

Generic Programming

——C++ Standard Template Library



哈工大(威海)经管学院

2011-9-12

说 明

高手们都说 STL(Standard Template Library)有用, STL 是现代 C++的精髓,不会 STL 就不会现代 C++编程,学完 C++当然要学 STL,云云。还有人说学完 C++再学 STL 就是再踏披荆之路。笔者认为,不管他人如何评说,毕竟 STL 是一个高效的 C++程序库,它被容纳于 C++标准程序库(C++ Standard Library)中,是 ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法。为广大 C++程序员们提供了一个可扩展的应用框架,高度体现了软件的可复用性。因此,作为未来的 IT 人士,应该对 STL 有所了解。另外,笔者一直觉得,目前国内高校专门给学生扩充 STL 知识或技术者尚不多见,应该说这是一个欠缺。为此,笔者专为我院 2010 级信息管理与信息系统专业的学生编纂了本资料,作为他们学习《面向对象程序设计/C++》和《数据结构》课程之后,扩充这些课程内容的粗浅读物。阅读本资料者需要具备一定的 C++和数据结构方面的基础知识。文中所有 STL 程序都在 VC++6.0 环境下调试通过。

水平有限,加之时间仓促,错误难免,请多指正!

2011-9-12 于威海

目 录

Generic Programming.....	I
第 1 讲 初识 STL.....	- 1 -
§1.1 什么是泛型程序设计.....	- 1 -
§1.2 什么是 STL.....	- 1 -
§1.3 STL 的核心组件.....	- 2 -
§1.3.1 容器 (container)	- 3 -
§1.3.2 算法 (algorithm)	- 3 -
§1.3.3 迭代器 (Iterator)	- 4 -
§1.3.4 函数对象(function object)	- 5 -
§1.3.5 容器、迭代器、算法和函数对象之间的关系.....	- 8 -
§1.4 建立 STL 程序的方法.....	- 8 -
第 2 讲 STL 容器.....	- 11 -
§2.1 序列式容器.....	- 11 -
§2.1.1 Vectors.....	- 11 -
§2.1.2 Deques.....	- 20 -
§2.1.3 Lists.....	- 23 -
§2.2 关联式容器.....	- 30 -
§2.2.1 操作 Set 和 Multisets 的成员函数及算法.....	- 30 -
§2.2.2 Set 和 Multisets 应用实例.....	- 32 -
§2.2.3 操作 Maps 和 Multimaps 的成员函数.....	- 32 -
§2.2.4 Maps 和 Multimaps 应用实例.....	- 35 -
§2.3 特殊容器——顺序容器配接器.....	- 38 -
§2.3.1 Stack(栈).....	- 38 -
§2.3.2 Queue(队列).....	- 40 -
§2.3.3 Priority Queues(优先级队列).....	- 41 -
§2.4 Strings 可以视为一种 STL 容器.....	- 43 -
第 3 讲 STL 迭代器.....	- 46 -
§3.1 迭代器.....	- 46 -
§3.2 迭代器的分类(Iterator Categorise)	- 48 -
§3.3 迭代器的算术操作.....	- 48 -
§3.4 迭代器的区间.....	- 50 -
§3.5 迭代器的辅助函数.....	- 51 -
§3.5.1 函数 advance(p, n)	- 51 -
§3.5.2 函数 distance(first, last)	- 52 -
§3.5.3 函数 Iter_swap(p1,p2).....	- 53 -
§3.6 迭代器适配器(Iterator Adapters).....	- 54 -
§3.6.1 Insert Iterators (安插型迭代器)	- 54 -
§3.6.2 Stream Iterators (流迭代器)	- 56 -
§3.6.3 Reverse Iterators (逆向迭代器)	- 58 -
第 4 讲 STL 函数对象.....	- 60 -
§4.1 什么是函数对象.....	- 60 -

§4.2	标准库中预定义的函数对象	- 62 -
§4.3	STL 中一元函数对象与二元函数对象的基类	- 66 -
§4.4	函数适配器	- 69 -
§4.5	STL 预定义函数对象应用示例	- 74 -
第 5 讲	STL 算法	- 76 -
§5.1	非变异算法	- 80 -
§5.2	变异算法	- 89 -
§5.3	排序算法	- 96 -
§5.4	数值算法	- 102 -
附录	STL 基本容器常用成员函数一览	- 106 -
§1	各类容器共性成员函数	- 106 -
§2	顺序容器和关联容器共有函数	- 106 -
§3	容器比较	- 106 -
§4	vector 容器常用成员函数	- 106 -
§5	deque 容器常用成员函数	- 107 -
§6	list 容器常用成员函数	- 108 -
§7	队列和堆栈常用成员函数	- 109 -
§8	优先队列常用成员函数	- 110 -
§9	集合常用成员函数	- 110 -
§10	映射常用成员函数	- 111 -

第 1 讲 初识 STL

§1.1 什么是泛型程序设计

所谓泛型程序设计（Generic Programming, GP），就是编写不依赖于具体数据类型的程序。C++中的模板(包括函数模版与类模版)是泛型程序设计的主要工具。

泛型程序设计的主要思想是将算法从特定的数据结构中分离出来，使算法成为通用的、可以适用于各种不同类型的容器（数据结构）。这样就不必为每种容器都写一套同样的算法，从而提高了软件的复用性。显然，泛型程序设计更加通用。

§1.2 什么是 STL

STL（Standard Template Library，标准模板库）是一个高效的 C++程序库。它被容纳于 C++标准程序库（C++ Standard Library）中，是 ANSI/ISO C++标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里常用的基本数据结构和基本算法。为广大 C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性。

从逻辑层次来看，在 STL 中体现了泛型程序设计的思想，引入了诸多新的概念，比如像容器（container）、算法（algorithmn）以及迭代器（iterator）等。与 OOP 中的多态（polymorphism）一样，泛型也是一种软件复用技术。

从实现层次看，整个 STL 是以类型参数化（type parameterized）的方式实现的。这种方式完全基于 C++标准中的模板（template），也就是说，整个 STL 几乎都是类模板和函数模板。

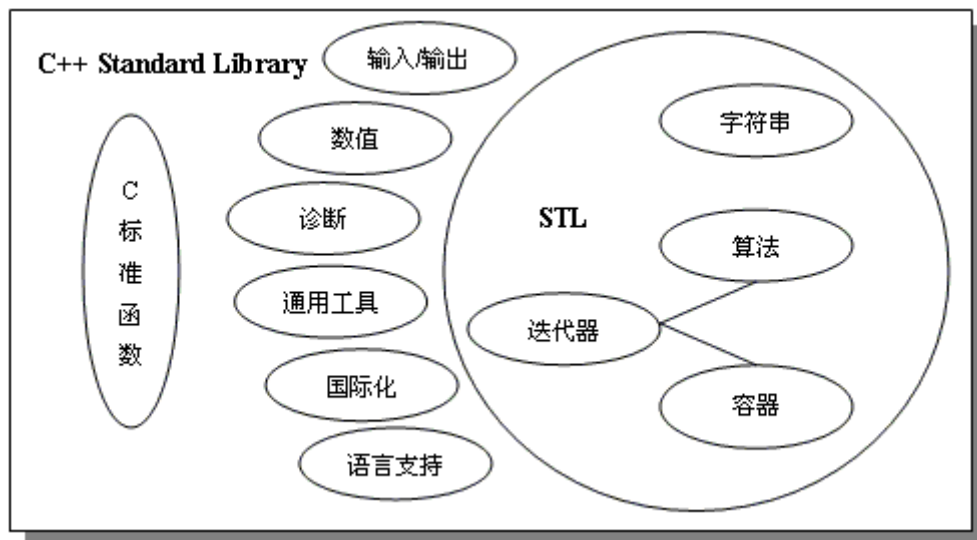


图 1-1 STL 和 C++标准函数库的组件

STL 最初是由 HP 公司被誉为 STL 之父的 Alexander Stepanov（出生于莫斯科）和 Meng Lee 开发的一个用于支持 C++泛型编程的模板库，1998 年被纳入 C++标准，成为 C++标准库的一部分，如图 1-1 所示。STL 是 C++发展的重要里程碑。由于 C++标准库有多种不同的实现，因此，STL 也有多种版本，但它们为用户提供的接口都遵守共同的标准。

§1.3 STL 的核心组件

STL 主要包含容器、算法、迭代器三大核心部分：

- STL 容器包含了绝大多数数据结构，如数组、链表、字符串、队列、堆栈、树等；
- STL 算法包含了诸如增、删、改、查、排序等系统(全局)函数模板。这些算法都是泛型算法，同一个算法可以适用于任意类型的容器；

- STL 迭代器类似指针一样,通过它的有序移动,把容器中的元素与算法关联起来,它是实现所有 STL 功能的基础所在。

下面首先看一个简单的 STL 程序。

【例 1-1】从标准输入设备键盘依次输入 5 个整数 5、-3、1、0 和 8，存入一个向量容器中，然后输出它们的相反数。

```
#include <iostream>
#include <vector>      //向量头文件
#include <iterator>    //迭代器头文件
#include <algorithm>    //算法头文件
#include <functional>  //函数对象(仿函数)头文件
using namespace std;  //导入命名空间 std

void main() {
    const int N = 5;
    vector<int> s(N); //定义一个含有 5 个 int 类型数据元素的向量容器对象 s
    vector<int>::iterator pos; //声明一个向量迭代器 pos，该向量元素为 int 型
    for (int i = 0; i < N; ++i)
        cin >> s[i]; //从键盘依次输入整数 5、-3、1、0 和 8
        //显然，向量容器可以随机访问

    cout<<"从键盘输入的 5 个整数是: "<<endl;
    for (pos = s.begin(); pos != s.end(); pos++)
        cout << *pos << ", "; //通过迭代器 pos 访问向量容器的元素
    cout<<endl;
    cout<<"输入的 5 个数的相反数是: "<<endl;
    transform(s.begin(), s.end(),
               ostream_iterator<int>(cout, ", "), negate<int>());
    cout << endl;
}
```

程序运行的结果如图 1-2 所示。

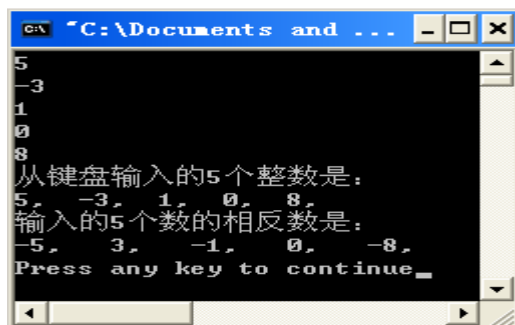


图 1-2 例 1-1 的运行结果

例 1-1 虽然简单,但却涉及到了 STL 的 3 种最关键、最核心的组件: 容器(Container)、算法(Algorithms)与迭代器(Iterator)。除此之外, STL 还有函数对象(Function object)、容器适配器(container adaptor)以及 STL 的其它标准组件。在早期的 STL 版本中, 函数对象又称为仿函数(functor)。下面分别对它们做简单介绍, 在后面还要分门别类地比较深入地介绍它们。

§1.3.1 容器(container)

STL 中的容器是一种存储有限个 T(Template)类型数据元素的集合, 每种容器都是一种数据结构。数据元素可以是类的对象本身, 如果数据类型 T 代表的是 Class 的话。当然, 这些数据元素也可以是其它数据类型的数据。例 1-1 中的 s 就是一种容器对象, 叫做向量容器。容器的内部实现是类模板。

STL 提供了七种基本容器, 它们被划分为两大类, 即序列式容器(Sequence Containers)和关联式容器(Associative Containers)。除了基本容器外, STL 还提供了一些其它类型的容器, 如容器适配器(容器适配器可以看作是由其它容器实现的容器)。

序列式容器中的元素顺序与元素值无关, 只与元素插入的次序和存放位置有关。STL 预先定义了三种序列式容器, 即 Vectors(向量)、Deque(双向队列)和 List(双向链表)。

关联式容器中的元素位置是按元素值的大小自动排序的, 缺省情况下为升序排列。其元素顺序与元素值有关, 与元素插入的先后次序无关。关联式容器的底层实现是二叉搜索树的形式。STL 中预先定义的关联式容器有 Sets 与 Multisets(集合与多重集合)以及 Map 与 Multimap(映射与多重映射)。

注意, 在 C++ 标准中, STL 的组件被组织为 13 个头文件: <vector>、<deque>、<list>、<set>、<map>、<stack>、<queue>、<iterator>、<functional>、<algorithm>、<numeric>、<memory>和<utility>。使用不同的容器或容器适配器, 需要包含不同的头文件。表 1-1 列出了常用的容器和容器适配器对应的头文件。因此, 今后在 STL 程序中使用哪种(或哪些)容器时, 必须首先将对应的那个(或哪些)头文件包含进来。

表 1-1 常用的容器和容器适配器对应的头文件

容器类别	数据结构(容器类)	实现头文件
序列式容器	向量(vector)	<vector>
	双向队列(deque)	<deque>
	双向链表(list)	<list>
关联式容器	集合(set)、多重集合(multiset)	<set>
	映射(map)、多重映射(multimap)	<map>
容器适配器	栈(stack)	<stack>
	队列(queue)、优先级队列(priority_queue)	<queue>

§1.3.2 算法(algorithm)

STL 提供了大约 70 多个已经实现了的算法的函数模板, 比如算法 find()用于在容器中查找等于某个特定值的元素。再如算法 sort()用于对容器中的元素进行排序等等。

这样一来, 只要熟悉了 STL 之后, 许多代码就可以被大大简化, 因为只要直接调用一

两个算法模板，就可以完成所需要的功能，从而大大提升软件生产率。

今后在软件的编写过程中，凡是要用到 STL 中的有关算法，必须先将算法所在的头文件包含进来，即：`#include<algorithm>`。

STL 中的算法分别被组织在头文件 `<algorithm>`、`<numeric>` 和 `<functional>` 当中，其中：
`<algorithm>` 是所有 STL 头文件中最大的一个（尽管它很好理解），它由一大堆函数模板组成。可以认为头文件 `<algorithm>` 中的每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等。

`<numeric>` 体积很小，只包括几个在序列上面进行简单算术(数值)运算的函数模板，如加法和乘法在序列上的一些操作。涉及到使用 STL 中的算术(数值)运算方法时，要包含此头文件。

`<functional>` 中定义了一些类模板，用以声明函数对象（又称为仿函数）。涉及到使用 STL 预定义好的仿函数时，要包含此头文件。

总之，算法是应用在容器上并以各种方式处理容器内容(数据元素)的行为或功能。在 STL 中，算法是由函数模板实现的。要特别注意，这些算法不是容器类的成员函数，相反，它们是独立于容器的全局函数(当然，容器本身也可能具有类似功能的公有成员函数可以使用，而且应该优先使用它们)。因此，不必通过容器对象来调用它们，直接调用即可。

令人吃惊的特点之一就是这些算法是如此地通用，不仅可以将其直接用于 STL 各种容器，而且还可以用于普通的 C++ 数组或任何其他应用程序指定的容器。

在例 1-1 中调用的 **`transform()`** 就是一个算法，其函数模板如下：

```
template<class InputIterator, class OutputIterator, class Unary_Function>
OutputIterator transform(InputIterator first, InputIterator last,
                        OutputIterator result, Unary_Function op)
{
    for(; first != last; ++ first, ++ last)
        * result = op(*first);
    return result;
}
```

`Transform()` 算法的功能是顺序遍历 *first* 和 *last* 两个迭代器所指区间内的诸元素，将每个元素的值作为二元函数对象 *op* 的参数，将 *op* 执行的返回值通过输出流迭代器 *result* 顺序输出。遍历完成后，*result* 迭代器的指向是输出的最后一个元素的下一个位置。

算法 **`transform()`** 会将迭代器返回，但本例的程序忽略了这个返回值。

§1.3.3 迭代器 (Iterator)

容器负责存储数据元素，算法负责加工处理数据元素，那么是谁负责将数据元素从容器里提取出来交给算法去处理呢？又是谁负责将算法处理过的数据元素写入容器呢？这项任务由迭代器负责完成。迭代器将容器与算法耦合起来，通过迭代器可以对容器内任意位置

上的元素进行定位和存取(access)。

可以把迭代器看作一个类似于指向容器中元素的被泛化了的普通指针。可以像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元素;反之,也可以像递减一个指针那样递减迭代器,使其依次指向容器中每一个前驱元素。尽管迭代器类似于指针,但迭代器决不是普通的指针。**它们的区别是:**指针是指向它所访问的数据元素的地址,而迭代器是指向它所访问的数据元素的位置(即容器的某个位置的下标)。但它们也有相同之处,即要访问它们所指的数据元素时,都是在其前面缀“*”运算符。

例 1-1 中的 `s.begin()`、`s.end()`以及 `ostream_iterator<int>(cout, ", ")`都是迭代器。其中

- `s.begin()`——指向向量容器 `s` 的第一个元素位置;
- `s.end()`——指向向量容器 `s` 的末尾位置(即最后一个元素所在位置的下一个位置);
- `ostream_iterator<int>(cout, ", ")`——是一个输出流迭代器(实例),其中的 `ostream_iterator` 是一个输出迭代器的类模板。本例中通过执行它的构造函数来建立一个输出流迭代器对象,构造时使用的 2 个参数是输出流对象 `cout` 和分隔符 `","`。`ostream_iterator` 类的实例并不指向 STL 容器的元素,而是一个输出流对象。假设 `p` 是该类型的一个变量(即 `ostream_iterator` 类的一个实例,也就是一个 `ostream_iterator` 迭代器),那么执行 `*p=x` 的结果就是将 `x` 输出到它所关联的输出流对象,并向其中输出一个分隔符。本例中,输出流迭代器关联到的输出流对象是 `cout`,分隔符是一个逗号,二者都是提供给构造函数的实参。

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,后面将深入讨论它。注意,每种类型的容器都可以定义它的迭代器(一个或多个),若程序中需要使用这类迭代器时可以直接定义。然而,当需要使用另外一类独立于 STL 容器、但在 STL 已经预定义了的迭代器时,如输入/出流迭代器,通常需要包含头文件 `<iterator>`。

§1.3.4 函数对象(function object)

STL 中有很多算法(即函数,包含在头文件 `algorithm` 中)都需要提供一个称为函数对象(Function Object)的参数。用户可以通过函数对象对容器中的元素实施特定的操作。

尽管在实现函数回调时广泛使用函数指针,但 C++STL 提供了比函数指针更好的实现回调函数的方法,这就是函数对象。通过函数对象不仅可以实现对函数的调用,而且十分灵活。从语法上讲,函数对象与普通函数的行为是类似的。在早期的 STL 版本中,函数对象又称为仿函数(functor)。

所谓函数对象,就是重载了函数运算符“`()`”的那个类的对象(class object)。下面举例说明如何定义和使用函数对象。

1、定义函数对象类

首先,声明一个重载了函数运算符“`()`”的普通类:

```
class Negate
{
public:
    int operator() (int n) { return -n;}    //重载函数运算符“()”
};
```

记住,在重载“`()`”操作符的语句中,第一个圆括弧总是空的,因为它代表重载的操作符是“`()`”;第二个圆括弧是参数列表。注意,通常在重载其它操作符时,参数数量是固定的(由

操作符涉及的操作数个数决定)，而重载“()”操作符时有所不同，它可以有任意多个参数。

因为在上面的 **Negate** 类中内建的操作(作用是取某数的相反值，此例中该数指定为 **int** 类型)是一元的（运算符“-”只有一个操作数），所以重载的“()”操作符也只有一个参数。返回类型与参数类型相同——本例中为 **int**。函数功能是返回与参数符号相反的整数。

2、使用函数对象

现在定义一个叫 **Callback()** 的函数来测试上面定义的函数对象类。**Callback()** 有两个参数：一个为 **int**，一个是对类 **Negate** 对象(即函数对象)的引用。**Callback()** 将函数对象 **neg** 作为一个普通的函数名使用：

```
#include <iostream>
using namespace std;

void Callback(int n, Negate & neg)    //第 2 个参数是对 Negate 类对象的引用
{
    int val = neg(n);    //调用仿函数 neg ()，即转化为调用重载的操作符“()”函数
    cout << val<<endl;
}
/* 注意，neg 是对象(调用构造函数生成)。编译器将语句
    int val = neg(n);
    转化为
    int val = neg.operator()(n);    //返回“-n”的值给 val
*/
```

通常，函数对象所在的类不必显式定义构造函数和析构函数。下面用主函数 **main()** 实现对 **Callback()** 的参数传递：

```
int main()
{
    Callback(5, Negate()); //第 2 个参数是通过缺省构造函数生成的一个临时对象
}
```

把上面的几段代码合并成一个完整的有关函数对象的程序如下：

```
#include <iostream>
using namespace std;

class Negate {
public:
    int operator() (int n) { return -n;}    //重载函数运算符“()”
};

void Callback(int n, Negate & neg)    //第 2 个参数是对 Negate 类对象的引用
{
    int val = neg(n);    //调用仿函数 neg ()，即调用重载的操作符“()”
    cout << val<<endl;}

int main() {
```

```
Callback(5, Negate() ); //第 2 个参数是通过缺省构造函数生成的一个临时匿名对象
}
```

程序运行结果如图 1-3 所示。

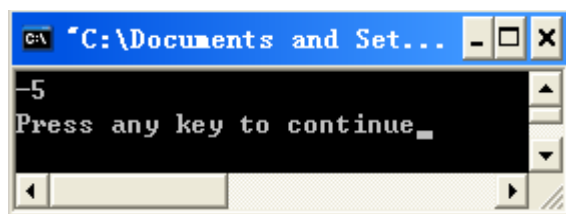


图 1-3

3、模板函数对象

从上面的代码可以看出，其数据类型被限制为 `int`，而通用性是函数对象的优势之一，那么如何创建具有通用性的函数对象呢？方法是使用模板，也就是将那个重载的函数操作符“`()`”的函数定义为类的成员函数模板，以使函数对象适用于任何数据类型，如 `double` 类型或 `char` 类型等。请看下面的代码：

```
class GenericNegate{
public:
    template <class T>
    T operator() (T t) {return -t;} //模板函数对象
};
#include <iostream>
#include <string>
using namespace std;

void main(){
    GenericNegate negate; //生成类的对象
    cout<< negate(5.3333)<<endl<<endl;           //double 类型
    cout<< negate('a')<<endl<<endl;             //字符类型
    cout<< negate(string("abcdefg"))<<endl<<endl; //string 类型
}
```

程序运行结果如图 1-4 所示。

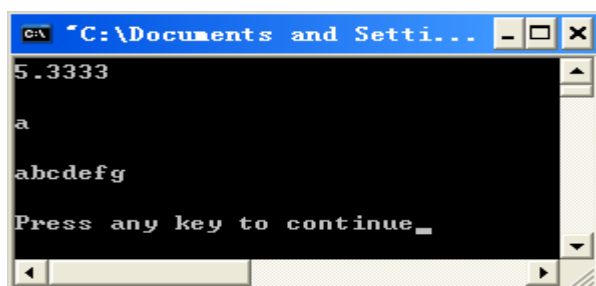


图 1-4

显然，如果用普通的回调函数实现上述的灵活性是相当困难的。

§1.3.5 容器、迭代器、算法和函数对象之间的关系

容器负责存储数据元素，算法负责处理数据元素。而迭代器则负责从容器中存取一个个数据元素提交给算法去进行处理，或者将算法处理完的数据元素放入容器。因此，迭代器将算法和容器连在一起，它们之间的关系如图 1-5 所示。

STL 把迭代器作为算法的参数而非把容器直接作为算法的参数。函数对象也可作为算法的参数。

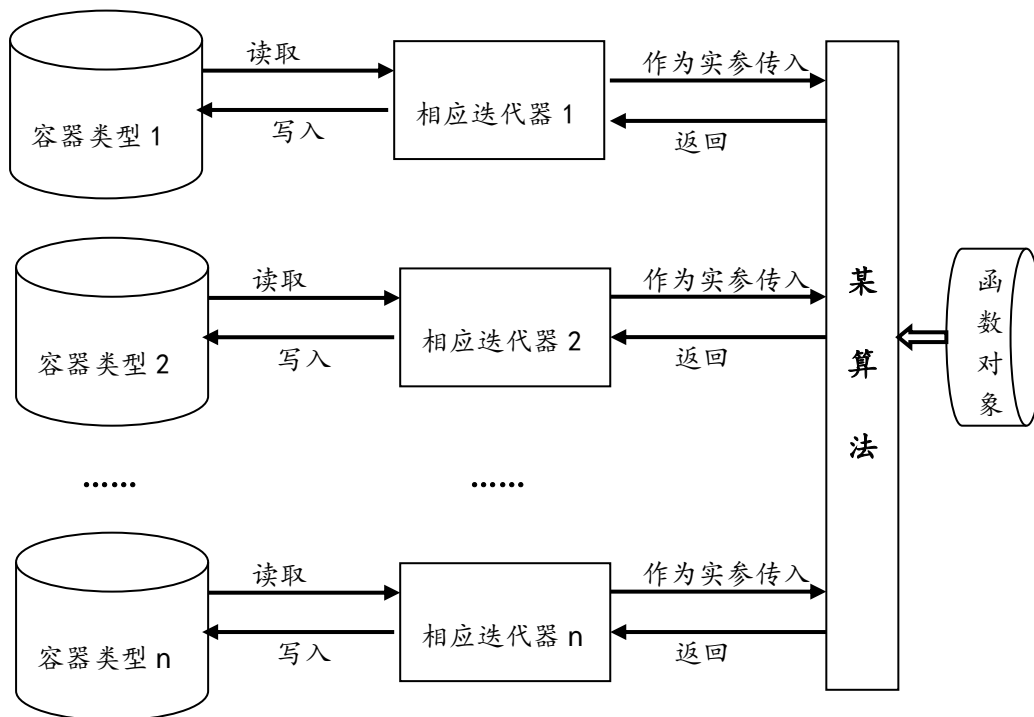


图 1-5 STL 组件及其之间的关系

§1.4 建立 STL 程序的方法

以 Visual C++6.0 开发平台为例。

(1) 启动 Visual C++6.0 后出现图 1-6 界面。在图 1-6 中，选择【文件】菜单→【新建】命令，出现图 1-7 界面。

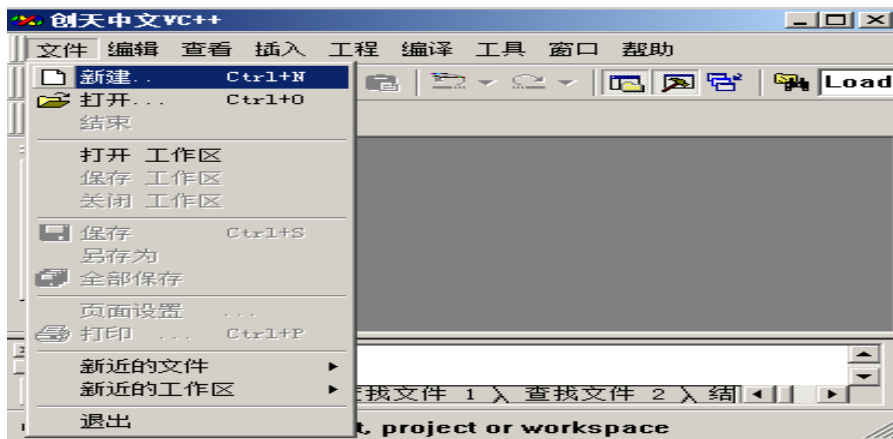


图 1-6



图 1-7

(2)在图 1-7 中, 选择【工程】→【Win32 Console Application】, 输入工程名, 例如 MyStl, 按【确定】按钮, 出现图 1-8 界面。

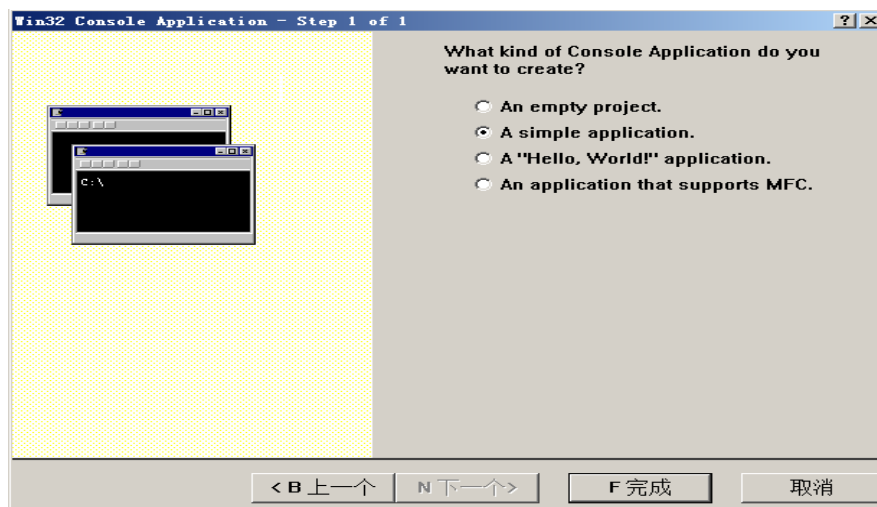


图 1-8

(3) 在图 1-8 中, 选择【A simple application】选项, 按【完成】按钮, 工程创建完毕。

(4)编辑文件。在图 1-9 中, 选择【文件】菜单, 接着在其下拉菜单中选择【新建】或【打开】源文件, 输入或打开程序, 如图 1-10 所示。

(5)编译、链接调试与执行程序, 如图 1-10 所示。



图 1-9

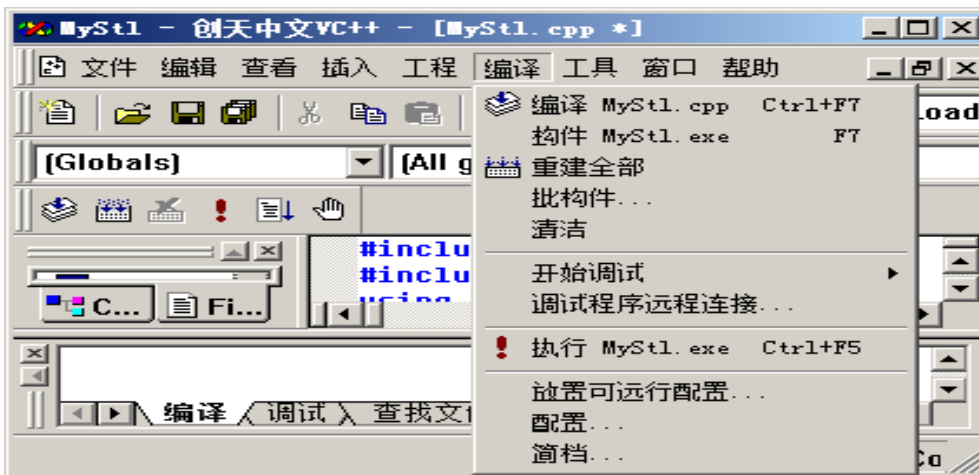


图 1-10

第 2 讲 STL 容器

容器 (Containers) 是用来存储和管理一组数据元素的数据结构, 每种容器都是一种数据结构。在第 1 讲中曾经指出: STL 提供了七种基本容器, 它们被分为两大类: 序列式容器 (Sequence Containers) 和关联式容器 (Associative Containers)。除了基本容器外, STL 还提供了几个容器配接器以及一些其它类型的容器。下面将分别介绍它们。

在正式介绍它们之前, 先做一项声明, 提醒大家注意!

各类容器的共性——各类容器一般来说都有下列成员函数, 并且, 由于它们都重载了比较运算符, 所以它们都允许比较判断运算(设 `c1` 和 `c2` 是两个类型相容的容器对象) :

- 默认构造函数: 提供容器默认初始化构造函数
- 复制构造函数: 将容器初始化为现有同类容器副本的构造函数
- 析构函数: 不再需要容器时进行内存整理的析构函数

- `c1.empty()`: 若 `c1` 内没有数据元素存在返回 `true`, 否则返回 `false`
- `c1.max_size()`: 返回 `c1` 中的最大元素个数
- `c1.size()`: 返回 `c1` 中当前实有元素个数
- `c1.swap(c2)`: 交换 `c1` 和 `c2` 的元素

下面的比较运算均返回 `true` 或者 `false`:

- `c1 = c2`
- `c1 < c2`
- `c1 <= c2`
- `c1 > c2`
- `c1 >= c2`
- `c1 == c2`
- `c1 != c2`

§2.1 序列式容器

序列式容器中元素的顺序与元素值无关, 只与元素插入的先后次序和存放位置有关。STL 预先定义好了三种序列式容器, 即 **Vectors** (向量)、**Deque** (双向队列) 和 **List** (双向链表)。

STL 中的每种序列式容器类都内嵌了一个 `iterator`(迭代器)类, 可以由它来定义该容器的 `iterator`。

§2.1.1 Vectors

`vector` 容器是数组的一个泛化推广。由于 `vector` 容器内的数据元素被存放在一块连续的内存里, 所以, 对 `vector` 容器内的数据元素可以通过下标法进行随机访问; 还可以使用容器的成员函数 `push_back(i)` 在容器尾端插入新元素 `i` 等; 还可以使用它的其它相关成员函数在 `vector` 的头部或其它任何位置进行插入、删除操作, 参见表 2-2。

注意: `vector` 并未提供在其头部插入新元素的 `push_front` 成员函数, 但可以使用功能相同的其它成员函数执行在 `vector` 的头部插入新元素, 如 `insert()` 成员函数。

vector 实际上是一个动态数组，具有内存自动管理功能。对于元素的插入和删除，**vector** 可以动态调整它所占用的内存空间大小。在 **STL** 程序中使用向量容器之前，必须包含头文件 `<vector>`。

此外，在定义向量类对象时，必须指定它存储的数据元素的数据类型。因为 **vector** 类是一个类模板。例如语句：“`vector<int> intList;`” 声明了一个存储的元素类型为 `int` 的向量容器对象 `intList`。类似地，语句：“`vector<string> stringList;`” 将 `stringList` 声明为一个存储的元素类型为 `string` 的向量容器对象。

§2.1.1.1 声明向量对象——vector 类中的构造函数

向量容器使用动态数组存储、管理数据元素。因为数组是一个随机访问的数据结构，所以可以随机访问向量中的元素。无论在数组中间，还是在开始处，插入一个元素都是费时的（因为要向后移动元素），特别是在数组非常大的时候更是如此。然而在数组末端插入元素却很快（因为不需要向后移动元素）。所以向量容器提供了一个在其末端插入元素的成员函数 `push_back(i)`。

STL 程序中使用 **Vector** 类的构造函数声明向量对象。**Vector** 类重载了多个构造函数，其中包括默认构造函数，当然还有拷贝构造函数和析构函数。因此，可以通过多种方式来声明和初始化一个向量容器对象（简称向量容器）。表 2-1 描述了怎样声明和初始化指定类型的向量容器。

表 2-1 声明和初始向量容器的方法(**vector** 类的构造函数)

语 句	作 用
<code>vector<elementType> vecList;</code>	创建一个没有任何元素的空向量 <code>vecList</code> （使用默认构造函数）
<code>vector<elementType> vecList(otherVecList)</code>	创建一个向量 <code>vecList</code> ，并使用向量 <code>otherVecList</code> 中的元素初始化该向量。向量 <code>vecList</code> 与向量 <code>otherVecList</code> 的类型必须相同
<code>vector<elementType> vecList(size);</code>	创建一个大小为 <code>size</code> 的向量 <code>vecList</code> ，并使用默认构造函数初始化该向量
<code>vector<elementType> vecList(n, elem);</code>	创建一个大小为 <code>n</code> 的向量 <code>vecList</code> ，该向量中所有的 <code>n</code> 个元素都初始化为 <code>elem</code>
<code>vector<elementType> vecList(begin, end);</code>	创建一个向量 <code>vecList</code> ，以区间 <code>(begin, end)</code> 做为元素的初值。

§2.1.1.2 操作向量——vector 类中的公有成员函数

在介绍了如何声明一个具体的向量容器类对象之后，下面开始讨论如何操作向量容器中的数据元素。必须掌握下面几种基本操作：

- 元素插入
- 元素删除
- 遍历向量容器中的元素
- 返回向量容器中元素的个数
- 判定向量容器是否空

假设 `vecList` 是一个向量类型容器对象。表 2-2 给出了在 `vecList` 中插入元素和删除元素的操作，这些操作都是 **vector** 类的公有成员函数。表 2-2 还说明了如何使用这些操作。其中多数成员函数被重载（即参数不同）。

表 2-2 向量容器各种基本操作的成员函数

操作类型	成员函数	作 用
删除	vecList.clear()	从容器中删除所有元素
	vecList.erase(position)	删除由 position 指定位置上的元素
	vecList.erase(beg, end)	删除从 beg 到 end-1 区间内的所有元素
	vecList.pop_back()	删除最后一个元素
插入	vecList.insert(position, elem)	将 elem 的一个副本插入到由 position 指定的位置上, 并返回新元素的位置
	vecList.insert(position, n, elem)	将 elem 的 n 个副本插入到由 position 指定的位置上
	vecList.insert(position, beg, end)	将从 beg 到 end-1 之间的所有元素的副本插入到 vecList 中由 position 指定的位置上
	vecList.push_back(elem)	将 elem 的一个副本插入到向量容器的末尾
重置大小	vecList.resize(num)	将元素个数改为 num。如果 size() 增加, 默认的构造函数负责创建这些新元素
	vecList.resize(num, elem)	将元素个数改为 num。如果 size() 增加, 默认的构造函数将这些新元素初始化为 elem

§2.1.1.3 在向量容器中声明迭代器

1、声明向量容器的迭代器

vector 类内嵌了一个 iterator 类, 它是 vector 类的一个 public 成员。通过 iterator 类可以声明向量容器的一个或多个迭代器, 例如语句:

```
vector<int>::iterator intVecIter;
```

将 intVecIter 声明为 int 类型的向量容器迭代器。

2、向量容器迭代器的运算表达式

```
++intVecIter    将迭代器 intVecIter 前移一个位置, 使其指向容器中的下一个元素;
--intVecIter   将迭代器 intVecIter 后移一个位置, 使其指向容器中的前一个元素;
*intVecIter    返回当前迭代器位置上的元素值。
```

3、容器的成员函数 begin()和 end()

不仅仅向量容器, 所有容器都包含成员函数 begin()和 end()。函数 begin()返回容器中第一个元素的位置; 函数 end()返回容器中最后一个元素的下一个位置。这两个函数都没有参数。

假设有下面的语句:

```
vector<int> intList; // intList 声明为元素为 int 类型的向量容器(对象)
```

```
vector<int>::iterator intVecIter;// intVecIter 声明为元素为 int 类型的向量容器的迭代器
```

那么, 执行完语句: intVecIter = intList.begin(); 结果迭代器 intVecIter 将指向容器 intList 中第一个元素所在的位置。第一个元素所在的位置就是第 0 号位置。

【示例】下面的 for 循环将 intList 中所有元素输出到标准输出设备上:

```
for (intVecIter = intList.begin(); intVecIter != intList.end(); ++intVecIter);
cout<<*intVecIter<<"    ";
```

可以通过表 2-3 中给出的操作 (均是向量容器的公有成员函数) 来直接访问向量容器中的元素。

表 2-3 访问向量容器中元素的其它操作(向量容器的其它成员函数)

表达式	作 用
<code>vecList.at(index)</code>	返回由 <code>index</code> 指定的位置上的元素
<code>vecList[index]</code>	返回由 <code>index</code> 指定的位置上的元素
<code>vecList.front()</code>	返回第一个元素（不检查容器是否为空）
<code>vecList.back()</code>	返回最后一个元素（不检查容器是否为空）

表 2-3 说明：可以按照数组的方式来处理向量中的元素（注意，在 C++ 中，数组下标从 0 开始，向量容器中第一个元素的位置号也是 0）。

向量类中还包含：返回容器中当前元素个数的成员函数 `size()`，返回可以插入到容器中的元素的最大个数的成员函数 `max_size()` 等。表 2-4 描述了一些操作（注意，假设 `vecCont` 是向量容器对象）。

表 2-4 计算向量容器大小的操作(向量容器的其它成员函数)

表达式	作 用
<code>vecCont.capacity()</code>	返回不重新分配空间可以插入到容器 <code>vecCont</code> 中的元素的最大个数
<code>vecCont.empty()</code>	若容器 <code>vecCont</code> 为空，返回 <code>true</code> ；否则，返回 <code>false</code>
<code>vecCont.size()</code>	返回容器 <code>vecCont</code> 中当前实有元素的个数
<code>vecCont.max_size()</code>	返回可以插入到容器 <code>vecCont</code> 中的元素的最大个数

【例 2-1】下面给出一个样本程序供进一步认识向量容器成员函数的用法

```
#include <iostream>
#include <vector>
using namespace std;

void main(){
    vector<int> intList;
    int i;
    intList.push_back(13); //在尾部插入元素 13
    intList.push_back(75);
    intList.push_back(28);
    intList.push_back(35);
    cout<<"Line 1: List Elements: ";
    for(i=0;i<4;i++)    //遍历向量容器
        cout<<intList[i]<<"    ";    //向量容器可以象数组那样随机访问
    cout<<endl;

    for(i=0;i<4;i++)
        intList[i] *=2;    //向量容器可以象数组那样来访问
    cout<<"Line 2: List Elements: ";
    for(i=0;i<4;i++)
        cout<<intList[i]<<"    ";    //向量容器可以象数组那样来访问
```

```

cout<<endl;

vector<int>::iterator listIt; //声明一个迭代器对象
cout<<"Line 3: list Elements: ";
for(listIt=intList.begin();listIt != intList.end();++listIt) //通过迭代器访问向量容器
    cout<<*listIt<<" ";
cout<<endl;

listIt=intList.begin();
++listIt;
++listIt;
intList.insert(listIt,88);
cout<<"Line 4: List Elements: ";
for(listIt = intList.begin();listIt !=intList.end();++listIt)
    cout<<*listIt<<" ";
cout<<endl;
}

```

运行程序输出结果如图 2-1 所示。

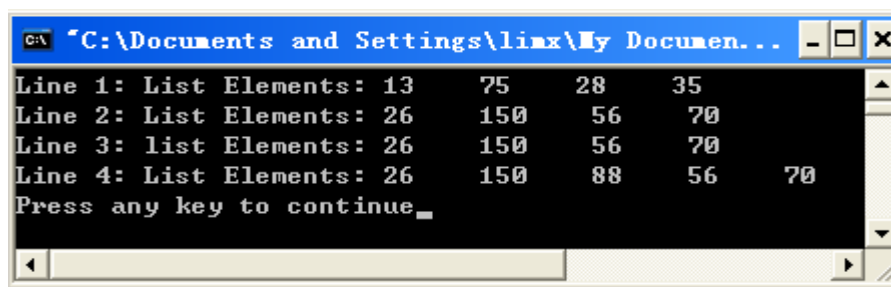


图 2-1 例 2-1 运行结果

§2.1.1.4 使用向量容器的成员函数操作向量容器的再举例

1、成员函数 **push_back()**

【功能】在容器尾端加入一个新数据元素

通过 **vector** 容器的构造函数可以创建一个 **vector** 容器对象(方式很多种),然后对 **vector** 容器初始化赋值(可以使用成员函数 **push_back()**在容器尾端加入元素),如果要对 **vector** 容器遍历的话,既可以常用数组下标方式随即访问,也可以用迭代器方式访问。

【例 2-2】下面的例子展示了如何用数组方式访问 **vector** 元素。

```

#include <vector>    //“vector”头文件
#include <iostream>
using namespace std;
int main(void){
    vector<int> v; //创建一个 vector 容器对象
    v.push_back(11); //在末端添加数据元素 11
    v.push_back(22);
    v.push_back(33);
    for(int i = 0; i < v.size(); i++)

```

```

        cout << "v[" << i << "] = " << v[i] << endl; //下标法, 随即访问
    return 0;
}

```

运行结果如图 2-2 所示。

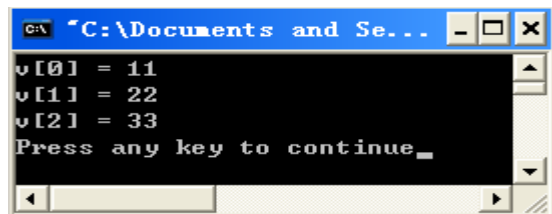


图 2-2 例 2-2 运行结果

2、成员函数 insert ()

【功能】在任意位置插入一个数据元素

除了 push_back()函数可以给 vector 容器尾部添加元素外, insert()函数则可以在任意位置插入元素。由于插入的时候要先将插入位置后面的元素后移, 因此, 与 push_back 函数相比, insert 函数耗时比 push_back 函数久。

【例 2-3】下面代码就展示了如何使用 insert 函数:

```

#include <vector>
#include <iostream>
using namespace std;
int main(void){
    vector<int> v;
    v.push_back(22);
    v.push_back(33);
    v.push_back(55);
    v.push_back(66);
    v.insert(v.begin() + 3, 44); //v.begin() + 3=0+3=3, 在 v 的第 3 个位置插入 44
    v.insert(v.begin(),11);
    v.insert(v.end(),77);
    for(int i = 0; i < v.size(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
    return 0;
}

```

运行结果如图 2-3 所示。

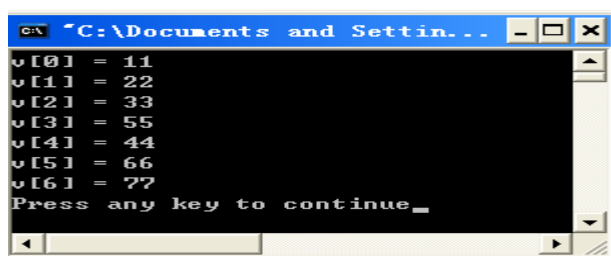


图 2-3 例 2-3 运行结果

2、成员函数 `erase()` 与 `clear()`

【功能】删除元素

学会了如何给 `vector` 容器添加元素以后，当然要知道怎么删除元素了。`vector` 容器提供了 `erase()` 函数，可以用来删除指定元素或者迭代器区间`[first, last)`内的元素。

`vector` 容器还提供另一个删除元素的函数是 `clear()` 函数，它在内部调用了 `erase()` 函数将`[begin(), end())`区间的元素全部删除，即清空 `vector` 容器。

【例 2-4】下面代码展示了如何使用 `erase` 成员函数和 `clear` 成员函数

```
#include <iostream>
#include <vector>
using namespace std;
void main(void)
{
    vector<int> v;
    v.push_back(11);
    v.push_back(22);
    v.push_back(33);
    v.push_back(44);
    v.push_back(55);
    v.erase(v.begin()+1);
    v.erase(v.begin()+2,v.begin()+4);
    vector<int>::iterator i;
    int j;
    for(i=v.begin(), j=0; i!=v.end(); i++, j++)
        cout<<"v[" << j << "] = " << *i<< endl;
    v.clear();
    cout << "vector clear()!" << endl;
}
```

运行结果如图 2-4 所示。

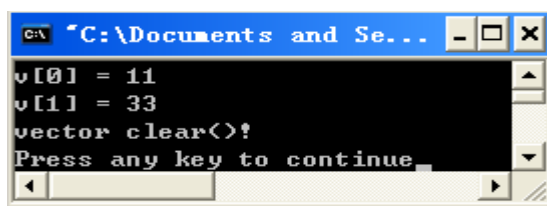


图 2-4 例 2-4 运行结果

//注：之前用的迭代器是正向，`vector` 还提供了反向迭代器(`reverse_iterator`，相应的有 `rbegin()`，`rend()`)，具体使用方法跟之前的一样，这里就不举例了。

4、成员函数 `swap()`

【功能】交换两个同类型 `vector` 容器的数据元素

【例 2-5】下面代码展示了如何使用 `swap` 函数：

```
#include <vector>
#include <iostream>
```

```

using namespace std;
void print(vector<int>& v){
    for(int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}
void main(void){
    vector<int> v1;
    v1.push_back(11);
    v1.push_back(22);
    v1.push_back(33);
    cout << "v1 = ";
    print(v1);

    vector<int> v2;
    v2.push_back(44);
    v2.push_back(55);
    v2.push_back(66);
    cout << "v2 = ";
    print(v2);
    v1.swap(v2); //交换 v1 和 v2 的元素
    cout << "After swap:" << endl;
    cout << "v1 = ";
    print(v1);
    cout << "v2 = ";
    print(v2); }

```

运行结果如图 2-5 所示。

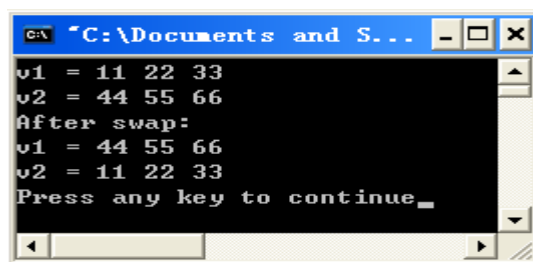


图 2-5 例 2-5 运行结果

5、其它几个成员函数

vector 容器还提供了一些可以用于统计的函数，可以用 **empty()**函数来判断容器是否为空；**size()**函数返回容器实际的元素个数(上面已经有用到过)；**max_size()**函数来获得容器的最大元素个数，**capacity()**函数来获得容器当前可容纳的 **vector** 元素个数(如果没有特殊声明容器可容纳元素个数的话，该函数返回值一般跟 **size()**是一样的，不过如果使用 **reserve** 函数来重新调整容器可容纳元素个数则会修改该函数的返回值，而不会修改 **size()**函数的返回值)，

【例 2-6】下面的代码展示了上面几个函数的用法：

```

#include <vector>
#include <iostream>
using namespace std;

void print(vector<int>& v){
    cout << "-----" << endl;
    cout << "empty = " << v.empty() << endl; //空返 1, 非空返 0
    cout << "size = " << v.size() << endl;
    cout << "capacity = " << v.capacity() << endl;
}

int main(void){
    vector<int> v;
    cout << "max_size = " << v.max_size() << endl;
    print(v);
    v.push_back(11);
    v.push_back(22);
    print(v);
    v.push_back(33);
    v.push_back(44);
    print(v);
    v.reserve(30);
    print(v);
    return 0;
}

```

运行结果如图 2-6 所示。

```

C:\Documents and ...
max_size = 1073741823
-----
empty = 1
size = 0
capacity = 0
-----
empty = 0
size = 2
capacity = 2
-----
empty = 0
size = 4
capacity = 4
-----
empty = 0
size = 4
capacity = 30
Press any key to continue

```

图 2-6 例 2-6 运行结果

容器还有很多成员函数，这里不再详细介绍了，下面仅对另外几个再简单说明一下：

Front() 函数可以返回容器首元素的引用 (容器不能为空)

back() 函数相应地返回末尾元素的引用(容器不能为空)

pop_back () 函数与 **push_back()**作用相反，用于删除容器末尾元素。

对于数组参数，**sizeof()**返回数组使用的内存字节数。由于这个原因，如果 **T** 是数组 **arr** 中元素的类型，则 **arr** 中元素的个数为 **sizeof(arr)/sizeof(T)**。使用这种方法决定数组元素的个数，可以保证能够不改变向量声明，而添加附加的数组元素并重新编译程序，所以可以如下声明向量 **intVector**：

```
int intArray[]={9, 2, 7, 3, 12}, i;
int arrsize=sizeof(intArray)/sizeof(int);
vector<int>intVector(intArray, intArray+arrsize);
```

向量类的默认构造函数：在声明中没有包含任何参数，最终的向量为空，size=0。如果不向空向量内添加新元素，这样的空向量是没有什么价值的。

§2.1.2 Deques

§2.1.2.1 Deque

Deque(发音类似“deck”)是“double-ended queue”的缩写，即“双端队列”。它是一个可以向前后两端发展的 dynamic array。可以在其头尾两端插入或删除元素，而且十分迅速。而在头部和中间部分插入或删除元素则要移动元素，所以比较费时。若程序中使用 Deque，那么，必需包含<deque>头文件。下面先看一个双端队列的例子。

【例 2-7】声明一个其元素为浮点数的 Deque，并在其头部依次插入 1.1~6.6 共 6 个元素，然后输出它们。Deque 容器的 **push_front()**成员函数是在 Deque 头部插入一个数据元素。

```
#include <iostream>
#include <deque>    //Deque 头文件
using namespace std;

int main(){
    deque<float> coll;    // deque container for floating-point elements

    // insert elements from 1.1 to 6.6 each at the front
    for (int i=1; i<=6; ++i) {
        coll.push_front(i*1.1);    // push_front()是在头部插入一个数据元素
    }

    // print all elements followed by a space
    for (int j=0; j<coll.size(); ++j) {    //遍历
        cout << coll[j] << ' ';
    }
    cout << endl;
}
```

运行结果如图 2-7 所示。

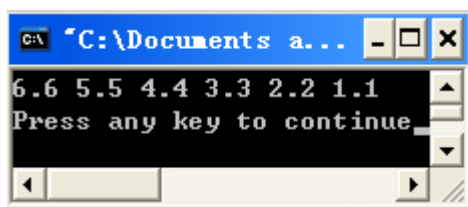


图 2-7 例 2-7 运行结果

注意：①也可以使用成员函数 **push_back()**在 deque 尾部追加元素；

②vector 并未提供在 vector 头部添加元素的 push_front()成员函数。

§2.1.2.2 Deque 的成员操作函数

Deque 是双向队列，操作 Deque 的成员函数列于表 2-5 中。

表 2-5 操作 Deque 的成员函数

函 数	描 述	备注
assign(beg,end)	给[beg; end) 区间赋值	
assign(n,elem)	将 n 个 elem 的拷贝赋值	
at(idx)	传回索引 idx 所指的数据，若 idx 越界，抛出 out_of_range	
back()	传回最后一个数据，不检查这个数据是否存在	
begin()	传回迭代器重的可一个数据	
clear()	移除容器中所有数据	
deque c1(c2)	复制一个 deque	构造函数
deque c(n)	创建一个 deque, 含有 n 个数据, 数据均已缺省构造产生	构造函数
deque c(n, elem)	创建一个含有 n 个 elem 拷贝的 deque	构造函数
deque c(beg,end)	创建一个以[beg;end)区间的 deque	构造函数
~deque()	销毁所有数据，释放内存	析构造函数
empty()	判断容器是否为空	
end()	指向迭代器中的最后一个数据地址	
erase(pos)	删除 pos 位置的数据，传回下一个数据的位置	
erase(beg,end)	删除[beg,end)区间的数据, 传回下一个数据的位置	
front()	传回地一个数据	
get_allocator	使用构造函数返回一个拷贝	
insert(pos,elem)	在 pos 位置插入一个 elem 拷贝, 传回新数据位置	
insert(pos,n,elem)	在 pos 位置插入 n 个 elem 数据, 无返回值	
insert(pos,beg,end)	在 pos 位置插入在[beg,end)区间的数据, 无返回值	
max_size()	返回容器中最大数据的数量	
pop_back()	删除最后一个数据	
pop_front()	删除头部数据	
push_back(elem)	在尾部加入一个数据	
push_front(elem)	在头部插入一个数据	
rbegin()	传回一个逆向队列的第一个数据	
rend()	传回一个逆向队列的最后一个数据的下一个位置	
resize(num)	重新指定队列的长度	
size()	返回容器中实际数据的个数	
c1.swap(c2)	将 c1 和 c2 元素互换	
swap(c1,c2)	同上操作	
operator []	返回容器中指定位置的一个引用	

*deque 容器类也有其缺省构造函数和拷贝构造函数

§2.1.2.3 Deque 综合示例

【例 2-8】下面再举一个例子说明 deque 的功能

```
#include <iostream>
```

```

#include <deque>
using namespace std;
typedef deque<int> INTDEQUE;

//定义一个从前向后显示 deque 队列全部元素的全局函数
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 容器功能的 main()函数
Void main(){
    INTDEQUE deq1;          // deq1 对象初始为空(调用缺省构造函数)
    INTDEQUE deq2(10,6);    //deq2 对象最初有 10 个值均为 6 的元素

    INTDEQUE::iterator i;   //声明一个名为i 的双向迭代器变量

    put_deque(deq1,"deq1"); //从前向后显示 deq1 中的数据

    put_deque(deq2,"deq2"); //从前向后显示 deq2 中的数据
    //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");

    //双端队列类的成员函数进行测试
    cout<<"deq1.at(4)="<<deq1.at(4)<<endl;
    cout<<"deq1[4]="<<deq1[4]<<endl;

```

```

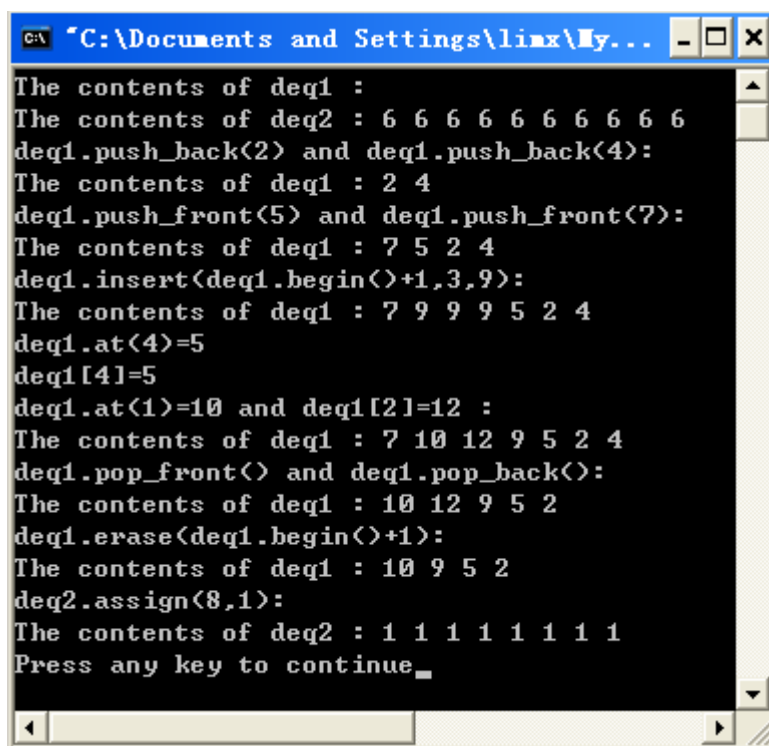
    deq1.at(1)=10;
    deq1[2]=12;
    cout<<"deq1.at(1)=10 and deq1[2]=12 :"<<endl;
    put_deque(deq1,"deq1");
    //从 deq1 序列的前、后各移去一个元素
    deq1.pop_front();
    deq1.pop_back();
    cout<<"deq1.pop_front() and deq1.pop_back():"<<endl;
    put_deque(deq1,"deq1");

    deq1.erase(deq1.begin()+1); //清除 deq1 中的第 2 个元素
    cout<<"deq1.erase(deq1.begin()+1):"<<endl;
    put_deque(deq1,"deq1");

    deq2.assign(8,1); //对 deq2 赋值并显示
    cout<<"deq2.assign(8,1):"<<endl;
    put_deque(deq2,"deq2");
}

```

程序运行结果如图 2-8 所示。



```

C:\Documents and Settings\linx\My...
The contents of deq1 :
The contents of deq2 : 6 6 6 6 6 6 6 6 6 6
deq1.push_back(2) and deq1.push_back(4):
The contents of deq1 : 2 4
deq1.push_front(5) and deq1.push_front(7):
The contents of deq1 : 7 5 2 4
deq1.insert(deq1.begin()+1,3,9):
The contents of deq1 : 7 9 9 9 5 2 4
deq1.at(4)=5
deq1[4]=5
deq1.at(1)=10 and deq1[2]=12 :
The contents of deq1 : 7 10 12 9 5 2 4
deq1.pop_front() and deq1.pop_back():
The contents of deq1 : 10 12 9 5 2
deq1.erase(deq1.begin()+1):
The contents of deq1 : 10 9 5 2
deq2.assign(8,1):
The contents of deq2 : 1 1 1 1 1 1 1 1
Press any key to continue_

```

图 2-8 例 2-8 运行结果

§2.1.3 Lists

List 是(带头结点的)双向链表(doublelinked list)，它是一种双线性列表。只能顺序访问它(从前向后或者从后向前逐个访问)。

List 与向量和双端队列两种容器类有一个明显的区别就是：它不支持随机访问(所以没

有成员函数 `at(pos)` 和 `operator[]`。要访问 `list` 中某个数据元素只能从表头或表尾处开始循环进行（可以通过迭代器来双向遍历读取）。`list` 容器不提供对 `capacity()` 和内存维护相关的接口，因为 `list` 容器中的每个元素分配自己的内存空间，当元素删除时，其对应的内存空间自动销毁。**List 的优势是可以在其任何位置上插入或删除元素，而且比较迅速(不需要移动元素)**。若程序中使用 `List` 容器，那么，必需先包含 `<list>` 头文件。下面先看一个 `List` 容器的例子。

【例 2-9】首先建立一个空 `List`，然后将‘a’至‘z’的所有字符插入其中，利用循环每次输出并删除第一个元素，直至输出所有元素。

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<char> coll;    // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);    ///成员函数 push_back ()在尾部追加一个元素
    }

    while (! coll.empty()) {    //成员函数 empty()是判断 List 容器是否为空
        cout << coll.front() << ' '; //成员函数 front()会返回第一个元素
        coll.pop_front();    //成员函数 pop_front()会删除第一个元素，但不返回
                             //第一个元素
    }
    cout << endl;
}
```

程序运行结果如图 2-9 所示。

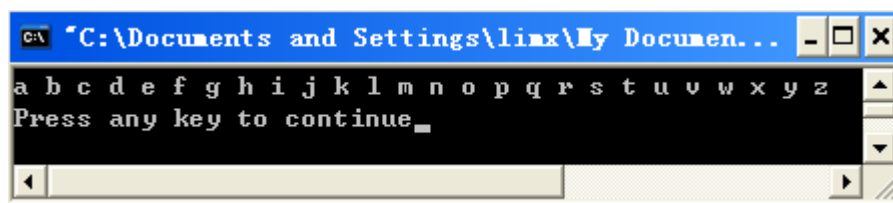


图 2-9 例 2-9 运行结果

§2.1.3.1 List 的操作

对 `List` 的操作主要是调用它的公有成员函数来实现，它的主要公有成员函数如下：

设 `list` 的对象 `list1` 和 `list2` 分别有元素 `list1(1,2,3)` 和 `list2(4,5,6)`。并且由语句“`list<int>::iterator it;`”声明了 `list` 的一个迭代器 `it`。

1. **list** 构造函数(以元素为 int 型为例)


```
list<int> L0;           // 产生一个空 list
list<int> L1(9);        // 建一个含 9 个元素的 list
list<int> L2(5,1);      // 建一个含 5 个元素其值均为 1 的 list
list<int> L3(L2);       // 建一个 L2 的 copy 的 list
list<int> L4(L0.begin(), L0.end()); //建一个含 L0 一个区域的元素的 list
```
2. **assign()**分配值，有两个重载


```
L1.assign(4,3);        // L1(3,3,3,3)
L1.assign(++list1.beging(), list2.end()); // L1(2,3)
```
3. **operator=** 赋值重载运算符


```
L1 = list1;           // L1(1,2,3)
```
4. **front()**返回第一个元素的引用


```
int nRet = list1.front()    // nRet = 1
```
5. **back()**返回最后一元素的引用


```
int nRet = list1.back()     // nRet = 3
```
6. **begin()**返回第一个元素的指针(iterator)


```
it = list1.begin();        // *it = 1
```
7. **end()**返回最后一个元素的下一位置的指针(list 为空时 end()=begin())


```
it = list1.end();
--it;                      // *it = 3
```
8. **rbegin()**返回链表最后一元素的后向指针(reverse_iterator or const)


```
list<int>::reverse_iterator it = list1.rbegin(); // *it = 3
```
9. **rend()**返回链表第一元素的下一位置的后向指针


```
list<int>::reverse_iterator it = list1.rend(); // *(--riter) = 1
```
10. **push_back()**增加一元素到链表尾


```
list1.push_back(4)         // list1(1,2,3,4)
```
11. **push_front()**增加一元素到链表头


```
list1.push_front(4)        // list1(4,1,2,3)
```
12. **pop_back()**删除链表尾的一个元素


```
list1.pop_back()           // list1(1,2)
```
13. **pop_front()**删除链表头的一元素


```
list1.pop_front()          // list1(2,3)
```

14. clear()删除所有元素

```
list1.clear(); // list1 空了, list1.size() = 0
```

15. erase()删除一个元素或一个区域的元素(两个重载函数)

```
list1.erase(list1.begin());           // list1(2,3)
list1.erase(++list1.begin(), list1.end()); // list1(1)
```

16. c.remove(val): 删除容器 c 中所有值为 val 的元素

```
list 对象 L1(4,3,5,1,4)
L1.remove(4);           // L1(3,5,1);
```

17. c.remove_if(op):删除容器 c 中所有使 op 为 true 的元素

```
// 小于 2 的值删除
bool myFun(const int& value) { return (value < 2); }
list1.remove_if(myFun); // list1(3)
```

18. empty()判断是否链表为空

```
bool bRet = L1.empty(); // 若 L1 为空, bRet = true, 否则 bRet = false。
```

19. max_size()返回链表最大可能长度

```
list<int>::size_type nMax = list1.max_size(); // nMax = 1073741823
```

20. size()返回链表中元素个数

```
list<int>::size_type nRet = list1.size(); // nRet = 3
```

21. resize()重新定义链表长度(两重载函数)

```
list1.resize(5) // list1 (1,2,3,0,0)用默认值填补
list1.resize(5,4) // list1 (1,2,3,4,4)用指定值填补
```

22. reverse()反转链表

```
list1.reverse(); // list1(3,2,1)
```

23. sort()对链表排序, 默认升序(可自定义回调函数)

- c.sort(): 以操作符< (升序) 来排序容器 c 中所有元素
- c.sort(op): 以运算符 op 来排序容器中所有元素

```
设 list 对象 L1(4,3,5,1,4)
L1.sort();           // L1(1,3,4,4,5)
L1.sort(greater<int>()); // L1(5,4,4,3,1)
```

24. merge()合并两个有序链表并使之有序

- c1.merge(c2): 假定容器 c1 和 c2 包含的元素都已经过排序(sorted with operation <), 该接口把 c2 容器中的所有元素移动到 c1 中并且也按照排序规则排序。
- c1.merge(c2, op): 假定容器 c1 和 c2 包含的元素都已经过排序(sorted with op),

该接口把 `c2` 容器中的所有元素移动到 `c1` 中并且也按照排序规则排序。

- `c.reverse()`: 反序容器中的元素次序

// 升序

```
list1.merge(list2);           // list1(1,2,3,4,5,6) list2 现为空
```

// 降序

```
L1(3,2,1), L2(6,5,4)
```

```
L1.merge(L2, greater<int>()); // list1(6,5,4,3,2,1) list2 现为空
```

25. splice()对两个链表进行拼接(三个重载函数) 拼接后第二个链表清空。要求参与拼接的链表相容

- `c1.splice(pos, c2)`: 把容器 `c2` 中的所有元素移动到容器 `c1` 的迭代器位置 `pos` 之前。

- `c1.splice(pos, c2, c2pos)`: 把容器 `c2` 中的 `c2pos` 位置的元素移动到容器 `c1` 的 `pos` 位置之前。(`c1` 和 `c2` 可以是同一个容器)。

- `c1.splice(pos, c2, c2beg, c2end)`: 把容器 `c2` 中的 `[c2beg, c2end)` 范围内的所有元素移动到容器 `c1` 的 `pos` 位置之前 (`c1` 和 `c2` 可以是同一个容器)。

```
list1.splice(++list1.begin(),list2);
```

```
// list1(1,4,5,6,2,3) list2 为空
```

```
list1.splice(++list1.begin(),list2,list2.begin());
```

```
// list1(1,4,2,3); list2(5,6)
```

```
list1.splice(++list1.begin(),list2,++list2.begin(),list2.end());
```

```
//list1(1,5,6,2,3); list2(4)
```

26. insert()在指定位置插入一个或多个元素(三个重载函数)

```
list1.insert(++list1.begin(),9); // list1(1,9,2,3)
```

```
list1.insert(list1.begin(),2,9); // list1(9,9,1,2,3)
```

```
list1.insert(list1.begin(),list2.begin(),--list2.end()); //list1(4,5,1,2,3);
```

27. swap()交换两个链表(两个重载)

```
list1.swap(list2); // list1 (4, 5, 6) list2 (1, 2, 3)
```

28. c.unique(): 删除容器中值相同位置相邻的额外元素, 只保留一个元素

```
L1(1,1,4,3,5,1)
```

```
L1.unique(); // L1(1,4,3,5,1)
```

29. c.unique(op): 删除容器中同时满足 `op` 条件位置相邻的额外元素, 只保留一个元素

```
bool same_integral_part (double first, double second)
```

```
{ return ( int(first)==int(second) ); }
```

```
L1.unique(same_integral_part);
```

§2.1.3.2 List 综合示例

为了更好地理解 List 容器的成员函数, 下面再列举两个例子。

【例 2-10】List 容器操作的综合例子之一

list 仍然包涵了 `erase()`, `begin()`, `end()`, `insert()`, `push_back()`, `push_front()` 这些基本成员函数。下面来演示一下 list 的其他函数功能, 涉及 `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();    //给 listn1 排序
    listn2.sort();    //给 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);
}
```

程序运行结果如图 2-10 所示。

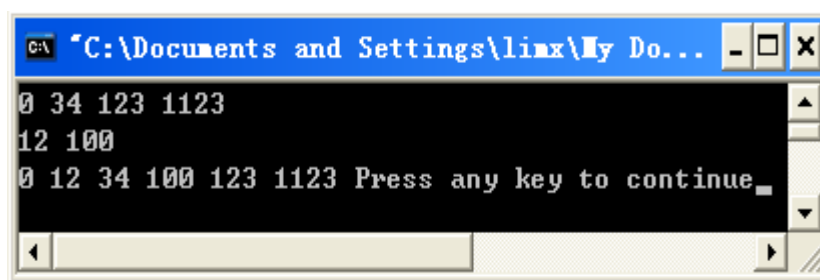


图 2-10 例 2-10 运行结果

【例 2-11】List 容器操作的综合例子之二

```

#include <iostream>
#include <list>
#include <algorithm>
#include <iterator>

using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "));
    cout << endl << "list2: ";
    copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "));
    cout << endl << endl;
}

int main()
{
    // create two empty lists
    list<int> list1, list2;

    // fill both lists with elements
    for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
    }
    printLists(list1, list2);

    // insert all elements of list1 before the first element with value 3 of list2
    // - find() returns an iterator to the first element with value 3
    list2.splice(find(list2.begin(), list2.end(), // destination position
        3),
        list1); // source list
    printLists(list1, list2);

    // move first element to the end
    list2.splice(list2.end(), // destination position
        list2, // source list
        list2.begin()); // source position
    printLists(list1, list2);

    // sort second list, assign to list1 and remove duplicates
    list2.sort();

```

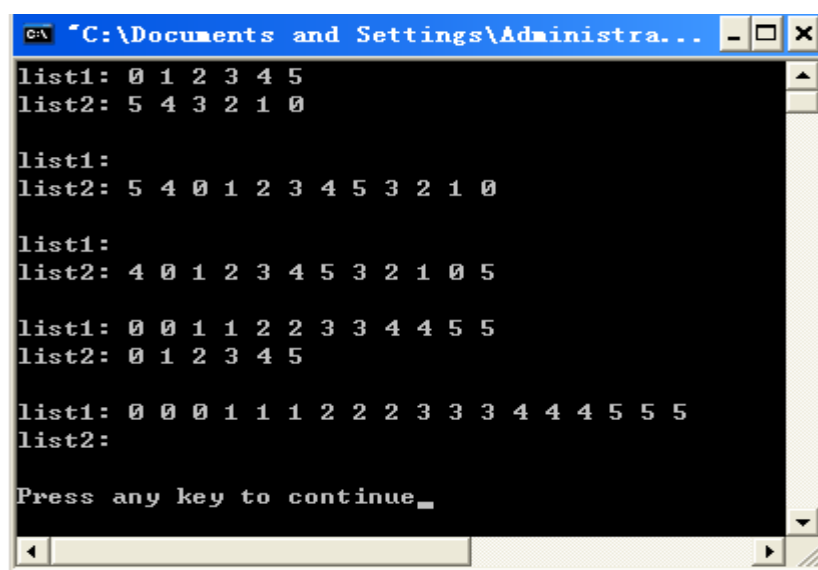
```

list1 = list2;
list2.unique();
printLists(list1, list2);

// merge both sorted lists into the first list
list1.merge(list2);
printLists(list1, list2);
}

```

程序运行结果如图 2-10 所示。



```

C:\Documents and Settings\Administra...
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:

Press any key to continue

```

图 2-11 例 2-11 运行结果

§2.2 关联式容器

对于关联式容器，一是要注意这类容器具有自动排序能力，即它们内部的元素总是时刻自动有序的；二是要注意使用 **Set**(集合) 和 **Multisets**(多重集合)之前，必须先包含头文件 **<set>**，而使用 **Map** 和 **Multimap** 之前，必须先包含头文件 **<map>**；三是要注意它们支持双向迭代器，但不支持随即迭代器，因此不能随即存取其元素。

一个集合 (**set**) 是一个容器，其所包含的元素的值是唯一的，而 **Multisets** 所包含的元素的值可以不唯一(即可以有重复元素)。注意，集合(**set**)与多重集合(**Multisets**)的内部数据组织是一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，因此，**set** 和 **Multiset** 内部的所有数据元素是自动有序的。

map (映射) 是经过排序的二元组集合，**map** 中的每个元素都是由两个值组成，其中 **key** 键值必须是唯一，而另一个值是该元素关联的数值。**multimap** (多重映射) 和映射 (**map**) 相似，但其中的键值可以有重复值。**map** 与 **multimap** 内部数据的组织也是一颗红黑树，所以它们内部所有的数据元素也都是自动有序的。

§2.2.1 操作 Set 和 Multisets 的成员函数及算法

(1)构造函数

定义一个元素为整数的集合 **intSet**，可以用：

```
set<int> intSet;
```

(2) 数据元素基本操作方面的成员函数

- 插入元素 i: `intSet.insert(i);`
- 删除元素 i (如果存在): `intSet.erase(i);`
- 判断元素 i 是否属于集合: `if (intSet.find(i) != intSet.end()) ...`
- 返回集合元素的个数: `intSet.size();`
- 判断集合是否空: `bool empty() const;`
- 将集合清为空集: `intSet.clear();`

问题: 如何判断一个元素是否在 **Set** 中?

方法 1: 用成员函数 `count()` 来判定关键字是否出现, 若出现返回值是 1, 否则返回 0。
比如查找关键字 1 是否出现:

```
int index = intSet.count(1);
```

方法 2: 用成员函数 `find()` 来定位数据元素出现位置, 它返回一个迭代器, 当 `set` 中包含该数据元素时, 返回数据元素所在位置的迭代器; 如果不包含, 返回的迭代器等于 `end` 函数返回的迭代器。比如查找关键字 1 是否出现:

```
if (intSet.find(1) == intSet.end())
    cout<<"Not Find "<<1<<endl;
else
    cout<<" Find value "<<1<<endl;
```

(3) 遍历方面的成员函数

- `iterator begin();` //返回首元素的迭代器指针
- `iterator end();` //返回尾元素后的迭代器指针, 而不是尾元素的迭代器指针
- `reverse_iterator rbegin();` //返回尾元素的逆向迭代器指针, 用于逆向遍历容器
- `reverse_iterator rend();` //返回首元素前的逆向迭代器指针, 用于逆向遍历容器

(4) 其它操作方面的成员函数

- `const_iterator lower_bound(const Key& key);` //返回容器元素等于 `key` 迭代指针,
//否则返回 `end()`
- `const_iterator upper_bound(const Key& key);`
- `int count(const Key& key) const;` //返回容器中元素值等于 `key` 的元素个数
- `pair<const_iterator, const_iterator> equal_range(const Key& key) const;` // 返回
//容器中元素值等于 `key` 的迭代指针[`first`, `last`)
- `const_iterator find(const Key& key) const;` //查找功能返回元素值等于 `key` 迭代器指针
- `void swap(set& s);` //交换单集合元素
- `void swap(multiset& s);` //交换多集合元素

(5) 集合的算法

STL 中的算法包含在 `<algorithm>` 头文件中, 集合的算法也包含在该头文件中。

- 集合的并: `set_union`
- 集合的交: `set_intersection`
- 集合的差: `set_difference`

§2.2.2 Set 和 Multisets 应用实例

【例 2-12】

```
#include <iostream>
#include <set>
using namespace std;

int main(){
    typedef std::set<int> IntSet;
    IntSet coll;    // set container for int values
    coll.insert(3); //所有关联式容器都提供一个 insert()函数，用以安插新元素，且
                    //自动完成排序

    coll.insert(1);
    coll.insert(5);
    coll.insert(4);
    coll.insert(1);
    coll.insert(6);
    coll.insert(2);

    IntSet::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

运行结果如图 2-12 所示。

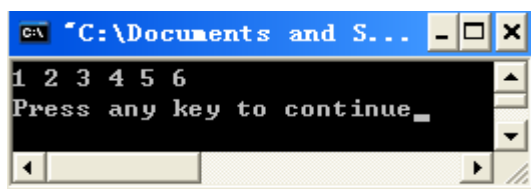


图 2-12 例 2-12 运行结果

§2.2.3 操作 Maps 和 Multimaps 的成员函数

`map` 是 STL 的一个关联容器，它提供一对一（其中第一个可以称为关键字，每个关键字只能在 `map` 中出现一次，第二个可称为该关键字对应的值，即即一个数据元素为 `key/value` 对的形式）的数据处理能力。`map` 内部数据的组织也是一棵红黑树，所以 `map` 内部所有的数据元素都是自动有序的。

那么什么是一对一的数据映射？比如一个班级中，每个学生的学号跟他的姓名就存在着——映射的关系，这个模型用 `map` 可以轻易描述，很明显学号用 `int` 描述，姓名用 `char *` 字符串描述，如下面的 `map` 描述代码：

```
map<int, char*> mapStudent;
```

一、STL 中的通用工具 `pair` 与 `make_pair` 的用法

C++ STL 预定义了几个通用工具类(或结构体)、函数等，结构体 `pair` 与函数 `make_pair()`

就是其中的两个。它们定义在通用工具头文件<utility>内，使用它们之前必需包含此头文件。

1、pair 的应用

Pair 的用途是将 2 个数据构造为一个数据元素(通过其构造函数)。当有这样的需求时就可以使用 pair 来构造，如 STL 中的容器 map 就是将 key 和 value 放在一起作为一个元素来保存的，因此，构造 map 的数据元素时就要用到 pair。pair 的另一个应用是，当一个函数需要返回 2 个值的时候，可以选择 pair 这种数据类型。pair 的实现是一个结构体，主要的两个成员变量是 first 和 second。因定义 pair 是使用 struct 不是 class，所以可以直接使用 pair 的成员变量(因为 struct 的成员都是 public 成员)。

std::pair 模型:

```
template <class T1, class T2>
struct pair
{
    T1    first;    //key 值部分
    T2    second;   //value 部分

    pair(const T1 a, const T2 b) : first(a), second(b) { } //构造函数
};
```

如：std::pair<int, float>(3, 1.1);将构造一个视为一个数据元素整体来对待的数对 (3, 1.1)，其中 3 为 int 型，1.1 为 float 型。

2、便捷函数 make_pair()

便捷函数 make_pair()的作用也是将 2 个数据构造为一个数据元素(make_pair()的返回值)。make_pair()的函数模版声明如下：

```
template <class T1, class T2>
pair<T1,T2> make_pair (const T1 x, const T2 y)
{
    return pair<T1,T2>(x, y); //调用 pair 类的构造函数
}
```

例如：std::make_pair(42, '@');与 std::pair<int, char>(42, '@');都是返回一个由(42, '@')组成的数对(42, '@')，显然，std::make_pair(42, '@')更简便，而 pair 在有些编译器下不允许这么简便写。

二、操作 Maps 和 Multimaps 的成员函数

(1)构造函数

map 共提供了 6 个构造函数，通常用图 2-13 所示的方法构造一个 Map。

```
map<int, char*> mapStudent;
```

关键字 值 map容器对象

图 2-13 构造 map 的常用方法

(2)数据的插入的三种方法

第一种方法：用 insert 函数插入 pair 数据

```
mapStudent.insert(pair<int, char*>(1, "student_one"));
```

第二种方法：用 insert 函数插入 value_type 数据

```
mapStudent.insert(map<int, char*>::value_type(1, "student_one"));
```

第三种方法：用数组方式插入数据

```
mapStudent[1] = "student_one";
```

```
mapStudent[2] = "student_two";
```

(3) 数据的删除

//如果要删除数据元素 1,用迭代器删除

```
map<int, string>::iterator iter;
```

```
iter = mapStudent.find(1);
```

```
mapStudent.erase(iter);
```

(4) 数据的查找

第一种方法：用 count 函数来判定关键字是否出现，返回 1，元素存在；否则返回 0。

比如查找关键字 1 是否存在？

```
int index = mapStudent.count(1);
```

第二种方法：用 find 函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器，如果 map 中没有要查找的数据，它返回的迭代器等于 end 函数返回的迭代器：

```
map<int, char*>::iterator iter;
```

```
iter = mapStudent.find(1);
```

```
if (iter != mapStudent.end())
```

```
    cout<<"Find, the value is "<<iter->second<<endl;
```

```
else
```

```
    cout<<"Do not Find"<<endl;
```

(5) 数据的其它基本操作

➤ 返回 map 元素的个数： mapStudent.size();

➤ 将 map 清为空集： mapStudent.clear();

➤ 判断 map 是否为空： mapStudent.empty();

➤ 数据的遍历： 比如用数组方式

```
int nSize = mapStudent.size();
```

```
for(int nIndex = 0; nIndex < nSize; nIndex++)
```

```
{
```

```
    cout<<mapStudent[nIndex]<<endl;
```

```
}
```

(6)其它操作函数

•void swap(map& s); //交换单映射元素

•void swap(multimap& s); //交换多映射元素

(7)特殊函数

- reference operator[](const Key& k); //仅用在单映射 map 类中，可以以数组的形式给映射添加键---值对，并可返回值的引用

§2.2.4 Maps 和 Multimaps 应用实例

【例 2-13】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 maplter;
    //char 是键的类型, int 是值的类型, less<char>要求升序
    //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(maplter=map1.begin();maplter!=map1.end();++maplter)
        cout<<" "<<(*maplter).first<<": "<<(*maplter).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<<endl;
    return 0;
}
```

程序运行结果如图 2-14 所示。

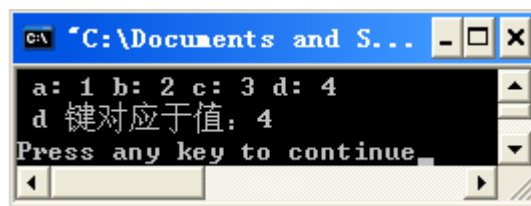


图 2-14 例 2-13 运行结果

从以上例程中可以看到 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。下面再看看 multimap:

【例 2-14】

```
#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 beautiful 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;
}

```

程序运行结果如图 2-15 所示。

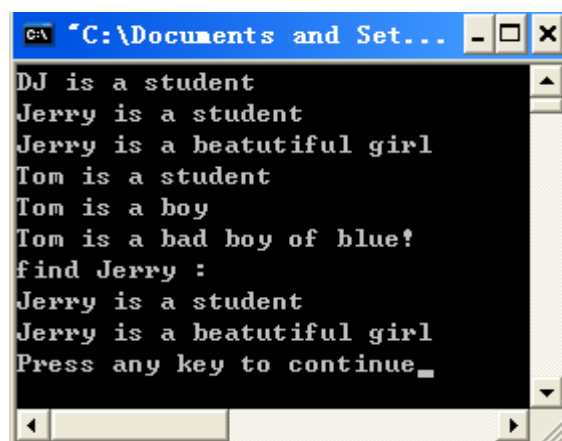


图 2-15 例 2-14 运行结果

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

【例 2-15】

```

#include <iostream>
#include <map>

#include <string>
using namespace std;

int main(){
    // type of the collection
    typedef multimap<int,string> IntStringMMap;

    IntStringMMap coll;          // container for int/string values

    // insert some elements in arbitrary order
    // - a value with key 1 gets inserted twice
    coll.insert(make_pair(5,string("tagged")));
    coll.insert(make_pair(2,string("a")));
    coll.insert(make_pair(1,string("this")));
    coll.insert(make_pair(4,string("of")));
    coll.insert(make_pair(6,string("strings")));
    coll.insert(make_pair(1,string("is")));
    coll.insert(make_pair(3,string("multimap")));

    /* print all element values
    * - iterate over all elements
    * - element member second is the value
    */
    IntStringMMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << pos->second << ' '; //输出 pos 所指数据元素的 second 部分
    }
    cout << endl;
}

```

程序运行结果如图 2-16 所示。

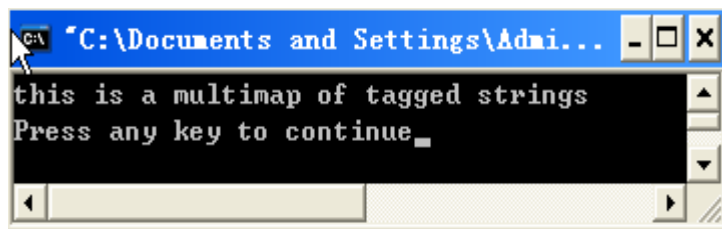


图 2-16 例 2-15 运行结果

【例 2-16】

```

#include <iostream>

```

```

#include <utility>
#include <string>
using namespace std;

int main () {
    pair <string,double> product1 ("tomatoes",3.25);
    pair <string,double> product2;
    pair <string,double> product3;

    product2.first = "lightbulbs";    // type of first is string
    product2.second = 0.99;          // type of second is double
    product3=make_pair(string("shoes"),20.0);

    cout << "The price of " << product1.first << " is $" << product1.second << "\n";
    cout << "The price of " << product2.first << " is $" << product2.second << "\n";
    cout << "The price of " << product3.first << " is $" << product3.second << "\n";
    return 0;
}

```

本程序运行结果如图 2-17 所示。

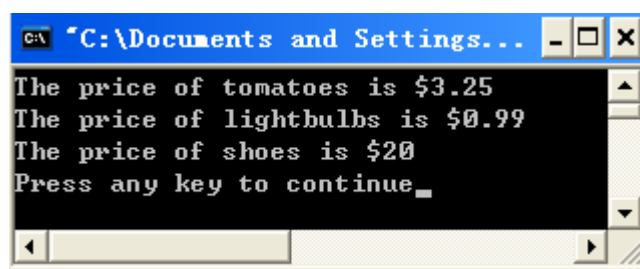


图 2-17 例 2-16 运行结果

§2.3 特殊容器——顺序容器配接器

为了满足特殊的需要，C++标准程序库还提供了一些特殊的、依据 `vector`、`list` 等基本顺序式容器预先定义好的容器配接器（Containers Adapters）。容器配接器包括：`Stacks`（栈）、`Queues`（队列）和 `Priority Queues`（优先级队列）。所谓容器配接器，就是使一种已存在的容器类型在实现时采用另一种的抽象类型的工作方式。

§2.3.1 Stack(栈)

`Stack` 容器是一个先进后出（FILO）的数据结构。值得注意的是迭代器不能在堆栈中使用，因为只有栈顶的元素才可以访问。使用 `stack` 时必须包含头文件 `<stack>`，并使用统一命名空间。

1、构造函数

构造函数用于声明一个堆对象，如：

```

stack();           //声明一个空栈
stack(copy);       //用一个栈对象 copy 初始化新栈
stack<int> s1;

```

```
stack<string> s2;
```

`stack` 模板类需要 2 个模板参数，第一个为元素类型，第二个为容器类型。其中元素类型是必要的。而容器类型缺省时，默认为 `deque`。

2、成员函数

所有的这些操作可以被用于堆栈。相等指堆栈有相同的元素并有着相同的顺序。

① `empty()`

语法: `bool empty();`

功能: 判断栈是否为空。如空返回 `true`, 否则返回 `false`。

② `pop()`

语法: `void pop();`

功能: 退栈。`pop()` 函数移除栈顶元素。

③ `push(x)`

语法: `void push(const TYPE &val);`

功能: `push(x)` 函数将 `x` 压栈, 使其成为栈顶元素。

④ `size()`

语法: `int size();`

功能: `size()` 函数返回当前堆栈中的元素数目。

⑤ 读取栈顶元素 `top()`

语法: `TYPE & top();`

功能: `top()` 函数栈顶元素的引用。

【例 2-17】

```
#include <iostream>
```

```
#include <stack>
```

```
using namespace std;
```

```
void main(){
```

```
    stack<int> st;
```

```
    // push three elements into the stack
```

```
    st.push(1);
```

```
    st.push(2);
```

```
    st.push(3);
```

```
    // pop and print two elements from the stack
```

```
    cout << st.top() << ' ';
```

```
    st.pop();
```

```
    cout << st.top() << ' ';
```

```
    st.pop();
```

```
    // modify top element
```

```
    st.top() = 77;
```

```

// push two new elements
st.push(4);
st.push(5);

// pop one element without processing it
st.pop();

// pop and print remaining elements
while (!st.empty()) {
    cout << st.top() << ' ';    st.pop(); }
cout << endl;
}

```

本程序运行结果如图 2-18 所示。

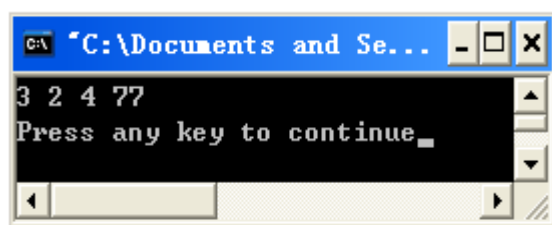


图 2-18 例 2-17 运行结果

§2.3.2 Queue(队列)

Queue 实现的是一个先进先出(FIFO)队列。使用 `queue` 时必须包含头文件 `<queue>`，并使用统一命名空间。

1、构造函数

```

queue()           //声明一个空队列;
queue(mycont)     //声明一个队列，它的初始内容复制自 mycont;

```

2、成员函数

- ① `push()`: //将一个新元素加入到队列的末端;
- ② `pop()`: //弹出队列中的第一个元素，返回的是一个 `void`;
- ③ `front()`: //存取队列中的第一个元素，返回的是一个引用;
- ④ `back()`: //存取队列中的最后一个元素，返回的是一个引用;
- ⑤ `empty()`: //测试队列是否为空;
- ⑥ `size()`: //获得队列中元素的个数;

3、常见的六种比较操作同样适用

【例 2-18】

```

#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main(){

```

```

queue<string> q;

// insert three elements into the queue
q.push("These ");
q.push("are ");
q.push("more than ");

// read and print two elements from the queue
cout << q.front();
q.pop();
cout << q.front();
q.pop();

// insert two new elements
q.push("four ");
q.push("words!");

// skip one element
q.pop();

// read and print two elements
cout << q.front();
q.pop();
cout << q.front() << endl;
q.pop();
// print number of elements in the queue
cout << "number of elements in the queue: " << q.size()
    << endl;
}

```

本程序运行结果如图 2-19 所示。

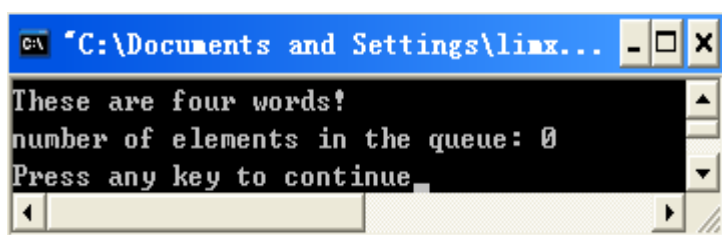


图 2-19 例 2-18 运行结果

§2.3.3 Priority Queues(优先级队列)

`priority_queue` 也是一个队列，不过该模版类可以保证每次通过 `pop`(出队)从它里面取得的元素都是剩下元素中优先级最高的，在每次通过 `push`(入队)添加元素时，整个序列都会在必要时进行重排，它使用堆算法保持序列的有序化。使用 `priority_queue` 时必须包含头文件 `<queue>`，并使用统一命名空间。

1、构造函数

- ① `priority_queue()` : // 声明一个按 `pred()` 排序的空优先队列;
- ② `priority_queue(pr)` : // 声明一个按函数对象 `pr` 排序的空优先队列;
- ③ `priority_queue(pr,mycont)` // 声明一个按 `pr` 排序的优先队列, 它的初始内容
 // 是从 `mycont` 中复制过来并排序的;
- ④ `priority_queue (first,last)` : // 声明一个按 `pred()` 排序的优先队列, 初始内容从
 // 指定序列中复制过来的;
- ⑤ `priority_queue(first,last,pr)` : // 声明一个按 `pr` 排序的优先队列, 它的初始内容是
 // 从指定的序列中复制过来并排序的;
- ⑥ `priority_queue(first,last,mycont,pr)` : // 声明一个按 `pr` 排序的优先队列, 它的初内
 // 容复制从 `mycont` 中指定的段并排序;

2、priority_queue 的主要成员函数如下:

- ① `push()`: // 向优先队列中压入一个新元素;
- ② `pop()`: // 从优先队列中弹出它的顶端元素, 返回值是 `void`;
- ③ `top()`: // 存取优先队列中顶端的元素, 返回的是一个常量的引用;
- ④ `empty()`: // 测试优先队列是否为空; 获得优先队列中元素的个数;

3、常用的六种比较操作同样适用

【例 2-19】

```
#include <iostream>
#include <queue>
using namespace std;

int main()
{
    priority_queue<float> q;
    // insert three elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // read and print two elements
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // insert three more elements
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
    // skip one element
    q.pop();
```

```

// pop and print remaining elements
while (!q.empty()) {
    cout << q.top() << ' ';
    q.pop();
}
cout << endl;
}

```

本程序运行结果如图 2-20 所示。

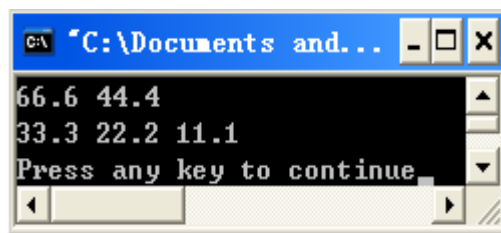


图 2-20 例 2-19 运行结果

§2.4 Strings 可以视为一种 STL 容器

STL 是个框架，除了提供标准容器外，也允许使用其它数据结构作为容器，如使用字符串或数组等作为容器。

Strings 可被视为以单个字符(characters)为元素的一种容器：若干个字符构成序列(即一个字符串)，可以在序列上来回移动遍历。因此，标准的 `string` 类也提供成员函数 `begin()` 和 `end()`，返回随机存取迭代器，可用来遍历整个 `string`。同时，`string` 还提供了诸如 `push_back()` 等成员函数。使用 `string` 类型，必须包含头文件 `<string>`。

1、string 类型的初始化(构造函数)

```

string s1;           //定义串 s1, s1 为空串
string s2(s1);       //用 s1 初始化 s2
string s3("abcdefg");
string s4(s3, 2);
string s5(s3.begin(), s3.begin()+2);
string s6(s3,2,3);
string s7(5,'x');
cin >> s1;
cout << s2 << endl; //输出 s2

```

2、string 对象基本操作的成员函数

`string` 对象提供的基本操作(成员函数)列于表 2-6 中。

3、string 应用举例

```

string s1("abc");
string s2("xyz");
s1+=s2+"1234";           //s1="abcxyz1234"

```

表 2-6 string 对象的基本操作

操作接口示例 s1、s2 均为 string 对象	说 明
s1=s2	将 s2 的副本赋值给 s1
s1.empty()	如果 s1 为空串，返回 true，否则返回 false
s1.size()	返回 s1 中字符的个数
s1[n]	返回 s1 中位置为 n 的字符，位置从 0 开始计
==, !=, <, <=, >, >=,	关系运算符保持原有的含义
s1+=s2 或 s1=s1+s2	将 s2 表示的串追加到 s1 的后面形成新串
s1.insert(pos, s2)	在下标为 pos 的元素前插入 s2 表示的串
s1.insert(pos, cp, len)	在下标为 pos 的元素前插入 cp 所指串的前 n 个字符 insert 操作还有很多用法，在这里不再一一列举
s1.c_str()	返回 s1 表示的串的首地址
s1.substr(pos, n);	返回一个 string 类型的字符串，包含 s1 中从下标 pos 开始连续 n 个字符
s1.append(s2)	将 s2 表示的串追加到 s1 的后面形成新串
s1.append(cp)	将字符指针 cp 指向的串追加到 s1 的后面形成新串
s1.replace(pos, len, s2)	删除 s1 中从下标 pos 开始的 len 个字符，并用 s2 替换之
s1.find(s2, pos)	从 s1 下标为 pos 的位置开始查找 s2，找到后返回 s2 第一次出现的位置，若没找到，返回值 string::npos
s1.rfind(s2, pos)	从 s1 下标为 pos 的位置开始查找 s2，找到后返回 s2 最后一次出现的位置，若没找到，返回值 string::npos
s1.find_first_of(s2,pos)	从 s1 下标为 pos 的位置开始查找 s2 中的任意字符，找到后返回第一次出现的位置。
s1.find_last_of(s2,pos)	从 s1 下标为 pos 的位置开始查找 s2 中的任意字符，找到后返回最后一次出现的位置。

```

bool tag = s1>s2 ? true:false;    //tag=0
int pos = 2;
s1.insert(pos, s2);                //s1 = "abxyzxyz1234"
s1.insert(pos, "12345", 4);        //s1 = "ab1234xyzxyz1234"
string s3 = s1.substr(pos, 6);     //s3 = "1234xy"
s3.append(s2);                    //s3 = "1234xyxyz"
s1.replace(pos, 4, "k");           //s1 = "abkxyzxyz1234"
int p = s1.find(s2, pos);          //p = 3
p = s1.rfind(s2);                 //p = 7
p = s1.find_first_of(s2);         //p = 3

```



```
p = s1.find_last_of(s2);    //p = 9
```

```
//拷贝到字符数组中
```

```
char * cp = new char [s1.size()+1];
```

```
strcpy(cp ,(char *) s1.c_str ());    //cp = "abkxyzxyz1234"
```

第3讲 STL 迭代器

§3.1 迭代器

迭代器 (Iterators) 是一个“可遍历容器内全部或部分元素”的对象。迭代器被用来指向容器中某个特定数据元素的位置 (如同下标), 其作用有点像指针, 但绝不是普通的指针。它是被泛化了的指针。迭代器多半作为参数传递给算法。

(1) 每种容器类型都定义了自己的迭代器类型, 如 vector 容器:

语句 `vector<int>::iterator iter;` 就定义了一个迭代器。由于 `iterator` 类是容器类的一个内嵌类, 所以这条语句是定义了一个名为 `iter` 的、`vector` 类的 `iterator` 类型的变量 (即 `iterator` 类的一个对象, 即迭代器 `iter`)。同时, 这条语句还指定 `vector` 的数据元素为 `int` 类型数据。

(2) 使用迭代器读取容器中的每一个数据元素, 如:

`vector<int> vector (10,1);` //创建一个含有 10 个整型数 1 为元素的向量容器 `vector`

```
//下面这条语句是将先前的 10 个数据元素分别改变为整型数 2
for(vector<int>::iterator iter=ivec.begin();iter!=ivec.end();++iter)
{
    *iter=2; //使用 * 访问迭代器所指向的元素
}
```

在 STL 中, 对迭代器本身的基本操作如表 3-1 所示。

表 3-1 对迭代器本身的基本操作

操作符	意 义
*	返回迭代器所指元素的值
++	将迭代器前进至下一个元素所在的位置
--	将迭代器退至上一个元素所在的位置
=	将某个元素的位置值赋给迭代器
==和!=	判断两个迭代器是否指向同一个位置
->	指向数据元素的某个字段域

(3) 各类容器都具有的两个最重要的成员函数

下面列出的两个函数是各类容器都具有的最重要的成员函数。在通过迭代器遍历容器的所有元素时要用到它们。

① **begin()** 让迭代器指向容器的第一个元素位置 (如果有元素的话);

② **end()** 让迭代器指向容器最后一个元素之后的那个位置。

【例 3-1】将 LIST 容器内的全部元素输出

```
#include <iostream>
```

```
#include <list>
```

```
using namespace std;
```

```
void main(){
```

```
    list<char> coll;    // list container for character elements
```

```
    // append elements from 'a' to 'z'
```

```
    for (char c='a'; c<='z'; ++c) {
```

```

        coll.push_back(c);
    }

    /* print all elements
       * - iterate over all elements
    */
    list<char>::const_iterator pos; // const_iterator 指定 pos 是一个“只读”迭代器
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}

```

程序运行结果如图 3-1 所示。

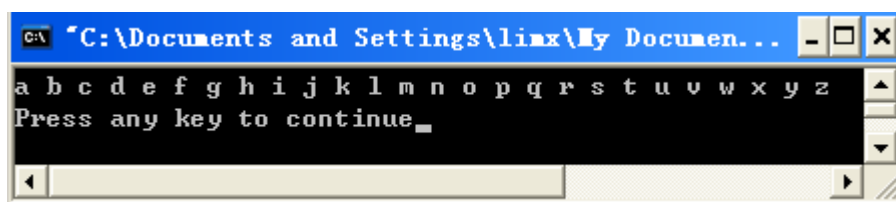


图 3-1 例 3-1 的运行结果

注：任何一种容器都可以定义三种类别的迭代器：

1、 container:: iterator

这种迭代器可以以“读/写”模式遍历容器诸元素，并且自身可以改变，例如：

```

for(vector<int>:: iterator citer=ivec.begin();citer!=ivec.end();citer++)
{
    cout<<*citer; //ok (读)
    *citer=3;      //ok (写)
}

```

2、 container:: const_iterator

这种迭代器只能以“读”模式遍历容器诸元素(元素值不可变)，并且自身可以改变，例如：

```

for(vector<int>::const_iterator citer=ivec.begin();citer!=ivec.end();citer++)
{
    cout<<*citer; //ok (读)
    *citer=3;      //error (写)
}

```

因此，不妨将这种迭代器叫做“指向常量的迭代器”。

3、 const container:: iterator

这种迭代器可以以“读/写”模式遍历容器诸元素(元素值可变)，但迭代器自身不可以改变，例如：

```

const vector<int>::iterator newiter=ivec.begin();
*newiter=11; //ok,可以修改指向容器的元素
newiter++;   // error, 这种迭代器本身不能被修改

```

因此，不妨将这种迭代器叫做“常迭代器”。

§3.2 迭代器的分类(Iterator Categorise)

根据 STL 中的分类, iterator 包括五类(注: 这些类型的迭代器非独立于容器):

1、Input Iterator(输入迭代器): 输入迭代器也可以称之为前向的只读访问器, 只能单步向前迭代元素(即只提供++操作), 不允许修改由该类迭代器引用的元素, 只可以用来从序列中读取数据, 如输入流迭代器。所有的容器的迭代器都具有输入迭代器的特征。

2、Output Iterator(输出迭代器): 输出迭代器也可以称之为前向的只写访问器, 该类迭代器和 Input Iterator 极其相似, 也只能单步向前迭代元素(即只提供++操作), 该类迭代器对元素只有写的能力, 只允许向序列中写入数据, 如输出流迭代器。

3、Forward Iterator(前向迭代器): 该类迭代器可以在一个正确的区间内进行读写操作。它拥有 Input Iterator 的所有特性和 Output Iterator 的部分特性, 以及单步向前迭代元素的能力。因此, 它既是输入迭代器又是输出迭代器, 并且只可以对序列进行前向的遍历。

4、Bidirectional Iterator(双向迭代器): 该类迭代器是在 Forward Iterator 的基础上还提供了单步向后迭代元素的能力, 在向前和向后两个方向上都可以对数据元素进行遍历。(即提供提供++操作, 又提供--操作)。

5、Random Access Iterator(随机访问迭代器): 该类迭代器也是双向迭代器, 它能够完成上面所有迭代器的工作。但它自己独有的特性就是可以像指针那样进行算术计算, 能够在序列中的任意两个位置之间进行跳转。

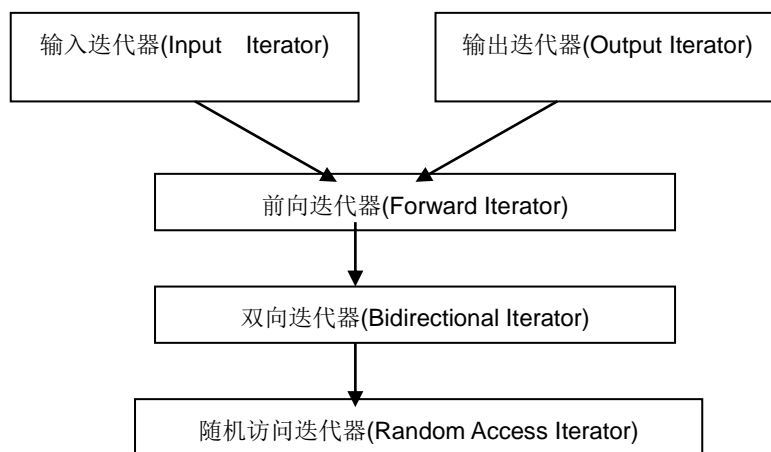


图 3-2 五种迭代器之间的关系

这五类迭代器的从属关系如图 3-2 所示, 其中箭头 B→A 表示 A 是 B 的强化类型, 也就是说如果一个算法要求 B, 那么 A 也可以应用于其中。例如, 一个前向迭代器肯定是输入迭代器, 也肯定是输出迭代器; 再如, 双向迭代器肯定是前向迭代器, 等等。各类容器支持的迭代器的类别如表 3-2 所示。各类迭代器允许的算术操作情况参见表 3-3。

§3.3 迭代器的算术操作

iterator 除了表 3-1 列出的能进行++、--等操作外, 还可以将 iter+n、iter-n 赋给一个新的 iteraor 对象。另外, 还可以使用一个 iterator 减去另外一个 iterator。表 3-3 列出了 iterator 允许的各种算术操作。例如:

```

const vector<int>::iterator newiter=ivec.begin(); //注意迭代器 newiter 自身不可改动
vector<int>::iterator newiter2=ivec.end();        //newiter2 迭代器可“读/写”
cout<<"\n"<<newiter2-newiter; //ok
  
```

表 3-2 各类容器支持(提供)的迭代器的类别

容器类别	支持的迭代器类别
vector	随机访问迭代器
deque	随机访问迭代器
list	双向迭代器
set	双向迭代器, 元素为常量
multiset	双向迭代器, 元素为常量
map	双向迭代器, key 为常量
multimap	双向迭代器, key 为常量
stack	容器适配器不支持迭代器
queue	容器适配器不支持迭代器
priority_queue	容器适配器不支持迭代器

表 3-3 iterator 的算术操作

千万要注意：每种迭代器均可进行包括表中前一种迭代器可进行的操作！			
序号	迭代器类型	迭代器操作	操作说明
1	所有迭代器	p++	后置自增迭代器
		++p	前置自增迭代器
2	输入迭代器	*p	复引用迭代器, 作为右值
		p=p1	将一个迭代器赋给另一个迭代器
		p==p1	比较迭代器的相等性
		p!=p1	比较迭代器的不等性
3	输出迭代器	*p	复引用迭代器, 作为左值
		p=p1	将一个迭代器赋给另一个迭代器
4	正(前)向迭代器	提供输入输出迭代器的所有功能	
5	双向迭代器（具有正向迭代器的所有功能）	--p	前置自减迭代器
		p--	后置自减迭代器
6	随机迭代器（具有双向迭代器的所有功能）	p+=i	将迭代器递增 i 位
		p-=i	将迭代器递减 i 位
		p+i	在 p 位加 i 位后的迭代器
		p-i	在 p 位减 i 位后的迭代器
		p[i]	返回 p 位元素偏离 i 位的元素引用
		p<p1	如果迭代器 p 的位置在 p1 前, 返回 true, 否则返回 false
		p>p1	如果迭代器 p 的位置在 p1 后, 返回 true, 否则返回 false
		p>=p1	p 的位置在 p1 的后面或同一位置时返回 true, 否则返回 false

§3.4 迭代器的区间

STL 算法的形参中常常包括一对输入迭代器，由它们构成容器数据元素的一个区间，该区间是容器数据元素的一个序列。STL 算法常以迭代器的区间作为输入，传递输入数据，算法将在该区间上进行操作。

设 $p1$ 和 $p2$ 是两个输入迭代器，用 $[p1, p2)$ 的形式表示它们构成容器数据元素的一个区间。该区间不包括 $p2$ 所指向的那个元素，所以它是个半开区间。那么，若 $p1$ 经过 n 次 ($n > 0$) 自增($++$)操作后满足 $p1 == p2$ ，区间 $[p1, p2)$ 就是一个合法的区间。

【例 3-2】

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;
    list<int>::iterator pos;

    // insert elements from 20 to 40
    for (int i=20; i<=40; ++i) {
        coll.push_back(i);
    }

    /* find position of element with value 3
       * - there is none, so pos gets coll.end()
    */
    pos = find (coll.begin(), coll.end(), // range
               3);                       // value

    /* reverse the order of elements between found element and the end
       * - because pos is coll.end() it reverses an empty range
    */
    reverse (pos, coll.end());

    // find positions of values 25 and 35
    list<int>::iterator pos25, pos35;
    pos25 = find (coll.begin(), coll.end(), // range
                 25);                       // value
    pos35 = find (coll.begin(), coll.end(), // range
                 35);                       // value

    /* print the maximum of the corresponding range
       * - note: including pos25 but excluding pos35
    */
}
```

```

*/
cout << "max: " << *max_element (pos25, pos35) << endl;

// process the elements including the last position
cout << "max: " << *max_element (pos25, ++pos35) << endl;
}

```

运行结果如图 3-3 所示。

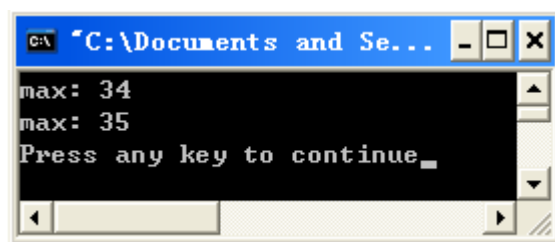


图 3-3 例 3-2 运行结果

§3.5 迭代器的辅助函数

C++标准程序库为迭代器提供了 `advance()`、`distance()`和 `Iter_swap()`三个辅助函数。

§3.5.1 函数 `advance(p, n)`

函数 `advance(p, n)`的作用是让迭代器 `p` 前进或后退 `n` 个元素，即：

若 `p` 是输入型迭代器，则使迭代器 `p` 执行 `n` 次自增操作 ($n > 0$)，表示让 `p` 前进 `n` 个元素；若 `p` 是输入型迭代器或随机访问型迭代器，则 `n` 可以取负值，表示让 `p` 后退 `n` 个元素。

【例 3-3】函数 `advance()`的应用

```

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main(){
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i); //在尾部插入元素
    }

    list<int>::iterator pos = coll.begin();

    // print actual element
    cout << *pos << endl;
    // step three elements forward

```

```

advance (pos, 3);

// print actual element
cout << *pos << endl;

// step one element backward
advance (pos, -1);

// print actual element
cout << *pos << endl;
}

```

运行结果如图 3-4 所示。

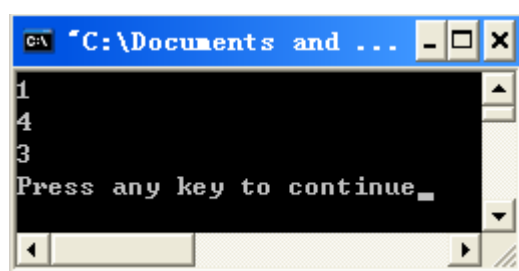


图 3-4 例 3-3 运行结果

§3.5.2 函数 distance(first, last)

函数 `distance(first, last)` 的作用是计算两个迭代器 `first` 和 `last` 之间的距离。即当 `first < last` 时，对 `first` 执行多少次“++”操作后才能够使得 `first == last` 成立。

【例 3-4】函数 `distance()` 的应用

```

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main(){
    list<int> coll;

    // insert elements from -3 to 9
    for (int i=-3; i<=9; ++i)
        coll.push_back(i);

    // search element with value 5
    list<int>::iterator pos;
    pos = find (coll.begin(), coll.end(), 5);
    if (pos != coll.end())
        // process and print difference from the beginning

```



```

        cout << "difference between beginning and 5: "
              << distance(coll.begin(),pos) << endl;
    else
        cout << "5 not found" << endl;
}

```

运行结果如图 3-5 所示。

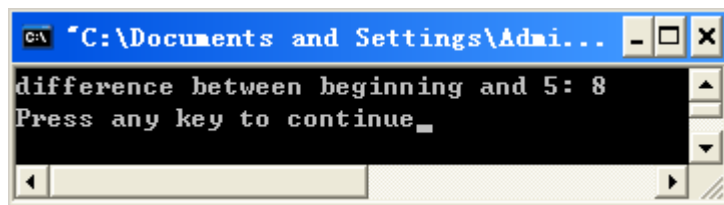


图 3-5 例 3-4 运行结果

§3.5.3 函数 Iter_swap(p1,p2)

函数 Iter_swap(p1,p2)的作用是交换迭代器 p1 和 p2 所指数据元素的值。

【例 3-5】函数 Iter_swap()的应用

```

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

```

/ PRINT_ELEMENTS()是一个泛型函数，可以打印一个字符串(也可以不指定)，然后打印容器的全部元素。容器名提供给第一个参数；第二个参数是可有可无的前缀字，用来在打印时放在所有元素之前。*

```

*/
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

```

```

int main()
{
    list<int> coll;

    //insert elements from 1 to 9

```

```

    for (int i=1; i<=9; ++i)
        coll.push_back(i);

    PRINT_ELEMENTS(coll);

    // swap first and second value
    iter_swap (coll.begin(), ++coll.begin());

    PRINT_ELEMENTS(coll);

    // swap first and last value
    iter_swap (coll.begin(), --coll.end());

    PRINT_ELEMENTS(coll);
}

```

运行结果如图 3-6 所示。

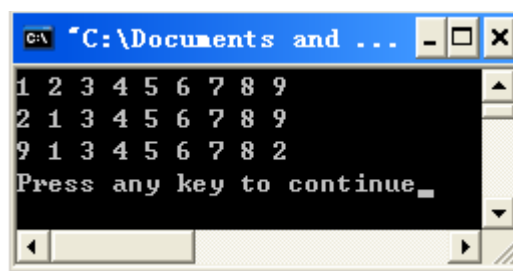


图 3-6 例 3-5 运行结果

§3.6 迭代器适配器(Iterator Adapters)

C++标准程序库提供了数个预先定义的特殊迭代器，称为迭代器适配器(Iterator Adapters)。如 Insert Iterators（安插型迭代器）、Stream Iterators（流迭代器）、Reverse Iterators（逆向迭代器）。

§3.6.1 Insert Iterators（安插型迭代器）

安插型迭代器可以使算法以插入(insert)方式而非覆盖(overwrite)方式动作。同时，它会使目标空间的大小按需要成长。有如下三种预定义好的安插型迭代器：

①back_inserter（安插于容器的最尾端）

这是一个追加动作，它只适用于具有 push_back()成员函数的 vector、deque 和 list 三种容器。

②front_inserter（安插于容器的最前端）

元素的安插位置与安插顺序相反。它只适用于具有 push_front()成员函数的 deque 和 list 两种容器。

③inserter（一般性安插器）

它的作用是将元素插入到“初始化时接收的第二个参数”所指位置的前方。它主要用于关联式容器。

【例 3-6】

```

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <algorithm>
using namespace std;

int main(){
    list<int> coll1;
    list<int>::iterator pos1;

    for (int i=1; i<=9; ++i)    // insert elements from 1 to 9 into the first collection
        coll1.push_back(i);

    cout<<"List:"<<endl;    //输出 list 内的元素
    for (pos1=coll1.begin(); pos1!=coll1.end(); ++pos1)
        cout << *pos1 << ' ';
    cout<<endl<<endl;

    // copy the elements of coll1 into coll2 by appending them
    vector<int> coll2;
    vector<int>::iterator pos2;
    copy (coll1.begin(), coll1.end(),    // source
          back_inserter(coll2));    // destination
    cout<<"vector:"<<endl;    //输出 vector 内的元素
    for (pos2=coll2.begin(); pos2!=coll2.end(); ++pos2)
        cout << *pos2 << ' ';
    cout<<endl<<endl;

    // copy the elements of coll1 into coll3 by inserting them at the front
    // - reverses the order of the elements
    deque<int> coll3;
    deque<int>::iterator pos3;
    copy (coll1.begin(), coll1.end(),    // source
          front_inserter(coll3));    // destination

    cout<<"Deque:"<<endl;    //输出 Deque 内的元素
    for (pos3=coll3.begin(); pos3!=coll3.end(); ++pos3)
        cout << *pos3 << ' ';
    cout<<endl<<endl;

    // copy elements of coll1 into coll4
    // - only inserter that works for associative collections
    set<int> coll4;

```

```

set<int>::iterator pos4;
copy (coll1.begin(), coll1.end(),      // source
      inserter(coll4,coll4.begin())); // destination
cout<<"Set:"<<endl; //输出 set 内的元素
for (pos4=coll4.begin(); pos4!=coll4.end(); ++pos4)
    cout << *pos4 << ' ';
    cout<<endl<<endl;
}

```

程序运行结果如图 3-7 所示。

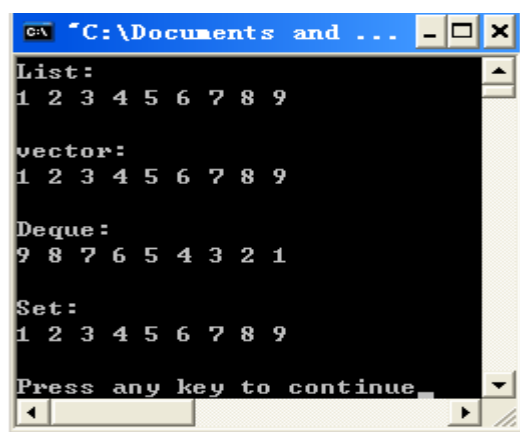


图 3-7 例 3-6 运行结果

§3.6.2 Stream Iterators（流迭代器）

流迭代器是一种迭代适配器，它们是类模板。由于它们独立于容器，所以使用时要包含头文件<iterator>。

流迭代器包括 `istream_iterator`(输入流迭代器)和 `ostream_iterator`(输出流迭代器)。`istream_iterator` 是用来从 `input stream` 流对象中连续读取指定的某种类型的数据元素。`ostream_iterator` 是用来向 `output stream` 流对象中连续写入指定的某种类型的数据元素。

1、ostream_iterator

`ostream_iterator` 将赋值操作转化为对输出运算符“<<”的操作。`ostream_iterator` 类的构造函数如下：

`ostream_iterator<T>(ostream):` 为 `ostream`(输出流)产生一个 `ostream_iterator`

`ostream_iterator<T>(ostream, delim):` 为 `ostream`(输出流)产生一个 `ostream_iterator`，各元素间以 `delim` 为分隔符（注意，`delim` 的数据类型是 `const char*`，即字符串）。

产生 `ostream_iterator` 对象时，必须提供一个 `output stream` 流对象(通常是“`cout`”)作为参数，`ostream_iterator` 将会把要输出的元素值写到 `output stream` 流对象上，如“`cout`”上。另一个参数 `delim` 可有可无。

2、istream_iterator

`istream_iterator` 将赋值操作转化为对输出运算符“>>”的操作。`istream_iterator` 类的构造函数如下：

`istream_iterator<T>():` 缺省构造函数将产生一个 *end-of-stream* 迭代器。判定输入是否结束。

`istream_iterator<T>(istream):` 为 `istream`(输入流)产生一个 `istream_iterator` 对象（可

能立刻读取第一个元素)。

产生 `istream_iterator` 对象时, 必须提供一个 `input stream` 流对象(通常是“`cin`”)作为参数。

【例 3-7】从键盘输入两个字符串: “aaaaaa”和“bbbbbb”, 然后输出出来。

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>    //使用独立于容器的迭代器时, 必须包含该头文件
using namespace std;

int main(){
    vector<string> coll;

    /* read all words from the standard input
     * - source: all strings until end-of-file (or error)
     * - destination: coll (inserting)
     */
    copy (istream_iterator<string>(cin),      // start of source
          istream_iterator<string>(),         // end of source
          back_inserter(coll));               // destination

    // sort elements
    sort (coll.begin(), coll.end()); //对所有元素排序

    /* print all elements without duplicates
     * - source: coll
     * - destination: standard output (with newline between elements)
     */
    unique_copy (coll.begin(), coll.end(),      // source
                 ostream_iterator<string>(cout, "\n")); // destination
}
```

程序运行结果如图 3-8 所示。

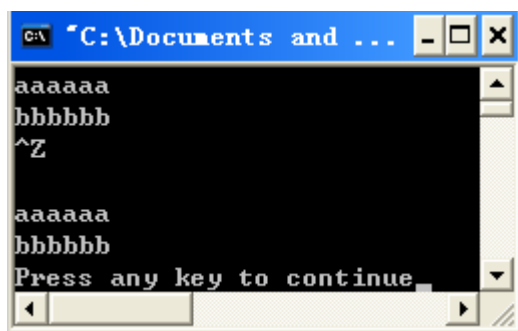


图 3-8 例 3-7 运行结果

对【例 3-7】做几点解释：

1、对下面这条 copy 语句中的三个参数解释如下：

```
copy (istream_iterator<string>(cin),    istream_iterator<string>(), back_inserter(coll));
```

① `istream_iterator<string>(cin)`——用 `cin` 作为参数，调用 `istream_iterator` 的有参构造函数，生成一个 `istream_iterator` 迭代器。模板参数 `string` 表示生成的这个迭代器专门用于读取该类型的元素。因此，`istream_iterator<string>(cin)` 是从标准输入流对象 `cin` 连续读取字符串；

② `istream_iterator<string>()`——调用 `istream_iterator` 的缺省构造函数，产生一个代表“流结束符号”的迭代器，它代表的含义是不能再从输入流中读取任何东西，实际上，它是输入的终点，即用作输入区间的终点；

③ `back_inserter(coll)`——借助它将读取到的数据拷贝并插入到容器 `coll` 中。

2、下面这条语句：

```
unique_copy (coll.begin(), coll.end(),
            ostream_iterator<string>(cout, "\n"));
```

① 算法 `unique_copy()` 的作用是将 `coll` 容器中的所有元素拷贝到目的端 `cout`，处理过程中算法 `unique_copy()` 会消除比邻的重复值；

② `ostream_iterator<string>(cout, "\n")`——调用有参构造函数生成一个输出流迭代器对象，并指定模板参数类型为 `string`。“`\n`”的作用是指元素之间的分隔符是换行(还可以是其它的分隔符，如“`\t`”或“ ”等)。

§3.6.3 Reverse Iterators（逆向迭代器）

`reverse iterators`（逆向迭代器）的作用是以逆向方式进行所有操作，即将递增运算转换为递减运算，将递减运算转换为递增运算。所有容器都可以通过其成员函数 `rbegin()` 和 `rend()` 产生 `reverse iterators`。

【例 3-8】

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // print all element in reverse order
    copy (coll.rbegin(), coll.rend(),    // source
          ostream_iterator<int>(cout, " ") // destination
    );
    cout << endl;
```

```
}
```

程序运行结果如图 3-9 所示。

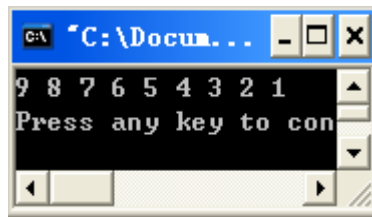


图 3-9 例 3-8 运行结果

对【例 3-8】的说明：

表达式 `coll.rbegin()` 返回一个由 `coll` 定义的 `reverse iterators`，并将其作为“对 `coll` 逆向遍历”的起点。它指向 `coll` 的尾部（最后一个元素的下一位置）。因此，`*coll.rbegin()` 表达式返回的是最后一个元素的值。

同样，`coll.rend()` 也返回一个由 `coll` 定义的 `reverse iterators`，并将其作为“对 `coll` 逆向遍历”的终点。它指向 `coll` 内第一个元素的前一个位置。

第4讲 STL 函数对象

每当提到 C++ STL，首先被人想到的是它的三大核心组件：Containers, Iterators 以及 Algorithms，即容器，迭代器和算法。容器为用户提供了常用的数据结构，算法是独立于容器的常用的基本算法，迭代器是由容器提供的一种接口，算法通过迭代器来操控容器。其实 C++ STL 还有另外一种重要组件，即函数对象（Function Object）。函数对象又称为仿函数(functor)。STL 中有很多算法（这些算法包含在 `algorithm` 头文件中）都需要为它提供一个称之为函数对象类型的参数。

其实，有关函数对象的内容在第 1 讲里已经讨论了一些，本讲继续介绍函数对象的概念，重点介绍 STL 中预定义的函数对象及其用法。

程序中使用 STL 预定义的函数对象，首先需要包含头文件：`#include <functional>`。

§4.1 什么是函数对象

所谓函数对象其实就是一个行为类似函数的“东西”，它可以没有参数，也可以带有若干参数，其功能是获取一个值，或者改变操作的状态。在 C++ 程序设计中，任何普通函数和任何重载了函数调用运算符“`()`”的类的实例都是函数对象，都可以作为参数传递给 STL 算法。

下面以 STL 算法中的 `accumulate()` 算法为例，介绍函数对象的设计与应用过程。算法 `accumulate()` 的函数模版原型如下：

```
template<class InputIterator, class Type, class BinaryFunction>
Type accumulate(InputIterator first, InputIterator last, Type val,
                BinaryFunction binaryOp);
```

其功能是对 `[first, last)` 区间内的数据进行累积运算，如累加、累乘等。究竟是哪种累积运算，完全由二元函数对象 `binaryOp` 决定；

`val` 为累积的初值；初值及该算法的返回值类型均为 `Type`；

下面使用两种方式定义表示数组元素乘法的函数对象，见例 4-1 和例 4-2。

【例 4-1】通过定义普通函数来定义函数对象

一般来说，用户设计的普通函数就是一种最简单的函数对象。本例首先设计一个普通的函数 `mult()` 作为函数对象传递给算法 `accumulate()`。

```
#include <iostream>
#include <numeric> //使用 accumulate() 算法必须包含数值算法头文件
using namespace std;

//定义一个普通函数作为函数对象传递给算法 accumulate(), 进行累积运算
int mult(int x, int y) { return x * y; };

int main() { //测试函数
    int a[] = { 1, 2, 3, 4, 5 };
    const int N = sizeof(a) / sizeof(int); //求 a 的元素个数
    cout << "数组 a 的元素乘积= "
         << accumulate(a, a + N, 1, mult) //全局普通函数 mult 作为函数对象
         << endl;
```



```

    return 0;
}

```

程序运行结果如图 4-1 所示。



图 4-1

【例 4-2】通过类来定义函数对象

```

#include <iostream>
#include <numeric>    //使用 accumulate()算法必须包含数值算法头文件
using namespace std;

class MultClass {
public:
    int operator() (int x, int y) const { return x * y; }    //必须重载 operator()函数
};

int main() {        //测试函数
    int a[] = { 1, 2, 3, 4, 5 };
    const int N = sizeof(a) / sizeof(int);    //求 a 的元素个数
    cout << "数组 a 的元素乘积= "
    << accumulate(a, a + N, 1, MultClass()) //调用默认构造函数生成临时匿名对象
    << endl;
    return 0;
}

```

程序运行结果也如图 4-1 所示。

下面再举一个例子(【例 4-3】), 以便进一步理解通过类来定义函数对象的方法。

【例 4-3】求整形向量各元素之和(即计算 $\sum_{i=1}^{100} i$)

```

#include<iostream>
#include<vector>
#include <functional>
#include<algorithm>
using namespace std;

class CSum
{
private:

```

```

        int sum;
    public:
        CSum() {sum = 0;}           //构造函数
        void operator()(int n)      //必须重载 operator()函数
        {
            sum += n;
        }
        int GetSum() {return sum;}
};

void main(){
    vector<int> v ;
    for(int i=1; i<=100; i++)
    { v.push_back(i); }
    CSum sObj = for_each(v.begin(), v.end(), CSum());
    /*在上面一条语句中, 由默认构造函数生成一个匿名临时函数对象。算法 for_each()
    是对给定的区间内每一个数据元素执行函数对象 CSum(), 即执行 operator()函数 */

    cout<<sObj.GetSum()<<endl;
}

```

运行结果如图 4-2 所示。

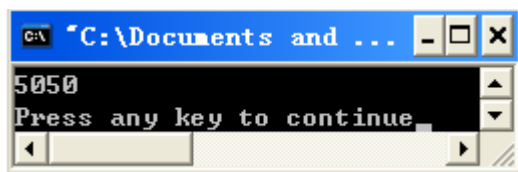


图 4-2 例 4-3 运行结果

对【例 4-3】的三点说明：

本例直接采用构造函数调用函数对象，如本例中 `for_each(v.begin(),v.end(),CSum())` 中的第 3 个参数 `CSum()`，也就是说：对本例而言，STL 知道 `CSum()` 对应着 `CSum` 类中重载的 `operator()` 函数；

具体有几个参数呢，由于 `for_each` 函数一次只能迭代出一个整形数，所以 STL 知道每迭代一次整形数，都要执行一次 `CSum` 中的 `operator()(int)` 函数，只需要一个参数；

`CSum sObj` 是用来接收 `for_each` 迭代函数对象的最终结果值的。

§4.2 标准库中预定义的函数对象

除了上面介绍的通过自定义函数对象的两种方法外，STL 也预定义了若干个函数对象，表 4-1 列出了 STL 预定义的所有仿函数。其中 `arg`、`arg1`、`arg2` 是仿函数的参数，它们一般为 `T` 类型。这些预定义好的函数对象都是重载了函数调用运算符 `operator()`。注意，STL 预定义的这些函数对象其实都是继承于 §4.3 介绍的两个基类的子类。

要使用这些仿函数，也必须包含头文件 `<functional>`。

标准 C++ 库根据重载的 `operator()` 函数的参数个数是 0 个、1 个或 2 个而将 STL 预定义的仿函数划分为以下几种类型，它们之间的关系如图 4-3 所示：

- 发生器(Generator)：没有参数且返回一个任意类型值的函数对象，例如随机数发生器。

- 一元函数(Unary Function): 只有一个任意类型的参数, 且返回一个可能不同类型值的函数对象。
- 二元函数(Binary Function): 有两个任意类型的参数, 且返回一个任意类型值的函数对象。
- 一元谓词(Unary Predicate): 返回 bool 型值的一元函数。一元谓词又叫做一元判定函数。
- 二元谓词(Binary Predicate): 返回 bool 型值的二元函数。二元谓词又叫做二元判定函数。

可以看出, STL 中函数对象最多仅适用于两个参数的情况(无参数、一元或二元情况), 但这也足以完成相当强大的功能。

表 4-1 STL 预定义的仿函数(仅一个一元仿函数)

	仿函数名称	类型	结果描述
算术类函数对象	加法: <code>plus<T>()</code>	二元函数	<code>arg1 + arg2</code>
	减法: <code>minus<T>()</code>	二元函数	<code>arg1 - arg2</code>
	乘法: <code>multiplies<T>()</code>	二元函数	<code>arg1 * arg2</code>
	除法: <code>divides<T>()</code>	二元函数	<code>arg1 / arg2</code>
	模取: <code>modules<T>()</code>	二元函数	<code>arg1 % arg2</code>
	否定: <code>negate<T>()</code>	一元函数	<code>-arg</code>
关系运算类函数对象	等同: <code>equal_to<T>()</code>	二元函数	<code>arg1 == arg2</code>
	不等于: <code>not_equal_to<T>()</code>	二元函数	<code>arg1 != arg2</code>
	大于: <code>greater<T>()</code>	二元函数	<code>arg1 > arg2</code>
	大于等于: <code>greater_equal<T>()</code>	二元函数	<code>arg1 >= arg2</code>
	小于: <code>less<T>()</code>	二元函数	<code>arg1 < arg2</code>
	小于等于: <code>less_equal<T>()</code>	二元函数	<code>arg1 <= arg2</code>
逻辑运算类函数对象	与: <code>logical_and<T>()</code>	二元函数	<code>arg1 && arg2</code>
	或: <code>logical_or<T>()</code>	二元函数	<code>arg1 arg2</code>
	非: <code>logical_not<T>()</code>	二元函数	<code>!arg1</code>

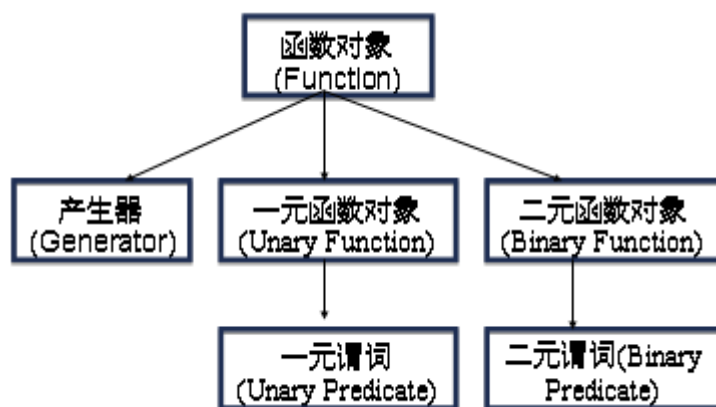


图 4-3 函数对象的关系

要特别注意使用表 4-1 个列出的 STL 预定义仿函数(它们都是继承于§4.3 介绍的两个基类的子类)的方法:

方法一: 先产生函数对象实例, 然后运用该实例执行函数功能, 如:

```
plus<int> plusobj;           //先产生函数对象 plus 的实例 plusobj
cout<< plusobj (2,4)<<endl; //然后运用实例 plusobj 执行函数功能
```

方法二: 直接以函数对象的临时对象(通过执行默认构造函数) 执行函数功能, 如例 4-4。
再如:

```
cout<< plus<int>() (2,4)<<endl;
```

下面举例说明如何使用 STL 预定义的函数对象。

一、STL 预定义的算术类基本函数对象使用示例

【例 4-4】通过 STL 提供的预定义的函数对象 `multiples<T>()` 同样可以实现例 4-1 或例 4-2 对元素连乘的操作。

```
#include <iostream>
#include <numeric>    //包含数值算法头文件
#include <functional> //包含标准函数对象头文件
using namespace std;
int main() {
    int a[] = { 1, 2, 3, 4, 5 };
    const int N = sizeof(a) / sizeof(int);

    //在下面的语句中将标准二元函数对象 multiplies<int>()传递给通用算法
    cout << "数组 a 的元素乘积= "
         << accumulate(a, a + N, 1, multiplies<int>())<< endl;
    return 0;
}
```

本程序运行结果也如同图 4-1 所示。

说明: 对于常规通用数据类型 `char`、`int`、`float`、`double`、`string` 而言, 可以直接套用 STL 预定义的基本函数对象, 如上面介绍的两种方法。但对于非常规数据类型(如复数类型), 则必须在定义该类型的类(如复数类)内重载相应的运算符。关于这方面的例子请参考有关书籍, 下同。

二、STL 预定义的关系运算类(一元谓词和二元谓词)函数对象使用示例

它们都返回 `bool` 型, 因此, 它们都是谓词。同样, 也要注意它们的使用方法。

【例 4-5】此例将二元函数对象 `greater()` 带入 `sort()` 算法, 从而实现把一个数组降序排列。默认情况下, `sort()` 算法使用 `less` 比较器进行比较, 实现的是升序排列。

```
#include <functional>
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
using namespace std;
```

```

int main(){
    int intArr[] = { 30, 90, 10, 40, 70, 50, 20, 80 };
    const int N = sizeof(intArr) / sizeof(int);
    vector<int> a(intArr, intArr + N);

    cout << "before sorting:" << endl;
    copy(a.begin(), a.end(), ostream_iterator<int>(cout, "\t"));
    cout << endl;

    sort(a.begin(), a.end(), greater<int>()); //降序排列

    cout << "after sorting:" << endl;
    copy(a.begin(), a.end(), ostream_iterator<int>(cout, "\t"));
    cout << endl;
    return 0;
}

```

运行结果如图 4-4 所示。

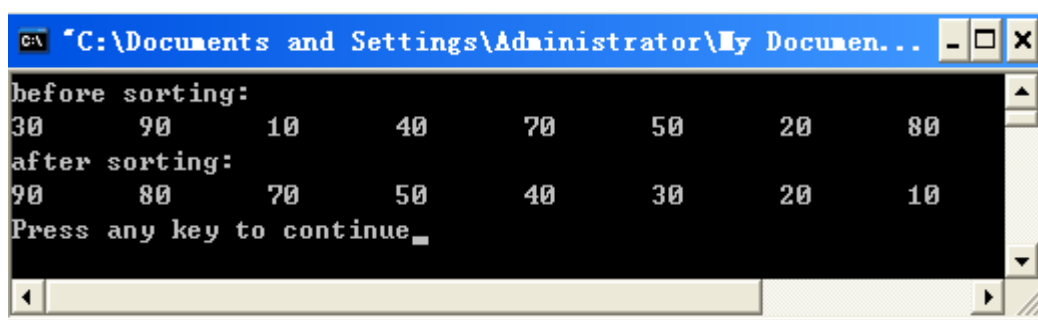


图 4-4 例 4-5 运行结果

三、STL 预定义的逻辑运算类(一元谓词和二元谓词)函数对象使用示例

【例 4-6】

```

#include <functional>
#include <iostream>
using namespace std;
void main(){
    //产生一些函数对象实体
    logical_and<int>andobj;
    logical_or<int>orobj;
    logical_not<int>notobj;

    //运行上述对象执行函数功能
    cout<<andobj(true,true)<<endl;    //1
    cout<<orobj(true,false)<<endl;    //1
    cout<<notobj(true)<<endl<<endl;    //0

    //利用临时对象(由默认构造函数生成)执行函数功能

```

```

    cout<<logical_and<int>()(3<5,6<9)<<endl;           //1
    cout<<logical_or<int>()(3<5,6>9)<<endl;             //1
    cout<<logical_not<int>()(3<5)<<endl;                //0
}

```

§4.3 STL 中一元函数对象与二元函数对象的基类

STL 中，定义了两个分别用于设计一元函数对象与二元函数对象的基类 `unary_function` 和 `binary_function`。**STL 中预定义的函数对象都分别继承自它们。**用户也可通过对它们的继承，来设计自己的一元函数对象与二元函数对象。

一、一元函数对象

基类 `unary_function` 的类模板原型如下：

```

template<class _A, class _R>
struct unary_function
{
    typedef _A argument_type;
    typedef _R result_type;
};

```

它有两个模板参数，`_A` 是输入参数，`_R` 是返回值类型，且此两个参数的类型是任意的，因此，它的动态特性非常强。

【例 4-7】设计一个一元函数对象对向量各元素进行处理

```

#include <iostream>
#include <iterator>
#include <algorithm>
#include <functional>
using namespace std;

void Add(int & ri) //普通全局函数
{
    ri += 5;
}

class Sub : public unary_function<int,void> //通常必需继承自基类 unary_function
{
public:
    void operator() (int & ri)
    {
        ri -= 5;    // ri = ri - 5
    }
};

void main()
{
    int a[10]={0,1,2,3,4,5,6,7,8,9};

```

```

    copy(a,a+10,ostream_iterator<int>(cout," "));
    cout << endl;

    for_each(a,a+10,Add); //普通全局函数 Add 作为函数对象传给算法 for_each
    copy(a,a+10,ostream_iterator<int>(cout," "));
    cout << endl;

    for_each(a,a+10,Sub()); //通过 Sub 类默认构造函数生成临时匿名对象
    copy(a,a+10,ostream_iterator<int>(cout," "));
    cout << endl;
}

```

程序运行结果如图 4-5 所示。

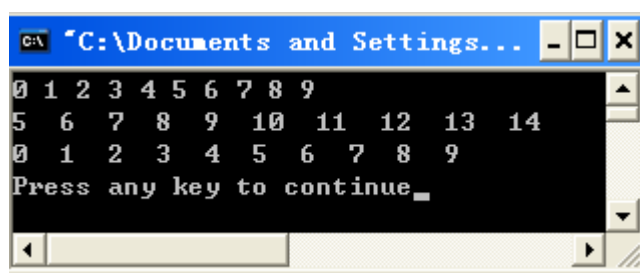


图 4-5 例 4-7 运行结果

二、二元函数对象

基类 `binary_function` 的类模板原型如下：

```

template<class Arg1, class Arg2, class Result>
struct binary_function
{
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};

```

它有三个模板参数，`Arg1`、`Arg2` 是输入参数，`Result` 是返回值类型，且此三个参数的类型是任意的，因此，它的动态特性非常强。

【例 4-8】设计一个二元函数对象 `binary_sort`，实现对向量元素升序排列

```

#include <functional>
#include <string>
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

```

```
class Student
{
public:
    string name;
    int grade;

    Student(string name1,int grade1)
    {
        name=name1;
        grade=grade1;
    }

    bool operator<(const Student &s) const
    {
        return grade<s.grade;
    }
};

ostream & operator<<(ostream & os,const Student &s)
{
    os<<s.name<<"\t"<<s.grade<<"\n";
    return os;
}

template<class _inPara1,class _inPara2>
class binary_sort:public binary_function<_inPara1,_inPara2,bool>
{
public:
    bool operator()(_inPara1 in1,_inPara2 in2)
    {
        return in1<in2;
    }
};

void main(){
    Student s1("zhangsan",60);
    Student s2("lisi",80);
    Student s3("wangwu",70);
    Student s4("zhaoliu",90);

    vector<Student> v;
    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);
}
```



```

v.push_back(s4);
sort(v.begin(),v.end(),binary_sort<Student &,Student &>()); //按升序排列
copy(v.begin(),v.end(),ostream_iterator<Student>(cout,"  ")); //升序结果输出
}

```

程序运行结果如图 4-6 所示。

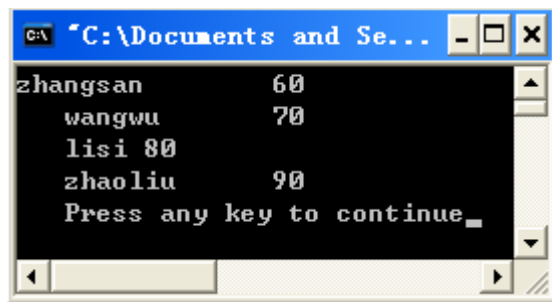


图 4-6 例 4-8 运行结果

说明：该函数的程序执行流程：

主程序中首先把 4 个学生对象依次放入向量 `v` 中，当执行排序函数 `sort` 时，调用二元函数类 `binary_sort` 中重载的 `operator()` 函数，当执行函数体 `in1<in2` 时，由于 `in1`、`in2` 的参数类型都是 `Student &`，因此，调用 `Student` 类中 `operator` 函数完成两个 `Student` 引用对象真正的比较功能，随后把返回值依次返回，`sort` 函数根据此返回值决定这两个 `Student` 对象是否交换。

§4.4 函数适配器

有时候，需要把一个二元函数对象转换为一个一元函数对象，或者组合几个二元谓词为一个谓词，从而得到一个简化了的仿函数，然后再把这个简化了的仿函数作为参数传递给某个算法(否则不符合该算法对参数的要求)。

所谓“函数适配器(Function Adapters)”就是指能够将一个仿函数和另一个仿函数(或某个值，或某个一般函数)结合起来的仿函数。函数适配器也声明于 `<functional>` 中。

表 4-2 列出了 STL 提供的函数适配器(它们也是类模版)，分为如下 4 大类：

- 绑定适配器——将 n 元函数对象的指定参数绑定为一个常数，得到 $n-1$ 元函数对象；
- 取反适配器——将指定谓词的结果取逻辑反；
- 指针函数适配器——对一般函数指针使用，使之能够作为其它函数适配器的输入；
- 成员函数适配器——对成员函数指针使用，把 n 元成员函数适配为 $n+1$ 元函数对象，该函数对象的第一个参数为调用该成员函数时的目的对象。

表 4-2 STL 提供的函数适配器

类 型	STL 提供的函数适配器	功 能 说 明
绑定 适配器	<code>binder1st</code>	将数值绑定到二元函数的第一个参数，适配成一元函数
	<code>binder2nd</code>	将数值绑定到二元函数的第二个参数，适配成一元函数
取反 适配器	<code>unary_negate</code>	将一元谓词的返回值适配成其逻辑反
	<code>binary_negate</code>	将二元谓词的返回值适配成其逻辑反
指针函数 适配器	<code>pointer_to_unary_function</code>	将普通一元函数指针适配成 <code>unary_function</code>
	<code>pointer_to_binary_function</code>	将普通二元函数指针适配成 <code>binary_function</code>

表 4-2(续)

成员函数 适配器	mem_fun_t	将无参数类成员函数适配成一元函数对象，第一个参数为该类的指针类型
	mem_fun_ref_t	将无参数类成员函数适配成一元函数对象，第一个参数为该类的引用类型
	mem_fun1_t	将单参数类成员函数适配成二元函数对象，第一个参数为该类的指针类型
	mem_fun1_ref_t	将单参数类成员函数适配成二元函数对象，第一个参数为该类的引用类型
	const_mem_fun_t	将无参数类常成员函数适配成一元函数对象，第一个参数为该类的常指针类型
	const_mem_fun_ref_t	将无参数类常成员函数适配成一元函数对象，第一个参数为该类的常引用类型
	const_mem_fun1_t	将单参数类成员函数适配成二元函数对象，第一个参数为该类的常指针类型
	const_mem_fun1_ref_t	将单参数类成员函数适配成二元函数对象，第一个参数为该类的常引用类型

为了简化函数适配器的构造，STL 还提供了若干函数适配器辅助函数(如表 4-3 所示)。通过这些函数适配器辅助函数可以直接地、隐式实现函数适配器的构造，无须通过函数适配器(是类)的构造函数来构造函数适配器的实例(否则书写其代码很麻烦)，从而大大简化或根本不用书写构造函数适配器的代码。

表 4-3 STL 函数适配器辅助函数

函 数	功 能 说 明
bind1st()	辅助构造 binder1st 适配器实例，帮定固定值到二元函数的第一个参数位置
bind2nd()	辅助构造 binder2st 适配器实例，帮定固定值到二元函数的第二个参数位置
not1()	辅助构造 unary_negate 适配器实例，生成一元函数的逻辑反函数
not2()	辅助构造 binary_negate 适配器实例，生成二元函数的逻辑反函数
mem_fun()	辅助构造 mem_fun_t 等成员函数适配器实例，返回一元或二元函数对象
mem_fun_ref()	辅助构造 mem_fun_ref 等成员函数适配器实例，返回一元或二元函数对象
ptr_fun()	辅助构造一般全局函数指针的 pointer_to_unary_function 或 pointer_to_binary_function 适配器实例

下面分别介绍几种函数适配器辅助函数的基本用法。

一、绑定、取反适配器

【例 4-9】绑定、取反适配器基本用法

```
#include <functional>
#include <algorithm>
#include <iostream>
```

```
using namespace std;
```

```

void main(){
    int a[] = {1,3,5,7,9,8,6,4,2,0};
    int nCount = count_if(a, a+sizeof(a)/sizeof(int), bind2nd(less<int>(), 4));
    cout << nCount << endl;        //输出 4

    nCount = count_if(a, a+sizeof(a)/sizeof(int), bind1st(less<int>(), 4));
    cout << nCount << endl;        //输出 5

    nCount = count_if(a, a+sizeof(a)/sizeof(int), not1(bind2nd(less<int>(), 4)));
    cout << nCount << endl;        //输出 6

    nCount = count_if(a, a+sizeof(a)/sizeof(int), not1(bind1st(less<int>(), 4)));
    cout << nCount << endl;        //输出 5

    sort(a, a+sizeof(a)/sizeof(int), not2(less<int>()));

    //下面的语句输出 9 8 7 6 5 4 3 2 1 0
    copy(a, a+sizeof(a)/sizeof(int), ostream_iterator<int>(cout, " "));
    cout << endl;
}

```

注：针对【例 4-8】着重理解以下的知识点：

(1)bind2nd(less<int>(),4):

less<int>()本身是一个二元函数对象，相当于普通函数 `bool less(T x,T y){ return x<y;}`；而 `bind2nd(less<int>(),4)`相当于普通函数 `bool less(T x, int n=4){return x<4}`，`bind2nd` 的作用是使 `less` 二元函数的第二个参数绑定为整形 4。因此，相当于把二元函数降低为一元函数，`count_if` 语句中是求数组 `a` 中小于 4 的数据个数是多少，可知符合条件的数有{1, 3, 2, 0}，共 4 个。

(2)bind1st(less<int>(),4):

less<int>()本身是一个二元函数对象，相当于普通函数 `bool less(T x,T y){ return x<y;}`；而 `bind1st(less<int>(),4)`相当于普通函数 `bool less(int n=4, T x){return 4<x}`，`bind1st` 的作用是使 `less` 二元函数的第 1 个参数绑定为整形 4。因此相当于把二元函数降低为一元函数，`count_if` 语句中是求数组 `a` 中大于 4 的数据个数是多少，可知符合条件的数有{5, 7, 9, 8, 6}，共 5 个。

(3)not1(bind2nd(less<int>(), 4)):

`bind2nd(less<int>(), 4)`相当于普通函数 `bool less(T x, int n=4){return x<4;}`，`not1` 后，相当于 `bool less(T x, int n=4){return !(x<4);}`，语义上相当于“不小于 4”，即求不小于 4 的数据个数是多少，可知符合条件的数有{5, 7, 9, 8, 6, 4}，共有 6 个。

(4)not1(bind1st(less<int>(), 4)):

`bind1st(less<int>(), 4)`相当于普通函数 `bool less(int n=4, T x){return 4<x;}`，`not1` 后，相当于 `bool less(int n=4, T x){return !(4<x);}`，语义上相当于“不大于 4”，即求不大于 4 的数据个数是多少，可知符合条件的数有{1, 3, 4, 2, 0}，共有 5 个。

(5)not2(less<int>()):

less<int>()相当于普通函数 `bool less(T x, T y){return x<y;}`，`not2` 后相当于 `bool less(T x, T y){return !(x<y);}`，即求数组 `a` 的降幂排序序列。

二、函数适配器

函数适配器有两种：

针对一般函数（非成员函数）而设计的函数适配器。这种情况在前面已经举过很多例子，下面再举一个例子(参见例 4-10)；

针对类的普通成员函数而设计的函数适配器，参见例 4-11。

【例 4-10】针对一般函数（非成员函数）而设计的函数适配器是最经常使用的用法，通过函数适配器对一个参数进行绑定。

```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;

class Person
{
public:
    Person(string init)
    {
        name=init;
    }
    string name;
};

bool shorter (const Person & t1, const Person & t2)  //普通全局函数
{
    Return  t1.name.length() < t2.name.length();
}

void main()
{
    vector<Person> iv;
    iv.push_back(Person("one"));
    iv.push_back(Person("two"));
    iv.push_back(Person("three"));
    iv.push_back(Person("four"));

    //将函数指针(函数名即是)传递进去
    sort(iv.begin(), iv.end(), shorter);  // shorter 函数作为函数对象

    for (vector<Person>::iterator it = iv.begin(); it != iv.end(); ++it)
        cout << (*it).name << " ";
    cout<<endl;
}
```

运行结果如图 4-7 所示。

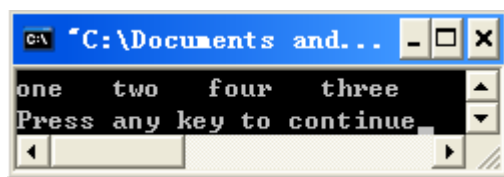


图 4-7 例 4-10 运行结果

【例 4-11】针对类的普通成员函数而设计的函数配接器的基本用法

这里所说的成员函数不包括 `operator()`，这种用法不多见。它是通过 `mem_fun_ref` 进行转换，将原本针对某个元素的函数调用转为调用被传递变量(*itr itr 为 iv 的迭代器)的成员函数。

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Student
{
    string strNO;    //学号
    string strName;  //姓名
public:
    Student(string strNO1, string strName1)
    {
        strNO=strNO1;
        strName=strName1;
    }
    bool show()      //学生信息显示成员函数(类的普通成员函数)
    {
        cout<<strNO<<" : "<<strName<<endl;
        return true;
    }
};

void main(){
    //针对 mem_fun_ref 程序段
    Student s1("1001","zhangsan");
    Student s2("1002","lisi");

    vector<Student> v;
    v.push_back(s1);
    v.push_back(s2);
    for_each(v.begin(),v.end(),mem_fun_ref(Student::show));
}
```

```

cout<<endl;

//针对 mem_fun 程序段
Student *ps1=new Student("1003","wangwu");
Student *ps2=new Student("1004","zhaoliu");

vector<Student *> pv;
pv.push_back(ps1);
pv.push_back(ps2);
for_each(pv.begin(),pv.end(),mem_fun(Student::show));
cout<<endl;
}

```

程序运行结果如图 4-8 所示。

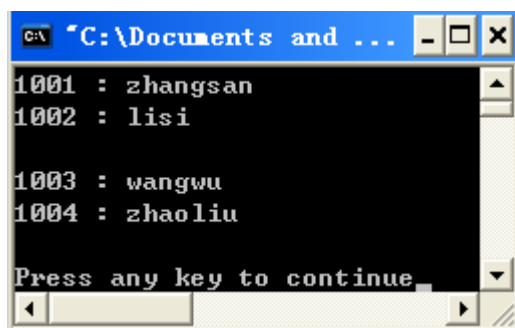


图 4-8 例 4-11 运行结果

针对例 4-11 主要理解点如下几点：

(1)mem_fun_ref、mem_fun 的区别：

若集合是基于对象的，形如 `vector<Student>`，则用 `mem_fun_ref`；若集合是基于对象指针的，形如 `vector<Student *>`，则用 `mem_fun`。

(2)以下调用写法是错误的：

`for_each(v.begin(), v.end(), Student::show)`，当在 STL 算法中调用成员函数时，一般要通过 `mem_fun_ref` 或 `mem_fun` 转换后才可以应用。

§4.5 STL 预定义函数对象应用示例

C++标准库定义了几个有用的函数对象，它们被放到 STL 算法中。例如，`sort()`算法以判断对象（predicate object）作为其第三个参数。判断对象是一个返回 Boolean 型结果的模板化的函数对象。可以向 `sort()`传递判断函数对象 `greater<>`或者 `less<>`来强行实现降序排序或升序排序：

```

#include <iostream>
#include <functional> // for greater<> and less<>
#include <algorithm> //for sort()
#include <vector>
using namespace std;

```

```
int main(){
vector<int> coll; //生成一个元素是 int 类型的向量容器 coll
//..填充向量
    coll.push_back(2);
    coll.push_back(5);
    coll.push_back(4);
    coll.push_back(1);
    coll.push_back(6);
    coll.push_back(3);
    cout << "向量中元素的初始顺序: "<<endl;
    for (int k=0; k<coll.size(); ++k)
        cout << coll[k] << " ";
    cout << endl<< endl;

    sort(coll.begin(), coll.end(), greater<int>()); //降序( descending )
    cout << "向量中元素降序排序的结果: "<<endl;
    for (int i=0; i<coll.size(); ++i)
        cout << coll[i] << " ";
    cout << endl<< endl;

    sort(coll.begin(), coll.end(), less<int>()); //升序 ( ascending )
    cout << "向量中元素升序排序的结果: "<<endl;
    for (int j=0; j<coll.size(); ++j)
        cout << coll[j] << " ";
    cout << endl;
}
```

程序运行结果如图 4-9 所示。

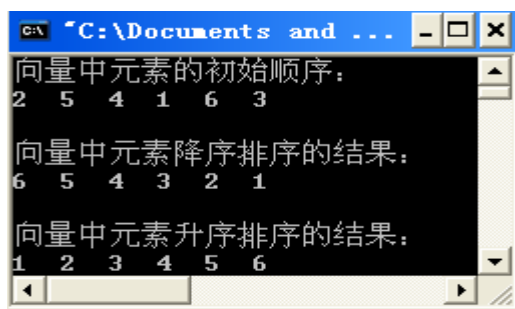


图 4-9

第 5 讲 STL 算法

为了处理各类容器内的数据元素，STL 提供了近百个标准算法(Algorithms)，包括搜索、排序、拷贝、修改、数值运算等十分基本而又普遍应用的算法。这些算法大致分为如下四类：

- 1、非可变序列算法：指不直接修改其所操作的数据元素的值或顺序的算法；
- 2、可变序列算法：指可以修改它们所操作的数据元素的值或顺序的算法；
- 3、排序算法：包括对序列进行排序和合并的算法、搜索算法以及有序序列上的集合操作；
- 4、数值算法：对容器的数据元素进行数值计算。

STL 算法部分主要由头文件<algorithm>,<numeric>,<functional>组成。在使用 STL 中的算法（函数）之前，首先必须包含头文件<algorithm>。对于数值算法须包含<numeric>头文件。头文件<functional>中定义了一些模板类，用来声明函数对象。

注意，算法并非容器的成员函数，而是一种全局的泛型函数，无需通过容器对象调用它们，直接使用即可。还要注意，有些算法和容器的某些成员函数具有相同的功能。

下面首先介绍 **for_each()** 算法：

for_each() 算法非常灵活，它可以以不同的方式存取、处理、修改每一个数据元素。

```
UnaryProc for_each (InputIterator beg, InputIterator end,
                  UnaryPredicate op)
```

- 其功能是对区间中的每一个元素调用：*op(elem)*

下面列举三个示例展示 **for_each()** 算法的使用。先设计一个头文件 **algostuff.hpp** 如下：

//algostuff.hpp

```
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>
#include <functional>
#include <numeric>
```

```
/* PRINT_ELEMENTS()
```

```
* - prints optional C-string optcstr followed by
* - all elements of the collection coll
* - separated by spaces
*/
```

```
template <class T>
```

```
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
```



```

{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

/* INSERT_ELEMENTS (collection, first, last)
 * - fill values from first to last into the collection
 * - NOTE: NO half-open range
 */
template <class T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(),i);
    }
}

```

【示例 1】将 print()传给 for_each(), 使得 for_each()对每一个元素调用 print(), 从而输出所有元素:

```

//foreach1.cpp
#include "alghostuff.hpp"
using namespace std;

// function called for each element
void print (int elem)
{
    cout << elem << ' ';
}

int main(){
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,9);

    // call print() for each element
    for_each (coll.begin(), coll.end(), // range
             print);                  // operation
    cout << endl;
}

```

程序输出结果如下：

```
1 2 3 4 5 6 7 8 9
```

【示例 2】利用仿函数改变每一个元素的内容：

/foreach2.cpp

```
#include "algotuff.hpp"
```

```
using namespace std;
```

```
// function object that adds the value with which it is initialized
```

```
template <class T>
```

```
class AddValue {
```

```
private:
```

```
    T theValue; // value to add
```

```
public:
```

```
// constructor initializes the value to add
```

```
AddValue (const T& v) : theValue(v) { }
```

```
// the function call for the element adds the value
```

```
void operator() (T& elem) const {
```

```
    elem += theValue;
```

```
}
```

```
};
```

```
int main(){
```

```
    vector<int> coll;
```

```
    INSERT_ELEMENTS(coll,1,9);
```

```
// add ten to each element
```

```
    for_each (coll.begin(), coll.end(), // range
```

```
        AddValue<int>(10)); // operation
```

```
    PRINT_ELEMENTS(coll);
```

```
// add value of first element to each element
```

```
    for_each (coll.begin(), coll.end(), // range
```

```
        AddValue<int>(*coll.begin())); // operation
```

```
    PRINT_ELEMENTS(coll);
```

```
}
```

程序输出如下：

```
11 12 13 14 15 16 17 18 19
```

```
22 23 24 25 26 27 28 29 30
```

```
Press any key to continue
```

【示例 3】利用的返回值。由于能够返回一项操作，所以可以在该项操作中处理返回结

果:

```
//foreach3.cpp
#include "alghostuff.hpp"

using namespace std;

// function object to process the mean value
class MeanValue {
private:
    long num;    // number of elements
    long sum;    // sum of all element values
public:
    // constructor
    MeanValue () : num(0), sum(0) {
    }

    // function call
    // - process one more element of the sequence
    void operator() (int elem) {
        num++;    // increment count
        sum += elem;    // add value
    }

    // return mean value (implicit type conversion)
    operator double() {
        return static_cast<double>(sum) / static_cast<double>(num);
    }
};

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,1,8);

    // process and print mean value
    double mv = for_each (coll.begin(), coll.end(), // range
        MeanValue());    // operation
    cout << "mean value: " << mv << endl;
}
```

程序输出如下:

```
mean value: 4.5
Press any key to continue
```

§5.1 非变异算法

非变异算法是一组不破坏操作数据元素的模板函数，用来对序列数据元素进行逐个处理、元素查找、子序列搜索、统计和匹配。非变异算法具有极为广泛的适用性，基本上可应用于各种容器。表 5-1 列出了主要的非变异算法。

表 5-1 主要的非变异算法

序号	功能	函数名称	说 明
1	循环	for_each()	遍历容器元素，对每个元素都执行某操作
2	查询	find()	找出等于某个值的第一次出现的那个元素的位置
		find_if()	找出符合某谓词的第一个元素的位置
		find_first_of()	找出等于“某数个值之一”的第一次出现的元素
		adjacent_find()	找出第一次相邻值相等的元素
		find_end()	找出一子序列的最后一次出现的位置
		search()	找出某一子区间第一次出现的位置
		search_n	找出某特定值连续 n 次出现的位置
3	计数	count()	统计某个值出现的次数
		count_if()	统计与某谓词(表达式)匹配的元素个数
4	比较	equal()	判断是否相等，即两个序列中的对应元素都相同时为真
		mismatch()	找出两个序列第一个相异的元素
		min_element()	返回最小值元素的位置
		max_element()	返回最大值元素的位置

下面通过示例对部分常用的非变异算法加以说明，以加深对它们的功能与用法的理解：

1、查找容器元素位置算法 find()

【功能】：它用于查找等于某值的第一个出现的那个元素。

形式 1: InputIterator **find**(InputIterator *beg*, InputIterator *end*, const T & *value*)

形式 2: InputIterator **find_if**(InputIterator *beg*, InputIterator *end*,
UnaryPredicate *op*)

形式 1 返回迭代器区间[*beg*,*end*)中第一个“元素值等于 *value*”的元素位置，即如果迭代器 *i* 所指的元素满足 **i=value*，则返回迭代器 *i*；未找到满足条件的元素，返回 *end*。

形式 2 返回迭代器区间[*beg*,*end*)中使一元判别式 *op(elem)* 为 true 的第一个元素；未找到返回 *end*。

【例 5-1】

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main(){
    int num_to_find = 6;
    vector<int> v1;
    for( int i = 0; i < 10; i++ )
        v1.push_back(2*i);
```

```
vector<int>::iterator result;

result = find( v1.begin(), v1.end(), num_to_find );
if( result == v1.end() )
    cout << "未找到任何元素匹配 " << num_to_find << endl;
else
    cout << "匹配元素的索引值是 " << result - v1.begin() << endl;
}
```

运行结果如图 5-1 所示。

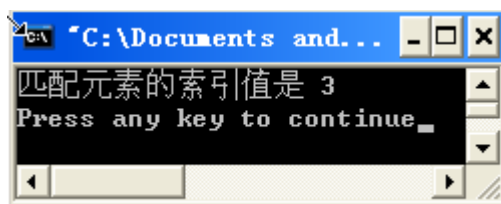


图 5-1 例 5-1 运行结果

2、条件查找容器元素位置算法 find_if()

【功能】：在迭代器区间 $[beg, end)$ 上，找出符合某谓词的第一个元素的位置；未找到元素，返回末位置 end 。

【例 5-2】在已经出现过。

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool divby5(int x)
{
    return x%5?0:1; //若 x 能被 5 整除返 0，否则返 1
}

void main(){
    vector<int> v(20);
    for(int i=0;i<v.size();i++)
    {
        v[i]=(i+1)*(i+3);
        cout<<v[i]<<' ';
    }
    cout<<endl;
    vector<int>::iterator ilocation;

    ilocation=find_if(v.begin(),v.end(),divby5);

    if(ilocation!=v.end())
        cout<<"找到第一个能被 5 整除的元素："<<*ilocation
```

```
<<endl<<"元素的索引位置是: "<<ilocation - v.begin()<<endl;
}
```

运行结果如图 5-2 所示。

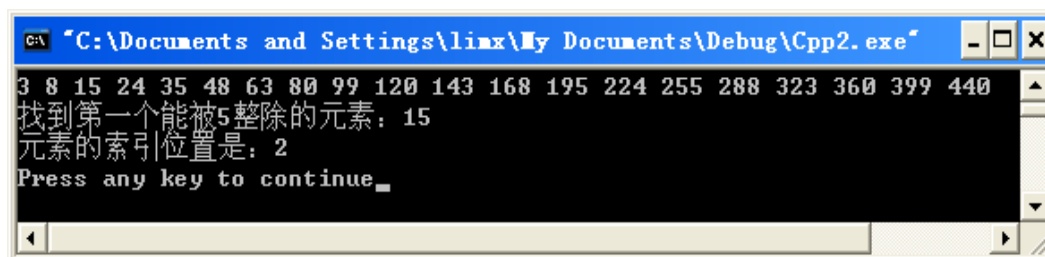


图 5-2 例 5-2 运行结果

3、统计容器中等于某给定值的元素个数算法 **count()**

形式 1: *difference_type* **count** (InputIterator *beg*,
InputIterator *end*, const T & *value*)

形式 2: *difference_type* **count_if**(InputIterator *beg*, InputIterator *end*,
UnaryPredicate *op*)

- 形式 1 计算区间[*beg*,*end*)中元素值等于的元素个数;
形式 2 计算区间[*beg*,*end*)中使一元判别式 *op(elem)*为 true 的元素个数。
- 返回值类型 *difference_type* 是表示迭代器间距的类型:
typename iterator_traits< InputIterator >::difference_type
- 关联式容器(sets,multisets,maps 和 multimaps)提供了一个等效的成员函数 count(),
用来计算等于某个 value 或某个 key 的元素个数。

【例 5-3】元素计数算法示例。先给出头文件 **algostuff.hpp** 如下:

```
//algostuff.hpp
#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>
#include <functional>
#include <numeric>

/* PRINT_ELEMENTS()
 * - prints optional C-string optcstr followed by
 * - all elements of the collection coll
 * - separated by spaces */
template <class T>
```

```

inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

/* INSERT_ELEMENTS (collection, first, last)
 * - fill values from first to last into the collection
 * - NOTE: NO half-open range
 */
template <class T>
inline void INSERT_ELEMENTS (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i) {
        coll.insert(coll.end(),i);
    }
}

```

下面的范例程序是根据不同的准则对元素进行计数：

```

// count1.cpp
#include "algostuff.hpp"
using namespace std;

bool isEven (int elem)
{ return elem % 2 == 0; }

int main(){
    vector<int> coll;
    int num;

    INSERT_ELEMENTS(coll,1,9);
    PRINT_ELEMENTS(coll,"coll: ");

    // count and print elements with value 4
    num = count (coll.begin(), coll.end(),    // range
                4);                            // value
    cout << "number of elements equal to 4:   " << num << endl;

    // count elements with even value
    num = count_if (coll.begin(), coll.end(), // range
                   isEven);                  // criterion
}

```

```

cout << "number of elements with even value: " << num << endl;

// count elements that are greater than value 4
num = count_if (coll.begin(), coll.end(), // range
               bind2nd(greater<int>(),4)); // criterion
cout << "number of elements greater than 4: " << num << endl;
}

```

本程序运行结果如下图 5-3 所示：

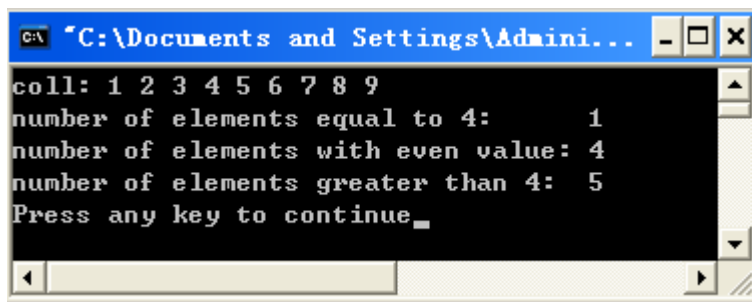


图 5-3 例 5-3 运行结果

4、求最大值和求最小值元素位置算法 `max_element()`算法和 `min_element()`

形式 1: `InputIterator max_element (InputIterator beg, InputIterator end)`

形式 2: `InputIterator max_element (InputIterator beg, InputIterator end, CompFunc op)`

形式 3: `InputIterator min_element (InputIterator beg, InputIterator end)`

形式 4: `InputIterator min_element (InputIterator beg, InputIterator end, CompFunc op)`

- 所有这些算法都返回区间 $[beg, end)$ 中最大或最小元素的位置；
- `op`用来比较两个元素：`op(elem1, elem2)`，如果 `elem1 < elem2` 成立，返回 `true`。

【例 5-4】`max_element()`算法和 `min_element()`算法

```

// minmax1.cpp
#include <cstdlib>
#include "algostuff.hpp"
using namespace std;

```

```

bool absLess (int elem1, int elem2)
{
    return abs(elem1) < abs(elem2);
}

```



```

int main(){
    deque<int> coll;

    INSERT_ELEMENTS(coll,2,8);
    INSERT_ELEMENTS(coll,-3,5);

    PRINT_ELEMENTS(coll);

    // process and print minimum and maximum
    cout << "minimum: "
        << *min_element(coll.begin(),coll.end())<< endl;
    cout << "maximum: "
        << *max_element(coll.begin(),coll.end())<< endl;

    // process and print minimum and maximum of absolute values
    cout << "minimum of absolute values: "
        << *min_element(coll.begin(),coll.end(),absLess)<< endl;
    cout << "maximum of absolute values: "
        << *max_element(coll.begin(),coll.end(),absLess)<< endl;
}

```

程序运行结果如图 5-4 所示:

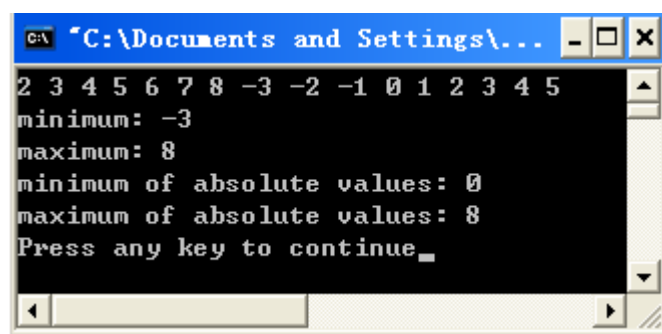


图 5-4 例 5-4 运行结果

5、子序列搜索算法 `search()`

【功能】：在一个序列中搜索与另一序列匹配的子序列首位置

形式 1: ForwardIterator1 `search` (ForwardIterator1 *beg*, ForwardIterator1 *end*
ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*)

形式 2: ForwardIterator1 `search` (ForwardIterator1 *beg*, ForwardIterator1 *end*
ForwardIterator2 *searchBeg*, ForwardIterator2 *searchEnd*, BinaryPredicate *op*)

●两种形式都返回区间[*beg*,*end*)内和区间[*searchBeg* , *searchEnd*)内完全吻合的第一个子区间内的第一个元素位置;

●形式 1 中, 子区间的元素必须完全等于[*searchBeg*, *searchEnd*)的元素;

●形式 2 中，子区间的元素和`[searchBeg, searchEnd)`的对应元素必须使二元判别式`op(eleme, searchEleme)`为 true;

●如果没有找到符合条件的子区间，两种形式都返回 `end`。

【例 5-5】搜索容器 v2 中的整个元素序列是否在 v1 中存在，若存在，返回其首位置

```
#include <vector>
```

```
#include <algorithm>
```

```
#include <iostream>
```

```
using namespace std;
```

```
void main(){
```

```
    vector<int> v1;
```

```
    cout<<"v1:";
```

```
    for(int i=0;i<5;i++){
```

```
        v1.push_back(i+5);
```

//注意：v1 定义时没有给定大小，因此这里不能直接使用赋值语句。

```
        cout<<v1[i]<<' ';
```

```
    }
```

```
    cout<<endl;
```

```
    vector<int> v2;
```

```
    cout<<"v2:";
```

```
    for(i=0;i<2;i++){
```

```
        v2.push_back(i+7);
```

```
        cout<<v2[i]<<' ';
```

```
    }
```

```
    cout<<endl;
```

```
    vector<int>::iterator ilocation;
```

```
    ilocation=search(v1.begin(),v1.end(),v2.begin(),v2.end());
```

```
    if(ilocation!=v1.end())
```

```
        cout<<"v2 的元素包含在 v1 中，起始元素为"<<"v1["
```

```
        <<ilocation-v1.begin()<<"]"<<endl;
```

```
    else
```

```
        cout<<"v2 的元素不包含在 v1 中"<<endl;
```

```
}
```

运行结果如图 5-5 所示。

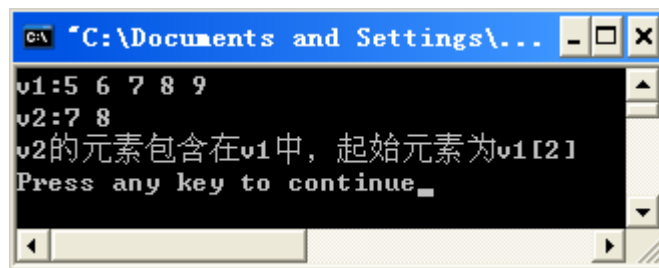


图 5-5 例 5-5 运行结果

6、重复元素子序列搜索算法 search_n()

【功能】：搜索序列中是否有一系列连续元素值均为某个给定值的子序列。若有，则返回其第一个元素的位置。

形式 1: InputIterator **search_n** (InputIterator *beg*, InputIterator *end*,
Size *count*, const T& *value*)

形式 2: InputIterator **search_n** (InputIterator *beg*, InputIterator *end*,
Size *count*, const T& *value* , BinaryPredicate *op*)

- 形式 1 返回区间[*beg,end*)中第一组连续 *count* 个元素值全部等于 *value* 的元素位置;
- 形式 2 返回区间[*beg,end*)中第一组连续 *count* 个元素使二元判别式 *op(elem,value)* 为 true 的元素位置;
- 如果没有找到匹配元素，两种形式都返回 *end*。

【例 5-6】

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
void main()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(8);
    v.push_back(8);
    v.push_back(8);
    v.push_back(8);
    v.push_back(6);
    v.push_back(6);
    v.push_back(8);
    vector<int>::iterator i;
    i=search_n(v.begin(),v.end(),3,8); //在 v 中找 3 个连续的元素，且它们的值均 8

    if(i!=v.end())
        cout<<"在 v 中找到 3 个连续的元素 8"<<endl;
    else
        cout<<"在 v 中未找到 3 个连续的元素 8"<<endl;
}
```

运行结果如图 5-6 所示。

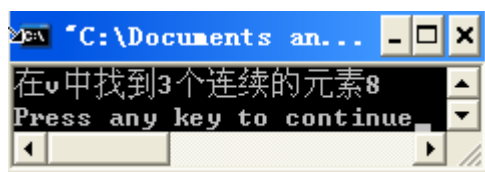


图 5-6 例 5-6 运行结果

7、搜索最后一个子区间算法 find_end()

【功能】：在指定范围内查找"由输入的另外一对 iterator 标志的第二个序列"的最后一次出现。找到则返回最后一对的第一个 ForwardIterator(前向迭代器)，否则返回输入的"另外一对"的第一个 ForwardIterator。

形式 1: ForwardIterator **find_end** (ForwardIterator *beg*, ForwardIterator *end*
ForwardIterator *searchBeg*, ForwardIterator *searchEnd*)

形式 2: ForwardIterator **find_end** (ForwardIterator *beg*, ForwardIterator *end*
ForwardIterator *searchBeg*, ForwardIterator *searchEnd*, BinaryPredicate *op*)

●两种形式都返回区间[*beg*,*end*)中和区间[*searchBeg* , *searchEnd*)内完全吻合的最后一个子区间内的第一个元素位置；

●形式 1 中，子区间的元素必须完全等于[*searchBeg*, *searchEnd*)的元素；

●形式 2 中，子区间的元素和[*searchBeg*, *searchEnd*)的对应元素必须使二元判别式 *op*(*eleme*, *searchEleme*)为 true；

●如果没有找到符合条件的子区间，两种形式都返回 *end*。

【例 5-6】

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
```

```
void main(){
    vector<int> v1;
    v1.push_back(-5);
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(-6);
    v1.push_back(-8);
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(-11);
    vector<int> v2;
    v2.push_back(1);
    v2.push_back(2);
    vector<int>::iterator i;
```

```
    i=find_end(v1.begin(),v1.end(),v2.begin(),v2.end());
```

```
    if(i!=v1.end())
```

```
        cout<<"v1 中找到最后一个匹配 v2 的子序列，位置在"
```

```
        <<"v1["<<i-v1.begin()<<"]"<<endl;
```

```
    else
```

```
        cout<<"v1 中未找到最后一个匹配 v2 的子序列\n";
```

```
}

```

运行结果如图 5-6 所示。

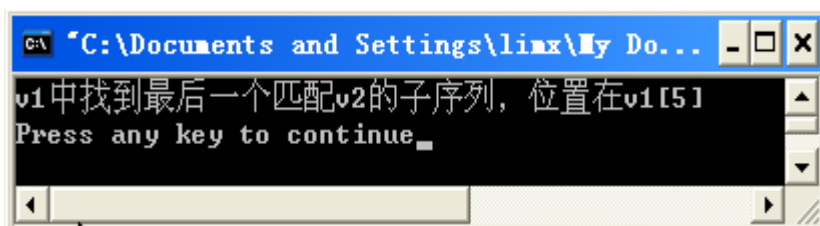


图 5-6 例 5-6 运行结果

§5.2 变异算法

变异算法的主要特点是修改容器中的元素，如修改容器中的元素值、元素复制 `copy`、改变容器中的元素顺序等。变异算法按照功能划分如表 5-2 所示。

表 5-2 变异算法的划分

序号	功能	函数名称	说 明
1	复制	<code>copy()</code>	从序列的第一个元素起复制到另一序列指定的开始处区间内
		<code>copy_backward()</code>	从序列的最后一个元素起复制某段区间
2	交换	<code>swap()</code>	交换两个元素
		<code>swap_ranges()</code>	交换指定范围的元素
		<code>iter_swap()</code>	交换由迭代器所指的两个元素
3	变换	<code>transform()</code>	变动(并复制)元素，将两个区间元素合并
4	替换	<code>replace()</code>	用一个给定值替换一些元素的值
		<code>replace_if()</code>	替换满足谓词的一些元素
		<code>replace_copy()</code>	复制序列时用一给定值替换元素
		<code>replace_copy_if()</code>	复制序列时替换满足谓词的元素
5	填充	<code>fill()</code>	用一给定值取代所有元素
		<code>fill_n()</code>	用一给定值取代前 n 个元素
6	生成	<code>generate()</code>	用一操作的结果取代所有元素
		<code>generate_n()</code>	用一操作的结果取代前 n 个元素

下面对部分变异算法举例说明，以加深对它们的功能与用法的理解：

1、元素复制算法 `copy()`

【例 5-8】

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

void main(){
    vector<int> v;
```

```

v.push_back(1);
v.push_back(3);
v.push_back(5);

list<int> l;
l.push_back(2);
l.push_back(4);
l.push_back(6);
l.push_back(8);
l.push_back(10);

//在双向链表 l 的开始处，将向量容器 v 的元素全部复制过来
copy(v.begin(),v.end(),l.begin());

list<int>::iterator i;
for(i=l.begin();i!=l.end();i++)
    cout<<*i<<' ';
cout<<endl;
}

```

运行结果如图 5-8 所示。

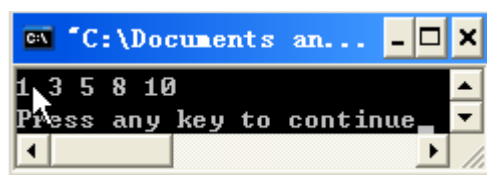


图 5-8 例 5-8 运行结果

2、元素变换算法 transform()

【功能】：变动(并复制)元素，将两个区间元素合并

【例 5-9】

```

#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int square(int x){ return x*x; }

void main(){
    vector<int> v;
    v.push_back(5);
    v.push_back(15);
    v.push_back(25);
}

```

```

cout<<"交换前向量容器的元素是: "<<endl;
vector<int>::iterator pos;
for(pos=v.begin(); pos!=v.end(); ++pos)
    cout<<*pos<<" ";
cout<<endl<<endl;

list<int> l(3);

transform(v.begin(),v.end(),l.begin(),square); // square 对诸元素平方

list<int>::iterator i;
cout<<"交换后链表容器的元素(经过平方计算)是: "<<endl;
for(i=l.begin();i!=l.end();i++)
    cout<<*i<<' ';
cout<<endl;
}

```

运行结果如图 5-9 所示。

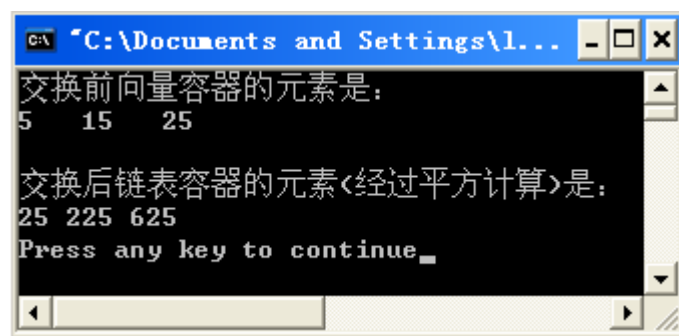


图 5-9 例 5-9 运行结果

3、替换算法 `replace()`

【功能】：算法将具有某特定值的元素替换为新值。

【例 5-10】创建一个整型向量容器，并在其尾部放入 13、25、27、25 和 29。然后用 100 替换值为 25 的元素。

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
void main()
{
    vector<int> v;
    v.push_back(13);
    v.push_back(25);
    v.push_back(27);
    v.push_back(25);
    v.push_back(29);
}

```

```

v.push_back(29);
cout<<"替换前向量容器的元素是: "<<endl;
vector<int>::iterator pos;
for(pos=v.begin(); pos!=v.end(); ++pos)
    cout<<*pos<<" ";
cout<<endl<<endl;

//在向量容器 v 中, 用值 100 替换值为 25 的元素的值
replace(v.begin(),v.end(),25,100);

cout<<"替换后向量容器的元素是: "<<endl;
for(pos=v.begin();pos!=v.end();pos++)
    cout<<*pos<<' ';
cout<<endl;
}

```

运行结果如图 5-10 所示。

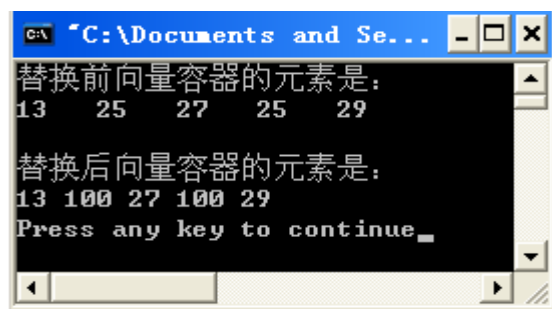


图 5-10 例 5-10 运行结果

4、条件替换 **replace_if**

【功能】：将符合某条件的元素替换为另一个值

【例 5-11】在整数 1~9 中，若某数不能被 2 除尽，则用 100 取代它

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
bool odd(int x){
    return x%2; //不能被 2 整除时返回 true
}
void main(){
    vector<int> v;
    for(int i=1;i<10;i++)
        v.push_back(i);

    replace_if(v.begin(),v.end(),odd,100);
}

```



```

vector<int>::iterator ilocation;
for(ilocation=v.begin();ilocation!=v.end();ilocation++)
    cout<<*ilocation<<' ';
    cout<<endl;
}

```

运行结果如图 5-11 所示。

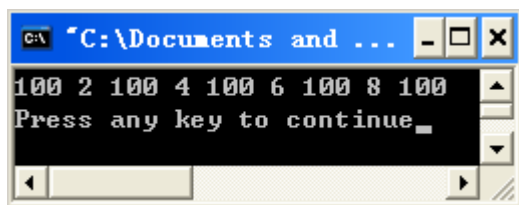


图 5-11 例 5-11 运行结果

5、n 次填充 fill_n

【功能】：用一给定值取代前 n 个元素

【例 5-12】

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

```

```

void main(){
    vector<int> v(10);

```

```

    fill_n(v.begin(),5,-1); //用“-1”填充 v 中的前 5 个元素

```

```

    vector<int>::iterator ilocation;
    for(ilocation = v.begin(); ilocation!=v.end(); ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;
}

```

运行结果如图 5-12 所示。

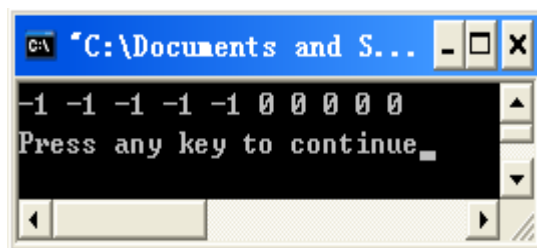


图 5-12 例 5-12 运行结果

6、替换 generate

【功能】：以某项操作的结果替换每个元素

【例 5-13】 随机生成 5 个元素，并放入向量容器的前头

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main(){
    vector<int> v(10);

    generate_n(v.begin(),5,rand); // 用 rand 算法生成 5 个随机数放入向量容器的前头

    vector<int>::iterator ilocation;
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;
}
```

运行结果如图 5-13 所示。

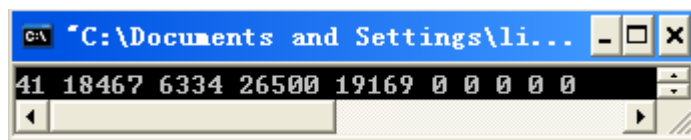


图 5-13 例 5-13 运行结果

7、条件移除 remove_if

【功能】：将满足条件的元素全部移除(注，此算法在标 5-2 中未给出)

【例 5-14】

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool even(int x) { return x%2?0:1;}

void main(){
    vector<int> v;
    for(int i=1;i<=10;i++)
        v.push_back(i);
    vector<int>::iterator ilocation,result;
    cout<<"移除前: ";
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;
    result=remove_if(v.begin(),v.end(),even); //移除能被 2 整除尽的元素
    cout<<"移除后: ";
```

```

for(ilocation=v.begin();ilocation!=result;ilocation++)
    cout<<*ilocation<<' ';
    cout<<endl;
}

```

运行结果如图 5-14 所示。

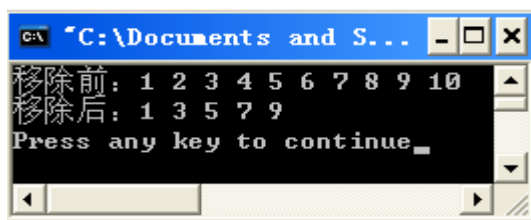


图 5-14 例 5-14 运行结果

8、剔除连续重复元素 unique

【功能】：移除比邻的重复元素(注，此算法在标 5-2 中未给出)

【例 5-15】

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main(){
    vector<int> v;
    v.push_back(2);
    v.push_back(6);
    v.push_back(6);
    v.push_back(6);
    v.push_back(9);
    v.push_back(6);
    v.push_back(3);
    vector<int>::iterator ilocation,result;
    result=unique(v.begin(),v.end());
    for(ilocation=v.begin();ilocation!=result;ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;
}

```

运行结果如图 5-15 所示。



图 5-15 例 5-15 运行结果

§5.3 排序算法

STL 有一系列算法都与排序有关。表 7-3 列出了 STL 中排序和搜索算法。同样，在 STL 程序中使用这些算法之前，必须包含头文件<algorithm>。常用的排序和搜索算法列在表 5-3 中。

表 5-3 STL 中常用的排序和搜索算法

序号	功能	函数名称	说 明
1	排 序	Sort	以很好的平均效率排序
		stable_sort	排序，并维持相同元素的原有顺序
		partial_sort	将序列的前一部分排好序
		partial_sort_copy	复制的同时将序列的前一部分排好序
2	第 n 个元素	nth_element	将第 n 各元素放到它的正确位置
3	二分检索	lower_bound	找到大于等于某值的第一次出现
		upper_bound	找到大于某值的第一次出现
		equal_range	找到（在不破坏顺序的前提下）可插入给定值的最大范围
		binary_search	在有序序列中确定给定元素是否存在
4	归 并	Merge	归并两个有序序列
		inplace_merge	归并两个接续的有序序列
5	序列结构上的集合操作	Includes	一序列为另一序列的子序列时为真
		set_union	构造两个集合的有序并集
		set_intersection	构造两个集合的有序交集
		set_difference	构造两个集合的有序差集
		set_symmetric_difference	构造两个集合的有序对称差集（并-交）
6	堆操作	make_heap	从序列构造堆/最大堆
		pop_heap	从堆中弹出元素
		push_heap	向堆中加入元素
		sort_heap	给堆排序，堆被破坏
7	最大和最小	min	两个值中较小的
		max	两个值中较大的
		min_element	序列中的最小元素
		max_element	序列中的最大元素
8	词典比较	lexicographical_compare	两个序列按字典序的第一个在前
9	排列生成器	next_permutation	按字典序的下一个排列
		prev_permutation	按字典序的前一个排列
10	数值算法	accumulate	累积和
		inner_product	内积
		partial_sum	局部求和
		adjacent_difference	临接差

下面通过举例来体会其中部分算法的作用和使用方法。

【例 5-16】堆操作包括：建堆、入堆、出堆、堆排序等算法

堆 (max-heap) 是二叉树，根结点总是大于等于其左右孩子结点 (递归定义)。在 STL 中，堆是最大堆 (max-heap)。堆中的元素被顺序地存储在一维数组中，且第一个元素值最大。

STL 提供四种堆操作算法：

1、make_heap()算法 将容器内的元素转化成 heap

形式一：

```
void make_heap(RandomAccessIterator beg,
               RandomAccessIterator end)
```

形式二：

```
void make_heap(RandomAccessIterator beg,
               RandomAccessIterator end,
               BinaryPredicate op)
```

- 两种形式都将区间[beg,end) 内的元素转化为 heap;
- op 是可有可无的二元判断式，它作为排序准则。

2、push_heap()算法 向堆中加入一个元素

形式一：

```
void push_heap (RandomAccessIterator beg,
                RandomAccessIterator end)
```

形式二：

```
void push_heap (RandomAccessIterator beg,
                RandomAccessIterator end,
                BinaryPredicate op)
```

● 两种形式都在本已经是 heap 的区间[beg,end) 尾部加入一个新元素，然后再将整个新的区间内的元素重新转化为 heap;

- op 是可有可无的二元判断式，它作为排序准则;
- 调用本函数之前，必须保证原来的整个区间已经是 heap。

3、pop_heap()算法 从堆中取出一个元素

形式一：

```
void pop_heap (RandomAccessIterator beg,
               RandomAccessIterator end)
```

形式二：

```
void pop_heap (RandomAccessIterator beg,
               RandomAccessIterator end,
               BinaryPredicate op)
```

● 一定要注意，以上两种形式都将[beg,end)内的最高元素，即第一个元素移到最后位置上，并将剩余区间[beg,end-1)内的元素组织起来，成为一个新的 heap。为了将移到最后位置上的那个已经出堆的元素物理地删除，一般在执行了 pop_heap()之后，紧接着要安排执行一次 pop_back()操作；

- op 是可有可无的二元判断式，它作为排序准则;
- 调用本函数之前，必须保证原来的整个区间已经是 heap。

4、sort_heap()算法 给堆排序，原容器不再是 heap 了

形式一：

```
void sort_heap (RandomAccessIterator beg,
               RandomAccessIterator end)
```

形式二：

```
void sort_heap (RandomAccessIterator beg,
               RandomAccessIterator end
               BinaryPredicate op)
```

● 以上两种形式都将 `heap[beg,end)` 转换成一个有序序列(默认升序)。注意，此算法一旦结束，该区间就不再是一个 heap 了。还要注意，原区间的最后是否有因 `pop_heap()` 操作而移来的元素；

- `op` 是可有可无的二元判断式，它作为排序准则；
- 调用本函数之前，必须保证原来的整个区间已经是 heap。

介绍了有关堆操作的四个算法之后，现研读本例如下代码：

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main(){
    vector<int> v;
    v.push_back(5);
    v.push_back(6);
    v.push_back(4);
    v.push_back(8);
    v.push_back(2);
    v.push_back(3);
    v.push_back(7);
    v.push_back(1);
    v.push_back(9);
    vector<int>::iterator ilocation;

    cout<<"此时向量 v 的元素为: \n";
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<" ";
    cout<<endl<<endl;
    make_heap(v.begin(),v.end()); //创建堆，即将一个区间转换成一个 heap
    cout<<"创建堆后，此时向量 v 的元素为: \n";
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<" ";
    cout<<endl<<endl;
    v.push_back(20);
```

```

cout<<"再向向量 v 尾部插入 20，但尚未压入堆，此时向量 v 的元素为：\n";
for(ilocation=v.begin();ilocation!=v.end();ilocation++)
    cout<<*ilocation<<" ";
cout<<endl<<endl;
push_heap(v.begin(),v.end()); //元素入堆（默认插入最后一个元素）
cout<<"将已加入向量 v 尾部的 20 压入堆后，此时向量 v 的元素为：\n";
for(ilocation=v.begin();ilocation!=v.end();ilocation++)
    cout<<*ilocation<<" ";
cout<<endl<<endl;
pop_heap(v.begin(),v.end()); //元素出堆(从 heap 中移除一个元素)
v.pop_back(); //把出堆元素物理删除

cout<<"从 heap 中移除一个元素后，此时向量 v 的元素为：\n";
for(ilocation=v.begin();ilocation!=v.end();ilocation++)
    cout<<*ilocation<<" ";
cout<<cout<<endl<<endl;

sort_heap (v.begin(),v.end()); //堆排序

cout<<"经堆排序后，此时向量 v 的元素为：\n";
for(ilocation=v.begin();ilocation!=v.end();ilocation++)
    cout<<*ilocation<<" ";
cout<<cout<<endl<<endl;
}

```

运行结果如图 5-14 所示（疑问：“004797E4”是什么？）。

图 5-16 例 5-16 运行结果

【例 5-17】堆排序 `sort_heap()` 算法(例 5-14 中已经用到此算法，此例可以省略)

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main()
{
    vector<int> v;
    v.push_back(3);
    v.push_back(9);
    v.push_back(6);
    v.push_back(3);
    v.push_back(17);
    v.push_back(20);
    v.push_back(12);
    vector<int>::iterator ilocation;

    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;

    make_heap(v.begin(),v.end()); //创建堆

    sort_heap(v.begin(),v.end()); //堆排序

    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl;
}
```

运行结果如图 5-17 所示。

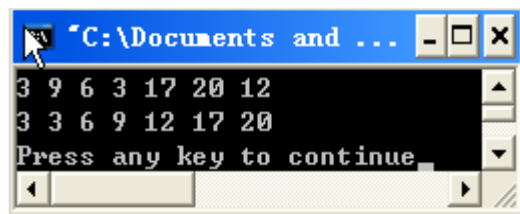


图 5-17 例 5-17 运行结果

【例 5-18】排序算法 `sort()` 算法和 `stable_sort()` 算法

它们都是对所有元素排序(默认为升序)。二者的区别是：`stable_sort()` 保证相等元素的原本相对次序在排序后保持不变(稳定性排序)。

不可以对 `list` 调用这些算法，因为 `list` 不支持随即存取迭代器。不过 `list` 提供了一个成员函数 `sort()`，可以用来对自身元素进行排序。

```
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)
```

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void main()
{
    vector<int> v;
    v.push_back(2);
    v.push_back(8);
    v.push_back(-15);
    v.push_back(90);
    v.push_back(26);
    v.push_back(7);
    v.push_back(23);
    v.push_back(30);
    v.push_back(-27);
    v.push_back(39);
    v.push_back(55);
    vector<int>::iterator ilocation;

    cout<<"排序前: "<<endl;
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
    cout<<endl<<endl;

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

    cout<<"排序后: "<<endl;
    for(ilocation=v.begin();ilocation!=v.end();ilocation++)
        cout<<*ilocation<<' ';
```

```
cout<<endl;
}运行结果如图 5-18 所示。
```

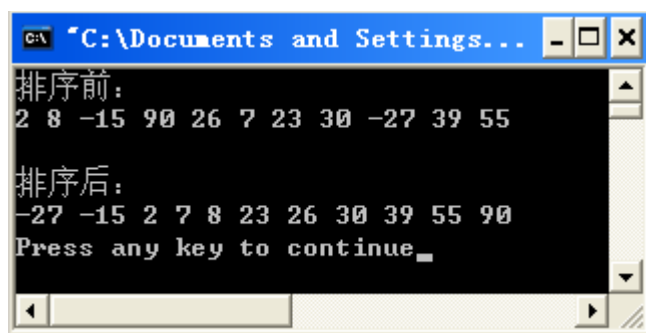


图 5-18 例 5-18 运行结果

§5.4 数值算法

STL 提供了四个通用的全局算法，使用这些算法需要包含头文件<numeric>。

一、**accumulate 算法**： 计算序列中所有元素的和。该函数重载有两个版本如下：

1、形式一

```
T accumulate(InputIterator beg, InputIterator end, T iniValue);
```

2、形式二

```
T accumulate(InputIterator beg, InputIterator end, T iniValue, BinaryFunc op);
```

形式一是计算 iniValue 和区间[beg, end) 内所有元素的总和。

形式二是计算 iniValue 和区间[beg, end) 内每一个元素进行 op 运算的结果。

●因此，对于如下数值序列：

a1 a2 a3 a4 ...

形式一是计算：iniValue + a1 + a2 + a3 + a4 + ...；

形式二是计算：iniValue op a1 op a2 op a3 op a4 op ...。

二、**inner_product**： 对两个序列做内积(对应元素相乘，再求和)，并将内积加到一个输入的初始值上。重载版本使用用户定义的操作。

1、形式一

```
T inner_product (InputIterator1 beg1, InputIterator1 end1,
                  InputIterator2 beg2, T iniValue);
```

2、形式二

```
T inner_product (InputIterator1 beg1, InputIterator1 end1,
                  InputIterator2 beg2, T iniValue
                  BinaryFunc op1, BinaryFunc op2);
```

形式一是计算并返回 $[beg1, end1)$ 区间和“以 $beg2$ 为起始的区间”的对应元素组（再加上 $iniValue$ ）的内积。具体地说，也就是针对“两区间内的每一组对应元素”调用以下表达式：

$$iniValue = iniValue + elem1 * elem2$$

形式二将 $[beg1, end1)$ 区间和“以 $beg2$ 为起始的区间”的对应元素组进行 $op2$ 运算，然后再和 $iniValue$ 进行 $op1$ 运算，并将结果返回。具体地说，也就是针对“两区间内的每一组对应元素”调用以下表达式：

$$iniValue = op1(iniValue, op2(elem1, elem2))$$

●因此，对于如下数值序列：

a1 a2 a3 ...

b1 b2 b3 ...

形式一是计算并返回： $iniValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + \dots$ ；

形式二是计算并返回： $iniValue \ op1 \ (a1 \ op2 \ b1) \ op1 \ (a2 \ op2 \ b2) \ op1 \ (a3 \ op2 \ b3) \ op1 \ \dots$ 。

三、partial_sum: 对序列中部分元素的值进行累加，并将结果保存在另一个序列中。

1、形式一

OutputIterator

```
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEed,
             OutputIterator destBeg );
```

2、形式二

OutputIterator

```
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEed,
             OutputIterator destBeg , BinaryFunc op);
```

形式一是计算区间 $[sourceBeg, sourceEed)$ 中每个元素的部分和，然后将结果写入以 $destBeg$ 为起点的目标区间。

形式二是将区间 $[sourceBeg, sourceEed)$ 中每个元素和其先前所有元素进行 op 运算，并将结果写入以 $destBeg$ 为起点的目标区间。

●因此，对于如下数值序列：

a1 a2 a3 ...

形式一是计算： $a1, a1 + a2, a1 + a2 + a3, \dots$ ；

形式二是计算： $a1, a1 \ op \ a2, a1 \ op \ a2 \ op \ a3, \dots$ 。

四、adjacent_difference: 计算序列中相邻元素的差，并将结果保存在另一个序列中。

1、形式一

OutputIterator

```
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEed,
             OutputIterator destBeg );
```

2、形式二

OutputIterator

```
partial_sum (InputIterator sourceBeg,
             InputIterator sourceEed,
             OutputIterator destBeg, BinaryFunc op);
```

形式一是计算区间[sourceBeg, sourceEed) 中每个元素和其紧邻前驱元素的差，并将结果写入以 destBeg 为起点的目标区间。

形式二是针对区间[sourceBeg, sourceEed) 中每个元素和其先前所有元素调用 op 操作，并将结果写入以 destBeg 为起点的目标区间。

●因此，对于如下数值序列：

a1 a2 a3 ...

形式一是计算：a1, a2 - a1, a3 - a2, a4 - a3 ...;

形式二是计算：a1, a2 op a1, a3 op a2, a4 op a3 ...。

【例 5-19】STL 数值算法举例

```
#include <iostream>
#include <numeric>
#include <functional>
#include <vector>
using namespace std;

void main()
{
    int iarray[]={1,2,3,4,5};
    vector<int>ivector(iarray,
                      iarray+sizeof(iarray)/sizeof(int));//此行求数组元素个数

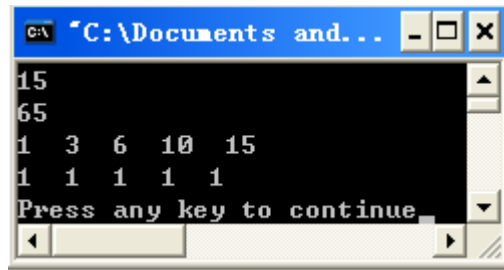
    cout<<accumulate(ivector.begin(),ivector.end(),0)<<endl; //元素的累计

    //向量的内积
    cout<<inner_product(ivector.begin(),ivector.end(),ivector.begin(),10)<<endl;

    //向量容器中元素局部求和
    partial_sum(ivector.begin(),ivector.end(),
                ostream_iterator<int>(cout," "));
    cout<<endl;

    //向量容器中相邻元素的差值
    adjacent_difference(ivector.begin(),ivector.end(),
                        ostream_iterator<int>(cout," "));
    cout<<endl;
}
```

运行结果如图 5-19 所示。



```
C:\Documents and...  
15  
65  
1 3 6 10 15  
1 1 1 1 1  
Press any key to continue
```

图 5-19 例 5-19 运行结果

附录 STL 基本容器常用成员函数一览

§1 各类容器共性成员函数

各类容器一般来说都有下列成员函数：

- 默认构造函数 提供容器默认初始化的构造函数。
- 复制构造函数 将容器初始化为现有同类容器副本的构造函数
- 析构函数 不再需要容器时进行内存整理的析构函数
- empty 容器中没有任何元素时返回 `true`, 否则返回 `false`
- max_size 返回容器中最大元素个数
- size 返回容器中当前元素个数
- operator= 将一个容器赋给另一个容器
- operator< 如果第一个容器小于第二个容器, 返回 `true`, 否则返回 `false`,
- operator<= 如果第一个容器小于或等于第二个容器, 返回 `true`, 否则返回 `false`
- operator> 如果第一个容器大于第二个容器, 返回 `true`, 否则返回 `false`
- operator>= 如果第一个容器大于或等于第二个容器, 返回 `true`, 否则返回 `false`
- operator== 如果第一个容器等于第二个容器, 返回 `true`, 否则返回 `false`
- operator!= 如果第一个容器不等于第二个容器, 返回 `true`, 否则返回 `false`
- swap 交换两个容器的元素

§2 顺序容器和关联容器共有函数

- begin 返回 `iterator` 或 `const_iterator`, 引用容器第一个元素位置
- end 返回 `iterator` 或 `const_iterator`, 引用容器最后一个元素后面一个位置
- rbegin 返回 `reverse_iterator` 或 `const_reverse_iterator`, 引用容器最后一个元素
- rend 返回 `reverse_iterator` 或 `const_reverse_iterator`, 引用容器首个元素前面一位
- erase 从容器中清除一个或几个元素
- clear 清除容器中所有元素

§3 容器比较

■ `vector` (连续的空间存储, 可以使用 `[]` 操作符) 快速的访问随机的元素, 快速的在末尾插入元素, 但是在序列中间位置的插入, 删除元素要慢, 而且如果一开始分配的空间不够的话, 有一个重新分配更大空间, 然后拷贝的性能开销。

■ `deque` (小片的连续, 小片间用链表相连, 实际上内部有一个 `map` 的指针, 因为知道类型, 所以还是可以使用 `[]`, 只是速度没有 `vector` 快) 快速的访问随机的元素, 快速的在开始和末尾插入元素, 随机的插入, 删除元素要慢, 空间的重新分配要比 `vector` 快, 重新分配空间后, 原有的元素不需要拷贝。对 `deque` 的排序操作, 可将 `deque` 先复制到 `vector`, 排序后在复制回 `deque`。

■ `list` (每个元素间用链表相连) 访问随机元素不如 `vector` 快, 随机的插入元素比 `vector` 快, 对每个元素分配空间, 所以不存在空间不够, 重新分配的情况。

■ `set` 内部元素唯一, 用一棵平衡树结构来存储, 因此遍历的时候就排序了, 查找也比较快。

■ `map` 一对一的映射的结合, `key` 不能重复。

§4 `vector` 容器常用成员函数

(1) 构造函数

- `vector();` 创建一个空 `vector`
- `vector(int nSize);` 创建一个 `vector`, 元素个数为 `nSize`
- `vector(int nSize, const T& t);` 创建一个 `vector`, 元素个数为 `nSize`, 且值均为 `t`

- `vector(const vector&);` 拷贝构造函数

(2)增加函数

- `void push_back(const T& x);` 向量尾部增加一个元素 `x`
- `iterator insert(iterator it, const T& x);` 向量中某一元素前增加一个元素 `x`
- `void insert(iterator it, int n, const T& x);` 向量中某一元素前增加 `n` 个相同元素 `x`
- `void insert(iterator it, const_iterator first, const_iterator last);` 向量中某一元素前插入另一个相同类型向量的`[first,last)`间的数据

(3)删除函数

- `iterator erase(iterator it);` 删除向量中某一个元素
- `iterator erase(iterator first, iterator last);` 删除向量中`[first,last)`中元素
- `void pop_back();` 删除向量中最后一个元素
- `void clear();` 删除向量中所有元素

(4)遍历函数

- `reference at(int pos);` 返回 `pos` 位置元素的引用
- `reference front();` 返回首元素的引用
- `reference back();` 返回尾元素的引用
- `iterator begin();` 返回向量头指针, 指向第一个元素
- `iterator end();` 返回向量尾指针, 不包括最后一个元素, 在其下面
- `reverse_iterator rbegin();` 反向迭代器, 功能等同 `iterator end()`
- `reverse_iterator rend();` 反向迭代器, 功能等同 `iterator begin()`

(5)判断函数

- `bool empty() const;` 向量是否为空? 若 `true`, 则向量中无元素。

(6)大小函数

- `int size() const;` 返回向量中元素的个数
- `int capacity() const;` 返回当前向量所能容纳的最大元素值
- `int max_size() const;` 返回最大可允许的 `vector` 元素数量值

(7)其它函数

- `void swap(vector&);` 交换两个同类型向量的数据
- `void assign(int n, const T& x);` 向量中第 `n` 个元素设置成元素 `x`
- `void assign(const_iterator first, const_iterator last);` 向量中`[first,last]`元素设置成当前向量元素

§5 deque 容器常用成员函数

(1)构造函数

- `deque();` 创建一个空 `deque`
- `deque(int nSize);` 创建一个 `deque`, 元素个数为 `nSize`
- `deque(int nSize, const T& t);` 创建一个 `deque`, 元素个数为 `nSize`, 且值均为 `t`
- `deque (const deque &);` 拷贝构造函数

(2)增加函数

- `void push_front(const T& x);` 双端队列头部增加一个元素 `x`
- `void push_back(const T& x);` 双端队列尾部增加一个元素 `x`
- `iterator insert(iterator it, const T& x);` 双端队列中某一元素前增加一个元素 `x`
- `void insert(iterator it, int n, const T& x);` 双端队列中某一元素前增加 `n` 个相同元素

`x`

- `void insert(iterator it, const_iterator first, const_iterator last);` 双端队列中某一元素前插入另一个相同类型向量的`[first,last)`间的数据

(3)删除函数

- `iterator erase(iterator it);` 删除双端队列中某一个元素
- `iterator erase(iterator first, iterator last);` 删除双端队列中`[first,last)`中元素
- `void pop_front();` 删除双端队列中最后一个元素
- `void pop_back();` 删除双端队列中最后一个元素
- `void clear();` 删除双端队列中所有元素

(4)遍历函数

- `reference at(int pos);` 返回 `pos` 位置元素的引用
- `reference front();` 返回首元素的引用
- `reference back();` 返回尾元素的引用
- `iterator begin();` 返回向量头指针, 指向第一个元素
- `iterator end();` 返回向量尾指针, 不包括最后一个元素, 在其下面
- `reverse_iterator rbegin();` 反向迭代器, 功能等同 `iterator end()`
- `reverse_iterator rend();` 反向迭代器, 功能等同 `iterator begin()`

(5)判断函数

- `bool empty() const;` 向量是否为空? 若 `true`, 则向量中无元素。

(6)大小函数

- `int size() const;` 返回向量中元素的个数
- `int max_size() const;` 返回最大可允许的双端队列元素数量值

(7)其它函数

- `void swap(vector&);` 交换两个同类型向量的数据
- `void assign(int n, const T& x);` 向量中第 `n` 个元素设置成元素 `x`
- `void assign(const_iterator first, const_iterator last);` 向量中`[first,last)`元素设置成当前向量元素

说明: `deque` 可以通过 `push_front` 直接在容器头增加元素, 通过 `pop_front` 直接删除容器头元素, 这一点是 `vector` 元素不具备的。

§6 list 容器常用成员函数

(1)构造函数

- `list<Elem> c;` 创建一个空的 `list`
- `list<Elem> c1(c2);` 拷贝另一个同类型元素的 `list`
- `list<Elem> c(n);` 创建 `n` 个元素的 `list`, 每个元素值由缺省构造函数确定
- `list<Elem> c(n, elem);` 创建 `n` 个元素的 `list`, 每个元素值为 `elem`

- `list<Elem> c(begin, end);` 由迭代器创建 `list`, 迭代区间为 `[begin, end]`

(2) 大小、判断空函数

- `int size() const` ; 返回容器元素个数
- `bool empty() const` ; 判断容器是否空, 若返回 `true`, 表明容器已空

(3) 增加、删除函数:

- `void push_back(const T& x)` ; `list` 容器尾元素后增加一个元素 `x`
- `void push_front(const T& x)` ; `list` 容器首元素前增加一个元素 `x`
- `void pop_back();` ; 删除容器尾元素, 当且仅当容器不为空
- `void pop_front();` ; 删除容器首元素, 当且仅当容器不为空
- `void remove(const T& x);` ; 删除容器中所有元素值等于 `x` 的元素
- `void clear();` ; 删除容器中所有元素
- `iterator insert(iterator it, const T& x = T());` ; 在迭代器指针 `it` 前插入元素 `x`, 返回 `x` 迭代器指针
- `void insert(iterator it, size_type n, const T& x);` ; 在迭代器指针 `it` 前插入 `n` 个相同元素 `x`
- `void insert(iterator it, const_iterator first, const_iterator last);` ; 把 `[first, last)` 间的元素插入迭代器指针 `it` 前
- `iterator erase(iterator it);` ; 删除迭代器指针 `it` 对应的元素
- `iterator erase(iterator first, iterator last);` ; 删除迭代器指针 `[first, last)` 间的元素

(4) 遍历函数

- `iterator begin();` ; 返回首元素的迭代器指针
- `iterator end();` ; 返回尾元素后的迭代器指针, 而不是尾元素的迭代器指针
- `reverse_iterator rbegin();` ; 返回尾元素的逆向迭代器指针, 用于逆向遍历容器
- `reverse_iterator rend();` ; 返回首元素前的逆向迭代器指针, 用于逆向遍历容器
- `reference front();` ; 返回首元素的引用
- `reference back();` ; 返回尾元素的引用

(5) 操作函数

- `void sort();` ; 容器内所有元素排序, 默认是升序
- `template <class Pred> void sort(Pred pr);` ; 容器内所有元素根据预判定函数 `pr` 排序
- `void swap(list& str);` ; 两 `list` 容器交换功能
- `void unique();` ; 容器内相邻元素若有重复的, 则仅保留一个
- `void splice(iterator it, list& x);` ; 队列合并函数, 队列 `x` 所有元素插入迭代指针 `it` 前, `x` 变成空队列
- `void splice(iterator it, list& x, iterator first);` ; 队列 `x` 中移走 `[first, end)` 间元素插入迭代指针 `it` 前
- `void splice(iterator it, list& x, iterator first, iterator last);` ; `x` 中移走 `[first, last)` 间元素插入迭代指针 `it` 前
- `void reverse();` ; 反转容器中元素顺序

§7 队列和堆栈常用成员函数

(1) 构造函数

- `queue(class T, class Container=deque<T>);` ; 创建元素类型为 `T` 的空队列, 默认容

器是 deque

•stack(class T, class Container=deque<T>); 创建元素类型为 T 的空堆栈，默认容器是 deque

(2)操作函数

- bool empty(); 如果队列(堆栈)为空返回 true, 否则返回 false
- int size(); 返回队列(堆栈)中元素数量
- void push(const T& t); 把 t 元素压入队尾(栈顶)
- void pop(); 当队列(栈)非空情况下, 删除队头(栈顶)元素 队列独有函数:
- T& front(); 当队列非空情况下, 返回队头元素引用
- T& back(); 当队列非空情况下, 返回队尾元素引用堆栈独有函数:
- T& top(); 当栈非空情况下, 返回栈顶元素的应用

§8 优先队列常用成员函数

(1)构造函数

•priority_queue(const Pred& pr = Pred(), const allocator_type& al = allocator_type());
创建元素类型为 T 的空优先队列, Pred 是二元比较函数, 默认是 less<T>

•priority_queue(const value_type *first, const value_type *last, const Pred& pr = Pred(), const allocator_type& al = allocator_type()); 以迭代器[first,last)指向元素创建元素类型为 T 的优先队列, Pred 是二元比较函数, 默认是 less<T>。

(2)操作函数

队列和堆栈共有函数:

- bool empty(); 如果优先队列为空返回 true, 否则返回 false
- int size(); 返回优先队列中元素数量
- void push(const T& t); 把 t 元素压入优先队列
- void pop(); 当优先队列非空情况下, 删除优先级最高元素
- T& top(); 当优先队列非空情况下, 返回优先级最高元素的引用

§9 集合常用成员函数

set、multiset 都是集合类, 差别在于 set 中不允许有重复元素, multiset 中允许有重复元素。

(1)构造函数

- set(const Pred& comp = Pred(), const A& al = A()); 创建空集合
- set(const set& x); 拷贝构造函数
- set(const value_type *first, const value_type *last, const Pred& comp = Pred(), const A& al = A()); 拷贝[first,last)之间元素构成新集合
- multiset(const Pred& comp = Pred(), const A& al = A()); 创建空集合
- multiset(const multiset& x); 拷贝构造函数
- multiset(const value_type *first, const value_type *last, const Pred& comp = Pred(), const A& al = A()); 拷贝[first,last)之间元素构成新集合

(2)大小、判断空函数

- `int size() const;` 返回容器元素个数
- `bool empty() const;` 判断容器是否空，若返回 `true`, 表明容器已空

(3) 增加、删除函数

- `pair<iterator, bool> insert(const value_type& x);` 插入元素 `x`
- `iterator insert(iterator it, const value_type& x);` 在迭代指针 `it` 处，插入元素 `x`
- `void insert(const value_type *first, const value_type *last);` 插入 `[first, last)` 间元素
- `iterator erase(iterator it);` 删除迭代指针 `it` 处元素
- `iterator erase(iterator first, iterator last);` 删除 `[first, last)` 迭代指针间元素
- `size_type erase(const Key& key);` 删除元素值等于 `key` 的元素

(4) 遍历函数

- `iterator begin();` 返回首元素的迭带器指针
- `iterator end();` 返回尾元素后的迭带器指针，而不是尾元素的迭带器指针
- `reverse_iterator rbegin();` 返回尾元素的逆向迭带器指针，用于逆向遍历容器
- `reverse_iterator rend();` 返回首元素前的逆向迭带器指针，用于逆向遍历容器

(5) 操作函数

- `const_iterator lower_bound(const Key& key);` 返回容器元素等于 `key` 迭代指针，否则返回 `end()`
- `const_iterator upper_bound(const Key& key);`
- `int count(const Key& key) const;` 返回容器中元素值等于 `key` 的元素个数
- `pair<const_iterator, const_iterator> equal_range(const Key& key) const;` 返回容器中元素值等于 `key` 的迭代指针 `[first, last)`
- `const_iterator find(const Key& key) const;` 查找功能，返回元素值等于 `key` 迭带器指针
- `void swap(set& s);` 交换单集合元素
- `void swap(multiset& s);` 交换多集合元素

§10 映射常用成员函数

常用的映射类是 `map`, `multimap`。在前述的各个容器中，仅保存着一样东西，但是在映射中将会得到两样东西：关键字以及作为对关键字进行查询得到的结果值，即一对值 `<Key, Value>`。`map` 单映射中 `Key` 与 `Value` 是一一对应的关系，`multimap` 多映射中 `Key` 与 `Value` 可以是一对多的关系。

(1) 构造函数

- `map(const Pred& comp = Pred(), const A& al = A());` 创建空映射
- `map(const map& x);` 拷贝构造函数
- `map(const value_type *first, const value_type *last, const Pred& comp = Pred(), const A& al = A());` 拷贝 `[first, last)` 之间元素构成新映射
- `multimap(const Pred& comp = Pred(), const A& al = A());` 创建空映射
- `multimap(const multimap& x);` 拷贝构造函数
- `multimap(const value_type *first, const value_type *last, const Pred& comp = Pred(), const A& al = A());` 拷贝 `[first, last)` 之间元素构成新映射

(2) 大小、判断空函数

- `int size() const;` 返回容器元素个数
- `bool empty() const;` 判断容器是否空, 若返回 `true`, 表明容器已空

(3) 增加、删除函数

- `iterator insert(const value_type& x);` 插入元素 `x`
- `iterator insert(iterator it, const value_type& x);` 在迭代指针 `it` 处插入元素 `x`
- `void insert(const value_type *first, const value_type *last);` 插入`[first, last)`间元素
- `iterator erase(iterator it);` 删除迭代指针 `it` 处元素
- `iterator erase(iterator first, iterator last);` 删除`[first, last)` 迭代指针间元素
- `size_type erase(const Key& key);` 删除键值等于 `key` 的元素

(4) 遍历函数

- `iterator begin();` 返回首元素的迭代器指针
- `iterator end();` 返回尾元素后的迭代器指针, 而不是尾元素的迭代器指针
- `reverse_iterator rbegin();` 返回尾元素的逆向迭代器指针, 用于逆向遍历容器
- `reverse_iterator rend();` 返回首元素前的逆向迭代器指针, 用于逆向遍历容器

(5) 操作函数

- `const_iterator lower_bound(const Key& key);` 返回键值等于 `key` 迭代指针, 否则返回 `end()`
- `const_iterator upper_bound(const Key& key);`
- `int count(const Key& key) const;` 返回容器中键值等于 `key` 的元素个数
- `pair<const_iterator, const_iterator> equal_range(const Key& key) const;` 返回容器中键值等于 `key` 的迭代指针`[first, last)`
- `const_iterator find(const Key& key) const;` 查找功能, 返回键值等于 `key` 迭代器指针
- `void swap(map& s);` 交换单映射元素
- `void swap(multimap& s);` 交换多映射元素

(6) 特殊函数

- `reference operator[](const Key& k);` 仅用在单映射 `map` 类中, 可以以数组的形式给映射添加键---值对, 并可返回值的引用。