

Последовательные контейнеры

Как уже отмечалось, есть две основные категории контейнеров: последовательные и ассоциативные. В этом разделе мы рассмотрим четыре последовательных контейнера (массивы (array), векторы, списки и очереди с двусторонним доступом), детально обсуждая, как работает каждый из них и какие в них имеются методы. Массивы (array), векторы (vector), двусторонние очереди (deque) и списки (list) поддерживают разные наборы операций, среди которых есть совпадающие операции. Разные виды контейнеров используют операции с одинаковыми именами и характеристиками, поэтому, например, метод `push_back()` для векторов будет ничуть не хуже работать со списками и очередями. Они могут быть реализованы с разной эффективностью:

Операции	Метод	vector	deque	list
Вставка в начало	<code>push_front</code>	-	+	+
Удаление из начала	<code>pop_front</code>	-	+	+
Вставка в конец	<code>push_back</code>	+	+	+
Удаление из конца	<code>pop_back</code>	+	+	+
Вставка в произвольное место	<code>insert</code>	(+)	(+)	+
Удаление из произвольного места	<code>erase</code>	(+)	(+)	+
Произвольный доступ к элементу	<code>[]</code> , <code>at</code>	+	+	-

Знак + означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества n элементов в контейнере. Знак (+) означает, что соответствующая операция реализуется за время, пропорциональное n . Для малых n время операций, обозначенных +, может превышать время операций, обозначенных (+), но для большого количества элементов последние могут оказаться очень дорогими.

Как видно из таблицы, такими операциями являются вставка и удаление произвольных элементов очереди и вектора, поскольку при этом все последующие элементы требуется переписывать на новые места.

Итак, *вектор* — это структура, эффективно реализующая произвольный доступ к элементам, добавление в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

Каждый пример в следующих разделах будет представлять несколько методов описываемого контейнера.

На рис. 1 показана работа некоторых особо важных методов для трех последовательных контейнеров.

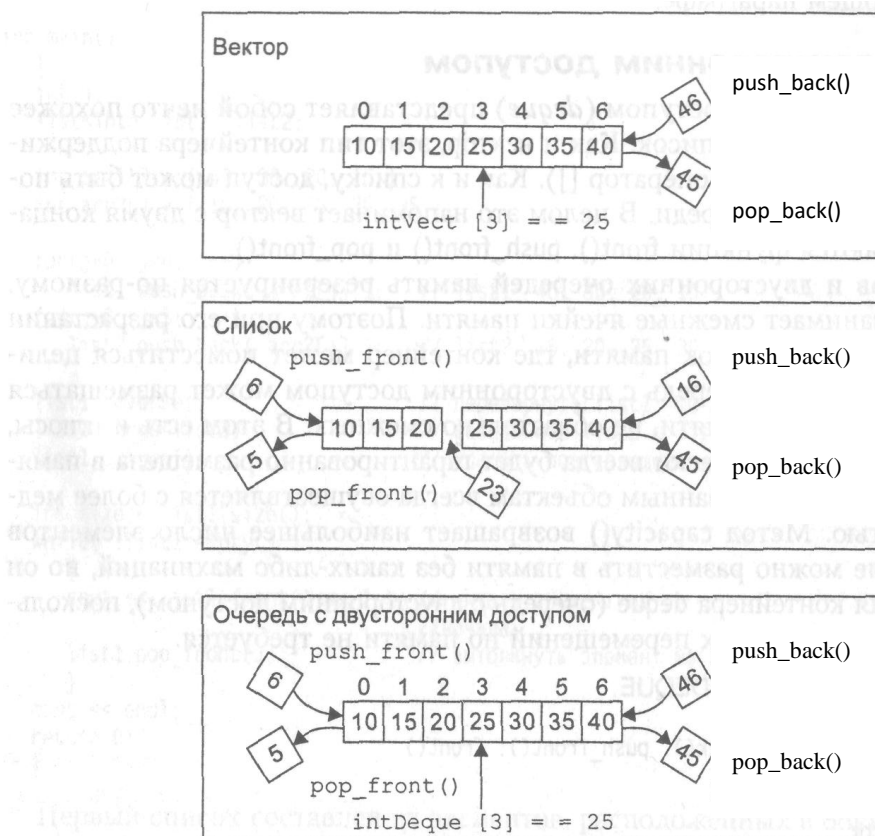


Рис. 1. Последовательные контейнеры

Массивы (std::array)

В C++ есть **фиксированные** и **динамические** массивы. Они очень полезны и активно используются в C++, но у них есть свои недостатки: фиксированные массивы **распадаются в указатели**, теряя информацию о своей длине; в динамических массивах проблемы могут возникнуть с освобождением памяти и с попытками изменить свою длину после выделения.

Поэтому, в стандартную библиотеку C++, добавили новые контейнеры, которые упрощают процесс управления массивами: std::array.

Шаблон классов

```
template <class Ty, std::size_t N>
```

```
class array;
```

описывает объект, управляющий последовательностью из N элементов типа Ty. Последовательность хранится как массив Ty в объекте array<Ty, N>.

Функция-член	Описание
array	Создает объект массива.
assign	Заменяет все элементы.
at	Обращается к элементу в указанной позиции.
back	Обращается к последнему элементу.
begin	Задаёт начало управляемой последовательности.

Функция-член	Описание
<code>cbegin</code>	Возвращает постоянный итератор произвольного доступа, указывающий на первый элемент в массиве.
<code>cend</code>	Возвращает постоянный итератор произвольного доступа, указывающий на позицию после последнего элемента массива.
<code>crbegin</code>	Возвращает константный итератор, который указывает на первый элемент в обратном массиве.
<code>crend</code>	Возвращает постоянный итератор, который указывает на позицию после последнего элемента в обратном массиве.
<code>data</code>	Получает адрес первого элемента.
<code>empty</code>	Проверяет наличие элементов.
<code>end</code>	Задаёт конец управляемой последовательности.
<code>fill</code>	Заменяет все элементы указанным значением.
<code>front</code>	Обращается к первому элементу.
<code>max_size</code>	Подсчитывает количество элементов.
<code>rbegin</code>	Задаёт начало обратной управляемой последовательности.
<code>rend</code>	Задаёт конец обратной управляемой последовательности.
<code>size</code>	Подсчитывает количество элементов.
<code>swap</code>	Меняет местами содержимое двух контейнеров.

Оператор	Описание
<code>array::operator=</code>	Заменяет управляемую последовательность.
Оператор <code>Array::[]</code>	Обращается к элементу в указанной позиции.

Примечания

У этого типа есть конструктор по умолчанию `array()` и оператор присваивания по умолчанию `operator=`. Тип удовлетворяет требованиям для aggregate. Поэтому объекты типа `array<Тy, N>` можно инициализировать с помощью агрегатного инициализатора. Например, применённая к объекту директива,

```
array<int, 4> ai = { 1, 2, 3 };
```

создаёт объект `ai`, содержащий четыре целочисленных значения, инициализирует первые три элемента как 1, 2 и 3 соответственно и инициализирует четвёртый элемент как 0.

```
#include "stdafx.h"
#include <iostream>
#include <array>

using namespace std;
```

```

template <class T>
void Print(const T& myarr)
{
    for (const auto & x : myarr)
        cout << x << ' ';
    cout << endl;
}

int main()
{
    array<int, 5> myarr; //Объявляем массив типа int длиной 5
    myarr = { 2, 4, 5, 6, 8 }; //Инициализируем массив начальными значениями.
    Print(myarr);
    array<int, 5> myarr1 = { 1, 5, 3, 7, 9 }; //Объявляем и инициализируем.

    array<int, 5> myarr2 { 5, 4, 3, 2, 1 }; //Объявляем и инициализируем.

    array<int, 5> myarr3(myarr); //Объявляем и инициализируем.
    myarr3 = { 7, 9 }; //Заменяем первые два значения, остальные обнуляем.
    Print(myarr3);

    return 0;
}

```

```

2 4 5 6 8
7 9 0 0 0
Для продолжения нажмите любую клавишу . . .

```

Подобно обычным фиксированным массивам, длина `std::array` должна быть установлена во время компиляции. `std::array` можно инициализировать с использованием списка инициализаторов или `uniform` инициализации:

```
std::array<int, 4> myarray = { 8, 6, 4, 1 }; // список инициализаторов
```

```
std::array<int, 4> myarray2 { 8, 6, 4, 1 }; // uniform инициализация
```

В отличие от стандартных фиксированных массивов, в `std::array` вы не можете пропустить (не указывать) длину массива:

```
std::array<int, > myarray = { 8, 6, 4, 1 }; // нельзя, должна быть указана длина массива
```

Также можно присваивать значения массиву с помощью списка инициализаторов:

```
std::array<int, 4> myarray;
```

```
myarray = { 0, 1, 2, 3 }; // Верно
```

```
myarray = { 8, 6 }; // Элементам 2 и 3 присвоен ноль!
```

```
myarray = { 0, 1, 3, 5, 7, 9 }; // Не допускается, слишком много элементов в списке
```

инициализаторов!

Доступ к значениям массива через оператор индекса осуществляется как обычно:

```
std::cout << myarray[1];
```

```
myarray[2] = 7;
```

Так же, как и в стандартных фиксированных массивах, оператор индекса не выполняет никаких проверок на диапазон. `std::array` поддерживает вторую форму доступа к элементам массива — функция `at()`, которая осуществляет проверку диапазона:

```
std::array<int, 4> myarray { 8, 6, 4, 1 };
```

```
myarray.at(1) = 7; // Элементу массива под индексом 1 присваиваем значение 7.
```

```
myarray.at(8) = 15; // Индекс 8 - некорректный, получим исключение.
```

В примере выше, вызов `myarray.at(1)` проверяет, есть ли элемент массива под индексом 1, и, поскольку он есть, возвращается ссылка на этот элемент. Затем мы присваиваем ему значение 7. Однако, вызов `myarray.at(8)` не работает, так как элемента под индексом 8 в массиве нет. Функция `at()` выдаёт ошибку, которая завершает работу программы (Примечание: на самом деле выбрасывается исключение типа `std::out_of_range`). Поскольку проверка диапазона выполняется, то `at()` работает медленнее (но безопаснее), чем оператор `[]`.

`std::array` автоматически делает все очистки после себя, когда выходит из области видимости, поэтому нет необходимости прописывать это вручную.

В следующем примере показано использования алгоритма `std::sort` для сортировки по убыванию и возрастанию.

```
// Example_3.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <iostream>
#include <array>
#include <algorithm> // для std::sort

int main()
{
    std::array<int, 5> myarray{ 8, 4, 2, 7, 1 };
    std::sort(myarray.begin(), myarray.end()); // Сортировка массива по
    возрастанию.

    for (const auto &element : myarray)
        std::cout << element << ' ';
    std::cout << std::endl;
    std::sort(myarray.rbegin(), myarray.rend()); // Сортировка массива по
    убыванию.

    for (const auto &element : myarray)
        std::cout << element << ' ';
    std::cout << std::endl;
```

```

        return 0;
}

```

Результат работы программы.

```

1 2 4 7 8
8 7 4 2 1

```

Пример использования функций front, back.

```

// Example_4.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <array>
#include <iostream>
using namespace std;

typedef array<int, 5> vector;

void Shift_Left(vector& v)
{
    int t = v.front();
    for (size_t i = 0; i < v.size() - 1; i++)
    {
        v.at(i) = v[i + 1];
    }
    v.back() = t;
}

int main()
{
    vector v = { 1,2,3,4,5 };
    Shift_Left(v);
    for (const auto &x : v)
    {
        cout << x << ' ';
    }
    cout << endl;
    return 0;
}

```

```

2 3 4 5 1

```

```

Для продолжения нажмите любую клавишу . . .

```

std::array — это отличная замена стандартных фиксированных массивов. Они более эффективны, так как используют меньше памяти. Единственными недостатками std::array по сравнению со стандартными фиксированными массивами являются немного неудобный синтаксис и то, что нужно явно указывать длину массива (компилятор не будет вычислять её за нас). Но это сравнительно незначительные нюансы. Рекомендуется использовать std::array вместо стандартных фиксированных массивов в любых нетривиальных задачах.

Векторы

Векторы — это, так сказать, умные массивы. Они занимаются автоматическим размещением себя в памяти, расширением и сужением своего размера по мере вставки или удаления данных. Векторы можно использовать в какой-то мере как массивы, обращаясь к элементам с помощью привычного оператора индексации []. В векторах случайный доступ выполняется очень быстро.

Также довольно быстро осуществляется добавление (или проталкивание) новых данных в конец вектора. Когда это происходит, размер вектора автоматически увеличивается для того, чтобы было, куда положить новое значение.

Для создания вектора можно воспользоваться следующими конструкторами (приведена упрощенная запись):

```
explicit vector(); // 1
explicit vector(size_type n, const T& value = T()); // 2
template <class InputIter > // 3
vector(InputIter first, InputIter last);
vector(const vector<T>& x); //4
```

Ключевое слово `explicit` используется для того, чтобы при создании объекта запретить выполняемое неявно преобразование при присваивании значения другого типа.

Конструктор 1 является конструктором по умолчанию.

Конструктор 2 создает вектор длиной `n` и заполняет его одинаковыми элементами — копиями `value`.

Поскольку изменение размера вектора обходится дорого, при его создании задавать начальный размер весьма полезно. При этом для встроенных типов выполняется инициализация каждого элемента значением `value`. Если оно не указано, элементы глобальных векторов инициализируются нулем.

Если тип элемента вектора определен пользователем, начальное значение формируется с помощью конструктора по умолчанию для данного типа. На месте второго параметра можно написать вызов конструктора с параметрами, создав, таким образом, вектор элементов с требуемыми свойствами (см. пример далее).

Элементы любого контейнера являются копиями вставляемых в него объектов. Поэтому для них должны быть определены конструктор копирования и операция присваивания.

Конструктор 3 создает вектор путем копирования указанного с помощью итераторов диапазона элементов. Тип итераторов должен быть «для чтения».

Конструктор 4 является конструктором копирования.

Примеры конструкторов:

// Создается вектор из 10 равных единице элементов:

```
vector<int> v2(10, 1);
```

// Создается вектор, равный вектору `v1`:

```
vector<int> v4(v1);
```

// Создается вектор из двух элементов, равных первым двум элементам `v1`:

```
vector<int> v3(v1.begin(), v1.begin() + 2);
```

```
// Создается вектор из 10 объектов класса monstr.
// (работает конструктор по умолчанию):
vector <monstr> m1(10);
// Создается вектор из 5 объектов класса monstr с заданным именем
// (работает конструктор с параметром char*):
vector <monstr> m2(5, monstr("Вася"));
```

В шаблоне vector определены операция присваивания и функция копирования:

```
vector<T>& operator=(const vector<T>& x);
void assign(size_type n, const T& value);
template <class InputIter>
void assign(InputIter first, InputIter last);
```

Здесь через T обозначен тип элементов вектора. Вектора можно присваивать друг другу точно так же, как стандартные типы данных или строки. После присваивания размер вектора становится равным новому значению, все старые элементы удаляются.

Функция assign в первой форме аналогична по действию конструктору 2, но применяется к существующему объекту. Функция assign во второй форме предназначена для присваивания элементам вызывающего вектора значений из диапазона, определяемого итераторами first и last, аналогично конструктору 3, например:

```
vector <int> v1, v2;
// Первым 10 элементам вектора v1 присваивается значение 1:
v1.assign(10,1);
// Первым 3 элементам вектора v2 присваиваются значения v1[5], v1[6]. v1[7]:
v2.assign(v1.begin() + 5, v1.begin() + 8);
```

Итераторы класса vector перечислены в табл. 12.

Доступ к элементам вектора осуществляется с помощью следующих операций и методов:

```
reference      operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference      at(size_type n);
reference      front();
const_reference front() const;
reference      back();
const_reference back() const;
```

Операция [] осуществляет доступ к элементу вектора по индексу без проверки его выхода за границу вектора. Функция at выполняет такую проверку и порождает исключение out_of_range в случае выхода за границу вектора. Естественно, что функция at работает медленнее, чем операция [], поэтому в случаях, когда диапазон определен явно, предпочтительнее пользоваться операцией:


```
for (int i = 0; i<v.size(); i++) cout << v[i] << " ";
```

В противном случае используется функция `at` с обработкой исключения:

```
try
{
    v.at(i) = v.at(...);
}
catch(out_of_range)
{ ... }
```

Операции доступа возвращают значение ссылки на элемент (`reference`) или константной ссылки (`const_reference`) в зависимости от того, применяются ли они к константному объекту или нет.

Методы `front` и `back` возвращают ссылки соответственно на первый и последний элементы вектора (это не то же самое, что `begin` — указатель на первый элемент и `end` — указатель на элемент, следующий за последним). Пример:

```
vector<int> v(5, 10);
v.front() = 100;

v.back() = 100;
cout << v[0] << " " << v[v.size() - 1]; //Вывод: 100 100
```

Функция **capacity** определяет размер оперативной памяти, занимаемой вектором:

`size_type capacity() const`; Память под вектор выделяется динамически, но не под один элемент в каждый момент времени (это было бы расточительным расходом ресурсов), а сразу под группу элементов, например, 256 или 1024. Перераспределение памяти происходит только при превышении этого количества элементов, при этом объем выделенного пространства удваивается. После перераспределения любые итераторы, ссылающиеся на элементы вектора, становятся недействительными, поскольку вектор может быть перемещен в другой участок памяти, и нельзя ожидать, что связанные с ним ссылки будут обновлены автоматически. Существует также функция выделения памяти **reserve**, которая позволяет задать, сколько памяти требуется для хранения вектора:

```
void reserve(size_type n);
```

Пример применения функции:

```
vector<int> v;
// Выделение памяти под 1000 элементов
v.reserve(1000);
```

Методы `push_back()`, `size()` и `operator[]`

Наш первый пример касается самых общих векторных операций.

Листинг 11. Программа VECTOR

```
//-----
// vector.cpp
// Демонстрация push_back(), operator[], size()
#include "stdafx.h"
#include <iostream>
```

```

#include <vector>
#include "windows.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Создать вектор типа int.
    vector<int> v ;
    v.push_back(10);      // внести данные в конец вектора
    v.push_back(11);
    v.push_back(12);
    v.push_back(13);
    // Вывести содержимое. 10 11 12 13
    for(const auto &x : v)
        cout << x << ' ';
    cout << endl;
    // Заменить новыми значениями.
    v[0] = 20;
    v[3] = 23;
    // Вывести содержимое. 20 11 12 13
    for(int j = 0; j < v.size(); j++)
        cout << v[j] << ' ';
    cout << endl;
    system("PAUSE");
    return 0;
}

```

Результат работы программы:

```

10 11 12 13
20 11 12 23
Для продолжения нажмите любую клавишу . . .

```

Для создания вектора `v` используется штатный конструктор вектора без параметров. Как с любыми контейнерами STL, для задания типа переменных, которые будут храниться в векторе, используется шаблонный формат (в данном случае это тип `int`). Мы не определяем размер контейнера, поэтому вначале он равен 0.

Метод `push_back()` вставляет значение своего аргумента в конец вектора (конец располагается там, где находится самый большой индекс). Начало вектора (элемент с индексом 0), в отличие от списков и очередей, не может использоваться для вставки новых элементов. Здесь мы проталкиваем значения 10, 11, 12 и 13 таким образом, что `v[0]` содержит 10, `v[1]` содержит 11, `v[2]` содержит 12, и `v[3]` содержит 13.

Как только в векторе появляются какие-либо данные, к ним сразу может быть получен доступ с помощью перегруженного оператора `[]`. Точно тот же прием, что и при работе с массивами, как видите. Очень удобно. Этот оператор мы использовали в нашем примере, чтобы заменить первый элемент с 10 на 20, а последний — с 13 на 23. Таким образом, результат программы получился следующим:

```
20 11 12 23
```

Метод `size()` возвращает текущее число элементов, содержащихся в контейнере. Для программы VECTOR это 4. Это значение используется в цикле `for` для вывода значений вектора на экран.

Есть еще один метод, `max_size()`, который мы не демонстрировали здесь. Он возвращает максимальный размер, до которого может расшириться контейнер. Это число зависит от типа хранимых в нем данных (понятно, что чем больше занимает один элемент

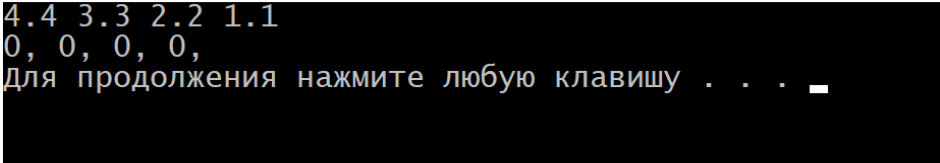
данного типа, тем меньше элементов мы сможем хранить), типа контейнера и операционной системы. Например, на нашей системе `max_size()` возвращает 1 073 741 823 для целочисленного вектора.

Методы `swap()`, `empty()`, `back()` и `pop_back()`

Следующий пример, `VECTCON`, демонстрирует еще несколько методов и векторов.

```
//-----  
// vectcon.cpp  
// демонстрация конструкторов, swap(), empty(), back(), pop_back()  
  
#include "stdafx.h"  
#include <iostream>  
#include <vector>  
#include "windows.h"  
using namespace std;  
  
int _tmain(int argc, _TCHAR* argv[])  
{  
    SetConsoleCP(1251);  
    SetConsoleOutputCP(1251);  
    // Массив типа double.  
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };  
    // Инициализация вектора массивом.  
    vector<double> v1(arr, arr + 4);  
    // Пустой вектор. Размер = 4.  
    vector<double> v2(4);  
    // Обменять содержимое v1 и v2.  
    v1.swap(v2);  
    // Пока вектор не будет пуст,  
    while (!v2.empty())  
    {  
        // вывести последний элемент  
        cout << v2.back() << ' ';  
        // и удалить его.  
        v2.pop_back();  
    }  
    cout << endl;  
    // Вывод: 4.4 3.3 2.2 1.1  
    for (const auto &x : v1)  
        cout << x << ", ";  
    cout << endl;  
    system("PAUSE");  
    return 0;  
}  
//-----
```

Результат работы программы:



```
4.4 3.3 2.2 1.1  
0, 0, 0, 0,  
Для продолжения нажмите любую клавишу . . . _
```

В этой программе мы использовали два новых конструктора векторов. Первый инициализирует вектор **v1** значениями обычного массива C++, переданного ему в качестве аргумента. Аргументами этого конструктора являются указатели на начало массива и на элемент «после последнего». Во втором конструкторе вектор **v2** инициализируется установкой его размера. Мы положили его равным **4**. Но значение самого вектора при инициализации не передается. Оба вектора содержат данные типа **double**.

Метод **swap()** обменивает данные одного вектора на данные другого, при этом порядок следования элементов не изменяется. В этой программе в векторе **v2** содержатся произвольные значения, которые будут обмениваются на значения из вектора **v1**. Результат работы программы таков:

4.4. 3.3. 2.2. 1.1

Метод **back()** возвращает значение последнего элемента вектора. Мы выводим его с помощью **cout**. Метод **pop_back()** удаляет последний элемент вектора.

Таким образом, при каждом прохождении цикла последний элемент будет иметь разные значения. (Немного удивительно, что **pop_back()** только удаляет последний элемент, но не возвращает его значение, как **pop()** при работе со стеком. Поэтому, в принципе, всегда нужно использовать **pop_back()** и **back()** в паре.)

Некоторые методы, например **swap()**, существуют и в виде алгоритмов. В таких случаях лучше предпочесть метод алгоритму. Работа с методом для конкретного контейнера обычно оказывается более эффективной. Иногда имеет смысл использовать и то, и другое. Например, такой подход можно использовать для обмена элементов двух контейнеров разных типов.

Методы **insert()** и **erase()**

Эти методы, соответственно, вставляют и удаляют данные при обращении к произвольной позиции контейнера. Их не рекомендуется использовать с векторами, поскольку в этом случае при каждой вставке или удалении приходится перекраивать всю структуру — разжимать, ужимать вектор. Все это не слишком эффективно. Тем не менее, когда скорость не играет очень большой роли, использовать эти методы и можно, и нужно. Следующий пример показывает, как это делается.

Листинг 13. Программа VECTINS

```
//-----  
  
// vectins.cpp  
// демонстрация методов insert(), erase()  
  
#include "stdafx.h"  
#include <iostream>  
#include <vector>  
#include "windows.h"  
  
using namespace std;  
  
template <class T>  
void Print(const T& myarr)  
{  
    for (const auto & x : myarr)  
        cout << x << ' ';  
    cout << endl;  
}  
  
int main()  
{  
    SetConsoleCP(1251);  
    SetConsoleOutputCP(1251);  
  
    vector<int> v { 100, 110, 120, 130 };  
    cout << "\nПеред вставкой: ";  
    // Вывести все элементы.  
    Print(v);
```

```

    //100, 110, 120, 130
    //Вставить 115 в позицию 2.
    v.insert(v.begin() + 2, 115);
    cout << "\nПосле вставки: ";
    // Вывести все элементы.
    Print(v);
    //100, 110, 115, 120, 130
    // Удалить элемент со 2 позиции.
    v.erase(v.begin() + 2);
    cout << "\nПосле удаления: ";
    // Вывести все элементы.
    Print(v);
    //100, 110, 120, 130
    system("PAUSE");
    return 0;
}
//-----

```

Результат работы программы:

```

Перед вставкой: 100 110 120 130
После вставки: 100 110 115 120 130
После удаления: 100 110 120 130
Для продолжения нажмите любую клавишу . . .

```

Метод **insert()** (по крайней мере, данная версия) имеет два параметра: будущее расположение нового элемента в контейнере и значение элемента. Прибавляем две позиции к результату выполнения метода **begin()**, чтобы перейти к элементу № 2 (третий элемент в контейнере, считая с нуля). Элементы от точки вставки до конца контейнера сдвигаются, чтобы было место для размещения вставляемого. Размер контейнера автоматически увеличивается на единицу.

Метод **erase()** удаляет элемент из указанной позиции. Оставшиеся элементы сдвигаются, размер контейнера уменьшается на единицу. Вот как выглядят результаты работы программы:

```

Перед вставкой: 100 110 120 130
После вставки: 100 110 115 120 130
После удаления: 100 110 120 130
Для продолжения нажмите любую клавишу . . .

```

Ниже приведён пример работы с вектором объектов класса. Класс **A** — класс объектов простая дробь.

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm>
//Определяет шаблонны функций контейнера,
//которые выполняют алгоритмы числовой обработки.
#include <numeric>
#include <functional>
#include "windows.h"

using namespace std;
class A
{
    int n, d;

```

```

public:
    A(int n = 0, int d = 1) : n(n), d(d){};
    A operator+(const A b)const
    {
        return A((n*b.d + b.n*d), d*b.d);
    };
    void Set_d(int d_) { d = d_; }
    int Get_d() const { return d; }
    string get() const
    {
        string a;
        ostringstream os;
        os << n << "/" << d;
        return os.str();
    };
    bool operator>(const A b)const
    {
        return n*b.d > d*b.n;
    };
    bool operator<(const A b)const
    {
        return n*b.d < d*b.n;
    };
};

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    //Вектор простых дробей
    vector<A> m;
    //Итератор для вектора
    vector<A>::iterator iter;
    //Проталкиваем дроби в вектор
    m.push_back(A(2));
    m.push_back(A(3));
    m.push_back(A(1));
    m.push_back(A(5));
    m.push_back(A(9));
    m.push_back(A(7));
    //Сортируем вектор по возрастанию
    sort(m.begin(), m.end(), less<A>());
    for (auto y : m) { // Копия 'x', почти всегда нежелательная.
        cout << y.get() << " ";
    }
    cout << "После сортировки по возрастанию." << endl;
    for (auto &y : m) { // Вывод типа по ссылке.
        // Наблюдает и/или изменяет на месте. Предпочтительно, когда требуется
изменить.
        y.Set_d(2);
    }

    for (const auto &y : m) { // Вывод типа по ссылке.
        // Наблюдает на месте. Предпочтительно, если изменение не требуется.
        cout << y.get() << " ";
    }
    cout << "После изменения знаменателя." << endl;

    for (iter = m.begin(); iter != m.end(); iter++)
        cout << iter->get() << " ";
    cout << "Вывод с помощью итератора." << endl;
    //Сортируем вектор по убыванию
    sort(m.begin(), m.end(), greater<A>());
    for (int j = 0; j != m.size(); j++)
        cout << m[j].get() << " ";
}

```

```

    cout << "После сортировки по убыванию." << endl;
    //Находим сумму
    A sum = accumulate(m.begin(), m.end(), A(), plus<A>());
    cout << sum.get() << " - Значение суммы " << endl;
    system("PAUSE");
    return 0;
}

```

Результат работы программы:

```

1/1 2/1 3/1 5/1 7/1 9/1 После сортировки по возрастанию.
1/2 2/2 3/2 5/2 7/2 9/2 После изменения знаменателя.
1/2 2/2 3/2 5/2 7/2 9/2 Вывод с помощью итератора.
9/2 7/2 5/2 3/2 2/2 1/2 После сортировки по убыванию.
864/64 - Значение суммы
Для продолжения нажмите любую клавишу . . .

```

Списки

Контейнер STL под названием список представляет собой дважды связный список, в котором каждый элемент хранит указатель на соседа слева и справа. Контейнер содержит адрес первого и последнего элементов, поэтому доступ к обоим концам списка осуществляется очень быстро.

Методы `push_front()`, `front()` и `pop_front`

Наш первый пример работы со списками демонстрирует вставку, чтение и извлечение данных.

Листинг 14. Программа LIST

```

// Листинг 14.cpp: определяет точку входа для консольного приложения.
//
// list.cpp
// демонстрация методов push_front(), front(), pop_front()

#include "stdafx.h"
#include <iostream>
#include <list>
#include <string>
#include "windows.h"
using namespace std;

template<class T>
void print(const string h, const T& s)
{
    cout << h << endl;
    for (const auto &x : s)
        cout << x << " ";
    cout << endl;
}

void Russia()
{
    SetConsoleCP(1251);
}

```

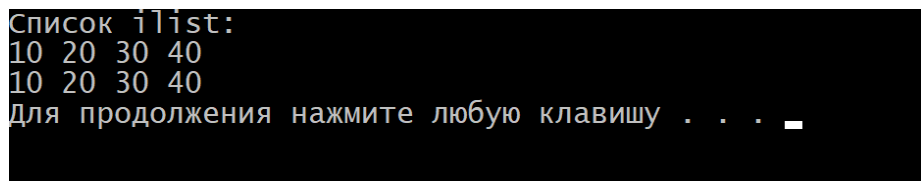
```

        SetConsoleOutputCP(1251);
    }

    int _tmain(int argc, _TCHAR* argv[])
    {
        Russia();
        list<int> ilist;
        // вставка элементов в конец
        ilist.push_back(30);
        ilist.push_back(40);
        // вставка элементов в начало
        ilist.push_front(20);
        ilist.push_front(10);
        // число элементов
        int size = ilist.size();
        //Печать содержимого списка.
        print("Список ilist: ", ilist);
        //Вывод значения головы и удаление его из списка.
        for(int j = 0; j < size; j++)
        {
            // Читать данные из начала.
            cout << ilist.front() << ' ';
            // Извлечение данных из начала.
            ilist.pop_front();
        }
        cout << endl;
        system("PAUSE");
        return 0;
    }

```

Поясним работу программы. Мы вставляем данные в конец и начало списка таким образом, чтобы при выводе на экран и удалении из начала сохранялся (следующий порядок их следования:



```

Список ilist:
10 20 30 40
10 20 30 40
Для продолжения нажмите любую клавишу . . . _

```

Методы `push_front()`, `front()` и `pop_front()` аналогичны методам `pop_back()`, `push_back()` и `back()`, которые мы уже видели при работе с векторами.

Помните, что использовать произвольный доступ к элементам списка нежелательно, так как он осуществляется слишком неторопливо для нормальной обработки данных. Поэтому оператор `[]` даже не определен для списков. Если бы произвольный доступ был реализован, этому оператору пришлось бы проходить весь список, перемещаясь от ссылки к ссылке, вплоть до достижения нужного элемента. Это была бы очень медленная операция. Если вы считаете, что программе может потребоваться произвольный доступ к контейнеру, используйте векторы или очереди с двусторонним доступом.

Списки целесообразно использовать при частых операциях вставки и удаления где-либо в середине списка. При этих действиях использовать векторы и очереди нелогично,

потому что все элементы над точкой вставки или удаления должны при этом быть сдвинуты. Что касается списков, то при тех же операциях изменяются лишь значения нескольких указателей. (Не бывает правил без исключений, и могут возникнуть ситуации, когда поиск нужной позиции вставки в списке будет тоже довольно длительной операцией.)

Для работы методов `insert()` и `erase()` требуются итераторы, поэтому мы отложим их подробное рассмотрение на будущее.

Методы `reverse()`, `merge()` и `unique()`

Некоторые методы используются только со списками. Других контейнеров, для которых они были бы определены, просто нет, хотя есть, конечно, алгоритмы выполняющие практически те же функции. В следующем примере показаны некоторые из таких методов. Программа начинается с заполнения двух целочисленных списков содержимым двух массивов.

Листинг 15. Программа LISTPLUS

```
// Листинг 15.cpp: определяет точку входа для консольного приложения.
//
// Демонстрация методов reverse(), merge() и unique()

#include "stdafx.h"
#include <iostream>
#include <list>
#include "windows.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    int j;
    //создаём два списка значений int
    list<int> list1, list2;

    int arr1[] = { 40, 30, 20, 10 };
    int arr2[] = { 15, 20, 25, 30, 35 };
    // list1: 40, 30, 20, 10
    for(j = 0; j < 4; j++)
        list1.push_back( arr1[j] );
    // list2: 15, 20, 25, 30, 35
    for(j = 4; j >= 0; j--)
        list2.push_front( arr2[j] );
    // перевернуть list1: 10 20 30 40
    list1.reverse();
    // объединить list2 с list1
    list1.merge(list2);
    // удалить повторяющиеся элементы 20 и 30
    list1.unique();
    int size = list1.size();
    while( !list1.empty() )
    {
        // читать элемент из начала
```

```

        cout << list1.front() << ' ';
        // вытолкнуть элемент из начала
        list1.pop_front();
    }
    cout << endl;
    //list2 - пуст
    cout << list2.size() << endl;
    system("PAUSE");
    return 0;
}

```

Первый список составлен из элементов, расположенных в обратном порядке, поэтому для начала мы его переворачиваем так, чтобы он был отсортирован по возрастанию. После этого действия списки выглядят так:

```

10 20 30 40
15 20 25 30 35

```

Затем выполняется функция `merge()`. С ее помощью объединяются списки `list2` и `list1`, результат сохраняется в `list1`. Его новое содержимое:

```

10 15 20 20 25 30 30 35 40

```

Наконец, мы применяем метод `unique()` к списку `list1`. Эта функция находит соседние элементы с одинаковыми значениями и оставляет только один из них. Содержимое списка `list1` выводится на экран:

```

10 15 20 25 30 35 40

```

Для вывода списка используются функции `front()` и `pop_front()`. Из них составляется цикл `for`, таким образом проходится весь список. Каждый элемент по очереди от головы до хвоста выводится на экран, затем выталкивается из списка. Впереди планеты всей после каждого шага оказывается новый элемент. В результате получается, что операция вывода на экран уничтожает список. Может быть, это не всегда то, что нужно, но, тем не менее, это единственный способ продемонстрировать возможности последовательного доступа к списку. Итераторы в этом смысле упрощают дело, но к их рассмотрению мы перейдём только в следующем параграфе.

В примере ниже с помощью списка обрабатываются объекты пользовательского типа.

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <sstream>
#include <list>
#include <algorithm>
#include <numeric> // Определяет шаблоны функций контейнера, которые выполняют алгоритмы
числовой обработки.
#include <functional>
#include "windows.h"

using namespace std;
class A
{
    int n, d;
public:
    A(int n = 0, int d = 1) : n(n), d(d){};
    A operator+(const A b) const
    {
        return A((n*b.d + b.n*d), d*b.d);
    }
}

```

```

};
string get() const
{
    string a;
    ostringstream os;
    os << n << "/" << d;
    return os.str();
};
bool operator>(const A b) const
{
    return n*b.d > d*b.n;
};
bool operator<(const A b) const
{
    return n*b.d < d*b.n;
};

};

template<class T> void print(const string h, const T& s)
{
    cout << h << endl;
    for (const auto &x : s)
        cout << x.get() << " ";
    cout << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    //Список простых дробей
    list<A> lst;
    //Итератор для списка
    list<A> :: iterator iter;
    //Проталкиваем дроби в вектор
    lst.push_back(A(2));
    lst.push_back(A(3));
    lst.push_back(A(1));
    lst.push_back(A(5));
    lst.push_back(A(9));
    lst.push_back(A(7));
    print("Список до сортировки.", lst);
    //Сортируем список по возрастанию
    lst.sort(less<A>());
    print("После сортировки по возрастанию.", lst);
    //Сортируем список по убыванию
    lst.sort(greater<A>());
    print("После сортировки по убыванию.", lst);
    //Находим сумму
    A sum = accumulate(lst.begin(), lst.end(), A(), plus<A>());
    cout << "Сумма элементов списка = " << sum.get() << endl;
    list<A> list{ { 10, 1 }, { 11, 1 }, { 12, 1 } };
    lst.splice(++lst.begin(), list);
    print("Список после слияния", lst);
    cout << endl;
    system("PAUSE");
    return 0;
}

```

Результат работы программы:

```

Список до сортировки.
2/1 3/1 1/1 5/1 9/1 7/1
После сортировки по возрастанию.
1/1 2/1 3/1 5/1 7/1 9/1
После сортировки по убыванию.
9/1 7/1 5/1 3/1 2/1 1/1
Сумма элементов списка = 27/1
Список после слияния
9/1 10/1 11/1 12/1 7/1 5/1 3/1 2/1 1/1

Для продолжения нажмите любую клавишу . . .

```

Очереди с двусторонним доступом

Очередь с двусторонним доступом (**deque**) представляет собой нечто похожее и на вектор, и на связный список. Как и вектор, этот тип контейнера поддерживает произвольный доступ (оператор []). Как и к списку, доступ может быть получен к началу и концу очереди. В целом это напоминает вектор с двумя концами. Поддерживаются функции `front()`, `push_front()` и `pop_front()`.

Для векторов и двусторонних очередей память резервируется по-разному. Вектор всегда занимает смежные ячейки памяти. Поэтому при его разрастании резервируется новый участок памяти, где контейнер может поместиться целиком. С другой стороны, очередь с двусторонним доступом может размещаться в нескольких сегментах памяти, не обязательно смежных. В этом есть и плюсы и минусы. Очередь практически всегда будет гарантированно размещена в памяти, но доступ к сегментированным объектам всегда осуществляется с более медленной скоростью. Метод `capacity()` возвращает наибольшее число элементов вектора, которые можно разместить в памяти без каких-либо махинаций, не определен для контейнера `deque` (очередь с двусторонним доступом), поскольку в данном случае никаких перемещений по памяти не требуется.

Листинг 16. Программа DEQUE

```

//-----
// Демонстрация методов push_back(), push_front(), front()
#include "stdafx.h"
#include <iostream>
#include <deque>
#include "windows.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    // Дека целых чисел типа int.
    deque<int> deq {70,80,90};
    // Проталкивание элементов в конец.
    deq.push_back(30);
    deq.push_back(40);
    deq.push_back(50);
    // Проталкивание элементов в начало.
    deq.push_front(20);
    deq.push_front(10);
    // Вывести элементы.
    for (int j = 0; j < deq.size(); j++)
        cout << deq[j] << ' ';
    cout << endl;
    // Изменение произвольного элемента контейнера.
    deq[2] = 33;
    // Вывести элементы.
    for (int j = 0; j < deq.size(); j++)

```

```

        cout << deq.at(j) << ' ';
    cout << endl;
    // Вывести элементы.
    for (const int &y : deq)
        cout << y << ' ';
    cout << endl;
    system("PAUSE");
    return 0;
}

```

//-----

Нам уже встречались примеры использования методов `push_back()`, `push_front()` и оператора `[]`. К очередям с двусторонним доступом они применяются точно так же, как и к другим контейнерам. Результат работы программы:

```

10 20 70 80 90 30 40 50
10 20 33 80 90 30 40 50
10 20 33 80 90 30 40 50
Для продолжения нажмите любую клавишу . . .

```