

C++ map 中的迭代器加数字

C++ string 中 find 函数的用法

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4.
5. int main(int argc, char* argv[])
6. {
7.     string pp = "abcd";
8.     int num1 = pp.find('b');//num1 == 1
9.     int num2 = pp.find('e');//num2 == -1
10.    system("pause");
11. }
```

查找成功返回字符在字符串中出现的位置，不成功返回-1

C++ 将向量的内容翻转

```
1. vector<int> vec1, vec2;
2. vec2 = vec1;
3. reverse(vec2.begin(), vec2.end());
```

向量 vec2 是向量 vec1 内容的翻转

C++ 链表的节点一般都是手动分配，如果是局部变量会有问题

```
1. while (l1 || l2)
2. {
3.     digits = 0;
4.     if (l1 != NULL)
5.     {
6.         digits += l1->val;
7.         l1 = l1->next;
8.     }
9.     if (l2 != NULL)
10.    {
11.        digits += l2->val;
```

```

12.         l2 = l2->next;
13.     }
14.     int tt = 2;
15.     digits += flag;
16.     /**这种用法错误**/
17.     ListNode t(digits % 10);
18.     p->next = &t;
19.     /**这种用法正确**/
20.     //p->next = new ListNode(digits % 10);
21.     flag = digits / 10;
22.     p = p->next;
23. }

```

节点手动分配内存存在堆上，一个点指向另一个点。如果是局部变量则指向这个节点的指针

应该在下次用不起来，因为局部变量的是在栈上，注意生存周期。

C++ 模板不能分离编译模式

```

1. #include <iostream>
2. #include <string>
3. #include "test.h"
4. using namespace std;
5.
6. int main(int argc, char* argv[])
7. {
8.     Solution<string> pp;
9.     pp.ReadData("testSet.txt");
10.    system("pause");
11. }

```

test.h

```

1. #ifndef TEST_H
2. #define TEST_H
3.
4. #include <vector>
5. using namespace std;
6.
7. template<typename T1>
8. class Solution
9. {

```

```

10. public:
11.     vector< vector<T1> > ReadData(const char *filename);
12. };
13.
14. #endif

```

test.cpp

```

1. #include <iostream>
2. #include <fstream>
3. #include <string>
4. #include "test.h"
5. using namespace std;
6.
7. template<typename T1>
8. vector< vector<T1> > Solution<T1>::ReadData(const char *filename)
9. {
10.     vector< vector<T1> > result;
11.     ifstream fin;
12.     fin.open(filename);
13.     vector<T1> row;
14.     int count = 0;
15.     while(!fin.eof())
16.     {
17.         T1 temp;
18.         fin>>temp;
19.         count++;
20.         row.push_back(temp);
21.         if(count == 3)
22.         {
23.             count = 0;
24.             result.push_back(row);
25.             row.clear();
26.         }
27.     }
28.     return result;
29. }
30.
31. template
32. vector< vector<string> > Solution<string>::ReadData(const char *filename);

```

C++ vector 从另一个 vector 中拷贝一部分

C++ switch 语句只接受常量

1、正确的写法

```
1. template<typename T>
2. Status arithmetic<T>::JudgeOperator(const string A)
3. {
4.     //switch 语句只接受常量
5.     switch(A[0])
6.     {
7.         case '+':
8.         case '-':
9.         case '*':
10.        case '/': return true;
11.        default: return false;
12.    }
13. }
```

2、错误的写法

```
1. template<typename T>
2. Status arithmetic<T>::JudgeOperator(const string A)
3. {
4.     switch(A)
5.     {
6.         case "+":
7.         case "-":
8.         case "*":
9.         case "/": return true;
10.        default: return false;
11.    }
12. }
```

C++ map 取 value

C++ 结构体中能否有函数

C++ unorderedmap、unorderedmultimap 的使用

C++ 默认参数

默认参数只需在函数声明的时候写上，在函数定义的时候不能用，否则会编译报错

C++ const 对象只能调用 const 成员函数

1、例子：

```
class A
{
    int value;
public:
    int getValue() {return value;};
    void setValue(int a){value=a;};
}

A myA;
/.... 一些操作 ..../

int test(const A tmpA)
{
    int i=tmpA.getValue();
    return i;
}
```

在 `int i=tmpA.getValue()`处，会发生编译错误，大体提示为：

XXX 的‘this’实参时丢弃了类型限定

因为对象 `tmpA` 被 `const` 修饰了，它只能调用 `const` 成员函数。

解决办法：要么把 `tmpA` 对象的 `const` 修饰去掉，要么在函数 `getValue` 的声明和定义处加上 `const` 修饰。

2、昨天在写一个 `HashTable` 的时候，出现了一个小错误，大概如下：

```
1. class HashTable {
2.     public:
3.         function_1() const {
4.             function_2(function_3());
5.         }
6.         function_2() {}
7.         function_3() {}
8.     };
```

类似这样的代码编译以后，会出现一个错误：将 const HashTable 做为

HashTable::function_2()的 this 实参时丢弃了类型限定。

将 const HashTable 做为 HashTable::function_3()的 this 实参时丢弃了类型限定。

后来在网上查阅，发现了出现这个错误的原因：因为 function_1()是一个 const 成员函数，

所以 function_1()的 this 指针是 const 指针，所以这个 this 指针指向的是一个 const 的

HashTable 的对象，而 **const 对象只能调用 const 成员函数**，而 function_2()和

function_3()不是 const 成员函数，自然就无法调用了。

相当于：

```
1. function_1() {  
2.     this->function_2(this->function_3());  
3. }
```

C++ 删除向量中相同的元素

```
sort( a.begin(), a.end() );  
a.erase( unique( a.begin(), a.end() ), a.end() );
```

C++ 找到某个值的出现位置

find, find_if, find_first_of 都是头文件 algorithm 中的函数，

map 容器提供了两个操作：count 和 find，用于检查某个键是否存在而不会插入该键。

m.count(k) 返回 m 中 k 的出现次数

m.find(k) 如果 m 容器中存在按 k 索引的元素，则返回指向该元素的迭代器。如果不存在，则返回超出末端迭代器

对于 map 对象，count 成员的返回值只能是 0 或 1。map 容器只允许一个键对应一个实例，所以 count 可有效地表明一个键是否存在。

而对于 multimaps 容器，count 的返回值将有更多的用途。

如果返回值非 0，则可以使用下标操作符来获取该键所关联的值，而不必担心这样做会在

map 中插入新元素：

```
int occurs = 0;
if (word_count.count("foobar"))
    occurs = word_count["foobar"];
```

当然，在执行 count 后再使用下标操作符，实际上是对元素作了两次查找。（count 查找

和下标查找）如果希望当元素存在时就使用它，则应该用 find 操作。

find 操作返回指向元素的迭代器，如果元素不存在，则返回 end 迭代器：

```
int occurs = 0;
map<string,int>::iterator it = word_count.find("foobar");
if (it != word_count.end())
    occurs = it->second;
```

如果希望当具有指定键的元素存在时，就获取该元素的引用，否则就不在容器中

创建新元素，那么应该使用 find。

1、find 运算

假设有一个 int 型的 vector 对象，名为 vec，我们想知道其中是否包含某个特定值。解决这

个问题最简单的方法是使用标准库提供的 find 运算：

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm> //包含通用算法
4. using namespace std;
5.
6. int main(int argc, char* argv[])
7. {
8.
9.     vector<int> scores;
10.    scores.push_back(100);
11.    scores.push_back(80);
12.    scores.push_back(45);
13.    scores.push_back(75);
14.    scores.push_back(99);
15.    scores.push_back(100);
16.    int search_value = 45;
```

```

17.     vector<int>::iterator it = find(scores.begin(), scores.end(), search_val
    ue); //找到 search_value 出现的位置
18.     cout<<"The value "<<search_value
19.         <<(it == scores.end() ? " is not present" : " is present")
20.         <<endl;
21.     system("pause");
22. }

```

类似地，由于指针的行为与作用在内置数组上的迭代器一样，因此也可以使用 `find` 来搜索

数组：

```

1int ia[6] = {27 , 210 , 12 , 47 , 109 , 83};
2int search_value = 83;
3int *result = find(ia , ia + 6 , search_value);
4cout<<"The value "<<search_value
5    <<(result == ia + 6 ? " is not present" : "is present")
6    <<endl;

```

如果需要传递一个子区间，则传递指向这个子区间的第一个元素以及最后一个元素的下一位置

的迭代器（或指针）。

例如，在下面对 `find` 函数的调用中，只搜索了 `ia[1]` 和 `ia[2]`：

```

//only search elements ia[1] and ia[2]
int *result = find(ia + 1 , ia
+ 3 , search_value);

```

2、find_if 运算

find_if 算法 是 `find` 的一个谓词判断版本，它利用返回布尔值的谓词判断 `pred`，检查迭代器

区间 `[first, last)` 上的每一个元素，如果迭代器 `iter` 满足 `pred(*iter) == true`，表示找到元素并

返回迭代器值 `iter`；未找到元素，则返回 `last`。

`find_if`：在序列中找符合某谓词的第一个元素。

```

1. #include <iostream>
2. #include <vector>
3. #include <algorithm> //包含通用算法
4. using namespace std;
5.
6. bool choose(int num)

```



```

7. {
8.     return num % 9 == 0;
9. }
10.
11. int main(int argc, char* argv[])
12. {
13.     vector<int> scores;
14.     scores.push_back(100);
15.     scores.push_back(80);
16.     scores.push_back(45);
17.     scores.push_back(75);
18.     scores.push_back(99);
19.     scores.push_back(100);
20.
21.     vector<int>::iterator it = find_if(scores.begin(), scores.end(), choose)
        ;//找到被 9 整除出现的位置
22.     cout<<"The value can / 9 "
23.         <<(it == scores.end() ? " is not present" : " is present")
24.         <<endl;
25.     system("pause");
26. }

```

3、find_first_of 运算

除了 find 之外，标准库还定义了一些更复杂的查找算法。当中的一部分类似 string 类的 find 操作，其中一个就是 find_first_of 函数。

这个算法带有**两对迭代器参数**来标记两端元素范围：**第一段范围内查找与第二段范围中任意元素匹配的元素，然后返回一个迭代器，指向第一个匹配的元素。如果找不到匹配元素，则返回第一个范围的 end 迭代器。**

假设 roster1 和 roster2 是两个存放名字的 list 对象，可使用 find_first_of **统计有多少个名字同时出现在这两个列表中：**

```

1. #include <iostream>
2. #include <vector>
3. #include <algorithm> //包含通用算法
4. using namespace std;
5.

```

```

6.  int main(int argc, char* argv[])
7.  {
8.      vector<int> roster1;
9.      roster1.push_back(100);
10.     roster1.push_back(80);
11.     roster1.push_back(45);
12.     roster1.push_back(75);
13.     roster1.push_back(99);
14.     roster1.push_back(100);
15.
16.     vector<int> roster2;
17.     roster2.push_back(83);
18.     roster2.push_back(67);
19.     roster2.push_back(45);
20.     roster2.push_back(63);
21.     roster2.push_back(99);
22.
23.     vector<int>::iterator it = roster1.begin();
24.     while((it = find_first_of(it, roster1.end(), roster2.begin(), roster2.end())) != roster1.end())
25.     {
26.         cout<<*it<<" ";
27.         it++;
28.     }
29.
30.     system("pause");
31. }

```

调用 `find_first_of` 查找 `roster2` 中的每个元素是否与第一个范围内的元素匹配，也就是在 `it` 到 `roster1.end()` 范围内查找一个元素。该函数返回此范围内第一个同时存在于第二个范围内的元素。在 `while` 的第一次循环中，遍历整个 `roster1` 范围。第二次以及后续的循环迭代则只考虑 `roster1` 中尚未匹配的部分。

循环条件检查 `find_first_of` 的返回值，判断是否找到匹配的名字。如果找到一个匹配，则使计数器加 1，同时给 `it` 加 1，使它指向 `roster1` 中的下一个元素。很明显可知，当不再有任何匹配时，`find_first_of` 返回 `roster1.end()`，完成统计。

find_first_of，带有两对迭代器参数。每对迭代器中，两个参数的类型必须精确匹配，但不要求两对之间的类型匹配。特别是，元素可存储在不同类型的序列中，只要这两个序列的元素可以比较即可。

在上述程序中，roster1 和 roster2 的类型不必精确匹配：roster1 可以使 list 对象，而 roster2 则可以使 vector 对象、 deque 对象或者是其他后面要学到的序列。只要这两个序列的元素可使用相等 (==) 操作符进行比较即可。如果 roster1 是 list< string>对象，则 roster2 可以使 vector<char*>对象，因为 string 标准库为 string 对象与 char* 对象定义了相等 (==) 操作符。

C++ 统计 vector 向量中指定元素出现的次数

1、利用 STL 通用算法统计 vector 向量中某个元素出现的次数：**count()**算法统计等于某个值的对象的个数。

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm> //包含通用算法
4. using namespace std;
5.
6. int main(int argc, char* argv[])
7. {
8.
9.     vector<int> scores;
10.    scores.push_back(100);
11.    scores.push_back(80);
12.    scores.push_back(45);
13.    scores.push_back(75);
14.    scores.push_back(99);
15.    scores.push_back(100);
16.    int num = 0;
17.    num= count(scores.begin(),scores.end(),100);//统计 100 出现的次数
18.    cout<<"times: "<<num<<endl;
19.    return 0;
20. }
```

2、使用 `count_if` 算法计算中的元素范围 `[first, last)` , 返回满足条件的元素的数量。

计算其中首字符是 'a' 的个数。

```
1. #include <iostream>
2. #include <string>
3. #include <vector>
4. #include <algorithm>
5. using namespace std;
6.
7. bool count_by_self(const string temp)
8. {
9.     return temp[0] == 'a';
10. }
11.
12. int main()
13. {
14.     vector<string> object;
15.
16.     object.push_back("afhj");
17.     object.push_back("dddfhj");
18.     object.push_back("aywmdf");
19.     object.push_back("iskfgdnej");
20.     object.push_back("augtedd");
21.
22.     int num = count_if(object.begin(), object.end(), count_by_self);
23.
24.     cout<<num<<endl;
25.
26.     system("pause");
27. }
```

C++ map 排序 (按照 value 值排序)

http://www.csdn123.com/html/itweb/20130922/130892_130908_130900.htm

```
1. bool cmp_by_value(const pair<string, int> &lhs, const pair<string, int> &rhs
2. )//定义的比较函数, 使得 sort 函数对 vector 中的 pair 对象按照 value 排序
3. {
4.     return lhs.second < rhs.second;
5. }
```

```

6. string majorityCnt(vector<string> classList)//如果没有特征可用于分类，则判断其中哪一类最多，将样本归为那一类
7. {
8.     map<string, int> classCount;
9.     for(int i = 0; i < classList.size(); i++)//统计各类出现的此处
10.    {
11.        classCount[classList[i]]++;
12.    }
13.
14.    vector<pair<string, int>> class_count(classCount.begin(), classCount.end());
15.    sort(class_count.begin(), class_count.end(), cmp_by_value);//按照 value 进行排序
16.    return class_count[0].first;
17. }

```

C++ 使用 ofstream 打开 txt 写入数据，每次打开 txt 都会使得以前的数据被清除掉，使用 in.open("Text.txt",ios_base::app);可以接着以前的数据写下去.

C++ map value 和 key 互换

C++ multimap 的使用

使用头文件 map

声明

```

1. multimap<string,string> authors;

```

元素的添加

```

1. //adds first element with key Barth
2. authors.insert(make_pair(string("Barth, John"),string("Sot-Weed Factor")));
3. //ok: adds second element with keyBarth
4. //每次调用 insert 总会添加一个元素，一个键可以对应多个实例，这些实例在容器中相邻存放
5. authors.insert(make_pair(string("Barth, John"),string("Lost in the Funhouse")));

```

元素的删除

```

1. string search_item("Kazuo Ishiguro");
2. //erase all elements with this key; returns number of elements removed
3. //删除该键所有元素并且返回删除元素的个数
4. multimap<string,string>::size_type cnt = authors.erase(search_item);

```

查找元素

方法 1： find count

```

1. //author we'll look for
2. string search_item("Alain de Botton");
3. //how many entries are there for this author
4. typedef multimap<string,string>::size_type sz_type;
5. sz_type entries = author.count(search_item);
6. //get iterator to the first entry for this author
7. multimap<string,string>::iterator iter = author.find(search_item);
8. //loop through the number of entries there are for this author
9. for(sz_type cnt = 0; cnt < entries; ++cnt, ++iter)
10. cout << iter->second << endl; //print each title

```

首先调用 count 确定某作者所写的书籍数目，然后调用 find 获得指向第一个该键所关联的元素的迭代器。for 循环迭代的次数依赖于 count 返回的值。在特殊情况下，如果 count 返回 0 值，则该循环永不执行。

方法 2： lower_bound upper_bound

[cpp] [view plain copy](#)

```

1. //definitions of authors and search_item as above
2. //beg and end denote range of elements for this author
3. typedef multimap<string,string>::iterator authors_it;
4. authors_it beg = authors.lower_bound(search_item),
5.           end = authors.upper_bound(search_item);
6. //loop through the number of entries there are for this author
7. while(beg != end){
8. cout << beg->second << endl; //print each title
9. ++beg;
10. }

```

在同一个键上调用 `lower_bound` 和 `upper_bound`，将产生一个迭代器范围，指示出该键所关联的所有元素。如果该键在容器中存在，则会获得两不同的迭代器：`lower_bound` 返回的迭代器指向该键关联的第一个实例，而 `upper_bound` 返回的迭代器则指向最后一个实例的下一位置。如果该键不在 `multimap` 中，这两个操作将返回同一个迭代器，指向依据元素的排列顺序该键应该插入的位置。

方法 3：`equal_range`

[cpp] [view plaincopy](#)

```
1. //definitions of authors and search_item as above
2. //pos holds iterators that denote range of elements of this key
3. pair<authors_it,authors_it> pos = authors.equal_range(search_item);
4. //loop through the number of entries there are for this author
5. while(pos.first != pos.second){
6.     cout<<pos.first->second<<endl; //print each title
7.     ++pos.first;
8. }
```

`pair` 对象的 `first` 成员存储 `lower_bound` 函数返回的迭代器，而 `second` 成员则记录

`upper_bound` 函数返回的迭代器。

leetcode 题目使用 `multimap` 来检测单词数组中的回文单词

For example:

Input: ["tea", "and", "ate", "eat", "den"]

Output: ["tea", "ate", "eat"]

```
1. #include <iostream>
2. #include <vector>
3. #include <map>
4. #include <set>
5. #include <utility> // make_pair()
6. #include <string>
7. #include <algorithm>
8. using namespace std;
9.
10. class Solution {
```

```

11. public:
12.     vector<string> anagrams(vector<string> &strs)
13.     {
14.         vector<string> result;
15.
16.         set<string> dif_key;
17.         multimap<string, int> key_num;
18.         for(int i = 0; i < strs.size(); i++)
19.         {
20.             string temp = strs[i];
21.             sort(temp.begin(), temp.end());
22.             dif_key.insert(temp); //存储所有不重复的键值
23.             key_num.insert(make_pair(temp, i)); //将所有键值和它的序号对应起来
24.         }
25.
26.         set<string>::iterator it;
27.         for(it = dif_key.begin(); it != dif_key.end(); it++)
28.         {
29.             /*****方法 1，使用 count 和 find*****/
30.             //multimap<string, int>::size_type num = key_num.count(*it);
31.             //if(num > 1)//有多个键值相同的证明它是 Anagrams 单词
32.             //{
33.             //    multimap<string, int>::iterator ip = key_num.find(*it);
34.             //    for(int j = 0; j < num; j++, ip++)
35.             //    {
36.             //        result.push_back(strs[ip->second]);
37.             //    }
38.             //}
39.
40.             /*****方法 1，使用 lower_bound 和 upper_bound*****/
41.             //multimap<string, int>::size_type num = key_num.count(*it);
42.             //if(num > 1)//有多个键值相同的证明它是 Anagrams 单词
43.             //{
44.             //    multimap<string, int>::iterator im = key_num.lower_bound(*it);
45.             //    multimap<string, int>::iterator in = key_num.upper_bound(*it);
46.             //    while(im != in)
47.             //    {
48.             //        result.push_back(strs[im->second]);
49.             //        im++;
50.             //    }
51.             //}
52.
53.             /*****方法 1，使用 equal_range*****/
54.             multimap<string, int>::size_type num = key_num.count(*it);

```



```

55.         if(num > 1)//有多个键值相同的证明它是 Anagrams 单词
56.         {
57.             pair<multimap<string, int>::iterator, multimap<string, int>::iterator> pos = k
               ey_num.equal_range(*it);
58.             while(pos.first != pos.second)
59.             {
60.                 result.push_back(strs[pos.first->second]);
61.                 pos.first++;
62.             }
63.         }
64.     }
65.     return result;
66. }
67. };
68.
69. int main(int argc, char* argv[])
70. {
71.     Solution p;
72.     vector<string> str;
73.     str.push_back("tea");
74.     str.push_back("and");
75.     str.push_back("ate");
76.     str.push_back("den");
77.     str.push_back("eat");
78.     p.anagrams(str);
79.     system("pause");
80. }

```

C++ pair、map、set 获取其中的元素

(1) 获取 pair 中的元素

```

1. pair<string, int> temp("James", 3);
2. temp.first == "James";
3. temp.second == 3;

```

(2) 获取 map 中的元素

```

1. map<string, int> temp;
2. temp.insert(make_pair("James", 3));
3. map<string, int>::iterator it = temp.begin();
4. it->first == "James";
5. it->second == 3;

```

6. // 或者使用下表操作
7. string value = temp["James"];

(3) 获取 set 中的元素

```
1. set<string> temp;  
2. temp.insert("James");  
3. set<string>::iterator it = temp.begin();  
4. *it == "James";
```

C++ 代码中出现的问题，迭代器无法加上整数

C++ 数字转字符串、字符串转数字

数字转字符串：

用 C++ 的 stringstream:

```
#include <sstream>  
#include <string>  
string num2str(double i)  
{  
    stringstream ss;  
    ss<<i;  
    return ss.str();  
}
```

字符串转数字：

```
int str2num(string s)  
{  
    int num;  
    stringstream ss(s);  
    ss>>num;  
    return num;  
}
```

上面方法很简便，缺点是处理大量数据转换速度较慢..

C library 中的 sprintf, sscanf 相对更快

可以用 sprintf 函数将数字输出到一个字符缓冲区中. 从而进行了转换...

例如：

已知从 0 点开始的秒数(seconds)，计算出字符串"H:M:S"，其中 H 是小时，M = 分钟，S = 秒

```
int H, M, S;
string time_str;
H=seconds/3600;
M=(seconds%3600)/60;
S=(seconds%3600)%60;
char ctime[10];

sprintf(ctime, "%d:%d:%d", H, M, S);    // 将整数转换成字符串

time_str=ctime;                        // 结果
```

与 sprintf 对应的是 sscanf 函数, 可以将字符串转换成数字

```
char str[] = "15.455";
int i;
float fp;

sscanf( str, "%d", &i );    // 将字符串转换成整数 i = 15

sscanf( str, "%f", &fp );    // 将字符串转换成浮点数 fp = 15.455000

//打印
printf( "Integer: = %d ", i+1 );
printf( "Real: = %f ", fp+1 );
return 0;
```

输出如下：

Integer: = 16

Real: = 16.455000

C++ STL 中 Map 的按 Key 排序和按 Value 排序

map 是用来存放<key, value>键值对的数据结构，可以很方便快速的根据 key 查到相应的 value。假如存储学生和其成绩（假定不存在重名，当然可以对重名加以区分），我们用

map 来进行存储就是个不错的选择。我们这样定义，map<string, int>，其中学生姓名用 string 类型，作为 Key；该学生的成绩用 int 类型，作为 value。这样一来，我们可以根据学生姓名快速的查找到他的成绩。

但是，我们除了希望能够查询某个学生的成绩，或许还想看看整体的情况。我们想把所有同学和他相应的成绩都输出来，并且按照我们想要的顺序进行输出：比如按照学生姓名的顺序进行输出，或者按照学生成绩的高低进行输出。换句话说，我们希望能够对 map 进行按 Key 排序或按 Value 排序，然后按序输出其键值对的内容。

一、C++ STL 中 Map 的按 Key 排序

其实，为了实现快速查找，map 内部本身就是按序存储的（比如红黑树）。在我们插入<key, value>键值对时，就会按照 key 的大小顺序进行存储。这也是作为 key 的类型必须能够进行<运算比较的原因。现在我们用 string 类型作为 key，因此，我们的存储就是按学生姓名的字典排序储存的。

【参考代码】

```
1. #include<map>
2. #include<string>
3. #include<iostream>
4. using namespace std;
5.
6. typedef pair<string, int> PAIR;
7.
8. ostream& operator<< (ostream& out, const PAIR& p) {
9.     return out << p.first << "\t" << p.second;
10. }
11.
12. int main() {
13.     map<string, int> name_score_map;
14.     name_score_map["LiMin"] = 90;
15.     name_score_map["ZiLinMi"] = 79;
16.     name_score_map["BoB"] = 92;
17.     name_score_map.insert(make_pair("Bing", 99));
18.     name_score_map.insert(make_pair("Albert", 86));
19.     for (map<string, int>::iterator iter = name_score_map.begin();
20.         iter != name_score_map.end();
21.         ++iter) {
22.         cout << *iter << endl;
23.     }
24.     return 0;
25. }
```

【运行结果】

```
Albert 86
Bing 99
BoB 92
LiMin 90
ZiLinMi 79
```

大家都知道 map 是 stl 里面的一个模板类，现在我们来看下 map 的定义：

```
1. template < class Key, class T, class Compare = less<Key>,
2.           class Allocator = allocator<pair<const Key,T> > > class map;
```

它有四个参数，其中我们比较熟悉的有两个: Key 和 Value。第四个是 Allocator，用来定义存储分配模型的，此处我们不作介绍。

现在我们重点看下第三个参数: class Compare = less<Key>

这也是一个 class 类型的，而且提供了默认值 less<Key>。less 是 stl 里面的一个函数对象，那么什么是函数对象呢？

所谓的函数对象：即调用操作符的类，其对象常称为函数对象（function object），它们是行为类似函数的对象。表现出一个函数的特征，就是通过“对象名+(参数列表)”的方式使用一个类，其实质是对 operator() 操作符的重载。

现在我们来看一下 less 的实现：

```
1. template <class T> struct less : binary_function <T,T,bool> {
2.     bool operator() (const T& x, const T& y) const
3.     {return x<y;}
4. };
```

它是一个带模板的 struct，里面仅仅对()运算符进行了重载，实现很简单，但用起来很方便，这就是函数对象的优点所在。stl 中还为四则运算等常见运算定义了这样的函数对象，与 less 相对的还有 greater：

```
1. template <class T> struct greater : binary_function <T,T,bool> {
2.     bool operator() (const T& x, const T& y) const
3.     {return x>y;}
4. };
```


map 这里指定 less 作为其默认比较函数(对象)，所以我们通常如果不自己指定 Compare，map 中键值对就会按照 Key 的 less 顺序进行组织存储，因此我们就看到了上面代码输出结果是按照学生姓名的字典顺序输出的，即 string 的 less 序列。

我们可以在定义 `map` 的时候，指定它的第三个参数 `Compare`，比如我们把默认的 `less` 指定为 `greater`：

【参考代码】

```
1. #include<map>
2. #include<string>
3. #include<iostream>
4. using namespace std;
5.
6. typedef pair<string, int> PAIR;
7.
8. ostream& operator<<(ostream& out, const PAIR& p) {
9.     return out << p.first << "\t" << p.second;
10. }
11.
12. int main() {
13.     map<string, int, greater<string> > name_score_map;
14.     name_score_map["LiMin"] = 90;
15.     name_score_map["ZiLinMi"] = 79;
16.     name_score_map["BoB"] = 92;
17.     name_score_map.insert(make_pair("Bing",99));
18.     name_score_map.insert(make_pair("Albert",86));
19.     for (map<string, int>::iterator iter = name_score_map.begin();
20.         iter != name_score_map.end();
21.         ++iter) {
22.         cout << *iter << endl;
23.     }
24.     return 0;
25. }
```

【运行结果】



```
ZiLinMi 79
LiMin 90
BoB 92
Bing 99
Albert 86
```

现在知道如何为 `map` 指定 `Compare` 类了，如果我们想自己写一个 `compare` 的类，让 `map` 按照我们想要的顺序来存储，比如，按照学生姓名的长短排序进行存储，那该怎么做呢？

其实很简单，只要我们自己写一个函数对象，实现想要的逻辑，定义 `map` 的时候把 `Compare` 指定为我们自己编写的这个就 ok 啦。

```

1. struct CmpByKeyLength {
2.     bool operator()(const string& k1, const string& k2) {
3.         return k1.length() < k2.length();
4.     }
5. };

```

是不是很简单！这里我们不用把它定义为模板，直接指定它的参数为 **string** 类型就可以了。

【参考代码】

```

1. int main() {
2.     map<string, int, CmpByKeyLength> name_score_map;
3.     name_score_map["LiMin"] = 90;
4.     name_score_map["ZiLinMi"] = 79;
5.     name_score_map["BoB"] = 92;
6.     name_score_map.insert(make_pair("Bing", 99));
7.     name_score_map.insert(make_pair("Albert", 86));
8.     for (map<string, int>::iterator iter = name_score_map.begin();
9.         iter != name_score_map.end();
10.        ++iter) {
11.         cout << *iter << endl;
12.     }
13.     return 0;
14. }

```

【运行结果】

```

BoB      92
Bing     99
LiMin    90
Albert   86
ZiLinMi  79

```

二、C++ STL 中 Map 的按 Value 排序

在第一部分中，我们借助 **map** 提供的参数接口，为它指定相应 **Compare** 类，就可以实现对 **map** 按 **Key** 排序，是在创建 **map** 并不断的向其中添加元素的过程中就会完成排序。

现在我们要从 `map` 中得到学生按成绩的从低到高的次序输出，该如何实现呢？换句话说，该如何实现 `Map` 的按 `Value` 排序呢？

第一反应是利用 `stl` 中提供的 `sort` 算法实现，这个想法是好的，不幸的是，`sort` 算法有个限制，利用 `sort` 算法只能对序列容器进行排序，就是线性的（如 `vector`，`list`，`deque`）。`map` 也是一个集合容器，它里面存储的元素是 `pair`，但是它不是线性存储的（前面提过，像红黑树），所以利用 `sort` 不能直接和 `map` 结合进行排序。

虽然不能直接用 `sort` 对 `map` 进行排序，那么我们可不可以迂回一下，把 `map` 中的元素放到序列容器（如 `vector`）中，然后再对这些元素进行排序呢？这个想法看似是可行的。要对序列容器中的元素进行排序，也有个必要条件：就是容器中的元素必须是可比较的，也就是实现了 `<` 操作的。那么我们现在就来看下 `map` 中的元素满足这个条件么？

我们知道 `map` 中的元素类型为 `pair`，具体定义如下：

```
1. template <class T1, class T2> struct pair
2. {
3.     typedef T1 first_type;
4.     typedef T2 second_type;
5.
6.     T1 first;
7.     T2 second;
8.     pair() : first(T1()), second(T2()) {}
9.     pair(const T1& x, const T2& y) : first(x), second(y) {}
10.    template <class U, class V>
11.        pair (const pair<U,V> &p) : first(p.first), second(p.second) { }
12. }
```

`pair` 也是一个模板类，这样就实现了良好的通用性。它仅有两个数据成员 `first` 和 `second`，即 `key` 和 `value`，而且

在 `<utility>` 头文件中，还为 `pair` 重载了 `<` 运算符，具体实现如下：

```
1. template<class _T1, class _T2>
2.     inline bool
3.     operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
4.     { return __x.first < __y.first
5.         || (!(__y.first < __x.first) && __x.second < __y.second); }
```

重点看下其实现：

```
1. __x.first < __y.first || (!(__y.first < __x.first) && __x.second < __y.second)
```

这个 `less` 在两种情况下返回 `true`，第一种情况：`__x.first < __y.first` 这个好理解，就是比较 `key`，如果 `__x` 的 `key` 小于 `__y` 的 `key` 则返回 `true`。

第二种情况有点费解： `!(__y.first < __x.first) && __x.second < __y.second`

当然由于`||`运算具有短路作用，即当前面的条件不满足是，才进行第二种情况的判断。第一种情况`__x.first < __y.first` 不成立，即`__x.first >= __y.first` 成立，在这个条件下，我们来分析下 `!(__y.first < __x.first) && __x.second < __y.second`

`!(__y.first < __x.first)`，看清出，这里是 `y` 的 `key` 不小于 `x` 的 `key`，结合前提条件，`__x.first < __y.first` 不成立，即 `x` 的 `key` 不小于 `y` 的 `key`

即： `!(__y.first < __x.first) && !(__x.first < __y.first)` 等价于 `__x.first == __y.first`，也就是说，第二种情况是在 `key` 相等的情况下，比较两者的 `value`（`second`）。

这里比较令人费解的地方就是，为什么不直接写 `__x.first == __y.first` 呢？这么写看似费解，但其实也不无道理：前面讲过，作为 `map` 的 `key` 必须实现`<`操作符的重载，但是并不保证`==`符也被重载了，如果 `key` 没有提供`==`，那么，`__x.first == __y.first` 这样写就错了。由此可见，`stl` 中的代码是相当严谨的，值得我们好好研读。

现在我们知道了 `pair` 类重载了`<`符，但是它并不是按照 `value` 进行比较的，而是先对 `key` 进行比较，`key` 相等时候才对 `value` 进行比较。显然不能满足我们按 `value` 进行排序的要求。

而且，既然 `pair` 已经重载了`<`符，而且我们不能修改其实现，又不能在外重复实现重载`<`符。

```
1. typedef pair<string, int> PAIR;
2. bool operator< (const PAIR& lhs, const PAIR& rhs) {
3.     return lhs.second < rhs.second;
4. }
```

如果 `pair` 类本身没有重载`<`符，那么我们按照上面的代码重载`<`符，是可以实现对 `pair` 的按 `value` 比较的。现在这样做不行了，甚至会出错（编译器不同，严格的就报错）。

那么我们如何实现对 `pair` 按 `value` 进行比较呢？第一种：是最原始的方法，写一个比较函数； 第二种：刚才用到了，写一个函数对象。这两种方式实现起来都比较简单。

```
1. typedef pair<string, int> PAIR;
2.
3. bool cmp_by_value(const PAIR& lhs, const PAIR& rhs) {
4.     return lhs.second < rhs.second;
5. }
6.
7. struct CmpByValue {
8.     bool operator()(const PAIR& lhs, const PAIR& rhs) {
9.         return lhs.second < rhs.second;
10.    }
11.};
```

接下来，我们看下 `sort` 算法，是不是也像 `map` 一样，可以让我们自己指定元素间如何进行比较呢？

```
1. template <class RandomAccessIterator>
2.     void sort ( RandomAccessIterator first, RandomAccessIterator last );
3.
4. template <class RandomAccessIterator, class Compare>
5.     void sort ( RandomAccessIterator first, RandomAccessIterator last, Compare
        comp );
```

我们看到，令人兴奋的是，`sort` 算法和 `map` 一样，也可以让我们指定元素间如何进行比较，即指定 `Compare`。需要注意的是，`map` 是在定义时指定的，所以传参的时候直接传入函数对象的类名，就像指定 `key` 和 `value` 时指定的类型名一样；`sort` 算法是在调用时指定的，需要传入一个对象，当然这个也简单，类名()就会调用构造函数生成对象。

这里也可以传入一个函数指针，就是把上面说的第一种方法的函数名传过来。（应该是存在函数指针到函数对象的转换，或者两者调用形式上是一致的，具体确切原因还不明白，希望知道的朋友给讲下，先谢谢了。）

【参考代码】

```
1. int main() {
2.     map<string, int> name_score_map;
3.     name_score_map["LiMin"] = 90;
4.     name_score_map["ZiLinMi"] = 79;
5.     name_score_map["BoB"] = 92;
6.     name_score_map.insert(make_pair("Bing",99));
7.     name_score_map.insert(make_pair("Albert",86));
8.     //把 map 中元素转存到 vector 中
9.     vector<PAIR> name_score_vec(name_score_map.begin(), name_score_map.end());

10.    sort(name_score_vec.begin(), name_score_vec.end(), CmpByValue());
11.    // sort(name_score_vec.begin(), name_score_vec.end(), cmp_by_value);
12.    for (int i = 0; i != name_score_vec.size(); ++i) {
13.        cout << name_score_vec[i] << endl;
14.    }
15.    return 0;
16. }
```

【运行结果】

```
ZiLinMi 79
Albert 86
LiMin 90
BoB 92
Bing 99
```

assert()函数用法总结

assert 宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行，原型定义：

```
#include <assert.h>
void assert( int expression );
```

assert 的作用是现计算表达式 **expression**，如果其值为假（即为 0），那么它先向 **stderr** 打印一条出错信息，然后通过调用 **abort** 来终止程序运行。请看下面的程序清单 **badptr.c**：

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int main( void )
{
    FILE *fp;

    fp = fopen( "test.txt", "w" );//以可写的方式打开一个文件，如果
    不存在就创建一个同名文件
    assert( fp );                //所以这里不会出错
    fclose( fp );

    fp = fopen( "noexitfile.txt", "r" );//以只读的方式打开一个文
    件，如果不存在就打开文件失败
    assert( fp );                //所以这里出错
    fclose( fp );                //程序永远都执行不到这
    里来
    return 0;
}
```

```
[root@localhost error_process]# gcc badptr.c
```

```
[root@localhost error_process]# ./a.out
```

```
a.out: badptr.c:14: main: Assertion `fp' failed.
```

已放弃使用 **assert()** 的缺点是，频繁的调用会极大的影响程序的性能，增加额外的开销。在调试结束后，可以通过在包含 **#include <assert.h>** 的语句之前插入 **#define NDEBUG** 来禁用 **assert** 调用，示例代码如下：

```
#include <stdio.h>
#define NDEBUG
#include <assert.h>
```

用法总结与注意事项：

- 1) 在函数开始处检验传入参数的合法性如：

```

int resetBufferSize(int nNewSize)
{
    //功能:改变缓冲区大小,
    //参数:nNewSize 缓冲区新长度
    //返回值:缓冲区当前长度
    //说明:保持原信息内容不变    nNewSize<=0 表示清除缓冲区
    assert(nNewSize >=0);
    assert(nNewSize <= MAX_BUFFER_SIZE);
    ...
}

```

2) 每个 **assert** 只检验一个条件, 因为同时检验多个条件时, 如果断言失败, 无法直观的判断是哪个条件失败, 如:

不好:

```
assert(nOffset>=0&& nOffset+nSize<=m_nInformationSize);
```

好:

```
assert(nOffset >=0);
assert(nOffset+nSize <= m_nInformationSize);
```

3) 不能使用改变环境的语句, 因为 **assert** 只在 **DEBUG** 个生效, 如果这么做, 会使用程序在真正运行时遇到问题, 如:

错误:

```
assert(i++<100);
```

这是因为如果出错, 比如在执行之前 **i=100**, 那么这条语句就不会执行, 那么 **i++** 这条命令就没有执行。

正确:

```
assert(i <100);
i++;
```

4) **assert** 和后面的语句应空一行, 以形成逻辑和视觉上的一致感。

5) 有的地方, **assert** 不能代替条件过滤。

字符串操作

1、string 字符串插入操作

```

public:
String Insert(
    int startIndex,
    char value
)

```

例：输出结果是“ABZCDE”

```
string temp = "ABCDE";  
  
temp.insert(temp.begin() + 2, 'Z');  
  
cout<<temp;
```

2、string 字符串删除操作

- (1) erase(pos,n); 删除从 pos 开始的 n 个字符，比如 erase(0,1)就是删除第一个字符
- (2) erase(position);删除 position 处的一个字符(position 是个 string 类型的迭代器)
- (3) erase(first,last);删除从 first 到 last 之间的字符（first 和 last 都是迭代器）

例：输出是 “ABE”

```
string temp = "ABCDE";  
  
temp.erase(2, 2);  
  
cout<<temp;
```

3、const 指针只能赋值给 const 指针。

例：

```
1. int lengthOfLastWord(const char *s)  
  
2. {  
  
3.     int len = strlen(s); //使用 strlen 函数可以得到指针字符串长度  
  
4.     const char *word = s; //const 指针只能赋值给 const 指针  
  
5. }
```

4、string 中没有直接翻转的函数

```
1. #include <string>
```

```
2. string temp;
```

```
3. temp.reserve(size); //将字符串的容量设置为至少 size
```

```
4.
```

```
5. #include <algorithm>
```

```
6. #include <string>
```

```
7. string temp;
```

```
8. reverse(temp.begin(), temp.end()); //将 temp 翻转
```

C++中 cin、cin.get()、cin.getline()、getline()、gets()等函数的用法

学 C++ 的时候，这几个输入函数弄的有点迷糊；这里做个小结，为了自己复习，也希望对后来者能有所帮助，如果有差错的地方还请各位多多指教（本文所有程序均通过 VC 6.0 运行）

1、cin

2、cin.get()

3、cin.getline()

4、getline()

5、gets()

6、getchar()

附:cin.ignore();cin.get()//跳过一个字符,例如不想要的回车,空格等字符

1、cin>>

用法 1：最基本，也是最常用的用法，输入一个数字：

```
#include <iostream>
using namespace std;
int main ()
{
    int a,b;
    cin>>a>>b;
    cout<<a+b<<endl;
}
```

输入：2[回车]3[回车]

输出：5

注意:>> 是会过滤掉不可见字符（如 空格 回车，TAB 等）

cin>>noskipws>>input[j];//不想略过空白字符，那就使用 **noskipws** 流控制

用法 2：接受一个字符串，遇“空格”、“TAB”、“回车”都结束

```
#include <iostream>
using namespace std;
int main ()
{
    char a[20];
    cin>>a;
    cout<<a<<endl;
}
```

输入: jkljkljkl

输出: jkljkljkl

输入: jkljkl jkljkl //遇空格结束

输出: jkljkl

2、**cin.get()**

用法 1： **cin.get(字符变量名)**可以用来接收字符

```
#include <iostream>
using namespace std;
int main ()
{
    char ch;
    ch=cin.get();          //或者 cin.get(ch);
    cout<<ch<<endl;
}
```

输入: jkljkljkl

输出: j

用法 2： **cin.get(字符数组名,接收字符数目)**用来接收一行字符串,可以接收空格

```
#include <iostream>
using namespace std;
int main ()
{
    char a[20];
    cin.get(a,20);
    cout<<a<<endl;
}
```

输入: jkl jkl jkl

输出: jkl jkl jkl

输入: abcdeabcdeabcdeabcdeabcde （输入 25 个字符）

输出: abcdeabcdeabcdeabcde （接收 19 个字符+1 个'\0'）

用法 3: `cin.get()`(无参数)没有参数主要是用于舍弃输入流中的不需要的字符,或者舍弃回车,弥补 `cin.get()`(字符数组名,接收字符数目)的不足.

```
#include <vector>
using namespace std;
int main(int argc, char* argv[])
{
    int n;
    char s[100];
    cin>>n;
    cin.get();// cin 后有回车符, 需要使用 cin.get()将其省略, 不影响 cin.getline()的输入
    while(n > 0)
    {
        n--;
        memset(s, '\0', sizeof(char) * 100);
        cin.getline(s, 5);
        cout<<s;
    }
    system("pause");
}
```

3、`cin.getline()` // 接受一个字符串, 可以接收空格并输出

```
#include <iostream>
using namespace std;
int main ()
{
    char m[20];
    cin.getline(m,5);
    cout<<m<<endl;
}
```

输入: jkljkljkl

输出: jklj

接受 5 个字符到 m 中, 其中最后一个为'\0', 所以只看到 4 个字符输出;

如果把 5 改成 20:

输入: jkljkljkl

输出: jkljkljkl

输入: jklf fjlsjf fj sdklf

输出: jklf fjlsjf fj sdklf

//延伸:

//cin.getline()实际上有三个参数, cin.getline(接受字符串的看哦那间 m,接受个数 5,结束字符)

//当第三个参数省略时, 系统默认为'\0'

//如果将例子中 cin.getline()改为 cin.getline(m,5,'a');当输入 jlkjkljkl 时输出 jklj, 输入 jkaljkljkl 时, 输出 jk

当用在多维数组中的时候, 也可以用 cin.getline(m[i],20)之类的用法:

```
#include<iostream>
#include<string>
using namespace std;
int main ()
{
    char m[3][20];
    for(int i=0;i<3;i++)
    {
        cout<<"\n 请输入第"<<i+1<<"个字符串: "<<endl;
        cin.getline(m[i],20);
    }
    cout<<endl;
    for(int j=0;j<3;j++)
        cout<<"输出 m["<<j<<"]的值:"<<m[j]<<endl;
}
```

请输入第 1 个字符串:

kskr1

请输入第 2 个字符串:

kskr2

请输入第 3 个字符串:

kskr3

输出 m[0]的值:kskr1

输出 m[1]的值:kskr2

输出 m[2]的值:kskr3

4、getline() // 接受一个字符串, 可以接收空格并输出, 需包含
"#include<string>"

```
#include<iostream>
#include<string>
using namespace std;
int main ()
{
    string str;
    getline(cin,str);
```

```
    cout<<str<<endl;  
}
```

输入: jkljkljkl

输出: jkljkljkl

输入: jkl jfksldfj jklsjfl

输出: jkl jfksldfj jklsjfl

和 `cin.getline()` 类似, 但是 `cin.getline()` 属于 `istream` 流, 而 `getline()` 属于 `string` 流, 是不一样的两个函数

关于 C++ 中的虚拟继承的一些总结

1. 为什么要引入虚拟继承

虚拟继承是多重继承中特有的概念。虚拟基类是为了解决多重继承而出现的。如: 类 D 继承自

类 B1、B2, 而类 B1、B2 都继承自类 A, 因此在类 D 中两次出现类 A 中的变量和函数。

为了节省内存空间, 可以将 B1、B2 对 A 的继承定义为虚拟继承, 而 A 就成了虚拟基类。

实现的代码如下:

```
class A
```

```
class B1:public virtual A;
```

```
class B2:public virtual A;
```

```
class D:public B1,public B2;
```

虚拟继承在一般的应用中很少用到, 所以也往往被忽视, 这也主要是在 C++ 中, 多重

继承是不推荐的, 也并不常用, 而一旦离开了多重继承, 虚拟继承就完全失去了存在的必

要因为这样只会降低效率和占用更多的空间。

2. 引入虚继承和直接继承会有什么区别呢

由于有了间接性和共享性两个特征，所以决定了虚继承体系下的对象在访问时必然会在时间和空间上与一般情况有较大不同。

2.1 时间：在通过继承类对象访问虚基类对象中的成员（包括数据成员和函数成员）时，都必须通过某种间接引用来完成，这样会增加引用寻址时间（就和虚函数一样），其实就是调整 this 指针以指向虚基类对象，只不过这个调整是运行时间接完成的。

2.2 空间：由于共享所以不必要在对象内存中保存多份虚基类子对象的拷贝，这样较之多继承节省空间。虚拟继承与普通继承不同的是，虚拟继承可以防止出现 diamond 继承时，一个派生类中同时出现了两个基类的子对象。也就是说，为了保证这一点，在虚拟继承情况下，基类子对象的布局是不同于普通继承的。因此，它需要多出一个指向基类子对象的指针。

3.笔试，面试中常考的 C++ 虚拟继承的知识点

第一种情况：

```
class a
{
    virtual void func();
};
class b:public virtual a
{
    virtual void foo();
};
```

第二种情况：

```
class a
{
    virtual void func();
};
class b:public a
{
    virtual void foo();
};
```

第三种情况

```
class a
{
    virtual void func();
    char x;
};
class b:public virtual a
{
    virtual void foo();
};
```

第四种情况：

```
class a
{
    virtual void func();
    char x;
};
class b:public a
{
    virtual void foo();
};
```

（1）如果不是虚继承，则基类、派生类不管有多少虚函数都使用一个虚指针。

（2）如果是虚继承，则基类使用一个虚指针，派生类使用一个虚指针，还要加上虚继承的指针。

如果对这四种情况分别求 sizeof(a)，sizeof(b)。结果是什么样的呢？下面是输出结果：

（在 vc6.0 中运行）

第一种：4, 12

第二种：4, 4

第三种：8, 16

第四种：8, 8

想想这是为什么呢？

因为每个存在虚函数的类都要有一个 4 字节的指针指向自己的虚函数表，所以每种情况的类 a 所占的字节数应该是没有什么问题的，那么类 b 的字节数怎么算呢？看“第一种”和“第三种”情况采用的是虚继承，那么这时候就要有这样的一个指针 vptr_b_a，这个指针叫虚类指针，也是四个字节；还要包括类 a 的字节数，所以类 b 的字节数就求出来了。而“第二种”和“第四种”情况则不包括 vptr_b_a 这个指针，这回应该木有问题了吧。

4.c++ 重载、覆盖、隐藏的区别和执行方式

既然说到了继承的问题，那么不妨讨论一下经常提到的重载，覆盖和隐藏

4.1 成员函数被重载的特征

- (1) 相同的范围（在同一个类中）；
- (2) 函数名字相同；
- (3) 参数不同；
- (4) virtual 关键字可有可无。

4.2 “覆盖”是指派生类函数覆盖基类函数，特征是：

- (1) 不同的范围（分别位于派生类与基类）；
- (2) 函数名字相同；
- (3) 参数相同；

(4) 基类函数必须有 virtual 关键字。

4.3 “隐藏”是指派生类的函数屏蔽了与其同名的基类函数，特征是：

(1) 如果派生类的函数与基类的函数同名，但是参数不同，此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载混淆）。

(2) 如果派生类的函数与基类的函数同名，但是参数相同，但是基类函数没有 virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

小结：说白了就是如果派生类和基类的函数名和参数都相同，属于覆盖，这是可以理解的吧，完全一样当然要覆盖了；如果只是函数名相同，参数并不相同，则属于隐藏。

4.4 三种情况怎么执行：

4.4.1 重载：看参数。

4.4.2 隐藏：用什么就调用什么。

4.4.3 覆盖：调用派生类。