**STL简介**

1 概况.......................................................... 2

1.1 STL是什么............................................... 2

1.2 为什么我们需要学习STL................................... 2

1.3 初识STL................................................. 2

1.4 STL 的组成.............................................. 5

2 容器.......................................................... 6

2.1 基本容器——向量（vector）.............................. 6

2.2  双端队列（deque容器类）............................... 9

2.3 表（List容器类）....................................... 10

2.4 集和多集（set 和multiset 容器类）：.................... 12

2.5 映射和多重映射（map 和multimap）....................... 13

3 算法（algorithm）：.......................................... 15

3.1 翻转和复制（reverse()和copy()）........................ 15

3.2 单值交换（Swap()）..................................... 16

3.3 查找（find()）......................................... 17

3.4 得到数目（Count()）.................................... 18

3.5 排序（sort()）......................................... 19

4 迭代器（itertor）............................................ 21

5 STL的其他标准组件........................................... 22

5.1 函数对象（functor或者funtion objects）................ 22

5.2 适配器（adapter）...................................... 23

**1 概况**

**1.1      STL是什么**

作为一个C++程序设计者，STL是一种不可忽视的技术。

Standard Template Library (STL)：标准模板库,更准确的说是 C++ 程序设计语言标准模板库。STL是所有C++编译器和所有操作系统平台都支持的一种库，说它是一种库是因为，虽然STL是一种标准，也就是说对所有的编译器来说，提供给C++程序设计者的接口都是一样的。也就是说同一段STL代码在不同编译器和操作系统平台上运行的结果都是相同的，但是底层实现可以是不同的。 令人兴奋的是，STL的使用者并不需要了解它的底层实现。 试想一下，如果我们有一把能打开所有锁的钥匙，那将是多么令人疯狂啊。

STL的目的是标准化组件，这样你就不用重新开发它们了。你可以仅仅使用这些现成的组件。STL现在是C++的一部分，因此不用额外安装什么。它被内建在你的编译器之内。

**1.2      为什么我们需要学习STL**

STL是 C++的ANSI/ISO 标准的一部分,可以用于所有C++语言编译器和所有平台(Windows/Unix/Linux..)。STL的同一版本在任意硬件配置下都是可用的；

STL 提供了大量的可复用软件组织。例如，程序员再也不用自己设计排序，搜索算法了，这些都已经是STL的一部分了。嘎嘎，有意思吧。

使用STL 的应用程序保证了得到的实现在处理速度和内存利用方面都是高效的，因为STL设计者们已经为我们考虑好了。

使用STL编写的代码更容易修改和阅读，这是当然的啦。因为代码更短了，很多基础工作代码已经被组件化了；

使用简单，虽然内部实现很复杂。

虽然，STL的优点甚多，但是STL的语法实在令初学者人头疼，许多人望而却步。可是STL是每个C++程序设计者迟早都要啃的一块骨头。

**1.3      初识STL**

下面让我们来看几段代码吧：

#include <iostream>

int main(void)

{

    double a[] = {1, 2, 3, 4, 5};

    std::cout<<mean(a, 5)<<std::endl;    // will print 3

    return 0;

}

好懂吧，除了那个std有点让人不舒服以外，这是一段普通的没有使用STL的C++代码。

再看下面一段：

#include <vector>

#include <iostream>

int main()

{

    std::vector<double> a;

    a.push\_back(1);

    a.push\_back(2);

    a.push\_back(3);

    a.push\_back(4);

    a.push\_back(5);

    for(int i = 0; i < a.size(); ++i)

{

        std::cout<<a[i]<<std::endl;

    }

    return 0;

}

如果你真的没有接触过STL的话，你会问，呀，vector 是啥呀？这是一段纯种的STL代码，看到尖括号了吧，知道那是模板了吧。看到a.push\_back(5)、a.size()你不感觉奇怪么？可是我们并没有定义这些函数啊。

#include <vector>

#include <iostream>

int main()

{

    std::vector< int > q;

    q.push\_back(10);

    q.push\_back(11);

    q.push\_back(12);

    std::vector< int > v;

    for(int i=0; i<5; ++i){

        v.push\_back(i);

    }

    std::vector<int>::iterator it = v.begin() + 1;

    it = v.insert(it, 33);

    v.insert(it, q.begin(), q.end());

    it = v.begin() + 3;

    v.insert(it, 3, -1);

    it = v.begin() + 4;

    v.erase(it);

    it = v.begin() + 1;

    v.erase(it, it + 4);

    v.clear();

    return 0;

}

这一段你又看到了新东西了吧：iterator、insert、erase、clear。不罗嗦了，等你看完这篇文章，回头再看就简单了。

关于模板的其他细节，读者可以参阅《C++ Templates 中文版》在这里，简单的介绍一下模板类和函数模板的概念。

模板是C++中实现代码重用机制的一种工具，可以实现类型参数化，把类型定义为参数。函数模板和类模板允许用户构造模板函数和模板类。

下面我们来看一段函数模板的例子：

#include<iostream>

#include<string>

using namespace std;

//定义函数模板

template<class T>   //template 是关键字，T 表示一种待实例化的类型

                    //template<typename T>  也是对的

T MAX(T a, T b)//函数模板，函数名为 max，此函数有2个T类型的参数，返回类型为T

{

  return (a>b)?a:b;

}

//在此例实例化的时候，T可以是多种类型的，int,char,string…

int main()

{

int x=2,y=6;

    double x1=9.123,y1=12.6543;

    cout<<"把T实例化为int:"<<MAX(x,y)<<endl;//实例化函数模板，把T实例化为int

    cout<<"把T实例化为double:"<<MAX(x1,y1)<<endl;  //把T实例化为double

}

下面再看看，类模板：

#include<iostream>

using namespace std;

//定义名为ex\_class的类模板

template < typename T>  class ex\_class

{

    T value;

public:

    ex\_class(T v) { value=v; }

    void set\_value(T v) { value=v; }

    T get\_value(void) {return value;}

};

//main()函数中测试ex\_class类模板

int main()

{

    //测试int类型数据

    ex\_class <int> a(5),b(10);

    cout<<"a.value:"<<a.get\_value()<<endl;

    cout<<"b.value:"<<b.get\_value()<<endl;

    //测试char类型数据

    ex\_class <char> ch('A');

    cout<<"ch.value:"<<ch.get\_value()<<endl;

    ch.set\_value('a');

    cout<<"ch.value:"<<ch.get\_value()<<endl;

    //测试double类型数据

    ex\_class <double> x(5.5);

    cout<<"x.value:"<<x.get\_value()<<endl;

    x.set\_value(7.5);

    cout<<"x.value:"<<x.get\_value()<<endl;

}

**1.4      STL 的组成**

STL有三大核心部分：容器（Container）、算法（Algorithms）、迭代器（Iterator），容器适配器（container adaptor），函数对象(functor)，除此之外还有STL其他标准组件。通俗的讲：

容器：装东西的东西，装水的杯子，装咸水的大海，装人的教室……STL里的容器是可容纳一些数据的模板类。

算法：就是往杯子里倒水，往大海里排污，从教室里撵人……STL里的算法，就是处理容器里面数据的方法、操作。

迭代器：往杯子里倒水的水壶，排污的管道，撵人的那个物业管理人员……STL里的迭代器：遍历容器中数据的对象。对存储于容器中的数据进行处理时，迭代器能从一个成员移向另一个成员。他能按预先定义的顺序在某些容器中的成员间移动。对普通的一维数组、向量、双端队列和列表来说，迭代器是一种指针。

下面让我们来看看专家是怎么说的：

容器（container）：容器是数据在内存中组织的方法，例如，数组、堆栈、队列、链表或二叉树（不过这些都不是STL标准容器）。STL中的容器是一种存储T（Template）类型值的有限集合的数据结构,容器的内部实现一般是类。这些值可以是对象本身，如果数据类型T代表的是Class的话。

算法（algorithm）：算法是应用在容器上以各种方法处理其内容的行为或功能。例如，有对容器内容排序、复制、检索和合并的算法。在STL中，算法是由模板函数表现的。这些函数不是容器类的成员函数。相反，它们是独立的函数。令人吃惊的特点之一就是其算法如此通用。不仅可以将其用于STL容器，而且可以用于普通的C＋＋数组或任何其他应用程序指定的容器。

迭代器(iterator)：一旦选定一种容器类型和数据行为(算法)，那么剩下唯一要他做的就是用迭代器使其相互作用。可以把达代器看作一个指向容器中元素的普通指针。可以如递增一个指针那样递增迭代器，使其依次指向容器中每一个后继的元素。迭代器是STL的一个关键部分，因为它将算法和容器连在一起。

下面我将依次介绍STL的这三个主要组件。

**2 容器**

STL中的容器有队列容器和关联容器，容器适配器（congtainer adapters：stack,queue，priority queue），位集（bit\_set），串包(string\_package)等等。  
　　在本文中，我将介绍list,vector，deque等队列容器，和set和multisets,map和multimaps等关联容器，一共7种基本容器类。  
　　队列容器（顺序容器）：队列容器按照线性排列来存储T类型值的集合，队列的每个成员都有自己的特有的位置。顺序容器有向量类型、双端队列类型、列表类型三种。

**2.1 基本容器——向量（vector）**

向量（vector容器类）：＃include <vector>，vector是一种动态数组，是基本数组的类模板。其内部定义了很多基本操作。既然这是一个类，那么它就会有自己的构造函数。vector 类中定义了4中种构造函数：

* 默认构造函数，构造一个初始长度为0的空向量，如：vector<int> v1;
* 带有单个整形参数的构造函数，此参数描述了向量的初始大小。这个构造函数还有一个可选的参数，这是一个类型为T的实例，描述了各个向量种各成员的初始值；如：vector<int> v2(n,0); 如果预先定义了：n,他的成员值都被初始化为0；
* 复制构造函数，构造一个新的向量，作为已存在的向量的完全复制，如：vector<int> v3(v2);
* 带两个常量参数的构造函数，产生初始值为一个区间的向量。区间由一个半开区间[first,last) 来指定。如：vector<int> v4(first,last）

下面一个例子用的是第四种构造方法，其它的方法读者可以自己试试。

//程序：初始化演示

#include <cstring>

#include <vector>

#include <iostream>

using namespace std;

int ar[10] = {  12, 45, 234, 64, 12, 35, 63, 23, 12, 55  };

char\* str = "Hello World";

int main()

{

    vector <int> vec1(ar, ar+10);   //first=ar,last=ar+10,不包括ar+10

    vector < char > vec2(str,str+strlen(str)); //first=str,last= str+strlen(str),

    cout<<"vec1:"<<endl;

    //打印vec1和vec2，const\_iterator是迭代器，后面会讲到

    //当然，也可以用for (int i=0; i<vec1.size(); i++)cout << vec[i];输出

    //size()是vector的一个成员函数

    for(vector<int>::const\_iterator p=vec1.begin();p!=vec1.end(); ++p)

        cout<<\*p;

        cout<<'\n'<<"vec2:"<<endl;

    for(vector< char >::const\_iterator p1=vec2.begin();p1!=vec2.end(); ++p1)

        cout<<\*p1;

    cout<<'\n';

    return 0;

}

为了帮助理解向量的概念，这里写了一个小例子，其中用到了vector的成员函数：begin()，end()，push\_back()，assign()，front()，back()，erase()，empty()，at()，size()。

#include <iostream>

#include <vector>

using namespace std;

typedef vector<int> INTVECTOR;//自定义类型INTVECTOR

//测试vector容器的功能

int main()

{

    //vec1对象初始为空

    INTVECTOR vec1;

    //vec2对象最初有10个值为6的元素

    INTVECTOR vec2(10,6);

    //vec3对象最初有3个值为6的元素，拷贝构造

    INTVECTOR vec3(vec2.begin(),vec2.begin()+3);

    //声明一个名为i的双向迭代器

    INTVECTOR::iterator i;

    //从前向后显示vec1中的数据

    cout<<"vec1.begin()--vec1.end():"<<endl;

    for (i =vec1.begin(); i !=vec1.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //从前向后显示vec2中的数据

    cout<<"vec2.begin()--vec2.end():"<<endl;

    for (i =vec2.begin(); i !=vec2.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //从前向后显示vec3中的数据

    cout<<"vec3.begin()--vec3.end():"<<endl;

    for (i =vec3.begin(); i !=vec3.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //测试添加和插入成员函数，vector不支持从前插入

    vec1.push\_back(2);//从后面添加一个成员

    vec1.push\_back(4);

    vec1.insert(vec1.begin()+1,5);//在vec1第一个的位置上插入成员5

    //从vec1第一的位置开始插入vec3的所有成员

    vec1.insert(vec1.begin()+1,vec3.begin(),vec3.end());

    cout<<"after push() and insert() now the vec1 is:" <<endl;

    for (i =vec1.begin(); i !=vec1.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //测试赋值成员函数

    vec2.assign(8,1);   // 重新给vec2赋值，8个成员的初始值都为1

    cout<<"vec2.assign(8,1):" <<endl;

    for (i =vec2.begin(); i !=vec2.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //测试引用类函数

    cout<<"vec1.front()="<<vec1.front()<<endl;//vec1第零个成员

    cout<<"vec1.back()="<<vec1.back()<<endl;//vec1的最后一个成员

    cout<<"vec1.at(4)="<<vec1.at(4)<<endl;//vec1的第五个成员

    cout<<"vec1[4]="<<vec1[4]<<endl;

    //测试移出和删除

    vec1.pop\_back();//将最后一个成员移出vec1

    vec1.erase(vec1.begin()+1,vec1.end()-2);//删除成员

    cout<<"vec1.pop\_back() and vec1.erase():" <<endl;

    for (i =vec1.begin(); i !=vec1.end(); ++i)

        cout << \*i << " ";

    cout << endl;

    //显示序列的状态信息

    cout<<"vec1.size(): "<<vec1.size()<<endl;//打印成员个数

    cout<<"vec1.empty(): "<<vec1.empty()<<endl;//清空

}

push\_back()是将数据放入vector（向量）或deque（双端队列）的标准函数。Insert()是一个与之类似的函数，然而它在所有容器中都可以使用，但是用法更加复杂。end()实际上是取末尾加一，以便让循环正确运行--它返回的指针指向最靠近数组界限的数据。

在Java里面也有向量的概念。Java中的向量是对象的集合。其中，各元素可以不必同类型，元素可以增加和删除，不能直接加入原始数据类型。

**2.2  双端队列（deque容器类）**

deque（读音：deck，意即：double queue，#include<qeque>）容器类与vector类似，支持随机访问和快速插入删除，它在容器中某一位置上的操作所花费的是线性时间。与vector不同的是，deque还支持从开始端插入数据：push\_front()。此外deque也不支持与vector的capacity()、reserve()类似的操作。

#include <iostream>

#include <deque>

using namespace std;

typedef deque<int> INTDEQUE;//有些人很讨厌这种定义法，呵呵

//从前向后显示deque队列的全部元素

void put\_deque(INTDEQUE deque, char \*name)

{

    INTDEQUE::iterator pdeque;//仍然使用迭代器输出

    cout << "The contents of " << name << " : ";

    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)

        cout << \*pdeque << " ";//注意有 "\*"号哦，没有"\*"号的话会报错

    cout<<endl;

}

//测试deqtor容器的功能

int main()

{

    //deq1对象初始为空

    INTDEQUE deq1;

    //deq2对象最初有10个值为6的元素

    INTDEQUE deq2(10,6);

    //声明一个名为i的双向迭代器变量

    INTDEQUE::iterator i;

    //从前向后显示deq1中的数据

    put\_deque(deq1,"deq1");

    //从前向后显示deq2中的数据

    put\_deque(deq2,"deq2");

    //从deq1序列后面添加两个元素

    deq1.push\_back(2);

    deq1.push\_back(4);

    cout<<"deq1.push\_back(2) and deq1.push\_back(4):"<<endl;

    put\_deque(deq1,"deq1");

    //从deq1序列前面添加两个元素

    deq1.push\_front(5);

    deq1.push\_front(7);

    cout<<"deq1.push\_front(5) and deq1.push\_front(7):"<<endl;

    put\_deque(deq1,"deq1");

    //在deq1序列中间插入数据

    deq1.insert(deq1.begin()+1,3,9);

    cout<<"deq1.insert(deq1.begin()+1,3,9):"<<endl;

    put\_deque(deq1,"deq1");

    //测试引用类函数

    cout<<"deq1.at(4)="<<deq1.at(4)<<endl;

    cout<<"deq1[4]="<<deq1[4]<<endl;

    deq1.at(1)=10;

    deq1[2]=12;

    cout<<"deq1.at(1)=10 and deq1[2]=12 :"<<endl;

    put\_deque(deq1,"deq1");

    //从deq1序列的前后各移去一个元素

    deq1.pop\_front();

    deq1.pop\_back();

    cout<<"deq1.pop\_front() and deq1.pop\_back():"<<endl;

    put\_deque(deq1,"deq1");

    //清除deq1中的第2个元素

    deq1.erase(deq1.begin()+1);

    cout<<"deq1.erase(deq1.begin()+1):"<<endl;

    put\_deque(deq1,"deq1");

    //对deq2赋值并显示

    deq2.assign(8,1);

    cout<<"deq2.assign(8,1):"<<endl;

    put\_deque(deq2,"deq2");

}

上面我们演示了deque如何进行插入删除等操作，像erase(),assign()是大多数容器都有的操作。关于deque的其他操作请参阅其他书籍。

**2.3 表（List容器类）**

　List（#include<list>）又叫链表，是一种双线性列表，只能顺序访问（从前向后或者从后向前），图2是list的数据组织形式。与前面两种容器类有一个明显的区别就是：它不支持随机访问。要访问表中某个下标处的项需要从表头或表尾处（接近该下标的一端）开始循环。而且缺少下标预算符：operator[]。

　　同时，list仍然包涵了erase(),begin(),end(),insert(),push\_back(),push\_front()这些基本函数，下面我们来演示一下list的其他函数功能。merge()：合并两个排序列表；splice()：拼接两个列表；sort()：列表的排序。

#include <iostream>

#include <string>

#include <list>

using namespace std;

void PrintIt(list<int> n)

{

    for(list<int>::iterator iter=n.begin(); iter!=n.end(); ++iter)

      cout<<\*iter<<" ";//用迭代器进行输出循环

}

int main()

{

    list<int> listn1,listn2;    //给listn1,listn2初始化

    listn1.push\_back(123);

    listn1.push\_back(0);

    listn1.push\_back(34);

    listn1.push\_back(1123);    //now listn1:123,0,34,1123

    listn2.push\_back(100);

    listn2.push\_back(12);    //now listn2:12,100

    listn1.sort();

    listn2.sort();    //给listn1和listn2排序

    //now listn1:0,34,123,1123         listn2:12,100

    PrintIt(listn1);

    cout<<endl;

    PrintIt(listn2);

    listn1.merge(listn2);    //合并两个排序列表后,listn1:0，12，34，100，123，1123

    cout<<endl;

    PrintIt(listn1);

}

上面并没有演示splice()函数的用法，这是一个拗口的函数。用起来有点麻烦。图3所示是splice函数的功能。将一个列表插入到另一个列表当中。list容器类定义了splice()函数的3个版本：

splice(position,list\_value);

splice(position,list\_value,ptr);

splice(position,list\_value,first,last);

list\_value是一个已存在的列表，它将被插入到源列表中，position是一个迭代参数，他当前指向的是要进行拼接的列表中的特定位置。

listn1:123,0,34,1123   listn2:12,100

执行listn1.splice(find(listn1.begin(),listn1.end(),0),listn2);之后，listn1将变为：123，12，100，0,34，1123。即把listn2插入到listn1的0这个元素之前。其中，find()函数找到0这个元素在listn1中的位置。值得注意的是，在执行splice之后，list\_value将不复存在了。这个例子中是listn2将不再存在。  
　　第二个版本当中的ptr是一个迭代器参数，执行的结果是把ptr所指向的值直接插入到position当前指向的位置之前.这将只向源列表中插入一个元素。  
　　第三个版本的first和last也是迭代器参数，并不等于list\_value.begin(),list\_value.end()。First指的是要插入的列的第一个元素，last指的是要插入的列的最后一个元素。

如果listn1:123,0,34,1123 listn2:12,43，87，100 执行完以下函数之后

listn1.splice(find(listn1.begin(),listn1.end(),0),++listn2.begin(),--listn2.end());

listn1:123,43,87,0,34,1123  listn2:12,100

以上，我们学习了vector,deque,list三种基本顺序容器，其他的顺序容器还有：slist,bit\_vector等等。

**2.4 集和多集（set 和multiset 容器类）：**

一个集合（#include<set>）是一个容器，它其中所包含的元素的值是唯一的。这在收集一个数据的具体值的时候是有用的。集合中的元素按一定的顺序排列，并被作为集合中的实例。如果你需要一个键/值对（pair）来存储数据，map（也是一个关联容器，后面将马上要讲到）是一个更好的选择。一个集合通过一个链表来组织，在插入操作和删除操作上比向量（vector）快，但查找或添加末尾的元素时会有些慢。

在集中，所有的成员都是排列好的。如果先后往一个集中插入：12，2，3，123，5，65 　　则输出该集时为：2，3，5，12，65，123

集和多集的区别是：set支持唯一键值，set中的值都是特定的，而且只出现一次；而multiset中可以出现副本键，同一值可以出现多次。

Set和multiset的模板参数：

template<class key, class compare, class Allocator=allocator>

第一个参数key是所存储的键的类型，第二个参数是为排序值而定义的比较函数的类型，第三个参数是被实现的存储分配符的类型。在有些编译器的具体实现中，第三个参数可以省略。第二个参数使用了合适形式的迭代器为键定义了特定的关系操作符，并用来在容器中遍历值时建立顺序。集的迭代器是双向，同时也是常量的，所以迭代器在使用的时候不能修改元素的值。

Set定义了三个构造函数：  
默认构造函数：

explicit set(const Compare&=compare());

如：set<int,less<int> > set1;

less<int>是一个标准类，用于形成降序排列函数对象。升序排列是用greater<int>。通过指定某一预先定义的区间来初始化set对象的构造函数：

template<class InputIterator> set(InputIterator, InputIterator,\ const Compare&=compare());

如：set<int ,less<int> >set2(vector1.begin(),vector1.end());

复制构造函数：

set（const set<Key,Compare&>）;

如：set<int ,less<int> >set3(set2);

下面我们来看一个简单的集和多集的插入例程：

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> set1;

    for(int i=0; i<10; ++i)

        set1.insert(i);

    for(set<int>::iterator p=set1.begin();p!=set1.end();++p)

        cout<<\*p<<"";

    if(set1.insert(3).second)//把3插入到set1中

//插入成功则set1.insert(3).second返回1，否则返回0

//此例中，集中已经有3这个元素了，所以插入将失败

        cout<<"set insert success";

    else

        cout<<"set insert failed";

    int a[] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};

    multiset<int> A;

    A.insert(set1.begin(),set1.end());

    A.insert(a,a+10);

    cout<<endl;

    for(multiset<int>::iterator p=A.begin();p!=A.end();++p)

    cout<<\*p<<" ";

    return 0;

}

**2.5 映射和多重映射（map 和multimap）**

映射和多重映射(#include<map>)基于某一类型Key的键集的存在，提供对T类型的数据进行快速和高效的检索。对map而言，键只是指存储在容器中的某一成员。Map不支持副本键，multimap支持副本键。Map和multimap对象包涵了键和各个键有关的值，键和值的数据类型是不相同的，这与set不同。set中的key和value是Key类型的，而map中的key和value是一个pair结构中的两个分量。Map支持下表运算符operator[],用访问普通数组的方式访问map，不过下标为map的键。在multimap中一个键可以对应多个不同的值。

下面的例程说明了map中键与值的关系。

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<char,int,less<char> > map1;

    map<char,int,less<char> >::iterator mapIter;

    //char 是键的类型，int是值的类型

    //下面是初始化，与数组类似

    //也可以用map1.insert(map<char,int,less<char> >::value\_type(''c'',3));

    map1['c']=3;

    map1['d']=4;

    map1['a']=1;

    map1['b']=2;

    for(mapIter=map1.begin();mapIter!=map1.end();++mapIter)

        cout<<" "<<(\*mapIter).first<<": "<<(\*mapIter).second;

    //first对应定义中的char键，second对应定义中的int值

    //检索对应于d键的值是这样做的：

    map<char,int,less<char> >::const\_iterator ptr;

    ptr=map1.find('d');

    cout<<'\n'<<" "<<(\*ptr).first<<" 键对应于值："<<(\*ptr).second;

    return 0;

}

　　从以上例程中，我们可以看到map对象的行为和一般数组的行为类似。Map允许两个或多个值使用比较操作符。下面我们再看看multimap:

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main()

{

    multimap<string,string,less<string> >mulmap;

    multimap<string,string,less<string> >::iterator p;

    //初始化多重映射mulmap:

    typedef multimap<string,string,less<string> >::value\_type vt;

    typedef string s;

    mulmap.insert(vt(s("Tom "),s("is a student")));

    mulmap.insert(vt(s("Tom "),s("is a boy")));

    mulmap.insert(vt(s("Tom "),s("is a bad boy of blue!")));

    mulmap.insert(vt(s("Jerry "),s("is a student")));

    mulmap.insert(vt(s("Jerry "),s("is a beatutiful girl")));

    mulmap.insert(vt(s("DJ "),s("is a student")));

    //输出初始化以后的多重映射mulmap:

    for(p=mulmap.begin();p!=mulmap.end();++p)

        cout<<(\*p).first<<(\*p).second<<endl;

    //检索并输出Jerry键所对应的所有的值

    cout<<"find Jerry :"<<endl;

    p=mulmap.find(s("Jerry "));

    while((\*p).first=="Jerry ")

    {

        cout<<(\*p).first<<(\*p).second<<endl;

        ++p;

    }

    return 0;

}

在map中是不允许一个键对应多个值的，在multimap中，不支持operator[],也就是说不支持map中允许的下标操作。

**3 算法（algorithm）：**

#inlcude <algorithm>

STL中算法的大部分都不作为某些特定容器类的成员函数，他们是泛型的，每个算法都有处理大量不同容器类中数据的使用。值得注意的是，STL中的算法大多有多种版本，用户可以依照具体的情况选择合适版本。中在STL的泛型算法中有4类基本的算法：

**变序型队列算法**：可以改变容器内的数据；

**非变序型队列算法**：处理容器内的数据而不改变他们；

**排序值算法：**包涵对容器中的值进行排序和合并的算法，还有二叉搜索算法、通用数值算法。（注：STL的算法并不只是针对STL容器，对一般容器也是适用的。）

**变序型队列算法：**又叫可修改的序列算法。这类算法有复制（copy）算法、交换（swap）算法、替代（replace）算法、删除（clear）算法，移动（remove）算法、翻转（reverse）算法等等。这些算法可以改变容器中的数据（数据值和值在容器中的位置）。

**3.1 翻转和复制（reverse()和copy()）**

下面介绍2个比较常用的算法reverse()和copy()。

#include <iostream>

#include <algorithm>

#include <iterator>

//下面用到了输出迭代器ostream\_iterator

using namespace std;

int main()

{

    int arr[6]={1,12,3,2,1215,90};

    int arr1[7];

    int arr2[6]={2,5,6,9,0,-56};

    copy(arr,(arr+6),arr1);//将数组aar复制到arr1

    cout<<"arr[6] copy to arr1[7],now arr1: "<<endl;

    for(int i=0;i<7;i++)

        cout<<" "<<arr1[i];

    reverse(arr,arr+6);//将排好序的arr翻转

    cout<<'\n'<<"arr reversed ,now arr:"<<endl;

    copy(arr,arr+6,ostream\_iterator<int>(cout, " "));//复制到输出迭代器

    swap\_ranges(arr,arr+6,arr2);//交换arr和arr2序列

    cout<<'\n'<<"arr swaped to arr2,now arr:"<<endl;

    copy(arr,arr+6,ostream\_iterator<int>(cout, " "));

    cout<<'\n'<<"arr2:"<<endl;

    copy(arr2,arr2+6,ostream\_iterator<int>(cout, " "));

    return 0;

}

revese()的功能是将一个容器内的数据顺序翻转过来，它的原型是：

template<class Bidirectional>

void reverse(Bidirectional first, Bidirectional last);

将first和last之间的元素翻转过来，上例中你也可以只将arr中的一部分进行翻转：

reverse(arr+3,arr+6); 这也是有效的。First和last需要指定一个操作区间。

Copy()是要将一个容器内的数据复制到另一个容器内，它的原型是：

  Template<class InputIterator ，class OutputIterator>

  OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

它把[first,last－1]内的队列成员复制到区间[result,result+(last-first)-1]中。泛型交换算法：

**3.2 单值交换（Swap()）**

Swap()操作的是单值交换，它的原型是：

template<class T>

void swap(T& a,T& b);

swap\_ranges()操作的是两个相等大小区间中的值，它的原型是：

  template<class ForwardIterator1, class ForwardIterator2>

  ForwardIterator2swap\_ranges(ForwardIterator1 first1,ForwardIterator1 last1, ForwardIterator1 first2);

交换区间[first1,last1-1]和[first2, first2+(last1-first1)-1]之间的值，并假设这两个区间是不重叠的。

非变序型队列算法，又叫不可修改的序列算法。这一类算法操作不影响其操作的容器的内容，包括搜索队列成员算法，等价性检查算法，计算队列成员个数的算法。我将用下面的例子介绍其中的find(),search(),count()：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int main()

{

    int a[10]={12,31,5,2,23,121,0,89,34,66};

    vector<int> v1(a,a+10);

    vector<int>::iterator result1,result2;//result1和result2是随机访问迭代器

    result1=find(v1.begin(),v1.end(),2);

    //在v1中找到2，result1指向v1中的2

    result2=find(v1.begin(),v1.end(),8);

    //在v1中没有找到8，result2指向的是v1.end()

    cout<<result1-v1.begin()<<endl; //3－0＝3或4－1＝3，屏幕结果是3

    cout<<result2-v1.end()<<endl;

    int b[9]={5,2,23,54,5,5,5,2,2};

    vector<int> v2(a+2,a+8);

    vector<int> v3(b,b+4);

    result1=search(v1.begin(),v1.end(),v2.begin(),v2.end());

    cout<<\*result1<<endl;

    //在v1中找到了序列v2，result1指向v2在v1中开始的位置

     result1=search(v1.begin(),v1.end(),v3.begin(),v3.end());

     cout<<\*(result1-1)<<endl;

    //在v1中没有找到序列v3，result指向v1.end(),屏幕打印出v1的最后一个元素66

     vector<int> v4(b,b+9);

     int i=count(v4.begin(),v4.end(),5);

     int j=count(v4.begin(),v4.end(),2);

     cout<<"there are "<<i<<" members in v4 equel to 5"<<endl;

     cout<<"there are "<<j<<" members in v4 equel to 2"<<endl;

     //计算v4中有多少个成员等于 5,2

     return 0;

}

**3.3 查找（find()）**

find()的原型是：

template<class InputIterator，class EqualityComparable>

InputIterator find(InputIterator first, InputIterator last, const EqualityComparable& value);

其功能是在序列[first,last-1]中查找value值，如果找到，就返回一个指向value在序列中第一次出现的迭代，如果没有找到，就返回一个指向last的迭代（last并不属于序列）。

search()的原型是：

template <class ForwardIterator1, class ForwardIterator2>

ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,                        ForwardIterator2 first2, ForwardIterator2 last2);

其功能是在源序列[first1,last1-1]查找目标序列[first2，last2-1]如果查找成功，就返回一个指向源序列中目标序列出现的首位置的迭代。查找失败则返回一个指向last的迭代。

**3.4 得到数目（Count()）**

Count()的原型是：

template <class InputIterator, class EqualityComparable>

iterator\_traits<InputIterator>::difference\_type count(InputIterator first,

InputIterator last, const EqualityComparable& value);

其功能是在序列[first,last-1]中查找出等于value的成员，返回等于value得成员的个数。

排序算法（sort algorithm）：这一类算法很多，功能强大同时也相对复杂一些。这些算法依赖的是关系运算。在这里我只介绍其中比较简单的几种排序算法：sort(),merge(),includes()

#include <iostream>

#include <algorithm>

using namespace std;

int main()

{

    int a[10]={12,0,5,3,6,8,9,34,32,18};

    int b[5]={5,3,6,8,9};

    int d[15];

    sort(a,a+10);

    for(int i=0;i<10;i++)

      cout<<" "<<a[i];

    sort(b,b+5);

    if(includes(a,a+10,b,b+5))

       cout<<'\n'<<"sorted b members are included in a."<<endl;

    else

       cout<<"sorted a dosn`t contain sorted b!";

    merge(a,a+10,b,b+5,d);

    for(int j=0;j<15;j++)

       cout<<" "<<d[j];

    return 0;

}

**3.5 排序（sort()）**

sort()的原型是：

template <class RandomAccessIterator>

void sort(RandomAccessIterator first, RandomAccessIterator last);

功能是对[first,last-1]区间内的元素进行排序操作。与之类似的操作还有：partial\_sort(), stable\_sort()，partial\_sort\_copy()等等。

merge()的原型是：

template <class InputIterator1, class InputIterator2, class OutputIterator>

OutputIterator merge(InputIterator1 first1, InputIterator1 last1,InputIterator2  first2, InputIterator2 st2,OutputIterator result);

将有序区间[first1,last1-1]和[first2,last2-1]合并到[result, result + (last1 - first1) + (last2 - first2)-1]区间内。

Includes()的原型是：

template <class InputIterator1, class InputIterator2>

bool includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);

其功能是检查有序区间[first2,last2-1]内元素是否都在[first1,last1-1]区间内，返回一个bool值。

通用数值算法（generalized numeric algorithms）：这一类算法还不多，涉及到专业领域中有用的算术操作，独立包涵于头文件<numeric>中。  
　　STL中的算法大都有多种版本，常见的版本有以下4中：

默认版本，假设给出了特定操作符；

一般版本，使用了成员提供的操作符；

复制版本，对原队列的副本进行操作，常带有 \_copy 后缀；

谓词版本，只应用于满足给定谓词的队列成员，常带有 \_if 后缀；

以上我们学习了STL容器和算法的概念，以及一些简单的STL容器和算法。在使用算法处理容器内的数据时，需要从一个数据成员移向另一个数据成员，迭代器恰好实现了这一功能。下面我们来学习STL迭代器 。

3.6 堆的用法（heap）  
STL里面的堆操作一般用到的只有4个。  
他们就是  
make\_heap();、pop\_heap();、push\_heap();、sort\_heap();  
他们的头函数是algorithm  
首先是make\_heap();  
他的函数原型是：  
void make\_heap(first\_pointer,end\_pointer,compare\_function);  
一个参数是数组或向量的头指针，第二个向量是尾指针。第三个参数是比较函数的名字  
。在缺省的时候，默认是大跟堆。（下面的参数都一样就不解释了）  
作用：把这一段的数组或向量做成一个堆的结构。范围是(first,last)  
然后是pop\_heap();  
它的函数原型是：  
void pop\_heap(first\_pointer,end\_pointer,compare\_function);  
作用：pop\_heap()不是真的把最大（最小）的元素从堆中弹出来。而是重新排序堆。它把first和last交换，然后将[first,last-1)的数据再做成一个堆。  
接着是push\_heap()  
void pushheap(first\_pointer,end\_pointer,compare\_function);  
作用：push\_heap()假设由[first,last-1)是一个有效的堆，然后，再把堆中的新元素加  
进来，做成一个堆。  
最后是sort\_heap()  
void sort\_heap(first\_pointer,end\_pointer,compare\_function);  
作用是sort\_heap对[first,last)中的序列进行排序。它假设这个序列是有效堆。（当然  
，经过排序之后就不是一个有效堆了）  
下面是例程：  
#include<algorithm>  
#include<cstdio>  
using namespace std;  
bool cmp(int a,int b)  
{  
    return a>b;  
}  
int main()  
{  
    int i,number[20]={29,23,20,22,17,15,26,51,19,12,35,40};  
    make\_heap(&number[0],&number[12]);  
    //结果是:51 35 40 23 29 20 26 22 19 12 17 15  
    for(i=0;i<12;i++)  
        printf("%d ",number[i]);  
    printf("\n");  
    make\_heap(&number[0],&number[12],cmp);  
    //结果：12 17 15 19 23 20 26 51 22 29 35 40  
    for(i=0;i<12;i++)  
        printf("%d ",number[i]);  
    printf("\n");  
    //加入元素8  
    number[12]=8;  
    //加入后调整  
    push\_heap(&number[0],&number[13],cmp);  
    //结果：8 17 12 19 23 15 26 51 22 35 40 20  
    for(i=0;i<13;i++)  
        printf("%d ",number[i]);  
    printf("\n");  
    //弹出元素8  
    pop\_heap(&number[0],&number[13],cmp);  
    //结果：12 17 15 19 23 20 26 51 22 29 35 40  
    for(i=0;i<13;i++)  
        printf("%d ",number[i]);  
    printf("\n");  
    sort\_heap(&number[0],&number[12],cmp);  
    //结果不用说都知道是有序的了！  
    for(i=0;i<12;i++)  
        printf("%d ",number[i]);  
    return 0;  
}

**4 迭代器（itertor）**

#include<iterator>

迭代器实际上是一种泛化指针，如果一个迭代器指向了容器中的某一成员，那么迭代器将可以通过自增自减来遍历容器中的所有成员。迭代器是联系容器和算法的媒介，是算法操作容器的接口。在运用算法操作容器的时候，我们常常在不知不觉中已经使用了迭代器。  
STL中定义了6种迭代器：

**输入迭代器**，在容器的连续区间内向前移动，可以读取容器内任意值；

**输出迭代器**，把值写进它所指向的队列成员中；

**前向迭代器**，读取队列中的值，并可以向前移动到下一位置（++p,p++）；

**双向迭代器**，读取队列中的值，并可以向前向后遍历容器；

**随机访问迭代器,** vector<T>::iterator，list<T>::iterator等都是这种迭代器；

**流迭代器**，可以直接输出、输入流中的值；

实际上，在前面的例子中，我们不停的在用迭代器。下面我们用几个例子来帮助理解这些迭代器的用法。  
下面的例子用到了输入输出迭代器：

#include <iostream>

#include <fstream>

#include <iterator>

#include <vector>

#include <string>

using namespace std;

int main()

{

    vector<string> v1;

    ifstream file("Text1.txt");

    if(file.fail())

    {

        cout<<"open file Text1.txt failed"<<endl;

        return 1;

    }

    copy(istream\_iterator<string>(file),istream\_iterator<string>(),inserter(v1,v1.begin()));

    copy(v1.begin(),v1.end(),ostream\_iterator<string>(cout," "));

    cout<<endl;

    return 0;

}

这里用到了输入迭代器istream\_iterator，输出迭代器ostream\_iterator。程序完成了将一个文件输出到屏幕的功能，先将文件读入，然后通过输入迭代器把文件内容复制到类型为字符串的向量容器内，最后由输出迭代器输出。Inserter是一个输入迭代器的一个函数(迭代器适配器)，它的使用方法是：

inserter (container ,pos);

container是将要用来存入数据的容器，pos是容器存入数据的开始位置。上例中，是把文件内容存入（copy()）到向量v1中。

**5 STL的其他标准组件**

**5.1 函数对象（functor或者funtion objects）**

#include<functional>

函数对象又称之为仿函数。函数对象将函数封装在一个对象中，使得它可作为参数传递给合适的STL算法，从而使算法的功能得以扩展。可以把它当作函数来使用。用户也可以定义自己的函数对象。下面让我们来定义一个自己的函数对象.

#include <iostream>

using namespace std;

struct int\_max{

int operator()(int x,int y){return x>y?x:y; }

};//operator() 重载了"（）"， (int x,int y)是参数列表

int main()

{

    cout<<int\_max()(3,4)<<endl;

    return 0;

}

这里的int\_max（）就是一个函数对象，struct关键字也可以用class来代替，只不过struct默认情况下是公有访问权限，而class定义的是默认私有访问权限。下面我们来定义一个STL风格的函数对象：

#include <iostream>

#include <vector>

using namespace std;

struct adder : public unary\_function<double, void>

{

    adder() : sum(0) {}

    double sum;

    void operator()(double x) { sum += x; }

};

int main()

{

    double a[5]={0.5644,1.1,6.6,8.8,9.9};

    vector<double> V(a,a+5);

    adder result = for\_each(V.begin(), V.end(), adder());

    cout << "The sum is " << result.sum << endl;

    return 0;

}

在这里，我们定义了一个函数对象adder()，这也是一个类，它的基类是unary\_function函数对象。unary\_function是一个空基类，不包涵任何操作或变量。只是一种格式说明，它有两个参数，第一个参数是函数对象的使用数据类型，第二个参数是它的返回类型。基于它所定义的函数对象是一元函数对象。（注：用关键字struct或者class定义的类型实际上都是"类"）

STL内定义了各种函数对象，否定器、约束器、一元谓词、二元谓词都是常用的函数对象。函数对象对于编程来说很重要，因为他如同对象类型的抽象一样作用于操作。

**5.2 适配器（adapter）**

适配器是用来修改其他组件接口的STL组件，是带有一个参数的类模板（这个参数是操作的值的数据类型）。STL定义了3种形式的适配器：容器适配器，迭代器适配器，函数适配器。

**容器适配器：**包括栈（stack）、队列(queue)、优先(priority\_queue)。使用容器适配器，stack就可以被实现为基本容器类型（vector,dequeue,list）的适配。可以把stack看作是某种特殊的vctor、deque或者list容器，只是其操作仍然受到stack本身属性的限制。queue和priority\_queue与之类似。容器适配器的接口更为简单，只是受限比一般容器要多；

**迭代器适配器：**修改为某些基本容器定义的迭代器的接口的一种STL组件。反向迭代器和插入迭代器都属于迭代器适配器，迭代器适配器扩展了迭代器的功能；

**函数适配器：**通过转换或者修改其他函数对象使其功能得到扩展。这一类适配器有否定器（相当于"非"操作）、帮定器、函数指针适配器。

next\_permutation(num,num+len)可以将数组num，从num[0]到num[len-1]改为按照字典序排列的下一个。

get.c\_str() get是一个string的对象，有c\_str()函数返回它的c语言字符串类型。