

Fork me on GitHub

幕三少

我们之所以会心累，就是常常徘徊在坚持和放弃之间，举棋不定。我们之所以会烦恼，就是记性太好，该记的，不该记的都会留在记忆里。我们之所以会痛苦，就是追求的太多。我们之所以不快乐，就是计较的太多，不是我们拥有的太少，而是我们计较的太多。

博客园

首页

新随笔

联系

订阅

管理

随笔 - 334 文章 - 1 评论 - 778

昵称：幕三少
园龄：9年2个月
粉丝：412
关注：40
+加关注

<	2018年9月						>
日	一	二	三	四	五	六	
26	27	28	29	30	31	1	
2	3	4	5	6	7	8	
9	10	11	12	13	14	15	
16	17	18	19	20	21	22	
23	24	25	26	27	28	29	
30	1	2	3	4	5	6	

常用链接

我的随笔

7

0

我的评论

1 vector

我的参与

向量 相当于一个数组

在内存中分配一块连续的内存空间进行存储。支持不指定vector

好文要顶

关注我

收藏该文



Fork me on GitHub

最新评论

我的标签

我的标签

WPF(16)

C#(15)

设计模式(6)

反射(4)

基础(3)

LINQ(3)

属性(2)

.net(2)

MVC(2)

SQLite(2)

更多

随笔分类

Android(12)

常大的内存空间预备进行存储，即capacituy () 函数返回的大小，当超过此分配的空间时再整体重新分配。一块内存存储，这给人以vector可以不指定vector即一个连续内存的大小的感觉。通常此默认的内存分配能完成大部分情况下的存储。

优点：(1) 不指定一块内存大小的数组的连续存储，即可以像数组一样操作，但可以对此数组
进行动态操作。通常体现在push_back() pop_back()

- (2) 随机访问方便，即支持[]操作符和vector.at()
- (3) 节省空间。

缺点：(1) 在内部进行插入删除操作效率低。
(2) 只能在vector的最后进行push和pop，不能在vector的头进行push和pop。
(3) 当动态添加的数据超过vector默认分配的大小时要进行整体的重新分配、拷贝与释放

2 list

双向链表

每一个结点都包括一个信息快Info、一个前驱指针Pre、一个后驱指针Post。可以不分配必须的内存大小方便的进行添加和删除操作。使用的是非连续的内存空间进行存储。

优点：(1) 不使用连续内存完成动态操作。
(2) 在内部方便的进行插入和删除操作
(3) 可在两端进行push、pop

缺点：(1) 不能进行内部的随机访问，即不支持[]操作符和vector.at()
(2) 相对于verctor占用内存多

3 deque

双端队列 double-end queue

deque是在功能上合并了vector和list。

优点：(1) 随机访问方便，即支持[]操作符和vector.at()
(2) 在内部方便的进行插入和删除操作
(3) 可在两端进行push、pop

缺点：(1) 占用内存多

7 0

使用区别：

1 如果你需要高效的随即存取，而不在乎插入和删除的效率，则应使

2 如果你需要大量的插入和删除，而不关心随即存取，则应使

3 如果你需要随即存取，而且关心两端数据的插入和删除，则应使

[好文要顶](#)

[关注我](#)

[收藏该文](#)



评论

大话

收藏

Fork me on GitHub

[Anjularjs](#)[ASP.NET\(11\)](#)[asp.net MVC](#)[C\(5\)](#)[C#\(48\)](#)[C++\(7\)](#)[EasyUI](#)[Extjs\(18\)](#)[HTML\(1\)](#)[Java\(13\)](#)[JavaScript\(22\)](#)[JQUERY\(7\)](#)[Linq\(6\)](#)[MVC\(2\)](#)[mysql\(2\)](#)[nodejs\(3\)](#)

C++STL中vector容器的用法

<http://xiamaogeng.blog.163.com/blog/static/1670023742010102494039234/>

vector是C++标准模板库中的部分内容，它是一个多功能的，能够操作多种数据结构和算法的模板类和函数库。vector之所以被认为是一个容器，是因为它能够像容器一样存放各种类型的对象，简单地说vector是一个能够存放任意类型的动态数组，能够增加和压缩数据。为了可以使用vector，必须在你的头文件中包含下面的代码：

```
#include <vector>
```

vector属于std命名域的，因此需要通过命名限定，如下完成你的代码：

```
using std::vector;    vector<int> v;
```

或者连在一起，使用全名：

```
std::vector<int> v;
```

建议使用全局的命名域方式：

```
using namespace std;
```

1. vector的声明

vector<ElemType> c; 创建一个空的vector

vector<ElemType> c1(c2); 创建一个vector c1，并用c2去初始化c1

vector<ElemType> c(n); 创建一个含有n个ElemType类型数据的vector;

vector<ElemType> c(n, elem); 创建一个含有n个ElemType类型数据的vector,并全部初始化为elem;

c.>vector<ElemType>(); 销毁所有数据,释放资源;

2. vector容器中常用的函数。(c为一个容器对象)

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

OpenGL(1)

Oracle(3)

python(8)

React

Revit(3)

Ruby(2)

SQLite(6)

SqlServer(4)

VS2010(9)

WCF(7)

webapi(3)

winform(3)

WPF(42)

程序人生(13)

工具使用技巧(36)

其他(15)

c.push_back(elem); 在容器最后位置添加一个元素elem

c.pop_back(); 删除容器最后位置处的元素

c.at(index); 返回指定index位置处的元素

c.begin(); 返回指向容器最开始位置数据的指针

c.end(); 返回指向容器最后一个数据单元的指针+1

c.front(); 返回容器最开始单元数据的引用

c.back(); 返回容器最后一个数据的引用

c.max_size(); 返回容器的最大容量

c.size(); 返回当前容器中实际存放元素的个数

c.capacity(); 同c.size()

c.resize(); 重新设置vector的容量

c.reserve(); 同c.resize()

c.erase(p); 删除指针p指向位置的数据，返回下指向下一个数据位置的指针（迭代器）

c.erase(begin,end) 删除begin,end区间的数据，返回指向下一个数据位置的指针（迭代器）

c.clear(); 清除所有数据

c.rbegin(); 将vector反转后的开始指针返回（其实就是原来的end-1）

c.rend(); 将vector反转后的结束指针返回（其实就是原来的begin-1）

c.empty(); 判断容器是否为空，若为空返回true，否则返回false

c1.swap(c2); 交换两个容器中的数据

c.insert(p,elem); 在指针p指向的位置插入数据elem,返回指向elem位置的指针

c.insert(p,n,elem); 在位置p插入n个elem数据，无返回值

c.insert(p,begin,end) 在位置p插入在区间[begin,end)的数据

7

0

3. vector中的操作

好文要顶

关注我

收藏该文



Fork me on GitHub

设计模式(9)

正则(1)

随笔档案

2018年9月 (1)

2018年8月 (1)

2018年7月 (3)

2018年6月 (6)

2018年5月 (2)

2018年4月 (4)

2018年3月 (13)

2018年2月 (3)

2018年1月 (4)

2017年12月 (1)

2017年10月 (1)

2017年8月 (5)

2017年7月 (3)

operator[] 如: c.[i];

同at()函数的作用相同，即取容器中的数据。

在上大致讲述了vector类中所含有的函数和操作，下面继续讨论如何使用vector容器；

1.数据的输入和删除。push_back () 与pop_back ()

 C++ STL中vector容器的用法 - 夏茂庚 - 夏茂庚

2.元素的访问

 C++ STL中vector容器的用法 - 夏茂庚 - 夏茂庚

3.排序和查询

 C++ STL中vector容器的用法 - 夏茂庚 - 夏茂庚

4.二维容器

 C++ STL中vector容器的用法 - 夏茂庚 - 夏茂庚

C++ STLList队列用法 (实例)

<http://www.cnblogs.com/madlas/articles/1364503.html>

C++ STL List队列用法 (实例)

2007-12-15 12:54

```
#include <iostream>
#include <list>
#include <numeric>
#include <algorithm>

using namespace std;

//创建一个list容器的实例LISTINT
typedef list<int> LISTINT;

//创建一个list容器的实例LISTCHAR
typedef list<char> LISTCHAR;
```

7

0

好文要顶

关注我

收藏该文





2017年6月 (8)

2017年5月 (8)

2017年4月 (2)

2017年3月 (1)

2017年2月 (9)

2017年1月 (2)

2016年12月 (1)

2016年11月 (1)

2016年10月 (1)

2016年9月 (1)

2016年7月 (2)

2016年6月 (4)

2016年5月 (1)

2016年4月 (1)

2016年3月 (2)

2016年1月 (1)

```

void main(void)
{
    //-----
    //用list容器处理整型数据
    //-----
    //用LISTINT创建一个名为listOne的list对象
    LISTINT listOne;
    //声明i为迭代器
    LISTINT::iterator i;

    //从前面向listOne容器中添加数据
    listOne.push_front (2);
    listOne.push_front (1);

    //从后面向listOne容器中添加数据
    listOne.push_back (3);
    listOne.push_back (4);

    //从前向后显示listOne中的数据
    cout<<"listOne.begin()--- listOne.end(): "<<endl;
    for (i = listOne.begin(); i != listOne.end(); ++i)
        cout << *i << " ";
    cout << endl;

    //从后向后显示listOne中的数据
    LISTINT::reverse_iterator ir;
    cout<<"listOne.rbegin()---listOne.rend(): "<<endl;
    for (ir =listOne.rbegin(); ir!=listOne.rend();ir++) {
        cout << *ir << " ";
    }
    cout << endl;

    //使用STL的accumulate(累加)算法
    int result = accumulate(listOne.begin(), listOne.end(),0);
    cout<<"Sum=" <<result << endl;
    cout<<"-----" << endl;
}

```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)


[2015年12月 \(4\)](#)
[2015年11月 \(2\)](#)
[2015年10月 \(4\)](#)
[2015年9月 \(3\)](#)
[2015年8月 \(4\)](#)
[2015年7月 \(6\)](#)
[2015年6月 \(5\)](#)
[2015年4月 \(4\)](#)
[2015年3月 \(1\)](#)
[2015年1月 \(2\)](#)
[2014年11月 \(1\)](#)
[2014年10月 \(5\)](#)
[2014年9月 \(1\)](#)
[2014年8月 \(5\)](#)
[2014年7月 \(7\)](#)
[2014年6月 \(5\)](#)

```

//-----
//用list容器处理字符型数据
//-----

//用LISTCHAR创建一个名为listOne的list对象
LISTCHAR listTwo;
//声明i为迭代器
LISTCHAR::iterator j;

//从前面向listTwo容器中添加数据
listTwo.push_front ('A');
listTwo.push_front ('B');

//从后面向listTwo容器中添加数据
listTwo.push_back ('x');
listTwo.push_back ('y');

//从前向后显示listTwo中的数据
cout<<"listTwo.begin()---listTwo.end(): "<<endl;
for (j = listTwo.begin(); j != listTwo.end(); ++j)
    cout << char(*j) << " ";
cout << endl;

//使用STL的max_element算法求listTwo中的最大元素并显示
j=max_element(listTwo.begin(),listTwo.end());
cout << "The maximum element in listTwo is: "<<char(*j)<<endl;
}

#include <iostream>
#include <list>

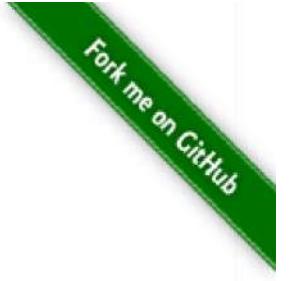
using namespace std;
typedef list<int> INTLIST;
//从前向后显示list队列的全部元素
void put_list(INTLIST list, char *name)
{
    INTLIST::iterator plist;

```

7

0

[好文要顶](#)
[关注我](#)
[收藏该文](#)

2014年5月 (1)

2014年4月 (2)

2014年3月 (3)

2014年1月 (2)

2013年12月 (7)

2013年11月 (7)

2013年10月 (4)

2013年9月 (17)

2013年8月 (29)

2013年7月 (46)

2013年6月 (25)

2013年5月 (11)

2013年4月 (4)

2010年9月 (1)

2010年8月 (2)

2010年7月 (7)

```

cout << "The contents of " << name << " : ";
for(plist = list.begin(); plist != list.end(); plist++)
    cout << *plist << " ";
cout<<endl;
}

//测试list容器的功能
void main(void)
{
//list1对象初始为空
INTLIST list1;
//list2对象最初有10个值为6的元素
INTLIST list2(10,6);
//list3对象最初有3个值为6的元素
INTLIST list3(list2.begin(),--list2.end());

//声明一个名为i的双向迭代器
INTLIST::iterator i;

//从前向后显示各list对象的元素
put_list(list1,"list1");
put_list(list2,"list2");
put_list(list3,"list3");

//从list1序列后面添加两个元素
list1.push_back(2);
list1.push_back(4);
cout<<"list1.push_back(2) andlist1.push_back(4):"<<endl;
    put_list(list1,"list1");

//从list1序列前面添加两个元素
list1.push_front(5);
list1.push_front(7);
cout<<"list1.push_front(5) andlist1.push_front(7):"<<endl;
    put_list(list1,"list1");

```

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

2010年6月 (4)

2010年5月 (5)

2010年4月 (6)

文章档案

2013年9月 (1)

Node.js学习的好去处

开源中国

由淘宝人建立的社区，内有Node中文文档

技术网站

CSDN

W3school

积分与排名

积分 - 317403

排名 - 646

```
//在list1序列中间插入数据
list1.insert( ++list1.begin(),3,9);
cout<<"list1.insert(list1.begin()+1,3,9):"<<endl;
put_list(list1,"list1");

//测试引用类函数
cout<<"list1.front()="<<list1.front()<<endl;
cout<<"list1.back()="<<list1.back()<<endl;

//从list1序列的前后各移去一个元素
list1.pop_front();
list1.pop_back();
cout<<"list1.pop_front() and list1.pop_back():"<<endl;
put_list(list1,"list1");

//清除list1中的第2个元素
list1.erase( ++list1.begin());
cout<<"list1.erase( ++list1.begin()):"<<endl;
put_list(list1,"list1");

//对list2赋值并显示
list2.assign(8,1);
cout<<"list2.assign(8,1):"<<endl;
put_list(list2,"list2");

//显示序列的状态信息
cout<<"list1.max_size(): "<<list1.max_size()<<endl;
cout<<"list1.size(): "<<list1.size()<<endl;
cout<<"list1.empty(): "<<list1.empty()<<endl;

//list序列容器的运算
put_list(list1,"list1");
put_list(list3,"list3");
cout<<"list1>list3: "<<(list1>list3)<<endl;
cout<<"list1<list3: "<<(list1<list3)<<endl;
```

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

最新评论

1. Re:Python正则进阶

你好！我是图书策划编辑，想邀请你写一本python相关的书，方便加下微信聊聊么？13647276727

--图书编辑雨露

2. Re:WPF里ItemsControl的分组实现 --listbox 实现分组

实用，点个赞。

--dino.c

3. Re:JPA、Hibernate、Spring data jpa之间的关系，终于明白了

讲的很清楚 赞

--发疯的man

4. Re:JSON C# Class Generator --由json字符串生成C#实体类的工具

然后发现json的键不能用c#变量名

--绿瞳

```
//对list1容器排序
list1.sort();
put_list(list1,"list1");

//结合处理
list1.splice(++list1.begin(),list3);
put_list(list1,"list1");
put_list(list3,"list3");
}
```

C++map 映照容器

<http://www.cppblog.com/vontroy/archive/2010/05/16/115501.html>

map映照容器的元素数据是一个键值和一个映照数据组成的，键值与映照数据之间具有一一映照的关系。

map映照容器的数据结构是采用红黑树来实现的，插入键值的元素不允许重复，比较函数只对元素的键值进行比较，元素的各项数据可通过键值检索出来。

**使用map容器需要头文件包含语句“#include<map>”，map 7 包含 0
multimap多重映照容器的定义。**

好文要顶

关注我

收藏该文



Fork me on GitHub

5. Re:JSON C# Class Generator -
--由json字符串生成C#实体类的工具

@smodi引用编辑 -> 选择性粘贴 -> 将json粘贴为类还真有，谢谢哈! ...

--幕三少

阅读排行榜

1. git命令 - 切换分支(57528)
2. python 函数参数的传递(参数带星号的说明)(39340)
3. HTML之marquee(文字滚动)详解(33525)
4. c++ list, vector, map, set 区别与用法比较(17510)
5. CodeBlocks "no such file or directory" 错误解决方案 (创建类找不到头文件) (12463)

评论排行榜

1. 最近几年的编程感悟 (1) (转) (7)

1、map创建、元素插入和遍历访问

**创建map对象，键值与映照数据的类型由自己定义。在没有指定比较函数时，
的插入位置是按键值由小到大插入到黑白树中去的，下面这个程序详细说明了如何操作
map容器。**

```

1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 int main()
11 {
12     //定义map对象，当前没有任何元素
13     map<string,float> m ;
14
15     //插入元素，按键值的由小到大放入黑白

```

7 0





- 2. 一个小程序引发的思考(54)
- 3. 使用WPF教你一步一步实现连连看(一) (46)
- 4. 看到他我一下子就悟了---委托(21)
- 5. 使用WPF教你一步一步实现连连看(二) (20)

推荐排行榜

- 1. 最近几年的编程感悟 (1) (转) (47)
- 2. 分享一篇写给程序员的文章 (很好) (20)
- 3. 使用WPF教你一步一步实现连连看(一) (19)
- 4. 最近几年的编程感悟 (2) (14)
- 5. 一个小程序引发的思考(12)

```

16   m["Jack"] = 98.5 ;
17   m["Bomi"] = 96.0 ;
18   m["Kate"] = 97.5 ;
19
20   //先前遍历元素
21   map<string,float> :: iterator it ;
22   for(it = m.begin() ; it != m.end() ; it++)
23   {
24       cout << (*it).first << ":" << (*it).second << endl ;
25   }
26
27   return 0 ;
28 }

29

```

运行结果：

Bomi : 96

Jack : 98.5

Kate : 97.5

7

0

程序编译时，会产生代号为“warning C47

好文要顶

关注我

收藏该文



Fork me on GitHub

警告的代号。可以在程序的头文件包含代码的前面使用 "#pragma warning(disable:4786)" 宏语句，强制编译器忽略该警告。4786号警告对程序的正确性和运行并无影响。

2、删除元素

map映照容器的 `erase()` 删除元素函数，可以删除某个迭代器位置上的元素、等于某个键值的元素、一个迭代器区间上的所有元素，当然，也可使用`clear()`方法清空map映照容器。

下面这个程序演示了删除map容器中键值为28的元素：

```
1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 int main()
11 {
```

7 0

[好文要顶](#) [关注我](#) [收藏该文](#)



Fork me on GitHub

```
12 // 定义map对象，当前没有任何元素  
13 map<int, char> m ;  
14 // 插入元素，按键值的由小到大放入黑白树中  
15 m[25] = 'm' ;  
16 m[28] = 'k' ;  
17 m[10] = 'x' ;  
18 m[30] = 'a' ;  
19 // 删除键值为28的元素  
20 m.erase(28) ;  
21 // 向前遍历元素  
22 map<int, char> :: iterator it ;  
23 for(it = m.begin() ; it != m.end() ; it ++)  
24 {  
25     // 输出键值与映照数据  
26     cout << (*it).first << " : " << (*it).second << endl ;  
27 }  
28 return 0 ;  
29 }  
30
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

运行结果：

10 : x

25 : m

30 : a

3、元素反向遍历

可以用反向迭代器**reverse_iterator**反向遍历**map**映照容器中的数据，它需要**rbegin()**方法和**rend()**方法指出反向遍历的起始位置和终止位置。

```
1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 int main()
11 { ... }
12 // 定义map对象，当前没有任何元素
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
13 map<int, char> m ;
14 //插入元素，按键值的由小到大放入黑白树中
15 m[25] = 'm' ;
16 m[28] = 'k' ;
17 m[10] = 'x' ;
18 m[30] = 'a' ;
19 //反向遍历元素
20 map<int, char> :: reverse_iterator rit ;
21 for( rit = m.rbegin() ; rit != m.rend() ; rit ++ )
22 {
23     //输入键值与映照数据
24     cout << (*rit).first << ":" << (*rit).second << endl ;
25 }
26 return 0 ;
27 }
```

28

运行结果:

30 : a

28 : k

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

25 : m

10 : x

4、元素的搜索

使用**find()**方法来搜索某个键值，如果搜索到了，则返回该键值所在的迭代器位置，否则，返回**end()**迭代器位置。由于**map**采用黑白树数据结构来实现，所以搜索速度是极快的。

下面这个程序搜索键值为28的元素：

```
1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 int main()
11 { ... }
12 // 定义map对象，当前没有任何元素
```

7

0

好文要顶

关注我

收藏该文





```
13 map<int, char> m ;  
14 //插入元素，按键值的由小到大放入黑白树中  
15 m[25] = 'm' ;  
16 m[28] = 'k' ;  
17 m[10] = 'x' ;  
18 m[30] = 'a' ;  
19 map<int, char> :: iterator it ;  
20 it = m.find(28) ;  
21 if(it != m.end()) //搜索到该键值  
22     cout << (*it).first << " : " << ( *it ).second << endl ;  
23 else  
24     cout << "not found it" << endl ;  
25 return 0 ;  
26 }  
27
```

5、自定义比较函数

将元素插入到map中去的时候，map会根据设定的比较函数将该元素放到该放的节点上去。在定义map的时候，如果没有指定比较函数，那么采用默认的 7 0 函数
键值由小到大的顺序插入元素。在很多情况下，需



 Fork me on GitHub

编写方法有两种。

(1) 如果元素不是结构体，那么，可以编写比较函数。下面这个程序编写的比

则是要求按键值由大到小的顺序将元素插入到map中

```
1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 //自定义比较函数 myComp
11 struct myComp
12 {
13     bool operator() (const int &a, const int &b)
14     {
15         if(a != b) return a > b ;
16     }
}
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

```
17 }  
18 } ;  
19  
20 int main()  
21 ...{  
22     //定义map对象，当前没有任何元素  
23     map<int, char> m ;  
24     //插入元素，按键值的由小到大放入黑白树中  
25     m[25] = 'm' ;  
26     m[28] = 'k' ;  
27     m[10] = 'x' ;  
28     m[30] = 'a' ;  
29     //使用前向迭代器中序遍历map  
30     map<int, char,myComp> :: iterator it ;  
31     for(it = m.begin() ; it != m.end() ; it ++)  
32         cout << (*it).first << " : " << (*it).second << endl ;  
33     return 0 ;  
34 }  
35
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

运行结果：

30 : a

28 : k

25 : m

10 : x

(2) 如果元素是结构体，那么，可以直接把比较函数写在结构体内。下面的程序详细说明了如何操作：

```
1 #include <map>
2 #include <string>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
7 using std :: string ;
8 using std :: map ;
9
10 struct Info
11 {
12     string name ;
```

7 0

好文要顶

关注我

收藏该文





Fork me on GitHub

```
13 float score ;  
14 //重载“<”操作符，自定义排列规则  
15 bool operator < (const Info &a) const  
16 {  
17     //按score由大到小排列。如果要由小到大排列，使用“>”号即可  
18     return a.score < score ;  
19 }  
20 } ;  
21  
22 int main()  
23 {  
24     //定义map对象，当前没有任何元素  
25     map<Info, int> m ;  
26     //定义Info结构体变量  
27     Info info ;  
28     //插入元素，按键值的由小到大放入黑白树中  
29     info.name = "Jack" ;  
30     info.score = 60 ;  
31     m[info] = 25 ;
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)



```
32     info.name = "Bomi" ;
33     info.score = 80 ;
34     m[info] = 10 ;
35     info.name = "Peti" ;
36     info.score = 66.5 ;
37     m[info] = 30 ;
38     //使用前向迭代器中序遍历map
39     map<Info,int> :: iterator it ;
40     for(it = m.begin() ; it != m.end() ; it++)
41     {
42         cout << (*it).second << " : ";
43
44         cout << ((*it).first).name << " : " << ((*it).first).score << endl ;
45     }
46 }
47
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

运行结果：

```
10 : Bomi  80  
30 : Peti  66.5  
25 : Jack   60
```

6、用map实现数字分离

对数字的各位进行分离，采用取余等数学方法是很耗时的。而把数字当成字符串，使用map的映照功能，很方便地实现了数字分离。下面这个程序将一个字符串中的字符当成数字，并将各位的数值相加，最后输出各位的和。

```
1 #include <string>  
2 #include <map>  
3 #include <iostream>  
4  
5 using std :: cout ;  
6 using std :: endl ;  
7 using std :: string ;  
8 using std :: map ;  
9  
10 int main()  
11 {
```

7 0

[好文要顶](#) [关注我](#) [收藏该文](#)

Fork me on GitHub

```
12 //定义map对象，当前没有任何元素  
13 map<char, int> m ;  
14  
15 //赋值：字符映射数字  
16 m['0'] = 0 ;  
17 m['1'] = 1 ;  
18 m['2'] = 2 ;  
19 m['3'] = 3 ;  
20 m['4'] = 4 ;  
21 m['5'] = 5 ;  
22 m['6'] = 6 ;  
23 m['7'] = 7 ;  
24 m['8'] = 8 ;  
25 m['9'] = 9 ;  
26 博主 /*/*上面的10条赋值语句可采用下面这个循环简化代码编写  
27 for(int j = 0 ; j < 10 ; j++)  
28 {  
29     m['0' + j] = j ;  
30 }
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
31  */
32  string sa, sb ;
33  sa = "6234" ;
34  int i ;
35  int sum = 0 ;
36  for ( i = 0 ; i < sa.length() ; i++ )
37      sum += m[sa[i]] ;
38  cout << "sum = " << sum << endl ;
39  return 0 ;
40 }
41
```

7、数字映照字符的map写法

在很多情况下，需要实现将数字映射为相应的字符，看看下面的程序：

```
1 #include <string>
2 #include <map>
3 #include <iostream>
4
5 using std :: cout ;
6 using std :: endl ;
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
7 using std :: string ;  
8 using std :: map ;  
9  
10 int main()  
11 {  
12     //定义map对象，当前没有任何元素  
13     map<int, char> m ;  
14  
15     //赋值：字符映射数字  
16     m[0] = '0' ;  
17     m[1] = '1' ;  
18     m[2] = '2' ;  
19     m[3] = '3' ;  
20     m[4] = '4' ;  
21     m[5] = '5' ;  
22     m[6] = '6' ;  
23     m[7] = '7' ;  
24     m[8] = '8' ;  
25     m[9] = '9' ;
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

```
26 //上面的10条赋值语句可采用下面这个循环简化代码编写
27 for(int j = 0 ; j < 10 ; j++)
28 {
29     m[j] = '0' + j ;
30 }
31 */
32 int n = 7 ;
33 string s = "The number is " ;
34 cout << s + m[n] << endl ;
35 return 0 ;
36 }
```

37

运行结果:

The number is 7

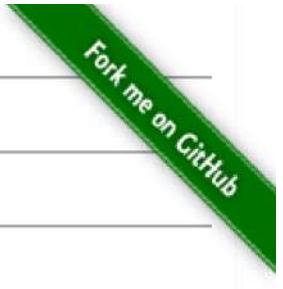
7 0

好文要顶

关注我

收藏该文





准模板库就是类与函数模板的大集合。STL共有6种组件：容器，容器适配器，迭代器，算法，函数对象和函数适配器。

1、容器

容器是用来存储和组织其他对象的对象。STL容器类的模板在标准头文件中定义。主要如下所示

7

0



Fork me on GitHub

头文件	内 容
<vector>	vector<T>容器表示一个在必要时可自动增加容量的数组。只能在矢量容器的末尾添加新元素。
<deque>	deque<T>容器实现一个双端队列。它等价于一个矢量，不过增加了向容器开头添加元素的能力。
<list>	list<T>容器是一个双向链表。
<map>	map<K, T>是一个关联容器，用关联链(类型为 K)存储确定键/对象对所在位置的每个对象(类型为 T)。映射中的每个键的值必须唯一。这个头文件也定义 multimap<K, T>容器，其中键/对象对中的键不需要唯一。
<set>	set<T>容器是一个映射，其中各对象作为自身的键。集合中的所有对象必须唯一。使用对象作为自身的键的一个后果是无法在集合中修改对象：要修改对象，必须先删除它，然后插入修改后的版本。 这个头文件也定义 multiset<T>容器，它与集合容器相似，只是其中的条目不需要唯一。
<bitset>	定义表示固定位数的 bitset<T>类模板。它通常用来存储表示一组状态或条件的标记(flag)。

表 10-1 中的容器表示 STL 中可用的所有容器，所有模板名称都在 std 命名空间中定义。T 是存储在容器中和使用键的地方的元素类型的模板类型形参，K 是键的类型。

Microsoft Visual C++也包括定义 hash_map<K, T>和 hash_set<K, T>的头文件<hash_map>和<hash_set>。它们是 map<K, T>和 set<K, T>容器的非标准变体，因为它们是在 stdext 命名空间中定义的(而不是在 std 中)。标准映射和集合容器用一个排序机制定位条目，而非标准 hash_map 和 hash_set 容器用散列机制定位条目。

①序列容器

基本的序列容器是上面图中的前三类：

7

0

好文要顶

关注我

收藏该文

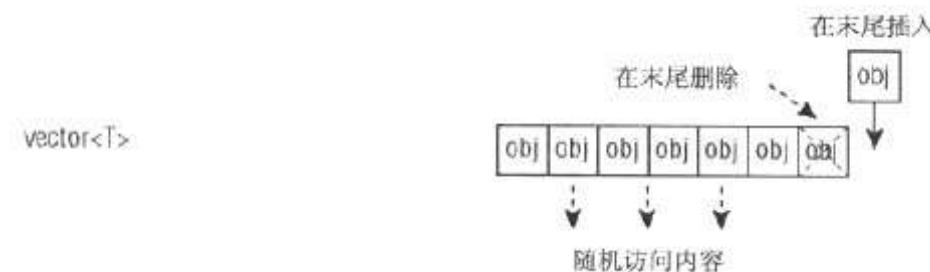


Fork me on GitHub

模 板	头 文件	说 明
<code>vector<T></code>	<code><vector></code>	创建一个表示存储 T 类型对象的动态数组的类
<code>list<T></code>	<code><list></code>	创建一个表示存储 T 类型对象的链表的类
<code>deque<T></code>	<code><deque></code>	创建一个表示存储 T 类型对象的双端队列的类

关于三者的优缺点主要是：

A:`vector<T>`矢量容器：可以随机访问容器的内容，在序列末尾添加或删除对象，但是因为是从尾部删除，过程非常慢，因为必须移动插入或删除点后面的所有对象。



矢量容器的操作：（自己以前有个表，贴出来大家看看）

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub



-10-24 20:41

Vector成员函数

函数	表述
c.assign(beg,end)	将[beg, end]区间中的数据赋值给c。
c.assign(n,elem)	将n个elem的拷贝赋值给c。
c.at(idx)	传回索引idx所指的数据，如果idx越界，抛出out_of_range。
c.back()	传回最后一个数据，不检查这个数据是否存在。
c.begin()	传回迭代器重的可一个数据。
c.capacity()	返回容器中数据个数。
c.clear()	移除容器中所有数据。
c.empty()	判断容器是否为空。
c.end()	指向迭代器中的最后一个数据地址。
c.erase(pos)	删除pos位置的数据，传回下一个数据的位置。
c.erase(beg,end)	删除[beg,end)区间的数据，传回下一个数据的位置。
c.front()	传回地一个数据。
get_allocator	使用构造函数返回一个拷贝。
c.insert(pos,elem)	在pos位置插入一个elem拷贝，传回新数据位置。
c.insert(pos,n,elem)	在pos位置插入n个elem数据。无返回值。
c.insert(pos,beg,end)	在pos位置插入在[beg,end)区间的数据。无返回值。
c.max_size()	返回容器中最大数据的数量。
c.pop_back()	删除最后一个数据。
c.push_back(elem)	在尾部加入一个数据。
c.rbegin()	传回一个逆向队列的第一个数据。
c.rend()	传回一个逆向队列的最后一个数据的下一个位置。
c.resize(num)	重新指定队列的长度。
c.reserve()	保留适当的容量。
c.size()	返回容器中实际数据的个数。
c1.swap(c2)	将c1和c2元素互换。
swap(c1,c2)	同上操作。
vector<Elem> c	创建一个空的vector。
vector <Elem> c1(c2)	复制一个vector。
vector <Elem> c(n)	创建一个vector，含有n个数据，数据均已缺省构造产生。
vector <Elem> c(n, elem)	创建一个含有n个elem拷贝的vector。
vector <Elem> c(beg,end)	创建一个以[beg,end)区间
c~vector <Elem>()	销毁所有数据，释放内存

7

0

好文要顶

关注我

收藏该文



其中的capacity表示容量，size是当前数据个数。矢量容器如果用户添加一个元素，容量已满，那么就增加当前容量的一半的内存，比如现在是500了，用户添加进第501个，那么他会再开拓250个，总共就750个了。所以矢量容器当你添加数据量很大的时候，需要注意这一点哦。。。

如果想用迭代器访问元素是比较简单的，使用迭代器输出元素的循环类似如下：

[cpp] [view](#) [plain](#) [copy](#)

1. `vector<int>::iterator`表示矢量容器`vector<int>`的迭代器。。。

[cpp] [view](#) [plain](#) [copy](#)

1. `for(vector<int>::iterator iter = number.begin(); iter<number.end(); iter++)`//这里的`iterator iter`算是一个指针了
2. `cout << " " << *iter;`

当然也可以用我们自己的方法，但是感觉用上面的更好一些。

7

0

[cpp] [view](#) [plain](#) [copy](#)

好文要顶

关注我

收藏该文



```
1. for(vector<int>::size_type i=0; i<number.size(); i++)  
2.     cout << " " << number[i]
```

Fork me on GitHub

排序矢量元素：

对矢量元素的排序可以使用<algorithm>头文件中定义的sort()函数模板来对一个矢量容器进行排序。但是有几点要求需要注意

1. sort()函数模板用<运算符来排列元素的顺序，所以容器中对象必须可以进行<运算，如果是基本类型，可以直接调用sort()，如果是自定义对象，必须对<进行运算符重载
2. 两个迭代器的指向必须是序列的第一个对象和最后一个对象的下一个位置。比如：
`sort(people.begin(), people.end()); // 这里两个参数就是迭代器的意思了`

B:deque<T>容器：非常类似vector<T>，且支持相同的操作，但是可以
在开头添加和删除。

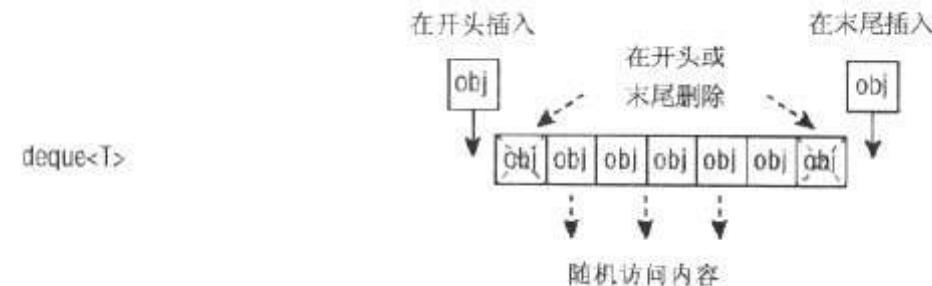
好文要顶

关注我

收藏该文



Fork me on GitHub



deque<T> 双端队列容器与矢量容器基本类似，具有相同的函数成员，但是有点不同的是它支持从两端插入和删除数据，所以就有了两个函数：**push_front**和**pop_front**。并且有两个迭代器变量

[cpp] [view plain](#) [copy](#)

```

1. <span style="font-size:18px;">#include <deque>
2. deque<int> data; //创建双端队列容器对象
3. deque<int>::iterator iter; //顺序迭代器
4. deque<int>::reverse_iterator riter; //逆序迭代器。
5. //iter和riter是不同的类型</span>

```

7

0

好文要顶

关注我

收藏该文



C:list<T>容器是双向链表，因此可以有效的在任何位置添加和删除。列表的缺点是不能随机访问内容，要想访问内容必须在列表的内部从头开始便利内容，或者从尾部开始。



②关联容器

map<K, T>映射容器:K表示键, T表示对象, 根据特定的键映射到对象, 可以进行快速的检索。

有关它的创建以及查找的操作作如下总结

[cpp] [view plain](#) [copy](#)

```

1. //创建映射容器
2. map<person, string> phonebook;
3.
4. //创建要存储的对象
5. pair<person, string> entry = pair<person, string>(pe

```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
6. 
7. //插入对象
8. phonebook.insert(entry); //只要映射中没有相同的键，就可以插入entry
9. 
10. //访问对象
11. string number = phonebook[person("mel", "Gibson")]; //如果这个键不存在，会默认将这个键插入
12. 
13. //如果不想在找不到的时候插入，可以先查找然后再检索
14. person key = person("mel", "Gibson");
15. map<person, string>::iterator iter = phonebook.find(key); //创建迭代器，就认为是指针就好了
16. 
17. if(iter != phonebook.end())
18.     string number = iter->second;
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

函数	说 明
begin()	返回一个指向映射中第一个条目的双向迭代器
end()	返回一个指向映射中最后一个条目的下一个位置的双向迭代器
rbegin()	返回一个指向映射中最后一个条目的逆向迭代器
rend()	返回一个指向是非曲直射中第一个条目的下一个位置的逆向迭代器
lower_bound()	接受一个键作为实参，如果有一个键大于或等于(下边界)指定键，则返回指向第一个条目的迭代器。如果该键不存在，将返回指向最后一个条目的下一个位置的迭代器
upper_bound()	接受一个键作为实参，如果有一个键大于(上边界)指定键，则返回一个指向第一个条目的迭代器。如果该键不存在，将返回指向最后一个条目的下一个位置的迭代器
equal_range()	接受一个键作为实参，并返回含两个迭代器的一对对象。这个对中的第一个成员指向指定键的下边界，第二个成员指向指定键的上边界。如果该键不存在，对中的两个迭代器都会指向映射中最后一个条目的下一个位置
swap()	将我们作为实参传递的映射中的条目与调用该函数的映射中的条目交换
clear()	删除映射中的所有条目
size()	返回映射中的元素数目
empty()	如果映射为空则返回 true，否则返回 false

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

2、容器适配器：

容器适配器是包装了现有的STL容器类的模板类，提供了一个不同的、通常更有限制性的功能。具体如下所示

头文件	内 容
<queue>	默认情况下，queue<T>容器由 deque<T>容器中的适配器定义，不过我们可以用list<T>容器定义它。我们只能访问队列中的第一个和最后一个元素，而且我们只能从后面添加元素，从前面删除元素。因此 queue<T>容器的运行过程或多或少就像我们在咖啡店排队一样 这个头文件还定义一个 priority_queue<T>容器，它是一个排列它所包含的元素顺序的队列，因此最大的元素总在最前面。只有前端的元素可以被访问或删除。默认情况下，优先级队列由 vector<T>中的适配器定义，不过我们可以用 deque<T>作为基础容器
<stack>	默认情况下，堆栈容器用 deque<T>容器中的适配器定义，不过我们可以用vector<T>或 list<T>容器定义它。堆栈是一种先进先出容器，因此添加或删除元素总是发生在顶部。我们只能访问顶部的元素

A:queue<T>队列容器：通过适配器实现先进先出的存储机制。我们只能向队列的末尾添加或从开头删除元素。push_back() pop_front()

代码：queue<string, list<string> > names;(这就是定义的一个 7 呀)是 0 列表创建队列的。适配器模板的第二个类型形参指定要使用的底层容器类型。要的如下

好文要顶

关注我

收藏该文



Fork me on GitHub

函 数	说 明
back()	返回对队列后端的元素的引用。这个函数有两个版本，一个版本返回 const 引用，另一个版本返回非 const 引用。如果队列为空，则返回的值不确定
front()	返回对队列前端的元素的引用。这个函数有两个版本，一个版本返回 const 引用，另一个版本返回非 const 引用。如果队列为空，则返回的值不确定

函 数	说 明
push()	将实参指定的元素添加到队列后端
pop()	删除队列前端的元素
size()	返回队列中的元素数目
empty()	如果队列为空则返回 true，否则返回 false

B:priority_queue<T>优先级队列容器：是一个队列，它的顶部总是具有最大或最高优先级。优先级队列容器与队列容器一个不同点是优先级队列容器不能访问队列后端的元素。

默认情况下，优先级队列适配器类使用的是矢量容器vector<T>，当然可以选择指定不同的序列容器作为基础，并选择一个备用函数对象来确定元素的优先级代码如下

7 0

[cpp] [view plain](#) [copy](#)[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
1. priority_queue<int, deque<int>, greater<int>> numbers;
```

C:stack<T>堆栈容器：其适配器模板在<stack>头文件中定义，默认情况下基于deque<T>容器实现向下推栈，即后进先出机制。只能访问最近刚刚进去的对象

[cpp] [view plain](#) [copy](#)

```
1. <span style="font-size:18px;">//定义容器
2. stack<person> people;
3. //基于列表来定义堆栈
4. stack<string, list<string>> names;</span>
```

基本操作如下：

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

函 数	说 明
<code>top()</code>	返回对堆栈顶部元素的引用。如果堆栈为空，则返回的值不确定。我们可以将返回的引用赋给一个 <code>const</code> 或非 <code>const</code> 引用，如果赋予后者，就可以在堆栈中修改对象
<code>push()</code>	向堆栈顶部添加实参指定的元素
<code>pop()</code>	删除堆栈顶部的元素
<code>size()</code>	返回堆栈中的元素数目
<code>empty()</code>	如果堆栈为空则返回 <code>true</code> ，否则返回 <code>false</code>

3、迭代器：

具体它的意思还没怎么看明白，书上介绍迭代器的行为与指针类似，这里做个标记 奋斗，看看后面的例子再给出具体的解释

具体分为三个部分：输入流迭代器、插入迭代器和输出流迭代器。

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

迭代器类别	说 明
输入与输出迭代器	这些迭代器读写对象的序列，可能仅使用一次。为了读或写第二次，必须获得一个新的迭代器。我们可以在迭代器上进行下列运算： <code>++iter</code> 或 <code>iter++</code> <code>*iter</code> 对于解引用运算，允许在输入迭代器的情况下进行只读访问，在输出迭代器的情况下进行只写访问
前向迭代器	前向迭代器结合了输入和输出迭代器的功能，因此我们可以对它们应用上面显示的运算，而且可以用它们进行访问和存储操作。也可以重用前向迭代器来以向前的方向遍历一组对象，我们想遍历几次就遍历几次
双向迭代器	双向迭代器提供了与前向迭代器相同的功能，此外，还允许 <code>--iter</code> 和 <code>iter--</code> 运算。这意味着我们可以后向也可以前向遍历对象序列

迭代器类别	说 明
随机访问迭代器	随机访问迭代器具有与双向迭代器相同的功能，不过还允许下列运算： <code>iter+n</code> 或 <code>iter-n</code> <code>iter +=n</code> 或 <code>iter -=n</code> <code>iter1-iter2</code> <code>iter1 < iter2</code> 或 <code>iter1 > iter2</code> <code>iter1 <= iter2</code> 或 <code>iter1 >= iter2</code> <code>iter[n]</code> 将一个迭代器递增或递减任意值 <code>n</code> 的能力允许随机访问对象的集合。最后一个使用 <code>[]</code> 运算符的运算等价于 <code>*(iter + n)</code>

7

0

看这一章的内容看的我有点抑郁了都，摘段课本介

好文要顶

关注我

收藏该文



Fork me on GitHub

<iterator>头文件中定义了迭代器的几个模板：①流迭代器作为指向输入或输出的指针，他们可以用来在流和任何使用迭代器或目的地之间传输数据。②插入迭代器可以将数据传输给一个基本序列容器。头文件中定义了两个流迭代器模板：**istream_iterator<T>**用于输入流，**ostream_iterator<T>**用于输出流。T表示从流中提取数据或写到流中的对象的类型。头文件还定义了三个插入模板：**insert<T>**，**back_insert<T>**和**front_inset<T>**。其中T也是指代序列容器中数据的类型。

输入流迭代器用下面的程序来说明下，可见具体注释

[cpp] [view plain](#) [copy](#)

```
1. #include <iostream>
2. #include <vector>
3. #include <numeric>
4. #include <sstream>
5.
6. using namespace std;
7.
8. int main()
9. {
10.     //定义矢量容器
11.     vector<int> numbers;
12.     cout << "请输入整数值，以字母结束:";
13.
14.     //定义输入流迭代器。注意两个不同
15.     //1、numberInput(cin)是指定迭代器指向流cin
16.     //2、numbersEnd没有指定，是默认的，默认构造了一个end
17.     istream_iterator<int> numbersInput(cin), numbers
```

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

```

18.
19. //用户输入，直到输入的不是int类型或者终止时结束。
20. while(numbersInput != numbersEnd)
21.     numbers.push_back(*numbersInput++);
22.
23. cout << "打印输出: " << numbers.at(3) << endl;
24.
25.
26. //如何指定输入流呢？
27.
28. //确定字符串
29. string data("2.1 3.6 36.5 26 34 25 2.9 63.8");
30.
31. //指定data为输入流input。需要头文件<iostream>
32. istringstream input(data);
33.
34. //定义迭代器
35. istream_iterator<double> begin(input), end;
36.
37. //计算数值和。
38. //accumulate为头文件<numeric>下定义的函数。
39. //第一个参数是开始迭代器，第二个是终止迭代器(最后一个值的下一个)。第三个是和的初值，注意必须用0.0，用
它确定数据类型是double
40. cout << "打印数据的总和: " << accumulate(begin, end, 0.0) << endl;
41. }

```

输出结果：

```

C:\WINDOWS\system32\cmd.exe
请输入整数值，以字母结束:6 5 3 1 4 5 a
打印输出: 1
打印数据的总和: 193.9
请按任意键继续... .

```

7 0

耽误时间太多。以后再写吧

好文要顶

关注我

收藏该文



4、算法：

算法是操作迭代器提供的一组对象的STL函数模板，对对象的一个操作，可以与前面的容器迭代器结合起来看。如下图介绍

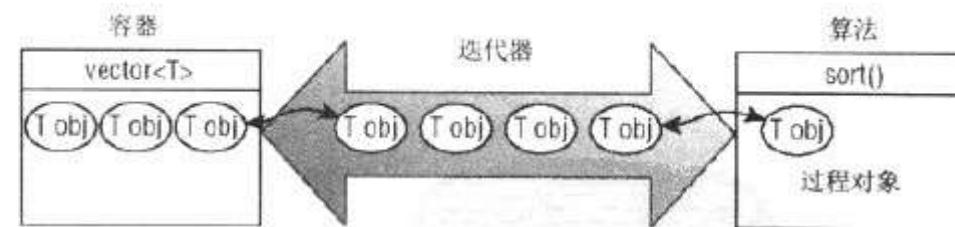


图 10-1

当我们向容器的内容应用算法时，我们提供指向容器内的对象的迭代器。算法用这些迭代器来访问容器内的对象，并在适当的时候将它们写回容器。例如，当我们向一个矢量的内容应用 sort() 算法时，就向 sort() 函数传递两个迭代器。一个指向第一个对象，另一个指向矢量中最后一个元素的下一个位置。sort() 函数用这些迭代器来访问对象进行比较，并将对象写回容器中以确定顺序。

算法在两个标准头文件中定义：`<algorithm>`头文件和`<numeric>`头文件。

5、函数对象：

函数对象是重载()运算符的类类型的对象。就是实现operator()()函数

7

0

函数对象模板在`<functional>`头文件中定义，必要时我们也可以定义自 _____ **函数**

做个标记 奋斗，等有具体实例来进行进一步的解

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

6、函数适配器：

函数适配器是允许合并函数对象以产生一个更复杂的函数对象的函数模板。

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。

下面举例说明什么是一对一的数据映射。比如一个班级中，每个学生的学号跟他的姓名就存在着一一映射的关系，这个模型用map可能轻易描述，很明显学号用int描述，姓名用字符串描述（本篇文章中不用char *来描述字符串，而是采用STL中string来描述），下面给出map描述代码：

Map<int, string> mapStudent;

7

0

1. map的构造函数

[好文要顶](#)[关注我](#)[收藏该文](#)

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;
```

2. 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))

```
#include <map>

#include <string>

#include <iostream>

Using namespace std;

Int main()

{
```

```
    Map<int, string> mapStudent;
```

```
    mapStudent.insert(pair<int, string>
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
mapStudent.insert(pair<int, string>(2, "student_two"));

mapStudent.insert(pair<int, string>(3, "student_three"));

map<int, string>::iterator iter;

for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)

{

    Cout<<iter->first<<"  "<<iter->second<<end;

}

}
```

第二种：用insert函数插入value_type数据，下面举例说明

```
#include <map>

#include <string>

#include <iostream>

Using namespace std;

Int main()

{
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
Map<int, string> mapStudent;

mapStudent.insert(map<int, string>::value_type (1,
"student_one"));

mapStudent.insert(map<int, string>::value_type (2,
"student_two"));

mapStudent.insert(map<int, string>::value_type (3,
"student_three"));

map<int, string>::iterator iter;

for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)

{

    Cout<<iter->first<<" "<<iter->second<<end;

}

}
```

第三种：用数组方式插入数据，下面举例说明

7 0
#include <map>

#include <string>

[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

```
#include <iostream>

Using namespace std;

Int main()

{

    Map<int, string> mapStudent;

    mapStudent[1] = "student_one";

    mapStudent[2] = "student_two";

    mapStudent[3] = "student_three";

    map<int, string>::iterator iter;

    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)

    {

        Cout<<iter->first<<" "<<iter->second<<end;

    }

}
```

7

0

以上三种用法，虽然都可以实现数据的插入，但是三种在效果上是完全一样的，用insert函数插入数据

[好文要顶](#)[关注我](#)[收藏该文](#)

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

7

0

```
#include <map>
```

```
#include <string>
```

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
#include <iostream>

Using namespace std;

Int main()

{

    Map<int, string> mapStudent;

    Pair<map<int, string>::iterator, bool> Insert_Pair;

    Insert_Pair = mapStudent.insert(pair<int, string>(1,
"student_one"));

    If(Insert_Pair.second == true)

    {

        Cout<<"Insert Successfully"<<endl;

    }

    Else

    {

        Cout<<"Insert Failure"<<endl;

    }

}
```

7

0



Fork me on GitHub

```
Insert_Pair = mapStudent.insert(pair<int, string>(1,
"student_two"));

If(Insert_Pair.second == true)

{

    Cout<<"Insert Successfully"<<endl;

}

Else

{

    Cout<<"Insert Failure"<<endl;

}

map<int, string>::iterator iter;

for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)

{

    Cout<<iter->first<<"  "<<iter->second<<endl;

}

}
```

7 0

好文要顶

关注我

收藏该文



Fork me on GitHub

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

```
#include <map>
#include <string>
#include <iostream>

Using namespace std;

Int main()
{
    Map<int, string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[1] = "student_two";
    mapStudent[2] = "student_three";
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++)
    {
        Cout<<iter->first<<" "<<iter->second<<endl;
    }
}
```

7

0

Cout<<iter->first<<" "<<iter->second<<endl;

[好文要顶](#)[关注我](#)[收藏该文](#)

}

}

3. 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();
```

4. 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

```
#include <map>
```

```
#include <string>
```

```
#include <iostream>
```

```
Using namespace std;
```

```
Int main()
```

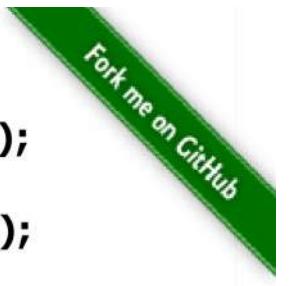
{

Fork me on GitHub

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
Map<int, string> mapStudent;

mapStudent.insert(pair<int, string>(1, "student_one"));

mapStudent.insert(pair<int, string>(2, "student_two"));

mapStudent.insert(pair<int, string>(3, "student_three"));

map<int, string>::reverse_iterator iter;

for(iter = mapStudent.rbegin(); iter != mapStudent.rend();

iter++)

{

    Cout<<iter->first<<" "<<iter->second<<end;

}

}
```

第三种：用数组方式，程序说明如下

```
#include <map>

#include <string>

#include <iostream>

Using namespace std;
```

7 0





Fork me on GitHub

Int main()

{

```
Map<int, string> mapStudent;  
  
mapStudent.insert(pair<int, string>(1, "student_one"));  
  
mapStudent.insert(pair<int, string>(2, "student_two"));  
  
mapStudent.insert(pair<int, string>(3, "student_three"));  
  
int nSize = mapStudent.size()  
  
//此处有误，应该是 for(int nIndex = 1; nIndex <= nSize; nIndex++)  
  
//by rainfish  
  
for(int nIndex = 0; nIndex < nSize; nIndex++)
```

{

```
Cout<<mapStudent[nIndex]<<end;
```

}

}

7

0

5. 数据的查找（包括判定这个关键字是否在n

[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

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，程序说明

```
#include <map>
#include <string>
#include <iostream>
Using namespace std;
```

Int main()

{

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

```
Map<int, string> mapStudent;

mapStudent.insert(pair<int, string>(1, "student_one"));

mapStudent.insert(pair<int, string>(2, "student_two"));

mapStudent.insert(pair<int, string>(3, "student_three"));

map<int, string>::iterator iter;

iter = mapStudent.find(1);

if(iter != mapStudent.end())

{

    Cout<<"Find, the value is "<<iter->second<<endl;

}

Else

{

    Cout<<"Do not Find"<<endl;

}

}
```

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

 Fork me on GitHub

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。

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

```
#include <map>
#include <string>
#include <iostream>
```

Using namespace std;

Int main()

{

Map<int, string> mapStudent;

mapStudent[1] = "student_one";

7

0

好文要顶

关注我

收藏该文



Fork me on GitHub

```
mapStudent[3] = "student_three";  
  
mapStudent[5] = "student_five";  
  
map<int, string>::iterator iter;  
  
iter = mapStudent.lower_bound(2);  
  
{  
  
    //返回的是下界3的迭代器  
  
    Cout<<iter->second<<endl;  
  
}  
  
iter = mapStudent.lower_bound(3);  
  
{  
  
    //返回的是下界3的迭代器  
  
    Cout<<iter->second<<endl;  
  
}
```

7

0

```
iter = mapStudent.upper_bound(2);
```

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

{

//返回的是上界3的迭代器

```
Cout<<iter->second<<endl;
```

}

```
iter = mapStudent.upper_bound(3);
```

{

//返回的是上界5的迭代器

```
Cout<<iter->second<<endl;
```

}

```
Pair<map<int, string>::iterator, map<int, string>::iterator>
```

```
mapPair;
```

```
mapPair = mapStudent.equal_range(2);
```

```
if(mapPair.first == mapPair.second)
```

7

0

{

```
cout<<"Do not Find"<<endl;
```

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
}

Else

{

    Cout<<"Find"<<endl;

}

mapPair = mapStudent.equal_range(3);

if(mapPair.first == mapPair.second)

{

    cout<<"Do not Find"<<endl;

}

Else

{

    Cout<<"Find"<<endl;

}

}
```

7 0

6. 数据的清空与判空

[好文要顶](#)[关注我](#)[收藏该文](#)

清空map中的数据可以用**clear()**函数，判定map中是否有数据可以用**empty()**函数，
它返回**true**则说明是空map

7. 数据的删除

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

```
#include <map>

#include <string>

#include <iostream>

Using namespace std;

Int main()
```

```
{

    Map<int, string> mapStudent;

    mapStudent.insert(pair<int, string>(1, "student_one"));

    mapStudent.insert(pair<int, string>(2, "student_two"));

    mapStudent.insert(pair<int, string>(3, "student_thr
```

7 0

//如果你要演示输出效果，请选择以下的一种，例

好文要顶

关注我

收藏该文





Fork me on GitHub

//如果要删除1,用迭代器删除

```
map<int, string>::iterator iter;  
iter = mapStudent.find(1);  
mapStudent.erase(iter);
```

//如果要删除1, 用关键字删除

```
Int n = mapStudent.erase(1); //如果删除了会返回1, 否则返回0
```

//用迭代器, 成片的删除

//一下代码把整个map清空

```
mapStudent.erase(mapStudent.begin(), mapStudent.end());
```

//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

//自个加上遍历代码, 打印输出吧

}

7

0

好文要顶

关注我

收藏该文



 Fork me on GitHub

8. 其他一些函数用法

这里有swap, key_comp, value_comp, get_allocator等函数，感觉到这些函数编程用的不是很多，略过不表，有兴趣的话可以自个研究

9. 排序

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

```
#include <map>

#include <string>

Using namespace std;

Typedef struct tagStudentInfo
```

{

Int nID;

String strName;

7

0

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
}StudentInfo, *PStudentInfo; //学生信息
```

```
Int main()
```

```
{
```

```
int nSize;
```

```
//用学生信息映射分数
```

```
map<StudentInfo, int>mapStudent;
```

```
map<StudentInfo, int>::iterator iter;
```

```
StudentInfo studentInfo;
```

```
studentInfo.nID = 1;
```

```
studentInfo.strName = "student_one";
```

```
mapStudent.insert(pair<StudentInfo, int>(studentInfo, 90));
```

```
studentInfo.nID = 2;
```

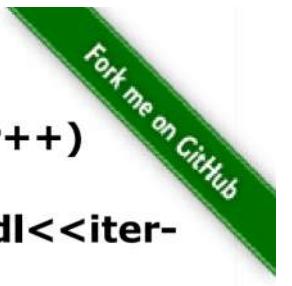
```
studentInfo.strName = "student_two";
```

7

0

```
mapStudent.insert(pair<StudentInfo, int>(studentInfo, 90));
```

好文要顶关注我收藏该文

Fork me on GitHub

```
for (iter=mapStudent.begin(); iter!=mapStudent.end(); iter++)  
    cout<<iter->first.nID<<endl<<iter->first.strName<<endl<<iter->second<<endl;  
  
}
```

以上程序是无法编译通过的，只要重载小于号，就OK了，如下：

```
Typedef struct tagStudentInfo  
{  
    Int      nID;  
    String   strName;  
    Bool operator < (tagStudentInfo const& _A) const  
{  
    //这个函数指定排序策略，按nID排序，如果nID相等的话，  
    //那么就按strName排序  
    If(nID < _A.nID)  return true;
```

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
If(nID == _A.nID) return strName.compare(_A.strName);  
0;  
Return false;  
}  
}StudentInfo, *PStudentInfo; //学生信息
```

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

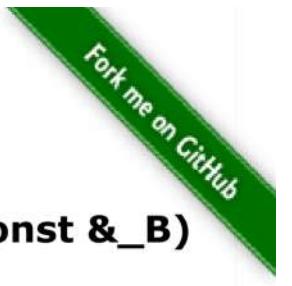
```
#include <map>  
  
#include <string>  
  
Using namespace std;  
  
Typedef struct tagStudentInfo  
{  
    Int    nID;  
    String strName;  
}StudentInfo, *PStudentInfo; //学生信息
```

7

0

Classs sort

[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

{

Public:**Bool operator() (StudentInfo const &_A, StudentInfo const &_B)****const**

{

If(_A.nID < _B.nID) return true;**If(_A.nID == _B.nID) return****_A.strName.compare(_B.strName) < 0;****Return false;**

}

};

Int main()

{

7

0

//用学生信息映射分数**Map<StudentInfo, int, sort>mapSt**[好文要顶](#)[关注我](#)[收藏该文](#)

Fork me on GitHub

```
StudentInfo studentInfo;  
  
studentInfo.nID = 1;  
  
studentInfo.strName = "student_one";  
  
mapStudent.insert(pair<StudentInfo, int>(studentInfo, 90));  
  
studentInfo.nID = 2;  
  
studentInfo.strName = "student_two";  
  
mapStudent.insert(pair<StudentInfo, int>(studentInfo, 80));  
  
}
```

10. 另外

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 $\log_2 N$ 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。

7

0

分类: C++[好文要顶](#)[关注我](#)[收藏该文](#)



Fork me on GitHub

« 上一篇: [由一道面试题来了解进程间的通信](#)
» 下一篇: [在多台PC之间同步Resharper所有设置的方法](#)

posted @ 2015-04-26 13:05 幕三少 阅读(17510) 评论(1) 编辑 收藏

评论列表

#1楼 2016-04-12 22:26 trigger007

回复 引用

博主总结的非常棒，但是想问一下，博文里鼠标移动会有小星星出现，这个是怎么做到的？谢谢

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论

昵称: Justry2015

评论内容:



7 0

好文要顶 关注我 收藏该文



评论 大众 吸溜



Fork me on GitHub

[提交评论](#) [退出](#) [订阅评论](#)

[Ctrl+Enter快捷键提交]

【推荐】超50万VC++源码：大型组态工控、电力仿真CAD与GIS源码库！

【免费】要想入门学习Linux系统技术，你应该先选择一本适合自己的书籍

【前端】SpreadJS表格控件，可嵌入应用开发的在线Excel

【直播】如何快速接入微信支付功能

7

0

好文要顶

关注我

收藏该文

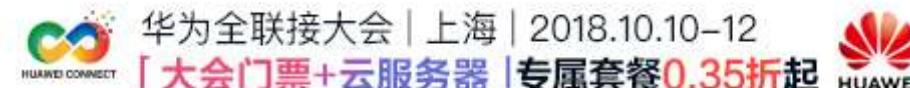


Fork me on GitHub



最新IT新闻:

- Comma.ai创始人George Hotz宣布辞去CEO职务
 - 特斯拉飓风山竹来袭时解锁电池容量 智能汽车OTA升级优势渐显
 - 连内饰都是智能的，宝马未来电动车都要长这样？
 - iPhone XS系列的「双卡双待」怎么用？苹果中国给了一份使用指南
 - 不依赖一颗汽车传感器，戴姆勒和博世自动代客泊车背后的秘密
- » 更多新闻...



最新知识库文章:

- 为什么说 Java 程序员必须掌握 Spring Boot ?
 - 在学习中，有一个比掌握知识更重要的能力
 - 如何招到一个靠谱的程序员
 - 一个故事看懂“区块链”
 - 被踢出去的用户
- » 更多知识库文章...

7

0