



STL: Концепция алгоритмов

- В STL для работы с контейнерами имеется *библиотека алгоритмов* с функциями для сортировки, поиска, копирования ...
- Алгоритмы – глобальные функции оперирующие с итераторами


 Существуют «специализированные» версии алгоритмов, например `list.sort()`, `map.find()`, которые более эффективны по сравнению с «глобальными»

- Для настройки алгоритмов предусмотрен механизм «подключения» пользовательских функций

 Основная задача при работе с алгоритмами состоит в написании пользовательских функций

STL Алгоритмы: простейшая классификация

- Немодифицирующие алгоритмы: поиск, подсчёт числа элементов ...
- Модифицирующие: удаление, вставка, замена ...
- Алгоритмы сортировки и работы с отсортированными последовательностями

 Имена алгоритмов в некоторой степени отражают то, что эти алгоритмы делают, однако будьте осторожны

Заголовочные файлы:

```
#include <algorithm>
#include <numeric>      // some of the STL algorithms are provided
                        // for numeric processing
#include <functional>  // for function objects and function adapters
```

Немодифицирующие алгоритмы

👉 Вызов этих функций не меняет контейнер

| | | |
|-----------------------------------------------|-------------------------------------------------|-------|
| <code>for_each()</code> | вызывает заданную функцию для каждого элемента | |
| <code>for_each_n()</code> | для первых N элементов | C++17 |
| <code>find()</code> | находит первый заданный элемент | |
| <code>find_if()</code> | находит первый элемент удовлетворяющий условию | |
| <code>count()</code> | число элементов равных заданному | |
| <code>count_if()</code> | число элементов удовлетворяющих условию | |
| <code>min_element()</code> | минимальный элемент | |
| <code>max_element()</code> | максимальный элемент | |
| <code>minmax_element()</code> | возвращает пару <i>min, max</i> | C++11 |
| <code>all_of()</code> , <code>any_of()</code> | проверяет, истинен ли предикат для всех, любого | |
| <code>none_of()</code> | или ни одного из элементов | C++11 |

STL: `for_each()`

```
for_each(beg, end, UnaryFunction op)
```

Один из самых универсальных алгоритмов:

- выполняет «функцию» `op(elem)` для всех элементов из `[beg, end)`, а возвращаемое значение `op()` игнорируется
- `for_each` возвращает копию функционального объекта `op()` после выполнения последнего вызова

👉 если `op(elem)` модифицирует элемент то `for_each()` изменит содержимое контейнера

● `for_each()` для печати элементов контейнера

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct print { // класс содержащий operator()
    print(int c=0) : count(c) {}
    void operator()(int x) { cout << x << ' '; ++count; }
    int count;
};

void main() {
    vector<int> V {1,2,3};
    print P = for_each(V.begin(),V.end(),print(0)); // print(0) is ctr
    cout << endl; // 1 2 3
    cout << "objects printed: " << P.count << endl; // objects printed: 3
}
```

Используем лямбда, только печать

C++11

```
// V = {1,2,3};  
for_each( begin(V), end(V), [](int x){cout << x << ' ';} );  
cout << endl; // 1 2 3
```

... С ПОДСЧЕТОМ КОЛИЧЕСТВА ВЫЗОВОВ

```
int count = 0;  
auto f = for_each( begin(V), end(V),  
    [&count](int x) -> void{ cout << x << ' '; ++count; } );  
cout << endl; // 1 2 3  
cout << count << "objects printed" << endl; // 3 objects printed  
cout << "check returned object f:";  
f(-123);  
cout << ": count= " << count << endl;  
Output: check returned object f:-123 : count= 4
```

● `for_each()` МОЖЕТ ИЗМЕНЯТЬ ЭЛЕМЕНТЫ В КОНТЕЙНЕРЕ

```
struct add_value {  
    add_value(int v = 0) { the_value=v; }  
    void operator()(int& elem) const { elem+=the_value; }  
private:  
    int the_value;  
};  
  
for_each(V.begin(), V.end(), add_value(10));  
for_each(V.begin(), V.end(), print() );  
cout << endl; // 11 12 13
```

Используем лямбда

C++11

```
for_each( begin(V), end(V), [](int& x) { x+=10; } );
```

👉 обратите внимание на тип аргумента: `int&`

STL: Минимальный и максимальный элементы

```
min_element (Iterator beg, Iterator end, [CompFunc op])  
max_element (Iterator beg, Iterator end, [CompFunc op])  
minmax_element (Iterator beg, Iterator end, [CompFunc op])
```

- возвращают **итератор**, а **minmax** пару итераторов, на минимальный/максимальный элемент из `[beg,end)`
- `op(elem1,elem2)` должен возвращать `true` если `elem1 < elem2`; версия без `op()` использует `operator<()`

```
V = {11, 2, 5, 36, 4, 14};  
vector<int>::iterator it1 = min_element(V.begin(),V.end());  
cout << "min_element= " << *it1 << endl; // min_element= 2  
vector<int>::iterator it2 = max_element(V.begin(),V.end());  
cout << "max_element= " << *it2 << endl; // max_element= 36
```


Используем лямбда: функцией сравнения по модулю 5

C++11

```
V = {11, 2, 5, 36, 4, 14};  
auto it5 = max_element( begin(V), end(V),  
    [](int x,int y){ return (x%5 < y%5); } ); // comparison func  
cout << "max(mod 5)= " << *it5 << endl; // max(mod 5)= 4
```

minmax_elements() и сравнение по модулю 13

C++11

```
auto min_max = minmax_element(begin(V),end(V),  
    [](int x,int y){ return (x%13 < y%13); } );  
cout << "min_max(mod 13)= (" << *min_max.first << ", "  
    << *min_max.second << ")" << endl; // min_max(mod 13)= (14, 11)
```

Модифицирующие алгоритмы

| | |
|------------------------------|-------------------------------------------|
| <code>transform()</code> | выполняет операцию с каждым элементом |
| <code>copy()</code> | копирует, начиная с первого элемента |
| <code>copy_backward()</code> | копирует в обратном порядке |
| <code>copy_n()</code> | копирует <code>n</code> элементов |
| <code>copy_if()</code> | копирует элементы удовлетворяющие условию |
| <code>replace(),...</code> | заменяет элементы удовлетворяющие условию |
| <code>generate(),...</code> | заменяет элементы результатом операции |
| <code>remove(),...</code> | удаляет элементы удовлетворяющие условию |
| <code>unique()</code> | удаляет соседние эквивалентный элементы |
| <code>fill()</code> | заполняет одним элементом |
| <code>iota()</code> | заполняет возрастающей серией |


STL: transform()

```
transform(srcBeg, srcEnd, destBeg, UnaryFunction op)
```

- выполняет функцию `op(elem)` для элементов из `[srcBeg, srcEnd)` и записывает возвращаемое значение в `[destBeg, ...)`
- возвращает позицию последнего элемента в принимающем контейнере
- `srcBeg` и `destBeg` могут принадлежать одному контейнеру

```
transform(srcBeg, srcEnd, destBeg, Unary op)
```



 принимающий контейнер должен быть достаточно большим чтобы вместить входящие элементы

● «Унарный минус» для вектора

```
V = {2, 3, 5, 7};  
transform ( begin(V), end(V),      //source range  
            begin(V),              //destination  
            [](int x){return -x;} ); //operation  
// result: V= -2 -3 -5 -7
```

● Умножение на -10 и сохранение в другом контейнере

```
list<int> L( size(V) );           //list of required size  
transform ( begin(V), end(V),     //source range  
            begin(L),             //destination  
            [](int x){return -10*x;} ); //operation  
// result: L= 20 30 50 70
```

● Итератор вставки в конец контейнера: `back_inserter()`

```
// V = -2 -3 -5 -7;    L= 20 30 50 70
transform ( begin(V), end(V),    //source range
            back_inserter(L),    //insert into the end of L
            [](int x){return x%3;} ); //operation
// result of back_inserter: L= 20 30 50 70 -2 0 -2 -1
```

● Итератор вставки общего вида: `inserter()`

```
transform ( begin(V), end(V),    //source range
            inserter(L,begin(L)), //insert before begin of L
            [](int x){return x/3;} ); //operation
// result of inserter(begin): L= 0 -1 -1 -2 20 30 50 70 -2 0 -2 -1
```

transform для двух «ИСТОЧНИКОВ»

```
transform(srcBeg1,srcEnd1,srcBeg2,destBeg,Binary op)
```



● Пример с итератором вставки в конец результирующего контейнера

```
vector<int> V1 {1, 2, 3};  
vector<double> V2 {5.1, 6.2, 7.3, 8.4};  
vector<double> Vres; // destination vector  
transform ( begin(V1), end(V1), //source-1 range  
            begin(V2),          //source-2 begin  
            back_inserter(Vres), //insert into the end of Vres  
            [](int x1,double x2){return x1+x2;} ); //operation  
// result: Vres= 6.1 8.2 10.3
```

STL: generate() и iota()

- `generate(Iterator beg, Iterator end, Func op)`
заменяет элементы контейнера на результат вызова `op()` (без аргумента)
- `iota(Iterator beg, Iterator end, T startValue)`
заполняет контейнер приращениями: `startValue, startValue+1, ...`

```
V.resize(5);  
generate ( begin(V), end(V),          //source range  
           [](){return rand()%1000;} ); //operation  
// generated: V= 807 249 73 658 930
```

```
array<int,10> A;  
iota( begin(A), end(A), 1 );  
// iota: A= 1 2 3 4 5 6 7 8 9 10
```

Алгоритмы сортировки

| | |
|---------------------------------|----------------------------------------------------------------------------------------------|
| <code>sort()</code> | сортировка в порядке возрастания |
| <code>partial_sort()</code> | сортировка первых <code>n</code> -элементов |
| <code>stable_sort()</code> | сортировка с сохранением порядка равных элементов |
| <code>partition()</code> | делит элементы на две группы относительно заданного предиката |
| <code>stable_partition()</code> | <code>partition</code> с сохранением порядка равных элементов |
| <code>nth_element()</code> | ставит на нужное место <code>n</code> -й элемент, и располагает все меньшие элементы до него |
| <code>binary_search()</code> | «поиск» элемента в отсортированном контейнере |
| <code>lower_bound()</code> | позиция первого элемента <u>не меньшего</u> заданного |
| <code>upper_bound()</code> | позиция первого элемента <u>большего</u> заданного |

STL: `sort()` и `partial_sort()`

`sort(beg, end, BinaryPredicate op)`

- `sort()` сортирует элементы в промежутке `[beg, end)` используя `op(elem1, elem2)` как критерий сортировки

`partial_sort(beg, endSort, end, BinaryPredicate op)`

- в `partial_sort()` сортированным будет лишь промежуток `[beg, endSort)` (`endSort ≤ end`)

Эффективность алгоритмов

- `sort()`: $n \times \log(n)$ сравнений C++11
- `sort()`: $n \times \log(n)$ в среднем, но n^2 в худшем случае C++03
- `stable_sort()`: $n \times \log^2(n)$ сравнений

● Сортировка «по умолчанию» – в возрастающем порядке

```
// V= 807 249 73 658 930  
vector<int> W(V);  
sort( begin(W), end(W) ); //sort in ascending order (default)  
// sort(<): W= 73 249 658 807 930
```

● Сортировка в убывающем порядке

```
sort( begin(V), end(V), //range  
      [](int x,int y){return x>y;} ); //sort in descending order  
// sort(>): V= 930 807 658 249 73
```

● Отсортировать первые несколько элементов: `partial_sort()`

```
W = {3,8,3,6,7,9,1,5,10,6};  
partial_sort ( begin(W),           //begin of the range  
               begin(W)+5,         //end of sorted range  
               end(W),             //end of full range  
               [](int x,int y){return x>y;} ); //sorting lambdas  
// sort(>) first five elements: W= 10 9 8 7 6 3 1 3 5 6
```

👉 Выражения `begin(W)+5` или `W.end()-2` допустимы лишь для контейнеров с произвольным доступом: `vector<>`, `array<>`

● `binary_search()` поиск элемента в отсортированном контейнере

```
W = {3,8,3,6,7,9,1,5,10,6};  
auto lgt = [](int x,int y){return x>y;}; //sorting lambdas  
sort( begin(W), end(W), lgt);  
// W(>)= 10 9 8 7 6 6 5 3 3 1  
bool is3 = binary_search( begin(W), end(W), 3, lgt); // lgt must be  
if( is3 ) {  
    cout << "W contains 3" << endl;  
} else {  
    cout << "W does not contain 3" << endl;  
}  
// W contains 3
```

👉 В `binary_search()` должен быть указан тот же алгоритм сортировки, что использовался при сортировке в `sort()`

● `lower_bound()` & `upper_bound()`

☞ позиция для вставки элемента в отсортированный контейнер не нарушающая сортировки

```
// W(>)= 10 9 8 7 6 6 5 3 3 1
for ( int i = 3; i < 5; ++i ) {
    auto posL = lower_bound( begin(W), end(W), i, lgt );
    auto posU = upper_bound( posL, end(W), i, lgt );
    cout << i << " can be inserted in interval ["
        << distance( begin(W), posL ) << ", "
        << distance( begin(W), posU ) << "]" << endl;
}
// 3 can be inserted in interval [7,9]
// 4 can be inserted in interval [7,7]
```

☞ элементы всегда вставляются **перед** указанным итератором