

КОНТЕЙНЕРЫ

Контейнеры — это классы, предназначенные для хранения однотипных данных, организованных определенным образом.

Контейнер управляет выделяемой памятью и предоставляет доступ к элементам с помощью функций, через итераторы или непосредственно.

Один и тот же вид контейнера можно использовать для хранения данных *различных типов*.

Эта возможность реализована с помощью шаблонов классов.

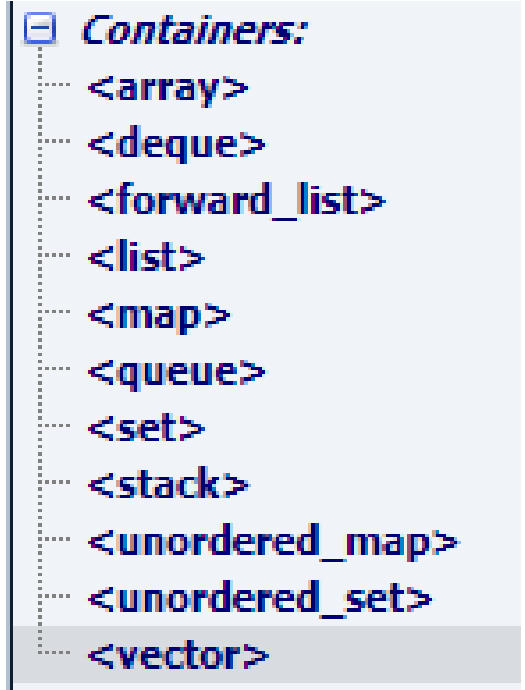
Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере.

Примерами контейнеров могут служить массивы, линейные списки или стеки.

STL содержит контейнеры, реализующие основные структуры данных — векторы, двусторонние очереди, списки и их разновидности, словари и множества.

Библиотеки C++, в которую входят контейнерные классы называют стандартной библиотекой шаблонов (**STL** — Standard Template Library).

Для использования определенного контейнера в программу необходимо добавить соответствующий заголовочный файл, который, как правило, называется по имени класса контейнера.



Контейнеры можно разделить на три категории:

последовательные контейнеры,
ассоциативные контейнеры,
контейнеры-адаптеры.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных величин в виде непрерывной последовательности.

Последовательные контейнеры отличаются способом доступа к элементам.

- векторы (vector),
- двусторонние очереди (deque)
- списки (list)
- стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue).

Последовательные контейнеры:

- **vector** — массив переменного размера,
- **deque** — двусторонние очереди,
- **list** — двухсвязный список,
- **forward_list** — односвязный список.
- **array** — массив фиксированного размера.
- **string**.

Ассоциативные контейнеры реализуют упорядоченные структуры данных с возможностью быстрого поиска данных по ключу.

Ассоциативные контейнеры построены на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент.

Ключ ассоциирован с элементом.

Ассоциативные контейнеры

- **map** — словари,
- **multimap** — словари с дубликатами,
- **set** — множества,
- **multiset** — множества с дубликатами,
- **bitset** — битовые множества.

Контейнерные классы обеспечивают стандартизованный интерфейс при их использовании.

Смысл одноименных операций для различных контейнеров одинаков, основные операции применимы ко всем типам контейнеров.

Стандарт определяет только интерфейс контейнеров.

Итераторы обеспечивают доступ к элементам контейнера, являются аналогом указателя на элемент.

Он используется для просмотра контейнера в прямом или обратном направлении.

При помощи **итераторов** можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам.

Функции итераторов

begin() возвращает итератор, который указывает на первый элемент контейнера.

end() возвращает итератор, который указывает на следующую позицию после последнего элемента.

Если контейнер пуст, то итераторы, возвращаемые методами *begin* и *end* совпадают.

rbegin() и **rend()** реверсивные итераторы.

Реверсивные итераторы позволяют перебирать элементы контейнера в обратном направлении.

Операции с итераторами

***iter:** получение элемента, на который указывает итератор;

++iter: перемещение итератора вперед для обращения к следующему элементу;

--iter: перемещение итератора назад для обращения к предыдущему элементу;

iter1 == iter2: два итератора равны, если они указывают на один и тот же элемент;

iter1 != iter2: два итератора не равны, если они указывают на разные элементы.

Вектор

```
#include <vector>
```

Вектор — класс контейнерного типа,
имитирует **динамический массив**.

Размера вектора автоматически
увеличиваться, при добавлении новых
элементов.

Доступа к отдельным элементам вектора
как в массиве.

Объявление вектора

1 способ :

создаётся пустой вектор

vector <тип> *имя вектора;*

vector <int> myVector;

vector <string> text;

2 способ:

Объявляется вектор с начальным размером.

Элементы вектора инициализируются значениями по умолчанию (0).

vector <тип> *имя вектора* (кол-во);

vector<int> myVector(10);



3 способ:

Все элементы объявленного вектора инициализируются одним заданным значением.

vector <тип> *имя вектора (кол-во, значен.);*

vector <int> myVector(10, 2);

vector<string> names (20, "text");



4 способ:

Все элементы вектора инициализируются заданными значениями.

vector <тип> *имя вектора* {значен1, зн2, ..};

vector <int> myVector {1,0, 2,9};

vector <int> vec1= {10, 20, 22, 69};

5 способ:

Вектор можно инициализировать другим объектом типа **vector**.

```
vector <int> v1;
```

Вектор можно создать как копию другого вектора использования операции присваивания.

```
vector <int> v2 (v1);
```

```
vector <int> v3= v1;
```

6 способ:

Вектор можно инициализировать с помощью массива. Вектору передаются два указателя – указатель на начало массива и на элемент, следующий за последним

```
int m[10]= { -2, -1, 0, 1, 2, 1, 0, 2, 4 };
```

6 элементов массива копируются в вектор:

```
vector < int > v1 ( m, m+6 );
```

копируются 3 элемента: m[2], m[3], m[4]

```
vector< int > v1 ( &m[ 2 ], &m[ 5 ] );
```

// пустой вектор

```
std::vector<int> v1;
```

// вектор v2 - копия вектора v1

```
std::vector<int> v2(v1);
```

// вектор v3 - копия вектора v1

```
std::vector<int> v3 = v1;
```

```
// пустой вектор
std::vector<int> v1;
// вектор v2 - копия вектора v1
std::vector<int> v2(v1);
// вектор v3 - копия вектора v1
std::vector<int> v3 = v1;
// вектор v4 состоит из 5 чисел
std::vector<int> v4(5);
// вектор v5 состоит из 5 чисел, каждое число равно 2
std::vector<int> v5(5, 2);
// вектор v6 состоит из чисел 1, 2, 4, 5
std::vector<int> v6{1, 2, 4, 5};
// вектор v7 состоит из чисел 1, 2, 4, 5
std::vector<int> v7 = {1, 2, 3, 5};
```


Обращение к элементам и их перебор

- **[index]** — получение элемента по индексу
- **at (index)** — метод доступа, к элементам коллекции, как [].
- **front()** — предоставляет доступ к первому элементу
- **back()** — предоставляет доступ к последнему элементу

```
vector<int> numbers= {1, 2, 3, 4, 5};  
int n1 = numbers.front();    // n1 = 1  
int n2 = numbers.back();    // n2 = 5  
int n3 = numbers[2];        // n3 = 3  
int n4 = numbers.at(3);     // n4 = 4
```

Итераторы

Итераторы обеспечивают доступ к элементам контейнера.

begin() — возвращает итератор, указывающий на начало коллекции.

end() — возвращает итератор, указывающий на конец коллекции. При этом он указывает не самый последний элемент, а на воображаемый элемент за последним.

Если контейнер пуст, то итераторы `begin` и `end` совпадают.

Если итератор `begin` не равен итератору `end`, то между ними есть как минимум один элемент.

```
vector<int> vec6 = { 8,9,6,7,4,5 };  
auto iter = vec6.begin();  
while (iter != vec6.end())    // пока не дойдем до конца  
{  
    // получаем элементы через итератор  
    cout << *iter << endl;  
    // перемещаемся вперед на один элемент  
    ++iter;  
}
```

Функции

size() – размер вектора.

Размер – это реальное количество элементов, хранящихся в данный момент в контейнере.

capacity() – емкость вектора.

Емкость – это максимальное количество элементов, которое может вместить контейнер без дополнительного выделения памяти.

push_back() – вставляет элемент в конец вектора.

При вызове `push_back()`, вектор выделяет больше памяти, чем требуется. В большинстве реализаций память увеличивается в два раза.

pop_back() – удаляет последний элемент.

```
vector<int> numbers;  
numbers.push_back(5);  
numbers.push_back(3);  
numbers.push_back(10);  
for (int n : numbers)  
    cout << n << "\\t";
```

```
vector< int > ivec;
```

```
cout << "ivec: PA3MEP: " << ivec.size( )  
<< " EMKOCTb: " << ivec.capacity( ) << endl;
```

```
for ( int ix = 0; ix < 24; ++ix ) {
```

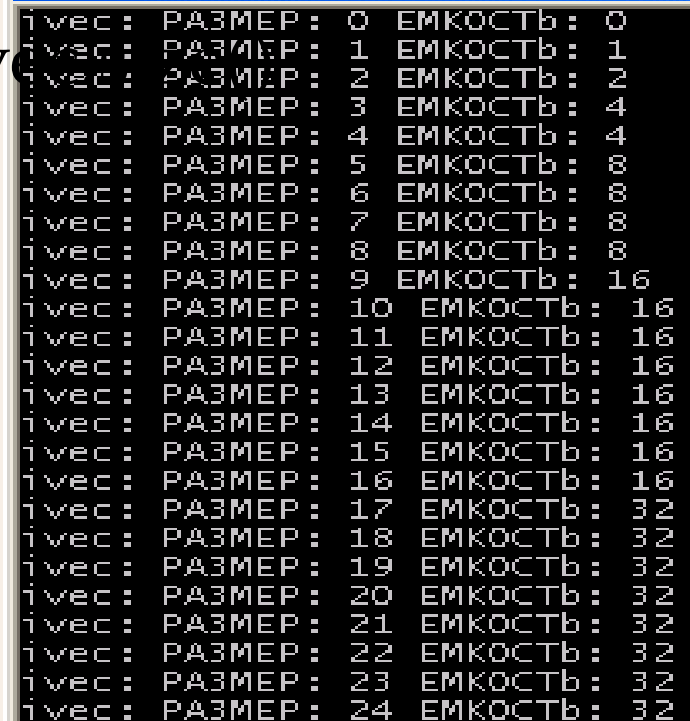
```
    ivec.push_back( ix );
```

```
    cout << "ivec: PA3MEP: " << ivec.size( )
```

```
    << " EMKOCTb: "
```

```
    << ivec.capacity( ) << endl;
```

```
}
```



ix	size	capacity
0	0	0
1	1	1
2	2	2
3	3	4
4	4	4
5	5	8
6	6	8
7	7	8
8	8	8
9	9	16
10	10	16
11	11	16
12	12	16
13	13	16
14	14	16
15	15	16
16	16	16
17	17	32
18	18	32
19	19	32
20	20	32
21	21	32
22	22	32
23	23	32
24	24	32

reserve (n)

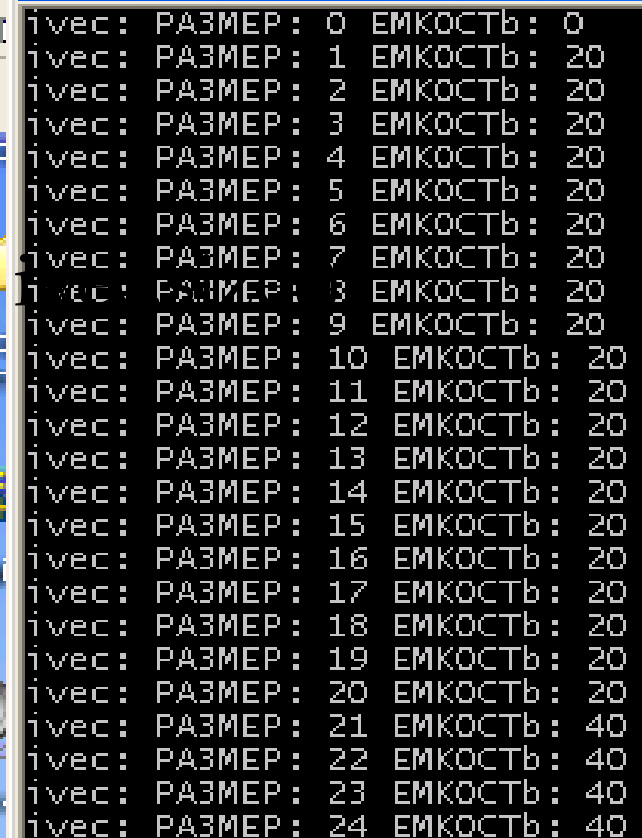
Выделяет дополнительную память в будущее пользование vector.

Вектор должен выделить столько памяти, чтобы вплоть до размера в **n** элементов дополнительных операций выделения памяти не потребовалось.


```
vector< int > ivec;  
cout << "ivec: PA3MEP: " << ivec.size()  
    << " EMKOCTb: " << ivec.capacity() << endl;
```

```
ivec.reserve ( 20 );
```

```
for ( int ix = 0; ix < 24; ++ix ) {  
    ivec.push_back( ix );  
    cout << "ivec: PA3MEP: " <<
```



```
ivec: PA3MEP: 0 EMKOCTb: 0  
ivec: PA3MEP: 1 EMKOCTb: 20  
ivec: PA3MEP: 2 EMKOCTb: 20  
ivec: PA3MEP: 3 EMKOCTb: 20  
ivec: PA3MEP: 4 EMKOCTb: 20  
ivec: PA3MEP: 5 EMKOCTb: 20  
ivec: PA3MEP: 6 EMKOCTb: 20  
ivec: PA3MEP: 7 EMKOCTb: 20  
ivec: PA3MEP: 8 EMKOCTb: 20  
ivec: PA3MEP: 9 EMKOCTb: 20  
ivec: PA3MEP: 10 EMKOCTb: 20  
ivec: PA3MEP: 11 EMKOCTb: 20  
ivec: PA3MEP: 12 EMKOCTb: 20  
ivec: PA3MEP: 13 EMKOCTb: 20  
ivec: PA3MEP: 14 EMKOCTb: 20  
ivec: PA3MEP: 15 EMKOCTb: 20  
ivec: PA3MEP: 16 EMKOCTb: 20  
ivec: PA3MEP: 17 EMKOCTb: 20  
ivec: PA3MEP: 18 EMKOCTb: 20  
ivec: PA3MEP: 19 EMKOCTb: 20  
ivec: PA3MEP: 20 EMKOCTb: 20  
ivec: PA3MEP: 21 EMKOCTb: 40  
ivec: PA3MEP: 22 EMKOCTb: 40  
ivec: PA3MEP: 23 EMKOCTb: 40  
ivec: PA3MEP: 24 EMKOCTb: 40
```

insert (pos, value) – добавление данных в vector.
вектор расширится на один элемент;

pos – элемент, перед которым будет вставлено значение.

value – значение элемента для вставки

```
char st[ ]="helo!";  
vector<char> vv (st, st+5);  
vv.insert (vv.begin( )+2,'L');
```



erase (p) —удаление элементов на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент

erase (begin, end) —удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end.

```
std::vector<int> numbers1 = { 1, 2, 3, 4, 5, 6 };  
auto iter = numbers1.begin();  
numbers1.erase(iter + 2);  
// numbers1 = { 1, 2, 4, 5, 6 }
```

clear () — удаляет все элементы вектора.

empty() — проверить вектор на пустоту.

```
vector<int> numbers = {1, 2, 3};  
if(numbers.empty())  
    cout << "Vector is empty" << endl;  
else  
    cout << "Vector has size " << numbers.size();
```

```
#include <vector.h>
#include <iostream.h>
#include <iterator.h>

void f1() {
    vector<int>::iterator it;
    vector< int > ivec;
    for ( int ix = 0; ix < 24; ++ix )
        ivec.push_back( ix );

    for (it = ivec.begin(); it != ivec.end(); ++it )

        cout << *it << ' ';

    copy (ivec.begin(), ivec.end(),
        ostream_iterator<int>(cout, " "));
```

```
include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "Header.h"
using namespace std;

int main(){
vector <man> student;
int a;
while (true) {
    cout << "Возможные действия:\n" <<
    "1. ввод данных.\n" <<
    "2. чтение данных из файла.\n" <<
    "3. запись данных в файл.\n" <<
    "4. вывод данных на экран.\n" <<
    "5. конец работы.\n" <<
    "введите номер выбранного пункта :\n";
    cin >> a;
    switch (a){
        case 1: vvod(student);break;
        case 2: read_f(student);break;
        case 3: write_f(student);break;
        case 4: consw(student);break;
        case 5: return 0;
    }
}
```

```
#pragma once  
struct man {  
    std::string fio;  
    int god;  
};
```

```
man* vvod(std::vector <man>& stud);  
void add_zap();  
void read_f(std::vector <man>& stud);  
void write_f(std::vector <man>& stud);  
void consw(std::vector <man>&);
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include "Header.h"
using namespace std;
```

```
man* vvod(vector <man>& stud) {
    man* st = new man;;
    cout << "введите имя:";
    cin >> st->fio;
    cout << "введите год:";
    cin >> st->god;
    stud.push_back(*st);
    return st;
}
```



```
void read_f(vector <man>& stud) {  
    man st;  
    stud.clear();  
    ifstream ff("spisok.txt");  
    if (!ff.is_open()) {  
        cout << "error\n";  
        return;  
    }  
    while (1) {  
        ff >> st.fio >> st.god;  
        if (ff.eof()) break;  
        stud.push_back(st);  
    }  
    cout << "\nsize=" << stud.size() << endl;  
}
```

```
void write_f(vector <man>& stud) {  
    ofstream ff("spisok.txt");  
    for (man var : stud) {  
        ff << var.fio<<" "<<var.god<<"\n";  
    }  
}
```

```
void consw(vector <man> &stud) {  
    for (man var : stud)  
    {  
        cout << var.fio << " " << var.god <<  
endl;  
    }  
}
```