C++标准函数库

1. template模板

1.定义：是针对一个或多个尚未明确的型别所撰写的函数或类别，使用template时，可以显示地或隐式地将型别当做参数传递

template<class T>

inline const T& max(const T &a,const T&b)

{

return a<b?b:a;

}

其中 T为任意数据类型

2.缺省模板参数

Template classes可以有缺省参数

template <class T,class container = vector<T>>

class MyClass;

如果只传给他一个参数，那么缺省可作为第二个参数使用

MyClass<int> x1;

3.关键字 typename

1).用来作为型别之前的标识符，给一类型又起了一个名字

2).可以在template中代替class

template<typename T> class MyClass;

二、命名空间

namespaces将不同的标识符号集合在一个具名作用域内，如果在namespace之内定义所有标识符号，则namespaces本身名称就成了唯一可能与其他全局符号冲突的标识符号，所以在标识符号前面加上namespace名字

处理方式与class类似

定义：

namespace josuttis{

class File;

void myGlobalFunc();

...

}

...

使用：

josuttis::File obj;

...

与class不同的是，namespace是开放的，可以在不同模块中定义和扩展

using namespace josuttis 会使namespace内的所有名字曝光，使用变量时不需要加作用域，但还是会出现名称冲突问题

三、关键字explicit

作用：禁止”单参数构造函数” 被用于自动型别转换。

阻绝以赋值语法进行带有转型操作的初始化

简单的理解就是阻止隐式转换的

class Stack{

public:

Stack(int s)

{

l = s;

}

private:

int l;

};

int main()

{

Stack s =40; //隐式转换调用构造函数

}

理解：

X x;

Y y(x); // 显示转换，根据型别X产生了一个型别Y的新对象

Y y = x; // 隐式转换，产生了一个型别Y的新对象

四、新的型别转换操作符（注意，这些操作符都只接收一个参数）

1.static\_cast

将以一种符合逻辑的方式转型。利用原值重建一个临时对象，并在设立初值时使用型别转换。处理相关类型的转换 指针到指针 整型到浮点型

double n = static\_cast<double>(a);

1. dynamic\_cast

处理基类到派生类型的转换

使用场合：基类必须有虚函数，即为多态 可以转换

Base \*p1 = new Base;

Derived \*p2 = dynamic\_cast<Derived\*>(p1);

1. const\_cast

用来移除const或volatile限定符

强制去掉const或volatile，除此之外不允许任何转换

4.reinterpret\_cast

处理互不相关类型的转换 指针到整型

int a=10;

double \*q = reinterpret\_cast<double \*>(a);

五、常量静态成员的初始化

类体内初始化一个const 静态数据成员时该成员必须仍然要被定义在类定义，但是因为这个静态数据成员的初始值是在类体中指定的, 所以在类定义之外的定义不能指定初始值,

class MyClass{

static const int num = 100;

...

}

const int MyClass :: num;

区分静态成员的初始化

通常静态数据成员在类声明中声明,在包含类方法的文件中初始化.初始化时使用作用域操作符来指出静态成员所属的类.但如果静态成员是整型或是枚举型const,则可以在类声明中初始化!!!

# include < iostream >

using namespace std ;

class test

{

public :

static int num;

} ;

int test : : num = 0;

void main( )

{

cout < < test : : num < < endl ;

test : : num = 20;

cout < < test : : num < < endl ;

}

一般地静态数据成员在该类定义之外被初始化如同一个成员函数被定义在类定义之

外一样,在这种定义中的静态成员的名字必须被其类名限定修饰,如上例中的

int test : : num = 0;

1. 头文件

c++中不需要.h

c的标准头文件也可以，但是必须采用前缀字符c

#include<cstdllib>

1. 配置器(Allocators)

采用特殊对象来处理内存配置和寻址，这样的对象称为配置器

c++标准程序库定义了一个缺省配置器

namespace std{

template <class T>

class allocator;

}

缺省配置器可在任何“配置器得以被当做参数使用”的地方担任默认值，缺省配置器会执行内存分配和回收的一般性手法，也就是呼叫new和delete操作符

1. 通用工具

1. Pairs(对组)

class pair 可以将两个值视为一个单元，C++标准程序库内多处使用了这个class，尤其容器类别map和multimap，就是使用pairs来管理其键值/实值(key/value)的承兑元素

任何函数返回两个值，也需要pair

1).头文件：<utility>

pair被定义为struct，而不是class 所以，所有的成员都是public，所以可以直接存取pair中的个别值

采用default构造函数生成一个pair，以两个“被该default构造函数个别初始化”的值作为初值。

std::pair<int,float>p; 以int和float来初始化p ，这两个构造函数都返回零值

2).Pair之间的比较

两个pair对象内的所有元素都相等，这两个pair对象就被视为相等

2.便捷函数 make\_pair()

template函数make\_pair() 使得无需写出型别，就可以生成一个pair对象

std::make\_pair(42,’@’);

等价于std::pair<int,char>(42,’@’);

3.Class auto\_ptr

头文件 #include<memory>

1).是一种智能型指针 帮助程序员防止“被异常抛出时发生资源泄漏”。

智能指针保证，无论在何种情形下，只要自己被摧毁，就一定连带释放其所指资源，而由于智能指针本身就是局部变量，所以无论是正常退出，还是异常退出，只要函数退出，它就一定会被销毁。

2). auto\_ptr是所指对象的拥有者，所以当身为对象拥有者的auto\_ptr被摧毁时，该对象也将遭到摧毁。auto\_ptr要求一个对象只能有一个拥有者，严禁一物二主。

std::auto\_ptr<ClassA> ptr(new ClassA);

等价于：ClassA \*ptr = new ClassA;

delete ptr;

这样就不再需要delete 也不需要catch

auto\_ptr 用operator->来指向对象中的成员，但是没有定义指针算术(++);

注意：auto\_ptr<> 不允许使用一般指针惯用的赋值(assign)初始化方式，必须直接使用数值来完成初始化。

std::auto\_ptr<ClassA> ptr2 = new ClassA ; //error

3).auto\_ptr拥有权的转移

a.使用auto\_ptr的copy构造函数将对象的拥有权交出去

std::auto\_ptr<ClassA> ptr1(new ClassA);

std::auto\_ptr<ClassA> ptr2(ptr1);

第一个语句中，ptr1拥有了那个new出来的对象，第二个语句中，拥有权由ptr1转交给ptr2，此后ptr2就拥有了那个new出来的对象，而ptr1不再拥有它，这样，对象就只会被delete一次-----在ptr2被销毁的时候

b.使用auto\_ptr的assignment操作符将对象拥有权交出去

std::auto\_ptr<ClassA> ptr1(new ClassA);

std::auto\_ptr<ClassA> ptr2;

ptr2 = ptr1;

如果ptr2被赋值之前正拥有另一个对象，赋值动作发生时会调用delete，将该对象删除

拥有者一旦交出拥有权，就两手空空，只剩一个null指针在手了。

只有auto\_ptr可以拿来当做另一个auto\_ptr的初值，普通指针是不行的。

std::auto\_ptr<ClassA> ptr;

ptr = new ClassA; //error

ptr = std::auto\_ptr<ClassA>(new ClassA);

1. .某个函数可以利用auto\_ptr将拥有权转交给另一个函数

a.某函数是数据的终点。即auto\_ptr以by value(传值)方式被当做一个参数传递给某函数，此时，调用端的参数获得了这个auto\_ptr的拥有权，如果函数不再将他传递出去，它所指的对象就会在函数退出时被删除。

void sink (std::auto\_ptr<ClassA>);

b.作为函数的返回值，当一个auto\_ptr被返回，其拥有权便转交给了调用端了

std::auto\_ptr<ClassA> f()

{

std::auto\_ptr<ClassA> ptr(new ClassA);

...

return ptr;

}

5).常数型auto\_ptr

const std::auto\_ptr<int> p(new int);

这样就无法交出拥有权

关键字const并非意味着你不能更改auto\_ptr所拥有的对象，而是意味着你不能更改auto\_ptr的拥有权。如果使用const auto\_ptr作为参数，对新对象的任何赋值操作都将导致编译期错误，就常数特性而言，const auto\_ptr比较类似常数指针，而非指向常数的指针

std::auto\_ptr<int> f()

{

const std::auto\_ptr<int> p(new int);

std::auto\_ptr<int> q(new int);

\*p = 42;

bad\_print(p); //error

\*p = \*q;

p = q; //error

return p;

}

6).auto\_ptr的错误应用

a.auto\_ptr之间不能共享拥有权

b.并不存在针对array而设计的auto\_ptr

auto\_ptr不可以指向array 因为auto\_ptr是透过delete而非delete[]来释放其所拥有的对象。

c.auto\_ptr绝非一个“四海通用”的智能型指针

d.auto\_ptr不能满足STL容器对其元素的要求

1. 辅助函数

算法程序库（头文件<algorithm>）内含有三个辅助函数，一个用来在两值之中挑选较大值，另一个用来在两值之间挑选较小值，第三个用来交换两值

1).max ()和min()

int i;

long j;

l = std::max(i.l); //errror

注意：min()和max()都要求他们所接受的两个参数的型别必须一致，如果不一致，则无法正确调用

可以明白的声明参数的型别

l = std::max<long>(i,l);

2).两值互换

std::swap(x,y);

5.头文件<cstddef>和<cstdlib>

<cstddef>

size\_t 一种无正负号的型别，用来表示大小(例如元素个数)

ptrdiff\_t 一种带有正负号的型号，用来表示指针之间的距离

<cstdlib>

exit() 会销毁所有static对象，将所有缓冲区清空，关闭所有IO通道，终止程序

abort() 立刻终止函数，不做任何清理工作

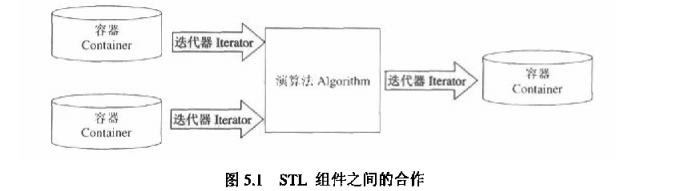
1. STL标准模板库

1.STL组件

容器(Containers):用来管理某类对象的集合

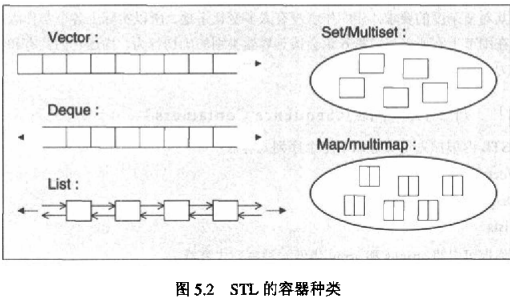
迭代器(Iterator): 用来在一个对象群集的元素上进行便历动作

算法(Algorithms): 用来处理群集内的元素，它们可以处于不同的目的而搜寻、排序、修改、使用那些元素。

STL的基本观念就是：将数据和操作分离，数据由容器类别加以管理，操作则由可定制的算法定义之，迭代器在两者之间充当粘合剂，使任何算法都可以和任何容器交互运作

2.容器(Containers)

用来管理一组元素，为了适应不同的需要，STL提供了不同类型的容器



总的来说：容器可分为两类

1).序列式容器：可序群集，其中每个元素均有固定位置，取决于插入时机和地点，和元素值无关 vector ,deque ,list

2).关联式容器：已序群集，元素位置取决于特定的排序准则 set multiset map multimap

关联式容器自动对其元素排序，也可以手动排序

序列式容器

(1).Vectors

将其元素置于一个dynamic array中加以管理，它允许随机存取，也就是说可以利用索引直接存取任何一个元素。

头文件 #include<vector>

声明 vector<int> a;

a.push\_back()函数为容器附加元素

a.size()成员函数返回容器中的元素个数

1. .Deques

是一个dynamic array 可以向两端发展，因此不论在尾部或头部安插元素都十分迅速。在中间部分安插元素则比较费时，因为必须移动其他元素。

头文件 #include<deque>

声明定义 deque<float> coll ;

coll.push\_front()函数用来安插元素，将元素安插在群集的前端

coll.push\_back()函数在尾部附加元素

(3).Lists

由双向链表实作而成，这意味着list内的每个元素都以一部分内存指示前驱元素和后继元素，List不提供随机存取，因此你要存取第10个元素，你必须沿着串链依次走过前9个元素。

优势是：在任何位置上执行安插或删除动作都非常迅速，因为只需改变连接（links）就行

头文件：#include<list>

声明： list<char> coll;

coll.empty()返回值判断容器中是否还有元素，返回false 表示容器中还有元素

coll.front()会返回第一个元素

coll.pop\_front();会删除第一个元素，但不会返回被删除的元素

(4).Strings

(5).arrays

array并非STL容器，没有类似size()和empty()等成员函数，但可以调用STL算法

关联式容器

关联式容器依据特定的排序准则，自动为其元素排序。排序准则以函数形式呈现，用来比较元素值（value）或元素键（key），缺省状态下以<进行比较

通常关联式容器由二叉树实作出来，在二叉树中，每个元素（节点）都有一个父节点和两个子节点；左子树的所有元素都比自己小，右子树的所有元素都比自己大。

(1).set

set的内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复

头文件 #include<set>

声明：set<int> coll;

coll.insert()函数用来插入数据

(2).Multisets

Multisets和set相同，只不过它允许重复元素

(3).map

map的元素都是“实值/键值”所形成的一个对组（key/value），每个元素有一个键，是排序准则的基础，每个键只能出现一次，不允许重复

头文件：#include<map>

声明： map<int,string> coll;

coll.insert(make\_pair(5,”a”));插入元素

允许使用[]安插元素

coll[key] = value;

(4).Multimaps

Multimap和map相同，允许重复元素

容器配接器

1. . stack 栈元素后进先出
2. .queue 队列先进先出
3. .Priority Queues 容器中的元素可以拥有不同的优先权，即基于一个排序准则

3.迭代器

(1). 迭代器是一个“可遍历STL容器内全部或部分元素”的对象，一个迭代器用来指出容器中的一个特定位置

基本操作 \* ++ == 　!= =

所有容器类别都提供有一些成员函数，使我们得以获得迭代器并以之遍访所有的元素

begin():返回一个迭代器，指向容器起始点，也就是第一个元素的位置

end(); 返回一个迭代器，指向容器结束点，结束点在最后一个元素之后 这样的迭代器又称为“逾尾”迭代器

声明: list<char>::const\_iterator pos; 声明一个指向list容器的迭代器

任何一种容器都定义有两种迭代器型别

container::iterator 迭代器以“读/写”模式遍历元素

container::const\_iterator 迭代器以“只读”模式遍历元素

pos = coll.begin();

pos = coll.end();

for(pos = coll.begin();pos!=coll.end();++pos)

cout<<\*pos<<endl;

前置式递增 ++pos 比后置式递增pos++效率高，后者需要一个额外的临时对象，因为它必须存放迭代器的原本位置并将它返回。

迭代器在键值对中的应用

map<int ,string>::iterator pos;

pos->first; //取key

pos->second; //取value

1. .迭代器分类

a.双向迭代器

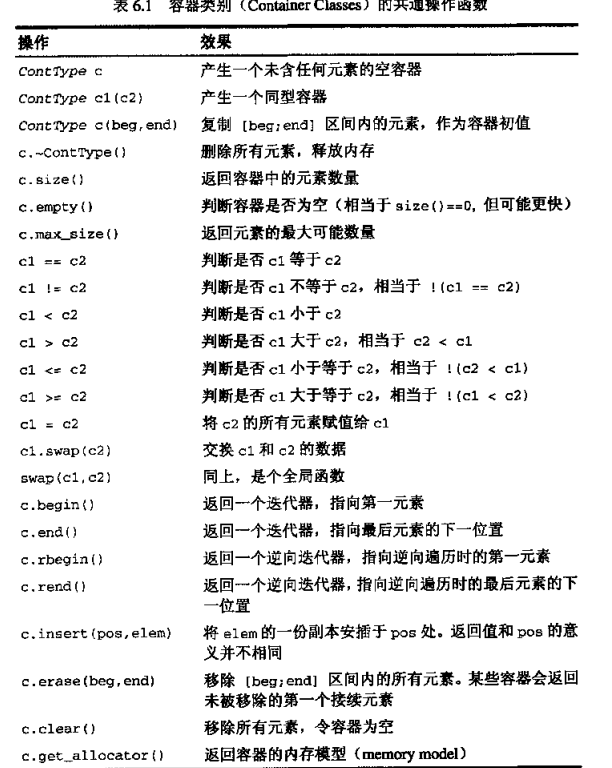
可以双向进行，以递增运算前进或以递减运算后退。list、set、multiset、map、multimap这些容器所提供的迭代器都属于此类。

b.随机存取迭代器

不但具备双向迭代器的所有属性，还具备随机访问能力。

vector 、deque、strings所提供的迭代器都属于此类

4.容器的共同操作



1. vector

(1). 动态数组 将其元素置于一个dynamic array中加以管理，它允许随机存取，也就是说可以利用索引直接存取任何一个元素。

(2).大小（size）和容量(Capacity)

vector之中用于操作大小的函数有size() ,empty(),max\_size()

另一个与大小相关的函数是capacity(),它返回vector实际能够容纳的元素数量，如果超越这个数量，vector就有必要重新配置内部存储器。

1).vector容量很重要的原因:

a.一旦内存重新配置，和vector元素相关的所有references、pointers、iterators都会失效

b.内存重新配置很耗时间

2).重新配置内存的方法：

a.使用reserve()保留适当容量，避免一再重新配置内存

std::vector<int> v;

v.reserve(80);

b.初始化期间就向构造函数传递附加参数，构造出足够的空间。如果你的参数是个数值，它将成为vector的起始大小

std::vector<T> v(5);

3).间接缩减vector容量的小窍门

a.两个vector交换内容后，两者的容量也会互换因此下面的例子虽然保留了元素，却缩减了容量

template<class T>

void shrinkCapacity(std::vector<T>& v)

{

std::vector<T> tmp(v);

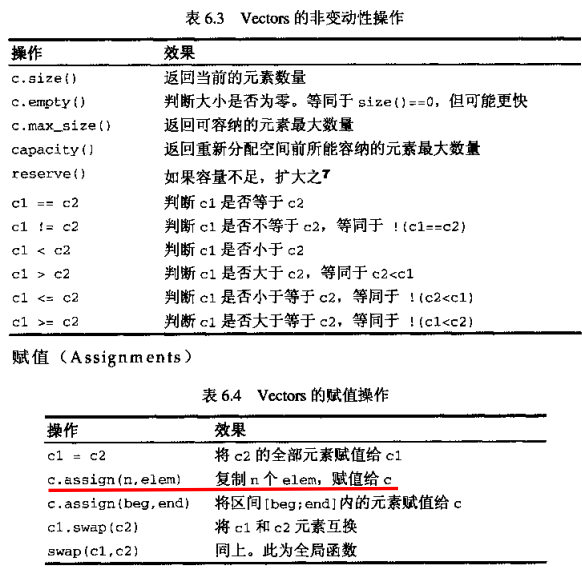
v.swap(temp);

}

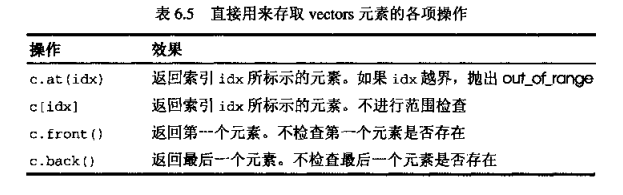
b.利用下面的语句直接缩减容量

std::vector<T>(v).swap(v);

1. .vector的操作函数



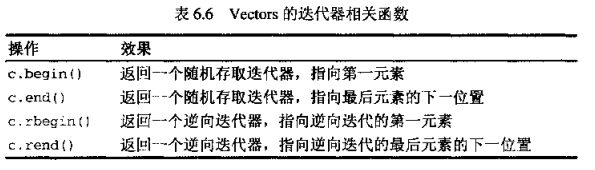
(4).元素存取



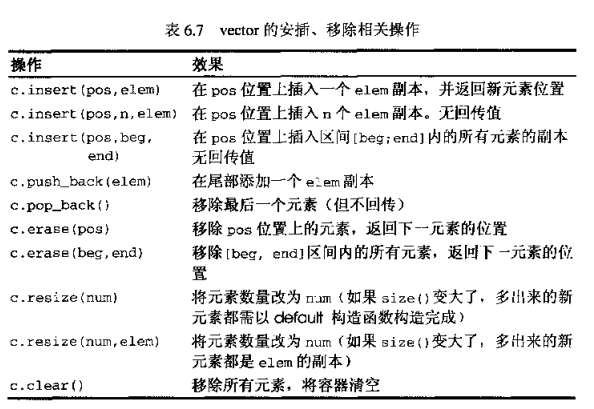
使用索引，确定索引有效。使用front和back必须确定容器不为空

(5).迭代器的相关函数

vector迭代器是r随机存取迭代器



(6).安插和移除元素



移除与某值相等的第一个元素

std::vector<int> coll;

std::vector<int>::iterator pos;

pos = find(coll.begin(),coll.end(),val);

if(pos!=coll.end()){

coll.erase(pos);

}

十一、Deques

1. 也是一种动态数组，与vector不同的是数组头尾都开放，因此能在头尾两端进行快速安插和删除

头文件：#include<deque>

2.与vector的比较

不同点：(1).两端都能快速安插元素和移除元素，vector只能在尾部插入，这些操作可以在分期摊还的常数时间内完成

(2).存取元素时，deque的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一点

(3).迭代器需要在不同区块间跳转，所以必须是特殊的智能型指针,非一般指针

(4).deque的max\_size()可能会更大

(5).deque不支持对容量和内存分配时机的控制。除了在头尾两端，在任何地方安插或删除元素，都将导致指向deque元素的任何pointers、reference、iterators失效。不过，deque的内存重分配优于vectors，因为其内部结构显示，deques不必再内存重分配时复制所有元素

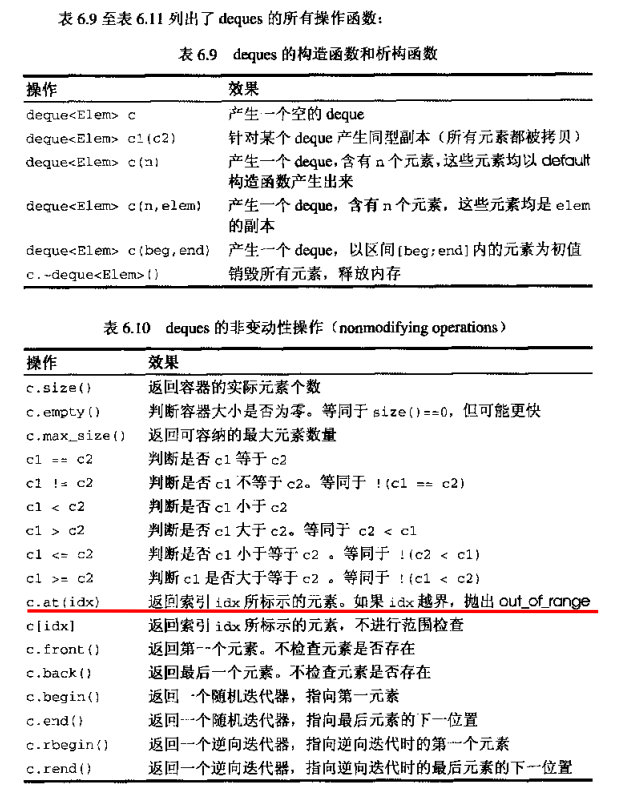
(6).deque的内存区块不再被使用时，会被释放，deque的内存大小时可缩减的

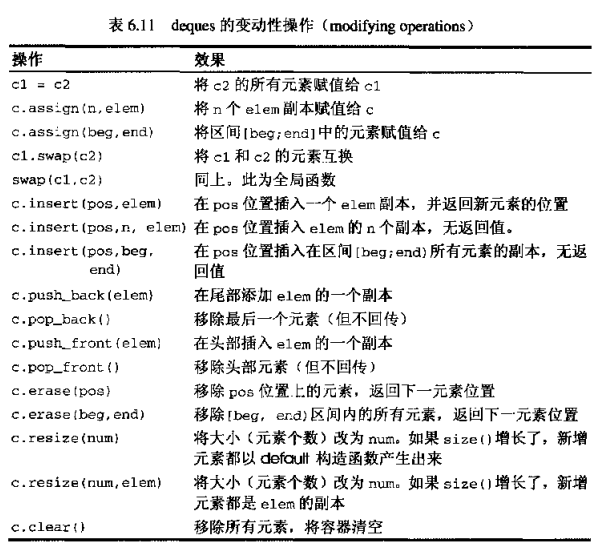
相同点：

(1).在中段部分安插、移除元素的速度相对较慢，因为所有元素都需要移动以腾出或填补空间

(2).迭代器属于随机存取迭代器

3.操作函数





十二、List

1.List使用一个双向链表来管理元素

头文件：#include<list>

2.特点：

(1).list不支持随机存取

(2).任何位置上执行元素的安插和移除都非常快，始终都是常数时间内完成，因为无需移动其他任何元素

(3).安插和删除动作并不会造成指向其它元素的各个pointers、reference、iterators失效

(4).异常处理，要么成功，要么什么都不发生

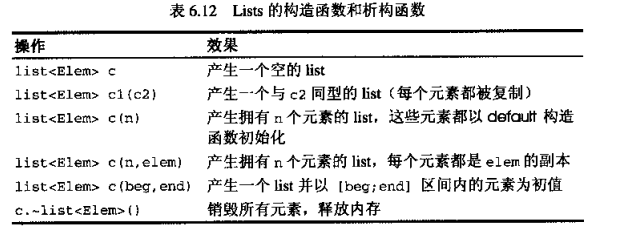
3.区别vector和deque

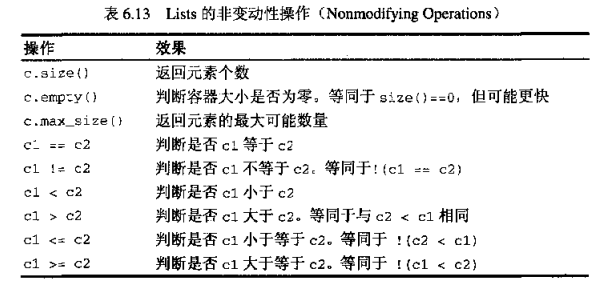
(1).不支持随机存取，不提空下标操作和at()

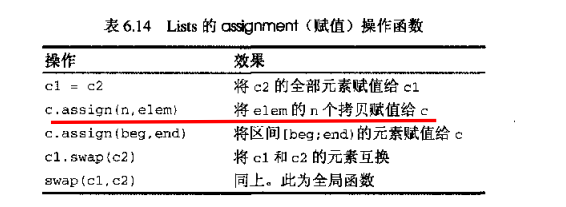
(2).不提供容量、空间重新分配等操作函数

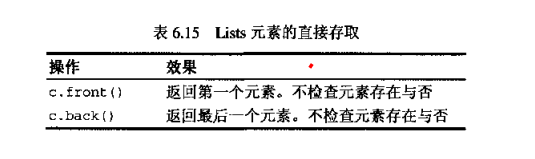
(3).提供特殊的成员函数，专门用于移动元素

4.list的操作函数

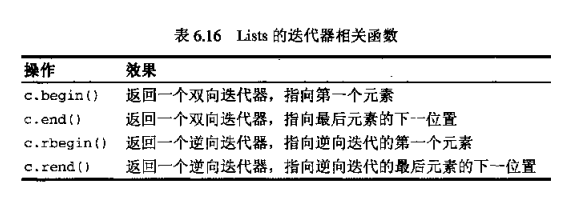




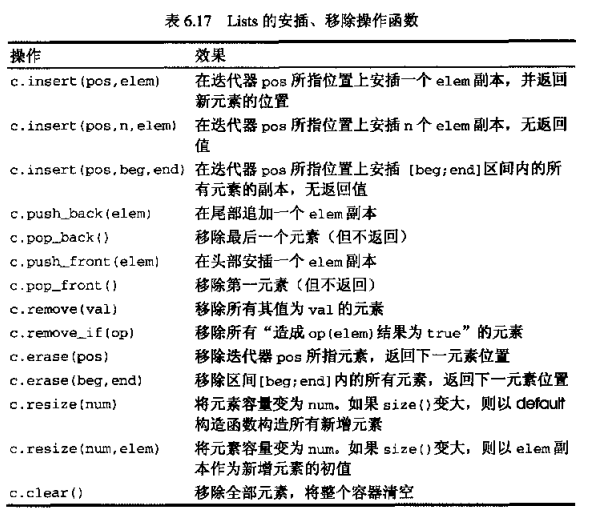


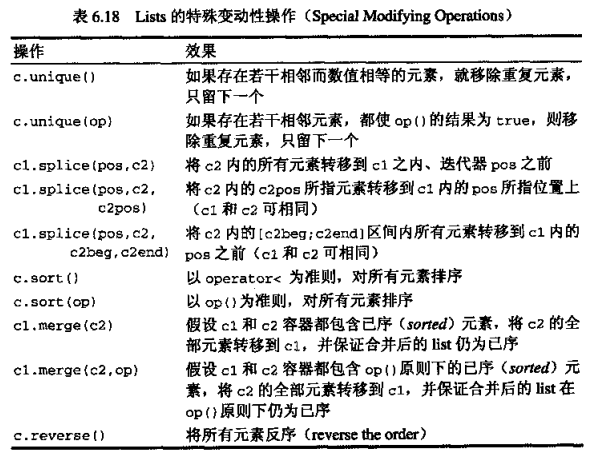


5.迭代器的相关函数

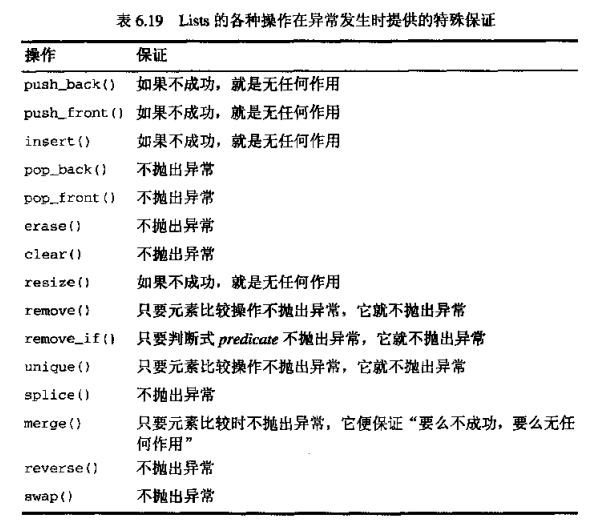


6.元素的安插和移除



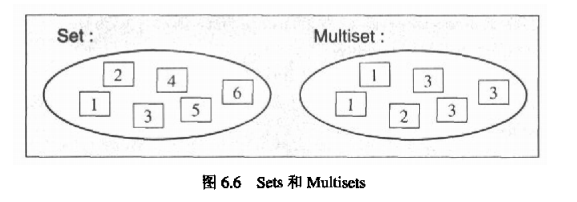


7.异常处理



1. set和multiset

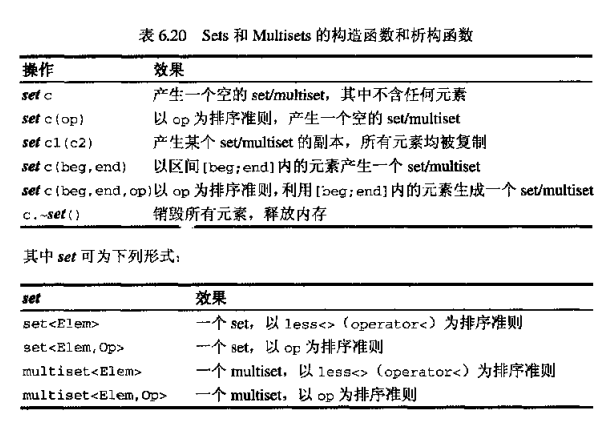
1.set和multiset会根据特定的排序准则，自动将元素排序，两者不同之处在于multiset允许元素重复而set不允许

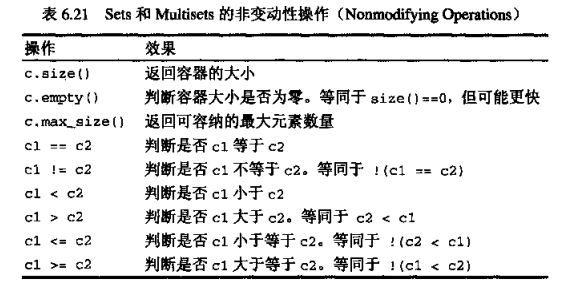


头文件：#include<set>

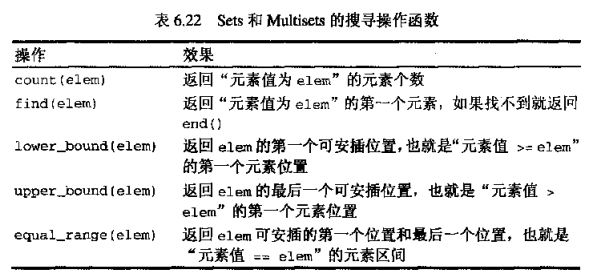
内部结构为平衡二叉树

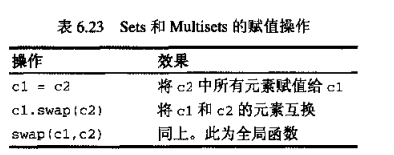
2.操作函数

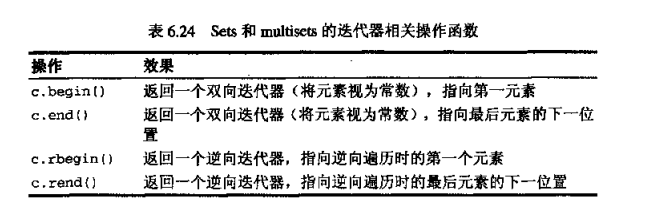




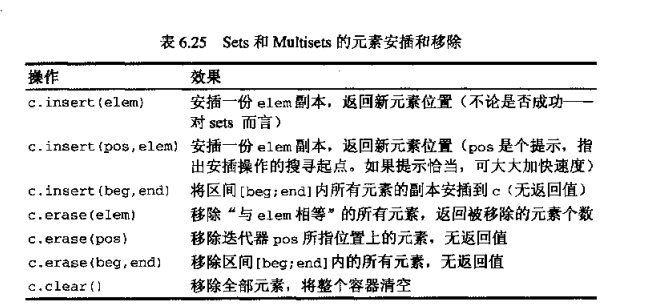
元素比较动作只能用于型别相同的容器





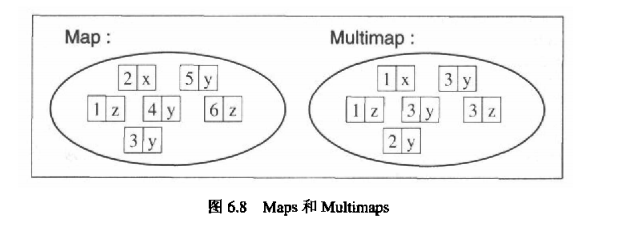


1. 安插和删除函数



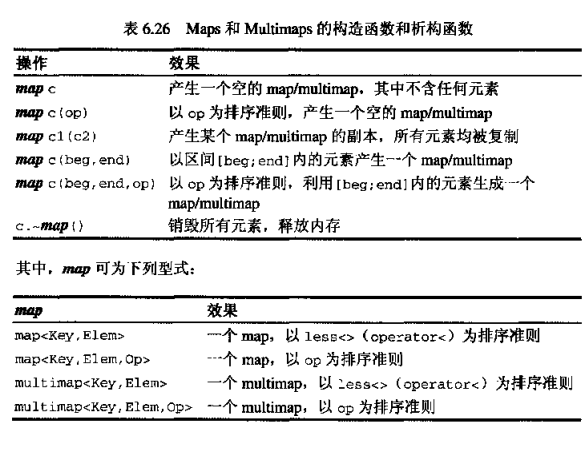
1. map和multimap
2. 将key/value pair当做元素，进行管理，可根据key的排序准则自动将元素排序，multimap允许重复元素，map不允许

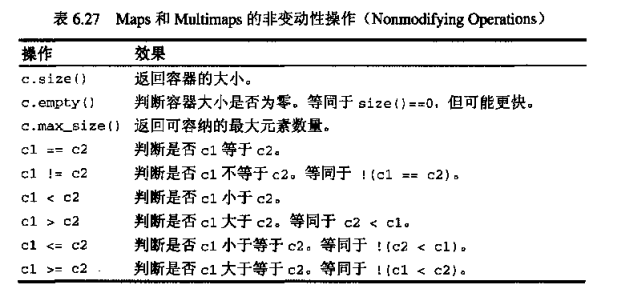
平衡二叉树实现

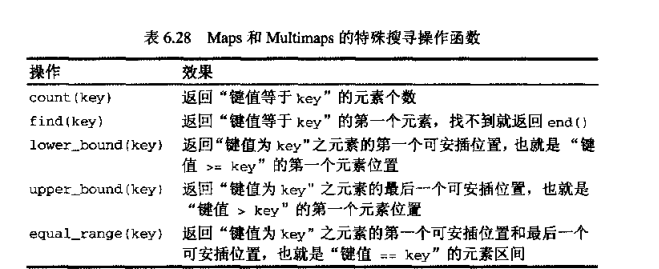


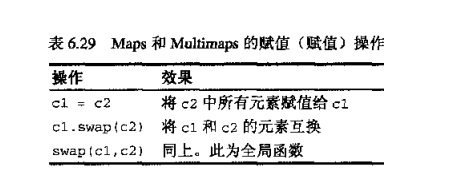
头文件 #include<map>

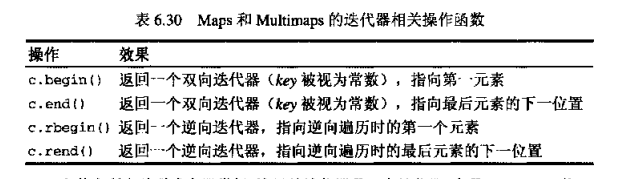
2.操作函数



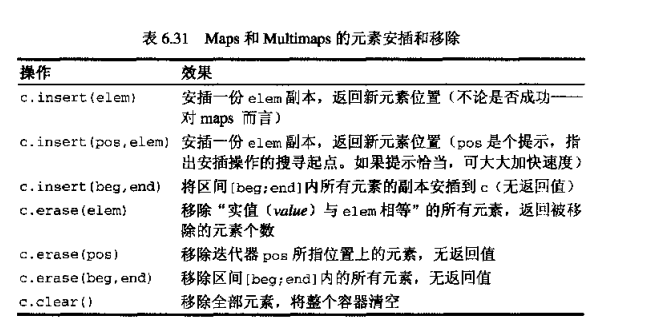






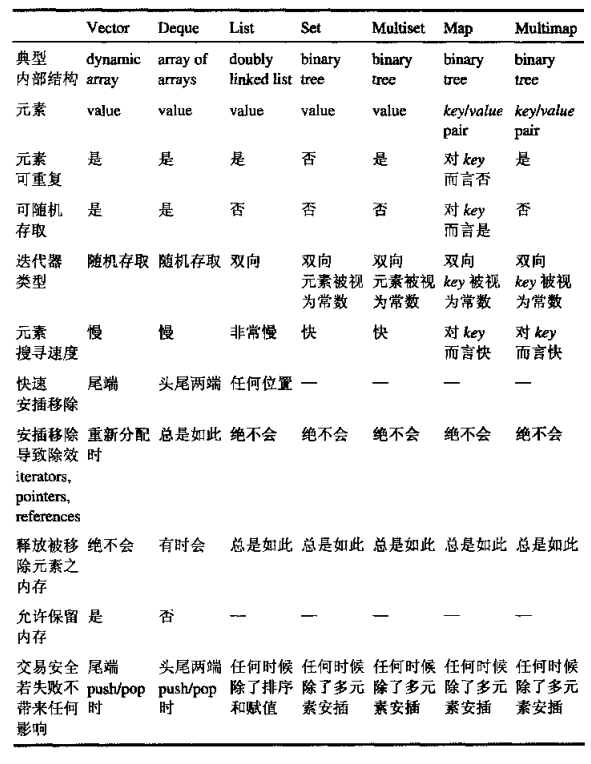


1. 元素的安插与移除



4.元素存取

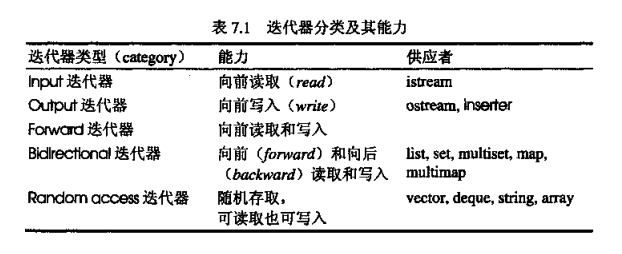
关联式容器不提供元素的直接存取，必须依靠迭代器，但是map是个例外，提供下标操作符，支持元素的直接存取，不过下标为元素的key。



1. 迭代器

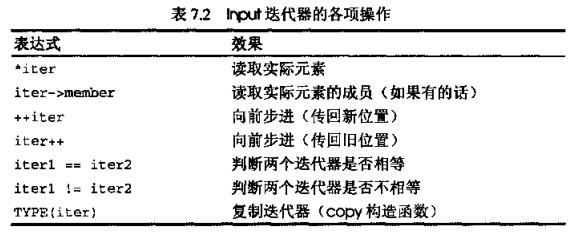
1.迭代器是一种“能够遍历某个序列内的所有元素”的对象。它可以透过与一般指针一致的接口来完成自己的工作。

2.迭代器的分类



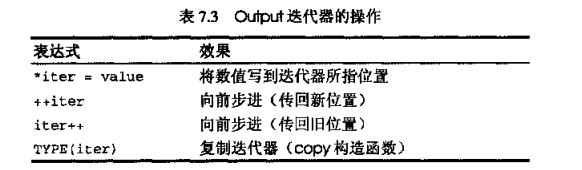
3.Input（输入）迭代器

只能一次一个向前读取元素，按此顺序一个个传回元素值



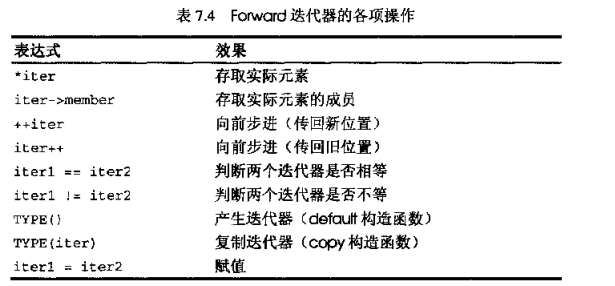
1. Output（输出）迭代器

将元素值一个个写入，即只能一个元素一个元素地赋新值



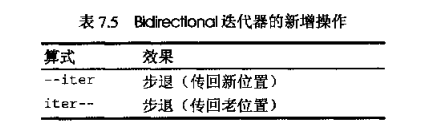
1. Forward（前向）迭代器

是Input迭代器和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。



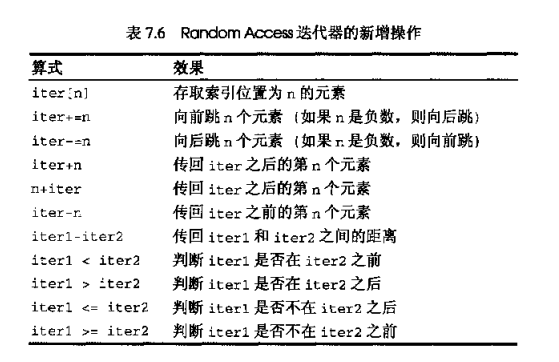
1. Bidirectional（双向）迭代器

在Forward迭代器的基础上增加了回头遍历的能力，即增加了减操作



1. Random Access（随机存取）迭代器

在Bidirectional迭代器的基础之上再增加随机存取能力，所以必须提供“迭代器算术运算”。



1. 迭代器相关辅助函数

三个辅助函数：advance() 、distance()、iter\_swap()

(1).advance()

可将迭代器的位置增加，增加的幅度由参数决定，即使迭代器一次前进（或后退）多个元素

#include<iterator>

void advance(InputIterator& pos,Dist n);

(2).distance()

用来处理两个迭代器之间的距离

#include<iterator>

void distance(InputIterator pos1,InputIterator pos2);

(3).iter\_swap()

交换两个迭代器所指的内容

#include<iterator>

void iter\_swap(ForwardIteraor1 pos1, ForwardIterator2 pos2);

迭代器的型别不必相同，但其所指的两个值必须可以相互赋值。

1. 仿函数

所谓仿函数，是一个定义operator()的对象：

FunctionObjectType fo;

fo(...);

其中表达式fo()系调用仿函数fo的operator(),而非调用函数fo()

可以将仿函数视为一般函数，只不过用的是一种更复杂的撰写手段，并非将所有语句放在函数体中,而是在仿函数类别的operator()体内撰写程序代码

class FunctionObjectType{

public:

void operator(){

statements

}

};

例：

#include <iostream>

#include<string>

#include <set>

#include <algorithm>

using namespace std;

class Person{

public:

string firstname() const;

string lastname() const;

};

class PersonSortCriterion{

public:

bool operator()(const Person& p1,const Person& p2)const{

return p1.lastname()<p2.lastname() ||

(!(p2.lastname()<p1.lastname())&&p1.firstname()<p2.firstname());

}

};

int main()

{

typedef set<Person,PersonSortCriterion> PersonSet;

PersonSet coll;

PersonSet::iterator pos;

for(pos=coll.begin();pos!=coll.end();++pos)

{

}

return 0;

}

此时的排序准则为PersonSortCriterion