i имеет тип данных int
16 auto d = 2.5; // d имеет тип данных double
17
18 /* Тип данных matrix совпадает с типом данных возвращаемого значения функции
19  InitMatrix() со снятыми ссылками. То есть, если функция возвращала int&, то переменная
20  matrix будет типа int и в ней будет содержаться копия возвращаемого значения. */
21 auto matrix = InitMatrix();
22
23 for (auto& elem : constArr) // elem имеет тип данных const int&
24   std::cout << elem;
```

Можно считать, что компилятор заменяет конструкцию вида

```
1 for (auto& elem : container) {
2   // ...
3 }
```

на конструкцию вида

```
1 for (auto it = container.begin(); it != container.end(); ++it) {
2   auto& elem = *it;
3   // ...
4 }
```

Таким образом, для того, чтобы по элементам пользовательского контейнера можно было проитерироваться при помощи цикла *range-based for*, в нём должны быть определены два публичных метода: `begin()`, `end()`, которые возвращают нечто, для чего определены операции `!=` и префиксная операция инкремента.

Отметим, что `auto` имеет смысл использовать для экономии места, когда тип данных переменной занимает много места, однако очевиден для разработчика. Тем не менее, не стоит этим злоупотреблять, в противном случае программа может стать плохо читаемой.

Задача 3 *Напишите шаблон и порождающую функцию для него*

```
template<typename IteratorType>
class Range;

template<typename IteratorType>
Range<IteratorType> MakeRange(IteratorType rangeBegin, IteratorType rangeEnd);
```

позволяющие проитерироваться при помощи цикла `range-based for` *по промежутку* `[rangeBegin, rangeEnd)`*. Конструктор шаблона принимает два итератора* `rangeBegin` *и* `rangeEnd` *типа* `IteratorType`*. Публичные методы* `begin()` *и* `end()` *возвращают итераторы на начало и конец промежутка соответственно. Например, для итерации по первым пяти элементам массива достаточно написать следующий код:*

```
std::array<int, 10> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Объявим массив

for (int elem : MakeRange(arr.begin(), arr.begin() + 5)) {
  // ...
}
```

Замечание 2.1 *Шаблон, предложенный в задаче 3 позволяет проитерироваться по некоторой части элементов контейнера при помощи цикла range-based for без копирования необходимых элементов в другой контейнер, иными словами, без накладных расходов.*

3 Контейнер std::vector

Контейнер `std::vector`, определённый в заголовочном файле `<vector>` представляет из себя оболочку над динамическим массивом. Он реализован в виде шаблона с одним обязательным аргументом — типом данных элемента. Вектор сам выделяет память при добавлении элемента, правильно реализует операции копирования, а также сам освобождает память при выходе из области видимости. Итераторы вектора имеют точно такие же свойства, как и итераторы контейнера `array`. Вектор поддерживает обращение к элементу по индексу.

```
/* Объявим вектор int'ов из десяти элементов. Все элементы будут автоматически
  проинициализированы значениями по умолчанию, то есть нулями. Обратите внимание,
  что в контейнерах array, а также в обычных статических или динамических массивах из C
  это не делается. */
std::vector<int> v(10);
// Вектор поддерживает чтение элемента по индексу (переопределена операция []).
std::cout << "The third element is equal to " << v[3] << std::endl;
```

```
v[4] = 40; // И запись элемента по индексу.

std::cout << "Enter two elements " << std::endl; // Считаем пару элементов.
for (std::size_t i = 2; i < 4u; i++)
  std::cin >> v[i];

// Распечатаем вектор
for (std::size_t i = 0; i < v.size(); i++)
  std::cout << v[i] << std::endl;

// Можно и с помощью итератора
for (std::vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
  std::cout << *it << std::endl;

// По вектору можно проитерироваться при помощи range-based for
for (int elem : v)
  std::cout << elem << std::endl;

std::vector<int>::iterator it = v.begin(); // Итератор на начало вектора.

// Точно так же как и у массива к итераторам вектора можно прибавлять числа.
it += 5; // Теперь it указывает на элемент вектора с номером 5.
std::cout << *it << std::endl; // Выведет v[5].
```

Таким образом, вектор поддерживает те же основные операции, что и контейнер `array`. Поясним внутреннюю работу с памятью в векторе на примере

```
#include <iostream>
#include <vector>    // Это для вектора

// Функция печатает полуинтервал [itBegin, itEnd) по константному итератору вектора.
void PrintInterval(std::vector<int>::const_iterator itBegin,
                   std::vector<int>::const_iterator itEnd) {
  std::cout << "Vector = [ ";
  for (std::vector<int>::const_iterator it = itBegin; it != itEnd; ++it) {
    std::cout << *it << ' ';
  }
  std::cout << "]" << std::endl;
}

int main() {
  std::vector<int> v; // Объявление вектора int'ов. По умолчанию в векторе 0 элементов.

  for(int i = 0; i < 10; i++) // Метод push_back() кладёт элемент в конец вектора.
    v.push_back(i * i % 10);  // То есть после последнего элемента.

  std::size_t vSize = v.size(); // Как обычно метод size() возвращает кол-во эл-тов.
  std::cout << "Vector has " << vSize << " elements." << std::endl; // Выведет 10.

  // Метод empty() позволяет узнать, пуст ли вектор.
```

```
24  std::cout << v.empty() << std::endl;
25
26  /* Метод push_back() переодически перевыделяет память и копирует все элементы
27     в новый массив большего размера. Делает он это как только свободное место в старом
28     массиве заканчивается. push_back() делает размер нового массива в два раза
29     больше старого. Метод capacity() позволяет узнать текущий размер (в эл-тах)
30     выделенной памяти. Всегда выполнено size() <= capacity(). */
31  std::cout << "We can insert " << v.capacity() - v.size()
32            << " element(s) without reallocation" << std::endl; // Выведет 6.
33
34  // Можно удалить последний элемент. При этом ёмкость (capacity) не изменится.
35  v.pop_back();
36
37  // Можно прочитать последний элемент.
38  std::cout << v.back() << std::endl;
39
40  /* Увеличит количество элементов с 9 до 15. Старые элементы сохранятся, новые станут
41  равны нулю. Обычный массив НЕ ОБНУЛЯЛ ЭЛЕМЕНТЫ. Вектор инициализирует новые элементы
42  значениями по-умолчанию. */
43  v.resize(15);
44
45  v.resize(20, 101); // Увеличит размер с 15 до 20. Старые элементы сохранятся.
46  // новые элементы будут равны 101 (последние 5). Также произойдёт перевыделение памяти.
47
48  /* Можно сразу объявить вектор заданного размера. В этом векторе будет 10 элементов,
49     которые проинициализируются нулём. */
50  std::vector<int> v2(10);
51
52  PrintInterval(v2.begin(), v2.end());    // Выводим результат
53
54  // Изменит размер вектора с 10 до 15 и заполнит весь вектор элементами, равными 27.
55  v2.assign(15, 27);
56  PrintInterval(v2.begin(), v2.end());    // Выводим результат
57
58  /* Можно просто зарезервировать память в векторе. Метод reserve() выделит память,
59    то есть сделает capacity равной 10. Однако, в векторе будет по-прежнему 0 элементов.
60    Если бы в векторе были элементы до этого, то reserve() бы их скопировала
61    в новый массив. Теперь их можно закинуть туда с помощью push_back(). */
62  std::vector<int> v3; // Создаём пустой вектор.
63  v3.reserve(10); // Резервируем память
64
65  for (int i = 0; i < 10; i++)
66    v3.push_back(i*i*i); // Закидываем элементы в зарезервированную память.
67
68  PrintInterval(v3.begin(), v3.end());    // Выводим результат
69
70  // Вектор можно инициализировать списком в фигурных скобках (C++11).
71  std::vector<int> v4 = {1, 3, 5, 7, 9};
72
73  std::cout << v4.size() << std::endl; // В векторе 5 элементов.
74
```

```
75   v4.clear(); // Очистка вектора (capacity не изменится).
76   return 0;
77 }
```

Вектор поддерживает операции добавления элементов в произвольную позицию (метод `insert()`) и удаления элементов (метод `erase()`) из произвольной позиции. Однако, эти операции имеют линейную сложность. Синтаксис этих методов совпадает с синтаксисом аналогичных методов списка, про который говорится в разделе 5, поэтому не будем на них здесь подробно останавливаться.

Задача 4 *Напишите функцию*

```
1 std::vector<double> GetGreaterThanMedian(std::istream& stream);
```

которая считывает все доступные дробные числа из потока `stream` *и возвращает вектор, содержащий элементы, большие медианы считанного массива. Этот вектор должен быть отсортирован в порядке убывания. Для сортировки можно использовать функцию* `QuickSort()` *из задачи 2 или аналогичную функцию* `std::sort()` *из заголовочного файла* `<algorithm>` .

```
1 std::vector<double> vec = ...;
2
3 // Сортировка по возрастанию
4 QuickSort(vec.begin(), vec.end());
5
6 // Сортировка по убыванию
7 QuickSort(vec.rbegin(), vec.rend());
```

Задача 5 *Напишите операцию вывода произвольного вектора* `<<` *в поток общего вида* `std::ostream` *в формате* `[a1, a2,..., aN]`*, а также операцию ввода произвольного вектора в указанном формате из потока общего вида. Гарантируется, что элементы вектора умеют считываться и записываться, то есть для них определены операции ввода/вывода, однако операция ввода может выдать ошибку во время считывания. При вводе пробелы могут стоять как угодно и в каком угодно количестве (то есть пробелы следует игнорировать). При выводе указанный формат должен строго соблюдаться. Если операция ввода не может считать вектор, она должна выставлять входному потоку состояние* `std::ios_base:failbit`.

```
1 template<typename ElemType>
2 std::ostream& operator<<(std::ostream& out, const std::vector<ElemType>& vec);
3
4 template<typename ElemType>
5 std::istream& operator>>(std::istream& in, std::vector<ElemType>& vec);
```

3.1 Инвалидация элементов вектора

Ссылки на элементы вектора, а также итераторы на эти элементы могут инвалидироваться при добавлении или удалении элементов. Рассмотрим пример

```
1   std::vector<int> vec = {1, 2, 3, 4, 5};
2   std::vector<int>::iterator it = vec.begin() + 3; // Указывает на число 4
```

```
3   const int& elem = vec[2]; // Указывает на число 3.
4
5   std::cout << *it << std::endl;      // Выведет 4.
6   std::cout << elem << std::endl;     // Выведет 3.
7
8   /* Удалим из вектора первый элемент, то есть число 1. Функция принимает
9     итератор на удаляемый элемент и возвращает итератор на элемент после удалённого. */
10  vec.erase(vec.begin());
11
12  std::cout << *it << std::endl;      // Выведет 5.
13  std::cout << elem << std::endl;     // Выведет 4.
```

Заметим, что значения поменялись. Это особенно странно для ссылки `elem`, которая была объявлена как `const`. Дело в том, что вектор удалил первый элемент и уплотнил оставшиеся элементы к началу. Таким образом, рассмотренные ссылка и итератор указывают на ту же позицию в массиве, однако сами значения в этих местах поменялись, иными словами, ссылка и итератор инвалидировались. Заметим, что при добавлении элементов в вектор может произойти ситуация ещё хуже: вектор может увеличить ёмкость, то есть выделить память, скопировать все элементы в неё, а старую память освободить. В таком случае обращение к элементу по ссылке или по итератору может привести к `segmentation fault`.

Отметим, что если бы мы удаляли элемент из конца вектора, то ссылки и итераторы на остальные элементы не инвалидировались бы.

4 Контейнер std::deque

Контейнер `std::deque` (*double ended queue*), определённый в заголовочном файле `<deque>`, представляет из себя блочный динамический массив. Память в нём выделяется фиксированными блоками, размер блока зависит от реализации стандартной библиотеки (в libstdc++ выделяется по 8 элементов). Память под каждый блок выделяется по отдельности. Таким образом, в отличие от вектора, дек при добавлении элемента увеличивает ёмкость (при необходимости) на постоянную величину, в то время как вектор увеличивает в два раза.

Точно так же, как и вектор, дек умеет быстро добавлять элементы в конец и удалять их из конца (за $O(1)$). Кроме того, дек умеет быстро добавлять элементы в начало, а также быстро удалять их из начала контейнера (за $O(1)$).

Итераторы дека поддерживают все те же операции, что и итераторы вектора. Однако из-за блочной структуры дека его итераторы перемещаются по контейнеру медленнее, чем итераторы вектора. Кроме того, доступ к элементу дека работает медленнее, чем доступ к элементу вектора за счёт двойного разыменования указателя.

С деком можно производить все те же операции (кроме `capacity()`), что и с вектором, поэтому приведём пример только тех методов, которых у вектора нет.

```
1  #include <iostream>
2  #include <deque>
3
4  int main() {
5    // Обязательный шаблонный параметр - тип данных элемента
6    std::deque<int> dq;
7
8    for (int i = 0; i < 10; i++)
9      dq.push_front(i * i); // Добавление элемента в начало.
```

```
10
11   dq.pop_front(); // Удаление элемента из начала дека.
12
13   std::cout << dq.front() << std::endl; // Вывод первого элемента дека.
14   return 0;
15 }
```

Отметим, что при добавлении элементов в начало или конец дека, а также при удалении элементов из начала или конца дек никогда не инвалидирует ссылки на остальные элементы. При этом итераторы инвалидируются. При добавлении в середину дека или при удалении элемента из середины дека инвалидируются все итераторы и ссылки.

Задача 6 *Напишите класс для расчёта средней температуры по больнице.*

```
1 struct Date { int year; int month; int day; };
2
3 struct Time { int hours; int minutes; int seconds; };
4
5 struct Record { Date recordDate; Time recordTime; int value; };
6
7 class TemperatureStats {
8  public:
9   void AddRecord(Record record);
10  double PerformReport(Date tillDate, Time tillTime);
11 private:
12  std::deque<Record> data;
13 };
```

Метод `AddRecord()` *добавляет запись в базу, метод* `PerformReport()` *считывает среднюю температуру с момента предыдущего замера не включительно до момента времени* `(tillDate, tillTime)` *включительно. Для структур* `Date`, `Time` *и* `Record` *необходимо написать операцию "меньше", сравнивающую их в календарном порядке. Сравнивать различные структуры между собой нет необходимости. Гарантируется, что в метод* `AddRecord()` *передаются моменты времени в порядке нестрогого возрастания. Новые переменные в класс добавлять не нужно. Гарантируется, что дата и время вводятся корректно. Температура измеряется в десятых долях градусов. Дни и месяцы нумеруются с нуля.*

5 Контейнер std::list

Контейнер `std::list`, определённый в заголовочном файле `<list>` представляет из себя двусвязный список. Он реализован в виде шаблона с одним обязательным параметром — типом данных элемента. Внутренне элементы списка хранятся в ячейках вида

```
1 template<typename ElemType>
2 struct ListNode {
3   ElemType data;
4   ListNode* prev;
5   ListNode* next;
6 };
```

Каждая ячейка ссылается на предыдущую и следующую. В каждой ячейке хранится один элемент.

Таким образом список поддерживает быстрое добавление и удаление элемента в произвольном месте списка. Однако, он не поддерживает произвольный доступ к элементу по индексу. Из каждой ячейки можно обратиться только к соседней. То есть для списка не определена операция `[]`.

У списка есть методы `push_back()`, `pop_back()`, `back()`, `push_front()`, `pop_front()`, `front()`, `size()` и `empty()`. Они работают точно так же как и в деке. Рассмотрим более подробно методы, специфичные именно для списка, поскольку в отличии от вектора и дека, эти методы для списка имеют сложность $O(1)$.

```
#include <iostream>
#include <vector>
#include <list>

int main() {
  // Можно инициализировать списком в фигурных скобках (C++11)
  std::list<int> myList = {1, 2, 3, 4, 5};

  std::list<int>::iterator it = myList.begin();

  /* К итератору списка нельзя прибавлять числа поскольку список не поддерживает
  произвольный доступ. Однако, работают функции next() и prev(), их скорость
  работы зависит линейно от величины сдвига. */
  auto it2 = std::next(it, 2);

  /* Можно удалять элемент по итератору. Функция возвращает итератор на следующий
   элемент. */
  std::list<int>::iterator itNextAfter = myList.erase(it2); // Удалит число 3.

  std::cout << *itNextAfter << std::endl; // выведет 4.

  /* Можно удалить полуинтервал. Функция возвращает итератор на следующий элемент
    после последнего удалённого. Сложность линейно зависит от длины полуинтервала. */
  myList.erase(itNextAfter, myList.end()); // Останется только [1, 2].

  /* Можно вставить элемент перед элементом, указанным итератором. Функция возвращает
  итератор на только что вставленный элемент. */
  std::list<int>::iterator insertedIt = myList.insert(std::next(myList.begin()), 11);

  for (int elem : myList)     // Выведет список [1, 11, 2].
    std::cout << elem << std::endl;

  // То же самое можно делать и с набором элементов, заданным полуинтервалом.
  std::vector<int> toInsert = {100, 101, 102, 103};
  /* Первый параметр задаёт позицию, куда вставить. Второй и третий параметры
    определяют полуинтервал, который нужно вставить. Функция возвращает итератор
    на первый вставленный элемент. Сложность линейно зависит от длины вставки. */
  myList.insert(std::next(myList.begin()), toInsert.begin(), toInsert.end());

  /* Можно перенести элементы из одного списка в другой или в другое место
    того же самого списка. */
```

```
  std::list<int> firstList = {1, 2, 3, 4, 5};
  std::list<int> secondList = {10, 11, 12, 13, 14, 15};

  /* Перенесём элементы [10, 11, 12] из второго списка в первый на место
    между числами 1 и 2. */
  /* Перенос элементов в другой список имеет сложность O(1). Однако стоимость подсчёта
    нового количества элементов списка линейно зависит от длины переносимого куска
    при условии, что списки не совпадают. */
  firstList.splice(next(firstList.begin()), // На какую позицию вставить
                   secondList,              // Из какого списка перенести полуинтервал
                   secondList.begin(),      // Начало переносимого полуинтервала
                   next(secondList.begin(), 3)); // Конец переносимого полуинтервала
  return 0;
}
```

В отличие от итераторов вектора, массива и дека, итераторы списка предоставляют только последовательный доступ. Они не поддерживают операцию прибавления числа. Их можно только инкрементировать и декрементировать.

Ещё одной особенностью списка является неинвалидация итераторов и ссылок при добавлении и удалении элементов (разумеется, кроме самого удаляемого элемента) в произвольную позицию.

Задача 7 *Напишите класс, реализующий внутренности однострочного текстового редактора.*

```
class TextEdit {
 public:
  TextEdit();
  void Insert(char c);
  void StepRight();
  void StepLeft();
  void StartSelection();
  void Copy();
  void Cut();
  void Paste();
  void Save(std::ostream& stream);

  const std::list<char>& Text() const { return text; }
  const std::list<char>& Clipboard() const { return clipboard; }
  const std::list<char>::const_iterator& Pos() const { return pos; }
  const std::list<char>::const_iterator& SelectionStartPos() const {
    return selectionStartPos;
  }
  bool SelectionMode() const { return selectionMode; }
  bool ReversedSelection() const { return reversedSelection; }
 private:
  std::list<char> text;                 // Введённый текст
  std::list<char> clipboard;            // Буфер обмена
  std::list<char>::const_iterator pos; // Текущая позиция курсора
  // Начало или конец выделенного промежутка
  std::list<char>::const_iterator selectionStartPos;
  bool selectionMode;                  // Включен ли режми выделения текста
```

```
28   bool reversedSelection;            // Необходимо ли инвертировать выделенный интервал
29 };
```

Поле `selectionMode` *равно* `true`*, если включен режим выделения текста. Поле* `reversedSelection` *служит для того, чтобы показать, что итератор* `pos` *указывает на элемент списка, расположенный раньше элемента, на который указывает итератор* `selectionStartPos`*. Итератор* `selectionStartPos` *указывает на символ, с которого начали выделять текст.*

Методы класса должны делать следующие действия:

- *конструктор инициализирует поле* `pos`*, полю* `selectionMode` *присваивает значение* `false`*;*
- *метод* `Insert()` *вставляет в текст букву на позицию каретки* `pos`*, каретка после завершения функции указывает на элемент, следующий за вставленным;*
- *метод* `Delete()` *удаляет текущий символ, каретка после завершения функции указывает на элемент, следующий за удалённым;*
- *метод* `StepLeft()` *перемещает каретку на одну позицию влево (если есть);*
- *метод* `StepRight()` *перемещает каретку на одну позицию вправо (если есть);*
- *метод* `StartSelection()` *выставляет переменную* `selectionStartPos` *в текущую позицию курсора, полю* `selectionMode` *присваивает значение* `true`*, поле* `reversedSelection` *инициализируется произвольным значением;*
- *метод* `Copy()` *сначала очищает буфер обмена, затем если включен режим выделения, копирует полуинтервал текста* `[selectionStartPos, pos)` *от* `selectionStartPos` *включительно до* `pos` *не включительно в буфер обмена (или полуинтервал* `[pos, selectionStartPos)` *если поле* `reversedSelection` *имеет значение* `true`*, то есть полуинтервал инвертировался в результате перемещений каретки);*
- *метод* `Cut()` *делает то же самое, что и* `Copy()`*, только дополнительно ещё и вырезает скопированный кусок из текста, после завершения функции каретка должна указывать на элемент, стоящий за последним удалённым;*
- *метод* `Paste()` *вставляет текст из буфера обмена в то место, на которое указывает каретка* `pos`*, после чего очищает буфер обмена, после завершения функции каретка должна указывать на элемент, стоящий за последним добавленным;*
- *метод* `Save()` *выводит текст в поток вывода.*

Кроме того, методы `Insert()`*,* `Delete()`*,* `Copy()`*,* `Cut()` *и* `Paste()` *присваивают полю* `selectionMode` *значение* `false`*. Методы* `StepLeft()` *и* `StepRight()` *при необходимости модифицируют переменную* `reversedSelection` *в режиме выделения текста.*

6 Другие контейнеры

Кроме обычного списка в стандартной библиотеке есть односвязный список `std::forward_list`, отличающийся от списка `std::list` тем, что каждая его ячейка хранит указатель только на одного соседа. В связи с этим по такому списку можно итерироваться только в одну сторону. Кроме того, немного меняются методы вставки и удаления элементов. Однако, принцип работы похож на обычный список.

В стандартной библиотеке также имеются адаптеры, основанные на других контейнерах. Как правило, они принимают необязательный, заданный по-умолчанию, шаблонный параметр — тип контейнера, на базе которого они работают. Среди них шаблоны `std::stack`, `std::queue` и шаблон `std::priority_queue`, интересный тем, что предоставляет доступ за $O(1)$ к самому большому элементу взамен логарифмической стоимости добавления и удаления элемента.

7 Особенности различных контейнеров

7.1 Массив std::array

- Выделяет память на стеке, что существенно быстрее выделения памяти из кучи;
- Размер массива фиксирован и определяется на этапе компиляции;
- Предоставляет произвольный доступ к элементу в том числе и по индексу;
- Не инвалидирует ссылки и итераторы.

7.2 Вектор

- Может менять (увеличивать) свою ёмкость на этапе выполнения программы;
- Предоставляет произвольный доступ к элементу в том числе и по индексу;
- Позволяет добавлять элементы в конец и удалять из конца за $O(1)$;
- Сложность вставки в начало или середину, как и сложность удаления оттуда линейно зависит от размера вектора;
- Инвалидирует ссылки и итераторы при добавлении/удалении элементов или увеличении размера.

7.3 Дек

- Может менять (увеличивать и уменьшать) свою ёмкость на этапе выполнения программы;
- Предоставляет произвольный доступ к элементу в том числе и по индексу, однако чуть медленнее, чем в векторе;
- Позволяет добавлять элементы в конец или начало, а также удалять из конца или начала за $O(1)$.
- Сложность вставки элемента в середину, а также удаления из середины линейно зависит от размера дека;
- Не инвалидирует ссылки при добавлении в конец или начало, а также при удалении элемента из конца или начала (кроме ссылки на удаляемый элемент). В остальных случаях ссылки инвалидируются. Итераторы инвалидируются всегда.

7.4 Список

- Может менять (увеличивать и уменьшать) свою ёмкость на этапе выполнения программы;
- Предоставляет последовательный доступ к элементам.
- Позволяет быстро добавлять на любую позицию и удалять элементы из любой позиции по итератору за $O(1)$.
- Не инвалидирует ссылки и итераторы.

Список литературы

[1] `https://en.cppreference.com/w/cpp/container`