



# 程序设计与算法(一)

## C语言程序设计

郭 炜

微信公众号



微博: <http://weibo.com/guoweiofpku>

**学会程序和算法，走遍天下都不怕!**

讲义照片均为郭炜拍摄



北京大学  
PEKING UNIVERSITY

信息科学技术学院

指定教材：

## 《新标准C++程序设计教程》

郭炜 编著

清华大学出版社

重点大学计算机专业系列教材

## 新标准C++程序设计教程

郭炜 编著



清华大学出版社



## STL 初步(一)

# STL概述

- STL: (Standard Template Library) 标准模板库
- 包含一些常用的算法如排序查找，还有常用的数据结构如可变长数组、链表、字典等。
- 使用方便，效率较高
- 要使用其中的算法，需要`#include <algorithm>`



北京大学  
PEKING UNIVERSITY

信息科学技术学院

排序算法 sort



美国拱门国家公园平衡石

## 用sort进行排序(用法一)

- 对基本类型的数组从小到大排序:

**sort (数组名+n1, 数组名+n2) ;**

n1和n2都是int类型的表达式, 可以包含变量

如果n1=0,则 + n1可以不写

将数组中下标范围为 [n1, n2) 的元素从小到大排序。下标为n2的元素不在排序区间内

## 用sort进行排序(用法一)

```
int a[] = {15,4,3,9,7,2,6};  
sort(a,a+7); //对整个数组从小到大排序
```

```
int a[] = {15,4,3,9,7,2,6};  
sort(a,a+3); // 结果: {3,4,15,9,7,2,6}
```

```
int a[] = {15,4,3,9,7,2,6};  
sort(a+2,a+5); //结果: {15,4,3,7,9,2,6}
```

## 用sort进行排序(用法二)

- 对元素类型为T的基本类型数组从大到小排序:

`sort(数组名+n1, 数组名+n2, greater<T>()) ;`

```
int a[] = {15,4,3,9,7,2,6};
```

```
sort(a+1,a+4,greater<int>()); // 结果: {15,9,4,3,7,2,6}
```



## 用sort进行排序(用法三)

- 用自定义的排序规则，对任何类型T的数组排序

`sort(数组名+n1, 数组名+n2, 排序规则结构名());`

- 排序规则结构的定义方式:

```
struct 结构名
{
    bool operator() ( const T & a1, const T & a2) {
        //若a1应该在a2前面，则返回true。
        //否则返回false。
    }
};
```

# sort排序规则注意事项

struct 结构名

```
{  
    bool operator()( const T & a1,const T & a2) {  
        //若a1应该在a2前面, 则返回true。  
        //否则返回false。  
    }  
};
```

排序规则返回 true, 意味着 a1 必须在 a2 前面  
返回 false, 意味着 a1 并非必须在 a2 前面

排序规则的写法, 不能造成比较 a1,a2 返回 true 比较 a2,a1 也返回 true

否则sort会 runtime error

比较 a1,a2 返回 false 比较 a2,a1 也返回 false, 则没有问题

## 用sort进行排序(用法三)

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

struct Rule1 //按从大到小排序
{
    bool operator()( const int & a1,const int & a2) {
        return a1 > a2;
    }
};

struct Rule2 //按个数从小到大排序
{
    bool operator()( const int & a1,const int & a2) {
        return a1%10 < a2%10;
    }
};
```

## 用sort进行排序(用法三)

```
void Print(int a[],int size) {
    for(int i = 0;i < size;++i)
        cout << a[i] << "," ;
    cout << endl;
}

int main()
{
    int a[] = { 12,45,3,98,21,7};
    sort(a,a+sizeof(a)/sizeof(int)); //从小到大
    cout << "1) "; Print(a,sizeof(a)/sizeof(int));
    sort(a,a+sizeof(a)/sizeof(int),Rule1()); //从大到小
    cout << "2) "; Print(a,sizeof(a)/sizeof(int));
    sort(a,a+sizeof(a)/sizeof(int),Rule2()); //按个数从小到大
    cout << "3) "; Print(a,sizeof(a)/sizeof(int));
    return 0;
}
```

1) 3,7,12,21,45,98,  
2) 98,45,21,12,7,3,  
3) 21,12,3,45,7,98,

## 用sort对结构数组进行排序(用法三)

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
struct Student {
    char name[20];
    int id;
    double gpa;
};
Student students [] = {
    {"Jack", 112, 3.4}, {"Mary", 102, 3.8}, {"Mary", 117, 3.9},
    {"Ala", 333, 3.5}, {"Zero", 101, 4.0}};
```

## 用sort对结构数组进行排序(用法三)

```
struct StudentRule1 { //按姓名从小到大排
    bool operator() (const Student & s1,const Student & s2) {
        if( strcmp(s1.name,s2.name) < 0)
            return true;
        return false;
    }
};

struct StudentRule2 { //按id从小到大排
    bool operator() (const Student & s1,const Student & s2) {
        return s1.id < s2.id;
    }
};

struct StudentRule3 { //按gpa从高到低排
    bool operator() (const Student & s1,const Student & s2) {
        return s1.gpa > s2.gpa;
    }
};
```

## 用sort对结构数组进行排序(用法三)

```
void PrintStudents(Student s[],int size){  
    for(int i = 0;i < size;++i)  
        cout << "(" << s[i].name << ","  
                << s[i].id <<"," << s[i].gpa << ") " ;  
    cout << endl;  
}
```

## 用sort对结构数组进行排序(用法三)

```
int main()
{
    int n = sizeof(students) / sizeof(Student);
    sort(students, students+n, StudentRule1()); //按姓名从小到大排
    PrintStudents(students, n);
    sort(students, students+n, StudentRule2()); //按id从小到大排
    PrintStudents(students, n);
    sort(students, students+n, StudentRule3()); //按gpa从高到低排
    PrintStudents(students, n);
    return 0;
}
```

```
(Ala, 333, 3.5) (Jack, 112, 3.4) (Mary, 102, 3.8) (Mary, 117, 3.9) (Zero, 101, 4)
(Zero, 101, 4) (Mary, 102, 3.8) (Jack, 112, 3.4) (Mary, 117, 3.9) (Ala, 333, 3.5)
(Zero, 101, 4) (Mary, 117, 3.9) (Mary, 102, 3.8) (Ala, 333, 3.5) (Jack, 112, 3.4)
```





北京大学  
PEKING UNIVERSITY

信息科学技术学院

## 二分查找算法



美国拱门国家公园

# STL中的二分查找算法

- STL提供在排好序的数组上进行二分查找的算法

`binary_search`

`lower_bound`

`upper_bound`

## 用binary\_search进行二分查找（用法一）

- 在从小到大排好序的基本类型数组上进行二分查找

**binary\_search(数组名+n1, 数组名+n2, 值);**

n1和n2都是int类型的表达式，可以包含变量

如果n1=0,则 + n1可以不写

查找区间为下标范围为[n1, n2)的元素，下标为n2的元素不在查找区间内  
在该区间内查找"等于"值的元素，返回值为true(找到) 或false(没找到)

"等于"的含义： $a \text{ 等于 } B \iff a < b \text{ 和 } b < a \text{ 都不成立}$

## 用binary\_search进行二分查找（用法二）

- 在用自定义排序规则排好序的、元素为任意的T类型的数组中进行二分查找

**binary\_search(数组名+n1, 数组名+n2, 值, 排序规则结构名());**

n1和n2都是int类型的表达式，可以包含变量  
如果n1=0,则 + n1可以不写

查找区间为下标范围为[n1, n2)的元素，下标为n2的元素不在查找区间内  
在该区间内查找"等于"值的元素，返回值为true(找到) 或false(没找到)

查找时的排序规则，必须和排序时的规则一致！

"等于"的含义：  $a \text{ 等于 } b \iff "a \text{ 必须在 } b \text{ 前面}" \text{ 和 } "b \text{ 必须在 } a \text{ 前面}" \text{ 都不成立}$

## 用binary\_search进行二分查找（用法二）

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
struct Rule //按个数从小到大排
{
    bool operator()( const int & a1,const int & a2) {
        return a1%10 < a2%10;
    }
};
void Print(int a[],int size) {
    for(int i = 0;i < size;++i) {
        cout << a[i] << "," ;
    }
    cout << endl;
}
```

## 用binary\_search进行二分查找（用法二）

```
int main() {  
    int a[] = { 12,45,3,98,21,7};  
    sort(a,a+6);  
    Print(a,6);  
    cout <<"result:"<< binary_search(a,a+6,12) << endl;  
    cout <<"result:"<< binary_search(a,a+6,77) << endl;  
    sort(a,a+6,Rule()); //按个位数从小到大排  
    Print(a,6);  
    cout <<"result:"<< binary_search(a,a+6,7) << endl;  
    cout <<"result:"<< binary_search(a,a+6,8,Rule()) << endl;  
    return 0;  
}
```

"等于"的含义:  $a \text{ 等于 } b \iff$  "a必须在b前面"和"b必须在a前面"都不成立

```
3,7,12,21,45,98,  
result:1  
result:0  
21,12,3,45,7,98,  
result:0  
result:1
```

## 用binary\_search进行二分查找（用法二）

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
struct Student {
    char name[20];
    int id;
    double gpa;
};

Student students [] = {
    {"Jack", 112, 3.4}, {"Mary", 102, 3.8}, {"Mary", 117, 3.9},
    {"Ala", 333, 3.5}, {"Zero", 101, 4.0}};
```

## 用binary\_search进行二分查找（用法二）

```
struct StudentRule1 { //按姓名从小到大排
    bool operator() (const Student & s1,const Student & s2) {
        if( strcmp(s1.name,s2.name) < 0)
            return true;
        return false;
    }
};

struct StudentRule2 { //按id从小到大排
    bool operator() (const Student & s1,const Student & s2) {
        return s1.id < s2.id;
    }
};

struct StudentRule3 { //按gpa从高到低排
    bool operator() (const Student & s1,const Student & s2) {
        return s1.gpa > s2.gpa;
    }
};
```



## 用binary\_search进行二分查找（用法二）

```
int main(){
    Student s;
    strcpy(s.name, "Mary");
    s.id= 117;
    s.gpa = 0;
    int n = sizeof(students) / sizeof(Student);
    sort(students, students+n, StudentRule1()); //按姓名从小到大排
    cout << binary_search( students , students+n, s,
                           StudentRule1()) << endl;

    strcpy(s.name, "Bob");
    cout << binary_search( students , students+n, s,
                           StudentRule1()) << endl;

    sort(students, students+n, StudentRule2()); //按id从小到大排
    cout << binary_search( students , students+n, s,
                           StudentRule2()) << endl;

    return 0;
}
```

## 用lower\_bound二分查找下界(用法一)

- 在对元素类型为T的从小到大排好序的基本类型的数组中进行查找

```
T * lower_bound(数组名+n1, 数组名+n2, 值);
```

返回一个指针 T \* p;

\*p 是查找区间里下标最小的, 大于等于"值" 的元素。如果找不到, p指向下标为n2的元素

## 用lower\_bound二分查找下界(用法二)

- 在元素为任意的T类型、按照自定义排序规则排好序的数组中进行查找

`T * lower_bound(数组名+n1, 数组名+n2, 值, 排序规则结构名());`

返回一个指针 `T * p;`

\*p 是查找区间里下标最小的，按自定义排序规则，**可以排在"值"后面的元素**。如果找不到，p指向下标为n2的元素

## 用upper\_bound二分查找上界(用法一)

- 在元素类型为T的从小到大排好序的基本类型的数组中进行查找

`T * upper_bound(数组名+n1, 数组名+n2, 值);`

返回一个指针 `T * p;`

\*p 是查找区间里下标最小的, 大于"值"的元素。如果找不到, p指向下标为n2的元素

## 用upper\_bound二分查找上界(用法二)

- 在元素为任意的T类型、按照自定义排序规则排好序的数组中进行查找

`T * upper_bound(数组名+n1, 数组名+n2, 值, 排序规则结构名());`

返回一个指针 `T * p;`

\*p 是查找区间里下标最小的，按自定义排序规则，**必须排在"值"后面的元素**。如果找不到，p指向下标为n2的元素

## lower\_bound,upper\_bound用法示例

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
struct Rule
{
    bool operator()( const int & a1,const int & a2) {
        return a1%10 < a2%10;
    }
};
void Print(int a[],int size) {
    for(int i = 0;i < size;++i) {
        cout << a[i] << "," ;
    }
    cout << endl;
}
```

## lower\_bound,upper\_bound用法示例

```
#define NUM 7
int main()
{
    int a[NUM] = { 12,5,3,5,98,21,7};
    sort(a,a+NUM);
    Print(a,NUM); // => 3,5,5,7,12,21,98,
    int * p = lower_bound(a,a+NUM,5);
    cout << *p << ", " << p-a << endl; //=> 5,1
    p = upper_bound(a,a+NUM,5);
    cout << *p << endl; //=>7
    cout << * upper_bound(a,a+NUM,13) << endl; //=>21
}
```

## lower\_bound,upper\_bound用法示例

```
sort(a,a+NUM,Rule());  
Print(a,NUM); //=>21,12,3,5,5,7,98,  
cout << * lower_bound(a,a+NUM,16,Rule()) << endl; // => 7  
cout << lower_bound(a,a+NUM,25,Rule()) - a<< endl; // => 3  
cout << upper_bound(a,a+NUM,18,Rule()) - a << endl; // => 7  
if( upper_bound(a,a+NUM,18,Rule()) == a+NUM)  
    cout << "not found" << endl; //=> not found  
cout << * upper_bound(a,a+NUM,5,Rule()) << endl; // =>7  
cout << * upper_bound(a,a+NUM,4,Rule()) << endl; // =>5  
return 0;  
}
```





北京大学  
PEKING UNIVERSITY

信息科学技术学院

## STL中的 平衡二叉树



美国拱门国家公园

## STL中的平衡二叉树数据结构

- 有时需要在大量增加、删除数据的同时，还要进行大量数据的查找

## STL中的平衡二叉树数据结构

- 有时需要在大量增加、删除数据的同时，还要进行大量数据的查找
- 希望增加数据、删除数据、查找数据都能在  $\log(n)$  复杂度完成

## STL中的平衡二叉树数据结构

- 有时需要在大量增加、删除数据的同时，还要进行大量数据的查找
- 希望增加数据、删除数据、查找数据都能在  $\log(n)$  复杂度完成
- 排序+二分查找显然不可以，因加入新数据就要重新排序

# STL中的平衡二叉树数据结构

- 有时需要在大量增加、删除数据的同时，还要进行大量数据的查找
- 希望增加数据、删除数据、查找数据都能在  $\log(n)$  复杂度完成
- 排序+二分查找显然不可以，因加入新数据就要重新排序
- 可以使用“平衡二叉树”数据结构存放数据，体现在STL中，就是以下四种“排序容器”：

`multiset`     `set`     `multimap`     `map`



北京大学  
PEKING UNIVERSITY

信息科学技术学院

multiset



美国拱门国家公园

# multiset用法

```
multiset<T> st;
```

- 定义了一个multiset变量st，st里面可以存放T类型的数据，并且能自动排序。开始st为空
- 排序规则：表达式 “a < b” 为true，则 a 排在 b 前面
- 可用 `st.insert`添加元素，`st.find`查找元素，`st.erase`删除元素，复杂度都是  $\log(n)$

## multiset 用法

```
#include <iostream>
#include <cstring>
#include <set> //使用multiset和set需要此头文件
using namespace std;
int main()
{
    multiset<int> st;
    int a[10]={1,14,12,13,7,13,21,19,8,8 };
    for(int i = 0;i < 10; ++i)
        st.insert(a[i]); //插入的是a [i]的复制品
    multiset<int>::iterator i; //迭代器, 近似于指针
    for(i = st.begin(); i != st.end(); ++i)
        cout << * i << ", ";
    cout << endl;
```

输出: 1,7,8,8,12,13,13,14,19,21,



## multiset 用法

```
i = st.find(22);    //查找22, 返回值是迭代器
if( i == st.end()) //找不到则返回值为 end()
    cout << "not found" << endl;
st.insert(22);    //插入 22
i = st.find(22);
if( i == st.end())
    cout << "not found" << endl;
else
    cout << "found:" << *i << endl;
//找到则返回指向找到的元素的迭代器
```

输出:

not found

found:22

```

    i = st.lower_bound(13);
//返回最靠后的迭代器 it, 使得[begin(),it) 中的元素
//都在 13 前面 , 复杂度 log(n)
    cout << * i << endl;
    i = st.upper_bound(8);
//返回最靠前的迭代器 it, 使得[it,end()) 中的元素
//都在 8 后面, 复杂度 log(n)
    cout << * i << endl;
    st.erase(i); //删除迭代器 i 指向的元素, 即12
    for(i = st.begin(); i != st.end(); ++i)
        cout << * i << ", ";
    return 0;
}

```

1, 7, 8, 8, 12, 13, 13, 14, 19, 21,

输出:

13

12

1, 7, 8, 8, 13, 13, 14, 19, 21, 22,

## multiset 上的迭代器

```
multiset<T>::iterator p;
```

➤ p是迭代器，相当于指针，可用于指向multiset中的元素。访问multiset中的元素要通过迭代器。

➤ 与指针的不同：

multiset上的迭代器可 ++ , --, 用 != 和 == 比较，不可比大小，不可加减整数，不可相减

## multiset 上的迭代器

```
multiset<T> st;
```

➤ `st.begin()` 返回值类型为 `multiset<T>::iterator`,  
是指向 `st` 中的头一个元素的迭代器

➤ `st.end()` 返回值类型为 `multiset<T>::iterator`,  
是指向 `st` 中的最后一个元素后面的迭代器

➤ 对迭代器 `++`, 其就指向容器中下一个元素, `--` 则令其指向上一个元素

# 自定义排序规则的multiset 用法

```
#include <iostream>
#include <cstring>
#include <set>
using namespace std;

struct Rule1 {
    bool operator()( const int & a,const int & b)      {
        return (a%10) < (b%10);
    } //返回值为true则说明a必须排在b前面
};

int main()  {
    multiset<int,greater<int> > st; //排序规则为从大到小
    int a[10]={1,14,12,13,7,13,21,19,8,8 };
    for(int i = 0;i < 10; ++i)
        st.insert(a[i]);
    multiset<int,greater<int> >::iterator i;
    for(i = st.begin(); i != st.end(); ++i)
        cout << * i << ", ";
    cout << endl;
```

输出：21,19,14,13,13,12,8,8,7,1,

## 自定义排序规则的multiset 用法

```
multiset<int,Rule1 > st2;  
//st2的元素排序规则为: 个位数小的排前面  
for(int i = 0;i < 10; ++i)  
    st2.insert(a[i]);  
multiset<int,Rule1>::iterator p;  
for(p = st2.begin(); p != st2.end(); ++p)  
    cout << * p << ",";  
cout << endl;  
p = st2.find(133);  
cout << * p << endl;  
  
return 0;  
}
```

输出:

1,21,12,13,13,14,7,8,8,19,  
13

## 自定义排序规则的multiset 用法

```
multiset<int, Rule1 > st2;  
//st2的元素排序规则为: 个位数小的排前面  
for(int i = 0; i < 10; ++i)  
    st2.insert(a[i]);  
multiset<int, Rule1>::iterator p;  
for(p = st2.begin(); p != st2.end(); ++p)  
    cout << * p << ", ";  
cout << endl;  
p = st2.find(133);  
cout << * p << endl;  
  
return 0;  
}
```

输出:

1, 21, 12, 13, 13, 14, 7, 8, 8, 19,  
13

find(x): 在排序容器中找一个元素y, 使得  
“x必须排在y前面” 和 “y必须排在x前面”  
都不成立

# 自定义排序规则的multiset 用法

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <set>

using namespace std;

struct Student {
    char name[20];
    int id;
    int score;
};

Student students [] = { {"Jack",112,78}, {"Mary",102,85},
    {"Ala",333,92}, {"Zero",101,70}, {"Cindy",102,78}};

struct Rule {
    bool operator() (const Student & s1,const Student & s2) {
        if( s1.score != s2.score)  return s1.score > s2.score;
        else    return (strcmp(s1.name,s2.name) < 0);
    }
};
```



# 自定义排序规则的multiset 用法

```
int main()
{
    multiset<Student,Rule> st;
    for(int i = 0;i < 5;++i)
        st.insert(students[i]); //插入的是students[i]的复制品
    multiset<Student,Rule>::iterator p;
    for(p = st.begin(); p != st.end(); ++p)
        cout << p->score <<" "<<p->name<<" "
            << p->id <<endl;
    Student s = { "Mary",1000,85};
    p = st.find(s);
    if( p!= st.end())
        cout << p->score <<" "<< p->name<<" "
            << p->id <<endl;
    return 0;
}
```

92 Ala 333  
85 Mary 102  
78 Cindy 102  
78 Jack 112  
70 Zero 101  
85 Mary 102



北京大学  
PEKING UNIVERSITY

信息科学技术学院

set



美国拱门国家公园

## set的用法

➤ set和multiset的区别在于容器里不能有重复元素

a和b重复  $\Leftrightarrow$  “a必须排在b前面” 和 “b必须排在a前面” 都不成立

➤ set插入元素可能不成功

# set的用法

```
#include <iostream>
#include <cstring>
#include <set>
using namespace std;
int main()
{
    set<int> st;
    int a[10] = { 1,2,3,8,7,7,5,6,8,12 };
    for(int i = 0;i < 10; ++i)
        st.insert(a[i]);
    cout << st.size() << endl;    //输出: 8
    set<int>::iterator i;
    for(i = st.begin(); i != st.end(); ++i)
        cout << * i << ",";    //输出: 1,2,3,5,6,7,8,12,
    cout << endl;
```

## set的用法

```
pair<set<int>::iterator, bool> result = st.insert(2);  
if( ! result.second ) //条件成立说明插入不成功  
    cout << * result.first << " already exists."  
    << endl;  
else  
    cout << * result.first << " inserted." << endl;  
return 0;  
}
```

输出:

2 already exists.

```
pair<set<int>::iterator, bool>
```

⇔

```
struct {  
    set<int>::iterator first;  
    bool second;  
};
```

# pair模板的用法

`pair<T1, T2>`类型等价于:

```
struct {  
    T1 first;  
    T2 second;  
};
```

例如: `pair<int, double> a;`  
等价于:

```
struct {  
    int first;  
    double second;  
} a;  
a.first = 1;  
a.double = 93.93;
```