

Стандартная библиотека шаблонов

Контейнеры, алгоритмы, итераторы

История STL (Standard Template Library)

- до включения в стандарт была сторонней разработкой HP, затем SGI;
- архитектура STL была разработана Александром Степановым и Менг Ли;
- включена в стандарт C++ весной 1994 года;
- в 1995 Александр Степанов получил Dr.Dobb's Excellence In Programming Award за создание STL, разделив премию с Линусом Торвальдсом.

Интервью с создателем C++ STL, 1995 г. Часть 1

<https://habr.com/ru/articles/166849/>

Интервью с создателем C++ STL, 1995 г. Часть 2

<https://h.amazingsoftworks.com/ru/post/167257/>

Определения

Контейнер — это способ организации хранения данных.

Алгоритм - функция, применяемая к контейнерам для обработки их данных различными способами.

Итераторы — это указатели на элементы контейнера. Алгоритмы общаются с контейнерами посредством итераторов.

Преимущества

Повышение надежности программ, переносимость и универсальность, уменьшение сроков разработки.

Недостатки

Возможное снижение быстродействия в зависимости от реализации компилятора

Виды контейнеров

Последовательные контейнеры - хранение конечного количества однотипных величин в виде непрерывной последовательности:

- векторы (vector);
- двусторонние очереди (deque);
- строки типа string;
- списки (list);
- адаптеры списков - стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue).

Ассоциативные контейнеры - быстрый доступ к данным по ключу (построены на основе сбалансированных деревьев):

- словари (map);
- словари с дубликатами (multimap);
- множества (set);
- множества с дубликатами (multiset).

Заголовочные файлы

algorithm,

vector,

iterator,

stack,

queue,

deque,

list,

map,

set,

memory,

numeric,

utility,

functional,

bitset

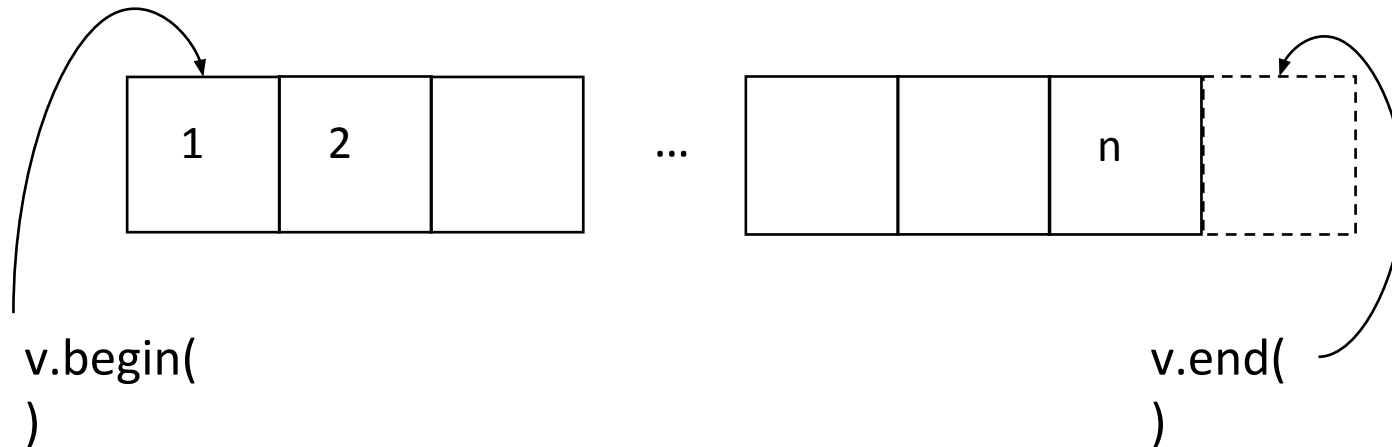
...

Типы, определенные в контейнерных классах

- `value_type`;
- `size_type`;
- `iterator`, `const_iterator`;
- `reverse_iterator`, `const_reverse_iterator`;
- `reference`, `const_reference`;
- `key_type`;
- `key_compare`;

Методы доступа к элементам контейнера

- `begin()`, `begin() const` – итератор начала
- `end()`, `end() const` – итератор конца
- `rbegin()`, `rbegin() const` – реверсивный итератор начала (конец)
- `rend()`, `rend() const` – реверсивный итератор конца (начало)



Методы общие для всех контейнеров

- `size()` – количество элементов;
- `max_size()` – максимальное количество элементов до перераспределения;
- `empty()` – проверка на пустоту.

Последовательный контейнер vector

vector – последовательный контейнер для представления динамического массива. Доступ к произвольному элементу, добавление/извлечение последнего элемента на $O(1)$

```
#include<vector>
```

```
template <class T, class a = Allocator<T> >  
class vector{  
    //тело класса  
};
```

Конструкторы класса vector

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

```
vector <int> v1;  
vector <int> v2(10);  
vector <Student> stud1(10);  
vector <int> v3(15, 5);  
vector <Student> Stud2(5, Student(1));  
vector <int> v4(v1);  
vector <int> v5(v3.begin(), v3.begin() + 2);
```

Присваивание в векторах

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

```
vector <int> v1, v2;
```

```
v1 = v2;
```

```
v1.assign(15, 5);
```

```
v2.assign(v1.begin() + 2, v1.begin() + 6);
```

Доступ к элементам вектора

- reference **operator []** (size_type n);
- const_reference **operator []** (size_type n) const;
- reference **at**(size_type n) ; // генерирует out_of_range;
- const_reference **at**(size_type n) const;
- reference **front**() – ссылка на 1-й элемент;
- const_reference **front**() const;
- reference **back**() – ссылка на последний элемент;
- const_reference **back**() const;

Пример доступа к элементам вектора

```
#include <stdexcept>
#include <vector>
using namespace std;

int main(){
try{
    vector<int> v(10, 5); // 5, 5, 5, 5, 5, 5, 5, 5, 5, 5
    v.front() = 100;      // 100, 5, 5, 5, 5, 5, 5, 5, 5, 5
    v.back()   = 200;      // 100, 5, 5, 5, 5, 5, 5, 5, 5, 200
    cout << v[v.size() - 1] << " " //OK
    //    << v[v.size()] << " " //Run Time Error
        << v.at(v.size()); //exception
}
catch(out_of_range){
    cout << "invalid index";
}
}
```

Функции вектора для работы с оперативной памятью

`size_type capacity() const` – распределенная память в виде числа элементов вектора;

`void reserve(size_type n)` – резервирует память (не будет перераспределяться, пока длина вектора $< n$);

`void resize(size_type sz, T c = T())` – перераспределяет память под вектор.

Изменение объектов класса vector

- void **push_back** (const T& value); - требуется наличие конструктора копирования для типа T
- void **pop_back** ();
- iterator **insert** (iterator position, const T& value);
- void **insert** (iterator position, size_type n, const T& value);
- template <class InputIter>
void **insert** (iterator position, InputIter first, InputIter last);
- iterator **erase**(iterator position);
- iterator **erase**(iterator first, iterator last);
- void **swap**();
- void **clear**() – удаляет все элементы вектора, size() возвращает 0, capacity не меняется
- iterator **emplace** (iterator pos, Args && args...) - emplace создает объект на месте

*insert, erase затрачивают много времени в отличии от push_back и pop_back, т.к. необходимо перестраивать структуру

Пример вызова функций вектора для работы с оперативной памятью

```
vector<int>vec;  
cout << vec.capacity() << endl; //0 0  
vec.push_back(0);  
cout << vec.capacity() << endl; //1 1  
vec.reserve(3);    // capacity - 3, size - 1  
vec.push_back(1);  // capacity - 3, size - 2  
vec.push_back(2);  // capacity - 3, size - 3  
vec.push_back(3);  // capacity - 6, size - 4  
vec.resize(3);     // capacity - 6, size - 3  
//vec: 0 1 2
```


Вывод элементов вектора

```
vector<int> v {1, 2, 3, 4, 5};  
// по индексам  
for (size_t i = 0; i != v.size(); i++)  
    cout << v[i] << ' '  
// по итераторам  
for (vector<int>::iterator i = v.begin(); i != v.end(); i++)  
    cout << *i << ' '  
for (auto i = v.begin(); i != v.end(); i++)  
    cout << *i << ' ';
```

Пример вывод с применением обратного итератора

```
vector<int> v {1, 2, 3, 4, 5};  
auto revIt = v.rbegin();  
while(revIt != v.rend())  
    cout << *revIt++ << ' ' ;  
//5 4 3 2 1
```

Пример изменения векторов

```
vector<int> v1(2), v2(4, 10); //v1: 0 0; v2: 10 10 10 10
vector<int> v3 {1, 2, 3, 4, 5}; //v3: 1 2 3 4 5
v3.insert(v3.begin() + 1, v2.begin(), v2.begin() + 2);
//v3: 1 10 10 2 3 4 5
v1.push_back(5); // v1: 0 0 5
v3.erase(v3.begin()); // v3: 10 10 2 3 4 5
v2.erase(v3.begin(), v3.begin() + 1); // v3: 10 2 3 4 5
v1.swap(v3); // v1: 10 2 3 4 5, v3: 0 0 5
```

*insert, erase затрачивают много времени в отличии от push_back и pop_back, т.к. необходимо перестраивать структуру

Операции сравнения векторов

```
vector<int> v, u;
for (int i = 0; i < 6; i++)
    v.push_back(i);
for (int i = 0; i < 3; i++)
    u.push_back(i + 1);
if (v < u) // true - lexicographically,
    //removed in C++20, see <=>
    cout << "v less" << endl;
else
    cout << "u less" << endl;
}
```

emplace и insert

```
struct Student{
int course;
Student() : course(0)           {cout << "Junior";}
Student(int course) : course(course) {cout << "Middle";}
Student(const Student &student) : course (student.course) {
                                cout << "Clone";}

};
```

```
int main() {
vector<Student> vec;           //Junior
vec.push_back(Student());     //Clone
vec.insert(vec.begin() + 1, Student(1)); //Middle Clone
vec.emplace(vec.begin() + 1, 3); //Middle
}
```

Последовательный контейнер list

- push_back, push_front
- pop_back, pop_front
- back, front
- insert
- erase
- reverse
- merge
- sort
- unique – только для соседних дубликатов
- swap
- remove(value)
- splice – перемещение, выполняется за $O(1)$, т.к. не хранится `_size`
- size – выполняется за $O(N)$, т.к. каждый раз вычисляется заново

Пример splice

Не копирует данные, перенаправляет указатели

```
ostream& operator << (ostream& ostr, const list<int> &list){  
    for (auto &i : list) ostr << " " << i;  
    return ostr;  
}
```

```
int main(){  
    list<int> list1 = {1, 2, 3, 4, 5};  
    list<int> list2 = {10, 20, 30, 40, 50};  
    auto it1 = list1.begin();
```

```
    it1++;
```

```
    //advance(it1, 1); //begin() + i does not work for list*
```

```
    auto it2 = list2.end(); advance(it2, -2);
```

```
    //list1.splice(it1, list2); //input all list2
```

```
    //list1.splice(it1, list2, it2); //input 1 el
```

```
    list1.splice(it1, list2, it2, list2.end()); //input range
```

```
    std::cout << "list1: " << list1 << "\n";
```

```
    std::cout << "list2: " << list2 << "\n";
```

```
}
```

*advance increments given iterator it by n elements

```
list1: 1 10 20 30 40 50 2 3 4 5  
list2:
```

```
list1: 1 40 2 3 4 5  
list2: 10 20 30 50
```

```
list1: 1 40 50 2 3 4 5  
list2: 10 20 30
```

stack - LIFO (last-in, first-out) data structure

Адаптер deque

- top
- empty
- size
- push
- emplace – at the top
- pop
- swap

Последовательный контейнер deque

- поддерживает произвольный доступ – [] за $O(1)$
- методы доступа к началу и концу дека, а также добавление и удаление из начало\конца за $O(1)$
- добавление и удаление из произвольной позиции за $O(n)$
- не обязательно занимает смежные ячейки, поэтому capacity нет
- более рациональное расширение, чем у вектора, т.к. нет необходимости копировать текущие элементы в новую область памяти

swap, shrink_to_fit, clear, insert, emplace, push_front, push_back, emplace_front, emplace_back, erase, resize, pop_front, pop_back

* shrink_to_fit - requests the removal of unused capacity

Псевдоконтейнер bitset

`==, !=, []`

`&=, |=, ^=, ~, <<=, >>=,`

`<<, >>` для `bitset` и для `stream`

`bool test(pos)` – то же, что `[]`, но `throw out_of_range`

`bool all()`, `bool any()`, `bool none()` – проверка единичных битов

`size_t count()` – число единичных битов

`size_t size()` – число бит всего

`bitset<N>& set()` – устанавливает все биты в `true`

`bitset<N>& set(pos, value = true)` – устанавливает `pos` бит в `true`

`bitset<N>& reset()` – устанавливает все биты в `false`

`bitset<N>& reset(pos)` – устанавливает `pos` бит в `false`

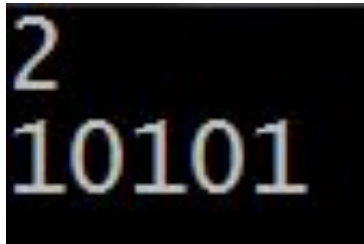
`bitset<N>& flip()` – равносильно `~` для всех битов

`bitset<N>& flip(pos)` – равносильно `~` для бита `pos`

`to_string`, `to_ulong`, `to_ullong`

Пример bitset

```
int main() {  
    bitset<5> b(10);  
    cout << b.count() << endl;  
    cout << b.flip().to_string();  
}
```

The image shows the output of the C++ code. It consists of two lines of text: the number '2' on the first line and the binary string '10101' on the second line. The text is white and set against a solid black rectangular background.

2
10101

Ассоциативный контейнер set

set – упорядоченное (по умолчанию — по возрастанию) множество уникальных ключей. Поиск, удаление, вставка за $O(\log N)$, обычно строятся на основе красно-черных деревьев

Сложность метода insert/erase/find – $\log n$

multiset - то же, но может содержать дубликаты

Методы:

size - размер

insert - вставка

erase - удаление

count – подсчет числа ключей

find - поиск

lower_bound() - поиск \geq

upper_bound() - поиск $>$

Ассоциативный контейнер set - вставка

```
string names[] = {"Alex", "Robert", "Peter", "Anna",  
"Robert", "Mark"};  
  
set<string> nameSet(names, names + 6);  
  
auto it = nameSet.insert("Lisa"); //pair<iterator, bool>  
cout << *(it.first); //Lisa in set  
cout << it.second; //true  
  
it = nameSet.insert("Robert");  
cout << *(it.first); //Robert in set  
cout << it.second; //false
```

Ассоциативный контейнер set - поиск

Сложность метода find – $\log n$

```
string names[] = {"Alex", "Robert", "Peter", "Anna", "Robert", "Mark"};
set<string, greater<string> > nameSet(names, names + 6);
nameSet.insert("Lisa"); nameSet.insert("Robert");
nameSet.insert("Nick"); nameSet.erase("Robert");
cout << "set size =" << nameSet.size() << endl;
auto it = nameSet.begin();
while(it != nameSet.end())
    cout << *it++ << endl;
string searchName;
cin >> searchName;
it = nameSet.find(searchName);
if(it == nameSet.end())
    cout << "There are no name " << searchName;
else
    cout << *it << " finded!";
```

Пример

```
set <string> tshortMania;  
string color;  
while(cin >> color){  
    auto it = tshortMania.insert(color);  
    if (it.second)  
        cout << "Buy it!\n";  
    else  
        cout << "Do not buy it!!!\n";  
}  
cout << "Mania`s coollection:\n";  
for(auto it : tshortMania)  
    cout << it << ' ';
```

Методы upper_bound, lower_bound

Сложность – $\log n$

```
string names[] = {"Alex", "Robert", "Peter", "Anna", "Robert", "Mark"};
set<string> nameSet(names, names + 6);
string lower, upper;
cout << "input range bounds for ex. A Dxx"<<endl;
cin >> lower >> upper;
set<string>::iterator it;
it = nameSet.lower_bound(lower);
while(it != nameSet.upper_bound(upper))
cout << *it++ << endl;
```


Компаратор для множества

```
struct strCmp {  
    bool operator()(const string& lhs, const string& rhs) const {  
        return lhs < rhs;  
    }  
};
```

```
set<string, strCmp> v;
```

pair

`pair<T1, T2>` хранит пару из переменных не обязательно одинаковых типов

```
pair<int, int> interval = {0, 42};
```

Первый элемент доступен через поле `.first`, а второй через `.second`.

```
pair<string, int> p1, p2 = make_pair("Hello", 2021);  
cout << p2.first << " " << p2.second << endl;  
p1.swap(p2);
```

```
cout << get<0>(p1) << " " << get<1>(p1) << endl;  
cout << get<string>(p1) << " " << get<int>(p1) << endl;
```

Ассоциативный контейнер map.

map – упорядоченный ассоциативный контейнер пар "ключ-значение" с уникальными ключами. Поиск, удаление, вставка за $O(\log N)$, обычно строятся на основе красно-черных деревьев

multimap - то же, но может содержать дубликаты. Порядок следования ключей дубликатов – по мере вставки

Методы

[] - значение

insert - вставка

erase - удаление

count – подсчет ключей

find - поиск

lower_bound – поиск \geq

upper_bound – поиск $>$

Пример работы с map

```
const int n = 5;
string name,
        names[n] = {"Zaicev", "Volkov", "Slonov", "Lisov", "Bobrov"};
int marks[n] = {3, 4, 2, 5, 5};
map<string, int> mapExam;
for(int j = 0; j < n; j++)
    mapExam.insert(make_pair(names[j], marks[j]));
for(auto it = mapExam.begin(); it != mapExam.end(); it++)
    cout << it->first << " " << it->second << endl;
```

Пример описания журнала с оценками на основе map

```
const int n = 5;
    names[n] = {"Zaicev", "Volkov", "Slonov", "Lisov", "Bobrov"};
int marks[n] = {3, 4, 2, 5, 5};
map<string, int> mapExam;
for(int j = 0; j < n; j++)
    mapExam[names[j]] = marks[j];

cout << "input name: ";
cin >> name;
cout << "Exam mark: " << mapExam[name] << endl;
for(auto it = mapExam.begin(); it != mapExam.end(); it++)
    cout << it->first << " " << it->second << endl;
```

Пример

```
map <string, int> tshortMania;  
string color;  
while(cin >> color) {  
    tshortMania[color]++;  
}  
cout << "Mania`s collection:\n";  
for(auto it : tshortMania)  
    cout << it.first << ':' << it.second << "\n";
```

Пример map (пример из cppreference)

```
void print_map(const std::map<std::string, int>& m){
    for (const auto& [key, value] : m)
        std::cout << key << " = " << value << "; ";
    std::cout << "\n";
}

int main(){
    std::map<std::string, int> m { {"CPU", 10}, {"GPU", 15}, {"RAM", 20} };
    print_map(m);
    m["CPU"] = 25;    // update an existing value
    m["SSD"] = 30;    // insert a new value
    print_map(m);
}
```

Пример map

Сколько повторяющихся элементов в векторе, с сохранением позиций дубликатов

```
vector<long> v{/**/};
map<long, vector<long>> m;
for(int i = 0; i < v.size(); i++)
    m[v[i]].push_back(i);
for(auto& [key, val] : m){
    cout << key << ": elem`s count - " << val.size()
        << ", positions:\n";
    for(auto i : val)
        cout << i << " ";
    cout << '\n';
}
```


Итераторы

- *Входной итератор* (cin или файл, только чтение) – inputIterator
- *Выходной итератор* (cout или файл, только запись) - outputIterator
- *Прямой итератор* (только ++) – forwardIterator

forward_list

- *Двухнаправленный итератор* (только ++ и --) – bidirectionalIterator

list, set, map

- *Итератор со случайным доступом* (++ , -- , + , -) – randomAccessIterator

vector, deque, stack

*<https://en.cppreference.com/w/cpp/iterator>

Допустимые операции над итераторами разных типов

Тип итератора	++	чтение = *i	запись *i =	--	[]
Произвольный доступ	+	+	+	+	+
Двунаправленный	+	+	+	+	-
Прямой	+	+	+	-	-
Входной	+	-	+	-	-
Выходной	+	+	-	-	-

Методы итераторов

- **advance** - увеличивает значение итератора на *n*, если *n* отрицательное, двунаправленный итератор уменьшается, поведение прямого оператора не определено.

```
list<int> list1 = {10, 20, 30, 40, 50};  
auto it1 = list1.begin();  
advance(it1, 2);
```

- **distance** - возвращает количество элементов между итераторами; не определено, если второй итератор недостижим из первого или наоборот.

```
cout << distance(it1, list1.end());
```

- **next/prev** - возвращает итератор, смещенный на *n*

```
auto nx = next(it1, 2);  
auto pr = prev(it1, 1);  
cout << *nx << ' ' << *pr; //50 20
```

*<https://en.cppreference.com/w/cpp/header/iterator>

Потоковые итераторы: ostream_iterator

```
#include<iterator>
```

```
int main() {
```

```
list<int> lst {1, 2, 3, 4, 5};
```

```
//---for console
```

```
ostream_iterator <int> osIter(cout, ", ");
```

```
/*---for file
```

```
ofstream ofile("file.dat");
```

```
ostream_iterator<int> osIter(ofile, " ");
```

```
*/
```

```
copy(lst.begin(), lst.end(), osIter);
```

```
}
```

Потоковые итераторы: istream_iterator

```
list<float> lst(5);  
/*---for console, press Ctrl+Z to end input for  
istream_iterator*/  
istream_iterator<float> cinIter(cin);  
istream_iterator<float> endOfStream;  
copy(cinIter, endOfStream, lst.begin());  
/*---for file  
ifstream ifile("file.dat");  
istream_iterator<float> fileIter(ifile);  
copy(fileIter, endOfStream, lst.begin());  
*/  
ostream_iterator<float> osIter(cout, ",");  
copy(lst.begin(), lst.end(), osIter);
```

Алгоритмы STL

min_element/max_element – поиск наибольшего/наименьшего значения

binary_search – бинарный поиск, значения должны быть упорядочены по возрастанию

upper_bound, lower_bound - поиск элемента не меньше (строго больше) заданного, значения должны быть упорядочены по возрастанию

find – первый элемент с указанным значением

count – количество элементов с указанным значением

equal – признак идентичности всех элементов двух контейнеров

search – последовательность значений одного контейнера идентичная последовательности другого

merge – слияние двух отсортированных последовательностей

sort – сортировка последовательности

for_each – выполнение функции для каждого элемента контейнера

transform – преобразование элементов контейнера

accumulate – сумма элементов заданного диапазона

rotate – циклический сдвиг вправо

find_if, count_if, ...

min_element/max_element

```
vector<int> v{2, 1, 3, 4, 6, 5};  
auto vmax = max_element(v.begin(), v.end());  
cout << "max element is: " << *vmax << ", at pos " <<  
vmax - v.begin() << "\n";
```

binary_search

только для контейнеров с произвольным доступом, возвращает bool

```
vector<int> v{2, 1, 3, 4, 6, 5};  
sort(v.begin(), v.end());  
int val = 3;  
if (binary_search(v.begin(), v.end(), val)  
    cout << "v contains " << val;
```


binary_search и компаратор

```
bool cmp(const int &l, const int &r) {  
    return (l % 10 < r % 10);  
}
```

```
int main() {  
    vector<int> v1{2, 1, 3, 4, 6, 5};  
    vector<int> v2{72, 91, 103, 4, 86, 15};  
    sort(v1.begin(), v1.end());  
    cout << binary_search(v1.begin(), v1.end(), 3) << "\n";  
  
    sort(v2.begin(), v2.end(), cmp);  
    cout << binary_search(v2.begin(), v2.end(), 3, cmp);  
}
```

upper_bound, lower_bound

логарифмическая сложность только для контейнеров с произвольным доступом, возвращает итератор на найденный элемент (или end), можно задать компаратор, сложность $\log N$

```
deque<int> v1{2, 1, 3, 4, 7, 5};  
sort(v1.begin(), v1.end());  
auto lb = lower_bound(v1.begin(), v1.end(), 5); // >=  
auto ub = upper_bound(v1.begin(), v1.end(), 5); // >  
cout << *lb << " " << *ub << "\n"; // 5 7
```

Алгоритм find

Сложность $O(n)$

```
int main() {  
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};  
    vector<int> v {1, 2, 3, 4, 5, 6, 7, 8};  
    int* ptrA = find(arr, arr + 8, 3);  
    auto ptrV = find(v.begin(), v.end(), 3);  
    cout << "pos value 3 in arr " << (ptrA - arr) << endl;  
    cout << "pos value 3 in vec " << (ptrV - v.begin());  
}
```

Алгоритмы count, equal

Сложность $O(n)$

```
int main() {  
vector<int> v {1, 2, 3, 4, 5, 4, 7, 8};  
cout << count(v.begin(), v.end(), 4);  
//-----  
if(equal(v.begin(), v.begin() + v.size() / 2, v.rbegin()))  
    cout << "symmetric";  
else  
    cout << "unsymmetric";  
}
```

Алгоритм search

```
int arr[] = {1, 2, 3, 4, 5, 3, 4, 8};
int sarr[] = {3, 4};
vector<int> vec (arr, arr + 8),
              svec(sarr, sarr + 2);
int* ptrA = search(arr, arr + 8, sarr, sarr + 2);
auto ptrV = search(vec.begin(), vec.end(), svec.begin(), svec.end());
auto ptrVA = search(vec.begin(), vec.end(), sarr, sarr + 2);
int *ptrAV = search(arr, arr + 8, svec.begin(), svec.end());
cout << "1-st sarr pos in arr " << (ptrA - arr) << endl;
cout << "1-st svec pos in vec " << (ptrV - vec.begin()) << endl;
```

Алгоритм merge

```
int arr[] = {1, 2, 3, 4, 5, 8};  
int sarr[] = {6, 7};  
int marr[8];  
vector<int> vec(arr, arr + 6), svec(sarr, sarr + 2), mvec(8);  
merge(arr, arr + 6, sarr, sarr + 2, marr);  
merge(vec.begin(), vec.end(), svec.begin(), svec.end(), mvec.begin());  
merge(vec.begin(), vec.end(), sarr, sarr + 2, mvec.begin());  
merge(arr, arr + 6, svec.begin(), svec.end(), marr);
```

Алгоритм sort

Тип сортировки по умолчанию – неубывание, должен существовать оператор сравнения $<$, можно задать компаратор

Сложность $n * \log n$

```
int main() {  
    float arr[] = {1.1, 5.1, 6.1, 3.1, 2.1, 7.1};  
    sort(arr, arr + 6); // default = less  
    // или sort(begin(arr), end(arr));  
    // также можно использовать компараторы  
    sort(arr, arr + 6, greater<float>());  
}
```

Предопределенные компараторы

<code>bool = equal_to(T, T)</code>	<code>X == Y</code>
<code>bool = not_equal_to(T, T)</code>	<code>X != Y</code>
<code>bool = greater(T, T)</code>	<code>X > Y</code>
<code>bool = less(T, T)</code>	<code>X < Y</code>
<code>bool = greater_equal(T, T)</code>	<code>X >= Y</code>
<code>bool = less_equal(T, T)</code>	<code>X <= Y</code>
<code>bool = logical_and(T, T)</code>	<code>X && Y</code>
<code>bool = logical_or(T, T)</code>	<code>X Y</code>
<code>bool = logical_not(T, T)</code>	<code>!X</code>

Пример sort с компаратором пользователя

Сигнатура компаратора

```
bool cmp(const Type1 &a, const Type2 &b);
```

```
struct Student{  
    string name;  
    int points;  
};
```

```
bool cmpStudents (const Student &s1, const Student &s2) {  
    return s1.points < s2.points;  
}
```

```
int main() {  
    Student group113[] = {Student("Ivan", 44), Student("Boris", 33),  
        Student("Olga", 22), Student("Anastasiya", 11)};  
    vector<Student> v(group113, group113 + 4);  
    sort(v.begin(), v.end(), cmpStudents); // ("Anastasiya", 11), ("Olga", 22), ...  
}
```

Алгоритм sort и функциональный объект пользователя

```
struct Student{
string name;
int points;
bool operator < (const Student &s) ;
};

bool Student::operator < (const Student &s) {
return points < s.points;
}

int main(){
Student group113[] = {Student("Ivan", 44),          Student("Boris",
33), Student("Olga", 22), Student("Anastasiya", 11)};
vector<Student> v(group113, group113 + 4);
sort(v.begin(), v.end()); // ("Anastasiya",11), ("Olga",22), ...
}
```

Алгоритм for_each

for_each(iterator, iterator, func)

- func – унарная, не должна изменять данные контейнера, возвращаемый тип игнорируется

```
void printArray(int value) {  
    //может быть template  
    cout << value << " ";  
}
```

```
int main() {  
    vector<int> v;  
    /*тут должно быть заполнение вектора*/  
    for_each(v.begin(), v.end(), printArray);  
}
```

Алгоритм transform

```
int absolute(int a){  
    return a < 0 ? -a : a;  
}
```

```
void print (int a){  
    cout << a << " ";  
}
```

```
int main(){  
    int a[3] {1, -2, 3};  
    vector<int> v{4, -5, 6};  
    //to the same container  
    transform(a, a + 3, a, absolute);  
    //to another container  
    transform(v.begin(), v.end(), a, absolute);  
    for_each(a, a + 3, print); //output a;  
}
```

Лямбда-выражения

Это более краткий компактный синтаксис для определения объектов-функций:

```
[] (параметры) { действия }
```

В круглых скобках указываются параметры (начиная со стандарта C++14 можно указывать значения по умолчанию), в фигурных скобках помещаются действия, выполняемые лямбда-выражением.

```
[]() { std::cout << "Hello" << std::endl; }
```

Каждый раз, когда компилятор встречает лямбда-выражение, он генерирует новый тип класса, который представляет объект-функцию. Например:

```
class __Lambda1234{public:  auto operator()() const { std::cout << "Hello" << std::endl; };
```

Такой класс имеет произвольное, но уникальное сгенерированное имя. А действия лямбда-выражения определяются в виде оператора (), причем вместо возвращаемого типа применяется слово auto.

Именованные лямбда-выражения

```
int main() {  
    // переменная hello представляет лямбда-выражение  
    auto hello { []() {std::cout << "Hello" << std::endl;} };  
  
    // через переменную вызываем лямбда-выражение  
    hello();    // Hello  
}
```

Лямбда-выражения с параметрами

```
auto print { [] (const std::string& text) {  
    std::cout << text << std::endl; } };  
// вызываем лямбда-выражение  
print("Hello World!");           // Hello World!  
print("Good bye, World...");     // Good bye, World...
```

Возвращение значения в лямбда-выражениях

```
// автовыведение типа возвращаемого значения  
auto sum { [] (int a, int b) {return a + b;} };
```

```
// вызываем лямбда-выражение  
std::cout << sum(10, 23) << std::endl;    // 33
```

```
// присваиваем его результат переменной  
int result { sum(1, 4) };  
std::cout << result << std::endl;          // 5
```

```
// явное указание типа возвращаемого значения  
auto sum { [] (int a, int b) -> double {return a + b;} };
```

```
// вызываем лямбда-выражение  
std::cout << sum(10, 23) << std::endl;    // 33
```

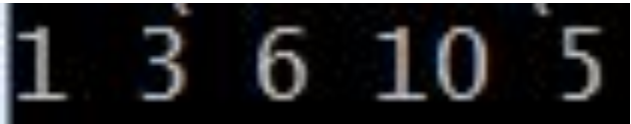

Лямбда-выражения в алгоритмах STL

```
transform(v.begin(), v.end(), v.begin(),  
          [](double i) ->double {return ++i;});  
for_each(v.begin(), v.end(),  
          [](double i){std::cout << i << std::endl;});
```

Алгоритм accumulate (<numeric>)

```
struct Student{
//must have all constructors
// to call plus<> you must have: T operator +(const T&) const
Student operator +(const Student& s) const {
    return Student (points + s.points); }
};

int main(){
list<int> lstInt {11, 12, 13, 14, 15};
accumulate(lstInt.begin(), lstInt.end(), 0);
accumulate(lstInt.begin(), lstInt.end(), 1, multiplies<int>());
accumulate(lstInt.begin(), lstInt.end(), 0, [](auto a, auto b) {
    auto c = a % 10 + b % 10;
    cout << c << " ";
    return c;});
list<Student> lstS {/**/};
accumulate(lstS.begin(), lstS.end(), Student(0), plus<Student>());
}
```



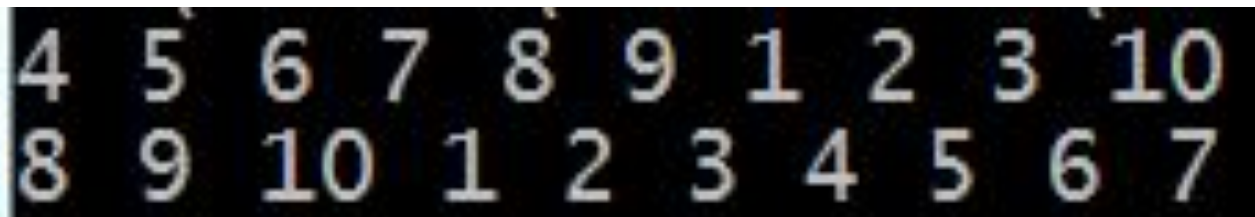
Пример rotate

Параметры:

ForwardIt first, ForwardIt n_first, ForwardIt last

Меняет элементы [first, last) таким образом, чтобы элемент n_first стал первым, а элемент n_first – последним

```
array<int, 10> arr1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
array<int, 10> arr2 (arr1);  
int k = 3;  
//left  
rotate(arr1.begin(), arr1.begin() + k, arr1.end() - 1);  
//right  
rotate(arr2.rbegin(), arr2.rbegin() + k, arr2.rend());
```



4	5	6	7	8	9	1	2	3	10
8	9	10	1	2	3	4	5	6	7

Предопределенные функциональные объекты (functional)

<code>T = plus (T, T)</code>	$X + Y$
<code>T = minus (T, T)</code>	$X - Y$
<code>T = multiplies (T, T)</code>	$X * Y$
<code>T = divides (T, T)</code>	X / Y
<code>T = modulus (T, T)</code>	$X \% Y$
<code>T = negate (T)</code>	$-X$

Алгоритм find_if и функциональные объекты

Функциональный объект – унарный предикат, м.б. функцией или классом-функтором (без данных, но с единственным методом)

```
struct Student{ /*...*/};
```

```
struct acsTTestSt1{ // класс-функтор  
bool operator () (const Student& s) const { return s.points > 50; }  
};
```

```
bool acsTTestSt2(const Student &s) { return s.points > 50;}
```

```
int main(){  
Student arr[] = {Student(11), Student (6), Student (88)};  
Student *p1 = find_if(arr, arr + 8, acsTTestSt1());  
Student *p2 = find_if(arr, arr + 8, acsTTestSt2);  
cout << p1 - arr << " " << p2 - arr; // 1 1  
}
```

Пример count, count_if

Параметры:

count(InputIt first, InputIt last, const T& value)

count_if(InputIt first, InputIt last, UnaryPredicate p);

```
bool even (int i) {  
    return i % 2 == 0;  
}
```

```
int main() {  
    vector<int> v {1, 2, 3, 4, 5, 4, 7, 8};  
    cout << count(v.begin(), v.end(), 4) << "\n";  
    cout << count_if(v.begin(), v.end(), even);  
}
```

Самостоятельное изучение: контейнеры

- `multimap`, `unordered_map`, `unordered_multimap`,
- `multiset`, `unordered_set`, `unordered_multiset`
- `forward_list`, `priority_queue`

Самостоятельное изучение: алгоритмы

- adjacent_find, mismatch
- swap_ranges, remove, fill, copy, generate, reverse, rotate, unique, shift_left, shift_right, replace
- is_sorted, stable_sort, nth_element
- includes, set_difference, set_intersection, set_union
- is_permutation, next_permutation, prev_permutation

Дополнительные источники

- <https://en.cppreference.com/w/cpp/header/algorithm>