

STL: Концепция алгоритмов

- В STL для работы с *контейнерами* имеются общие *алгоритмы* обработки: сортировка, поиск, копирование и тому подобные


👉 Существуют «специализированные» версии алгоритмов: `list.sort()`, `map.find()` ... более эффективные по сравнению с «глобальными»

- Алгоритмы – глобальные функции оперирующие с *итераторами*
- 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

☞ Немодифицирующие алгоритмы оставляют контейнер абсолютно таким же как он был до вызова

<code>for_each()</code>	вызывает заданную функцию для каждого элемента
<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)

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;
};

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
```

Lambda expression: только печать

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

... С ПОДСЧЕТОМ КОЛИЧЕСТВА ВЫЗОВОВ

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


`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
```

Lambda expression:

(C++11)

```
for_each( V.begin(), V.end(), [](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  
auto it2 = max_element(V.begin(),V.end());  
cout << " max_element= " << *it2 << endl; // max_element= 36
```


Пример: функцией сравнения по модулю 5

```
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_element(mod 5)= " << *it5 << endl;
```

Output: max_element(mod 5)= 4

minmax_elements() и сравнение по модулю 13

```
V = {11,2,5,36,4,14};  
auto min_max = minmax_element(begin(V),end(V),  
    [](int x,int y){ return (x%13 < y%13); } ); // comparison func  
cout << " mod 13: min= " << *min_max.first  
    << " max= " << *min_max.second << endl;
```

Output: mod 13: min= 14 max= 11

Модифицирующие алгоритмы

<code>transform()</code>	выполняет операцию с каждым элементом
<code>copy()</code>	копирует, начиная с первого элемента
<code>copy_backward()</code>	копирует, начиная с последнего элемента
<code>copy_n()</code>	копирует <code>n</code> элементов
<code>copy_if()</code>	копирует элементы удовлетворяющий условию
<code>replace(), ...</code>	заменяет элементы
<code>generate()</code>	заменяет элементы результатом операции
<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 (V.begin(), V.end(),           // source range  
           V.begin(),                     // destination range  
           [](int x){return -x;});        // operation  
// result: V = -2 -3 -5 -7
```

● Умножение на -10 и сохранение в другом контейнере

```
list<int> L(V.size());                    // ATTENTION to the size  
transform (V.begin(), V.end(),           // source range  
           L.begin(),                     // destination range  
           [](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 (V.begin(), V.end(),          // source range
           back_inserter(L),           // insert into the end of L
           [](int x){return x%3;});    // operation
// result: L= 20 30 50 70 -2 0 -2 -1
```

● Итератор вставки общего вида: `inserter()`

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

STL: generate() и iota()

- `generate(Iterator beg, Iterator end, Func op)`
Копирует элементы возвращаемые функцией `op()` в `[beg,end)`
- `iota(Iterator beg, Iterator end, T startValue)`
Копирует `startValue, startValue+1, startValue+2 ...` в `[beg,end)`

```
V.resize(5);  
generate(V.begin(), V.end(),           // source range  
         [](){return rand()%1000;} ); // operation  
// generated: V= 383 886 777 915 793
```

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

Алгоритмы сортировки

<code>sort()</code>	сортирует все элементы
<code>stable_sort()</code>	сортировка с сохранением порядка равных элементов
<code>partial_sort()</code>	сортирует первые <code>n</code> -элементов
<code>nth_element()</code>	ставит на нужное место <code>n</code> -й элемент
<code>partition()</code>	перемещает элементы удовлетворяющий условию в начало
<code>stable_partition()</code>	то же что <code>partition()</code> сохраняя относительный порядок элементов

👉 для `list` лучше использовать «собственный» алгоритм: `list.sort()`

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

```
// generated V= 383 886 777 915 793  
vector<int> V2(V);  
sort(V2.begin(),V2.end()); // sort V2 with operator<()  
// sort(<): V= 383 777 793 886 915
```

● Сортировка в убывающем порядке

```
sort(V.begin(),V.end(), // range  
     [](int x,int y){return x>y;}); // sorting lambda  
// sort(>): V= 915 886 793 777 383
```

● Отсортировать первые несколько элементов: `partial_sort()`

```
W = {3,8,3,6,7,9,1,5,10,6};  
// sort first 5-elements  
partial_sort(W.begin(),                // beginning of the range  
             W.begin()+5,              // end of sorted range  
             W.end(),                  // end of full range  
             [](int x,int y){return x>y;}); // sorting lambda  
cout << " sort(>) first 5el: W= " << W << endl;  
// sort(>) first 5el: W= 10 9 8 7 6 3 1 3 5 6
```

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

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

☞ элементы всегда вставляются **перед** указанным итератором