C++标准库笔记目录

[**一、** **智能指针** 2](#_Toc8135593)

[**二、** **外覆器** 5](#_Toc8135594)

[**三、** **辅助函数** 6](#_Toc8135595)

[**四、** **一些头文件** 7](#_Toc8135596)

[**五、** **标准模板库** 8](#_Toc8135597)

[**六、** **STL容器** 19](#_Toc8135598)

[**七、** **细探STL容器成员** 30](#_Toc8135599)

[**八、** **迭代器** 41](#_Toc8135600)

[**九、** **STL函数对象及Lambda** 49](#_Toc8135601)

[**十、** **STL算法** 53](#_Toc8135602)

[**十一、** **特殊容器** 64](#_Toc8135603)

[**十二、** **字符串** 65](#_Toc8135604)

[**十三、** **并发** 69](#_Toc8135605)

1. **智能指针**
2. **shared\_ptr**

shared\_ptr提供了共享式拥有语义。即，多个shared\_ptr可以共享或者拥有同一对象，对象的最末一个拥有者有责任销毁对象，并清理对象相关的所有资源。

Tips1:因为”接受单一pointer作为唯一实参”的构造函数是explicit，所以不能够使用赋值符来直接构造一个shared\_ptr。要么就使用便捷函数make\_shared().如:

Shared\_ptr<string> Pnico = make\_shared<string>(“nico”);

Tips2:可以为shared\_ptr声明专属deleter。方法是在其构造函数的参数列中，加入第二实参。这个第二实参可以是个函数对象(operator())、函数、或是一个lambda函数。

Tips3:shared\_ptr提供的default deleter调用的是delete而不是delete[],所以shared\_ptr只有是由new建立起来的单一对象，default delete方才能够适用。否则，如果是一个array of object，那么需要传递一个函数、函数对象、或者lambda，让他们针对传入的寻常指针调用delete[]。

TIPS4:确保对象只被一组(可以有很多个)shared\_ptr拥有。如果还被其他类型的普通指针拥有，可能会导致重复销毁。

TIPS5:如果要建立一个”得自this的shared\_ptr”,那么可以继承class std::enable\_shared\_from\_this<>，然后可以使用一个派生的成员函数shared\_from\_this()建立起一个源自this的正确shared\_ptr。

TIPS6:一个empty的shared\_ptr不能分享对象拥有权，所以use\_count返回0.

TIPS7:一旦拥有权转移至一个”已拥有其他对象”的shared\_ptr(move操作)，deleter就会针对先前被拥有的那个对象调用。

TIPS8:两个shared\_ptr进行赋值操作，那么右值原对象的引用计数+1，左值对象的引用计数-1。（表示左shared\_ptr共享右值shared\_ptr的对象）

TIPS9:一般而言，shared\_ptr并不线程安全。因此，为避免data race造成的不明确行为，当在多线程中以shared\_ptr指向同一对象，必须使用诸如mutex和lock等技术。

1. **Weak\_ptr**

Weak\_ptr允许“共享但不拥有”某对象，即其不会增加对象的引用计数。一旦最末一个拥有该对象的shared pointer失去拥有权，任何weak\_pointer都会自动变空。

另外，weak\_ptr不能够使用操作符\*和->访问对象，而必须建立另一个shared\_pointer.

Shared\_prt使用时会存在“环形引用问题”，引入weak\_ptr就可以解决了。

Tips1:weak\_ptr的use\_count返回的是shared\_ptr的拥有次数。

Tips2:weak\_ptr可以为空。而使用lock()会产出一个shared pointer,因此在那个shared pointer寿命期间，该对象的引用计数会多1.这是处理weak pointer所共享对象的**唯一途径**！

1. **Unique\_ptr**

此类ptr实现了独占式拥有的概念。意味着其可以确保一个对象和其相应资源在同一时间只被一个pointer拥有，一旦拥有者被销毁或者变为empty，或者开始拥有另一个对象，先前拥有的那个对象就会被销毁，其拥有的任何资源也会被释放。

Class unique\_ptr继承自class auto\_ptr,后者由C++98引入但是已经不再被认可。Class unique\_ptr提供了一个更加简明干净的接口，比auto\_ptr更不容易出错。

Class unique\_ptr的目的:

1. 获得某些资源
2. 执行某些操作
3. 将取得的资源释放掉

TIPS1:unique\_ptr有着与寻常pointer相似的接口，可使用操作符\*和->等。但是其不提供运算符如++等，这被视为优点。因为pointer算术运算通常是麻烦的来源。

TIPS2:class unique\_ptr<>不允许以赋值语法将一个寻常的pointer当作初值。因此必须直接初始化unique\_ptr，像下面这样:

Std::unique\_ptr<int> up(new int) //OK

Std::unique\_ptr<int> up = new int; //ERROR

TIPS3:unique\_ptr可以是empty。可以对其赋值nullptr或调用reset。

TIPS4:可以调用release(),以获得unique\_ptr()拥有的对象并放弃所有权。

Std::string sp = up.realse() //up is a unique\_ptr

TIPS5:不可以对unique pointer执行copy或者assign,但可以使用C++的move语义来转移所有权。如下：

Std::unique\_ptr<ClassA> up1(new ClassA);

Std::unique\_ptr<ClassA> up2(std::move(up1));

TIPS6:如果将一个由std::move()建立起来的unique\_ptr以rvalue reference的身份当作函数实参，那么被调用的函数参数将会取得unique\_ptr的所有权，因此，如果该函数不再转移所有权，对象会在函数结束时被delete

TIPS7:当函数返回一个unique\_ptr，其拥有权会转移至调用端的场景内。C++11规定，编译器应该自动尝试加上move。

TIPS8:在class内使用unique\_ptr可以避免资源泄露。如果使用unique\_ptr代替寻常pointer，就不再需要析构函数，因为对象被删除会连带所有成员删除。

TIPS9:C++标准库为unique\_ptr提供了一个偏特化版本用来处理array，这个版本会在遗失对象的所有权时，自动为对象调用delete[],而无需与shared\_ptr一样自定义一个deleter。值得注意的是，这个偏特化版本不提供\*和->,而是提供[]，用来访问array中的每一个对象。

TIPS10:当所指的对象要求不止是调用delete或者deleter，就只能具体指定自己的deleter。而其定义方式略不同于shared\_ptr。必须指明deleter的类型作为第二个template实参。该类型可以实函数引用，或者是个函数指针，或者是个函数对象。

**4.auto\_ptr**

Auto\_ptr是C++98标准库提供的，如今已被C++11明确声明不再支持。其目的是为了提供如今unique\_ptr的语义，不过却导致了一些问题

1. 设计它时，C++尚未有move语义让构造函数和assignment操作符使用，然而其目标仍然是提供转移语义。是copy和assignment操作符还是获得了move语义，但却可能导致严重问题，特别是当auto\_ptr为实参时。
2. 不存在deleter所表示的语义，因此只能使用它处理”以new分配之单一对象”
3. 由于其是C++标准库提供的最早且唯一的smart pointer，所以常常被误用，特别是它僭越地提供了如今class shared\_ptr所提供地拥有共享权语义。

关于非自觉遗失所有权地危险，如下例:

Template <typename T>

Void bad\_print(std::auto\_ptr<T> p){

If(p.get()==NULL)

Std::cout << “NULL”;

Else

Std::cout<<\*p;

}

无论何时，当某个auto\_ptr被传递给bad\_print()时，它所拥有地对象就会被删除。原因是，被当作参数传递地那个auto\_ptr的拥有权被转移给了参数p，而p会在函数结束前就删除它所拥有的对象。这可能引发致命错误，如下

Std auto\_ptr<int> p(new int);

\*p = 42;

Bad\_print(p);

\*p = 18; //RUNTIME ERROR

传递auto\_ptr给容器时，特别容易发生上述行为。现在，有了unique\_ptr,这样错误不可能再发生。因为你必须很明显地必须使用std::move()传递实参。

1. **外覆器**
2. **Reference Wrapper**

声明于<functional>中的class std::reference\_wrapper<>主要用来“喂”reference给function template，后者原本以by value的方式接受参数。对于一个给定类型T,这个class提供ref()用以隐式转换为&T.一个cref()的调用隐式转换为const T&，这往往允许function template得以操作reference而不需要另写特化版本。

这个特性被用于各个地方，如:

Make\_pair()用此创建一个pair<> of reference

Make\_tuple()用此创建一个tuple<> of reference

Binder于是能绑定reference

Thread用此能够以by reference的方式传递实参

另外，class reference\_wrapper使得可以使用reference作为最高级对象，例如作为array或者STL容器的元素类型。

1. **Function Type Wrapper**

Class std::function<>，提供多态覆盖，可以概括function pointer记号。这个class允许把可调用对象(callable object,如function,member function,.function object和lambda)当作最高级对象。

使用member function时，”借以调用它们”的那个对象必须被当作第一实参。比如下面

Class C{

Publiuc :

Void memfunc(int x, int y) const;

};

Std::function<void(const C&,int,int)> mf;

Mf = $C::memfunc;

Mf(C(),42,77);

值得注意的是，执行一个函数调用，却没有标的物可以调用。将会抛出std::bad\_function\_call。

1. **辅助函数**

**1.两值互换(swapping)**

函数swap()用来交换两个对象的值，其泛化实现定义于<utility>。

在其内部，数值被**moved或moved assigned**。运用这个函数，可以交换任意变量x和y的值。在C++11前，数值是被assigned或者copied。

Namespace std{

Template <typename T>

Inline void swap(T &a,T &b)…{

T tmp(std::move(a));

a = std::move(b);

b = std::move(tmp);

}

}

当然，只有参数类型提供了copy和move语义，上述调用才会成功。

Swap()最大的优势在于，通过模板特化或者函数重载，可以为复杂类型提供特殊的实现版本！

自C++11起，C++提供了一个针对array的重载版本

Namespace std{

Template <typename T,size\_t N>

Void swap(T(&a)[N],T(&b)[N]){

Noexcept(noexcept(swap(\*a,\*b)));

}

}

**2.Clock和Timer**

过去,C和POSIX提供系统时间接口，允许从秒转换到毫秒、再到微妙乃至纳秒。问题是每次转换都需要一个新接口，针对这个问题，C++11起提供一个精度中立的程序库，也被称为chrono程序库，其特性被定义于<chrono>.

另外，C++的标准库也提供一个基本的C和POSIX接口，用来处理日历时间。当然，还可以使用C++11开始的thread程序库，等待某个线程或者程序一段时间。

TIPS1:sleep\_for()和sleep\_until()，由this\_thread提供用以停滞线程

TIPS2:try\_lock\_for()和try\_lock\_until(),用来在等待一个mutex时指定最大时间段

TIPS3:wait\_for()和wait\_until()，用来等待某条件成里或者等待一个future时指定最大时间段。

1. **一些头文件**

**1.<cstddef>的各项定义**

|  |  |
| --- | --- |
| 标识符 | 意义 |
| NULL | 指针值，表示未定义或者无 |
| nullptr\_t | nullptr的类型 |
| size\_t | 一种无正负号类型，用来表示大小 |
| ptrdiff\_t | 一种带正负号类型，表示指针间距 |
| max\_align\_t | 所有环境最大齐位对于的类型 |
| offserof(type,mem) | 表示成员mem在某个结构/联合体中的偏移量 |

C++保证NULL就是0(类型可为int或者long)，在C中通常为(void\*)0，然而这在C++中不正确，因为NULL必须是整数类型，否则无法将NULL赋予一个pointer。因为C++没有定义从void\*到任何其他类型的自动转型操作！NULL同时也定义于<cstdio>、<cstdlib>、<cstring>、<ctime>、<cwhar>、<clocale>内。

**2.<cstring>中的定义式**

|  |  |
| --- | --- |
| 定义 | 意义 |
| Memchr(const void\*,int,size\_t) | 在ptr所指的前len哥byte中找出字符c |
| Memcmp(const void\*,int,size\_t) | 比较ptr1、ptr2的前len个byte |
| Memcpy(void\*,const void\*,size\_t ) | 将src的前len个bytes数据复制到dst内 |
| Memove(void\*,const void\*,size\_t) | 将src的前len个bytes数据复制到dst内(可重叠) |
| Memset(void\*,int,size\_t) | 将ptr所指的前len个字符赋值为c |

Cstring中这些定义式十分重要，用以设定、拷贝、搬移内存数据。

1. **标准模板库**

**5.1STL组件**

若干精心勾画的组件共同合作，构筑起STL的基础。这些组件中最关键的是容器、迭代器和算法。

STL的基本观念是把数据和操作分离(从某种意义上说，STL的概念OOC最初思想是矛盾的，然而这么做有很重要的原因，就是可以将各种容器与各种算法结合起来，在很小的框架内达到非常大的弹性)。数据内容由容器管理，操作则可以定义算法，迭代器在两者间作为粘合剂。

STL另一个重要特性是，所有组件都可以针对任意类型运作。因此其为泛型编程的一个出色代表。

5.2容器

容器用来管理一大群元素，为了适应不同需要，STL提供了不同容器，如Array、Vector、deque、List、Forward-List、Set、Map等。

总体而言，可分为三大类容器：序列式容器(有序集合，即array、vector、deque、kust和forward-list)、关联式容器(已排序集合，包括map、multimap、set、multiset)、无序容器(unordered collection)。

其中，有序容器常以array或者linked-list实现。关联式容器常以二叉树实现。无序容器常以hash table实现。

**5.2.1序列式容器**

**Vector:**Vector将元素置于一个动态数组管理。其允许随机访问。在尾部添加或者删除元素都很快，但是在中段安插或者删除就比较费时---因为在中段安插或删除后部的元素都要进行移动以保持相对次序。

TIPS：在尾部插入/删除语句对应push\_back()和pop\_back();

**Deque(双队列容器):**这是一个动态数组，可以向两端发展。因此不论在头部还是尾部插入都十分迅速。同理，在中段插入比较费时。

Tips:使用push\_back()和push\_front()分别在头尾部插入元素。

**Array:**一个array对象是在某个固定大小的数组内管理元素。因此，不可以改变元素个数，只能改变元素值。

**List:**List由双向链表实现，不提供随机访问。List的优势是，在任何位置上进行插入或者删除都很快速。因为只需要改变链接就好。

**Forward List:**C++11后，标准库提出了新容器forward list。其是由元素构成的单向linked list，原则上forward list就是一个受限的list。不支持任何向后移动或者效率低下的操作。因此，其不提供成员函数如push\_back()乃至size()

**5.2.2关联式容器**

关联式容器的优点是，可以很快找出一个具有特定value的元素，因为其具备对数复杂度。缺点是，不可以直接改变元素的value，否则会破坏元素的自动排序。此类容器一般用二叉树实现。

**Set:**元素依据其value自动排序，每个元素只能出现一次，不允许重复

**Multiset:**和set的区别是，元素可以重复

**Map:**每个元素都是key/value。其中key是排序的准则，每个key只可以出现一次，不允许重复。Map也可以视为一种关联式数组，就是“索引为任意类型”的数组。Map的底层实现一般是红黑树

**Multimap:**和Map的区别是，元素可以重复。Multimap可以被当作字典使用。

**5.2.3无序容器**

在无序容器中，元素没有明确的次序。通常以hash table实现，内部是一个由linked list组成的array(即采用链接法解决冲突的哈希表)。无序容器的主要优点是，当打算查找一个特定值的元素，其速度可能快过关联式容器。事实上其提供的是常量复杂度，前提是有一个很号的hash函数，可能需要提供很多内存作为bucket。

**Unordered set**:无序元素的集合，每个元素仅能出现一次

**Unordered multiset:**与unordered set的区别是可以有重复元素

**Unordered map:**每个key只能出现一次，不允许重复

**Unordered multimap:和unordered map**的区别是，可以有重复的key，也可以用作字典。

**5.2.4关联式容器**

不论是map还是unordered map，都是键值对形成的集合。每个元素带有独一无二的key，如此的集合可以被视为一个关联式数组，也就是索引并非整数的array。也因此，刚才的那两个容器都提供了下标操作符[ ].

比如下面程序

Unordered\_map<string,float> coll;

Coll[“VAT1”] = 0.16

Coll[“VAT1”] += 0.03;

Cout<<”VAT 1 IS ”<<coll[“VAT1”];

这里[]操作符的行为和寻常数组下标操作符有些不同:其索引非整数；它接受一个新索引，并以此建立和安插一个新元素，该元素以该索引为key，所以绝对不能给其一个无效索引。

也就是说，程序中的

Coll[“VAT1”] = 0.16

语句乃是建立一个新元素，其key为“VAT1”，value为0.16.如果没有右值，则使用default构造，对于基础类而言初值为0.

从C++11开始，可以使用at()访问元素的value，只需要传给它key就行。这种情况下如果找不到对应的key，会导致一个out\_of\_range异常。

而语句

Coll[“VAT1”] += 0.03;

Cout<<”VAT 1 IS ”<<coll[“VAT1”];

获得的是对value的读/写权力。

**5.2.5容器适配器**

除了上述个根本容器，为了满足特殊需求，C++标准库还提供了一些所谓的容器适配器，它们也是预先定义好的容器，提供一定限度的接口，用以应付特殊需求。

**Stack:**栈容器，其中元素采取LIFO(后进先出)管理策略

**Queue:队**列容器，其元素采用FIFO(先进先出)管理策略，也就是说，它是一个寻常的缓冲区。

**Priority Queue:**内部元素有优先权，这种容器相当于这样的缓冲区:”下一元素永远是容器中优先级最高的元素”，如果同时有多个最高优先级元素，其次序没有明确定义。

**5.3迭代器(Iterator)**

自C++11起，可以使用range-based for循环来处理所有元素。然而如果只是要找出某些元素而无需处理所有元素，应当迭代所有元素，直到找到目标。因此需要这样一个概念:以一个对象表现出容器元素的位置，其被称作迭代器。

迭代器是一个”可遍历STL容器全部元素的对象”，基本操作如下

Operator \*:返回当前位置上的元素，如果该元素有成员，可以用->操作符来取用之

Operator ++:令迭代器前进至下一元素。大多数迭代器还可以使用- -退至前一元素.

Operator ==和!=:判断两个迭代器是否指向同一位置

Operator =:对迭代器赋值(也就是指明迭代器所指向的元素位置)

这些操作符和运用指针操作寻常的数组元素时的接口一致。差别在于，迭代器是智能指针。

所有容器类型都提供一些基本的成员函数，使得我们得以取得迭代器并以之遍历所有元素.

Begin():返回一个迭代器(container::iterator类型，不是常量型)，指向容器的起点

End(),返回一个迭代器，指向容器终点---注意，终点是最末元素的下一位置，这样的迭代器被称作逾尾迭代器。

任何容器都定义有两种迭代器类型

Container::iterator

Container::const iterator

分别为读写模式和只读模式。

如果容器是空的，那么coll.begin()就是coll.end()，循环根本不会被执行。

TIPS1:一般使用++pos而不是pos++，因为前置式递增效率比后置式递增高。后者内部需要一个临时对象，因为它必须存放迭代器的原本位置并返回之。

TIPS2：自C++11起提供cbegin()和cend（）,其返回container::const iterator类型。

TIPS3：ranged-base for循环，如下

For(type elem : coll){

…

}

被解释为

For(auto pos = coll.begin(),end=coll.end();pos!=end;++pos){

Type elem = \*pos;

…

}

这里旧可以理解为什么声明elem为一个常量引用可以避免非必要复制了。如果不这么做，elem会被初始化为\*pos的拷贝。

**5.3.1迭代器种类**

根据能力不同，迭代器被分为5种，STL预先定义好的所有容器，其迭代器均属于以下三种分类

1. 前向迭代器:只能够用累加操作符向前迭代，如class forward\_list。其他容器如unordered\_set、uordered\_multiset、unordered\_map和unordered\_multimap也都至少为此类别(但标准库为它们提供了双向迭代器)
2. 双向迭代器:可以进行递增、递减操作。List、set、multiset、map和multimap都属于这种类型。
3. 随机访问迭代器:不但具有双向迭代器的所有属性，还具有随机访问能力，也就是说其提供了迭代器算术运算的必要操作符。可以进行增加、减少任意偏移量，计算两个迭代器的距离，使用< >号进行比较操作。Vector、deque、array和string提供的迭代器都属于此类。

除此之外，STL还提供了两个类别

1. 输入型迭代器:向前迭代时能够读取/处理value。Input stream属于此类
2. 输出型迭代器:向前迭代时能够涂写value。Inserter和outstream都属于此类

注意，为了写出尽可能与容器类型无关的泛型代码，最好不要使用随机迭代器访问操作。你可以使用所有迭代器都具备的操作符 **\* 、++、 ==、 !=、 =** 等，但是比如< >这类只有特定类型的迭代器才支持。

**5.4算法**

为了处理容器内的元素，STL提出了一些标准算法，包括查找、排序、拷贝、重新排序、修改、数值运算等基本而普遍的算法。

为了调用算法，必须包含头文件<algorithm> (某些需要特定的头文件)。

以下展示基本的算法:

min\_element()/max\_element():返回区间内最小/最大元素对应位置的迭代器。

Sort():将一段区间内的所有元素加以排序，可以选择性地传入一个排序基准，默认为<(递增)

Find():在一段区间内查找某值，返回具有某值地第一个元素对应的迭代器

Reserve():反转区间内的元素顺序

**5.4.1区间**

所有算法都是用来处理一或多区间内的元素。这样的区间可以(但非必须)涵盖容器内的所有元素。为了操作容器元素的某个子集，我们必须将区间首尾当作两个实参传给算法，而不是一口气把容器传进去。

所有算法处理的都是半开区间---包括起始元素位置但不包括末尾元素位置，用数学表示是

[beigin,end)

半开区间的优点是单纯，可以免除对空集做特殊处理。

**5.4.2处理多重区间**

有个算法可以同时处理多重区间。通常必须设定第一个区间的起点和终点。至于其他区间，只需要设定起点即可。终点通常可以由第一区间的元素推导出来。比如下面片段种，equal()从头开始逐一比较coll1和coll2内的所有元素如下：

If(equal(coll1.begin(),coll1.end(),

Coll2.beign())) {

…

}

于是，coll2中参与比较的元素数量，间接取决于coll1内的元素数量。故可以总结:如果某个算法被用来处理多重区间，那么调用它时，必须确保第二(以及其他)区间内所拥有的元素不少于第一个区间内的元素。

**5.5迭代器之适配器**

迭代器是一个纯抽象概念。任何东西，只要行为类似迭代器，它就是一个迭代器。因此可以撰写一些class具备迭代器接口。但有着不同的行为。C++标准库亦提供了数个预定义的特殊迭代器，被称作迭代器适配器。

**5.5.1Insert Iterator**

这种适配器可以使算法以“安插“而非覆写的方式运作，用以解决”目标空间不足“的问题。

Insert iterator内部将接口做了新的定义:

1、单步前进不会引起任何动静。

2、如果对某个元素赋值，会引发对其所属集合的插入动作。插入位置是在最前、最后还是特定位置上，需要视三种不同的insert iterator而定。

Back inserter:插入到最末

Front inserter:插入到最前

General inserter:插入到所指位置前方。

**5.5.2Stream Iterator**

其被用于读写stream。它们提供了必要的抽象特性，使得来自键盘的输入象是个集合，可以从中读取内容。同样道理，也可以把一个算法的输出结果重新导向到某个文件或者屏幕上。

**5.5.3Reverse iterator**

Reverse iterator会造成算法逆向操作，其内部将对递增转换为递减，反之亦然。所有提供双向访问迭代器的容器都可以通过成员函数rbegin()和rend()产生一个泛型迭代器。

注意:rbegin()对应于容器的末尾元素，而rend()同是逾尾位置，只不过是在容器内第一个元素的前一个位置.

同样的，只读方式形式为crbegin()和crend();

**5.6更易型算法**

表示会“移除或重排或修改“元素的算法。

5.6.1移除元素(remove)

Remove()并没有改变集合内的元素数量，变化的只是元素的次序:当某个元素被remove，其后序元素依次向前覆盖。

即数值为3的元素被其后的元素覆盖了，但集合种的元素数量没变。事实上，这个算法返回了一个新的终点。可以利用该终点获得新区间、缩减后的容器大小，或者是被删除的容器个数。

List<int> ::iterator end = remove(coll.begin(),coll.end(),3);

另一种做法是，借由测定集合之逻辑终点和实际终点的距离，获得“被删除元素的数量“

distance(end,coll.end());

如果需要单一语句来删除元素，可以这么做

Coll.erase(remove(coll.begin(),coll.end(),3),coll.end());

为什么算法不自己调用erase呢？这个问题正好点出STL为了获取弹性而付出的代价。一般来说，迭代器对自己所属容器一无所知，任何以“迭代器访问容器“的算法，都不得通过迭代器调用容器雷所属的任何成员函数。

**5.6.2更易关联式和无序容器**

更易型算法(指那些会移除、重排、修改元素的算法)若用于关联式或者无序容器，会出问题。关联式和无序容器不能被当作操作目标。关联式和无序容器的迭代器均被声明为指向常量的value或key。如果更动关联式容器或无序容器的元素，会导致编译错误。

那么，如何从关联式和无序容器种删除元素呢？当然是调用它们自己的成员函数。每一种关联式和无序容器都提供用以移除元素的成员函数，例如可以调用erase().

注意，容器类提供了多个不同的erase()成员函数。

**5.6.3算法vs成员函数**

就算我们符合种种条件，得以使用某个算法，那也未必旧一定是好。容器本身可能提供功能相似而效能更佳的成员函数。

如果高效是你的目标，那么应该总是优先使用成员函数。

**5.7以函数作为算法的实参**

有些算法可以接受用户自定义的辅助性函数，借以提高弹性和能力。这些函数将被算法内调用。

**5.7.1以函数作为算法实参的使用范例**

最简单的例子莫过于for\_each(),针对区间内的每一个元素，调用一个由用户指定的函数。

For\_each(coll.cbegin(),coll.cend(),print);

**5.7.2判断式**

判断式是一种特殊的辅助函数，所谓predicate，它会返回布尔值，常被用来指定作为排序准则或者查找准则。

**Unary Predicate(单参判断式)**

单参判断式会检查唯一实例的某项属性。比如下面程序:

Bool isPrime(int number)

{

…

}

Auto pos = find\_if(coll.begin(),coll.end(),isPrime);

在这个例子种，算法find\_if()在给定区间内寻找使”传入之单参判断式”运算结果为true的第一个元素。

Binary Predicate(双参判断式)

双参判断式的典型用途是，比较(实例)的两个实参的特定属性

比如下面,根据每个人的姓名，对一组元素进行排序:

Bool personSortCriterion(const Person &p1,const Person &p2){

…

}

Sort(coll.begin(),coll.end(),personSortCriterion);

注意，也可以使用函数对象来实现一个排序准则。这种做法的优点是，制作出来的准则将会是个类型，可以用来声明一个“某种类型为排序准则的set“之类的事。

**5.8使用Lambda**

Lambda始于C++11，是一种“在表达式或语句内指明函数行为“的定义式。这导致可以定义对象来描述函数行为，并将这些对象以inline实参形式传给算法作为predicate，或者是其他用途。

比如下面表达式

[](double d){return d\*d\*d}

就定义了一个·lambda，表述一个函数对象，返回某个double的三次方值。

**使用lambda的好处**

运用lambda具体指定“STL框架内部将采取的行为“，可以解决先前存在的许多缺点。

**Lambda的局限**

Lambda并非在每一方面都保持优势，但它无法拥有“跨越多次调用“都能被保存下来的内部状态。如果需要这个状态，必须在外围作用域种声明一个对象或变量，将他们以传引用的方式传入lambda。与此相比，函数对象允许你封装内部状态。

**5.9函数对象(Function Object)**

传递给算法的“函数形实参“不一定得是函数，可以是行为类似函数的对象。这种对象称为函数对象，或者仿函数

**5.9.1定义一个函数对象**

定义方法即是在类的内部重载()操作符，即可用对象+()表示一个函数调用。

函数对象是一种带状态的函数，行为像“function“的对象被称为”smart function“，因为它们的能力超越了operator(),因为它们的能力超过了operator().

每个函数对象有其自己的类型，寻常函数，唯有签名式不同时，才算类型不同。而函数对象即使签名式相同，也可以有不同类型。

函数对象通常比寻常函数速度快。对template而言，由于更多细节在编译器就已经确定，所以通常可能进行更好的优化。

**5.9.2预定义的函数对象**

C++标准库提供了若干预定义的函数对象，涵盖了许多基础运算，有了它们，很多时候就不必自己费心去写函数对象了。比如下例

Operator<之默认排序准则调用的是less<>，所以如果声明

Set<int> coll;

会被扩展为set<int,less<int>> coll;

既然如此，反向排序声明为

Set<int,greater<int>> coll;

**5.9.3Binder**

可以使用特殊的function adpter，或者所谓的binder，将预定义的函数对象和其他数值结合在一起。

**5.9.4函数对象vs Lambda**

Lambda是一种隐式的预定义函数对象，lambda通常提供“用以定义使用于STL算法内的函数行为“的更只管做法。此外，lambda应该和函数对象一样快。

然而，Lambda也有若干缺点。

**5.10容器内的元素**

**5.10.1容器元素的必要条件**

STL容器元素必须满足以下三个基本需求:

1. 元素必须可复制或者搬移。
2. 元素必须可悲assignment操作符加以搬移或者复制，容器和算法以新元素覆盖旧元素时使用的是assignment操作符
3. 元素必须可以被一个析构函数销毁。当元素被移除，容器会销毁该元素的内部拷贝。因此，析构函数一定不是private，此外，一如C++的做法，析构函数一定不可以抛出异常，否则世事难料。

**5.10.2Value语义vs Reference语义**

通常，所有容器都会建立元素拷贝，返回的也是元素拷贝。这意味着你的容器内的元素与你放进去的东西相等但非同一。如果你修改容器的元素，实际上改变的是拷贝而不是原件。

Value被复制意味着STL容器所提供的value语义。它们容纳的是你安插对象的值，而非对象本身，然而现实中也许需要用到reference语义，让容器容纳元素的引用。

STL只支持value语义，不支持reference语义，这当然是利弊参半。好处是：

复制元素很简单

使用引用容易导致错误，必须确保引用的对象健在，并效信对付偶尔出现的环式引用。

缺点是：

复制元素可能会导致不良效率，有时候甚至无法复制。

无法在数个不同的容器种管理同一份对象。

为了利用value实现reference语义，可以使用smart pointer。此外，还可以使用std::reference\_wrapper<>这样的外覆器让STL持有reference.

1. **STL容器**

**6.1容器的共通能力和共通操作**

所有的STL容器必须满足的三个最核心能力是:

1. 所有容器提供value语义而非reference语义。故容器进行元素安插动作时，内部实施的时copy和/或move动作。
2. 元素再容器内有特定排序。每一种容器都会提供若干“返回迭代器”的操作函数。
3. 一般而言，各项操作并非绝对安全，也就是它们不会检查每一个可能发生的错误

容器的共通操作包括:

1. 初始化，每个容器提供了一个default构造函数，一个copy构造函数和一个析构函数。
2. 赋值(assignment)和swap()
3. 与大小相关的操作
4. 比较。注意，无序容器只定义了==和!=。除了无序容器外，常用的比较操作符==、!=、<、<=、>、>=都依据以下三个规则被定义出来
5. 比较动作的两端必须属于同一类型的容器
6. 如果两个容器的所有元素依序相等，那么这两个容器相等。操作符==用来检查元素是否相等
7. 采用字典式变焦，来判断某个容器是否小于另一个
8. 元素访问

**容器提供的类型**

|  |  |  |
| --- | --- | --- |
| 类型 | 必要 | 效果 |
| size\_type | Yes | 用于表现容器大小，无符号 |
| difference\_type | Yes | 带正负号的差距值类型 |
| value\_type | Yes | 元素类型 |
| reference | Yes | 元素的reference类型 |
| const\_reference | Yes | 常量元素的reference类型 |
| iterator | Yes | Iterator的类型 |
| const\_iterator | Yes | 指向只读元素的迭代器类型 |
| pointer | No | 指向元素的指针类型 |
| const\_pointer | No | 指向只读元素的指针类型 |

**6.2Array**

**6.2.1Array的能力**

Array式一种有序集合，允许随机访问，可以再常量时间内访问任意元素。其具备以下能力

1. 初始化(array<>)是唯一一个，无任何东西被指定为初值时，会被预初始化的容器(可能不明确，而非0)。
2. Swap()和move语义.iterator和reference不会随着元素的置换而改变所指容器，影响只会让其指向同一容器内的不同元素。
3. 大小size

**6.2.2Array的操作**

(1)建立/销毁/复制（利用default、copy或者move来初始化均可）

(2)非更易操作

(3)赋值(按元素依次赋值)

(4)访问。可以利用索引、或者at()函数、以及range-based for循环

(5)迭代器相关函数(begin()、end()、cbegin()、cend()、rbegin()、rend()、crbegin()、crend())

**6.2.3把Array当成C-style Arra**y

就像对待class vector<>一样，C++标准库保证array<>的所有元素一定位于连续且相邻的内存内。于是，无论何处，只要可以使用寻常的C-style array，就可以使用Array<>。

**6.2.4tuple接口**

Array提供tuple接口，因此可以用表达式tuple\_size<array name>::value取得元素个数。用tuple\_element<array name>::type取得元素类型。用get()取得某特定元素。

**6.3Vector**

Vector塑模出一个动态数组。它本身是“将元素置于动态数组中加以管理”的一个抽象概念。

**6.3.1Vector的能力**

Vector将元素复制到内部的动态数组中。元素之间总是存在一定的顺序，所以vector是一种有序集合。Vector支持随机访问，提供随机访问迭代器，所以适用于任何STL算法。

如果再末端附加或者删除元素，vector的效率相当好，单如果在前端或者中段安插/删除元素，效率就不怎么样了。因为作用点之后每个元素都要移动到另一位置，而每一次移动都需要调用assignment操作符。

vector提供的操作大小的函数有size()、empty()和max\_size().另一个与大小有关的capacity()返回vector实际可容纳的元素量。

Vector的容量之所以重要，有两个原因:

1. 一旦内存重新分配，vector元素相关的所有迭代器、引用、指针都会失效
2. 内存重新分配很耗时

所以，如果程序管理了与”vector元素相关”的引用、迭代器、指针，而程序执行速度对我们至关重要的话，就必须考虑容量问题！

**Tips1:**可以实用reserve()保留适当容量，避免重新分配内存

**Tips2:**避免重新分配内存的另一个方法是，初始化期间向构造函数，构建足够的空间。如果实参是个数值，它将成为vector的起始大小。

Vector的容量概念和string类似，但有一个大不同:**vector不能使用reserve()缩减容量**。

C++11引入了一个vector新函数shrink\_to\_fit():一个不具有强制力的要求，可以缩减容量以符合当前的元素个数。在C++11前，有一个间接缩减vector的小窍门，两个vector内容交换后，容量也会交换。

**Tips3:**swap()之后原先所有的引用、指针、迭代器都换了指向对象，它们仍然指向原本的位置。换句话说，这些引用、指针、迭代器均失效了。Shrink\_to\_fit()同样如此。

**6.3.2Vector的操作**

(1)构建、复制和销毁

(2)非更易操作

(3)赋值

(4)元素访问。欲访问vector的所有元素，必须使用range-based for循环、特定的操作函数或者迭代器。和array一样，vector能通过索引返回元素。

(5)迭代器相关函数

(6)安插和移除元素。依据STL惯例，必须保证传入的实参合法，即迭代器必须指向一个合法位置，且区间的起始位置不能在结束位之后。一如既往，当pop\_back()被调用时，确保容器非空是程序员的责任。

关于效能，以下情况可以预期安插动作和移除动作会比较快些。

1. 在容器尾部安插或者删除元素
2. 容量一开始就够大
3. 安插多个元素时，调用一次比调用多次来的快。

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.push\_back(element) | 附加一个elem的拷贝到尾部 |
| c.pop\_back() | 移除最后一个元素，但不返回它 |
| c.insert(pos,elem) | 在pos前方插入一个elem拷贝，返回新元素位置 |
| c.insert(pos,n,elem) | 在pos前方插入n个elem拷贝，返回第一个新元素位置 |
| c.insert(pos,begin,end) | 在pos前方插入区间内的所有元素的拷贝，返回同上 |
| c.insert(pos,initlist) | 在pos前方插入初值列元素的拷贝，返回同上 |
| c.emplace(pos,args…) | 在pos前方插入一个以args为初值的元素，并返回其位置 |
| c.emplace\_back(args…) | 附加一个以args为初值的元素于末尾，不返回任何东西 |
| c.erase(pos) | 移除pos上的元素，返回下一个元素的位置 |
| c.erase(begin,end) | 移除区间内的元素，返回下一元素的位置 |
| c.resize(num) | 将元素量变为num(如果size变大,多的元素都采用默认构造) |
| c.resize(num,elem) | 将元素量变为num(如果size变大,多的元素都为elem拷贝) |
| c.clear() | 移除所有元素，清空容器 |

**6.3.3 Class vector<bool>**

C++标准库针对元素类型为bool的vector<>专门设计了一个特化版本，目的是获取一个优化的vector使其消耗空间远小于一般vector而实现出来。一般实现版本会为每个bool元素分配至少1byte空间。而vector<bool>的特化版本内部只用1bit存放元素，空间节省8倍，不过这里有个小麻烦:C++最小可定址值乃是以byte为单位，所以上述vector特化版本必须对reference和iterator做特殊处理。

因此，vector<bool>无法满足其他所有vector的规定。例如vector<bool>::iterator并不是一个真正的随机访问迭代器。所以某些template代码有可能适用于任何类型的vector，唯独不适用于vector<bool>.

Vector<bool>提供了一些特殊的bit操作。可以利用之方便操作bit和flag。

由于vector<bool>大小可动态改变，还可以将之视为可动态改变大小的bit-filed。如果需要固定大小的bitfiled，应该使用bitset而非vector<bool>。

**6.4Deque**

容器deque和vector非常相似，其也采用动态数组来管理元素，提供随机访问，并有着和vector几乎一模一样的接口。不同的是deque的头尾均开放，是双向的，因此能在头尾两端进行快速安插和删除。

由于其性质同vector类似，这里不多做赘述。

**6.5List**

List(指的是容器类List<>的一个实例)使用双向串列(双向链表)管理元素。

**6.5.1List的能力**

(1)List不支持随机访，如果要访问某个元素，需要顺着链表逐一递进。

(2)任何位置上执行元素的安插和移除操作都非常块，始终是常量时间。因为实际上内部只做了一些指针操作

(3)安插和删除动作不会使其他元素的各个指针、引用、迭代器等失效

(4)List对于异常的处理方式是，要么操作成功，要么什么都不发生。

(5)List提供front()、push\_front()、pop\_front()、back()、push\_back()、pop\_back()等操作函数。

(6)List不提供下标操作符，也不提供at()

(7)List不提供容量和空间重新分配等操作，也没有必要

(8)List提供不少特殊成员函数，专门用于移动和删除元素。

**6.5.2List元素的安插和移除(只列举部分，其余和其他vector类似)**

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.erase(pos) | 移除迭代器位置pos上的元素，返回下一元素位置 |
| c.erase(beg,end) | 移除区间内元素，返回下一元素位置 |
| c.remove(val) | 移除所有值为val的元素 |
| c.remove\_if(op) | 移除所有造成表达式op为true的元素 |

**6.6Foward List**

Forward List始于C++11，内部以一个单链表管理元素。

**6.6.1foward list的能力**

观念上，forward list是一个行为受限的list，不能走回头路。凡是list没有提供的功能，它也不提供。它的优点是占用内存少，行动也略快些。

相比较list，forward list有以下约束

1. forward list只提供前向迭代器，而非双向迭代器(没有rbegin之类…)
2. Forward list不提供成员函数size()
3. Forward list没有指向最末元素的锚点。基于此，fowawrd list不提供用以处理最末元素的成员函数如back()、push\_back()和pop\_back()
4. 对于所有“令元素安插或被删除于forward list某特定位置上”的成员函数，forward list提供特殊版本。原因是必须传递第一个被处理的元素的前一位置。这些函数名都带有一个后缀after，例如insert\_after()代替insert()，将新元素安插于第一实参所表示的元素之后。也就是受，它在那个位置上附加(append)一个元素.
5. 基于同样原因，forward list提供before\_begin()和cbefore\_begin(),它们会产生出一个元素之前的一个虚拟元素的位置，可被用来让名称带有后缀afer的算法交换第一元素。

除了这些差异，forward list的行为就像一个list。

**6.7Set和Multiset**

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。

只要是可依据某排序准则被比较的任意类型T都可以成为set或者multiset的元素类型。可有可无第二个template实参来定义排序准则，默认为less(<，即为升序)。可有可无第三个实参来定义内存模型，默认的内存模型是allocator，由C++标准库提供。

**6.7.1set和multiset的能力。**

和所有标准的关联式容器类似，set和multiset通常以平衡二叉树完成----虽然C++标准没有明确指定，但由set和multiset各项操作的复杂度可以得出这个结论。

事实上，set和multiset通常以红黑树实现。它保证安插时最多只会做两个重新链接动作，而且达到某一元素的最长路径的深度，最多只是最短路径深度的两倍。

**6.7.2Set和Multiset的操作函数**

有两种方式可以定义排序准则

1. 以template参数定义之，例如:

Std::set<int.std::greater<int>> coll;

1. 以构造函数参数定义之。采用default构造函数时，传入排序准则作为实参即可。

特殊的查找函数

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.count(val) | 返回元素值为val的元素个数 |
| c.find(val) | 返回元素值为val的第一个元素 |
| c.lower\_bound(val) | 返回key为val第一个可安插位置(即key>=val的第一个位置) |
| c.upper\_bound(val) | 返回key为val最后一个可安插位置(即key>val的第一个位置) |
| c.eqaul\_range() | 返回val可被安插的第一个和最后一个位置。也就是“元素值==val”的区间。 |

**6.8Map和Multiamp**

Map和multimap将键值对当作元素进行管理。它们可以根据key的排序准则自动为元素排序。Multimap允许元素重复而map不允许。

**6.8.1Map和multimap的能力**

和其他所有关联式容器一样，Map/Multimap通常以平衡二叉树完成。通常set、map、multimap、multiset使用相同的内部结构。因此，可以把set和multiset视为特殊的map和multimap。只不过set元素的key、value是同一对象。因此，map和multimap拥有set和multiset的所有能力和所有操作。当然，差异还是有的，首先它们的元素是键值对，其次，map可以作为关联式数组来运用。

Map和multimap会根据元素的key进行自动排序。这么依赖，根据已知key来查找某个元素时就有很好的效率，而根据已知的value查找元素时，效率就会变得很糟糕。自动排序使得map和multimap身上有了一条重要限制:不能直接改变元素的key，因为这样会破坏正确次序。要改变key，必须先移除拥有该key的元素，然后插入新的键值对。从迭代器角度看，元素的key是常量。至于元素的value倒是可以直接修改，当然前提是value非常量。

**6.8.2将Map视为关联式数组**

通常，关联式容器不提供元素的直接访问，必须借助迭代器。不过map和无序容器是例外去。Non-const map提供了下标操作符，支持元素的直接访问。这里的下标即为元素的key。

C++11提供了另一个成员函数at，可以用于const和non-const map。

|  |  |
| --- | --- |
| 操作 | 效果 |
| c[key] | 安插一个带着key的元素---如果尚未存在于容器内。返回一个指向带着key的元素的引用.(only for nonconstant maps) |
| c.at[key] | 返回一个reference指向带着key的元素 |

at()会依据它收到的元素的key来取得元素的value，如果不存在这样的元素则抛出异常out\_of\_range。

**Tips1:**map可以通过全局的find\_if()来查找特定value的元素。

**Tips2:**Multimap可以用来当作字典使用。

**6.9无序容器**

Hash table(散列表)，作为一个重要的用于集合的数据结构，自TR1起，带有hash table特性的容器终于进入C++标准。

以hash table为基础的容器，其内的元素没有清晰明确的次序。严格地说，C++标准库称呼unordered 容器为unordered associative容器。当然一般称为unordered容器。如果说associative容器则指代旧式地associative容器(即C++98引入的set、multiset、map、multimap)。

Unordered set和multiset都是存放特定类型的个别value。而unordered map和multimap则存放的元素都是键值对。

Unordered set和unordered map都不允许key重复。但是unordered multiset和unordered multimap都允许key重复。

**6.9.1unordered 容器的能力**

Unordered容器有若干指明的性质，基于以下假设:

1. 这些hash table采用链表法解决冲突，于是一个hash code和一个linked list关联起来。
2. 上述的linked list是单链还是双链取决于实现。C++ 标准只保证其至少是forward list。
3. 关于重新散列，有各种实现策略

--传统做法是，在单一insert和erase动作出现时，有时会发生一次内部数据重组织

--所谓递进式做法是，渐进改变bucket或者slot的数量，这对即时环境特别有用，因为在其中突然放大hash table的代价太高。

内部使用hash table，其主要优点是，它惊人的运行期行为。假设拥有良好的hashing策略，并有着良好的实现，可以保证在安插、删除、查找元素时获得摊提(amortized)常量时间。

但是，unordered容器比起寻常的associative容器，也有一些缺点

1. 无序容器不提供<、>、<=和>=用以安排布置这些容器的多重实例。然而提供了==和!=
2. 不提供lower\_bound()和upper\_bound()
3. 由于iterator至少是个forward iterator，因此反向iterator包括rbegin()、rend()等都不提供

由于元素的key具体关系到元素的位置，所以不可以直接改动元素的key。因此，很像associative容器那样，要改动，就要先删除。

**Tips1:**无序容器不提供直接访问元素操作。

**Tips2:**通过迭代器进行的访问有其束缚:从迭代器的角度看，元素的value是常量。

身为程序员，可以指定若干会影响hash table行为的参数

1. 你可以指定bucket的最小值
2. 你可以(有时必须)提供自己的hash function
3. 你可以(有时候必须)提供自己的等价准则
4. 你可以指定一个最大负载系数，一旦超过就会自动rehashing
5. 你可以强迫rehashing

但你不能影响以下行为:

1. 成长系数，那是自动rehashing时用来成长或者缩小list of bucket的系数
2. 最小负载系数，用来强制执行rehashing(当容器中的元素个数缩减)

**Tips3:**rehashing只能发生在以下调用之后:insert()、rehash()、reserve()、clear()

**Tips4:**erase()绝对不会造成指向元素的指针、引用、迭代器失效。因此，即使删除了几百个元素，那些bucket的大小并不会改变。但是如果在之后安插一个元素，bucket的大小可能会改变。

**Tips5:**在那些支持等价key的容器内(比如unordered multiset和unordered multimap)，带有等价key的元素将会相邻排列。Rehashing以及其他可能于内部改变元素次序的操作，都会维持带有等价key的元素的相对次序。

**6.9.2布局操作**

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.hash\_function() | 返回hash函数 |
| c.key\_eq() | 返回相等性判断式 |
| c.bucket\_count() | 返回当前bucket数目 |
| c.max\_bucket\_count() | 返回bucket可能最大数目 |
| c.load\_factor(0 | 返回当前负载系数 |
| c.max\_load\_factor() | 返回当前最大负载系数 |
| c.max\_load\_factor(val) | 设定最大负载系数 |
| c.rehash(bnum) | 将容器rehash，使其bucket个数最少为bnum |
| c.reserve(num) | 将容器rehahs，使其空间至少可有num个元素 |

**6.9.3特殊查找操作**

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.count(val) | 返回元素值为val的元素个数 |
| c.find(val) | 返回元素值为val的第一个元素。如果找不到就返回val |
| c.equal\_range(val) | 返回val可被安插的第一个位置和最后一个位置，也就是元素值==val的区间 |

**6.9.4安插和移除操作**

除了使用初值列之外，还有三种方式可以将一个value放进一个无序容器中。

1. 使用value\_type
2. 使用pair<>
3. 使用make\_pair()

**6.9.5bucket接口**

我们有可能通过一个特定的bucket接口访问个别bucket，用以暴露整个hash table的内部状况。如下表所示

|  |  |
| --- | --- |
| 操作 | 效果 |
| c.bucket\_count() | 返回当前的bucket个数 |
| c.bucket(val) | 返回val将被找到的那个bucket编号 |
| c.bucket\_size(bucketidx) | 返回第bucketidx个bucket所包含的元素个数 |
| c.begin(bucketidx) | 返回前向迭代器，指向第bucketidx个bucket的头部 |
| c.end(bucketidx) | 返回前向迭代器，返回第bucketidx个bucket的尾部 |
| c.cbegin(bucketidx) | 返回常量类型的前向迭代器，指向与begin一致 |
| c.cend(bucketidx) | 返回常量类型的前向迭代器，指向与end一致 |

**6.9.6使用unordered map作为associative array**

就像map一样，unordered map也提供一个下标操作符来直接访问元素。还提供一个相应的成员函数at().

|  |  |
| --- | --- |
| 操作 | 效果 |
| c[key] | 安插一个带着key的元素---如果尚未存在于容器内。返回一个指向带着key的元素的引用.(only for nonconstant maps) |
| c.at[key] | 返回一个reference指向带着key的元素 |

**6.10各种容器的使用时机**

1、默认情况下应当使用vector，因为vector内部结构最为简单

2、如果经常需要在序列头尾安插、移除元素，应该采用deque。

3、如果需要经常在容器中段执行安插、移除等操作，可以考虑用list

4、如果需要容器对于异常的处理使得“每一次若操作不成功便无任何作用”，也应选用list

5、如果需要经常精准地查找某个元素，应当使用“依据该准则进行hash“的unordered set或者multiset。然而，hash内部是无序的，如果必须依赖元素的次序，应当使用set或者multiset，它们根据查找准则对元素排序。

6、如果想要处理键值对，请用unordered map或unordered multimap，如果元素次序很重要则采用map

7、如果需要关联式数组，应当使用unordered map。如果元素次序很重要则用map

8、如果需要字典结构，应当使用unorederd multimap，如果元素次序很重要则使用multimap。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Array | Vector | Deque | List | Forward List | 关联式数组 | 无序容器 |
| 来源 | TR1 | C++98 | C++98 | C++98 | C++11 | C++98 | TR1 |
| 典型结构 | Static array | 动态数组 | Array of arrays | 双向链表 | 单链表 | 平衡二叉树 | Hash table |
| 元素形式 | Value | Value | Value | Value | Value | Value或键值对 | Value或键值对 |
| 元素可重复 | Yes | Yes | Yes | Yes | Yes | Multi类型可以 | Multil类型可以 |
| 迭代器类型 | 随机访问 | 随机访问 | 随机访问 | 双向 | 单向(向前) | 双向,key为常量 | 单向,key常量 |
| 扩/缩 | 不可 | 一端可 | 两端可 | 各处 | 各处 | 各处 | 各处 |
| 随机访问 | Yes | Yes | Yes | 否 | 否 | 否 | 几乎是 |
| 元素查找 | 慢 | 慢 | 慢 | 很慢 | 很慢 | 快 | 很快 |
| 安插/移除致iterator失效 | 无 | 重新分配时 | 总是如此 | 绝不 | 绝不 | 绝不 | Rehashing会 |
| 安插/移除致reference、pointer失效 | 无 | 重新分配时 | 总是如此 | 绝不 | 绝不 | 绝不 | 局部 |
| 允许保留内存 | 无 | Yes | 否 | 无 | 无 | 无 | 是(bucket) |
| 允许释放被移除元素内存 | 无 | Shirink\_to\_fit | 有时 | 总是 | 总是 | 总是 | 有时 |
| 事物安全(要么成功要么无效) | 否 | 尾端push/pop | 头尾两端push/pop | 安插/移除 | 安插/移除 | 单元素安插/移除 | 同左侧一样 |

1. **细探STL容器成员**

**7.1创建、复制和销毁**

容器提供下列构造、析构函数，大多数构造函数允许你传入一个分配器(allocator)作为一个附加实参。

**Container::container()**

·Default构造函数

·建立一个新得空容器

**Explicit container::container(const CompFunc& cmpPred)**

·以cmpPred为准则建立一个空容器

·排序准则必须定义一个strict weak ordering

·提供者:set、multiset、map、multimap

**Container::container(initializer-list)**

·建立一个新容器，以初值列内的元素为初值

·对于array，这个操作是隐式的

**Container::container(initializer-list,const CompFunc& cmpPred)**

·建立一个容器,排序准则为cmpPred，以初值列内的元素为初值

·排序准则必须定义一个strict weak ordering

·提供者:set、multiset、map、multimap

**Container::container(const container &c)**

·拷贝构造函数

·建立既有容器c的一个新拷贝

·针对c中的每一个元素调用拷贝构造函数

·对于array，这个操作是隐式定义的

**Container::container(container &&c)**

·Move构造函数

·建立一个新容器，以既有容器c的元素为初值

·经过此调用后，c仍然有效，单其内却含“不确定”

·对于array，这个操作也是隐式定义的

**Explicit container::container(size\_type num)**

·建立一个容器，可含有num个元素

·元素由其default构造函数创建

·提供者:vector、deque、list和forward list

**Explicit container::container(size\_type num,const T&value)**

·建立一个容器，可含num个元素

·所有元素都是value的拷贝

·T是元素类型

·对于string,value并非传引用

·提供者:vector、deque、list、forward list和string

**Container::(InputIterator beg,InputIterator end)**

·建立一个容器，以[begin,end)内的所有元素为初值

·此函数是个成员模板

**Container::(InputIterator beg,InputIterator end，**

**Const CompFunc& cmpPred)**

·产生一个排序准则为cmpPred，并以区间[begin,end)内所有元素为初值的容器

·此函数是个成员模板

·排序准则必须定义一个strict weak ordering

·提供者:set、multiset、map、multimap

**Container::~container()**

·析构函数

·移除所有元素，并释放内存

·对每个元素均调用析构函数

**附录:strick weak ordering的意义**

1. **必须是非对称的**。对于operator<而言，如果x<y为true，那么y<x为false。对于判断式op而言，如果op(x,y)为true,op(y,x)则为false
2. **必须是可传递的**。如果x<y为true，y<z为true，那么x<z为true。对于判断式op而言，如果op(x,y)为true,op(y,z)为true,那么op(x,z)为true
3. **必须是自反的**。比如x<x永远为false。而对于判断式op，op(x,x)永远为false
4. **必须具有等效传递性**，大体意思是，如果a等于b且b等c，那么a等于c

C++11保证安插和抹除动作都会保存等效元素的相对次序。

**7.2 非更易操作**

**7.2.1大小相关操作**

**Bool container::empty() const**

·检查容器是否为空

·相当于begin()==end()，但是更快

·复杂度:常量时间

**Size\_type container::size() const**

·返回现有元素的数量

·欲检查容器是否为空应用empty(),因为其更快

·复杂度:常量时间

**Size\_type container::max\_size() const**

·返回容器可内含的最大元素个数

·这是一个与实现技术相关的值，可能取决于容器的内存模型。更准确而言，由于vector容器常常使用一个单一内存区段，所以vector的这个值往往小于其他容器

·复杂度:常量时间

**7.2.2元素比较**

Bool operator ==

Bool operator !=

·返回两个容器相等或不等

·两个容器如果具有相同数量的元素，且元素内容相同，便是相等。但是unordered容器还包括必须“想等的元素有着相同的次序”

·复杂度:一般是线性

Bool operator <

Bool operator <=

Bool operator >

Bool operator >=

·返回两个同类型容器的比较结果

·检验某容器是否小于另一容器，采用的是字典式比较

·复杂度:线性

·提供者:array、vector、deque、list、forward list、set、multiset、map、multimap和string

**7.2.3关联式和无序容器的特有非更易操作**

**Size\_type container::count(const T& value) const**

·返回与value相等的元素个数

·T是被排序值的类型。在set和multiset内，T是元素类型，而在map和multimap中，T是key的类型

·复杂度:线性

**Iterator container::find(const T& value)**

**Const\_iterator container::find(const T& value) const**

·返回第一个“值与value相等”的元素的位置

·如果找不到就返回end()

·T是被排序值的类型。在set和multiset内，T是元素类型，而在map和multimap中，T是key的类型

**Iterator container::lower\_bound (const T& value)**

**Const\_iterator container::lower\_bound(const T& value) const**

·返回一个iterator指向“根据排序准则，可安插value拷贝的第一个位置”

·如果找不到就返回end()

·返回的iterator指向”其值大于等于value”的第一个元素的位置

·T是被排序值的类型。在set和multiset内，T是元素类型，而在map和multimap中，T是key的类型

复杂度:对数

提供者:set、multiset、map和multimap

**Iterator container::upper\_bound (const T& value)**

**Const\_iterator container::upper\_bound(const T& value) const**

·返回一个iterator指向“根据排序准则，可安插value拷贝的最后一个位置”

·如果找不到就返回end()

·返回的iterator指向”其值大于value”的第一个元素的位置

·T是被排序值的类型。在set和multiset内，T是元素类型，而在map和multimap中，T是key的类型

复杂度:对数

提供者:set、multiset、map和multimap

**Pair<iterator,iterator>container::equal\_range (const T& value)**

**Pair<const iterator, const iterator>container::equal\_range(const T& value) const**

·返回一对iterator，指向“根据排序准则，可安插value拷贝的第一个和最后一个位置”

·返回的区间内的元素，其值皆等于value

·相当于make\_pair(lower\_bound(value),upper\_bound(value))

·返回的区间是半开的，即[begin,end)内的元素都等于value

·复杂度:对数

**7.3 赋值**

**Container& container::operator = (const container& c)**

·这是一个拷贝赋值操作符

·将c的所有元素赋值给现有容器，亦即以c的所有元素替换现有的所有元素

·这个操作符会针对覆写的元素调用其赋值操作符，针对被附加的元素调用其拷贝构造函数，针对被移除的元素调用其析构函数

**Container& container::operator = (const container&& c)**

·这是一个move assignment操作符

·将c的所有元素搬移给\*this，亦即以c的元素替换现有元素

·调用之后c仍有效，但是内容不明确

·始于C++11

**Container& container::operator = (initializer-list)**

·将初值列的所有元素赋值给现有容器，亦即以传入的元素的拷贝替换现有元素

·这个操作符会针对被覆写的元素调用其assignment操作符，针对被附加的元素调用其拷贝构造函数，针对被移除的元素调用其析构函数

·始自C++11

**Void container::assign(initializer-list)**

·将initializer-list的所有元素赋值给现有容器，亦即以传入的元素的拷贝替换现有元素

·始自C++11

**Void array::fill(const T& value)**

·将value赋值给所有元素，亦即以value的拷贝替换现有元素

·始自C++11

**Void container::assign(size\_type num,const T& value)**

·赋值num个value，亦即以num个value的拷贝替换现有元素

·T必须是元素类型

**Void container::assign(InputIterator begin，InputIterator end)**

·赋值[beg,end)区间内的所有元素，即以该区间内的所有元素替换掉现有元素

·此函数为一个成员模板、

**Void container::swap(container& c)**

**Void swap(container& c1,container& c2)**

·和c交换内容；或是c1和c2交换内容

·上述两个操作都互换容器的元素、排序准则、相等性准则、hash函数对象(如果有的话)。所有指向元素的指针、引用和迭代器，都会交换其容器，因此它们此后仍然指向相同(被交换后的)元素

·array内部不能够只是交换指针，因此swap带有线性复杂度。而完成后iterator指和reference指向原本的容器，但是不同的元素。

·对于关联式容器，只要比较准则进行复制或者赋值时不抛出异常，本函数就不会抛出异常。对于无序容器，只有当相等判断式或者hash函数对象抛出异常，本函数才会抛出异常。对于其他所有容器，一律不抛出异常

·复杂度:一般常量，array线性

·基于其复杂度，当你不需要被赋值的对象，宁可使用swap而不使用拷贝赋值操作。

**7.4 元素直接访问**

**Refernce container::at(size\_type idx)**

**Const refernce container::at(size\_type idx) const**

·返回索引idx代表的元素(第一个元素索引为0)

·如果传入无效索引，会抛出out\_of\_range异常

·后序的改动或内存重新分配，可能会导致这里的引用失效

·如果调用者保证idx合法有效，那么最好使用速度更快的[]

·提供者:array、vector、deque和string

**T& map::operator at(const key\_type& key)**

**Const T& map::operator at(const key\_type& key) const**

·返回map之中key所对应的value

·如果没有任何元素的key等于key，会抛出out\_of\_range异常

·始自C++11

**Reference container::operator[ ](size\_type idx)**

**Const reference container::operator[ ](size\_type idx) const**

·两者都返回索引idx所代表的元素

·如果传入一个无效索引会导致不确定索引，故调用者必须保证索引有效，否则应该使用at()

·后序的改动或者重新分配内存会导致这里返回的reference失效

**T& map::operator[ ](const key\_type& key)**

**T& map::operaotr[ ](key\_type &&key)**

·关联式数组的operator[ ]

·在map中，会返回key对应的value

·如果不存在以key为键的元素，本操作会自动创建一个新元素，其键由传入的key而来，其值由value则由value类型之default构造函数给予。不存在所谓的无效索引

·对于第二形式，key的状态此后变得不确定(move语义)

·上述第二形式始于C++11

·提供者:map和unordered map

**Reference container::front()**

**Const\_reference container::front() const**

·返回第一个元素

·调用者必须确保容器内拥有元素，否则会导致不确定行为

·对string的支持源自C++11

·提供者:array、vector、deque、list、forward list和string

**Reference container::back()**

**Const\_reference container::back() const**

·返回最后一个元素(索引为size()-1)

·调用者必须确保容器内拥有元素，否则会导致不确定行为

·对string的支持源自C++11

·提供者:array、vector、deque、list和string

**T\* container::data()**

**Const T \* container::date() const**

·返回一个带着所有元素的C风格的数组

·本函数用来将一个标准库的array的所有元素传递给一个C-style array

·String只有第二形式

·对array和vector的支持始于C++11

**7.5 产生迭代器的操作**

|  |  |
| --- | --- |
| 容器 | 迭代器种类 |
| Array | 随机访问 |
| Vector | 随机访问 |
| Deque | 随机访问 |
| List | 双向 |
| Forward List | 单向 |
| Set | 双向，元素为常量 |
| Multiset | 双向，元素为常量 |
| Map | 双向，key为常量 |
| Multimap | 双向，key为常量 |
| Unordered set | 前向，元素为常量 |
| Unordered multiset | 前向，元素为常量 |
| Unordered map | 双向，key为常量 |
| Unordered multimap | 双向，key为常量 |
| String | 随机访问 |

**Iterator container::begin()**

**Const\_iterator container::begin() const**

**Const\_iterator container:cbegin() const**

·返回一个iterator，指向容器的起始处

·如果容器为空，此操作相当于end()或者cend()

·unordered容器也提供带有参数实值得begin和cbegin，用以提供bucket接口

·cbegin()始于C++11

**Iterator container::end();**

**Const\_iterator container::end() const**

**Const\_iterator container:cend() const**

·返回一个iterator，指向容器尾端(最后一个元素得下一位置)

·如果容器为空，此操作相当于begin()或者cbegin()

·unordered容器也提供带有参数实值的end和cend，用以提供bucket接口

·cend()始于C++11

**Reverse\_iterator container::rbegin()**

**Const\_reverse\_iterator container::rbegin() const**

**Const\_reverse\_iterator container:crbegin() const**

·返回一个反向迭代器，指向反向迭代器的起点(即正向的最后一个元素)

·如果容器为空，此操作相当于rend()或者crend()

·crbegin()始于C++11

**Reverse\_iterator container::rend()**

**Const\_reverse\_iterator container::rend() const**

**Const\_reverse\_iterator container:crend() const**

·返回一个反向迭代器，指向反向迭代器的终点(即正向第一个元素的更前方)

·如果容器为空，此操作相当于rbegin()或者crbegin()

·crend()始于C++11

**7.6重设大小**

**Void container::resize(size\_type num)**

**Void container::resize(size\_type num,const T& value)**

·将容器大小改为num

·如果size()原本就是num，则两者皆无效

·如果num大于size()，则在容器末尾附加额外元素。第一形式通过default构造函数来构建新元素，第二形式则用value的拷贝来构建

·如果num小于size()，则移除尾端元素，获得新的大小。每个被移除元素的析构函数都会被调用

·对于vector和deque，这些函数都可能造成”指向其他元素”的迭代器和引用无效。对于所有其他容器，“指向其他元素”的迭代器和引用永远保持有效。

·对于vector和deque，这些函数要么成功，要么无效，前提是元素的构造函数和赋值操作符不抛出异常。对于list和forward list，如果失败不会带来任何影响

·在C++11之前，value是以传值的方式传递

·对于string，value是以传值的方式传递

·提供者:vector、deque、list、forward list和string

**7.7 List和Forward List的特殊成员函数**

**Void list::remove(const T& value)**

**Void list::remove\_if(UnaryPredicate op)**

·remove()移除所有带有value值得元素

·remove\_if()移除所有让单参数判断式op返回true的元素

·两个形式都会调用被移除元素的析构函数

·留下的元素保持原先次序(stable)

·这是remove()的特殊版本，因为其确实将元素删除了(比如vector要真正删除需要erase)

**Void list::unique()**

**Void list::unique(BinaryPredicate op)**

·移除(foward)list内相邻而重复的元素，使得每一个元素都不同于下一个

·第一形式会将所有“和前一元素相等的元素”移除

·第二形式的意义是，任何以一个元素elem，如果其前一元素是e，而elem和e造成的双参判断式为true，就移除elem。

·两个形式都会调用被移除元素的析构函数

·如果元素比较动作不抛出异常，这些函数亦不抛出异常

**Void list::splice(const\_iterator pos,list& source)**

**Void list::splice(const\_iterator pos,list&& source)**

·将source的所有元素搬移到\*this，并将它们安插到iterator pos所指向的位置

·调用之后，source将被清空

·如果source和\*this相同，会导致不明确行为

·调用者必须确定pos是\*this的一个有效位置，否则会导致不明确行为

·指向source的元素的所有指针、引用、迭代器仍然有效，此后他们呢属于this

·此函数不抛出异常

·第二形式出自C++11.在C++11前用的是iterator而不是const\_iterator

**Void list::sort()**

**Void list::sort(CompFunc cmpPred)**

·对list内的所有元素排序

·第一形式以operator<进行排序

·第二形式调用CompFunc比较两元素

·Value相同的元素，顺序保持不变

·提供者:list、forward list

·这是sort()赫stable\_sort()的算法特殊版本

**Void list::reverse()**

·将(foward)list内的元素次序颠倒

·这是reverse()算法的特殊版本

·此函数不抛出异常

·提供者：list和forward list

**7.7 容器的策略接口**

**7.7.1更易型策略函数**

**Void container::reserve(size\_type num)**

·内部保留若干内存，至少能容纳num个元素

·对于vector，这个调用只能增加容量。因此如果num小于或等于当前真实容量，其没有效用

·对于unordered容器，这个调用等价于rehash，且会让iterator失效，并改变元素的次序

·对于string，num可有可无(默认是0)，如果num小于实际容量，这个调用将是个不被绑定的缩减请求

·这个操作可能造成指向元素的迭代器(针对vector、string)、引用和指针失效。然而它保证，一旦被调用，安插过程中不会发生重分配，直到有个安插动作造成大小超越num。因此，reserve()可以增加速度并且帮助指针、引用、迭代器的有效性

·如果num>max\_size()就会抛出lengh\_error异常，如果内存分配失败就会抛出一个相应的异常

·自C++11起可以作用域无序容器

**Void container::shrink\_to\_fit()**

·缩减内部内存，使刚好符合实际的元素数量

·此调用时一个非必被绑定的请求，意思是实现可忽略这个调用，以便有回旋空间允许“实现专属之优化”。因此不保证此后的capacity()==size()为true

·这个操作可能造成指向其中元素的指针、引用、迭代器失效

·始于C++11。在此之前如果要缩减vector容量，一般是用swap交换一大一小两个容器。

·提供者:vector、deque和string

**Void container::rehash(size\_type bnum)**

·改变无序容器的bucket数目，至少为bnum

·此操作会让迭代器失效，改变元素之间的相对次序，会改变元素落在哪个bucket内。这个操作不会让指向元素的指针或者引用失效。

·如果有个异常被抛出，此操作不会带来任何影响( 除非是容器的hash函数或者比较函数抛出的)

·对于unordered multiset、unordered multimap,rehashing会保存相等元素之间的相对次序

·提供者:unordered set、unordered multiset、unordered map、unordered multimap

**7.7.2Unordered容器的bucket接口**

**Size\_type container::bucket\_count() const**

返回unordered容器的当前bucket数量

**Size\_type container::max\_bucket\_count() const**

·返回unordered容器最大可能的bucket数量

**Size\_type container::bucket(const key\_type& key) const**

·返回一个bucket编号，该bucket是带有于key等价键的元素落脚处

·返回值必须在区间0-----bcket\_count()内

·如果bucket\_count()是0.返回值不明确

**Size\_type container::bucket\_size(size\_type bucketIdx) const**

·返回编号为bucketidx的bucket内的元素数量

·如果bucketidx不是区间0-----bcket\_count()内的一个有效索引，此操作的结果不明确

**Local\_iterator container::begin(size\_type bucketIdx)**

**Const\_local\_iterator container::begin(size\_type bucketIdx) const**

**Const\_local\_iterator container::cbegin(size\_type bucketIdx) const**

·所有的三个形式都返回一个iterator，指向编号为bucketIdx的bucket内的所有元素的起点

·如果编号为bucketIdx的bucket为空，此调用相当于container::end(bucketIdx)或container::cend(bucketIdx)

·如果bucketIdx不是有效区间内的索引，操作结果不明确

**Local\_iterator container::end(size\_type bucketIdx)**

**Const\_local\_iterator container::end(size\_type bucketIdx) const**

**Const\_local\_iterator container::cend(size\_type bucketIdx) const**

·所有的三个形式都返回一个iterator，指向编号为bucketIdx的bucket内的所有元素的终点(最末元素的下一个位置)

·如果编号为bucketIdx的bucket为空，此调用相当于container::begin(bucketIdx)或container::cbegin(bucketIdx)

·如果bucketIdx不是有效区间内的索引，操作结果不明确

**附录:什么样的情况下会导致指向容器内元素的指针、引用、迭代器失效呢？**

如果元素本身存储在array类似的动态(一整段连续内存空间)容器内，那么插入、删除或是各种可能造成重新分配的操作都可能使得原本指向元素的迭代器、指针、引用失效。比如vector、deque。但是像list这种，因为插入、删除等操作并不会引起原本元素的内存空间的重分配(都是通过指针来操作的)，这种就不会造成迭代器、指针、引用的失效

而在无序容器中,bucket是连续的内存区域，而迭代器正是基于bucketidx的，所以rehash会造成原本指向元素的迭代器失效。但是指针和引用并不依赖于bucketidx，且元素存储在list中，自然不会造成指针和引用的失效。

在关联式容器内，道理同list，也绝对不会造成迭代器、引用和指针的失效。

所以，为了保证指针、迭代器、引用的有效性，以及免于重新分配内存，可使用reserve的方式来为诸如vector等容器预先留有足够的空间。

1. **迭代器**

**8.1迭代器种类**

|  |  |  |
| --- | --- | --- |
| 迭代器种类 | 能力 | 提供者 |
| Output迭代器 | 向前写入 | Ostream,inserter |
| Input迭代器 | 向前读取一次 | Istream |
| Forward迭代器 | 向前读取 | Forward list,无序容器 |
| Bidirectional迭代器 | 向前和向后读取 | List、关联式容器 |
| Random-acess迭代器 | 以随机访问方式读取 | 序列式容器、String、C数组 |

**Output迭代器**

Output迭代器允许一步步前行并且搭配write动作。因此可以一个一个元素地赋值，不能够使用output迭代器对同一区间迭代两次。

一个典型地pure output迭代器的例子是:将元素写至标准输出设备(比如屏幕或者打印机)。如果采用两个output迭代器写至屏幕，第二个字将跟在第一个字后面，而不是覆盖第一个字。

**Input迭代器**

Input迭代器只能读取元素一次，如果复制input迭代器，并命令原input迭代器和新产生的拷贝都向前读取，可能会遍历到不同的值。

所有迭代器都具备input迭代器的能力，而往往更强。

**前向迭代器**

前向迭代器是一种input迭代器，并且在前进读取时提供额外保证。和input迭代器不同的是，两个前向迭代器如果指向同一元素，operator==会返回true。

实际上标准库也允许无序容器使用双向迭代器，而不仅仅是前向。

如果forward迭代器履行了output迭代器应有的条件，那么它就是一个mutable forward迭代器，既可用于读，也可用于写。

**双向迭代器**

双向迭代器在前向迭代器的基础上增加了回头迭代的能力。换而言之，其支持递减操作符。

如果双向迭代器履行了output迭代器应有的条件，那么它就是一个mutable bidirectional迭代器，既可用于读，也可用于写。

**随机访问迭代器**

随机访问迭代器在双向迭代器的基础上增加了随机访问的能力。因此它必须提供iterator算术运算(和寻常的pointer算术运算相当)。也就是说，它能增减某个偏移量，计算距离，并诸如运用<和>等关系操作符进行比较。

随机访问迭代器主要由String、Wstring、C-style array和序列式容器中的一部分提供(deque、Array、Vector等数组类型的)

如果random-access迭代器履行了ouput迭代器应有的条件，那么它就是一个mutable random-access迭代器，即可用于读，也可用于写。

**附:Vector迭代器的递增和递减**

一般而言可以递增或递减临时性迭代器，但对于array、vector和string就不行(在某些编译平台上)，比如下例

If(coll.size()>1)

{

Std::sort(++coll.begin(),coll.end());

}

编译++coll.beign()可能会失效，实际情况取决于平台。但如果用deque代替vector就可以通过编译。

这种奇怪问题的原因在于，vector、array和string等迭代器通常被实现为寻常指针。而C++不允许你修改任何基础类型(包括pointer)的临时值，但对于struct和class则允许。

为了保证可移植，C++11提供了next()，所以上述程序应当这么写

If(coll.size()>1){

Std::sort(std::next(coll.begin()),coll.end());

}

(Ps:

++a表示取a的地址，增加它的内容，然后把值放在寄存器中

a++表示取a的地址，把它的值装入寄存器，然后增加内存中的a的值。操作用到的都是寄存器中的值！

故前置++的返回类型是引用，后置++的返回临时对象(这样会增加额外的构造和析构)。这意味着，前置++返回的是左值，后置++返回的是右值。任何右值都不应被赋值)

**8.2 迭代器相关辅助函数**

**Advance()**

Advance()可以将迭代器的位置增加，增加幅度由实参决定。

Void advance(InputIterator& pos, Dist n)

此函数总能根据迭代器种类采取最优方案，这归功于它使用了iterator trait(迭代器特征).

**Next()和prev()**

C++11提供者两个新增的辅助函数，允许前进/后退迭代器的位置

FowardIterator next(FowardIterator pos)

FowardIterator next(FowardIterator pos,Dist n)

·导致forward迭代器前进1个或者n个位置

·如果处理的是双向和随机迭代器，n可以为负值

·Dist是类型std::iterator\_traits<FowardIterator>::difference\_type

·其内部将对一个临时对象调用advance(pos,n)

·注意，next()并不检查是否会跨越序列的end()，因此调用者必须自行担保其结果有效

BidrectionalIterator prev(BidirectionalIterator pos)

BidrectionalIterator prev(BidirectionalIterator pos,Dist n)

·导致双向迭代器pos后退1个或者n个位置

·n可以为负值，导致前向移动

·Dist是类型std::iterator\_traits<FowardIterator>::difference\_type

·其内部将对一个临时对象调用advance(pos,-n)

·注意，prev()并不检查是否会跨越序列的begin()，因此调用者必须自行担保其结果有效

**Distance()**

函数distance()用来处理两个迭代器间的距离

Dist distance(InputIterator pos1,InputIterator pos2)

·返回两个input迭代器pos1和pos2之间的距离

·两个迭代器必须指向同一个容器

·如果不是随机访问迭代器，则从pos1开始必须能够前进到pos2.

·Dist是类型std::iterator\_traits<FowardIterator>::difference\_type

这种函数能根据迭代器种类采用最佳实现手法。对于随机访问迭代器，返回值仅仅是pos2-pos1，具备常量时间复杂度。而对于其他种类迭代器，distance会不断递增pos1，直到能够到达pos2为止，然后返回递增次数。

**Iter\_swap()**

这个简单的辅助函数用来交换两个迭代器所指的元素值。

Void iter\_swap(FowardIterator1 pos1,FowardIterator2 pos2)

·交换迭代器pos1和pos2所指的值

·迭代器类型不必相同，但其所指的两个值必须可以相互赋值

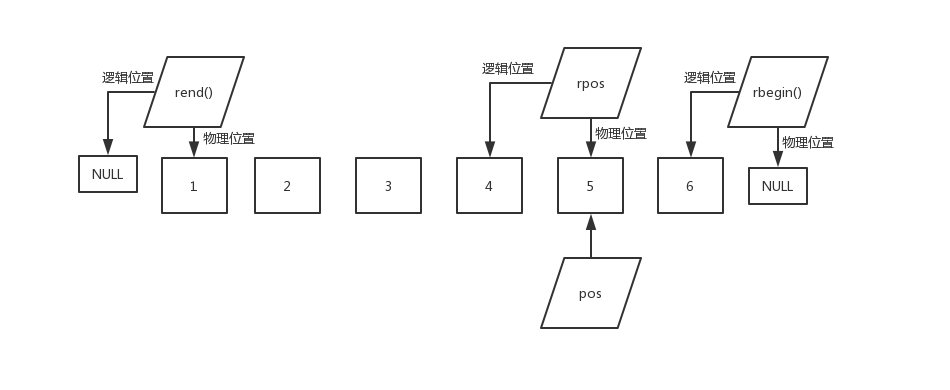
**8.3 迭代器适配器**

**8.3.1反向迭代器**

Reverse迭代器重新定义了递增和递减运算，使得其行为恰好颠倒。因此，如果使用此迭代器，算法将以反向次序处理元素。

容器的成员函数rbegin()和rend()各返回一个reserve迭代器。

可以将正常的迭代器转化为反向迭代器，当然，原本的那个迭代器必须具有双向移动能力。但是要注意的是，转换前后迭代器的逻辑位置发生了变化(物理位置不变)。



如上图所示。实际上，rbegin()的物理位置和end()是一样的，但是，其逻辑位置却指向了end的前一位置。而对齐进行\*操作，返回值将会是逻辑位置的值。

类似地，如果原本pos所指地元素是5，将其转化为反向迭代器后，其虽然物理位置仍处于元素5所在的位置，但是逻辑位置却指向了5的前一位置，也就是4！

这种转换实际上是倒置了原本的半开区间，使得转换后每一个位置上的元素仍然有效。

反向迭代器也提供了一个base()，可以将反向迭代器转换为正常的迭代器。

**8.3.2Insert迭代器**

Insert迭代器，也称作inserter，用来将“赋予新值”的动作转换为“安插新值”的动作。借由这种迭代器，算法将执行安插行为而非覆写行为。所有的insert迭代器都隶属ouput迭代器种类，所以它只提供赋值能力(assign就是overwrite)

在inserter中，operator\*是无效的，仅仅返回一个\*this,所以此时pos和\*pos等效。

注意，对于这样的表达式

\*pos =value;

如果是inserter，那么共分为两步，第一步先返回迭代器的位置，第二部将赋值转化为安插动作(调用push\_back()、push\_front()或者insert()成员函数)

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | Class名 | 实际作用函数 | 调用式 |
| Back Inserter | Back\_insert\_iterator | Push\_back(value) | Back\_inserter(cont) |
| Front Inserter | Front\_insert\_iterator | Push\_front(value) | Front\_inserter(cont) |
| General Inserter | Insert\_iterator | Insert(pos,value) | Inserter(cont,pos) |

(注:这里cont表示容器名)

比如下例

Vector<int> coll;

Back\_inserter(coll) = 20;

相当于在容器coll的末尾插入20.

**8.3.3Stream(串流)迭代器**

Stream迭代器时一种迭代器适配器。借由它，可以把stream当成算法的来源端和目的端。更明确地说，一个istream迭代器可以用来从input stream读取元素，而一个ostream迭代器可以用来对output stream写入元素。

**Ostream迭代器**

Ostream迭代器可以将“被赋值”写入output stream。用了它，算法就可以直接写入stream。其实现机制与inserter类似，唯一区别在于ostream迭代器将赋值动作转化为output动作(通过operator<<),如此依赖算法就可以使用寻常的迭代器接口直接对stream执行涂写动作。

**Istream迭代器**

Istream迭代器是Ostream迭代器的拍档，用来从input stream读取元素。通过input stream迭代器，算法可以从stream直接读取数据。

建立istream迭代器时，必须提供一个input stream作为实参，迭代器将从中读取数据，然后它经由input迭代器的寻常接口，使用operator>>读取元素。然而，读取动作有可能失败(EOF,或者读取错误)，此外算法的数据来源端也需要一个“终点位置”。为解决这些问题，你可以使用一个所谓的end-of-stream迭代器。它以istream迭代器的default构造函数生成。只要有任何一次读取失败，所有的istream迭代器都会变成end-of-stream迭代器。所以经过一次读取后，就应该将istream迭代器拿来和end-of-stream迭代器比较一番，看看这个迭代器是否还有效。

**8.3.4Move(搬移)迭代器**

从C++11开始，提供了一个迭代器适配器，用来将任何对底层元素的处理转换为一个move操作。如下

Std::list<std::string> s;

…

Std::vector<string> v1(s.begin(),s.end());

Std::vector<string> v2(make\_move\_iterator(s.begin()),make\_move\_iterator(s.end()));

这些迭代器的应用之一是，让算法以move语义代替copy语义，将元素从一个区间放到另一个区间。

注意，唯一保证元素只被读取或处理一次的迭代器种类是input iterator，因此，通常只有当某算法有个来源端，其内要求input iterator，并且有目的段，其内使用ouput iterator。这时候使用move迭代器才有意义。

**8.4 Iterator Trait(迭代器特性)**

迭代器可以区分为不同类型，每个类型都代表特定的迭代器能力。如果能根据不同的迭代器种类将操作行为重载，将会很有用，甚至很必要。、

通过迭代器标志(tag)和特性(trait,由<iterator>提供)，可以实现这样的重载。

C++标准库为每个迭代器种类都提供了一个迭代器标志，用来作为迭代器的标签。

Namespace  std{

Struct ouput\_iterator\_tag{

};

Struct input\_iterator\_tag{

};

Struct forward\_iterator\_tag{

:public input\_iterator\_tag;

};

Struct bidirectional\_iterator\_tag{

:public forward\_iterator\_tag;

};

Struct random\_access\_iterator\_tag{

:public bidirectional\_iterator\_tag;

};

}

注意，这里应用了继承，所以说，任何forward迭代器都是一种input迭代器。然而请注意，forward迭代器tag只派生自input迭代器tag，和output迭代器的tag没有关系，实际上forward迭代器的某些特性的确不符合ouput迭代器的要求。

如果撰写泛型代码，可能不止对迭代器种类感兴趣，还需要了解迭代器所指的元素类型。C++标准库提供了一种特殊的模板结构来定义所谓的迭代器特性(iterator trait)，该结构包含了迭代器相关的所有信息，为迭代器应具备的所有类型定义提供了一致的接口。

Namespace std{

Template <typename T>

Struct iterator\_traits{

Typedef typename T::itertator\_category iterator\_category;

Typedef typename T::Value\_type value\_type;

Typedef typename T::difference\_type difference\_type;

Typedef typename T::pointer pointer;

Typedef typename T::reference reference;

}

}

在这个模板中，T表示迭代器类型。有了它，就可以撰写出任何运用迭代器种类或其元素类型的泛型码。例如一下表示可以取得迭代器类型为T的元素类型:

Typename std::iterator\_traits<T>::value\_type (调用iterator\_traits<T>类中的value\_type详单与T类型迭代器的value\_type)

这个trait结构有两个优点:

1. 确保每一个迭代器都提供了必要的类型定义
2. 能够针对特定的迭代器实行特化

**8.4.1为迭代器编写泛型函数**

借由iterator trait，可以编写这样的泛型函数:根据迭代器而派生某种类型定义。或根据迭代器种类而采用不同的实现代码。

**运用迭代器的value\_type**

这些算法内部需要一个以元素类型为类型的临时变量，可以声明如下:

Typename std::iterator\_traits<T>::value\_type tmp;

其中T是迭代器的类型。

现在，tmp就是一个T类型迭代器中的元素类型的变量了。

**运用迭代器的iterator\_category**

如果希望针对不同的迭代器种类采用不同的实现方案，需要下面两个步骤

1. 让你的template函数将迭代器作为附加实参，调用另一个函数

Template<typename Iterator>

Inline void foo(Iterator beg,Iterator end)

{

Foo(beg,end,std::iterator\_traits<Iterator>::iterator\_category());

}

1. 针对不同的迭代器种类实现出上述的调用函数

Template<typename BiIterator>

Void foo(BiIterator beg,BiIterator end,std::bidirectional\_iterator\_tag)

{

…

}

Template<typename RaIterator>

Void foo(RaIterator beg,RAIterator end,std::random\_access\_iterator\_tag)

{

…

}

其中，针对random-acess迭代器而写的版本支持随机访问操作，针对双向迭代器的则不可以。基于迭代器标志继承体系，可以写出一份实现代码但使用于多个迭代器种类。

1. **STL函数对象及Lambda**

**9.1Fuction Object**

所谓function object就是一个定义了operator()的对象。如下

FunctionObjectType fo；

fo(…)

这里，表达式fo()系调用function object fo的operator()而非调用函数fo。可以将function object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。

函数对象比一般的函数更灵活，因为它可以拥有状态。实际上，对于相同的函数对象类，你可以有两个不同的实例。寻常的函数无法做到这一点

每个函数对象都有其类型，因此可以将函数对象当作模板参数传递，从而指定某种行为，还有一个好处，容器类型也会因为函数对象的不同而不同。

执行速度上，函数对象比函数指针更快。

所谓判断式，就是返回Boolean值的函数或者函数对象。对STL而言并非所有返回Boolean值的函数都是合法的判断式。这可能会导致出人意料的行为。

**9.2预定义的Function Object和Binder**

C++标准库提供了许多预定义的function object和binder，后者允许你合成更多精巧的function object。

**预定义的函数对象**

预定义的function object 如下

|  |  |
| --- | --- |
| 表达式 | 效果 |
| Negate<type>() | -param |
| Plus<type>() | Param1+param2 |
| Minus<type>() | Param1-param2 |
| multiplies<type>() | Param1\*param2 |
| Divedes<type>() | Param1/param2 |
| Moldulus<type>() | Param1%param2 |
| Equal\_to<type>() | Param1 == param2 |
| Not\_equal\_to<type>() | Param1 != param2 |
| Less<type>() | Param1 < param2 |
| Greater<type>() | Param1>param2 |
| Less\_equal<type>() | Param1<=param2 |
| Greater\_equal<type>() | Param1 >=param2 |
| Logical\_not<type>() | !param |
| Logical\_and<type>() | Param1&&param2 |
| Logical\_or<type>() | Param1||param2 |
| Bit\_and<type>() | Param1&param2 |
| Bit\_or<type>() | Param1|param2 |
| Bit\_xor<type>() | Param1^param2 |

**函数适配器和binder**

所谓函数适配器，是指能够将不同的函数对象(或是和某值或某寻常函数)结合起来的东西，它自身也是个函数对象。然而随着时间过去，函数对象的合成方法有了变化。事实上，C++98提供的所有这些方面的特性已经因为C++11的到来全部过时。C++11提供了更具弹性的适配器。

自C++11开始，标准库提供的函数适配器如下

|  |  |
| --- | --- |
| 表达式 | 效果 |
| Bind(op,args…) | 将arqs绑定给op |
| Mem\_fn(op) | 调用op，把它当作某个对象的成员函数 |
| Not1(op) | 单参反相:!op(param) |
| Not2(op) | 双参反相:!op(param1,param2) |

最终要的适配器就是binder，它允许你：

·在既有的或者预定义的函数对象之外另配及合成新的函数对象

·调用全局函数

·针对对象，指针和智能指针调用成员函数

一般而言bind()用来将参数绑定于可调用对象。因此，如果一个函数、成员函数、函数对象或lambda需要若干参数，你可以将参数绑定为“明白指出的”或“被传入的”实参。欲明白指出实参，你只需写出其名称，欲使用被传入的实参，则可以利用预定义占位符\_1、\_2、…它们被定义于命名空间std::placeholders内。

Binder的一个典型应用时，当使用C++标准库提供之预定义函数对象时，具体指定参数。

比如下例:

Std::bind(std::plus<int>(),std::placeholders::\_1,10);

它所表现的函数对象内部调用了plus<>，以占位符\_1作为它的第一参数，以10作为第二参数。占位符\_1表示实际传入此表达式的第一实参。于是，对于任何传给此表达式的任何实参，这个函数对象都会产生出“实参+10”的结果值。

再比如下面这个例子

Auto pos = std::find\_if(coll.begin(),coll.end(),std::bind(std::greater<int>(),\_1,42))

其会返回第一个大于42的元素

下例是一个嵌套的binder

Bind(multiplies<int>(),bind(plus<int>(),\_1,10),2);

其会把10加在任何传入的实参上，然后再将结果乘以2。

**运用binder调用成员函数**

如下例

Bind(&Person::print,\_1)

定义一个函数对象，其内针对传入得Person调用param1.printf().也就是说，由于第一个实参是一个成员函数，下一个实参将会定义“用以调用成员函数”的对象(占位符\_1所表示)

其他任何实参都会被传递给该成员函数，这意味着：

Bind(&Person::print2,\_1,”Person ”)

定义出一个函数对象，其内针对传入的person调用param1.print2(“Person ”)

原则上也可以直接传递对象，如下

Person n(”nico”);

Bind(&Person::print2,\_1,”This is ”)(n);

这将会调用n.print2(“This is ”);

**Mem\_fn() Adapter**

对于成员函数，可以使用mem\_fn() adapter，那就不再需要以占位符表示其调用者

Std::for\_each(coll.begin(),coll.end,std::mem\_fn(&Person::print))

若有额外实参被传递给成员函数，mem\_fn()就拿第一实参作为调用者(对象)，其他实参当作成员函数的实参。

Std ::mem\_fn(&Person::print)(n) //calls n.print();

Std ::mem\_fn(&Person::print2)(n,”Person: ”) // call n.print2(“Person: ”);

然而，如果要为函数对象绑定实参，还是必须调用bind()

Std::for\_each(coll.begin(),coll.end().

Std::bind(std::mem\_fn(&Person::print2),

Std::placeholders::\_1,

“Person: ”));

|  |  |
| --- | --- |
| 表达式 | 效果 |
| Bind1st(op,arg) | 调用op(arg,param) |
| Bind2nd(op,arg) | 调用op(param,arg) |
| Ptr\_fun(op) | 调用\*op(param)或\*op(param1,param2) |
| Mem\_fun(op) | 调用op()视为一个对象指针的成员函数 |
| Mem\_fun\_ref(op) | 调用op()视为一个对象的成员函数 |
| Not1(op) | 单参反向 |
| No2(op) | 双参反向 |

**9.3运用Lambda**

Lambda的引入始于C++11,这带来一种很具威力又十分便利的方法，允许我们提供局部技能，特别是用来具体指明算法和成员函数的细节。

Lambda可以提供和binder相似的作用

Lambda可以提供带有状态的函数对象相似的作用

Lambda可以调用全局函数和成员函数

Lambda可以作为hash函数、排序准则或者相等准则。

1. **STL算法**

**10.1算法分类**

不同的算法满足不同的需求，所以可以根据其主要目的加以分类。

为了顾名思义，STL设计者为算法命名时，引入了两个特殊后缀。

**后缀\_if:**如果某算法有连个形式，参数个数相同，但第一形式要求传入一个值。，第二形式要求传入一个函数或function object，那么后缀\_if就派得上用场。例如find()用来查找具有某值的元素，find\_if则接受一个被当作查找准则的函数或者函数对象，查找第一个满足该准则的元素。

**后缀copy:**这个后最用来表示此算法中元素不只被操作，还会被复制到标的区间。例如reserve() 将区间中的元素颠次序。而reserve\_copy()则是逆序将元素赋值到另一个区间。

STL算法大致可以分类如下

·非更易型算法

·移除型算法

·变序型算法

·排序算法

·已排序区间算法

·数值算法

**10.2 for\_each算法**

For\_each()算法非常灵活，它允许你以不同方式访问、处理、修改每一个元素，然而注意，自C++11起，range-based-for循环提供了更方便自然的行为，因此，for\_each恐将日渐丧失其重要性。

For\_each(InputIterator beg,InputIterator end,UnaryProc op)

·对区间[beg,end)中的每一个元素调用op

·返回op(它已在算法内部被改动过)的一个副本

·op可以改动元素

·op的任何返回值都会被忽略

·复杂度:线性

**10.3非更易型算法(只列举常用且比较重要的)**

**10.3.1元素计数**

diffrence\_type Count(inputIterator beg,inputIterator end,const T& value)

diffrence\_type Count(inputIterator beg,inputIterator end,UnaryPredicate op)

·第一形式返回区间内元素值等于value的元素个数

·第二形式返回区间内元素值让op为true的元素个数

·返回类型是diffrence\_type,用以表现iterator间距的类型

·关联和无序容器提供了一个类似的成员函数count(),用来计算“以某给定值为key”的元素个数

·复杂度:线性

**10.3.2查找元素**

InputIterator find (inputIterator beg,inputIterator end,const T& value)

InputIterator find\_if(inputIterator beg,inputIterator end,UnaryPredicate op)

InputIterator find\_if\_not(inputIterator beg,inputIterator end,UnaryPredicate op)

·第一形式返回区间内第一个元素值等于value的元素位置

·第二形式返回区间内第一个元素值满足让op为true的元素位置

·第三形式返回区间内第一个元素值造成一下op为false的元素位置

·如果没有找到匹配元素，它们均返回end()

·如果是已排序区间，应当使用lower\_bound()、upper\_bound()、equal\_range()和binary\_search()等算法以获得更高效能

·关联和无序容器提供一个等效的成员函数find(),拥有较好的复杂度

·复杂度:线性

**10.3.3区间的比较**

Bool equal(InputIterator1 beg,InputIterator1 end,InputIterator2 Cmp)

Bool equal(InputIterator1 beg,InputIterator1 end,InputIterator2 Cmp,binaryPredicate op)

·第一形式判断区间内的元素是否都和“以cmp开头的区间“内的元素相等

·第二形式判断区间内的元素和“以cmp开头的区间“内的对应元素是否都能够造成op为true。

·op不应在函数调用过程中改变状态

·op不应改动传入的实参

·调用者必须确保以cmp开头的区间内有足够的元素

·当序列不相等时，如果想要了解其中的不同，应该使用mismatch()算法

·复杂度:线性

Pair<InputIterator1,InputIterator2>

Mismatch(InputIterator1 beg,InputIterator1 end,InputIterator2 cmpBeg)

Pair<InputIterator1,InputIterator2>

Mismatch(InputIterator1 beg,InputIterator1 end,InputIterator2 cmpBeg,BinaryPredicate op)

·第一形式返回区间内和“以cmpBeg起始的区间“内第一组两两相异的元素。

·第二形式返回区间内和“以cmpBeg起始的区间”内第一组造成op为false的元素(op(elem,cmpelem));

·如果没有找到相异点，就返回“以end和第二序列对应元素组成”的pair<>。这并不意味着两个序列相等，因为第二序列可能含有更多元素

·如果想了解两个序列是否相等，应该使用equal()算法。

·复杂度:线性

**10.4更易型算法(只列举常用且比较重要的)**

**10.4.1复制元素**

OutputIterator

copy(InputIteraotr sourceBeg,InputIterator sourceEnd,OutputIterator destBeg)

OutputIterator

Copy\_if(InputIteraotr sourceBeg,InputIterator sourceEnd,

OutputIterator destBeg,UnaryPredicate op)

OutputIterator

Copy\_n(InputIterator sourceBeg,num,OutputIterator destBeg)

·上述算法都将区间内的所有元素复制到以destBeg为起点的目标区间内

·它们都返回目标区间内最后一个被复制元素的下一位置。也就是第一个未被覆写的元素的位置。

·关于copy()，destBeg不可以处于[sourceBeg,sourceEnd)区间内。关于copy\_if(),源区间和目标区间不可重叠。

·copy()正向遍历

**10.4.2搬移元素**

OutputIterator

Move(InputIterator sourceBeg,InputIterator sourceEnd,OutputIterator destBeg)

·本算法将区间内的所有元素搬移至以destBeg为起点的区间内

·它们会针对每一个元素调用

\*destElem = std::move(\*sourceElem)

因此，如果元素类型有提供move语义，源端元素从此不再明确

·它们会返回目标区间的最后一个被复制元素的下一位置，也就是第一个未被覆写的元素

·move()的destBeg不可以处于[sourceBeg,sourceEnd)区间内

·move()正向遍历

·调用者必须确保目标区域有足够的空间，要不就得使用insert iterator

**10.4.3转换和结合元素**

OutputIterator

Transform(InputIterator sourceBeg,InputIterator sourceEnd,

OutputIterator destBeg,UnaryFunc op)

·针对区间内的每一个元素调用op(elem)并将结果写入以destBeg起始的区间内

·返回目标区间的最后一个被转换元素的下一位置。也就是第一个未被覆写的元素位置

·调用者必须确保目标有足够空间

·如果想以某值替换掉符合给定准则的元素，应该使用replace()

·复杂度:线性

OutputIterator

Transform(InputIterator1 source1Beg,InputIterator1 source1End,

InputIterator2 source2Beg,

OutputIterator destBeg,

BinaryFunc op)

·针对第一区间[source1Beg,source1End)以及“从source2Beg开始的第二源区间”的对应元素，调用

Op(source1Elem,source2Elem)

并将结果写入destBeg起始的区间内

·返回目标区间内最后一个被转换元素的下一位置，也就是第一个未被覆写的元素位置

·调用者必须确保第二源区间有足够空间

·调用者必须确保目标区间有足够空间，不然就使用insert iterator

·source1Beg，source2Beg和destBeg可能相同，因此可以让元素自己和自己结合，然后将结果覆盖自己

·复杂度:线性

**10.4.4互换元素**

FowardIterator2

Swap\_ranges(FowardIterator1 beg1，FowardIterator1 end1，ForwardIterator2 beg2)

·将第一区间内的元素和以“beg2开始区间”内的对应元素互换

·返回第二区间内最后一个被交换元素的下一位置

·调用者必须确保目标区间有足够空间

·两个区间不得重叠

·如果要将相同类型的两个容器内的所有元素都互换，应该使用swap()成员函数，因为该成员函数通常具备常量复杂度

·复杂度:线性

**10.4.5赋值**

Void

fill(FowardIterator beg,ForwardIterator end,const T&newValue)

Void

Fill\_n(OutputIterator beg,size num,const T&newValue)

·fill()将区间内每一个元素都赋予新值newValue

·fill\_n()将从beg开始的前num个元素赋予新值newValue.如果num值为负则不做任何事

·调用者必须确保目标区间有足够空间，否则调用insert iterator

·自C++11起，fill\_n()返回被改动元素的下一位置(beg+num)，如果num是负数就返回beg。

·复杂度:线性

**10.4.6替换元素**

Void

Replace(ForwardIterator beg,ForwardIterator end,const T&oldValue,const T&newValue)

Void

Replace\_if(ForwardIterator beg,ForwardIterator end,UnaryPredicate op ,const T&newValue)

·replace()将区间内每一个与oldValue相等的元素替换为newValue

·replace\_if()将区间内每一个让op为true的元素替换为newValue

·复杂度:线性

OutputIterator

Replace\_copy(ForwardIterator sourceBeg,ForwardIterator sourceEnd,

OutputIterator destBeg,

const T&oldValue,const T&newValue)

OutputIterator

Replace\_copy\_if(ForwardIterator sourceBeg,ForwardIterator sourceEnd,

OutputIterator destBeg,

UnaryPredicate op,const T&newValue)

·replace\_copy()是copy()和replace()的结合。它将源区间中的元素复制到以destBeg为起点的区间，然后将目标区间内与oldValue相等的元素替换为newValue

·replace\_copy\_if()将源区间中的元素复制到以destBeg为起点的区间内，然后将目标区间中使得op(实际为op(elem))为true的元素替换为newValue.

·两个算法都返回目标区间中最后一个被复制元素的下一位置，也就是第一个未被覆写的元素位置。

·调用者必须确保目标区间有足够空间，否则使用insert iterator

·复杂度:线性

**10.5移除型算法(只列举常用且比较重要的)**

**10.7.1移除某些元素**

ForwardIterator

Remove(ForwardIterator beg,ForwardIterator end,const T&value)

ForwardIterator

Remove\_if(ForwardIterator beg,ForwardIterator end,UnaryPredicate op)

·remove()会移除区间内每一个与value相等的元素

·remove\_if()会移除区间内每一个让op(Elem)为true的元素

·两个算法都返回被改动序列的新的逻辑终点(也就是最后一个未被移除元素的下一位置）

·这些算法会把原本置于后面的未移除元素向前移动，覆盖被移除元素

·未被移除元素保持次序上不变

·调用者在调用此算法后，应当保证此次采用其所返回的新逻辑终点，不再使用原终点end()

·由于元素会被改动，所以这些算法不可以被用于关联式或者无序容器。

·list提供了一个效果相同的成员函数remove()，效能较高，原因是它不重新赋值，而是重新链接pointer(此为真正意义上的移除)

**10.7.2移除重复元素**

FowardIterator

Unique(FowardIterator beg,FowardIterator end)

FowardIterator

Unique(FowardIterator beg,FowardIterator end,

BinaryPredicate op)

·以上两种形式都会移除连续重复元素种的多余元素

·第一形式将区间内所有与前一元素相等的元素移除，因此，源序列必须经过排序才能够使用这个算法移除所有重复元素

·第二个形式将每一个位于元素e之后并造成op(e,elem)为true的所有elem元素移除。换而言之，上述predicate并非拿元素和其原本的前一元素比较，而是拿它和“经处理后仍健在”的前一元素比较

·两个形式都返回被改动的序列的新逻辑终点，也就是最后一个未被移除元素的下一位置

·两个算法将原本位置在后的未移除元素向前移动，覆盖掉被移除的元素

·未被移除元素在相对次序上保持不变

·调用者在使用此算法后，应当保证从此使用新的逻辑终点，而非原逻辑终点end()

·List提供了一个相同的成员函数unique(),其不会重新赋值，而是重新链接指针

·复杂度:线性

**10.5变序型算法(只列举常用且比较重要的)**

**10.5.1反转元素次序**

Void

Reverse(BidirectionalIterator beg,BidirectionalIterator end)

OutputIterator

Reverse\_copy(BidirectionalIterator sourceBeg,BidirectionalIterator sourceEnd,

OutputIterator destBeg)

·reverse()会将区间内的元素全部反转次序

·reverse\_copy()会将源区间内的元素复制到destBeg为起始的目标区间，并在复制过程种颠倒次序

·reverse\_copy()返回目标区间内的最后一个被复制元素的下一位置，也就是第一个未被覆盖的元素

·List提供了一个效果相同的成员函数reverse()，但它并非重新赋值，而是重新链接pointer，因此效能更佳

·复杂度:线性

**10.6排序算法(只列举常用且比较重要的)**

**10.6.1对所有元素排序**

Void

Sort(RandomAccessIterator beg,RandomAccessIterator end)

Void

Sort(RandomAccessIterator beg,RandomAccessIterator end,BinaryPredicate op)

Void

Stable\_sort(RandomAccessIterator beg,RandomAccessIterator end)

Void

Stable\_sort(RandomAccessIterator beg,RandomAccessIterator end,BinaryPredicate op)

·sort()和stable\_sort()的第一形式对区间内的所有元素使用opertaor<进行排序(结果造成升序)

·sort()和stable\_sort()的第二形式使用op(elem1,elem2)作为排序准则，对区间内的所有元素进行排序

·sort()和stable\_sort()的区别是，后者保证相等之各元素的相对次序在排序后保持不变

·不可以对list或者forward list调用这些算法，因为两者都不提供随机访问，不过它们自己提供了成员函数sort()可对自身元素排序

·sort()保证了很不错的平均效能n-log-n

·复杂度:sort()---平均n-log-n,

Stable\_sort()—如果系统有足够内存，那么就是n-log-n，否则就是n-log-n x log-n

**10.6.2局部排序**

Void

Partial\_sort(RandomAcessIterator beg,RandomAccessIterator sortEnd,

RandomAccessIterator end)

Void

Partial\_sort(RandomAcessIterator beg,RandomAccessIterator sortEnd,

RandomAccessIterator end,BinaryPredicate op)

·第一形式以operator<对区间[beg,end)内的元素进行排序，使得[beg,sortEnd)内的元素处于已排序状态

·第二形式以op(elem1,elem2)为准则进行局部排序

·和sort()不同，partial\_sort()不会对全部元素进行排序，一旦第一个元素至sortEnd的元素都排序妥当，就立刻停止。

·如果sortEnd等于end，partial\_sort()会对整个序列进行排序，平均效能不如sort()

**10.6.3根据第n各元素排序**

Void

Nth\_element(RandomAccessIterator beg,RandomAccessIterator nth,

RandomAccessIterator end)

Void

Nth\_element(RandomAccessIterator beg,RandomAccessIterator nth,

RandomAccessIterator end,BinaryPredicate op)

·两种形式都对区间内的元素排序，使得第n各位置上的元素就位。也就是说，所有在位置n之前的元素都小于等于它，所有在位置n之后的元素都大于等于它。这样就得到了根据n位置上元素分割而来的两个子序列。如果你只需要n各最大或者最小元素，但不要求它们处于排序状态，这个算法就很有用

·第一形式用operator<作为排序准则

·第二形式使用op(elem1,elem2作为排序准则)

·复杂度:平均为线性

**10.6.4heap算法**

就排序而言，堆排序是一种特殊的元素组织方式，heap可以被视为一个序列式集合实现而成的二叉树，具有两个性质:

1. 第一个元素总是最大
2. 总是能够在对数时间内增加或者删除元素

Heap是实现priority queue(其内元素会自动排序)的一个理想结构，因此heap算法也被priority\_queue容器使用。为了处理heap，STL提供了4个算法

1. make\_heap()将某区间内的元素转化为heap
2. push\_heap()对heap增加一个元素
3. pop\_heap()对heap取出一个元素、
4. sort\_heap()将heap转化为一个已排序群集(转化后它就不再是heap了)

**10.7已排序区间算法(只列举常用且比较重要的)**

**10.7.1查找元素**

Bool

Binary\_search(ForwardIterator beg,ForwardIterator end,const T& value)

Bool

Binary\_search(ForwardIterator beg,ForwardIterator end,const T& value,

BinaryPredicate op)

·两种形式都用来判断已排序区间内是否包含和value等值的元素

·如果想要获得所查找的元素位置，应该使用lower\_bound、upper\_bound、或者equal\_range()

·调用者必须确保，进入算法之际，工作区间已排序

·复杂度:如果搭配随机访问迭代器僵尸对数复杂度，否则就是线性复杂度

Lower\_bound()、upper\_bound()、equal\_range()之前已介绍过，这里不详细描述。

**10.7.2合并元素**

OutputIterator

Merge(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg)

OutputIterator

Merge(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg,BinaryPredicate op)

·两者都是将源区间[sourceBeg1,sourceEnd1)和[sourceBeg2,SourceEnd2)内的元素合并，使得“以destBeg为起始的目标区间”内涵两个源区间的所有元素

·目标区间的所有元素都将处于已排序的状态下

·两者都返回目标区间“最后一个被复制元素”的下一位置，也就是第一个未被覆盖的元素

·op是个可有可无的binary predicate，被视为排序准则

·源区间不会有任何变化

**两个已排序集合的并集**

OutputIterator

Set\_union(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg)

OutputIterator

Set\_union(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg,BinaryPredicate op)

·目标区间的所有元素都处于已排序状态

·同时出现于两个源区间内的元素，在并集区间将只会出现一次，但是如果原来的源区间存在重复元素，则目标区间也会存在重复元素----重复的次数是两个源区间内的重复次数的较大值。

·调用者必须确保源区间一开始就已排序

**两个集合的交集**

OutputIterator

Set\_intersection(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg)

OutputIterator

Set\_ intersection (InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg,BinaryPredicate op)

·目标区间的所有元素都处于已排序状态

·如果原来的源区间存在重复元素，则目标区间也会存在重复元素----重复的次数是两个源区间内的重复次数的较小值。

·调用者必须确保源区间一开始就已排序

**两个集合的差集**

OutputIterator

Set\_difference(InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg)

OutputIterator

Set\_difference (InpurtIterator source1Beg,InputIterator source1End,

InputIterator source2Beg,InputIterator source2End,

OutputIterator destBeg,BinaryPredicate op)

·目标区间所有元素都处于排序状态

·如果源区间有重复元素，目标区间也会有重复元素---重复的次数等于第一源区间的重复次数减去第二源区间的重复次数，如果第二源区间的重复次数大于第一源区间的重复次数，目标区间的对应重复次数将会是0

·两者都返回目标区间内被合并的最后一个元素的下一位置

·op是个可有可无的排序准则

·调用者必须确保源区间一开始就已排序

1. **特殊容器**

**11.1Stack(堆栈)**

Class stack<>实现出一个LIFO(后进先出)的数据结构。可以使用push()将任意数量的元素放入stack,也可以使用pop()将元素依其插入的相反次序从容器种移除。

**核心接口**

Push() 将一个元素放入stack内

Top() 返回stack内的“栈顶”元素

Pop() 移除stack内的“栈顶”元素

**11.2Queue(队列)**

Class queue<>实现出一个FIFO(先进先出)的数据结构。它是一个典型的数据缓冲区构造。

**核心接口**

Push() 将一个元素放入queue内

Front() 返回队列最前端(最早被安插)的元素

Back() 返回队列最末尾的元素

Pop() 移除最前端(最早被安插)的元素

**11.3Priority Queue(优先级队列)**

Class priority\_queue<>实现出一个队列，其中的元素依照优先级被读取。它的接口和queue非常接近。换句话说，优先级队列内的元素已经根据值进行了排序。

**核心接口**

Push() 将一个元素放入优先级队列内

Top() 返回优先级队列最前端(最大值)的元素

Pop() 从优先级队列中移除最前端(最大)的元素，如果有多个一样优先级的元素，移除哪个并不明确。

**11.4Bitset**

Bitset构造出了一个内含bit或者boolean值且大小固定的array。当你需要管理各式flag，并且以flag的任意组合来表现变量时，就可以运用bitset。

注意，不可以改变bitset的bit个数，这个数量的具体值是由template实参决定的。如果需要一个可变长度的bit容器，可以考虑用vector<bool>。

Bitset的一个强有力的特色就是可以在整数值和bit序列之间互相转化，只需要创建一个临时的bitset就可以办到，

如下，把267转换成一个24位的bits

Cout<<bitset<24>(267)<<endl;

注意，bitset内的bit个数应该小于sizeof(unsigned long long);

1. **字符串**

**C++11带来的变化**

·string如今提供了front()和back()，用来访问第一元素和最末元素。此外还提供了shirink\_to\_fit()用来缩小容量

·string如今提供了方便的函数来将其转化为数值，以及反向转换

·data()和c\_str()不会再造成“指向string”的reference、iterator和pointer的无效化

·string如今提供move语义以及初值列

**12.1String提供的操作**

|  |  |
| --- | --- |
| 操作 | 效果 |
| =,assign() | 赋予新值 |
| Swap() | 交换两个string内容 |
| +=,append(),push\_back() | 添加字符 |
| Insert() | 插入字符 |
| Erase()、pop\_back() | 删除字符，pop\_back始于C++11 |
| Clear() | 移除所有字符，使之为空 |
| Resize() | 改变字符数量(在尾部添加或删除) |
| Replace() | 替换字符 |
| + | 串接 |
| ==,!=,<,<=,>,>=.compare() | 比较 |
| Empty() | 判断是否为空 |
| Size(),length() | 返回字符数量 |
| Max\_size() | 返回字符最大可能数目 |
| Capacity() | 返回重分配前的字符容量 |
| Reserve() | 保留一定内存用以容纳一定数量字符 |
| Shrink\_to\_fit() | 缩减内以符合当前字符数量 |
| [ ],at() | 访问某单一字符 |
| Front(),back() | 访问第一或最末字符 |
| >>,getline() | 从一个stream读取某值 |
| << | 将某值写入stream |
| Stoi(),stol(),stoll() | 将字符串转化为带符号的整数 |
| Stoul,stoull() | 将字符串转换为不带符号的整数 |
| Stof(),stod(),stold() | 将stirng转换为浮点数 |
| To\_string(),to\_wstring() | 将整数、浮点数转换为string |
| Copy() | 将string复制为一个字符array(C-string) |
| Data(),c\_str() | 将string内容以C-string形式返回 |
| Substr() | 返回某个子字符串 |
| Find functions() | 查找某个子字符串或字符 |
| Begin(),end() | 提供正常的迭代器操作 |
| Cbegin(),cend() | 提供const迭代器支持 |
| Rbegin(),rend() | 提供reverse迭代器支持 |
| Crbegin(),crend() | 提供const reverse迭代器支持 |
| Get\_allocater() | 返回分配器 |

**12.2String的构造和析构函数**

注意，不能以单一字符初始化某字符串，如下

Std::string(‘x’);

但是却可以这么做

Std::string(1,’x’);

Or

Std::string({‘x’});

这表示存在一个隐式转换将const char\*转化为string，但不存在一个从char到string的自动类型转换。

|  |  |
| --- | --- |
| 表达式 | 效果 |
| String s | 生成一个空的string s |
| String s(str) | 拷贝构造函数，创建一个str的拷贝 |
| String s(rvstr) | Move构造函数 |
| String s(str,stridx) | 将string str内处于stridx的部分当作s初值 |
| String s(str,stridx,strlen) | 将string str内处于stridx，且长度顶多strlen的部分当作s初值 |
| String s(cstr) | 以C-string cstr作为字符串s的初值 |
| String s(chars,charslen) | 以字符数组chars的钱charslen个字符作为字符串s的初值 |
| String s(num,c) | 生成一个字符串，包含num个c字符 |
| String s(beg,end) | 以[beg,end)区间内的字符作为字符串s的初值 |
| String s(inilist) | 以初值列作为s的初值 |
| s.~string() | 销毁所有字符，释放内存 |

**12.2大小和容量**

Size和length()返回string的现有字符数，两个函数等效。根据STL概念，size是获取容器元素个数的通用函数，而length()一般对应于C-string的length()函数，返回string长度。

Max\_size()返回string最多能包含的字符数。String通常包含单独一个内存区块的所有字符，所以可能与PC本身的限制有关。返回值一般是索引最大值-1

Capacity()返回重新分配内存之前，string所能包含的最大字符数。

让string有足够的容量是很重要的，原因有二：

1. 重新分配会造成所有指向string的指针、引用和迭代器失效
2. 重新分配很耗费时间

容量概念应用于string和应用于vector是相同的，但有一个显著差异:对string可以调用reserve()来缩减实际容量，但vector的reserve()没有这项异能。拿一个小于现有容量的实参来调用reserve()实际就是一种非强制性的缩减请求。如果实参小于现有字符数，则被视为“非强制性合身缩减”的请求。也就是说，你可能想要缩减容量至某个目标，但不一定能如愿。String reserve()的实参默认为0，所以调用reserve()且不给实参就是一种非强制性合身缩减。

自C++11起提供了shirink\_to\_fit()，实现了和reserve()相同的非强制性合身缩减效果。

C++标准规定，唯有在回应reserve()或者shirink\_to\_fit()调用时，容量才有可能缩减。因此，即使发生字符串被删除或改变时，其他任何字符只要位于“被操作字符”之前，那些指向它们自身的引用、指针和迭代器就仍然有效。

**12.3元素访问**

·和之前所言一样。Operator[ ]不检查索引是否有效，at()则会检查。

·一般而言，最末字符的更后位置也是有效的，因此，当前实际字符数可被当成一个有效索引。此情况下operator[ ]返回的是由字符default构造函数所产生的字符，就string对象而言是char ’\0’

·front()相当于[0]，意味着对于空的string，也就是EOF

·对于at(),“当前字符个数”不是有效索引

·对于一个空的string调用back()会导致不明确行为。

**12.4 子字符串以及字符串接合**

可以使用成员函数substr()从string身上提取出子串，例如

Std::string s(“interchangeability”);

s.substr(); //返回s的拷贝

s.substr(11); //返回从第11个字符之后的所有字符

s.substr(5,6); //返回从第5个字符开始的长度为6的子串

s.substr(s.find(‘c’)); //返回从字符c开始的字串

可以使用operator+把两个string接合起来，如

Std::string s1(“enter”);

Std::string s1(“nation”);

Std::string i18n;

I18n = ‘i’ + s1.substr(1) + s2 + “aliz” + s2.substr(1);

**12.4 搜索和查找**

|  |  |
| --- | --- |
| String函数 | 效果 |
| Find() | 查找第一个与value相等的字符，返回其索引 |
| Rfind() | 查找最后一个与value相等的字符，返回其索引 |
| Find\_first\_of() | 查找第一个与value中的某值相等的字符，返回其索引 |
| Find\_last\_of() | 查找最后一个与value中的某值相等的字符，返回其索引 |
| Find\_first\_not\_of() | 查找第一个与value中的任何值都不相等的字符，返回其索引 |
| Find\_last\_not\_of() | 查找最后一个与value中的任何值都不相等的字符，返回其索引 |

所有查找函数都返回字符串中“符合查找条件”的第一个字符的索引。如果查找不成功就返回npos，这些查找函数都采用下面的体系

·第一实参永远是被查找的对象

·第二实参指出string内的查找起点

·第三实参指出欲查找的字符个数

使用string的npos时要格外小心，如果要检查函数返回值，一定要使用类型string::size\_type，不能以int或者unsigned作为返回值的类型，否则返回值与string::npos之间的比较可能无法正常进行！

1. **并发**

**13.1高级接口async()和Future**

“以多线程允许程序”的最佳起点就是C++标准库中的std::async()和Class std::future<>提供的高级接口。

·async()提供一个接口，让一段机能或者一个回调对象若是可能的话在后台运行，成为一个独立线程。

·Class future<>允许等待线程结束并获得其结果(一个返回值，或者也许是一个异常)

例:如果需要计算两个操作的和

Func1() + Func2()

原本程序需要先调用func1再调用func2。如果运用async和future可以修改如下

Std::future<int> result1(std::async(func1)); //或者auto result1(std::async(func1))

Int result2 = func2();

Int result = result1.get() + result2;

在这里，async尝试将其所获得的函数立刻异步启动于一个分离线程内。基于两个原因，返回future object是必要的:

1、它允许取得“传递给async()的那个函数”的未来结果。这个future object已受到“被启动函数”返回类型的特化，如果启动的是个返回“无物”的后台任务，就会是std::future<void>

2、它必须存在确保“目标函数”或快或慢会被调用。

随着get()被调用，以下三件事情之一会发生

1. 如果func1被async启动于一个分离线程且已结束，立刻获得结果
2. 如果func1被启动但尚未结束，get会引发停滞待func1结束后产生结果
3. 如果func1尚未启动，会强迫启动如同一个同步调用，get会引发停滞直到产生结果

这样的行为很重要，因为确保了在单线程环境中，或是当async无法启动新线程时，程序仍可以有效运作。

**Launch策略**

可以强迫async绝不拖延目标函数的执行，只要明确传入一个launch策略用以指挥之。

Std::future<long> result1 = std::async(std::launch::async,func1);

如果异步调用在此无法实现，程序会抛出异常。有了这个async发射策略，就不必调用get了，因为如果返回的future生命即将结束，这个程序必定会等待func1结束。因此，如果不调用get，当离开future object作用域时，程序会等待后台任务结束。

如果不将std::async(std::launch::async,…)的结果赋值出去，调用者会在此停滞到目标函数结束，那就相当于一个完完全全的同步调用。

与此类似，可以强制延缓执行，以std::launch::defferred为策略传给aysnc()。下面的做法允许你延缓func1()直到对f调用get()

Std：：future<…>f(std::async(std::launch::defferred,func1));

这保证了func1绝对不会再没有get()或wait()的情况下启动。这个策略的特别之处在于允许你写出缓式求值。

**等待和轮询**

一个future只能被get()一次。之后future将处于无效状态，而这种状态只能借由对future调用valid()来检测。此情况下对它的任何调用都会导致不明确行为。

但是future也提供一个接口，允许等待后台操作完成而不处理其结果。这个接口可以被调用一次以上，也可以结合一个时间段或者时间点以限制等待时间。

只要对某个future调用wait(),就可以强制启动该future象征的线程并等待这一后台操作终止。

Std::future<…>f(std::async(func1));

f.wait();

另外还有两个函数，它们并不强制启动线程

1、wait\_for，给予一个时间段，就可以让“异步、运行中”的操作等待一段时间

f.wait\_for(std::chrono::seconds(10))

2、使用wait\_until()就可以等待直到某个特定时间点

f.wait\_until(std::sys\_clock::now()+std::chrono::minutes(1));

无论使用上述何者都可以返回以下三种东西之一:

·std::future\_status::defferred

·std::future\_status::timeout

·std::future\_status::ready

**TIPS:如果使用async(),如果使用async(),就应该以传值的方式来传递所有处理目标函数的必要object，使用async()只需要局部拷贝。如果复制成本太高，就让那些object以常量引用的方式传递，且不使用mutable。**

**13.2 Shared Future**

尽管class std::future提供了处理并发运算之未来结果的能力。但是只能处理该结果一次，否则可能导致不明确行为。

然而，有时候，多次处理并发运算之未来结果是合理的，特别当多个其他线程都想处理这份结果时。基于这个目的，C++标准库提供了class std::shared\_future.于是可以多次调用get(),导致相同结果，或是抛出同一份异常。

有时，为了在声明式中使用auto，可以改而调用成员函数share(),如下

Auto f = async(queryNumber).share();

**13.3 低层接口Thread和Promise**

除了高级接口async()和(shared)future,C++标准库还提供了一个启动及处理线程的低层接口。

**13.3.1 Class std::thread**

欲启动某个线程，只需要先声明一个class std::thread对象，并将任务当作初始实参，然后要么就等待它结束，要么就将它卸离(detach)。

Void doSomething();

…

Std::thread t(doSomething);

…

t.join();

就像对待async()一样，可以传入任何callable object(可以是函数、成员函数、函数对象、lambda)，并且可以夹带任何可能的实参。

此外，这是个低层接口，所以我们可能会感兴趣这一接口和高级的async相比下不提供哪些性质：

·Class thread没有所谓的发射策略，C++标准库永远试着将目标函数启动于一个新线程中。

·没有接口可以处理线程结果。唯一可以获得的是一个独一无二的线程ID。

·如果发生异常，但未捕捉于线程之外，程序会立刻终止并调用std::terminate()

·你必须声明是否“想要等待线程结束”或打算将它“自母体卸离使它运行于后台而不受任何限制”

·如果你让线程运行于后台但是main()结束了，所有线程都会被鲁莽而硬性终止。

**附:join和detach的区别**

（1）当使用join()函数时，主调线程(main函数里有一个主调线程)阻塞，等待被调线程终止，然后主调线程回收被调线程资源，并继续运行；上面这段话的意思就是，使用join(),线程运行完,main函数才能结束。

（2）当使用detach()函数时，主调线程继续运行，被调线程驻留后台运行，主调线程无法再取得该被调线程的控制权。当主调线程结束时，由运行时库负责清理与被调线程相关的资源。上面这段话的意思就是，使用detach(),main函数不用等待线程结束才能结束。有时候线程还没运行完，main函数就已经结束了。

当thread::join()函数被调用后，调用它的线程会被block，直到线程的执行被完成。基本上，这是一种可以用来知道一个线程已结束的机制。当thread::join()返回时，OS的执行的线程已经完成，C++线程对象可以被销毁。

当thread::detach()函数被调用后，执行的线程从线程对象中被分离，已不再被一个线程对象所表达--这是两个独立的事情。C++线程对象可以被销毁，同时OS执行的线程可以继续。如果程序想要知道执行的线程何时结束，就需要一些其它的机制。join()函数在那个thread对象上不能再被调用，因为它已经不再和一个执行的线程相关联。

**当心Detached Thread**

卸离后的线程很容易形成问题—如果它们使用nonlocal资源的话。问题出在你丧失了对于detached thread的控制，没有轻松的办法可以得知它是否运行，以及运行多久。因此，绝对不要让一个detached thread访问任何寿命已经结束的object，基于这个理由，“以传引用方式传递变量和object”给线程，总是带有风险，强烈建议以传值的方式传递。

所以，考虑以下数点作为detached thread的一般性准则。

·Detached thread宁可只访问local copy。

·如果detached thread用上了一个global or static object，应当做以下事情之一

-确保这些对象在“对它们进行访问”之所有detached thread都结束之前不被销毁。

-调用quick\_exit()方式结束程序。

**Thread ID**

所谓thread ID隶属于特殊类型std:thread::id，其值独一无二。唯一允许对thread ID进行的操作是“比较”，以及调用output操作符输出至某个stream。

线程识别的唯一方法是，将线程启动时的ID存储下来，以此为唯一识别值。注意，已结束线程的ID可能会被系统拿去重复使用。

**13.3.2 Promise**

欲传数值给线程，可以仅仅把它们当作实参传递，如果需要线程的运行结果，可以用传引用的方式来传递。

然而，另一个用来传递运行结果和异常的一般性机制是：class std::promise。所谓promise object是future object的配对兄弟，二者都能暂时持有一个shared state。但是future object允许你取回数据，promise object让你提供数据(借由set\_...()函数群中的一个)。

如下：

Std::promise<std::string> p;

Std::thread t(doSomething,std::ref(p));

t.detach();

…

Std::future<std::string> f(p.get\_future());

std::cout<< f.get()<<endl;

在包含<future>(其中有promise的声明)之后，你可以声明一个promise object，令它针对“持有值”或者“返回值”特化。

Promise内部会建立一个shared state,在这里被用来存放一个相应类型值或一个异常，并可被future object取其数据当作线程结束。

一旦shared state存有某值或者某个异常，其状态就转变为ready。于是你可以在它处取出其内容。但是“取出”动作需要借助一个“共享相同shared state”的future object。为此，在main()内对promise object调用get\_future，建立起这以相应的future object。

**13.3.3 Class package\_task<>**

Async()予你一个权柄，使你得以处理task的结果，该task是你尝试立刻启动于后台的。然而有时候虽然你需要处理一个后台task的成果，你其实不需要立刻启动该task。比如，thread pool可以控制何时运行多少个后台task同时运行，此情况下不再这么写:

Double compute(int x, int y);

Std::future<double> f = std::async(compute,7,5);

…

Double res = f.get();

而是写成这样

Double compute(int x, int y);

Std::packaged\_task<double(int,int)> task(compute);

Std::future<double> f = task.get\_future;

…

Task(7,5);

…

Double res = f.get();

其中的task通常启动于某个分离线程中。

**13.4 细说启动线程**

|  |  |  |  |
| --- | --- | --- | --- |
| 启动线程 | 返回值 | 抛出异常 | |
| Std::async | 返回由Std::future<>提供的值或者异常 | | Use  A  Shared  state |
| Std::package\_task | 返回由Std::future<>提供的值或者异常 | |
| Std::thread | 在std::promise<>中设置值或者异常，并且在std::future<>中处理 | |
| Std::thread | 使用shared variables | 抛出异常std::exception\_ptr | |

·低层接口class thread让我们得以启动线程，为了返回数据，我们需要可共享的变量(global或者static变量，或是以实参传递的变量)。为了返回异常，可利用类型std::exception\_ptr

·shared state的概念使得我们能够以一种较便捷的方式处理返回值或者异常。搭配低层接口所提供的promise我们可以建立一个shared state然后通过一个future来处理它

·在高级层面中，class packaged\_task或async()会自动建立一个shared state，它会因一个return语句或一个未被捕获的异常而设置妥

·package\_task允许我们建立一个“带着shared state”的object，但我们必须明确写出何时启动线程

·若是使用std::async()，我们无须关心线程何时真正启动，我们唯一确知的是当需要结果时就调用get()。

**Shared state**

Shared\_state允许“启动以及控制后台机能”的object能够和“处理其结果的”object相互沟通。因此，shared state必须能够持有被启动之目标函数以及某些状态和结果。

Shared state如果持有其函数运行结果，我们说它是ready。Shared state通常被实现为一个引用计数对象，当它被最后一个使用者释放时即被销毁。

**细说async()**

Std::async()是一个辅助函数，用来分离线程中的某个启动函数。因此，如果低层平台支持多线程，你可以让函数并发运转，如果底层平台不支持，也没有任何损失。

然而，async()的真正行为复杂的多，且高度取决于launch策略，后者可作为第一实参。以下便以应用程序开发者的角度来描述async()的三个形式

Future async(std::launch::async,F func,args…)

·尝试启动func并且予以实参args，形成一个异步任务(一个并行线程)

·如果以上办不到就会抛出std::system\_error异常。

·被启动的线程保证在程序结束前完成，除非程序中途失效。

·以下情况会结束线程

-对返回值的future调用get()或者wait()

-如果最后一个指向“返回之future所代表的shared state”的object被销毁

·这意味着对async()的调用会造成停滞直到func完成---如果async的返回值未被使用的话。

Future async(std::launch::deferred,F func,args…)

·形成一个推迟任务，当我们对返回的future调用wait()或者get()，那个推迟任务即被同步调用

·如果未曾调用get或者wait，那么这个推迟任务绝对不会启动

Future async(F func,args…)

·系统会根据当前形式选择一个发射策略，也就是说，如果“立即发射”行不通的话，就使用推迟发射

·也就是说，如果async()可以为func启动一个新线程，就那么做。否则func会被推迟，直到我们对返回的future调用get()或者wait()

·这个调用唯一保证是，一旦我们对返回的future调用get或者wait，func一定会被调用且完成

·如果没有使用get或者wait，func可能永远不调用

·注意，如果无法异步调用func，本形式的async不会抛出system\_error异常

**细说future**

Future用来表现某一个操作的成果:可能是个返回值或是一个异常。这份成果被管理于一个shared state内，后者可被std::async()或者是一个std::package\_task或是一个promise生成出来。这份成果也许尚未存在，因此future持有的也可能是“生成该成果“的每一件必要的东西。

如果future是被async()返回且其相关的task受到推迟，对它调用get()或wait()会同步启动该task。注意，wait\_for()和wait\_until()都不会令一个被推迟的任务启动。

注意，只可调用get一次，因为get()会让future处于无效状态。

面对无效的future，调用其析构函数、move assignment操作符或者valid()以外的任何操作，都会导致不可预期的行为。

Future不提供拷贝构造函数和拷贝赋值操作符，以确保绝对不会有两个object共享某一后台操作之状态。

**细说shared future**

Class shared\_future<>提供的语义接口和class future<>基本相同，但有以下差异

·允许多次调用get

·支持copy语义

·get是个const成员函数，返回一个const reference指向存储于shared state的值.而class std::future的get()是个non-const成员函数，返回一个move-assigned拷贝(如果此性质未被支持则返回一个copy)

·不提供share()

Get的返回值不是拷贝，这一事实造成了若干风险。除了生命周期考虑，还可能形成data race。当作用于相同数据而又相互冲突的动作有着不明确的次序时，就会导致不明确的行为。

**细说class std::promise**

Promise对象用来临时持有一个值或者一个异常。一般而言promise可以持有一个shared state。

值得注意的是，你只能调用get\_future一次，第二次会抛出异常，如果没有相应的shared state也会抛出异常。

所有用来设定数值或者异常的成员函数都是线程安全的，也就是说，犹如有个mutex确保每次只有某一个成员函数可以修改shared state

**细说class std::package\_task**

Class std::package\_task<>被用来同时持有目标函数及其“成果“。你可以拿着相应的目标函数来初始化task，然后通过对task实施operator()来调用该目标函数。最后，可以对此task取一个future以便于处理其成果。

**细说class std::thread**

Thread object和线程之间的关联始于我们将一个callable object指派给它，并夹带有可能的实参。这个关联的结束有两个情况:一是由于join()或是由于detach().不论哪一个函数都必须在thread object生命结束前或是在一个新的thread object被move assignment之前被调用。否则程序会因std::terminate()而中止。

如果thread object被关联至某个线程，它就是所谓的joinable。此情况下joinable()会取得true，调用get\_id()会得到thread ID，其值不同于std::thread::id()所得。

Thread ID有自己的类型std::thread::id。其default构造函数会产出一个独一无二的ID用以表示“非线程“。如果没有任何线程被关联，调用thread::get\_if()获得的便是那个值。如果thread object被关联到了某个线程，调用thread::get\_id()便会获得另一个独一无二的id。

注意，detached thread不应该访问生命已结束的object。这意味着当我们结束程序时，必须确定detached thread不会访问global/static object。

|  |  |
| --- | --- |
| 操作 | 效果 |
| t.join() | 等待关联线程完成工作，然后令此object变成nonjoinable |
| t.detach() | 接触t和线程之间的关联并让线程继续运行 |

**13.5线程同步化与并发问题**

多个线程并发处理相同的数据而又不曾同步化，那么唯一安全的情况是所有线程只读取数据。

不同的对象拥有各自的内存区，这一保证在C++11前并不存在。C++98/03都是只针对单线程而发展的标准。严格说，在C++11前，并行处理不同的对象会导致不可预期的行为。

除非另有说明，C++标准库提供的函数通常不支持“写或读“动作与另一个”写“动作(写至同一笔数据)并发执行。

为解决并发数据的三个主要问题，需要建立以下概念

·Atomicity(不可切割的):这意味着读或者写一个变量，或是一连串语句，其行为是独占的、排他的、无任何打断，因此一个线程不可能读取到“因另一线程而造成“的中间状态。

·order(次序):我们需要一些方法来保证“具体指定之语句次序“

C++标准库提供了许多方法来处理这些概念，让程序在并发方面获得额外保证：

·可以使用future和promise，它们都保证atomicity和order:一定是在形成成果之后才设定shared state，这意味着读和写不会并发发生。

·可以使用mutex和lock

·可以使用condition variable

·可以使用atomic data type确保每次对变量的访问动作都是不可切割的

·可以使用atomic data type的低层接口

**13.6 Mutex和Lock**

Mutex即互斥体，是个object，用来协助采取独占排他方式控制“对资源的并发访问“。这里所谓资源可能是个object，或是多个object的组合。为了获得独占式的资源访问能力，相应的线程必须lock mutex，这样可以防止其他线程lock mutex，直到第一个线程unlock为止。

使用方式如下：

Int val；

Std::mutex valMutex;

此后每次访问这个资源都必须锁定这个mutex以求独占。例如某个线程可能会这么做

valMutex.lock();

if(val>0){

f(val);

}

valMutex.unlock();

而另一个线程也可能这样访问同一资源:

valMutex.lock();

++val;

valMutex.unlock();

有一点很重—但凡可能发生并发存取的地方都应该使用同一个mutex，不论读或者写皆是如此。

这个简单的解决办法可能变得十分复杂:你应该确保异常---它会终止独占-也解除相应的mutex，否则相应的资源可能永远被锁住。此外也可能出现死锁情境—两个线程在释放它们的lock前彼此等待对方的lock。（一种情形，此时执行程序中两个或多个线程发生永久堵塞（等待），每个线程都在等待被其他线程占用并堵塞了的资源。例如，如果线程A锁住了记录1并等待记录2，而线程B锁住了记录2并等待记录1，这样两个线程就发生了死锁现象。）

C++标准库试图处理这些问题(但是目前仍然无法从概念上根本解决)。举个例子，面对异常你不应该自己lock/unlock,而是应当使用RAII守则，以此构造造函将获得资源，而析构函数---甚至当异常造成生命周期结束时它也会被调用---则为我们释放资源。为了这个目的，C++标准库提供了class std::lock\_guard；

Int val;

Std::mutex valMutex;

…

{

Std::lock\_guard<std::mutex> lg(valMutex) //lock and automatically unlock；

If(val >= 0) f(val);

} //在这里确保锁会被自动释放(生命周期结束)

**递归的(recursive) Lock**

有时候递归锁定是必要的，如下:

Std::mutex dbvMutex;

Void createTable(…)

{

Std::lock\_guard<std::mutex> lg(dbvMutex)

}

Void createTableandInsertData(…)

{

Std::lock\_guard<std::mutex> lg(dbMutex);

createTable(…);

}

很显然，在调用后一函数时发生了死锁。因为它锁住了dbMutex后再调用createTable，而createTable发现dbMutex已经被锁住了，于是会阻塞到dbMutex可用，但是这绝对不会发生。

凭借着使用recursive\_mutex，上述行为将不再有问题，这个mutex允许同一线程锁住多次，并且在最近以此相应的unlock时释放lock。  
Std:: recursive\_mutex dbvMutex;

Void createTable(…)

{

Std::lock\_guard<std::recursive\_mutex> lg(dbvMutex)

}

Void createTableandInsertData(…)

{

Std::lock\_guard<std:: recursive\_mutex > lg(dbMutex);

createTable(…);

}

**尝试性的Lock**

有时候想要获得一个lock，但是如果不成功的话它不想永远block，针对这种情况mutex提供了成员函数try\_lock(),它试图取得一个lock，成功就返回true，失败就返回false。

为了能够使用lock\_guard,你可以传一个额外的实参adopt\_lock给其构造函数。

Std::mutex m;

While(m.try\_lock()==false){

Do something;

}

Std::lock\_guard<std::mutex> lg(m,std::adopt\_lock);

…

**处理多个lock**

通常一个线程只处理一个mutex。然而有时候需要锁定多个mutex，这种情况若用先前的lock机制来应付，可能变得复杂且具有风险:你或许取得第一个lock却拿不到第二个lock，或许发生死锁---如果以不同的次序去处理相同的lock。

C++标准库为此提供了若干便捷函数，用以锁定多个mutex，例如:

Std::mutex m1;

Std::mutex m2;

…

{

Std::lock (m1,m2);

Std::lock\_guard<std::mutex> lockM1(m1,std::adopt\_lock);

Std::lock\_guard<std::mutex> lockM2(m2,std::adopt\_lock);

…

}

全局函数std::lock()会锁住它收到的所有mutex，而且阻塞直到所有mutex被锁定或直到抛出异常，如果是后者，所有锁定的mutex都会被解锁。一如既往，成功锁定后你可以并且应该使用adopt\_lock作为第二实参，确保任何情况下这些mutex在离开作用域时会被解锁。

以此相同的方式可以尝试“取得多个lock“且”若并非所有lock都可用也不至于造成阻塞“。全局函数std::try\_lock()会在取得所有lock的情况下返回-1，否则返回第一个失败的lock索引，且如果这样的话成所有成功的lock又会被unlock；

**Class unique\_lock**

除了class::lock\_guard<>,C++标准库还提供了class unique\_lock<>,它对付mutex更有弹性。Class unique\_lock提供的接口和lock\_guard相同，而又允许明确写出何时以及如何锁定或解锁其mutex。因此object可能(也可能不)拥有一个被锁住的mutex。这和lock\_guard不同，后者生命中总是锁定一个mutex。

Unique\_lock可以为一个不一定得锁定得mutex提供一个lock guard。如果它在析构期间仍然锁定mutex，它会调用unlock。但可以明确控制它是否有一个关联得mutex以及是否这个mutex被锁住。

**13.7 细说Mutex和Lock**

C++标准库提供了以下mutex class

·Class std::mutex,同一时间只能被一个线程锁定，如果它被锁住，其他任何lock都会阻塞。

·Class std::recursive\_mutex,允许在同一时间多次被同一线程获得其lock，典型应用是:函数捕获一个lock并调用另一函数而后者再次捕获相同的lock

·Class std::timed\_mutex额外允许你传递一个时间段或者时间点，用来定义其多长时间可以尝试捕捉一个lock

·Class std::recursive\_timed\_mutex 允许同一线程多次取得其lock,可指定期限

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **操作** | **Mutex** | **Recursive\_mutex** | **Timed\_mutex** | **Recursive\_Timed\_mutex** |
| Lock() | 捕获mutex(若无阻塞则阻塞) | | | |
| Try\_lock() | 捕获mutex (若无阻塞则返回false) | | | |
| Unlock() | 解锁被锁定的mutex | | | |
| Try\_lock\_for() | - | - | 试着在时间段内捕获一个lock | |
| Try\_lock\_until() | - | - | 试着捕获一个lock直到某个时间点 | |
| 多个lock | 否 | 是(同一线程) | 否 | 是(同一线程) |

**细说Class lock\_guard**

|  |  |
| --- | --- |
| 操作 | 效果 |
| Lock\_guard lg(m) | 为mutex m建立一个lock guard |
| Lock\_guar lg(m,adopt\_lock) | 为已经锁定的mutex m建立一个lock guard |
| Lg.~lock\_guard() | 解锁mutex并销毁lock guard |

**13.8 只调用一次**

有时候某些机能初次被某个线程使用后，其他线程也不需要它。典型例子是缓式初始化。

单线程的做法很简单，以一个bool flag表示此机能是否已经使用。

但是这样的做法不适合于多线程环境。因为如果两个或者多个线程检查初始化是否尚未发生然后启动初始化，就可能发生data race。你必须针对并发存取保护“检查及初始化”程序区

一如既往，可以调用mutex，单C++标准库为此提供了一个特殊解法---std::once\_flag 以及std::call\_once

Std::once\_flag oc; //全局的flag

…

Std::call\_once(oc,initialize) //如果没有初始化就初始化之

传给call\_once的第一实参必须是相应的once\_flag,下一实参代表可调用对象，还可再加上其他实参给被调用函数使用。

**13.9 Condtion variable**

有时候，被不同线程执行的task必须彼此等待，所以对并发操作实现同步化除了data race之外还有其他原因。

所谓条件变量，是个变量，借由它，一个线程可以唤醒一个或者多个其他等待中的线程。

原则上，条件变量的运作如下:

·你必须同时包含<mutex>和<condition\_variable>并声明一个mutex和一个condition variable

·那个激发“条件终于满足”的线程必须被调用

·那个等待“条件终于被满足”的线程必须被调用

然而，发生wakeup不一定意味着线程所需要的条件以及掌握了。准确地说，wakeup之后仍需要代码去验证“实际条件已达成”。如下：

{

Std::unique\_lock<std::mutex> ul(readyMutex);

readyConVar.wait(ul,[]{return readyFlag});

}

在这里，条件变量的wait成员函数是如此使用的：吧mutex readyMutex的lock u

当作第一实参，把一个lambda当作第二实参，用来二次检测条件是否真的满足。其效果是wait会不断调用该第二实参，直到它返回true，故这段代码实际效果相当于以下

{

Std::unique\_lock<std::mutex> ul(readyMutex);

While(!readyFlag){

readyConVar.wait(ul);

}

}

值得注意的是，这里必须使用unique\_lock而不能使用lock\_guard,因为wait内部会明确对mutex进行解锁和锁定(毕竟lock\_guard只会在析构时才能解锁)。

TIPS:由于可能发生假醒，当条件满足，仅仅通知是不够得，等待者还必须在苏醒之后两次检查该条件。

**13.10 Atomic**

从必要的资源和潜藏的独占访问来看，mutex也许是个相对昂贵的操作，所以也许值得以atomic代替mutex和lock。

Atomic的高层接口所提供的操作将使用默认保证，不论内存的访问次序如何，这个默认保证了顺序一致性。意思是发生在线程之中的atomic操作保证一定像代码出现的次序那样地发生。

欲使用之，应当包含头文件<atomic>

然后使用std::atomic<> class template声明一个atomic object

Std::atomic<bool> readFlag(False);

注意，总是应该将atomic object初始化，因为其默认构造函数并不完全初始化它(并非值不明确，而是lock没有初始化)。

处理atomic的最重要两个操作是

Store()赋予一个新值。

Load()取得当前值。

重点是，这些值都保证是atomic，所以不需要和以前一样“需要mutex的保护才能设置readyFlag”

对于atomic类型，可以继续使用有用而寻常的操作，比如赋值、自动转换为整型、递增、递减等。

然而，为了提供atomic，某些惯常操作可能有些许偏差。比如赋值操作返回的是被赋予值，而不是返回一个reference指向“接受该值”的atomic。

**Atomic的低层接口**

Atomic的低层接口意味着不保证顺序一致性，因此编译器和硬件有可能重排对atomic的处理次序。

因为这类操作经常犯错，即使是专家也得格外小心。。所以，这里。。。略

C++标准库笔记，OVER。 2019.05.07