

Лекция 0. Контейнеры и итераторы

Автор: Набиев Марат

Компьютеры созданы для того, чтобы эффективно обрабатывать данные. А если этих данных много, то удобней всего хранить их в каких-то контейнерах (в каких-то языках это называются коллекции). Эти контейнеры должны обеспечить эффективный доступ к данным: возможность добавлять, удалять, изменять. У каждого вида контейнера есть свои плюсы и минусы. В C++ эти контейнеры описаны в библиотеке STL.

1. `std::array`

для подключения надо написать `#include <array>`

Array имеет фиксированный размер, по способу работы очень напоминает обычные массивы, но array более безопасный, т. к. можно отловить ошибку выхода за границы массива, а с обычными массивами вы даже не заметите этого.

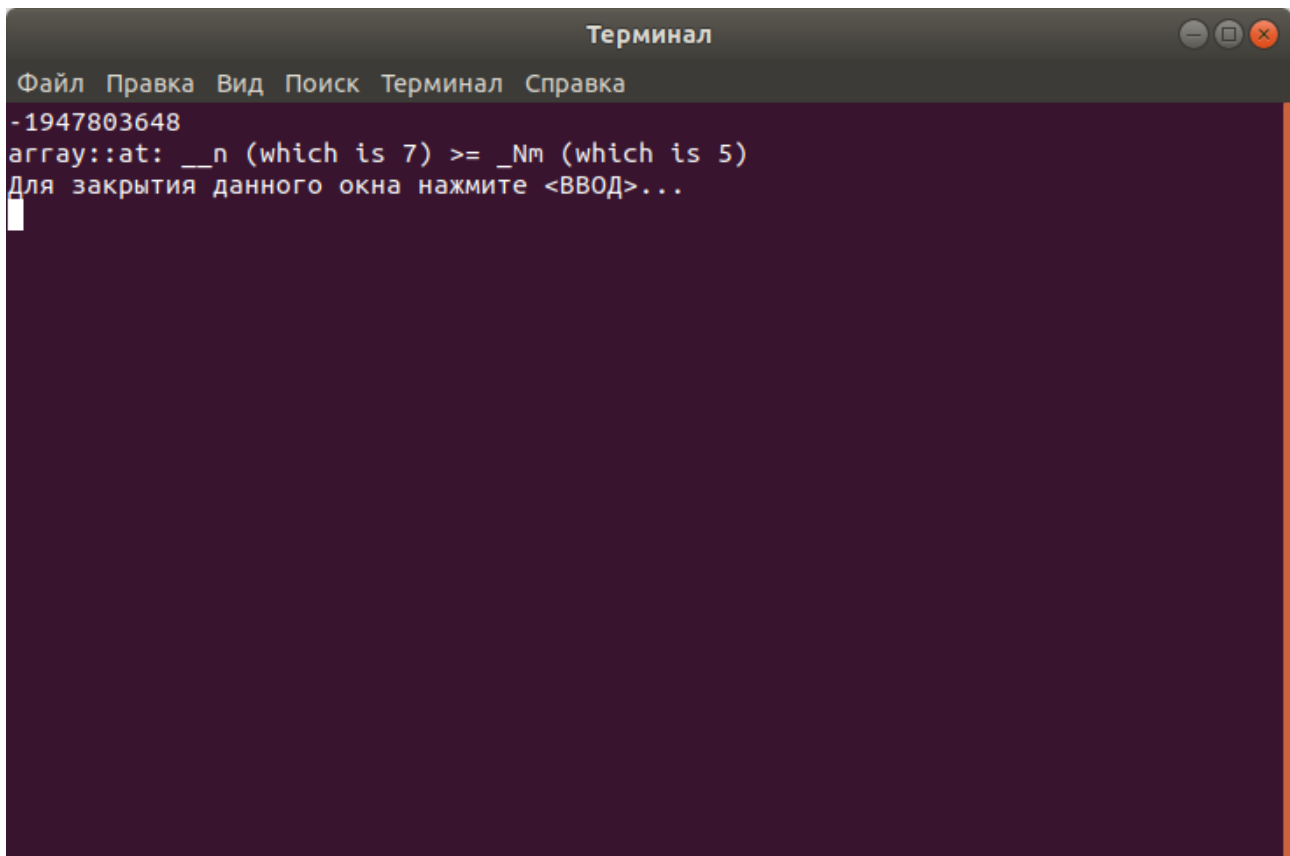
```
#include <iostream>
#include<array>

int main()
{
    //создаем массив из 5 элементов
    std::array<int, 5> a = {9,1,3,6,0};
    //создали аналогичный обычный массив
    int b[5] = {1,4,56,7,1};
    //выводим то, что за границей
    std::cout<<b[6]<<std::endl;

    //попробуем отловить ошибку
    try{
        std::cout<<a.at(7)<<std::endl;
    }
    catch(std::out_of_range e)
    {
        //выводим сообщение об ошибке
        std::cout<<e.what()<<std::endl;
    }

    return 0;
}
```

Что вывелось



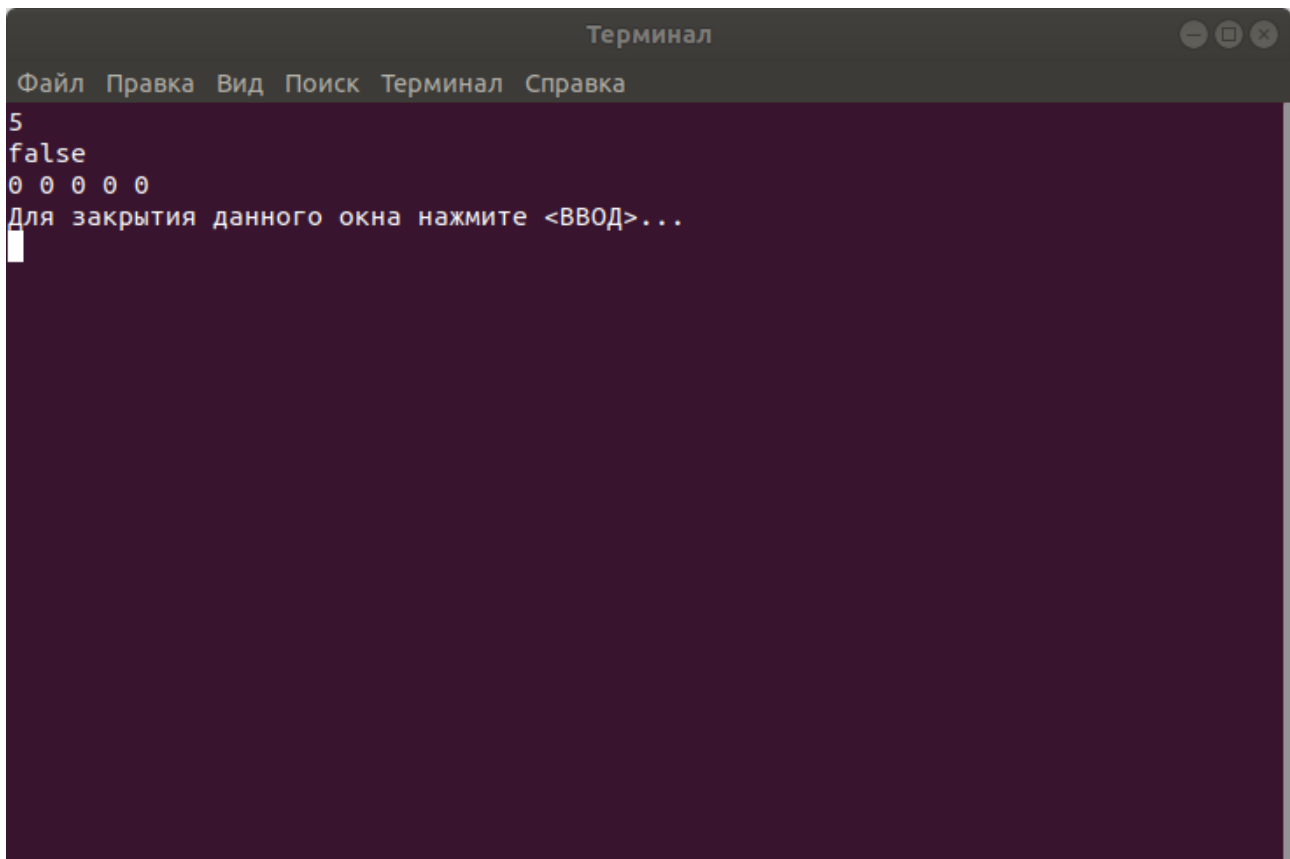
```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
-1947803648
array::at: __n (which is 7) >= _Nm (which is 5)
Для закрытия данного окна нажмите <ВВОД>...
```

Также есть следующие методы `size` — для получения размера, `empty` — узнать не пустой ли массив и также `fill`, который заполняет весь массив какими-то значениями

```
int main()
{
    //создаем массив из 5 элементов
    std::array<int, 5> a = {9,1,3,6,0};
    std::cout<<a.size()<<std::endl;
    //boolalpha для вывода булевой переменной словами
    std::cout<<std::boolalpha<<a.empty()<<std::endl;

    a.fill(0);
    for(int i: a)
    {
        std::cout<<i<<' ';
    }
    std::cout<<std::endl;

    return 0;
}
```



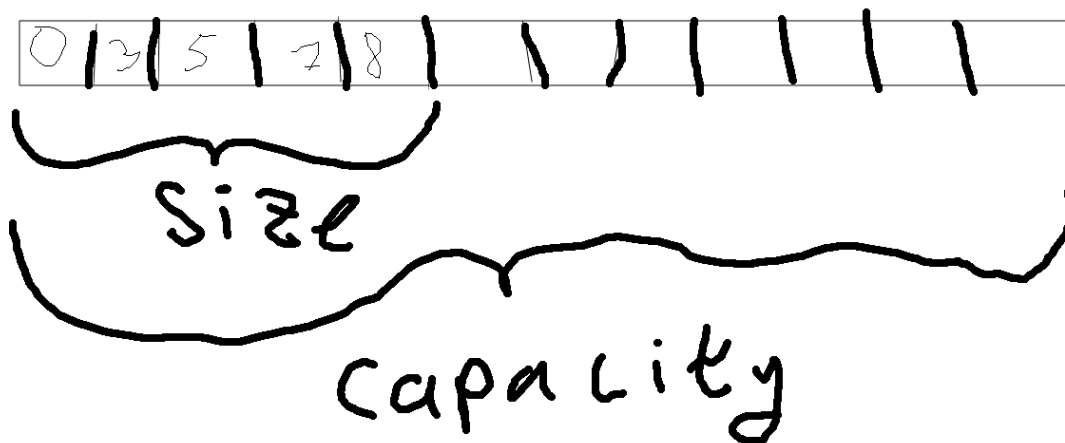
```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
5
false
0 0 0 0 0
Для закрытия данного окна нажмите <ВВОД>...
```

Этот контейнер довольно неинтересен с той стороны, что надо заранее предполагать какой размер он имеет, поэтому его используют довольно редко.

2. `std::vector`

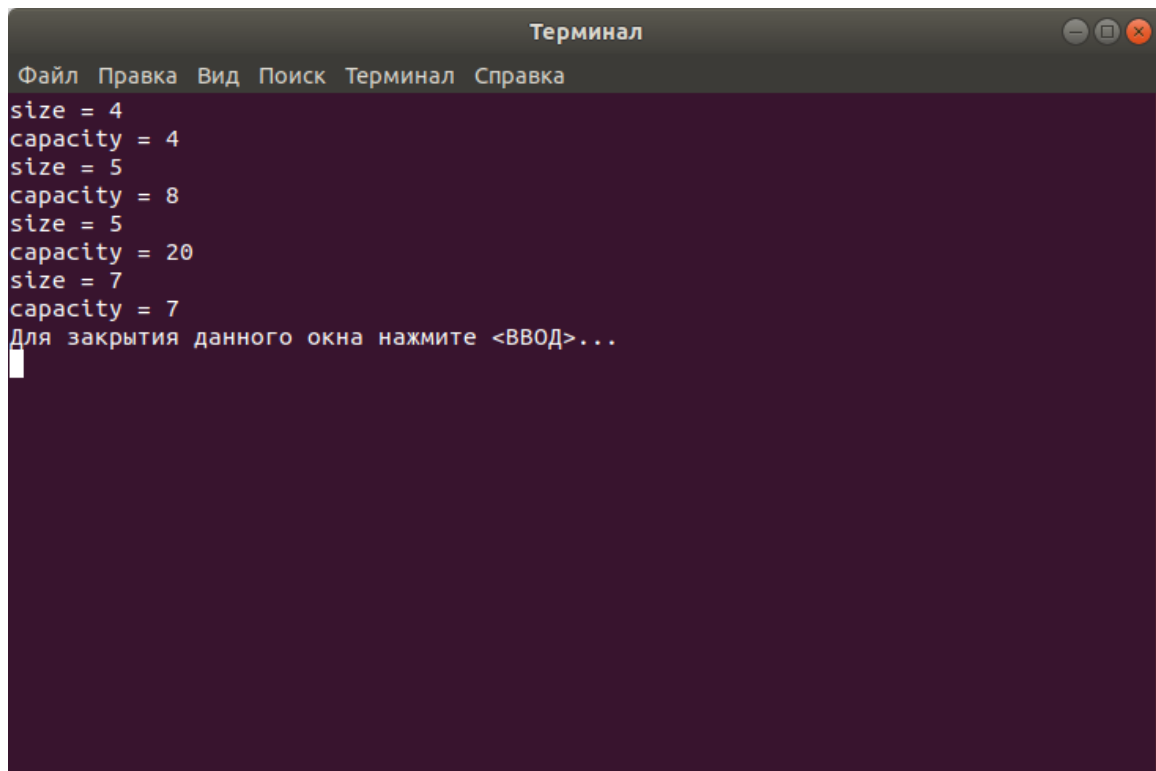
Подключается при помощи `#include <vector>`

Достаточно популярный контейнер, который используется чаще всего. Здесь размер массива может меняться, при чем есть зарезервированная часть и есть реальный размер. `Size` — размер массива с данными, а `capacity` - то, что у нас есть. Если `size < capacity`, то добавление в конец выполняется за константное время. Если равны, то размер зарезервированной части увеличивается (в 1.5-2 раза), старые данные копируются в новый массив и в конец добавляется добавляемый элемент. При помощи метода `reserve` — можно заранее выделить память, но не использовать ее. Это надо в том случае, если вы знаете, примерно сколько памяти вам надо, и для того, чтобы ускорить программу, заранее выделяете память. А при помощи метода `shrink_to_fit` можно очистить неиспользуемую память. Есть методы `push_back` — для добавления в конец и `pop_back`, чтобы удалить последний элемент



```
//создаем массив из 5 элементов
std::vector<int> a = {1,3,4,5};
std::cout<<"size = "<<a.size()<<std::endl;
std::cout<<"capacity = "<<a.capacity()<<std::endl;
//добавляем в конец
a.push_back(12);
//опять вывели размеры
std::cout<<"size = "<<a.size()<<std::endl;
std::cout<<"capacity = "<<a.capacity()<<std::endl;

//резервируем память
a.reserve(20);
std::cout<<"size = "<<a.size()<<std::endl;
std::cout<<"capacity = "<<a.capacity()<<std::endl;
//добавляем в конец пару элементов
a.push_back(1);
a.push_back(3);
//убираем неиспользуемую память
a.shrink_to_fit();
std::cout<<"size = "<<a.size()<<std::endl;
std::cout<<"capacity = "<<a.capacity()<<std::endl;
return 0;
```



```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
size = 4
capacity = 4
size = 5
capacity = 8
size = 5
capacity = 20
size = 7
capacity = 7
Для закрытия данного окна нажмите <ВВОД>...
```

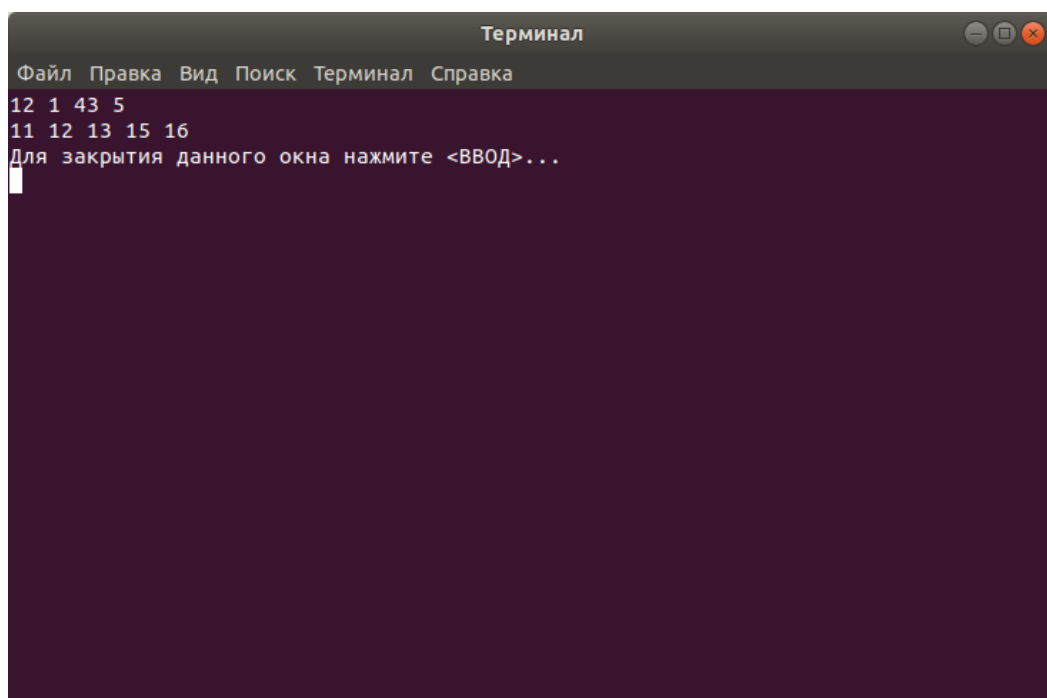
Также есть операторы сравнения `==`, `<`, `>` и д.р. сравниваются в лексикографическом порядке. Вроде все. Если что добавлю.

3. `std::forward_list` и `std::list`

Для подключения `#include<forward_list>` и `#include<list>` соответственно.

Это односвязный и двусвязный списки. В отличие в методах только то, что в односвязный список можно добавлять и удалять только из начала списка, а в двусвязный список можно добавлять и удалять и из начала, и из конца. Но в списках нельзя обращаться к элементам при помощи оператора `[i]`. Т.к. это неэффективно и разработчики не добавили эту возможность :D

```
//односвязный список
std::forward_list<int> a = {1,43,5};
//добавили в начало
a.push_front(12);
//распечатали
for(int i: a)
{
    std::cout<<i<<" ";
}
std::cout<<std::endl;
//двусвязный список
std::list<int> b = {12,13,15};
//добавили в начало и конец
b.push_back(16);
b.push_front(11);
for(int i: b)
{
    std::cout<<i<<" ";
}
std::cout<<std::endl;
```



Терминал

Файл Правка Вид Поиск Терминал Справка

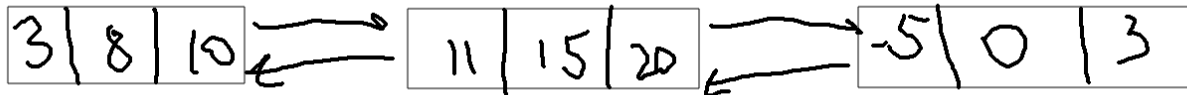
```
12 1 43 5
11 12 13 15 16
Для закрытия данного окна нажмите <ВВОД>...
```

4. `std::deque`

Для подключения `#include <deque>`

Очень классный контейнер, довольно эффективный. Собрал все хорошие стороны вектора и списка

Организован примерно следующим образом:



Можно добавлять в начало и конец, обращаться к элементам по индексам и все такое. Круто короче.

```
int main()
{
    std::deque<int> a = {1,3,4};
    //в начало
    a.push_front(4);
    //в конец
    a.push_back(12);
    a[3] = 12;
    for(int i: a)
    {
        std::cout<<i<<" ";
    }
    std::cout<<std::endl;
    //размер
    std::cout<<"size = "<<a.size()<<std::endl;

    return 0;
}
```

```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
4 1 3 12 12
size = 5
Для закрытия данного окна нажмите <ВВОД>...
```

5. `std::stack`

для подключения `#include<stack>`

Обычный стек. Организован по принципу: последний вошел , первый вышел (LIFO — last in, first out). Для добавления используется метод `push`, для удаления `pop`, для того, чтобы посмотреть верхний элемент метод `top`.

```
std::stack<int> s;
s.push(3);
s.push(1);
std::cout<<s.top()<<std::endl;
s.pop();
std::cout<<s.top()<<std::endl;
return 0;
```



```
Терминал
Файл Правка Вид Поиск Терминал Справка
1
3
Для закрытия данного окна нажмите <ВВОД>...
```

6. `std::queue`

Для подключения `#include<queue>`

Очередь, добавление и удаление элементов организовано по принципу FIFO (first in first out — первый вошел, первый вышел). Для добавления метод `push`, для удаления `pop`, для получения первого `front`, для получения последнего `back`

```
std::queue<int> s;
s.push(3);
s.push(1);
std::cout<<s.front()<<std::endl;
std::cout<<s.back()<<std::endl;
s.pop();
std::cout<<s.front()<<std::endl;
return 0;
```

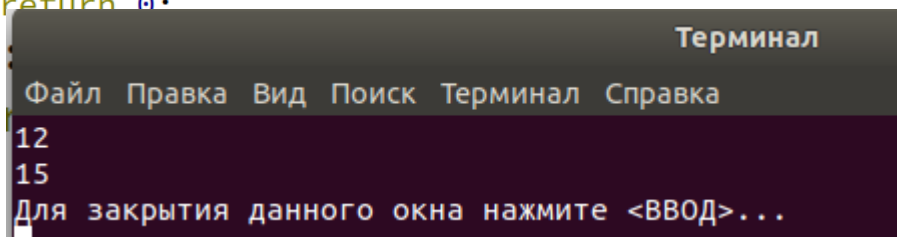
```
Терминал
Файл Правка Вид Поиск Терминал Справка
3
1
1
Для закрытия данного окна нажмите <ВВОД>...
```

7. `std::priority_queue`

Для подключения `#include<queue>`

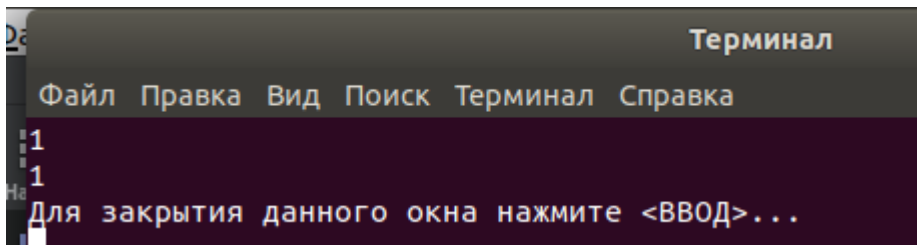
Фишка этой очереди в том, что он всегда отсортирован. По умолчанию в порядке возрастания. Для добавления элемента метод `push`, для удаления `pop`, чтобы посмотреть элемент `top`

```
//по умолчанию по возрастанию
std::priority_queue<int, std::vector<int>>> a;
a.push(12);
a.push(1);
std::cout<<a.top()<<std::endl;
a.push(15);
std::cout<<a.top()<<std::endl;
return 0;
```



Если хотим по убыванию, то из `#include<functional>` надо взять функциональный элемент `std::greater` (если хотите лекцию функциональным элементам и лямбда функциям, то скажите).

```
#include<functional>
int main()
{
    //по убыванию
    std::priority_queue<int, std::vector<int>, std::greater<int>>> a;
    a.push(12);
    a.push(1);
    std::cout<<a.top()<<std::endl;
    a.push(15);
    std::cout<<a.top()<<std::endl;
    return 0;
}
```



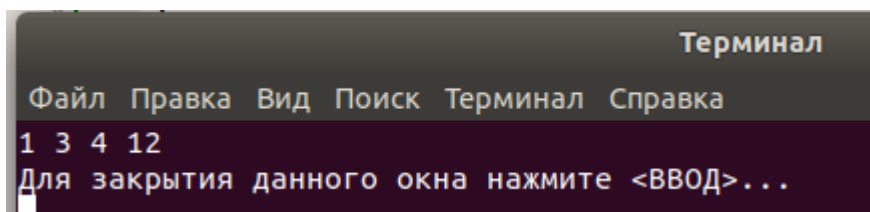
```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
1
1
Для закрытия данного окна нажмите <ВВОД>...
```

7. `std::set`

для подключения `#include<set>`

Этот контейнер работает как множества, т. е. все элементы уникальны. Ходят слухи, что элементы хранятся в бинарном дереве, а именно в красно-черном. Поэтому добавление, удаление, поиск достаточно быстро работает. Примерно за логарифм. Добавляется при помощи метода `insert`.

```
std::set<int>a = {1,3,4};
a.insert(12);|
a.insert(4);
for(int i: a)
{
    std::cout<<i<<" ";
}
std::cout<<std::endl;
return 0;
```



```
Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
1 3 4 12
Для закрытия данного окна нажмите <ВВОД>...
```

8. `std::map`

`#include<map>`

Хранит пары ключ-значение. Например в телефонной книге по фамилии можно найти номер. Сделаем пример с цифрами. Добавляется также при помощи метода `insert`.

```

std::map<int, std::string> nums= {{1, "One"}, {2, "Two"}, {3, "Three"}, {4, "Four"}};
nums.insert({5, "Five"});
for(std::pair<int, std::string> it: nums)
{
    std::cout<<it.first<<" : "<<it.second<<std::endl;
}
//find возвращает итератор, если он равен end,
//значит его нет
bool has = nums.find(3)!=nums.end();
std::cout<<std::boolalpha<<has<<std::endl;|
return 0;

```

```

Терминал
Файл  Правка  Вид  Поиск  Терминал  Справка
1 : One
2 : Two
3 : Three
4 : Four
5 : Five
true
Для закрытия данного окна нажмите <ВВОД>...

```

Итераторы.

У всех контейнеров разная внутренняя реализация. Чтобы не писать один алгоритм обработки данных для каждого контейнера отдельно, придумали унифицированный интерфейс доступа к данным - итераторы. Каждый итератор указывает на отдельный элемент контейнера. Так для каждого контейнера существует итератор, который знает как обращаться к следующему элементу и знает как передвигаться по элементам. По синтаксису итераторы похожи на указатели. т. е. Поддерживаются следующие операции:

*it - разыменовывание

`it++` - переход к следующему

`it+n` — перейти на `n`

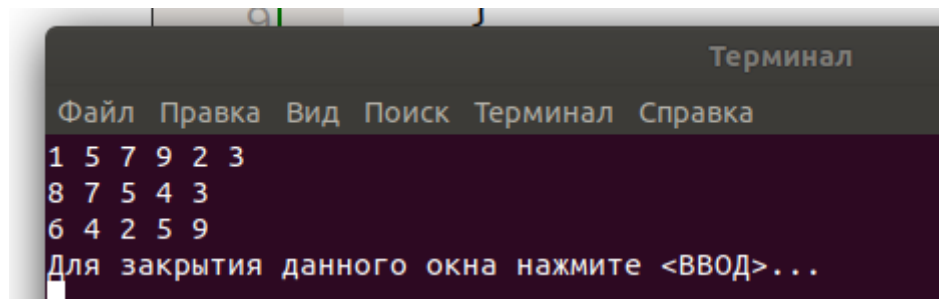
`it1 - it2` — вычислить расстояние

и поддерживаются операции сравнения.

Для всех контейнеров есть 2 особых итератора: `begin` — который указывает на первый элемент, `end` — указывает на следующий за последним.

Рассмотрим пример, который просто печатает все элементы контейнера.

```
#include <iostream>
#include<vector>
#include<array>
//Написали один метод для печати
template<class It>
void printAll(It begin, It end)
{
    for(It i = begin; i!= end; i++)
    {
        std::cout<<*i<<" ";
    }
    std::cout<<std::endl;
}
int main()
{
    const int N = 5;
    //вектор
    std::vector<int> a = {1,5,7,9,2,3};
    //массив
    std::array<int, N> b = {8,7,5,4,3};
    //обычный массив
    int c[N] = {6,4,2,5,9};
    //передаем первый итератор и следующий за последний
    printAll(a.begin(), a.end());
    printAll(b.begin(), b.end());
    //для обычных массивов это просто следующие указатели
    printAll(c, c+N);
    return 0;
}
```



На этом вроде все. Надеюсь до этого момента кто-нибудь дочитает.

Д/З: самому реализовать vector со всеми функциями. Если что пишите мне.