

STL算法

郭 炜 刘家瑛



北京大学



STL算法分类

- STL中的算法大致可以分为以下七类:
 - 不变序列算法
 - 变值算法
 - 删除算法
 - 变序算法
 - 排序算法
 - 有序区间算法
 - 数值算法

算法

大多重载的算法都是有**两个版本**的

- 用 “==” 判断元素是否相等, 或用 “<” 来比较大小
- 多出一个类型参数 “Pred” 和函数形参 “Pred op” :

通过**表达式 “op(x,y)” 的返回值**: ture/false

→判断x是否 “等于” y , 或者x是否 “小于” y

如下面的有两个版本的min_element:

iterator **min_element**(iterator first, iterator last);

iterator **min_element**(iterator first, iterator last, **Pred op**);



1. 不变序列算法

- 该类算法不会修改算法所作用的容器或对象
- 适用于顺序容器和关联容器
- 时间复杂度都是 $O(n)$

算法名称	功 能
min	求两个对象中较小的(可自定义比较器)
max	求两个对象中较大的(可自定义比较器)
min_element	求区间中的最小值(可自定义比较器)
max_element	求区间中的最大值(可自定义比较器)
for_each	对区间中的每个元素都做某种操作



1. 不变序列算法

算法名称	功 能
count	计算区间中等于某值的元素个数
count_if	计算区间中符合某种条件的元素个数
find	在区间中查找等于某值的元素
find_if	在区间中查找符合某条件的元素
find_end	在区间中查找另一个区间最后一次出现的位置(可自定义比较器)
find_first_of	在区间中查找第一个出现在另一个区间中的元素 (可自定义比较器)
adjacent_find	在区间中寻找第一次出现连续两个相等元素的位置(可自定义比较器)



1. 不变序列算法

算法名称	功 能
search	在区间中查找另一个区间第一次出现的位置(可自定义比较器)
search_n	在区间中查找第一次出现等于某值的连续n个元素(可自定义比较器)
equal	判断两区间是否相等(可自定义比较器)
mismatch	逐个比较两个区间的元素，返回第一次发生不相等的两个元素的位置(可自定义比较器)
lexicographical_compare	按字典序比较两个区间的大小(可自定义比较器)



find:

```
template<class InIt, class T>
```

```
InIt find(InIt first, InIt last, const T& val);
```

- 返回区间 [first,last) 中的迭代器 i ,使得 $*i == val$

find_if:

```
template<class InIt, class Pred>
```

```
InIt find_if(InIt first, InIt last, Pred pr);
```

- 返回区间 [first,last) 中的迭代器 i, 使得 $pr(*i) == true$



for_each:

```
template<class InIt, class Fun>
```

```
Fun for_each(InIt first, InIt last, Fun f);
```

- 对[first, last)中的每个元素e, 执行f(e), 要求 f(e)不能改变e



count:

```
template<class InIt, class T>
```

```
size_t count(InIt first, InIt last, const T& val);
```

- ▲ 计算[first, last) 中等于val的元素个数(x==y为true算等于)

count_if:

```
template<class InIt, class Pred>
```

```
size_t count_if(InIt first, InIt last, Pred pr);
```

- ▲ 计算[first, last) 中符合pr(e) == true 的元素e的个数



min_element:

```
template<class FwdIt>
```

```
FwdIt min_element(FwdIt first, FwdIt last);
```

- 返回[first,last) 中最小元素的迭代器, 以 “<” 作比较器
- 最小指没有元素比它小, 而不是它比别的不同元素都小
- 因为即便 $a \neq b$, $a < b$ 和 $b < a$ 有可能都不成立

max_element:

```
template<class FwdIt>
```

```
FwdIt max_element(FwdIt first, FwdIt last);
```

- 返回[first,last) 中最大元素(不小于任何其他元素)的迭代器
- 以 “<” 作比较器



```
#include <iostream>
#include <algorithm>
using namespace std;
class A {
    public:
        int n;
        A(int i):n(i) { }
};
bool operator<( const A & a1, const A & a2) {
    cout << "< called" << endl;
    if( a1.n == 3 && a2.n == 7 )
        return true;
    return false;
}
```



```
int main() {  
    A aa[] = { 3,5,7,2,1 };  
    cout << min_element(aa,aa+5)->n << endl;  
    cout << max_element(aa,aa+5)->n << endl;  
    return 0;  
}
```

输出：
< called
< called
< called
< called
3
< called
< called
< called
< called
7



2. 变值算法

- 此类算法会修改源区间或目标区间元素的值
- 值被修改的那个区间, 不可以是属于关联容器的

算法名称	功 能
for_each	对区间中的每个元素都做某种操作
copy	复制一个区间到别处
copy_backward	复制一个区间到别处, 但目标区前是从后往前被修改的
transform	将一个区间的元素变形后拷贝到另一个区间



2. 变值算法

算法名称	功 能
swap_ranges	交换两个区间内容
fill	用某个值填充区间
fill_n	用某个值替换区间中的n个元素
generate	用某个操作的结果填充区间
generate_n	用某个操作的结果替换区间中的n个元素
replace	将区间中的某个值替换为另一个值
replace_if	将区间中符合某种条件的值替换成另一个值
replace_copy	将一个区间拷贝到另一个区间，拷贝时某个值要换成新值拷过去
replace_copy_if	将一个区间拷贝到另一个区间，拷贝时符合某条件的值要换成新值拷过去



transform

```
template<class InIt, class OutIt, class Unop>
```

```
OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
```

- ▲ 对[first,last)中的每个迭代器l,
 - 执行 `uop(* l);` 并将结果依次放入从 x 开始的地方
 - 要求 `uop(* l)` 不得改变 * l 的值
- ▲ 本模板返回值是个迭代器, 即 `x + (last-first)`
 - x可以和 first相等



```
#include <vector>
#include <iostream>
#include <numeric>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;
class CLessThen9 {
    public:
        bool operator()( int n) { return n < 9; }
};
void outputSquare(int value ) { cout << value * value << " "; }
int calculateCube(int value) { return value * value * value; }
```




```
int main() {  
    const int SIZE = 10;  
    int a1[] = { 1,2,3,4,5,6,7,8,9,10 };  
    int a2[] = { 100,2,8,1,50,3,8,9,10,2 };  
    vector<int> v(a1,a1+SIZE);  
    ostream_iterator<int> output(cout, " ");  
    random_shuffle(v.begin(),v.end());  
    cout << endl << "1) ";  
    copy( v.begin(),v.end(),output);  
    copy( a2,a2+SIZE,v.begin());  
    cout << endl << "2 ) ";  
    cout << count(v.begin(),v.end(),8);  
    cout << endl << "3 ) ";  
    cout << count_if(v.begin(),v.end(),CLessThan9());  
}
```

输出:

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

2) 2

3) 6

//1) 是随机的



```
cout << endl << "4) ";  
cout << * (min_element(v.begin(), v.end()));  
cout << endl << "5) ";  
cout << * (max_element(v.begin(), v.end()));  
cout << endl << "6) ";  
cout << accumulate(v.begin(), v.end(), 0); //求和
```

输出:

4) 1

5) 100

6) 193



```
cout << endl << "7) ";  
for_each(v.begin(), v.end(), outputSquare);  
vector<int> cubes(SIZE);  
transform(a1, a1+SIZE, cubes.begin(), calculateCube);  
cout << endl << "8) ";  
copy(cubes.begin(), cubes.end(), output);  
return 0;  
}
```

输出：

7)10000 4 64 1 2500 9 64 81 100 4

8)1 8 27 64 125 216 343 512 729 1000



```
ostream_iterator<int> output(cout ,“ ”);
```

- 定义了一个 `ostream_iterator<int>` 对象,
可以通过`cout`输出以 “ ”(空格) 分隔的一个个整数

```
copy (v.begin(), v.end(), output);
```

- 导致`v`的内容在 `cout`上输出



copy 函数模板(算法)

```
template<class InIt, class OutIt>
```

```
OutIt copy(InIt first, InIt last, OutIt x);
```

- 本函数对每个在区间 $[0, \text{last} - \text{first})$ 中的 N 执行一次
 $*(x+N) = *(first + N)$, 返回 $x + N$

对于`copy(v.begin(),v.end(),output);`

- `first` 和 `last` 的类型是 `vector<int>::const_iterator`
- `output` 的类型是 `ostream_iterator<int>`



copy 的源代码:

```
template<class _II, class _OI>
inline _OI copy(_II _F, _II _L, _OI _X)
{
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```



```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main(){
    int a[4] = { 1,2,3,4 };
    My_ostream_iterator<int> oit(cout,"");
    copy(a,a+4,oit); //输出 1*2*3*4*
    ofstream oFile("test.txt", ios::out);
    My_ostream_iterator<int> oitf(oFile,"");
    copy(a,a+4,oitf); //向test.txt文件中写入 1*2*3*4*
    oFile.close();
    return 0;}
```

// 如何编写 My_ostream_iterator?



- copy 的源代码:

```
template<class _II, class _OI>
inline _OI copy(_II _F, _II _L, _OI _X){
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```

- 上面程序中调用语句“copy(a,a+4,oit)”实例化后得到copy如下:

```
My_ostream_iterator<int> copy(int * _F, int * _L, My_ostream_iterator<int> _X)
{
    for (; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return (_X);
}
```




- ▲ My_ostream_iterator类应该重载 “++” 和 “*” 运算符
- ▲ “=” 也应该被重载

```
#include <iterator>
```

```
template<class T>
```

```
class My_ostream_iterator:public iterator<output_iterator_tag, T>{
```

```
    private:
```

```
        string sep; //分隔符
```

```
        ostream & os;
```

```
    public:
```

```
        My_ostream_iterator(ostream & o, string s):sep(s), os(o){ }
```

```
        void operator ++() { }; // ++只需要有定义即可, 不需要做什么
```

```
        My_ostream_iterator & operator * () { return * this; }
```

```
        My_ostream_iterator & operator = ( const T & val)
```

```
        { os << val << sep; return * this; }
```

```
};
```



In-Video Quiz

1. find算法判断两个元素相等时，用哪个运算符？

A) == B) = C) < D) !=

2. 若要使下面这段程序编译能够通过，以下哪个表达式不是必须有定义的？

```
class A { };
```

```
A obj;
```

```
int a[] = {1,2,3,4,5};
```

```
copy(a,a+5,obj);
```

A) ++obj

B) -- obj

C) *obj

D) 以上三个都可以没定义