一篇文章搞定【所有】STL八股文

原创 陈同学 陈同学在搬砖 2021-12-16 09:36

加我微信chen079328 拉你进技术群

STL八股文

Q1 什么是STL?

STL包含了以下几个部分

容器:本质上是一种类模板 对存放数据的数据结构的实现

算法:各种算法的实现 本质上是函数模板

迭代器:算法和容器的胶合剂 迭代器是一种将operator*，operator->，operator++，operator--等指针相关操作予以重载的类模板。所有STL容器都附带有自己专属的迭代器。

仿函数:重载了（）的类或者类模板

配接器: 在已有容器基础上实现的接口

空间配置器:用于空间配置和管理的类模板

Q2 讲讲STL中各容器的用法?

Q3 讲讲STL中的空间配置器?

空间配置器allocator是什么东西呢?

allocator是STL的六大组件之一. 其作用就是为各个容器管理内存(内存开辟 内存回收) 其实allocator配置的对象不只是内存,它也可以向硬盘索取空间. 但是在这里我们主要考虑对内存的管理

我们在使用STL库的时候不用去考虑各种内存的操作,就是因为allocator已经帮我们做好了内存配置的工作 那么allocator是怎么实现内存的管理的呢? 举个简单的例子

vectorvec;

我们在声明上面的vec的时候,allocator是怎么工作的呢? 其实对于一个标准的STL 容器，当Vetorvec 的真实语句应该是

vetor<int, allocator>vec

第二个参数就是我们用到的allocator,这里是标准版本的空间配置器,也就是说不同版本的STL里面都会有这个(STL的不同版本)

但是在SGI版本的STL中,它还有一个更高效的空间配器,名字叫alloc 用这个空间配置器的时候不能用标准写法,它不接受任何参数

也就是说在SGI版本的STL里面有两个空间配置器 一个是标准的allocator,一个是SGI它自己做的alloc

两种配置器的用法是这样的

vetor<int, allocator>vec//标准的allocator空间配置器

vetor<int, alloc>vec//alloc空间配置器

关于空间配置器的具体原理可以看我写的下面这篇博客

https://blog.csdn.net/vjhghjghj/article/details/88647336 （复制后到浏览器打开）

Q4 讲讲STL中的迭代器?

0. 介绍

迭代器算法和容器的胶合剂 迭代器是一种 将operator*，operator->，operator++，operator--等指针相关操作予以重载的类模板。所有STL容器都附带有自己专属的迭代器。

1.分类 及操作

在STL中，迭代器主要分为5类。

`输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机訪问迭代器`

每种迭代器均可进行包括表中前一种迭代器可进行的操作。 输入迭代器和输出迭代器是最低级的迭代器。后三种迭代器都是对该迭代器的一种派生， 这五类迭代器的从属关系，如下图所示，

其中箭头A→B表示，A是B的强化类型，这也说明了如果一个算法要求B，那么A也可以应用于其中。

- Input（输入）迭代器

只能一次一个向前读取元素，按此顺序一个个传回元素值。表2.1列出了Input迭代器的各种操作行为。Input迭代器只能读取元素一次，如果你复制Input迭代器，并使原Input迭代器与新产生的副本都向前读取，可能会遍历到不同的值。纯粹Input迭代器的一个典型例子就是“从标准输入装置（通常为键盘）读取数据”的迭代器。

表达式	功能表述
*iter	读取实际元素
iter->member	读取实际元素的成员（如果有的话）
++iter	向前步进（传回新位置）
iter++	向前步进（传回旧位置）
iter1 == iter2	判断两个迭代器是否相同
iter1 != iter2	判断两个迭代器是否不相等
TYPE(iter)	复制迭代器（copy 构造函数）

- Output（输出）迭代器

Output迭代器和Input迭代器相反，其作用是将元素值一个个写入。表2.2列出Output迭代器的有效操作。operator*只有在赋值语句的左手边才有效。Output迭代器无需比较（comparison）操作。你无法检验Output迭代器是否有效，或“写入动作”是否成功。你唯一可以做的就是写入、写入、再写入。

表达式	功能表述
*iter = value	将元素写入到迭代器所指位置
++iter	向前步进（传回新位置）
iter++	向前步进（传回旧位置）
TYPE(iter)	复制迭代器（copy 构造函数）

- Forward（前向）迭代器

Forward迭代器是Input迭代器与Output迭代器的结合，具有Input迭代器的全部功能和Output迭代器的大部分功能。表2.3总结了Forward迭代器的所有操作。Forward迭代器能多次指向同一群集中的同一元素，并能多次处理同一元素。

```
表达式                  功能表述

*iter               存取实际元素

iter->member        存取实际元素的成员

++iter              向前步进（传回新位置）

iter++              向前步进（传回旧位置）

iter1 == iter2      判断两个迭代器是否相同

iter1 != iter2      判断两个迭代器是否不相等

TYPE()              产生迭代器（default构造函数）

TYPE(iter)          复制迭代器（copy构造函数）

iter1 == iter2      复制
```

- Bidirectional（双向）迭代器

Bidirectional（双向）迭代器在Forward迭代器的基础上增加了回头遍历的能力。换言之，它支持递减操作符，用以一步一步的后退操作。

```
p++                             后置自增迭代器
++p                             前置自增迭代器
--p                               前置自减迭代器
p--                               后置自减迭代器
```

- Random Access（随机存取）迭代器

Random Access迭代器在Bidirectional迭代器的基础上再增加随机存取能力。因此它必须提供“迭代器算数运算”（和一般指针“指针算术运算”相当）。也就是说，它能加减某个偏移量、能处理距离（differences）问题，并运用诸如<和>的相互关系操作符进行比较。以下对象和型别支持Random Access迭代器：

可随机存取的容器（vector, deque）

strings（字符串，string，wstring）

一般array（指针）

```
p++    后置自增迭代器
++p                                 前置自增迭代器
p+=i                                 将迭代器递增i位
p-=i                                   将迭代器递减i位
p+i                                    在p位加i位后的迭代器
p-i                                     在p位减i位后的迭代器
p[i]                                    返回p位元素偏离i位的元素引用
p<p1                                 如果迭代器p的位置在p1前，返回true，否则返回false
p<=p1                               p的位置在p1的前面或同一位置时返回true，否则返回false
p>p1                                 如果迭代器p的位置在p1后，返回true，否则返回false
p>=p1                               p的位置在p1的后面或同一位置时返回true，否则返回false
```

各容器支持的迭代器的类别如下：

```
容器                    支持的迭代器类别
vector                  随机访问
deque                   随机访问
set                       双向
multimap                双向
multiset                双向
list                      双向
map                     双向
queue                   不支持
stack                     不支持
```

2.迭代器的失效

迭代器失效指的是迭代器原来所指向的元素不存在了或者发生了移动，此时假设不更新迭代器，将无法使用该过时的迭代器。迭代器失效的根本原因是对容器的某些操作改动了容器的内存状态（如容器又一次载入到内存）或移动了容器内的某些元素。

使vector迭代器失效的操作

1.向vector容器内加入元素(push_back,insert)　向vector容器加入元素分下面两种情况：

1）若向vector加入元素后，整个vector又一次载入。即前后两次vector的capacity()的返回值不同一时候，此时该容器内的全部元素相应的迭代器都将失效。

2）若该加入操作不会导致整个vector容器载入，则指向新插入元素后面的那些元素的迭代器都将失效。

2,删除操作(erase,pop_back,clear)　　vector运行删除操作后，被删除元素相应的迭代器以及其后面元素相应的迭代器都将失效。

3.resize操作：调整当前容器的size　　调整容器大小对迭代器的影响分例如以下情况讨论：

A.若调整后size>capacity，则会引起整个容器又一次载入，整个容器的迭代器都将失效。

B.若调整后size<capacity，则不会又一次载入，详细情况例如以下：

B1.若调整后容器的size>调整前容器的size，则原来vector的全部迭代器都不会失效。

B2.若调整后容器的size<调整前容器的size，则容器中那些别切掉的元素相应的迭代器都将失效。

4.赋值操作(v1=v2　v1.assign(v2))　　会导致左操作数v1的全部迭代器都失效，显然右操作数v2的迭代器都不会失效。

5.交换操作(v1.swap(v2))

因为在做交换操作时。v1,v2均不会删除或插入元素，所以容器内不会移动不论什么元素。故v1,v2的全部迭代器都不会失效。

使deque迭代器失效的操作

1.插入操作（push_front,push_back,insert)　　deque的插入操作对迭代器的影响分两种情况：

1).在deque容器首部或尾部插入元素不会是不论什么迭代器失效；

2).在除去首尾的其它位置插入元素会使该容器的全部迭代器失效。

2.删除操作

1).在deque首、尾删除元素仅仅会使被删除元素相应的迭代器失效；

2).在其它不论什么位置的删除操作都会使得整个迭代器失效。

使list/map/set迭代器失效的操作

因为list/map/set容器内的元素都是通过指针连接的。list实现的数据结构是双向链表，而map/set实现的数据结构是红黑树，故这些容器的插入和删除操作都只需更改指针的指向，不会移动容器内的元素。故在容器内添加元素时，不会使不论什么迭代器失效，而在删除元素时，只会使得指向被删除的迭代器失效。

Q5 讲讲STL中的vector?

- 1.迭代器(成员变量)

vector维护的是一个连续的线性空间, 由于是连续线性空间, 所以其迭代器所要进行的一些操作比如:*,->,+,-,++,--等等普通的指针都可以满足 所以vector的迭代器就是普通指针. 通过普通指针也可让vector随机存取(所以vector的迭代器是Random Access Iterator) 看一下vector源码中,它的迭代器是怎么定义的

```
emplate<class T,class Alloc=alloc>
class vector{
   public:
       typedef     T             value_type;
       typedef     value_type*   iterator;//vector的迭代器是普通指针
};
```

因此，如果客户端写出这样的代码：

```
vector<int>::iterator ivite;
vector<Shape>::iterator svite;
ivite的型别就是int*,svitede 的型别就是Shape*
```

- 2.数据结构(成员变量)

在将vector的数据结构之前,先说一下vector的存储方式, vector在使用时正常来说会有一部分正在使用的空间和一部分备用空间, 总的空间大小就叫做容量

容量永远大于等于其大使用空间大小,如果相等就是满载了 增加新元素时,如果元素超过了原来的容量, 则下次再有新元素时整个vector就得另觅居所了 在另外一个居所上,容量将扩充至原来的两倍,如果还不够,则继续扩充

这个过程会经历"重新开辟新内存->搬运元素->释放原来的内存" vector实现的关键在于它在内部定义了三个迭代器(指针), 一个start指向正在使用空间的头,一个fininsh指向正在使用空间的尾, 一个end_of_storage表示目前可用空间的尾,

源码如下

```
template<class T,class Alloc=alloc>
class vector{
...
protected:
    iterator start;
    iterator finish;
    iterator end_of_storage;
};
```

演示如下图

通过这三个迭代器,就可以实现我们想要的很多操作,比如提供首尾标示,大小,容量,空容器判断,[ ]运算符,最前端元素,最后端元素等

```
template <class T,class Alloc=alloc>
class vector{
...
public:
    iterator begin(){return start;}
    iterator end(){return finish;}
    size_type size() const {return size_type(end()-begin());}
    size_type capacity() const {return size_type(end_of_storage-begin());}
    bool empty() const {return begin()==end();}
    reference operator[](size_type n){return *(begin()+n);}
    reference front(){return *begin();}
    reference back() {return *(end()-1);}
...
};
```

3.内存管理

内存扩容

新增元素：Vector通过一个连续的数组存放元素， 如果集合已满，在新增数据的时候，就要分配一块更大的内存， 将原来的数据复制过来，释放之前的内存，在插入新增的元素；

对vector的任何操作，一旦引起空间重新配置， 指向原vector的所有迭代器就都失效了 ； 初始时刻vector的capacity为0， 塞入第一个元素后capacity增加为1； 不同的编译器实现的扩容方式不一样， VS2015中以1.5倍扩容，GCC以2倍扩容。

```
代码
#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> vec;
    cout << vec.capacity() << endl;
    for (int i = 0; i<10; ++i)
    {
        vec.push_back(i);
        cout << "size: " << vec.size() << endl;
        cout << "capacity: " << vec.capacity() << endl;
    }
    system("pause");
    return 0;
}
```

可以根据输出看到，vector是以2倍的方式扩容的。这里让我产生了两个疑问： 1.为什么要成倍的扩容而不是一次增加一个固定大小的容量呢？2.为什么是以两倍的方式扩容而不是三倍四倍，或者其他方式呢？

第一个问题：

以成倍方式增长

```
假定有 n 个元素,倍增因子为 m;
完成这 n 个元素往一个 vector 中的
push_back操作,
需要重新分配内存的次数大约为 logm(n);
第 i 次重新分配将会导致复制 m^(i)
 (也就是当前的vector.size() 大小)个
旧空间中元素;
n 次 push_back 操作
所花费的时间复制度为O(n):
m / (m - 1), 这是一个常量,
均摊分析的方法可知,
vector 中 push_back 操作的
时间复杂度为常量时间.
```

一次增加固定值大小

```
假定有 n 个元素,每次增加k个;
第i次增加复制的数量为为: 100i
n 次 push_back 操作
所花费的时间复杂度为O(n^2):
均摊下来每次push_back
操作的时间复杂度为O(n);
```

总结：对比可以发现采用采用成倍方式扩容， 可以保证常数的时间复杂度， 而增加指定大小的容量 只能达到O(n)的时间复杂度， 因此，使用成倍的方式扩容

。

第二个问题：

根据查阅的资料显示，考虑可能产生的堆空间浪费，成倍增长倍数不能太大， 使用较为广泛的扩容方式有两种，以2二倍的方式扩容，或者以1.5倍的方式扩容。以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和， 导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间：

知乎上看到一个很好的解释： C++ STL中vector内存用尽后，为啥每次是两倍的增长，而不是3倍或其他数值？借用他的一张图来说明2倍与1.5倍的区别：

- push_back()函数

当我们以push_back（）
将新元素插入于vector尾端时，
该函数首先检查是否还有备用空间，
如果有就直接在备用空间上构造函数，
并调整迭代器finish，
使vector变大。
如果没有备用空间，
就扩充空间
（重新配置，移动数据，释放原空间）
正如我们上面所说的
增加新元素时，
如果元素超过了原来的容量，
则下次再有新元素时
整个vector就得另觅居所了
在另外一个居所上，
容量将扩充至原来的两倍，
如果还不够，则继续扩充
这个过程会经历
"重新开辟新内存->搬运元素->释放原来的内存"
所谓动态增加大小，
并不是在原空间之后接续新空间
（因为无法保证原空间之后尚有可供配置的空间），
而是以原大小的两倍另外配置一块较大空间，
然后将原内容拷贝过来，
然后才开始在原内容之后构造新元素，
并释放原空间。
因此，对vector的任何操作，
一旦引起空间重新配置，
指向原vector的
所有迭代器就都失效了。

```
void push_back(const T& x)
{
   if(finish!=end_of_storage)
   {
       //还有备用空间
       construct(finish,x);//全局函数
       ++finish;
   }
   else
   {
       insert_aux(end(),x);//vector member function
   }
}
```

我们知道，只有在调用析构函数的时候，vector 才会自动释放缓冲区。那么，如何根据自己的需要，强制释放缓冲区呢？有两种方法：

```
// 方法一：
vector<int>().swap(v1);
 ```
```

//方法二：vectorv_temp; v1.swap(v_temp);

分析：方法一是将 v1 直接和空的vector交换，原内存当然就被销毁了。　　第二种方法是曲线销毁，先定义一个临时变量。由于临时变量没有被初始化，所以，缓冲区大小为0.那么，当 V1 与它交换后，v1 原来占用的缓冲区就被销毁了。而临时变量 v_temp 调用析构函数来进行释放空间。
```

5.相关问题

vector与list的区别与应用?

(1) vector为存储的对象分配一块连续的地址空间，随机访问效率很高。但是插入和删除需要移动大量的数据，效率较低。尤其当vector中存储

的对象较大，或者构造函数复杂，则在对现有的元素进行拷贝的时候会执行拷贝构造函数。

(2) list中的对象是离散的，随机访问需要遍历整个链表，访问效率比vector低。但是在list中插入元素，尤其在首尾插入，效率很高，只需要改变元素

(3) vector是单向的，而list是双向的；

(4) 向量中的iterator在使用后就释放了，但是链表list不同，它的迭代器在使用后还可以继续用；链表特有的；

使用原则：

(1) 如果需要高效的随机存取，而不在乎插入和删除的效率，使用vector；

(2) 如果需要大量高效的删除插入，而不在乎存取时间，则使用list；

(3) 如果需要搞笑的随机存取，还要大量的首尾的插入删除则建议使用deque，它是list和vector的折中；

vector和deque的区别?
deque与vector的主要不同之处在于：

1. 两端都能快速安插和删除元素，这些操作可以在分期摊还的常数时间（amortized constant time）内完成。

2. 元素的存取和迭代器的动作比vector稍慢。

3. 迭代器需要在不同区块间跳转，所以它非一般指针。

4. 因为deque使用不止一块内存（而vector必须使用一块连续内存），所以deque的max_size()可能更大。

5. 不支持对容量和内存重新分配时机的控制。不过deque的内存重分配优于vector，因为其内部结构显示，deque不必在内存重分配时复制所有元素。

6. 除了头尾两端，在任何地方安插或删除元素，都将导致指向deque元素的所有pointers、references、iterators失效。

7. deque的内存区块不再被使用时，会自动被释放。deque的内存大小是可自动缩减的。

8. deque与vector组织内存的方式不一样。在底层，deque按“页”（page）或“块”（chunk）来分配存储器，每页包含固定数目的元素。而vector只分

deque的下述特性与vector差不多：

1. 在中部安插、删除元素的速度较慢。

2. 迭代器属于random access iterator（随机存取迭代器）。

优先使用vector，还是deque?

c++标准建议：vector是那种应该在默认情况下使用的序列。如果大多数插入和删除操作发生在序列的头部或尾部时，应该选用deque。

Q6 讲讲STL中的list?

0.概述

list是什么？ list本质上就是双向链表 相对与上一篇所讲的vector,我们知道双向链表有它自己的优点,首先空间利用比较灵活,所以省空间. 而且插入删除元素都是常数时间 下面将参照list的源码,将其分为下面几部分去讲

1.结点

既然list是链表,那就必须得定义它自己的结点类 在源码中它的结点是这样子的

2.迭代器

不像vector是连续的内存空间,所以可以直接用指针作为迭代器 在 list中由于是链式存储的,所以得定义一个属于自己的迭代器 直接看一些它的源码

可以看到在迭代器类中 有一个成员变量 node,用来存放该迭代器指向的结点, 可以通过构造函数给node指定指向的结点, 也可以给通过复制构造函数给node指定好另一个迭代器的node指向的结点. 由于list只支持双向迭代器(因为双向链表要支持前移 后移),所以对运算符++ ,-- ,* ,->进行了重载. 注意 在list中插入结点和删除某个结点并不会使其他结点的迭代器失效, 而vector就并非如此,因为插入元素可能导致原有内存重新分配 导致原有迭代器失效

3.数据结构

list是一个双向循环链表,既然是循环,也就是说尾结点的下一个就是头结点,头结点的上一个就是尾结点. 同时只要用一个结点指针指向了其中一个结点,就可以遍历到其他任意结点 在list中定义了一个空白的尾结点指针 用node表示 如下图表示

通过这个node,就可以表示出其他任意一个节点了,这样就可以实现我们的一些list的基本操作,

比如返回头结点(即返回node->next就可以了),返回尾结点(返回node本身就行了),返回头结点值,返回尾结点值,以及判空,返回长度

我们看一下在源码中是怎么实现的

4.内存管理

4.1内存分配与释放

4.2内存使用

6.其他问题

- list和queue与vector之间的区别?

list不再能够像vector一样以普通指针作为迭代器， 因为其节点不保证在存储空间中连续存在；list插入操作和结合才做都不会造成原有的list迭代器失效;

list不仅是一个双向链表， 而且还是一个环状双向链表，所以它只需要一个指针；

list不像vector那样 有可能在空间不足时做重新配置、数据移动的操作， 所以插入前的所有迭代器在插入操作之后都仍然有效；

deque是一种双向开口的连续线性空间， 所谓双向开口， 意思是可以在头尾两端分别做元素的插入和删除操作；可以在头尾两端分别做元素的插入和删除操作；

deque和vector最大的差异， 一在于deque允许常数时间内对起头端进行元素的插入或移除操作， 二在于deque没有所谓容量概念， 因为它是动态地以分段连续空间组合而成， 随时可以增加一段新的空间并链接起来， deque没有所谓的空间保留功能。

Q7 讲讲STL中的deque?

0.概述

什么是deque呢? 一句话来概括 ,deque是一种 双向开口的"连续"线性空间存储数据的数据结构(注意这里的连续打了双引号,后面会解释)

咱们来对比一下deque和vector的一些区别

优点:

1.vector是单向开口 deque是双向开口.也就是说vector只能在一端插入和删除数据,而deque在两端都可以插入和删除数据

(其实vector也可以在头部插入和删除,但是头部插入,后面的元素就得后移,所以效率奇差,一般不采用)

2.如第一条所说,vector在头部插入删除元素效率很低,但是deque在头部插入和删除元素是常数时间.

3.vector是一端连续线性空间,满了以后,在重新分配空间 搬运元素,回收原来空间.但是deque的连续线性空间的"连续"其实是一种假象,实际上他是动态的以分段连续空间组合而成的(这一点到后面会有讲解),所以deque没有必要提供空间保留功能

缺点:

1.vector和deque的迭代器都属于Random Acess Iterator ,

且vector的迭代器本质上就是指针,而deque由于结构的复杂性,他的迭代器也会相应复杂得多,从而也影响了一些deque成员函数计算复杂性,所以一般可以用vector的地方尽量用vector.

比如要对deque的元素排序时,为了提高效率,一般会把deque元素复制到vetor然后在vector里面排好序复制回deque.

下面将通过源码 分几个部分来讲解deque

1. 中控器与迭代器

原理讲解

现在来说一下为什么deque的连续是一种"假象" deque本质上是由一段一段小的连续子空间拼在一起的,

当要在deque已满的情况下在尾部插入元素,deque会在尾部再加入一段小的连续子空间.

同样在deque已满的情况下在头部插入元素,deque会在头部再加入一段小的连续子空间.

然后通过某种办法来把这些分段的连续子空间维护其整体连续的假象.这个办法就是 中控器.

这样就避免了vector每次空间满了以后的"重新配置,搬运元素 回收内存"的麻烦 带来了上一节所说的三个优点

同时由于设计相对会复杂一点,所以就导致了deque实现复杂度会大很多.

下面来具体讲一讲整体连续的假象是怎么实现的.

所谓的中控器实际上就是一个指针数组(即数组元素是指针),叫map(注意不是STL的map容器)

上面讲的每一段小的连续子空间叫deque的缓冲区 然后map数组的每一个元素,即每一个指针都会指向一个相应的缓冲区,

当map数组满了以后,需要再找一个更大的空间来作为map,如下图

中控器对这些一段一段的缓冲区有一个很好的组织之后,下面就是迭代器发挥作用了. 一个deque会有两个迭代器,start和finish

start 迭代器负责起始的那块缓冲区,finish迭代器负责最后的那块缓冲区 每个迭代器会有四个指针域cur first last node

node指向在map中该缓存区所处的位置 first指向当前缓冲区的整体空间的头部

last指向当前缓冲区的整体空间的尾部

对于finish迭代器,cur指向当前缓冲区已用空间的最后一个元素位置(的下一个位置),

对于start迭代器,cur指向当前缓冲区已用空间的第一个元素位置 具体表示如下图

Q8 讲讲STL中的map/set?

实现原理

map/set/multi_map/multi_set底层的实现都是红黑树， 关于红黑树的原理可以看这篇文章

https://blog.csdn.net/vjhghjghj/article/details/88779703

关于map/multi_map

- 和set类似 ,map的底层实现也是红黑树RB-tree,下面说一下map的几个特性
- map的所有元素是pair,什么是pair?pair就是同时包含了实值(value)和键值(key)的一个值,pair的第一个元素视为键值 第二个元素是实值 ,它的源码定义如下
- map所有的元素根据pair里面的键值自动排序, 两个元素之间不允许有相同的键值
- 和set一样,我们不能通过map的迭代器更改元素里的键值key,因为改变了key会破坏map的组织规则,
- 但是可以通过迭代器更改实值value 因为实值value并不影响map的组织顺序.
- 和set一样,当删除和插入map中一个元素时,对其他元素的迭代器不会有影响,其他迭代器不会失效
- map底层调用红黑树,即RB-Tree模板类.set要执行的所有操作,RB-tree差不多都提供了.所以几乎所有的map操作行为都是调用RB-tree的操作行为而已
- multimap的特性以及用法和map完全相同,唯一的差别在于它允许键值重复 因此它采用的底层机制是RB-tree的insert_equal()而非insert_unique()

关于set/multi_set

set是什么?set即集合 它的主要特性有以下几个

- 由于set的底层是红黑树实现的,而红黑树又是一种二叉排序树,所以set里面所有的元素键值会被自动排序,而且不允许两个元素有相同的键值,因为那样的话在树里面就不知道怎么排序了
- 不像map那样有key(键值)和value(实值),set的key和value是一体的, 通过set的迭代器不能改变set里面的元素键值.因为改变了键值,红黑树的结点就得重新排列了,就会破坏set的组织.所以set的迭代器被定义为红黑树底层的const_iterator,杜绝写入操作
- 当对某个元素插入或删除时,其余迭代器依然是保持有效的.
- set底层调用红黑树,即RB-Tree模板类.set要执行的所有操作,RB-tree差不多都提供了.所以几乎所有的set操作行为都是调用RB-tree的操作行为而已
- multiset的特性以及用法和set完全相同,唯一的差别在于它允许键值重复 因此它采用的底层机制是RB-tree的insert_equal()而非insert_unique()