**STL关联容器概述**

2011年07月09日 16:00:11 [yfk](https://me.csdn.net/yfkiss) 阅读数 6647

版权声明：本文为博主原创文章，遵循[CC 4.0 by-sa](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/yfkiss/article/details/6594859>

**1. 概念** 

STL容器大的方向分为两类，序列式容器和关联式容器，这两者通过数据在容器内的排列来区分，关联容器是通过键(key)存储和读取元素的，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。  
标准的STL序列容器包括：vector、list、deque、heap（算法呈现）、stack(适配器)、queue（适配器）、priority\_queue(适配器)。标准的STL关联式容器包括：set、multiset、map、multimap。SGI STL还提供了一些非标准的关联式容器，eg：hash\_table、hash\_set。

**2. 底层实现**先了解几个概念：  
**二叉搜索树**是一种特殊的二叉树，其具有如下性质：  
1) 若左子树不空，则左子树所有结点的值均小于它的根结点的值  
2）若右子树不空，则右子树所有节点的值均大于它的根节点的值  
3）左右子树也分别为二叉搜索树  
二叉搜索树支持各种动态集合操作，包括：插入、查找、删除，其操作的时间复杂度与树的高度成正比，在遇到二叉树极端不平衡的情况下，其形状就与链表是一样的，二叉树插入、查找、删除的时间复杂度都退化为O（n）。  
**平衡二叉搜索树**是一种特殊的二叉搜索树，其没有一个节点深度过大，不同的平衡条件，造就不同的效率表现。常见的平衡二叉搜索树有：AVL-tree和RB-tree。  
**关联容器一般以平衡二叉搜索树作为内部数据结构，RB-tree的应用尤其广泛。  
RB-tree**是许多平衡二叉查找树的一种，一颗有n个内结点的红黑树的高度至多为2lg(n+1)，它能保证在最坏情况下，基本的动态集合操作时间为O(lgn)。

**3. set / multiset  
3.1 概念**  
set不区分键值和实值，其键值就是实值。顾名思义，可以把set当做集合使用，由于set的底层是平衡二叉搜索树，因此其在插入、查询和删除时都是O(lgn)的时间复杂度。set和multiset唯一的不同是，set不允许任何两个元素有相同的值，而multiset允许键值重复。  
  
**3.2 迭代器**  
set的迭代器本质上是const\_iterator，这是因为RB-tree的结构依赖于数据的组织，如果允许通过iterator改变set元素值，会严重破坏RB-tree结构。此外，**当对set进行插入和删除时，除了影响指向操作元素本身的迭代器之外，不影响指向其它元素的迭代器。  
  
3.3 Set API**(constructor) Construct set (public member function)  
(destructor) Set destructor (public member function)  
operator= Copy container content (public member function)  
**Iterators:**begin Return iterator to beginning (public member function)  
end Return iterator to end (public member function)  
rbegin Return reverse iterator to reverse beginning (public member function)  
rend Return reverse iterator to reverse end (public member function)  
**Capacity:**  
empty Test whether container is empty (public member function)  
size Return container size (public member function)  
max\_size Return maximum size (public member function)  
**Modifiers:**  
insert Insert element (public member function)  
erase Erase elements (public member function)  
swap Swap content (public member function)  
clear Clear content (public member function)  
**Observers:**  
key\_comp Return comparison object (public member function)  
value\_comp Return comparison object (public member function)  
**Operations:**  
find Get iterator to element (public member function)  
count Count elements with a specific key (public member function )  
lower\_bound Return iterator to lower bound (public member function)  
upper\_bound Return iterator to upper bound (public member function)  
equal\_range Get range of equal elements (public member function)  
**Allocator:**  
get\_allocator Get allocator (public member function)  
  
**3.4 set实例：**

1. #include <iostream>
2. #include <set>
3. using namespace std;
5. int \_tmain(int argc, \_TCHAR\* argv[])
6. {
7. set<int> myset;
8. set<int>::iterator it, itlow, uplow;
9. int i;
11. *// 插入数据*
12. cout << "insert elements " << endl;
13. for (i=0; i<10; i++)
14. myset.insert(i\*2);
16. int num = 4;
18. *//查询、删除数据*
19. cout << "find and delete element" << 4 << endl;
20. it=myset.find(num);
21. if(it != myset.end())
22. {
23. myset.erase(it);
24. }
26. num = 7;
28. *// 元素边界*
29. itlow = myset.lower\_bound(num);
30. uplow = myset.upper\_bound(num);
31. cout << num << " lower bound is " << \*itlow << endl;
32. cout << num << " upper bound is " << \*uplow << endl;
33. pair<set<int>::iterator,set<int>::iterator> ret = myset.equal\_range(7);
34. cout << num << " lower bound is " << \*ret.first << endl;
35. cout << num << " upper bound is " << \*ret.second << endl;
37. *// check数据是否在容器中*
38. if (myset.count(num)>0)
39. cout << num << " is an element of myset.\n";
40. else
41. cout << num << " is not an element of myset.\n";
43. *//输出容器大小*
44. cout << "set size: " << (int) myset.size() << endl;
46. *//输出容器内元素*
47. cout << "myset contains:";
48. for (it=myset.begin(); it!=myset.end(); it++)
49. {
50. cout << " " << \*it;
51. }
52. cout << endl;
54. system("pause");
55. return 0;
56. }

输出：  
insert elements  
find and delete element4  
7 lower bound is 8  
7 upper bound is 8  
7 lower bound is 8  
7 upper bound is 8  
7 is not an element of myset.  
set size: 9  
myset contains: 0 2 6 8 10 12 14 16 18

**4. map/ multimap  
4.1 概念**  
和set相比，map同时拥有实值(value)和键值(key)，其每一个元素都是pair，pair的第一个元素是键值，第二个元素是实值。map和multimap的区别在于，map不允许两个元素拥有相同的键值，而multimap允许存在重复的键值。  
pair定义如下：

1. template <class T1, class T2>
2. struct pair
3. {
4. typedef T1 first\_type;
5. typedef T2 second\_type;
7. T1 first;
8. T2 second;
9. }

**4.2 迭代器**  
map和set的底层实现都是很RB-tree，其迭代器特点和set一致。 **4.3 Map API**(constructor) Construct map (public member function)  
(destructor) Map destructor (public member function)  
operator= Copy container content (public member function)  
**Iterators**:  
begin Return iterator to beginning (public member function)  
end Return iterator to end (public member function)  
rbegin Return reverse iterator to reverse beginning (public member function)  
rend Return reverse iterator to reverse end (public member function)  
**Capacity**:  
empty Test whether container is empty (public member function)  
size Return container size (public member function)  
max\_size Return maximum size (public member function)  
**Element access:**  
operator[] Access element (public member function)  
**Modifiers**:  
insert Insert element (public member function)  
erase Erase elements (public member function)  
swap Swap content (public member function)  
clear Clear content (public member function)  
**Observers**:  
key\_comp Return key comparison object (public member function)  
value\_comp Return value comparison object (public member function)  
**Operations**:  
find Get iterator to element (public member function )  
count Count elements with a specific key (public member function)  
lower\_bound Return iterator to lower bound (public member function)  
upper\_bound Return iterator to upper bound (public member function)  
equal\_range Get range of equal elements (public member function)  
**Allocator**:  
get\_allocator Get allocator (public member function)  
  
**4.4 map实例：**

1. #include <iostream>
2. #include <map>
3. using namespace std;
5. int \_tmain(int argc, \_TCHAR\* argv[])
6. {
7. map<char,int> mymap;
8. map<char,int>::iterator it;
9. pair<map<char,int>::iterator,bool> ret1;
10. pair<map<char,int>::iterator,map<char,int>::iterator> ret;
12. *//插入元素*
13. mymap['a']=10;
14. mymap['b']=20;
15. mymap['c']=30;
16. mymap['d']=40;
17. ret1=mymap.insert (pair<char,int>('c',500) );
18. if (ret1.second==false)
19. {
20. cout << "element 'c' already existed";
21. cout << " with a value of " << ret1.first->second << endl;
22. }
24. *//寻找、删除*
25. it=mymap.find('d');
26. mymap.erase (it);
28. *//边界*
29. ret = mymap.equal\_range('b');
30. cout << "lower bound points to: ";
31. cout << ret.first->first << " => " << ret.first->second << endl;
32. cout << "upper bound points to: ";
33. cout << ret.second->first << " => " << ret.second->second << endl;
35. *// 输出容器元素*
36. cout << "mymap contains:\n";
37. for ( it=mymap.begin() ; it != mymap.end(); it++ )
38. cout << (\*it).first << " => " << (\*it).second << endl;
40. system("pause");
41. return 0;
42. }

输出：  
element 'c' already existed with a value of 30  
lower bound points to: b => 20  
upper bound points to: c => 30  
mymap contains:  
a => 10  
b => 20  
c => 30  
  
**5. 结语**关联容器是把一个键值与一个元素联系起来，并使用该键来加速诸如查找、插入以及删除元素等操作。**和序列容器相比，在使用键值来查找元素时，关联容器有更好的表现。**

**c++ 关联容器用法详解（set与map）**

**三、关联容器**

关联容器与序列容器有着根本性的不同，序列容器的元素是按照在容器中的位置来顺序保存和访问的，而关联容器的元素是按关键元素来保存和访问的。关联容器支持高效的关键字查找与访问。两个主要的关联容器类型是map与set。

**1.set**

**1.1 简介：**set里面每个元素只存有一个key，它支持高效的关键字查询操作。set对应数学中的“集合”。

**1.2 特点：**

* 储存同一类型的数据元素（这点和vector、queue等其他容器相同）
* 每个元素的值都唯一（没有重复的元素）
* 根据元素的值自动排列大小（有序性）
* 无法直接修改元素
* 高效的插入删除操作

**1.3 声明：**set<T> a

set<int> a;

**1.4 常用函数**

以下设 set<T> a,其中a是T类型的set容器。

|  |  |  |
| --- | --- | --- |
| 表达式 | 返回类型 | 说明 |
| a.begin() |  | 返回指向第一个元素的迭代器 |
| a.end() |  | 返回指向超尾的迭代器 |
| a.clear() |  | 清空容器a |
| a.empty() |  | 判断容器是否为空 |
| a.size() |  | 返回当前容器元素个数 |
| a.count(x) |  | 返回容器a中元素x的个数 |

**1.6 插入元素：**

* a.insert(x) :其中a为set<T>型容器，x为T型变量

1. set<int> a={0,1,2,9};
2. a.insert(6);
3. for(auto it = a.begin();it != a.end();it++) cout << \*it;//输出01269

* a.insert(first,second):其中first为指向区间左侧的迭代器，second为指向右侧的迭代器。作用是将first到second区间内元素插入到a（左闭右开）。

1. set<int> a = {0,1,2,9};
2. set<int> b = {3,4,5};
3. auto first = b.begin();
4. auto second = b.end();
5. a.insert(first,second);
6. for(auto it = a.begin();it != a.end();it++) cout << \*it;

插入元素会自动插入到合适的位置，使整个集合有序

**1.7 删除元素：**

* a.erase(x)：删除建值为x的元素
* a.erase(first,second)：删除first到second区间内的元素（左闭右开）
* a.erase(iterator):删除迭代器指向的元素
* set中的删除操作是不进行任何的错误检查的，比如定位器的是否合法等等，所以用的时候自己一定要注意。

**1.8 lower\_bound 和 upper\_bound 迭代器：**

* lower\_bound（x1）:返回第一个不小于键参数x1的元素的迭代器
* upper\_bound（x2）:返回最后一个大于键参数x2的元素的迭代器
* 由以上俩个函数，可以得到一个目标区间，即包含集合中从'x1'到'x2'的所有元素

1. #include<iostream>
2. #include<set>
3. #include<algorithm>
4. using namespace std;
5. int main()
6. {
7. set<int> a = {0,1,2,5,9};
8. auto it2 = a.lower\_bound(2);//返回指向第一个大于等于x的元素的迭代器
9. auto it = a.upper\_bound(2);//返回指向第一个大于x的元素的迭代器
10. cout << \*it2 << endl;//输出为2
11. cout << \*it << endl;//输出为5
12. return 0;
13. }

**1.9 set\_union() 与 set\_intersection()**

set\_union():对集合取并集

set\_union()函数接受5个迭代器参数。前两个迭代器定义了第一个集合的区间，接下来的俩个迭代器定义了第二个集合的区间，最后一个迭代器是输出迭代器，指出将结果集合复制到什么位置。例如：要将A与B的集合复制到C中，可以这样写：

1. #include<iostream>
2. #include<set>
3. #include<algorithm>
4. using namespace std;
5. int main()
6. {
7. set<int> A = {1,2,3}, B= {2,4,5},C;
8. set\_union(A.begin(),A.end(),B.begin(),B.end(),
9. insert\_iterator<set<int> >(C,C.begin()));
10. for(auto it = C.begin();it != C.end();it++)
11. cout << \*it <<" ";
12. return 0;
13. }

注意：

其中第五个参数不能写C.begin(),原因有两个：首先，关联集合将建看作常量，所以C.begin()返回的迭代器是常量迭代器，不能作为输出迭代器(详情请参考迭代器相关概念)。其次，与copy()相同，set\_union()将覆盖容器中已有的数据，并且要求容器用足够的空间容纳新信息，而C不满足，因为它是空的。

解决方法：可以创建一个匿名的insert\_iterator,将信息复制给C。如上述代码所为。另一种方法如下：

1. set\_union(A.begin(),A.end(),B.begin(),B.end(),
2. inserter(C,C.begin()));//调用inserter

set\_intersection():对集合取交集，它的接口与set\_union()相同。

附：使用set\_union()和set\_intersection()还有另一种技巧。由于需要五个迭代器，看起来会很累赘和麻烦，如果多次使用会增加出错的几率，所以我们可以试试用宏定义的方法来简化代码。如下：

1. #include<iostream>
2. #include<set>
3. #include<algorithm>
4. using namespace std;
5. #define ALL(x) x.begin(),x.end()
6. #define INS(x) inserter(x,x.begin())
7. int main()
8. {
9. set<int> A = {1,2,3}, B= {2,4,5},C;
10. set\_union(ALL(A),ALL(B),INS(C));
11. for(auto it = C.begin();it != C.end();it++)
12. cout << \*it <<" ";
13. return 0;
14. }

其中使用到了宏定义。

**1.10 set的几个问题：**

（1）为何map和set的插入删除效率比用其他序列容器高？

因为对于关联容器来说，不需要做内存拷贝和内存移动。set容器内所有元素都是以节点的方式来存储，其节点结构和链表差不多，指向父节点和子节点。因此插入的时候只需要稍做变换，把节点的指针指向新的节点就可以了。删除的时候类似，稍做变换后把指向删除节点的指针指向其他节点也OK了。这里的一切操作就是指针换来换去，和内存移动没有关系。

（2）为何每次insert之后，以前保存的iterator不会失效？

iterator这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢(当然被删除的那个元素本身已经失效了)。相对于vector来说，每一次删除和插入，指针都有可能失效，调用push\_back在尾部插入也是如此。因为为了保证内部数据的连续存放，iterator指向的那块内存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使时push\_back的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，申请新的更大的内存，复制已有的数据元素到新的内存，最后把需要插入的元素放到最后，那么以前的内存指针自然就不可用了。特别时在和find等算法在一起使用的时候，牢记这个原则：不要使用过期的iterator。

（3）当数据元素增多时，set的插入和搜索速度变化如何？

如果你知道log2的关系你应该就彻底了解这个答案。在set中查找是使用二分查找，也就是说，如果有16个元素，最多需要比较4次就能找到结果，有32个元素，最多比较5次。那么有10000个呢？最多比较的次数为log10000，最多为14次，如果是20000个元素呢？最多不过15次。看见了吧，当数据量增大一倍的时候，搜索次数只不过多了1次，多了1/14的搜索时间而已。你明白这个道理后，就可以安心往里面放入元素了。

**2.map**

**2.1 简介：**如果说set对应数学中的“集合”，那么map对应的就是“映射”。map是一种key-value型容器，其中key是关键字，起到索引作用，而value就是其对应的值。与set不同的是它支持下标访问。头文件是<map>

**2.2 特点：**

* 增加和删除节点对迭代器的影响很小(高效的插入与删除)
* 快速的查找（同set）
* 自动建立key-value的对应，key和value可以是任何你需要的类型
* 可以根据key修改value的记录
* 支持下标[]操作

**2.3 声明：**map<T1,T2> m

其中T1是key类型，T2是value类型，m就是一个T1-T2的key-value。

map<string,int> m;//声明一个key为string，value为int的map型容器

下述代码更清楚的解释了map容器的特点：

1. #include<iostream>
2. #include<map>
3. using namespace std;
4. int main()
5. {
6. map<string,int> m;
7. m["abc"] = 5;
8. m["cdf"] = 6;
9. m["b"] = 1;
10. for(auto it = m.begin();it != m.end();it++)
11. cout << it->first <<" " << it->second << endl;
12. return 0;
13. }

在上述代码中，m容器被按照key的字典序升序排列了，而且我们可以通过将key当作索引来获取value的值。（同时这也是一种插入方法）

**2.4 插入元素：**

* 使用insert()函数插入pair类型的元素
* 使用下标操作向map容器中插入元素

1. map<string,int> m;
2. m.insert(make\_pair("b",6));//insert插入
3. m["a"] = 5;//使用下标插入

**2.5 删除元素：**

* erase(key):删除键为key的元素
* erase(it):删除迭代器it所指向的元素

1. #include<iostream>
2. #include<map>
3. using namespace std;
4. int main()
5. {
6. map<string,int> m;
7. m.insert(make\_pair("b",6));
8. m["a"] = 5;
9. m["c"] = 5;
10. m["d"] = 5;
11. m["e"] = 5;
13. m.erase("d");
15. auto pr = m.begin();
16. m.erase(pr);
18. for(auto it = m.begin();it != m.end();it++)
19. cout << it->first <<" " << it->second << endl;
20. return 0;
21. }

**2.6 map容器的遍历：**

* 使用迭代器遍历（代码如上）

注：使用迭代器遍历map容器，其中每一个元素可以看成是pair类型的，访问第一个位置的key值可以用it->first访问，第二个位置value的值可以用it->second访问，其中it是指向该元素的迭代器。

**2.7 常用函数：**

下表中m为map类型的容器，it为和m同类型的迭代器，key表示该类型的一个键。

|  |  |  |
| --- | --- | --- |
| 表达式 | 返回类型 | 说明 |
| m.Count(key) |  | 返回map中key出现的次数（0或1） |
| m.find(key) | 迭代器 | 返回指向key位置的迭代器.若无则返回m.end() |
| m.insert(make\_pair( ) ) |  | 插入一个元素(必须以pair形式插入) |
| m.erase(it) |  | 删除迭代器it所指向的元素 |
| m.erase(key) |  | 删除键值为key的元素 |
| m.size() |  | 返回m中元素的个数 |
| m.clear() |  | 清空m容器 |
| m.empty() | bool | 判断容器是否为空。空则返回true |
| m.lower\_bound(key) | 迭代器 | 返回指向第一个键值不小于key的元素的迭代器 |
| m.upper\_bound(key) | 迭代器 | 返回指向第一个键值大于key的元素的迭代器 |

**3.重要例题：**

题目来自UVa12096

有一个专门为了集合运算而设计的“集合栈”计算机。该机器有一个初始为空的栈，并且支持以下操作：

* PUSH：空集“{}”入栈
* DUP：把当前栈顶元素复制一份后再入栈
* UNION：出栈两个集合，然后把二者的并集入栈
* INTERSECT：出栈两个集合，然后把二者交集入栈
* ADD：出栈两个集合，然后把先出栈的集合加入到后出栈的集合中，把结果入栈

每次操作后，输出栈顶集合的大小（即元素个数）。例如，栈顶元素是A={ {}，{ {} } }，，下一个元素是B= { {}，  { { {} } }  }，则：

* UNION操作将得到{ {}，{ {} }，{ { {} } }  }，输出3
* INTERSECT操作将得到{ {} }，输出1
* ADD操作将得到{ {},{ { {} } } ,{ {},{ {} } }  },输出3

输出不超过2000个操作，并且保证操作均能顺利进行（不需要对空栈执行出栈操作）

1. #include<cstdio>
2. #include<iostream>
3. #include<algorithm>//set\_union等函数定义在这里
4. #include<vector>
5. #include<set>
6. #include<map>
7. #include<stack>
9. #define ALL(x) x.begin(),x.end()
10. #define INS(x) inserter(x,x.begin()) //注意宏的括号和inserter
12. using namespace std;
14. typedef set<int> Set;
15. map<Set,int> IDCache;
16. vector<Set> setCache;
17. int t,n;
18. char cmd[10];
19. int getID(Set s){
20. if(IDCache.count(s))return IDCache[s];
21. setCache.push\_back(s); //将新集合加入Setcache
22. return IDCache[s]=setCache.size()-1;//将ID加入map ，同时返回新分配的ID值
23. }
25. int main(){
26. scanf("%d",&t);
27. while(t--){
28. scanf("%d",&n);
29. // setCache.clear();
30. stack<int> s;
31. while(n--){
32. scanf(" %s",&cmd);
33. if(cmd[0]=='P')s.push(getID(Set()));
34. else if(cmd[0]=='D')s.push(s.top());
35. else{
36. Set s1=setCache[s.top()];s.pop();
37. Set s2=setCache[s.top()];s.pop();
38. Set x;
39. if(cmd[0]=='U')set\_union(ALL(s1),ALL(s2),INS(x));
40. if(cmd[0]=='I')set\_intersection(ALL(s1),ALL(s2),INS(x));
41. if(cmd[0]=='A'){ x=s2; x.insert(getID(s1)); }
42. s.push(getID(x));
43. }
44. printf("%d\n",setCache[s.top()].size());
45. }
46. puts("\*\*\*");
47. }
48. }

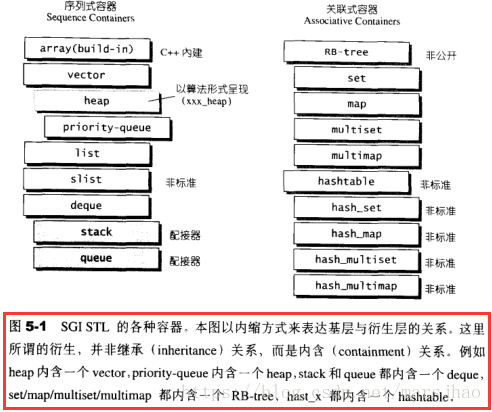
**STL关联容器总结**

2018年08月26日 20:49:58 [marsjhao](https://me.csdn.net/marsjhao) 阅读数 405

版权声明：本文为博主原创文章，遵循[CC 4.0 by-sa](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/marsjhao/article/details/82084322>

**一、关联式容器**



STL 关联容器分为 set（集合）和 map（映射表）两大类，及其衍生体 multiset 和 multimap。这些容器的底层机制均以 RB-tree（红黑树）实现。RB-tree 也是一个独立容器，但并不开放使用。

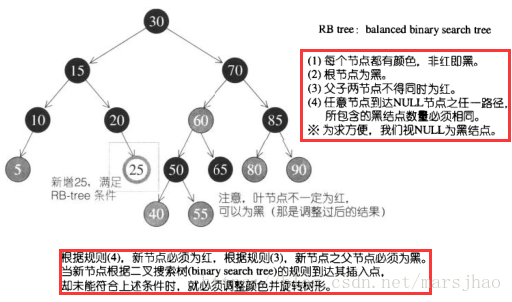
SGI STL 还提供一个不在标准规格的关联式容器 hash\_table（散列表），以及以 hash\_table 为底层机制而完成的 hash\_set散列集合、hash\_map散列映射表、hash\_multiset散列多键集合、hash\_multimap散列多键映射表。

关联容器，类似关联式数据库，每个元素都有一个键值key和一个实值value。关联式容器的内部结构是一个平衡二叉树，包括 AVL-tree、RB-tree、AA-tree，STl 中运用的是 RB-tree红黑树。

**二、红黑树 RB-tree**

1. 基本性质

* RB-tree 是一个二叉搜索树。
* 每个节点只能是红色或黑色（图中深色代表黑色，浅色代表红色）。
* 根节点为黑色。
* 红色节点的子节点必须为黑。
* 任一节点至NULL（树的尾端）的任何路径，包含黑色节点数相同。



**三、set**

1. 基本性质

* set 元素的键值就是实值，实值就是键值。
* set 不允许两个元素有相同的键值。
* 不可以通过 set 的迭代器改变其元素值，即 set iterator 是一种 constant iterators。
* 当对 set 进行插入删除操作后，原有的迭代器依然有效，被删除元素的迭代器除外。

2. STL 提供一组 set / multiset 相关算法，详见算法部分笔记。

**四、map**

1. 基本性质

* map 的所有元素都是 pair，同时拥有实值value和键值key。pair 的第一元素为键值，第二元素为实值。
* map 不允许两个元素拥有相同的键值。
* 不可以通过 map 的迭代器改变其元素的键值，可以修改元素的实值。
* 当对 map 元素进行增加或删除操作后，原有的迭代器依然有效，被删除的元素的迭代器除外。

**五、multiset、multimap**

multiset 和 multimap 的特性及用法与 set 和 map 完全相同，唯一差别在于允许键值重复。

# stl关联容器

2016年04月15日 21:43:35 [weixiaopaj](https://me.csdn.net/weixiaopaj) 阅读数 105

版权声明：本文为博主原创文章，遵循[CC 4.0 by-sa](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/weixiaopaj/article/details/51164969>

# stl关联容器

* 总共有8个关联容器，体现在三个不同的纬度上：   
  1)是一个map,还是一个set   
  2)是否允许重复的关键字   
  3)是否需要按照顺序保存

## 使用关联容器

map<string,size\_t> word\_count;*//string到size\_t的空map*

string word;

while(cin>>word)

++word\_count[word];*//提取word的计数器并将其加1*

for(const auto &w : word\_count)*//对map中的每个元素*

cout<<w.first<<" occurs "<<w.second

<< ((w.second > 1)?"times":"time") <<endl;

* 使用下标运算符时，如果不存在，则创建一个新的元素，值初始化

map<string,size\_t> word\_count;

set<string> excludes = {"The","But","And","Or","An","A"};

string word;

while(cin>>word)

if(excludes.find(word) == exclude.end())

++word\_count[word];

## 定义关联容器

map<string,size\_t> word\_count;*//空容器*

*//列表初始化*

set<string> excludes = {"The","But","And","Or","An","A"};

*//三个元素；authors将映射为名*

map<string,string> authors = {{"Joyce","James"}};

### 关键字类型的限制

* “行为正常”的<运算符的要求   
  1. 两个关键字不能够同时“小于等于”对方
  2. 如果k1“小于等于”k2,且k2“小于等于”k3,则k1“小于等于”k3
  3. 如果两个关键字均不“小于等于”对方，则两者等价
* 比较操作也是容器类型的一部分

multiset<Sales\_data,decltype(compareIsbn)\*> bookstore(compareIsbn);

### pair类型

* pair上的操作

pair<T1,T2> p;*//p是一个pair，两个类型分别为T1和T2的成员都进行了值初始化*

pair<T1,T2> p(v1,v2);*//p是一个成员类型为T1和T2的pair,first和second分别使用v1和v2初始化*

pair<T1,T2> p = {v1,v2};

make\_pair(v1,v2);*//返回一个用v1,v2初始化的pair,类型由v1和v2推断*

## 关联容器操作

* pair的关键字是const的,我们可以改变pair的值，但是无法改变关键字成员的值

auto map\_it = word\_count.begin();

*//\*map\_it是指向pair<const string,size\_t>对象的引用*

cout<<map\_it->first;

cout<<" "<<map\_it->second;

map\_it->first = "new key";*//错误！！！，first是const的*

++map\_it->second;*//正确*

* set的迭代器是const的

### 遍历迭代器

auto map\_it = word\_count.cbegin();

while(map\_it!=word\_count.cend()){

cout<<map\_it->first<<" occurs "<<map\_it->second <<" times "<<endl;

++map\_it;

}

### 添加元素

map<string,size\_t> word\_count;*//从string到size\_t的空map*

string word;

while(cin>>word){

*//插入一个元素，关键字等于word,值为1*

auto ret = word\_count.insert({word,1});

if(!ret.second)*//word已经在word\_count中*

++ret.first->second;

}

* 如果是multimap类型，insert函数返回是指向新元素的迭代器,无需返回bool值

### 删除元素

* 有三种erase方法可以调用   
  1. c.erase(k)   
     从c中删除每个关键字为k的元素。返回一个size\_type值，指出删除元素的数量
  2. c.erase(p)   
     从c中删除迭代器p指向的元素
  3. c.erase(b,e)   
     从c中删除迭代器对b和e所表示的范围中的元素。返回e

### map的下表操作

* 下标 VS at   
  1. c[k] 返回关键字为k的元素；如果k不在c中，添加一个关键字为k的元素，对其进行值初始化
  2. c.at(k) 若k不在c中，抛出一个out\_of\_range异常

### 访问元素

* find / count
* 对于有序容器来说,lower\_bound / upper\_bound
* 在mutlimap和multiset中查找元素

string search\_item("Alain de Botton");

auto entries = authors.count(search\_item);

auto iter = authors.find(search\_item);

while(entries){

cout<<iter->second<<endl;

++iter;

*--entries;*

}

for(auto pos = c.equal\_range(search\_item);pos.first!=pos.second;++pos.first)

cout<<pos.first->second<<endl;

## 无序容器

* 通常，无序容器和有序容器可以互相替换

### 无序容器对关键字类型的要求

1. haser函数
2. == 运算符

**STL—— 关联容器**

2014年07月05日 17:02:21 [demystify](https://me.csdn.net/xiaoxiaoyusheng2012) 阅读数 377

版权声明：本文为博主原创文章，遵循[CC 4.0 by-sa](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

本文链接：<https://blog.csdn.net/xiaoxiaoyusheng2012/article/details/28672787>

1、关联容器与顺序容器的本质区别

     关联容器通过键（key）存储和读取元素，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

2、关联容器的类型

     map 、set、multimap、multiset

3、pair类型

     pair类型的比较：

     p1<p2 :两个pair对象之间的小于运算，其定义遵循字典次序：如果p1.first < p2.first 或者 !(p1.first < p2.first) && (p1.second< p2.second)，则返回true;

    p1 == p2: 如果两个pair对象的first和second成员依次相等，则这两个对象相等，(该运算使用其元素的==操作符)

4、map类型

map<k, v> m; //创建空的map对象，其键和值得类型分别为k和v;

map<k, v> m(m2);

map<k, v> m(b,e);

注意：键值应能够比较

map<K, V>::key\_type   //在map中，用作索引的键的类型

map<K, V>::mapped\_type  //在map中，键所关联的值得类型

map<K, V>::value\_type  //一个pair类型，它的first元素具有const map<K, V>::key\_type类型，而second元素则为map<K, V>::mapped\_type类型。

注意：map的value\_type是pair类型，它的值成员可以修改，但键成员不能修改（类型为const map<K,V>::key\_type,只读）。

           map迭代器进行解引用将产生pair类型的对象

     <1>、map类型的操作

     \*\*1、通过下标访问map对象

     注意:用下标访问不存在的元素将导致在map容器中添加一个新的元素，它的键即为该下标值。

     例：map<string, int> word\_count;

            word\_count["Anna"] = 1;

            将有以下操作：

            在word\_count中查找键为Anna的元素，没有找到；

            将一个新的键-值对插入到word\_count中，它的键是const string类型，保存为Anna, 而它的值初始化为0；

            读取新插入的元素，并将它的值赋为1；

     map下标操作返回的左值是特定键所关联的值。

    <2>、map::insert的使用

           注意insert的返回值；

           是一个pair类型，first成员是map迭代器，second成员是bool型，first成员指向map中已存在的值，second标识插入操作是否成功。

    <3>、查询操作

         m.count(k);//返回m中k键出现的次数。

         m.find(k);//查找m容器中是否存在k键，存在返回指向该元素的迭代器，不存在，返回超出末端的迭代器。

    <4>、从map中删除元素

       m.erase(k);//删除m中键为k的元素，返回size\_type类型的值，表示删除的元素的个数。

       m.erase(p);//从m中删除迭代器p所指向的元素。p必须指向m中确实存在的元素，而且不能等于m.end(),返回void类型。

       m.erase(b,e);//从m中删除一段范围内的元素，该范围由迭代器对b和e标记。b和e必须标记m中的一段有效范围。返回void类型。

5、set类型

     map是键-值对的集合，set容器是单纯的键的集合

     set支持大部分的map操作，但是set不支持下标操作符，set容器中存储的键必须为唯一的，而且不能修改。

6、multimap和multiset类型

     1\*\*、注意multimap不支持下标操作（因为某个键可能对应多个值）

     2\*\*、在multimap和multiset中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。

     3\*\*、在multimap和multiset中查找元素所使用的三种方法

      <1>、find和count

      <2>、lower\_bound和upper\_bound

              m.lower\_bound(k):返回一个迭代器，指向键不小于k的第一个元素

              m.upper\_bound(k);返回一个迭代器，指向键大于k的第一个元素

              m.equal\_range(k);返回一个迭代器的pair对象，它的first成员等价于m.lower\_bound(k),而second成员等价于

              m\_upper\_bound(k);

     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。

     如果查找的键不在multimap中，则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。

     例：

1. typedef multimap<string, string>::iterator author\_it;
2. author\_it beg = author.lower\_bound("Barth,John");
3. author\_it end = author.upper\_bound("Barth,John");
5. while(beg != end)
6. {
7. cout<<beg->first <<"---"<<beg->second<<endl;
8. beg++;
9. }

     <3>、equal\_range函数

1. pair<author\_it, author\_it> pos = author.equal\_range("Barth,John");
2. while(pos.first != pos.second)
3. {
4. cout<<pos.first->first<<"---"<<pos.first->second<<endl;
5. ++pos.first;
6. }

# [C++中的STL中map用法详解](https://www.cnblogs.com/fnlingnzb-learner/p/5833051.html)

Map是STL的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在map中出现一次，第二个可能称为该关键字的值）的数据 处理能力，由于这个特性，它完成有可能在我们处理一对一数据的时候，在编程上提供快速通道。这里说下map内部数据的组织，map内部自建一颗红黑树(一 种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的，后边我们会见识到有序的好处。

1、map简介

map是一类关联式容器。它的特点是增加和删除节点对迭代器的影响很小，除了那个操作节点，对其他的节点都没有什么影响。

对于迭代器来说，可以修改实值，而不能修改key。

2、map的功能

自动建立Key － value的对应。key 和 value可以是任意你需要的类型。

**根据key值快速查找记录，查找的复杂度基本是Log**(N)，如果有1000个记录，最多查找10次，1,000,000个记录，最多查找20次。

快速插入Key -Value 记录。

快速删除记录

根据Key 修改value记录。

遍历所有记录。

3、使用map

使用map得包含map类所在的头文件

#include <map>  //注意，STL头文件没有扩展名.h

map对象是模板类，需要关键字和存储对象两个模板参数：

std:map<**int**,string> personnel;

这样就定义了一个用int作为索引,并拥有相关联的指向string的指针.

为了使用方便，可以对模板类进行一下类型定义，

**typedef**map<**int**,CString> UDT\_MAP\_INT\_CSTRING;

UDT\_MAP\_INT\_CSTRING enumMap;

4、       map的构造函数

map共提供了6个构造函数，这块涉及到内存分配器这些东西，略过不表，在下面我们将接触到一些map的构造方法，这里要说下的就是，我们通常用如下方法构造一个map：

map<int, string> mapStudent;

5、     数据的插入

在构造map容器后，我们就可以往里面插入数据了。这里讲三种插入数据的方法：

第一种：用insert函数插入pair数据，下面举例说明(以下代码虽然是随手写的，应该可以在VC和GCC下编译通过，大家可以运行下看什么效果，在VC下请加入这条语句，屏蔽4786警告 ＃pragma warning (disable:4786) )

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //数据的插入--第一种：用insert函数插入pair数据
2. #include <map>
4. #include <string>
6. #include <iostream>
8. **using namespace std;**
10. int main()
12. {
14. map<int, string> mapStudent;
16. mapStudent.insert(pair<int, string>(1, "student\_one"));
18. mapStudent.insert(pair<int, string>(2, "student\_two"));
20. mapStudent.insert(pair<int, string>(3, "student\_three"));
22. map<int, string>::iterator iter;
24. **for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)**
26. cout<<iter->first<<' '<<iter->second<<endl;
28. }

第二种：用insert函数插入value\_type数据，下面举例说明

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //第二种：用insert函数插入value\_type数据，下面举例说明
3. #include <map>
5. #include <string>
7. #include <iostream>
9. **using namespace std;**
11. int main()
13. {
15. map<int, string> mapStudent;
17. mapStudent.insert(map<int, string>::value\_type (1, "student\_one"));
19. mapStudent.insert(map<int, string>::value\_type (2, "student\_two"));
21. mapStudent.insert(map<int, string>::value\_type (3, "student\_three"));
23. map<int, string>::iterator iter;
25. **for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)**
27. cout<<iter->first<<' '<<iter->second<<endl;
29. }

第三种：用数组方式插入数据，下面举例说明

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //第三种：用数组方式插入数据，下面举例说明
3. #include <map>
5. #include <string>
7. #include <iostream>
9. **using namespace std;**
11. int main()
13. {
15. map<int, string> mapStudent;
17. mapStudent[1] = "student\_one";
19. mapStudent[2] = "student\_two";
21. mapStudent[3] = "student\_three";
23. map<int, string>::iterator iter;
25. **for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)**
27. cout<<iter->first<<' '<<iter->second<<endl;
29. }

以上三种用法，虽然都可以实现数据的插入，但是它们是有区别的，当然了第一种和第二种在效果上是完成一样的，用insert函数插入数据，在数据的 插入上涉及到集合的唯一性这个概念，即当map中有这个关键字时，insert操作是插入数据不了的，但是用数组方式就不同了，它可以覆盖以前该关键字对 应的值，用程序说明

mapStudent.insert(map<int, string>::value\_type (1, "student\_one"));

mapStudent.insert(map<int, string>::value\_type (1, "student\_two"));

上面这两条语句执行后，map中1这个关键字对应的值是“student\_one”，第二条语句并没有生效，那么这就涉及到我们怎么知道insert语句是否插入成功的问题了，可以用pair来获得是否插入成功，程序如下

pair<map<int, string>::iterator, bool> Insert\_Pair;

Insert\_Pair = mapStudent.insert(map<int, string>::value\_type (1, "student\_one"));

我们通过pair的第二个变量来知道是否插入成功，它的第一个变量返回的是一个map的迭代器，如果插入成功的话Insert\_Pair.second应该是true的，否则为false。

下面给出完成代码，演示插入成功与否问题

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //验证插入函数的作用效果
2. #include <map>
4. #include <string>
6. #include <iostream>
8. **using namespace std;**
10. int main()
12. {
14. map<int, string> mapStudent;
16. pair<map<int, string>::iterator, bool> Insert\_Pair;
18. Insert\_Pair = mapStudent.insert(pair<int, string>(1, "student\_one"));
20. **if(Insert\_Pair.second == true)**
22. cout<<"Insert Successfully"<<endl;
24. **else**
26. cout<<"Insert Failure"<<endl;
28. Insert\_Pair = mapStudent.insert(pair<int, string>(1, "student\_two"));
30. **if(Insert\_Pair.second == true)**
32. cout<<"Insert Successfully"<<endl;
34. **else**
36. cout<<"Insert Failure"<<endl;
38. map<int, string>::iterator iter;
40. **for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)**
42. cout<<iter->first<<' '<<iter->second<<endl;
44. }

大家可以用如下程序，看下用数组插入在数据覆盖上的效果

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //验证数组形式插入数据的效果
3. #include <map>
5. #include <string>
7. #include <iostream>
9. **using namespace std;**
11. int main()
13. {
15. map<int, string> mapStudent;
17. mapStudent[1] = "student\_one";
19. mapStudent[1] = "student\_two";
21. mapStudent[2] = "student\_three";
23. map<int, string>::iterator iter;
25. **for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)**
27. cout<<iter->first<<' '<<iter->second<<endl;
28. }

6、      map的大小

在往map里面插入了数据，我们怎么知道当前已经插入了多少数据呢，可以用size函数，用法如下：

Int nSize = mapStudent.size();

7、     数据的遍历

这里也提供三种方法，对map进行遍历

第一种：应用前向迭代器，上面举例程序中到处都是了，略过不表

第二种：应用反相迭代器，下面举例说明，要体会效果，请自个动手运行程序

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //第二种，利用反向迭代器
3. #include <map>
5. #include <string>
7. #include <iostream>
9. **using namespace std;**
11. int main()
13. {
15. map<int, string> mapStudent;
17. mapStudent.insert(pair<int, string>(1, "student\_one"));
19. mapStudent.insert(pair<int, string>(2, "student\_two"));
21. mapStudent.insert(pair<int, string>(3, "student\_three"));
23. map<int, string>::reverse\_iterator iter;
25. **for(iter = mapStudent.rbegin(); iter != mapStudent.rend(); iter++)**
27. cout<<iter->first<<"  "<<iter->second<<endl;
29. }

第三种，用数组的形式，程序说明如下：

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //第三种：用数组方式，程序说明如下
3. #include <map>
5. #include <string>
7. #include <iostream>
9. **using namespace std;**
11. int main()
13. {
15. map<int, string> mapStudent;
17. mapStudent.insert(pair<int, string>(1, "student\_one"));
19. mapStudent.insert(pair<int, string>(2, "student\_two"));
21. mapStudent.insert(pair<int, string>(3, "student\_three"));
23. int nSize = mapStudent.size();
25. //此处应注意，应该是 for(int nindex = 1; nindex <= nSize; nindex++)
26. //而不是 for(int nindex = 0; nindex < nSize; nindex++)
28. **for(int nindex = 1; nindex <= nSize; nindex++)**
30. cout<<mapStudent[nindex]<<endl;
32. }

8、    查找并获取map中的元素（包括判定这个关键字是否在map中出现）

在这里我们将体会，map在数据插入时保证有序的好处。

要判定一个数据（关键字）是否在map中出现的方法比较多，这里标题虽然是数据的查找，在这里将穿插着大量的map基本用法。

这里给出三种数据查找方法

第一种：用count函数来判定关键字是否出现，其缺点是无法定位数据出现位置,由于map的特性，一对一的映射关系，就决定了count函数的返回值只有两个，要么是0，要么是1，出现的情况，当然是返回1了

第二种：用find函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器，如果map中没有要查找的数据，它返回的迭代器等于end函数返回的迭代器。

**查找map中是否包含某个关键字条目用find**()**方法，传入的参数是要查找的key，在这里需要提到的是begin**()**和end**()两个成员，

分别代表map对象中第一个条目和最后一个条目，这两个数据的类型是iterator.

程序说明

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. #include <map>
3. #include <string>
5. #include <iostream>
7. **using namespace std;**
9. int main()
11. {
13. map<int, string> mapStudent;
15. mapStudent.insert(pair<int, string>(1, "student\_one"));
17. mapStudent.insert(pair<int, string>(2, "student\_two"));
19. mapStudent.insert(pair<int, string>(3, "student\_three"));
21. map<int, string>::iterator iter;
23. iter = mapStudent.find(1);
25. **if(iter != mapStudent.end())**
27. cout<<"Find, the value is "<<iter->second<<endl;
29. **else**
31. cout<<"Do not Find"<<endl;
33. **return 0;**
34. }

通过map对象的方法获取的iterator数据类型是一个std::pair对象，包括两个数据 iterator->first和 iterator->second分别代表关键字和存储的数据。

第三种：这个方法用来判定数据是否出现，是显得笨了点，但是，我打算在这里讲解

lower\_bound函数用法，这个函数用来返回要查找关键字的下界(是一个迭代器)

upper\_bound函数用法，这个函数用来返回要查找关键字的上界(是一个迭代器)

例如：map中已经插入了1，2，3，4的话，如果lower\_bound(2)的话，返回的2，而upper-bound（2）的话，返回的就是3

Equal\_range函数返回一个pair，pair里面第一个变量是Lower\_bound返回的迭代器，pair里面第二个迭代器是Upper\_bound返回的迭代器，如果这两个迭代器相等的话，则说明map中不出现这个关键字，

程序说明

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. #include <map>
3. #include <string>
5. #include <iostream>
7. **using namespace std;**
9. int main()
11. {
13. map<int, string> mapStudent;
15. mapStudent[1] = "student\_one";
17. mapStudent[3] = "student\_three";
19. mapStudent[5] = "student\_five";
21. map<int, string>::iterator iter;
23. iter = mapStudent.lower\_bound(1);
25. //返回的是下界1的迭代器
27. cout<<iter->second<<endl;
29. iter = mapStudent.lower\_bound(2);
31. //返回的是下界3的迭代器
33. cout<<iter->second<<endl;
35. iter = mapStudent.lower\_bound(3);
37. //返回的是下界3的迭代器
39. cout<<iter->second<<endl;
41. iter = mapStudent.upper\_bound(2);
43. //返回的是上界3的迭代器
45. cout<<iter->second<<endl;
47. iter = mapStudent.upper\_bound(3);
49. //返回的是上界5的迭代器
51. cout<<iter->second<<endl;
53. pair<map<int, string>::iterator, map<int, string>::iterator> mappair;
55. mappair = mapStudent.equal\_range(2);
57. **if(mappair.first == mappair.second)**
59. cout<<"Do not Find"<<endl;
61. **else**
63. cout<<"Find"<<endl;
65. mappair = mapStudent.equal\_range(3);
67. **if(mappair.first == mappair.second)**
69. cout<<"Do not Find"<<endl;
71. **else**
73. cout<<"Find"<<endl;
75. **return 0;**
76. }

9、    从map中删除元素

移除某个map中某个条目用erase（）

该成员方法的定义如下：

iterator erase（iterator it);//通过一个条目对象删除

iterator erase（iterator first，iterator last）//删除一个范围

size\_type erase(const Key&key);//通过关键字删除

clear()就相当于enumMap.erase(enumMap.begin(),enumMap.end());

这里要用到erase函数，它有三个重载了的函数，下面在例子中详细说明它们的用法

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. #include <map>
3. #include <string>
5. #include <iostream>
7. **using namespace std;**
9. int main()
11. {
13. map<int, string> mapStudent;
15. mapStudent.insert(pair<int, string>(1, "student\_one"));
17. mapStudent.insert(pair<int, string>(2, "student\_two"));
19. mapStudent.insert(pair<int, string>(3, "student\_three"));
21. //如果你要演示输出效果，请选择以下的一种，你看到的效果会比较好
23. //如果要删除1,用迭代器删除
25. map<int, string>::iterator iter;
27. iter = mapStudent.find(1);
29. mapStudent.erase(iter);
31. //如果要删除1，用关键字删除
33. int n = mapStudent.erase(1);//如果删除了会返回1，否则返回0
35. //用迭代器，成片的删除
37. //一下代码把整个map清空
39. mapStudent.erase( mapStudent.begin(), mapStudent.end() );
41. //成片删除要注意的是，也是STL的特性，删除区间是一个前闭后开的集合
43. //自个加上遍历代码，打印输出吧
45. }

10、    map中的swap用法

map中的swap不是一个容器中的元素交换，而是两个容器所有元素的交换。

11、     排序 ·  map中的sort问题

map中的元素是自动按Key升序排序，所以不能对map用sort函数；

这里要讲的是一点比较高深的用法了,排序问题，STL中默认是采用小于号来排序的，以上代码在排序上是不存在任何问题的，因为上面的关键字是int 型，它本身支持小于号运算，在一些特殊情况，比如关键字是一个结构体，涉及到排序就会出现问题，因为它没有小于号操作，insert等函数在编译的时候过 不去，下面给出两个方法解决这个问题。

第一种：小于号重载，程序举例。

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. #include <iostream>
2. #include <string>
3. #include <map>
4. **using namespace std;**
6. **typedef struct tagStudentinfo**
8. {
10. int      niD;
12. string   strName;
14. bool operator < (tagStudentinfo **const& \_A) const**
16. {     //这个函数指定排序策略，按niD排序，如果niD相等的话，按strName排序
18. **if(niD < \_A.niD) return true;**
20. **if(niD == \_A.niD)**
22. **return strName.compare(\_A.strName) < 0;**
24. **return false;**
26. }
28. }Studentinfo, \*PStudentinfo; //学生信息
30. int main()
32. {
34. int nSize;   //用学生信息映射分数
36. map<Studentinfo, int>mapStudent;
38. map<Studentinfo, int>::iterator iter;
40. Studentinfo studentinfo;
42. studentinfo.niD = 1;
44. studentinfo.strName = "student\_one";
46. mapStudent.insert(pair<Studentinfo, int>(studentinfo, 90));
48. studentinfo.niD = 2;
50. studentinfo.strName = "student\_two";
52. mapStudent.insert(pair<Studentinfo, int>(studentinfo, 80));
54. **for (iter=mapStudent.begin(); iter!=mapStudent.end(); iter++)**
56. cout<<iter->first.niD<<' '<<iter->first.strName<<' '<<iter->second<<endl;
58. **return 0;**
59. }

第二种：仿函数的应用，这个时候结构体中没有直接的小于号重载，程序说明

**[cpp]** [view plain](http://blog.csdn.net/shawn_hou/article/details/38035577) [copy](http://blog.csdn.net/shawn_hou/article/details/38035577)

1. //第二种：仿函数的应用，这个时候结构体中没有直接的小于号重载，程序说明
3. #include <iostream>
5. #include <map>
7. #include <string>
9. **using namespace std;**
11. **typedef struct tagStudentinfo**
13. {
15. int      niD;
17. string   strName;
19. }Studentinfo, \*PStudentinfo; //学生信息
21. **class sort**
23. {
25. **public:**
27. bool operator() (Studentinfo **const &\_A, Studentinfo const &\_B) const**
29. {
31. **if(\_A.niD < \_B.niD)**
33. **return true;**
35. **if(\_A.niD == \_B.niD)**
37. **return \_A.strName.compare(\_B.strName) < 0;**
39. **return false;**
41. }
42. };
44. int main()
46. {   //用学生信息映射分数
48. map<Studentinfo, int, sort>mapStudent;
50. map<Studentinfo, int>::iterator iter;
52. Studentinfo studentinfo;
54. studentinfo.niD = 1;
56. studentinfo.strName = "student\_one";
58. mapStudent.insert(pair<Studentinfo, int>(studentinfo, 90));
60. studentinfo.niD = 2;
62. studentinfo.strName = "student\_two";
64. mapStudent.insert(pair<Studentinfo, int>(studentinfo, 80));
66. **for (iter=mapStudent.begin(); iter!=mapStudent.end(); iter++)**
68. cout<<iter->first.niD<<' '<<iter->first.strName<<' '<<iter->second<<endl;
69. }

由于STL是一个统一的整体，map的很多用法都和STL中其它的东西结合在一起，比如在排序上，这里默认用的是小于号，即less<>，如果要从大到小排序呢，这里涉及到的东西很多，在此无法一一加以说明。

还要说明的是，map中由于它内部有序，由红黑树保证，因此很多函数执行的时间复杂度都是log2N的，如果用map函数可以实现的功能，而STL Algorithm也可以完成该功能，建议用map自带函数，效率高一些。

下面说下，map在空间上的特性，否则，估计你用起来会有时候表现的比较郁闷，由于map的每个数据对应红黑树上的一个节点，这个节点在不保存你的 数据时，是占用16个字节的，一个父节点指针，左右孩子指针，还有一个枚举值（标示红黑的，相当于平衡二叉树中的平衡因子），我想大家应该知道，这些地方 很费内存了吧，不说了……

12、

      map的基本操作函数：

     C++ maps是一种关联式容器，包含“关键字/值”对

**begin**()         返回指向map头部的迭代器

**clear**(）        删除所有元素

**count**()         返回指定元素出现的次数

**empty**()         如果map为空则返回true

**end**()           返回指向map末尾的迭代器

**equal\_range**()   返回特殊条目的迭代器对

**erase**()         删除一个元素

**find**()          查找一个元素

**get\_allocator**() 返回map的配置器

**insert**()        插入元素

**key\_comp**()      返回比较元素key的函数

**lower\_bound**()   返回键值>=给定元素的第一个位置

**max\_size**()      返回可以容纳的最大元素个数

**rbegin**()        返回一个指向map尾部的逆向迭代器

**rend**()          返回一个指向map头部的逆向迭代器

**size**()          返回map中元素的个数

**swap**()           交换两个map

**upper\_bound**()    返回键值>给定元素的第一个位置

**value\_comp**()     返回比较元素value的函数