这是本小人书。原名是《using stl》，不知道是谁写的。不过我倒觉得很有趣，所以化了两个晚上把它翻译出来。我没有对翻译出来的内容校验过。如果你没法在三十分钟内觉得有所收获，那么赶紧扔了它。文中我省略了很多东西。心疼那，浪费我两个晚上。

译者：kary

contact:karymay@163.net

**STL概述**

STL的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但这种分离确实使得STL变得非常通用。例如，由于STL的sort()函数是完全通用的，你可以用它来操作几乎任何数据集合，包括链表，容器和数组。

**要点**

STL算法作为模板函数提供。为了和其他组件相区别，在本书中STL算法以后接一对圆括弧的方式表示，例如sort()。

STL另一个重要特性是它不是面向对象的。为了具有足够通用性，STL主要依赖于模板而不是封装，继承和虚函数（多态性）——OOP的三个要素。你在STL中找不到任何明显的类继承关系。这好像是一种倒退，但这正好是使得STL的组件具有广泛通用性的底层特征。另外，由于STL是基于模板，内联函数的使用使得生成的代码短小高效。

提示

确保在编译使用了STL的程序中至少要使用-O优化来保证内联扩展。STL提供了大量的模板类和函数，可以在OOP和常规编程中使用。所有的STL的大约50个算法都是完全通用的，而且不依赖于任何特定的数据类型。下面的小节说明了三个基本的STL组件：

1）           迭代器提供了访问容器中对象的方法。例如，可以使用一对迭代器指定list或vector中的一定范围的对象。迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。但是，迭代器也可以是那些定义了operator\*()以及其他类似于指针的操作符地方法的类对象。

2）           容器是一种数据结构，如list，vector，和deques ，以模板类的方法提供。为了访问容器中的数据，可以使用由容器类输出的迭代器。

3）           算法是用来操作容器中的数据的模板函数。例如，STL用sort()来对一个vector中的数据进行排序，用find()来搜索一个list中的对象。函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高度复杂容器的任何数据结构上使用。

**头文件**

为了避免和其他头文件冲突， STL的头文件不再使用常规的.h扩展。为了包含标准的string类，迭代器和算法，用下面的指示符：

#include <string>

#include <iterator>

#include <algorithm>

如果你查看STL的头文件，你可以看到象iterator.h和stl\_iterator.h这样的头文件。由于这些名字在各种STL实现之间都可能不同，你应该避免使用这些名字来引用这些头文件。为了确保可移植性，使用相应的没有.h后缀的文件名。表1列出了最常使用的各种容器类的头文件。该表并不完整，对于其他头文件，我将在本章和后面的两章中介绍。

**表 1. STL头文件和容器类**

| **#include** | **Container Class** |
| --- | --- |
| <deque> | deque |
| <list> | list |
| <map> | map, multimap |
| <queue> | queue, priority\_queue |
| <set> | set, multiset |
| <stack> | stack |
| <vector> | vector, vector<bool> |

**名字空间**

你的编译器可能不能识别名字空间。名字空间就好像一个信封，将标志符封装在另一个名字中。标志符只在名字空间中存在，因而避免了和其他标志符冲突。例如，可能有其他库和程序模块定义了sort()函数，为了避免和STL地sort()算法冲突，STL的sort()以及其他标志符都封装在名字空间std中。STL的sort()算法编译为std::sort()，从而避免了名字冲突。

尽管你的编译器可能没有实现名字空间，你仍然可以使用他们。为了使用STL，可以将下面的指示符插入到你的源代码文件中，典型地是在所有的#include指示符的后面：

using namespace std;

**迭代器**

迭代器提供对一个容器中的对象的访问方法，并且定义了容器中对象的范围。迭代器就如同一个指针。事实上，C++的指针也是一种迭代器。但是，迭代器不仅仅是指针，因此你不能认为他们一定具有地址值。例如，一个数组索引，也可以认为是一种迭代器。

迭代器有各种不同的创建方法。程序可能把迭代器作为一个变量创建。一个STL容器类可能为了使用一个特定类型的数据而创建一个迭代器。作为指针，必须能够使用\*操作符类获取数据。你还可以使用其他数学操作符如++。典型的，++操作符用来递增迭代器，以访问容器中的下一个对象。如果迭代器到达了容器中的最后一个元素的后面，则迭代器变成past-the-end值。使用一个past-the-end值得指针来访问对象是非法的，就好像使用NULL或为初始化的指针一样。

提示

STL不保证可以从另一个迭代器来抵达一个迭代器。例如，当对一个集合中的对象排序时，如果你在不同的结构中指定了两个迭代器，第二个迭代器无法从第一个迭代器抵达，此时程序注定要失败。这是STL灵活性的一个代价。*STL不保证检测毫无道理的错误。*

**迭代器的类型**

对于STL数据结构和算法，你可以使用五种迭代器。下面简要说明了这五种类型：

        *Input iterators* 提供对数据的只读访问。

        *Output iterators* 提供对数据的只写访问

        *Forward iterators* 提供读写操作，并能向前推进迭代器。

        *Bidirectional iterators*提供读写操作，并能向前和向后操作。

        *Random access iterators*提供读写操作，并能在数据中随机移动。

尽管各种不同的STL实现细节方面有所不同，还是可以将上面的迭代器想象为一种类继承关系。从这个意义上说，下面的迭代器继承自上面的迭代器。由于这种继承关系，你可以将一个Forward迭代器作为一个output或input迭代器使用。同样，如果一个算法要求是一个bidirectional 迭代器，那么只能使用该种类型和随机访问迭代器。

**指针迭代器**

正如下面的小程序显示的，一个指针也是一种迭代器。该程序同样显示了STL的一个主要特性——它不只是能够用于它自己的类类型，而且也能用于任何C或C++类型。[Listing 1](http://www.csdn.net/editor/Editor.htm#Listing 1), iterdemo.cpp, 显示了如何把指针作为迭代器用于STL的find()算法来搜索普通的数组。

**表 1. iterdemo.cpp**

#include <iostream.h>

#include <algorithm>

using namespace std;

#define SIZE 100

int iarray[SIZE];

int main()

{

iarray[20] = 50;

int\* ip = find(iarray, iarray + SIZE, 50);

if (ip == iarray + SIZE)

cout << "50 not found in array" << endl;

else

cout << \*ip << " found in array" << endl;

return 0;

}

在引用了I/O流库和STL算法头文件（注意没有.h后缀），该程序告诉编译器使用std名字空间。使用std名字空间的这行是可选的，因为可以删除该行对于这么一个小程序来说不会导致名字冲突。

程序中定义了尺寸为SIZE的全局数组。由于是全局变量，所以运行时数组自动初始化为零。下面的语句将在索引20位置处地元素设置为50,并使用find()算法来搜索值50:

iarray[20] = 50;

int\* ip = find(iarray, iarray + SIZE, 50);

find()函数接受三个参数。头两个定义了搜索的范围。由于C和C++数组等同于指针，表达式iarray指向数组的第一个元素。而第二个参数iarray + SIZE等同于*past-the-end* 值，也就是数组中最后一个元素的后面位置。第三个参数是待定位的值，也就是50。find()函数返回和前两个参数相同类型的迭代器，这儿是一个指向整数的指针ip。

**提示**

必须记住STL使用模板。因此，STL函数自动根据它们使用的数据类型来构造。

为了判断find()是否成功，例子中测试ip和 *past-the-end* 值是否相等：

if (ip == iarray + SIZE) ...

如果表达式为真，则表示在搜索的范围内没有指定的值。否则就是指向一个合法对象的指针，这时可以用下面的语句显示：:

cout << \*ip << " found in array" << endl;

测试函数返回值和NULL是否相等是不正确的。不要象下面这样使用：

int\* ip = find(iarray, iarray + SIZE, 50);

if (ip != NULL) ... // ??? incorrect

当使用STL函数时，只能测试ip是否和*past-the-end* 值是否相等。尽管在本例中ip是一个C++指针,其用法也必须符合STL迭代器的规则。

**容器迭代器**

尽管C++指针也是迭代器，但用的更多的是容器迭代器。容器迭代器用法和iterdemo.cpp一样，但和将迭代器申明为指针变量不同的是，你可以使用容器类方法来获取迭代器对象。两个典型的容器类方法是begin()和end()。它们在大多数容器中表示整个容器范围。其他一些容器还使用rbegin()和rend()方法提供反向迭代器，以按反向顺序指定对象范围。

下面的程序创建了一个矢量容器（STL的和数组等价的对象），并使用迭代器在其中搜索。该程序和前一章中的程序相同。

**Listing 2. vectdemo.cpp**

#include <iostream.h>

#include <algorithm>

#include <vector>

using namespace std;

vector<int> intVector(100);

void main()

{

intVector[20] = 50;

vector<int>::iterator intIter =

find(intVector.begin(), intVector.end(), 50);

if (intIter != intVector.end())

cout << "Vector contains value " << \*intIter << endl;

else

cout << "Vector does not contain 50" << endl;

}

注意用下面的方法显示搜索到的数据：

cout << "Vector contains value " << \*intIter << endl;

**常量迭代器**

和指针一样，你可以给一个迭代器赋值。例如，首先申明一个迭代器：

vector<int>::iterator first;

该语句创建了一个vector<int>类的迭代器。下面的语句将该迭代器设置到intVector的第一个对象，并将它指向的对象值设置为123：:

first = intVector.begin();

\*first = 123;

这种赋值对于大多数容器类都是允许的，除了只读变量。为了防止错误赋值，可以申明迭代器为：

const vector<int>::iterator result;

result = find(intVector.begin(), intVector.end(), value);

if (result != intVector.end())

\*result = 123; // ???

**警告**

另一种防止数据被改变得方法是将容器申明为const类型。

『呀！在VC中测试出错,正确的含义是result成为常量而不是它指向的对象不允许改变，如同int \*const p;看来这作者自己也不懂』

**使用迭代器编程**

你已经见到了迭代器的一些例子，现在我们将关注每种特定的迭代器如何使用。由于使用迭代器需要关于STL容器类和算法的知识，在阅读了后面的两章后你可能需要重新复习一下本章内容。

**输入迭代器**

输入迭代器是最普通的类型。输入迭代器至少能够使用==和!=测试是否相等；使用\*来访问数据；使用++操作来递推迭代器到下一个元素或到达*past-the-end* 值。

为了理解迭代器和STL函数是如何使用它们的，现在来看一下find()模板函数的定义：

template <class InputIterator, class T>

InputIterator find(

InputIterator first, InputIterator last, const T& value) {

while (first != last && \*first != value) ++first;

return first;

}

**注意**

在find()算法中，注意如果first和last指向不同的容器，该算法可能陷入死循环。

**输出迭代器**

输出迭代器缺省只写，通常用于将数据从一个位置拷贝到另一个位置。由于输出迭代器无法读取对象，因此你不会在任何搜索和其他算法中使用它。要想读取一个拷贝的值，必须使用另一个输入迭代器（或它的继承迭代器）。

**Listing 3. outiter.cpp**

#include <iostream.h>

#include <algorithm> // Need copy()

#include <vector> // Need vector

using namespace std;

double darray[10] =

{1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9};

vector<double> vdouble(10);

int main()

{

vector<double>::iterator outputIterator = vdouble.begin();

copy(darray, darray + 10, outputIterator);

while (outputIterator != vdouble.end()) {

cout << \*outputIterator << endl;

outputIterator++;

}

return 0;

}

**注意**

当使用copy()算法的时候，你必须确保目标容器有足够大的空间，或者容器本身是自动扩展的。

**前推迭代器**

前推迭代器能够读写数据值，并能够向前推进到下一个值。但是没法递减。replace()算法显示了前推迭代器的使用方法。

template <class ForwardIterator, class T>

void replace (ForwardIterator first,

ForwardIterator last,

const T& old\_value,

const T& new\_value);

使用replace()将[first,last]范围内的所有值为old\_value的对象替换为new\_value。*:*

replace(vdouble.begin(), vdouble.end(), 1.5, 3.14159);

**双向迭代器**

双向迭代器要求能够增减。如reverse()算法要求两个双向迭代器作为参数:

template <class BidirectionalIterator>

void reverse (BidirectionalIterator first,

BidirectionalIterator last);

使用reverse()函数来对容器进行逆向排序:

reverse(vdouble.begin(), vdouble.end());

**随机访问迭代器**

随机访问迭代器能够以任意顺序访问数据，并能用于读写数据（不是const的C++指针也是随机访问迭代器）。STL的排序和搜索函数使用随机访问迭代器。随机访问迭代器可以使用关系操作符作比较。

random\_shuffle() 函数随机打乱原先的顺序。申明为：

template <class RandomAccessIterator>

void random\_shuffle (RandomAccessIterator first,

RandomAccessIterator last);

使用方法：

random\_shuffle(vdouble.begin(), vdouble.end());

**迭代器技术**

要学会使用迭代器和容器以及算法，需要学习下面的新技术。

**流和迭代器**

本书的很多例子程序使用I/O流语句来读写数据。例如：

int value;

cout << "Enter value: ";

cin >> value;

cout << "You entered " << value << endl;

对于迭代器，有另一种方法使用流和标准函数。理解的要点是将输入/输出流作为容器看待。因此，任何接受迭代器参数的算法都可以和流一起工作。

**Listing 4. outstrm.cpp**

#include <iostream.h>

#include <stdlib.h> // Need random(), srandom()

#include <time.h> // Need time()

#include <algorithm> // Need sort(), copy()

#include <vector> // Need vector

using namespace std;

void Display(vector<int>& v, const char\* s);

int main()

{

// Seed the random number generator

srandom( time(NULL) );

// Construct vector and fill with random integer values

vector<int> collection(10);

for (int i = 0; i < 10; i++)

collection[i] = random() % 10000;;

// Display, sort, and redisplay

Display(collection, "Before sorting");

sort(collection.begin(), collection.end());

Display(collection, "After sorting");

return 0;

}

// Display label s and contents of integer vector v

void Display(vector<int>& v, const char\* s)

{

cout << endl << s << endl;

copy(v.begin(), v.end(),

ostream\_iterator<int>(cout, "\t"));

cout << endl;

}

函数Display()显示了如何使用一个输出流迭代器。下面的语句将容器中的值传输到cout输出流对象中:

copy(v.begin(), v.end(),

ostream\_iterator<int>(cout, "\t"));

第三个参数实例化了ostream\_iterator<int>类型，并将它作为copy()函数的输出目标迭代器对象。“\t”字符串是作为分隔符。运行结果：

$ g++ outstrm.cpp

$ ./a.out

Before sorting

677 722 686 238 964 397 251 118 11 312

After sorting

11 118 238 251 312 397 677 686 722 964

这是STL神奇的一面『确实神奇』。为定义输出流迭代器，STL提供了模板类ostream\_iterator。这个类的构造函数有两个参数：一个ostream对象和一个string值。因此可以象下面一样简单地创建一个迭代器对象：

ostream\_iterator<int>(cout, "\n")

该迭代起可以和任何接受一个输出迭代器的函数一起使用。

**插入迭代器**

插入迭代器用于将值插入到容器中。它们也叫做适配器，因为它们将容器适配或转化为一个迭代器，并用于copy()这样的算法中。例如，一个程序定义了一个链表和一个矢量容器:

list<double> dList;

vector<double> dVector;

通过使用front\_inserter迭代器对象，可以只用单个copy()语句就完成将矢量中的对象插入到链表前端的操作：

copy(dVector.begin(), dVector.end(), front\_inserter(dList));

三种插入迭代器如下：

        *普通插入器* 将对象插入到容器任何对象的前面。

        *Front inserters* 将对象插入到数据集的前面——例如，链表表头。

        *Back inserters* 将对象插入到集合的尾部——例如，矢量的尾部，导致矢量容器扩展。

使用插入迭代器可能导致容器中的其他对象移动位置，因而使得现存的迭代器非法。例如，将一个对象插入到矢量容器将导致其他值移动位置以腾出空间。一般来说，插入到象链表这样的结构中更为有效，因为它们不会导致其他对象移动。

**Listing 5. insert.cpp**

#include <iostream.h>

#include <algorithm>

#include <list>

using namespace std;

int iArray[5] = { 1, 2, 3, 4, 5 };

void Display(list<int>& v, const char\* s);

int main()

{

list<int> iList;

// Copy iArray backwards into iList

copy(iArray, iArray + 5, front\_inserter(iList));

Display(iList, "Before find and copy");

// Locate value 3 in iList

list<int>::iterator p =

find(iList.begin(), iList.end(), 3);

// Copy first two iArray values to iList ahead of p

copy(iArray, iArray + 2, inserter(iList, p));

Display(iList, "After find and copy");

return 0;

}

void Display(list<int>& a, const char\* s)

{

cout << s << endl;

copy(a.begin(), a.end(),

ostream\_iterator<int>(cout, " "));

cout << endl;

}

运行结果如下：

$ g++ insert.cpp

$ ./a.out

Before find and copy

5 4 3 2 1

After find and copy

5 4 1 2 3 2 1

可以将front\_inserter替换为back\_inserter试试。

如果用find()去查找在列表中不存在的值，例如99。由于这时将p设置为*past-the-end* 值。最后的copy()函数将iArray的值附加到链表的后部。

**混合迭代器函数**

在涉及到容器和算法的操作中，还有两个迭代器函数非常有用：

        advance() 按指定的数目增减迭代器。

        distance() 返回到达一个迭代器所需（递增）操作的数目。

例如：

list<int> iList;

list<int>::iterator p =

find(iList.begin(), iList.end(), 2);

cout << "before: p == " << \*p << endl;

advance(p, 2); // same as p = p + 2;

cout << "after : p == " << \*p << endl;

int k = 0;

distance(p, iList.end(), k);

cout << "k == " << k << endl;

advance()函数接受两个参数。第二个参数是向前推进的数目。对于前推迭代器，该值必须为正，而对于双向迭代器和随机访问迭代器，该值可以为负。

使用 distance()函数来返回到达另一个迭代器所需要的步骤。

**注意**

distance()函数是迭代的，也就是说，它递增第三个参数。因此，你必须初始化该参数。未初始化该参数几乎注定要失败。

**函数和函数对象**

STL中，函数被称为算法，也就是说它们和标准C库函数相比，它们更为通用。STL算法通过重载operator()函数实现为模板类或模板函数。这些类用于创建函数对象，对容器中的数据进行各种各样的操作。下面的几节解释如何使用函数和函数对象。

**函数和断言**

经常需要对容器中的数据进行用户自定义的操作。例如，你可能希望遍历一个容器中所有对象的STL算法能够回调自己的函数。例如

#include <iostream.h>

#include <stdlib.h> // Need random(), srandom()

#include <time.h> // Need time()

#include <vector> // Need vector

#include <algorithm> // Need for\_each()

#define VSIZE 24 // Size of vector

vector<long> v(VSIZE); // Vector object

// Function prototypes

void initialize(long &ri);

void show(const long &ri);

bool isMinus(const long &ri); // Predicate function

int main()

{

srandom( time(NULL) ); // Seed random generator

for\_each(v.begin(), v.end(), initialize);//调用普通函数

cout << "Vector of signed long integers" << endl;

for\_each(v.begin(), v.end(), show);

cout << endl;

// Use predicate function to count negative values

//

int count = 0;

vector<long>::iterator p;

p = find\_if(v.begin(), v.end(), isMinus);//调用断言函数

while (p != v.end()) {

count++;

p = find\_if(p + 1, v.end(), isMinus);

}

cout << "Number of values: " << VSIZE << endl;

cout << "Negative values : " << count << endl;

return 0;

}

// Set ri to a signed integer value

void initialize(long &ri)

{

ri = ( random() - (RAND\_MAX / 2) );

// ri = random();

}

// Display value of ri

void show(const long &ri)

{

cout << ri << " ";

}

// Returns true if ri is less than 0

bool isMinus(const long &ri)

{

return (ri < 0);

}

所谓断言函数，就是返回bool值的函数。

**函数对象**

除了给STL算法传递一个回调函数，你还可能需要传递一个类对象以便执行更复杂的操作。这样的一个对象就叫做函数对象。实际上函数对象就是一个类，但它和回调函数一样可以被回调。例如，在函数对象每次被for\_each()或find\_if()函数调用时可以保留统计信息。函数对象是通过重载operator()()实现的。如果TanyClass定义了opeator()(),那么就可以这么使用：

TAnyClass object; // Construct object

object(); // Calls TAnyClass::operator()() function

for\_each(v.begin(), v.end(), object);

STL定义了几个函数对象。由于它们是模板，所以能够用于任何类型，包括C/C++固有的数据类型，如long。有些函数对象从名字中就可以看出它的用途，如plus()和multiplies()。类似的greater()和less-equal()用于比较两个值。

**注意**

有些版本的ANSI C++定义了times()函数对象，而GNU C++把它命名为multiplies()。使用时必须包含头文件<functional>。

一个有用的函数对象的应用是accumulate() 算法。该函数计算容器中所有值的总和。记住这样的值不一定是简单的类型，通过重载operator+()，也可以是类对象。

**Listing 8. accum.cpp**

#include <iostream.h>

#include <numeric> // Need accumulate()

#include <vector> // Need vector

#include <functional> // Need multiplies() (or times())

#define MAX 10

vector<long> v(MAX); // Vector object

int main()

{

// Fill vector using conventional loop

//

for (int i = 0; i < MAX; i++)

v[i] = i + 1;

// Accumulate the sum of contained values

//

long sum =

accumulate(v.begin(), v.end(), 0);

cout << "Sum of values == " << sum << endl;

// Accumulate the product of contained values

//

long product =

accumulate(v.begin(), v.end(), 1, multiplies<long>());//注意这行

cout << "Product of values == " << product << endl;

return 0;

}

编译输出如下：

$ g++ accum.cpp

$ ./a.out

Sum of values == 55

Product of values == 3628800

『注意使用了函数对象的accumulate()的用法。accumulate() 在内部将每个容器中的对象和第三个参数作为multiplies函数对象的参数,multiplies(1,v)计算乘积。VC中的这些模板的源代码如下：

// TEMPLATE FUNCTION accumulate

template<class \_II, class \_Ty> inline

\_Ty accumulate(\_II \_F, \_II \_L, \_Ty \_V)

{for (; \_F != \_L; ++\_F)

\_V = \_V + \*\_F;

return (\_V); }

// TEMPLATE FUNCTION accumulate WITH BINOP

template<class \_II, class \_Ty, class \_Bop> inline

\_Ty accumulate(\_II \_F, \_II \_L, \_Ty \_V, \_Bop \_B)

{for (; \_F != \_L; ++\_F)

\_V = \_B(\_V, \*\_F);

return (\_V); }

// TEMPLATE STRUCT binary\_function

template<class \_A1, class \_A2, class \_R>

struct binary\_function {

typedef \_A1 first\_argument\_type;

typedef \_A2 second\_argument\_type;

typedef \_R result\_type;

};

// TEMPLATE STRUCT multiplies

template<class \_Ty>

struct multiplies : binary\_function<\_Ty, \_Ty, \_Ty> {

\_Ty operator()(const \_Ty& \_X, const \_Ty& \_Y) const

{return (\_X \* \_Y); }

};

引言：如果你想深入了解STL到底是怎么实现的，最好的办法是写个简单的程序，将程序中涉及到的模板源码给copy下来，稍作整理，就能看懂了。所以没有必要去买什么《STL源码剖析》之类的书籍，那些书可能反而浪费时间。』

**发生器函数对象**

有一类有用的函数对象是“发生器”(generator)。这类函数有自己的内存，也就是说它能够从先前的调用中记住一个值。例如随机数发生器函数。

普通的C程序员使用静态或全局变量 “记忆”上次调用的结果。但这样做的缺点是该函数无法和它的数据相分离『还有个缺点是要用TLS才能线程安全』。显然，使用类来封装一块：“内存”更安全可靠。先看一下例子：

**Listing 9. randfunc.cpp**

#include <iostream.h>

#include <stdlib.h> // Need random(), srandom()

#include <time.h> // Need time()

#include <algorithm> // Need random\_shuffle()

#include <vector> // Need vector

#include <functional> // Need ptr\_fun()

using namespace std;

// Data to randomize

int iarray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

vector<int> v(iarray, iarray + 10);

// Function prototypes

void Display(vector<int>& vr, const char \*s);

unsigned int RandInt(const unsigned int n);

int main()

{

srandom( time(NULL) ); // Seed random generator

Display(v, "Before shuffle:");

pointer\_to\_unary\_function<unsigned int, unsigned int>

ptr\_RandInt = ptr\_fun(RandInt); // Pointer to RandInt()//注意这行

random\_shuffle(v.begin(), v.end(), ptr\_RandInt);

Display(v, "After shuffle:");

return 0;

}

// Display contents of vector vr

void Display(vector<int>& vr, const char \*s)

{

cout << endl << s << endl;

copy(vr.begin(), vr.end(), ostream\_iterator<int>(cout, " "));

cout << endl;

}

// Return next random value in sequence modulo n

unsigned int RandInt(const unsigned int n)

{

return random() % n;

}

编译运行结果如下：

$ g++ randfunc.cpp

$ ./a.out

Before shuffle:

1 2 3 4 5 6 7 8 9 10

After shuffle:

6 7 2 8 3 5 10 1 9 4

首先用下面的语句申明一个对象：

pointer\_to\_unary\_function<unsigned int, unsigned int>

ptr\_RandInt = ptr\_fun(RandInt);

这儿使用STL的单目函数模板定义了一个变量ptr\_RandInt，并将地址初始化到我们的函数RandInt()。单目函数接受一个参数，并返回一个值。现在random\_shuffle()可以如下调用：

random\_shuffle(v.begin(), v.end(), ptr\_RandInt);

在本例子中，发生器只是简单的调用rand()函数。

**关于常量引用的一点小麻烦（不翻译了，VC下将例子中的const去掉）**

**发生器函数类对象**

下面的例子说明发生器函数类对象的使用。

**Listing 10. fiborand.cpp**

#include <iostream.h>

#include <algorithm> // Need random\_shuffle()

#include <vector> // Need vector

#include <functional> // Need unary\_function

using namespace std;

// Data to randomize

int iarray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

vector<int> v(iarray, iarray + 10);

// Function prototype

void Display(vector<int>& vr, const char \*s);

// The FiboRand template function-object class

template <class Arg>

class FiboRand : public unary\_function<Arg, Arg> {

int i, j;

Arg sequence[18];

public:

FiboRand();

Arg operator()(const Arg& arg);

};

void main()

{

FiboRand<int> fibogen; // Construct generator object

cout << "Fibonacci random number generator" << endl;

cout << "using random\_shuffle and a function object" << endl;

Display(v, "Before shuffle:");

random\_shuffle(v.begin(), v.end(), fibogen);

Display(v, "After shuffle:");

}

// Display contents of vector vr

void Display(vector<int>& vr, const char \*s)

{

cout << endl << s << endl;

copy(vr.begin(), vr.end(),

ostream\_iterator<int>(cout, " "));

cout << endl;

}

// FiboRand class constructor

template<class Arg>

FiboRand<Arg>::FiboRand()

{

sequence[17] = 1;

sequence[16] = 2;

for (int n = 15; n > 0; n—)

sequence[n] = sequence[n + 1] + sequence[n + 2];

i = 17;

j = 5;

}

// FiboRand class function operator

template<class Arg>

Arg FiboRand<Arg>::operator()(const Arg& arg)

{

Arg k = sequence[i] + sequence[j];

sequence[i] = k;

i--;

j--;

if (i == 0) i = 17;

if (j == 0) j = 17;

return k % arg;

}

编译运行输出如下:

$ g++ fiborand.cpp

$ ./a.out

Fibonacci random number generator

using random\_shuffle and a function object

Before shuffle:

1 2 3 4 5 6 7 8 9 10

After shuffle:

6 8 5 4 3 7 10 1 9

该程序用完全不通的方法使用使用rand\_shuffle。Fibonacci 发生器封装在一个类中，该类能从先前的“使用”中记忆运行结果。在本例中，类FiboRand 维护了一个数组和两个索引变量I和j。

FiboRand类继承自unary\_function() 模板:

template <class Arg>

class FiboRand : public unary\_function<Arg, Arg> {...

Arg是用户自定义数据类型。该类还定以了两个成员函数，一个是构造函数，另一个是operator()（）函数，该操作符允许random\_shuffle()算法象一个函数一样“调用”一个FiboRand对象。

**绑定器函数对象**

一个绑定器使用另一个函数对象f()和参数值V创建一个函数对象。被绑定函数对象必须为双目函数，也就是说有两个参数,A和B。STL 中的帮定器有：

        bind1st() 创建一个函数对象，该函数对象将值V作为第一个参数A。

        bind2nd()创建一个函数对象，该函数对象将值V作为第二个参数B。

举例如下：

**Listing 11. binder.cpp**

#include <iostream.h>

#include <algorithm>

#include <functional>

#include <list>

using namespace std;

// Data

int iarray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

list<int> aList(iarray, iarray + 10);

int main()

{

int k = 0;

count\_if(aList.begin(), aList.end(),

bind1st(greater<int>(), 8), k);

cout << "Number elements < 8 == " << k << endl;

return 0;

}

Algorithm count\_if()计算满足特定条件的元素的数目。 这是通过将一个函数对象和一个参数捆绑到为一个对象，并将该对象作为算法的第三个参数实现的。 注意这个表达式:

bind1st(greater<int>(), 8)

该表达式将greater<int>()和一个参数值8捆绑为一个函数对象。由于使用了bind1st()，所以该函数相当于计算下述表达式：

8 > q

表达式中的q是容器中的对象。因此，完整的表达式

count\_if(aList.begin(), aList.end(),

bind1st(greater<int>(), 8), k);

计算所有小于或等于8的对象的数目。

**否定函数对象**

所谓否定(negator)函数对象，就是它从另一个函数对象创建而来，如果原先的函数返回真，则否定函数对象返回假。有两个否定函数对象：not1()和not2()。not1()接受单目函数对象，not2()接受双目函数对象。否定函数对象通常和帮定器一起使用。例如，上节中用bind1nd来搜索q<=8的值：

count\_if(aList.begin(), aList.end(),

bind1st(greater<int>(), 8), k);

如果要搜索q>8的对象，则用bind2st。而现在可以这样写：

start = find\_if(aList.begin(), aList.end(),

not1(bind1nd(greater<int>(), 6)));

你必须使用not1，因为bind1nd返回单目函数。

**总结：使用标准模板库 (STL)**

尽管很多程序员仍然在使用标准C函数，但是这就好像骑着毛驴寻找Mercedes一样。你当然最终也会到达目标，但是你浪费了很多时间。

尽管有时候使用标准C函数确实方便(如使用sprintf()进行格式化输出)。但是C函数不使用异常机制来报告错误，也不适合处理新的数据类型。而且标准C函数经常使用内存分配技术，没有经验的程序员很容易写出bug来。.

C++标准库则提供了更为安全，更为灵活的数据集处理方式。STL最初由HP实验室的Alexander Stepanov和Meng Lee开发。最近，C++标准委员会采纳了STL，尽管在不同的实现之间仍有细节差别。

STL的最主要的两个特点：数据结构和算法的分离，非面向对象本质。访问对象是通过象指针一样的迭代器实现的；容器是象链表，矢量之类的数据结构，并按模板方式提供；算法是函数模板，用于操作容器中的数据。由于STL以模板为基础，所以能用于任何数据类型和结构。