不要重复发明轮子

快捷开发　信手拈来

学习本书内容需要您已经有了一定的C/C++基础

本书力求将C++ STL所涉及的问题，一一列出并结合实例详细阐述，同时融入作者多年实践经验，目的只有一个：让初学者能够少走些弯路。

本书的示例程序皆是作者亲自编写或从MSDN中摘录，精练分析，简单易学，有助于读者掌握STL开发技能，同时希望读者在学习的时候不要漏掉任何一个例题。

C++ STL 标准程序库开发指南

【第2版】

闫常友　王敏　编著

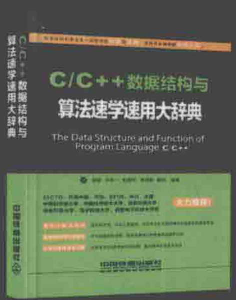
中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

扫码下载书中源代码
下载地址：http://www.crphdm.com/2016/1208/13033.shtml

不要重复发明轮子

C++ STL 标准程序库开发指南【第2版】

中国铁道出版社编程语言畅销图书推荐：



书 名：C/C++数据结构与算法速学速用大辞典
定 价：55.00元
ISBN 978-7-113-21699-3

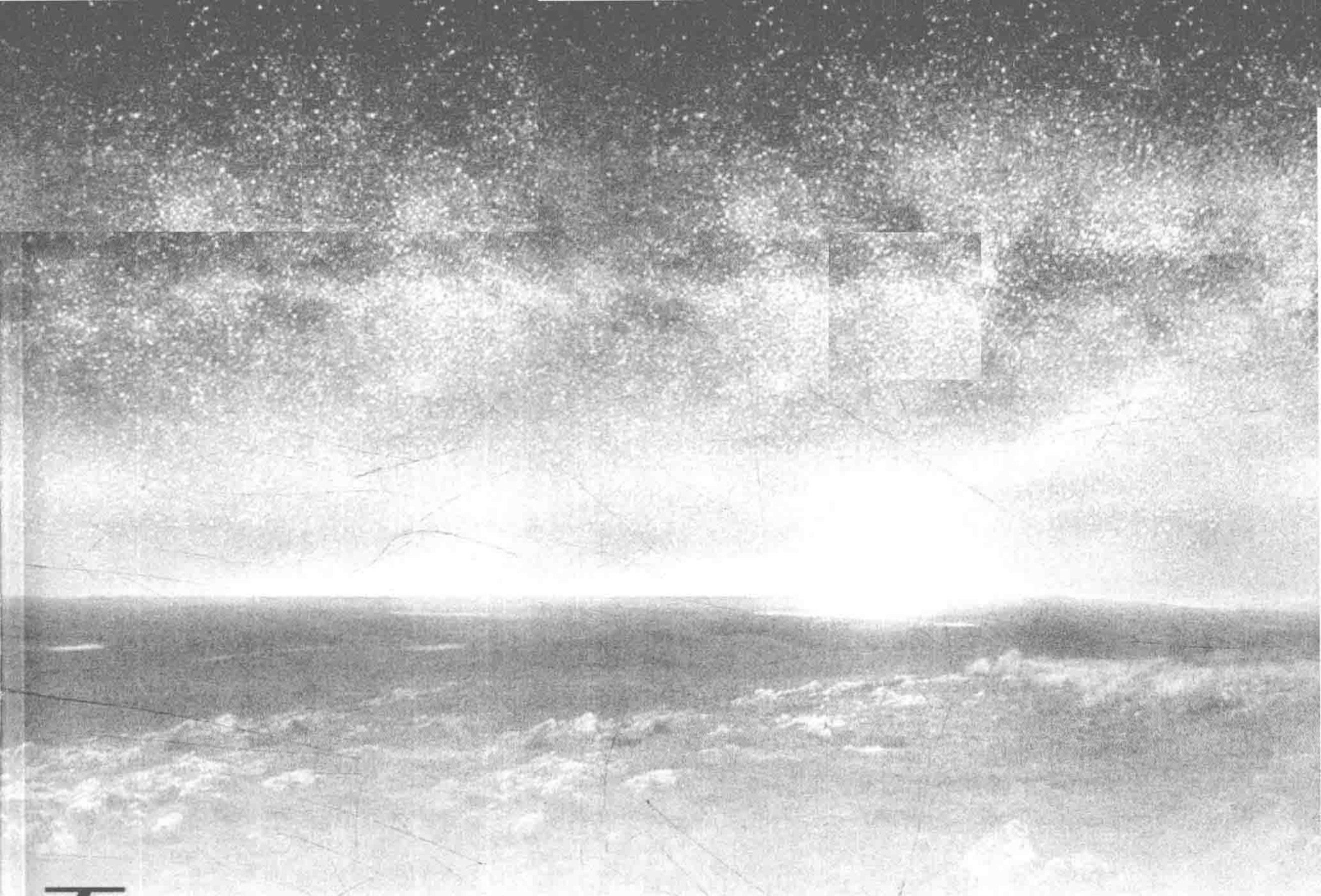
书 名：Java常用算法手册（第3版）（含盘）
定 价：59.80元
ISBN 978-7-113-17379-1

书 名：移动Web开发从入门到精通
定 价：59.80元
ISBN 978-7-113-21985-7

上架建议：计算机/程序设计/C++

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE
地址：北京市西城区右安门西街8号
邮编：100054
网址：http://www.tdpress.com

ISBN 978-7-113-22377-9
9 787113 223779 >
定价：79.00元



不要重复发明轮子 C++ STL 标准程序库开发指南

【第2版】

闫常友 王敏 编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

内 容 简 介

本书献给喜欢 C++ STL 的朋友，当您看见这本书时您应该对 C++ STL 已有一定的了解，希望继续深造；或者您已经暗下决心来学习它。那么本书将是您最佳的选择。

如果您已经有了一些 C/C++基础，那么学习起来会更加轻松。本书分为 14 章，按照章节的先后顺序，由浅入深地讲解 C++ STL 应用开发技术。本书力求将 STL 涉及的问题一一列出讲解，使初学者能够少走弯路，并且所有的示例程序都是作者亲自编写或从 MSDN 中摘录的，简单易学，有助于读者掌握 STL 的知识。

图书在版编目（CIP）数据

C++ STL标准程序库开发指南 / 闫常友，王敏编著．—2 版．—北京：中国铁道出版社，2017.1

ISBN 978-7-113-22377-9

Ⅰ．①C… Ⅱ．①闫… ②王… Ⅲ．①C 语言－程序设计 Ⅳ．①TP312.8

中国版本图书馆 CIP 数据核字（2016）第 232920 号

书　　名：C++ STL 标准程序库开发指南（第 2 版）
作　　者：闫常友　王　敏　编著

责任编辑：荆　波　　　**读者服务热线：**010-63560056
责任印制：赵星辰　　　**封面设计：**MXK DESIGN STUDIO

出版发行：中国铁道出版社（北京市西城区右安门西街 8 号　　邮政编码：100054）
印　　刷：中国铁道出版社印刷厂
版　　次：2013 年 5 月第 1 版　2017 年 1 月第 2 版　2017 年 1 月第 1 次印刷
开　　本：787mm×1 092mm　1/16　**印张：**33.25　**字数：**892 千
书　　号：ISBN 978-7-113-22377-9
定　　价：79.00 元

版权所有　侵权必究

凡购买铁道版图书，如有印制质量问题，请与本社读者服务部联系调换。电话：（010）51873174

打击盗版举报电话：（010）51873659

前　言　Foreword

在开发语言中，C++的应用越来越广泛。而 C++ STL 是标准的 C++模板库，是算法和其他一些标准组件的集合，可以说是世界上众多技术人员多年经验的总结。STL 的目的是标准化组件，这样就不用重复开发，即可使用现成的组件，提高了开发效率。STL 是 C++标准的一部分，开发时不用额外安装插件包。

编写一本全面的、透彻的 C++ STL 的书籍，有助于广大程序开发人员深入掌握 C++ STL 的编程技巧。从 2003 年 10 月 15 日，国际标准 ISO/IEC-14882 second edition 颁布以来，已经过去了十多个年头。2011 年夏天，笔者对目前的 C++类书籍做了调研。除国外的翻译版本之外，国内出版的书籍少之又少，图书的质量良莠不齐。限于国外专家的书写习惯和翻译人员的水平，多数翻译版本表达不够准确。国内出版的多数书籍不能深刻地体现 C++ STL 的博大精深。

国内的程序开发人员对 C++和 STL 的学习和掌握，也停留在较低的水平。当我们打开 C++或 STL 的函数声明或函数定义时，我们应该为 C++语言专家的深邃思想和逻辑思维而折服。

我们在学习使用 C++ STL 时，有时会深刻地认识到自身对计算机语言知识的匮乏，并且深刻地认识现代社会及未来社会中计算机语言的重要性。不管哪一种语言，其算法库的博大精深是毋庸置疑的。程序员在编程过程中只有充分利用这些算法库，才能发挥事半功倍的效果。当然每种算法库也有自己的局限性，遇到这种情况就需要程序员自己编写自定义的算法了。

这里不再赘述，也许您现在已对 C++ STL 产生了浓厚的兴趣。那么尝试读一读本书，相信您会爱上本书。

STL 的发展和本书改版

C++ STL 是学好算法之后的关键学习点，本书配套我社《C/C++常用算法手册》使用，可以在学习算法知识之后，来学习这套积累了程序员大牛智慧结晶的 STL，可以应用在关键的算法和工作应用中。

本次改版，我们主要把代码重新进行了调试，修改了上一版本书中因为匆忙而产生的代码小错误，并对全书所有实例的讲解意图进行了更加详细的说明。大规模重写了全书的第 1 章，让读者更加明白 C++模板技术的起源和用途，明白 C++ STL 的重要性。让读者的学习台阶不再陡峭，读者学习了 C++语言，结合 C/C++算法类图书，即可学习本书。

本书适合的读者

- 有一定的 C/C++语言基础，想从事 C++程序较高层级开发的初学者；
- 学习过 C++语言，需要用 C++做一定规模开发的读者；
- 热爱 C/C++程序开发的所有读者。

作者团队

闫常友，高级工程师，电力系统及其自动化专业，九三学社社员，热爱计算机语言和软件开发，尤其对 C/C++系列有独到的见解和深厚的感情。

王敏，高级工程师，电力系统及其自动化专业，长期从事 C/C++语言的开发和项目实践，经验丰富。

致谢

本书的编写是非常艰难的。首先作者对目前市场上现有的 C/C++书籍做了深入调研，利用了无数个夜晚，编写了所有的例题，并完成调试。感谢王敏女士编写了第 1 章的部分内容。感谢本书的修订者袁静静完成了本书的修订，这使得读者对 C++的理解更加清晰，对本书的内容，也更加完善。其次，感谢负责本书的第一个审读者兼读者刘霞。她在写作过程中给予了我很多的鼓励和支持，使我得以写完本书。再次，还要感谢一下笔者对程序开发多年来的深深的热爱。笔者从 1998 年喜欢上 C/C++语言，目前已近不惑之年，把自己之前的一些想法和经验通过本书表达出来，也是值得庆幸的。

闫常友

2016 年 8 月

目　录　Contents

第 1 章　类模板简介

第 2 章 C++中的字符串

第 3 章 容器

第 4 章 C++中的算法

第 5 章 迭代器（Iterator）

第 6 章　STL 的数值计算

第 7 章　输入/输出流

第 8 章 异常处理

第 9 章　通用工具

第 10 章 语言支持

第 11 章 检测库详解

第 12 章 国际化库详解

第 13 章 仿函数

第 14 章 配置器

参考文献

第 1 章　类模板简介

在现今的C++标准模板库中，几乎所有东西都被设计为template形式，不支持模板，就无法使用标准程序库。模板可以认为是针对一个或多个尚未明确的类型而编写一套函数或类型。模板是C++的一个新特性。通过使用模板，C++允许推迟对某些类型的选择，直到想使用模板或者对模板进行专门化处理时。使用模板，可以让程序员面对相似而又略有不同的特性时，更快捷地编写代码，提高开发效率。

本章主要讲述C/C++语言中常用的一些基本概念，如何定义类模板，介绍成员模板、友元模板、函数模板、类模板的参数，最后介绍模板库。

1.1　C++为什么需要模板功能

理解了这个话题，程序员才能深度理解STL，才能用好STL，所以，本书的开篇，让我们深入浅出地理解C++的引入模板功能的思想。

1.1.1　计算机实在是太傻了

在大多数人的眼里，计算机既神秘又能干。

但在程序员的眼里，计算机实在是又蠢又笨。只不过运算的比人类速度快（电子的速度吗），记忆力好（能存储很多数据）。而且不给指令还什么都干不了。就是给指令，写一个程序，计算机也不灵活。比如，在C++中，同样一个加法，居然要给出不同的数据类型的运算方式。

整数型，我们写这样一个函数：

```
int add(int m,int n)
{
return m+n;
}
```

浮点型，我们写这样一个函数：

```
float add(float i,float j)
{
return i+j;
}
……
```

每一种类型，我们都必须给计算机写一个函数。

感觉实在是太烦琐了，可是，计算机就是不灵活，它们不具备人类的基本思维。

所以，我们常说电子计算机只是外延和加强了人类的某些功能，

如果说电子计算机能替代人类统治世界，说这话的人实在是太外行了。

1.1.2 类和函数重载部分解决了这个问题

可是计算机程序，就是最大限度来模拟人类的思维，让计算机不断接近人类的认知能力，成为人类最好的外延工具之一。所以，远比计算机聪明的科学家和工程师想了很多办法。其中一个是面向对象开发技术。

通过类的封装和函数重载，部分解决了这个问题。比如我们常常这样做。

```
class calc
{
……
int add(int m,int n)
{
return m+n;
}
float add(float i,float j)
{
return i+j;
}
……

}
```

这样的类产生后，用 calc.add 方法时，就不用考虑这两个参数的具体数据类型。

某种程度上，实现了一定的柔性处理，计算机变得聪明了一点点儿。

但是还是不够，不够自然，不够完美，工程师们继续努力……

1.1.3 泛型编程完美思路

人们设想能否研制一种编程机制，让一个函数适应所有的数据类型，包括自己定义的。

后来，人们实现了这种机制，既然适应所有的数据类型，就称为泛型编程（Generic Programming）。于是现在 C++也支持这种泛型编程机制。

泛型编程最初提出时的动机很简单直接，就是发明一种语言机制，能够帮助实现：

（1）一个通用的标准容器库。所谓通用的标准容器库，就是要能够做到，比如用一个 List 类存放所有可能类型的对象。

（2）泛型编程让编写完全一般化并可以重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。泛型即是指具有在多种数据类型上皆可操作的含义，这样算法与数据结构完全分离，其中算法是泛型的，不与任何特定数据结构或对象类型联系在一起。

泛型编程的代表作品 STL 是一种高效、泛型、可交互操作的软件组件。STL 以迭代器（iterators）和容器（containers）为基础，是一种泛型算法（generic algorithms）库，容器的存在使这些算法有东西可以操作。STL 包含各种泛型算法（algorithms）、泛型迭代器（iterators）、泛型容器（containers）

以及函数对象（function objects）。STL 并非只是一些有用组件的集合，它是描述软件组件抽象需求条件的一个正规而有条理的架构。

泛型的第一个好处是编译时的严格类型检查。这是集合框架最重要的一个特点。此外，泛型消除了绝大多数的类型转换。如果没有泛型，当使用集合框架时，不得不进行类型转换。

如果对上面的话感觉有些云里雾里，可以后面再研究。但关于泛型编程的理解可以用下面一句话概括：它是把数据类型（包括自己定义的 class 类型）作为一种参数传递进来的机制。

1.1.4　C++的模板

我们需要让程序更加智能，C++很需要泛型这种新的编程模式，于是引入了模板这个功能。也就是说在 C++中，引入了关键字 template。

使用模板是为了实现泛型，可以减轻编程的工作量，增强函数的重用性。注意，增加重用性几乎是编程的基本思路，不然也不会发明函数这种功能了。

例如，前面将两个变量相加的函数 add;

如果不是用模板我们需要针对不同的类型写很多个功能相同的函数，例如 int、float 等，而使用模板则只需使用一个函数即可：

```
template<class T>
T add(T m, T n)
{
return m+n;
}
```

调用时可以显示说明类型。例如：add<int>(3, 5)，使用简单的用法和使用普通函数是相同的：add（3, 5）。

当然，还可以自己定义数据类型，甚至自己定义的类，其具体方法请参照本书后面的章节。

1.1.5　C++ STL 的渊源

那么，有了泛型编程思想，有了模板功能，C++程序员们就可以编写出很多通用的针对不同数据类型的算法，其中，STL 脱颖而出，成为 C++标准，只需掌握这个标准，就不用自己费心费力从头开始编写一些算法。这就是所谓的开发基本思想：不要重复发明轮子。

简单地说，STL（Standard Template Library），标准模板库，是一个具有高可用性的、高效的 C++程序库。它被容纳在 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。你刚开始可以理解为 C++内置的函数库，类算法框架库等。

STL 最开始是惠普实验室开发的一系列软件的统称，它是由 Alexander Stepanov、Meng Lee 和 David R Musser 在惠普实验室工作时所开发出来的。因为实在太优秀了，为了避免其他人重复开发，后来被 C++标准组织投票成为 C++标准的一部分。从根本上说，STL 是一些“容器”的集合，这些“容器”有 list、vector、set、map 等，都是我们熟悉的数据结构。STL 也是算法和其他一些组件的集合。这里的“容器”和算法的集合指的是世界上很多聪明人多年的杰作。STL 的目的是标准化组件，这样就不用重新开发，可以使用现成的组件。STL 被内建在 C++的编译系统之内，是 C++的一部分，因此不用额外安装。

1.2 C++基本概念通览

虽然读者大概都熟悉 C++或者 Java，不过还是要简单回顾一下，加深概念。本节主要介绍 C/C++语言常用的一些基本概念，主要包括：命名空间、头文件、面向对象的程序设计、C++中的声明和定义、最简单的 C++程序、指针、函数、文件、程序的编译和链接、程序的启动和终止、异常处理。通过对这些基本概念的学习，实现对这些基本概念的正确使用，从而更加有利于程序员方便、高效地开发代码。

1.2.1 命名空间

命名空间（name space）是指标识符的可见范围或者有效范围。定义命名空间的目的是防止出现名称冲突现象。命名空间是 C++中一项较新的特性。为了将多个程序员（或厂商）开发的代码便捷、高效地组合起来，防止重复的函数名、类名等，命名空间可以将不同的代码封装在有效的范围内。

命名空间需要使用 using 来声明，并且每个命名均需要使用 using。STL（标准模板库）采用命名空间技术来应对大规模软件开发的难题。C++标准库内的所有标识符都被定义在一个名为 std 的命名空间中，因而得以直接使用指定的标志符号：std。例如：

```
std::cout<<2.0<<std::endl;
```

如果在上述语句之前，加以声明：

```
using std::cout;
using std::endl;
```

那么前面的例句可以写为：

```
cout<<2.0<<endl;
```

如果嫌麻烦，需要逐个声明名称，可以直接声明使用 std 空间，例如：

```
using namespace std;
......
cout<<2.0<<endl;
```

显而易见，使用 C++标准库的命名空间 std 之后，可以任意使用标准库中的函数和变量，如例 1-1 所示。

例 1-1

```
#include <iostream>
#include <stdio.h>
using namespace std;
void main()
{
  cout<<"欢迎您开始步入 C++标准模板库!"<<endl;
  getchar();//等待输入任意键，之后程序退出
   return;
}
```

提示：在例 1-1 中使用了命名空间 std。

使用方法：在 main()函数前添加语句如下。

```
using namespace std;
```

之后在 main()函数中，可以直接使用 cout 函数和 endl 函数。语句 using namespace std 可以放在源代码的任意位置，不同的位置表示 std 的不同有效范围。

总结：本节主要讲述了命名空间的概念及其最简单的使用方法。通过例 1-1 可以使读者对命名空间有一个直接的认识。

1.2.2　头文件

在例 1-1 中，语句#include <iostream>中的 iostream 就是被 include 语句包含的头文件。在原来的 C 语言中，头文件都是以 x.h 的形式存在的。而在 C++中，头文件只有名字，没有扩展名。例如，iostream。原有的 C 语言体系中的头文件部分仍然以 x.h 的形式存在，部分头文件被升级到 C++体系中而不复存在，这部分头文件均在原有名字的前面添加字符 c 来加以标志。例如：cmath，ctring。这类头文件在使用时代码如下：

```
#include  <cmath>
#include  <cstring>
```

值得注意的是，在书写头文件名时，一定要认真，不要添加其他字符，例如：空格。有些编译程序会提示无法找到该头文件，但 Visual C++6.0 编译程序可以避免类似的问题。

头文件的功能主要是将原程序片段收集到一起，形成一个提供给编译程序的文件。一般情况下，头文件中只包含各种声明、常量定义、预编译、注释、类型定义、模板定义等。常规的函数定义、数据定义、导出的模板定义等不应该出现在头文件中。

C++标准库头文件是标准库的外在表现形式。使用标准库的唯一途径就是包含相应的头文件。标准库头文件没有后缀（如下面例 1-2 中头文件 cmath 的使用）。

例 1-2

```
#include <iostream>
#include <cmath>
#include <stdio.h>
using namespace std;                                    //使用命名空间 std
void main()
{
 float pi=3.1415916;
 float radius=0;
 float area=0.0;
 float girth=0.0;
 cout<<"请输入圆的半径:"<<endl;
     cin>>radius;
 area=pi*pow(radius,2);                                 //使用数学函数 pow();乘方
 girth=2*pi*radius;
 cout<<"area="<<area<<";  girth="<<girth<<endl;

 getchar();
 getchar();                                             //等待程序退出
 return;
}
```

提示：在例 1-2 中，使用了 C 语言常规函数 pow(double x, douley)，是幂函数。这个函数有两个参数 x 和 y，函数的作用是求解 x 的 y 次方。

使用头文件主要是为了提供类、函数、变量的声明，从而可以方便地使用这些类、函数、变量等。标准库同样提供了大量的头文件，当使用这些头文件时，标准库的强大功能在瞬间向我们敞开了大门。

1.2.3 面向对象的程序设计

本节主要讲解什么是“面向对象的程序设计”。

面向对象是 C/C++语言解决问题的一种方法。原来的程序开发是面向过程的。面向过程的程序设计主要考虑解决问题的先后顺序、措施安排等，具有典型的过程性。而面向对象的程序设计主要是建立在各种对象基础上的软件开发，每一个具有独立特性和相应功能的个体均可以作为一个对象加以处理。

人类社会历史就充满了“抽象”的概念。例如：天、地、道等。在软件开发和程序设计领域，同样存在诸多“抽象”的对象。根据软件开发的需要，每个对象被赋予不同的属性和方法（函数）。当该对象被定义时，同时可以方便地使用该对象的属性和方法。以前我们学习过 C++中类的概念，类的对象一般有构造函数、析构函数等。类的对象可以被复制、被传递、被作为其他类的成员，甚至有些类的成员还可以直接参加数学计算。

面向对象（Object Oriented，OO）是当前计算机界关心的重点，它是 20 世纪 90 年代软件开发方法的主流。在对对象有了简单的认识之后，广大程序员开始对面向对象的程序设计进行各种尝试和研究，历经几十年的发展和变革，才有了今天的面向对象软件的开发技术。而今面向对象的概念和应用已超越了程序设计和软件开发，扩展到了很宽的范围。

在 C++中提出了“类”的概念。大家对类也有一定的认识，它是 C++语言的一种新特性、新类型。C++类是一种将抽象转换为用户自定义类型的 C++方法。该方法将数据表示和操纵数据的方法组合成一个整洁的包。

对于类的实现，一般包括两方面：类的声明；类方法的定义。类的声明是指以数据成员的方式描述数据部分，以成员函数的方式描述公有借口。类方法的定义是指描述如何实现类成员函数。通俗地讲，类的声明提供了类的蓝图，而类的定义则提供了细节。下面举一个最简单的例子。

提示：类 Calculator 作为一个简单的计算器，用于实现一些最简单的数学计算功能，如例 1-3 所示。

例 1-3

```
#include <iostream>                         //包含输入/输出流功能的头文件
#include <cmath>                            //包含数学函数库的头文件
#include <stdio.h>                          //包含标准输入/输出的头文件
using namespace std;                        //使用命名空间 std
class Calculator                            //声明类 Calculator
{
public:                                     //将下述定义为公共成员
    Calculator();                           //构造函数
    ~ Calculator();                         //析构函数
    float x;                                //成员变量 x
    float y;                                //成员变量 y
```

```
    float Adder(float x,float y);                       //加法运算
    float Substration(float x,float y);                 //减法运算
    float Multiplication(float x,float y);              //乘法运算
    float Division(float x,float y);                    //除法运算
    float CPow(float x,float y);                        //乘方运算
    float CSqrt(float x);                               //开方运算
};                                                      //类声明结束
Calculator::Calculator()                                //定义构造函数
{
    x=0;y=0;
}
Calculator::~Calculator()                               //定义析构函数
{

}
float Calculator::Adder(float x,float y)                //定义实现加法运算的成员函数
{
    float He=0;
    He=x+y;
    return He;
}
float Calculator::Substration(float x,float y)          //定义实现减法运算的成员函数
{
    float Cha=0;
    Cha=x-y;
    return Cha;
}
float Calculator::Multiplication(float x,float y) //定义实现乘法运算的成员函数
{
    float Ji=0;
    Ji=x*y;
    return Ji;
}
float Calculator::Division(float x,float y)             //定义实现除法运算的成员函数
{
    float Shang=0;
    if(y==0.0)
        return -1;
    Shang=x/y;
    return Shang;
}
float Calculator::CPow(float x,float y)                 //定义实现乘方运算的成员函数
{
    float ChengFang=0;
    ChengFang=pow(x,y);
    return ChengFang;
}
float Calculator::CSqrt(float x)                        // 定义实现开平方运算的成员函数
{
    float sqrtC=0.0;
    sqrtC=sqrt(x);
    return sqrtC;
```

```
}
void main()                                                 //程序入口，main ( ) 函数
{
    float x=0;
    float y=0;
    cout<<"请输入 x:" ;
     cin>>x;                                                //输入数据 x
    cout<<endl;
    cout<<"请输入 y:" ;
     cin>>y;                                                //输入数据 y
    cout<<endl;
    Calculator my;                                          //定义类的对象
    float He=my.Adder (x,y);
    cout<<"两数之和: "<<He<<endl;
    float Cha=my.Substration(x,y);
    cout<<"两数之差: "<<Cha<<endl;
    float Ji=my.Multiplication(x,y);
    cout<<"两数之积: "<<Ji<<endl;
    float Shang=my.Division(x,y);
    cout<<"两数之商: "<<Shang<<endl;
    float Pow=my.CPow(x,y);
    cout<<"x 的 y 次方: "<<Pow<<endl;
    float KaiFang=my.CSqrt(x);
    cout<<"x 的平方根: "<<KaiFang<<endl;
    cout<<"任意键程序退出";
    getchar();
}
```

例 1-3 的执行效果见图 1-1。

总结：在例 1-3 中，通过声明、定义一个类 Calculator，来说明了如何实现用实用类来进行面向对象的编程。通过定义类，形成一个具有一定功能的通用类型或通用模板；使用这个类的对象时，就可以调用模板中所有的成员函数。面向对象技术要高于面向过程技术。面向对象的程序开发使每个对象具备一定的特殊“能力”，而这些“能力”一般也是通过面向过程的技术实现的。通过本节的学习，主要理解“面向对象”的概念即可。

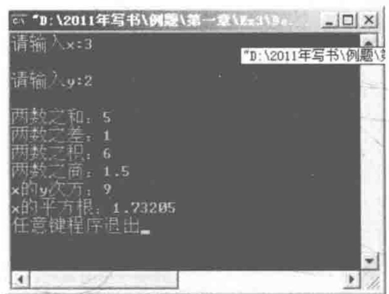


图 1-1　例 1-3 的执行效果

1.2.4　C++中的声明和定义

在程序开发领域，声明是指当一个计算机程序需要调用内存空间时，对内存发出的“占位”指令。

定义是指将声明的变量、函数、类等呈现或描述出来，为其提供一个意义相当的表达，并表明其与众不同之处，最终目的是显示其内涵。

在平时的软件开发过程中，常见的声明和定义语句列举如下（注意中文注释语句）：

例如：

```
int a;                                          //声明一个整型变量 a
float b;                                        //声明一个浮点型变量 b
struct S                                        //声明一个结构 S
```

```
    {
      int a;
      float b;
    };
extern  int  x;                              //声明一个外部变量 x
void  count(bool  YN,  int&  counter);
```

上述语句均是典型的声明语句。

```
void count (bool YN, int& counter )          //函数 count 的定义
  {
     if (YN)                                 //如果 YN 是 true
        {
        counter++;                           //计数器加 1
        }
  }
```

上述代码对函数 count（bool YN, int& counter）进行定义，实现函数的功能代码。

下面通过例 1-4 来验证上述函数的定义。

例 1-4

```
#include  <iostream>
using namespace std;
void  count(bool  YN,  int&  counter);       //函数声明
void count (bool YN,  int&  counter )        //函数定义
    {
      if (YN)
      {
      counter++;
      }
    }
void main()                                  //main()主函数
    {
      int counter=0;                         //计数器
      bool YN=1;                             //逻辑变量，用于当条件满足时，退出 while 循环
      while(YN)                              //
      {
      count(YN,counter);                     //计算循环次数
      cout<<counter<<endl;                   //输出次数至屏幕
      if(counter>5)                          //判断循环次数是否大于 5 次
      {
         YN=0;                               //如果循环次数超过 5 次，将逻辑变量 YN 置 0
      }
      }
    }
```

例 1-4 的执行效果见图 1-2。

图 1-2　例 1-4 的执行效果

通过上述例子可知，在 C++中声明和定义仍然是程序开发所必需的，沿袭了 C 语言的特点。一个名字（标识符）在程序中使用之前必须首先声明。即在使用一个标识符之前，必须刻画清楚其类型，便于编译器识别其类型。在 C++程序中，每个标识符必须有一个定义。通俗地

讲，定义就是给标识符确定了一个值。对于常量，其值是不变的；而对于非常量的标识符，其值是可以被修改的。有时声明和定义可以同时进行，参见下面两行语句。

```
double pi=3.1415026;
char* ch="Here.";
```

下面简单介绍 typedef 的使用。typedef 是为每个类型声明一个新的名字（标识符），而不是一个类型的对象。例如：

```
typedef int INT_32;
typedef short INT_16
typedef unsigned char UCHAR;
```

总结：本节使用通俗易懂的语句和范例，简明扼要地讲述了 C++语言中的声明和定义。这一部分内容是程序开发最基础的部分，希望读者能反复阅读，彻底理解这一部分的含义。

1.2.5 最简单的 C++程序

每个 C++程序必须有一个 main()函数。函数中所有的语句分别实现各自的功能，序列组成 main()函数的主体。例如：

```
#include <iostream>
using namespace std;
void main()
{   cin.get();
    return;
}
```

上述代码被执行，只需按下键盘的任意键，程序即退出运行。下面从全面的角度，尽可能地包括各方面知识，再举一例，目的是为了说明如何使用 C++模板库创建一个最简单的 C++程序。

提示：在例 1-5 中，程序包含两个头文件<iostream>和<list>，使用命名空间 std。<iostream>头文件的使用，使得程序可以使用 C++模板库中的输入流和输出流，具体来说就是可以使用 cin 和 cout 实现屏幕输入和屏幕输出。<list>头文件使程序可以使用模板库中的容器 list。在访问容器中的元素时，使用了 list 模板的迭代器。

读者应尽量读懂这段程序代码，如果觉得有些内容无法理解（例如：迭代器），就先记住这些内容，随着本书章节的延伸，自然就明白了。

例 1-5

```
#include <iostream>
#include <list>
using namespace std;
struct PERSON{                          //定义结构 PERSON
    int id;
    int sex;
    double core;
    void clear()
    {
      id=0;
      sex=0;
      core=0;
    }
```

```
};
void main()
{
    PERSON temp;
    list<PERSON>C1;                //声明一个 list 容器的对象 C1，容器中元素是 PERSON 类型的对象
    int id_temp;
    int sex_temp;
    double core_temp;
    int size=0;
    C1.clear();
    int counter=0;
    cout<<"This is a simplest C++ Example!\n "<<endl;
    cout<<"任意键开始......";
    cin.get();                     //按回车键等待
    while(counter<5)               //输入数据
    {
        cout<<"请输入 ID:";
        cin>>id_temp;
        cout<<"请输入性别:";
        cin>>sex_temp;
        cout<<"请输入分数:";
        cin>>core_temp;
        temp.id=id_temp;
        temp.sex=sex_temp;
        temp.core=core_temp;
        C1.push_back(temp);
        memset(&temp,0,sizeof(PERSON));
        counter++;
    }
    cout<<"回车键继续......";
    cin.get();                                                   //输入等待
    size=C1.size();
    cout<<endl;
    list<PERSON>::iterator Iter;                                 //声明容器迭代器类型变量 Iter
    for(Iter=C1.begin();Iter!=C1.end();Iter++)                   //遍历输出容器中的各元素
    {
        temp.clear();
        temp=*Iter;
        cout<<"ID: "<<temp.id<<", SEX: "<<temp.sex<<",
Core: "<<temp.core<<endl;
    }
    cout<<"任意键退出程序......";//<<endl;
    cin.get();
    return;
}
```

例 1-5 的执行效果见图 1-3。

在例 1-5 中，读者应该重点理解程序的总体架构和模板的使用，不要执着于个别生僻的知识点。在今后的学习中，编程知识会逐渐丰富，慢慢达到信手拈来的程度。上述程序中读者应该重点理解 C++程序中 list 容器的使用方法：如何使用模板 list，如何使用模板

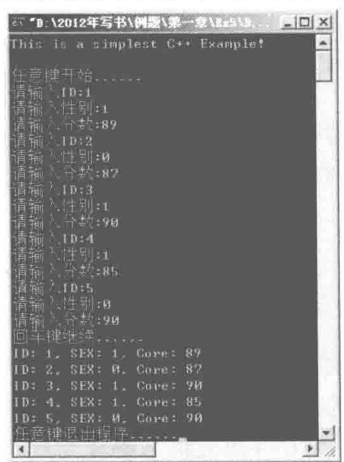


图 1-3　例 1-5 的执行效果

的迭代器，如何遍历和访问容器中的元素等知识。

1.2.6 指针

指针是 C 语言中一个重要的概念，也是 C 语言的重要特色。C++沿袭了 C 语言的这个特色。正确而灵活地使用指针，可以有效地表示复杂的数据结构、动态分配内存、便捷地使用字符串、方便地使用数组、直接处理内存、函数可以返回多个数值等。

在内存中，变量存储在特定的地址，通过地址可以找到所需的变量，变量在内存中的地址“指向”该变量单元。一个变量的地址称为该变量的“指针”。如果有一个变量用来存储另一变量的地址，则称它为“指针变量”。

在学习过程中，要注意区分指针和指针变量。指针指的是该变量在内存中的存储地址；而指针变量指的是该变量的值是一个指向其他变量的指针值。在学习指针的过程中，首先要学习两个运算符：

&：取地址运算符；

*：指针运算符。

下面通过例 1-6 来理解什么是指针，什么是指针变量。

例 1-6

```
#include <iostream>
using namespace std;
void main()
{
    long Aa=11;                                    //给变量 Aa 赋值
    long *B_Pointer=NULL;                          //声明指针变量 B_Pointer,并将其赋
值为空
    B_Pointer=&Aa;                                 //将变量 Aa 的内存地址保存在
B_Pointer 中
    cout<<"Aa 的值: "<<Aa<<"  ,Aa 的内存地址:  "<<B_Pointer<<endl;;          //输出
    cin.get();                                     //等待程序退出
}
```

例 1-6 的执行效果见图 1-4。

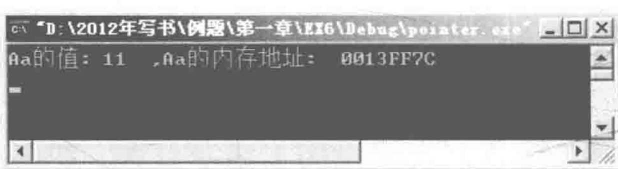


图 1-4 例 1-6 的执行效果

下面继续进行指针的学习。

在 C++语言中，指针和数组基本等价的原因在于指针算术和 C++内部处理数组的方式。指针变量增加 1 或减少 1 之后，增加或减少的量等于指针指向的类型的字节数。例如，指向 double 型变量的指针加 1 之后，数值将增加 8。

C++将数组名解释为数组第 1 个元素的地址。例如：数组 wages[20]存储 20 个 double 型数据，即：

```
double *pw=wages;
```

其实 wages=&wages[0]。现在指针 pw 指向 wages[0]，而*（pw+1）将显示 wages[1]的数值，说明（pw+1）指向数组的第二个元素。

指针可以动态创建，也可以静态创建。静态创建指的是在声明时即给定数组的长度。例如：

```
Int array[10];
```

动态创建指针需要使用 new 操作符，即数组的长度是未知的，在程序运行时为数组分配内存空间。这类指针或数组名必须使用 delete[]指令释放其占用的内存。例如：

```
cin>>size;
int* pi=new int[size];
......
delete [] pi;
```

指针和数组的关系可以扩展到字符串。

提示 1：动态分配内存会导致内存泄漏。如果使用 new（在自由存储空间或堆中）操作符创建变量之后，没有使用 delete 将其释放，会导致内存泄漏。即使包含指针的内存由于作用域规则和对象生命周期的原因而被释放，在自由存储空间上动态分配的变量或结构将继续存在。实际上将会无法访问自由存储空间中的结构，因为指向该内存的指针无效。这部分内存在程序的整个生命周期都不可使用。即这些内存被分配出去，而无法收回。一旦内存泄漏严重，会导致程序的可用内存被耗尽，出现内存耗尽错误，导致程序崩溃。

提示 2：使用指针的目的是希望能直接映射到程序运行所在的机器上的地址机制。部分机器可以对字节寻址，另一些机器可以从机器字中取出字节。

1.2.7　函数

函数是 C/C++程序的最基本组成构件。C/C++的源程序是由函数组成的。

1．函数的概念

读者在学习 C 语言阶段时对函数有了一定程度的理解。本节再复习函数的概念。以利于后面模板库的学习。

C/C++程序至少需要一个 main()函数。程序往往由多个函数组成。函数是 C++源程序的基本模块，通过调用相应的函数，可以实现特定的功能。C/C++语言提供了强大的、极为丰富的库函数。另外，用户或程序开发者可以把自己的思想或算法编写成一个个相对独立的函数模块。

从函数定义的角度，函数可分为库函数和用户定义函数两种。库函数由 C/C++系统提供，使用时只需包含相应的头文件即可直接调用。用户自定义函数是根据用户的需求编写的函数，不仅要在程序中定义函数本身，还要在 main()函数模块中进行类型说明。

函数还可以分为有返回值函数和无返回值函数两种。有返回值函数被调用后将向调用者返回一个执行结果，称为函数返回值。有返回值的函数必须在函数声明和定义中明确函数返回值的类型。无返回值函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。在定义此类型函数时，函数返回值类型为 void。

函数还可以具有参数。从这个角度划分，函数可以分为无参函数和有参函数。无参函数是指函数定义、函数说明及函数调用过程中均不带参数；有参函数是指在函数定义及函数说明时都有参数，一般称为形式参数。函数调用时必须给出参数，称为实际参数。函数调用时，主函数将把实参的值

传送给形参，供被调用的函数使用。

对于函数的声明和定义，举例说明：

```
#define pi=3.1415926
double area(float Radius)                  //函数声明，函数为有返回值函数，并且拥有一个参数
double area(float Radius)                  //函数定义，函数返回值为 double 类型
{
double A=0;
A=pi*Radius*Radius;
return A;
}
```

2. 函数的调用

函数是如何被调用的呢？

在 C 语言中，函数调用的一般形式为：

```
函数名（实际参数表）
```

如果函数没有参数，则调用时不用填写参数表；参数表中的参数可以是常数、变量或其他类型的数据以及表达式，各参数之间用逗号分隔。

需要说明的是：函数定义时填写的参数表是形式参数，简称形参；当调用函数时，填入的参数是实际参数，简称实参。

函数调用时，将实际参数传递给形参表可以有两种形式：赋值调用；引用调用。“赋值调用”形式是将参数的值传递给函数的形参表。此时函数中形参的值发生变化，不会影响函数调用时使用的变量（实际参数）。“引用调用”形式是将实际参数的地址复制给形式参数。当函数调用时，这个地址用来访问所使用的实际参数。这意味着在函数调用过程中，形参的数值变化会直接反映到实际参数的值。下面举例说明这两种调用形式的区别。

提示：例 1-7 中包含两个自定义函数 pow2(float x)和 pow3(float* y)，函数 pow2()是赋值调用形式；函数 pow3()是引用调用形式，但在函数 pow3()中参数 y 并没有发生变化，因此在输出变量 B 时，B 的值没有变化。

例 1-7

```
#include <iostream>
#include <cmath>
using namespace std;
double pow2(float x);                                   //声明
double pow3(float y);                                   //声明
double pow2(float x)                                    //定义
{
    double z=0;
    z=x*x;
    return z;
}
double pow3(float* y)                                   //定义
{
    double z=0;
    z=(*y)*(*y)*(*y);
    return z;
```

```
}
void main()
{
    double A=10;
    float B=20;
    double C=0;
    double D=0;
    C=pow2(A);
    cout<<"A:  "<<A<<"  ; Pow2(A):  "<<C<<endl;
    D=pow3(&B);
    //D=pow3(&A);
    cout<<"B:  "<<B<<"  ; Pow3(B):  "<<D<<endl;
    cin.get();
}
```

例 1-7 的执行效果见图 1-5。

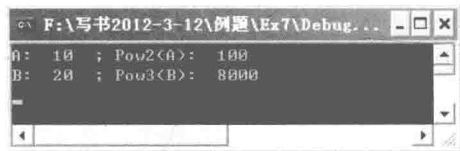


图 1-5　例 1-7 的执行效果

总结：函数调用主要是学习如何使用函数，如何调用函数，理解函数的形参和实参，理解被调用函数的返回值。

3. 函数的递归调用形式

函数可以实现自我调用。如果在函数内部调用了函数自身，则称这个函数为“递归”。例如，计算阶乘的函数代码，就是使用了递归形式。3 的阶乘等于 1×2×3，即 6。一般计算阶乘的代码见例 1-8。

例 1-8

```
#include <iostream>
using namespace std;
int Factor(int n)
{
    int answer=0;
    if(n==1)                                        //递归结束条件
        return (1);
    answer=Factor(n-1)*n;                           //函数的递归调用
    return answer;
}
void main()
{
    int n=5;
    int result=0;
    result=Factor(n);                               //求整数 5 的阶乘
    cout<<"5!= 5x4x3x2x1 = "<<result<<" "<<endl;    //输出计算结果
    cin.get();                                      //等待程序退出
}
```

例 1-8 的执行效果见图 1-6。

总结：函数的递归调用形式实现时最关键的是将“递归结束条件”放在递归调用的前面，并且“递归结束条件”一定要书写正确，否则会带来不必要的麻烦！

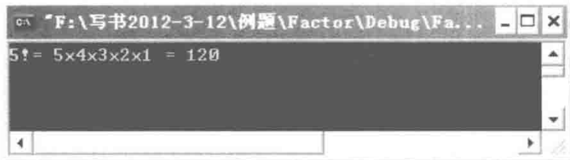


图 1-6　例 1-8 的执行效果

4. 函数的通用性和效率

函数的通用性一般用来指某些通用性比较强的函数，通用函数可以被许多程序员使用，并且通用函数需要的变量和数据必须用函数参数的形式传递。使用参数传递数据，除了有助于函数可用在多种情况下之外，还可以提高函数代码的可读性。

C 语言是基于函数的语言，函数是必不可少的。但函数调用的效率比较低，对于特别简单的功能模块，可以采用内嵌代码的形式。程序对执行速度要求比较高时，可考虑内嵌代码的形式。下面用例 1-9 来加以说明。

例 1-9

```
01  #include <iostream>
02  using namespace std;
03  void main()
04  {
05      int x;
06      for(x=1;x<11;++x)
07          printf("%d ",x*x);
08      printf("\n");
09      cin.get();
10  }
```

例 1-9 中代码语句第 06、07 两行可以单独编写成一个函数，但是函数调用的效率较低，此处体现了内嵌代码的优越性。

5. 函数中的变量

变量以非数字的符号来表达，一般用拉丁字母。变量是常数的相反。变量的用处在于能一般化描述指令的方式。变量通常用一个英文字母表示。

（1）局部变量和全局变量

局部变量也称为内部变量。局部变量是在函数内作定义说明的。其作用域仅限于函数内，离开该函数后再使用这种变量是非法的。

全局变量也称为外部变量，它是在函数外部定义的变量。全局变量不属于哪一个函数，而属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量，一般应全局变量说明。全局变量的说明符为 extern。但在一个函数之前定义的全局变量，在该函数内使用可以不再加以说明。

（2）变量的存储类别

在使用函数时涉及不同的存储方式：静态存储方式；动态存储方式。全局变量使用静态存储方式；函数的形参和普通局部变量使用动态存储方式。

在函数中还可以声明或定义 4 种类型的变量。这 4 种类型分别是：auto，static，register，extern。

- auto 类型是默认类型，可以省略不写。

- static 类型属于静态存储类型，当希望函数中的局部变量的值在函数调用结束后不消失而保留原值时，可用关键字 static 声明该局部变量。
- C 语言允许将局部变量得值放在 CPU 中的寄存器中，这种变量称作“寄存器变量”，用关键字 register 作声明。
- 如果外部变量不在文件的开头定义，其有效的作用范围只限于定义到文件末尾。函数需要引用该外部变量，则应该在引用之前用关键字 extern 对该变量作“外部变量声明”，表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

下面举例说明 4 种变量的存储类别。

首先定义一个文件 my.cpp，在文件中添加以下代码：

```
#pragma once
int A=1;
int B=2;
```

创建例 1-10 的主文件如下：

例 1-10

```
#include <iostream>
#include <cstring>
#include <string>
using namespace std;
extern int A;                                       //外部变量
extern int B;                                       //外部变量
void main()
{
   auto int x=0;                                    //auto 类型整数变量
   register int y=0;                                //register 类型整数变量
   static int count=0;                              //静态存储类型整型变量
   char *ch=new char();
   sprintf(ch,"%d,%d,%d,%d,%d.",x,y,count,A,B);
   cout<<ch<<endl;                                  //输出
}
```

将 my.cpp 文件和 ex10.cpp 放在一起，或者将 my.cpp 文件加入 ex10.cpp 所属工程中。在例 1-10 中包含了 4 种变量的存储类型，虽然例 1-10 中没有给出各种变量的详细说明，但最重要的是给出了外部变量的使用方法。例 1-10 的执行效果见图 1-7。

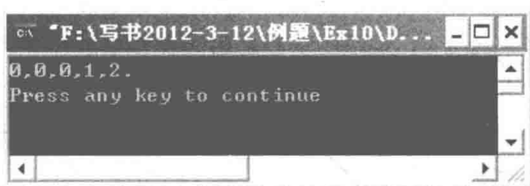


图 1-7　例 1-10 的执行效果

总结：本节要认真体会函数中各种变量的使用方法。

1.2.8　文件

C/C++中的文件是非常重要的概念。所谓“文件”一般是指，存储在外部介质上数据的集合，

并且是一组相关数据的有序集合。当文件被使用时，将其调入内存中。

文件可分为两种：一种是上述所说的，就是驻留在磁盘或其他外部介质上的一个有序数据集，可以是数据文件、可执行程序等；另一种是设备文件，是指和主机相联的各种设备，像显示器、键盘、打印机等；外部设备一般可以看作一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。

在 C 语言中，文件的类型可以分为两种：缓冲型和非缓冲型。文件系统的读写因此也分为两种方法：缓冲文件系统一般用来处理文本文件；非缓冲文件系统可以用来处理二进制文件。C 语言函数库中包含了大量的文件处理函数。当需要使用这批函数时，需要在源程序中包含头文件“stdio.h”。

同样在 C++语言中，关于文件的处理功能更加高级，新封装了部分模板和部分类。常用的类包括：

- ifstream 和 wifstream，用来读取文件；wifstream 是字读取操作。
- ofstream 和 wofstream，用来将数据写入文件；wofstream 是字读取操作。
- fstream 和 wfstream，用于读/写文件；wfstream 是字读取操作。
- filebuf 和 wfilebuf，用来进行实际的字符读写；wfilebuf 是字读/写操作类。

注意：使用上述几个类时，需要包含头文件<fstream>。

下面介绍在 C++ STL（模板库）中如何使用文件操作。

1. 文件标识

在 C 语言中，打开文件时一般需要使用标识符来表明文件的使用方式。同样，在 C++语言中，打开文件时同样需要使用文件标识符（文件模式常量）来说明文件的使用方式。常用的文件模式常量包括：

```
ios_base::in          打开文件，用于读取
ios_base::out         打开文件，用于改写或写入
ios_base::ate         打开文件，用于将文件指针移到文件末尾
ios_base::app         打开文件，写入时始终添加入尾端
ios_base::trunc       打开文件，将先前的内容移除
ios_base::binary      以二进制形式打开文件。
```

2. 文件的读写

本小节主要讲述如何使用 ifstream 类和 ofstream 类。这两个类均包含了一批各自的成员函数。这些成员函数主要用于打开和关闭文件，设置文件缓冲区，读取和写入文件，文件指针定位等。下面举例说明文件的读/写操作。例 1-11 实现的功能是将源文件中的内容读取出来，将这些内容再写入目标文件。

例 1-11

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{
    ifstream f1;
    ofstream f2;
    char filename1[256];
```

```
    char filename2[256];
    char content[256];
    cout<<"请输入文件名(源):";                              //输入源文件名
    cin>>filename1;
    cout<<"请输入文件名(目的):";                            //输入目标文件名
    cin>>filename2;
    f1.open(filename1,ios::in);                             //打开源文件，用于读取
    f2.open(filename2,ios::out);                            //打开目标文件，用于写入
    while(!f1.eof())
    {
        f1.getline(content,128);                            //读取文件中数据
        f2<<content<<endl;                                  //将数据写入目标文件
    }
    f1.close();
    f2.close();
}
```

3. 文件流状态检查

C++文件流类从 ios_base 类那里继承了一个流状态成员。这个流状态成员指出流状态的信息：一切顺利，已达文件尾，I/O 操作失败等。例如：fail()；is_open()；eof()。

4. 使用临时文件

软件开发过程中，经常需要使用临时文件。临时文件的存在是短暂的，必须受程序控制。在 C++中，创建临时文件、复制另一个文件的内容并删除文件其实都很简单。首先，需要为临时文件制订一个命名方案，确保每个文件都被指定为独一无二的文件名。cstdio 库函数中声明的 tmpnam()标准函数可以帮助您。函数原型如下：

```
char* tmpnam(char* pszname)
```

tmpnam()函数创建一个临时文件名（注意：仅仅产生一个文件名），将它放在 pszname 指向的 C 风格字符串中。常量 L_tmpnam 和 TMP_MAX 限制了文件名包含的字符数，以及在确保当前目录中不生成重复文件名的情况下，tmpnam()可以被调用的最多次数。临时文件的使用方法见例 1-12。

提示：在例 1-12 的代码中，头文件 cstio 其实是原 C 语言中的头文件 stdio.h，变量 TMP_MAX 在本例中值为 32 767，表明临时文件名字符长度的变量 L_tmpnam 的值在本例中是 14。文件名是随机产生的，所有的临时文件输出到屏幕上。

例 1-12

```
#include <iostream>
#include <cstdio>
void main()
    {
    using namespace std;
    cout<<"This system can generate up tp "<<TMP_MAX<<" temporary names of up to "<<L_
    tmpnam<<" characters.\n";
    char pszName[L_tmpnam]={'\0'};
    cout<<"Here are ten names: \n";
      for(int i=0;i<10;i++)
      {
        tmpnam(pszName);                                    //生成临时文件名
```

```
        cout<<pszName<<endl;
      }
      return;
    }
```

总结：文件操作在 C 语言中如果已经熟练，再来学习本节是比较轻松的。唯一不同的是，在 C++ 中使用了模板库和常用的 I/O 类。读者可以执行附例中的可执行程序，或者重新编译源文件之后执行，体验一下产生临时文件名的过程。

1.2.9　程序的编译和链接

程序员使用任一编辑软件将编写好的 C++程序输入计算机，并以文本文件的形式保存在计算机的磁盘上。编辑的结果是建立 C++源程序文件。C++程序一般使用小写英文字母，常量和其他用途的符号可以用大写字母。C++编译器对大、小写字母是有区别的。C++语言的关键字必须是小写。

1. 程序编译

程序编译是指将编辑好的源文件翻译成二进制目标代码的过程。编译过程是使用 C 语言提供的编译程序完成的。不同操作系统下的各种编译器的使用命令不完全相同，使用时应注意计算机环境。编译时，编译器首先要对源程序中的每一个语句检查语法错误。当发现错误后，会提示错误的位置和错误类型的信息。要再次调用编辑器进行查错修改，之后再编译，直至排除所有语法和语义错误。正确的源程序文件经过编译后在磁盘上生成目标文件。

C/C++程序需要经过多个步骤才能生成。除了要定义问题需求、设计并在计算机上输入代码外，必须通过计算机生成一个可执行文件。源代码包含指挥计算机运行的 C/C++语句。

编译器是一个计算机程序，读入代码，如果源代码“语法正确”，将把该程序生成机器码。用户将一个源文件提交给编译器后，首先进行的是该文件的预处理，即完成宏处理，并按照#include 指令引进所有头文件。预处理之后的结果被称为被编译单位。这种编译单位才是编译器真正的工作对象，是 C++语言规则所描述的对象。

2. 程序的链接

编译后产生的目标文件是可重定位的程序模块，不能直接运行。

链接程序：是把目标文件和其他分别进行编译生成的目标程序模块及系统提供的标准库函数链接在一起，生成可以运行的可执行文件的过程。

链接过程使用 C++语言提供的链接程序完成，链接器把机器码和函数库代码连接起来，生成一个可执行文件，生成的可执行文件存在磁盘中。

如果程序运行在个人计算机上，该程序必须在个人计算机上进行编译和链接。但符合 ISO C++标准的源代码可以在不同机器上编译和链接，这种情况称为代码可移植。标准化的语言应该是可移植语言。

在所有的编译单位中，对所有函数、类、模板、变量、名字空间、枚举和枚举符的名字的使用都必须保持一致，除非被显式的描述为局部的东西。所有名字空间、类、函数等都应该在它们出现的各个编译单位中有适当的声明，而且声明都应该一致的引用同一个实体。

如果一个名字可以在与其定义所在的编译单位不同的地方使用，即具有外部链接；如果某个名字只能在其定义所在的编译单位内部使用，即被称为具有内部链接。

1.2.10　程序的启动和终止

对于任何一个 C/C++程序，程序执行时总是从 main()函数开始，之后依次或者顺序执行 main()函数中的程序代码，实现其中其他函数的各种功能。

这使人们联想到在生活、工作中，人们总是将“大功能”分解成“小功能”，以便于实现。在 C/C++中，主函数 main()就是所谓的“大功能”。例如：一个“做菜”程序，总的过程就是 main()函数，在主函数中根据情况，调用“买菜”、“切菜”、“烧油”、“放葱花”等一系列步骤（子函数）。一般在可执行程序中，必须要主函数 main()作为程序运行的入口。其余函数均由 main()或其他一般函数调用。

一个 C/C++的源程序经过编译、链接之后，会生成扩展名为 exe 的可执行文件。扩展名为 exe 的程序可以在操作系统下直接运行，即可以由系统启动。对于 main()函数，可以在程序运行时传递参数。

对于非 Windows 程序，程序的终止一般调用 C++函数库的 return 函数，return 函数既可以返回一个数值，也可以不返回数值。程序终止还可以使用 C 函数库中的 exit()函数和 abort()函数。

exit()函数调用时，程序正常退出，退出之前程序会完成一切需要完成的工作。例如，释放分配的内存单元等。abort()函数是“放弃”的意思，函数执行时，程序不做“善后”处理，直接退为运行。

C/C++还提供其他进程控制函数，例如：terminate()，system()等。一般情况下，当程序处理异常时一般调用 terminate()函数；当在程序中需要调用其他命令或者可执行文件时，可选用 system()函数。在第 10 章会详细介绍。

1.2.11　异常处理

后续章节会详细地介绍异常处理。本小节只是简单介绍异常处理的概念。

程序运行时会遇到某种错误，导致程序无法正常运行。C++的异常类为处理这种情况提供了一种功能强大而灵活的工具。异常处理机制是 C++的新功能。异常处理机制可以看作是编译时的类型检查和歧义性控制机制在运行中的对应物。人们应该认识到，错误的处理是一件非常困难的工作。而 C++的异常处理机制为程序员提供了一种处理错误的方式，在给定的系统结构中，以最自然的方式去处理这些错误。异常机制使处理错误的复杂性更加清晰可见。

最初的 C++程序中出现异常时，检测到异常的程序段可以通过产生或抛出异常，使主进程知晓“异常已经发生”。C++异常处理代码是 catch 子句。当异常被 try 中的语句抛出时，系统通过查看 try 块后的 catch 子句列表，来查找能够处理该异常的 catch 子句。

在 C++ STL 中定义了标准库异常类 exception，类的声明包含在头文件<exception>中，但是异常类 exception 不能捕捉所有的异常。exception 异常类只是为人们提供了一种更便捷的异常处理方法而已。

1.3　类模板定义

从本节开始，本书正式开始讲述 C++ STL（模板库）的内容。STL 的一个重要特点是数据结构和算法的分离。这种分离使得 STL 确实非常通用。本节涉及的内容主要包括：模板库、STL 之父、类模板的英文原始定义、类模板实例化、类模板的成员函数、类模板的静态成员。

1.3.1 模板库（STL）

STL 另一个重要特性是它不是面向对象的。STL 主要依赖于模板。这使得 STL 的组件具有广泛通用性的底层特征。由于 STL 是基于模板的，内联函数的使用使得生成的代码短小高效。STL 是 ANSI/ISO C++标准中最新的也是极具革命性的一部分。STL 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。

从实现层次看，整个 STL 是以模板作为基石的。

STL 的背后蕴含着泛型化程序设计（GP）的思想，在这种思想里，大部分基本算法被抽象、被泛化，独立于与之对应的数据结构，用于以相同或相近的方式处理各种不同情形。“泛型化”是根据英语翻译过来的，以笔者的理解，称为“普通”更恰当。本书仍称为泛型。

1.3.2 STL 之父——Alexander Stepanov

Alexander Stepanov，出生于莫斯科，被世人誉为 STL 之父。20 世纪 70 年代，他开始研究将算法从诸多具体应用中抽象出来的可能性。这就是泛型思想的最早雏形。他和纽约州立大学教授 Deepak Kapur，伦塞利尔技术学院教授 David Musser 共同开发了一种称作 Tecton 的语言。Tecton 语言最终没有取得实用性的成果。之后，Alexander Stepanov 又和他人一起，合伙建立了一些大型程序库。当时，面向对象的程序设计思想存在一些问题，例如：抽象数据类型概念的缺陷。Alexander Stepanov 希望通过对软件各部分的分类，逐渐形成软件设计的概念性框架。

大约在 1987 年，Alexander Stepanov 在贝尔实验室工作期间，开始采用 C++语言进行泛型化软件库研究。当时 C++语言还没有模板的概念，现在我们已经认识到模板对于 C++是多么的重要！另外 Alexander Stepanov 开发了庞大的算法库。1988 年 Alexander Stepanov 进入惠普的实验室工作，之后返回到泛型化算法的研究。从最初项目组的 8 人，减少至 2 人，最终信念与汗水终于铸成包含大量数据结构和算法部件的庞大运行库。这便是现在 STL 的雏形，即 HP STL。1993 年 9 月 Alexander Stepanov 为 ANSI/ISO C++标准委员会做了一个相关演讲。1994 年 3 月，Alexander Stepanov 向 ANSI 递交建议书，希望 STL 成为 C++标准库的一部分，遗憾的是由于其思想的新颖性，他被拒绝了。之后，Alexander Stepanov 对原有的 STL 进行了改进，增加了封装内存模式信息的新模块（allocator），使 STL 的大部分功能独立于具体的内存模式，并独立于具体的操作系统平台。1994 年 7 月，在滑铁卢会议上，ANSI 学会最终通过提案，决定将 STL 正式纳入 C++标准化进程之中。STL 终于成为 C++家族的一员。

1998 年，ANSI/ISO C++标准正式确定后，STL 被纳入整个 C++标准库中。

1.3.3 类模板的英文原始定义

在 ISO/IEC 14882:2003(E)中，类模板的英文定义是：“A class template defines the layout and operation for an unbounded set of related types。”较恰当的中文翻译应该是：对于一些非紧密相关的数据类型，类模板定义了它们共同的设计（类的结构）和操作（子函数）等。

1.3.4 类模板实例化

类模板仅仅是模板。如何使用模板就涉及类模板的实例化问题。模板实例化一般指使用模板类和模板参数生成一个类声明的过程。

提示：从上述对模板实例化的概念可知，使用模板和模板参数可以产生一个类的声明。同样，使用其他的模板和相关参数也可以生成一个其他的 C/C++实例。例如，使用函数模板和相关的模板参数生成（实例化）相应的函数。

模板可以使用较短的源代码定义生成出代码，是非常强有力的程序开发方法。使用时要避免大量几乎相同的函数定义胀破存储器。

模板的具体形式是：template <class Type > 。

后面章节会逐渐介绍很多容器类，例如：list，stack，map 等。

在实例化过程中，需要声明一个类型为模板类的对象，即使用所需的具体类型替换通用类型名。例如：

```
list<int> mylist;
stack<int> mystack;
```

至此，终于说明了如何将类模板实例化。

提示：类模板实例化，说白了就是怎样使用类模板生成类，这样才能使用生成的类定义对象。本节没有大篇幅地讲述类模板的抽象知识，而是通过讲道理的形式，对类模板的功能等加以阐述。之后直接给出实例形式，很好地避免了读者开始学习就掉入抽象的模板理论中。

下面用例 1-13 说明类模板的使用方法。

提示：注意代码中的加黑代码。

例 1-13

```
#include <iostream>
#include <list>
#include <ctime>
using namespace std;
void mysleep(int second)                //自己编写的延时函数
{clock_t st;
st=clock();
while(clock()-st<second*CLOCKS_PER_SEC);
}
void main()
{
int count=5;
float number=0.0;
list<int> mylist;                       //对 list 模板类进行实例化，list<int>
cout<<"请任意输入 5 个数字:"<<endl;
while (count--)
{
   cin>>number;
   mylist.push_back(number);            //将输入数据压入列表 list
}
list<int>::iterator iter;
for(iter=mylist.begin();iter!=mylist.end();iter++)    //输出列表 list 的元素
   cout<<*iter<<" ,";
cout<<endl;
return;
}
```

1.3.5 类模板的成员函数

类模板的成员函数可以被类模板实例化产生的类所拥有。每个类模板都拥有自己相应的成员函数，这些函数可以被模板的实例调用。

类模板中成员函数也是用模板实现的，当使用时必须将其实例化。具体形式如下：

（1）必须以关键字 template 开头；

（2）必须指出是哪个类的成员；

（3）类名必须包含模板形参。

例如：

```
template <class T> ret-type Queue::<T>::member-name
```

对于 Queue 类的 destroy()函数，其定义源代码为（摘自《C++Primer 中文版》第 4 版 814 页）：

```
template <class Type> void Queue<Type>::destroy()
{
while(!empty())
   pop();
}
```

对于 vector 的 insert()函数，其定义源代码为（摘自《STL 源码剖析》第 124 页）：

```
template <class T, class Alloc>
void vector<T,Alloc>::insert(iterator position,size_type n,const T&x)
{
......
}
```

提示：在这里主要是明白其中的道理。本书的重点是讲解 STL 的使用，对于 STL 内部深奥的理论和逻辑了解即可。

1.3.6 类模板的静态数据成员

在 C++中，类的成员变量被声明为 static 型，意味着它为该类的所有实例所共享，即当某个类的实例修改了该静态成员变量，其修改值为该类的其他所有实例所见。下面用例 1-14 详细说明静态成员的使用方法。

如果类模板中有成员被声明为 static 型，例如：

```
template <class T>                              //声明下面是一个类模板
class BClass
{
public:
   static int count;                            //静态成员变量
   static long size;                            //静态成员变量
public:
   static long GetSize(){                       //静态成员函数
     return size;
   }
   static int GetCount(){                       //静态成员函数
     return count;
   }
};
```

在上述代码中，BClass 作为类模板的名称，count 和 size 是类模板的成员变量，GetSize()和GetCount()是类模板的成员函数。

如果在程序中需要使用类模板的成员变量，必须在程序中类外部对该 static 静态成员进行定义。例如：

```
template <class T> int BClass<T>::count=0;
template <class T> long BClass<T>::size=2;
```

如果在程序中需要使用类模板的成员函数，必须在程序中使用类模板创建对象，进而使用对象访问该成员函数；或者使用类模板的作用域操作符直接访问成员函数。例如：

```
BClass<int>  myO;
cnt=myO.GetCount();
nsize= myO.GetSize();
```

完整的使用静态成员的示例见例 1-14。

例 1-14

```
#include <iostream>
using namespace std;
template <class T>
class BClass
{
public:
static int count;
static long size;
public:
static long GetSize(){
 return size;
 }
static int GetCount(){
 return count;
 }
};
template <class T> int BClass<T>::count=0;    //在类模板外部定义类模板的静态成员变量
count
template <class T> long BClass<T>::size=2;    //在类模板外部定义类模板的静态成员变量
size
void main()
{
  int cnt;
  long nsize;
  BClass<int>  myO;
  myO.count=2;
  myO.size=3;
  cnt=myO.GetCount();
  nsize= myO.GetSize();
  cout<<cnt<<" ,"<<nsize<<endl;
}
```

例 1-14 的执行效果见图 1-8。

总结：本节主要讲述了如何在类模板中使用静态成员。虽然略有些难度，但经过上述讲解之后，相信读者都会掌握。类模板的静态成员函数可以通过类模板定义对象或作用域操作符的形式调用；而类模板的静态成员变量必须在程序中（类外部）声明。详见随书光盘中源代码。

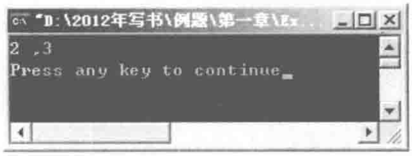


图 1-8　例 1-14 的执行效果

1.4　成员模板

C++模板支持“成员模板”新特性。即模板可以用作结构、类或模板类的成员。在完全实现 STL 设计的过程中，必须使用这项特性。成员模板的定义一般为任意类（可以是类模板，也可以不是类模板）可以拥有类模板或函数模板作为其成员。

在 ISO/IEC 14882:2003(E)中，关于成员模板的说明是：一个模板可以在一个类或类模板中声明，这样的类模板称为成员模板；成员模板的定义既可以在类（或类模板）定义内部，也可以在类（或类模板）定义的外部；当类模板的成员模板在类模板定义的外部定义时，应该完整地指定类模板的参数和成员模板的参数。例如：

```
template <class T>template <class I> void queue::assign(I beg, I end);
```

上述代码中 template <class T>是类模板，作为第一个模板形参表；template<class I>是成员模板，作为第二个模板形参表。

下面举例说明：

```
template <class T>class string                          //类模板 string
{
public:
    template<class T2>int compare(const T2&);
    ......
}
template <class T>template<class T2>int string<T>::compare(const T2& s)
                                                        //在类外部定义成员函数
{
    ......
}
```

上述代码在类模板 string 中，使用了另一个类模板类型的成员函数。

```
template <typename T> class bC
{
public:
    template <typename V> class HC                      //类模板 HC 作为模板 bC 的成员
    {
    public:
    V m;
    V n;
    public:
    HC(){};
    void show(){cout<<m<<endl;};
    }
HC <T> A;
HC<int>B;
public:
```

```
   bC(){};
   template <typename U>U show_m(){};          //类模板定义成员函数
}
```

上述代码举例说明使用类模板作为另一类模板的成员。另外，成员模板具有一些使用规则。

- 成员模板遵循常规访问控制；
- 成员模板不能为虚（virtue）；
- 局部类不能拥有成员模板；
- 析构器不能是模板类型；
- 成员函数模板不能重载基类的虚函数；
- 成员模板具有复杂的转换功能。

总结：本节的重点是对成员模板的理解。最简单的理解即在类模板中包含类模板，被包含的类模板作为其成员。

1.5　友元模板

友元机制允许一个类将对其非公有制成员的访问权授予指定的函数或类。一般友元声明以关键字 friend 开始。友元只能出现在类定义的内部。而友元的声明可以出现在类中的任何地方。一般友元声明被成批地放在类定义的开始或结尾。最简单的使用友元机制的例子如下：

```
class Screen{
       friend class Windows;
       ......
}
```

在上例中，类 Windows 被声明为类 Screen 的友元类，之后在类 Windows 中可以使用类 Screen 的非公有资源，包括其私有成员。友元可以是普通非成员函数，其他类的成员函数或整个类。

在 ISO/IEC 14882:2003(E)中，类或类模板的友元可以是函数模板或者类模板。友元模板可以在类（或类模板）内部被声明。友元函数可以在类（或类模板）中定义，而友元类模板不可以在类（或类模板）中被定义。

类模板中友元定义主要分为以下 3 类：

（1）非模板函数、类作为实例类的友元；

（2）模板函数、模板类作为同类型实例类的友元；

（3）模板函数、类作为不同类型实例类的友元。

下面分别举例说明。

（1）非模板函数、类作为实例类的友元。

举例如下：

```
class A{
void AF();
};
template <class Type>
class Queue
{
friend class B;                          //类 B 不需要事先声明
friend void A();                         //函数 A()
```

```
friend void A::AF();                                    //类 A 必须事先声明
}
```

（2）模板函数、模板类作为同类型实例类的友元。

举例如下：

```
template <class Type> class  A{…};
template <class Type> void D()(B <Type>);
template <class Type> class C{ void Cf();…};
template <class Type>class B
{
friend class A<Type>;                          //模板类 A 需要先定义或声明
friend void D(B <Type>);                       //模板函数 A()需要先定义或声明
friend void C<Type>::Cf();                     //模板类 C 必须先定义
};
```

（3）模板函数、类作为不同类型实例类的友元。

举例如下：

```
template <class T>class B
{
    friend<class Type>friend class A;
    template <class Type>friend void D(B <Type>);
    template <class Type>friend void C::Cf();
}
```

1.6 函数模板

对于学习和使用 STL 的程序员或者研究人员，可能更需要快速地掌握和使用 STL 提供的各种容器。但人们很快发现，函数模板的作用极其强大。

函数模板定义一个无限的相关函数集合。当函数除了数据类型不一致外，其余的处理全部相同，此时函数模板随之诞生。这也是创建函数模板的原因。

函数模板可以定义参数化的非成员函数，使程序员能够用不同类型的参数调用相同的函数，由编译器决定该采用哪种类型，从模板函数中生成相应的代码。

提示： 在例 1-15 的源代码中，template <typename T>void print(const T& var)是函数的定义，其中 T 代表数据类型或者类，函数 print 的参数是通用数据类型 T。

例 1-15

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>void print(const T& var) //也可以使用 template<class T>声明函数模板
{
    cout<<var<<endl;
}
void main()
{
    string str("Hello Beijing!");
    print(str);
    cin.get();
}
```

例 1-15 的执行效果见图 1-9。

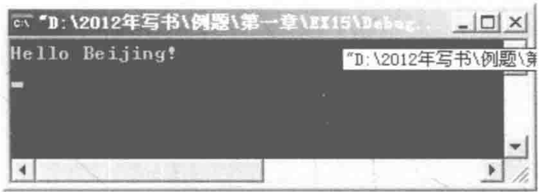


图 1-9　例 1-15 的执行效果

提示： 使用模板函数时，可以使用例题中的 template <typename T> 定义模板函数，也可以使用 template <class T>，两者没有区别。目前程序员多用后者，主要是关键字 typename 的诞生时间很短，多数人还不知道或者不习惯使用。

例 1-16 介绍了函数模板的使用方法。如果函数模板需要定义多个参数，可以对它们随意命名。但每个参数必须以关键词开头，彼此之间使用逗号分隔。这样，函数模板的便捷性取代了多态函数的繁杂性，意义深远。

例 1-16

```
#include <iostream>
#include <string>
using namespace std;
template <class T> void print(T& ii, T& jj)                    //函数模板
{  cout<<ii<<endl;
   cout<<jj<<endl;
}
void main()
{  string strA("Hello  Beijing!");
  string strB("I am a programmer!");
  print(strA,strB);
  cin.get();
}
```

例 1-16 的执行效果见图 1-10。

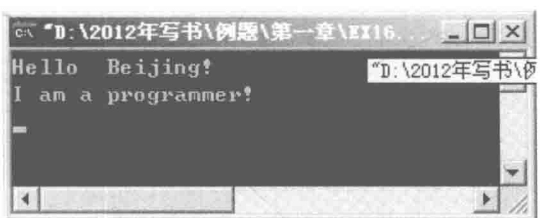


图 1-10　例 1-16 的执行效果

在例 1-16 中，模板函数的两个参数是同一类型，还可以使用多种类型的多个参数，见例 1-17。

提示： 请注意下面源代码中的黑体字。

例 1-17

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

```
template <class T1,class T2> void print(T1&  ii, T2&  jj)          //模板函数
{
  cout<<ii<<endl;
  cout<<jj<<endl;
}
void main()
{
  string strA("Hello  Beijing!");                                  //定义 string
对象
  int B=50;
  print(strA,  B);                                                 //输出
  cin.get();
}
```

例 1-17 的执行效果见图 1-11。

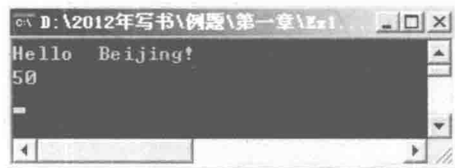


图 1-11　例 1-17 的执行效果

总结：使用模板函数实现多个参数时，函数模板的参数既可以是同类型数据，也可以是不同类型数据。参见例 1-16 和例 1-17。

函数模板功能极其强大，但也存在个别情况。当待比较的函数模板没有提供正确的操作符或者方法，程序不会对此进行编译。为了避免此类错误，可以使用函数模板与同名的非模板函数重载，即函数的定制。本书不再赘述。

下面举一个函数模板的实例（例 1-18），供读者参考。例 1-18 实现的功能是将任何数据类型转换为字符串，还可以将字符串转换为其他数据类型。

（1）创建模板函数 toString()和 fromString()，并将其放在头文件 Ex18.h 中。

例 1-18

```
#ifndef Ex18_H
#define Ex18_H
#include <iostream>
#include <string>
#include <sstream>
template <typename T> T fromString(const std::string& s)   //模板函数，将字符串转化成
                                                            其他数据类型
{
    std::istringstream  is(s) ;
    T t;
    is>>t;
    return t;
}
template <typename T>std::string  toString(const T& t)     //模板函数，将其他数据类型
                                                             转换成字符串
{
    std::ostringstream s;
    s<<t;
    return s.str();
}
#endif
```

（2）创建主程序 Ex18.cpp。

```
#include "Ex18.h"
#include <iostream>
#include <complex>
using namespace std;
void main()
{    int i=1234;
     float j=567.34;
     complex<float> c(2.5,4.1);
     cout<<"i==\""<<toString(i)<<"\""<<endl;          //将数字转换为字符串，并输出
     cout<<"i==\""<<toString(j)<<"\""<<endl;          //将数字转换为字符串，并输出
     cout<<"i==\""<<toString(c)<<"\""<<endl;          //将数字转换为字符串，并输出
     //......
     i=fromString<int>(string("1234"));               //将字符串转换为数字
     j=fromString<float>(string("567.34"));           //将字符串转换为数字
     c=fromString< complex<float> >(string("(2.5,4.1)"));  //将字符串转换为数字
     cout<<"i==\""<<i<<"\""<<endl;
     cout<<"i==\""<<j<<"\""<<endl;
     cout<<"i==\""<<c<<"\""<<endl;
}
```

（3）例 1-18 的执行效果见图 1-12。

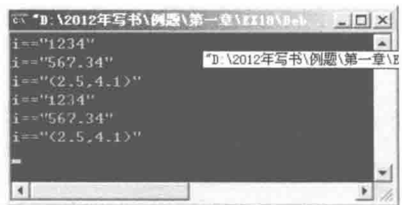


图 1-12　例 1-18 的执行效果

总结：本节主要讲述了函数模板的一些特性和使用方法。通过 4 个例题，详细演示了函数模板的使用。希望读者在阅读文字内容的同时，认真阅读本节的 4 个例题。

1.7　类模板的参数

在前面几节的学习过程中，已经涉及了模板的参数。本小节将较为详细地对模板的参数进行阐述。

模板的参数可以是类型参数，也可以有常规类型的参数。并且一个模板可以有多个参数。

如果是类型参数，类型参数可以采用 typename 或 class 关键字指明。

```
template <class T1, class T2, class T3> class classname;
template<typename T1, typename T2, typename T3> class classname;
```

如果是常规类型参数，即通常的参数定义。

```
template<class Type, int size> class Queue;
```

对于上例，实例化时可以采用如下形式：

```
Queue<int , 100> iq;
```

如果在声明类模板时，已经明确了模板参数：

```
template <class Type=int , int size=1024> class Queue;
```

在实例化时可采用以下几种形式：

```
Queue<int , 100> iq1;
Queue<int> iq2;
Queue<>iq3;
```

整数参数可以用来提供大小或界限。

1.7.1 关键字 typename 的使用

typename 关键字告诉编译器其后的名称为一个类型，可以用其来创建实例。例如：

```
template <class T> class Y
{
    typedef  typename T::A TA;
    TA*  abc;
    ……
}
```

在上例中，如果没有 typename 关键字，编译程序则无法理解 T::A 是什么。

由以上可知，typename 关键字主要用于解决由类模板机制带来的语义歧义，使模板声明更加直观。

类模板及其成员函数使用 tyepname 关键字限定所有对参数化类型中定义类型的引用。编译器看到 typename 关键字，就知道其后紧跟的是数据类型，在分析代码时假定类型的名字通过实例来填充。标准 C++要求，如果类型在模板化的类中定义，在模板中引用该类型时，都要使用 typename 限定符。下面用例 1-19 来说明 typename 的用法。

例 1-19

```
#include <iostream>
template <class T> class Table{                          //模板类 Table
    typename T::iter t;                                  //typename 关键字说明 T 代表类型
public:
    explicit Table(const typename T::iter& ti):t(ti)    //构造器
    {
    }
    void show()                                          //输出
    {
        typename T::iter* y;
        y=&t;
        std::cout<<(*y)->p<<std::endl;
    }
};
class card{
public:
    typedef card* iter;
    int p;
    card(int pos):p(pos)                                 //构造器
    { }
};
void main()
{
    card pc(3);
    Table<card>S_Table(&pc);                             //使用类 card 作为类模板 Table 的参数
    S_Table.show();                                      //输出
    return;
}
```

例 1-19 的执行效果见图 1-13。

总结：请读者认真研究例 1-19，体会其中的意思。

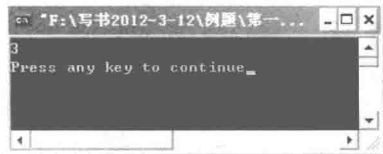


图 1-13　例 1-19 的执行效果

1.7.2　关键字 typename 与关键字 class

标准 C++委员会把 typename 关键字引入标准 C++中，将其应用于使用 class 来声明参数化类型的模板中。在引入 typename 之前，模板实现早已在模板声明中使用 class 关键字，例如：

```
template <class T1, class T2>
```

在引入关键字 typename 之后，标准 C++允许在模板声明中用 typename 关键字替代 class，例如：

```
template <typename T1, typename T2>                          //
```

typename 的使用见例 1-20。

例 1-20

```
#include <iostream>
using namespace std;
template <typename T>T doubleV(T Val)                     //模板函数 double(T Val)
{
   return Val*2;
};
template<typename T1, typename T2>class TwoThing{         //模板类
   T1 th1;
   T2 th2;
   T1 th3;
public:
   TwoThing(T1 tt1,T2 tt2)
   {
      th1=tt1;
      th2=tt2;
      th3=doubleV<T1>(tt1);
   }
   void show()                                            //输出
   {
      cout<<th1<<" , "<<th2<<" , "<<th3<<endl;
   }
};
void main()
{
   int m=123;
   double n=456.789;
   TwoThing<int, double> my(m,n);                         //定义模板对象
   my.show();
   return;
}
```

例 1-20 的执行效果见图 1-14。

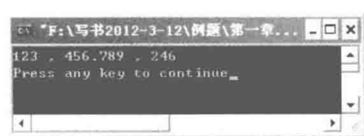


图 1-14　例 1-20 的执行效果

总结：本节主要讲述了模板参数。开始部分讲述了模板参数的概念和意义；之后分两个问题详细讲述了关键字 typename。本节内容是重点学习内容。

1.8 模板库简介

STL 的代码从广义上讲分为 3 类：algorithm（算法）、container（容器）和 iterator（迭代器），几乎所有的代码都采用了模板类和模板函数的方式。在 C++标准中，STL 被组织为 13 个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。本小节主要介绍模板库（STL）的演变历史、组成构件、基本结构以及 STL 编程概述。

1.8.1 C 语言和 STL 的演变历史

C 语言是 1972 年由美国的 Dennis Ritchie 设计发明的，并首次在 UNIX 操作系统的计算机上使用。C 语言由早期的汇编语言 BCPL 发展演变而来。1970 年，在 AT&T 贝尔实验室的 Ken Thompson 根据 BCPL 语言设计出较先进的语言并取名为 B 语言，最后导致 C 语言的问世。随着微型计算机的日益普及，出现了很多 C 语言版本。由于没有统一的标准，使 C 语言之间出现了不一致之处。美国国家 ANSI 为 C 语言制定一套 ANSI 标准，成为现行的 C 语言标准。

早期的 C 语言主要用于 UNIX 系统。由于 C 语言的强大功能和各方面的优点，它逐渐被人们认识，到 20 世纪 80 年代 C 语言开始进入其他操作系统，并很快在各种计算机上得到广泛应用，成为当代最优秀的程序设计语言之一。

C 语言的表现能力和处理能力极强。它不仅具有丰富的运算符和数据类型，便于实现各类复杂的数据结构。还可以直接访问内存物理地址，甚至进行位操作。C 语言可以实现对硬件的编程操作，便捷方便。

1983 年贝尔实验室的 Bjarne Strou-strup 推出了 C++。C++进一步扩充和完善了 C 语言，使其成为面向对象的程序设计语言。最初 C++主要用于小型计算机系统，直至 1988 年出现了第一个用于 PC 的 ZORTECH C++2.0 编译系统，1989 年出现了 TURBO C++2.0 编译器。之后 Borland 公司从 1991 年陆续推出了 Borland C++2.0/3.0/4.0 系统，而微软公司直到 1992 年，才推出基于 DOS 的 MS C/C++7.0 系统。1993 年微软推出了面向 Windows 的 Visual C++1.0 系统，直至 1998 年才推出 Visual C++6.0。

C 语言提供了具有可适应性的、强大的抽象机制，用于对问题进行抽象；这种语言结构允许程序员创建和使用新的类型，这些新的类型则可以与实际应用中所包含的概念相适应。在 C++的最新发展过程中，C++新增模板新特性。通过使用模板，程序具备更好的代码重用性能。1994 年 7 月，美国国家标准研究所通过投票决定，将 STL 纳入 C++标准，使之成为 C++库的重要组成部分。1997 年 C++标准完成了最近一次的修改，官方名称为 ISO14882。

STL 译为标准模板库。STL 从根本上讲是“容器”的集合，也是组件的集合。容器包括：list、vector、set、map 等；组件包括：迭代器等。STL 的目的是标准化组件，和 Visual C++中的 ATL 相似。STL 是 C++的一部分，不用额外安装，被内建在支持 C++的编译器中。STL 的算法是标准算法，可以把 STL 已经定义好的算法应用在容器的对象上。

1.8.2 STL 的组件

STL 是 C++标准程序库的核心。STL 内所有组件都由模板构成，其元素可以是任意型别。程序员通过选用恰当的群集类别，调用其成员函数和算法数据，就万事大吉了。代价是 STL 并不好懂。

本节将介绍 STL 的各种组件。组件中最主要的是容器、迭代器、算法和仿函数。

容器，用来管理某类对象的集合。鱼缸是用来盛放金鱼的容器，见图 1-15。每一种容器都有其优点和缺点。为应对程序的各种需求，STL 准备了很多容器类型。容器可以是 arrays 或是 linked lists，或者每个元素有特别的键值。

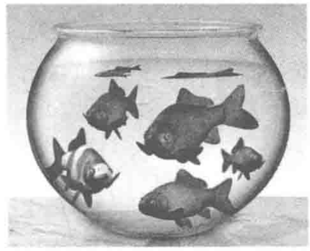


图 1-15　鱼缸

迭代器，用来在一个对象群集的元素上进行遍历动作。对象群集可能是容器，也可能是容器的一部分。迭代器的主要用途是为容器提供一组很小的公共接口。利用这个接口，某项操作可以行进至群集内的下一个元素。每种容器都提供了各自的迭代器，迭代器了解该容器的内部结构，所以能够正确行进。迭代器的接口和一般指针差不多。

算法，用来处理群集内的元素。可以出于不同目的搜寻、排序、修改、使用那些元素。所有容器的迭代器都提供一致的接口，通过迭代器的协助，算法程序可以用于任意容器。

注意：STL 的基本观念是将数据和操作分离。数据由容器类别加以管理，操作则由可定制的算法定义。迭代器在两者之间充当黏合剂，算法可以和容器交互运作。

STL 的另一个特性是组件可以针对任意型别运作。“标准模板库”的名字就表示“可接受任意型别”的模板，这些型别均可执行必要操作。

在 STL 中容器又包括各种类型：序列式容器，关联式容器。而迭代器的功能主要是遍历容器内全部或部分元素的对象。迭代器可以划分为 5 种类属，归属两种类型：双向迭代器、随机存取迭代器。STL 中提供的算法包括：搜寻、排序、复制、重新排序、修改、数值运算等最基本、最普遍的算法。在 STL 中，大量运用仿函数。仿函数具有泛型编程强大的威力和纯粹抽象概念的特征。

1.8.3　STL 的基本结构

本节详细介绍 STL 的基本结构。

STL 是 C++通用库，由容器、算法、迭代器、仿函数和内存配置器组成。这 5 部分的关系见图 1-16。

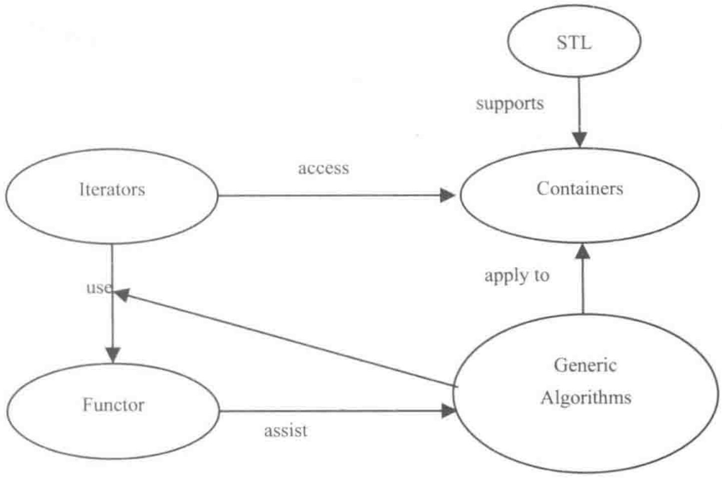


图 1-16　STL 结构图

1. 容器

STL 主要包含容器类。容器类是可以包含其他对象的类，就像数组和队列堆栈等数据结构包含整数、小数、类等数据成员一样。STL 可以包含常见的向量类、链表类、双向队列类、集合类和图类等。每个类都是一种模板，这些模板可以包含各种类型的对象。下述代码是常用的 vector 赋值方法。

```
vector<int> l;
for(int i=0;i<100;i++)
    l.push_back(i);
```

下例采用 map 容器进行二维元素的管理。

```
map<string, string,less<string>>cap;          //按从小到大排序
cap["Ark"]="Little Rock";
cap["New"]="Albany";
```

map 类似二维数组，但比二维数组灵活得多。

目前，STL 中已经提供的容器如表 1-1 所示。

表 1-1　STL 中提供的容器

名　称	说　明
vector<T>	是一种向量
list<T>	是一个双向链表容器；完成了标准 C++数据结构中链表的所有功能
queue<T>	是一种对列容器，完成了标准 C++数据结构中队列的所有功能
stack<T>	是一种栈容器，完成了标准 C++数据结构中栈的所有功能
deque<T>	是双端对列容器，完成了标准 C++数据结构中栈的所有功能
priority_queue<T>	是一种按值排序的对列容器
set<T>	是一种集合容器
multiset<T>	是一种允许出现重复元素的集合容器
map<key,val>	是一种关联数组容器
multimap<key, val>	是一种允许出现重复 key 值的关联数组容器

以上容器设计高效，还提供了接口。程序员可以在任何适当的地方使用它们。

容器可以分为两大类：序列式容器和关联式容器。序列式容器主要有 vector、list 和 deque；关联式容器包括 set、map、multiset 和 multimap 等容器模板类。

2. 算法

STL 提供了非常多的数据结构算法。这些算法在命名空间 std 的范围内定义，通过包含头文件<algorithm>来获得使用权。

常见的部分算法如下：

```
for_each()
find()
find_if()
count()
count_if()
replace()
replace_if()
copy()
unique_copy()
sort()
equal_range()
merge()
```

STL 中所有的算法都是基于模板实现的。

3. 迭代器

通俗地讲，迭代器就是指示器。迭代器技术能够使程序反复地对 STL 容器的内容进行访问，非常快捷和重要。反复的访问意味着一次可以访问一个或多个元素。迭代器为访问容器提供了通用的方法，类似于 C++的指针。当参数化类型是 C++内部类型时，迭代器即 C++指针。STL 定义了 5 种类型的指示器，根据使用方法而命名。

每种容器都支持某种类别的迭代器，常见的迭代器类别包括：输入、输出、前向、双向和随机访问。下面详细介绍：

（1）输入迭代器主要用于为程序中需要的数据源提供输入接口，数据源一般指容器、数据流等。输入迭代器只能够从一个序列中读取数值，迭代器可以被修改、被引用等。

（2）输出迭代器主要用于输出程序中已经得到的数据结果（容器，数据流）。输出迭代器只能够向一个序列写入数据，输出迭代器也可以被修改和被引用。

（3）双向迭代器既可以用来读又可以用来写，双向迭代器与前向迭代器相类似。双向迭代器可以同时进行前向和后向元素操作。所有的 STL 容器都提供了双向迭代器功能，既有利于数据的写入和读出，还有利于提供更加灵活的数据操作。有的容器甚至提供了随机接入迭代器。

（4）随机接入迭代器可以通过跳跃的方式访问容器中的任意数据，使数据的访问非常灵活。随机访问迭代器具有双向迭代器的所有功能，是功能最强大的迭代器类型。

迭代器的诞生使算法和容器分离成为可能。算法是模板，其类型依赖于迭代器，不会局限于单一容器。不同的 STL 算法需要不同类型的迭代器来实现相应的功能。因为不同类型的 STL 容器支持不同类型的迭代器，所以不能对所有容器使用相同的算法。

4. 仿函数

理解了算法、迭代器、容器之后，已经了解了 STL 的大部分内容。STL 还有两个最重要的内容：仿函数和内存配置器。本小节简单介绍仿函数，下一小节介绍内存分配器。

STL 包含大量仿函数。仿函数可以理解为函数的一般形式。仿函数有几种不同的约束，对于编程来说，仿函数非常重要。在标准 C++规范中，函数调用一般使用指针，当需要调用函数时，只需告诉函数的地址即可。例如：

```
static int cmp(int* I, int *j){return (*i-*j);}
```

上一行代码定义了一个函数，当需要调用此函数时，只需提供函数的地址即可。例如：

```
qsort(a,10,sizeof(int), cmp);
```

此方法的最大缺陷是效率低，为了提高效率，STL 定义了仿函数概念。下例定义了一个仿函数，其特征就是使用 operator 来定义。例如：

```
struct three_mul
{
   bool operator()(int& v)
   {
   return (v%3= =0)
   }
}
```

上面代码定义了一个仿函数。通过运算符定义能显著提高效率。例如：

```
for_each(myvector.begin(),myvector.end(),three_mul);   //
```

5. 内存配置器

STL 包括底层的内存分配和释放。Allocators 非常少见，在此可以忽略，在后面章节将专门介绍。配接器可以实现不同类之间的数据转换。最常用的配接器有 istream_iterator，提供了函数 copy 的接口。Adapter 对于 STL 技术来说非常重要。STL 提供 3 种容器配接器：stack<Container>，queue<Container>和 and deque<Container>，具体应用见例 1-21。

例 1-21

```
#include <iostream>
#include <stack>
using namespace std;
void main()
{
  stack<int> st;                                  //定义堆栈对象
  for(int i=0;i<10;i++)
      st.push(i);                                 //将数据压入堆栈
  while(!st.empty())
  {
      cout<<st.top()<<" ";                        //弹出堆栈的第一个元素，并输出
      st.pop();                                   //弹出堆栈元素
  }
  cout<<endl;
  cin.get();                                      //任意键退出
}
```

例 1-21 的执行效果见图 1-17。

图 1-17　例 1-21 的执行效果

1.8.4 STL 的编程概述

STL 作为 C++通用库，主要由迭代器、算法、容器、仿函数、内存分配器和配接器 6 部分组成。

程序员使用 STL 能够容易地实现许多种标准类型的容器，并且便捷地操作容器。对于编程人员来说，标准化组件意味着直接使用现成的组件，不用重复开发。使用 STL 最重要的是掌握基本理论和编程方法，了解 STL 编程技术，必须深刻掌握 STL 容器技术和 STL 迭代器技术。

STL 提供了一组表示容器、迭代器、仿函数和算法的模板。容器是类似数组的单元，可存储若干个值。

STL 容器是同质的，即存储的值类型相同；算法是完成特定任务的处方，迭代器能够用来遍历容器的对象，与能够遍历数组的指针类似，是广义指针；仿函数是类似函数的对象，可以是类对象或函数指针。STL 可以构造各种容器和执行各种算法操作。

下面以矢量为例，简要讲述矢量模板的使用。在数学计算和 STL 模板中，vector 对应数组，提

供与 valarray 和 ArrayTP 类似操作。而 STL 为使 vector 类成为通用的，在头文件<vector>中定义了 vector 模板。创建 vector 模板对象，使用通常的<type>表示法指出要使用的类型。并且 vector 模板使用动态内存，可以使用初始化参数决定矢量的数量。例如：

```
#include <vector>
using namespace std;                                    //使用命名空间 std;
vector <int > ratings(5);                               //定义矢量对象
int n;
cin>>n;                                                 // 输入矢量大小
vector <double> scores(n);                              //定义矢量动态内存
```

分配器是用来管理对象内存的。在 STL 容器模板中，一般都有一个可选的模板参数。例如：

```
template <class T, class Allocator = allocator<T>>      //矢量模板
class vector
{
   … …
}
```

如果省略该模板参数的值，则容器模板将默认使用 allocator<T>类。allocator 类以标准形式使用 new 和 delete 内存管理方式。例 1-22 中创建了两个 vector 对象，一个是 int 规范，另一个是 string 规范。

提示：请关注例 1-22 代码中的加黑代码。

例 1-22

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
const int NUM=5;
void main()
{ vector<string>names(NUM);                             //定义矢量对象
  vector<int>sexs(NUM);                                 //同上
  cout<<"Please Do Exactly As Told. You Will enter\n"<<NUM<<" Personal Name and Their
Sex.\n";
  int i=0;
  for(i=0;i<NUM;i++)                                    //输入信息
  {    cout<<"Enter title # " <<i+1<<": ";
       getline(cin,names[i]);                           //获取输入信息
       cout<<"Enter sex(0/1) # ";
       cin>>sexs[i];                                    //获取输入信息
       cin.get();                                       //等待
  }
  cout<<"Thank you. You entered the following: \n"<<"name/sex"<<endl;
  for(i=0;i<NUM;i++)                                    //输出信息
  {    cout<<names[i]<<"\t"<<sexs[i]<<endl;
  }
  return;
}
```

例 1-22 的执行效果见图 1-18。

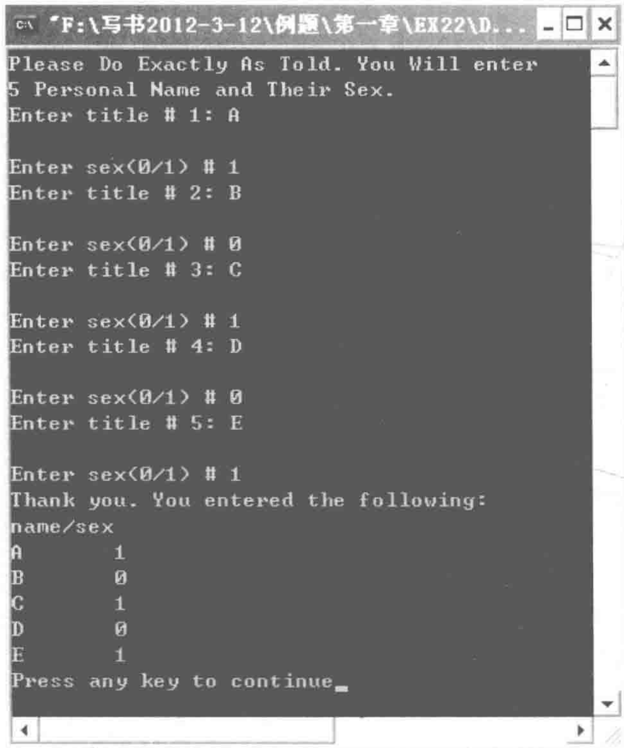


图 1-18　例 1-22 的执行效果

总结：本节简要介绍了 STL 模板的一些特性。并以 vector 矢量为例，简要讲述其使用方法。

1.8.5　学习重点提示

本节主要对 STL 模板库进行了简介。主要介绍了 C 语言和 STL 的演变历史、STL 的组件、STL 的基本结构、STL 的编程概述，共四部分内容。

读者应该重点学习“STL 的组件”、“STL 的基本结构”两部分内容。“STL 的编程概述”部分读者认真学习即可。大量的关于编程的内容在后面章节会逐渐深入介绍。

1.9　本章小结

本章主要介绍了 C++和 STL 模板库的基本概念，介绍了类模板定义、成员模板、友元模板、函数模板和类模板的参数等内容。在 1.7 节详细介绍了模板库，学习了模板库的历史、模板库的组件、模板库的基本结构以及模板库编程概述等内容。

第 2 章　C++中的字符串

本章讲述 C++标准程序库中的字符串处理类 string。在最初的 C 语言中，头文件 string.h 提供了一系列字符串函数，早期的 C++为处理字符串提供了各自的类。string 类由头文件<string>支持，该类包含了大量方法，并且包括了若干构造函数，用于字符串赋给变量、合并字符串、比较字符串和访问各个元素的重载操作符、查找字符和子字符串的方法等。到目前为止，string 已经得到广泛理解并被大家接受及使用。

C++从 C 继承的字符串概念仍然是以'\0'为结束符的 char 数组。C++标准库中的 string class 可以将 string 作为一个型别，可以实现复制、赋值和比较，不必担心内存大小及占用内存实际长度等具体问题。现今世界，数据处理大部分是字符串处理，对于早期的 C 语言和类 FORTRAN 语言的开发语言来说，是非常重要的进步。在早期的开发语言中，字符串的处理是非常复杂的。

本章将详细讲述字符串库类简述、字符的特点、字符串类（basic_string）、字符串通用函数、字符串联接、字符串 IO 操作、搜索和查找、字符串对迭代器的支持、字符串对配置器的支持等内容。

2.1　字符串库简述

本节主要介绍字符串类和智能指针 auto_ptr。字符串的表现形式多种多样，像 TCHAR，std::string，BSTR 等。字符串类均起源于 C 语言的字符串，而最初 C 语言的字符串是字符的数组。单字节字符串顺序存放各个字符串，并用'\0'字节表示字符串结束。在 C 语言中，已存在部分字符串处理函数，例如：strcpy()，sprintf()，stoi()等，只能用于单字节字符串。在标准库中，还有仅用于 Unicode 字符串的函数，例如：wcscpy()，swprintf()，_wtol()等。

多数人都习惯于使用指针++和- -操作符来遍历字符串。有时也使用数组处理字符串中的字符。无论 ASCII 码字符串还是 Unicode 字符串，使用指针均能够正确无误地返回要寻求的字符位置。

在 STL 中只有一个字符串类，即 basic_string。basic_string 管理一个'\0'结尾的字符数组。字符类型由模板参数决定。通常，basic_string 被处理为不透明的对象，靠获得只读指针来访问缓冲区，写操作是由 basic_string 的成员函数实现的。STL 的 C++标准程序库中的 string 类，使用时不必担心内存是否充足、字符串长度等问题。string 作为类出现，其集成的操作函数（方法）足以完成多数情况下的需要。可以使用“=”进行赋值，使用“==”进行等值比较，使用“+”做串联。

为了使用 string 类，必须包含头文件<string>。在 STL 库中，basic_string 有两个预定义类型：包含 char 的 string 类型和包含 wchar 的 wstring 类型。

string 类的 string::npos 可以同时定义字符串的最大长度，通常设置为无符号 int 的最大值。String 类包含了 6 个构造函数。string 类支持两种输入方式：cin 方式和 getline()方式。简单示例如下：

```
string stuff;
cin>>stuff;
getline(cin, stuff);
```

上述三行代码，第一行是声明 string 类的对象 stuff，第二行是从屏幕读入输入的字符串；第三行同样实现第二行代码的功能。

string 库提供了许多其他功能，例如，删除字符串的部分或全部；用一个字符的部分或全部替换另一个字符串的部分或全部；插入、删除字符串中的数据；比较、提取、复制、交换等操作。

在 STL 中还提供了另一个模板类：auto_ptr 类。主要用于管理动态内存分配。如果使用 new 函数分配堆中的内存，而又不记得回收这部分内存，会导致内存泄漏。必须使用 delete 语句释放该内存块。即使在函数末端添加了 delete 语句释放内存，还需要在任何跳出该函数的语句（如抛出异常）之前添加释放内存的处理，例如，goto 语句，throw 语句。auto_ptr 模板定义了类似指针的对象，将 new 获得的地址赋给该对象。当 auto_ptr 对象过期时，析构函数将使用 delete 来释放内存。将 new 返回的地址赋值给 quto_ptr 对象时，无须记住还需要释放这些内存。在 auto_ptr 对象过期时，内存将自动被释放。

在 C++语言中，使用 STL 中的 auto_ptr 对象，必须包含头文件<memory>。该文件包括 auto_ptr 模板。使用通常的模板句法来实例化所需类型的指针。auto_ptr 构造函数是显式的，不存在从指针到 auto_ptr 对象的隐式类型转换，代码如下：

```
auto_ptr <double> pd;
double *p_reg=new double;
pd=p_reg;                                   //不允许
pd=auto_ptr<double>(p_reg);                 //允许
auto_ptr<double> pauto=p_reg;               //不允许
auto_ptr<double>pauto(p_reg);               //允许
```

模板可以通过构造函数将 auto_ptr 对象初始化为一个常规指针。auto_ptr 是一个智能指针，但特性远比指针要多。值得注意的是，在使用 auto_ptr 时，只能配对使用 new 和 delete。

提示：只能对 new 分配的内存使用 auto_ptr 对象，不要对由 new[]分配的或通过声明变量分配的内存使用它。

总结：C++库中 auto_ptr 对象是一种智能指针。智能指针是一种类，即其对象的特征类似于指针。智能指针存储 new 分配的内存地址，也可以被解除引用。智能指针是一个类对象，可以修改和扩充简单指针的行为。智能指针可以建立引用计数，从而多个对象可以共享由智能指针跟踪的同一个值。当使用该值的对象数为 0 时，智能指针将删除该值。智能指针可以提高内存的使用效率，帮助防止内存泄漏。

2.2 字符的特点

“字符”本身是个有趣的概念。在纸上或者屏幕上，字符“C”仅仅是一段曲线而已。在计算机中，用一个 8 位字节存储该字符，并赋值为 67；C 还是拉丁字母的第 3 个字母；是碳原子的缩写形

式。很偶然的机会，C 被用来表示一种程序设计语言的名字。目前，字符集是在字符与整数值之间的一种映射关系。

C++程序员通常都假定能够使用美国字符集（ASCII），但 C++允许程序员有缺少某些字符的可能性。包含 ASCII 里所没有的字符的字符集对其提出了巨大的挑战。许多语言（例如中文、丹麦文、法文、冰岛文、日文）无法用 ASCII 正常写出来。即使扩充到 16 位字符集，也无法将所有人类所知的字符放在同一字符集中。据说已经存在 32 位字符集能保存每一个字符，但尚未得到推广。

作为 C++语言，允许程序员使用任何字符集作为字符串的字符，也可以使用扩充字符集或可移植的数值编码。原则上字符串能以任何带有正确的复制操作的类型作为其字符类型。标准 string 要求作为其字符类型的类型不包含用户自定义复制操作，有助于字符串 I/O 的简化与提高效率。

字符类型的性质由字符特征类（char_traits）定义。字符特征类是下述模板的专门化。

```
template <class Ch> struct char_traits{ }
```

所有的字符特征类均定义在名称空间 std 中，标准的字符特征类由<string>给出。通用字符串特征类 char_traits 本身无任何属性，针对特定字符类型的专门 char_traits 具有属性。

对于标准字符串模板的实例 basic_string 依赖诸多类型和函数。一个类型作为 basic_string 字符类型，必须提供支持上述功能的字符特征类（char_traits）。

C++的字符串类模板 basic_string 还集合了大量的字符串处理函数。大量函数的使用方法将在 2.3 节介绍。

2.3　字符串类模板（basic_string 模板类）

标准库字符串功能的基础是模板 basic_string，该模板提供了许多成员和方法，与标准容器类似。该模板的声明如下：

```
template<class Ch, class Tr=char_traits<Ch>, class A=allocator<Ch>>class std::basic_string
{
public:
......
}
```

在上述模板声明中，共包含 3 个参数：Ch，Tr，A。

第 1 个参数（class Ch）是单个字符所属型别；

第 2 个参数（class Tr=char_traits<Ch>）是特性类别，提供字符串类别中所有的字符核心操作。该特性类别规定了“复制字符”或“比较字符”的做法；如果不指定该特性类别，会根据现有的字符型别采用默认的特性类别。

第 3 个参数（class A=allocator<Ch>）带有默认值，定义字符串类别所采用的内存模式，通常设定为“默认内存模型 allocator”。该模板及其相关功能都定义在名称空间 std 中，由头文件<string>给出。其中包含了两个 typedef，为最常用的串类型提供方便的名字。即 C++标准程序库提供了两个 basic_string< > 特化版本。

```
typedef basic_string <char> string;          //定义一个 basic_string 类型的容器
typedef basic_string<wchar> wstring;         //定义一个宽字符的 basic_string 类型的容器
```

其中 wstring 是便于使用宽字符集，例如 Unicode 或某些欧洲字符集。但所有的字符串类型均使用相同接口，用法和问题都一样。在本书中，仍以 string 表示任何字符串型别。

basic_string 有些类似于 vector，但 basic_string 还提供典型的字符串操作，例如：子串检索。basic_string 没有提供像 vector 那样的一组完整操作。一般情况下，string 不会直接使用数组或者 vector 实现，为了很好地支持 string 常见应用，实现中需要尽量减少复制，对短的字符串不使用自由存储空间，允许对长的字符串进行简单修改。

basic_string<T>没有虚函数，这点和其他标准库类型一致。当需要设计更复杂的文字处理类时，可以用它加以实现。

与其他标准容器一样，basic_string 还提供了一组成员类型名，程序员能使用这些与串相关的类型。

例如：

```
typedef  Tr traits_type;                         //类型定义 Tr 代表类型 trait_type
typedef  typename Tr::char_type value_type       //
typedef  A allocator_type                        //类型定义 A 代表类型 allocator_type
......
```

basic_string 除支持最简单的 basic_string<char>之外，还支持许多不同种类的字符串，例如：

```
typedef basic_string<unsigned char> Ustring;
tyepdef basic_string<Jchar> Jstring;             //日文字符串
```

无论怎么定义字符串，模板 basic_string 的大量方法均可以便捷使用。模板 basic_string<Ch>能存放集合 Ch 中的任何字符，特别是 string 中的 0。“字符类型” Ch 的行为必须像字符，但它不能有用户确定的复制构造函数、析构函数和复制赋值。

字符串类 string 包含了一系列的成员和函数，但也有个别功能没有实现，例如：正则表达式和较复杂的文本处理功能。

总体来说，string 类使得字符串操作变得简单了。程序员可以定义 string 类型的对象，string 类的重载操作符和成员函数使字符串处理非常容易。在 2.4 节将介绍如何使用 string 类，详细讲解 string 类的各种成员和函数的使用方法。

2.4 字符串操作的通用函数

程序员定义 string 对象，string 类自身可以管理内存，程序员不必操心内存分配细节。string 类提供的各种操作函数大致分为：构造器和析构器，大小和容量，元素存取（访问），字符串比较，字符串内容修改和替换，字符串连接，字符串 I/O 操作，字符串的搜索和查找。本节还将介绍静态数据类型 npos、string 类的迭代器以及配置器等内容。表 2-1 列出了 string 类的方法汇总。

表 2-1 string 类的方法汇总

函数名称	效果	函数名称	效果
构造函数	产生或复制字符串	empty()	判断字符串是否为空
析构函数	销毁字符串	capacity()	返回重新分配之前的字符容量
=，assign	赋予新值	reserve()	保留内存以存储一定数量的字符
swap	交换两个字符串的内容	[], at()	存取单一字符
+=,append(), push_back()	添加字符	>>，getline()	从 stream 中读取某值

续表

函数名称	效　果	函数名称	效　果
insert()	插入字符	<<	将值写入 stream
erase()	删除字符	copy()	将内容复制为一个 C-string
clear()	移除全部字符	c_str()	将内容以 C-string 形式返回
resize()	改变字符数量	data()	将内容以字符数组形式返回
replace()	替换字符	substr()	返回子字符串
+	串联字符串	find()	搜寻某子字符串或字符
= =,!=,<,<=,>,>=,compare()	比较字符串内容	begin(), end()	提供正向迭代器支持
size(), length()	返回字符数量	rbegin(), rend()	提供逆向迭代器支持
max_size()	返回字符的最大可能个数	get_allocator()	返回配置器

2.4.1　构造器和析构器

构造器函数带有 4 个参数，其中 3 个具有默认值。要初始化一个 string 类，可以使用 C 风格字符串、另一个 string 类型对象、C 风格字符串的部分、另一个 string 的部分或其他字符序列。值得注意的是：不能使用字符或者整数去初始化字符串。

常见的 string 类构造函数有以下几种形式：

```
string  strs                          //生成空字符串
string  s(str)                        //生成字符串 str 的复制品
string  s(str, stridx)                //将字符串 str 中始于 stridx 的部分，作为构造函数的初值
string  s(str, strbegin, strlen)      //将字符串 str 中始于 strbegin、长度为 strlen 的部分作
                                        为字符串初值
string  s(cstr)                       // 以 C_string 类型 cstr 作为字符串 s 的初值
  string  s( cstr, char_len)          //以 C_string 类型 cstr 的前 char_len 个字符串作为字符
                                        串 s 的初值
string  s(num, c)                     //生成一个字符串，包含 num 个 c 字符
string  s(strs, beg, end)             //以区间[beg, end]内的字符作为字符串 s 的初值
```

析构函数的形式如下：

```
~string()                             //销毁所有内存，释放内存
```

如果字符串只包含一个字符，使用构造函数对其初始化时，使用以下两种形式比较合理。

```
std::string  s('x');                  //错误
std::string  s(1,'x');                //正确
```

或

```
std::string  s("x");                  //正确
```

上述内容涉及的 C_string 在这里要做简要介绍。C_string 一般认为是常规的 C++字符串。目前，存在一个从 const char* 到 strings 的隐式型别转换，却不存在从 string 对象到 C_string 的自动型别转换。对于 string 类型的字符串，可以通过函数 c_str()返回该 string 类对象对应的 C_string。

通常在程序中，整个程序应该坚持使用 string 类对象，直到必须将内容转换为 char*时才将其转换为 C-string。

以上内容可参考例 2-1。请读者关注中文注释。

例 2-1

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string  str("12345678");
char ch[]="abcdefgh";
string  a;                              //定义一个空字符串
string  str_1(str);                     //构造函数,全部复制
string  str_2(str,2,5);  //构造函数,从字符串 str 的第 2 个元素开始,复制 5 个元素,赋值给 str_2
string  str_3(ch,5);                    //将字符串 ch 的前 5 个元素赋值给 str_3
string  str_4(5,'X');                   //将 5 个'X'组成的字符串 “XXXXX” 赋值给 str_4
string  str_5(str.begin(),str.end()); //复制字符串 str 的所有元素给字符串 str_5
cout<<str<<endl;
cout<<a<<endl;
cout<<str_1<<endl;
cout<<str_2<<endl;
cout<<str_3<<endl;
cout<<str_4<<endl;
cout<<str_5<<endl;
}
```

例 2-1 的执行效果见图 2-1。

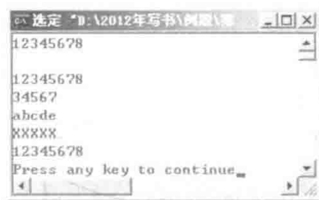


图 2-1　例 2-1 的执行效果

通过对上述内容的学习，对 string 类的构造函数和析构函数应该有初步了解，并能够学会使用构造函数创建 string 对象。

2.4.2　大小和容量

对于 string 类的对象包含 3 种求解大小的函数：size()和 length()；max_size()；capacity()。

（1）size()和 length()。这两个函数会返回 string 对象中的字符个数；两个函数的执行效果相同。

（2）max_size()。函数 max_size 返回 string 类对象最多包含的字符数。一旦程序使用长度超过 max_size()的 string 操作，编译器会抛出 length_error 异常。

（3）capacity()。重新分配内存之前，string 对象能包含的最大字符数。

string 类还有一个函数 reserve()。当调用该函数时，得以保留一定的预留容量。reserve()的默认参数为 0。

下面用例 2-2 说明上述几个函数的使用方法。

例 2-2

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
int size=0;
int length=0;
unsigned long maxsize=0;
```

```
int capacity=0;
string str("12345678");
string str_custom;
str_custom.reserve(5);
str_custom=str;
size=str_custom.size();
length=str_custom.length();
maxsize=str_custom.max_size();
capacity=str_custom.capacity();
  cout<<"size= "<<size<<endl;
cout<<"length= "<<length<<endl;
cout<<"maxsize= "<<maxsize<<endl;
cout<<"capacity= "<<capacity<<endl;
}
```

例 2-2 的执行效果见图 2-2。

由例 2-2 可知，string 类对象 str_custom 调用函数 reserve()时，似乎并没有起到重新分配内存的目的。笔者所用编译器为 Visual C++6.0。

修改上述代码，删除语句 str_custom.reserve(5)，在代码 str_custom=str 之后添加代码如下：

```
str_custom.resize(5);
```

例 2-2 修改后的执行效果见图 2-3。

图 2-2　例 2-2 的执行效果

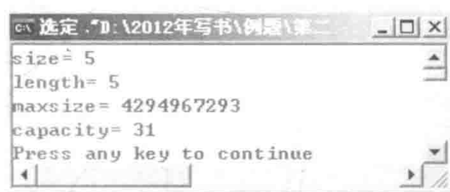


图 2-3　例 2-2 修改后的执行效果

当重新设置 string 类对象 str_custom 的大小之后，重新求解 str_custom 的大小和设置的数值一致。

2.4.3　元素存取（访问）

string 是 C++中的字符串。字符串是一种特殊类型的容器，专门设计来操作字符序列。字符串元素的访问是被允许的。一般有两种方法访问单一字符：下标操作符[]和成员函数 at()。两者都返回指定索引的对应位置上的字符。第 1 个字符索引（下标）为 0，最后的字符索引为 length()-1。两种访问方法存在区别。

提示：1. 下标操作符[] 在使用时不检查索引的有效性。如果下标超出字符的长度范围，会引起不必要的麻烦；对于常量字符串，使用下标操作符时，字符串的最后字符（即'\0'）是有效的。对应 string 对象（常量型）最后一个字符的下标是有效的，调用返回字符'\0'。

2. 函数 at()z 在使用时会检查下标是否有效。如果给定的下标超出字符的长度范围，系统会抛出 out_of_range 异常。

为了允许更改 string 内容，下标操作符[]的非常量版本和函数 at()都返回字符的“引用”。如果

字符串发生重分配行为（例如：重新被赋值），这些“引用”立即失效。下面用例 2-3 加以说明：

例 2-3

```
#include <iostream>
#include <string>
void main()
{const std::string cS("conststring");
std::string s("abcde");
char temp=0;
char temp_1=0;
char temp_2=0;
char temp_3=0;
char temp_4=0;
char temp_5=0;
temp=s[2];                                 //获取字符'c'
temp_1=s.at(2);                            //获取字符'c'
temp_2=s[s.length()];                      //未定义行为，但visual c++6.0执行时未报错
temp_3=cS[cS.length()];                    //指向字符'\0'
temp_4=s.at(s.length());                   //程序异常
temp_5=cS.at(cS.length());                 //程序异常
std::cout<<temp<<temp_1<<temp_2<<temp_3<<temp_4<<temp_5<<std::endl;
}
```

通过对上述代码的分析，可知要理解字符串的存取需要多实践、多尝试，并且要牢记基础知识和基本规则。

为了修改 string 的字符串的内容，下标操作符和函数 at()都返回字符的“引用”。当字符串的内存被重新分配以后，可能会发生执行错误。详见例 2-4。

例 2-4

```
#include <iostream>
#include <string>
void main()
{
std::string s("abcde");
std::cout<<s<<std::endl;
char& r=s[2];                              //建立引用关系
char* p=&s[3];                             //建立引用关系
r='X';                                     //修改内容
*p='Y';                                    //修改内容
std::cout<<s<<std::endl;                   //输出
s="12345678";                              //重新赋值
r='X';                                     //修改内容
*p='Y';                                    //修改内容
std::cout<<s<<std::endl;                   //输出
}
```

例 2-4 的执行效果见图 2-4。

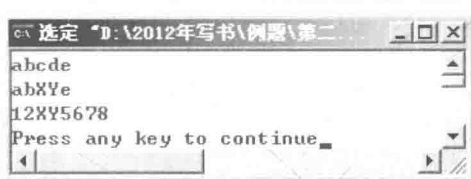


图 2-4　例 2-4 的执行效果

在例 2-4 中，使用 Visual C++6.0 编译器编译，字符串被重新赋值后，修改其中某位置字符的值，仍然成功。这和《C++标准程序库》中的描述并不一致。从某种意义上讲，字符串被重新赋值后，其原来的引用关系已经没有意义了。

2.4.4　字符串比较

字符串可以和类型相同的字符串比较，也可以和具有同样字符类型的数组比较。basic_string 类模板既提供了比较运算符：>，　<，　= =，　>=，　<=，　!=等；还提供了方法 compare()。对于方法 compare()，如果相比较的两个子串相同，compare()函数返回 0，否则返回非零值。

1．compare()函数

类 basic_string 的函数 compare()的原型为：

```
int compare (const basic_string& s) const;
int compare (const Ch* p) const;
int compare(size_type pos, size_type n, const basic_string& s)const;
int compare(size_type pos, size_type n, const basic_string& s, size_type pos2, size_type n2) const;
int compare(size_type pos, size_type n, const Ch* p, size_type n2=npos)const;
```

如果在使用 compare()函数时，参数中出现了位置和大小，比较时只能使用指定的子串。例如：

```
s.compare(pos,n,s2);
```

如果参与比较的两个串值相同，则函数返回 0；如果字符串 s 按字典序要先于 s2，则返回负值；否则返回正值。下面以例 2-5 说明如何使用 srting 类的函数 compare()。请注意代码中的中文注释语句。

例 2-5

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
 string A("aBcdef");
 string B("AbcdEf");
 string C("123456");
 string D("123dfg");
    //下面是各种比较方法
 int m=A.compare(B);                          //完整的A和B的调整
 int n=A.compare(1,5,B);                      //"Bcdef"和"AbcdEf"的比较
 int p=A.compare(1,5,B,4,2);                  //"Bcdef"和"Ef"比较
 int q=C.compare(0,3,D,0,3);                  //"123"和"123"比较
 cout<<"m="<<m<<", n="<<n<<", p="<<p<<", q="<<q<<endl;
 cin.get();
 return;
}
```

例 2-5 的执行效果见图 2-5。

总结：从例 2-5 可知，string 类的比较函数 compare()使用非常方便，并且能区分字母的大小写。建议大家多使用。比较运算符使用更加方便，在后面加以详细介绍。

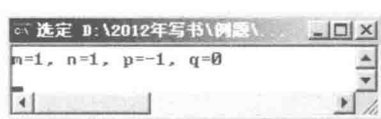


图 2-5　例 2-5 的执行效果

2．比较运算符

string 类的常见运算符包括：>，<，= =，>=，<=，!=。其中文意义分别为“大于”、“小于”、“等

于”、“大于等于”、“小于等于”、“不等于”。

比较运算符使用起来非常方便，这里就不再介绍其函数原型了，直接使用即可。详见例 2-6。

例 2-6

```
#include <iostream>
#include <string>
using namespace std;
void TrueOrFalse(int x)
{
cout<<(x?"True":"False")<<endl;
}
void main()
{
string S1="DEF";
string CP1="ABC";
string CP2="DEF";
string CP3="DEFG";
string CP4="def";
cout<<"S1= "<<S1<<endl;
cout<<"CP1= "<<CP1<<endl;
cout<<"CP2= "<<CP2<<endl;
cout<<"CP3= "<<CP3<<endl;
cout<<"CP4= "<<CP4<<endl;
cout<<"S1<=CP1 returned ";
TrueOrFalse(S1<=CP1);
cout<<"S1<=CP2 returned ";
TrueOrFalse(S1<=CP2);
cout<<"S1<=CP3 returned ";
TrueOrFalse(S1<=CP3);
cout<<"CP1<=S1 returned ";
TrueOrFalse(CP1<=S1);
cout<<"CP2<=S1 returned ";
TrueOrFalse(CP2<=S1);
cout<<"CP4<=S1 returned ";
TrueOrFalse(CP4<=S1);
cin.get();
}
```

程序的执行结果如下：

```
S1= DEF
CP1= ABC
CP2= DEF
CP3= DEFG
CP4= def
S1<=CP1 returned False
S1<=CP2 returned True
S1<=CP3 returned True
CP1<=S1 returned True
CP2<=S1 returned True
CP4<=S1 returned False
```

总结：从 string 类的比较运算符可知，比较运算符可以方便地实现字符串的大小比较。在使用时注意两个参加比较的字符串，任何一个都不能为 NULL。否则程序会异常退出。

2.4.5　字符串内容修改和替换

字符串内容的变化可以分为两种：修改和替换。本节分别介绍修改和替换。

1．更改内容

可以使用多个函数修改字符串的值。例如：assign()，operator=，erase()，交换（swap），插入（insert）等。另外还可以通过 append()函数实现添加字符。下面一一介绍各函数的使用方法。

（1）assign()函数

assign() 函数可以直接给字符串赋值；既可以将整个字符串赋值给新串，也可以将字符串的子串赋值给新串。其在 basic_string 类中的原型为：

```
basic_string& assign(const E *s);                        //直接使用字符串赋值
basic_string& assign(const E *s, size_type n);           //限定被初始化的字符长度
basic_string& assign(const basic_string& str, size_type pos, size_type n);
                                                         //将 str 的子串赋值给调用串
basic_string& assign(const basic_string& str);           //使用字符串的“引用”赋值
basic_string& assign(size_type n, E c);                  //使用 n 个重复字符赋值
basic_string& assign(const_iterator first, const_iterator last);      //使用迭代器赋值
```

以上几种方法均会在例 2-7 中有所体现，代码如下：

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string str1("123456");
string str;
str.assign(str1);                                  //直接赋值
cout<<str<<endl;
str.assign(str1,3,3);                              //赋值子串
 cout<<str<<endl;
 str.assign(str1,2,str1.npos);                     //从位置 2 至末尾的子串赋值
 cout<<str<<endl;
 str.assign(5,'X');                                //重复 5 个 'X' 字符
 cout<<str<<endl;
string::iterator itB;
string::iterator itE;
itB=str1.begin();
itE=str1.end();
str.assign(itB,(--itE));                           //从第一个至倒数第二个元素，赋值给字符串 str
 cout<<str<<endl;
   ……
}
```

（2）operator=

“operator=”的功能就是赋值。

（3）函数 erase()

函数原型为：

```
iterator erase(iterator first, iterator last);
iterator erase(iterator it);
basic_string& erase(size_type p0 = 0, size_type n = npos);
```

函数的使用方法如下：

```
str.erase(str.begin(),str.end());
或 str.erase(3);
```

（4）swap()函数

函数原型为：

```
void swap(basic_string& str);
```

函数的使用方法如下：

```
string str2("abcdefghijklmn");
str.swap(str2);
```

（5）insert()函数

函数原型为：

```
basic_string& insert(size_type p0, const E *s);                 //插入1个字符至字符串s前面
basic_string& insert(size_type p0, const E *s, size_type n);  //将s的前3个字符插入p0位置
basic_string& insert(size_type p0, const basic_string& str); //将字符串str插入p0位置
basic_string& insert(size_type p0, const basic_string& str, size_type pos, size_type n);
                                                                //选取str的子串
basic_string& insert(size_type p0, size_type n, E c);          //在下标p0位置插入n个字符c
iterator insert(iterator it, E c);                              //在it位置插入字符c
void insert(iterator it, const_iterator first, const_iterator last);
                                                                //在字符串前插入字符
void insert(iterator it, size_type n, E c);                     //在it位置重复插入n个字符c
```

函数的使用方法如下：

```
string A("ello");
string B("H");
B.insert(1,A);
cout<<B<<endl;
//.............
A="ello";
B="H";
B.insert(1,"yanchy ",3);
cout<<B<<endl;
//............
A="ello";
B="H";
B.insert(1,A,2,2);
cout<<B<<endl;
//..............
A="ello";
B="H";
B.insert(1,5,'C');
cout<<B<<endl;
//..............
A="ello";
B="H";
```

```
string::iterator it=B.begin()+1;
const string::iterator itF=A.begin();
const string::iterator itG=A.end();
B.insert(it,itF,itG);
cout<<B<<endl;
```

（6）append()函数

函数原型为：

```
basic_string& append(const E *s);                    //在原始字符串后面追加字符串 s
basic_string& append(const E *s, size_type n); //在原始字符串后面追加字符串 s 的前 n 个字符
basic_string& append(const basic_string& str, size_type pos, size_type n);
                                       //在原始字符串后面追加字符串 s 的子串 s[pos,….,pos+n-1]
basic_string& append(const basic_string& str); //在原始字符串之后追加字符串 str
basic_string& append(size_type n, E c);              //追加 n 个重复字符
basic_string& append(const_iterator first, const_iterator last);      //使用迭代器追加
```

函数的使用方法如下：

```
A="ello";
B="H";
 cout<<A<<endl;
 cout<<B<<endl;
 B.append(A);
 cout<<B<<endl;
//...............
A="ello";
B="H";
 cout<<A<<endl;
 cout<<B<<endl;
 B.append("12345",2);
cout<<B<<endl;
//................
A="ello";
B="H";
 cout<<A<<endl;
 cout<<B<<endl;
 B.append("12345",2,3);
cout<<B<<endl;
//................
A="ello";
B="H";
 cout<<A<<endl;
 cout<<B<<endl;
 B.append(10,'a');
cout<<B<<endl;
//................
A="ello";
B="H";
 cout<<A<<endl;
 cout<<B<<endl;
 B.append(A.begin(),A.end());
cout<<B<<endl;
```

下面通过例 2-7 分别介绍这些函数的使用方法。

例 2-7

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string str1("123456");                          //定义字符串
string str2("abcdefghijklmn");
string str;
str.assign(str1);                               //使用字符串 str1 初始化字符串 str
cout<<str<<endl;
str.assign(str1,3,3);                           //使用 str1 的一部分初始化字符串 str
 cout<<str<<endl;
 str.assign(str1,2,str1.npos);                  //使用 str1 的一部分初始化字符串 str
 cout<<str<<endl;
 str.assign(5,'X');                             //使用 5 个'X'初始化字符串 str
 cout<<str<<endl;
string::iterator itB;
string::iterator itE;
itB=str1.begin();                               //获取字符串起始位置
itE=str1.end();                                 //获取字符串末端位置指针
str.assign(itB,(--itE));
 cout<<str<<endl;
//////////////////////////
str=str1;
 cout<<str<<endl;
/////////////////////////
str.erase(3);                                   //删除元素
cout<<str<<endl;
str.erase(str.begin(),str.end());
 cout<<":"<<str<<":"<<endl;
/////////////////////////
str.swap(str2);                                 //交换元素
cout<<str<<endl;
/////////////////////////
string A("ello");
string B("H");
B.insert(1,A);                                  //插入元素
cout<<B<<endl;
///.............
A="ello";
B="H";
B.insert(1,"yanchy ",3);                        //插入元素
cout<<"插入: "<<B<<endl;
///.............
A="ello";
B="H";
B.insert(1,A,2,2);                              //插入元素
cout<<"插入: "<<B<<endl;
///..............
A="ello";
B="H";
B.insert(1,5,'C');                              //插入元素
cout<<"插入: "<<B<<endl;
///..............
A="ello";
```

```
B="H";
string::iterator it=B.begin()+1;
const string::iterator itF=A.begin();
const string::iterator itG=A.end();
B.insert(it,itF,itG);                             //插入内容
cout<<"插入:"<<B<<endl;
///..............
A="ello";
B="H";
 cout<<"A= "<<A<<", B= "<<B<<endl;
 B.append(A);                                     //追加字符串
 cout<<"追加:"<<B<<endl;
//...............
A="ello";
B="H";
 cout<<"A= "<<A<<", B= "<<B<<endl;
 B.append("12345",2);                             //追加字符串
cout<<"追加:"<<B<<endl;
//................
A="ello";
B="H";
 cout<<"A= "<<A<<", B= "<<B<<endl;
 B.append("12345",2,3);                           //追加字符串
cout<<"追加:"<<B<<endl;
//................
A="ello";
B="H";
 cout<<"A= "<<A<<", B= "<<B<<endl;
 B.append(10,'a');                                //追加字符串
cout<<"追加:"<<B<<endl;
//................
A="ello";
B="H";
cout<<"A= "<<A<<", B= "<<B<<endl;
 B.append(A.begin(),A.end());                     //追加字符串
cout<<"追加:"<<B<<endl;
cin.get();
}
```

例 2-7 的执行效果见图 2-6，可以根据源代码逐项对照。

图 2-6　例 2-7 的执行效果

2. 替换

如果在一个字符串中标识出具体位置，便可以通过下标操作修改个别位置的字符值，或者替换整个子串。完成此项工作需要使用 string 类的 replace()函数。该函数的原型为：

```
    (1) basic_string& replace(size_type p0, size_type n0, const E *s);
    // 使用字符串 s 中的 n 个字符，从源串的位置 p0 处开始替换
    (2) basic_string& replace(size_type p0, size_type n0, const E *s, size_type n);
    //使用字符串 s 中的 n 个字符，从源串的位置 p0 处开始替换 1 个字符
    (3) basic_string& replace(size_type p0, size_type n0, const basic_string& str);
    // 同 (1)
    (4) basic_string& replace(size_type p0, size_type n0, const basic_string& str, size_type
pos, size_type n);
    // 使用串 str 的子串 str[pos,pos+n-1]替换源串中内容，从位置 p0 处开始替换，替换字符 n0 个
    (5) basic_string& replace(size_type p0, size_type n0, size_type n, E c);
    // 使用 n 个字符'c'替换源串中位置 p0 处开始的 n0 个字符
    (6) basic_string& replace(iterator first0, iterator last0, const E *s);
    // 使用迭代器替换，和 (1) 用法类似
    (7) basic_string& replace(iterator first0, iterator last0, const E *s, size_type n);
    // 和 (2) 类似
    (8) basic_string& replace(iterator first0, iterator last0, const basic_string& str);
    // 和 (3) 类似
    (9) basic_string& replace(iterator first0, iterator last0, size_type n, E c);
    // 和 (5) 类似
    (10) basic_string& replace(iterator first0, iterator last0, const_iterator first,
const_iterator last);
    // 使用迭代器替换
```

函数的使用方法见例 2-8。

例 2-8

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string var("abcdefghijklmn");
const string dest("1234");
string dest2("567891234");
var.replace(3,3,dest);                                          //替换
cout<<"1: "<<var<<endl;
//.....................
var="abcdefghijklmn";
var.replace(3,1,dest.c_str(),1,3);                              //替换
cout<<"2: "<<var<<endl;
//.....................
var="abcdefghijklmn";
var.replace(3,1,5,'x');                                         //替换
cout<<"3: "<<var<<endl;
//.....................
string::iterator itA,itB;
string::iterator itC,itD;
itA=var.begin();
```

```
    itB=var.end();
    var="abcdefghijklmn";
    var.replace(itA,itB,dest);                                    //替换
    cout<<"4: "<<var<<endl;
    //.....................
    itA=var.begin();                                              //起始位置
    itB=var.end();                                                //终止位置
    itC=dest2.begin()+1;
    itD=dest2.end();
    var="abcdefghijklmn";
    var.replace(itA,itB,itC,itD);                                 //代替
    cout<<"5: "<<var<<endl;
    //.....................
    var="abcdefghijklmn";
    var.replace(3,1,dest.c_str(),4);                              //这种方式会限定字符串替换的最大长度
    cout<<"6: "<<var<<endl;
    //.....................
    }
```

例 2-8 的执行效果见图 2-7。

总结：本节讲述了诸多字符串修改和替换的函数及其使用方法，对每个函数均给出了例题。由于每个函数可能有多个原型，希望读者根据自己的情况，掌握其中的一种或两种，满足自己使用的需要。并能够认真阅读本章节中的源代码，对照例题的执行效果，彻底掌握本节内容。

图 2-7　例 2-8 的执行效果

2.4.6 字符串连接

在 C++STL 模板库中，可以使用 operator+将两个字符串拼接起来。拼接就是将两个子串连接起来，将其中一个放在另一个串之后，形成一个新串。例如：

```
string str1("Inter");
string str2("national");
string str3=str1+str2;                                   //字符串连接
cout<<str3<<endl;
```

输出结果为：

```
International
```

2.4.7 字符串 I/O 操作

operator<<和 operator>>提供了 C++语言的字符串输入/输出功能。operator<<可以实现将字符写入一个流中（例如：ostream）；operator>>可以实现将以空白或回车为结束的完整的字符读入对应的字符串中，开头和结尾的空白字符不包括到字符串中。

还有一个常用的函数 getline()，函数原型包括以下两种形式：

```
template<class CharType, class Traits, class Allocator> basic_istream<CharType, Traits>& getline
    (basic_istream<CharType, Traits>& _Istr,basic_string<CharType,Traits, Allocator>& _Str
    );
```

```
    //上述原型包含 2 个参数，第 1 个参数是输入流；第 2 个参数是保存输入内容的字符串。
    template<class CharType, class Traits, class Allocator> basic_istream<CharType, Traits>& getline
    (basic_istream<CharType, Traits>& _Istr, basic_string<CharType, Traits, Allocator>& _Str,
    CharType _Delim
    );
    //上述原型包含 3 个参数：第 1 个参数是流，第 2 个参数保存输入的字符串，第 3 个参数指定分界符。
```

该函数实现读取一个整行的所有字符到字符串中。在读取字符时，遇到文件结束符、分界符、回车符时，将终止读入，文件结束符、分界符、回车符在字符串中不保存；当已读入字符超过字符串所能容纳的最大字符数时，终止读入。下面分别按两种函数原型举例说明，见例 2-9。

例 2-9

```
#include <iostream>
#include <string>
using namespace std;
void main()
{string s1,s2;
getline(cin,s1);                           //从屏幕读取一行字符
getline(cin, s2, ' ');                     //从屏幕读取一行字符
cout<<"You inputed chars are: "<<s1<<endl;
cout<<"You inputed chars are: "<<s2<<endl;
}
```

例 2-9 的执行效果见图 2-8。

提示：观察图 2-8，其中输入的第二行字符中间包含空格字符。空格之后的字符没有被存储到字符串 s2 中。

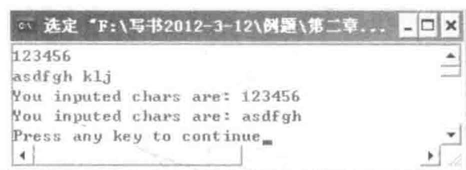


图 2-8　例 2-9 的执行效果

2.4.8　字符串的搜索和查找

在 C 语言、C++语言中，实现字符串查找功能的函数非常多。在大家的习惯中，搜索查找往往和替换联系在一起，字符串替换的内容已经在 2.4.5 节中讲述过了。本节重点讲述在 STL 模板库中的字符串的搜索查找。

在 STL 中，字符串查找功能可以实现搜索单个字符、搜索子串，可以实现前向搜索、后向搜索，可以分别实现搜索第一个和最后一个满足条件的字符（或子串），实现搜索第一个和最后一个与给定参考值不相等的字符（或子串）。如果查找函数 find()和其他函数没有搜索到期望的字符（或子串），函数返回 npos；如果搜索成功，则返回搜索到的第 1 个字符或子串的位置。

提示：关于 npos：npos 是一个无符号整数值，初始值为-1。当搜索失败时，npos 表示“没有找到（not found）”或“所有剩余字符”。

值得注意的是，所有查找函数 find()的返回值均是 size_type 类型，是无符号整数类型，是表明字符串元素的个数或者字符在字符串中的位置。

下面分别介绍和字符查找相关的函数。

1．find () 函数和 rfind()函数

find 函数的原型主要有 4 种。读者可以根据实际情况选择使用。

```
size_type find(value_type _Ch, size_type _Off = 0) const;
//上 1 行代码中 find()函数的第 1 个参数是被搜索的字符，第 2 个参数是在源串中开始搜索的下标位置
size_type find(const value_type* _Ptr, size_type _Off = 0) const;
//上 1 行代码中 find()函数的第 1 个参数是被搜索的字符串，第 2 个参数是在源串中开始搜索的下标位置
size_type find(const value_type* _Ptr, size_type _Off = 0,size_type _Count) const;
//上 1 行代码中第 1 个参数是被搜索的字符串，第 2 个参数是源串中开始搜索的下标，第 3 个参数
//是关于第 1 个参数的字符个数，可能是_Ptr 的所有字符数，也可能是_Ptr 的子串字符个数
size_type find(const basic_string& _Str,size_type _Off = 0) const;
//函数的第 1 个参数是被搜索的字符串，第 2 个参数是在源串中开始搜索的下标位置
```

rfind()函数的原型和 find()函数的原型类似，参数情况也类似。

上述两个函数原型的使用方法见例 2-10。

例 2-10

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string str_ch("for");
string str("Hi, Peter, I'm sick. Please bought some drugs for me.");
string::size_type m= str.find('P',5);
string::size_type rm= str.rfind('P',5);
cout<<"Example-find(): The position(forward) of 'P' is: "<<(int)m<<endl;
cout<<"Example-rfind(): The position(reverse) of 'P' is: "<<(int)rm<<endl;
string::size_type n= str.find("some",0);                //查找
string::size_type rn= str.rfind("some",0);              //逆向查找
cout<<"Example-find(): The position(forward) of 'some' is: "<<(int)n<<endl;
cout<<"Example-rfind(): The position(reverse) of 'some' is: "<<(int)rn<<endl;
string::size_type mo= str.find("drugs",0,5);
string::size_type rmo= str.rfind("drugs",0,5);
cout<<"Example-find(): The position(forward) of 'drugs' is: "<<(int)mo<<endl;
cout<<"Example-rfind(): The position(reverse) of 'drugs' is: "<<(int)rmo<<endl;
string::size_type no= str.find(str_ch,0);
string::size_type rno= str.rfind(str_ch,0);
cout<<"Example-find(): The position of 'for' is: "<<(int)no<<endl;
cout<<"Example-rfind(): The position of 'for' is: "<<(int)rno<<endl;
cin.get();
}
```

例 2-10 的执行效果见图 2-9。

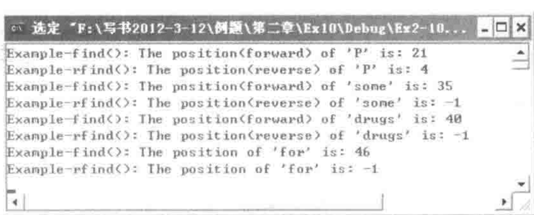


图 2-9　例 2-10 的执行效果

2. 函数 find_first_of ()和函数 find_last_of ()

函数 find_first_of ()实现在源串中搜索某字符串的功能，函数返回值是被搜索字符串的第 1 个字

符第 1 次出现的下标（位置）。如果查找失败，函数返回 npos。

函数 find_last_of()同样实现在源串中搜索某字符串的功能。所不同的是，函数返回值是被搜索字符串的最后 1 个字符的下标（位置）。如果查找失败，函数返回 npos。

上述两个函数的原型分别为：

```
size_type find_first_not_of(value_type _Ch, size_type _Off = 0) const;
size_type find_first_of(const value_type* _Ptr,size_type _Off = 0) const;
size_type find_first_of(const value_type* _Ptr, size_type _Off,size_type _Count) const;
size_type find_first_of(const basic_string& _Str, size_type _Off = 0) const;

size_type find_last_of(value_type _Ch, size_type _Off = npos) const;
size_type find_last_of(const value_type* _Ptr,size_type _Off = npos) const;
size_type find_last_of(const value_type* _Ptr, size_type _Off,size_type _Count)const;
size_type find_last_of(const basic_string& _Str,size_type _Off = npos) const;
```

例 2-11 详细阐述了上述两个函数的使用方法。这两个函数和 find()及 rfind()的使用方法相同，具体参数的意义亦相同。

例 2-11

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
string str_ch("for");
string str("Hi, Peter, I'm sick. Please bought some drugs for me.");
int length=str.length();                                      //获取字符串 str 的长度
string::size_type m= str.find_first_of('P',0);                //查找第一个字符'P'
string::size_type rm= str.find_last_of('P',(length-1));
cout<<"Example-find_first_of(): The position(forward) of 'P' is: "<<(int)m<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'P' is: "<<(int)rm<<endl;
string::size_type n= str.find_first_of("some",0);              //
string::size_type rn= str.find_last_of("some",(length-1));     //查找最后一个
cout<<"Example-find_first_of(): The position(forward) of 'some' is: "<<(int)n<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'some' is: "<<(int)rn<<endl;
string::size_type mo= str.find_first_of("drugs",0,5);
string::size_type rmo= str.find_last_of("drugs",(length-1),5);
cout<<"Example-find_first_of(): The position(forward) of 'drugs' is: "<<(int)mo<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'drugs' is: "<<(int)rmo<<endl;
string::size_type no= str.find_first_of(str_ch,0);
string::size_type rno= str.find_last_of(str_ch,(length-1)); /查找最后一个
cout<<"Example-find_first_of(): The position of 'for' is: "<<(int)no<<endl;
cout<<"Example-find_last_of(): The position of 'for' is: "<<(int)rno<<endl;
cin.get();
}
```

例 2-11 的执行效果见图 2-10。

3．函数 find_first_not_of ()和函数 find_last_not_of ()

函数 find_first_not_of ()的函数原型为：

```
size_type find_first_not_of(value_type _Ch,size_type _Off = 0) const;
//
size_type find_first_not_of(const value_type* _Ptr,size_type _Off = 0) const;
```

```
//
size_type find_first_not_of(const value_type* _Ptr, size_type _Off,size_type _Count)
const;
//
size_type find_first_not_of(const basic_string& _Str,size_type _Off = 0) const;
//
```

find_first_not_of ()函数实现在源字符串中搜索与指定字符（串）不相等的第 1 个字符。函数的参数意义和前面几个函数相同，使用方法和前面几个函数也基本相同。详见例 2-12。

例 2-12

```
#include <iostream>
#include <string>
using namespace std;
void main()
{string str_ch("for");
string str("Hi, Peter, I'm sick. Please bought some drugs for me.");
int length=str.length();
string::size_type m= str.find_first_not_of('P',0);   //查找第一个不是'P'的元素坐标，下同。
string::size_type rm= str.find_last_not_of('P',(length-1));
cout<<"Example-find_first_of(): The position(forward) of 'P' is: "<<(int)m<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'P' is: "<<(int)rm<<endl;
string::size_type n= str.find_first_not_of("some",0);
string::size_type rn= str.find_last_not_of("some",(length-1));
cout<<"Example-find_first_of(): The position(forward) of 'some' is: "<<(int)n<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'some' is: "<<(int)rn<<endl;
string::size_type mo= str.find_first_not_of("drugs",0,5);
string::size_type rmo= str.find_last_not_of("drugs",(length-1),5);
cout<<"Example-find_first_of(): The position(forward) of 'drugs' is: "<<(int)mo<<endl;
cout<<"Example-find_last_of(): The position(reverse) of 'drugs' is: "<<(int)rmo<<endl;
string::size_type no= str.find_first_not_of(str_ch,0);
string::size_type rno= str.find_last_not_of(str_ch,(length-1));
cout<<"Example-find_first_of(): The position of 'for' is: "<<(int)no<<endl;
cout<<"Example-find_last_of(): The position of 'for' is: "<<(int)rno<<endl;
cin.get();
}
```

例 2-12 的执行效果见图 2-11。

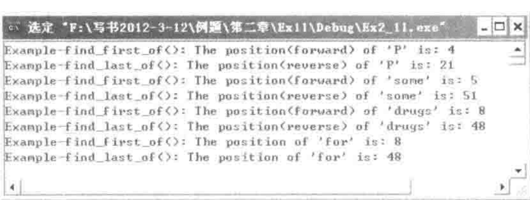


图 2-10　例 2-11 的执行效果

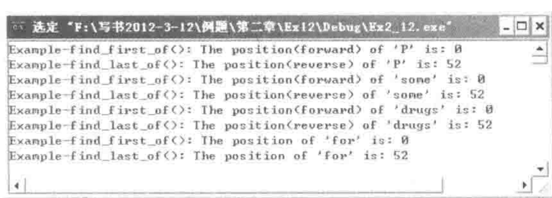


图 2-11　例 2-12 的执行效果

总结：本节主要讲述 C++STL 中的字符串查找函数。对于以上 6 个查找函数，其使用形式大致是相同的，对于每个函数均配备了实例作为参考。希望读者能认真对照例题，深刻理解这 6 个函数的使用

方法，对函数每个参数的意义有所体会。本书的最大特点就是对每个章节的每个知识点全部配备例题，对每个涉及的函数均进行实例分析。

2.4.9 字符串对迭代器的支持

理解迭代器是理解 STL 的关键所在。模板库使算法独立于存储的数据类型，并且独立于使用的容器类型。STL 定义了 5 种迭代器，根据所需的迭代器类型对算法进行描述。这 5 种迭代器分别是输入迭代器、输出迭代器、正向迭代器、双向迭代器和随机访问迭代器。对于这 5 种迭代器都可以执行解除引用操作（*操作符），还可进行比较。本节主要讲述 basic_string 类（或 string 类）中迭代器的使用。

提示：本节是本书第一次介绍迭代器。后面章节还会有专门介绍。本节进行简要介绍。读者对迭代器有大概印象即可。

basic_string 类模板和 string 类提供了常规的迭代器和反向迭代器。string 是字符的有序群集。C++标准库为 string 提供了相应接口，便于将字符串当作 STL 容器使用。使用迭代器，可以遍历 string 内所有字符；当然也可以使用其他算法遍历 string 内的全部字符。能够访问字符串中的每个字符是进行排序、逆向重排、找出最大值等操作的前提。

string 类的迭代器是随机存取迭代器，即其支持随机存取。任何一个 STL 算法都可以与其搭配。通常 string 的“迭代器型别”由 string class 本身定义，通常可以被简单地定义为一般指针。对迭代器而言，如果发生重分配，或其所指值发生某些变化，迭代器会失效。

对于 string 类，和使用迭代器相关的成员函数有很多，主要包括：begin()，end()，rbegin()，rend()，构造器，append()，assign()，insert()，erase()，replace()等。下面通过例 2-13 说明迭代器在 string 类中的使用方法。

例 2-13

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
void main()
{
string s("The zip code of Hondelage in Germany is 38108.");
cout<<"Original: "<<s<<endl;
string sd(s.begin(),s.end());                              //构造函数中使用迭代器
cout<<"Destination: "<<sd<<endl;
transform(sd.begin(),sd.end(),sd.begin(),toupper);         //算法中使用迭代器（仿函数）
cout<<"Destination (All Toupper)): "<<sd<<endl;
string sd1;
sd1.append(sd.begin(),(sd.end()-7));                       //append()函数中使用迭代器
cout<<"Destination sd1: "<<sd1<<endl;
//.......
string sd2;
string::reverse_iterator iterA;
string temp="0";
for(iterA=sd.rbegin();iterA!=sd.rend();iterA++)            //reverse_iterator
{
```

```
    temp=*iterA;
     sd2.append(temp);
    }
    cout<<"Destination sd2: "<<sd2<<endl;
    sd2.erase(0,15);                                         //erase()函数中使用迭代器
    cout<<"Destination sd2(Erased 15 chars): "<<sd2<<endl;
    //...
    string::iterator iterB=sd2.begin();
    string sd3=string("12345678");
    sd2.insert(sd2.begin(),sd3.begin(),sd3.end());           //insert()函数中使用迭代器
    cout<<"Destination sd2(Insert 8 chars): "<<sd2<<endl;
    //..
    sd2.replace(sd2.begin(),sd2.end(),"This is an Example of Replace!");
    cout<<"Destination sd2(Replace All): "<<sd2<<endl;       // replace()函数中使用迭代器
    }
```

例 2-13 的执行效果见图 2-12。

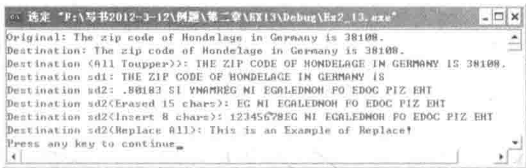


图 2-12　例 2-13 的执行效果

总结：上述内容对迭代器在 string 类及其相关成员函数中的使用进行了详细的描述。希望读者认真体会。

2.4.10　字符串对配置器的支持

对于 STL 的学习，首先应该理解配置器。配置器在 C++STL 中是非常重要的。在本节将结合 string 类对其进行简单介绍。使用 STL 必然涉及容器，而容器中存储了大量的数值，这必然需要分配内存空间。配置器的作用就是为容器开辟内存！

配置器最早是为了将内存模型抽象化而提出并开发的。所以内存配置器分配内存时是按该对象的个数，而不是按字节数，这区别于原来的 new[]和 new 操作符。另外，配置器最大的优点在于：配置器实现了将算法、容器与物理存储细节分隔。配置器可以提供一套分配与释放内存的标准方式，并提供用作指针类型和引用类型的标准名字。目前来看，配置器是一种纯粹的抽象，行为上像分配器的类型都可以看作是配置器。

C++STL 提供了标准分配器，目的是为用户提供更多的服务。basic_string 模板以及 string 类均提供了对常见配置器的相关支持。在 basic_string 类模板中包含 1 个配置器类型的成员：allocator_type。对于 string 对象，其可以作为配置器类的对象使用；对于 string 类而言，等价于 allocator<char>，即分配数据类型为 char 的内存，便于 string 类的对象存储 char 型字符。

同时 basic_string 类模板的第 3 个参数也是配置器模板参数。basic_string 类模板的形式如下：

```
template <class CharType,class Traits=char_traits<CharType>, class Allocator=allocator<
CharType> >class basic_string
```

而 string 类的声明形式如下：

```
typedef basic_string<char> string;
```

对于 basic_string 类模板，其第 1 个参数是 CharType，第 2 个参数和第 3 个参数的默认值和 charType 均相关。在声明 string 类时，参数 char 取代模板中的 CharType，string 即成为模板的实例，同时模板中的第 3 个参数成为“class Allocator=allocator <char>”，其意义为 string 中对象的内存类型为 char 型。

string 类还提供了 1 个和配置器相关的函数：get_allocator()，其函数原型为：

```
allocator_type  string::get_allocator() const
```

函数返回 string 类的内存模型对象，可以用于构造新的字符串。例 2-14 介绍了该函数的使用方法。

例 2-14

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;
void main()
{
string s("abcd");
basic_string<char> s1(s.get_allocator());          //定义容器 s1
basic_string<char>::allocator_type aT=s1.get_allocator();
string::size_type se=s1.size();
cout<<se<<endl;
}
```

由于在 string 类中，其成员 allocator 是保护成员，难以直接调用对内存的直接配置。本节仅简要介绍配置器的概念和使用方法。关于配置器的详细使用见后面相应的章节。

总结：对于配置器，一般情况下都是使用默认。对于本节内容，读者了解即可。

2.5 本章小结

本章内容较多，首先对字符串库进行了详细的介绍，之后介绍字符的特点。在 2.3 节，对字符串类模板进行了简要介绍。2.4 节对字符串模板类进行了深入研究，对其中包括的绝大部分内容进行了详细阐述。

字符串是任何程序员都必须大量使用的数据类型。对于绝大多数程序员来说，掌握字符串类的基础知识及其使用方法是必需的。

第 3 章 容器

容器是管理序列的类，通过容器类提供的成员函数，实现各种对序列中元素的操作，是一种高度的泛型抽象。部分算法也可以应用于容器序列的控制，关于算法的内容在第 4 章介绍。

STL 提供了各种容器模板类，通常这些模板类包括：向量（vector）、列表（list）、双队列（deque）、集合（set）、多重集合（multiset）、映射（map）、多重映射（multimap）。通常，向量（vector）可以认为是包含 1 个或 N 个更多元素的数组；列表（list）是由节点组成的双向链表，每个节点包含 1 个元素；双队列（deque）是包含 N 个连续的指向不同元素的指针组成的数组；集合（set）是由节点组成的，每个节点包含 1 个元素，节点之间以某种谓词排序；多重集合（multiset）是允许存在两个次序相等的元素集合；映射（map）是由{键，值}对组成的集合，同样以某种谓词排序，谓词和数据对有某种作用；多重映射（multimap）是允许键对包含相等次序的映射。

STL 在实现诸多容器类的同时，还实现了部分序列式容器的适配器（adapter）。容器的适配器是针对原有的基本容器的不足，对原有基本容器功能的补充。所有的适配器都不提供迭代器，元素访问通过专有的接口函数实现。

本章重点介绍各种容器的定义和使用方法。

提示：本书最大的特点就是，在介绍基础知识之后，对每个知识点全部配备了较翔实的例题，便于读者在阅读的同时，有机会动手尝试。本章例题凝练了作者众多心血，希望读者认真学习。

3.1 容器的概念

在汉语中，容器是指用来包装或装载物品的存储器（例如：箱、罐、坛）或成形或柔软不成形的包覆材料。在 C++语言中，STL 提供了大量的容器类。容器类的对象可以认为是容器。在编程实现过程中，可以认为“容器是用来存储和组织其他对象的对象”。容器适配器严格地说并不是容器，而是使用容器的对象，是在容器的基础上发展起来的。

3.1.1 容器成员和函数

STL 提供的每种容器，都提供了许多操作行为和成员。作为容器的成员，必须满足 3 个条件：

（1）元素必须是可复制的。所有容器均会产生 1 份元素副本，不会发生 alias 现象；所有容器操作行为传回的均是其元素的副本。这导致复制构造函数的执行非常频繁。

（2）元素必须是可指派（assign）的。容器的成员函数以及 STL 的各种算法（algorithms）均可利用 assign()函数为元素设定新值。

（3）元素必须是可释放的（经过析构函数释放内存）。使用者从容器中将元素删除时，容器必须释放其元素所占的内存。按这种需求，析构函数不能设置为 private 类型。

作为容器的成员函数（操作），也具有一些共同具备的能力。其中 3 个最重要的能力是：

（1）容器均能提供 value，而非 reference（引用）。涉及元素操作时，元素满足上述的 3 个条件。

（2）所有的元素自动形成顺序。即按此顺序可以多次遍历所有元素。这些容器均包含可返回迭代器的函数，使用这些迭代器可遍历元素。这是算法和容器的关键接口实现。

（3）函数使用者必须确保传递的参数符合要求。否则，可能会导致未定义的行为，通常 STL 不会抛出异常。

以上均为容器中成员函数的核心竞争力。所有的容器均包含部分共有的成员函数。主要包括：初始化、容器大小、比较、赋值和交换等。

3.1.2 容器的种类和数据结构

STL 容器通常分为 3 种：序列式容器，关联性容器，容器配接器。

（1）序列式容器（sequence 容器），容器中的元素是有序的，但未排序。包括：vector 动态数组、deque 双向队列、list 双向串行。

（2）关联性容器，容器中的元素都经过排序。包括：set，multiset，map，multimap 和 hash（哈希）table。

（3）容器配接器，是以某种 STL 容器作为底，修改其接口，具备各自的特点。包括：stack，queue，priority_queue。

STL 容器的数据结构也包括 3 种：string 字符串，bitset，valarray。

（1）string 字符串，其内保存字符。string 并不是真正的类，而是 1 个 basic_string 类的 typedef。其定义：

```
    template  <class  chart  ,  class  traits=char_traits<chart>,  class  Allocator=
allocator<chart>> class basic_string;
    typedef basic_string<char> string;
```

（2）bitset，其内保存 bits 的结构体。每个 bit 表示 1 个标志（flag）。其长度固定，长度便是模板的自变量。详见例 3-1。

例 3-1

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
void main()
{
        bitset<10> bs1(7);              //bitset 中长度为 10 个 bit; 初始化为十进制数 7.
        bitset<10> bs2(string("1000101011"));     //初始化为 10 个字符长度的字符串
        cout<<bs1<<endl;                          //输出
        cout<<bs2<<endl;                          //输出
        cin.get();                                //任意键退出
}
```

例 3-1 的执行效果见图 3-1。

在图 3-1 中，第 1 行字符恰好是十进制数 7 的二进制表现形式；第 2 行字符正好是字符串“1000101011”。

（3）valarray，是数学中的线性数列概念的呈现。其好处是可以像处理单一数值一样，对整个 valarray 中的每个元素实现运算。下面以例 3-2 为例进行介绍。

例 3-2

```
#include <iostream>
#include <valarray>
using namespace std;
template <typename T> void printValarray(const valarray<T>& va)
{
    for(int i=0;i<va.size();i++)
        cout<<va[i]<<"  ";
    cout<<endl;
}
void main()
{
       valarray<int> val(4);                    //包含 4 个元素，但未指定元素的值
       printValarray(val);                      //输出
       valarray<int>va2(3,4);                   //包含 4 个元素，其数值均为 3;
       printValarray(va2);
       int ia[]={1,2,3,4,5,6};                  //定义整数型数组
       valarray<int> va3(ia,sizeof(ia)/sizeof(ia[0]));//动态数组大小和数组 ia 的元素个数相同
       printValarray(va3);
       valarray<int>va4(ia+1,4);                //4 个元素,数值分别为数组 ia 的前 4 个元素加 1
       printValarray(va4);
       val=(va2+va4)*va4;                       //给 val 赋值
       printValarray(val);
}
```

例 3-2 的执行效果见图 3-2。

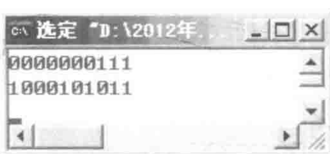


图 3-1　例 3-1 的执行效果

图 3-2　例 3-2 的执行效果

提示：在例 3-2 中，请读者注意 valarray 类型对象的使用方法。

3.2　序列式容器概述

对于基本容器，通过增加要求来实现对其的改进。序列是一种重要的改进，6 种 STL 容器（deque，list，queue，priority_queue，stack，vector）都是序列。序列最重要的特点就是可以在首端删除元素，在尾端添加元素。尤其是双向序列，允许在两端添加和删除元素。序列中包含至少一种迭代器，从而保证元素按特定的顺序排列，不会在两次迭代间发生变化。

数组和链表均是序列，但分支结构不是序列。序列中元素具有确定顺序，使用时可以将元素插入特定位置、删除特定区间等操作。标准库提供的基本序列包括：vector、list、deque；同时还包括3 个适配器：stack、queue、priority_queue。

vector、list、deque 3 种序列不可能互相实现，同时又不损失效率。另一方面，stack 和 queue 都可以在这 3 种基本序列的基础上高效实现。适配器的内容将在 3.4 节介绍。

vector、list、deque 3 种容器在使用时，需要分别包含相应的头文件。前面讲过，容器都是类模板，要定义或实现某种特定的容器对象，必须在容器名后加 1 对尖括号，括号中提供容器中存放元素的数据类型。

```
#include <vector>
#include <list>
#include <deque>

vector <string> svec;
list <int> ilist;
deque <long> litem;
```

本节主要讲述这 3 种序列式容器。下面分小节逐一阐述。

3.3 序列式容器——vector 类模板

vector 是定义于名称空间（namespace）std 内的模板，其定义在头文件<vector>中。

```
template <class T, class Allocator=allocator<T>> class vector;
```

vector 中的元素可以是任意型别 T，必须具备可设置、可复制两个属性。模板的第 2 个参数是关于空间配置器设置的，用于定义内存模型，默认内存模型是 C++标准库提供的 allocator，对于一般的程序员来说，可有可无。

vector 是最简单的序列式容器，支持随机访问元素。当然了，有时候显得效率低一些。vector 的大小与容量之间有重要差别。容量肯定是大于或等于其大小的。

vector 就像一个动态数组，是典型的“将元素置于动态数组中加以管理”的抽象概念。然而，在 C++标准中，并未规定必须以动态数组来实现 vector，仅规定了相应的条件和操作复杂度。在程序开发过程中，程序员使用 vector 作为动态数组是非常方便的。vector 的迭代器是随机存取迭代器，对任何一个 STL 算法均奏效。尤其在末端或删除元素时，vector 性能相当好。

vector 可以实现数据结构中的队列、数组和堆栈的所有功能。vector 一旦定义了类对象之后，就可以“装”东西了。

下面介绍 vector 类模板的使用。

3.3.1 vector 类基础

vector 类模板使用之前，需要定义该模板的对象，并对其初始化。vector 容器对象的大小和容量也是使用的重要前提，就像使用数组一样，数组的大小决定了数组能否正常使用。

1. vector 对象定义

定义 vector 类对象，主要是使用 std 名称空间的 vector 类模板，见例 3-3。

例 3-3

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{
        vector<string>myvt;                                  //定义类模板对象
        cout<<"OK!"<<endl;
}
```

在例 3-3 中，语句 vector<string>myvt ，使用类模板 vector 定义类模板的对象 myvt，容器 myvt 中保存元素的数据类型是 string。当然，也可以存储其他数据类型的元素，例如：int、long、double 等。

命名空间 std 是类的包容器（或者称作“更大的容器”）。上述程序在编译时，会出现 4 个警告信息（warning）,警告信息的编号为 c4786。为避免出现警告信息（warning），可以在代码中添加语句：#pragma warning (disable:4786)。新修改后的代码如下：

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{    vector<string>myvt;
     cout<<"OK!"<<endl;
     cout<<"成功消除编译警告信息!"<<endl;
}
```

第一行代码的作用是消除警告信息 c4786。

提示：注意上例中#pragma 语句的用法。

2．vector 类对象初始化

vector 类对象实现初始化操作可以使用函数 push_back()。下面在例 3-3 中添加部分代码，实现对 vector 类对象 myvt 的初始化。

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{    vector<string>myvt;
     myvt.push_back("1. Beijing City.");
     myvt.push_back("2. Tianjin City.");
     myvt.push_back("3. Shanghai City.");
     myvt.push_back("4. Chongqing City.");
     cout<<"OK!"<<endl;
     cout<<"成功消除编译警告信息!"<<endl;
```

```
    vector<string>::iterator it;
    for(it=myvt.begin();it!=myvt.end();it++)
      cout<<*it<<endl;
    cin.get();
}
```

上面代码执行之后，容器 myvt 中将包含 4 个字符串。函数 push_back()将对象放入容器中，有时会错误地把信息 push_back()到容器中，此时可以使用 pop_back()将其弹出。

既然 vector 类模板实例化的对象是容器，那么按“容器”的物理含义，对于既定的容器，必然存在容器的容量。vector 的对象也可以使用 reserve()函数预先设置容器的大小。例如：

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{   vector<string>myvt;
    myvt.reserve(4);
    myvt.push_back("1. Beijing City.");
    myvt.push_back("2. Tianjin City.");
    myvt.push_back("3. Shanghai City.");
    myvt.push_back("4. Chongqing City.");
    cout<<"OK!"<<endl;
    cout<<"成功消除编译警告信息!"<<endl;
    vector<string>::iterator it;
    for(it=myvt.begin();it!=myvt.end();it++)
        cout<<*it<<endl;
   cin.get();
}
```

例 3-3 的执行效果见图 3-3。

该语句可以实现设置容器的容量为 4 个字符串。一般情况下，定义 vector 类的对象后，如果没有设置容器的大小，是不允许直接给容器中的元素赋值的。即下面一段代码执行时，程序会发生异常。

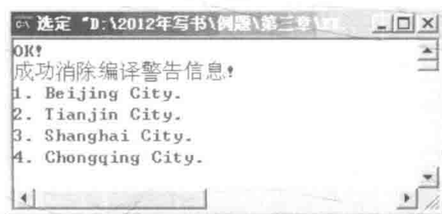


图 3-3　例 3-3 的执行效果

```
vector<string> myvt;
myvt[0]=1;
myvt[1]=2;
myvt.reserve(4);
```

3. 容器的大小和容量

vector 类模板定义了两个函数（size 和 capacity），实现统计容器元素的目的；还定义了两个函数（resize 和 reserve），实现设置容器大小的目的。

函数 size()和 capacity()可以统计容器中元素的数量。size()函数返回容器中现有的元素数量；capacity()函数返回容器中实际能够容纳的元素数量。函数 max_size()可以返回容器所能容纳的最大元素数量。一般情况下，元素数量超越 capacity()返回的数值时，vector 有必要重新配置内部存储器。

函数 reserve()可以预先设置容器的大小；函数 resize()可以修改容器的大小。函数 reserve()可以保留适当的容量，避免重新配置内存。当然，在定义容器对象时，通过传递函数的形式，也可以设置 vector 的起始大小。

以上知识点见例 3-3。

```
#pragma warning(disable:4786)
#include <iostream>
#include <string>
#include <vector>
using namespace std;
void main()
{   vector<string>myvt;
    myvt.reserve(4);
    cout<<"The size is 4."<<endl;
    myvt.push_back("1. Beijing City.");
    myvt.push_back("2. Tianjin City.");
    myvt.push_back("3. Shanghai City.");
    myvt.push_back("4. Chongqing City.");
    cout<<"OK!"<<endl;
    cout<<"成功消除编译警告信息!"<<endl;
    vector<string>::iterator it;
    for(it=myvt.begin();it!=myvt.end();it++)
        cout<<*it<<endl;
    int m=myvt.size();                              //获取容器的大小
    int n=myvt.capacity();                          //获取容器的容量
    int m1=myvt.max_size();                         //容器的最大容量
    cout<<"vector:myvt, size is "<<m<<endl;
    cout<<"vector:myvt, capacity is "<<n<<endl;
    cout<<"vector:myvt, maxsize is "<<m1<<endl;
    myvt.resize(10);                                //重置大小
    cout<<"resize: 10."<<endl;
    int n1=myvt.capacity();
    int n2=myvt.size();
    cout<<"vector:myvt, capacity is "<<n1<<endl;
    cout<<"vector:myvt, size is "<<n2<<endl;
    for(it=myvt.begin();it!=myvt.end();it++)        //输出向量的内容
    {    if(*it=="")
            cout<<"******"<<endl;
        cout<<*it<<endl;
    }
    cin.get();
}
```

完整的例 3-3 的执行效果见图 3-4。

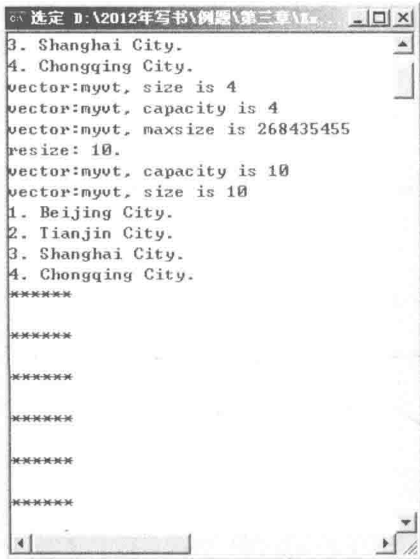


图 3-4　完整的例 3-3 的执行效果

提示：上例中请关注 size()，capacity()，reserve()，resize()函数的用法。

3.3.2　vector 类的成员函数

向量最基本的应用包括：判断向量是否为空；遍历向量元素；使用算法。下面逐一介绍。

1. 判断向量是否为空

向量模板 vector<T>提供了一个函数：empty()。此函数可以判断向量容器中元素是否为 0，如果向量为空，函数返回真；如果向量不为空，函数返回非真。函数 empty()的函数原型为：

```
template<class _TYPE, class _A> bool vector::empty() const;
```

下面举例说明 empty()的使用方法。在例 3-4 中，首先定义了结构体类型 ST，然后定义了初始化函数 Origin()，该函数可以根据指定的参数，实现对向量的初始化，为向量添加内容。

例 3-4

```
#include <iostream>
#include <vector>
using namespace std;
struct ST{                                              //定义结构体
int id;
double db;
};
void Origin(int num, vector<ST>& vt)                    //初始化函数
{ int m=num;
  ST temp;
  for(int i=0;i<m;i++)
  {    temp.id=i+1;
       temp.db=(i+1)*10;
       vt.push_back(temp);                              //初始化向量
  }
```

```
}
void main()
{  ST tmp;
   vector<ST> myvt;                              //定义向量
   Origin(5,myvt);                               //初始化向量
   int size=myvt.size();                         //统计向量中元素个数
   cout<<"size: "<<size<<endl;                   //输出向量个数
   while(!myvt.empty())                          //判断向量是否为空
   {   tmp=myvt.back();                          //如果非空，输出其末尾元素，并弹出
       cout<<"id "<<tmp.id<<", db: "<<tmp.db<<endl;
       myvt.pop_back();
   }
}
```

例 3-4 的执行效果见图 3-5（a）。

图 3-5（a）　例 3-4 的执行效果

请读者认真对照程序输出结果和源代码。

上面简单介绍了函数 empty()，顺便介绍函数 clear()。函数 clear()可以将容器中的所有元素移出，将容器清空。

提示：请关注上例中 empty()函数在 while 循环中的用法。

2．遍历 vector 型容器

遍历 vector 中的元素，必须使用循环语句 for 或者 while。可以使用迭代器方法实现遍历，也可以通过使用 at()函数和循环语句实现。下面在例 3-4 中添加两个函数 Iter_for()和 at_for()，代码如下：

```
void Iter_for(vector<ST>& vt)                    //使用迭代器遍历 vector 型容器
{ ST temp;
  vector<ST>::iterator iter;
  for(iter=vt.begin();iter!=vt.end();iter++)
  {    temp=*iter;
       cout<<"id:  "<<temp.id<<",  db:  "<<temp.db<<endl;
  }
}
void at_for(vector<ST>& vt)                      //使用函数 at()遍历 vector 型容器
{ ST temp;
  int i=0;
  int m=vt.size();
  for(i=0;i<m;i++)
  {    temp=vt.at(i);
       cout<<"id:  "<<temp.id<<",  db:  "<<temp.db<<endl;
  }
}
```

修改 main()函数如下：

```
#include <iostream>
#include <vector>
using namespace std;
struct ST{
int id;
double db;
};
void Origin(int num, vector<ST>& vt)
```

```
{ int m=num;
  ST temp;
  for(int i=0;i<m;i++)
  {    temp.id=i+1;
       temp.db=(i+1)*10;
       vt.push_back(temp);
  }
}
void Iter_for(vector<ST>& vt)                          //使用迭代器访问向量
{ ST temp;
  vector<ST>::iterator iter;
  for(iter=vt.begin();iter!=vt.end();iter++)
  {    temp=*iter;
       cout<<"id:  "<<temp.id<<",  db:  "<<temp.db<<endl;
  }
}
void at_for(vector<ST>& vt)                            //使用函数 at 访问向量
{ ST temp;
  int i=0;
  int m=vt.size();
  for(i=0;i<m;i++)
  {    temp=vt.at(i);
       cout<<"id:  "<<temp.id<<",  db:  "<<temp.db<<endl;
  }
}
void main()
{ ST tmp;
  vector<ST> myvt;
  Origin(5,myvt);
  int size=myvt.size();
  cout<<"size: "<<size<<endl;
  cout<<"Iterator output!"<<endl;
  Iter_for(myvt);
  cout<<"at() output!"<<endl;
  at_for(myvt);
  cout<<"empty() usage:"<<endl;
  while(!myvt.empty())
  {     tmp=myvt.back();
        cout<<"id "<<tmp.id<<", db: "<<tmp.db<<endl;
        myvt.pop_back();
  }
}
```

上述两种方法均较好地实现了对元素的遍历。

提示：请关注上例中如何遍历 vector 型中的元素。

使用例 3-4 遍历向量型容器的执行效果见图 3-5（b）。

3. 使用算法

使用 STL vector，还可以通过使用部分算法，实现对 vector 的操作。下面以 for_each()和 count()两个函数为例。在例 3-5 中，定义了一个模板函数 Original()，两个全局函数 out()和 greater95()。通

过例 3-5，既复习了函数模板的用法，还提前了解了算法的使用。

例 3-5

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Student{
  int id;
  double score;
};
template <class T> void Original(T& myvt)
{  Student temp;
   temp.id=1;
   temp.score=90;
   myvt.push_back(temp);
   temp.id=2;
   temp.score=95;
   myvt.push_back(temp);
   temp.id=3;
   temp.score=98;
   myvt.push_back(temp);
   temp.id=4;
   temp.score=97;
   myvt.push_back(temp);
   temp.id=5;
   temp.score=95;
   myvt.push_back(temp);
   temp.id=6;
   temp.score=90;
    myvt.push_back(temp);
}
void out(Student& stu)
{ cout<<"id: "<<stu.id<<" , score: "<<stu.score<<endl;              //输出信息
}
bool greater95(Student& stu)
{ if(stu.score>=95.0)                                               //如果大于 95，就返回"true"
          return 1;
  else
          return 0;
}
void main()
{   vector<Student> myvt;                                           //声明向量容器
   vector<Student>::iterator iter;                                  //声明迭代器
   int countV=0;
   Original(myvt);                                                  //初始化
   for_each(myvt.begin(),myvt.end(),out);                           //for_each 算法
   countV=count_if(myvt.begin(),myvt.end(),greater95);              //统计score>95.0的元素个数
   cout<<"The Number of the elements >95.0: "<<countV<<endl; //输出符合条件的元素个数
}
```

例 3-5 的执行效果见图 3-6。

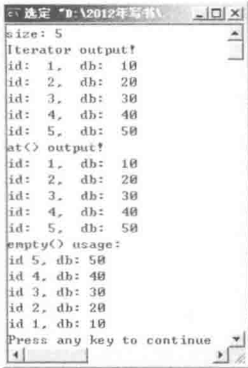


图 3-5（b） 使用例 3-4 遍历向量型容器的执行效果

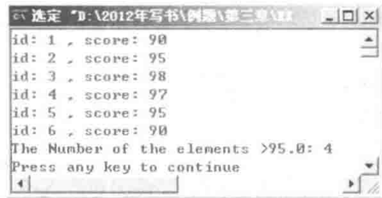


图 3-6 例 3-5 的执行效果

总结：请关注函数模板的使用，关注算法 for_each()的使用。最重要的是关注如何定义 for_each()的子进程。对于算法的子进程，后面还会有更多的介绍。读者不必急于掌握其原理，首先应该“照葫芦画瓢”学会使用方法。随着使用次数的增加，逐渐增加印象，逐渐深刻理解其内涵。

3.3.3 vector 高级编程

这里主要介绍 vector 容器相关的元素访问方法，迭代器相关函数，元素查找和搜索，vector 型容器中的字符串处理，元素排序，插入元素，删除元素，对象交换等内容。

1. 元素访问方法

按 C 和 C++的惯例，第一个元素的下标为 0；最后一个元素的索引下标为 size()-1，即第 n 个元素的下标为 n-1。可以直接访问 vector 型容器中元素的操作方法主要包括：at()，[]，front()，back()。

```
vector<typename T> c;
   c.at (index);
   c[index];
   c.front ( ) ;
   c.back();
```

其中，c.at(index) 函数返回值是引用。既可以取出元素值，也可以对元素赋值；c[index] 同样返回值是引用类型。既可以取出元素，也可以对元素赋值；必须确定下标有效。c.front() 返回第一个元素；c.back() 返回最后一个元素。

2. 迭代器相关函数

vector 类模板提供部分常规函数来获取迭代器。迭代器是随机访问迭代器，通过迭代器甚至可以操作所有算法。和迭代器相关的函数主要包括：begin()，end()，rbegin()，rend()。其中 begin()指向第一个元素；end()指向最后元素的下一个位置；rbegin()指向逆向迭代的第一个元素；rend()指向逆向迭代的最后元素的下一个位置。函数 begin()和 end()返回值类型均为迭代器类型；函数 rbegin()和 rend()返回值类型为逆向迭代器。

3. 元素查找和搜索

查找 vector 型容器中的元素，可以使用 STL 的通用算法 find()。有条件搜索相关元素可以使用算法 find_if()。这两个算法函数均可以使用迭代器，两个迭代器参数决定了搜索和查找的范围。函数返回值均为迭代器类型。

find()函数的原型为：

```
template<class InputIterator, class T> inline
  InputIterator find(InputIterator first, InputIterator last, const T& value)
```

find_if()函数的原型为：

```
template<class InputIterator, class T, class Predicate> inline
  InputIterator find_if(InputIterator first, InputIterator last, Predicate predicate)
```

先用例 3-6 说明如何实现在 vector 型容器中的元素查找和搜索。

例 3-6

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
void print(const int& temp)
{ cout<<temp<<endl;
}
void main()
{    const int ARRAY_SIZE = 8 ;
   int IntArray[ARRAY_SIZE] = { 1, 2, 3, 4, 4, 5, 6, 7 } ;
  vector<int> myvt;
  int* location_index=NULL;
  for(int i=0;i<8;i++)
     myvt.push_back(IntArray[i]);
  for_each(myvt.begin(),myvt.end(),print);
  location_index=find(myvt.begin(),myvt.end(),2);                     //find
  cout<<"数字 2 的下标是 :"<<(location_index-myvt.begin())<<endl;
  location_index=find_if(myvt.begin(),myvt.end(),bind2nd(greater<int>(),5));
  cout<<"第一个大于 5 的数字的下标是 :"<<(location_index-myvt.begin())<<endl;
}
```

例 3-6 的执行效果见图 3-7。

提示：掌握 find()函数和 find_if()函数的使用，尤其关注 find_if()函数中的条件表达式。

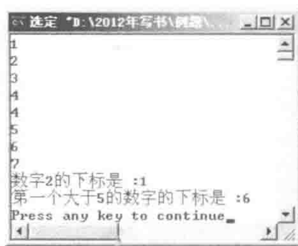


图 3-7　例 3-6 的执行效果

4. vector 型容器中的字符串处理

使用 vector 管理字符串，可以处理字符串中的每个字符，并且 vector 可以自动管理内存。字符串的每一个字符作为向量 vector 容器的元素，使用容器的所有成员函数可以方便完成字符串的操作。可参考本书第 2 章关于字符串的内容。

5. 元素排序

实现对 vector 容器中元素的排序需要使用算法 sort()或其他的排序算法。有些容器支持对特殊算法的实现，使用算法排序，有助于提高计算性能。但 vector 向量没有提供具有排序算法的成员函数。下面简要介绍使用 sort()算法实现对 vector 型容器中元素的排序，见例 3-7。

例 3-7

```
#include <iostream>
```

```
#include <vector>
#include <string>
#include <algorithm>
using namespace std;
class student{
public:
  student(const string &a,double b):name(a),score(b){}
  string name;
  double score;
  bool operator<(const student& m) const
  {    return score<m.score;
  }
};
bool name_sort_less(const student& m, const student& n)          //定义子进程
{ return m.name<n.name;
}
bool name_sort_greater(const student& m, const student& n)       //定义子进程
{ return m.name>n.name;
}
bool score_sort(const student& m, const student& n)              //定义子进程
{ return m.score>n.score;
}
void print(student& S)                                           //定义子进程
{ cout<<S.name<<"        "<<S.score<<endl;
}
void Original(vector<student>& V)                                //初始化
{ student st1("Tom", 74);
  V.push_back(st1);
  st1.name="Jimy";
  st1.score=56;
  V.push_back(st1);                                              //压入向量
  st1.name="Mary";
  st1.score=92;
  V.push_back(st1);
  st1.name="Jessy";
  st1.score=85;
  V.push_back(st1);
  st1.name="Jone";
  st1.score=56;
  V.push_back(st1);
  st1.name="Bush";
  st1.score=52;
  V.push_back(st1);
  st1.name="Winter";
  st1.score=77;
  V.push_back(st1);
  st1.name="Ander";
  st1.score=63;
  V.push_back(st1);
  st1.name="Lily";
  st1.score=76;
  V.push_back(st1);
```

```
  st1.name="Maryia";
  st1.score=89;
  V.push_back(st1);
 }
void main()
{
   vector<student> vect;
   Original(vect);
   cout<<"----Before sorted.-----"<<endl;
   for_each(vect.begin(),vect.end(),print);           //输出容器中的元素
   sort(vect.begin(),vect.end());                     //按 score 从小到大排序(使用 operator<)
   cout<<"----After sorted by score.-----"<<endl;
   for_each(vect.begin(),vect.end(),print);           //输出排序结果
   sort(vect.begin(),vect.end(),name_sort_less);      //按 name 从小到大排序
   cout<<"----After sorted by name.-----"<<endl;
   for_each(vect.begin(),vect.end(),print);           //输出排序结果
   sort(vect.begin(),vect.end(),score_sort);          //按 score 从大到小排序
   cout<<"----After sorted by name.-----"<<endl;
   for_each(vect.begin(),vect.end(),print);           //输出容器中的元素
   sort(vect.begin(),vect.end(),name_sort_greater); //按 name 从大到小排序
   cout<<"----After sorted by name.-----"<<endl;
   for_each(vect.begin(),vect.end(),print);           //输出排序结果
}
```

在例 3-7 中，重点讲述了如何使用算法 sort()实现对 vector 向量容器中的元素进行排序。在使用 sort()算法时，针对元素的特点，实现了 4 种排序：分别对 name 和 score 进行了从大到小和从小到大排序。

程序的执行结果如下：

```
----Before sorted.-----
Tom          74
Jimy         56
Mary         92
Jessy        85
Jone         56
Bush         52
Winter       77
Ander        63
Lily         76
Maryia       89
----After sorted by score.-----
Bush         52
Jimy         56
Jone         56
Ander        63
Tom          74
Lily         76
Winter       77
Jessy        85
Maryia       89
Mary         92
----After sorted by name.-----
```

```
Ander        63
Bush         52
Jessy        85
Jimy         56
Jone         56
Lily         76
Mary         92
Maryia       89
Tom          74
Winter       77
----After sorted by score.-----
Mary         92
Maryia       89
Jessy        85
Winter       77
Lily         76
Tom          74
Ander        63
Jimy         56
Jone         56
Bush         52
----After sorted by name.-----
Winter       77
Tom          74
Maryia       89
Mary         92
Lily         76
Jone         56
Jimy         56
Jessy        85
Bush         52
Ander        63
```

提示：掌握 sort()算法的使用，并学会定义其子进程。

6. 插入元素

在现有的vector型容器中插入或增加一个新的元素，可以使用push_back()函数将元素加入vector型对象（容器）的末尾。还可以使用 insert()函数将对象插入 vector 中任意位置，即 insert()函数可以实现在迭代器指向位置插入元素。insert()函数返回值为迭代器，指向刚插入的元素。还可以在迭代器指定位置插入多个连续的不同元素，还可以插入多个值相同的元素。函数 push_back()和函数 insert()的原型如下：

```
template<class _TYPE, class _A>void vector::push_back(const _TYPE& X); //push_back()模型

iterator insert(iterator it, const T& x = T());              //insert()函数的 3 个模型
void insert(iterator it, size_type n, const T& x);
void insert(iterator it,const_iterator first, const_iterator last);
```

下面通过例 3-8 说明函数 push_back()和 insert()的使用方法。

例 3-8

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
using namespace std;
void OutToScreen(int& Ele)                                    //输出元素
{cout<<Ele<<", ";
}
void main()
{vector<int> myvt;                                            //定义 vector 容器型对象
 for(int i=0;i<10;i++)
    myvt.push_back(i);                                        //初始化容器的元素
 myvt.insert(myvt.begin(),-1);                                //在向量起始位置之前插入元素-1
 for_each(myvt.begin(),myvt.end(),OutToScreen);               //将向量中的元素输出
 cout<<endl;
 myvt.insert(myvt.end(),-2);                                  //在向量末尾之前插入-2
 for_each(myvt.begin(),myvt.end(),OutToScreen);
 cout<<endl;
 vector<int> vt2;                                             //定义新的数值序列
 vt2.push_back(-8);
 vt2.push_back(-9);
 myvt.insert(myvt.end(),vt2.begin(),vt2.end());               //插入多个数值
 for_each(myvt.begin(),myvt.end(),OutToScreen);
 cout<<endl;
 myvt.insert(myvt.begin(),3,0);                               //插入多个相同的数值
 for_each(myvt.begin(),myvt.end(),OutToScreen);
 cout<<endl;
}
```

例 3-8 的执行效果见图 3-8。请对照源代码和执行结果深刻理解元素插入方法。

图 3-8 例 3-8 的执行效果

7. 删除元素

在定义 vector 型容器时，需要初始化其元素。并且定义 vector 时同时使用另外 STL 容器中的区间数据来初始化。该 STL 容器不必是 vector 型对象，只要元素的类型相同即可。例如：

```
vector<int> Harry;
Harry.push_back(1);
Harry.push_back(2);
vector<int>Bill(Harry.begin(), Harry.end());
```

删除容器中的元素，可以使用 3 个成员函数：pop_back()、erase()和 clear()。还可以使用算法库的算法 remove()来实现。

pop_back()函数删除最后 1 个元素，函数 erase () 可以删除由迭代器指定的元素，也可以删除区间范围的元素。clear()函数实现删除向量 vector 的所有元素，相当于 erase(begin(), end())。3 个函数的使用方法详见例 3-9。

例 3-9

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

```
using namespace std;
void OutToScreen(int& Ele)                                    //输出元素至屏幕
{cout<<Ele<<", ";
}
void main()
{vector<int> myvt;
 for(int i=0;i<10;i++)
    myvt.push_back(i);
 for_each(myvt.begin(),myvt.end(),OutToScreen);               //输出 vector 中的元素
 cout<<endl;
 cout<<"----------------------------"<<endl;
 while(!myvt.empty())                                         //判断向量是否为空
 {
 myvt.pop_back();                                             //弹出向量
 for_each(myvt.begin(),myvt.end(),OutToScreen);               //输出向量中的元素
 cout<<endl;
 }
 myvt.clear();                                                //清空向量型容器
 for(int j=0;j<10;j++)
    myvt.push_back(j);
 for_each(myvt.begin(),myvt.end(),OutToScreen);               //输出
    cout<<endl;
 cout<<"----------------------------"<<endl;
 while(!myvt.empty())                                         //判断向量是否为空
 {
 myvt.erase(myvt.begin());                                    //删除开始第一个元素
 for_each(myvt.begin(),myvt.end(),OutToScreen);               //输出向量容器中的元素
 cout<<endl;
 }
}
```

例 3-9 的执行效果见图 3-9。

8. 对象交换

vector 类模板还提供了成员函数 swap()，用于两个 vector 型容器之间的元素互换。如果两个参与交换的 vector 类型相同，对象交换会瞬间完成。如果两个参与交换的 vector 对象中的元素类型不同，在实现对象交换的过程中，需要执行非常复杂的操作。函数 swap()的原型如下：

```
template<class T, class A> void swap (const vector<T,A>&v1, const vector <T,A>&v2;
```

下面举例说明。

例 3-10

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void OutToScreenI(int& Ele)
{cout<<Ele<<",  ";
}
void main()
{vector<int> ci,cd;
for(int i=0;i<10;i++)
```

```
    {    ci.push_back(i);
         cd.push_back(i*3);
    }
    cout<<"vector-ci-below:"<<endl;
    for_each(ci.begin(),ci.end(),OutToScreenI);
    cout<<endl;
    cout<<"vector-cd-below:"<<endl;
    for_each(cd.begin(),cd.end(),OutToScreenI);
    cout<<endl;
    cout<<"---------swap---------"<<endl;
    ci.swap(cd);
    cout<<"vector-ci-below:"<<endl;
    for_each(ci.begin(),ci.end(),OutToScreenI);
    cout<<endl;
        cout<<"vector-cd-below:"<<endl;
    for_each(cd.begin(),cd.end(),OutToScreenI);
    cout<<endl;
}
```

例 3-10 的执行效果见图 3-10。

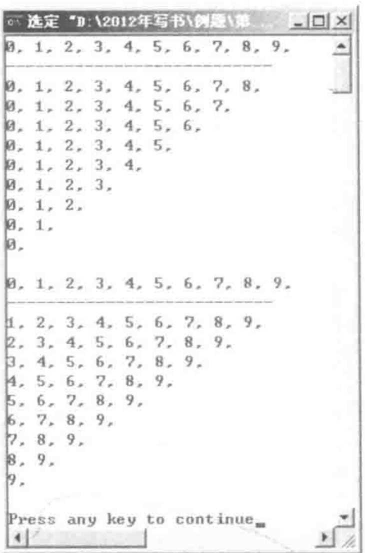


图 3-9　例 3-9 的执行效果

图 3-10　例 3-10 的执行效果

9．vector<bool>类

C++标准程序库专门针对元素类别为 bool 的 vector 设计了特殊版本，目的是获取优化的 vector。其所占用的内存空间远小于一般的 vector 模板实例化的 bool 型向量（容器）。普通的 vector 类模板会分配 1 个 byte 空间，而 vector<bool>只占用 1 个 bit 存储单个元素。所占内存空间小 8 倍。而 C++的最小寻址通常以 byte 为单位，在类 vector<bool>中需针对 reference 和 iterator 进行特殊考虑。类 vector<bool>的操作要比普通的 vector 慢很多，所有元素操作必须转换为 bit 操作。后面章节还会介绍 bitset 的使用，程序员应该优先使用 bitset，而不是 vector<bool>。

类 vector<bool>不仅仅是向量型的容器类，还提供位操作，非常方便地操作“位”和“标志”。所有用于元素存取的函数，返回值均为应用型（reference）。

vector<bool>类提供了位取反函数 flip()，既可以实现对容器中所有元素取反，还可以对某“位”

取反。下面举一简单例子进行说明，见例 3-11。

例 3-11

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void print(bool& Ele)
{
   cout<<Ele;
}
void main()
{
   int   X[]={1,0,1,0,0,0,1,1,1};
   vector<bool> vt;
   vector<bool>::iterator it;
   int i=0;
   for(i=0;i<9;i++)
   {
      vt.push_back(bool(X[i])) ;
   }
   for_each(vt.begin(),vt.end(),print);
   cout<<endl;
    vt[3]=bool(1);
    for_each(vt.begin(),vt.end(),print);
    cout<<endl;
}
```

例 3-11 的执行效果见图 3-11。

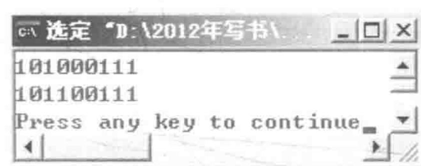


图 3-11　例 3-11 的执行效果

提示：本节主要讲述了 vector 模板类，主要讲述了 vector 模板类（容器）的定义和容量，基本成员函数使用方法和 vector 容器的高级编程。在学习使用 vector 容器的同时，讲述了部分算法的学习。

3.4　序列式容器——list 类模板

模板类 list 是一个容器，list 是由双向链表来实现的，每个节点存储 1 个元素。list 支持前后两种移动方向。list 和 vector 类似，提供了对元素的随机访问。list 的优势在于任何位置执行插入和删除动作都非常迅速，因为改变的仅仅是链接而已。在 list 中移动元素要比在 vector 和 deque 中快得多。list 模板类是定义于命名空间（namespace）std 中的，该类模板的声明形式为

```
template <class T, class Allocator=allocator<T>> class list;
```

使用 list 需要包含头文件<list>。前面已经讲过，任意型别 T 只要具备可设置性和可复制性，即可作为 list 元素。模板的第 2 个参数可有可无，适用于指定内存模型。在模板中也指定了默认的内存模型为 allocator。List 的内部结构和 vector 不同，存在较明显的区别。

（1）list 不支持随机存取。

（2）在 list 的任何位置执行元素的插入和移除都非常快，可以迅速实现。插入和删除动作不会

影响指向其他元素的指针、引用、迭代器，不会造成失效。

（3）list 不支持随机存取，不提供下标操作符和 at()函数。

（4）list 没有提供容量、空间重新分配等操作函数，每个元素都有自己的内存。

（5）list 也提供了特殊成员函数，专门用于移动元素。和同名的算法相比，速度更快。

list 模板类实现了标准 C++数据结构中链表的所有功能。list 一旦定义了类对象之后，就可以完成链表操作。

下面介绍 list 类模板的使用。

3.4.1　list 的定义和容量

1. list 的定义和构造函数

该模板类描述的对象控制是一个元素类型为 T 的可变长度序列。该序列以双向链表的方式存储，链表中每个元素都包含一个类型为 T 的成员。list 对象通过存储于其内部的一个类 A 的分配器对象进行存储空间的分配和释放。该分配器对象必须拥有和模板类 allocator 同样的外部接口。

在头文件 list 中定义了 4 种构造函数。

```
    explicit list(const _A& _Al = _A()): allocator(_Al),_Head(_Buynode()), _Size(0)
        {
        }
    explicit list(size_type _N, const _Ty& _V = _Ty(),const _A& _Al = _A()) : allocator(_Al),
_Head(_Buynode()), _Size(0)
        {
        insert(begin(), _N, _V);
        }
    list(const _Myt& _X): allocator(_X.allocator),_Head(_Buynode()), _Size(0)
        {
        insert(begin(), _X.begin(), _X.end());
        }
    list(const _Ty *_F, const _Ty *_L, const _A& _Al = _A()): allocator(_Al), _Head(_Buynode()),
_Size(0)
        {
        insert(begin(), _F, _L);
    }
```

以上 4 种构造函数中，第 1 个是默认的构造函数，表示要创建空 list 对象，根据以上构造函数，list 对象定义时一般有以下几种方法：

```
    list<A> listname;
    list<A> listname(size);
    list<A> listname(size,value);
    list<A> listname(elselist)
    list<A> listname(first, last)
```

以上几种定义 list 型对象的方法中，第 1 种形式最简单。第 1 种示例可以创建 1 个空的 list 对象，其中可以容纳类型为 A 的元素，其名称为 listname。例如：

```
    list<int> mylist
```

而第 2 种示例可以创建初始大小为 size 的 list 对象；第 3 种示例可以创建初始大小为 size，每个元素初始值为 value 的 list 对象；第 4 种示例用复制构造函数，从现有的 list 中创建新的 list 对象；

第 5 种方法创建 1 个 list 对象，并从其他 list 对象中复制由迭代器指定范围的多个元素。

2. 元素的赋值

list 模板类提供了两个成员函数 push_front()和 push_back()，用来把新的元素插入 list 对象中。push_front()用来在容器中的头部插入新元素，由于 vector 在头部插入新元素的效率非常低，所以只在 list 中存在该函数。还有另外两个函数 pop_front()和 pop_back()分别用于获取容器的头和容器的尾元素。这 4 个函数的原型分别为：

```
void push_front(const T& x);
void push_back(const T& x);
void pop_front();
void pop_back();
```

与 vector 类模板不同的是，vector 模板类只具有 push_back()和 pop_back()。而 list 模板类具有 4 个相应的函数，充分说明 list 是双向链表。

3. 容器的容量

list 对象作为容器，和其容量相关的成员函数包括：resize()，size()，max_size()。其函数原型分别为：

```
size_type size() const;
size_type max_size() const;
void resize(size_type n, T x = T());
```

函数 size()和 max_size()的返回类型均为 size_type 型，即 unsigned int 类型。size()用于返回 list 对象（容器）中元素的个数；max_size()用于返回 list 对象最大允许容量，一般是一个非常大的整数。函数 resize()用于重新调整 list 对象的大小。

4. 迭代器相关

list 模板类包含和迭代器相关的函数主要有：begin()，front()，rbegin()，end()，back()，rend()。这几个函数的原型分别为：

```
const_iterator begin() const;
iterator begin();
reference front();
const_reference front() const;
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
const_iterator end() const;
iterator end();
reference back();
const_reference back() const;
const_reverse_iterator rend() const;
reverse_iterator rend();
```

下面以例 3-12 说明如何使用以上 4 个知识点。通过实例学习，希望读者对 list 型容器的最基本知识有所了解，理解 list 的定义、容量、赋值方法以及该容器中和迭代器相关的成员函数。

例 3-12

```
#pragma warning(disable:4786)
#include <iostream>
```

```
#include <list>
#include <algorithm>
#include <string>
//#include <iomanip.h>
using namespace std;
template <class T>void print(const T& Ele)
{    cout<<" "<<Ele<<";"<<endl;
}
void Print_D(double& Ele)                                    //格式化输出
{    cout.width(5);                                          //格式化输出，宽度5个字符，
     cout.precision(1);                                      //保留1位小数点
     cout<<std::fixed<<Ele<<", ";                            //以定点数形式输出
}
void Print_I(int& Ele)
{    cout<<Ele<<",  ";
}
```

以上代码请关注第 1 行和 3 个不同的 print()函数的使用方法。

```
void main()
{    list<string>mylist_string;
     list<double>mylist_double(6);
     //上述是2种list型容器的定义形式
     //---------初始化mylist_string
     mylist_string.push_front("1:  Jack");
     mylist_string.push_front("2:  Tom");
     mylist_string.push_front("3:  Mike");
     //---------初始化mylist_double
     mylist_double.push_front(10.0);
     mylist_double.push_front(20.0);
     mylist_double.push_front(30.0);
     mylist_double.push_front(40.0);
     mylist_double.push_front(50.0);
     //下述是三种容器定义形式
     list<int> mylist_int(6,0);
     list<double>mylist_double2(6,0.0);
     list<int>elselist(mylist_int);
     list<double>Iterlist(mylist_double.begin(),mylist_double.end());
     //----输出各个容器中的元素
     cout<<"the string list: mylist_string is below:"<<endl;
     list<string>::iterator iter_String;
     for(iter_String=mylist_string.begin();iter_String!=mylist_string.end();iter
_String++)
     {    string temp=*iter_String;
          print(temp);
     }
     cout<<"the double list- mylist_double is below:"<<endl;
     for_each(mylist_double.begin(),mylist_double.end(),Print_D);
     cout<<endl;
     cout<<"the double list- mylist_double2 is below:"<<endl;
     for_each(mylist_double2.begin(),mylist_double2.end(),Print_D);
     cout<<endl;
     cout<<"the double list- Iterlist is below:"<<endl;
```

```
        for_each(Iterlist.begin(),Iterlist.end(),Print_D);
        cout<<endl;
        cout<<"the int list- mylist_int is below:"<<endl;
        for_each(mylist_int.begin(),mylist_int.end(),Print_I);
        cout<<endl;
        cout<<"the int list- elselist is below:"<<endl;
        for_each(elselist.begin(),elselist.end(),Print_I);
        cout<<endl;
        //各容器的容量
        int size=mylist_string.size();
        int maxsize=mylist_string.size();
        mylist_string.resize(5);
        size=mylist_double.size();
        maxsize=mylist_double.max_size();
        mylist_double.resize(5);
        size=mylist_double2.size();
        maxsize=mylist_double2.max_size();
        mylist_double2.resize(5);
        size=Iterlist.size();
        maxsize=Iterlist.max_size();
        Iterlist.resize(5);
        size=mylist_int.size();
        maxsize=mylist_int.max_size();
        mylist_int.resize(5);
        size=elselist.size();
        maxsize=elselist.max_size();
        elselist.resize(5);
        //----再次输出各个容器中的元素
        cout<<"the string list: mylist_string is below:"<<endl;
        for(iter_String=mylist_string.begin();iter_String!=mylist_string.end();iter
_String++)
        {     string temp=*iter_String;
              print(temp);
        }
        cout<<"the double list- mylist_double is below:"<<endl;
        for_each(mylist_double.begin(),mylist_double.end(),Print_D);
        cout<<endl;
        cout<<"the double list- mylist_double2 is below:"<<endl;
        for_each(mylist_double2.begin(),mylist_double2.end(),Print_D);
        cout<<endl;
        cout<<"the double list- Iterlist is below:"<<endl;
        for_each(Iterlist.begin(),Iterlist.end(),Print_D);
        cout<<endl;
        cout<<"the int list- mylist_int is below:"<<endl;
        for_each(mylist_int.begin(),mylist_int.end(),Print_I);
        cout<<endl;
        cout<<"the int list- elselist is below:"<<endl;
        for_each(elselist.begin(),elselist.end(),Print_I);
        cout<<endl;
        //使用迭代器相关的函数
        list<double>::iterator Iter_D;
        list<double>::reverse_iterator Iter_rD;
```

```
    double tmp=0.0;
    Iter_D=mylist_double.begin();
    tmp=*Iter_D;
    cout<<"The beginning element of the mylist_double:"<<endl;
    cout<<tmp<<endl;
    Iter_rD=mylist_double.rbegin();
    tmp=*Iter_rD;
    cout<<"The reverse beginning element of the mylist_double:"<<endl;
    cout<<tmp<<endl;
    Iter_D=mylist_double.end();
    tmp=*Iter_D;
    cout<<"The ending element of the mylist_double:"<<endl;
    cout<<cout.scientific<<tmp<<endl;
    Iter_rD=mylist_double.rend();//
    tmp=*Iter_rD;
    cout<<"The reverse ending element of the mylist_double:"<<endl;
    cout<<tmp<<endl; //
    tmp=mylist_double.front();
    cout<<"The front element of the mylist_double:"<<endl;
    cout<<tmp<<endl; //
    tmp=mylist_double.back();
    cout<<"The back element of the mylist_double:"<<endl;
    cout<<tmp<<endl;
}
```

在例 3-12 的源代码中，请关注中文注释。并且在例题中还使用了模板函数和仿函数，使用了 for_each 算法，还对容器 list 的大小进行重新设置。为防止程序编译时显示过多的警告信息，在程序起始位置添加语句：#pragma warning(disable:4786)。读者应对照源代码和程序输出，认真理解。

例 3-12 的执行效果见图 3-12。

图 3-12　例 3-12 的执行效果

提示：

1. 在例 3-12 中，由于函数 end()所指向的元素分别是容器中最后一个元素，程序中输出 end()返回的数值时，并不是最后一个元素的数值。函数 rend()同理。由此可知，在使用迭代器进行 for 循环时，一般循环终止条件是(iterator!=end())，即循环至 end()返回值指向的位置时（恰好是容器中最后一个元素的后面），停止循环。

2. 在函数 Print_D(double& Ele)中，使用了 cout 的格式化输出。请读者关注。

3.4.2 list 容器基础成员函数

list 容器最基础的编程内容包括：判断 list 内容是否为空，元素存取，元素重置，交换两个容器内容，list 序列中元素的插入和删除等。

1. 判断容器是否为空

如果 list 型容器为空，成员函数 empty()返回 true。函数 empty()的原型为：

```
bool empty() const;
```

下面用例 3-13 加以说明。

例 3-13

```
#include <iostream>
#include <list>
using namespace std;
void main()
{    list<double>mylist;                                    //创建 list 容器
     mylist.push_back(10.2);                                //压入 1 个数值
     bool empty=0;
     if(mylist.empty())                                     //如果容器为空，
         cout<<"The list is empty!"<<endl;                  //输出信息
     else                                                   //否则
     {    empty=mylist.empty();
          cout<<mylist.front()<<", "<<empty<<endl; //输出容器中的元素和函数 empty()的返回值
     }
}
```

从例 3-13 可知，使用 empty()还是比较方便的。

2. list 容器中元素的存取和访问

list 型容器不提供成员函数 at()和操作符 operator[]，对容器中的元素访问有些不便。值得庆幸的是，还可以使用迭代器进行元素的访问，见例 3-14。

例 3-14

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(double& Ele)
{    cout<<Ele<<" ,";
}
void main()
{    list<double>mylist;
     mylist.push_back(11.1);
```

```
    mylist.push_back(21.5);
    mylist.push_back(31.6);
    mylist.push_back(41.7);
    int count=mylist.size();
    for_each(mylist.begin(),mylist.end(),print);
    cout<<endl;
    list<double>::iterator Iter_S;
    Iter_S=mylist.begin();
    cout<<"The third element is "<<*(++(++(++Iter_S)))<<endl;
}
```

例 3-14 的执行效果图见图 3-13。

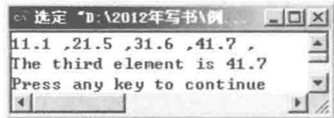


图 3-13　例 3-14 的执行效果

3. 元素重置

在 list 型容器中提供了可以重置元素值的成员函数 assign()。使用函数 assign()可以修改容器中任意元素的数值，甚至可以修改多个连续元素的数值。assign()函数的原型如下：

```
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
```

下面举例说明其使用方法，见例 3-15。

例 3-15

```
#include <iostream>
#include <list>
using namespace std;
//void print(list<double>& mylist)
template<class T>void print(list<T>& mylist)
{   list<T>::iterator Iter;
    mylist.reverse();
  for(Iter=mylist.begin();Iter!=mylist.end();Iter++)
  {  cout<<*Iter<<",  ";
  }
  cout<<endl;
}
void main()
{   list<double> list_One,list_Two, list_Three;
    double Ele=0.0;
    for(int i=0;i<10;i++)
    { Ele=i+i/10.0;
      list_One.push_front(Ele);
    }
    print(list_One);
    list_Two.assign(5,5.6);
    print(list_Two);
    list_Three.assign(list_One.begin(),list_One.end());
    print(list_Three);
}
```

在例 3-15 中，针对 assign()函数的两种用法均举出了实例。在需要初始化 list 型容器或者复制 list 型容器中的元素时，使用 assign()是非常方便的。例 3-15 的执行效果见图 3-14。

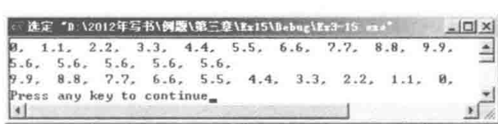


图 3-14　例 3-15 的执行效果

提示：学习函数 assign()使用方法的同时，请关注模板函数 print()。

4．交换两个 list 容器的内容

模板类 list 提供了成员函数 swap()。可以实现两个 list 型容器（对象）的内容交换。通过 swap()函数完成两个 list 容器中元素交换的同时，容器的大小也发生了变化。在 STL 的算法库中，也提供了 swap()函数。作为通用算法同样可以实现交换两个 list 容器中的内容。在例 3-15 的基础上，进行适当的修改，添加容器交换的内容见例 3-16。

例 3-16

```
#include <iostream>
#include <list>
using namespace std;
template<class T>void print(list<T>& mylist)
{   list<T>::iterator Iter;
    mylist.reverse();
   for(Iter=mylist.begin();Iter!=mylist.end();Iter++)
   {   cout<<*Iter<<", ";
   }
   cout<<endl;
}
void main()
{   list<double> list_One,list_Two, list_Three;
    double Ele=0.0;
    for(int i=0;i<10;i++)
    { Ele=i+i/10.0;
       list_One.push_front(Ele);
    }
    cout<<"List_One 输出:"<<endl;
    print(list_One);
    list_Two.assign(5,5.6);
    cout<<"List_Two 输出:"<<endl;
    print(list_Two);
    list_Three.assign(list_One.begin(),list_One.end());
    cout<<"list_Three 输出:"<<endl;
    print(list_Three);
    list_One.swap(list_Two);         //对 list_One 和 list_Two 中的元素进行交换
    cout<<"list_one 和 list_Two 交换之后: "<<endl;
    cout<<"List_One 输出:"<<endl;
    print(list_One);
    cout<<"List_Two 输出:"<<endl;
    print(list_Two);
    swap(list_One,list_Three);       //使用 swap ( ) 的算法形式进行两个 list 型容器的内容交换
    cout<<"list_one 和 list_Three 交换之后: "<<endl;
    cout<<"List_One 输出:"<<endl;
    print(list_One);
    cout<<"list_Three 输出:"<<endl;
    print(list_Three);
}
```

例 3-16 的执行效果见图 3-15。

5．元素的插入和删除

模板 list 和其他型容器一样，提供了丰富而灵活的插入和删除元素操作。尤其 list 型容器提供了

较完善的插入和删除操作。list 型容器可以在序列的开头和队尾灵活地插入和删除元素。涉及的成员函数主要包括：push_back()，push_front()，pop_back()，pop_front()，insert()，erase()和 clear()。push_front()和 push_back()均可以轻松地插入元素至序列中，pop_back()和 pop_front()亦可以实现轻松地从首端或尾端将元素从序列中移除。这 4 个函数在前面章节已有涉及，本小节不再详细介绍这 4 个函数的使用。本小节重点介绍函数 insert()、erase()和 clear()。同样可以实现元素删除的成员函数 remove()和 remove_if()在后面介绍。

成员函数 insert()的原型为以下 3 种形式：

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
void insert(iterator it, const_iterator first, const_iterator last);
```

第 1 种形式的作用在于把某个元素插入指定位置；第 2 种形式的目的是把某个具体值的多个备份插入 list 中迭代器所指起始位置；第 3 种形式主要用于把一个指定范围的多个元素插入 list 中迭代器所指向的范围中。函数的使用方法和 vector 型容器的同名成员函数近似。

成员函数 erase()的原型为以下 2 种形式：

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

同样，第 1 种形式用于删除迭代器指向的元素；第 2 种形式用于删除迭代器所定义的范围。函数的使用方法和 vector 型容器的同名成员函数近似。

成员函数 clear()相当于使用 erase()删除序列中的所有元素，即 erase(begin(), end())。修改例 3-9，使之适用于 list 型容器，注意代码中的加黑部分，见例 3-17。

例 3-17

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void OutToScreen(int& Ele)
{   cout<<Ele<<", ";
}
void main()
{   list<int> mylt;                                         //定义容器型对象
    for(int i=0;i<10;i++)
        mylt.push_back(i);                                  //添加元素
    for_each(mylt.begin(),mylt.end(),OutToScreen);          //输出容器中的元素
    cout<<endl;
    cout<<"----------------------------"<<endl;
    while(!mylt.empty())                         //每次从尾部取出一个元素，之后输出，直至为“空”
    {mylt.pop_back();
    for_each(mylt.begin(),mylt.end(),OutToScreen);
    cout<<endl;
    }
    mylt.clear();                                           //清空容器
    for(int j=0;j<10;j++)                                   //重新添加元素
        mylt.push_back(j);
    for_each(mylt.begin(),mylt.end(),OutToScreen);          //输出容器中所有元素
    cout<<endl;
    cout<<"----------------------------"<<endl;
```

```
    while(!mylt.empty())                    //删除容器中第一个元素，输出剩余元素，直至为空
    {mylt.erase(mylt.begin());
    for_each(mylt.begin(),mylt.end(),OutToScreen);
    cout<<endl;
    }
}
```

例 3-17 的执行效果见图 3-16。

图 3-15　例 3-16 的执行效果

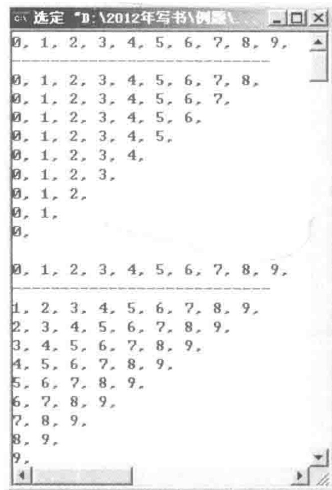


图 3-16　例 3-17 的执行效果

总结：在例 3-17 中，在例 3-9 的基础上仅修改头文件<vector>为<list>，将容器的定义语句 vector<int>myvt 修改为 list<int>mylt。请关注下述代码中的黑体字。并重点体会 pop_front()、pop_back()、erase()和 clear()的使用方法。

3.4.3　运算符函数

同样，list 型容器也提供了大量的模板函数，即提供了大量运算符函数。常见的包括：operator==，operator<，operator!=，operator<=，operator>，operator>=等。下面分别详细介绍。

1．operator= =

operator[]的基本使用方法主要是读取 list 中的元素，而运算符 operator= =则是判断语句，用于判断两个 list 容器是否相等。如果相同，返回 true。其原型如下：

```
bool  operator==(const  list<Type,  Allocator>&  _Left,  const  list<Type,  Allocator>&
_Right);
```

上式表明，参与比较的两个 list 对象的格式应该完全一样，不能使用两个类型不同的容器进行比较。

2．operator <

运算符 operator<依然是用于判断。它用来判断两个 list 容器是否“前者小于后者”，如果“前者小于后者”，返回 true；否则返回 false。其原型如下：

```
bool operator<(const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

其使用方法和 operator= =是相同的。

3．operator!=

运算符 operator!=用于判断两个 list 型容器是否“不相等”。如果两者不同，返回 true；否则返回 false。其原型如下：

```
bool operator!=(const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

4．operator <=

运算符 operator<=用于判断两个 list 型容器是否“前者小于或等于后者”。如果“前者小于或等于后者”，返回 true；否则返回 false。其原型如下：

```
bool operator<=(const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

5．operator>

运算符 operator>用于判断两个 list 型容器是否“前者大于后者”，如果“前者大于后者”，返回 true；否则返回 false。

```
bool operator> (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

6．operator >=

运算符 operator>=用于判断两个 list 型容器是否“前者大于或等于后者”。如果“前者大于后者”，运算符返回 true，否则返回 false。其语法如下：

```
bool operator>= (const list<Type, Allocator>& _Left, const list<Type, Allocator>& _Right);
```

下面使用例 3-18 说明以上 6 个运算符的使用方法。

例 3-18

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int& Ele)
{  cout <<Ele<<" ";
}
void main()
{  list<int> L1,L2;
   L1.push_back(1);
   L1.push_back(2);
   L2.assign(L1.begin(),L1.end());
   cout<<"list L1: ";
   for_each(L1.begin(),L1.end(),print);
   cout<<endl;
   cout<<"list L2: ";
   for_each(L2.begin(),L2.end(),print);
   cout<<endl;
   if(L1==L2)
      cout<<"L1 and L2 are equal!"<<endl;
   L2.push_back(3);
   L1.push_back(1);
   cout<<"list L1: ";
```

```
    for_each(L1.begin(),L1.end(),print);
    cout<<endl;
    cout<<"list L2: ";
    for_each(L2.begin(),L2.end(),print);
    cout<<endl;
    if(L1<L2)
        cout<<"L1 is less than L2."<<endl;
    else if(L1>L2)
        cout<<"L1 is greater than L2."<<endl;
    if(L1!=L2)
        cout<<"L1 is not equal to L2."<<endl;
}
```

例 3-18 的执行效果见图 3-17。

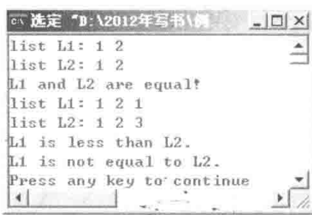


图 3-17　例 3-18 的执行效果

3.4.4　其他重要成员函数

list 型容器具有一些特殊函数：函数 merge()，remove()，remove_if()，sort()，splice()，unique()。

1．函数 merge()和函数 sort()

函数 merge()可以实现将两个 list 型对象合并成一个 list 对象。函数的功能在于把原型中的 list 容器对象作为函数参数，插入目标 list（函数的调用者）中。合并之后的容器中元素是按从小到大升序排列的。函数 merge()的原型为：

```
void merge(list& x);
void merge(list& x, greater<T> pr);
```

list 型容器还提供了具有排序功能的成员函数：sort()，用于对 list 容器中元素进行排序。函数 sort()默认排序方式是从小到大。函数原型为：

```
void sort()
void sort(greater<T>pr)
```

两个函数的具体使用方法参见例 3-19。

例 3-19

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int& Ele)
{   cout<<Ele<<" ";
}
void main()
{   list<int> L1,L2,L3;
    list<int>::iterator I1,I2,I3;
```

```
    L1.push_back(1);
    L1.push_back(5);
    L2.push_back(2);
    L2.push_back(3);
    L3.push_back(7);
    L3.push_back(8);
    cout<<"L1 : ";
    for_each(L1.begin(),L1.end(),print);     //输出 L1 中的内容
    cout<<endl;
    cout<<"L2 : ";
    for_each(L2.begin(),L2.end(),print);     //输出 L2 中的内容
    cout<<endl;
    cout<<"L3 : ";
    for_each(L3.begin(),L3.end(),print);     //输出 L3 中的内容
    cout<<endl;
    cout<<"L1 merges L2 and L3 :";
    L1.merge(L2);                            //合并 L1 和 L2
    L1.merge(L3);                            //合并 L1 和 L3
    for_each(L1.begin(),L1.end(),print); //可知，在 list 合并之后，所有元素自动按从小到大排序
    cout<<endl;
    L1.sort(greater<int>());                 //降序排序
    cout<<"L1 : ";
    for_each(L1.begin(),L1.end(),print);     //所有元素输出（降序）
    cout<<endl;
    L1.sort();                               //默认按从小到大排序
    cout<<"L1 : ";
    for_each(L1.begin(),L1.end(),print);     //所有元素自动按从大到小排序
    cout<<endl;
}
```

在上述代码中，两个 list 型容器合并之后，新容器中的元素是自动排序的。语句为：

```
L1.sort(greater<int>())
```

此语句可以实现对容器中的元素降序排列。函数的参数“greater<int>()”是 STL 中的预定义仿函数。容器类 list 的成员函数 sort()默认排序方式是升序。

例 3-19 的执行效果见图 3-18。

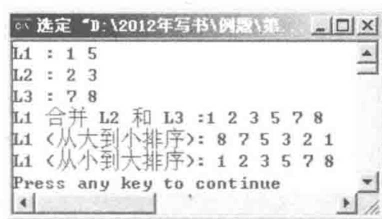


图 3-18　例 3-19 的执行效果

2. 函数 remove()和函数 remove_if()

删除 list 中的对象可以使用多个成员函数：pop_back，pop_front，erase，clear 等常见函数。在 list 中提供了 remove()函数，此函数的原型如下：

```
void remove(const Type&  _Val);
```

函数可以实现删除 list 容器中某个具体的元素。使用 remove()函数不需要指定具体位置，只要告诉需要删除的元素的值，就可以直接删除所有相应的元素。这是它和其他删除函数不同的地方。值得注意的是，使用函数 remove()并会不改变容器中元素的顺序。同时，由于在使用 remove()时，会删除掉所有等于参数（指定数值_Val）的元素，程序员在写程序时，要谨慎使用。

函数 remove_if()是有条件地删除所有等于参数值的 list 容器中的元素。函数 remove_if()的原型如下：

```
template <class Pred> void remove_if(Pred pr)
```

函数 remove_if()仅删除满足条件“Pred pr”的相应元素。函数的参数“Pred pr”是一元谓词。在 Visual C++6.0 中，Pred 的定义如下：

```
typedef binder2nd<not_equal_to<_Ty> > Pred;
```

如果 remove_if()函数只有当 Pred pr 为 true 时，才能正确执行删除操作。那么在 visual C++6.0 编译环境下，函数执行结果是删除和指定参数 pr 不相等的所有元素。参见例 3-20，学习两个“删除”函数的用法。

例 3-20

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int& Ele)
{    cout<<Ele<<",  ";
}
bool is_Even(int & Ele)
{    return (Ele%2==1);
}
void Origin(list<int>& L, int num)
{    int temp;
     L.clear();
     for(int i=0;i<num;i++)
     {    temp=i+1;
          L.push_back(temp);
     }
     for_each(L.begin(),L.end(),print);
     cout<<endl;
}
void main()
{    list<int> L1;
     Origin(L1,9);
     int temp;
     temp=9;
     L1.push_back(temp);
     temp=8;
     L1.push_back(temp);
     cout<<"Ouput the list \'L1\':"<<endl;
     for_each(L1.begin(),L1.end(),print);
     cout<<endl;
     L1.remove(9);
     for_each(L1.begin(),L1.end(),print);
     cout<<endl;
     L1.remove_if(bind2nd(not_equal_to<int>(),1));
     for_each(L1.begin(),L1.end(),print);
     cout<<endl;
}
```

例 3-20 的执行效果见图 3-19。

提示：学习函数 remove()函数和 remove_if()函数的同时，请详读这两个函数的功能。最重要的是关注成员函数 remove_if()的使用。这不同于算法 remove_if()的使用，算法 remove_if()在使用时，要求更宽泛一些。

3．函数 splice()

前面讲述了容器合并成员函数 merge()。函数 merge()并不是非常灵活，它具有一定的局限性。容器 list 提供了另一个函数 splice()，函数 splice()的语法原型如下：

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first, iterator last);
```

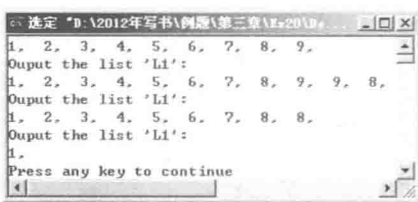
上述 3 种形式既可以把 list 型对象 x 插入迭代器指针 it 指定的位置后面；也可以将 x 中的某个元素（first 指向的元素）插入 it 后面；最后一种形式可以将 x 中某个范围（first，last）内的元素插入 it 后面。值得注意的是，一旦合并完成，参数 x 中会减少相应数目的元素。这些元素已经转移走了。尤其是第一种形式，一旦合并函数 splice()执行成功之后，参数 x 中将不包含任何元素，是空容器。修改例 3-19，使其用于函数 splice 的学习，见例 3-21。

例 3-21

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int& Ele)
{   cout<<Ele<<" ";
}
void main()
{   list<int> L1,L2,L3,L0;
    //list<int>::iterator I1,I2,I3;
    L1.push_back(1);
    L1.push_back(5);
    L2.push_back(2);
    L2.push_back(3);
    L3.push_back(7);
    L3.push_back(8);
    L0.push_back(9);
    L0.push_back(-1);
    cout<<"L1 : ";
    for_each(L1.begin(),L1.end(),print);
    cout<<endl;
    cout<<"L2 : ";
    for_each(L2.begin(),L2.end(),print);
    cout<<endl;
    cout<<"L3 : ";
    for_each(L3.begin(),L3.end(),print);
    cout<<endl;
    cout<<"L0 : ";
    for_each(L0.begin(),L0.end(),print);
    cout<<endl;
    cout<<"L1 合并 L2:";
    L1.splice(L1.end(),L2);
```

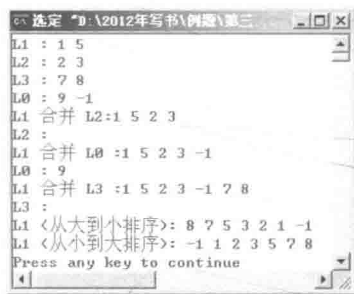
```
    for_each(L1.begin(),L1.end(),print);
    cout<<endl;
    cout<<"L2 : ";
    for_each(L2.begin(),L2.end(),print);
    cout<<endl;
    cout<<"L1 合并 L0 :";
    L1.splice(L1.end(),L0,(++L0.begin()));
    for_each(L1.begin(),L1.end(),print);
    cout<<endl;
    cout<<"L0 : ";
    for_each(L0.begin(),L0.end(),print);
    cout<<endl;
    cout<<"L1 合并 L3 :";
    L1.splice(L1.end(),L3,L3.begin(),L3.end());
    for_each(L1.begin(),L1.end(),print); //可知，在 list 合并之后，所有元素自动按从小到大排序
    cout<<endl;
    cout<<"L3 : ";
    for_each(L3.begin(),L3.end(),print);
    cout<<endl;
    L1.sort(greater<int>());                  //所有元素自动按从大到小排序
    cout<<"L1 (从大到小排序): ";
    for_each(L1.begin(),L1.end(),print);
    cout<<endl;
    L1.sort();                                //默认按从小到大排序
    cout<<"L1 (从小到大排序): ";
    for_each(L1.begin(),L1.end(),print);     //所有元素自动按从大到小排序
    cout<<endl;
}
```

例 3-21 的执行效果见图 3-20。



```
1, 2, 3, 4, 5, 6, 7, 8, 9,
Ouput the list 'L1':
1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8,
Ouput the list 'L1':
1, 2, 3, 4, 5, 6, 7, 8, 8,
Ouput the list 'L1':
1,
Press any key to continue
```

图 3-19　例 3-20 的执行效果



```
L1 : 1 5
L2 : 2 3
L3 : 7 8
L0 : 9 -1
L1 合并 L2:1 5 2 3
L2 :
L1 合并 L0 :1 5 2 3 -1
L0 : 9
L1 合并 L3 :1 5 2 3 -1 7 8
L3 :
L1 (从大到小排序): 8 7 5 3 2 1 -1
L1 (从小到大排序): -1 1 2 3 5 7 8
Press any key to continue
```

图 3-20　例 3-21 的执行效果

提示：学习容器 list 的成员函数 splice()时，要关注合并以后，参数 x 表示的容器中的元素数目。同时应该注意不同的编译系统，对于 STL 的容器模板可能略有差别。

4．函数 unique()

对于 list 容器中存储的元素，可能存在连续的多个元素数值相等的情况，使用成员函数 unique()会移除重复元素，仅留下 1 个。成员函数 unique()的原型包括以下两种：

```
void unique( );
template<class BinaryPredicate>void unique(BinaryPredicate _Pred );
```

函数 splice()假定容器中元素是已排序的，因此所有相同的元素都是相邻的。不相邻的重复元素不能被移除。第二种函数形式，只有当元素满足条件表达式_Pred 时，该元素才被删除。由以上可

知，成员函数 unique()并不能保证序列中元素值的唯一性，而仅仅是将相邻的重复元素保留一个。函数的第二个原型：template<class BinaryPredicate>void unique (BinaryPredicate _Pred)，其中参数 BinaryPredicate _Pred 在 Visual C++6.0 中其具体形式如下：

```
not_equal_to<_Ty> _Pr2
```

所以，在使用第二种形式时，实现的功能是仅保留和第一个元素相等的元素，参数仅提供“函数”，即仅提供谓词。此时参数的实例化问题较突出，和前面的 remove_if()函数有相似之处。详见例 3-22。

例 3-22

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void Print(int& Ele)
{   cout<<Ele<<" ";
}
void main()
{ list<int>L1,L2;
  not_equal_to<int> Pred;
  L1.push_back(1);
  L1.push_back(2);
  L1.push_back(3);
  L1.push_back(1);
  L1.push_back(2);
  L1.push_back(3);
  L1.push_back(5);
  L1.push_back(7);
  L2.assign(L1.begin(),L1.end());
  for_each(L1.begin(),L1.end(),Print);
  cout<<endl;
  for_each(L2.begin(),L2.end(),Print);
  cout<<endl;
  L1.sort();
  L1.unique();
  for_each(L1.begin(),L1.end(),Print);
  cout<<endl;
  L2.sort();
  L2.unique(Pred);
  for_each(L2.begin(),L2.end(),Print);
  cout<<endl;
}
```

例 3-22 的执行效果见图 3-21。

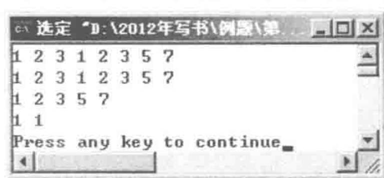


图 3-21　例 3-22 的执行效果

提示：使用 list 容器的成员函数 unique 时，尽量先使用 sort()对容器中的元素进行排序；函数 unique()的第二种形式使用时，参数仅是“谓词”，而不提供具体的数值！

5. 函数 reverse()

成员函数 reverse()实现将容器中所有元素用与原来相反的顺序排列。函数原型如下：

```
void reverse();
```

在使用函数 reverse()时，不需要任何参数，仅调用即可。容器中所有元素会按与原来相反的顺序排列。

总结：请注意！再次对 list 模板类做简要小结。list 型容器是非常重要的一种模板类，遗憾的是没有提供操作符[]和成员函数 at()。list 容器提供了大多数关于序列的操作功能。在学习以上内容的同时，要注意反复多读几遍，不要贪图速度。本书对每个知识点都进行了详细的、明了的讲解，读者应尽量学会所有的知识点。

3.5 序列式容器——deque（双端队列）类模板

“deque”是简写形式，其原意为“double-ended queue”。

模板类 deque 提供了对序列随机访问的功能，可以实现在序列两端进行快速插入和删除操作的功能，并可以在需要时修改自身大小。容器 deque 是典型的双端队列，完成了标准 C++数据结构中队列的所有功能。

容器 deque 采用动态数组来管理序列中的元素，提供随机存取，和 vector 具有几乎类似的接口。模板类（容器）deque 的最重要特征是在 deque 两端放置元素和删除元素是高效的，原因在于 deque 型序列开放了序列的两端，即头尾均开放，可以实现在序列两端进行快速的插入和删除。当需要向序列两端频繁地插入或删除数据元素时，最佳的容器为 deque。

作为动态数组形式，容器 deque 在具有其优势的同时，必然也存在其局限性。在 deque 型队列中间插入元素是非常费时费力的，必须移动其他元素。

容器 deque 的数据结构逻辑图见图 3-22。

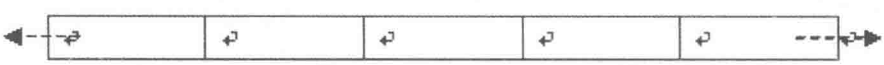


图 3-22　容器 deque 的结构逻辑图

3.5.1 容器 deque 和容器 vector 的对比

容器 deque 和 vector 相比，具有诸多优越之处。

（1）deque 可以在两端迅速插入和删除元素，而 vector 只提供了成员函数 push_back()和 pop_back()；

（2）存取元素时，容器 deque 会稍慢一些；

（3）容器 deque 的迭代器指针是智能型指针；

（4）在内存区块受限制的系统中，deque 型容器可以包含更多元素，因为 deque 型容器使用多块内存；

（5）容器 deque 不支持对容器和内存重分配时机的控制；

（6）deque 的内存区块不使用时，会被释放；

（7）在序列中间插入和删除元素时，deque 的速度很慢，需要移动相关的所有元素；

（8）容器 deque 的迭代器属于随机存取迭代器。

3.5.2　容器 deque 的定义和容量

1．定义

deque 是“double-ended queue”的缩写。deque 是动态数组，可以向两端发展。在 STL 库的头文件<deque>中定义了 4 种构造函数：

```
explicit deque(const A& al = A());
explicit deque(size_type n, const T& v = T(), const A& al = A());
deque(const deque& x);
deque(const_iterator first, const_iterator last, const A& al = A());
```

第 1 种构造函数定义了一种空的可控序列；第 2 种构造函数定义了 n 个相同的元素 x；第 3 种构造函数通过另一个 deque 容器定义，实现从另一个 deque 容器复制到新建容器；第 4 种构造函数通过另一个容器的某范围内元素，创建新容器。

根据以上 4 种构造函数，可以有以下方法用于定义 deque 对象。

```
deque<typename T> name;
deque<typename T> name(size);
deque<typename T> name(size,value);
deque<typename T> name(elsedeque);
deque<typename T> name(elsedeque.first(), elsedeque.end());
```

上述 5 种方法，第 1 种方法创建容纳类型 T 的空 deque 容器对象；第 2 种方法创建初始大小为 size 的 deque 对象；第 3 种方法创建初始大小为 size，每个元素初始值为 value 的 deque 对象；第 4 种方法用复制构造函数从现有的 deque 容器 elsedeque 中创建新的 deque 对象；第 5 种方法使用迭代器创建 deque 容器对象。

2．容量

deque 为度量数据成员数量和容器的大小，提供以下几种方法：resize(size_type n, T x=T()) ：把 deque 对象的大小重新调整为 n，并把对象中的新元素初始化为 x；max_size()：返回 deque 对象中的最大允许值；size()：返回 deque 的大小，即容器中元素的个数。

下面通过例 3-23 体会上述知识点。

例 3-23

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Print(int& Ele)
{    cout<<Ele<<" ";
}
void main()
{   deque<int>D1;
   deque<int>D2(5);
   deque<int>D3(10,2),D31(10,3);
   deque<int>D4(D3);
   deque<int>D5(D31.begin(),D31.end());
   cout<<"D1:"<<endl;
```

```
    for_each(D1.begin(),D1.end(),Print);
    cout<<endl;
    cout<<"D2:"<<endl;
    for_each(D2.begin(),D2.end(),Print);
    cout<<endl;
    cout<<"D3:"<<endl;
    for_each(D3.begin(),D3.end(),Print);
    cout<<endl;
    cout<<"D31:"<<endl;
    for_each(D31.begin(),D31.end(),Print);
    cout<<endl;
    cout<<"D4:"<<endl;
    for_each(D4.begin(),D4.end(),Print);
    cout<<endl;
    cout<<"D5:"<<endl;
    for_each(D5.begin(),D5.end(),Print);
    cout<<endl;
    int size=D2.size();
    cout<<"The size of D2: "<<size<<endl;
    int Msize=D2.max_size();
    cout<<"The maxsize of D2: "<<Msize<<endl;
    D31.resize(5,'A');
    cout<<"D31:"<<endl;
    for_each(D31.begin(),D31.end(),Print);
    cout<<endl;
    D31.resize(7,0);
    cout<<"D31:"<<endl;
    for_each(D31.begin(),D31.end(),Print);
    cout<<endl;
}
```

例 3-23 的执行效果见图 3-23。

图 3-23 例 3-23 的执行效果

总结：在例 3-23 中，对 5 种 deque 容器对象的定义形式分别用实例进行了阐述，读者应该迅速掌握。

3.5.3 deque 容器基础成员函数

1. 赋值

deque 容器提供了两个数据成员函数 push_front()和 push_back()用于把新元素插入 deque 对象中。push_front()用于在容器的头部插入新元素；push_back()用于在容器的尾部插入新元素。同理，函数 pop_front()和 pop_back()分别用于获取容器的头、尾两个元素。

注：在 vector 型容器中，在序列最前端插入新元素的效率非常低，所以 vector 中不包括在序列头部插入新元素的成员函数。

deque 容器还提供了运算符[]。operator[]的函数原型如下：

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

通过赋值运算符“=”，也可以实现对容器中元素的赋值，见例 3-24。

例 3-24

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Print(int& ele)
{    cout<<ele<<" ";
}
void main()
{  deque<int> D1;
   D1.push_front(0) ;
   D1.push_front(1);
   D1.push_front(2);
   D1.push_front(3);
   D1.push_front(4);
   D1.push_back(1);
   D1.push_back(2);
   D1.push_back(3);
   D1.push_back(4);
   cout<<"D1:"<<endl;
   for_each(D1.begin(),D1.end(),Print);                //注意观察元素在序列中的排列位置
   cout<<endl;
   D1[4]=9;                                             //修改序列中的中间一个元素
   cout<<"D1:"<<endl;
   for_each(D1.begin(),D1.end(),Print);                //注意观察元素在序列中的排列位置
   cout<<endl;
}
```

例 3-24 的执行效果见图 3-24。

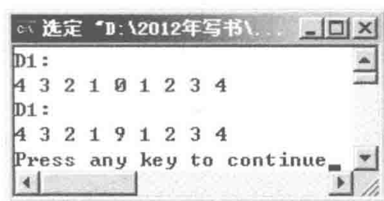


图 3-24　例 3-24 的执行效果

2. 迭代器相关函数

在 deque 容器中，提供了返回 deque 对象的迭代器型指针或引用的成员函数，主要分为以下两类：

（1）begin，rbegin，end，rend；

（2）back，front。

第（1）类中 4 个函数返回值均为迭代器型，或逆迭代器型；第（2）类中 2 个函数返回值为引用，代码如下：

```
const_iterator begin() const;
iterator begin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
const_iterator end() const;
iterator end();
const_reverse_iterator rend() const;
reverse_iterator rend();
reference back();
const_reference back() const;
reference front();
const_reference front() const;
```

（3）判断迭代器是否为空

deque 型容器提供了成员函数 empty()。如果容器为空，函数 empty()返回值为真。函数原型如下：

```
bool empty() const
```

（4）元素的访问

deque 型容器提供了用于随机访问容器中元素的成员函数 at()，的语法如下：

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

（5）deque 序列的元素值重置技术

容器模板 deque 提供了函数 assign()，函数 assign()可以方便地重置 deque 中某个元素的数值。函数的原型如下：

```
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
```

利用例 3-25，对上述（2），（3），（4），（5）的内容加深印象，加深理解。

例 3-25

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Out(double& Ele)
{    cout.width(5);
     cout.precision(1);
     cout<<std::fixed<<Ele<<" ";
}
void main()
{    deque<double>::iterator Iter;
     deque<double>::reverse_iterator rIter;
     deque<double> D1,D2,D3;
     for(int i=0;i<10;i++)
     {    D1.push_front(90+i/10.0);
     }
     cout<<"All the element of 双端序列 D1:" <<endl;
     for_each(D1.begin(),D1.end(),Out);
     cout<<endl;
     Iter=D1.begin();
     double begin=*Iter;
     Iter=D1.end();
     double end=*(--Iter);
     ///////////////
     rIter=D1.rbegin();
     double rbegin=*(rIter);
     rIter=D1.rend();
     double rend=*(--rIter);                              //注意是“--”
     double front=D1.front();
     double back=D1.back();
     cout<<" begin :"<<begin<<","<<" end :"<<end<<","<<" front :"<<front<<","<<" back :
"<<back<<endl;
     cout<<" reverse begin :"<<rbegin<<" reverse end :"<<rend<<","<<endl;
     if(D1.empty())
```

```
	{	cout<<"双端序列为空."<<endl;
	}
	else
	{	int size=D1.size();
		cout<<"双端序列中包含"<<size<<"个元素."<<endl;
	}
	double five=D1.at(5);
	cout<<"第五个元素是"<<five<<"."<<endl;
	D2.assign(6,0);
	D3.assign(D1.begin(),D1.end());
	cout<<"All the element of 双端序列 D2:" <<endl;
	for_each(D2.begin(),D2.end(),Out);
	cout<<endl;
	cout<<"All the element of 双端序列 D3:" <<endl;
	for_each(D3.begin(),D3.end(),Out);
	cout<<endl;
}
```

例 3-25 的执行效果见图 3-25。

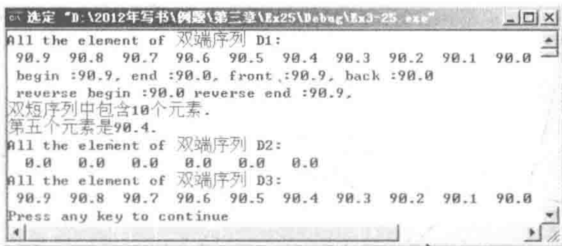


图 3-25　例 3-25 的执行效果

例 3-25 对（2）、（3）、（4）、（5）的内容进行了详细的实例化。请读者认真阅读，深刻体会。

3.5.4 deque 容器的高级编程

1. 容器元素交换

在 deque 中提供了成员函数 swap 进行两个 deque 变量的交换操作。通过 swap 函数可以完成两个 deque 变量的相互内容交换。函数原型如下：

```
Template <class T, class Allocator>void swap(deque<T,Allocator>& x, deque<T,Allocator>&y);
```

swap()既可以作为 deque 的成员函数来使用，还可以作为通用算法使用，两者功能是一致的。

2. 插入和删除

Deque 容器提供了丰富而又灵活的插入和删除元素操作。容器 deque 除了提供 push_front()，push_back()，pop_front()，pop_back()实现插入和删除功能之外，函数 insert()能实现把某个元素插入指定的位置，也可以把指定范围的多个元素插入 deque 中迭代器所指位置，还可以把某个具体数值的多个备份重复插入 deque 的迭代器所指位置。函数 erase()主要用来删除 deque 中的元素，既可以删除某个具体位置的元素，也可以删除指定范围的多个元素。函数 clear()用于删除容器中所有元素。

（1）insert()

函数 insert()的原型如下：

```
iterator insert(iterator it, const T& x = T());
```

```
void insert(iterator it, size_type n, const T& x);
void insert(iterator it, const_iterator first, const_iterator last);
```

第一种原型的作用在于把某个元素插入指定位置，返回值为被插入元素的位置；第二种原型的作用是把某个值的多个备份插入 deque 容器中迭代器所指位置。

（2）erase()和 clear()

在进行 deque 操作时，经常需要进行元素的删除操作，包括在 deque 的头部、尾部和中间任何位置进行该操作。由于 deque 是双端队列，在队列的头部和尾部删除操作最简单、最高效。容器模板 deque 提供了 pop_front()和 pop_back()两个专门函数，实现从队列的头部和尾部删除元素。成员函数 erase()较这两个函数更加灵活实用。函数 erase()可以删除迭代器指向的任意单个元素或相联的多个元素。函数 clear()可以无条件地删除容器中所有元素，clear()可以被 erase(begin，end)所代替。

（3）查找

模板类 deque 没有提供与“查找”和“搜索”相关的函数。使用头文件<algorithm>中声明的算法 find()，同样可以实现对 deque 型容器的查找。详见下述代码的中文注释。

以上内容详见例 3-26。

例 3-26

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void Out(double& Ele)
{   cout.width(5);
    cout.precision(1);
    cout<<std::fixed<<Ele<<" ";
}
void main()
{    deque<double>D1,D2;
     for(int i=0;i<10;i++)                                //初始化容器
     {    D1.push_front(90+i/10.0);
     }
     cout<<"All the element of 双端序列 D1:" <<endl;
     for_each(D1.begin(),D1.end(),Out);                   //输出容器 D1 中的内容
     cout<<endl;
     for(i=0;i<10;i++)
     {    D2.push_front(80+i/10.0);
     }
     cout<<"All the element of 双端序列 D2:" <<endl;
     for_each(D2.begin(),D2.end(),Out);                   //输出容器 D2 中的内容
     cout<<endl;
     D1.swap(D2);                                         //交换容器 D1 和 D2 中的内容
     cout<<"D1 swap D2 :"<<endl;
     cout<<"All the element of 双端序列 D1:" <<endl;
     for_each(D1.begin(),D1.end(),Out);                   //输出 D1 中的所有元素
     cout<<endl;
     cout<<"All the element of 双端序列 D2:" <<endl;
     for_each(D2.begin(),D2.end(),Out);                   //输出 D2 中的所有元素
     cout<<endl;
     D1.insert(D1.begin(),-1.0);                          //插入三个元素
```

```
    D1.insert(D1.end(),-1.0);
    D1.insert(D1.begin()+6,-0.0);
    cout<<"All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(),D1.end(),Out);
    cout<<endl;
    D1.erase(D1.begin()+1);                                        //删除两个元素
    D1.erase(D1.begin()+10);
    cout<<"All the element of 双端序列 D1:" <<endl;
    for_each(D1.begin(),D1.end(),Out);
    cout<<endl;
    D2.clear();
    cout<<"D2 has  been already cleared!"<<endl;
    cout<<"All the element of 双端序列 D2:" <<endl;
    for_each(D2.begin(),D2.end(),Out);
    cout<<endl;
    deque<double>::iterator it=find(D1.begin(),D1.end(),80.5) ;        //使用 find 算法
    int step=(it-D1.begin());                                      //求出所照元素的位置（下标）
    cout<<"find 80.5. in D1 ,  its Index: "<<step<<" ."<<endl;
}
```

例 3-26 的执行效果见图 3-26。

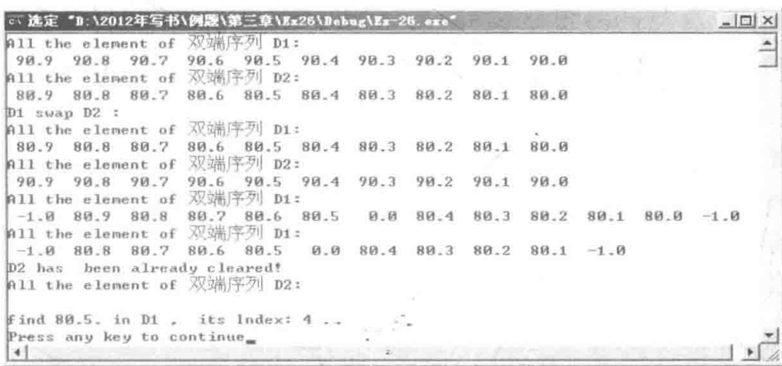


图 3-26　例 3-26 的执行效果

3.5.5　deque 的模板函数

模板类 deque 提供了大量模板函数，模板函数使 deque 容器功能异常强大，同时也使编程异常便捷。在 deque 容器中主要包括运算符函数，也包括 swap()函数（前面已介绍），此处主要介绍运算符函数。

以模板函数形式实现的运算符主要包括：[]，== ，< ，! =，<=，>，>=。

- “[]”用于访问容器中任意元素；
- “==”用于判断两个容器是否相等；
- “<”用于判断两个容器是否相同，如果前者小于后者，返回 true，否则返回 false；
- “! =”用于判断两个容器是否不相等，如果不相等，返回 true，如果相等，返回 false；
- “<=”用于判断两个容器是否相等，如果前者不大于后者，返回 true，否则返回 false；
- “>”用于判断两个容器是否相等，如果前者大于后者，返回 true，否则返回 false；
- “>=”用于判断两个容器是否相等，如果前者不小于后者，返回 true，否则返回 false。

上述模板函数不受数据类型限制，顺序型容器全部提供了这些成员函数，使用方法也相同，在此不再赘述。

3.6 关联式容器概述

关联式容器是最常见也是最有用的容器。关联式容器其实是关联数组概念的推广。

关联式容器依据特定的排序准则，自动为其元素排序。关联式容器中的元素都经过排序，是已排序的。所有关联式容器都有一个可供选择的 template 参数，指明排序原则。排序准则以函数形式呈现，用于比较元素值或元素键。模板情况下以“operator<”进行比较，当然程序员也可以提供自定义的比较函数，从而定义出不同的排序准则。

通常关联式容器由二叉树实作出来。在二叉树中，每个元素都有一个父节点和两个子节点；左子树的所有元素都比自己小，右子树的所有元素都比自己大。关联式容器的差别在于元素的类型以及处理重复元素的方式。

关联式容器的另一个优点：提供了对元素的快速访问。但是却不能实现任意位置的操作。

主要的关联式容器包括：set，multiset，map，multimap。其中 set 可视为一种特殊的 map，其元素的值即键值。前两种容器是在<set>头文件中声明和定义的，后两种是在<map>头文件中声明和定义的。

set 是一种随机存取的容器，其关键词和数据文件是同一个值。set 对象中的所有元素必须具有唯一值，即不能包含重复的元素。set 对象可以使程序按照次序来存储一组数值，在一个集合中元素既作为被存储的数据又作为数据的关键值。集合的本质就是一个有序的排列。

multiset 是另一种类型的容器，其关键词和数据元素是同样的值。与 set 不同的是，它可以包含重复的元素。multiset 对象可以使程序按照次序存储一组数值。

map 是一种关联数组容器，包含成对数据的容器，一个值是实际数据值，另一个值是用来寻找数据的关键值，一个特定的关键词只能与一个元素相联系。map 是排序结构体，键值是独一无二的。事实上，map 的内部结构和 set 是一样的。set 可以认为是一种特殊的 map，其键值和实值是同一个。

multimap 是一种允许出现重复键关键值的关联数组容器。与 map 对象不同，一个关键词可以和多个元素相联系，multimap 允许键值重复。

下面将主要介绍上述 4 种关联式容器。

3.7 关联式容器——set/multiset 类模板

集合（set）型容器能顺序存储一组值。在一个集合中，集合元素既充当存储的数据，又充当数据的关键码。集合更像一个有序链表，元素以升序的顺序存储。

3.7.1 集合 set 的定义

模板类 set 的声明如下：

```
template <class Key, class Traits=less<Key>, class Allocator=allocator<Key> >class set
```

其中，参数 Key 是存储在集合中元素的数据类型；参数 Traits 是用于实现集合内部排序的仿函数，默认为 less<Key>；参数 Allocator 代表集合的内存配置器，负责内存的分配和销毁。

使用集合（set）时，需要包含头文件<set>。集合（set）的构造函数包含以下几种：

```
explicit set( const Traits& _Comp);
explicit set( const Traits& _Comp, const Allocator& _Al);
set( const _set& _Right);
template<class InputIterator> set(InputIterator _First,InputIterator _Last );
template<class InputIterator> set(InputIterator _First,
                            InputIterator _Last,
                            const Traits& _Comp);
template<class InputIterator>set( InputIterator _First,
                            InputIterator _Last,
                            const Traits& _Comp,
                            const Allocator& _Al );
```

上述是 6 种集合（set）的创建方法。第 1 种形式最简单，定义一个空对象（容器），参数包含了一个谓词，用于容器内部排序；第 2 种形式除包含一个谓词之外，还加入了内存配置器；第 3 种形式用于使用已存在的集合（set）定义新的集合（set）；第 4 种形式包含使用迭代器指定固定范围，用于定义新的集合（set）；第 5 种形式比第 4 种形式增加参数_Comp，从而可以自定义排序规则；第 6 种形式在第 5 种形式的基础上增加内存配置器的内容。

在本节之前，本书已经大量使用了“排序”。在此，对排序准则进行简要描述。所谓排序准则：

（1）必须是“反对称的”。对操作符“operator<”而言，如果 x < y 为真，则 y < x 为假。对判断式 predicate op()而言，如果 op(x,y)为真，则 op(y,x)为假。

(2) 必须是“可传递的”。对操作符“operator<”而言，如果 x < y 为真且 y < z 为真，则 x < z 为真。对判断式 predicateop()而言，如果 op(x,y)为真且 op(y,x)为真，则 op（x,z）为真。

（3）必须是“非自反的”。对操作符“operator<”而言，x < x 永远为假，是不可能的。对判断式 predicate op()而言，op(x,x)永远为假。

总结：以上内容是 STL 学术理论的一部分。STL 先建立起一个抽象概念阶层体系，形成一个软件组件分类学，最后再以实际工具（template）将各个概念实现。这些理论架构在《Generic Programming and the STL》一书中有诸多描述，其中译本为《泛型程序设计与 STL》。

在具体实现排序准则时，可有两种形式实现。第一，在模板中以参数形式实现；第二，以构造函数参数定义。

```
(1) std::set<int, std::greater<int>>s1;           //第一种形式实现排序规则
(2) set<int>s2(less<int>());                       //第二种形式实现排序规则
```

集合（set）具有排序功能（multiset 同样也有），也具有其局限性。元素一旦被输入容器中，就无法再对其进行必要的访问，因为元素的位置改变了。原有的输入顺序被自动排序功能打乱。要改变元素的值，需要先删除原有的元素，再重新插入新元素。所以，set 和 multiset 没有提供任何直接存取元素的成员函数。虽然 STL 定义了很多种 set 定义方法，但 Visual C++6.0 并没有提供所有的定义方法，Linux 环境下的 C++编译系统可以实现。

通过例 3-27 对以上内容加以详细说明。

例 3-27

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

```
void OutPut(set<int> &s)
{   set<int>::iterator    it;
    for(it=s.begin();it!=s.end();it++)
        cout<<" "<<*it<<", ";
    cout<<endl;
}
void OutPutM(multiset<int> &s)
{
multiset<int>::iterator    it;
for(it=s.begin();it!=s.end();it++)
cout<<" "<<*it<<", ";
cout<<endl;
}
void main()
{  set<int>s1;
  s1.insert(10);
  s1.insert(15);
  s1.insert(25);
  s1.insert(20);
  s1.insert(30);
  s1.insert(33);
  s1.insert(5);
  s1.insert(20);
  OutPut(s1);
  set<int>s2(less<int>());
  s2.insert(10);
  s2.insert(13);
  s2.insert(11);
  s2.insert(19);
  s2.insert(17);
  s2.insert(21);
  OutPut(s2);
  set<int>::allocator_type s1_Alloc;
  s1_Alloc=s1.get_allocator();
  set<int>s3(less<int>(),s1_Alloc);
  s3.insert(1);
  s3.insert(5);
  s3.insert(2);
  OutPut(s3);
  set<int>s4(s1);
  OutPut(s4);
  multiset<int>sm1;
  sm1.insert(10);
  sm1.insert(15);
  sm1.insert(25);
  sm1.insert(20);
  sm1.insert(30);
  sm1.insert(33);
  sm1.insert(5);
  sm1.insert(20);
  OutPutM(sm1);
}
```

例 3-27 的执行效果见图 3-27。

例 3-27 中的几种定义形式，在 Visual C++6.0 中调试通过。在模板中使用多个参数的定义形式不能调试通过。另外，除了允许元素重复之外，multiset 的定义形式和 set 是相同的。

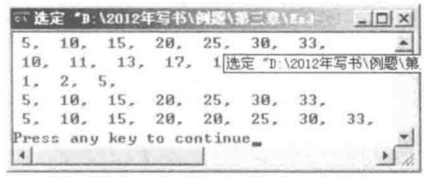


图 3-27　例 3-27 的执行效果

提示：在 main()函数的第 9 行，输入了一个重复元素“20”，但程序输出时，并没有这个元素。说明 set 中不允许有重复的元素。

3.7.2　set 和 multiset 的容量、搜寻和统计

集合型容器 set（multiset）和其他容器一样，提供了获取容器容量的函数（size()）、判断容器是否为空的函数（empty()）以及返回容器可容纳最大元素数量的成员函数 max_size()。

集合型容器 set（multiset）还提供了元素个数统计函数 count()。

集合型容器 set（multiset）提供了查找函数 find()，还提供了 low_bound()，upper_bound()，equal_range()。

以上几个函数的原型如下：

```
size_type size( ) const;
bool empty( ) const;
size_type max_size( ) const;
size_type count( const Key& _Key) const;
iterator find( const Key& _Key) const;
const_iterator find( const Key& _Key) const;
const_iterator lower_bound(const Key& _Key) const;
iterator lower_bound(const Key& _Key) const;
const_iterator upper_bound(const Key& _Key) const;
iterator upper_bound(const Key& _Key) const;
pair <const_iterator, const_iterator> equal_range (const Key& _Key) const;
pair <iterator, iterator> equal_range (const Key& _Key) const;
```

函数 find(Key)的功能是返回键值为 Key 的元素的位置，函数返回值是迭代器类型。

函数 count(Key)的功能是统计键值为 Key 的元素个数。

函数 low_bound(const Key& _Key)的返回值类型是迭代器类型，该迭代器指向集合中键值大于等于参数 Key 的第一个元素。

函数 upper_bound(onst Key& _Key)的返回值类型是迭代器类型，该迭代器指向集合中键值大于参数 Key 的第一个元素。

函数 equal_range (const Key& _Key)的返回值类型是迭代器对（pair），该迭代器对（pair）的两个迭代器分别指向集合中键值大于并等于参数 key 的第一个元素和集合中键值大于参数 Key 的第一个元素。

下面使用例 3-28 来说明以上几个函数的使用方法。

例 3-28

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
```

```
using namespace std;
void Print(int& Ele)
{   cout<<Ele<<", ";
}
void main()
{   typedef set<int> SET;
    typedef multiset<int> MSET;
    SET s1;
    MSET s2;
    SET::iterator it;
    MSET::iterator Mit;
    pair<SET::iterator, SET::iterator>p1;
    pair<MSET::iterator, MSET::iterator>Mp1;
    s1.insert(10);
    s1.insert(15);
    s1.insert(21);
    s1.insert(17);
    s2.insert(11);
    s2.insert(16);
    s2.insert(20);
    s2.insert(18);
    s2.insert(20);                                      //以上是初始化
    cout<<"The Set S1 is below:"<<endl;
    for_each(s1.begin(),s1.end(),Print);
    cout<<endl;
    cout<<"The Multiset S2 is below:"<<endl;
    for_each(s2.begin(),s2.end(),Print);
    cout<<endl;
    int size=s1.size();                                 //计算容量
    int Msize=s2.size();
    cout<<"The size of the set s1: "<<size<<endl;
    cout<<"The size of the Multiset s2: "<<Msize<<endl;
    int max_size=s1.max_size();                         //最大容量
    int max_Msize=s2.max_size();
    cout<<"The max_size of the set s1: "<<max_size<<endl;
    cout<<"The max_size of the Multiset s2: "<<max_Msize<<endl;
    int cnt=s1.count(10);                               //指定元素的个数
    int Mcnt=s2.count(20);
    cout<<"The count of \'10\' in set s1: "<<cnt<<endl;
    cout<<"The count of \'20\' in multiset s2: "<<Mcnt<<endl;
    it=s1.find(21);                                     //查找元素
    Mit=s2.find(18);
    cout<<"The element with a key '21' is: "<<*it<<endl;
    cout<<"The element with a key '18' is: "<<*Mit<<endl;
    it=s1.lower_bound(15);                              //查找元素
    Mit=s2.lower_bound(16);
    if(it==s1.end())
    {   cout<<"The element with a key 15 in s1 doesn't exist."<<endl;
    }
```

```
    else
    {   cout<<"The element with a key 15 in s1 is: "<<*it <<endl;
    }
    if(Mit==s2.end())
    {   cout<<"The element with a key 16 in s2 doesn't exist."<<endl;
    }
    else
    {   cout<<"The element with a key 16 in s2 is: "<<*Mit <<endl;
    }
    it=s1.upper_bound(15);                              //查找元素
    Mit=s2.upper_bound(16);
    if(it==s1.end())
    {   cout<<"The element with a key >15 in s1 doesn't exist."<<endl;
    }
    else
    {   cout<<"The element with a key >15 in s1 is: "<<*it <<endl;
    }
    if(Mit==s2.end())
    {   cout<<"The element with a key >16 in s2 doesn't exist."<<endl;
    }
    else
    {   cout<<"The element with a key >16 in s2 is: "<<*Mit <<endl;
    }
    p1=s1.equal_range(15);                              //查找元素
    Mp1=s2.equal_range(16);
    if(it==s1.end())
    {
    cout<<"The element with a key >=15 in s1 doesn't exist."<<endl;
    }
    else
    {
       cout<<"The element with a key =15 in s1 is: "<<*p1.first <<endl;
        cout<<"The element with a key >15 in s1 is: "<<*p1.second<<endl;
    }
    if(Mit==s2.end())
    {
       cout<<"The element with a key >16 in s2 doesn't exist."<<endl;
    }
    else
    {
       cout<<"The element with a key =16 in s2 is: "<<*Mp1.first <<endl;
       cout<<"The element with a key >16 in s2 is: "<<*Mp1.second <<endl;
    }
}
```

例 3-28 的执行效果见图 3-28。

图 3-28　例 3-28 的执行效果

3.7.3　set 和 multiset 的迭代器相关函数和赋值函数

set 和 multiset 提供了所有容器均提供的基本赋值操作："="，swap()。赋值操作的两端容器必须具有相同型别。尽管比较准则本身不可能相同，但型别必须相同。

与容器 set 和 multiset 相关的迭代器函数主要包括：begin()，end()，rbegin()，rend()。其中，begin()和 end()是双向迭代器，rbegin()和 rend()是逆向迭代器。对于迭代器，所有的元素都被视为常数，确保不会人为改变元素值，打乱既定顺序。因此，无法对 set 和 multiset 元素调用任何变动性算法。并且对于 set 和 multiset 不能使用 remove()算法，要移除 set 和 multiset 中的元素，必须使用它们提供的成员函数。

函数 swap()的原型如下：

```
void swap(set& str);
```

函数 begin()和 end()的原型如下：

```
const_iterator begin() const
const_iterator end() const;
```

函数 rbegin()和 rend()的原型如下：

```
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;
```

例 3-29 对以上函数进行了具体介绍。

例 3-29

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void print(double& Ele)
{    cout<<Ele<<" ";
}
void main()
{    set<double> s1,s2;
```

```
    s1.insert(11);
    s1.insert(21);
    s1.insert(17);
    s1.insert(19);
    s1.insert(9);
    s1.insert(13);                                          //初始化
    cout<<"s1: "<<endl;
    for_each(s1.begin(),s1.end(),print);                    //输出 s1
    cout<<endl;
    s2=s1;
    cout<<"s2: "<<endl;
    for_each(s2.begin(),s2.end(),print);                    //输出 s2
    cout<<endl;
    s2.insert(31);
    s2.insert(24);
    cout<<"s2: "<<endl;
    for_each(s2.begin(),s2.end(),print);                    //输出 s2
    cout<<endl;
    s1.swap(s2);
    cout<<"s1: "<<endl;
    for_each(s1.begin(),s1.end(),print);                    //交换
    cout<<endl;
    cout<<"s2: "<<endl;
    for_each(s2.begin(),s2.end(),print);                    //输出
    cout<<endl;
    set<double>::iterator its;
    set<double>::reverse_iterator rits;
    its=s1.begin();                                         //第一个元素
    cout<<"The first Element of sequence \'s1\':"<<*its<<endl;
    its=s1.end();                                           //最后一个元素
    cout<<"The last Element of sequence \'s1\':"<<*(--its)<<endl;
    rits=s1.rbegin();                                       //逆向第一个元素
    cout<<"The first Element of sequence in reverse direction.\'s1\':"<<*rits<<endl;
    rits=s1.rend();                                         //逆向最后一个元素
    cout<<"The last Element of sequence in reverse direction.\'s1\':"<<*(--rits)<<endl;
}
```

例 3-29 的执行效果见图 3-29。

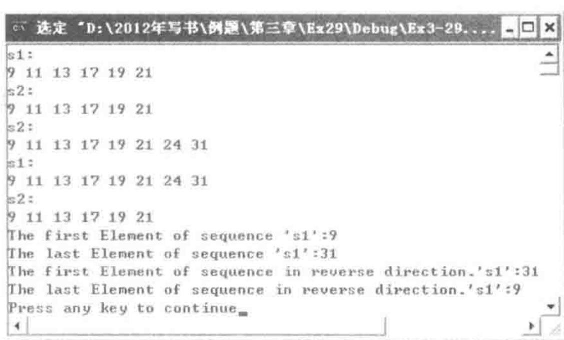


图 3-29　例 3-29 的执行效果

3.7.4 set 和 multiset 的插入和移除

容器 set 和 multiset 的元素插入和删除函数主要包括：insert()，erase()，clear()。迭代器指向有效位置，序列起点不能位于终点之后，不能从空容器中删除元素（使用之前要判断容器的容量）。函数 insert()可以加入一个对象、一个对象的若干复制，或者一个范围内的对象。插入位置是使用迭代器 iterator 指定的。

函数 insert()和 erase()的原型如下：

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
void insert(const value_type *first, const value_type *last);

iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

上述第一种模型中，函数 insert()的返回值是 pair<iterator,bool>类型。pair 型返回值的第 1 个参数是迭代器指针，代表插入元素的位置；pair 型返回值的第 2 个参数是 bool 类型，代表安插是否成功。

函数 erase()可以删除一个由迭代器指定的元素，也可以删除一个范围内的元素。第 1 种语法的功能是将迭代器所指向的元素删除，第 2 种语法是将迭代器所值范围内的元素全部删除，第 3 种语法的功能是将元素 Key 删除。

提示：请读者注意，在例 3-30 中，第一次使用了算法 distance()，用于计算容器中两个元素的距离。在本例中也第一次较详细地使用了 pair 类型的两个成员 first 和 second，这两个成员分别代表 pair 对象中的两个元素。例中仅使用了 set 类型，multiset 和 set 的区别仅在于可存在重复元素。

下面请参考例 3-30。

例 3-30

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void print(int& Ele)
{   cout<<Ele<<" ";
}
template <typename T>void printSet(set<T>& s)
{   for_each(s.begin(),s.end(),print);
    cout<<endl;
}
void main()
{   set<int>s1,s2;
    pair<set<int>::iterator,bool> p1;
    s1.insert(10);
    s1.insert(11);
    s1.insert(13);
    s1.insert(21);
    s1.insert(17);
```

```
    cout<<"s1: "<<endl;
    printSet(s1);
    p1=s1.insert(12);
    if(p1.second==true)
    {   cout<<"The element 12 be inserted successfully."<<endl;
    }
    else
    {   cout<<"The element 12 already exist in s1."<<endl;
    }
    cout<<"s1: "<<endl;
    printSet(s1);
    p1=s1.insert(17);
    if(p1.second==true)
    {   cout<<"The element 17 be inserted successfully."<<endl;
        cout<<"The position of 17 is:"<<distance(s1.begin(),p1.first)+1<<endl;
    }
    else
    {   cout<<"The element 17 already exist in s1."<<endl;
        cout<<"The position of 17 is:"<<distance(s1.begin(),p1.first)+1<<endl;
    }
    cout<<"s1: "<<endl;
    printSet(s1);
    s2=s1;
    s2.insert(25);
    cout<<"s2: "<<endl;
    printSet(s2);
    s2.erase(25);
    cout<<"s2: (after erasing 25.) "<<endl;
    printSet(s2);
    s2.erase(s2.begin());
    cout<<"s2: (after erasing *begin() "<<endl;
    printSet(s2);
    s2.erase((++s2.begin()),(--s2.end()));
    cout<<"s2: (after erasing from ++begin() to --end()"<<endl;
    printSet(s2);
}
```

例 3-30 的执行效果见图 3-30。

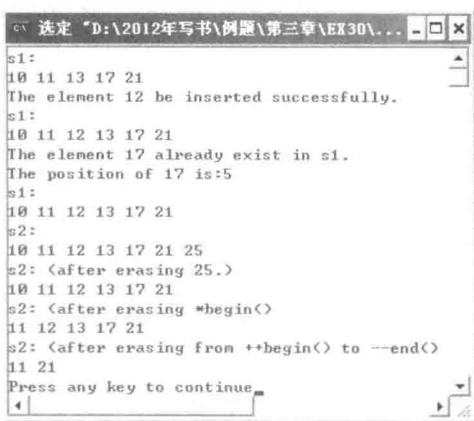


图 3-30　例 3-30 的执行效果

3.7.5 set 和 multiset 的比较运算符

容器之间的比较只能用于相同型别。元素和排序则必须有相同的型别，否则编译时会产生错误。在 ISO/IEC14882 中，提供了关于运算符的声明和定义。运算符的使用和序列式容器是一致的。

Visual C++6.0 中没有提供比较运算符的成员函数，但 Visual C++6.0 中提供了另外两个函数：key_comp()，value_comp()。函数 key_comp()用于键值比较，函数 value_comp()用于实值比较。这两个函数的原型为：

```
key_compare key_comp() const;
value_compare value_comp() const;
```

函数 key_comp()的返回值是 key_compare 类型，key_compare 决定了集合中元素的排列顺序。通过对返回值的测试，可以确定序列排序形式。函数 value_comp()的返回值是 value_compare 类型。其意义和 key_compare 是一致的。

1. 键值比较

例 3-31 用来说明如何使用函数 key_comp()。请读者认真阅读下面的源代码。

例 3-31

```
#pragma warning(disable:4786)
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void Print(double& Ele)
{    cout<<Ele<<" ";
}
template <typename T>void PrintS(set<T>& s)
{    for_each(s.begin(),s.end(),Print);
     cout<<endl;
}
template <typename T>void PrintM(multiset<T,greater<T> >& s)
{    for_each(s.begin(),s.end(),Print);
     cout<<endl;
}
using namespace std;
void main()
{    set<double,less<double> >s1;
     s1.insert(1);
     s1.insert(5);
     s1.insert(7);
     s1.insert(4);
     multiset<double,greater<double> >s2;
     s2.insert(7);
     s2.insert(9);
     s2.insert(6);
     s2.insert(4);
     set<double,less<double> >::key_compare kc1=s1.key_comp();
     multiset<double,greater<double> >::key_compare kc2=s2.key_comp();
     cout<<"s1:"<<endl;
     PrintS(s1);
```

```
    cout<<"s2:"<<endl;
    PrintM(s2);
    if(kc1(2,3)==true)
    {   cout<<"The function object of s1 return true."<<endl;
    }
    else
    {   cout<<"The function object of s1 return false."<<endl;
    }
    if(kc2(2,3)==true)
    {   cout<<"The function object of s2 return true."<<endl;
    }
    else
    {   cout<<"The function object of s2 return false."<<endl;
    }
}
```

例 3-31 的执行效果见图 3-31。

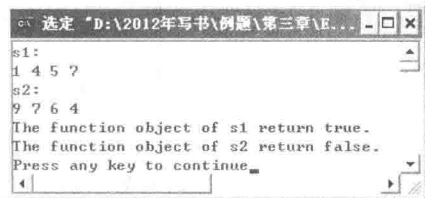


图 3-31　例 3-31 的执行效果

总结：在本例中，关于 set 和 multiset 两个类型，例 3-31 第一次使用了较复杂的定义对象形式（见源代码第 19 行和第 24 行）。在使用时，注意“less<double>”和其后面的“>”之间一定要有一个空格，否则编译时会出错。

2. 实值比较

函数 value_comp()的使用方法和 key_comp()的使用方法相同，只不过函数的返回类型为 value_compare 型。在此不再重新举例，对例 3-31 稍微改造即可。

3.8 关联式容器——map/multimap 类模板

容器 map 是键-值对的集合。容器 map 类型通常可以理解为关联数组（associative array）。可以使用键作为下标来获取对应的值，正如内置数组类型一样。而关联的本质在于元素的值与某个特定的键相关联，而不是通过在数组中的位置来获取。

map 是由许多对的键值组成的排序结构体，键值独一无二。

容器类型 multimap 和容器类型 map 基本是一致的。只是 multimap 允许重复元素，而 map 不允许。

容器类型 map 和 multimap 的示意图见图 3-32。

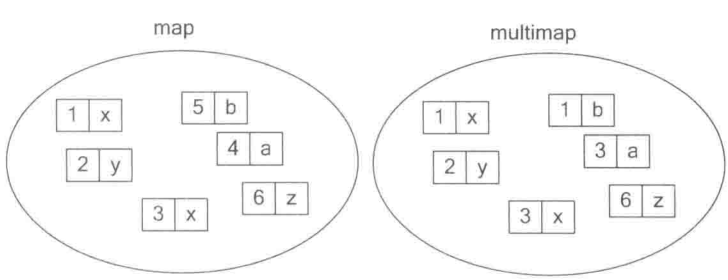


图 3-32　容器 map 和 multimap 的示意图

在 STL 中，两种型别的模板如下：

```
template <class Key,class T,class Compare=less<Key>, class Allocator=allocator<pair
<const Key,T >>>class map;

template <class Key, class T, class Compare=less<Key>,class Allocator=allocator<pair
<const Key,T>>> class multimap;
```

第一个参数是容器中元素的 key（键），第二个 template 参数被当作元素的 value。前两个参数 key 和 value 必须具备可赋值（assignable）和可复制（copyable）的性质。并且由于容器内部要排序，所以 key 必须是可以比较（comaprable）的。第三个参数可有可无，用于定义排序准则。元素的顺序仅由 key 决定，和 value 是无关的。默认排序准则是“less<>”。

由以上内容可知，map 对象可以使程序按照次序存储一组数值。每个元素均由一个键实现排序和检索。在上一节介绍的 set/multiset 容器中，元素既作为关键值，又作为数值。而在 map/multimap 中，实现了键和值以“数据对”（pair）的形式出现。

3.8.1 map 和 multimap 基础

为了使用 map 和 multimap，必须包含头文件<map>。在内存中，map 和 multimap 的对象均是以数据对的形式存放在二叉树中的。

1．map 和 multimap 的定义、初始化

容器 map 和 multimap 的构造函数，如表 3-1 所示。

表 3-1 容器 map 和 multimap 的构造函数

构造函数	备注
map c / multimap c	产生一个空的 map 或 multimap，不包含任何因素
map c (op) / multimap c(op)	以 op 为排序准则，产生一个空的 map/multimap
map c1(c2) / multimap c1(c2)	产生某个 map/multimap 对象的副本，所有元素均被复制
map c(beg, end) / multimap c(beg, end)	以区间(beg, end)内的元素产生一个新的 map/multimap
map c(beg, end,op) / multimap c(beg, end,op)	以区间(beg, end)内的元素产生一个新的 map/multimap，排序准则为 op

容器 map 和 multimap 的形式如表 3-2 所示。

表 3-2 容器 map 和 multimap 的形式

map	效果
map<key, Elem>	以 less<>为排序准则
map<key, Elem,op>	以 op 为排序准则
multimap<key, Elem>	以 less<>为排序准则
multimap <key, Elem,op>	以 op 为排序准则

通过了解表 3-1 和表 3-2，基本可以实现对 map 的定义。根据表格中的内容任意组合即可。例如：

```
map <key, Elem> c;
map <key, Elem> c(op);
map <key, Elem> c(beg, end);
… …
```

通常定义排序准则有两种方法：以模板参数定义；以构造函数参数定义。在表 3-1 和表 3-2 中已经体现，在此进行简单的阐述。

（1）使用模板参数定义排序准则

例如：

```
map<float, string, greater<float> >m1;
```

此时，排序准则是型别的一部分。而实际的排序准则是容器产生的函数对象（仿函数）。

（2）以构造函数参数定义排序准则

以构造函数参数的形式定义排序准则，只有当构造函数被执行时，排序准则才有效。执行期才能获得排序准则，而且程序要用到不同的排序准则。

下面用例 3-32 来阐述以上内容。

例 3-32

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef pair<int, double> CustomPair;
void Print(map<int,double>& m)
{   map<int, double>::iterator it;
    for(it=m.begin();it!=m.end();it++)
    {   CustomPair p1=(pair<int, double>)(*it);
        cout<<p1.first<<",  ";
        cout<<std::fixed<<cout.precision(2)<<p1.second<<";  "<<endl;
    }
}
void PrintM(multimap<int,double>& m)
{   multimap<int, double>::iterator it;
    for(it=m.begin();it!=m.end();it++)
    {   CustomPair p1=(pair<int, double>)(*it);
        cout<<p1.first<<",  ";
        cout<<std::fixed<<cout.precision(2)<<p1.second<<";  "<<endl;
    }
}
void PrintG(map<int,double,greater<int> >& m)
{   map<int, double,greater<int> >::iterator it;
    for(it=m.begin();it!=m.end();it++)
    {   CustomPair p1=(pair<int, double>)(*it);
        cout<<p1.first<<",  ";
        cout<<std::fixed<<cout.precision(2)<<p1.second<<";  "<<endl;
    }
}
void PrintG_M(multimap<int,double,greater<int> >& m)
{   multimap<int, double,greater<int> >::iterator it;
    for(it=m.begin();it!=m.end();it++)
    {   CustomPair p1=(pair<int, double>)(*it);
        cout<<p1.first<<",  ";
        cout<<std::fixed<<cout.precision(2)<<p1.second<<";  "<<endl;
    }
```

```
}
void PrintL_M(map<int,double,less<int> >& m)
{    map<int, double,less<int> >::iterator it;
     for(it=m.begin();it!=m.end();it++)
     {    CustomPair p1=(pair<int, double>)(*it);
          cout<<p1.first<<",  ";
          cout<<std::fixed<<cout.precision(2)<<p1.second<<";  "<<endl;
     }
}
void main()
{ map<int, double>::iterator itm;
 map<int,double,greater<int> >::iterator itmG;
 map<int,double,less<int> >::iterator itmL;
 map<int,double> m1;
 map<int,double,greater<int> >m2;
 multimap<int,double> m3;
 multimap<int,double,greater<int> >m4;
 m1.insert(CustomPair(1,2.0));
 m1.insert(CustomPair(2,5.0));
 m1.insert(CustomPair(3,7.0));
 m1.insert(CustomPair(4,8.0));
 m1.insert(CustomPair(5,11.0));
 m1.insert(CustomPair(6,6.0));
 cout<<"m1: "<<endl;
 Print(m1);
 m2.insert(CustomPair(1,2.0));
 m2.insert(CustomPair(2,5.0));
 m2.insert(CustomPair(3,7.0));
 m2.insert(CustomPair(4,8.0));
 m2.insert(CustomPair(5,11.0));
 m2.insert(CustomPair(6,6.0));
 cout<<"m2(greater<int>:)"<<endl;
 PrintG(m2);
 m3.insert(CustomPair(1,2.0));
 m3.insert(CustomPair(2,5.0));
 m3.insert(CustomPair(3,7.0));
 m3.insert(CustomPair(4,8.0));
 m3.insert(CustomPair(5,11.0));
 m3.insert(CustomPair(6,6.0));
 cout<<"m3:"<<endl;
PrintM(m3);
 m4.insert(CustomPair(1,2.0));
 m4.insert(CustomPair(2,5.0));
 m4.insert(CustomPair(3,7.0));
 m4.insert(CustomPair(4,8.0));
 m4.insert(CustomPair(5,11.0));
 m4.insert(CustomPair(6,6.0));
 cout<<"m4(greater<int>:)"<<endl;
 PrintG_M(m4);
 map<int,double>::allocator_type ma;
 ma=m2.get_allocator();
 map<int,double> m5(less<int>(),ma);
```

```
    m5.insert(CustomPair(16,1.0));
    m5.insert(CustomPair(15,7.0));
    m5.insert(CustomPair(24,9.0));
    m5.insert(CustomPair(23,13.0));
    m5.insert(CustomPair(32,21.0));
    m5.insert(CustomPair(11,27.0));
    cout<<"m5(less<int>:)"<<endl;
    PrintL_M(m5);
   }
```

例 3-32 的执行效果见图 3-33。

总结：在例 3-32 中，对于 map/multimap 的多种定义形式，均进行了尝试。在输出函数中，还使用了格式化输出的方法。读者要多体会。

2. map 的容量

在 map 和 multimap 中，定义了两个成员函数 size()和 max_size()，用来确定 map 和 multimap 的数据成员的数量。两个函数的语法如下：

```
size_type size() const;
size_type max_size() const;
```

函数的使用方法见例 3-33。

例 3-33

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
void main()
{map<int,double,greater<int> >m1;
 multimap<int,double,greater<int> >m2;
 typedef pair<int,double> mypair;
 m1.insert(mypair(1,5.0));
 m1.insert(mypair(2,7.0));
 m1.insert(mypair(3,7.0));
 m1.insert(mypair(4,17.0));
 m1.insert(mypair(5,11.0));
 m2.insert(mypair(1,4.0));
 m2.insert(mypair(2,8.0));
 m2.insert(mypair(3,9.0));
 m2.insert(mypair(4,12.0));
 m2.insert(mypair(5,19.0));
 int size=m1.size();                                  //计算 size
 int max_size=m1.max_size();                          //计算 max_size
 cout<<"the size of map m1 : "<<size<<", the max_size of map m1 : "<<max_size<<endl;
 int m_size=m2.size();
 int m_max_size=m2.max_size();
 cout<<"the size of multimap m2 : "<<m_size<<", the max_size of multimap m2 : "<<m_max_
size<<endl;
 }
```

例 3-33 的执行效果见图 3-34。

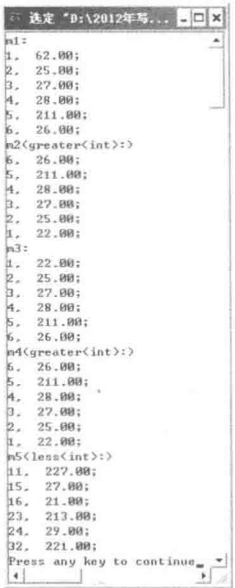


图 3-33　例 3-32 的执行效果

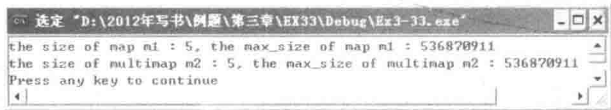


图 3-34　例 3-33 的执行效果

3.8.2　map 和 multimap 成员函数

1. 判断空函数

判断一个 map/multimap 是否为空是很重要的。如果 map 或者 multimap 为空，成员函数 empty() 返回 true。函数的语法如下：

```
bool empty() const;
```

2. 遍历容器

容器 map 和 multimap 不支持元素直接存取。元素的存取需要经过迭代器实现，并且 map 和 multimap 的迭代器均是双向迭代器。此类成员函数包括：begin()，end()，rbegin()，rend()。函数原型如下：

```
const_iterator begin() const;
iterator begin();
const_iterator end() const;
iterator end();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
const_reverse_iterator rend() const;
reverse_iterator rend();
```

下面用例 3-34 说明上述 4 个函数的使用。

例 3-34

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef pair<int,double> mypair;
void Origin(map<int,double,greater<int> >& m,multimap<int,double,greater<int> >& mm)
```

```
{m.insert(mypair(1,5.0));
 m.insert(mypair(2,7.0));
 m.insert(mypair(3,7.0));
 m.insert(mypair(4,17.0));
 m.insert(mypair(5,11.0));
 mm.insert(mypair(1,4.0));
 mm.insert(mypair(2,8.0));
 mm.insert(mypair(3,9.0));
 mm.insert(mypair(4,12.0));
 mm.insert(mypair(5,19.0));
}
void Print(map<int,double,greater<int> >& m,multimap<int,double,greater<int> >& mm)
{map<int,double,greater<int> >::iterator it;
 multimap<int,double,greater<int> >::iterator itM;
 int size=m.size();
 cout<<"map: "<<endl;
 for(it=m.begin();it!=m.end();it++)
 {  mypair temp=*it;
    cout<<temp.first<<", "<<temp.second<<", "<<endl;
 }
 int Msize=mm.size();
 cout<<"multimap: "<<endl;
 for(itM=mm.begin();itM!=mm.end();itM++)
 {  mypair temp=*itM;
    cout<<temp.first<<", "<<temp.second<<", "<<endl;
 }
 cout<<endl;
}
void main()
{map<int,double,greater<int> >m1;
 multimap<int,double,greater<int> >m2;
 map<int,double,greater<int> >::iterator it;
 map<int,double,greater<int> >::reverse_iterator itR;
 multimap<int,double,greater<int> >::iterator itM;
 multimap<int,double,greater<int> >::reverse_iterator itRM;
 Origin(m1,m2);
 Print(m1,m2);
 it=m1.begin();
 mypair tmp=*it;
 cout<<"The first element is : "<<tmp.first<<", "<<tmp.second<<";"<<endl;
 it=m1.end();
 tmp=*(--it);
 cout<<"The last element is : "<<tmp.first<<", "<<tmp.second<<";"<<endl;
 itM=m2.begin();
 tmp=*itM;
 cout<<"The first element is : "<<tmp.first<<", "<<tmp.second<<";"<<endl;
 itM=m2.end();
 tmp=*(--itM);
 cout<<"The last element is : "<<tmp.first<<", "<<tmp.second<<";"<<endl;
 itR=m1.rbegin();
 tmp=*itR;
 cout<<"The reverse first element is : "<<tmp.first<<", "<<tmp.second<<";"<<endl;
```

```
    itR=m1.rend();
    tmp=*(--itR);
    cout<<"The reverse last element is : "<<tmp.
first<<", "<<tmp.second<<";"<<endl;
    itRM=m2.rbegin();
    tmp=*itRM;
    cout<<"The reverse first element is : "<<tmp.
first<<", "<<tmp.second<<";"<<endl;
    itRM=m2.rend();
    tmp=*(--itRM);
    cout<<"The reverse last element is : "<<tmp.
first<<", "<<tmp.second<<";"<<endl;
    cin.get();
}
```

例 3-34 的执行效果见图 3-35。

总结：在例 3-34 中，主要讲述了 4 种和迭代器相关的函数的使用方法。请读者关注数据类型 pair 的使用方法。

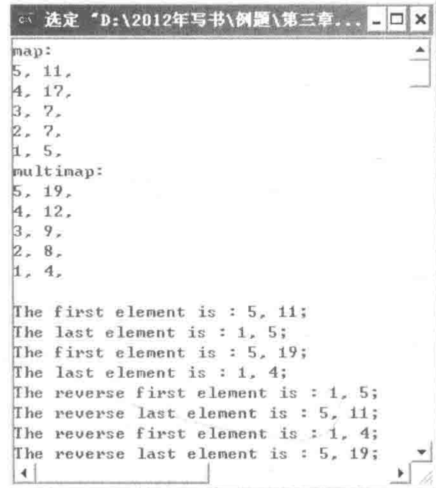


图 3-35　例 3-34 的执行效果

3.8.3　map 和 multimap 的高级编程

本节主要讲述 map/multimap 的插入和删除、元素交换函数 swap()、元素个数统计、查找元素和元素的随机访问、元素大小比较、获取内存分配器、map 作为关联式数组、通常数组可作为 STL 容器。

1. map/multimap 的插入和删除

容器 map/multimap 的成员函数 insert()可以加入一个对象、一个对象的若干复制，或者某个范围内的对象。函数 insert()把一个或若干个元素插入 iterator 指示的位置。所插入的元素将出现在 iterator 指出的位置之前。函数 insert()的原型为：

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
void insert(const value_type *first, const value_type *last);
```

第 1 种语法的功能是将元素 x 插入 map 的末尾，第 2 种语法的功能是将元素 x 插入迭代器 it 之前，第 3 种语法的功能是将迭代器(first, last)指定范围的元素插入 map 中。

函数 erase()的原型如下：

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

函数 erase()可以用来删掉由一个 iterator 指出的元素，也可以删掉一个范围的元素。第 1 种语法的功能是将迭代器 it 所指向的元素删除；第 2 种语法的功能是将迭代器(first,end)所指范围中的元素全部删除；第 3 种语法的功能是将元素 key 删除。

同样，容器 map 还提供了成员函数 clear()。

```
void clear() const;
```

函数 clear()可以删除容器中的所有元素，并且比函数 erase()的管理功能更强大而且更加灵活。函数 clear()等效于：

```
erase(map::begin(), map::end())
```

下面使用例 3-35 来详细讲解对于容器 map/multimap 的插入和删除操作。

例 3-35

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef pair<int,double>mypair;
void print( map<int,double,greater<int> >& m)
{  map<int,double,greater<int> >::iterator it;
   mypair tmp;
   int size=m.size();
  if (size<=0)
    cout<<"******"<<endl;
  else
  {  for(it=m.begin();it!=m.end();it++)
     {
     tmp=(mypair)*it;
     cout<<tmp.first<<"; "<<tmp.second<<endl;
     }
     cout<<endl;
  }
}
void main()
{map<int,double,greater<int> >m1;
 map<int,double,greater<int> >m3;
 multimap<int,double,greater<int> >m2;
 multimap<int,double,greater<int> >::iterator itmA,itmB;
 m1.insert(mypair(1,5.0));
 m1.insert(mypair(2,10.5));
 m1.insert(mypair(3,11.5));
 m1.insert(mypair(4,13.5));
 m1.insert(mypair(5,15.1));
 cout<<"m1: "<<endl;
 print(m1);
 m2.insert(mypair(1,9.5));
 m2.insert(mypair(2,12.5));
 m2.insert(mypair(3,9.2));
 m3=m1;
 int size=m1.size();
 cout<<"The size of map m1: "<<size<<endl;
 size=m2.size();
 cout<<"The size of map m2: "<<size<<endl;
 m3.erase(m3.begin());
 cout<<"m3: "<<endl;
 print(m3);
    m3.erase(m3.begin(),--m3.end());
 cout<<"m3: "<<endl;
    print(m3);
 m1.erase(2);
 cout<<"m1: "<<endl;
    print(m1);
```

```
    m3.clear();
    cout<<"m3: "<<endl;
    print(m3);
}
```

例 3-35 的执行效果见图 3-36。

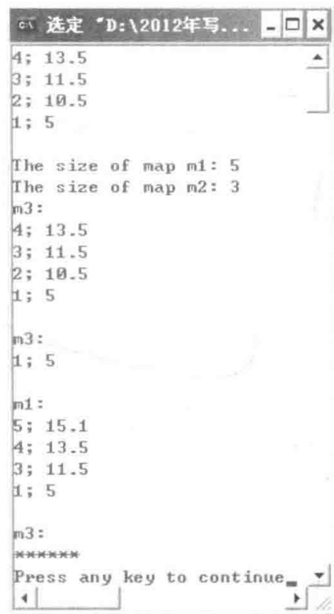


图 3-36　例 3-35 的执行效果

2．元素交换函数 swap()

map 和 multimap 还提供了元素交换函数 swap()。参与交换的两个对象必须类型一致。函数原型如下：

```
void swap(map& str);
```

无论在 map 还是 multimap 中，函数 swap()的使用方法是一致的。在实际编程中，需要把两个变量进行交换。函数 swap()一般包括两种形式：swap(m1,m2)和 m1.swap(m2)。值得注意的是，第二种方法只是 m1 的内容发生改变。修改例 3-35，在源代码最后添加以下代码：

```
m1.swap(m3);
cout<<"m1: "<<endl;
print(m1);
cout<<"m3: "<<endl;
print(m3);
```

之后，即可以实现容器 m1 和容器 m2 的内容交换。

3．元素个数统计、查找元素和元素的随机访问

count()函数的功能是返回元素在 map /multimap 中出现的次数。对于 map 型容器，其元素具有唯一性，不包含重复的元素，函数返回值只有两个："0" 或者 "1"。对于 multimap 型容器，函数功能和 map 型是一致的，唯一区别在于：multimap 允许数据重复，函数 count()的返回值不仅仅局限于 0 和 1，函数会返回元素的重复个数。count()的原型如下：

```
size_type count(const Key& key) const;
```

实现查找元素功能的成员函数是 find()。其功能是返回指向 key 的迭代器，如果该迭代器定义的是常迭代器，返回的类型是 const_iterator。如果和键值 key 对应的元素不存在，则函数返回迭代器 end()。还可以使用 STL 的通用算法 find()和 find_if()。这两个函数可以使用迭代器 iterator。通常第一个迭代器指着开始的位置，第二个迭代器指着停止处理的地方。并且第二个迭代器指向的元素不被处理。对于 multimap 型容器，其中存在多个相同的元素，在使用 find()函数时，返回的是第一个出现所查找到元素的迭代器。

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

元素的随机访问功能包括两个函数 upper_bound()和 lower_bound()；还有函数 equal_range()。

函数 equal_range()实现的功能是返回容器中元素上下限的两个迭代器，由于迭代器是常迭代器，所以 equal_range()可以返回 map/multimap 容器元素中上下限的两个常迭代器，如果元素没找到，则返回最后一个元素，函数返回的是 end()。函数返回的两个迭代器中，第一个迭代器和函数 lower_bound()返回值一样；第二个迭代器和函数 upper_bound()返回值一样。

函数 upper_bound(Key key)返回值是返回指向大于 key 元素的迭代器；lower_bound()的功能在于返回指向 key 前面元素的迭代器，即 key 是“界限”。

上述 3 个函数的原型如下：

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>equal_range(const Key& key) const;
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

下面以例 3-36 讲解上述几个函数的使用方法。

例 3-36

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
#include <algorithm>
using namespace std;
typedef pair<int,double>mypair;
map<int,double,greater<int> >::iterator it1;
multimap<int,double,greater<int> >::iterator it2;
void print(map<int,double,greater<int> >& m)
{    mypair tmp;
     for(it1=m.begin();it1!=m.end();it1++)
     {  tmp=(mypair)*it1;
        cout<<" "<<tmp.first<<";  "<<tmp.second<<endl;
     }
     cout<<endl;
}
void printM(multimap<int,double,greater<int> >& m)
{    mypair tmp;
     for(it2=m.begin();it2!=m.end();it2++)
     {  tmp=(mypair)*it2;
        cout<<" "<<tmp.first<<";  "<<tmp.second<<endl;
     }
     cout<<endl;
}
void main()
{
     typedef map<int,double,greater<int> > Map;
     typedef multimap<int,double,greater<int> > MMap;
     Map m1;
     MMap m2;
     mypair temp;
     pair<Map::iterator,Map::iterator> p1;
     pair<MMap::iterator,MMap::iterator>p2;
     m1.insert(mypair(1,5.6));
     m1.insert(mypair(2,8.2));
     m1.insert(mypair(3,9.5));
     m1.insert(mypair(4,10.5));
     m1.insert(mypair(5,4.5));
     print(m1);
     m2.insert(mypair(1,5.6));
     m2.insert(mypair(2,8.2));
     m2.insert(mypair(2,9.5));
```

```
        m2.insert(mypair(4,10.5));
        m2.insert(mypair(4,4.5));
        printM(m2);
        int size=m1.count(2);
        int Msize=m2.count(4);
        cout<<"The count of key 2 in the map m1: "<<size<<endl;
        cout<<"The count of key 4 in the map m2: "<<Msize<<endl;
        it1=m1.find(5);
        temp=*it1;
        cout<<"The element of key 5:---- "<<temp.first<<"; "<<temp.second<<endl;
        it2=m2.find(2);
        temp=*it2;
        cout<<"The element of key 2:---- "<<temp.first<<"; "<<temp.second<<endl;
        p1=m1.equal_range(3);
        cout<<"The lower_bound(3) and upper_bound(3):---- "<<p1.first->second<<";  "<<p1.
second->second<<endl;
        p2=m2.equal_range(4);
        cout<<"The lower_bound(4) and upper_bound(4):---- "<<p2.first->second<<";  "<<p2.
second->second<<endl;
        it1=m1.lower_bound(4);
        temp=*it1;
        cout<<"The lower_bound(4):---- "<<temp.first<<";  "<<temp.second<<endl;
        it2=m2.upper_bound(5);
        temp=*it2;
        cout<<"The upper_bound(5):---- "<<temp.first<<";  "<<temp.second<<endl;
    }
```

例 3-36 的执行效果见图 3-37。

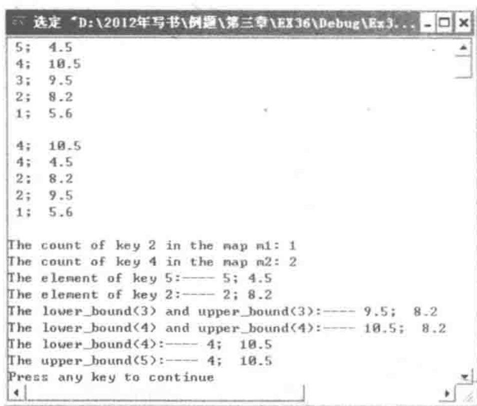


图 3-37　例 3-36 的执行效果

4. 元素大小比较

序列式容器中，可采用<、>等操作符函数进行容器大小比较。对于 map/multimap，同样提供了一系列操作运算符函数：==， <， !=， >， >= ， <=。还提供了特殊的函数，主要有 key_comp 和 value_comp。

（1）键值比较

通过函数 key_comp 可以进行键值的比较，返回类型为 key_comp，key_comp 决定了 map 中元素的排列顺序。函数 key_comp 的原型如下：

```
key_compare key_comp() const;
```

下面用例 3-37 对 key_comp 的使用方法进行阐释。

例 3-37

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
void main()
{  typedef  map<int,double,less<int> > MAP;
   typedef  multimap<int,double, greater<int> > M_MAP;
   MAP m1;
   M_MAP m2;
   MAP::key_compare kc1=m1.key_comp();
   M_MAP::key_compare kc2=m2.key_comp();
   bool r=kc1(2,3);
   bool M_r=kc2(3,4);
   if(r)
   { cout<<"kc1(2,3)返回值为 true. "<<endl;
   }
   else
   { cout<<"kc1(2,3)返回值为 false. "<<endl;
   }
   if(M_r)
   {  cout<<"kc2(2,3)返回值为 true. "<<endl;
   }
   else
   {  cout<<"kc2(2,3)返回值为 false. "<<endl;
   }
}
```

例 3-37 的执行效果见图 3-38。在例 3-37 中，由于容器 m1 是按 less 规则排列的，容器 m2 是按 greater 规则排列的，所以函数 kc1 返回值为 true，函数 kc2 返回值为 false。

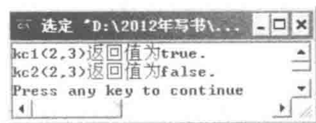


图 3-38　例 3-37 的执行效果

（2）实值比较

通过函数 value_comp 可以进行实值比较，返回类型为 value_compare。value_compare 决定了 map 中元素的排列顺序，函数 value_compare 的原型如下：

```
value_compare value_comp() const;
```

其用法和 key_comp()类似，首先定义键值类型 vc1 和 vc2，之后通过 vc1 和 vc2 进行排列顺序的测试。对于按 less 规则排序的 map/multimap 容器和对于按 greater 规则排序的 map/multimap 容器，分别产生不同的返回值，具体应用见例 3-38。

例 3-38

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
typedef  map<int,double,less<int> > MAP;
typedef  multimap<int,double,greater<int> > M_MAP;
typedef  pair<MAP::iterator, bool> mypair;
```

```
typedef  pair<int,double> myp;
void print(MAP& m)
{    MAP::iterator it;
     myp  temp_p;
     for(it=m.begin();it!=m.end();it++)
     {    temp_p=(*it);
          cout<<temp_p.first<<", "<<temp_p.second<<endl;
     }
     cout<<endl;
}
void print_M(M_MAP& m)
{    M_MAP::iterator it;
     myp  temp_p;
     for(it=m.begin();it!=m.end();it++)
     {    temp_p=(*it);
          cout<<temp_p.first<<", "<<temp_p.second<<endl;
     }
     cout<<endl;
}
void main()
{    MAP m1;
     M_MAP m2;
     M_MAP::iterator itA,itB;
     mypair p1,p2;
     MAP::value_compare vc1=m1.value_comp();
     M_MAP::value_compare vc2=m2.value_comp();
     p1=m1.insert(map<int,double>::value_type(1,10));
     p2=m1.insert(map<int,double>::value_type(2,5));
     cout<<"输出容器m1: "<<endl;
     print(m1);
     if(vc1(*p1.first,*p2.first)==true)
     {    cout<<" 元素(1,10)在元素(2,5)之前"<<endl;
     }
     else
     {    cout<<" 元素(1,10)在元素(2,5)之后"<<endl;
     }
     itA=m2.insert(multimap<int,double>::value_type(1,20));
     itB=m2.insert(multimap<int,double>::value_type(2,50));
     cout<<"输出容器m2: "<<endl;
     print_M(m2);
     if(vc2(*itA,*itB)==true)
     {    cout<<" 元素(1,20)在元素(2,50)之前"<<endl;
     }
     else
     {    cout<<" 元素(1,20)在元素(2,50)之后"<<endl;
     }
}
```

例 3-38 的执行效果见图 3-39。

总结：在例 3-37 和例 3-38 中，主要讲述了和排序相关的两个函数 key_comp 和 value_comp，它们的使用方法大概类似。这两个函数的主要作用是判断容器中元素的排序规则。

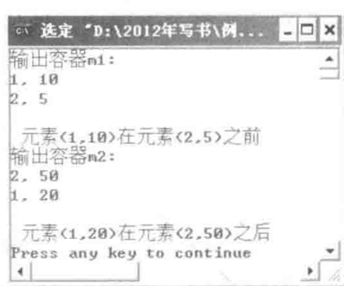


图 3-39　例 3-38 的执行效果

5. 获取内存分配器

函数 get_allocator()用于返回 map 或 multimap 的内存配置器，内存配置器类似指针的首地址，用于指明对象的初始存储位置，获取容器的内存分配器至关重要。函数的原型如下：

```
Allocator get_allocator() const;
```

函数只能读取内存分配器，不能改变内存分配器，即不能改变容器变量所存储的位置量。用例 3-39 对配置器进行简单介绍。

例 3-39

```
#pragma warning(disable:4786)
#include <iostream>
#include <map>
using namespace std;
void main()
{typedef map<int,double> MAP;
MAP::allocator_type m1_alloc;
MAP::allocator_type m2_alloc;
MAP::allocator_type m3_alloc;
MAP::allocator_type m4_alloc;
MAP m1,m2,m3;
m1_alloc=m1.get_allocator();
m2_alloc=m2.get_allocator();
m3_alloc=m3.get_allocator();
MAP m4(less<int>(),m1_alloc);
m4_alloc=m4.get_allocator();
if(m1_alloc==m4_alloc)
{    cout<<"配置器相同！ "<<endl;
}
else
{    cout<<"配置器不同. "<<endl;
}
}
```

例 3-39 的执行效果见图 3-40。

6. map 作为关联式数组

通常关联式容器不提供元素的直接存取，必须依靠迭代器。但 map 型容器提供下标操作符，支持元素的直接存取。下标操作符的索引值并非元素整数位置，而是元素的键值 key。使用键值索引有时会带来诸多问题。例如：容器中不存在指定键值的元素，会导致自动安插该元素，并且新元素的默认 value 由构造函数提供。通常，所有的基本数据类型都提供了默认的构造函数，并都以 0 为初值。

7. 通常数组可作为 STL 容器

大家都知道 string 可以被视为容器，其内元素均为字符。通常的数组不是类，也不可能提供成员函数。但是可以通过普通指针，使用 STL 的算法实现对普通数组的操作，具体应用见例 3-40。

例 3-40

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

```
void main()
{int col[]={1,5,7,2,8};
sort(col,col+5);
copy(col,col+5,ostream_iterator<int>(cout," "));
cout<<endl;
}
```

例 3-40 的执行效果见图 3-41。

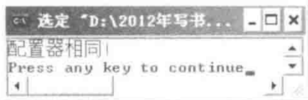


图 3-40　例 3-39 的执行效果

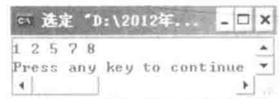


图 3-41　例 3-40 的执行效果

当然，除了上述方法之外，还可以通过自定义类或者自定义类模板的形式，实现对数组的访问和操作。

总结：本节主要讲述了 4 种关联式容器：set， multiset， map，multimap。分别详细地讲述了各种容器的各种成员与函数及其使用。对每个容器均认真讲述了其构造函数、容量大小、遍历容器、元素查找、插入和删除、交换等。同时在讲述知识点的同时，也适当补充了部分其他的知识。例如：算法 for_each 的使用和算法 sort()的使用等。

3.9　特殊容器用法

STL 提供了部分比较特殊的容器，并涉及部分数据结构的算法。这些容器模板主要包括：位段类模板、堆栈类模板、队列类模板和优先队列模板。本节主要讲述这些类模板的使用方法。读者尤其应该关注位段类模板、堆栈类模板和队列类模板的学习。

3.9.1　bitset 类模板

在例 3-1 中简单介绍了 bitset 类，本节专门讲解 bitset 类的使用。bitset 创造一个内含位或布尔值且大小固定的数组（array）。当需要管理各种标识，并需要以标识的任意组合表现变量时，即可使用 bitset 类模板。在 C 语言或者 C++语言中，通常使用型别 long 来作为 bits array，再通过&，!，~等位操作符，操作各个位。bitset 类的优点在于可容纳任意个数的位，并提供各项操作。将 bitset 作为 0 和 1 组成的序列，进行读写。

对于既定的 bitset 变量或对象，其内位的个数是不能改变的。bitser 类中变量或对象的位的数量是由模板参数决定的。如果需要使用可变长度的容器，可考虑使用模板 vector<bool>。

模板参数并不是一个型别，而是一个无符号型整数。

如果在定义对象时，使用的模板参数不同，模板的型别就会不同。因此，在对 bitset 型对象进行比较和组合时，两个对象或多个对象的位个数必须相同。类模板 bitset 的构造函数包括以下形式：

```
bitset();                                          //位段的构造函数
bitset(unsigned long val);                         //位段的构造函数
explicit bitset(const string& str,size_t pos = 0, size_t n = -1);    //位段的构造函数
```

第 1 种形式的构造函数将复位 bitset 型对象中所有的位（例如：对 bitset 对象进行全部清零）。

第 2 种形式的构造函数的参数为整数，该数值的二进制形式对应的位为 1。

第 3 种形式的构造函数使用字符串对 bitset 对象实现初始化。第 3 种构造函数形式需要详细说明如下：

参数 str 是用来初始化对象的初始字符串；参数 pos 的含义表明：在字符串 str 中，从 pos 位置开始的子字符串“才是赋值给类 bitset 模板对象的字符串”，即赋值时采用的是 str 的某个字串；参数 n 代表的是从 pos 位置开始的、长度为 n 的子字符串，该子字符串用来给 bitset 类对象赋值，bitset 类模板的成员函数如表 3-3 所示。

表 3-3　bitset 类模板的成员函数

函数名称	功能描述
size()	返回位的个数
count()	返回位值为“1”的位个数
any()	判断是否有任何位为“1”
none()	判断是否所有的值为“0”
test(size_t idx)	判断位“idx”是否为“1”
set()	设置所有位为“1”
set(size_t idx, in value)	设定位置“idx”上的值为 value
reset()	复位所有的位为“0”
reset(size_t idx)	将位置 idx 上的位设置为“0”
flip()	反转所有位的值。
flip(size_t idx)	反转“idx”位置上的位。
to_ulong()	返回所有位代表的整数
==	判断两个 bitset 型对象是否“相等”
! =	判断两个 bitset 型对象是否“不相等”
^=	异或位操作
|=	位或操作
&=	逻辑与操作
<<=	所有位向左移动 n 个位置
>>=	所有位向右移动 n 个位置
[size_t idx]	返回 idx 位置上的位值
=	赋值
~	返回该位的补码（反转值）
~()	反转原对象的位，作为产生新对象的初值
<<()	向左移动原对象 n 个位置，作为产生新对象的初值
>>()	向右移动原对象 n 个位置，作为产生新对象的初值
&()	两个 bitset 对象逐一“与”操作
|()	两个 bitset 对象逐一“或”操作
^()	两个 bitset 对象逐一“异或”操作
to_string()	以字符串形式表示 bitset 对象中的二进制

例 3-41 用于阐释 bitset 的使用。

例 3-41

```
#include <iostream>
#include <bitset>
#include <string>
using namespace std;
void print(bitset<16>& b)                                  //输出位段的内容
{  int i=0;
   int bsize=b.size();
   for(i=0;i<bsize;i++)
   {   cout<<b[i];
   }
   cout<<"   :"<<"The size of bitset :"<<bsize<<endl;
}
void main()
{  string str="0011111111111111110000";                     //定义字符串
   bitset<16> b1;                                          //定义位段 b1
   bitset<16> b2(25);                                      //定义位段 b2
   bitset<16> b3(str,2,16);                                //定义位段 b3
   print(b1);
   print(b2);
   print(b3);
   int c1=b1.count();                                      //统计"1"的个数
   int c2=b2.count();
   int c3=b3.count();
   cout<<"b1 's count : "<<c1<<", "<<"b2 's count : "<<c2<<", "<<"b3 's count : "<<c3<<", "<<endl;
   bool l1=b1.any();                                       //判断位段中是否至少包含一个"1"
   bool l2=b2.any();
   bool l3=b3.any();
   cout<<"b1 's any() : "<<l1<<", "<<"b2 's any() : "<<l2<<", "<<"b3 's any() : "<<l3<<", "<<endl;
   bool n1=b1.none();                                      //判断位段中是否全部为'0'
   bool n2=b2.none();
   bool n3=b3.none();
   cout<<"b1 's none() : "<<n1<<", "<<"b2 's none() : "<<n2<<", "<<"b3 's none() : "<<n3<<", "<<endl;
   bool t1=b1.test(2);                                     //判断改位是'1'还是'0'
   bool t2=b2.test(2);
   bool t3=b3.test(2);
   cout<<"b1 's test() : "<<t1<<", "<<"b2 's test() : "<<t2<<", "<<"b3 's test() : "<<t3<<", "<<endl;
   cout<<"set(5): "<<endl;
   b1.set(5,1);                                            //设置"位"
   b2.set(5,1);
   b3.set(5,1);
   print(b1);
   print(b2);
   print(b3);
   cout<<"reset(5): "<<endl;
   b1.reset(5);                                            //复位
   b2.reset(5);
   b3.reset(5);
```

```
    print(b1);
    print(b2);
    print(b3);
    b1.flip();                                           //取反
    b2.flip();
    b3.flip();
    cout<<"flip: "<<endl;
    print(b1);
    print(b2);
    print(b3);
    unsigned long ul1= b1.to_ulong();                    //尾段转换成数值
    unsigned long ul2= b2.to_ulong();
    unsigned long ul3= b3.to_ulong();
    cout<<"to_ulong : "<<endl;
    cout<<ul1<<endl;
    cout<<ul2<<endl;
    cout<<ul3<<endl;
    string s1=b1.to_string(); //位段转换成字符串
    string s2=b2.to_string();
    string s3=b3.to_string();
    cout<<"to_string"<<endl;
    cout<<s1.c_str()<<endl;
    cout<<s2.c_str()<<endl;
    cout<<s3.c_str()<<endl;
}
```

例 3-41 的执行效果见图 3-42。

图 3-42　例 3-41 的执行效果

总结：bitset 类模板有些类似于 C 语言中的"位段"，只不过 bitset 是以类模板的形式存在的，可以实例化为 bitset 型类或者 bitset 型容器。STL 为 bitset 提供了一系列的成员函数，便于对该型容器的操作和访问。本小节读者应该重点关注一下 bitset 型容器的用途，之后掌握 bitset 型容器的具体使用方法。

3.9.2　stack 类模板

堆栈是大家熟悉的数据结构。本小节讲述模板库（STL）中的堆栈算法。

1．stack 概述

类模板 stack（栈）实例化一个栈容器（后进先出，LIFO）或栈对象。使用函数 push()将任意数量的元素置入栈（stack）中，使用函数 pop()将元素依次反序地从栈中移除。在 STL 中，使用栈（stack）必须要包含头文件<stack>。在头文件中，类模板 stack 的定义如下：

```
namespace std{template <class T, class Container=deque<T> >class stack; }
```

第 1 个 template 参数代表元素型别。带有默认值的第 2 个 template 参数用来定义 stack 内部存放元素所用的实际容器，默认采用 deque。选择 deque（没有选择 vector）的目的主要是因为 deque 移

除元素时会释放内存，并不必在重新分配时复制所有元素。

例如：

```
stack<int> s1;
```

栈 stack 将各项操作转化为内部容器的对应调用，实际上可以使用任何序列式容器（vector，list）支持 stack，并支持 back()，push_back()，pop_back()等操作即可。

```
std::stack<int, std::vector<int> >sv;
```

2. 核心接口

栈（stack）的核心接口是 3 个成员函数：push()，top()，pop()。函数 push()将元素压入栈中，函数 pop()从栈中移除元素，函数 top()返回栈内的下一个元素。

函数 push()和函数 pop()并无返回值，函数 top()却返回下一个元素，但并不移除。函数 top()和 pop()存在一个问题：当栈中为空时，调用这两个函数会导致失败，前提是栈（stack）必须非空。

类模板 stack 为便于操作和访问，还提供了其余几个函数：size()， empty()，getallocator()等。

下面用例 3-42 对 stack 类模板进行了具体说明。

例 3-42

```
#pragma warning(disable: 4786)
#include <iostream>
#include <stack>
#include <string>
#include <list>
#include <assert.h>
using namespace std;
void main()
{ stack<string,list<string> > s1;
 s1.push("Haidian District.");
 cout<<"\"Haidian District.\" are pushed into the stack.  "<<endl;
 s1.push("Chaoyang District.");
 cout<<"\"Chaoyang District.\" are pushed into the stack.  "<<endl;
 s1.push("Xuanwu District.");
 cout<<"\"Xuanwu District.\" are pushed into the stack.  "<<endl;
 s1.push("Xicheng District.");
 cout<<"\"Xicheng District.\" are pushed into the stack.  "<<endl;
 assert(s1.size()==4);
 assert(s1.top()=="Xicheng District.") ;
 cout<<"The Elements are poped from the stack."<<endl;
 while(s1.size())
    {   cout<<"  "<<s1.top()<<endl;
    s1.pop();
    }
}
```

例 3-42 的执行效果见图 3-43。

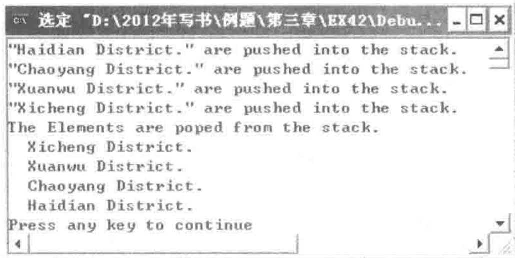


图 3-43　例 3-42 的执行效果

总结：除使用 STL 库定义的类模板之外，程序员还可以根据实际情况，创建自定义的 stack 类模板。

3.9.3　队列 queue 类模板

队列也是常见的数据结构算法。本小节讲述模板库（STL）中的队列算法。

1．队列 queue 概述

类模板 queue 可以实例化队列（queue），即 FIFO（先进先出）型队列。队列（queue）和栈（stack）的不同之处在于：队列是双端的，元素压入时是从一端，元素移除时是从另一端；而栈是单端口的容器，元素的压入和移除均是从同一端口。通俗地讲，queue 是典型的数据缓冲区结构。

为便于使用队列（queue），必须包含头文件<queue>。在头文件<queue>中，类模板 queue 的定义如下：

```
namespace std{template <class T, class Container=deque<T> > class queue;}
```

和 stack 一样，类模板声明中的第一个参数表示元素的型别，第二个元素用来定义队列内部用于存放元素的容器类别，默认值为 deque。最简单的 queue 实例化容器的例子如下：

```
std::queue<std::string> buffer;
```

实际上 queue 是单纯地把各项操作转化为内部容器的对应调用。可以使用任何序列式容器来支持 queue，前提是该序列式容器要包括成员与函数：front()，back()，push_back()，pop_front()等。假定使用序列式 list 型容器来容纳元素，队列的声明形式可如下所示：

```
std::queue<std::string, std::list<std::string> >buffer;
```

2．核心成员函数

queue 的核心接口主要由成员函数 push()，front()，back()，pop()构成。成员函数 push()会将元素压入队列（queue）中；front()返回队列中的第 1 个元素；back()会返回 queue 中最后 1 个元素，pop()会从 queue 中移除 1 个元素。

和 stack 一样，queue 型容器的成员函数 pop()在移除元素时，并不返回该元素。front()和 end()返回队列中对应的元素，但并不移除或移动该元素。

和 stack 一样，当 queue 型容器中元素个数为 0 时，使用函数 front()，back()和 pop()会导致调用失败。在使用这些函数之前，最好先调用 size()或 empty()函数，判断容器是否为空。

3．使用队列 queue 型容器

本节主要讲述了如何使用 queue 型容器。使用两个例子进行详细的说明：例 3-43 用于阐述 STL 中队列（queue）的使用；例 3-44 用于阐述如何使用自定义的队列。

例 3-43　STL 中队列（queue）的使用

```
#pragma warning(disable:4786)
#include <iostream>
#include <queue>
#include <string>
#include <list>
#include <assert.h>
using namespace std;
void main()
{ queue<string,list<string> > q1;                          //使用 list 型容器
 q1.push("Haidian District.");                             //压入队列
 cout<<"\"Haidian District.\" are pushed into the queue.  "<<endl;
 q1.push("Chaoyang District.");                            //压入队列
 cout<<"\"Chaoyang District.\" are pushed into the queue.  "<<endl;
 q1.push("Xuanwu District.");                              //压入队列
 cout<<"\"Xuanwu District.\" are pushed into the queue.  "<<endl;
 q1.push("Xicheng District.");
 cout<<"\"Xicheng District.\" are pushed into the queue.  "<<endl;
 assert(q1.size()==4);                                     //断言
 assert(q1.front()=="Haidian District.") ;                 //断言
 cout<<"The Elements are poped from the stack by order."<<endl;
 while(q1.size())
 {   cout<<"  "<<q1.front()<<endl;
     q1.pop();                                             //取出缓冲区中的元素
 }
}
```

例 3-43 的执行效果见图 3-44。

下面使用例 3-44 来进行自定义队列的使用说明。当然 STL 中标准的 queue 型队列通常运行速度较快，在使用时，方便性和安全性均较高。但是程序员仍然可以自己编写自定义式的队列。例 3-44 选自《C++标准程序库自修教程与参考手册》（侯捷/孟岩 译）。

例 3-44　如何使用自定义队列

```
#pragma warning(disable:4786)
#include <iostream>
#include "queue.cpp"
#include <list>
#include <string>
#include <assert.h>
using namespace std;
void main()
{   try{
 Queue<string> q1;
 q1.push("Haidian District.");
 cout<<"\"Haidian District.\" are pushed into the queue.  "<<endl;
 q1.push("Chaoyang District.");                            //压入队列
 cout<<"\"Chaoyang District.\" are pushed into the queue.  "<<endl;
     q1.push("Xuanwu District.");
 cout<<"\"Xuanwu District.\" are pushed into the queue.  "<<endl;
 q1.push("Xicheng District.");
 cout<<"\"Xicheng District.\" are pushed into the queue.  "<<endl;
 assert(q1.size()==4);                                     //宏断言
 assert(q1.front()=="Haidian District.") ;
 cout<<"The Elements are poped from the queue by order."<<endl;
```

```
    while(q1.size())
    {   cout<<"  "<<q1.front()<<endl;
        q1.pop();                                        //弹出队列
    }
  }
  catch(const exception& e)
  {   cerr<<"Exception: "<<e.what()<<endl;
  }
}
```

例 3-44 的执行效果见图 3-45。

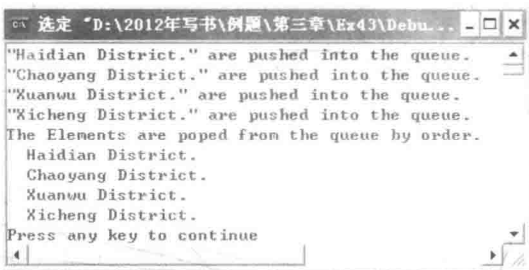


图 3-44　例 3-43 的执行效果

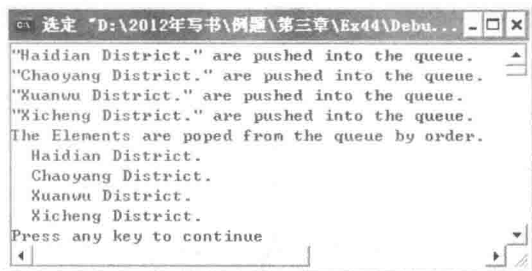


图 3-45　例 3-44 的执行效果

其中自定义模板类 queue 在头文件 queue.cpp 中。queue.cpp 的源代码如下：

```
//queue.cpp
#ifndef QUEUE_CPP
#define QUEUE_CPP
#include <list>
#include <exception>
template <class T>
class Queue{
protected:
    std::list<T> c;
public:
    class ReadEmptyQueue:public std::exception{
    public:
        virtual const char* what() const throw()
        { return "Read Empty Queue.";
        }
    };
    typename std::list<T>::size_type size() const{
        return c.size();
    }
bool empty() const
{   return c.empty();
}
void push(const T& elem)
{   c.push_back(elem);
}
T pop()
{   if(c.empty())
    {   throw ReadEmptyQueue();
    }
    T elem(c.front());
```

```
        c.pop_front();
        return elem;
    }
    T& front()
    {   if(c.empty())
        {   throw(ReadEmptyQueue());
        }
        return c.front();
    }
};
#endif
```

总结：本节主要讲述了如何使用队列 queue 以及 queue 的特性。本节最后讲述如何自定义队列。

3.9.4 Priority Queues 类模板

本节主要讲述标准模板库（STL）中的优先队列容器及相关的算法。

1．优先队列概述

优先队列（priority_queue）类模板可实例化 queue 型容器，其中元素根据优先级被读取。类模板的接口和 queue 非常接近。在此类队列中，元素并不是按顺序存储在容器中的，而是按优先级顺序存储在容器中的。即在此类队列中，被压入的元素已经按优先级进行了自动排序。默认排序准则是"<"。优先队列和常规队列同样，使用时需要包含头文件<queue>。例如：

```
#include <queue>
```

优先队列的定义如下：

```
    namespace std{template <class T, class Container = vector<T>,class Compare=less<typename
Container::value_type> >
    class priority_queue;}
```

在上述式中，第 1 个参数是元素的型别；第 2 个参数定义了内部用来存放元素的容器，默认容器是 vector；第 3 个元素定义了排序准则，默认情况是以"operator<"作为比较准则。例如：

```
std::priority_queue<float> buffer;
```

和前述几种特殊容器一样，容器 priority_queue 也是把各项操作转化为内部容器的对应调用。程序开发人员可以使用任何序列式容器来支持 priority_queue，前提是这些序列式容器要支持 front()，push_back()，pop_back()等操作。

值得一提的是，在容器 priority_queue 中需要用到 STL 中的"堆"（heap）算法，因此其内部容器必须支持随机存取迭代器。比如：deque 型容器。

如果想定义自己的排序准则，必须传递一个函数（或仿函数）作为二元判断式，用于比较两个元素并作为排序准则。

2．类模板的核心成员函数

优先队列 priority_queue 的核心接口主要由成员函数 push()，top()，pop()组成。这 3 个函数的使用和前述的几种特殊容器相同。当容器中为空时，调用 top()和 pop()时会失败。

和前面几种容器不同的是，有限型队列提供了多种形式的构造函数。在 STL 中，stack、bitset、queue 在各自的模板中，均提供了一种构造函数。而 priority_queue 中，提供了两种构造函数：

（1）Visual C++中提供的构造函数

```
explicit priority_queue(const Pred& pr = Pred(), const allocator_type& al = allocator_type());
priority_queue(const value_type *first, const value_type *last, const Pred& pr = Pred(), const allocator_type& al = allocator_type());
```

（2）STL 提供的构造函数

```
explicit priority_queue (const Compare& x=Compare(),const Container&=Container());
template <class InputIterator> priority_queue(InputIterator first, InputIterator last, const Compare& x=Compare(), const Container& = Container());
```

此外，priority_queue 提供了两个操作符“==”和“<”，并提供了函数 size()和 empty()。

例 3-45 用于说明 priority_queue 的使用方法。

例 3-45

```
#include <iostream>
#include <queue>
#include <vector>
#include <list>
#include <queue>
using namespace std;
void print(double& Ele)
{cout<<Ele<<", ";
}
template <class T>void Out(priority_queue <T, deque<T>,less<T> >& p)
{ while(!p.empty())
 {   cout<<p.top()<<", ";
    p.pop();                                         //弹出队列
 }
 cout<<endl;
}
template <class T>void OutG(priority_queue <T, deque<T>,greater<T> >& pg)
{ while(!pg.empty())
{   cout<<pg.top()<<", ";
   pg.pop();
 }
 cout<<endl;
}
void main()
{priority_queue <double,deque<double>,less<double> >p1,p2;
 p1.push(11.5);                                      //压入队列
 p1.push(22.5);
 p1.push(32.5);
 p1.push(21.1);
 p1.push(15.6);
 p1.push(8.9);
 p1.push(55.0);
 p2=p1;
 Out(p1);
 p1=p2;
```

```
    priority_queue <double,deque<double>,greater<double> >p3;
    while(p2.size())
     {
       p3.push(p2.top());
       p2.pop();
     }
     OutG(p3);
}
```

例 3-45 的执行效果见图 3-46。

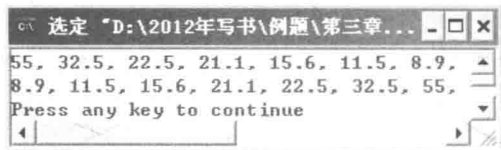


图 3-46　例 3-45 的执行效果

总结：本节主要讲述了如何使用优先队列 priority_queue，其相对于队列 queue 的不同之处在于：优先队列实现了内部自动排序，可根据实际情况自定义排序准则，也可以自己编写函数或仿函数用于内部优先级的确定。在 Visual C++开发环境中，优先队列 priority_queue 的容器可使用 vector 和 deque 两种顺序式容器，在使用顺序式容器 list 时，出现了编译错误。并且，在例 3-45 中，对两种排序准则均进行了尝试。读者应该认真阅读，举一反三。

3.10　本章小结

本章主要讲述了 4 部分内容。第 1 部分介绍了容器的概念，详细地讲解了容器的基本概念和基本原理；第 2 部分介绍了序列式容器：vector，list，deque，主要涉及这些容器的使用及各自的特殊功能；第 3 部分重点讲述了 4 种关联式容器，是本章的重点内容，其中 map 又较 set 更复杂，掌握这些关联式容器，对提高软件开发水平有较大的帮助；第 4 部分讲述了部分较特殊的容器：bitset，stack，queue，还有优先式队列（priority_queue）。

通过本章的讲述，已经把 STL 中所有的容器全部介绍完毕。本章是全书的核心。本章除例 3-44 之外，全部例题由笔者自己编写，并全部调试通过，请读者认真阅读，希望有所收获。

第 4 章 C++中的算法

以有限的步骤解决逻辑或数学上的问题，称为算法。具体的算法是指对解题方案准确而完整的描述，是一系列解决问题的清晰指令。算法是用系统的方法描述解决问题的策略机制。算法能够实现对于一定规范的输入，在有限时间内获得所要求的输出。一般情况下，算法大致分为基本算法、数据结构的算法、数论与代数算法、计算几何算法、图论算法、动态规划及数值分析、加密算法、排序算法、检索算法、随机化算法、并行算法等一系列算法。

C++标准程序库也提供了一系列的算法。算法一般被设计用来处理迭代器区间。迭代器是一个“可遍历 STL 容器内全部或部分元素”的对象。迭代器的当前值指向容器中的特定位置。STL 中算法的概念可以理解为使用迭代器处理容器中元素的方法。

4.1 算法库简介

STL 的算法一般采用“覆盖（Overwrite）”模式而不是“安插（Insert）”模式。调用时必须保证目标区间拥有足够的元素空间。当然也可以使用“安插（Insert）”型迭代器访问容器。

为了提高灵活性和功效，STL 的算法允许使用者传递自定义的操作（函数），方便由 STL 算法调用。这些操作（函数）既可以是一般函数，也可以是仿函数。如果返回值是 bool 类型，称为条件判断式。

对于算法函数设计，有两个主要的通用部分。首先，这些函数都使用模板来提供通用类型；其次，这些函数都使用迭代器来提供访问容器中数据的通用表示。所以，可以将数值存储于数组中，或者存储在链表中，还可以将对象存储在树结构中。

注意：由于指针是一种特殊的迭代器，使用时更要注意其各种用法，防止发生失误。

STL 中的算法库分成 4 种：非修改式序列算法、修改式序列算法、排序和相关算法、通用数字算法。

（1）非修改式序列算法不改动容器中元素的次序，也不改动元素值。一般通过 input 迭代器和 forward 迭代器完成工作，可用于所有的标准容器。

（2）修改式序列算法一般不直接改变容器中元素的值，或者在复制到另一区间的过程中改变元素值。修改式序列算法还包括移除性质或删除性质的算法。移除一般只是在逻辑上“移除”元素，

不改变容器的大小和容器中的元素个数。"移除"和"删除"是不同的算法。

（3）排序和相关算法包括多个排序函数和其他各种函数，包括集合操作等。C++的 STL 涵盖了所有的变序性算法。排序一般是指通过对容器中元素的赋值和交换，改变元素顺序。排序算法的复杂度通常低于线性算法，要动用随机存取迭代器。变序性算法不能以关联式容器作为目标。因为关联式容器的元素都被视为常数，不能变更。

（4）通用数字算法一般包括将区间的内容累积、内部乘积、小计、计算相邻对象差等函数。

4.2 非修改性序列算法

非修改性序列性算法是不需要使用循环而从序列中找出某些东西的基本工具。算法的操作不会改变容器中的元素值，也不会改变元素的次序。下面依次介绍：

- for each 算法。
- 元素计数算法。
- 最小值和最大值算法。
- 搜寻算法。
- 区间比较算法。

4.2.1 for each 算法

for each 算法非常灵活，可以非常方便地同时处理修改每一个元素。函数的定义形式：

```
for_each ( Iterator begin  Iterator end,  proc op  )
```

说明：for_each 算法实现对区间[begin, end]中每个元素均调用进程 op。

该函数的使用给出 3 个例题：

（1）最普通地使用算法 for_each;

（2）在使用 for_each 的例题中使用仿函数;

（3）介绍使用 for_each()的返回值。

参见例 4-1，其功能是实现将向量中的元素值输出到屏幕上。请读者关注每行代码后面的汉语注释。

例 4-1

```
#include <iostream>                                        //包含头文件 iostream
#include <vector>                                          //包含头文件 vector
#include <algorithm>                                       //包含头文件 algorithm
using namespace std;                                       //使用命名空间 std
template <class T>                                         //使用类模板定义类 T
void FillValue(T& vect, int first, int last)  //定义函数 FillValue,函数的功能是给对象
{                                                          //vect 的每个元素赋值
  if(last >= first)
  {
  for(int i=first;i<=last;++i)
    vect.insert(vect.end(),i);
  }
  else
```

```
    {
    cout<<" The indexes is error: last < first. "<<endl;
    }
}
void print(int elem)                                   //在屏幕上输出函数的参数elem
{
    cout<<elem<<" ";
}
void main()                                            //程序入口main()函数
{
    vector <int> myvector;                             //使用向量模板定义向量myvector
    FillValue(myvector, 1,10);                         //给向量赋值
    for_each(myvector.begin(),myvector.end(),print);//输出向量中的每个元素的值
    cout<<endl;                                        // 换行
}
```

例 4-1 的执行效果见图 4-1。

图 4-1　例 4-1 的执行效果

例 4-2 是利用仿函数改变容器中每个元素的值。所谓仿函数即使一个类使用时象调用函数。其实现时类中包含函数 operator()，之后这个类就有了类似函数的行为，就是一个仿函数类。当使用类似函数调用形式 classname()时，函数 operator()被自动执行。

提示：例 4-2 的倒数第 3 行代码在调用时首先产生一个 Multiple<int>型的临时对象，对象的初始赋值为 2，代码实现对向量 myvector 的每个元素的值都乘以 2。

仿函数，又称为函数对象，是 STL（标准模板库）六大组件（容器、配置器、迭代器、算法、配接器、仿函数）之一；仿函数虽然小，但却极大地拓展了算法的功能，几乎所有的算法都有仿函数版本，代码如下。

```
for_each(myvector.begin(), myvector.end(), Multiple<int>(2) );
```

例 4-2

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
    if(last >= first)
    {
    for(int i=first;i<=last;++i)
```

```
        vect.insert(vect.end(),i);
    }
    else
    {
    cout<<" The indexes is error: last < first. "<<endl;
    }
}
void print(int elem)
{
    cout<<elem<<" ";
}
template <class T>
class Multiple{                                                      //定义类 Multiple
private:
    T theValue;
public:
    Multiple(const T& v): theValue(v)
    {
    }
    void operator()(T& elem) const                                   //添加成员函数
    {  elem*=theValue;
    }
} ;
void main()
{
    vector <int> myvector;
    FillValue(myvector, 1,10);
    for_each(myvector.begin(), myvector.end(), print);
    cout<<endl;
    for_each(myvector.begin(), myvector.end(), Multiple<int>(2) );//看程序前面的解释
    for_each(myvector.begin(),myvector.end(),print);
    cout<<endl;
}
```

例 4-2 的执行效果见图 4-2。

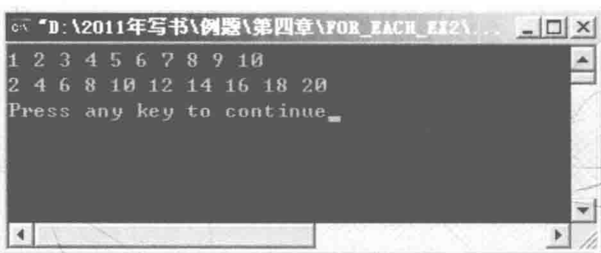


图 4-2　例 4-2 的执行效果

例 4-3 用来示范如何使用算法 for_each()的返回值。首先定义类 SUM，在 SUM 中定义函数 operator() (int elem)，用于计算容器中元素的值的和，函数 operator double()用于返回计算求得的总和。

提示：注意关注下面源代码中的注释文字。

例 4-3

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
   if(last >= first)
   {   for(int i=first;i<=last;++i)
      vect.insert(vect.end(),i);
   }
   else
   {   cout<<" The indexes is error: last < first. "<<endl;
   }
}
void print(int elem)
{
   cout<<elem<<" ";
}
class SUM{                                                    //定义 SUM 类
private:
   long sum_D;
public:
   SUM(): sum_D(0){
   }
   void operator()(int elem)                                  //实现求和功能
   {  sum_D+=elem;
   }
   operator double() {                                        //返回求解的“和”
       return static_cast<double>(sum_D);
    }
};
void main()
{
   vector <int> myvector;
   FillValue(myvector, 1,10);                                 //给向量 myvector 赋值
   for_each(myvector.begin(),myvector.end(),print);    //输出向量 myvector 中的元素
   cout<<endl;
   double sum=for_each(myvector.begin(),myvector.end(),SUM());  //求和
   cout<<"The Sum: "<<sum<<endl;
}
```

例 4-3 的执行效果见图 4-3。

图 4-3　例 4-3 的执行效果

总结：本节重点讲述了 for each 算法。通过 3 个知识点的讲解和 3 个例题的实践，既熟悉了 for_each()算法的使用，还学习了关于仿函数的知识和技巧。最需要掌握的是 for_each()算法的使用， for_each()算法的第 3 个参数采用仿函数的形式，仿函数的调用以及仿函数的返回值。

4.2.2 元素计数算法

STL 的算法库提供了元素计数的功能，即求得容器中元素的总个数。这个功能是由算法 count()来实现的。另外还提供了条件记数的算法 count_if()。

函数形式如下：

```
count( Iterator begin , Iterator end , const T& value)
count( Iterator begin, Iterator end, UnaryPredicate op)
```

说明：函数 count(Iterator begin，Iterator end) 将统计在迭代器区间[begin, end]范围之内等于 value 的元素个数；函数 count(Iterator begin，Iterator end ，UnaryPredicate op)只有当 op 参数为“真”时，才统计元素的个数。

为了说明算法 count()的使用方法，见例 4-4。

提示：值得注意的是，在例题中使用了 bind2nd(greater<int>(),2)，其意义是数值大于 2 的条件表达式。这种使用形式记住即可，随着使用次数的增多，印象会逐渐加深。

例 4-4

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)                    //赋值
{
  if(last >= first)
  {
  for(int i=first;i<=last;++i)
    vect.insert(vect.end(),i);
  }
  else
  {
  cout<<" The indexes is error: last < first. "<<endl;
  }
}
void print(int elem)                                             //输出到屏幕
{
cout<<elem<<" ";
}
bool isEven(int elem)                                            //是否是偶数
{
  return elem%2==0;
}
void main()
```

```
{
    vector <int> myvector;                                        //定义向量
    FillValue(myvector, 1,10);                                    //给向量赋值
    for_each(myvector.begin(),myvector.end(),print);              //输出到屏幕
    cout<<endl;
    int ct=count(myvector.begin(),myvector.end(),4);              //统计等于 4 的元素个数
    int ctif=count_if(myvector.begin(),myvector.end(),isEven);        //统计偶数的个数
    int ctg=count_if(myvector.begin(),myvector.end(),bind2nd(greater<int>(),2));
                                                                  //统计大于 2 的元素个数
    cout<<"Number of the element equals to 3:"<< ct<<endl;            //输出
    cout<<"Number of the elements are  larger than 1:"<<ctif<<endl; //输出
    cout<<"Number of the elements are larger than 2: "<<ctg<<endl;  //输出
    cout<<endl;
}
```

例 4-4 的运行效果见图 4-4。

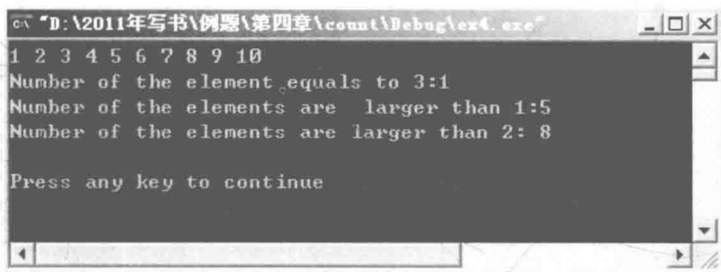


图 4-4　例 4-4 的执行效果

总结：通过例 4-4 的学习可知算法 count()可以统计任何容器（例 4-4 是向量）的元素个数，还可以通过条件表达式，统计满足相应条件的元素个数。请读者注意各种条件的表达形式和使用方法。

4.2.3　最小值和最大值算法

STL 算法库中包含了最小值算法和最大值算法，并且还提供了复杂形式的比较。函数形式如下。

最小值算法：

```
Iterator min_element(Iterator beg, Iterator end)
Iterator min_element(Iterator begin, iterator end, compFunc op)
```

最大值算法：

```
Iterator max_element(Iterator beg, Iterator end)
Iterator max_element(Iterator begin, iterator end, compFunc op)
```

上述两种形式的函数，第一种形式是以“operator<”进行元素比较；第二种形式中参数 op 用来比较两个元素 op（elem1, elem2），如果第一个元素小于第二个元素，应当返回 true，如果存在多个最小值或最大值，算法返回找到的第一个最小或最大值。这 4 个函数的使用详细见例 4-5。

例 4-5

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

```
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
  if(last >= first)
  {
  for(int i=first;i<=last;++i)
    vect.insert(vect.end(),i);
  }
  else
  {
  cout<<" The indexes is error: last < first. "<<endl;
  }
}
void print(int elem)
{
  cout<<elem<<" ";
}
bool AbsLess(int elem1, int elem2)                    //使用绝对值进行比较
{
  return abs(elem1)<abs(elem2);
}
void main()
{
  vector <int> myvector;
  FillValue(myvector, -3,12);                          //给向量赋值
  for_each(myvector.begin(),myvector.end(),print);    //输出向量中的元素
  cout<<endl;
  cout<<"minimum: "<<*min_element(myvector.begin(),myvector.end())<<endl;
                                                       //获得最小的元素值
  cout<<"maximum: "<<*max_element(myvector.begin(),myvector.end())<<endl;
                                                       //获得最大的元素值
  cout<<"minimum of absolute value: "<<*min_element(myvector.begin(),myvector.
end(),AbsLess)
   <<endl;                                             //获得向量中绝对值最小的元素
 cout<<"maximum of absolute value: "<<*max_element(myvector.begin(),myvector.
end(),AbsLess)
  <<endl;                                              //获得向量中绝对值最大的元素
 }
```

例 4-5 的执行效果见图 4-5。

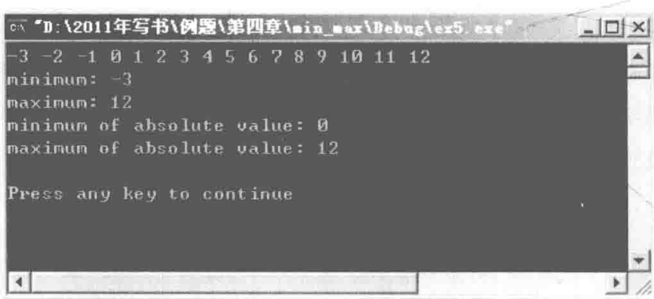


图 4-5　例 4-5 的执行效果

总结：min_element()算法和 max_element()算法的使用通过例 4-5 进行说明。在例 4-5 中既实现了无条件的最小值、最大值求取，还实现了利用仿函数的有条件最大值、最小值的求取。在例题中仿函数的作用是利用元素绝对值进行比较。

4.2.4 搜寻算法

STL 算法库提供了一系列搜索算法，可以获得第一个和给定值的元素的位置。同时也提供了条件搜索算法。

（1）第 1 组搜索算法函数（搜寻第一个匹配元素）：

```
Iterator find (Iterator begin , Iterator end , const T& value);
Iterator find_if (Iterator begin , Iterator end, UnaryPredicate op);
```

函数说明：

第一种形式函数返回区间[begin, end]中第一个“元素值等于 value”的元素的位置；第二种形式函数返回在区间[begin , end]中满足条件 op（elem）的第一个元素。如果函数搜索失败，返回值为 end。

上述两个函数的使用见例 4-6。

例 4-6

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
  if(last >= first)
  {
  for(int i=first;i<=last;++i)
    vect.insert(vect.end(),i);
  }
  else
  {
  cout<<" The indexes is error: last < first. "<<endl;
  }
}
void print(int elem)
{
  cout<<elem<<" ";
}
void main()
{
  vector <int> myvector;
  FillValue(myvector, -3,12);
  for_each(myvector.begin(),myvector.end(),print);
  cout<<endl;
  vector <int>::iterator pos1;
  pos1=find(myvector.begin(),myvector.end(),5);
  vector<int>::iterator pos2;
```

```
    pos2=find_if(myvector.begin(),myvector.end(),bind2nd(greater<int>(),3));
//   cout<<"The position of the element equaling to 5 :"<<distance(myvector.begin(),
pos1)+1<<endl;
//   cout<<"The position of the first element being larger than 3 :"<<distance
(myvector.begin(),pos2)+1
//   <<endl;
    cout<<"第一个等于 5 的元素位置 :"<<distance(myvector.begin(),pos1)+1<<endl;
    cout<<"第一个大于 3 的元素位置 :"<<distance(myvector.begin(),pos2)+1<<endl;
}
```

例 4-6 的执行效果见图 4-6。

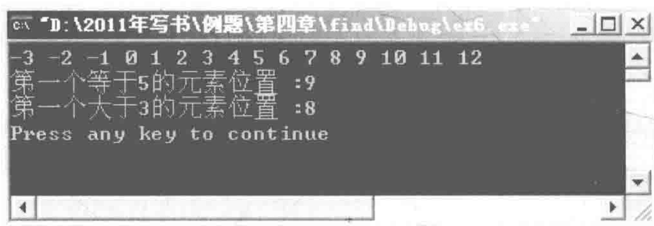


图 4-6　例 4-6 的执行效果

（2）第 2 组搜索算法函数（搜寻前 n 个连续匹配值）：

```
Iterator search_n (Iterator begin , Iterator end , Size count , const T& value );
Iterator search_n (Iterator begin , Iterator end , Size count , const T& value ,
BinaryPredicate op );
```

函数说明：

第一个函数返回区间[begin , end]中的第一组“连续 count 个元素值全等于 value”的元素位置；第二个函数返回区间[begin , end]中第一组满足条件 op 为“真”的“连续 count 个元素”。如果搜索失败，函数返回值为 end。这两个函数的使用方法见例 4-7。

提示：关注例题中“greater<int>()”这个参数的使用，这也是类模板的使用方法之一。

例 4-7

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
   if(last >= first)
   {
   for(int i=first;i<=last;++i)
      vect.insert(vect.end(),i);
   }
   else
   {
   cout<<" The indexes is error: last < first. "<<endl;
```

```
    }
}
void print(int elem)
{
    cout<<elem<<" ";
}
void main()
{
    vector <int> myvector;
    FillValue(myvector, -3,12);
    for_each(myvector.begin(),myvector.end(),print);
    cout<<endl;
    vector <int>::iterator pos1;
    pos1=search_n(myvector.begin(),myvector.end(),4,3);
    if(pos1!=myvector.end())
    {
        cout<<"4 个连续等于 3 的元素的起始位置: "<<distance(myvector.begin(),pos1)+1<<endl;
    }
    else
    {
        cout<<"没有 4 个连续等于 3 的元素"<<endl;
    }
    vector<int>::iterator pos2;
    pos2=search_n(myvector.begin(),myvector.end(),4,3,greater<int>());
    if(pos2!=myvector.end())
    {
      cout<<"4 个连续大于 3 的元素的起始位置 :"<<distance(myvector.begin(),pos2)+1<<endl;
    }
    else
    {
      cout<<"第一个大于 3 的元素位置 :"<<distance(myvector.begin(),pos2)+1<<endl;
    }
}
```

例 4-7 的执行效果见图 4-7。

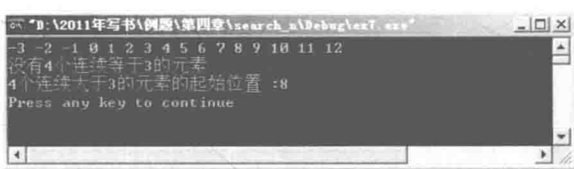


图 4-7　例 4-7 的执行效果

（3）第 3 组搜索算法函数:（搜寻第一个子区间）

```
    Iterator search(Iterator1 begin , Iterator1 end, Iterator2 searchbegin, Iterator2 searchend);
    Iterator search(Iterator1 begin , Iterator1 end, Iterator2 searchbegin, Iterator2 searchend, BinaryPredicate op );
```

函数说明:

这两个函数都返回和区间[searchbegin, searchend]完全吻合的第一个字区间的第一个元素位置。第二个函数只有在 op 为“真”时才被执行。如果搜索失败，两个函数均返回 end。这一组函数使用

比较简单，详见例 4-8。读者要认真研究函数 checkEven()的定义和调用过程。

提示：在 search 函数调用过程中，checkEven()的两个参数分别从向量和数组 checkEvenArg 中获得。checkEven()的第一个参数 elem 是向量 myvector 中的逐个元素，第二个参数 even 是从数组 checkEvenArg 中的元素，按顺序逐一代入。

例 4-8

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
   if(last >= first)
   {
   for(int i=first;i<=last;++i)
      vect.insert(vect.end(),i);
   }
   else
   {
   cout<<" The indexes is error: last < first. "<<endl;
   }
}
void print(int elem)
{
   cout<<elem<<" ";
}
 bool checkEven(int elem, bool even)
{
   if(even)
   {
      return elem%2==0;
   }
   else
   {
      return elem%2==1;
   }
}
void main()
{
   vector <int> myvector;
   vector <int> subvector;
   bool checkEvenArg[3]={true,false,true};
   FillValue(myvector, -3,12);
   FillValue(subvector,-1,3);
   for_each(myvector.begin(),myvector.end(),print);
   cout<<endl;
   for_each(subvector.begin(),subvector.end(),print);
   cout<<endl;
```

```
    vector<int>::iterator pos1;
    pos1=search(myvector.begin(),myvector.end(),subvector.begin(),subvector.end());
    if(pos1!=myvector.end())
    {   cout<<"子串在原串中的位置"<<distance(myvector.begin(),pos1)+1<<endl;
    }
    else
    {   cout<<"没有搜索到需要的子串"<<endl;
    }
    vector<int>::iterator pos2;
    pos2=search(myvector.begin(),myvector.end(),checkEvenArg,checkEvenArg+3,
checkEven);
    cout<<"满足'偶，奇，偶'顺序的子串起始位置："<<distance(myvector.begin(),pos2)+1<<endl;
  }
```

例 4-8 的执行效果图见图 4-8。

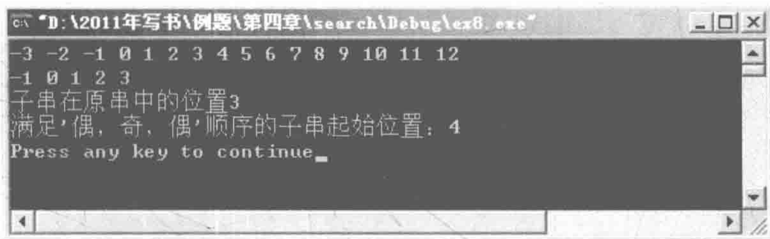


图 4-8　例 4-8 的执行效果

（4）第 4 组搜索算法：（搜寻最后一个子区间）

```
    Iterator find_end (Iterator begin , Iterator end , Iterator2 searchbegin, Iterator2
searchend,);
    Iterator find_end (Iterator1 begin , Iterator1 end, Iterator2 searchbegin, Iterator2
searchend, BinaryPredicate op );
```

函数说明：

这一组函数都返回区间[begin , end]之中“和区间[searchbegin, searchend]完全吻合”的最后一个子区间内的第一个元素位置。其中，第二个函数只有在条件表达式 op 为“真”时才有意义。如果函数搜索失败，返回 end。这两个函数的使用和前面的使用大相径庭，直接给出例 4-9。

例 4-9

```
    #include <functional>
    #include <iostream>
    #include <vector>
    #include <algorithm>
    using namespace std;
    template <class T>
    void FillValue(T& vect, int first, int last)
    {
      if(last >= first)
      {
      for(int i=first;i<=last;++i)
        vect.insert(vect.end(),i);
      }
      else
```

```
    {
    cout<<" The indexes is error: last < first. "<<endl;
    }
}
void print(int elem)
{
    cout<<elem<<" ";
}
void main()
{
    vector <int> myvector;
    vector <int> subvector;
    FillValue(myvector, -3,12);
    FillValue(myvector, -3,6);
    FillValue(subvector,-1,3);
    for_each(myvector.begin(),myvector.end(),print);
    cout<<endl;
    for_each(subvector.begin(),subvector.end(),print);
    cout<<endl;
    vector<int>::iterator pos1;
    pos1=find_end(myvector.begin(),myvector.end(),subvector.begin(),subvector.end());
    if(pos1!=myvector.end())
    {
       cout<<"最后一个子串在原串中的位置: "<<distance(myvector.begin(),pos1)+1<<endl;
    }
    else
    {
       cout<<"没有搜索到需要的子串."<<endl;
    }
}
```

例 4-9 的执行效果见图 4-9。

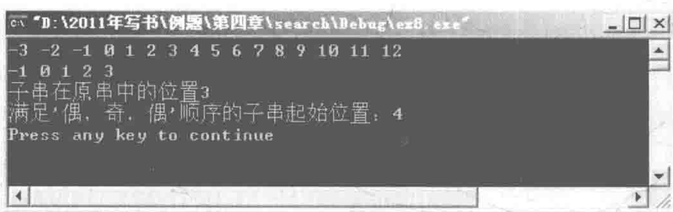


图 4-9　例 4-9 的执行效果

（5）第 5 组搜索函数（搜寻某些元素的第一次出现位置）

```
    Iterator find_first_of(Iterator begin, Iterator end , Iterator2 searchbegin,
Iterator2 search end);
    Iterator find_first_of(Iterator begin, Iterator end , Iterator2 searchbegin,
Iterator2 search end, BinaryPredicate op);
```

函数说明：

这两个函数都返回“子串[searchbegin, searchend]”在区间[begin , end]中出现的第一个元素的位置；第二个函数执行时要满足条件表达式 op 为“真”，才有意义。

下面的例 4-10 实现了两个功能：在原向量（myvector）中搜索和向量（subvector）完全吻合

的子向量第一次出现的位置，然后从后面反向搜索和向量（subvector）完全吻合的子向量第一次出现的位置。

提示：请读者关注 vector<int>::reverse_iterator rpos 这条语句。

例 4-10

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class T>
void FillValue(T& vect, int first, int last)
{
  if(last >= first)
  {
  for(int i=first;i<=last;++i)
    vect.insert(vect.end(),i);
  }
  else
  {
  cout<<" The indexes is error: last < first. "<<endl;
  }
}
void print(int elem)
{
  cout<<elem<<" ";
}
void main()
{
  vector <int> myvector;
  vector <int> subvector;
  FillValue(myvector, -3,12);
  FillValue(myvector, -3,6);
  FillValue(subvector,-1,3);
  for_each(myvector.begin(),myvector.end(),print);
  cout<<endl;
  for_each(subvector.begin(),subvector.end(),print);
  cout<<endl;
  vector<int>::iterator pos1;
  pos1=find_first_of(myvector.begin(),myvector.end(),subvector.begin(),subvector.
end());
  if(pos1!=myvector.end())
  {
    cout<<"第一个子串在原串中的位置: "<<distance(myvector.begin(),pos1)+1<<endl;
  }
  else
  {
    cout<<"没有搜索到需要的子串."<<endl;
  }
```

```
    vector<int>::reverse_iterator rpos;
    rpos=find_first_of(myvector.rbegin(),myvector.rend(),subvector.begin(),subvector.
end());
    cout<<"原串中最后一个字串的位置: "<<distance(myvector.begin(),rpos.base())<<endl;
  }
```

例 4-10 的执行效果见图 4-10。

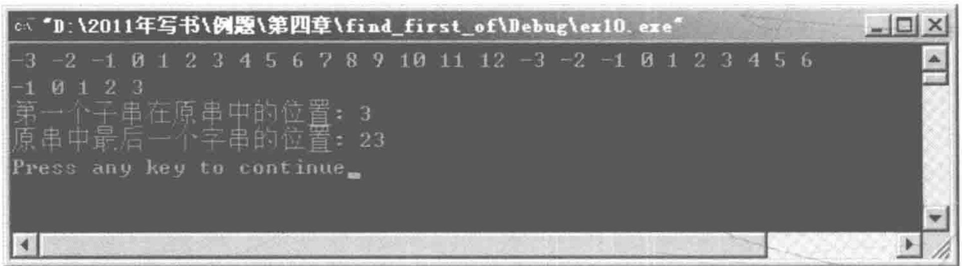


图 4-10　例 4-10 的执行效果

（6）第 6 组搜索函数（搜寻两个连续相等的元素）

使用函数 adjacent_find()可实现搜寻序列中两个连续相等元素的功能。函数的原型如下：

```
    template<class FwdIt>  FwdIt adjacent_find(FwdIt first, FwdIt last);
    template<class FwdIt, class Pred>  FwdIt adjacent_find(FwdIt first, FwdIt last, Pred
pr);
```

上述两种形式的函数声明形式，第一种形式返回在区间[first , last]中第一对“连续相等两个元素”之中的第一个元素的位置；第二种形式返回区间[first , last]中第一对“连续两个元素均使二元判断式 pr 为 true”其中的第一个元素。第二种形式本质还是属于有条件搜索，搜索符合条件 pr 的两个连续元素，并返回第一个元素的位置。

此函数既可以用于普通的 C 语言数组，也可以用于类似 vector 的容器。下面用例 4-11 来加以说明。

例 4-11

```
    #include <iostream>
    #include <algorithm>
    #include <vector>
    using namespace std;
    void print(int& Ele)
    {
      cout<<Ele<<", ";
    }
    bool doubleS(int ele1,int ele2)
    {
      return ele1*2==ele2;
    }
    void main()
    {
      vector<int> v1;
      vector<int>::iterator it;
      v1.push_back(1);
      v1.push_back(3);
      v1.push_back(3);
```

```
    v1.push_back(2);
    v1.push_back(4);
    v1.push_back(5);
    v1.push_back(5);
    v1.push_back(0);
    for_each(v1.begin(),v1.end(),print);
    cout<<endl;
    it=adjacent_find(v1.begin(),v1.end());
    if(it!=v1.end())
    {
       cout<<"The position is :"<<distance(v1.begin(),it)+1<<endl;
    }
    it=adjacent_find(v1.begin(),v1.end(),doubleS);
    if(it!=v1.end())
    {
       cout<<"The position is :"<<distance(v1.begin(),it)+1<<endl;
    }
}
```

例 4-11 的执行效果见图 4-11。

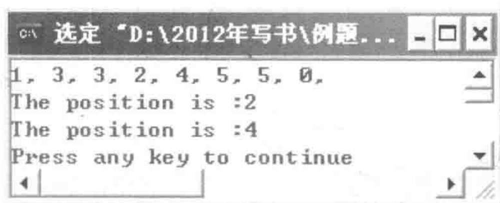


图 4-11　例 4-11 的执行效果

4.2.5　区间比较算法

STL 提供了 3 个比较函数：equal()，　mismatch()，lexicographical_compare()。

函数 equal()可以实现两个容器对象的比较，如果两个对象相等，函数返回 true。该函数只能比较属于同一类型的不同变量。

函数 mismatch()用于寻找两个容器对象之间两两相异的对应元素，如果没有找到相异点，也并不意味这两个对象完全相同，因为这两个对象的容量不一定相同。如果需要判断两个对象相等，要使用 equal()算法。如果没有找到相异点，函数返回值是一个 pair，以第一个对象的 end 元素和第二个对象的对应元素组成。如果找到相异点，该相异点的对应元素组成一个 pair，并被函数返回。

函数 lexicographical_compare()属于字典式比较。所谓字典式比较是指：两个容器中的元素一一比较；直到发生以下情况 ：

（1）如果两个元素不相等，则两个元素的比较结果即为两个序列的比较结果；

（2）如果两个容器中的元素数量不相等，则元素较少的序列小于另一个序列，即若第一序列元素数量较少，则比较结果为 true；

（3）如果两序列均没有更多的元素做比较，则两序列相等，比较结果为 false。

3 个函数的原型分别为：

```
   template<class InIt1, class InIt2>bool equal(InIt1 first, InIt1 last, InIt2 x);
   template<class InIt1, class InIt2, class Pred> bool equal(InIt1 first, InIt1 last,
InIt2 x, Pred pr);
   template<class InIt1, class InIt2> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1
```

```
last, InIt2 x);
    template<class InIt1, class InIt2, class Pred> pair<InIt1, InIt2> mismatch(InIt1
first, InIt1 last, InIt2 x, Pred pr);
    template<class InIt1, class InIt2> bool lexicographical_compare(InIt1 first1, InIt1
last1,InIt2 first2, InIt2 last2);
    template<class InIt1, class InIt2, class Pred> bool lexicographical_compare(InIt1
first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

下面用例 4-12～例 4-14 对上述 3 个函数的使用进行说明。

例 4-12

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
void Origin(vector<int>& v)
{  v.push_back(1);
   v.push_back(2);
   v.push_back(3);
   v.push_back(4);
   v.push_back(5);
   v.push_back(6);
   v.push_back(7);
}
void Origin2(vector<int>& v)
{   v.push_back(3);
   v.push_back(4);
   v.push_back(5);
   v.push_back(6);
   v.push_back(7);
   v.push_back(8);
   v.push_back(9);
}
bool relationship(int e1, int e2)
{  int tmp=e1+2;
   return  e2==tmp;
}
void Print(vector<int>& v)
{  vector<int>::iterator it;
   for(it=v.begin();it!=v.end();it++)
      cout<<*it<<",  ";
   cout<<endl;
}
void main()
{   vector<int> v1,v2;
    Origin(v1);
    Origin2(v2);
    cout<<"vector v1: "<<endl;
    Print(v1);
    cout<<"vector v2: "<<endl;
    Print(v2);
    bool eq=equal(v1.begin(),v1.end(),v2.begin());
    if(eq)
    {       cout<<"v1 == v2"<<endl;
```

```
    }
    else
    {      cout<<"v1 != to v2"<<endl;
    }
    bool eq2=equal(v1.begin(),v1.end(),v2.begin(),relationship);
    if(eq2)
    {      cout<<"v2[i]==v1[i]+2"<<endl;
    }
    else
    {      cout<<"v2[i]!=v1[i]+2"<<endl;
    }
}
```

例 4-12 的执行效果见图 4-12。

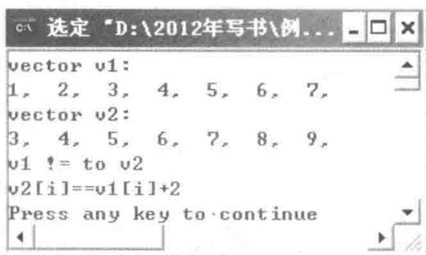


图 4-12　例 4-12 的执行效果

例 4-13 用于说明函数 mismatch()的用法。

例 4-13

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
void Print(int& Ele)
{   cout<<Ele<<", ";
}
void main()
{list<int> l1;
   vector<int> l2;
   pair<list<int>::iterator,vector<int>::iterator> p1;
   l1.push_back(1);
   l1.push_back(2);
   l1.push_back(3);
   l1.push_back(4);
   l1.push_back(7);
   l1.push_back(5);
   l2.push_back(1);
   l2.push_back(2);
   l2.push_back(3);
   l2.push_back(5);
   l2.push_back(6);
   l2.push_back(4);
   for_each(l1.begin(),l1.end(),Print);
   cout<<endl;
   for_each(l2.begin(),l2.end(),Print);
```

```
  cout<<endl;
   p1=mismatch(l1.begin(),l1.end(),l2.begin());
  if(p1.first==l1.end())
  {   cout<<"no mismatch."<<endl;
  }
  else
  {   cout<<"The first mismatch: ";
    cout<<*p1.first<<",  "<<*p1.second<<endl;
  }
  p1=mismatch(l1.begin(),l1.end(),l2.begin(),less_equal<int>());
  if(p1.first==l1.end())
  {   cout<<"no mismatch."<<endl;
  }
  else
  {   cout<<"The first mismatch: ";
    cout<<*p1.first<<",  "<<*p1.second<<endl;
  }
}
```

例 4-13 的执行效果见图 4-13。

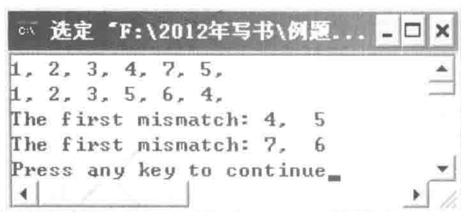


图 4-13　例 4-13 的执行效果

例 4-14 用于说明函数 lexicograpghical_compare()的使用方法。

例 4-14

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
using namespace std;
void Print(int& Ele)
{   cout<<Ele<<", ";
}
void main()
{list<int> l1;
  vector<int> l2;
  l1.push_back(1);
  l1.push_back(2);
  l1.push_back(3);
  l1.push_back(4);
  l1.push_back(5);
  l2.push_back(1);
  l2.push_back(2);
  l2.push_back(3);
  l2.push_back(5);
  l2.push_back(4);
  cout<<"l1: "<<endl;
```

```
    for_each(l1.begin(),l1.end(),Print);
    cout<<endl;
    cout<<"l2: "<<endl;
    for_each(l2.begin(),l2.end(),Print);
    cout<<endl;
    bool b=lexicographical_compare(l1.begin(),l1.end(),l2.begin(),l2.end());
    if(b)
    {   cout<<"l1 < l2. "<<endl;
    }
    else
    {   cout<<"l1 >= l2. "<<endl;
    }
}
```

例 4-14 的执行效果见图 4-14 。

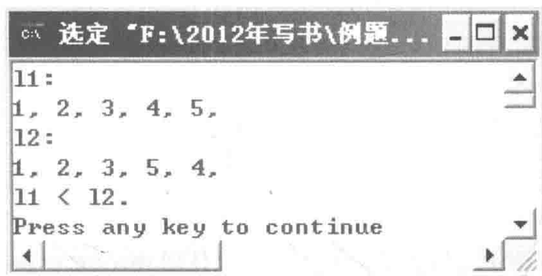


图 4-14　例 4-14 的执行效果

提示：以上 3 个函数除其各自的一般形式外，均提供了自定义比较规则。使用自定义比较规则时，其实现的功能已经超越了其本身名称的意义。equal()和 lexicographical_compare()返回值均为 bool 类型，而函数 mismatch()的返回值为 pair 型。读者应逐渐理解这 3 个函数的意义。

4.3　变动性算法

前面讲述的非变动性算法对其所作用的容器内容不做任何修改。在实际编程中，经常需要对所操作的容器进行修改和写操作。

在 STL 中，提供诸多功能完善的算法，能够对所作用的容器进行修改的算法称为变动性算法（或修正序列算法）。改变容器中元素的值一般包括两种方法：

（1）使用迭代器遍历序列的过程中，直接改变元素的值；

（2）在元素复制过程中，改变元素的值。

本节对变动性算法进行分类，主要包括：复制、转换、互换、赋值、替换、逆转、旋转、排列 8 种。

提示：在实现本节的上述算法时，其目标容器或目标区间必须不能是关联式容器，因为关联式容器是自动内部排序的。关联式容器的元素均被视为常数，也是不允许作为变量形式使用的，否则无法保证其内部自动排序。

4.3.1　复制

元素复制是指在两个容器变量之间进行元素传递，STL 提供复制算法进行元素复制。STL 中实现复制功能的算法包括两种形式：copy()和 copy_backward()。两个函数的原型如下：

```
OutputIterator copy(InputIterator _First, InputIterator _Last, OutputIterator _DestBeg );
BidirectionalIterator2 copy_backward( BidirectionalIterator1 _First, BidirectionalIterator1
_Last, BidirectionalIterator2
    _DestEnd );
```

两个函数形式具有较大区别，函数的参数和返回值类型也不相同。函数 copy()实现将(_First, _Last)指定范围的所有元素复制到另一个容器中由_DestBeg 指定的起始位置。函数 copy_backward()的输入参数均是双向迭代器，返回值也是双向迭代器，唯一不同的是：函数从指定范围的最后一个元素开始复制，之后向前直到第一个元素。

由上述内容可知：copy()函数是正向遍历序列，copy_backward()函数是逆向遍历序列。这两个函数的使用方法不同，也给程序员带来了问题。例如，当源区间和目标区间存在重复区域时，会导致这些问题：

（1）如果要把一个区间复制到前端，应使用 copy()，此时目标位置应在_First 之前；

（2）如果要把一个区间复制到后端，应使用 copy_backward()。目标位置_DestEnd 应该在 sourceEnd 之后。总之，当_DestEnd 处在（_First, _Last）范围内时，程序员要慎重考虑。

值得一提的是：

（1）STL 并没有提供 copy_if()算法。如果希望实现有条件地复制容器中的元素，可以使用算法 remove_copy_if()；

（2）如果希望在复制过程中逆转元素的顺序，可以使用 reverse_copy()算法；

（3）程序员要确保目标区间有足够空间，否则需要使用 insert 迭代器；

（4）如果实现两个容器间所有元素的复制，可以不使用 copy()和 copy_backward()，而使用 assign()算法；

（5）如果希望在复制过程中删除部分元素，可使用算法 remove_copy()和 remove_copy_if()。

（6）如果在复制过程中改变元素的数值，需要使用 transform()算法和 replace_copy()算法。

（7）如果目标容器是空容器，需要使用插入型迭代器（InsertIterator）。

例 4-15 讲述了复制的使用方法。

例 4-15

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;
void print(int& ele)
{   cout<<ele<<", ";
}
void main()
{  int dim[]={1,3,5,7,9,11,32,54,65};
   vector<int> v1;
   v1.assign(dim,dim+9);
   cout<<"vector v1: "<<endl;
   for_each(v1.begin(),v1.end(),print);
   cout<<endl;
   list<int>l1,l2;
   copy(v1.begin(),v1.end(),back_inserter(l1));
   cout<<"list l1: "<<endl;
   for_each(l1.begin(),l1.end(),print);
   cout<<endl;
```

```
    //输出 l2
    l2=l1;
    cout<<"list l2: "<<endl;
    copy(l2.begin(),l2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    copy_backward(v1.begin()+2,v1.begin()+7,l2.end());
    cout<<"list l2(modified) : "<<endl;
    copy(l2.begin(),l2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-15 的执行效果见图 4-15。

图 4-15　例 4-15 的执行效果

提示：

（1）函数 copy_backward (_first, _end, _dest) 在使用时，源数据是源容器中范围[_first, _end]内的元素；在目标容器中，被复制的元素应该保留在目标指针_dest 之前，并且在复制过程中是将[_first, _end]内的元素按从倒数第一个至正数第一个的顺序复制，存在指针_dest 前面的内存空间。因此，在指定指针_dest 时，要留出足够的存放空间。

（2）关注 back_inserter()的使用。

（3）关注如何将数据复制至 cout 中。

4.3.2　转换

STL 中提供了函数 transform()，实现了将源区间的元素复制至目标区间，复制和修改元素一气呵成；还可以将两个区间的元素合并，并将结果写入目标区间。函数原型为：

```
    template<class InIt, class OutIt, class Unop> OutIt transform(InIt first, InIt last,
OutIt x, Unop uop);
    template<class InIt1, class InIt2, class OutIt, class Binop> OutIt transform(InIt1
first1, InIt1 last1, InIt2 first2, OutIt x, Binop bop);
```

第一种形式实现的功能：针对源区间[sourceBeg, sourceEnd]中的每个元素均调用函数 uop(ele)，并将结果写入以 destBeg 为起始的目标区间内。函数返回值是“最后一个被转换元素”的下一个位置。在使用函数时，要确保有足够的空间存放这些元素，否则需要使用插入型迭代器。transform()函数可以变动序列中的每个元素，在这方面和算法 for_each()相似。

第二种形式实现的功能：针对源区间[first1, last1]中的每个元素，和从 first2 开始的第二个源区间的对应元素，调用函数或规则 bop（ele1,ele2），并将结果写入以 x 为起始的目标区间。函数返回值和第一种形式相同。在使用时，程序员需要确保第二源区间具有足够的容器，至少具有和第一源区间同样的容量。同样在使用时，要确保具有足够的空间（容量），否则需要使用插入型迭代器。值

得一提的是，两个源区间可以相同，目标区间也就可以和源区间相同。

下面以例 4-16 说明上述 transform()算法的使用方法。

例 4-16

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
void print(int& Ele)
{   cout<<Ele<<", ";
}
void main()
{
int dim[]={1,2,3,4,5,6,7,8,9};
  vector<int>v1;
  list<int>l2,l3;
  v1.assign(dim,dim+9);
  cout<<"vector v1: ";
  for_each(v1.begin(),v1.end(),print);
  cout<<endl;
  //l2 中的元素分别乘以-1
  transform(v1.begin(),v1.end(),back_inserter(l2),negate<int>());
  cout<<"list l2: ";
  for_each(l2.begin(),l2.end(),print);
  cout<<endl;
  //l2 中的元素分别乘以 10
  transform(l2.begin(),l2.end(),l2.begin(),bind2nd(multiplies<int>(),10));
  cout<<"list l2(*10): ";
  for_each(l2.begin(),l2.end(),print);
  cout<<endl;
   cout<<"list l2(-): ";
  //l2 中的元素没有变化
  transform(l2.rbegin(),l2.rend(),ostream_iterator<int>(cout,", "),negate<int>());
  cout<<endl;
   cout<<"list l2(/2, reverse_direction): ";
  //l2 中的元素没有变化
  transform(l2.rbegin(),l2.rend(),ostream_iterator<int>(cout,", "),bind2nd
(divides<int>(),2));
  cout<<endl;
  //以上是第一种形式的用法
  cout<<"list l2: ";
   for_each(l2.begin(),l2.end(),print);
  cout<<endl;
  cout<<"list l2(*v1): ";
  transform(v1.begin(),v1.end(),l2.begin(),l2.begin(),multiplies<int>());
  copy(l2.begin(),l2.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"list l2(squared): ";
   //l2 中的元素没有变化
   transform(l2.begin(),l2.end(),l2.begin(),ostream_iterator<int>(cout,", "),multiplies
<int>());
```

```
     cout<<endl;
    cout<<"list l2: ";
     for_each(l2.begin(),l2.end(),print);
    cout<<endl;
    cout<<"l3( v1+l2 ): ";
    transform(v1.begin(),v1.end(),l2.begin(),back_inserter(l3),plus<int>());
     for_each(l3.begin(),l3.end(),print);
     cout<<endl;
    cout<<"cout (l2-l3) : ";
    //l2、l3 中的元素没有变化
    transform(l2.begin(),l2.end(),l3.begin(),ostream_iterator<int>(cout,", "),minus
<int>());
    cout<<endl;
}
```

例 4-16 的执行效果见图 4-16。

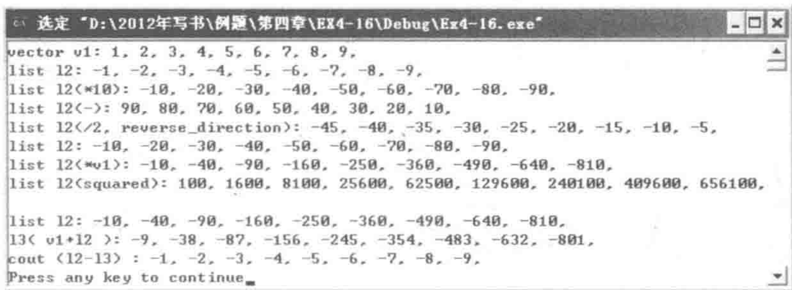


图 4-16　例 4-16 的执行效果

请读者参照程序的执行结果，对照源代码和注释，体会算法 transform()的用途和使用方法。函数 transform()既可以使用 STL 既定的函数或仿函数作为规则，还可以使用自定义的函数或仿函数作为规则。自定义函数规则参见例 4-17。例 4-17 使用仿函数的形式实现了 transform()的自定义规则传递。自定义类作为仿函数使用，可以任意编写变换 transform()的功能，从而实现程序员的目的。

例 4-17

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
void print(int& Ele)
{   cout<<Ele<<", ";
}
template <class T> class customFun{
public:
   T Parm;
public:
  customFun(const T& _Val):Parm(_Val)
  {}
  int operator()(T&elem ) const
  {   return elem*Parm;
  }
```

```
    };
    template <class T> class customFun2{
    public:
        T Parm;
    public:
      customFun2(const T& _Val):Parm(_Val)
      {}
      int operator()(T& elem ,T& elem2 ) const
      {   return (elem+elem2)*Parm;
      }
    };

    void main()
    {   int dim[]={1,2,3,4,5,6,7,8,9};
      vector<int>v1,v2;
      list<int> l1;
      v1.assign(dim,dim+9);
      copy(v1.begin(),v1.end(),back_inserter(l1));
      cout<<"vector v1: ";
      for_each(v1.begin(),v1.end(),print);
      cout<<endl;
      transform(v1.begin(),v1.begin()+5, v1.begin(),customFun<int>(4));
       cout<<"vector v1(transformed): ";
      for_each(v1.begin(),v1.end(),print);
      cout<<endl;
     cout<<"list l1: ";
      for_each(l1.begin(),l1.end(),print);
      cout<<endl;
      transform(v1.begin(),v1.end(),l1.begin(),l1.begin(),customFun2<int>(2));
     cout<<"vector l1(=(v1+l1)*2): ";
      for_each(l1.begin(),l1.end(),print);
      cout<<endl;
    //transform(v1.begin(),v1.end(),l1.begin(),ostream_iterator<int>(cout,", "),customFun2
<int>(2));
    //   cout<<endl;
    //   transform(v1.begin(),v1.end(),l1.begin(),back_inserter(v2),customFun2
<int>(2));
    //   cout<<"vector l1(=(v1+l1)*2): ";
    // copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
    // cout<<endl;
    }
```

例 4-17 的执行效果见图 4-17。

```
选定 "D:\2012年写书\例题\第四章\Ex4-17\Debug\Ex4-17....
vector v1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector v1(transformed): 4, 8, 12, 16, 20, 6, 7, 8, 9,
list l1: 1, 2, 3, 4, 5, 6, 7, 8, 9,
vector l1(=(v1+l1)*2): 10, 20, 30, 40, 50, 24, 28, 32, 36,
Press any key to continue
```

图 4-17　例 4-17 的执行效果

总结：

（1）在上例中定义了两个仿函数类。请读者认真阅读第二个仿函数类，此仿函数是针对 transform()算法的第二种形式编写的。

（2）在 main()函数的末尾，包含了 6 行注释语句，前 2 行是一组，后 4 行是一组，可以使用这两组代码替换 main()函数中第二种形式的 transform()算法的使用，即 main()函数中代码第 16 行至第 19 行。

（3）例 4-17 较完整地阐释了算法 transform()的自定义运算规则，同时在本书中第一次给出了多参数的仿函数例子，有利于开拓读者的思路和视野，为读者今后编写出更好的仿函数打下基础。

4.3.3　互换

STL 的大部分容器均提供了成员函数 swap()，用于两个不同容器对象之间的元素交换。STL 同时还提供了算法 swap()。该算法的函数原型如下：

```
template<class  T> void swap(T& a, T& b);
template<class FwdIt1, class FwdIt2>  FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
```

第一种形式的函数 swap()用于交换两个类型相同的容器对象，即交换两个容器对象的所有元素；

第二种形式的函数 swap_ranges()可以交换两个容器中的部分元素。函数 swap_ranges()的第一个参数 first 和第二个参数 last 用来指定第一个用于交换的容器 1 中的元素的范围，用参数 x 表示：在第二个容器中，以迭代器 x 为起始位置的元素被交换。下面用例 4-18 来阐释这两个算法的使用方法。

例 4-18

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
   int dim[]={1,2,3,4,5,6,7,8,9};
   int dim2[]={11,12,13,14,15,16,17,18,19};
   vector<int> v1;
   vector<int> v2;
   v1.assign(dim,dim+9);
   cout<<"vector v1: ";
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   copy(dim2,dim2+9,back_inserter(v2));
   cout<<"vector v2: ";
   copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   swap(v1,v2);
      cout<<"vector v1: ";
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
     cout<<"vector v2: ";
```

```
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    swap_ranges(v1.begin(),v1.begin()+5,v2.begin());
      cout<<"vector v1: ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
      cout<<"vector v2: ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-18 的执行效果见图 4-18。

图 4-18　例 4-18 的执行效果

4.3.4　赋值

STL 提供了 4 个算法函数，用于给容器赋值。这 4 个函数分别是：fill()，fill_n()，generate()，generate_n()。

算法函数 fill_n()和 generate_n()只给指定区间前的那个数值赋值。

算法函数 fill()和 fill_n()用于给每个元素赋予相同的数值；算法函数 generate()和 generate_n()在执行时会调用函数的子进程或仿函数(g)，产生新值，并赋予容器中的元素。4 个函数的原型如下：

```
template<class FwdIt, class T>  void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T> void fill_n(OutIt first, Size n, const T& x);
template<class FwdIt, class Gen>void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen> void generate_n(OutIt first, Dist n, Gen g);
```

值得一提的是，算法函数 fill_n()和 generate_n()均指定了赋值区间的起始位置和赋值元素的个数。详见例 4-19。

例 4-19

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int Fibonacci(void)
{   static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2 ;
    f1 = f2 ;
    f2 = r ;
```

```
    return f1 ;
  }
void main()
{  vector<int> v1;
   vector<int> v2(5,0),v3(6,0);
   int dim[]={1,2,3,4,5,6,7,8,9};
   v1.assign(dim,dim+9);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   fill(v1.begin(),v1.begin()+4,9);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   fill_n(v1.begin(),5,20);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   /*----*/
   generate(v2.begin(),v2.end(),rand);
   copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   /*----*/
   generate_n(v2.begin(),3,rand);
   copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   /*--自定义序列---*/
   generate_n(v3.begin(),6,Fibonacci);
   copy(v3.begin(),v3.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
}
```

例 4-19 的执行效果见图 4-19。

总结：算法 generate()和 generate_n() 除了可以使用 STL 提供的函数作为子进程或仿函数用于产生元素，还可以使用自定义的函数产生新的元素值。甚至使用特定的算法，产生数值序列。例如：在例 4-19 中，main()函数最后即产生 1 个"斐波那契"数列。

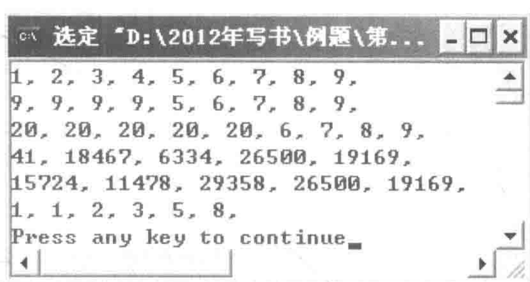


图 4-19　例 4-19 的执行效果

4.3.5　替换

在 STL 中可以通过替换函数 replace()代替容器中需要替换的元素。STL 提供了两种形式的替换算法。其原型分别为：

```
template<caass FwdIt, class T> void replace(FwdIt first, FwdIt last, const T& vold,
```

```
const T& vnew);

    template<class FwdIt, class Pred, class T> void replace_if(FwdIt first, FwdIt last,
Pred pr, const T& val);
```

两个函数均是条件性的替换：

第一种形式的 replace()是替换容器（序列）的[first, last]范围中和 vold 值相等的元素，将这些元素的值替换为 vnew。

第二种形式的 replace_if()是替换容器的[first, last]范围中能够使规则（一元判断式）pr 为 true 的元素，并将这些元素的值修正为 val。

上述两个函数属于比较简单的操作。下面用例 4-20 来阐释这两个函数的使用方法。

例 4-20

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
void main()
{   vector<int> v1;
   int dim[]={1,2,3,4,5,6,7,8,9};
   v1.assign(dim,dim+9);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   replace(v1.begin(),v1.end(),7,99);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   replace_if(v1.begin(),v1.end(),bind2nd(less<int>(),5),11);
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
}
```

例 4-20 的执行效果见图 4-20。

图 4-20　例 4-20 的执行效果

4.3.6　逆转

逆转是指将容器或序列中的元素按逆向顺序反转，第一个元素变成最后一个元素，第二个元素变成倒数第一个元素，依次反转。STL 提供了两个可以实现逆转功能的算法函数：reverse()和 reverse_copy()。

这两个函数均使用双向迭代器。两个函数的原型如下：

```
template <class Bidirectional Iterator> void reverse (BidirectionalIterator first,
```

```
BidirectionalIterator last)
    template <class Bidirectional Iterator, class OutputIterator>OutputIterator
reverse_copy (BidirectionalIterator first,
    BidirectionalIterator last, OutputIterator result);
```

算法 reverse()会将区间[beg, end]中的元素全部逆序；reverse_copy()会将源区间[sourceBeg, sourceEnd]内的元素复制到“以 destBeg 起始的目标区间”，并在复制过程中颠倒安置次序。算法 reverse_copy()返回区间内最后一个被复制元素的下一位置，即第一个未被覆盖的元素。

在使用时，程序员必须确知目标区间足够大，否则就需要使用插入型迭代器。

值得注意的是，上述两个函数的参数是迭代器，通过迭代器可以指定需要逆转的元素的范围，既可以逆转整个容器中的所有元素，还可以逆转容器中的部分元素。

以上函数用例 4-21 加以说明。

例 4-21

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;
void print(int& ele)
{   cout<<ele<<", ";
}
void main()
{ int dim[]={1,2,3,4,5,6,7,8};
  vector<int> v1,v2;
  list<int> l1,l2;
  v1.assign(dim,dim+8);
  reverse(v1.begin(),v1.end());
  cout<<"vector v1: ";
  for_each(v1.begin(),v1.end(),print);
  cout<<endl;
  copy(v1.rbegin(),v1.rend(),back_inserter(l1));
  cout<<"list l1: ";
  for_each(l1.begin(),l1.end(),print);
  cout<<endl;
  reverse(v1.begin()+2,v1.end()-2);
  cout<<"vector v1: ";
  for_each(v1.begin(),v1.end(),print);
  cout<<endl;
  list<int>::iterator posf=l1.begin();
  list<int>::iterator pose=l1.end();
  advance(posf,2);
  advance(pose,-2);
  reverse(posf,pose);
  cout<<"list l1: ";
  for_each(l1.begin(),l1.end(),print);
  cout<<endl;
  reverse_copy(v1.begin(),v1.end(),back_inserter(v2));
  cout<<"vector v2: ";
  for_each(v2.begin(),v2.end(),print);
```

```
    cout<<endl;
    cout<<"list l2: ";
    reverse_copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-21 的执行效果见图 4-21。

图 4-21 例 4-21 的执行效果

总结：

通过本节和前面几节的学习，读者应该意识到一个问题：为什么没有使用 assign 直接赋值一个常量数组到 list 型的容器呢？

这是因为 list 容器的迭代器在移动时，使用的方法和 vector 是不同的。vector 型迭代器在移动时可以使用加法运算（如例 4-21 中所示），而 list 型迭代器需要使用 advance()函数。本节在此前介绍了 advance()。advance()可以方便地实现容器的迭代器的前进和后退。只有随机存取迭代器才可以使用加、减法运算，直接将偏移量加到迭代器上。

4.3.7 旋转

STL 提供了对于容器或序列的旋转算法：即 rotate()和 rotate_copy()。函数 rotate()将容器中的元素或序列看作一个环，旋转这些元素直至原来 middle 处的元素到达 first 位置。旋转运算是使序列中的元素按照一个环的方式旋转，而不是简单的左移。函数 rotate_copy()产生一个旋转后的副本，即在旋转之后会复制参加旋转的元素。两个函数的原型如下：

```
    template <class ForwardIterator> void rotate(ForwardIterator first, ForwardIterator
middle, ForwardIterator last)


    template <class ForwardIterator, class OutputIterator>OutputIterator rotate(
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last,
    OutputIterator result
    )
```

值得一提的是，函数 rotate_copy()的源区间和目标区间不允许重复。例 4-22 阐释了上述两个函数的使用方法。

例 4-22

```
    #include <iostream>
```

```
#include <vector>
#include <algorithm>
using namespace std;
void main()
{
   int dim[]={1,2,3,4,5,6,7,8,9};
   vector<int> v1,v2;
   v1.assign(dim,dim+9);
   v2.assign(v1.begin(),v1.end());
   cout<<"vector v1: ";
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   cout<<"vector v1(first rotation): ";
   rotate(v1.begin(),v1.begin()+4,v1.end());
   copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   cout<<"vector v1(second rotation): ";
   rotate_copy(v1.begin(),v1.begin()+3,v1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   cout<<"vector v1(third rotation): ";
   rotate(v2.begin(),v2.begin()+2,v2.end()-3);
   copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   cout<<"vector v1(fourth rotation): ";
   rotate_copy(v2.begin(),v2.begin()+3,v2.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
}
```

例 4-22 的执行效果见图 4-22。

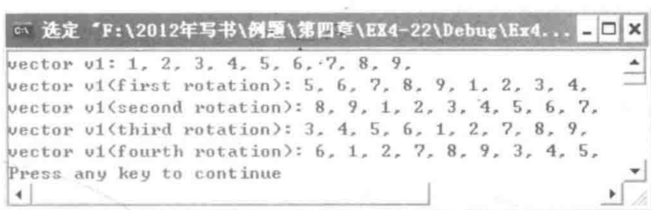


图 4-22　例 4-22 的执行效果

4.3.8　排列

本节主要介绍容器（序列）中元素的“排列元素”、“重排元素”、“将元素向前搬移”3 种操作。“排列元素”会改变容器（序列）中的元素次序，排序方式为字典式“正规”排序。“重排元素”是指对容器中的所有元素进行随机排序，一般有两种形式：符合均匀分布随机的排序；按指定规则打乱容器（序列）中的元素次序。“前向搬移”是按指定的一元判断式向前搬移元素，当一元判断式的值为 true 时，算法会向前移动符合条件的元素，函数返回值是使一元判断式为“false”的第一个元素位置。以上各函数的原型如下。

排列元素：

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,BidirectionalIterator last);
```

```
                                                                    //升序
    template<class BidirectionalIterator, class Compare>
    bool   next_permutation(BidirectionalIterator  first,BidirectionalIterator  last,
Compare comp);                                                      //降序
```

重排元素：

```
    template<class  RandomAccessIterator>   void  random_shuffle(RandomAccessIterator
first,RandomAccessIterator last);                                   //均匀分布
    template<class RandomAccessIterator, class RandomNumberGenerator>  void random_shuffle
(RandomAccessIterator first,  RandomAccessIterator last,  RandomNumberGenerator& rand);
                                                                    //按指定规则
```

将元素向前搬移：

```
    template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator  partition(BidirectionalIterator  first, BidirectionalIterator
last,  Predicate  pred);                                            //不排序
    template<class BidirectionalIterator, class Predicate>
    BidirectionalIterator       stable_partition(   BidirectionalIterator        first,
BidirectionalIterator last,  Predicate pred);
                                                                    //会保留一个相对的次序（排序）
```

面对上面 3 段代码，我们可以看出在排列操作中有 6 种排序方式，例 4-23～例 4-25 讲述了以上 6 种排序方式。

例 4-23

```
    #include <iostream>
    #include <vector>
    #include <deque>
    #include <algorithm>
    using namespace std;
    void main()
    {   deque<int> d1;
       vector<int> v1;
       int dim[]={1,2,3};
       v1.assign(dim,dim+3);
       cout<<"vector v1:  ";
       copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
       cout<<endl;
       cout<<"queue d1(Original):  ";
       copy(v1.begin(),v1.end(),back_inserter(d1));
       copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
       cout<<endl;
       while(next_permutation(d1.begin(),d1.end()))
       {   copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
          cout<<endl;
       }
       cout<<"queue d1(排序后):  ";
       copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
       cout<<endl;
       cout<<"queue d1(准备重新按降序排列):  ";
       copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
       cout<<endl;
       while(prev_permutation(d1.begin(),d1.end()))
       {   copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
```

```
        cout<<endl;
    }
    cout<<"queue d1(降序排列后): ";
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    while(prev_permutation(d1.begin(),d1.end()))
    {   copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
        cout<<endl;
    }
    cout<<"queue d1(再次降序排列后): ";
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-23 的执行效果见图 4-23。

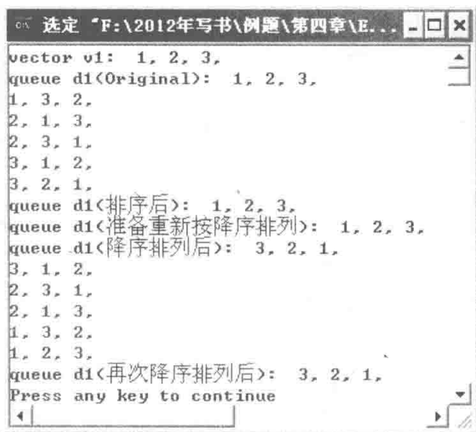


图 4-23　例 4-23 的执行效果

例 4-24

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
class myRandom                                          //创建一个可产生随机数的仿函数
{
public:
    ptrdiff_t operator()(ptrdiff_t max)
    {
        double tmp;
        tmp=static_cast<double>(rand())/static_cast<double>(RAND_MAX);
        return static_cast<ptrdiff_t>(tmp*max);
    }
};
void main()
{
    vector<int> v1;
    int dim[]={1,2,3,4,5,6,7,8,9};
    v1.assign(dim,dim+9);
```

```
    cout<<"vector v1:  ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
                                                  //输出最初的 vector 容器中的元素
    cout<<endl;
    random_shuffle(v1.begin(),v1.end());
    cout<<"vector v1(random ):  ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;                                   //输出打乱顺序后的 vector 容器中的元素
    sort(v1.begin(),v1.end());
    cout<<"vector v1(sorted again ):  ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;                                   //输出排序后的 vector 容器中的元素
    myRandom mrd;
    random_shuffle(v1.begin(),v1.end(),mrd);
    cout<<"vector v1(shuffled again ):  ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;                                   //输出第二次打乱顺序后的 vector 容器中的元素
}
```

例 4-24 的执行效果见图 4-24。

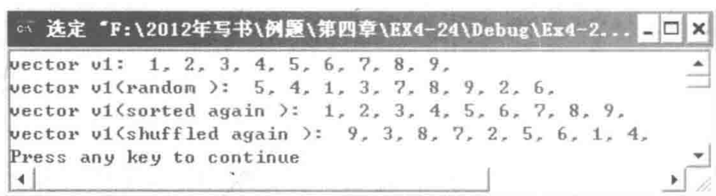


图 4-24　例 4-24 的执行效果

例 4-25 （引自《C++标准程序库》（侯捷译））

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{
    int dim[]={1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    vector<int>v2;
    v1.assign(dim,dim+9);
    v2=v1;
    cout<<"vector v1: ";
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"vector v2: ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    vector<int>::iterator pos1,pos2;
    pos1=partition(v1.begin(),v1.end(),not1(bind2nd(modulus<int>(),2)));
    pos2=stable_partition(v2.begin(),v2.end(),not1(bind2nd(modulus<int>(),2)));
    cout<<"vector v1(modulus, 2, partition): ";
```

```
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"vector v2(modulus, 2, stable_partition): ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-25 的执行效果见图 4-25。

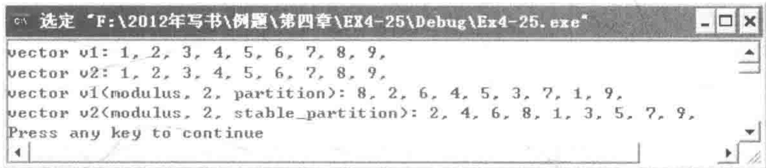


图 4-25　例 4-25 的执行效果

上述 3 个例题分别讲述了 6 个和排序有关的算法，其功能各不相同。但归根结底是通过不同的方式实现对容器（序列）中元素的排序。看似用途不大，但是各种妙处希望读者多理解、多体会。

4.4　排序及相关操作

STL 提供了多种算法用来对容器（序列）中的全部或部分元素进行排序。对于程序员来说，如果局部排序能够满足需要的话，应尽量使用局部排序。当然，关联式容器可以实现自动排序。但是对全体元素进行一次性排序要比时刻维护它们，使它们保持已序状态要高效得多。

对于一个序列，排序是最简单也是最有用的算法之一。因此也不必为 STL 提供了那么多的排序算法而惊讶。STL 提供的排序算法都是功能较强大、机制较完善的。这些排序算法都是基于随机访问的容器元素的方法。这些函数的数量甚至超过 25 个，有些在前面章节已经介绍过，本章将逐步介绍剩余的排序算法。

4.4.1　全部元素排序

算法 sort()和算法 stable_sort()支持对容器（序列）中的所有元素进行排序。算法 sort()和 stable_sort()需要访问随机存取迭代器，只能适用于 vector 型和 deque 型容器。由于 list 型容器不支持随机存取型迭代器，所以不能使用这两个算法，但是 list 型容器提供了 sort()成员函数，用于自身元素的排序，代码如下。

```
    template<class RanIt> void sort(RanIt first, RanIt last);
    template<class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);


    template<class RanIt> void stable_sort(RanIt first, RanIt last);
    template<class RanIt, class Pred> void stable_sort(RanIt first, RanIt last, Pred
pr);
```

stable_sort()能够保持序列或容器中元素的相对顺序不改变，而 sort()则不能做到这点。上述两个算法函数均具有两种形式的函数原型，并且其第二种形式均带有二元判断式 pr(elem1, elem2)，作为排序准则。顾名思义，stable_sort() 会比 sort() 更加稳定、可靠。上述原型中 RanIt 代表 RandomAccessIterator。

上述两个算法函数的默认排序准则是：利用“operator<” 实现升序排序。可以利用规则表达式实现降序排序，甚至可以使用自定义的规则表达式。

上述内容详见例 4-26。

例 4-26

```
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{   int dim[]={1,2,3,4,5,6,7,8,9};
    deque<int> d1;
    copy(dim,dim+9,back_inserter(d1));
    //copy(dim,dim+9,back_inserter(d1));
    cout<<"deque d1(Original): ";
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    //sort(d1.begin(),d1.end());
    stable_sort(d1.begin(),d1.end());
    cout<<"deque d1(sorted by '<'): ";
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    //sort(d1.begin(),d1.end(),greater<int>());
    stable_sort(d1.begin(),d1.end(),greater<int>());
    cout<<"deque d1(sorted by '>'): ";
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-26 的执行效果见图 4-26。

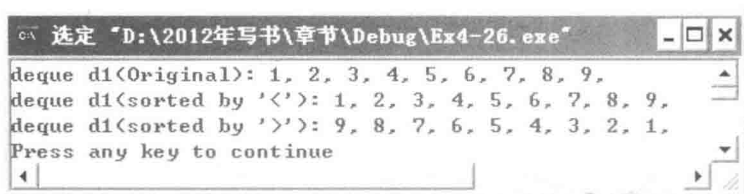


图 4-26　例 4-26 的执行效果

用户还可以自定义二元判断式。函数的定义如下：

```
bool CustomDefineFun(T  elem1, T  elem2)
{
    bool  res=0;
    ……                                    //使用 elem1 和 elem2 进行运算，并对 res 的值进行修改
    return res;                           //一定要返回 bool 型变量值
}
```

上述省略部分在此仅举两例。

（1）若 T 为 int 型，省略部分可为：res=(elem1>elem2);

（2）若 T 为 string 型，省略部分可为：res=(elem1.length()>elem2.length())。

4.4.2　局部排序

STL 还提供了局部排序算法：partial_sort()和 partial_sort_copy()。这两个函数的原型分别为：

```
    void partial_sort( RandomAccessIterator _First, RandomAccessIterator _SortEnd,
RandomAccessIterator _Last );
    void partial_sort( RandomAccessIterator _First, RandomAccessIterator _SortEnd,
RandomAccessIterator _Last,
        BinaryPredicate _Comp );

    RandomAccessIterator partial_sort_copy( InputIterator _First1, InputIterator
_Last1, RandomAccessIterator _First2,
        RandomAccessIterator _Last2 );
    RandomAccessIterator partial_sort_copy( InputIterator _First1, InputIterator
_Last1, RandomAccessIterator _First2,
        RandomAccessIterator _Last2, BinaryPredicate _Comp );
```

1．算法函数 partial_sort()

算法函数 partial_sort ()的功能是实现对区间[_First, _Last]内的元素进行排序，排序后区间[_First, _SortEnd]内的元素处于有序状态。函数 partial_sort()还可以使用二元判断式_Comp(elem1, elem2)，排序后区间[_First, _SortEnd]内的元素是有序的。partial_sort() 仅仅是对容器（序列）中的部分元素进行排序，在需要的情况下可以节约时间，因为不会对剩余元素进行不必要的排序。如果参数_SortEnd 和参数_Last 相等的话，则可实现对全部元素的排序。

经过 partial_sort()排序之后的容器（序列）中，已序的元素被放在容器（序列）的前面，非已序的部分放在容器的后面。

算法函数 partial_sort()的默认排序规则是“operator <”。

2．算法函数 partial_sort_copy()

算法函数 partial_sort_copy()将容器(序列)的元素从区间[_First1, _Last1]复制到目标区间[_First2, _Last2]，同时对这部分元素进行排序。函数返回值是“最后一个被复制元素”的下一位置。被排序和复制的元素数量是源区间[_First1,_Last1]和目标区间[_First2,_Last2]两者所含元素数量中较小的那个数量。如果源区间[_First1,_Last1]的元素数量小于目标区间[_First2,_Last2]的元素数量，则容器（序列）中所有元素均参与排序。

函数 partial_sort_copy()还可使用二元判断式，用于指定排序规则。

通过以上两部分的阐释，可发现算法函数 partial_sort_copy()的功能和 partial_sort()基本相似，只不过需要指定一个独立的容器去接受复制。该独立容器中的元素是已序的。

提示：上述几个适用于排序的算法函数均使用随机存取型迭代器，随机存取型迭代器可以适用于 deque 型、vector 型容器，还可使用于普通的 C 语言数组，例 4-27 中可详细了解。

例 4-27

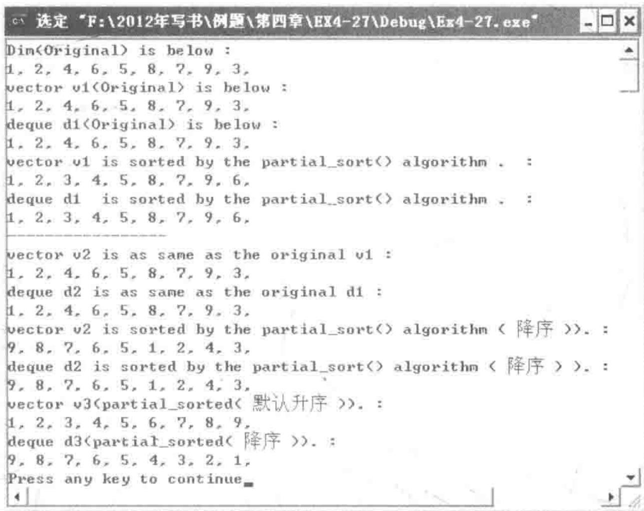
```
#include <iostream>
#include <deque>
#include <vector>
#include <algorithm>
```

```
#include <functional>
using namespace std;
void main()
{   int dim[]={1,2,4,6,5,8,7,9,3};     //初始数组，用来初始化 vector 型和 deque 型容器
  vector<int> v1,v2,v3(9);
  deque<int>d1,d2,d3(9);
  v1.assign(dim,dim+9);                                          //初始化 v1
  copy(v1.begin(),v1.end(),back_inserter(d1));                   //初始化 d1
  cout<<"Dim(Original) is below : "<<endl;
  copy(dim,dim+9,ostream_iterator<int>(cout,", "));              //输出数组 dim
  cout<<endl;
  cout<<"vector v1(Original) is below : "<<endl;
  copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));    //输出 v1
  cout<<endl;
  cout<<"deque d1(Original) is below : "<<endl;
  copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));    //输出 d1
  cout<<endl;
  v2=v1;
  d2=d1;
  partial_sort(v1.begin(),v1.begin()+5,v1.end());        //局部排序 v1 的前 5 个元素
  cout<<"vector v1 is sorted by the partial_sort() algorithm .  : "<<endl;
  copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));//输出局部排序之后的 v1 的结果
  cout<<endl;
  partial_sort(d1.begin(),d1.begin()+5,d1.end());               //局部排序 d1
  cout<<"deque d1  is sorted by the partial_sort() algorithm .  : "<<endl;
  copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", ")); //输出局部排序之后的 d1
  cout<<endl;
  cout<<"-----------------"<<endl;
  cout<<"vector v2 is as same as the original v1 : "<<endl;
  copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
                                          //输出 v2，和 v1 的初始内容相似
  cout<<endl;
  cout<<"deque d2 is as same as the original d1: "<<endl;
                                          //输出 d2 的内容，和 d1 的初始内容相似
  copy(d2.begin(),d2.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;

  partial_sort(v2.begin(),v2.begin()+5,v2.end(),greater<int>());
                                          //对 v2 的前 5 个元素进行降序排序
  cout<<"vector v2 is sorted by the partial_sort() algorithm ( 降序 )). : "<<endl;
  copy(v2.begin(),v2.end(),ostream_iterator<int>(cout,", "));
                                          //输出降序排序之后的 v2
  cout<<endl;
  partial_sort(d2.begin(),d2.begin()+5,d2.end(),greater<int>());
                                          //降序排序 d2 的前 5 个元素
  cout<<"deque d2 is sorted by the partial_sort() algorithm ( 降序 ) ). : "<<endl;
  copy(d2.begin(),d2.end(),ostream_iterator<int>(cout,", "));    //输出排序后的 d2
  cout<<endl;
  partial_sort_copy(dim,dim+9,v3.begin(),v3.end());
                                          //对初始数组 dim 排序，并复制到 v3 中
  cout<<"vector v3(partial_sorted( 默认升序 )). : "<<endl;
  copy(v3.begin(),v3.end(),ostream_iterator<int>(cout,", "));    //输出 v3
```

```
    cout<<endl;
    partial_sort_copy(v3.begin(),v3.end(),d3.begin(),d3.end(),greater<int>());
                                                //对 v3 进行降序排序，并复制到 d3 中
    cout<<"deque d3(partial_sorted( 降序 )). : "<<endl;
    copy(d3.begin(),d3.end(),ostream_iterator<int>(cout,", "));     //输出 d3
    cout<<endl;
}
```

例 4-27 的执行效果见图 4-27。



```
选定 "F:\2012年写书\例题\第四章\EX4-27\Debug\Ex4-27.exe"
Dim(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v1(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
deque d1(Original) is below :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v1 is sorted by the partial_sort() algorithm .  :
1, 2, 3, 4, 5, 8, 7, 9, 6,
deque d1  is sorted by the partial_sort() algorithm .  :
1, 2, 3, 4, 5, 8, 7, 9, 6,
------------------
vector v2 is as same as the original v1 :
1, 2, 4, 6, 5, 8, 7, 9, 3,
deque d2 is as same as the original d1 :
1, 2, 4, 6, 5, 8, 7, 9, 3,
vector v2 is sorted by the partial_sort() algorithm ( 降序 )). :
9, 8, 7, 6, 5, 1, 2, 4, 3,
deque d2 is sorted by the partial_sort() algorithm ( 降序 ) ). :
9, 8, 7, 6, 5, 1, 2, 4, 3,
vector v3(partial_sorted( 默认升序 )). :
1, 2, 3, 4, 5, 6, 7, 8, 9,
deque d3(partial_sorted( 降序 )). :
9, 8, 7, 6, 5, 4, 3, 2, 1,
Press any key to continue
```

图 4-27 例 4-27 的执行效果

4.4.3 根据某个元素排序

STL 提供了算法函数 nth_element()。该算法可以对指定区间内的元素进行排序，并使第 n 个位置上的元素就位。即所有在位置 n 之前的元素都小于等于它，所有在位置 n 之后的元素都大于等于它。因此，可得到根据位置 n 上的元素分割出两个子序列。第一子序列的元素全部小于第二子序列的元素。如果程序开发人员只需要 n 个最大或最小元素，但不要求这些元素是已序的，此时本算法会提供很大帮助。

使用函数 nth_element()之后的序列中，元素是已序的，并且所有小于第 n 个元素的元素出现在该元素前面，而大于第 n 个元素的元素出现在该元素后面。

函数原型如下：

```
    template<class RandomAccessIterator> void nth_element( RandomAccessIterator first,
RandomAccessIterator nth,
        RandomAccessIterator last);
    template<class  RandomAccessIterator,  class  Compare>   void  nth_element
(RandomAccessIterator first,
        RandomAccessIterator nth, RandomAccessIterator last, Compare comp);
```

在上述两种形式的函数原型中，迭代器属于随机存取型迭代器。第一种形式属于常规形式，第二种形式包含二元判断式，并使用二元判断式 comp(elem1,elem2)作为排序规则。

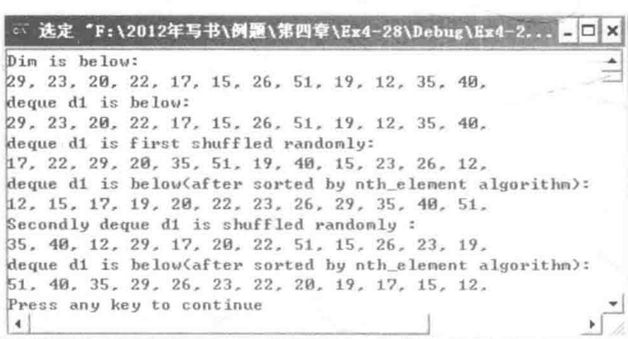
函数 nth_element()和 partial_sort()有相似之处。其区别在于：partial_sort()在实现局部排序时，指定位置作为划分整个区间的界线；而 nth_element()在实现排序时，是指定区间中的某个元素作为

划分整个区间的界线。具体说明见例 4-28。

例 4-28

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <functional>
using namespace std;
void main()
{   int dim[]={29,23,20,22,17,15,26,51,19,12,35,40};
    deque<int> d1;
    copy(dim,dim+12,back_inserter(d1));
    cout<<"Dim is below: "<<endl;
    copy(dim,dim+12,ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"deque d1 is below: "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    random_shuffle(d1.begin(),d1.end());
    cout<<"deque d1 is first shuffled randomly: "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    nth_element(d1.begin(),d1.begin()+4,d1.end());
    cout<<"deque d1 is below(after sorted by nth_element algorithm): "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    random_shuffle(d1.begin(),d1.end());
    cout<<"Secondly deque d1 is shuffled randomly : "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    nth_element(d1.begin(),d1.begin()+6,d1.end(),greater<int>());
    cout<<"deque d1 is below(after sorted by nth_element algorithm): "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-28 的执行效果见图 4-28。



```
选定 "F:\2012年写书\例题\第四章\Ex4-28\Debug\Ex4-2...
Dim is below:
29, 23, 20, 22, 17, 15, 26, 51, 19, 12, 35, 40,
deque d1 is below:
29, 23, 20, 22, 17, 15, 26, 51, 19, 12, 35, 40,
deque d1 is first shuffled randomly:
17, 22, 29, 20, 35, 51, 19, 40, 15, 23, 26, 12,
deque d1 is below(after sorted by nth_element algorithm):
12, 15, 17, 19, 20, 22, 23, 26, 29, 35, 40, 51,
Secondly deque d1 is shuffled randomly :
35, 40, 12, 29, 17, 20, 22, 51, 15, 26, 23, 19,
deque d1 is below(after sorted by nth_element algorithm):
51, 40, 35, 29, 26, 23, 22, 20, 19, 17, 15, 12,
Press any key to continue
```

图 4-28　例 4-28 的执行效果

提示：程序的执行结果可能和读者想象的不一样。请详细阅读并深刻思考。

4.4.4　堆（Heap）操作

“堆”属于计算机领域的一个术语。在计算机算法领域，“堆”通常用于指一种组织序列元素的方式。“堆”的第一个元素通常是具有最大值的元素。STL 的算法库中提供了部分堆操作算法：push_heap()，pop_heap()，sort _ heap ()，make_heap()等。

“堆”（Heap）应用于排序，是一种特别的元素组织方式，是以序列式群集实做而成的二叉树。它具有两大性质：堆中的第一个元素的值总是最大；能够在对数时间内增加或移除一个元素。

堆的默认排序规则为“operator <”。程序开发人员也可以使用自定义的排序准则。下面分别讲述各种“堆”算法。

1．算法 make_heap()

算法 make_heap()的函数模型如下：

```
void make_heap(RandomAccessIterator _First, RandomAccessIterator _Last );
void make_heap(RandomAccessIterator _First, RandomAccessIterator _Last, BinaryPredicate _Comp);
```

上述两种形式的函数模型，均实现将区间[_First, _last]中的元素转化为“堆”。参数_Comp 是二元判断式，是排序准则。使用“堆”处理的条件是：元素多于 1 个。如果仅有 1 个元素，没必要使用“堆”，也可以认为单个元素本身就是“堆”。

2．算法 push_heap()

算法 push_heap()实现在其参数指定区间中插入一个元素，新插入的元素放在堆栈末尾，即尾指针处。使用算法 push_heap()时，必须保证区间原有的元素已是既定的“堆”。算法 push_heap()的函数原型如下：

```
void push_heap( RandomAccessIterator _First, RandomAccessIterator _Last);
void push_heap( RandomAccessIterator _First, RandomAccessIterator _Last, BinaryPredicate _Comp );
```

3．算法 pop_heap

算法 pop_heap()的作用是删除在迭代器[_First,_Last]指定范围（“堆”）内的元素序列中的最大元素（第一个元素）。剩余的元素成为一个新的“堆”。算法 pop_heap() 的原型如下：

```
void pop_heap(RandomAccessIterator _First, RandomAccessIterator _Last );
void pop_heap(RandomAccessIterator _First, RandomAccessIterator _Last, BinaryPredicate _Comp);
```

4．算法 sort_heap

算法 sort_heap()的作用是对其参数迭代器指定的范围内的元素进行排序。其函数原型如下：

```
void sort_heap(RandomAccessIterator _First, RandomAccessIterator _Last );
void sort_heap(RandomAccessIterator _First, RandomAccessIterator _Last, BinaryPredicate _Comp );
```

算法 sort_heap()可以使堆[beg, end]转化为一个已序的序列。算法执行之后，该区间及其区间内的元素，就不再是“堆”了。

提示：堆的内部数据结构为二叉树。输出的堆数据有时看上去有些混乱，但其实并不混乱，它在内存中是以二叉树的形式存在的。

以上内容用例 4-29 进行说明。

例 4-29

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void main()
{   //序列中元素发生变化之后，二叉树全部重新排列
    vector<double> v1;
    double dim[]={1.2, 4.3, 5.6, 2.3, 7.8, 8.2, 4.2, 8.1, 5.3};
    int cnt=sizeof(dim)/sizeof(double);
    v1.assign(dim,dim+cnt);
    cout<<"The Original Vector Elements are below: "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    //make_heap
    make_heap(v1.begin(),v1.end());
    cout<<"The heap is below made by the vector v1: "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    pop_heap(v1.begin(),v1.end());   //
    v1.pop_back();      //移除最后一个元素
    cout<<"The heap is below which the first element has been erased. : "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    //push
    v1.push_back(13.2);
    push_heap(v1.begin(),v1.end());
    cout<<"The heap is below which the new element has been pushed. : "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    //sort_heap;  after sort_heap, the vector will not be a heap.
    sort_heap(v1.begin(),v1.end());
    cout<<"After the heap is sorted by \'sort_heap\', The vector is not a heap. : "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
}
```

例 4-29 的执行效果见图 4-29。

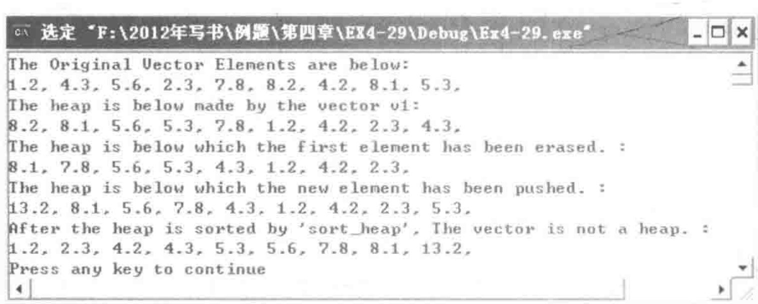


图 4-29　例 4-29 的执行效果

4.4.5　合并排序

本节介绍的算法用来将两个容器或区间的元素合并，合并运算在本节中主要包括总和、并集、交集等。

1. 两个已序集合的总和（merge）

算法 merge()可用来处理两个已序集合的总和。其函数原型如下：

```
    OutputIterator merge(InputIterator first1, InputIterator last1,InputIterator
first2, InputIterator last2, OutputIterator x);
    OutputIterator merge(InputIterator first1, InputIterator last1,InputIterator
first2, InputIterator last2, OutputIterator x,
    Compare comp);
```

上述两种形式均是将两个区间[first1, last1]和[first2,last2]内的元素合并，并将合并后的所有元素保存在以 x 为起始的目标区间。目标区间内的所有元素都将按顺序排列。函数返回值为目标区间的最后一个被复制元素的下一位置。二元判断式 comp 是可有可无的，用户可以自定义二元判断式，作为序列的排序准则。merge()操作对两个源区间没有任何影响。

源区间（源容器）中的元素应该是已序的。在多数情况下，merge()算法可以将两个无序的容器（源区间）合并到一个无序的目标区间中。通常，使用 copy()函数会更方便些。在使用 merge()算法时，要保证目标区间足够大，否则需要使用插入型迭代器。

一般情况下，目标区间和源区间不允许重复。但是，算法 merge()没有保证新生成的区间（容器）中元素的唯一性。

对于 list 型容器，提供了一个特殊成员函数，用于合并两个 list 型容器中的元素。

2. 两个已序集合的并集（union）

算法 set_union()用于实现将已序的两个区间内的元素合并，获得以指定起始位置开始的目标区间，新区间的元素来自于两个源区间，但是重复的元素被唯一化。函数原型如下：

```
    OutputIterator set_union(InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _Result );
    OutputIterator set_union(InputIterator1 _First1, InputIterator1 _Last1, InputIterator2
_First2, InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate _Comp);
```

上述函数模型，[_First1, _Last1]和[_First2, _Last2]表示两个源区间，参数_Result 代表新生成的目标区间的起始位置。

当源区间内本身就含有重复元素时，在使用算法 set_union()时，会保留这些重复元素。此时元素的重复个数是两个源区间中重复个数较大的那个数值。

_Comp 是一个二元判断式，可作为排序准则。

3. 两个已序集合的交集（intersection）

算法 set_intersection()用于实现将已序的两个区间的元素合并，新生成区间内的元素应该是那些同时属于两个源区间（容器）的元素。如果两个源区间内原来存在重复元素，新生成的目标区间内也包含重复元素，重复个数为两个源区间中较小的那个重复个数。

算法 set_intersection()的函数原型如下：

```
    OutputIterator set_intersection(InputIterator1 _First1, InputIterator1 _Last1,
InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result );
```

```
    OutputIterator  set_intersection(InputIterator1  _First1,  InputIterator1  _Last1,
InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate
_Comp );
```

函数返回值是 OutputIterator 型迭代器，迭代器指向目标区间的最后一个被“合并”元素的下一个位置。目标区间内所有元素均按顺序排列。

4. 两个已序集合的差集（difference）

算法 set_difference()实现在两个源区间之间的求差（difference），新生成的区间中的元素仅仅包含在第一个源区间中的元素。算法 set_difference()的函数原型如下：

```
    OutputIterator  set_difference(InputIterator1  _First1,  InputIterator1  _Last1,
InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result);
    OutputIterator  set_difference(InputIterator1  _First1,  InputIterator1  _Last1,
InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate
_Comp);
```

第一个源区间中所有存在于第二个源区间中的元素将不存在于新生成的区间中。和前面的合并算法一样，新生成的目标区间的元素均是已序的。

如果源区间内有重复元素，则目标区间内相应的也会包含重复元素。重复元素的个数是第一源区间内的重复个数减去第二源区间内的相应的重复个数，如果第二个源区间内的重复个数大于第一源区间内的相应重复个数，目标区间内的对应重复个数将等于 0。

算法对源区间不做任何修改，并且两个源区间都是已序的。目标区间和源区间不允许重叠。

另外还存在另一算法 set_symmetric_difference()。该算法实现两个源区间的合并，并生成新区间，新区间中不包含两个源区间中共同存在的元素。和前面的合并算法一样，新生成的目标区间的元素均是已序的。

算法的函数原型如下：

```
    OutputIterator  set_symmetric_difference( InputIterator1  _First1,  InputIterator1
_Last1, InputIterator2 _First2,
        InputIterator2 _Last2, OutputIterator _Result );
    OutputIterator  set_symmetric_difference( InputIterator1  _First1,  InputIterator1
_Last1,IInputIterator2 _First2,
        InputIterator2 _Last2, OutputIterator _Result, BinaryPredicate _Comp );
```

5. 连贯的已序区间合并算法

算法 inplace_merge()可以将两个已序区间[beg1, end1beg2]和[end1beg2, end2]的元素合并，使区间 [beg1,beg2]成为两者之和。算法的函数原型如下：

```
    void inplace_merge( BidirectionalIterator _First, BidirectionalIterator _Middle,
BidirectionalIterator _Last );
    void inplace_merge( BidirectionalIterator _First, BidirectionalIterator _Middle,
BidirectionalIterator _Last,
            BinaryPredicate _Comp  );
```

下面以例 4-30 来阐释上述几种合并算法的使用方法。

例 4-30

```
    #pragma warning(disable:4786)
    #include <iostream>
```

```
#include <list>
#include <set>
#include <algorithm>
#include <functional>
using namespace std;
/*
 merge, set_union, set_difference, set_intersection ,
*/
void print(int& elem)
{
  cout<<elem<<", ";
}
void main()
{
  int dim[]={1,2,3,4,5,6,7,8};
  int dim2[]={3,4,5,6,7,8,9,10};
  list<int> l1;
  set<int> s1;
  for(int i=0;i<sizeof(dim)/sizeof(int);i++)
    l1.push_back(dim[i]);
  for(i=0;i<sizeof(dim2)/sizeof(int);i++)
    s1.insert(dim2[i]);
  cout<<"list l1: "<<endl;
  copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"list s1: "<<endl;
  copy(s1.begin(),s1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"l1 and s1 are merged: "<<endl;
  merge(l1.begin(),l1.end(),s1.begin(),s1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"l1 and s1 are merged by algorithm set_union: "<<endl;
  set_union(l1.begin(),l1.end(),s1.begin(),s1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"l1 and s1 are merged by algorithm set_difference: "<<endl;
  set_difference(l1.begin(),l1.end(),s1.begin(),s1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"l1 and s1 are merged by algorithm set_intersection: "<<endl;
  set_intersection(l1.begin(),l1.end(),s1.begin(),s1.end(),ostream_iterator<int>(cout,", "));
  cout<<endl;
  cout<<"l1 and s1 are merged by algorithm inplace_merge: "<<endl;
  copy(s1.begin(),s1.end(),back_inserter(l1));
  list<int>::iterator pos;
  pos=find(l1.begin(),l1.end(),3);
  pos=find(++pos,l1.end(),3);
  pos++;
  //int d=distance(l1.begin(),pos);
  inplace_merge(l1.begin(),pos,l1.end());    //inplace_merge没有实现排序的效果
  l1.sort();
  for_each(l1.begin(),l1.end(),print);
  cout<<endl;
}
```

例 4-30 的执行效果见图 4-30。

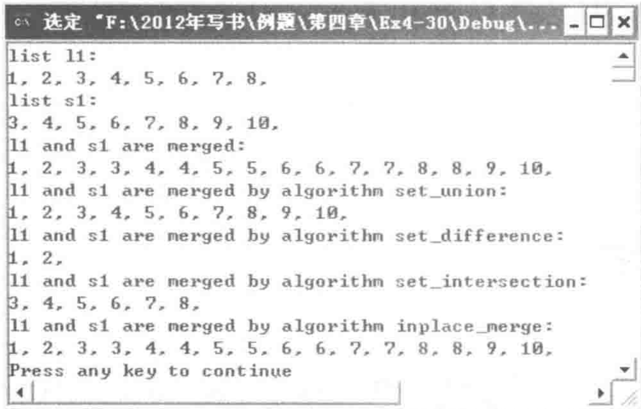


图 4-30　例 4-30 的执行效果

4.4.6　搜索

1. 算法 binary_search()

在 STL 中，对于已序区间提供了搜索算法 binary_search()。它可实现在已序区间中搜寻指定元素。算法的函数原型如下：

```
bool binary_search(ForwardIterator _First,ForwardIterator _Last,const Type& _Val);
bool binary_search(ForwardIterator _First, ForwardIterator _Last, const Type& _Val,
BinaryPredicate _Comp );
```

以上两种函数形式均可用来判断已序区间[_First, _Last]中是否包含值为_Val 的元素。_Comp 是一个可有可无的二元判断式，用来作为排序准则。如果搜寻被搜寻元素的位置，应使用 lower_bound()、upper_bound()和 equal_range()。

使用算法 binary_search()之前，被搜索的区间必须是已序的。

2. 算法 includes()

算法 includes ()可用于在指定源区间中检查若干值是否存在。在 Visual C++6.0 的 msdn 文档中提供的函数原型如下：

```
inline bool  includes(InputIterator1 First1,InputIterator1 Last1,InputIterator2
First2,InputIterator2 Last2)
```

而 C++语言的 ISO_IEC 标准中提供了两种函数形式，其中一种与上述形式相同，另一种形式如下：

```
Bool  includes(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
InputIterator2 last2, Compare comp )
```

3. 搜索第一个或（和）最后一个可能位置

STL 还提供了算法 lower_bound()、equal_range()和 upper_bound()。算法 lower_bound()返回第一个“大于等于 value”的元素位置。这是指定元素的第一个位置。算法 equal_range()和 lower_bound()一样，要求被搜索的源区间是已序的。算法 equal_range()的返回值其实是 lower_bound()和 upper_bound()的共同返回值。

注意：使用 lower_bound()和 upper_bound()时，如果未找到指定的元素，函数返回 end()指向位置。

以上 3 个算法函数的使用方法见例 4-31。

例 4-31

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
#include <functional>
using namespace std;
void main()
{  int dim[]={1,2,3,4,5,6,7,8,9};
   list<int> l1;
   list<int>::iterator posL,posU;
   pair<list<int>::iterator,list<int>::iterator> range_Iterator;
   copy(dim,dim+9,back_inserter(l1));
   copy(dim,dim+9,back_inserter(l1));
   cout<<"list l1 is as below. "<<endl;
   copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   l1.sort();
   cout<<"list l1 has been sorted. "<<endl;
   copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   bool sh=binary_search(l1.begin(),l1.end(),9);
   if(sh)
   {  cout<<"9 can be searched in list l1."<<endl;
   }
   else
   {  cout<<"9 can't' be searched in list l1."<<endl;
   }
   vector<int> s;
   int dim2[]={3,4,5,6};
   s.assign(dim2,dim2+4);
   sh=includes(l1.begin(),l1.end(),s.begin(),s.end());
   if(sh)
   {  cout<<"vector s can be included in list l1. "<<endl;
   }
   else
   {  cout<<"vector s can't' be included in list l1. "<<endl;
   }
   posL=lower_bound(l1.begin(),l1.end(),8);
   posU=upper_bound(l1.begin(),l1.end(),8);
   range_Iterator=equal_range(l1.begin(),l1.end(),8);
   if(posL!=l1.end())
     cout<<"8 can be positioned "<<distance(l1.begin(),posL)<<". "<<endl;
 if(posU!=l1.end())
      cout<<"8 can be positioned "<<distance(l1.begin(),posU)<<" secondly. "<<endl;
 cout<<"8 can be positioned from "<<distance(l1.begin(),range_Iterator.first)<<" to
"<<distance(l1.begin(),
 range_Iterator.second)<<endl;
 }
```

例 4-31 的执行效果见图 4-31。

图 4-31　例 4-31 的执行效果

4.5 删除算法

删除算法指的是删除容器（或序列）中满足相应条件的元素。

1. 算法 remove()

查找并删除容器（区间）中的元素是最基本的容器操作之一。STL 提供了算法 remove()，它可以直接进行元素删除操作，算法的函数原型如下：

```
ForwardIterator remove( ForwardIterator _First, ForwardIterator _Last, const Type& _Val );
```

算法 remove()的功能是实现在迭代器[_First, _Last]指定范围内删除所有值为_Val 的元素。

切记！函数返回值是删除元素之后容器（或区间）的新末尾位置，此返回值是非常重要的。

2. 算法 remove_if()

STL 还提供了条件删除算法：remove_if()。条件删除算法 remove_if()会移除容器（区间）中每个“使其条件判断式为 true”的元素。

被删除元素之后的元素会按顺序向前移动。由于删除算法会导致元素变动，所以它不能应用于关联式容器，只能应用于顺序式容器。对于那些关联式容器，需要使用算法 erase()来实现删除容器中元素的目的。对于容器类型 list，在其类模板中提供了一个成员函数 remove()，具有更加好的性能。算法 remove_if()的函数原型如下：

```
ForwardIterator remove_if(ForwardIterator First,ForwardIterator Last, Predicate Pred )
```

3. remove_copy()和 remove_copy_if()

算法 remove_copy()和 remove_copy_if()用于在复制过程中移除相关的元素。如果使用条件删除语句，只有当一元判断式为“true”时，才正确执行算法 remove_if()。两个算法函数的原型如下：

```
OutputIterator remove_copy( ForwardIterator First, ForwardIterator Last, OutputIterator Result,const T& Value )
OutputIterator remove_copy_if(InputIterator First, InputIterator Last, OutputIterator _Result,Predicate _Pred);
```

其中，算法 remove_copy_if()是 copy()和 remove_if()的组合，其将源区间[_First, _Last]内的元素复制到以_Result 为起始的目标区间中。在复制过程中，删除所有使一元判断式_Pred 为“true”的元素。在使用这两个算法时，如果目标区间的容量不够大，必须使用插入型迭代器。

函数返回值为目标容器（区间）的最后一个元素的位置。

4. 移除重复元素

在 vector、list、deque、multiset 和 multimap 等类型的容器中，允许元素重复出现，但是有时需要删除重复的元素。STL 提供了算法 unique()，可以实现容器中的元素是唯一的。代码如下：

```
ForwardIterator unique(ForwardIterator _First,ForwardIterator _Last );
ForwardIterator unique(ForwardIterator _First,ForwardIterator _Last, BinaryPredicate _Comp );
```

上述两种 unique()的语法形式，其区别在于增加了比较函数_Comp()。函数返回值是修正后的容器最后一个元素位置。

5. 复制过程中移除重复元素

STL 提供了两种形式的算法 unique_copy() 。其函数原型如下：

```
OutputIterator unique_copy(InputIterator _First, InputIterator _Last, OutputIterator _Result );
OutputIterator unique_copy(InputIterator _First, InputIterator _Last, OutputIterator _Result,BinaryPredicate _Comp,);
```

算法 unique_copy()均实现复制容器（区间）[_First, _Last]中的元素至以_Result 为起始位置的目标容器或目标区间，并移除重复的元素。移除重复元素的前提是源区间必须是已序的。

函数返回值是最后一个被复制的元素的下一位置。同样，在使用时必须确保目标区间有足够的容量，否则需要使用插入型迭代器。当判断式_Comp 为 true 时，将该元素删除。此判断式并非用来将元素和其原本的前一个元素比较，而是将它和未被移除的前一个元素比较。

下面以例 4-32 来说明上述几个删除算法的使用方法。

例 4-32

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
bool mylessthan(int ele1,int ele2)
{
if(ele2>3||ele1>3)
     return true;
 else
     return false;
 }
void main()
{int source[]={1,4,4,6,1,2,2,3,1,6,6,6,5,7,5,4,4};
   int num=sizeof(source)/sizeof(int);
   vector<int> l1,l11(num,0);
   vector<int> l2(num,0),l3(num,0);
   vector<int>::iterator it1;
   l1.assign(source,source+num);
   cout<<"The Original vector l1 is as below. "<<endl;
   copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
   cout<<endl;
   it1=remove(l1.begin(),l1.end(),1);
   l1.erase(it1,l1.end());
```

```
    cout<<"The Original vector l1 is as below after removing 1. "<<endl;
     copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    it1=remove_if(l1.begin(),l1.end(),bind2nd(greater<int>(),6));
                                        //条件表达式 bind2nd(greater<int>(),6)
    l1.erase(it1,l1.end());
    cout<<"Removed the elements which are larger than 6. "<<endl;
     copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    it1=remove_copy(l1.begin(),l1.end(),l2.begin(),6);      //返回目标容器的终止位置
    cout<<"Removing the elements 6. "<<endl;
     copy(l2.begin(),it1,ostream_iterator<int>(cout,", "));
    cout<<endl;
    it1=remove_copy_if(l2.begin(),l2.end(),l3.begin(),bind2nd(less<int>(),3));
                                                            //返回目标容器的终止位置
    cout<<"Removing the elements <3. "<<endl;
     copy(l3.begin(),it1,ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"Unique the elements. "<<endl;
    it1=unique(l3.begin(),l3.end());
     copy(l3.begin(),--it1,ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"Unique_copy the elements. "<<endl;
    l1.assign(source,source+num);
    it1=unique_copy(l1.begin(),l1.end(),l11.begin());
     copy(l11.begin(),--it1,ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"Unique_copy(if) the elements. "<<endl;
    l1.assign(source,source+num);
    it1=unique_copy(l1.begin(),l1.end(),l11.begin(),mylessthan);
     copy(l11.begin(),--it1,ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 4-32 的执行效果见图 4-32。

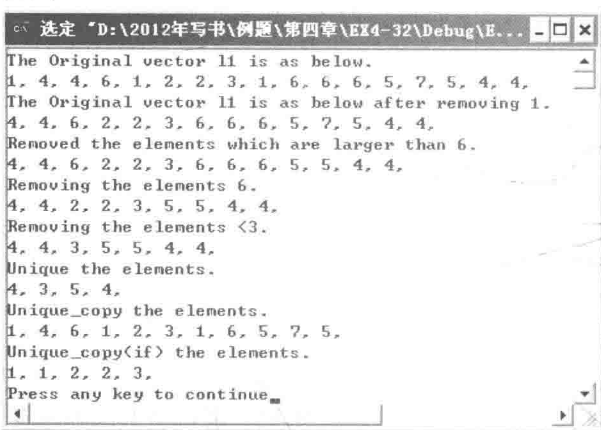


图 4-32　例 4-32 的执行效果

提示：本节讲述了 5 种和删除操作有关的算法，这 5 种算法各有特色，值得读者认真学习。如果学会了删除容器中的元素，对程序员的益处将非常大。

4.6　本章小结

本章重点介绍了 C++ STL 中的各种算法。按大类分主要包括：非修改性序列算法，变动性算法，排序及相关操作，删除算法等。通过本章的学习，读者应该较全面地掌握了 STL 的各种算法，尤其是较常用的算法，例如：for_each()，copy()，max(min)，compare()，swap()，transform()，reverse()以及各种排序、排列算法和 remove()等。

第 5 章 迭代器（Iterator）

迭代器是连接容器和算法的纽带，它为数据提供了一种抽象的观点，使写算法的人不必关心各种各样的数据结构和具体细节。迭代器提供了一个数据访问的标准模型，缓解了要求容器提供一组更广泛的访问操作的压力。

迭代器用对象序列作为它所支持的数据访问模型。将具有数组和字节形式的低级数据模型映射到高级的对象模型。迭代器是每个程序员都需要熟悉的概念之一。

5.1 迭代器及其特性

迭代器本身是一种对象。所以，即使不是迭代器，但其行为类似迭代器的东西，也是迭代器。迭代器不是通用指针，而是指向数组的指针概念的抽象。通俗地讲，迭代器就是一个指示器。迭代器技术能够使程序反复地对 STL 容器内容进行访问。当参数化类型为 C++内部类型时，迭代器是 C++的指针。STL 定义了 5 种类型的指示器，根据使用方式而命名。每种容器类型都支持某种类别的迭代器，这些类别分别为：输入、输出、前向、双向和随机访问。

输入迭代器主要用于为程序中需要的数据源提供输入接口，此处数据源可以是容器、数据流等。输入迭代器只能够从一个序列中读取数据，此类迭代器可以被修改、引用并进行比较。

输出迭代器主要用于输出程序中已经得到的数据结果，此处结果数据指的是容器、数据流等。输出迭代器只能向一个序列写入数据，此类迭代器可以被修改和引用。

前向迭代器可以随意访问序列中的元素，许多 STL 算法函数需要提供前向迭代器。前向迭代器可以用来读也可以用来写，结合了输入和输出迭代器的功能，并能够保存迭代器的值，以便从其原先位置开始重新遍历。

双向迭代器既可以用来读也可以用来写，双向迭代器可以被增值和减值，并且可以同时进行前向和后向元素的操作。所有的 STL 容器都提供了双向迭代器功能，以有利于数据的写入和读出。

随机访问迭代器可以通过跳跃的方式访问容器中的任意数据，从而使数据的访问非常灵活。随机访问迭代器作为功能最强大的迭代器类型，具有双向迭代器的所有功能，能够使用算法和所有的迭代器比较功能。

迭代器是指向序列元素的指针概念，关键的属性如下：

- 当前被指向的元素，用“*”或“->”表示；

- 指向下一个元素，迭代器的增量使用运算符“++”；
- 相等，使用运算符“==”；
- 只有随机存取迭代器可以通过加减整数，取得相对地址；
- 除了输出迭代器之外，可以获得任意两个迭代器之间的位置（使用 distance 函数）。

5.2　头文件<iterator>简述

所有容器都定义其各自的迭代器型别，当程序开发人员使用某种容器的迭代器时，不需要含有专门的头文件。但是逆向（reverse）迭代器被定义于头文件<iterator>之中。

5.3　迭代器类型详述

前面已讲过，迭代器是一种“能够遍历某个序列内的所有元素”的对象，可以通过与一般指针一致的接口来完成自己的工作。迭代器奉行纯抽象概念：任何东西，只要行为类似迭代器，就是一种迭代器。不同的迭代器具有不同的“能力”（行进和存取能力）。某些算法需要特定的迭代器能力，能力是很重要的概念。

不同种类的迭代器可以排成如图 5-1 所示的层次结构。

图 5-1　各种迭代器的关系

一方面，不同的算法需要不同的迭代器作为参数。另一方面，有时同一个算法对于不同的迭代器也可能有效率不同的实现。为支持基于迭代器类别进行重新解析，标准库提供了 5 个类，分别代表 5 个迭代器类别：输入型迭代器；输出型迭代器；前向迭代器；双向迭代器；随机存取迭代器。

输入型迭代器用于向前读取；输出型迭代器用于向前写入；前向迭代器用于读取和写入；双向迭代器可用于向前和向后读取及写入；随机存取迭代器既可用于读取也可用于写入。

5.3.1　输入型迭代器

输入型迭代器（InputIterator）：只能一次一个地向前读取元素，并按此顺序传回元素值。InputIterator 迭代器只能读取元素一次。如果复制输入型迭代器，原 InputIterator 迭代器和新产生的副本都向前读取，会遍历到不同的值。所有的迭代器都具备输入型迭代器的能力，纯粹迭代器的典型例子是“从标准输入装置读取数据”的迭代器。同一个值不会被读取两次，一旦从输入流读入一个字后，下次读取时就会传回另一个字。

如果两个输入型迭代器占用同一个位置，则两者相同。输入型迭代器的操作如下。

- *：用于从迭代器中读取实际元素；
- ->：读取实际元素的成员；
- ++：无论是放在迭代器的前边还是后边，均代表向前步进；
- ==：判断两个迭代器是否相等；

- ！=：判断两个迭代器是否不相等；
- TYPE（iter）：复制迭代器。

使用递增迭代器时，应尽量使用前置式递增操作符，而不是后置式递增操作符，因为前置式递增操作符性能更好。

5.3.2 输出型迭代器

输出型迭代器和输入型迭代器相反，其作用是将元素值逐个写入。即只能逐个元素进行赋值，不能使用 OutputIterator 型迭代器对同一序列进行两次遍历。输出型迭代器可以实现“*”、“++”和复制操作。一般迭代器可以读取和写入元素值，几乎所有迭代器都具有输出型迭代器的功能。

输出型迭代器可举两例：一个是“将元素写至标准输出装置“的迭代器，如果采用两个输出迭代器写至屏幕，第二个字将跟在第一个字后面，而不是覆盖第一个字；另一个典型例子是插入器（inserter），所谓插入器是用来将元素值插入容器内的一种迭代器。

5.3.3 前向迭代器

前向迭代器是输入型迭代器和输出型迭代器的结合，具有输入型迭代器的全部功能和输出型迭代器的大部分功能。前向迭代器的所有操作见表 5-1。

表 5-1 前向迭代器的各项操作

表达式	效果	表达式	效果
*Iter	存取实际元素	Iter1!=iter2	判断两个迭代器不相等
Iter->number	存取实际元素成员	TYPE()	产生迭代器（默认构造函数）
++Iter	向前步进	TYPE(iter)	复制迭代器（copy）
Iter++	向前步进	Iter1=iter2	赋值
Iter1==iter2	判断两个迭代器相等		

前向迭代器能多次指向同一集合的同一元素，并能多次处理同一元素。

- Output 迭代器在使用时，无须检查是否抵达序列尾端，可直接写入数据。Output 迭代器不提供比较操作，不能将 Output 迭代器和尾端迭代器相比较。
- Forward 迭代器在使用之前，需要确定迭代器是否有效？因此，无论使用哪种迭代器，在循环时应尽量使用 begin()和 end()作为循环的起始。

5.3.4 双向迭代器

双向迭代器在前向迭代器的基础上增加了回头遍历的功能。可以支持递减运算符，可以实现一步一步的后退操作。双向迭代器的各项操作见表 5-2。

表 5-2 双向迭代器的各项操作

表达式	效果	表达式	效果
--iter	步退（传回新位置）	Iter--	步退（传回老位置）

5.3.5 随机存取迭代器

随机存取迭代器在双向迭代器的基础上增加了随机存取能力。因此，必须增加（提供）迭代器

算数运算。即可以实现迭代器加减某个偏移量、能处理距离问题，并可运用诸如“<”和“>”等相互关系运算操作符实现比较操作。

可以支持随机存取迭代器的容器对象或数据类型包括：

```
vector, deque,strings(string, wstring),普通数组 array。
```

随机存取迭代器的各项操作见表 5-3。

表 5-3　随机迭代器的各项操作

表达式	效果	表达式	效果
Iter[n]	存取索引位置为 n 的元素	Iter1–iter2	传回 iter1 和 iter2 的距离
Iter+=n	前进 n 个元素	Iter1<iter2	判断 iter1 是否在 iter2 之前
Iter–=n	后退 n 个元素	Iter1>iter2	判断 iter1 是否在 iter2 之后
Iter+n	传回 iter 之后的第 n 个元素	Iter1<=iter2	判断 iter1 是否在 iter2 之后
n+iter	传回 iter 之后的第 n 个元素	Iter1>=iter2	判断 iter1 是否不在 iter2 之前
Iter–n	传回 iter 之前的第 n 个元素		

随机存取迭代器提供了上述丰富的运算功能，同时也带来诸多麻烦。例如下面几例会导致诸多问题。

```
 vect.end()-1;                                   //迭代器指针可能指向 begin()之前
 pos+=2;                                         //迭代器指针可能超越 end()位置
for(pos=vect.begin(); pos!=vect.end(); pos+=2)   //迭代器可能超越容器末尾
{
… …
}
```

提示：随机存取迭代器对 list、sets 和 maps 是无效的。

5.3.6　vector 迭代器的递增和递减

迭代器的递增问题较复杂，程序员可以递增或递减暂时性迭代器，对于 vector 和 string 两种容器类型来说则不同（下面内容摘自《C++标准程序库》）。

```
std::vector<int> v;
if(v.size()  > 1)
{
sort(++v.begin(),v.end())
}
```

上述代码在编译时，通常会出错。但如果将容器类型换成 deque，就可以通过编译。对于 vector 的其他操作，有时也可以通过编译。

上述现象的原因一般为：vector 迭代器在实际操作中通常作为一般指针。C++不允许修改任何基本型别的暂时值，对于 struct 和 class 则可以。如果迭代器被实例化为一般指针，编译会失败；如果迭代器被实例化为 class，则编译可以成功。deque、lists、sets 和 maps 总是能通过编译，这些容器对象的迭代器不可能被实例化为一般指针。

5.4　迭代器配接器

C++标准程序库提供了多个预先定义的特殊迭代器，即所谓迭代器配接器（iterator adapters）。

迭代器配接器使算法能够以逆向模式、安插模式进行工作，还可以和流配合。迭代器配接器不仅能起辅助作用，还能赋予整个迭代器抽象概念更强大的能力。

5.4.1 逆向迭代器

Reverse 迭代器是一种配接器，它能重新定义递增运算和递减运算，使其行为正好倒置。算法以逆序次序来处理元素。所有标准容器都允许使用 Reverse 迭代器来遍历元素。下面用例 5-1 来加以说明。

例 5-1

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int ele)
{
 cout<<ele<<", ";
}
void main()
{
 list<int> l1;
 for(int i=1;i<=9;i++)
     l1.push_back(i);
 for_each(l1.begin(),l1.end(),print);
 cout<<endl;
 for_each(l1.rbegin(),l1.rend(),print);
 cout<<endl;
}
```

例 5-1 的执行效果见图 5-2。

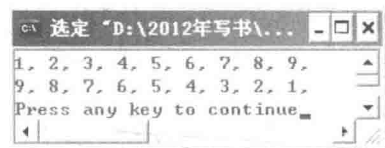


图 5-2 例 5-1 的执行效果

在例 5-1 中，容器的成员函数 rbegin()和 rend()会分别传回一个 Reverse 迭代器，就像 begin()和 end()的返回值一样，共同定义出一个半开区间。其中 rbegin()传回逆向遍历的第一元素位置，即实际的最后一个元素位置；rend()传回逆向遍历时最后一个元素的下一个位置，即第一个元素的前一个位置（已经不属于本容器了）。

通常迭代器可以转化成一个 Reverse 迭代器。原本那个迭代器必须具有双向移动能力。下面用例 5-2 说明这个现象。

```
vector<int> ::iterator pos;
vector<int> v1;
pos=find(v1.begin(), v1.end(),5);
 vector<int>::reverse_iterator rpos(pos);
 cout<<"*rpos"<<*rpos<<endl;
```

上述代码实现将迭代器转化为逆迭代器，并输出其值。

例 5-2

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(int ele)
```

```
{
 cout<<ele<<", ";
}
void main()
{
 list<int> l1;
 for(int i=1;i<=9;i++)
     l1.push_back(i);
 cout<<"vector :"<<endl;
 for_each(l1.begin(),l1.end(),print);
 cout<<endl;
 list<int>::iterator it;
 it=find(l1.begin(),l1.end(),5);
 cout<<"pos: "<<*it<<endl;
 list<int>::reverse_iterator itR(it);
 cout<<"rpos: "<<*itR<<endl;
}
```

例 5-2 的执行效果见图 5-3。

逆向迭代器还可以转换为正常迭代器。逆向迭代器类模板提供了一个 base()成员函数。base()成员函数实现将逆迭代器类型的迭代器转化为正常迭代器。

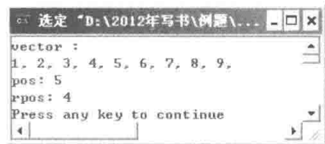


图 5-3　例 5-2 的执行效果

5.4.2　插入型迭代器

Insert 迭代器称为 Inserters，用来将“赋值新值”操作转换为“安插新值”操作。通过迭代器，算法可以执行安插行为而非覆盖行为。所有 Insert 迭代器都隶属于 Output 迭代器类型，它只提供赋值新值的能力。

Insert 迭代器可以把上述这样的赋值动作转化为安插操作。实际上这里面有两个操作：运算符“*”传回到迭代器当前位置，然后由“operator=”赋值新值。

Insert 迭代器通常使用下面两个技巧：

（1）operator* 在实际操作中被认为是一个无实际动作的动作，简单传回*this。对 Insert 迭代器来说，*pos 与 pos 等价；

（2）赋值动作被转化为安插动作。事实上 Insert 迭代器会调用容器的 push_back()、push_front()或 insert()成员函数。

对于一个 Insert 迭代器，插入新值时，可以采用两种形式：pos=value，*pos=value。但正确的表达式应该是*pos=value。

Insert 迭代器分 3 种：后插入型、前插入型和产生型。它们之间的区别在于插入位置。在迭代器初始化时，一定要清楚所属的容器是哪一种。每一种插入型迭代器均由一个对应的便捷函数加以生成和初始化，见表 5-4。

表 5-4　插入型迭代器的分类

名　　称	class	其所调用的函数	生 成 函 数
BackInserter	back_insert_iterator	push_back	back_inserter(cont)
FrontInserter	front_insert_iterator	push_front	front_inserter(cont)
GeneralInserter	insert_iterator	Insert(pos,value)	Inserter(cont,pos)

容器本身必须支持 Insert 迭代器所调用的函数，否则该 Insert 迭代器就不可用。back_inserters 只能用在 vector、deque、list、string 等类型的容器上， front_inserter 只能用在 deque 和 list 容器上。原因很简单，该容器必须具备与该类型迭代器对应的可调用的成员函数。

backInserter 生成时必须根据其所属容器进行初始化。函数 back_inserter()为此提供了方便途径。在前面章节已经大量使用了 back_inserter()函数。back_inserter()在安插元素时，会造成指向该 vector 的其他迭代器失效。string 也提供了 STL 容器接口，包括 push_back()。可以使用 back_inserter()为 string 型容器追加字符串。

frontInserter 通过成员函数 push_front()将一个元素值加在容器头部。而 push_back()仅在 deque 和 list 中有所实现。而 C++标准函数库中就只有这两个容器支持 frontInserter。frontInserter 生成时必须根据所属容器进行初始化。函数 front_inserter()提供了便捷途径。值得一提的是：在插入多个元素时，frontInserter 以逆序方式插入，因为它总是将后一个元素插入前一个元素的前面。

generalInserter 根据两个参数初始化：

（1）容器；

（2）待安插位置。

迭代器内部以“待安插位置”为参数，调用成员函数 insert()。便捷函数 inserter()则能提供更方便的手段产生 generalInserter 并初始化。本类迭代器对所有的标准容器均适用，所有容器都有 insert()成员函数。然而对于关联式容器而言，安插位置仅是标识，元素的真正位置要根据其实值或键值而定。

安插操作完成后，generatorInserter 获得被安插的元素的位置，为确保该迭代器的位置始终有效，在 deque、vector 和 string 中，该 generalInserter 本身会失效。每一次插入操作都会使指向容器的所有迭代器失效。

关联式容器的定制型 Inserter：对于关联式容器，generalInserter 的“位置参数”只不过是个提示，用于强化速度，如果使用不当反而可能导致比较糟糕的性能。如果逆序安插，提示会较缓慢，甚至会导致安插错误。这也是 STL 的缺陷之一。下面用例 5-3 加以说明。

例 5-3

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void main()
{
 int dim[]={1,2,3,4,5,6};
 deque<int> d1;
 cout<<"deque d1(1): "<<endl;
 copy(dim,dim+6,back_inserter(d1));
 copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
 cout<<endl;
 cout<<"deque d1(2): "<<endl;
    front_inserter(d1)=11;
 front_inserter(d1)=22;          //前端插入
 copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
 cout<<endl;
 cout<<"deque d1(3): "<<endl;
 copy(d1.begin(),d1.end(),front_inserter(d1));
 copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
```

```
    cout<<endl;
    inserter(d1,d1.end())=33;
    inserter(d1,d1.end())=44;
    cout<<"deque d1(4): "<<endl;
    copy(d1.begin(),d1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    deque<int> d2;
    copy(d1.begin(),d1.end(),inserter(d2,d2.begin()));
    cout<<"deque d2(5): "<<endl;
    copy(d2.begin(),d2.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 5-3 的执行效果见图 5-4。

图 5-4　例 5-3 的执行效果

5.4.3　流迭代器

流迭代器是一种迭代器配接器，可以把流当成算法的原点和终点。流迭代器是特殊用途的输入和输出迭代器，程序能管理与 I/O 流相关的数据。插入和反向迭代器由迭代器适配器形成。一个 istream 迭代器可以用来从 input stream 中读取元素，而 ostream 迭代器可以用来对 output stream 写入元素。流迭代器的特殊形式是所谓的 stream 缓冲区迭代器，用来对 stream 缓冲区进行直接读取和写入操作。

1．ostream 迭代器

ostream 迭代器可以将被赋予的值写入 output stream 中。ostream 迭代器将赋值操作转化为运算符 operator<<，算法就可以使用一般的迭代器接口直接对 stream 执行动作，见表 5-5。

表 5-5　ostream 迭代器的各项操作

算　　式	效　　果
ostream_iterator<T>(ostream)	为 ostream 产生一个 ostream 迭代器
ostream_iterator <T>(ostream,delim)	为 ostream 产生一个 ostream 迭代器，各元素间以 delim 为分隔符
*iter	无实际操作
Iter=value	将 value 写到 ostream，如：ostream<<value
++iter	无实际操作
Iter++	无实际操作

注：delim 为分隔符。

生成迭代器时，必须提供一个 output stream 作为参数，迭代器将会把元素写到该 output stream 身上。另一个参数可有可无，是个字符串，作为每个元素值之间的分隔符。分隔符的型别是 const char*。如果使用 string 的对象（变量），在使用时一定要使用成员函数 c_str()以获得该对象的内容。例如：

```
string delim;
ostream_iterator<int>cout, delim.c_str());
```

流迭代器的模板为：

```
template < class Type , class CharType = char, class Traits = char_traits<CharType>>
```

其中第一个参数代表被插入流的数据类型；第二个参数默认是 char；第三个参数是可选的，默认是 char_traits<CharType>。

输出流迭代器必须满足输出迭代器的所有要求，在算法中使用输出流迭代器时，可以直接访问输出流。下面用例 5-4 来加以说明。

例 5-4

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
using namespace std;
void main()
{
    int dim[]={1,2,3,4,5,6,7,8,9};
    vector<int> v1;
    ostream_iterator<int> iter(cout,"\n");
    v1.assign(dim,dim+9);
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout));
    cout<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,"< "));
    cout<<endl;
    string delim(", ");
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,delim.c_str()));
    cout<<endl;
    *iter=22;
    iter++;
    *iter=33;
    iter++;
    *iter=44;
}
```

例 5-4 的执行效果见图 5-5。

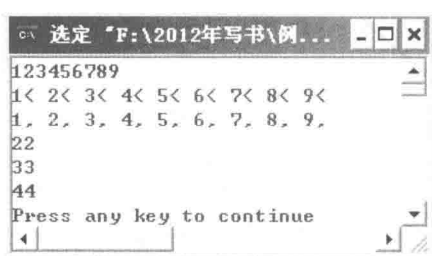


图 5-5　例 5-4 的执行效果

2. istream 迭代器

istream 迭代器是 ostream 迭代器的"伙伴"，用于从 input stream 读取元素。通过 istream 迭代器，算法可以从 stream 中直接读取元素。然而，istream 迭代器较 ostream 迭代器稍微复杂一些。本来"读取"工作就比"写"复杂。

产生 istream 迭代器时，必须提供一个 input stream 作为参数，迭代器将从其中读取数据。一般经由 input 迭代器的通用接口，利用 operator>>读取元素。读取动作可能失败，此外算法需要知道区间是否到达终点。

为了解决这些问题，可以使用 end-of-stream 迭代器，它利用 istream 迭代器的默认构造函数生成。只要读取一次失败，所有的 istream 迭代器会变成 end-of-stream 迭代器。所以，每次读取结束后，应该将 istream 迭代器和 end-of-stream 迭代器进行比较，观察迭代器是否合理合法。

istream 迭代器的构造函数会将 stream 打开，读取第一个元素。在确实需要用到一个 istream 迭代器之前，不要过早定义它，上述做法是必要的，否则一旦 operator* 被调用，将无法传回第一个元素。这和版本有关系，某些版本会延缓第一次读取操作，直到第一次 operator* 被调用，见表 5-6。

表 5-6　istream 迭代器的各项操作

算　　式	效　　果
istream_iterator<T>()	产生一个 end-of-stream 迭代器
istream_iterator <T>(istream)	为 istream 迭代器产生一个迭代器（可能立刻读取一个元素）
*iter	传回先前读取的值（如果构造函数未读取第一个元素值，则本式执行读取任务）
iter->member	传回先前读取的元素成员
++iter	读取下一个元素，并传回其位置
iter++	读取下一个元素，传回迭代器指向的前一个元素
iter1==iter2	检验 iter1 和 iter2 是否相等
iter1!=iter2	检查 iter1 和 iter2 是否相等

istream 迭代器类模板如下：

```
    template<class T, class chart-char, class traits=char_traints<chart>, class
Distance=ptrdiff_t> class istream_iterator
```

类模板的第 1 个参数是数据类型；第 2 个参数和第 3 个参数确定 stream 型别，第 4 个参数用来指定迭代器距离表示的型别。

两个 istream 迭代器相等的条件如下：

（1）两者都是 end-of-stream 迭代器；

（2）两者均可进行读取操作，并指向相同的 stream。

下面用例 5-5 来加以说明。

例 5-5

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
using namespace std;
void main()
```

```
{
    vector <int> v1;
    istream_iterator<int> inputCin(cin);
    istream_iterator<int> IEof;
    while(inputCin!=IEof)
    {
        *inputCin;
        v1.push_back(*inputCin);
        ++inputCin;
    }
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
}
```

例 5-5 的执行效果见图 5-6。

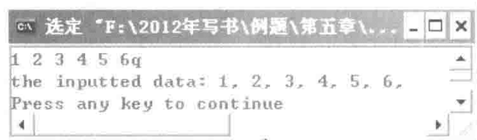


图 5-6　例 5-5 的执行效果

在例 5-5 中，定义的流迭代器输入数据类型为 int，一旦输入非 int 型数据，输入即面临结束，while 循环退出。

5.5　迭代器辅助函数

C++标准程序库（STL）为迭代器提供了 3 个辅助函数：advance()、distance()和 iter_swap()。前二者给所有迭代器提供前进和后退的能力；第三者允许程序开发人员交换两个迭代器的数值。前面章节已经讲述过，只有随机存取迭代器才可以自由地前进和后退，方便地使用偏移量，而函数 advance()使所有类型的迭代器均可以方便地前进和后退；函数 distance()可以计算统一容器中两个迭代器之间的距离；函数 iter_swap()可以方便交换两个迭代器所指向元素的值。

5.5.1　advance()迭代器前进函数

迭代器辅助函数 advance()可使相关的迭代器前进和后退，增加的速度由参数决定，可以使迭代器前进和后退一个或多个元素。函数原型如下：

```
    template<class InputIterator, class Distance> void advance( InputIterator& _InIt,
Distance _Off );
```

函数的功能：实现将输入型迭代器 _Init 前进或后退 _Off 个元素。对于参数_Off，前进为正数，后退为负数。只有当迭代器类型为双向型迭代器和随机存取型迭代器时，参数_Off 为负值，表示后退。原型中 Distance 是模板（template）型别，通常为整数型别。函数 advance()在移动迭代器指针时若不检查是否超过序列的 end()，可能会导致迭代器已到达尾部（首部），还要继续增加迭代器的数值。

函数 advance()也有其局限性。比如：

（1）只有随机存取迭代器才能方便地移动迭代器的位置，非随机存取迭代器需要使用 advance

函数移动迭代器指向的位置，其执行性能并不佳；

（2）advance()没有返回值。

但这并不影响 advance()的优点，正是 advance()使迭代器的移动变得容易了！

下面用例 5-6 来加以说明。

例 5-6

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void main()
{
 int dim[ ]={1,2,3,4,5,6,7,8,9};
 vector<int> v1;
 vector<int>::iterator it;
 v1.assign(dim,dim+9);
 copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));     //复制
 cout<<endl;
 it=v1.begin();
 advance(it,5);
 cout<<"The sixth element: "<<*it<<endl;
 advance(it,-3);
 cout<<"The third element: "<<*it<<endl;
}
```

例 5-6 的执行效果见图 5-7。

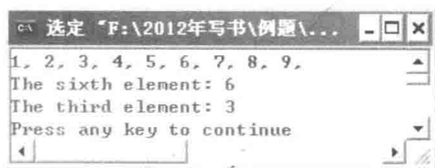


图 5-7　例 5-6 的执行效果

5.5.2　distance()迭代器距离

函数 distance()用来处理两个迭代器之间的距离。使用该函数时，需要包含头文件<iterator>。函数原型为：

```
ptrdiff_t distance(InIt first, InIt last);
```

函数返回值 ptrdiff_t 是整数类型 integer，first 和 last 分别代表两个迭代器。当然，两个迭代器必须是同一容器的迭代器。如果不是随机存取型迭代器，从 first 开始往前走必须能到达 last，即 last 必须和 first 相同或在其后。

此函数能够根据两个迭代器传回它们之间的距离。对于随机存取型迭代器，距离是常数。对于其他型迭代器，距离具有线性复杂度。对于非随机存取迭代器，使用 distance()时，性能和效果并不是很好，应尽力避免使用。

下面用例 5-7 来加以说明。

例 5-7

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

```
#include <algorithm>
using namespace std;
void main()
{
 int dim[]={1,2,3,4,5,6,7,8,9};
 list<int>l1;
 copy(dim,dim+9,back_inserter(l1));
 copy(l1.begin(),l1.end(),ostream_iterator<int>(cout,", "));
 cout<<endl;
 list<int>::iterator pos;
 pos=find(l1.begin(),l1.end(),5);
 if(pos!=l1.end())
 {
     cout<<"The distance between beginning and 5: "<<distance(l1.begin(),pos)<<endl;
//使用 distance
 }
 else
 {
     cout<<"5 not found."<<endl;
 }
}
```

例 5-7 的执行效果见图 5-8。

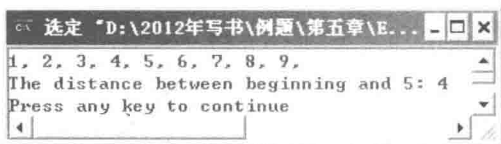


图 5-8　例 5-7 的执行效果

5.5.3　iter_swap()交换两个迭代器所指内容

函数 iter_swap()用来交换两个迭代器的元素值。函数的原型为：

```
void iter_swap( ForwardIterator1 First, ForwardIterator2 Second )
```

交换迭代器 First 和 Second 的所指向的值。迭代器的型别不一定相同，但所指向的两个数值必须可以互相赋值。

下面用例 5-8 来加以说明。

例 5-8

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;
void main()
{
 int dim[]={1,2,3,4,5,6,7,8,9};
 double Vdim[]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
 vector<int> v1;
 list<double> l1;
 v1.assign(dim,dim+9);
 copy(Vdim,Vdim+9,back_inserter(l1));
```

```
    cout<<"vector v1: "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"list l1: "<<endl;
    copy(l1.begin(),l1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    vector<int>::iterator itV;
    itV=v1.end();
    itV--;
    iter_swap(v1.begin(),itV);
    cout<<"vector v1(swap first.): "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    list<double>::iterator itLs,itLe;
    itLs=l1.begin();
    advance(itLs,2);
    itLe=l1.end();
    advance(itLe,-3);
    iter_swap(itLs,itLe);
    cout<<"list l1 (swap first.): "<<endl;
    copy(l1.begin(),l1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
    itV=v1.begin();
    iter_swap(itLs,itV);
    cout<<"vector v1 (swap second.): "<<endl;
    copy(v1.begin(),v1.end(),ostream_iterator<int>(cout,", "));
    cout<<endl;
    cout<<"list l1 (swap second.): "<<endl;
    copy(l1.begin(),l1.end(),ostream_iterator<double>(cout,", "));
    cout<<endl;
}
```

例 5-8 的执行效果见图 5-9。

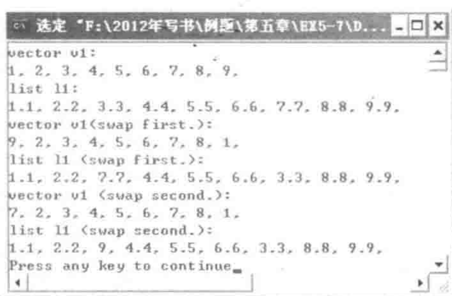


图 5-9　例 5-8 的执行效果

5.6　本章小结

本章重点讲述了 5 部分内容：第 1 部分概要地讲述迭代器的概念及其特性；第 2 部分讲述使用迭代器及相关函数需要包含的头文件；第 3 部分是本章的重点，详细讲述了迭代器的类型（输入型、输出、前向、双向、随机存取）以及迭代器如何实现递增和递减。第 4 部分同样是本章的重点，讲述迭代器的配接器，主要从 3 个方面讲述了迭代器：逆向迭代器、插入型迭代器和流迭代器。第 5 部分讲述迭代器的辅助函数，主要包括：advance()、distance()和 iter_swap()。

第 6 章　STL 的数值计算

本章讲述 C++ STL 标准程序库的数值相关组件，其中包括复数（complex）、数值数组（value arrays），以及从 C 标准程序库继承而来的全局数值函数。C++在数值方法上的优越性使 C++被广泛应用于涉及复杂数值的科学与工程计算，逐步代替传统的 FORTRAN 语言。STL 支持数值计算，展现了其迷人的侧面。本章围绕 STL 中的内容，主要讲述 4 个方面的内容。

- 复数运算
- 数组（向量）运算
- 通用数值运算
- 全局性数学函数

6.1　复数运算

C++标准库中提供了一个 template class complex<>，用于实现复数操作。复数是由实部和虚部组成的数值。虚部的特点是“其平方值为负数”。即复数虚部带着 i，i 是-1 的平方根。

类 complex 定义于头文件<complex>中。使用复数运算相关的函数时，需要包含头文件<complex>。

```
#include <complex>
```

类 complex 的定义：

```
template<class Type>  class complex
```

其中 template 的参数 Type 被用来作为复数的实部和虚部的标量型别。STL 还为标准程序库提供了针对标量型别 float、double、long double 的特殊版本。

6.1.1　最简单的复数运算例题

本节介绍一个最简单的复数运算例题。具体内容见例 6-1。

例 6-1

```
#include <iostream>
#include <complex>
using namespace std;
void main()
```

```
{complex<double> c1(3.0,4.0); //real, image
 complex<float> c2(polar(5.0,1.0));//magnitude, phase
 cout<<"c1: "<<c1<<endl;
 cout<<"c2: "<<c2<<endl;
 cout<<"c1: magnitude: "<<abs(c1)<<"(squared magnitude: "<<norm(c1)<<")"<<"phase angle: "
<<arg(c1)<<endl;
 cout<<"c2: magnitude: "<<abs(c2)<<"(squared magnitude: "<<norm(c2)<<")"<<"phase angle: "
<<arg(c2)<<endl;
 cout<<"c1 conjugated: "<<conj(c1)<<endl;
 cout<<"c2 conjugated: "<<conj(c2)<<endl;
 cout<<"c1*2.0=: "<<(c1*2.0)<<endl;
 cout<<"c1+c2= : "<<(c1+complex<double>(c2.real(),c2.imag()))<<endl;
}
```

例 6-1 的执行效果见图 6-1。

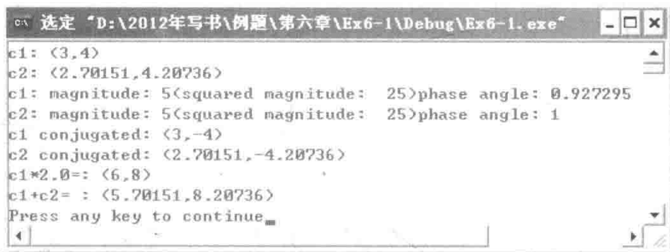


图 6-1　例 6-1 的执行效果

6.1.2　复数成员函数

复数类 complex 的成员函数包括：构造函数，实部函数（real）和虚部函数（imog），运算符函数。本节主要讲述两个构造函数、实部函数和虚部函数。

1. 构造函数

STL 数值库中复数类 complex 提供了两个构造函数，其形式略有不同。函数的声明如下：

```
complex(const T& re = 0, const T& im = 0);
complex(const complex& x);
```

上述两种构造函数形式均较简单，第一种形式利用参数 re 和 im 确定需要构造的复数对象，第二种形式是利用原有的复数对象，构造新的复数对象。代码如下：

```
complex<float> c1(1.5,2.5);          //构造复数对象 c1
complex<float> c2(c1);               //使用复数对象 c1 来构造复数对象 c2
```

2. 实部函数（real）和虚部函数（imag）

复数类 complex 还提供了专门针对复数对象实部和虚部的函数。函数原型如下：

```
T real() const;
T imag() const;
```

在 C++ STL 的复数库中，仅指定了上述形式的函数声明。上述函数可以仅返回（获取）复数对象的实部或虚部。在 Visual C++6.0 中，还可以使用函数 real()和 imag()修改复数对象的实部数值和虚部数值。例如：

```
complex<float> c1(1.5,2.5);
complex<float> c2(c1);
```

```
float re=3.5;
float im=5.2;
c2.real(re);                //修改复数对象 c2 的实部为 3.5
c2.imag(im);                //修改复数对象 c2 的虚部为 5.2
```

6.1.3 复数运算符

复数和实数一样，需要参与各种数学运算。STL 提供了“+”、“-”、“*”、“/”、“=”等运算符。还包括其他较复杂的运算符：“+=”、“-=”、“*=”和“/=”以及部分逻辑运算符，例如：“==”、“！=”。

6.1.4 复数运算

复数运算指在上一节中提到的简单数学运算，还包括部分较复杂的运算。本小节除对上一节涉及的简单数学运算进行举例说明之外，还将讲述绝对值（模）函数、绝对值平方、复数相位、输入与输出、共轭函数、极坐标形式、比较等运算。

1. 算术运算

下面的代码中对上一小节中提到的算术运算均进行了举例说明。复数运算比较复杂，复数可以和复数进行运算，复数也可以和实数进行运算，见例 6-2。

例 6-2

```
#include <iostream>
#include <complex>
using namespace std;
void main()
{complex<float> c1(1.5,2.5);
 complex<float> c2(c1);
 complex<float> c3;
 float re=3.5;
 float im=5.2;
 c2.real(re);               //实部
 c2.imag(im);               //虚部
 cout<<"c1: "<<c1<<";  c2: "<<c2<<endl;
 c3=c1+c2;
 cout<<"c3=c1+c2 : "<<c3<<endl;
 c3=c1-c2;
 cout<<"c3=c1-c2 : "<<c3<<endl;
 c3=c1*c2;
 cout<<"c3=c1*c2 : "<<c3<<endl;
 c3=c1/c2;
 cout<<"c3=c1/c2 : "<<c3<<endl;
 c1+=c2;
 cout<<"c1+=c2 : "<<c1<<endl;
 c1-=c2;
 cout<<"c1-=c2 : "<<c1<<endl;
 c1*=c2;
 cout<<"c1*=c2 : "<<c1<<endl;
 c1/=c2;
 cout<<"c1/=c2 : "<<c1<<endl;
 bool cmp=(c1==c2);
 if(cmp)
    cout<<"c1==c2."<<endl;
 else
```

```
        cout<<"c1!=c2."<<endl;
}
```

例 6-2 中复数的算术运算结果见图 6-2。

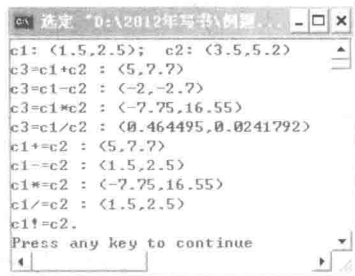


图 6-2　例 6-2 中复数的算术运算结果

2. 其他运算

复数还可以进行其他一些简单的运算，例如：绝对值函数（abs）、绝对值的平方（norm）、复数的相位（arg）、输入与输出、共轭函数（conj）、极坐标函数（polar）等。

（1）绝对值函数（abs）

函数原型如下：

```
T abs(const complex<T>& x);
```

复数的绝对值即复数的模，绝对值的计算公式为：

$$\sqrt{cmp.real()^2 + cmp.imag()^2}$$

（2）绝对值的平方（norm）

函数原型如下：

```
T norm(const complex<T>& x);
```

复数的绝对值平方的求取需要使用函数 norm()。复数绝对值平方的计算公式如下：

$$cmplx.real()^2 + cmplx.imag()^2$$

（3）复数的相位（arg）

函数原型如下：

```
T arg(const complex<T>& x);
```

函数 arg()返回复数的相位，返回值的单位是弧度，是复数的极坐标形式的相位角。相位角的计算公式为：

$$a\tan 2(cmplx.imag() + cmplx.real())$$

（4）输入与输出

复数的输入与输出，可以使用>>和<<。

（5）共轭函数（conj）

函数原型如下：

```
complex<T> conjg(const complex<T>& x);
```

共轭函数可以归入超越函数的范围。在此进行介绍。

复数的共轭函数 conj()产生并返回一个复数。共轭复数是指虚部与原复数的虚部互为反相。

（6）极坐标函数（polar）

函数 polar()的返回值是复数类型，函数的原型如下：

```
complex<T> polar(const T& rho, const T& theta = 0);
```

此函数可用于使用极坐标和相位的形式创建一个复数对象。例如在 6.1.1 节中的代码：

```
complex<float> c2(polar(5.0,1.0));//magnitude, phase
```

下面对上述几个函数进行举例，说明各自的使用方法，见例 6-3。

例 6-3

```
float a1=abs(c1);
float a2=abs(c2);
cout.precision(3);
cout<<"abs(c1): "<<a1<<"  ;"<<"  abs(c2): "<<a2<<endl;
a1=norm(c1);
a2=norm(c2);
cout.precision(3);
 cout<<"norm(c1): "<<a1<<"  ;"<<"  norm(c2): "<<a2<<endl;
a1=arg(c1);
a2=arg(c2);
cout.precision(3);
cout<<"arg(c1): "<<a1<<"("<<(a1*180/3.1415926)<<")"<<"  ;"<<"  arg(c2): "<<a2<<
      "("<<(a2*180/3.1415926)<<")"<<endl;
complex<float> c11;
complex<float> c12;
c11=conj(c1);
c12=conj(c2);
cout<<"conj(c1) : "<<c11<<endl;
cout<<"conj(c2) : "<<c12<<endl;
```

上述代码的执行结果如下：

```
abs(c1): 2.92  ;  abs(c2): 6.27
norm(c1): 8.5  ;  norm(c2): 39.3
arg(c1): 1.03(59)  ;  arg(c2): 0.978(56.1)          //括号内数字为角度
conj(c1) : (1.5,-2.5)
conj(c2) : (3.5,-5.2)
```

总结：本小节主要讲述复数运算。涉及复数类对象（复数）的各种操作、各种运算以及超越函数的使用。

6.1.5 复数的超越函数

复数的超越函数一般包括三角函数和指数函数。三角函数主要是指 sin、cos、tan、sinh、cosh 和 tanh 等。指数函数主要是指幂函数，以 e 为底的幂函数，平方根函数，自然对数函数，以 10 为底的对数。

本小节将依次介绍上述各种超越函数。

1. 三角函数

C 语言提供的三角函数只有 6 种：sin，cos， tan， sinh，cosh，tanh。三角函数的函数原型分别为：

```
complex<T> sin(const complex<T>& x);
```

```
complex<T> cos(const complex<T>& x);
complex<T> tan(const complex<T>& x);
complex<T> sinh(const complex<T>& x);
complex<T> cosh(const complex<T>& x);
complex<T> tanh(const complex<T>& x);
```

复数的三角函数计算是依托自然指数实现的。假定复数 $z=x+j\times y$，则 sin(z)的计算式如下：

$$\sin(z)=\sin(x+j\times y)=\sin x(\frac{e^{y}+e^{-y}}{2})+j\cos x(\frac{e^{y}-e^{-y}}{2})$$

$$\cos(z)=\cos(x+j\times y)=\cos x(\frac{e^{y}+e^{-y}}{2})-j\sin x(\frac{e^{y}-e^{-y}}{2})$$

其余的三角函数均可由正弦函数、余弦函数和自然指数获得。

提示：Visual C++6.0 没有提供复数的正切函数。

例 6-4 是对 6 种三角函数的实例说明。

例 6-4

```
c11=sin(c1);
cout<<"sin(c1) = "<<c11<<endl;
c11=cos(c1);
cout<<"cos(c1) = "<<c11<<endl;
c11=sin(c1)/cos(c1);
cout<<"tan(c1) = "<<c11<<endl;
c11=sinh(c1);
cout<<"sinh(c1) = "<<c11<<endl;
c11=cosh(c1);
cout<<"cosh(c1) = "<<c11<<endl;
c11=sinh(c1)/cosh(c1);
cout<<"tanh(c1) = "<<c11<<endl;
```

程序的执行结果如下：

```
sin(c1) = (6.12,0.428)
cos(c1) = (0.434,-6.04)
tan(c1) = (0.00193,1.01)
sinh(c1) = (-1.71,1.41)
cosh(c1) = (-1.88,1.27)
tanh(c1) = (0.968,-0.0926)
```

2. 其他超越函数

其他超越函数主要是指：幂函数，以 e 为底的幂函数，平方根函数，自然对数函数，以 10 为底的对数。

（1）幂函数

复数可以进行指数运算，即幂运算。STL 为复数提供了幂函数。函数原型如下：

```
template<class T>
   complex<T> pow(const complex<T>& x, int y);
   complex<T> pow(const complex<T>& x, const T& y);
   complex<T> pow(const complex<T>& x, const complex<T>& y);
   complex<T> pow(const T& x, const complex<T>& y);
```

上述几种形式中，参数 x 为底，参数 y 为指数。例如：

```
c3=pow(c1,c2);
cout<<"pow(c1,c2)= "<<c3<<endl;
c3=pow(c1,2);
cout<<"pow(c1,2)= "<<c3<<endl;
//  c3=pow(3.0,c1);                                //在 Visual C++6.0 环境中编译没有通过
//  cout<<"pow(3.0,c1)= "<<c3<<endl;               //在 Visual C++6.0 环境中编译没有通过
```

上述代码的执行结果如下：

```
pow(c1,c2)= (-0.193,0.0501)
pow(c1,2)= (-4,7.5)
```

（2）以 e 为底的幂函数 exp()

以 e 为底的幂函数（exp()）的函数原型如下：

```
template<class T>  complex<T> exp(const complex<T>& x);
```

函数的输入参数和返回值均为复数。例如：

```
c3=exp(c2);
cout<<"exp(c2)= "<<c3<<endl;
```

上述代码的执行结果如下：

```
exp(c2)= (15.5, -29.3)
```

（3）平方根函数 sqrt()

复数的平方根函数原型如下：

```
template<class T> complex<T> sqrt(const complex<T>& x);
```

函数的输入参数和输出参数均为复数类型。例如：

```
c3=sqrt(c1);
cout<<"sqrt(c1)= "<<c3<<endl;
c3=sqrt(complex<float>(-5,0));
cout<<"sqrt(-5)= "<<c3<<endl;
```

上述代码的执行结果如下：

```
sqrt(c1)= (1.49,0.841)
sqrt(-5)= (0,2.24)
```

（4）对数函数

对数函数可分为以 10 为底的对数函数和自然对数函数。函数原型如下：

```
template<class T>  complex<T> log(const complex<T>& x);
template<class T> complex<T> log10(const complex<T>& x);
```

简单举例如下：

```
c3=log(c1);
cout<<"log(c1)= "<<c3<<endl;
c3=log10(c1);
cout<<"log10(c1)= "<<c3<<endl;
```

上述代码的执行结果如下：

```
log(c1)= (1.07,1.03)
```

```
log10(c1)= (0.465,0.447)
```

总结：本节讲述了复数的超越函数。超越函数是最常见、最常使用的数学函数，这些函数的使用方法较简单，读者了解即可。复数类最主要的成员函数包括：real()，image()，abs()，norm，arg()，"<<"，">>"，conj()，polar()等。读者应熟练掌握，尤其是参与计算工作较多的程序开发人员，更是应该认真阅读本章的内容，从而体验 C/C++博大的函数库、类库和模板库。

6.2 数组（向量）运算

STL 提供了一个数组类，class valarray。类 valarray 用于实现数值数组的运算。valarray 代表一个数学概念：数值线性序列。该序列是一维的，但程序员可以通过运用特殊技巧得到多维效果。所谓的特殊技巧，即使用"索引"能力和强大的"子集"能力。因此，valarray 是向量和矩阵运算的基础。数值计算工作多依赖于浮点数值的一维向量，下标从零开始。类 valarray 的设计目的就是加速常用数值向量的计算。

类 valarray 的设计原则不是通用性和易用性，而是高效利用计算机性能，尤其是高性能计算机。如果程序员编程的目标是灵活性和通用性，则没必要使用类 valarray，使用前面讲述的容器即可。

6.2.1 类 valarray

通俗地讲，valarray 是经过优化的向量（vector）。类 valarray 是由 4 个用于描述 valarray 中各个部分的辅助类型支持的。这 4 个类分别是：

slice_array，gslice_array，mask_array 和 indirect_array。

其中 slice_array 和 gslice_array，mask_array 和 indirect_array 用于存放临时数值或数据，这会在后面章节介绍。

使用类 valarray 时，必须包含头文件<valarray>。本节主要讲述类 valarray 的构造函数、下标、赋值以及数学运算各项操作（成员函数）、超越函数、子集操作。子集操作在后面的章节还会介绍。

1. 类 valarray 的构造函数

类 valarray 的构造函数一般有以下几种形式：

```
valarray();
explicit valarray(size_t n);
valarray(const T& val, size_t n));
valarray(const T *p, size_t n);
valarray(const slice_array<T>& sa);
valarray(const gslice_array<T>& ga);
valarray(const mask_array<T>& ma);
valarray(const indirect_array<T>& ia);
```

构造类 valarray 的对象时，最简单的形式是将元素的数量作为输入参数。即上述构造函数的第二种形式和第三种形式。第二种形式的参数 size_t n 用于指定数组的大小，第三种形式的参数 n 用于指定数组的大小，参数 val 用作所有元素的初始默认值。这和 STL 中的容器的构造函数不同。多数容器的构造函数中，第一个参数用于指定容器的大小，第二个参数用于指定容器中元素的初始默认值。

构造函数的第三种形式可以使用已知数组对 valarray 对象进行初始化。下面用例 6-5 来加以说明。

例 6-5

```
int dim[]={1,2,3,4,6,7,8,9};
valarray<int> val1(10);
valarray<int> val2(0,10);
valarray<int> val3(dim,sizeof(dim)/sizeof(int));
valarray<int> val4(val2);
cout<<"valarray val1: "<<endl;
print(val1);
cout<<"valarray val2: "<<endl;
print(val2);
cout<<"valarray val3: "<<endl;
print(val3);
cout<<"valarray val4: "<<endl;
print(val4);
```

上述代码的执行结果如下：

```
valarray val1:
-842150451, -842150451, -842150451, -842150451, -842150451, -842150451, -8421504
51, -842150451, -842150451, -842150451,
valarray val2:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
valarray val3:
1, 2, 3, 4, 6, 7, 8, 9,
valarray val4:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

2. 下标、赋值以及数学运算

类 valarray 还提供了一系列的下标操作[]、数学运算等。下标操作符[]使编程人员可以方便地访问数组中的每个元素，并获取数值子集。类 valarray 还定义了所有的普通数学运算：+、-、*、模数、反相、位操作、位比较、逻辑操作、赋值操作等。数组的乘法运算是对应元素相乘，即：

```
val=va2*va3
```

等同于：

```
val[0]=va2[0]*va3[0];
val[1]=va2[1]*va3[1];
… …
```

如果参与计算的数组容量不一致，则运算结果未有定义。

3. 成员函数

类 valarray 提供了一系列的成员函数：size()，sum()，max()，min(), resize()，shift()，cshift()，apply()，free() 。

（1）函数 size()：用于获取数组的大小。

（2）函数 max()：用于获取数组中的最大值。

（3）函数 min()：用于获取数组中的最小值。

（4）函数 sum()：用于获取数组中所有元素的总和。

（5）函数 resize()：用于实现重新设置数组的大小。

（6）函数 shift()和 cshift()：函数 shift()用于实现逻辑移位，函数 cshift()用于实现循环移位。移位的意思是将数组中的元素位置顺次移动。

（7）函数 apply()的原型如下：

```
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
```

其中 fn(T)是指定的函数。函数 apply()的功能是：使用函数 fn()处理数组中的每个元素。

（8）函数 free()的原型如下：

```
void free();
```

函数 free()删除数组的所有元素，并将数组长度设置为 0。

完整的例 6-6 源代码如下：

例 6-6

```
#include <iostream>
#include <valarray>
//#include <>
using namespace std;
void print(valarray<int>& array )
{  int size=array.size();
   for(int i=0;i<size;i++)
      cout<<array[i]<<", ";
   cout<<endl;
}
int fn(int ele)
{  int r=0;
   r=ele*2;
   return r;
}
void main()
{  int dim[]={1,2,-3,4,5,6,-7,8,9};
   valarray<int> val1(10);
   valarray<int> val2(-1,10);
   valarray<int> val3(dim,sizeof(dim)/sizeof(int));
   valarray<int> val4(val2);
   valarray<int> val5;
   cout<<"valarray val1: "<<endl;
   print(val1);
   cout<<"valarray val2: "<<endl;
   print(val2);
   cout<<"valarray val3: "<<endl;
   print(val3);
   cout<<"valarray val4: "<<endl;
   print(val4);
   val5=abs(val3);
   cout<<"abs(val3)= "<<endl;
   print(val5);
   int s=val3.size();
```

```
    int ma=val3.max();
    int mi=val3.min();
    cout<<"The size of val3 is : "<<s<<endl;
    cout<<"The maximum value in val3 is : "<<ma<<endl;
    cout<<"The minimum value in val3 is : "<<mi<<endl;
    //
    val3.resize(15);
    cout<<"val3 is resized, val3: "<<endl;
    print(val3);
    int He=val3.sum();
    cout<<"The sum of val3 is : "<<He<<endl;
    val5=val3.shift(5);
    cout<<"val3 move 5 bits left. "<<endl;
    print(val5);
    val5=val3.shift(-5);
    cout<<"val3 move 5 bits right. "<<endl;
    print(val5);
    val5=val3.cshift(8);
    cout<<"val3 move 8 bits left circulatedly. "<<endl;
    print(val5);
    val5=val3.cshift(-8);
    cout<<"val3 move 8 bits right. circulatedly"<<endl;
    print(val5);
    val5=val3.apply(&fn);
    cout<<"val3 which has been delt by fn is below : "<<endl;
    print(val5);
    val1.free();
    val2.free();
    val3.free();
    val4.free();
    val5.free();
}
```

例 6-6 的执行效果见图 6-3。

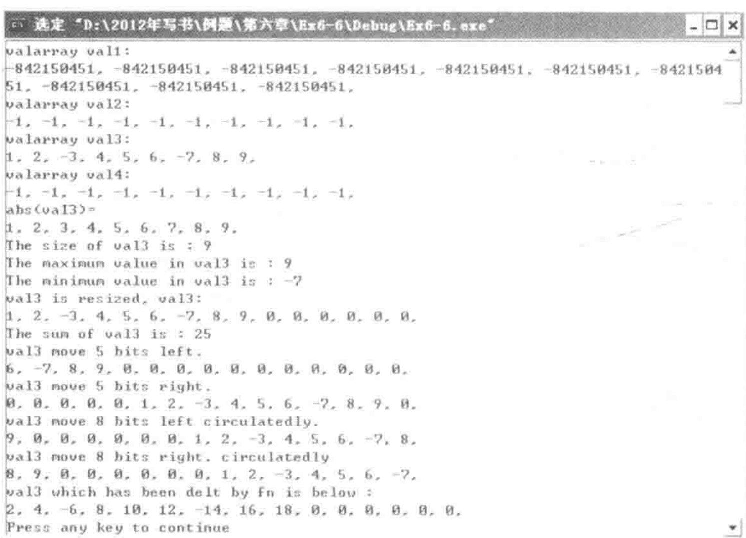


图 6-3 例 6-6 的执行效果

提示：请读者认真读例题。例题是经过作者认真思考和精心处理的，精心安排了各种编程细节。

4．超越函数

对于类 valarray，可允许的超越函数包括：abs()，pow()，exp()，sqrt()，log()，log10()，sin()，cos()，tan()，sinh()，cosh()，tanh()，asin()，acos()，atan()，atan2()。以上函数均返回一个新的 valarray 型的新数组，元素个数和该函数输入的数组大小相同。上述函数的功能是针对数组中的每个元素分别进行超越函数计算。详见例 6-7。

（1）函数 abs()

函数原型如下：

```
template<class T> valarray<T> abs(const valarray<T>& x);
```

函数 abs()的功能是对数组中的每个元素进行绝对值计算，获取每个元素的绝对值。

（2）函数 pow()

函数原型如下：

```
template<class T> valarray<T> pow(const valarray<T>& x, const valarray<T>& y);
template<class T> valarray<T> pow(const valarray<T> x, const T& y);
template<class T> valarray<T> pow(const T& x, const valarray<T>& y);
```

使用函数 pow()时，需要包含数学库头文件<cmath>。对于类 valarray 的对象（数组），函数 pow()主要有上述 3 种形式。在 Visual C++6.0 环境中，本书作者仅编译通过了函数 pow()的第一种形式。

（3）函数 exp()

函数原型如下：

```
template<class T> valarray<T> exp(const valarray<T>& x);
```

使用函数 exp()时，同样需要包含头文件<cmath>。

（4）函数 sqrt()

函数原型如下：

```
template<class T>  valarray<T> sqrt(const valarray<T>& x);
```

使用函数 sqrt 时，同样需要包含头文件<cmath>。

（5）函数 log()和 log10()

函数原型如下：

```
template<class T> valarray<T> log(const valarray<T>& x);
template<class T> valarray<T> log10(const valarray<T>& x);
```

使用函数 log()和 log10()时，同样需要包含头文件<cmath>。

（6）三角函数

函数原型如下：

```
template<class T> valarray<T> sin(const valarray<T>& x);
template<class T>inline valarray<T> cos(const valarray<T>& x);
template<class T>inline valarray<T> tan(const valarray<T>& x);
template<class T>inline valarray<T> sinh(const valarray<T>& x);
template<class T>inline valarray<T> cosh(const valarray<T>& x);
template<class T>inline valarray<T> tanh(const valarray<T>& x);
template<class T> inline valarray<T> asin(const valarray<T>& x);
```

```
template<class T> inline valarray<T> acos(const valarray<T>& x);
template<class T> inline valarray<T> atan(const valarray<T>& x);
template<class T> inline valarray<T> atan2(const valarray<T>& x, const valarray<T>& y);
template<class T> inline valarray<T> atan2(const valarray<T> x, const T& y);
template<class T> inline valarray<T> atan2(const T& x, const valarray<T>& y);
```

使用三角函数时，同样需要包含头文件<cmath>。

上述三角函数中，函数 atan2()有 3 种形式。本文作者在 Visual C++6.0 环境中，仅调试通过了第一种形式。

上述超越函数的使用方法详见例 6-7。

例 6-7

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
void print(valarray<double>& array )
{int size=array.size();
 cout.precision(3);
 for(int i=0;i<size;i++)
     cout<<array[i]<<", ";
 cout<<endl;
}
void main()
{double dim[]={1,2,-3,4,5,6,-7,8,-9};
 valarray<double> v1(dim,sizeof(dim)/sizeof(double));
 valarray<double> v2;
 valarray<double> v3(2,9);
 cout<<"v1: "<<endl;
 print(v1);
 v2=abs(v1);                    //绝对值
 cout<<"v2=abs(v1)= :"<<endl;
 print(v2);
 v2=pow(v1,v3);                 //幂
 cout<<"v2=pow(v1,v3)= :"<<endl;
 print(v2);
 v2=exp(v1);                    //指数
 cout<<"v2=exp(v1)= :"<<endl;
 print(v2);
 v2=sqrt(v1);                   //开平方。如果元素为负值。输出时仅表明结果为虚数
 cout<<"v2=sqrt(v1)= :"<<endl;
 print(v2);
 v2=log(v1);                    //对数
 cout<<"v2=log(v1)= :"<<endl;
 print(v2);
 v2=log10(v1);                  //以 10 为底的对数
 cout<<"v2=log10(v1)= :"<<endl;
 print(v2);
 v2=sin(v1);                    //正弦
 cout<<"v2=sin(v1)= :"<<endl;
 print(v2);
```

```
    v2=cos(v1);                          //余弦
    cout<<"v2=cos(v1)= :"<<endl;
    print(v2);
    v2=tan(v1);                          //正切
    cout<<"v2=tan(v1)= :"<<endl;
    print(v2);
    v2=sinh(v1);                         //sinh
    cout<<"v2=sinh(v1)= :"<<endl;
    print(v2);
    v2=cosh(v1);                         //cosh
    cout<<"v2=cosh(v1)= :"<<endl;
    print(v2);
    v2=tanh(v1);                         //tanh
    cout<<"v2=tanh(v1)= :"<<endl;
    print(v2);
    v2=asin(v1);                         //反正弦
    cout<<"v2=asin(v1)= :"<<endl;
    print(v2);
    v2=acos(v1);                         //反余弦
    cout<<"v2=acos(v1)= :"<<endl;
    print(v2);
    v2=atan(v1);                         //反正切
    cout<<"v2=atan(v1)= :"<<endl;
    print(v2);
    v2=atan2(v1,v3);                     //反正切
    cout<<"v2=atan2(v1,v3)= :"<<endl;
    print(v2);
}
```

例 6-7 的执行效果见图 6-4。

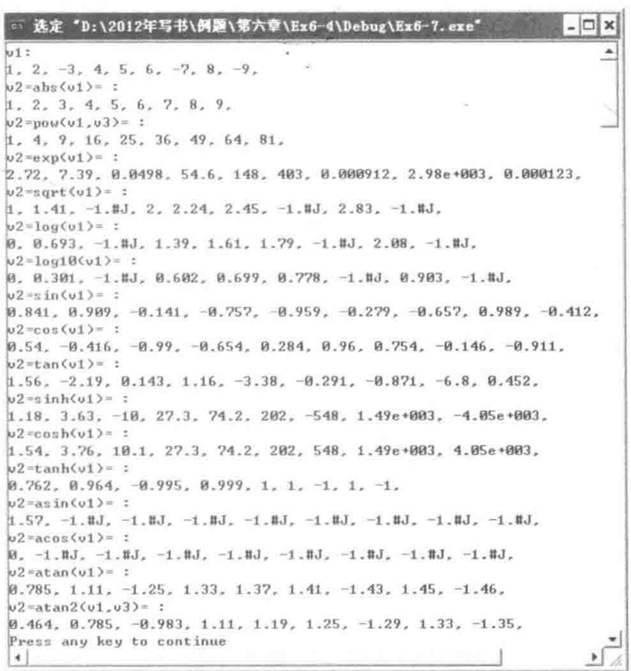


图 6-4　例 6-7 的执行效果

总结：本节讲述了数组类 valarray 的内容。主要涉及的内容包括数组的最基本使用、访问、基本操作以及超越函数。希望读者对照给出的源代码，认真阅读源代码并熟练掌握。

6.2.2 数组子集类——slice 类和类模板 slice_array

slice 是切割的意思，即将一个向量作为任意维数组去操作。“切割”是在一个 valarray 中，中间距为 n 的多个元素。slice 的跨步是切割中两个元素的距离（间隔的下标个数）。每次切割均具备 3 个属性：

（1）起始下标；

（2）元素个数；

（3）元素的间距。

切割可以从一个数组中提出部分元素，成为一个新的数组（valarray）。

从一个 valarray 和一个 slice，可以构造出感觉上类似 valarray，而实际上是一种由切割描述的数组子集。使用 slice 方式可以创建数组的各种各样的子集。STL 没有提供矩阵类，使用 slice 方式可构造出有不同需要而优化的各种矩阵。因此矩阵的表示即是一个 valarray，通过切割（slice）的方式，使 valarray 类型数组拥有了维数。

为方便运算和操作，需要将数组的子集转换成类，slice_array 类应运而生。slice_array 是为切割（slice）提供内部辅助的类别。对用户而言，slice_array 类内部是完全透明的，但类 slice_array 的所有构造函数和赋值操作符均是私有类型（private）。

slice_array 类对以下操作均有定义：

（1）将同一值赋予每一个元素；

（2）赋值另一个 valarray；

（3）调用任何一个赋值复合运算。

如果希望完成其他操作，必须先将子集转换成 valarray 类的对象才可以。

同一个 valarray 的不同 slice 方式，可以组合成诸多子集，其结果可以存入 valarray 类型的另一个子集中。在此基础上，valarray 型数组可作为二维矩阵看待。

对于 valarray，用户不会直接创建一个 valarray 类的对象。实际情况是，根据编程人员描述 valarray 类对象的下标，为给定切割（slice）建立一个 slice_array 类型的数组。值得一提的是，slice_array 是禁止复制的，但 slice 是允许复制的。

下面列举两个例题作为本节学习的范例，见例 6-8 和例 6-9。

例 6-8

```
#include <valarray>
#include <iostream>
#include <cmath>
using namespace std;
void myprint(valarray<double>& v)        //输出数组
{
 int size=v.size();
 for(int i=0;i<size;i++)
     cout<<v[i]<<", ";
 cout<<endl;
}
void main()
```

```
{
 valarray <double> v1(12),v2;
 for(int i=0;i<12;i++)
 {
     v1[i]=2*(i+1);
 }
 cout<<"v1(Original): "<<endl;
 myprint(v1);
 valarray<double> tv1=valarray<double>(v1[slice(0,4,3)]);
 valarray<double> tv2=valarray<double>(v1[slice(2,4,3)]);
 v1[slice(0,4,3)]=pow(tv1,tv2);
 cout<<"v1(Calculated): "<<endl;
 myprint(v1);
 valarray<double> v3(v1[slice(0,4,3)]);
 cout<<"v3: "<<endl;
 myprint(v3);
}
```

例 6-8 的执行效果见图 6-5。

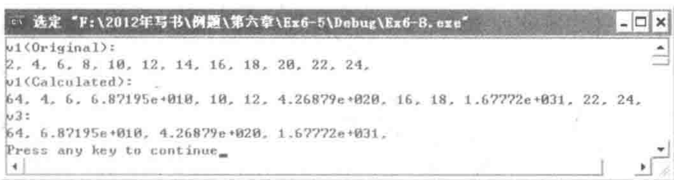


图 6-5　例 6-8 的执行效果

例 6-9

```
#include <iostream>
#include <valarray>
using namespace std;
void myprint(valarray<double>& v)                    //输出数组
{
 int size=v.size();
 for(int i=0;i<size;i++)
      cout<<v[i]<<", ";
 cout<<endl;
}
void main()
{
 valarray<double> v1(12),v2;
 for(int i=0;i<12;i++)
      v1[i]=2*(i+1);
 cout<<"v1(Original) : "<<endl;
 myprint(v1);
 slice myslice(0,4,3);
 v2=v1[myslice];
 cout<<"v2(v1[slice(0,4,3)])  :"<<endl;
 myprint(v2);
}
```

例 6-9 的执行效果见图 6-6。

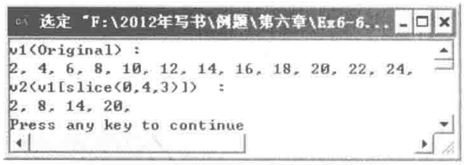


图 6-6　例 6-9 的执行效果

总结：读者要认真阅读上述两个例题，体会 slice 的用法。上述例题是作者精心编排的，去掉了模糊不清的内容，尽量让读者一看就懂。

6.2.3　类 gslice 和类模板 gslice_array

gslice 是 general slice 的简称。gslice 具有以下 3 个属性：起始索引，元素数量，元素间距。gslice 的元素数量和间距也是数组，其中的元素个数和其维度相同。gslice 既可以处理一维数组、二维数组，还可以处理三维数组。

gslice 是一个广义切割，会包含 n 个切割的所有信息。同样，gslice_array 提供了与 slice_array 同样的一组成员。gslice_array 不能直接由用户构造或复制。客观地讲，gslice_array 即使用 gslice 作为一个 valarray 的下标的结果。

由以上可知，gslice 和 slice 的区别在于 gslice 能够运用数组来定义大小和间距，除此之外两者相同。

（1）当需要定义 valarray 数组的具体子集时，可以将 gslice 作为参数传递给 valarray 的下标操作符；

（2）若 valarray 数组是常量，则子集表达式将导致一个新的 valarray 数组；

（3）若 valarray 数组不是常量，则子集表达式将导致一个 gslice_array；

（4）gslice_array 类提供赋值操作符和复合赋值操作符，用以修改子集内的元素；

（5）通过型别转换，可以将 gslice_array 和其他 valarray 以及 valarray 子集组合起来。

下面讲述如何使用 gslice 实现处理一维数组和多维数组。

（1）对于一维 valarray 类型数组，如何使用 gslice 产生一维子集呢？

```
valarray v1[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
```

如果 gslice 为 gslice(2,4,3)，则返回的子集为{2,5,8,11}。

（2）对于一维 valarray 类型数组，如何使用 gslice 产生二维子集呢？

```
valarray<int> v1[15]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
int length[]={2,4 };
int stride[]={5,3};
```

若 gslice 为 gslice(1,length, stride)，则返回的子集，如例 6-10 所示：

```
{1, 4,7,10
6, 9,12,15 }
```

（3）对于一维 valarray 类型数组，如何使用 gslice 产生三维子集呢？

```
valarray<int> v1[15]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19, 20};
int length[]={3,2,4};
```

```
int stride[]={30,10,3};
```

若 gslice 为 gslice(1,length,stride)，则返回的子集如例 6-10 所示。

例 6-10

```
#include <iostream>
#include <valarray>
using namespace std;
void main()
{
 valarray<size_t> v1(20),v2,v3,v11;
 for(size_t i=0;i<20;i++)
 {
     v1[i]=i;
     cout<<i<<", ";
 }
 cout<<endl;
 valarray<size_t> len(5,1);
 valarray<size_t> stri(3,1);
 gslice g1(1,len,stri);
 v11=v1[g1];
 cout<<"subcollection of one dimension: "<<endl;
 int size=v11.size();
 for(size_t j=0;j<size;j++)
 {
     v1[j]=j;
     cout<<j<<", ";
 }
 cout<<endl;
 //以上是一维子集
 size_t lengthv[]={2,4};
 size_t stridev[]={5,3};
 valarray<size_t>length(lengthv,2);       //构造赋值
 valarray<size_t>stride(stridev,2);       //构造赋值
 gslice g(1,length,stride);
 v2=v1[g];
 cout<<"subcollection of two dimension: "<<endl;
    size=v2.size();
 for(j=0;j<size;j++)
 {
     v2[j]=j;
     cout<<j<<", ";
 }
 cout<<endl;
 //以上是二维子集
 size_t lengthv2[]={2,2,3};
 size_t stridev2[]={3,2,3};
 valarray<size_t>length2(lengthv2,3);     //构造赋值
 valarray<size_t>stride2(stridev2,3);     //构造赋值
 gslice g2(1,length2,stride2);
 v3=v1[g2];
 cout<<"subcollection of three dimension: "<<endl;
```

```
    size=v3.size();
    for(j=0;j<size;j++)
    {
        v3[j]=j;
        cout<<j<<", ";
    }
    cout<<endl;
    //以上是三维子集
}
```

例 6-10 的执行效果见图 6-7。

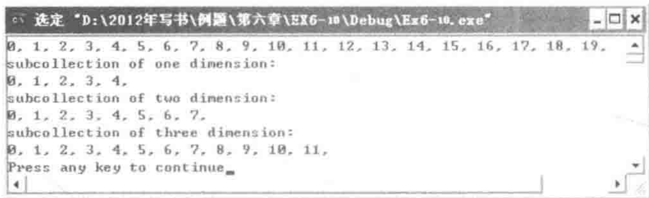


图 6-7　例 6-10 的执行效果

提示： 读者要认真阅读上述例题。

6.2.4　类 mask_array

数组类 valarray 还提供了另一种描述数组子集的方式：屏蔽子集。valarray 类的对象经过“屏蔽”处理后，其返回结果为 valarray<bool>型。将“屏蔽”作为数组（valarray）的下标，值为 true 的位表明对应的 valarray 型数组中的元素将作为结果的一部分。在此基础上，也同样可以在 valarray 的特定子集上操作。和 slice_array 一样，mask_array 也不能由程序开发者构造或复制。mask_array 是将一个 valarray<bool>型对象用作 valarray 数组下标的结果。屏蔽 valarray 的元素个数不能多于以它作为下标的那个 valarray 的元素个数。

由上述说明可知，mask_array 就是使用布尔表达式屏蔽相应元素的结果。具体地说，mask_array 和所有的 valarray subsets 一样，具有以下属性：

（1）若要定义某 valarray 型数组的具体子集，可以将布尔型的 valarray 作为参数传给 valarray 的下标操作符；

（2）如果 valarray 是常量，子集表达式将导致新的 valarray；

（3）若 valarray 型数组不是常量，子集表达式将导致一个 mask_array，后者以 reference 语意来表现 valarray 的相应元素。

（4）mask_array 提供赋值操作符和复合赋值操作符，用来修改子集中的元素；

（5）通过型别转换，将 mask_array 与其他 valarray 和 valarray 的子集组合起来。

mask_array 的使用方法详见例 6-11。

例 6-11

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
void myprint(valarray<double>& v)
```

```
{     int size=v.size();
     for(int i=0;i<size;i++)
          cout<<v[i]<<", ";
     cout<<endl;
}
void main()
{    double dim[12]={1,2,3,4,5,6,7,8,9,10,11,12};
     valarray<double> v1(dim,sizeof(dim)/sizeof(double)),v2,v3;
     cout<<"valarray(Original) v1:  "<<endl;
     myprint(v1);
     bool B[]={0,1,0,1,0,0,1,1,1};
     valarray<bool> mask_array(B,9);
     v2=v1[mask_array];
     cout<<"mask_array v2:"<<endl;
     myprint(v2);
     v1[v1>8.0]=33.0;
     cout<<"valarray v1: "<<endl;
     myprint(v1);
     v1[v1<8.0]=valarray<double>(v1[v1<8.0])+50.5;
     myprint(v1);
     bool B2[12];
     for(int i=0;i<12;i++)
     {    B2[i]=(bool)fmod(v1[i],2);
     }
     for(i=0;i<12;i++)
     {  cout<<B2[i];
     }
     cout<<endl;
     valarray<bool> vB2(B2,12);
     v1[!vB2]=0;
     myprint(v1);
}
```

例 6-11 的执行效果见图 6-8。

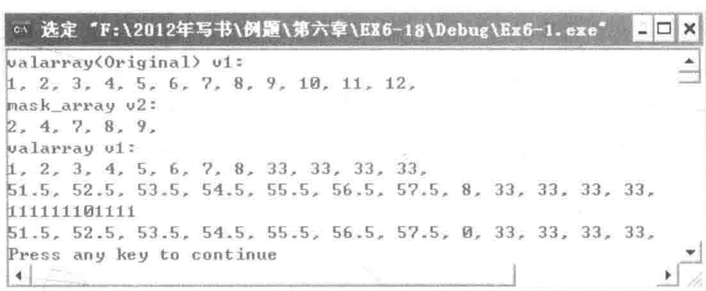


图 6-8　例 6-11 的执行效果

总结：读者要认真阅读上述例题。不要辜负笔者精心编排例题的一片苦心。

6.2.5　类 indirect_array

间接数组子集也是创建数组子集的一种方式，还可以任意排列元素。这是定义数组子集的第 4 种方法。同样，通过传递一个索引数组，即可定义一个 valarray 的子集。间接数组子集可以使用类

indirect_array 来表示。但是类 indirect_array 不能直接由用户构造或复制，和 mask_array 的用法一样，indirect_array 是将 valarray<size_t>用作 valarray 的下标而产生的子集。作为下标的 valarray 的元素个数必须小于原始 valarray 数组。

另外，indirect_array 和所有的 valarray 的子集一样，具有以下属性：

（1）若要定义某个 valarray 的子集，可以使用元素型别为 size_t 的 valarray 作为参数传递给 valarray 的下标操作符；

（2）若 valarray 是常量，则子集表达式将导致新的 valarray；

（3）若 valarray 不是常量，子集表达式将导致一个 indirect_array 型数组，后者以 reference 语意来表现 valarray 的相应元素。

（4）indirect_array 型数组提供赋值操作符和赋值复合操作符来修改子集中的元素；

（5）通过型别转换，可以将 indirect_array 型数组和其他 valarray 以及 valarrays 子集组合起来。

提示：用来指定子集的索引无须排序，并且可以重复出现。

以上内容用例 6-12 来加以说明。

例 6-12

```
#include <iostream>
#include <valarray>
#include <cmath>
using namespace std;
template<class T>void myprint(valarray<T>& v)
{
    int size=v.size();
    for(int i=0;i<size;i++)
    {
    cout<<v[i]<<", ";
    }
    cout<<endl;
}
void main()
{
    double dim[]={1,2,3,4,5,6,7,8,9,10,11,12};
    valarray<double> v(dim,12),v2,v3;
    cout<<"valarray v(Original) : "<<endl;
    myprint(v);
    valarray<size_t> vi(4);
    vi[0]=4;
    vi[1]=6;
    vi[2]=7;
    vi[3]=1;
    v2=v[vi];
    cout<<"valarray v2(indirect_array) : "<<endl;
    myprint(v2);
    v3=pow(v,v2);
    cout<<"v3=pow(v,v2) : "<<endl;
    myprint(v3);
}
```

例 6-12 的执行效果见图 6-9。

图 6-9　例 6-12 的执行效果

6.3　通用数值计算

在 STL 中，按照<algorithm>中非数值算法的风格，提供了 4 个通用的数值算法。这 4 个算法的声明在头文件<numeric>中。这 4 个算法包括：accumulate()，inner_product()，partial_sum()和 adjacement_ difference()。这 4 个算法的功能分别是：求和；内积；部分元素之和；元素的相邻差。

6.3.1　求和（accumulate）

算法 accumulate()的原型如下：

```
    template<class InputIterator, class Type> Type accumulate( InputIterator _First,
InputIterator _Last, Type _Val );
    template<class InputIterator, class Type, class Fn2> Type accumulate( InputIterator
_First, InputIterator _Last,
    Type _Val, BinaryOperation _Binary_op );
```

上述算法的第一种形式是计算序列[_First, _Last]的数值总和，并将该数值总和添加至数值_Val 上。第二种形式对序列[_First, _Last]的每个元素，均使用谓词_op 进行处理，处理之后的结果依次累加，所得总和再添加至数值_Val 上。

谓词 op 的原型如下：

```
    _Binary_op (_Val , *Iter)
```

其中，*Iter 代表序列中的每个元素。_Val 即是算法 accumulate 中的参数_Val。

以上内容用例 6-13 来加以说明。

例 6-13

```
    #include <iostream>
    #include <vector>
    #include <algorithm>
    #include <numeric>
    #include <functional>
    using namespace std;
    void myprint(int& ele)
    {   cout<<ele<<", ";
    }
    int myop(int orig,int ele)
    {   return ele*2+orig;
    }
    void main()
```

```
{   vector<int> vt;
    int dim[]={1,2,3,4,5,6,7,8,9,10,11,12};
    vt.assign(dim,dim+12);
    for_each(vt.begin(),vt.end(),myprint);
    cout<<endl;
    int sum=accumulate(vt.begin(),vt.end(),0);
    cout<<"The sum of vector vt is : "<<sum<<endl;
    sum=accumulate(vt.begin(),vt.end(),0,myop);
    cout<<"The sum2 of vector vt is : "<<sum<<endl;
}
```

例 6-13 的执行效果见图 6-10。

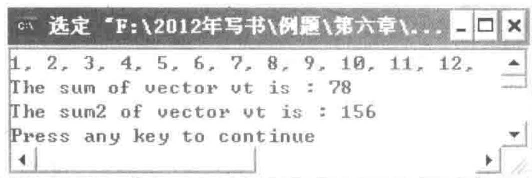


图 6-10　例 6-13 的执行效果

6.3.2　内积（inner_product）

内积是通过把第一个序列中的元素与第二个序列中相同位置的元素相乘，然后把计算的结果相加而得到的。内积算法的函数原型如下：

```
    template<class InIt1, class InIt2, class T> T product(InIt1 first1, InIt1 last1,
Init2 first2, T val);
    template<class InIt1, class InIt2, class T, class Pred1, class Pred2> T product(InIt1
first1, InIt1 last1, Init2 first2, T val,
    Pred1 pr1, Pred2 pr2);
```

上述第一种形式计算并返回[beg, end]区间和“以 beg2 为起始的区间”的对应元素组的内积。具体地说，是针对“两区间内的每一组对应元素”调用以下表达式：

```
initValue=initValue+elem1*elem2
```

上述第二种形式是对区间[beg,end]和“以 beg2 为起始区间”内的对应元素进行 op2 运算，然后再和 initValue 进行 op1 运算，并将结果返回。具体地说，针对“两区间内的每一组对应元素”调用以下表达式：

```
initValue=op1(initValue,op2(elem1,elem2))
```

由上述内容可知，对于两个数值序列：

```
a1,a2,a3,a4,......
b1,b2,b3,b4,......
```

上述两个算法分别计算并返回：

```
initValue + (a1*b1)+(a2*b2)+(a3*b3)+......


initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1.....
```

需要注意的几点：

（1）如果第一个区间为空，则两者返回 initValue；

（2）程序开发者必须确保“以 beg2 为起始的区间”内含足够的元素空间；

（3）谓词（或函数）op1 和 op2，都不允许修改其传入参数内容；

（4）op1 和 op2 两个函数或两个谓词，均是两个参数。

以 6-14 为例，请读者认真研究内积算法的使用方法。

例 6-14

```
#include <iostream>
#include <list>
#include <numeric>
#include <algorithm>
using namespace std;
void myprint(int ele)
{
   cout<<ele<<", ";
}
int op1(int initV,int ele3)
{
   return initV+ele3;
}
int op2(int ele1,int ele2)
{
   return ele1*2+ele2*3;
}
void main()
{
    int dim[]={1,2,3,4,5,6,7,8,9,10,11,12};
    int dim2[]={3,4,5,6,7,8,9,10,11,12,13,14};
    list<int> l1,l2;
    copy(dim,dim+12,back_inserter(l1));
    for_each(l1.begin(),l1.end(),myprint);
    cout<<endl;
    copy(dim2,dim2+12,back_inserter(l2));
    for_each(l2.begin(),l2.end(),myprint);
    cout<<endl;
    int prod=inner_product(l1.begin(),l1.end(),l2.begin(),0);
    cout<<"inner_product of l1 and l2 is : "<<prod<<endl;
    prod=inner_product(l1.begin(),l1.end(),l2.begin(),0,op1,op2);
    cout<<"inner_product of l1 and l2 with function op1 and op2 is : "<<prod<<endl;
    int sum=0;
    for(int j=0;j<12;j++)
    {   sum+=(dim[j]*2+dim2[j]*3);
    }
    cout<<"inner_product of l1 and l2 with function op1 and op2 is : "<<sum<<endl;
//由此证明定义的两个函数 op1                               //  和 op2 是正确的。
}
```

例 6-14 的执行效果见图 6-11。

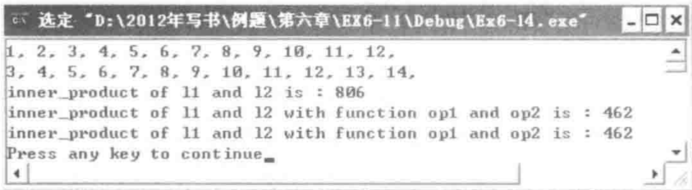


图 6-11 例 6-14 的执行效果

总结：在多数内积算法的例题中，op1 和 op2 采用 multiplies<int>()和 plus<int>()。这样定义 op1 和 op2，读者并不能确切地理解 op1 和 op2 到底是如何进行运算的？例 6-14 中定义了两个自定义谓词 op1 和 op2，代码清楚，功能简单，便于读者理解。

6.3.3 部分和（partial_sum）

算法 partial_sum()的功能是对迭代器[_First, _Last]指明范围内的元素进行部分求和。计算结果保存在从_Result 开始的序列。部分求和的计算步骤：首先把原序列的第 1 个元素赋值给新序列的第 1 个元素；然后把原序列的前两个元素相加，把结果赋值给新序列的第 2 个元素；把原序列的第 2 和第 3 个元素相加，把结果赋值给新序列的第 3 个元素；依此类推，直到计算完原序列的全部元素为止。

算法 partial_sum()的函数原型如下：

```
   template<class InputIterator, class OutIt> OutputIterator partial_sum( InputIterator
_First, InputIterator _Last,
            OutputIterator _Result );
   template<class InputIterator, class OutIt, class Fn2>OutputIterator partial_sum
(InputIterator _First, InputIterator _Last,
            OutputIterator _Result, BinaryOperation  _Binary_op );
```

上述两种形式中，第一种形式计算源区间[_First, _Last]中每个元素的部分和，然后把结果写入以 result 为起点的目标区间。第二种形式包含一个谓词_Binary_op，对于源区间中的每个元素要进行_Binary_op 处理。

通过以上描述可知，对于序列 a,b,c,d。算法 partial_sum 产生序列 a,a+b,a+b+c,a+b+c+d。

值得注意的问题包括以下几条：

（1）两种形式都返回目标区间内“最后一个被写入的值”的下一位置；

（2）源区间和目标区间可以相同；

（3）程序开发者必须确保目标区间足够大，否则需要使用插入型迭代器；

（4）谓词_Binary_op 不允许修改传入其内的参数。

目前参考资料给出的谓词多为 multiplies<T>()等类型。

下面以例 6-15 说明算法 partial_sum()的使用方法。谓词为自定义函数。

例 6-15

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

```
void myprint(int& ele)
{   cout<<ele<<", ";
}
int op(int pre_ele,int ele)
{   return pre_ele+ele*2;
}
void main()
{   vector<int> v1,v2,v3;
    int dim[]={1,2,3,4,5,6,7,8,9,10,11,12};
    v1.assign(dim,dim+12);
    cout<<"vector v1: "<<endl;
    for_each(v1.begin(),v1.end(),myprint);
    cout<<endl;
    partial_sum(v1.begin(),v1.end(),back_inserter(v2));
    cout<<"vector v2: "<<endl;
    for_each(v2.begin(),v2.end(),myprint);
    cout<<endl;
    partial_sum(v1.begin(),v1.end(),back_inserter(v3),op);
    cout<<"vector v3: "<<endl;
    for_each(v3.begin(),v3.end(),myprint);
    cout<<endl;
}
```

例 6-15 的执行效果见图 6-12。

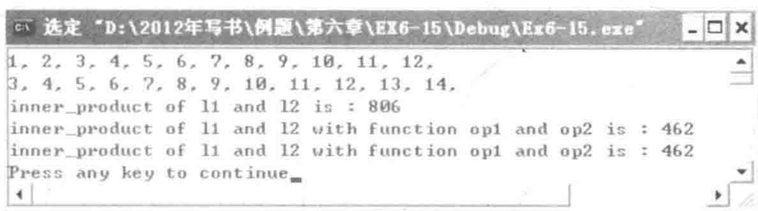


图 6-12　例 6-15 的执行效果

总结：请读者认真阅读例题，尤其是谓词函数的定义。

6.3.4　序列相邻差（adjacent_difference）

算法 adjacent_difference ()用于计算序列的相邻差。算法的功能是计算由迭代器指定的源区间[_First, _Last]内相邻元素的差值，并将结果赋予至以_Result 为起始的序列中。相邻差值的计算步骤为：首先把源序列的第 1 个元素赋值给新序列的第 1 个元素；再把源序列的第 2 个元素减去第 1 个元素，其结果赋值给新序列的第 2 个元素；然后把源序列的第 3 个元素减去第 2 个元素，并将结果赋值给新序列的第 3 个元素；依此类推，直至计算完源序列的全部元素为止。同样，算法 adjacent_difference()也提供了带有谓词的形式。

算法的函数原型如下：

```
    template<class InputIterator, class OutIterator>OutputIterator adjacent_difference
(InputIterator _First,  InputIterator
      _Last, OutputIterator _Result );
    template<class InputIterator, class OutIterator, class BinaryOperation> OutputIterator
adjacent_difference(InputIterator _First, InputIterator _Last,OutputIterator _Result,
BinaryOperation  _Binary_op );
```

算法模型的第一种形式，计算源区间[_First,_Last]中元素的相邻差，并将计算结果保存至以_Result 为起始的新序列中。

算法模型的第二种形式，针对源区间[_First, _Last]中的每个元素和其紧邻的前一个元素调用谓词处理，并将结果写入以_Result 为起始的目标序列中。源序列的第一个元素只是被单纯地复制。

以下几个问题值得注意：

（1）算法函数的返回值为目标区间内“最后一个被写入的值”的下一位置；

（2）算法的第一种形式相当于将一个绝对值序列转换为一个相对值序列；

（3）源区间和目标区间可以相同；

（4）程序开发者需要确保目标区间足够大，否则需要使用插入型迭代器；

（5）谓词函数_Binary_op 不能修改传入其内的参数。

由以上可知，对于序列 a,b,c,d,……，算法 adjacent_difference()将产生新序列：a,b-a, c-b, d-c,……。

提示：算法 partial_sum()和算法 adjacent_difference()的效果是完全相对的，两种算法之间是逆向操作的关系。

下面以例 6-16 说明算法 adjacent_difference()的使用方法。

例 6-16

```
#include <vector>
#include <list>
#include <numeric>
#include <functional>
#include <iostream>
#include <algorithm>
using namespace std;
void myprint(int& ele)
{   cout<<ele<<", ";
}
int op(int ele2,int ele1)
{   return ele2*5-ele1*3;
}
void main()
{   int dim[]={1,2,3,4,5,6,7,8,9,10,11,12};
    vector<int> v1;
    v1.assign(dim,dim+12);
    for_each(v1.begin(),v1.end(),myprint);
    cout<<endl;
    vector<int>v2(v1.size());
    vector<int>::iterator it;
    it=adjacent_difference(v1.begin(),v1.end(),v2.begin());
    cout<<"v1 is dealt by the algorithm adjacent_difference :"<<endl;
    for_each(v2.begin(),v2.end(),myprint);
    cout<<endl;
    adjacent_difference(v1.begin(),v1.end(),v2.begin(),op);
    cout<<"v1 is dealt by the algorithm adjacent_difference with pred  op: "<<endl;
    for_each(v2.begin(),v2.end(),myprint);
```

```
    cout<<endl;
}
```

例 6-16 的执行效果见图 6-13。

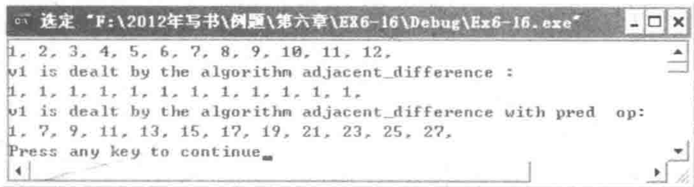


图 6-13　例 6-16 的执行效果

总结：请读者认真阅读例题，对照前面讲述的内容，体会算法 adjacent-difference()的调用效果。

6.4　全局性数学函数

头文件<cmath>和<cstdlib>提供了从 C 语言继承下来的全局性数值函数。这些函数包括：pow()，exp()，sqrt()，log()，log10()，log()，sin()，cos()，tan()，sinh()，cosh()，tanh()，asin()，acos()，atan()，atan2()，ceil()，floor()，fabs()，fmod()，frexp()，ldexp()，modf()。

这些函数在 C 语言中，其函数声明均包含在头文件<math.h>和<stdlib.h>中。在 C++中，这些函数改在头文件<cmath>和<cstdlib>中声明。

和 C 语言不同的是，C++将某些操作函数针对不同型别进行了重载，从而导致部分 C 函数失去价值。例如：C 语言提供的 abs()、labs()、fabs()，分别计算 int、long、double 类型变量或常数的绝对值；而 C++的 abs()完成了重载，具备了上述 3 个函数的统一功能。

头文件<cstdlib>内定义的数值函数包括：abs()，labs()，div()，ldiv()，srand()，rand()。确切地说，所有浮点数的数值函数，均对 float、double、long double 3 个型别实现重载。但同时也带来了新的问题，传入的数值类型为整数时，会引发模棱两可的情况。例如：

```
sqrt(7);        //不合理
```

应该是：

```
sqrt(7.0);
```

或：

```
float x=7;      //OK
sqrt(x);        //OK
```

程序库设计者处理这一问题时，还没有尽善尽美。有些提供了重载，有些遵循标准规格，有些针对所有的数值类别加以重载，有些借助预处理器在各种策略间切换。在应用程序的开发过程中，模棱两可的情况时有发生。可移植性高的代码，需要参数型别有精确的匹配。

下面给出上述数学函数的声明形式。

```
float  abs  (float)
float  acos (float)
float  asin (float)
float  atan (float)
```

```
float  atan2 (float)
float  ceil (float)
float  cos (float)
float  cosh (float)
float  exp  (float)
float  fabs  (float)
float  floor  (float)
float  fmod  (float)
float  frexp  (float)
float  ldexp  (float)
float  log  (float)
float  log10  (float)
float  modf  (float)
float  pow  (float, float*)
float  pow  (float,  float)
float  sin  (float)
float  sinh  (float)
float  sqrt  (float)
float  tan  (float)
float  tanh  (float)
double  abs  (double)
double  pow  (double, int)
long double  abs  (long double)
long double  acos  (long double)
long double  asin  (long double)
long double  atan  (long double)
long double  atan2  (long double)
long double  ceil   (long double)
long double  cos   (long double)
long double  cosh  (long double)
long double  exp   (long double)
long double  fabs  (long double)
long double  floor  (long double)
long double  fmod  (long double, long double)
long double  frexp  (long double, int*)
long double  ldexp  (long double, int)
long double  log    (long double)
long double  log10  (long double)
long double  modf  (long double)
long double  pow   (long double)
long double  pow  (long double)
long double  sin  (long double)
long double  sinh  (long double)
long double  sqrt  (long double)
long double  tan  (long double)
long double  tanh  (long double)
```

总结：本节仅简单介绍了部分全局性数学函数，这些函数的使用已经非常普遍。希望读者在今后的学习和程序编写过程中，尽量多使用这些全局性的数学函数。

6.5　本章小结

本章主要有 4 个重点内容：复数运算、数组（向量）运算、通用数值计算、全局性数学函数。

通用数值计算只有在数学或者和数学紧密相关的场合才会使用。

全局性数学函数的使用极为普遍，在图形编程、工程计算、理论研究以及其他更宽泛的范围内，数学函数是非常重要的，希望读者认真学习并加以使用。

复数运算对于在校的研究生或高年级本科生尤为重要，学习了这部分内容之后，可以方便地使用 C++以及 STL 实现课题研究。

数组（向量）运算也是数学中的重要内容。在线性代数或矩阵论中均有涉及，学习这部分内容，对这部分数学课程的学习大有帮助。读者通过自己编写程序，可以实现对数学知识的更加深刻的理解。

第 7 章　输入/输出流

每个程序必须包含输入与输出，即使没有输入，也必须包含输出，否则程序的执行结果将无法观察。C++使用了很多较为高级的语言特性来实现输入与输出，包括：类，派生类，函数重载，虚函数，模板和多重继承等。要真正理解 C++如何实现 I/O 功能，必须了解 C++的诸多内容。

IOStream 程序库提供了一系列 IO classes。用于 I/O（输入/输出）的各个 classes，是 C++标准程序库的重要组成部分。标准程序库中的 IO classes 不仅局限于文件、屏幕或键盘，它形成了一套富有弹性的框架，用于任意数据格式化，处理（存取）任意外部表述。为了标准程序库的一致性和某些新需要，IO classes 进行了数度修改，但基本原则始终如一。最近几年，I/O 逐渐被国际化，并且 IOStream classes 均已被泛型化，并支持不同的字符表述，使用时需要包含头文件<iosfwd>。在 STL 中，和其他部分一样，IOStream 的所有符号均定义于命名空间 std 内。std::IOStream 类库从标准输入中读取输入，并在标准输出上显示结果。而 IOStream 类库具有的能力远不止读/写系统控制台，还包括更复杂的格式化和文件 I/O 流等内容。流被用于在不同的机器间传递对象，用于加密消息流、数据压缩和对象的持续性存储，甚至包括大量其他工作。

本章将介绍使用 istream 类对象 cin 和 ostream 类对象 cout 进行输入和输出的基本方法，同时使用 ifstream 和 ofstream 对象进行文件输入和输出。还将介绍 C++的输入和输出类。用于文件输入和输出的 C++工具都是基于 cin 和 cout 的基本类定义。ANSI/ISO C++标准委员会的工作是让 C++ I/O 与已有的 C 语言 I/O 更加兼容。

7.1　IOStream 简介

IOStream 类具有处理格式化显示的技术。本节将介绍 Stream 类对象、Stream 类别、Stream 类的操作符。

7.1.1　Stream 对象

1. 输入流和输出流对象

C++I/O 由 stream 完成。所谓 stream 是一条数据“流”，字符序列在其中“川流不息”。按面向对象的原则，stream 是由某个类别定义出来的具有特定性质的对象。C++流被实现为类模板。std::cout 和 std::cin 的对象是 std::ios 的派生类的全局实例对象。尽管后面将使用 std::ios 定义的常量和函数—

一包括 std::ios::beg 和 std::ios::setprecision(int)。程序主要处理的对象类型均派生自 std::ios 类。这些类对象调用的成员函数均是在 ios 类中定义的。

Stream 主要包括输出流和输入流。输出操作被解读为“数据流入 stream”，输入操作则是“数据流出 stream”。还包括一些为标准 I/O 通道而定义的全局对象。

C++的 iostream 类库管理了诸多细节。例如，在程序中包含 iostream 文件将自动创建 8 个 Stream 对象（4 个用于窄字符，4 个用于宽字符）：

（1）cin 对象对应标准输入 stream。默认情况下，stream 被关联到标准输入设备（键盘）。wcin 对象与此类似，处理的是 wchar_t 类型；

（2）cout 对象与标准输出 stream 相对应。默认情况下，这个 stream 被关联到标准输出设备（显示器）。wcout 对象与此类似，处理的是 wchar_t 类型；

（3）cerr 对象与标准错误 stream 相对应，可用于显示错误信息。默认情况下，这个 stream 被关联到标准输出设备（显示器）。这个 stream 没有缓冲，意味着信息将被直接发送给屏幕，不会等到缓冲区填满或新的换行符。wcerr 对象与此类似，处理的是 wchar_t 类型；

（4）clog 对象同样对应标准错误 stream。默认情况下，这个 stream 被关联到标准输出设备（通常为显示器）。这个 stream 有缓冲。wclog 对象与此类似，但处理的是 wchar_t 类型。

当 iostream 文件为程序声明一个 cout 对象时，该对象将包含存储了与输出有关的信息的数据成员，例如：显示数据时使用的字段宽度、小数位数、显示整数时采用的计数方法以及描述用来处理输出流的缓冲区的 streambuf 对象的地址。cout 对象凭借 streambuf 对象的帮助，管理着流中的字节流。

2．文件流对象

文件流对象是用于文件操作时，程序需要实例化文件流类的对象，文件流类包括 3 个：ofstream、ifstream 和 fstream。这 3 个对象分别支持文件输入、输出以及兼具输入和输出。程序需要在流类的构造函数中包括实参，或者实例化未初始化的文件流对象。

7.1.2　Stream 类别

对于不同种类的 I/O（输入、输出、文件存取），对于不同的 stream classes ，其中最重要的是：

- class istream，定义 input stream，可用来读取数据；
- class ostream，定义 output stream，可用来写出数据；
- class ofstream，实现文件输入与输出。

1．istream 类和 ostream 类

两者分别具体实现自 template class basic_istream<>和 basic_ostream<>，以 char 作为字符型别。事实上整个 IOStream 程序库均不依赖任何特定的字符型别，而以一个 template 参数替代。此种参数化在 string classes 也存在。

IOStream 程序库定义了整个型别为 istream 和 ostream 的全局对象，对应标准的 I/O 通道。前面已讲述：

cin 是供使用者输入用的标准输入通道，对应于 cstdin。操作系统通常将它和键盘连接。

cout 是供使用者输出用的标准输出通道，对应于 cstdout。操作系统通常将它和监视器（屏幕）连接。

cerr 是所有错误信息所使用的标准错误输出通道，对应于 cstderr。操作系统通常将它和监视器连接。默认情况下，cerr 无缓冲装置。

clog 是标准日志通道，没有对应物。默认情况下，操作系统将它连接入 cerr 所连接的装置，但 clog 设有缓冲装置。

“正常输出”和“错误信息的输出”需要分离，让程序以不同的方式对待两种输出。例如：可以将正常输出重新定向至某个文件，而同时仍然令错误信息显示于控制台。前提是操作系统必须支持标准 I/O 通道的重定向功能。这种分离方式起源于 UNIX 的 I/O 重定向概念。

ostream 类派生自 std::ios 类，用来处理流的输出。std::cout 对象是 std::ostream 类的外部对象。cout 对象在库中声明，cout 的 extern 声明出现在<iostream>中，所包含的<iostream>的程序均可使用 cout 对象。程序通过使用重载“<<”插入符来写 std::ostream 对象。std::ostream 类具有的“<<”插入运算符足以支持把大多数标准 C++数据类型写到输出流中。

std::istream 类管理流输入的方式和 std::ostream 类管理流输出的方式相同。在头文件<iostream>中声明，std::cin 对象从标准输入设备中读取数据。std::istream 类使用重载“>>”抽取运算符来读取输入。重载“>>”抽取运算符足以支持读取标准 C++数据类型的数据，用户自定义的类可以通过重载“>>”抽取运算符来从 std::istream 对象中读取数据。

2. ofstream 类

C++ I/O 类软件包处理文件输入和输出的方式与处理标准输入和输出的方式非常相似。要写入文件，需要创建一个 ofstream 对象，并使用 ostream 方法，如“<<”插入符或 write()。要读取文件，需要创建一个 ifstream 对象，并使用 istream 方法。例如：“>>”抽取操作符或 get()。文件管理是非常复杂的，在使用时必须将文件和流关联，并且还分多种操作模式（只读，只写，读写）；写文件时，可能需要创建新文件、取代旧文件或添加到旧文件中，还可能需要在文件中来回移动。为实现对文件的复杂操作，还增加了用于文件输入的类（ifstream）、文件输出的类（ofstream）、同步文件输入输出类（fstream）

7.1.3 Stream 操作符

operator>>和 operator<<被相应的 stream classes 重载，分别用于输入和输出。C++移位操作符变成了 I/O 操作符。例如：

```
int a,b;
std::cin >> a >> b;                              //输入整型变量 a,b 的数值
std::cout<<"a: "<<a <<"b: "<<b<<std::endl;        //输出整型变量 a,b 的数值
```

7.1.4 操控器（Manipulators）

大部分输出语句的最后，会写一个称为操控器的东西，即 std::endl。操控器是专门用来操控 stream 的对象，仅改变输入或格式化输出的解释方式，例如：数值进制 dec（10 进位），hex（16 进位），oct（8 进位）。用于 ostream 的操控器并不凭空造出输出数据，用于 istream 的操控器不会吃掉输入数据。部分操控器会发生立即操作，例如：用于“刷新 output 缓冲区”或“跳过 input 缓冲区空格”的那些操控符。

操控器 endl 的功能是终止一行代码，主要完成两项任务：

（1）输出换行符号“\n”；

（2）刷新 output 缓冲区；

IOStream 程序库中最重要的一些操控器包括：endl，ends，flush，ws。

```
endl：属于ostream类，输出'\n'，并刷新output缓冲区；
ends：属于ostream类，输出'\0'；
flush：属于ostream类，刷新output缓冲区；
ws：属于istream类，读入并忽略空格。
```

总结：本节主要介绍与 IOStream 类有关的基本概念：Stream 对象，Stream 类别，Stream 操作符，操控器等，以便于后续章节的学习。对上述内容，希望读者认真阅读，对这些概念能基本了解。

7.2 IOStream 基本类和标准 IOStream 对象

本节主要讲述和 IOStream 类相关的头文件、标准 Stream 操作符、Stream 状态以及标准输入和输出函数。

7.2.1 和 IOStream 类相关的头文件

关于 stream classes 的诸多定义包含在头文件<iosfwd>，<streambuf>，<istream>，<ostream>和<iostream>中。大部分头文件主要用于 C++标准程序库的内部组织。IOStream 程序使用者只需含入拥有各个 stream classes 声明式的<iosfwd>，在运用输入或输出功能时，分别含入<istream>或<ostream>。如果不需要使用标准 stream 对象，则不需要含入<iostream>。

对于特殊的 stream 特性，例如，参数化的操控器、file streams、string stream，需要含入其他头文件。

7.2.2 标准 Stream 操作符

C/C++的操作符“>>”和“<<”，分别用于位左移和位右移，然而 basic_istream<>和 basic_ostream<>重载它们，使之成为标准的 I/O 操作符。本节主要介绍 output 操作符、input 操作符、特殊型别的 I/O、操作符的格式化等内容。

1. output 操作符

```
basic_ostream<charT, traits>& operator
<<(basic_ostream<charT,traits>&(*pf)(basic_ostream<charT,traits>&))
basic_ostream<charT,traits>&operator<<(basic_ios<charT,traits>&(*pf)(basic_ios<charT,trait
s>&)
basic_ostream<charT,traits>&operator<<(ios_base&(*pf)(ios_base&))
basic_ostream<charT, traits>&operator<<basic_streambuf<charT,traits>* sb);
```

以上给出了输出操作符的 4 种声明形式。basic_ostream（ostream 和 wostream 也算）将“<<”定义为 output 操作符，对所有基本型别均进行重载，包括：char*，void* ，bool。stream 将其 output 操作符定义为：把第二参数发送到相应的 stream 中。操作符“<<”会被重载，第二参数可以是任意型别。C++标准程序库使用相同的机制提供 string、bitset 及 complex 等类型的 output 操作符。在前面章节，已经大量使用“<<”符号用于输出，这里不再详述。输出机制的可扩展性，使使用者自定义型别天衣无缝地融入 I/O 系统中，和 C 语言的 printf()输出机制相比，是跨越式进步。现在，程序员在输出数据时，不仅仅局限于标准型别，还可以包括自定义型别。

操作符“<<”不仅可以输出单个对象，还可以输出多个对象。即使用“<<”，可以将输出操作

符串成一串，“<<”的输出次序是由左到右。

2. input 操作符

```
basic_istream<charT,traits>&operator>>(basic_istream<charT,traits>&*pf)(basic_i
stream<charT,traits>&)
basic_istream<charT,traits>&operator>>    (basic_ios<charT,traits>&
(*pf)(basic_ios<charT,traits>&));
basic_istream<charT,traits>& operator>>(ios_base& (*pf)(ios_base&))
```

basic_istream 将“>>”定义为 input 操作符。和 basic_ostream 类似，basic_istream 对基本型别（char*，void*，bool）重载了操作符“>>”。Stream input 操作符被定义为：将读入的数值存储在第二参数。参数的传递方向是箭头方向。

和“<<”同样，程序开发者可以对任意型别重载 input 操作符，并串联运用它们。

3. 特殊型别的 I/O

标准 I/O 操作符还定义了 bool、char* 和 void* 型别的输入和输出。“>>”和“<<”还可以运用在自定义类型对象的输出和输入。

bool 默认情况下，布尔值的读取和输出均以数字的形式出现。在 C 语言中，true 代表“真”或“1”，false 代表“假”或“0”。如果读入的数值既非 1 也非 0，即被认为是错误的。ios 类可能会抛出相应的异常。

最重要的是设立 stream 的格式化选项，以字符串形式对布尔量进行 I/O 操作。目前仍然是国际化的议题，多数情况下采用字符串“true”和“false”，特殊的 locale objects 会使用不同的字符串，这种情况在不同的国家、不同的语系、各自存在不同的形式。

对于 char 和 wchar_t，经由操作符“>>”读取一个 char 或 wchar_t 字符时，一般开头的空格会被去掉，如果需要保留读入的所有字符（包括空格），需要清除 skipws 标志或利用成员函数 get()。

对于 char*型的 C 字符串，在读入该类型时，起头的空格会跳过，一直到非空格或文件结束为止，是否跳过起头空格，可由标志 skipws 控制。以上阐述说明，C 语言的字符串读入的最大长度远超过 80 字符。因此，在 C++语言中的 string 型对象，可容纳足够的字符。使用 string 代替 char*可使程序更轻松、更安全，另外 string 还提供更便捷的函数（getline），可以逐行地读入数据。尽量使用 string 会为程序员提供更快速、更完善的编程效率。

对于 void*，操作符“<<”和“>>”为指针的打印和读入提供方便。型别为 void*的参数被传递至 output 操作符时，其地址将被输出，而不是输出指针指向的内容。请看下面例 7-1 的输入/输出实例。

例 7-1

```
#include <iostream>
using namespace std;
void main()
{
 char* cstring="Hello";        //输出地址时需要使用 void*
 cout<<"string \" "<<" "<<cstring<<" \" "<<"is located at address: "<<
     static_cast<void*>(cstring)<<endl;
  cout<<"string \" "<<" "<<cstring<<" \" "<<"is located at address: "<<
     (void*)cstring<<endl;
}
```

程序输出结果如下：

```
string " Hello " is located at address: 0046E04C
string " Hello " is located at address: 0046E04C
```

借助 input 操作符读入地址，需要注意：地址是临时性的，同一对象的地址在程序每次执行时可能会不同。地址的输出和输入的可能应用：交换对象地址或共享内存。

对于 stream Buffers（串流缓冲区），操作符“>>”和“<<”可以直接用于读/写或改写 stream buffer，运用 C++ IO stream 实现复制文件的最快办法。

自定义型别在使用时比较方便。要考虑所有可能的格式和错误条件时，需要付出更多努力。对于自定义型别扩展标准 IO 机制，会有更复杂的讨论。

4. 操作符的格式化

在前面的章节，已经涉及在输出时使用 cout。在输出时，可以使用多种格式显示数字。C++的 IOStream 库使用操作符、标记和成员函数支持类似的格式化显示。对于简单的数值列表显示，cout 对象会工作得非常好。使用标准 IOStream 系统的格式化，以多种方式显示输出。

5. 带参数的操作符

为便于程序开发者可以不直接以标志位的方式处理流的状态，标准库还提供部分函数。用于操控类似状态，关键思想是在读写对象之间插入一个修改状态的操作。

例如：函数 flush 可以显式的刷新输出缓冲区。

```
cout<<x<<flush<<y<<flush;
```

例如：noskipws 操控符可以在输入时省略前面的空格。

```
cin>>noskipws>>x;
```

在上述语句中，noskipws 可以代替函数 cin.unsetf(ios_base::skipws)。

下面主要讲述带参数的操控符。

带参数的操控符非常有用。例如：cout.precision(int *np*)。函数的执行结果是，使用 4 位精度输出随后输出的变量或其他类型对象。

```
double d=20.123456;
cout.precision(4);        //精度保留 4 位
cout<<d<<endl;            //输出 d
```

输出结果：

```
20.12
```

6. 标准 IO 操作符

标准库针对各种各样的格式状态和状态变化提供了一批操控符。标准操控符定义在名称空间 std 中。针对 ios_base 的操控符由<ios>给出。针对 istream 和 ostream 的操控符分别在<istream>和<ostream>中给出，在<iostream>中同时给出。其他更复杂的标准操控符则由头文件<iomanip>给出。例如：

```
boolalpha()               //用符号形式表示真和假
noboolalpha()             //unsetf(ios_base::boolalpha
showbase()                //输出八进制和十六进制时，分别加各自的前缀
noshowbase()              //unsetf(ios_base::showbase
showpoint()
noshowpoint()             //unsetf(ios_base::showpoint)
```

```
showpos()
noshowpos()                    //unsetf(ios_base::showpos)
skipws()                       //跳过字符串起始的空格
noskipws()                     //unsetf(ios_base::skipws)
uppercase()                    //使用 X 和 E，而不是 x 和 e
nouppercase()                  //使用 x 和 e，而不是 X 和 E
internal()                     //调整
left()                         //值后填充
right()                        //值前填充
dec ( )                        //整数基数
hex()                          //整数基数 16
oct()                          //整数基数 8
fixed()                        //浮点格式
scientific()                   //科学格式
endl()                         //回车换行，并刷新
ends()                         //输出 '\0'
flush()                        //刷新流
ws()                           //去掉"空白"字符
resetiosflags()                //清楚标志
setiosflags()                  //设置标志
setbase()                      //基于 b 输出整数
setfill()                      //用 c 作为填充字符
setprecision()                 //n 位数字
setw()                         //下个域宽为 n 个字符
```

如果使用不带参数的操控符，需要加括号。要使用带参数的标准操控符，应记住包含头文件<iomanip>。

例如：

```
double d1=30.3245;
cout<<setprecision(4)<<d<<endl;          //设置输出精度为 4 位
```

输出结果如下：

```
30.32
```

7. 用户自定义操作符

程序开发者按照标准的风格使用各自的操控符。precision 对所有的输出操作具有持续性的作用，而 width()仅对下一个数值输出操作起作用。作为程序开发人员，往往希望在按某种预定格式输出浮点数时，事情可以变得非常简单，而又不影响在这个流上随后的输出操作。最基本的想法是定义一个表示格式的类，用另一个类表示一个格式再加一个需要格式化的值，之后让运算符"<<"按照格式向 ostream 输出该值。下面的例 7-2 就是给我们示范如何处理这种情况。

例 7-2

```
#include <iostream>
#include <iomanip>
using namespace std;
void Of(double x)
{
 cout.precision(6);
 cout<<cout.scientific<<x<<endl;         //使用科学计数法形式输出
```

```
}
void main()
{
... ...
 double d2=12345678.987634568;
 Of(d2);
... ...
}
```

程序员可以通过编写自定义的格式化输出函数，实现将数据按各自需要的格式输出。

7.2.3　Stream 状态

Stream 维护一种状态，标志 IO 成功，并能指出不成功的原因。每个流都包含相关联的状态。如果状态是 good，则说明操作是成功的；下一个输入操作可能成功，否则会失败。在读取变量时，操作状态如果不是 good，该流将不产生任何作用。流状态为 fail 或 bad 之间的差异很微妙。如果状态为 fail，则可以假设流未被破坏，并且没有字符丢失；如果状态是 bad，就意味着彻底失败。

1．stream 状态常数

stream 定义了一些型别为 iostate 的常数，用以反映 stream 的状态。iostate 是 class ios_base 的成员，具体型别由实例版本决定，即 iostate 未被限定是列举型别、整数型别。iostate 型别的常量主要包括：goodbit，eofbit，failbit，badbit。goodbit 代表一切都好；eofbit 代表文件末尾标志；failbit 代表错误，某个 IO 操作未成功；badbit 代表毁灭性错误，不确定状态。failbit 和 badbit 相比，后者是更严重的错误。两者的区别在于：

failbit：某项操作未能完成，但 stream 大体 OK，即设立这个位；通常是由于读入格式错误，例如程序需要读入整数，却遇到字符。

badbit：　若 stream 由于不明原因而损坏或丢失数据，即设立这个位。例如某个 stream 被定为指向“文件起点”的更前方。

eofbit 经常和 failbit 同时出现，在 eof 之后试图读取数据时，会检测出 end-of-file 状态。在最后一个字符被读取时，eofbit 并未出现，再次试图读取字符时，往往 eofbit 和 failbit 同时被设置，因为读取操作失败了。

上述常数并非全局性，而仅定义于 class ios_base 类中，需要添加作用域操作符。例如：

```
std::ios_base::eofbit
```

使用 ios_base 派生类别也行。以上标志均由类 basic_ios 维护，能够在 basic_istream 和 basic_ostream 的所有对象中使用。stream buffer 没有状态标志，单个 stream buffer 可被多个 stream object 共享。上述标志仅能反映最后一次操作的 stream 状态，因为这些标志可能被以前的操作设置。

2．stream　状态相关成员函数

stream 当前状态的一些标志可以由一些成员函数来访问。这些成员函数主要包括：good，eof，fail，bad，rdstate，clear，clear(state)，setstate(state)。下面一一介绍。

（1）good()：若 stream 正常无误，返回 true，表示 goodbit 已设置。函数返回值为 bool 类型，如果 basic_ios 的 rdstate 等于 goodbit 时，函数返回 1，否则返回 0。

（2）bad()：发生毁灭性错误时，返回 true。此时 badbit 被设置。函数返回值为 bool 类型，如果 basic_ios 的 rdstate&badbit 是非零，函数返回 1，否则返回 0。

举例如下：

```
#include <iostream>
void main( void )
{
   using namespace std;
   bool b = cout.bad( );                        //返回 badbit 的值
   cout << b << endl;
   b = cout.good( );                            //返回 goodbit 的值
   cout << b << endl;
}
```

输出结果：

```
0
1
```

（3）eof()：若程序在读取文件时，遇到流末尾 end-of-file 时，返回 true。表示 eofbit 已设置。

（4）fail()：若发生错误，返回 true。表示 failbit 或 badbit 被设置为 1。

举例如下：

```
 fstream fs;
 int n=1;
 fs.open(".\\eof.txt");                        //打开文件
 cout<<fs.eof()<<endl;                         //是否处于文件末尾
 fs>>n;                                        //输入整型变量 n
 cout<<fs.eof()<<endl;                         //是否处于文件末尾
输出结果:
0
1
```

（5）rdstate()：返回当前已设置的所有标志；函数返回值为 iostate 类型。

（6）clear()：清除当前所有标志后；返回值为空。

（7）clear(state)：清除当前所有标志后，设置标记 state；返回值为空。

（8）setstate(state)：设置 state 标志。函数返回值为空。其实质是调用函数 clear(_State|rdstate)。参数 state 代表需要设置的标志位。

需要说明一点：被设置的位仅能反映过去曾发生的事情，如果某次操作后，发现某个或若干位被设置，无法确定究竟是该次或先前操作导致的该结果。如果想通过标志了解或获取错误信息，在操作前应先设置 goodbit()。并且对于每种标志的访问，必须使用正确的访问方法。

举例如下：

```
    fstream fx;
    fx.open(".\\test.txt",ios::out);                              //打开文件
    fx.clear();                                                   //清除所有标志位
    cout<<"badbit: "<<(fx.rdstate()&ios::badbit)<<" failbit: "<<(fx.rdstate()&ios::
failbit)<<" eofbit: "
  <<(fx.rdstate()&ios::eofbit)<<" badbit: "<<(fx.rdstate()&ios::badbit)<<endl;
                                                                  //显示各标志位
    fx.clear( ios::badbit | ios::failbit | ios::eofbit );         //清除标志位
    cout<<"badbit: "<<(fx.rdstate()&ios::badbit)<<" failbit: "<<(fx.rdstate()&ios::
failbit)<<" eofbit: "
  <<(fx.rdstate()&ios::eofbit)<<" badbit: "<<(fx.rdstate()&ios::badbit)<<endl;
                                                                  //显示各标志位

    fx.close();
```

输出结果如下：

```
badbit: 0 failbit: 0 eofbit: 0 badbit: 0
badbit: 4 failbit: 2 eofbit: 1 badbit: 4
```

3．stream 状态和异常

（1）stream 状态

stream 定义了两个用于 bool 表达式的函数 operator void* ()和!()。C++的异常处理机制可以处理错误和异常。在多数情况下，stream 不会抛出异常。标准化之后的 stream 允许对任何状态标志进行定义，此状态标志被设置时，会引发异常。异常处理机制包含了函数 exception()。下面将主要介绍上述 3 个函数的使用方法。

函数 operator　void* ()和!()，并不是显式的调用该函数，而是隐含式的调用。例如：

```
while(std::cin)
{
… …
}
```

结构中的 bool 条件并不一定要转化为 bool，只要能够转化成某个整数型别或指针型别即可。转换为 void* 是为了在同一表达式中读入对象并测试是否成功。例如：

```
If(std::cin>>x)
{
}
```

此时的 cin 用于条件判断，cin 会调用 operator void*，返回“stream 是否发生错误”。

通常，在 while 循环中，有时使用 eof 作为循环终止的条件，此时一般 failbit 和 badbit 均被设置。

使用“>>”操作符时，默认情况下起头空格会被跳过。但如果从流中输入的型别是 char，空格是有意义的。

使用 operator!进行反相测试，会返回“stream 是否发生错误”，此时标志位 failbit 和 badbit 被设置为 true。

```
If(!std::cin>>x)
{
}
```

注意：上述 cin 是 stream 的对象，而是用!cin 时，是描述 cin 状态的布尔值。

（2）处理异常

调用函数 exception 时，可以获得引发目前异常的标志。调用带参数的 exception()，如果参数代表的标志被设置，会引发相应异常。函数 exception()的原型为：

```
exceptions(flags)          //设定“会引发异常”的标志
exception()                //返回引发异常的标志
```

函数的第二种形式，如果返回值是 goodbit，表示没有任何异常被抛出；函数的第一种形式是设置相应的标志位，并引发相应异常。如果输入的参数是 goodbit 或者是 0，则不会引发异常。

抛出异常是“程序调用 clear()或 setstate()之后”，设立该标志之后，如果该标志已被设置并且未被清除，同样会抛出异常。抛出的异常是 std::ios_base::failure 对象，派生自类 exception。

需要指出的是，获取错误信息的唯一可移植方式是借助函数 what()。强调一下，只有函数 what()的操作具有移植性。但 what()的返回值不具有移植性，需要根据实际情况判断函数的返回结果。

stream 的异常处理能力主要是在读取"格式化数据"时，尽显身手。即使使用异常，仍然存在诸多问题。当读取数据至文件末尾时，end-of-file 带来的异常随之发生。通过检查流状态，可以分清楚到底是由于数据错误还是由于达到文件末尾而触发的异常。

下面用例 7-3 来说明 exception()的使用方法。

例 7-3

```
#include <exception>
#include <iostream>
#include <istream>
#include <ostream>
using namespace std;
void main()
{
 ios::iostate olde=cin.exceptions();                         //返回 cin 的异常标志位
 cout<<"old exceptions: "<<olde<<endl;                       //输出该标志位
 int x=0;
 try{
 cin.exceptions(ios::eofbit|ios::failbit|ios::badbit);      //设置引发异常的标志位
 //cin.exceptions(ios::goodbit);
 cin>>x;
 }
 catch(exception& e)
 {
     cout<<"exception: "<<e.what()<<endl;                   //输出异常类别
 }
}
```

程序执行时，本应该输入一个整型数据，读者可以输入一个阿拉伯字母。执行结果如下：

```
old exceptions: 0
n
exception: ios::failbit set
```

7.2.4 标准输入和输出函数

能够取代标准流操作符">>"和"<<"的部分成员函数一般用于读写无格式数据。通常，在读取数据时，不跳过起始空格，对异常的处理方式也不同于格式化 IO 函数；发生意外（异常）后，badbit 通常会设立。若异常掩码中包含 badbit，会重新抛出该异常，和格式化函数一样，无格式相关函数会产生一个 sentry（岗哨）对象。本节主要讲述输入用成员函数和输出用成员函数。

1. 输入函数

Istream 是用于输入的一个流类。还有 wistream 类和模板类 basic_istream。C++ STL 中用于读取字符序列的成员函数主要包括：get(s,num)，get(s,num,t)，getline(s,num)，getline(s,num,t)，read (s,num)，readsome (s,num)。

输入运算符">>"有一个限制，前面的空格会自动跳过。当某行输入字符时，只有那些非空字符进入接收字符的变量中，前面的空格会被越过。数据输入时，如果是在非起始位置出现空格字符，输入将会停止。之后的文字或字符会被读取到下一个存储流的对象中，并且中间的空格也会消失。

命名空间 std 中的 ios::skipws 标记是不能对">>"运算符起作用的。如果要保留输入序列中的空格，需要使用函数 get()或 getline()。

命名空间 istream 类的 read()成员函数是输入函数，主要用于和输出函数 write()对应。输入的二

进制数据存入缓冲区中，并且会包括数据中的空格。通常会用于文件的读写操作中。

函数 get()、getline()、read()和 readsome()的函数原型分别为：

（1）函数 get()的函数原型

```
int get();&
istream& get( char* pch, int nCount, char delim = '\n' );
istream& get( unsigned char* puch, int nCount, char delim = '\n' );
istream& get( signed char* psch, int nCount, char delim = '\n' );
istream& get( char& rch );
istream& get( unsigned char& ruch );
istream& get( signed char& rsch );
istream& get( streambuf& rsb, char delim = '\n' );
```

上述第 1 种形式返回从流中读入的序列中的字符。第 2 种形式返回流中读入的字符序列，读入的数据存储在字符型数组 pch 中，函数结束的条件是：已经输入了 nCount-1 个字符；遇见回车换行符‘\n’；或者遇见文件尾标志 end-of-file。第 3 种形式和第 2 种形式近似，只不过输入的数据流存储在无符号型字符数组 puch 中。第 4 种形式和第 2 种型式近似，只不过输入的数据流存储在有符号型字符数组 psch 中。

第 5 种形式表示从输入流中提取单个字符，并存储在 rch 中；第 6 种形式和第 5 种形式近似，表示从输入流中提取单个字符，并存储在无符号字符型变量 ruch 中；第 7 种形式表示从输入流中抽取单个字符，存储在有符号字符变量 rsch 中；第 8 种形式表示从输入流中抽取字符序列，并存储在 streambuf 型对象中，抽取时遇上指定的分隔符或遇见文件结束标志 eof 时，抽取过程终止。

（2）函数 getline()的函数原型

```
istream& getline( char* pch, int nCount, char delim = '\n' );
istream& getline( unsigned char* puch, int nCount, char delim = '\n' );
istream& getline( signed char* psch, int nCount, char delim = '\n' );
```

上述 3 种形式中，第 1 种形式 getline()实现从输入流中抽取字符序列，并保存在字符型数组 pch 中。抽取过程的终止条件是：已经输入了 nCount-1 个字符；遇见回车换行符 ‘\n’；或者遇见文件尾标志 end-of-file。

第 2 种形式和第 3 种形式均和第 1 种形式近似。

（3）read()函数

```
istream& read( char* pch, int nCount );
istream& read( unsigned char* puch, int nCount );
istream& read( signed char* psch, int nCount );
```

以上给出 3 种型式的 read()函数原型。第 1 种形式实现读取 nCount 个字符，并存储在字符串 pch 中，返回数据流，数据流的状态可表明读取工作是否成功。

需要注意：在读入过程中，字符串 pch 中的字符不会由于字符终止符号结束。必须确保字符串 pch 的容量大于 nCount 个字符。读取过程中，如果遇见 eof 标识会出现错误，failbit 和 eofbit 会被设置。

第 2 种形式、第 3 种形式和第 1 种形式近似。

（4）readsome()函数

```
streamsize istream::readsome(char *str, streamsize count);
```

上述函数最多可读入字符个数 count，并存储在字符串 str 中。函数返回值是读取的字符个数。

str 内的字符串不会自动以字符串终止符号结束。使用时，编程人员需要确保字符串数组 str 中预留足够的空间，能够保存 count 个字符。

和 read()函数不同，readsome()会读取 streambuffer 内的所有有效字符。遇见文件末尾标志 eof 时，不会报错，并且不设置标志位 eofbit 和 failbit。

另外还有函数 gcount()，ignore()，peek()，unget()，putback()，tellg()和 seekg()。

函数 gcount()返回上次“非格式化读取操作”所读入的字符个数。

ignore()有 3 种形式，所有形式均实现提取字符，并将这些字符舍弃不用。函数原型为：

```
basic_istream<charT, traits>& ignore(streamsize n=1, int_type delim=traits::eof())
```

在使用时，可使用以下 3 种形式：

```
ignore();
ignore(streamsize count);
ignore(streamsize count, int delim);
```

peek()函数返回输入序列中下一个被读入的字符，但并不读取该字符，如果输入序列中没有数据，函数返回 eof。函数 peek()的原型为：

```
int_type peek()
```

unget()函数将上一次读取的字符重新放回 stream 中，下一次从输入流中读取时，便可以将这些字符读取出来。

tellg()和 seekg()主要用于文件 I/O，后面会有所描述。

2．输出函数

命名空间 std 中包含两个成员函数，实现把字符和存储块写入输出流对象中，可以选择使用重载“<<”插入运算符把字符和存储块写至输出流对象中。函数 put()把单个字符写至输出流中。还有另外两个函数 write()和 flush()。函数 put() 的原型如下：

```
ostream& ostream::put(char c)
```

功能：实现将参数 c 写入流中。返回值的状态可以说明写入是否成功。

函数 write()的原型如下：

```
ostream& write( const char* pch, int nCount );
ostream& write( const unsigned char* puch, int nCount );
ostream& write( const signed char* psch, int nCount );
```

上述 3 个函数，实现功能均是：将字符串 pch（或 puch，psch）中的 nCount 个字符写至输出流中。

返回值类型为输出流类型，返回流的状态字可以说明该写入操作是否成功。值得注意的是：字符串终止符号不会终止写入操作，程序员必须确保字符串中至少包含 nCount 个字符，否则会导致不可预期的行为。

函数 flush()的原型为：

```
ostream& ostream:: flush();
```

函数 flush()的功能是：刷新 stream 的缓冲区，将所有缓冲区数据强制写入其所属的设备或 IO 通道。

另外，函数 tellg()和 seekg()用以改变写入位置，主要与文件 IO 有关。

这几个函数的应用，请模仿下面的例 7-4 和例 7-5。

例 7-4

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{     char chdim[7]={0,0,0,0,0,0,0};
  cout<<"int cin.get(): "<<endl;
  cin.get(chdim,6,'\n');                    //获取输入
  cout.write(chdim,6);                      //输出
  cout<<endl;
  cin.get();                                //获取输入
  cout<<"cin.get(char ch) : "<<endl;
  char ch;
  cin.get(ch);
  cout.put(ch);                             //输出该字符
  cout<<endl;
  cin.get();
  cout<<"cin.getline(char* ch,nCount,\\n)"<<endl;
  cin.getline(chdim,6,'\n');                //获取该行数据
  cout.write(chdim,6);                      //输出该行数据
  cout<<endl;
  char dim[20];
  fstream fio;                              //流
  fio.open(".\\test.txt",ios::in);          //打开文件
  fio.read(dim,20);                         //从文件中获取
  cout.write(dim,20);                       //输出获取的数据
  fio.close();                              //关闭数据流
  cout<<endl;
}
```

例 7-4 的执行效果见图 7-1。

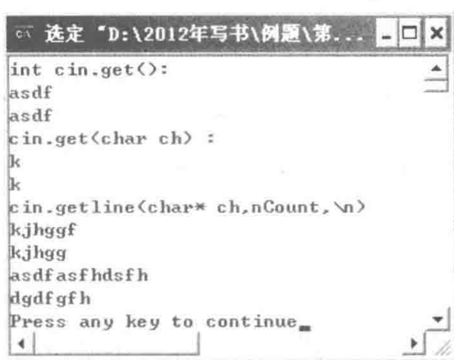


图 7-1　例 7-4 的执行效果

例 7-5

```
#include <iostream>
#include <fstream>
#include <istream>
#include <string>
```

```
using namespace std;
void main()
{
 char chardim[30];
 char ch[6];
 fstream fio;
 fio.open(".\\test5.txt",ios::in|ios::out);       //打开文件
 int cnt=10;
 int gcnt=0;
 istream i0=fio.read(chardim,cnt);                //从流中读取 cnt 个字符
 if(i0.good())
 {
   cout<<"Read success!"<<endl;
 }
 gcnt=fio.gcount();                               //统计个数
 string buf;
 buf.assign(chardim,10);                          //存入字符串
 cout<<"The input chars from file test5.txt are: "<<buf<<endl;
 cout<<"The count of the last read function: "<<gcnt<<endl;
 istream is=fio.ignore(3,EOF);                    //忽略
 buf.erase(buf.begin(),buf.end());                //清除
 i0=fio.read(ch,5);                               //读字符
 if(i0.good())
 {
    cout<<"Read success!"<<endl;
 }
 buf.assign(ch,5);
    cout<<"Secondly The input chars from file test5.txt are: "<<buf<<endl;
 char chc=fio.peek();
 if(chc==EOF)
 {
    cout<<"file eof reach."<<endl;
 }
 else
 {
   cout<<"some char has existed in file."<<endl;
 }
 i0=fio.unget();
    i0=fio.read(ch,5);
 if(i0.good())
 {
    cout<<"Read success!"<<endl;
 }
 buf.assign(ch,5);
    cout<<"Thirdly The input chars from file test5.txt are: "<<buf<<endl;
 fio.flush();
 fio.close();
}
```

例 7-5 的执行效果见图 7-2。

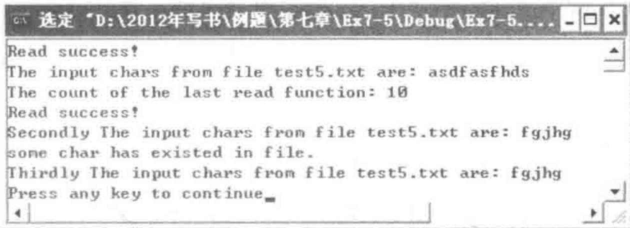


图 7-2　例 7-5 的执行效果

7.3　格式化

前面各章节涉及的输出，多数是按照默认规则将对象转为字符序列，但程序员更需要细节控制。例如：需要控制一个输出操作所用的空格数或数值输出的格式。有时候可能更需要以多种格式显式数字。C++中 IOStream 库利用操纵符、标记和成员函数支持格式化显示。

对于格式化 I/O 输出，最重要的是格式标志，可定义诸如：数字精度、充填字符、数字进制等。另一个重要的概念是根据特定国家和地区的不同人群习惯，实现格式调整的国际化。本节主要介绍格式化的问题，国际格式化的问题在后面章节专门介绍。

7.3.1　格式标志

格式标志是非常重要的。在流输入和流输出过程中，格式化是要重点掌握的。只有掌握了格式化的内容，才能随心所欲地实现输入和输出的高效和便捷。

1. 格式化标记

std::ios 类定义了部分格式化标记。这些标记具有互斥的位值，通过“或”（OR）操作，形成位掩码。函数 setiosflag()和函数 resetiosflag 操纵符，成员函数 setf()和 unsetf()，可接受这些标记和通过“或”（OR 操作）形成的掩码做实参。格式化标记通常包括 16 种：

```
std::ios::boolalpha
std::ios::hex
std::ios::internal
std::ios::left
std::ios::oct
std::ios::right
std::ios::scientific
std::ios::showbase
std::ios::showpoint
std::ios::showpos
std::ios::skipws
std::ios::stdio
std::ios::unitbuf
std::ios::uppercase
std::ios::dec
std::ios::fixed
```

2. 格式操控符

class ios_base 提供数个成员，用于定义各种 I/O 格式，实现访问格式标志的功能。这些函数包括：

setf()，unsetf()，flags()，flags(flags)和 copyfmt(stream)。这些函数可以处理所有的格式定义。函数 setf()和 unsetf()分别用于设置和清除一个或多个标志；二元操作符“or”可将多个标志合并，从而一次操控多个标志。函数 setf()以第二参数为掩码，清楚该掩码所标识的所有标志，之后设置第一参数所代表的标志。如果 setf()输入一个参数，则稍有差异。使用函数 flags()可以一次操控所有格式标志。调用无参数的 flags()会返回当前标志；传给 flags()一个参数，以该参数作为新格式标志状态，返回先前状态。输入一个参数的 flags()会清除所有标志，并将状态设置为传入的标志。函数 flags()非常有用，可用于存储当前标志状态，并在适当时机恢复。函数 copyfmt()实现从 stream 中复制所有格式定义。

setiosflags()和 resetiosflags()可以在改写语句或输入语句中设定或清理标志。使用这两个函数需要包含头文件<iomanip>。函数 setbase()、setfill()、setiosflags()、setprecision()、setw()。

下面详细介绍上述操控符。

- setbase(int base)：实现基于 base 输出整数。
- setfill()：设置作为填充字符的字符。
- setprecision()：设置浮点数的精度（即数字的个数或位数）。
- setw()：设置显示字段的宽度。

使用“<<”符号将操纵符插入输出流中，也可以使用“>>”运算符插入输入流中。典型的格式操纵符包括：

std::endl，std::ends，std::dec，std::flush，std::hex，std::oct，std::ws。其含义分别为：

- std::endl　　在输出流中插入‘\n’，刷新流
- std::ends　　插入空字符‘\0’
- std::dec　　按十进制显示整数
- std::flush　　刷新流
- std::hex　　按十六进制显示整数
- std::oct　　按八进制显示整数
- std::ws　　跳过输入流中的整数
- std::fixed　　浮点数输出

3. 格式化函数详解

std::ios 类包含很多控制流格式的函数，通过类对象调用这些函数。下面介绍部分格式化函数。

- fill()　返回当前填充字符；
- fill(int ch)　设置填充字符，返回先前填充字符
- precision()　返回当前浮点精度
- precision(int p)　设置精度 p，返回先前的精度
- setf(int m)　设置掩码 m 的 OR 表达式中的标记，返回先前的标记
- setf(int m1, int m2)　设置掩码 m1，关闭掩码 m2
- unsetf(int m)　关闭掩码 m 的 OR 表达式中的标记
- width()　放回当前的宽度
- idth(int w)　设置宽度为 w，返回先前的宽度

7.3.2　bool 类型数据的格式控制

STL 中对于 bool 类型数据的格式控制有明确的使用方法。标识 boolalpha 定义了布尔值的读写

格式：数字表示或文字表示。如果 boolalpha 被设置，输出时便以文字表示，否则输出时以数字表示。使用数字表示时，false 始终是 0，true 始终是 1。在输入时，如果遇到非“0 和 1”的数值，程序会报错，并设置 failbit。

如果此标志被设置，布尔量会以文字形式表示。读入的字符串必须为“true”或“false”。实际表述法还和 stream locale object 有关系。标准的“C”locale object 使用字符串“true”和“false”表示布尔值。STL 定义了两个操控器：boolalpha()和 noboolalpha()。Boolalpha()强制使用文字表示法，并设立标志 ios::boolalpha；noboolalpha()强制使用数字表示法，清除标志 ios::boolalpha。

举例如下：

```
bool b=0;
std::cout<<std::noboolalpha<<b<<"= ="<<std::boolalpha<<b<<std::endl;//输出逻辑变量
```

7.3.3　详解“字段宽度、充填字符和位置调整”

在 7.3.1 节讲述了各种格式标志和操控符。本小节详细讲解如何使用格式化中的字段宽度、充填字符和格式化中的位置。

1．字段宽度

函数 width()用来定义字段宽度。函数的原型包括两种，即有参数和无参数：

```
int width( int nw );
int width() const;
```

对于输出而言，width()定义了最小字段，此设置应用于下一次格式化输出。调用该函数时，如果无参数，函数返回当前字段宽度；如果传入一个整数，则会改变字段宽度并返回当前宽度。字段的最小默认值是 0。该函数的作用仅限于“下一次”格式化输出。

函数 setw()同样可以为输出的数值设置字段宽度，功能相当于 width()。函数 setw()的原型如下：

```
setw( int nw );
```

setw()可以将设置宽度命令插入输出流、输入流中，比单独调用格式化函数更加方便。并且，当需要显示的数值比设置的字段宽度大时，将显示整个数据值，这会影响后面的输出行的数据格式。

值得一提的是，在输入流中设定字段宽度时，如果使用字符数组和字符型指针，尤其是字符型指针，不能用 sizeof(char*) 来计算该字符型数组的字符数目。举例如下：

正确：

```
char  chdim[81];
cin>>setw(sizeof(chdim))>>chdim;
```

错误：

```
char* s;
cin>>setw(sizeof(s))>>s;
```

如果使用 string 类型，则不会有以上麻烦，举例如下。

正确：

```
string buf;
cin>>buffer;
```

2. 充填字段

函数 fill()定义用来充填“格式化表述”和最小字段之间的充填字符。默认的充填字符是空格符。

```
char fill( char cFill );
char fill() const;
```

函数 setfill() 同样可以定义充填字符，相当于 fill()。

```
setfill( int nFill );
```

举例如下：

```
cout<<setw(8)<<setfill('_')<<-3.14<<" "<<42<<endl;
```

3. 格式化中的位置

输入和输出中的字段位置，一般指字段的左对齐、右对齐和“符号靠左对齐，数值靠右对齐”。

标识 left 代表靠左对齐；标识 right 代表靠右对齐；标识 internal 代表符号靠左对齐，数值靠右对齐；none 是默认的靠右对齐。使用字段位置时，如果输出数值或字符所占宽度小于字段设置宽度，其余位置需要由特定字符填充。

单一字符的对齐方式在标准化过程中发生变化。标准化之前，面对单个字符，忽略字段宽度，直至下一次多字符格式化输出时才使用。在输入和输出时，设置字段位置有两种方法：setf()、unsetf()和 setiosflag()、resetiosflags()。

两个函数的原型如下：

```
long setf( long lFlags );
long setf( long lFlags, long lMask );
long unsetf( long lFlags );

smanip( long ) resetiosflags( long lFlags );
smanip( long )  setiosflags( long lFlags );
```

以上函数包含在头文件< iomanip>和<iostream>中。下面用例 7-6 来加以说明。

例 7-6

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
 bool varb=1;
 cout<<std::noboolalpha<<varb<<"=="<<std::boolalpha<<varb<<endl;
 float varf=123.4512;
 int wid=cout.width();                    //返回原有的宽度设置
 cout.width(6);                           //设置输出宽度
 cout<<varf<<endl;
 cout<<setw(7)<<varf<<endl;               //设置输出字段宽度
 cout.width(6);
 cout<<cout.fill('_')<<varf<<endl;    //使用字符"_"填充
 cout<<setw(9)<<setfill('*')<<varf<<"  "<<45<<endl;//使用字符"*"填充
 cout<<setw(9)<<setfill('
')<<setiosflags(ios::right)<<varf<<";  :"<<setw(9)<<45.234<<endl;
```

```
    double varf1=123.4512;              //
    double varf2=456.892;               //

cout<<setw(10)<<setfill(' ')<<setiosflags(ios::right)<<varf1<<";"<<resetiosflags(ios::left)<<setw(10)
    <<setfill(' ')<<setiosflags(ios::right)<<varf2<<endl; //使用空格填充，右对齐输出 varf1，varf2
    resetiosflags(ios::left);           //恢复左对齐
    cout<<varf1<<" ; "<<varf2<<endl;    //
   //注意一下输出没有将格式控制直接插入流输出中，是因为下述格式控制调用时均有返回值；
   //程序运行时，会将这些函数的返回值输出，带来不必要的麻烦
    cout.width(12);
    cout.fill('*');
    cout.setf(ios_base::right,ios_base::adjustfield);//右对齐
    cout<<varf1<<";"<<endl;
    cout<<varf2<<";"<<endl;
       cout.width(12);
    cout.fill('*');
    cout.setf(ios_base::right,ios_base::adjustfield);
       cout<<varf2<<";"<<endl;
    cout.width(12);
    cout.fill('*');
    cout.setf(ios_base::left,ios_base::adjustfield);
    cout<<varf1<<";"<<endl;
       cout.width(12);
    cout.fill('*');
    cout.setf(ios_base::left,ios_base::adjustfield);
       cout<<varf2<<";"<<endl;
}
```

例 7-6 的执行效果见图 7-3。

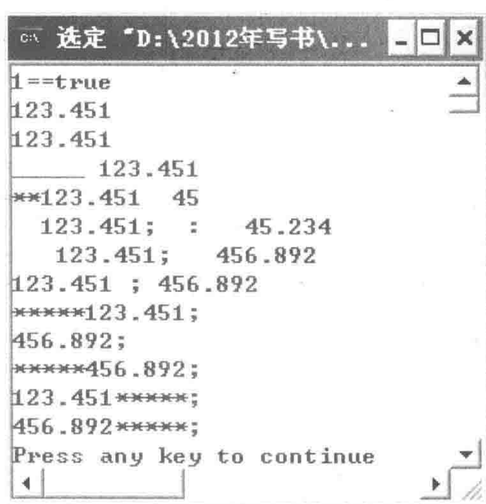


图 7-3　例 7-6 的执行效果

7.3.4　正记号与大写字符

格式标志 showpos 和 uppercase 用于改变数值的一般表述。这两个标志会影响数值的正负号和

字母大小写。标志 showpos 使用时会在正数前加上正记号；标志 uppercase 使用时会实现使用大写字符的功能，并且标志 uppercase 还可用于十六进制格式表述的整数，以及“科学计数法表述”的浮点数。通常，在输出时，字母均为小写，并省略正记号。虽然这是一个小问题，但使用时会方便读者阅读。具体的标识含义见表 7-1。

表 7-1　数值符号和大小写的操控标识符

操控标识符	意　　义
showpos	强制输出正数前的正记号（ios::showpos）
noshowpos	强制忽略正数前的正记号（ios::noshowpos）
uppercase	强制字母大写（ios::uppercase）
nouppercase	强制字母小写（ios::nouppercase）
showbase	显示数值进制（ios::showbase）
noshowbase	不显示数值进制（清除 ios::showbase）

例 7-7

```
    double var1=198534.98236;
    cout.setf(ios::showpos|ios::uppercase);          //大写
    cout<<scientific<<setprecision(4)<<setw(16)<<setfill(' ')<<setiosflags(ios_base::
right)<<var1<<endl; //
    cout.unsetf(ios::uppercase);                     //取消大写
    cout<<scientific<<setprecision(4)<<setw(16)<<setfill(' ')<<setiosflags(ios_base::
right)<<var1<<endl; //
    double var2=-198534.98236;
    cout.setf(ios::showpos|ios::uppercase);          //大写
    cout<<scientific<<setprecision(4)<<setw(16)<<setfill(' ')<<setiosflags(ios_base::
right)<<var2<<endl; //
    cout.unsetf(ios::uppercase);                     //取消大写
    cout<<scientific<<setprecision(4)<<setw(16)<<setfill(' ')<<setiosflags(ios_base::
right)<<var2<<endl; //
```

输出结果如下：

```
+1.9853E+005
+1.9853e+005
-1.9853E+005
-1.9853e+005
```

提示：虽然提供了标识符 ios::noshowpos 和 ios::nouppercase，但在 Visual C++ 6.0 环境中无法使用 setf()函数设置标识符 noshowpos 和标识符 nouppercase。所以在例 7-7 中使用了 unsetf()函数。

7.3.5　数值进制

常见的数值进制包括：八进制、十进制和十六进制。这 3 种进制分别使用 3 种标志：oct，dec，hex。标志 oct 代表是以八进制进行读/写；标志 dec 代表是以十进制进行读/写；标志 hex 代表是以十六进制进行读/写。如果使用参数 none，代表以十进制输出，在读入时要根据实际情况视起始字符而定，见表 7-2。

表 7-2　整数进制

操控标识符	意　义
oct	以八进制进行读/写
dec	以十进制进行读/写
hex	以十六进制进行读/写
none	以十进制输出，读取时视起始字符而定
showbase	显示数值进制（设置标志 ios::showbase）
noshowbase	不显示数值进制（清除 ios::showbase）

一旦进制被改变，除非重新设置相关标志，否则会持续应用于后继的整数处理过程。在默认情况下，使用十进制。输入流和输出流（IOStream）不支持二进制，使用二进制进行读/写时，可使用类 bitset，实现以二进制形式读写整数值。如果没有使用进制标志或使用了多个进制标志，输出时会采用十进制。

在输入时，进制标志有时也会产生影响。上述的进制标志被设置后，读取的数字会按该进制处理。如果没有设置进制标志，起始字符将决定进制。例如：以 0x 或 0X 起始的十六进制，以 O 起始的八进制，其余被视为十进制数字。

标志 showbase 可以 C/C++惯例显示数值进制。标志 showbase 一旦被设置，会自动显示出相应的数字进制。对于八进制会自动以 O 开头，十六进制会以 0x 或 0X 开头。

使用上述进制标志一般有两种方法：用 setf()和 unsetf()直接设置需要指定的标志；在设置标志时，同时自动清理同组的其他标志。函数 setf()和函数 unsetf()的原型如下：

```
void  setf( fmtflags _Mask);
fmtflags  setf( fmtflags _Mask,fmtflags _Unset);

void unsetf( fmtflags _Mask);
```

其中函数 setf()的第二种形式 setf(fmtflags_Mask, fmtflags _Unset)，第一个参数_Mask 是需要打开的标志符，第二个参数_Unset 是需要关闭的标志符。该形式的返回值是原有的标志符。具体应用请看下面的例 7-8。

例 7-8

```
#include <iostream>
using namespace std;
void main()
{
int vari=123458;
 cout<<showbase<<uppercase<<" vari : "<<vari<<" vari(oct): "<<oct<<vari<<" vari
(hex):"<<hex<<vari<<endl;
cout<<endl;
//以下需要分别调用各函数，而不是将函数插入输入、输出流中
cout.setf(ios::dec|ios::uppercase|ios::showbase);
cout<<" vari : "<<vari;
cout.setf(ios::oct,ios::basefield);
cout<<" vari(oct): "<<vari;
cout.setf(ios::hex,ios::basefield);                        //十六进制输出时，X 大写
cout<<" vari(hex): "<<vari<<endl;
}
```

输出结果：

```
vari : 123458  vari(oct): 0361102  vari(hex): 0X1E242
vari : 123458  vari(oct): 0361102  vari(hex): 0X1E242
```

7.3.6 浮点数输出

浮点数通过格式和精度控制进行输出。通常有以下 3 种形式的输出：

（1）一般格式，具体实现时为输出数据选一种表现形式，使之能在可用空间内以最佳形式维持这个值，精度表述了数字的最大位数，此种格式对应于 printf()的%g 输出格式。

（2）科学格式（scientific），通常用小数点前一位数字以及一个指数部分的方式表现数值，精度表述的是小数点之后的最大允许位数，和 printf()的参数%e 相关。

（3）定格格式（fixed），将值表示为一个整数部分，后面小数点及一个小数部分，精度表述的是小数点之后的最大允许位数，相当于 printf()的%f。通过状态操控函数来控制浮点数输出的格式。

需要指出的是，精度控制对于浮点数实现的是舍入操作，而不是简单的截断操作。精度控制函数 precision()不影响整数的输出。

通常涉及的 stream 标志，流（stream）成员函数实现浮点数输出的控制。和前面的讲述一致，多种标志分别为：ios::fixed，ios::scientific，ios::none。

```
ios::fixed 使用小数计数法;
ios::scientific 使用科学计数法;
ios::none 使用上述两者中最合适的。
```

默认情况下，参数 ios::fixed 和 ios::scientific 未被设置，此时输出的形式取决于实际情况。还可以通过使用标识 ios::showpoint 来强制书写小数点，并补 0 直至足够的精度为止。前面已讲述过，ios::showpos 可以用来输出“+”号，ios::uppercase 可以用来指定科学计数法中字符 e 的大写和小写。

流成员函数 precision()用于定义精度，precision()返回当前的浮点数精度，调用函数 precision(val)时，val 为新设置的浮点数精度，并返回原来的设置值。一旦使用科学计数法，precision()定义小数位数，余数是通过四舍五入，而不是被截断的。另外，函数 setprecision()也可以设置数值输出的精度，使用函数 setprecision()时，需要包含头文件<iomanip>。而且 setprecision()可以直接插入数据输出流中，而 precision()却不能直接插入输入流和输出流中。

科学计数法和定点数的输出控制方法有 3 种：使用函数 setiosflags()；直接将关键字（fixed）插入输出流（例如 cout）中；使用函数 setf()设置浮点类型，或者将标记 scientific 直接插入输出流中。具体这些数字的输出方法，请看下例 7-9。

例 7-9

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
 double vard=15.432;
 cout<<setprecision(4)<<vard<<endl;
 cout.precision(6);
 cout<<vard<<endl;
cout<<" ios::fixed usage: "<<endl;                    //浮点数输出
 cout<<setiosflags(ios::fixed)<<vard<<endl;
```

```
    cout<<" ios::scientific: "<<endl;
   cout<<" fixed usage: "<<endl;
    cout<<fixed<<vard<<endl;                       //浮点数输出
    cout<<" scientific usage: "<<endl;
    cout<<scientific<<vard<<endl;
}
```

输出结果：

```
15.43
15.432
ios::fixed usage:
15.432000
ios::scientific:
fixed usage:
15.432000
scientific usage:
1.543200e+001
```

7.3.7　一般性格式定义

通常的格式标志一般包括 skipws 和 unitbuf。skipws 格式标志，一旦被调用之后读取数值时，会自动跳过起始空格；nounitbuf 格式标志每次输出后，会清空 output 缓冲区。在默认情况下，设置 ios::skipws 意味着读取数值时会跳过起始空格，因为数字之间的空格是无意义的，不需要读取。这也导致了使用“>>”时，无法读取空字符和空格字符，它们被忽略了。

标志字 ios::unitbuf 用于控制 output 缓冲区。若 ios::unitbuf 被设立，即不使用缓冲装置，每次输出时均清空（flush）缓冲区。在默认情况下，没有设立此标志，不过 cerr 和 wcerr 已预先设置了它。

上述两个标志字均有其相反的形式，比如 noskipws 和 nounitbuf。值得注意的是，要区别它们的不同含义。skipws 代表着省略起始空格；noskipws 代表着读取输入流中的空格，不跳过起始空格，而将其读出；unitbuf 的默认状态字是 nounitbuf，状态字 unitbuf 一旦被设置，缓冲区非空时会被处理（清除）；而使用状态字 nounitbuf 时，每次输出结束后，并不清除输出缓冲区。

以上 4 种标志字的对应标志分别为：ios::skipws，ios::unitbuf。设置和清除时使用相同的标志，可以使用操纵符 setiosflags()和 resetiosflags()，还可以使用流 cout 的 setf()函数和 unsetf()函数。

例 7-10

```
resetiosflags(ios::skipws);
char strtest[10];
cin>>strtest;
cout<<strtest<<endl;
cout.unsetf(ios::unitbuf);                  //取消格式
cin>>strtest;
cout<<strtest<<endl;
```

输出结果：

```
sdfghe
sdfghe
 12345asdf
12345asdf
```

7.4 StreamBuffer 类介绍

StreamBuffer 类是比较重要的缓冲区类。本节对流缓冲区、缓冲区迭代器和自定义缓冲区均进行了详细介绍。

7.4.1 Stream 缓冲区

通常，stream 并不负责实际读写操作，而是委托给 stream buffer 实现。输出流将某些字符放入缓冲区，之后某个时刻，这些字符被写到输出设备上。此类缓冲区可被称为 Streambuf。类 streambuf 为缓冲区管理内存，并提供用于填充的缓冲区、访问缓冲区内容、刷新缓冲区和管理缓冲区内存的类方法。

在此尤其要讲述在使用 cout 输出字符时，刷新输出缓冲区。由于 ostream 类对 cout 对象处理的输出进行缓冲，输出不会立即发送到目标地址，而会被存在缓冲区中，直至缓冲区被填满。之后，程序通过刷新缓冲区，将其中的内容发送出去，并清空缓冲区，以存储新的数据。通常，缓冲区的大小为 512 字节或其整数倍。当标准输出是硬盘上的连接的文件时，缓冲可以节省大量时间。程序员不希望程序为发送 512 个字节，而存取磁盘 512 次。将 512 个字节收集到缓冲区中，之后一次性将其写入硬盘会更有效率。

对于屏幕输出，首先填充缓冲区的重要性要低得多。当重新表述消息“Press any key to continue”以便使用 512 个字节填充缓冲区时，非常不方便。幸运的是，屏幕输出时，程序没必要等待缓冲区被填充。例如换行符发送至缓冲区之后，缓冲区即被刷新。而多数的 C++语言，是在输入即将发生时刷新缓冲区。控制符 flush 和 endl 均可实现刷新缓冲区，而控制符 endl 在刷新缓冲区时，还将插入一个换行符。

对于程序员和程序开发者来说，类 basic_streambuf 仅仅是发送（sent）或提取（extracted）字符的地方。通常有两个公共函数，用于写入字符，它们是 sputc ()和 sputn()。函数 sputc()调用发生错误时，会返回 traits_type::eof()，之后 traits_type 是类中的型别定义。函数 sputn()将写入由第二参数指定的字符数，除非 stream 缓冲区无法使用或消耗该字符。这两个函数使用时，不考虑字符串终止符号，返回的是实际写出的字符数。

访问 streambuf 缓冲区的接口非常复杂。对于输入而言，必须时刻监视未耗用的字符。解析时，字符最好能被送回 stream 缓冲区。为此，stream buffer 类特别提供了相应的函数，见表 7-3。

表 7-3　stream buffer 类的公用成员函数（输入）

输入成员函数	含　　义
in_avail()	返回有效字符的下界
sgetc()	返回当前字符，并不耗用它
sbumpc()	返回当前字符，并耗用它
snextc()	耗用当前字符并返回下一个字符
sgetn(b,n)	读取 n 个字符，并将其存储到缓冲区中
sputbackc(c)	将指定字符 c 返回 stream 缓冲区中
sungetc()	退回至前一字符

函数 in_avail()用于确定缓冲区中至少有多少个有效字符，并确定从键盘读取数据时不会阻塞。值得注意的是“至少”，它表示实际上可能包含更多的有效字符。

函数 sgetc()不必移动至下一字符即可获得当前字符。函数 sbumpc()读取当前字符并移动至下一个字符，并使之成为当前字符。函数 snextc()将下一个字符视为当前字符，之后读取。这 3 个函数如果调用失败，均会返回 traits_type::eof()。函数 sgetn()读取字符序列并发送至缓冲区中，其参数可以代表欲读取的字符数，返回值是实际读取的字符数目。

函数 sputbackc()和 sungetc()被用于后退一步，前一个字符成为当前字符。函数 sputbackc()可将前一字符替换为其他字符。使用这两个函数，只能后退一个字符。还有部分函数用于存取局部对象、改变位置或影响缓冲区，见表 7-4。

表 7-4　难以分类的 stream buffer 类公用函数

输入成员函数	含　义
pubimbue(loc)	为 stream 缓冲区安装 locale (loc)
getloc()	返回当前的 locale
pubseekpos(pos)	将当前位置重新设定为某绝对位置
pubseekpos(pos, which)	将当前位置重新设定为某绝对位置，并可指定 IO 方向
pubseekoff(offset,rpos)	将当前位置重新设定为另一位置的相对位置
pubseekoff(offset,rpos,which)	将当前位置重新设定为另一位置的相对位置，并可指定 IO 方向
pubsetbuf(b,n)	影响缓冲行为

pubimbue()和 getloc()用于国际化议题。pubimbue()在 stream 缓冲区中安装一个新的 locale 对象，并返回前一个被安装的 locale 对象：getloc()返回当前的 locale 对象。

函数 pubsetbuf()试图对缓冲区的缓冲策略进行控制，是否有效则取决于具体的 streambuf 类。例如，对于 string stream 缓冲区运用 pubsetbuf()将毫无意义。即使用于 file stream 缓冲区，只能在第一个 I/O 操作之后以 pubsetbuf(0,0)方式调用才起作用。如果调用出错，函数返回 0，否则返回该 stream 缓冲区。

函数 pubseekoff()和 pubseekpos()控制读/写操作的当前位置，无论控制读或写，都取决于最后一个参数（型别为 ios_base::openmode），如果没有特别指定，参数默认值为 ios_base::in|ios_base::out。一旦设置 ios_base::in，读取位置会跟着改变，一旦设置 ios_base::out，改写位置会跟着变化。函数 pubseekpos()会把 stream 当前位置移动至其第一个参数指示的绝对位置上，函数 pubseekoff()会把 stream 当前位置移动至相对位置，偏移量由第一个参数决定，起始位置由第二个参数决定，标志字一般是：ios_base::cur、ios_base::beg 或 ios_base::end。两个函数均返回 stream 所在位置或无效位置，将函数结果拿来和对象 pos_type 比较，检查出无效的 stream 位置。如果要获取 stream 的当前位置，需要使用函数 pubseekoff()。即：

```
streambuffer sbuf;
sbuf.pubseekoff(0, std::ios::cur)
```

7.4.2 缓冲区迭代器

流缓冲区和其他任意的存储空间一样，可以使用迭代器进行访问。使用流缓冲区的迭代器类，其实也是使用 stream 成员函数的一种形式。streambuffer 类提供的迭代器：（1）符合输入迭代器和输出迭代器的规格和要求；（2）从 stream 缓冲区读取或写入单个字符时，流缓冲区的迭代器将字符层面的输入与输出，归纳入 C++标准程序库的算法管辖范围内。

模板类 istreambuf_iterator 和 ostreambuf_iterator 用于从型别为 basic_streambuf 的对象中读取或写入单个字符。使用上述迭代器需要包含头文件<iterator>。流缓冲区迭代器是流迭代器的特殊形式，它与流迭代器的唯一区别在于元素是字符。

1. Output Stream 缓冲区迭代器

举例如下：

```
std::ostreambuf_iterator <char> bufwrite(std::cout);
std::string hello("hello, world\n");
std::copy(hello.begin(), hello.end(),bufwriter);
```

第一行代码根据 cout 构造一个输出迭代器，型别为 ostreambuf_iterator。除了传递 output stream 之外，可以直接传递指针，指向 stream 缓冲区。第二行和第三行代码分别实现定义一个字符串和将之复制到输出缓冲区的功能，copy()函数执行之后，字符串 string 随之被输出到屏幕上。

输出缓冲区迭代器的操作函数和输出流的迭代器近似（见表 7-5）。使用一个缓冲区将迭代器初始化，调用 fail()检查迭代器能否用于输出。如果任何一个字符的预写入操作失败，函数 fail()会返回 true。此时，运算符 operator 进行的任何改写操作均无效。

表 7-5　输出缓冲区迭代器的各项功能

输入成员函数	含　　义
ostreambuf_iterator<char>(ostream)	为流 ostream 产生一个输出流缓冲区迭代器
ostreambuf_iterator<char>(buffer_ptr)	为 buffer_ptr 所指的缓冲区产生一个输出流缓冲区迭代器
*iter	无操作，返回 iter
iter=c	调用 sputc(c)，对缓冲区写入字符 c
++iter	无操作，返回 iter
iter++	无操作，返回 iter
failed()	判断输出流迭代器是否能执行改写操作

2. Input Stream 缓冲区迭代器

输入流缓冲区迭代器的所有操作函数和类 istream 迭代器的所有操作函数近似（见表 7-6）。通常，成员函数 equal()用于判断两个输入流缓冲区迭代器是否相等。当两个流缓冲区迭代器都是或都不是 end-of-stream 迭代器时，两者均视为相等。

表 7-6　输入流缓冲区迭代器的各项操作

输入成员函数	含　　义
istreambuf_iterator<char>()	产生一个 end-of-stream 迭代器
istreambuf_iterator<char>(istream)	为 istream 建立一个 input stream 缓冲区迭代器，并可能调用 sgetc()读取第一个字符
Istreambuf_iterator<char>(buffer_ptr)	为 buffer_ptr 所指向的输入流产生一个输入流缓冲区迭代器，并调用 sgetc()读取第一个字符
*iter	返回当前字符，即是先前以 sgetc()读取的字符（如果构造函数未执行读取操作，在此执行）
++iter	以 sbumpc()读取下一个字符，并返回其位置
iter++	以 sbumpc()读取下一个字符，返回一个迭代器
equal ( )	判断两个迭代器是否相等
= =	判断两个迭代器是否相等
! =	判断两个迭代器是否不相等

有两点需要注意：

（1）从流当前位置到流尾部之间的范围，由两个迭代器定义出来：istreambuf_iterator<charT, traits>(stream)和 istreambuf_iterator<charT, traits>()，其中 stream 的型别是 basic_istream<charT, traits>或 basic_strrambuf<charT, traits>。

（2）不可能以 istreambuf_iterators 建立出一个子序列。

下面给出一个缓冲区迭代器的实例。实现功能为：使用流缓冲区迭代器简单地输出所有读取到的字符见例 7-11。

例 7-11

```
#include <iostream>
#include <algorithm>
using namespace std;
void main()
{
 ostreambuf_iterator<char> bufwrite(cout);
 string hello("Hello, world!.\n");
 copy(hello.begin(),hello.end(),bufwrite);          //复制字符串至缓冲区
 istreambuf_iterator<char> inpos(cin);
 istreambuf_iterator<char> endpos;
 ostreambuf_iterator<char> outpos(cout);
 while(inpos!=endpos)
 {
    *outpos=*inpos;
    ++inpos;
    ++outpos;
 }
}
```

输出结果如下：

```
Hello, world!.
asdfghjk,lmn
asdfghjk,lmn
```

7.4.3　自定义缓冲区

流缓冲区是一种 I/O 缓冲区，接口由类 basic_streambuf 定义。针对字符型别 char 和 wchar，标准程序库分别提供预先定义好的流缓冲区（streambuf）和宽字符流缓冲区（wstreambuf）。尤其在特殊通道上通信时，各类可以当作基类，要实现这一点儿，必须对流缓冲区的操作有所了解。缓冲区的主要接口由 3 个指针构成。函数 eback()、gptr()、egptr()返回的指针构成了 read(input)缓冲区的界面。函数 pbase()、pptr()、epptr()返回的指针构成了 write(output)缓冲区的接口。这些指针分别由 I/O 操作操控，后者会导致相关 I/O 通道上的相关响应。精确操作将分为读取和写入。

尤其对于程序开发者而言，实现自定义输出缓冲区是很重要的。

输出缓冲区一般由 3 个指针维护，分别由函数 pbase()、pptr()、epptr()获得。它们表示的意义如下：

（1）pbase()是指 output stream 缓冲区的起始位置；

（2）pptr()是当前写入位置；

（3）epptr()是 output 缓冲区的结尾，指向“被缓冲的最后一个字符”的下一个位置。

pbase()至 pptr()之间的序列字符已被写至相应的输出通道，但尚未清空。

成员函数 sputc()可以写入一个字符。如果当时有个空的改写位置，字符就会被复制到该位置上。之后，指向当前改写位置的那个指针会加 1。如果缓冲区是空的，就调用虚函数 overflow()将 output 缓冲区的内容发送至对应的输出通道中。这个函数可能有效地将字符送至某种“外部表述”。基类 basic_streambuf 所实例化的 overflow()函数只返回 end-of-file()，表示没有更多字符被写入。

成员函数 sputn()可以用来一次写入多个字符。该函数把实际任务委派给虚函数 xsputn()。后者可针对“多个字符”做出更有效的操作。类 basic_streambuf 中的 xsputn()对每个字符调用 sputn()，会改写 xsputn()为不是必要的。通常，“同时写入多个字符”会比“一次写入一个字符”效率高得多，所以 sputn()可以用于优化对字符序列的处理。

同时，广大程序员也应该认识到：对于一个 stream 缓冲区，写入数据时不必一定要采取缓冲行为，而可以令字符即刻发送。此时，默认构造函数会自动维护“输出缓冲区”的指针设置为 0 或 NULL。

1. 函数 overflow()详解

通过上述描述，给出以下实例。例中并未采取缓冲行为，对每个字符都调用 overflow()。

简要介绍一下虚函数 overflow()。Visual C++软件开发环境的 basic_streambuf 类的保护（protected）成员虚函数 overflow()的说明中提到：overflow()是保护型虚成员函数，每当有新的字符被插入空的缓冲区内时，函数 overflow()即被调用。函数的默认返回值是 traits_type::eof。如果函数调用失败，将返回 traits_type::eof 或抛出一个异常；否则，函数返回值为 traits_type::not_eof()。

函数的原型如下：

```
virtual int_type overflow(int_type _Meta = traits_type::eof( ));
```

参数_Meta 并不和 traits_type::eof 相等，在函数被调用时，将努力插入字符型变量_Meta，因为参数是 int_type 型，执行过程中被插入输出缓冲区中的是进行了数值类型转换的参数，即 traits_type::to_char_type()。

以上功能是通过多种方式实现的：

- 如果“写”的位置有效，元素被存储至写的位置，并增加输出缓冲区的下一个指针；
- 通过分配新的或额定的存储空间至输出缓冲区，以确保每个“写”的位置均有效；
- 通过“写出”至一些外部目标地址，部分或所有的元素均保证在输出缓冲区的起始位置和下一个指针之间。

虚函数 overflow()和 sync()、underflow()，定义了输出缓冲区类的特性。每个类会采用不同的方式执行 overflow()函数，但调用流类的接口是相同的。函数 overflow()是最频繁调用的函数之一，通常被流缓冲区类（streambuf）的函数调用，例如 sputc()和 sputn()。其他的流类也随时调用 overfow()函数。

函数 overflow()耗用输出区域中指针 pbase 和指针 pptr 之间的字符，并重新初始化这些区域。函数 overflow()必须也耗用字符 nCh，或者它可能选择将字符放置入新的输出区域，以至在下一次调用时被耗费。

在不同的派生类中，耗用（consume）的定义是多种多样的。例如，文件缓冲区类 filebuf“写字符”到文件时，类 strstreambuf 保持这些字符在缓冲区中，并响应 overflow()的调用扩张缓冲区。通过释放旧的缓冲区，并使用新的、更大的缓冲区取代旧的缓冲区，实现扩张内存的目的。同时，指针的调整也是必需的。

鉴于以上阐述，在实现自定义流缓冲区类时，一定要派生出保护类型虚成员函数 overflow()。

2. 自定义输出缓冲区最简单的实例

下面实现一个最简单的自定义缓冲区实例，见例 7-12。此例题摘自《C++标准程序库（侯捷译）》一书。

例 7-12

```
#include <iostream>
#include <streambuf>
#include <locale>
#include <cstdio>
using namespace std;
class outbuf: public std::streambuf
{
protected:
 virtual int_type overflow(int_type c)
 {
       if(c!=EOF)
       {
           c=std::toupper(c,getloc());
           if(putchar(c)==EOF)
           {
               return EOF;
           }
       }
       return c;
 }
};
void main()
{
 outbuf ob;
 std::ostream out(&ob);                         //自定义输出缓冲区
 int num=56;
 out<<"56 十六进制数值: "<<std::hex<<std::showbase<<num<<endl;
}
```

输出结果：

提示： 函数 getloc()是获取基本流缓冲区类对象的场合（或场景）。

```
56 十六进制数值: 0X38
```

对于较复杂的自定义输出，其他任意目标的写入操作（例如文件、socket 连接名或流缓冲区），若向目标端改写数据，只需实例化相应的 overflow()即可。有必要时也需要实例化 xsputn()函数，有助于提高效率。为方便地构造流（stream）缓冲区，实例化特殊类，用于将构造函数参数传递给相应的流（stream）缓冲区。例 7-13 中使用函数 fprintf()写入文件，使用 ostream 类的继承类，用以维护流缓冲区。并且，如果需要实例具备缓冲能力的流缓冲区，“写”缓冲区需要使用函数 setp()初始化，并使用函数 sync()实现同步。

函数 setp()的原型如下：

```
void setp(char_type *_Pbeg, char_type *_Pend);
```

函数的作用是保存参数_Pbeg 和_Pend 限定的输出缓冲区内容。

函数 sync()的原型：

```
int sync()
```

函数的作用是实现控制流和外部关联流（设备）之间的同步。

这些函数的应用，我们用例 7-13 来加以说明。

例 7-13

```
#include <iostream>
#include <io.h>
#include <streambuf>
#include <cstdio>
using namespace std;
static const int bufferSize=10;
class outbuf:public std::streambuf{
protected:
char buffer[bufferSize];
public:
 outbuf()
 {
       setp(buffer,buffer+(bufferSize-1));     //输出
 }
 virtual ~outbuf()
 {
       sync();
 }
protected:
 int flushBuffer()
 {
       int num=pptr()-pbase();
       if(write(1,buffer,num)!=num)
       {
          return EOF;
       }
       pbump(-num);
       return num;
 }
 virtual int_type overflow(int_type c)
 {
       if(c!=EOF){
          *pptr()=c;
          pbump(1);
       }
       if(flushBuffer()==EOF)
       {
          return EOF;
       }
       return c;
 }
 virtual int sync()
 {
       if(flushBuffer()==EOF)
       {
          return -1;
       }
       return 0;
```

```
    }
};
class fdostream: public std::ostream{
protected:
 outbuf buf;
public:
 fdostream(int fd):ostream(0)
 {
       rdbuf(&buf);
 }
};
void main()
{
   fdostream out(1);                                    //设置输出缓冲区
   out<<"51 hexadecimal: "<<std::hex<<std::showbase<<51<<endl;
}
```

程序的执行结果如下：

```
51 hexadecimal: 0x33
```

3. 自定义输入缓冲区

输入和输出的机制基本相同。对输入而言，可能不进行最后的读取操作。函数 sungetc()或 sputbackc()可用于存储 stream（流）缓冲区最后一次读取前的状态。可能需要读取下一字符但并不移动读取位置。实例化“从 stream 缓冲区中读取”的操作和实例化“向 stream 缓冲区写入数据”的操作相比，必须改写（重载）更多的函数。

同样，stream 缓冲区以 3 个指针维护一个 read 缓冲区，这些指针通过成员函数 eback()、gptr()和 egptr()获取。

（1）eback()指的是 input 缓冲区的起始位置，或者回退区的尾端。如果不采取特殊措施，字符最多只能被回退到这个位置；

（2）gptr()指的是当前的“读取位置”；

（3）egptr()指的是 input 缓冲区的尾端。

读取位置和结束位置之间的字符已经从外部表述装置被传至程序内存，但仍等待着程序的处理。函数 sgetc()或 sbumpc()可以读取单一字符。两者的差别在于后者会令“读取指针”前进，前者不会。如果缓冲区读取完毕，就不再有可用字符了。缓冲区必须重新获得补给。这项工作可由虚函数 underflow()完成，函数 underflow()负责读取数据。如果没有可用字符，函数 sbumpc()会调用虚函数 uflow()，而 uflow()的默认行为即调用 underflow()，移动（前进）“读取指针”。基类 basic_streambuf 中对 underflow()的默认做法是令它返回 EOF，意味着不可能以此默认版本读取字符。

函数 sgetn()用于一次读取多个字符。这个函数把任务委派给虚函数 xsgetn()，后者的默认做法是简单地对每个字符调用 sbumpc()。像重载函数 xsputn()一样，通过改进 xsgetn()实现优化多个字符的读取过程。

输入和输出还有不同，对于输入而言，仅仅改写一个函数是不够的，必须建立缓冲区，至少实例化函数 underflow()和 uflow()。因为 underflow()不会将“读取指针”移动到当前字符之后，可以被 sgetc()调用，移至下一字符，需以缓冲区操作函数或 uflow()完成。任何一个具备字符读取功能的 stream 缓冲区，必须实例出 underflow()。若 underflow()和 uflow()都实例化了，就没必要建立缓冲区了。

成员函数 setg()可以建立一个 read()缓冲区。该函数有 3 个参数，依次为：

（1）一个指针，指向缓冲区头部（eback()）；

（2）一个指针，指向当前读取位置（gptr()）；

（3）一个指针，指向缓冲区尾部（egptr()）。

和 setp()不同的是，setg()包括 3 个参数，这是必需的。它能定义出用来储存“将被回退给 stream”的字符空间。因此，一旦指向 read 缓冲区的指针已经被设定好，会有部分字符被读取但仍存放在缓冲区内。运用 sputbackc()和 sungetc()可将字符回退到 read 缓冲区中，sputbackc()可以将回退字符作为参数，并确保该字符确实是被读取的字符。如果可能，这两个函数会将读取指针退回一步。只有在读取指针不指向 read 缓冲区头部的时候，才能实现。如果到达缓冲区头部，试图退一字符，虚函数 pbackfail()会被调用。通过改写函数，可以实例出“即使在这种情况下也能恢复原读取位置”的机制。基类 basic_streambuf 并未定义相应操作，实际上不可能回退任意字符。如果不使用缓冲区的 streams，函数 pbackfail()应该被实例化出来，这些 streams 通常假设至少有一个字符可以被回退。若新缓冲区只用于读取，会产生新的问题：即缓冲区没有将旧数据保存下来，连一个字符也无法回退。实例化 undereflow() 时经常把当前缓冲区的最后数个字符移到头部，之后才添加新读取的字符。允许在调用 pbackfail()之前回退一些字符。下面用例 7-14 来加以说明。

例 7-14

```
#include <iostream>
#include <streambuf>
#include <cstring>
#include <io.h>
using namespace std;
static const int bufferSize=10;
class inbuf:public std::streambuf
{
protected:
    char buffer[bufferSize];
public:
 inbuf()
 {
       setg(buffer+4,buffer+4,buffer+4);
 }
protected:
 virtual int_type underflow(){
       if(gptr()<egptr())                        //未至文件末尾
       {
          return *gptr();
       }
       int numputback;
       numputback=gptr()-eback();               //当前位置至文件开头的距离
       if(numputback>4){
          numputback=4;
       }
       memcpy(buffer+(4-numputback),gptr()-numputback,numputback);
       int num;
       num=read(0,buffer+4,bufferSize-4);
       if(num<=0){
```

```
            return EOF;
        }
        setg(buffer+(4-numputback),buffer+4,buffer+4+num);
        return *gptr();
    }
};
void main()
{
    inbuf ib;
    std::istream in(&ib);                               //自定义输入缓冲区
    char c;
    for(int i=1;i<=20;i++)
    {
        in.get(c);
        cout<<c<<flush;
        if(i==8)
        {
            in.unget();
            in.unget();
        }
    }
    cout<<endl;
}
```

输出结果如下：

```
asdfghj
asdfghj
j
```

总结：请读者认真阅读上述几个例题，体会如何使用自定义输出缓冲区和自定义输入缓冲区取代 cout 和 cin。认真体会各函数的用法。

7.5　基于字符串的流

在 STL 的头文件<sstream>中定义了 4 个类模板和 6 个类型，均是和 basic_string 相关的流缓冲区类型。其功能主要是实现将流附着在 string 上，即通过流所提供的格式化功能，从 string 中读取或写入 string。同时，这样的流也被命名为 stringstream。在头文件中的定义如下：

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem>
> class basic_stringstream :
    public basic_iostream<Elem, Tr>
```

7.5.1　streambuf 类

类模板 basic_stringbuf 的定义形式如下：

```
template <class Elem, class Tr = char_traits<Elem>, class Alloc = allocator<Elem> > class
basic_stringbuf : public basic_streambuf<Elem, Tr>
```

其中包含 5 个公用类型定义：

```
public:
```

```
typedef   charT char_type;
typedef   typename  traits::int_type   int_type;
typedef   typename  traits::pos_type  pos_type;
typedef   typename  traits::off_type   off_type;
typedef traits  traits_type;
```

两个构造函数：

```
explicit basic_stringbuf(ios_base::openmode which =ios_base::in|ios_base::out);
explicit basic_stringbuf(const basic_string<charT, traits, Allocator>& str, ios_base::
openmode which=ios_base::in|ios_base::out);
```

一个 get()函数和一个 set()函数：

```
basic_string<charT, traits, Allocator> str() const;
void str(const basic_string<charT,traits,Allocator>&s);
```

重载了 6 个虚函数：

```
virtual int_type underflow();
virtual int_type pbackfail(int_type c=traits::eof());
virtual int_type overflow(int_type c=traits::eof());
virtual basic_streambuf<charT,traits>*setbuf(charT*, streamsize);
virtual  pos_type  seekoff(off_type  off,  ios_base::seekdir  way,  ios_base::openmode  which
=ios_base::in
 |ios_base::out);
virtual pos_type seekpos(pos_type sp, ios_base::openmode which =ios_base::in|ios_base::out);
```

下面对其各成员函数和构造函数进行详细说明。

两个构造函数的主要功能是构造基类 basic_streambuf 的对象，并初始化基类，使用参数 which 初始化模式。

```
explicit basic_stringbuf(ios_base::openmode which =ios_base::in|ios_base::out);
explicit   basic_stringbuf(const   basic_string<charT,   traits,   Allocator>&   str,
ios_base::openmode which=ios_base::in|ios_base::out);
```

成员函数 str()的功能是获取字符串缓冲区中的内容，或者初始化输入/输出序列。

虚函数 underflow()将参数指定的字符放入输入缓冲区，如果可能的话，需要 3 种方法：

（1）如果输入序列的当前位置有效，并且 if traits::eq_int_type(c,traits::eof()) 返回 false，且 traits::eq(to_char_type(c),gptr()[-1]) returns true , assigns gptr()-1 to gptr()，上述条件均满足时，函数返回字符 c。

（2）如果 traits::eq_int_type(c, traits::eof()) 返回 false，并且输入序列的当前位置有效，模式为输出（ios::out）时，指定字符 c 给*(--gp())，函数返回 traits::not_eof(c)。

总结： 函数 traits::eof()表明函数调用失败。

成员函数 overflow()的功能是将参数指定的字符 c 送入输出缓冲区，如果可能的话，有两种办法：

（1）当 traits::eq_int_type(c,traits::eof())返回 false，并输出缓冲区当前位置有效，函数会调用函数 sputc(c)，如果调用成功，函数返回字符 c。

（2）如果 traits::eq_int_type(c,traits::eof())返回 true，且没有字符被添加(append)，函数调用成功时，返回一个数值，调用失败时，返回 traits::eof()。

成员函数 seekoff()的功能是在一个可控序列中，改变流的位置。函数 seekpos()的功能和 seekoff()近似。

7.5.2　类模板 basic_istringstream

类模板 basic_istringstream 支持读取 basic_string 类的对象。它使用 basic_stringbuf 类对象控制相关的存储区域。

类模板 basic_istringstream 的定义形式如下：

```
    template<class charT, class traits=char_traits<charT>, class Allocator=allocator
<charT>>
    class basic_istringstream: public basic_istream<charT,traits>
```

包括 5 个公用类型定义：

```
    typedef  charT  char_type;
    typedef  typename  traits::int_type   int_type;
    typedef  typename  traits::pos_type  pos_type;
    typedef  typename  traits::off_type   off_type;
    typedef  traits  traits_type;
```

两个构造函数：

```
    explicit basic_istringstream (ios_base::openmode which = ios_base::in);
    explicit basic_ istringstream(const basic_string<charT, traits, Allocator>& str,
ios_base::openmode which=ios_base::in);
```

3 个成员函数：

```
    basic_stringbuf(charT, traits, Allocator>)* rdbuf() const;
    basic_string<charT, traits,Allocator>str() const;
    void str(const basic_string<charT, traits,Allocator>&s);
```

其两个构造函数主要用于构造类的对象及其初始化缓冲区。

成员函数 rdbuf()获取缓冲区内的指针，函数 str()可以获取或指定缓冲区的内容。

7.5.3　类模板 basic_ostringstream

类 basic_ostringstream 支持写入 basic_string 类型对象。它使用 basic_stringbuf 型缓冲区对象控制相关的内存区域。为方便下面描述，假定标志 sb 代表字符串缓冲区对象（stringbuf object）。类模板 basic_ostringstream 的定义如下：

```
    template<class charT, class traits=char_traits<charT>, class Allocator=allocator
<charT>>
    class basic_ostringstream : public basic_ostream<charT, traits>
```

包括 5 个公用类型定义：

```
    typedef  charT  char_type;
    typedef  typename  traits::int_type   int_type;
    typedef  typename  traits::pos_type  pos_type;
    typedef  typename  traits::off_type   off_type;
    typedef  traits  traits_type;
```

两个构造函数：

```
    explicit basic_ostringstream(ios_base::openmode which =ios_base::out);
    explicit basic_ostringstream(const basic_string<charT, traits, Allocactor>& str,
ios_base::openmode
```

```
which =ios_base::out);
```

3 个成员函数：

```
basic_stringbuf<charT, traits, Allocator>*rdbuf() const;
basic_string<charT, traits, Allocator> str() const;
void str(const basic_string<charT,traits, Allocator>&s);
```

构造函数实现构造类的对象，并初始化缓冲区。

成员函数 rdbuf()获取缓冲区的指针。

函数 str()既可以获取缓冲区中的内容，也可以指定缓冲区中的内容。

7.5.4 类模板 basic_stringstream

类 basic_stringstream 既可以支持读取 basic_string 类对象，也可以支持写入 basic_string 类对象。同样，该类使用缓冲区 basic_stringbuf 类对象控制相关联的序列。类模板的定义如下：

```
    template<class charT, class traits=char traits<charT>,class Allocator=allocator
<charT>>class basic
    stringstream : public basic_iostream,traits>
```

包括 5 个公用类型定义：

```
typedef  charT  char_type;
typedef  typename  traits::int_type   int_type;
typedef  typename  traits::pos_type  pos_type;
typedef  typename  traits::off_type   off_type;
typedef  traits                         traits_type;
```

两个构造函数：

```
    explicit basic_stringstream(ios_base::openmode which =ios_base::out|ios_base::in);
    explicit  basic_stringstream(const  basic_string<charT,  traits,Allocator>&  str,
ios_base::openmode which =ios_base::out|ios_base::in);
```

两个构造函数实现构造类的对象并初始化缓冲区。

成员函数 rdbuf()获取缓冲区的指针；成员函数 str()获取缓冲区中的内容或设置缓冲区中的内容。

7.6 基于文件的流

stream 可用来存取文件。C++标准程序库提供 4 个 template classes，预先定义 4 个标准特化版本。这 4 个标准特化版本分别是：

（1）template class basic_ifstream<>及其特化版本 ifstream 和 wifstream，用于读取文件；

（2）template class basic_ofstream<>及其特化版本 ofstream 和 wofstream，用于将数据写入文件；

（3）template class basic_fstream<>及其特化版本 fstream 和 wfstream，用于读写文件；

（4）template class basic_filebuf<>及其特化版本 filebuf 和 wfilebuf，被其他的 file stream classes 用于进行实际字符的读写工作。

以上这些类的使用，均需要包含头文件<fstream>。和 C 语言的文件存取机制相比，C++文件流类的最大好处是文件的自动管理。文件在构造时自动打开，析构时自动关闭。这都是因为使用了非常好的构造函数和析构函数。

对于既可读取亦可改写的流 stream，不允许在读写操作之间任意转换其读写属性。而有些时候

需要在开始读写过程之前转换流属性，需要进行一个 seek 操作，到达当前位置，再转换读写属性。如果已经到达文件末尾，可立即接着写入字符。

文件流对象以某个 C-string 为构造函数参数，会自动打开该字符所代表的文件，用于读和写。读或写操作是否成功，会体现在流 stream 的状态中。文件中的内容既可以单个字符地读取和输出，也可以使用文件缓冲区的指针，一次性输出其文件的内容。例如：

```
while(file.get(c))
cout.put(c);
```

或者：

```
cout<<file.rdbuf();
```

7.6.1　文件标志及其使用

文件描述符是代表文件的标志。每个流会被附着在一个开启的 I/O 通道上。文件描述符可以将文件流初始化。

文件描述符是一个整数，用于辨识被开启的 I/O 通道。尤其在 UNIX 系统中，文件描述符是属于底层接口的。并且通常 C++系统预先定义了 3 个文件描述符：

（1）0 代表标准输入通道；

（2）1 代表标准输出通道；

（3）2 代表标准错误信息通道。

这些通道可能连接文件、控制台、其他进程或 I/O 设备。

目前，C++ STL 没有提供“运用文件描述符将流 stream 附着在 I/O 通道”的可能性。并且这部分功能也不具备移植性，目前多数操作系统也不存在这样的标准接口。

1．文件简介

为了准确控制文件处理模式，类 ios_base 定义了一组标志，其型别都是 openmode，类似 fmtflags 的位掩码型别，这些标志见表 7-7。

表 7-7　ios_base 类的标志

标　志	意　义
in	打开，用于读写（ifstream 的默认模式）
out	打开，用于改写（ofstream 的默认模式）
app	写入时始终添加于尾端
ate	打开文件之后令读写位置移至文件尾端
trunc	将先前的文件内容移除
binary	不要替换特殊字符

标志字 binary 使得 stream 能够封锁特殊字符或字符序列的转换。对于以前的 MS-DOS 操作系统的文字文件，每一行结束时均是以两个字符（“回车 CR”和“换行 LF”）表示的。在正常模式下，会将以上两个字符替换为 newline（换行）字符。对于二进制模式，则不会进行上述转换。当文件是二进制时，需要使用标识 binary。当复制文件时，将源文件字符逐一读出，不做任何修改地写入目标文件。如果文件是文本类型，不需要使用 binary。此时处理换行符号时，则改为两个字符。

部分 C++版本还提供了 nocreate 和 noreplace，但这两个关键标志不是标准的内容。

多个标志可以使用操作符“|”组合起来，最终结果作为构造函数的第二参数。例如：

```
std::ofstream file("a.txt",std::ios::out|std::ios::app)
```

标志之间的并不是可以任意组合的。常见的“C++标志组合”和“C 的文件打开函数 fopen()所使用之接口字符串”间的关联见表 7-8。

表 7-8　常见的 C++标志组合

Ios_base 标志	意　　义	C 模式
in	读取（文件必须存在）	“r”
out	清空之后改写（有必要才产生）	“w”
out|trunc	清空之后改写（有必要才产生）	“w”
out|app	添加（有必要才产生）	“a”
in|out	读和写；最初位置在起始点（文件必须存在）	“r+”
in|out|trunc	先清空，再读写（有必要才产生）	“w+”

需要指出的是，无论是读文件还是写文件，在打开文件时，真正执行打开文件操作的是 stream buffer class，对于文件的实际操作才是由 stream class 实现的。

file stream 所拥有的文件，可以显式地被开启和关闭。C++ STL 定义了 3 个成员函数。这 3 个函数分别是：

```
open(name)
open(name, flags)
close()
isopen()
```

需要养成的一个好习惯是：打开文件之后，完成必要的操作并使用完毕之后，在关闭文件之前，一定要使用 clear()函数将设置于文件尾端的状态标志清除。因为在 C++标准中，文件是可以被多个用户共享的。

2. 简单的文件 I/O

使用程序实现文件的写入，需要以下步骤：

（1）创建一个 ofstream 对象，管理输出流；

（2）该对象与特定的文件需要进行关联；

（3）使用 cout 的方式使用此对象，区别在于：输出将进入文件，而不是屏幕。

使用 ofstream 类，需要包含头文件<fstream>。对于大多数实现来说，包含该文件便会自动包括 iostream 文件，因此可以不必显示地包含<iostream>。ofstream 类在创建对象时，同时分配缓冲区空间。

使用缓冲空间实现文件操作，可以大大提高文件传输数据的速度。

同样，实用程序实现文件读取时，也需要 3 个步骤：

（1）创建一个 ifstream 对象，实现管理输入流；

（2）此对象会与特定的文件关联起来；

（3）使用 cin 的方式使用此对象。

读文件的步骤和写文件的步骤是相似的。同样需要包含头文件<fstream>，之后声明一个 ifstream 类的对象，并与文件名关联。之后就可以像使用 cin 一样，使用该流处理文件了。

需要指出的是：当输入流和输出流对象过期时，该流至文件的连接会自动关闭。当然也可以使用 close()方式显示地关闭文件和流之间的联系。当调用 close()函数时，并不会真正的删除流，仅仅断开流到文件的连接而已。该流仍然存在，该流的缓冲区也仍然存在，并且该流还可以重新连接到同一个文件或另一个文件。下面用例 7-15 来加以说明。

例 7-15

```
#include <iostream>     //need
#include <fstream>
#include <string>
using namespace std;
void main()
{
  string filename;
  cout<<"Enter filename for new file: ";
  cin>>filename;
  ofstream fout(filename.c_str());                    //将流与文件关联
  fout<<"For your eyes only!\n";                      //输出至文件
  cout<<"Enter your secret number: ";
  float secret;
  cin>>secret;
  fout<<"Your secret number is "<<secret<<endl;   //将变量 secret 输出至文件
  fout.clear();                                       //清除文件状态标识
  fout.close();
  ifstream fin(filename.c_str());
  cout<<"Here are the contents of "<<filename.c_str()<<": \n";
  char ch;
  while(fin.get(ch))                                  //读取文件的一种形式
  {
     cout<<ch;
  }
  cout<<"Done! \n";
  fin.clear();
  fin.close();
  cout<<"Other file read test:\n";
  ifstream fin2(filename.c_str());
  cout<<"Here are the contents of "<<filename.c_str()<<": \n";
  cout<<fin2.rdbuf();                                 //读文件的另一种形式
  fin2.clear();
  fin2.close();
}
```

在上述代码中，ofstream fout(filename.c_str())实现将文件流和数据文件关联，fout<<"For your eyes only!\n"实现将字符串"For your eyes only!\n"写入文件中。fout<<"Your secret number is "<<secret<<endl 实现将变量输出至文件中，fout.clear()实现清楚文件状态标志，get(ch)是读取文件的一种形式，而 fin2.rdbuf()是读取文件的另一种形式。

例 7-15 的执行效果见图 7-4。

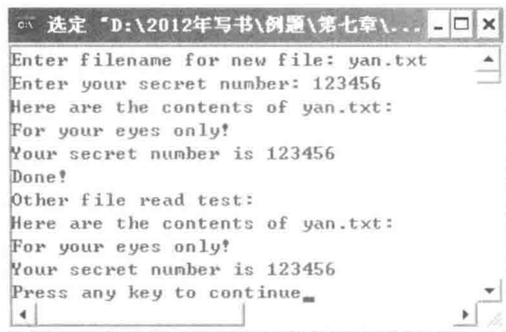


图 7-4　例 7-15 的执行效果

3. 打开文件和文件模式

在前面已经讲述了文件描述符和文件模式等内容。本小节主要讲述文件的打开和文件模式。前面的例题在使用文件时，在流定义时将流和文件关联，打开文件。流 ofstream 类还定义了 open()函数和 is_open()函数，实现文件的打开和是否打开的判断。

类 ifstream 的 open()函数声明如下：

```
void open(const char *s,ios_base::openmode mode = ios_base::in);
```

is_open()的函数声明如下：

```
bool is_open();
```

类 ofstream 的 open()函数声明如下：

```
void open(const char *s, ios_base::openmode mode = ios_base::out | ios_base::trunc);
```

is_open()的函数声明如下：

```
bool is_open();
```

C++文件流从 ios_base 类处继承了一个流状态成员。该成员存储了标明流状态的信息：一切顺利、已达文件尾、I/O 操作失败等。如果一切顺利，流状态为零。和其他状态一样，流状态也是通过将特定位设置为 1 而记录的。对于文件流，这些状态字也包括了检查打开文件是否成功。例如：要打开的文件未找到或不存在时，failbit 会被设置为 1。

举例如下：

```
istream fin;
fin.open(filename);
if(fin.fail())
{
}
```

函数 is_open()用于检查打开文件是否顺利。ifstream 类和 ofstream 类均提供了函数 is_open()，可以提供判断文件是否被正确打开的功能。例如：

```
if(fin.is_open())
{
.... ...
}
```

前面已经介绍过文件模式，此处再补充一些关于文件模式的内容。

文件模式描述的是文件被如何使用：读、写、追加等。流和文件关联时，可以提供指定文件模式的第二个参数。例如：

```
ifstream fin("abc.txt",mode);
```

或者：

```
ofstream fout;
fout.open("abc.txt",mode);
```

对于 C++，ios_base 类定义了一个 openmode 类型，用于表示模式；与 fmtflags 和 iostate 类型相同，是 bitmask 类型。选择 ios_base 类中定义的多个常量来指定模式。

无论是 istream 类或者 ofstream 类，其构造函数和 open()函数均能够提供两个参数，其第二个参数即是用来表示文件模式的。位操作符 OR(|)用于将两个位值合并成一个可用于设置两个位的值。模式必须显示地提供，fstream 类没有提供默认的模式默认值。在创建类对象时，必须显式地提供模式。

ios_base::trunc 标记意味着打开已有的现存文件，用于接收程序输出。以前的内容将被删除。此类行为极大地降低了耗尽磁盘空间的危险。也能想象这样的情况，不希望打开文件时将其内容删除。当 C++提供其他选择时，例如：在文件末尾追加新内容，即可使用 ios_base::app 模式。

标准 C++根据 ANSI C 标准 I/O 定义了部分文件 I/O。实现下面的 C++语句时：

```
ifstream fin(filename, C++mode)
```

其中，C++模式是一个 openmode 值，例如 ios_base::in。在 C++语言中，模式是相应的 C 模式字符串，例如“r”。在前面的内容中，C++模式和 C 模式的对应关系已经列出。值得注意的是，ios_base::ate 和 ios_base::app 均会将文件指针指向将要打开的文件末尾。但是，ios_base::app 模式仅允许将数据添加至文件末尾，而 ios_base::ate 模式会将指针放到文件末尾。

（1）追加文件

在文件末尾追加数据时，程序会维护一个存储清单的文件。该程序首先显示文件当前的内容。在文件被打开之后，使用 is_open()方法来检查该文件是否存在。之后，程序以 ios_base::app 模式打开文件，进行输出。程序请求用户从键盘输入时，会将其添加到文件中。之后，程序显示修订后的文件内容。

在很早之前，文件 I/O 可能是 C++最不标准的部分。很多编译器均不遵守当前标准。下面给出一个关于给文件追加内容的例子，见例 7-16。

例 7-16

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
using namespace std;
const string filename="test.txt";
void main()
{
ifstream fin;
fin.open(filename.c_str(),ios_base::in);            //打开文件
if(fin.is_open())
{
```

```
        cout<<"Here are the current contents of the \'"<<filename<<"\' :"<<endl;
        cout<<fin.rdbuf();                                    //输出文件内容
        cout<<endl;
    }
    fin.clear();
    fin.close();
    ofstream fout;
    fout.open(filename.c_str(),ios_base::out|ios_base::app);
    if(!fout.is_open())
    {
        cerr<<"Can't open "<<filename<<" file for output.\n";
        exit(EXIT_FAILURE);
    }
    cout<<"Enter new file contents (to be append):\n";
    string buffer;
    while(getline(cin,buffer)&&buffer.size()>0)          //输入整行数据
    {
        fout<<buffer<<endl;
    }
        fout.clear();
        fout.close();                                     //关闭文件
    fin.open(filename.c_str(),ios_base::in);
        if(fin.is_open())
        {
            cout<<"Here are the current contents of the \'"<<filename<<"\' :"<<endl;
            cout<<fin.rdbuf();
            cout<<endl;
        }
    fin.clear();
    fin.close();
       return;
}
```

程序输出结果：

```
Here are the current contents of the 'test.txt' :
这里是北京.abcdefg
lkjhgf
这是最后一行了。
abefjus
qwertyuiop
再添加一行吧。

Enter new file contents (to be append):
再加一行。


Here are the current contents of the 'test.txt' :
这里是北京.abcdefg
lkjhgf
这是最后一行了。
abefjus
qwertyuiop
```

```
再添加一行吧。
再加一行。

Press any key to continue
```

（2）二进制文件

std::ios::binary 打开模式实参指定文件被打开为二进制文件，而不是文本文件。值得提醒的是，在 UNIX 系统中，二进制文件和文本文件没有区别；在 windows 系统中，实参 std::ios::binary 是有意义的。文本文件和二进制文件之间可以使用以下方法进行转换：

- 在程序向二进制文件写入新行符“\n”时，文件系统将写入单个新行符。多数平台上，新行符与换行符（0xoa）相同。
- 在程序向文本文件写入新行符时，文件系统将写入两个字符：回车符（0x0d）和换行符（0x0a）。
- 程序从二进制文件中读取一个新行符时，文件系统将把单个新行符读取到存储器中；
- 在程序从文本文件中读取一对回车/换行符时，文件系统将把一对字符转换成存储器中的一个新行符；
- 在程序从文本文件中读取单个新行符时，换行符前面没有回车符，文件系统把一个新行符插入存储器。

由以上内容可知，上述方法不仅涉及新行符表示的转换，还涉及文件位置的操作——seeking 和 telling。文本文件在存储器中的数据表示与磁盘上数据表示的长度不同，是存储器中单个新行符与磁盘之间回车符和换行符之间双向转换造成的。在文本和二进制文件有区别的平台上，文本文件中的 seeking 和 telling 是不可靠动作。程序需要在文件内查找和告知当前文件的位置，程序应该使用二进制模式打开文件。无论在读文件时，还是在写文件时，均需要完整地处理和换行有关的符号“\r\n”，而不是仅仅处理符号“\n”或 std::endl。

将数据存储在文件中时，二进制格式具体地说是计算机内部表示。计算机不是存储字符，而是存储这个值的 64 位二进制表示。对于字符而言，二进制表示与文本表示是一样的，即字符的 ASCII 码的二进制表示。对于数字而言，二进制表示与文本表示有很大差别。每种格式都有自己的优点，文本格式便于读取，使用编辑器或字处理器来读取或编辑文本文件，能很方便地实现将文本文件在计算机系统之间的传输。二进制格式对于数字而言，比较精确。不会有转换误差或舍入误差。以二进制格式保存数据的速度非常快，因为另一个系统使用另一种内部表示，无法将数据传输给该系统。同一系统上不同的编译器可能使用不同的内部结构表示。必须编写一个将数据转换成另一种数据的程序。

使用二进制格式，更为有利的是，可以实现整块地写入和读取数据。通常在实现文件的读取和写入时，多数情况下使用的是函数 write()和 read()。下面引用 C++Primer Plus（第五版）中的例题 17.19（本书例 7-17），来说明二进制文件的读取和写入。

例 7-17

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <string>
using namespace std;
const string filename="planets.dat";
```

```
    inline void eatline(){while(std::cin.get()!='\n')continue;}
    struct planet
    {   char name[20];
    double population;
    double g;
    };
    void main()
    {   planet pl;
    cout<<fixed<<right;                                        //定点输出，右靠齐
    ifstream fin;
    fin.open(filename.c_str(),ios_base::in|ios_base::binary);
    if(fin.is_open())
    {   cout<<"Here are the current contents of the "<<filename<<"file: \n";
        while(fin.read((char*)&pl,sizeof(pl)))          //读取
        {cout<<setw(20)<<pl.name<<":"<<setprecision(0)<<setw(12)<<pl.population<<
setprecision(2)
        <<setw(6)<<pl.g<<endl;
        }
        fin.clear();                                           //清除所有标志位
        fin.close();                                           //关闭文件
    }
    ofstream fout(filename.c_str(),ios_base::out|ios_base::app|ios_base::binary);
    if(!fout.is_open())
    {   cerr<<"Can't open"<<filename<<" file for output: \n";
        exit(EXIT_FAILURE);
    }
    cout<<"Enter planet name (enter a blank line to quit):\n";
    cin.get(pl.name,20);
    while(pl.name[0]!='\0')
    {   eatline();
        cout<<"Enter planetary population: ";
        cin>>pl.population;
        cout<<"Enter planet's acceleration of gravity: ";
        cin>>pl.g;
        eatline();
        fout.write((char*)&pl,sizeof(pl));
        cout<<"Enter planet name (enter a blank line to quit): \n";
        cin.get(pl.name,20);
    }
    fout.close();
    fin.clear();                                               //打开文件前，清除所有标志位
    fin.open(filename.c_str(),ios_base::in|ios_base::binary);
    if(fin.is_open())
    {   cout<<"Here are the new contents of the "<<filename<<"\n";
        while(fin.read((char*)&pl,sizeof(pl)))
        {cout<<setw(20)<<pl.name<<":"<<setprecision(0)<<setw(12)<<pl.population<<
setprecision(2)
    <<setw(6)<<pl.g<<endl;
        }
        fin.close();
    }
    cout<<"Done. \n";
```

```
return;
}
```

例 7-17 的执行效果见图 7-5。

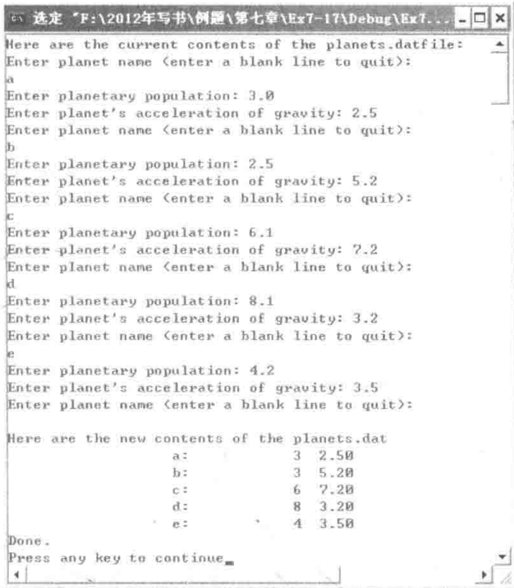


图 7-5 例 7-17 的执行效果

4. 命令行处理技术

文件处理程序通常使用命令行参数来指定文件。尤其在原有的 MS-DOS、UNIX 或者现有的 Windows 操作系统的 cmd 运行环境中。命令行参数是用户在使用程序时即时输入的，即在命令行中输入的参数。例如：

在 UNIX 操作系统中使用命令 wc，用于统计文件信息：

```
wc a.dat b.dat c.dat d.dat e.dat
```

此时，wc 是程序名，a.dat，b.dat，c.dat，d.dat 和 e.dat 是命令行参数传递给程序的文件名。对于 C/C++来说，在命令环境中运行的程序能够访问命令行参数，方法是使用 main()函数的参数。对于一个普通的 main()函数，其形式多为：

```
int main(int argc, char *argv[])
```

其中参数 argc 是命令行中的参数个数，其中包括命令名本身。在上例中，所谓命令名本身即是指命令 wc 本身。参数 argv 变量为一个指针，指向 char 型指针。将参数 argv 看做指针数组，其中的指针指向命令行参数，argv[0]是一个指针，指向存储第一个命令行参数的字符串的第一个字符，依此类推。argv[0]是命令行中的第一个字符串。例如，在上例中，argv[1]代表文件名 a.dat，依此类推。对于上例：

```
wc a.dat b.dat c.dat d.dat e.dat
```

main()函数的各参数分别为：argc 等于 6，argv[0]为 “wc”，argv[1]为 “a.dat”，argv[2]为 “b.dat”，argv[3]为 “c.dat”，argv[4]为 “d.dat”，argv[5]为 “e.dat”。

可以通过在 main()函数的起始位置填写部分语句，将 main()函数的各参数进行输出。当然，命令行参数与命令行操作系统紧密相关。其他程序可能允许使用命令行参数。下面用例 7-18 来加以说明。

例 7-18

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
void main(int argc, char* argv[])
{
if(argc==1)
{    cerr<<"Usage: "<<argv[0]<<" filename[s]\n";          //输出程序的名称
     exit(EXIT_FAILURE);
}
else
{    int count=argc;
     for(int i=0;i<count;i++)
     {    cout<<"Paramter: "<<argv[i]<<endl;               //输出程序的参数
     }
}
}
```

例 7-18 的执行效果见图 7-6。

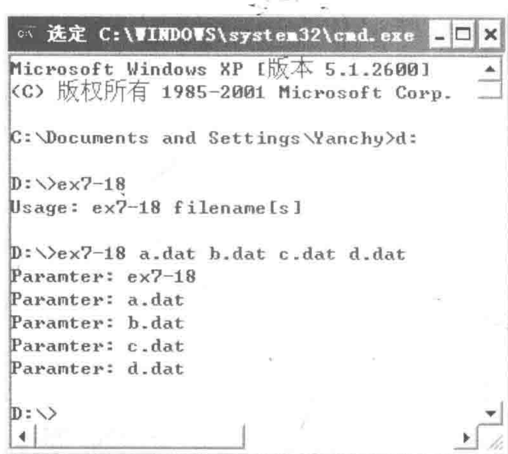


图 7-6 例 7-18 的执行效果

7.6.2 随机存取

随机存取指的是将内容直接移动到文件的任何位置，直接存取文件任何位置的内容。随机存取常被用于数据库文件，程序维护一个独立的索引文件，该文件指出在主数据文件中的位置。程序可以直接跳到这个位置，读取其中的数据。如果文件由长度相同的记录组成，这种方法实现最简单。每条记录表示一组相关的数据。

和 stream 读写位置相关的成员函数包括：

对于输入流：

```
tellg(),   seekg(pos),   seekg(offset, rpos);
```

对于输出流：

```
tellp(),   seekp(pos),   seekp(offset, rpos);
```

上述函数分别以特殊字符 g 或 p 结尾，g 代表“get”，p 代表“put”。用于读写的位置函数定义

于 basic_istream，用于改写的位置函数定义于 basic_ostream。然而，并不是所有的输入流和输出流均能够支持读写定位。例如：cin、cout 和 cerr 就不支持读写定位。

当程序读写文件时，它将协调地移动输入缓冲区中的输入指针和输出缓冲区中的输出指针。在文件中，移动文件指针也需要一种方式。fstream 类继承了两个方法：seekg()和 seekp()。前者将输入指针移到指定位置，后者将输出指针转移到指定的文件位置。将 seekg()用于 ifstream 对象，将 seekp()用于 ofstream 对象。

函数 seekg() 和 seekp()可以接受一个绝对位置或一个相对位置。对于绝对位置，必须采用 tellg()和 tellp()，会返回一个绝对位置，型别为 pos_type。在文件中，逻辑位置和实际位置可能并不相同，这和字节表示法有关系，是非常复杂的问题。

如果是相对位置，偏移值可以和 3 个位置相关，通常 3 个参数被定义于 ios_base 类中，型别均为 seekdir。这 3 个参数是：ios_base::beg，ios_base::cur，ios_base::end。

对于文件位置的偏移量，其类型都属于 off_type 型别，是 streamoff 的间接定义。由于 streamoff 是带正负号的整数型别，使用整数当作 stream 的偏移值。

无论是文件位置的绝对值，还是偏移值，其位置均在文件长度中有效，如果该位置超越了文件起头，在文件结尾会导致未定义的行为。

函数 seekg()和 seekp()的函数原型分别为：

```
basic_istream& seekg( pos_type _Pos);
basic_ostream& seekp(pos_type _Pos);
//
basic_istream& seekg( off_type _Off,ios_base::seekdir _Way);
basic_ostream& seekp(off_type _Off,ios_base::seekdir _Way);
```

两个函数的第一种形式仅包含一个参数，参数代表距离文件开头的特定距离的位置；两个函数的第二种形式包含两个参数，第一个参数的位置描述是相对于第二个参数的，第二个参数可以使用 3 个值：

```
ios_base::beg, ios_base::end, ios_base::cur。
```

输入流和输出流还分别提供了函数 tellg()和 tellp()，用于检查文件指针的当前位置，函数返回值均为表示当前位置的 streampos 值。tellg()对应于输入流，tellp()对应于输出流。创建 fstream 对象时，输入指针和输出指针将一前一后地移动，是分别独立的。对于同一个文件，如果同时使用 ifstream 对象来管理输入流，使用 ofstream 来使用输出流，则输入指针和输出指针将彼此独立移动。此时，tellp()和 tellg()的返回值是不同的，是独立的。下面用例 7-19 和例 7-20 来加以说明。

例 7-19

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void printfile(string filename)
{
   ifstream fin(filename.c_str());
cout<<fin.rdbuf();                //输出缓冲区内容
fin.seekg(0);                     //文件指针放至文件头
cout<<fin.rdbuf();                //输出文件内容
}
```

```
void main(int argc, char* argv[])
{
for(int i=1;i<argc;i++)
{
    cout<<"The contents of file "<<argv[i]<<" is below.\n"<<endl;
    printfile(string(argv[i]));
}
}
```

例 7-19 的执行效果见图 7-7。

图 7-7　例 7-19 的执行效果

例 7-20

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
string filename;
void main(int argc, char* argv[])
{
char file[20];
cout<<"Input filename: ";

cin>>file;
filename=string(file);
if(filename.empty())
{
    exit(EXIT_FAILURE);
}
cout <<"This example is used for test seekg(p) or tellg(p). "<<endl;
fstream finout,fout;
//ofstreambuf
```

```
finout.open(filename.c_str(),ios_base::in|ios_base::out);
if(finout.is_open())
{
    cout<<finout.rdbuf();                        //输出文件内容
    finout.seekg(0);
    cout<<finout.rdbuf();
}
finout.seekg(0,ios_base::beg);
fout.open("b.txt",ios_base::out);
if(fout.is_open())
    {
    fout<<finout.rdbuf();
    fout.close();
    }
cout<<endl;
fout.open("c.txt",ios_base::out);
if(fout.is_open())
    {
    finout.seekp(25,ios_base::beg);
    fout<<finout.rdbuf();
    fout.close();
    }
cout<<endl;
fout.open("d.txt",ios_base::out);
if(fout.is_open())
    {
    finout.seekp(51,ios_base::beg);             //重新定位文件指针
    fout<<finout.rdbuf();
    fout.close();
    }
cout<<endl;
}
```

例 7-20 的执行效果见图 7-8。

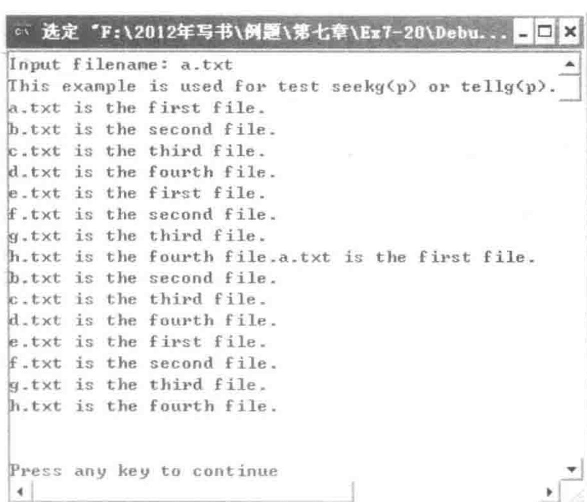


图 7-8　例 7-20 的执行效果

总结：本节使用两个例题，说明了文件随机存取的使用方法。两个例题中，文件均为文本文件，这样便于读者校对源代码和程序的输出结果。主要讲述了 seekg()和 seekp()的使用方法。tellg()和 tellp()的使用非常简单，没有在例题中体现。

7.6.3 4 个类模板简介

本小节简要介绍一下 4 个类模板。在本章的前面已经对 4 个类模板进行了使用，但并没有对这 4 个类模板的全部功能和全部成员函数进行透彻的讲解和说明。

1. 类模板 basic_filebuf

类 basic_filebuf 是从类 basic_streambuf 派生而来的，属于名称空间 std。此类包含了 5 个类型声明：

```
typedef  charT char_type;
typedef  typename traits::int_type  int_type;
typedef  typename traits::pos_type  pos_type;
typedef  typename traits::off_type  off_type;
typedef  traits  traits_type;
```

还包含了一个构造函数和一个析构函数：

```
basic_filebuf();
virtual ~basic_filebuf();
```

类的成员函数有 3 个：is_open()，open()，close()。函数原型分别为：

```
bool is_open( );
basic_filebuf *open( const char *_Filename, ios_base::openmode _Mode);
basic_filebuf *close( );
```

其中函数 open()返回值是指针类型。

此类还包括几个重载函数：showmanyc()，underflow()，uflow()，pbackfail()，overflow()，setbuf()，seekoff()，seekpos()，sync()，imbue()。

类 basic_filebuf 使输入流和输出流同各自的文件关联起来。

- 函数 is_open()，如果文件打开成功，函数返回 true，否则函数返回 false。
- 函数 open()和 close()，实现文件或流的打开和关闭。
- 函数 showmanyc()可以实现判断出输入流中有多少字符可以被读取。
- 函数 underflow() 在前面已经讲过。
- 函数 uflow()从输入流中读取当前字符，并返回为整数值形式。
- 函数 pbackfail()将字符放回输入流中。
- 函数 overflow()前面已经讲过。
- 函数 seekpos()使用绝对位置，随机访问文件。
- 函数 seekoff()使用相对位置，随机访问文件。
- 函数 sync()实现缓冲区和文件的同步。
- 函数 imbue()默认是不做任何事情。

2. 类模板 basic_ifstream

类 basic_ifstream 是从类 basic_istream 派生而来的。同样包含了 5 个类型声明：

```
typedef  charT char_type;
```

```
typedef  typename traits::int_type  int_type;
typedef  typename traits::pos_type  pos_type;
typedef  typename traits::off_type  off_type;
typedef  traits  traits_type;
```

两个构造函数：

```
basic_ifstream();
explicit basic_ifstream(const char* s, ios_base::openmode mode=ios_base::in);
```

4 个成员函数：

- 函数 rdbuf()是获取流缓冲区的指针；
- 函数 is_open()用于判断流是否被正确打开；
- 函数 open()用于打开文件；
- 函数 close()用于关闭文件和输入流之间的关联。

3. 类模板 basic_ofstream

类模板 basic_ofstream 是从类 basic_ostream 派生而来的。同样，此类包含了 5 个类型声明：

```
typedef  charT char_type;
typedef  typename traits::int_type  int_type;
typedef  typename traits::pos_type  pos_type;
typedef  typename traits::off_type  off_type;
typedef  traits  traits_type;
```

两个构造函数：

```
basic_ofstream()
explicit basic_ofstream(const char*s, ios_base::openmode mode=ios_base::out)
```

还包括 4 个成员函数：

- 函数 rdbuf()获取流缓冲区的指针；
- 函数 is_open()用于判断是否正确打开文件；
- 函数 open()用于打开文件，如果打开成功，将建立文件和流之间的关联；
- 函数 close()用于关闭流和文件之间的关联。

4. 类模板 basic_fstream

类 basic_fstream 是从类 basic_iostream 派生而来的。同样，该类包括 5 个类型声明：

```
typedef  charT char_type;
typedef  typename traits::int_type  int_type;
typedef  typename traits::pos_type  pos_type;
typedef  typename traits::off_type  off_type;
typedef  traits  traits_type;
```

还包括两个构造函数：

```
basic_fstream()     //可用于仅仅创建一个流对象
explicit basic_fstream(const char *s, ios_base::openmode mode=ios_base::in|ios_base::out)
//创建流对象的同时打开文件
```

还包括 4 个成员函数：

- 函数 rdbuf()用于获取流缓冲区的指针；
- 函数 is_open()用于判断是否正确打开文件；

- 函数 open()用于打开文件，如果打开成功，将建立该文件和流之间的关联；
- 函数 close()用于关闭流和文件之间的关联。

7.6.4 C 库中的文件存取功能概述

在本节简单介绍了 C 语言中的关于文件存取的部分函数和宏。在头文件<cstdio>中，涉及文件存取的宏见表 7-9。

表 7-9 头文件<cstdio>中包括的宏

宏 名 称	功 能
BUFSIZ	用户自定义缓冲区大小
FOPEN_MAX	可以同时打开的最大文件数目
SEEK_CUR	文件指针的当前位置
TMP_MAX	函数 tmpnam()可以产生的最大文件数目
_IONBF	缓冲区类型
stdout	标准输出
EOF	文件结束标志
L_tmpnam	函数 tmpnam()产生的临时文件的文件名的长度
SEEK_END	文件末尾
_IOFBF	缓冲区类型
stderr	标准错误
FILENAME_MAX	文件名的最大允许长度
NULL	空
SEEK_SET	文件开始位置
_IOLBF	缓冲区类型
stdin	标准输入

头文件<cstdio>中包含的数据类型见表 7-10。

表 7-10 头文件<cstdio>中包括的数据类型

宏 名 称	功 能
FILE	文件结构
fpos_t	长整型数据类型
size_t	无符号整型

该头文件中包括声明的函数见表 7-11。

表 7-11 头文件<cstdio>中包括声明的函数

函 数 名 称	功 能
clearerr(FILE* stream)	//复位流的错误标志
fgets ( )	//从流中获取一串字符
fscanf	//从流中按格式读取
gets	//从标准输入流中获取一行数据
rename	//重新命名文件

续表

函数名称	功　　能
tmpfile	//创建临时文件
fclose	//关闭文件
fopen	//打开文件
fseek	//随机访问文件
perror	//打印错误信息
rewind	//重新定义文件指针到文件起始位置
tmpnam	//创建临时文件名
feof	//判断是否在文件结尾
fprintf	//向文件格式输出
fsetpos	//设置流位置标识
printf	//格式化输出
scanf	//格式化读取
ungetc	//将字符放回流缓冲区中
ferror	//检查流中的错误
fputc	//向文件中输出字符
ftell	//判断文件指针位置
putc	//输出字符
setbuf	//将指定的存储区域和流缓冲区位置关联
vfprintf	//向文件中写入
fflush	//清空缓冲区
fputs	//向文件中输出一串字符
fwrite	//向文件中写入数据
putchar	//输出字符
setvbuf	//控制流缓冲区和缓冲区的大小
vprintf	//格式化输出
fgetc	//从文件中读取一个字符
fread	//从文件中读数据
getc	//从标准输入中获取一个字符
puts	//向标准输出中输出一行数据
sprintf	//格式化字符串，输出数据至字符串
vsprintf	//格式化输出字符串
fgetpos	// 获取流的文件位置标识
freopen	// 重设文件指针
getchar	// 获取字符
remove	// 删除
sscanf	// 从字符串中格式化输入

7.7 本章小结

本章主要讲述输入和输出流的相关内容。讲述了 IOStream 流的基本内容，并讲述了流中的格式化方法，对流缓冲区进行了详细讲解。介绍了基于字符串的流和文件流，并重点讲述了基于文件的流的使用方法。

本章是本书的学习重点之一，希望读者能够认真理解，反复阅读。

第 8 章 异常处理

在本书第 1 章，曾简要提到了异常处理的内容。

异常处理是一种 C++特性。异常处理将会使程序按照有序、有组织和一致的方式截取和处理异常条件——错误。异常处理允许使用一段程序来感知和分发错误条件，用另一段程序来处理错误。检测错误的代码可能不知道错误处理的策略，代码可以是库中的类和函数，也可能包含其他种类的处理错误的代码。通常错误来源于程序的隐含错误或致命错误（无效等）。函数库不可能知晓所有能检测出的可能发生的异常。函数将错误汇报给正在使用的程序，需要一定的前提。总而言之，异常处理是一个较复杂的过程。

8.1 异常概念和基本思想

本节主要讲述异常（exception）。异常是程序开发必须重视的内容。本节涉及的内容主要包括：

异常的概念，异常的分类，异常的捕捉和处理，资源管理，异常和效率，异常的描述，未捕捉的异常。

8.1.1 异常的概念

异常处理，英文名为 exceptional handling。异常处理分离了接收和处理错误的代码。异常处理功能理清了编程者的思绪，同时增强了代码的可读性，方便维护者的阅读和理解。

异常处理（又称为错误处理）功能提供了处理程序运行时出现的任何意外或异常情况的方法。异常处理使用 try、catch 和 finally 关键字来尝试可能未成功的操作以及在事后清理资源。

异常处理通常是防止未知错误产生所采取的处理措施。异常处理使程序员不用绞尽脑汁去考虑各种错误，为处理多种错误提供了有效的方法，使编程效率大大提高。

在 C++语言中，异常通常由 throw 关键字的应用程序代码生成。

异常处理通常有以下两种理论模型：

（1）“终止模型”（Java 与 C++所支持的模型）。在这种模型中，所有的错误被假设是非常关键的，错误发生时通常程序无法返回到异常发生之处继续执行。异常被抛出后，说明产生的错误已经无法挽回。

（2）“恢复模型”。异常处理的功能是修正程序产生的错误，并重新尝试调动出问题的方法（或

函数)，并认为二次调用会成功。

通常，对于恢复模型，异常处理完毕之后能继续执行程序。异常的抛出更像是对方法（或函数）的调用。如果把 try 块放在 while 循环中，可以不断地进入 try 块，直到截获可能的异常发生或者获得程序员满意的结果。

由以上描述可知，恢复模型显得更有魅力，更加吸引广大程序员。多数操作系统是支持恢复模型的异常处理的。但是完美的处理不一定是最佳选择，多数程序员更喜欢“终止模型”。主要的理由是，程序必须关注异常抛出的地点，关注导致无法包含或编写依赖于抛出位置的非通用性代码。这增加了代码编写和维护的困难程度。

上述对异常的描述，可以归结为一句话：异常会使对某个函数的调用动作中止，而该函数正是异常发生之处。异常处理机制有能力将某个对象作为参数，回传给函数调用者。但并不是使用的“回调函数”(回调函数属于信号处理机制范畴)。异常处理和信号处理是不同的概念。

8.1.2 异常的分类

通过异常处理，C++标准程序库可以在“不污染”函数接口的情况下处理异常。一旦遭遇异常情况，通过“抛出异常”停止正常的处理过程。语言本身或标准程序库所抛出的所有异常，均派生于基类 exception。

类 exception 的声明如下：

```
class exception {
public:
    exception( ) throw( );
    exception(const exception& right) throw( );
    exception& operator=(const exception& right) throw( );
    virtual ~exception( ) throw( );
    virtual const char *what( ) const throw( );
    };
```

它包含了两个构造函数，这两个构造函数全部包括异常描述(后面讲述)。一个右侧赋值操作符，一个虚析构函数和一个虚 what()函数。

类 exception 是多个标准异常类别的基类，它们共同构成类的体系。标准异常类通常分为以下 3 组：

（1）语言本身支持的异常；

（2）C++标准程序库发出的异常；

（3）程序作用域（scope of a program）之外发出的异常。

第一种异常是语言本身支持的异常。此类异常多用于支撑某些语言的特性。此类异常处理不是 STL 的一部分，而是核心语言的内容。一旦操作失败，会抛出异常。全局操作符 new 一旦操作失败，会抛出 bad_alloc 异常。在较复杂的程序中，尤其大量使用 new 的程序中，此类异常可能会随时发生，是一种非常重要的异常。程序执行时，当加诸于 reference 身上的“动态型别转换操作”失败时，动态 dynamic_cast 通常会抛出 bad_cast 异常。对于型别辨识过程，赋予 typeid 的参数一旦是零或空指针，typeid 会抛出 bad_typeid 异常。值得一提的是：如果发生非预期的异常，bad_exception 异常会即时处理。此时通常函数 bad_exception 会调用函数 unexpected()。

类 bad_alloc 的声明如下：

```
class bad_alloc : public exception
{};
```

类 bad_cast 的声明如下：

```
class _CRTIMP bad_cast : public exception {
public:
   bad_cast(const __exString& what_arg) : exception (what_arg) {}
};
```

第二种异常是 C++标准程序所发生的异常。C++标准程序库总是派生自 logic_error。从理论上讲，它能够通过一些手段，使程序避免逻辑错误。逻辑错误通常指的是包括违背逻辑前提或违反类的不变性。C++标准程序库提供 4 个逻辑错误类别，分别是：invalid_argument，length_error，out_of_range，domain_error。

类 logic_error 的声明如下：

```
class logic_error : public exception {
public:
  logic_error(const string& message);
};
```

上述类 logic_error 的构造器的参数 string& message 使用于描述异常的信息，以便于在使用 what() 函数时，返回该 message。下同。

invalid_argument 通常表示无效参数，例如:使用 char 类型对 bitset 初始化；该类的声明如下：

```
class invalid_argument  : public logic_error {
public:
  invalid_argument(const string& message);
};
```

length_error 通常指某个行为“可能超越了最大极限”，例如：字符串中字符数目过多。该类的声明如下：

```
class length_error : public logic_error {
public:
  length_error(const string& message);
  };
```

out_of_range 通常指参数值“不在预期范围内”，例如：在 array 型容器中或字符串 string 中采用错误的索引。该类的声明如下：

```
class out_of_range : public  logic_error {
public:
  out_of_range(const string& message);
};
```

domain_error 通常指专业领域内的错误。该类的声明如下：

```
class domain_error : public  logic_error {
public:
  domain_error(const string& message);
  };
```

STL 的 I/O 部分提供名为 ios_base::failure 的特殊异常。数据流由于错误或由于到达文件末尾而发生状态改变时，可能会抛出异常。

由于标准程序库会用到语言特性及客户所写的程序代码，可能间接抛出任何异常。尤其是何时分配存储空间，都有可能抛出 bad_alloc 异常。

标准程序库的任何具体的实例化类及其对象，都可能提供额外的异常类别。使用非标准类别将导致程序无法移植。必要时需要非常痛苦地修改源程序，因此使用标准异常是非常必要的。

第 3 种异常是程序作用域之外发生的异常。派生自 runtime_error 的异常，用来指出“不在程序范围内，且不容易回避”的事件。C++标准程序库对“执行期错误”提供 3 个类：

range_error：指出内部计算时发生区间错误。该类的声明如下：

```
class range_error : public runtime_error {
public:
   range_error(const string& message);
};
```

overflow_error：指出算术运算发生上溢位。该类的声明如下：

```
class overflow_error : public runtime_error {
public:
   overflow_error(const string& message);
};
```

underflow_error：指出算术运算发生下溢位。该类的声明如下：

```
class underflow_error : public runtime_error {
public:
   underflow_error(const string& message);
};
```

类 runtime_error 的声明如下：

```
class runtime_error : public exception {
public:
   runtime_error(const string& message);
};
```

使用标准异常类时，需要包含头文件<exception>，<new>，<typeinfo>，<ios>和<stdexcept>。在头文件<exception>中包含类 exception 和 bad_exception 的定义声明。bad_alloc 类在头文件<new>中声明，bad_cast 和 bad_typeid 在头文件<typeinfo>中声明。ios_base::failure 声明于头文件<ios>中。其他多数异常声明于头文件<stdexcept>中。

8.1.3 异常的捕捉和处理

在 C 语言中，采用两种方法处理异常。需要对每个函数调用错误测试，或者调用 setjmp()和 std::longjmp()来截取错误条件。

第一种方法使用类似全局宏 errno 的变量、零或错误函数的返回值来报告错误种类，方法可靠但非常烦琐。程序员无法全部考虑各种错误的可能性。其中 errno 包含在头文件<cerrno>中，其声明如下：

```
namespace std {
   using ::errno;
}
```

第二种方法是使用 setjmp()和 std::longjmp()函数。这两个函数的使用更加接近于 C++异常处理所能达到的效果，能够依次、自动地把堆栈还原至函数调用层次结构中较高层位置所记录的状态。setjmp()和 std::longjmp()两个函数会截取和处理不需要瞬时终止程序的错误条件。程序使用 setjmp()指定返回的位置，并使用 std::longjmp()返回到该位置。

上述两种异常处理办法是不能完好地解决异常处理的。

在 C++中，通常使用 try 块和 catch 块处理异常的捕捉和处理。try 块中可以执行一些 C++函数，能检测执行过程中的错误，并能在出错之后恢复。try 块的形式如下：

```
try{
//c++ statements
}
```

在 try 块之外执行的代码不能检测出或处理异常情况，即在某个 try 块外面的代码发生异常时，程序会束手无策。try 块还可以嵌套其他的 try 块，一旦发生意外，try 块会调用其他能够检测和处理异常的函数。

catch 块用于处理异常。理论上 catch 块出现在 try 块之后，并且 catch 块包含形参。catch 块的形式如下：

```
catch(type argument)
{
// error-handling code
}
```

catch 块"捕捉"到由 try 块中某个 throw 语句抛出的异常。try 块之后列出多个具有不同形参的列表来处理函数。

例如：

```
try{
}
catch(int err){
}
catch(char* msg){
}
```

不同的 catch()处理块通过形参列表中的类型区分，不必给 catch 形参列表中的形参命名。如果形参被命名，即声明了具有名字的对象，异常检测代码会传递形参值。如果形参没有被命名，将丢弃异常检测代码中的 throw 语句的实参。

throw 语句是用来抛出异常的。为了检测出异常并跳转到某个 catch 处理函数，C++函数发出 throw 语句，语句中包含的数据类型与适当的 catch 处理函数的形参列表相匹配。例如：

```
throw "An error has occured";
```

该 throw 语句跳转到具有 char*形参列表的 catch 异常处理函数。throw 语句还会还原堆栈，通过调用对象的析构函数清除在 try 块内声明的所有对象。之后，throw 会调用匹配的 catch 处理函数，传递形参对象。

提示：如果 catch()的形参为省略号（即 catch(...)），则 catch 处理函数能捕捉到所有未曾被捕捉的异常。通常 catch(...)块放在所有 catch 块的最后。

下面使用一个例子讲述异常处理的问题。

```
try{
throw "error information".          //抛出异常信息
throw errorcode;                    //抛出错误代码
}
catch(…)
```

```
{
......
}
```

8.1.4 资源管理

当函数申请了其所占有的资源时，为保证系统的运行，必须正确地释放此类资源。即通常所说的“正确释放”，这要求申请资源的函数在返回其调用者之前完成资源的释放。例如，一旦在调用函数 fopen()之后，调用 fclose()之前，程序运行出了问题，会出现异常退出，但此时却没有来得及调用 fclose()函数。即使是常规的 return 语句实现问题，也会导致 use_file()退出而没有关闭该文件。

在使用时，可以将很多代码包裹在一个 try 块内，便于捕捉所有的异常，关闭文件并重新抛出这个异常。此方法过于烦琐，且代价高昂，容易诱发各类错误，让程序开发人员深感厌恶。所幸还有另一种更好的解决办法。

对于类的对象，通常由构造函数创建，并由析构函数销毁。只要适当地利用带有构造函数和析构函数的类对象，就可以处理此种资源的申请和释放问题了。例如，在使用文件时，程序员可以定义文件类，其中包含其构造函数和析构函数。一旦该类的对象在其作用域结束时将被销毁，析构函数会关闭相应的文件（当然，需要在析构函数中添加相应的代码）。无论函数是正常结束，还是非正常退出，析构函数总是会被调用的。异常处理机制使程序员可以从主要算法中删去处理错误的代码，这样代码更简单。

1. 构造函数和析构函数的使用

利用局部对象管理资源的技术通常是“资源申请即初始化”。该技术依赖于构造函数和析构函数的性质和这两个函数与异常处理的相互关系。

当某个对象的构造函数执行完毕时，此对象才被认为已经明确建立。之后，堆栈回退时才为该对象调用析构函数。由子对象组成的对象的构造函数需要等待所有子对象的构造函数均完成执行才完毕。

对于数组的构造函数，执行时将持续到数组所有元素均构造完成，数组的构造函数才执行完毕。

对象的构造函数努力去保证对象能够完整、准确地创建起来。如果目标无法实现，书写良好的构造函数会将系统的状态恢复到对象构造之前的情况。理想状况是：按朴素的方式写出构造函数能达到的可能情况，不使它们处理的对象处于某种“片面构造”的状态。使用“资源申请即初始化”技术，可以实现完全构造类对象。

在资源申请简单模型的支持下，编写构造函数的人不必去专门写显式的异常处理代码。这样，程序员不必去追踪其他情况。但这也带来了很多问题，例如：大量的初始化工作需要放在构造函数中才更加保险，才不会导致剧烈的异常给整个程序或操作系统带来致命的冲击。

2. STL 的 auto_ptr 类

类 auto_ptr 的对象可以使用指针去初始化，且能够以与指针同样的方式间接访问。在 auto_ptr 类对象退出作用域时，所指的对象将被隐式地自动删除。类 auto_ptr 型指针具有与常规指针不一样的复制意义：在将一个 auto_ptr 型指针复制给另一个指针之后，原来的 auto_ptr 不再指向任何东西。复制 auto_ptr 将导致对其自身的修改，所以 const auto_ptr 不可以被复制。

类 auto_ptr 的声明：

```
template <class Type> class auto_ptr
```

使用类 auto_ptr 时，需要包含头文件<memory>。多个 auto_ptr 拥有同一个对象的效果是无定义的。最可能的情况是使该对象无法被彻底删除。

任何程序均需要具备从破坏中恢复起来的能力。不可能所有的程序都需要使用“资源申请即初始化”技术、auto_ptr 和 catch 块等技术去保护该程序。对于一些简单程序，一旦发生异常，最有效的办法是使该进程夭折。此时，程序会释放所申请的所有资源，重新运行程序。

3．构造函数中的异常和 new

在类的构造函数中，如果大量使用 new()分配存储空间，一旦发生异常，能够完整地释放这些存储空间吗？回答是肯定的。

但是在程序中使用 new()函数时，释放该内存空间时需要使用 delete()函数。

异常是从构造函数报告的问题中提供解决方案。由于构造函数不能顺利返回独立的值供调用程序检查，通常的解决办法有 4 种：

（1）返回处于错误状态的对象，相信用户有办法检查其状态；

（2）设置非局部变量指出创建失败，相信用户能检查该变量；

（3）在构造函数中不做初始化，依靠用户第一次使用对象之前调用的某个初始化函数；

（4）将对象标记为“未初始化”，让该对象调用的成员函数完成实际的初始化工作，并允许该函数在初始化失败时报告错误。

异常处理机制使构造失败信息能从构造函数的内部传递出来。程序员可以实现安排一些 throw()，例如：某类的对象遭遇超量存储时，可以在程序中使用判断语句，一旦遭遇此类情况，即可将其抛出。

尤其是处理那些多种资源的构造函数，“资源申请即初始化”技术是最安全最优秀的方法。从根本上讲，该技术把处理多种资源的问题，归结为一种反复应用处理单一资源技术的问题。

如果类的成员在初始化时抛出异常，按照默认的约定，该异常将传至类的构造函数位置。构造函数本身通常会将完整的函数体包括在 try 块内，函数本身是无法捕捉这些异常的。

另外，构造函数是无法复制的。复制构造函数可以通过抛出异常的方式发出失败信号。在此种情况下，实际上没有创建对象。例如：对于某容器类，在复制构造函数时需要分配存储并复制元素，这可能导致异常的抛出。抛出异常之前，复制构造函数需要释放已经申请到的所有资源。尤其是复制赋值操作会申请资源，也可能通过抛出异常而结束。但在抛出异常之前必须保证每个操作对象均保持在某种合法状态。

4．析构函数中的异常

从异常处理的角度来说，析构函数中也可能发生异常。通常此时析构函数会在两种情况下被调用：

（1）正常调用：作为某个作用域正常退出的结果，作为 delete 操作的结果。

（2）在异常处理中被调用：在堆栈回退过程中，异常处理机制退出作用域时，其中包含析构函数的对象。

对于第二种情况，不能让析构函数抛出异常。如果真的在析构函数中发生了异常的抛出，即是异常处理机制的重大失败。在析构函数中抛出异常而导致析构函数退出程序，违背了 STL 的基本原则。

为防止上述情况的发生，在析构函数中，可以使用 try 块和 catch 块。例如，对于类 C，其析构

函数可以如下编写：

```
X::~X()
{
    try{
    }
  catch(…){
   }
}
```

如果有异常被抛出之后未被捕捉，STL 中的函数 uncaught_exception()会返回 true。此函数多应用在析构函数中，调用时可以获取合适的返回值，从而确定对象是被正常销毁还是作为堆栈回退的一部分。

5. 资源耗尽

申请资源时，一旦失败，会导致很大的问题。例如：程序在打开文件时文件不存在，或者耗尽了所有自由的存储空间。通常会有两种解决方案：

（1）唤醒机制，请求调用某个程序，纠正问题，之后继续执行；

（2）结束当前计算，并返回某个调用程序。

对于第一种方式，预先需要准备一个调用程序来帮助处理某段未知的代码中出现的资源申请问题。

对于第二种方式，调用程序必须准备好应付某个资源申请失败的情况。第二种方法更为简单一些，也更为实用。第二种方法能够使系统保持抽象层次之间的较好的隔离。采取终止策略时，终止的不是整个程序，而是其中个别的计算。“终止”是传统的描述策略的术语，表示从“失败”的计算返回到与某个调用过程相关的错误处理器，而不试图去修复坏的状况，之后从检查出问题的那个位置继续。

在 C++中唤醒模型通常由函数调用机制支持，终止模型由异常处理机制支持。要抛出异常，必须要求存在一个被抛出的对象。每个 C++实现时都要求保留足够的存储，在存储耗尽时需要抛出 bad_alloc()。由于抛出其他对象而导致存储耗尽的情况也是会发生的。

8.1.5 异常和效率

在没有抛出异常之前，异常处理的实现在运行时没有任何额外开销。抛出异常并不比调用函数的开销大。在完成这些的同时，要维持与 C 语言在调用序列、排错规范等方面的兼容性，而又不引起显著的额外存储开销，这是非常困难的。那些能代替异常的东西是需要代价的。在大多数的规模程序或系统软件中，更多的代码（甚至超过一半）实际上是用于错误处理和异常处理的。对于下述形式的代码：

```
void g(int );
void f()
{
string s;
//
g(1);
g(2);
}
```

如果 g()会抛出异常，f()必须包含部分代码，使异常发生时会得到即时处理。例如：确保字符串

变量 s 正确销毁。如果函数 g()不采用抛出异常的方式，也会使用另一种方式报告发生的错误。与之相比较，使用常规方式写出的处理错误的代码，需要包含大量的 if 判断和 return 语句。就像排除地雷一样，走一步就要判断一步。

异常描述可能非常有助于改进所生成的代码。但是，传统的 C 函数不会抛出异常。在大部分程序中，每个 C 函数均可以用空抛出描述 throw()的声明。举例如下：

```
void function () throw()
```

上述函数定义中 throw()的参数表为空，说明此函数不能抛出任何异常。大部分传统的 C 语言函数都不能抛出异常。所以在编写程序时，每个 C 函数都可以用空抛出声明。对于标准 C 库函数只有很少的几个函数可以抛出异常，例如：atexit()和 qsort()。举例如下：

```
int atexit( void ( __cdecl *func )( void ) );
```

函数 atexit()指定函数 func()在程序终止时被调用。如果调用成功，函数返回 0；如果调用失败，函数返回非零。进程函数 func()返回值必须为 void，并且函数 func()不允许包含参数。

函数 qsort()在前面讲述算法的章节已有介绍。

利用这些函数可以生成更好的代码。值得注意的是：在为某个 C 函数空异常描述 throw()之前，要思考一下该函数是否有必要抛出异常。如果有需要抛出异常的话，还是要采用空异常描述。

8.1.6 异常的描述

1. 异常描述

抛出或捕捉异常对某个函数与其他函数的关系产生影响。会抛出异常集合作为函数声明的部分就具有非常重要的价值（前面已经简单提过空抛出的问题）。例如：

```
void function(int a ) throw (x2, x3)
```

在上例中，函数 function(int a)可能抛出两个异常类 x2 和 x3 的对象和从这两个异常类型派生的异常，不会抛出其他类型异常。当函数描述了可能抛出的异常时，即有效地为其调用者提供了一种保证。在函数的执行过程中，如果试图废止所做的保证，此意图会被转换成一个对 std::unexpected()的调用。unexpected()的默认意义是 std::terminate()，一般会转而调用函数 abort()。

在功能方面：

```
void function throw(x2, x3)
   {
   //部分代码
   }
```

它等价于：

```
void function()
try
{
  //代码
}
catch(x2)
{
   throw;
}
catch(x3)
```

```
{
    throw;
}
catch(...)
{
std::unexpected();
}
```

它的优点在于函数的声明属于界面（即接口），界面是函数的调用者明显可以看到的。函数的定义通常不是可用的内容。即使有机会访问所有库的源代码，广大程序员也不希望经常去查看库函数的源代码。另外，带有异常描述的函数，要比使用 try 块和 catch 块简捷得多。

如果函数声明中不包含异常描述，假定该函数可能抛出任何其他的异常，但又不希望抛出某些异常，可以使用空抛出声明形式（前面已提到）。

举例如下：

```
int function()              //可能抛出异常
int function()  throw()     //不会抛出任何异常
```

2. 异常描述检查

编译过程中可能会遗漏违反界面描述的情况。编译时检查可以完成大部分工作。异常描述的方式应该是假定函数将要抛出可能要抛出的所有异常。如果函数的声明中包含了异常描述，该函数的每个声明（或定义）都必须包含完全一致的异常类型集合的异常描述。

假如在函数声明时包含了 throw()，而在函数的定义时没有包含 throw()，则是错误的。例如：

```
int function() throw(std::bad_alloc)
......
int function()
{
......
}
```

以上内容就是错误的。

上面对异常描述进行了较详细的讲解，但读者也没必要为此恐慌。当某处发生异常时，不必强迫人们对相关的异常描述做完全的更新，不必强制性要求重新编译受影响的代码。大多数系统应该能够在一种部分更新的状态中，依靠对未预期异常的动态检查照常工作。对于大规模系统的维护是必不可少的要求，但大规模更新的代价极其昂贵。

尤其是当需要覆盖一个虚函数时，函数所带的异常描述必须至少是和那个虚函数的异常描述一样受限的。

总而言之，对于我们广大程序开发人员，异常处理是非常复杂的过程，我们只要掌握最简洁的使用形式即可，确保在使用过程中万无一失。而没有必要去追求使用形式的多样性和复杂性。

3. 未预期的异常

异常描述可能导致调用函数 unexpected()。如果不希望出现此种调用的情况，可在程序调试期间，通过细心地组织异常和界面描述，尽力避免此种调用。即尽力拦截对 unexpected()的调用，并使之无害化。

函数 unexpected()的声明如下：

```
void unexpected( );
```

C++标准要求表明：如果被抛出的异常不包括在 throw 列表中，unexpected()会被调用。当前的实现不支持这种情况，例如直接调用 unexpected()时，是通过调用该函数的句柄。如果 unexpected()函数在程序中被直接调用时，unexpected()的句柄是最新通过 set_unexpected()设置。其使用方法可参考例 8-4。unexpected()处理器不可能返回至它的调用者，它可能通过以下条件终止执行：

（1）异常描述中的类型列表中类型的对象，或任意类型的对象被抛出时，unexpected()处理器可以在程序中直接调用。

（2）抛出 bad_exception 类型的对象。

（3）在调用 terminate()、abort()或者 exit()过程中。

函数 set_unexpected()的声明如下：

```
unexpected_handler  set_unexpected(  unexpected_handler  _Pnew  ) throw( );
```

参数 Pnew 是指定的被调用的函数，函数返回值是原有的被调用的函数。

通常，设计良好的子系统 Y 常常将它所有的异常从类 Yerr 中派生。例如：

```
class Some_Yerr: public Yerr{/*......*/}
```

包含如下声明的函数：

```
void function() throw(Xerr, Yerr, exception);
```

把所有的 Yerr 传递给它的调用者。特别是 function()可以通过将 Some_Yerr 传给其调用者的方式去处理它。函数 function()中的 Yerr 不会触发 unexpected()。由标准库中抛出的所有异常均是由 exception 派生出的。

4．异常映射

当某个异常发生时，即终止程序的策略过于严厉。此时，需要将 unexpected()的行为修改为其他的能接受的方式。

最简便的方法是将标准库异常 std::bad_exception 加入某个异常描述中。此时，unexpected()会抛出一个 bad_exception，而不去调用某个难以处理的函数。

举例如下：

```
class X{};
class Y{};
void function() throw(X, std::bad_exception)
{
throw Y();  //抛出“bad”异常
}
```

异常描述将捕捉未预期的异常 Y，之后再抛出一个 bad_exception 类型的异常。

类 bad_exception 异常通常没什么“坏”的地方，仅仅提供一种机制，并没有调用 terminate()那样“冷酷”。但丢失了引起问题的那个异常的所有信息。

8.1.7　未捕捉的异常

前面已经讲过，如果某个异常未被捕捉，会调用函数 std::terminate()。当异常处理机制发现堆栈损坏时，或者在由某个异常抛出而导致的堆栈回退过程中，被调用的析构函数企图通过抛出异常而退出时，都会调用函数 std::terminate()。通俗地讲，未捕捉的异常是没有为其准备 catch()处理函数的异常，或是 throw()所执行的析构函数抛出的异常。此异常将造成 std::terminate()函数被调用，该函

数调用 std::abort()以终止程序。

未预期的异常由 set_unexpected()确定的_unexpected_handler 处理。类似情况是，对未捕捉的异常响应应该由_uncaught_handler 确定，而它由<exception>中的 std::set_terminate()设置：

```
typedef void(*terminate_handler)();
terminate_handler set_terminate(terminate_handler);
```

返回值给出的是通过 set_terminate()设置的前一个函数。

提出 terminate()的原因是，在少数情况下，必须能用较粗糙的错误处理技术终止异常处理过程。例如：terminate()可能被用于结束某个进程，甚至用于重新初始化一个系统。采用 terminate()的意图是作为一种严格限制，当由异常处理机制所实现的错误恢复策略失败之后，应该进入另一个容错策略层次，即可应用之。

通常，按默认的规定，terminate()会调用函数 abort()。对大部分用户而言，这是一种正确选择。特别是排除错误阶段。

_uncaught_handler 被假定是不会返回其调用者的，否则 terminate()将调用 abort()。函数 abort()表示从程序中非正常退出。使用函数 exit()终止程序，函数的返回值可以向外围系统表明程序是正常结束还是非正常结束。

在程序因为未捕捉的异常而终止时，是否调用有关的析构函数的问题将由具体实现决定。不调用析构函数是至关重要的，将使程序能够从排错系统中重新唤醒。在其他系统中，系统的结构决定在检索异常处理器的过程中，不调用析构函数是不可能的。

如果要在发生未捕捉异常时保证进行彻底的清理工作，可以在所有真正需要关注的异常处理器之外，再为 main()添加捕捉一切的处理器。举例如下：

```
int main()
try{
   // code
}
catch(std::range_error)
{
   // code
   cerr<<"range error : not agan!\n";
}
catch(std::bad_alloc)
{
   cerr<<"new ran out of memory\n";
}
catch(...)
{
   //...
}
```

上述形式的代码，即可捕捉程序可能发生的所有异常，除在全局变量的构造和析构中抛出的异常之外，没有办法捕捉全局变量初始化期间抛出的异常。对于非局部静态对象初始化中出现的 throw，获取控制的唯一办法是通过 set_unexpected()，即应该尽可能地避免全局变量的其他原因。

一旦异常被捕捉，通常无法确定其抛出位置，也意味着信息的丢失。在有些 C++开发环境中，对某些程序和某些开发人员而言，信息的丢失会导致不捕捉程序的设计本身无法恢复的异常。

总结：8.1 节主要讲述了异常及其相关的部分概念和异常处理的基本原理。希望读者能认真阅读，对异常处理能有较清楚地认识即可。任何知识的学习均是循序渐进的过程，开始的时候最重要的是了解和理解其基本概念和基本原理，随着今后实践机会和实践经验的逐渐增加，对异常处理的认识会更加深刻。希望读者不要指望一下子学会异常处理的全部内容。同样，对于 C++ STL 的其他内容也是如此，都需要经历循序渐进的过程。

8.2　异常类及几个重要问题

标准模板库（STL）提供了异常类（exception）。其余的异常类均派生于一个基类 exception。本节涉及的内容包括：类 exception，函数 abort()，堆栈解退，错误代码，异常的迷失，异常处理的局限性。

8.2.1　类 exception

STL 中类 exception 除了构造函数、析构函数之外，仅有一个虚成员函数 what()。函数 what()用以获取“型别本身以外的附加信息”，函数返回值是一个以 null 结束的字符串。类 exception 的声明如下：

```
class exception {
public:
   exception( ) throw( );
   exception(const exception& right) throw( );
  exception& operator=(const exception& right) throw( );
  virtual ~exception( ) throw( );
   virtual const char *what( ) const throw( );
   };
```

在上述类声明中，有 3 个构造函数和 1 个析构函数，均被设置了空抛出（throw()）。其虚成员函数 what()也被设置了空抛出。

函数 what()返回的字符串可以被转换为 wstring 并便于被显示。what()返回的 C-string 在所属的异常对象被摧毁后，即不再有效。函数 what()是唯一被用来描述异常种类的虚函数。唯一可能实现的另一个异常评估手段是根据异常的精确型别，自己得出结论。最简单的例子如下：

```
try{
//...
}
catch(const std::exception& error ){
std::cerr<<error.what()<<std::endl;
}
```

前面已经说明，C++异常的主要目的是为设计容错程序提供语言级支持，即异常使得在程序设计中包含错误处理功能更简便，防止事后采取严格的错误处理方式。异常的灵活性和相对方便性激励程序员在条件允许的情况下在程序中加入错误处理功能。异常是一种特性，类似于类，可以改变程序员的编程方式。

在头文件<exception>中定义了异常类，C++可以将其作为其他异常类的基类。程序会引发 exception，同样也可以将 exception 类用作基类。what()是虚函数，因为生成从 exception 类派生的自定义类可以重新定义该函数。举例如下：

```
#include  <exception>
class bad_cl: public std::exception
{
public:
const char* what(){return "bad argument to bad_cl ";}
......
};
```

可以使用 std::exception 类处理派生的异常，在同一个基类处理程序中捕获此类信息。

```
try{
 //...
}
catch(std::exception& e)
{
cout<<e.what()<<endl;
}
```

当然也可以分别捕获这些异常，前面已有介绍。

STL 定义了很多基于 exception 的异常类型。头文件 exception 提供 bad_exception，供函数 unexpected()使用。

1. stdexcept 异常类

头文件 stdexcept 定义其他几个异常类。首先，该文件定义了 logic_error 类和 runtime_error 类，均是以公有方式从 exception 类派生而来。类 logic_error 的声明如下：

```
class logic_error : public exception {
public:
  logic_error(const string& message);
};
```

该类的构造函数接受一个 string 对象作为参数，该参数为 what()函数提供了返回值（message.data）。

类 runtime_error 的声明如下：

```
class runtime_error : public exception {
public:
  runtime_error(const string& message);
};
```

同样，类的构造函数的参数是 string 型对象。该参数为 what()函数提供了返回值（message.data）。

除上述两类之外，还有类 domain_error，invalid_argument，length_error 和 out_of_range。这些类和 logic_error 的情况类似。

对于 domain_error 类，数学函数通常有定义域和值域。定义域由参数的可能值组成，值域由函数可能的返回值组成。例如：如果程序员在开发程序时，将不合理的参数传递给函数 std::sin()，会导致程序抛出 domain_error 类异常。

异常类 invalid_argument 指出给函数传递一个意外的值。例如：如果函数希望接受一个字符串，其中每个字符要么是 0 要么是 1，当传递的字符串中包含其他字符时，该函数会引发 invalid_argument 类异常。

异常类 length_error 用于指出没有足够的空间来执行所需的操作。例如：string 类的 append()方法在合并获取的字符串长度超过最大允许长度时，会引发 length_error 异常。

异常类 out_of_range 通常用于指示索引错误，即索引的数值超出了正常容量的上下限。例如：对于某数组，使用 operator()[]进行访问时，使用的索引无效会引发 out_of_range 异常。

异常类 runtime_error 描述了可能在运行期间难以预计和防范的错误，每个类的名称指出其报告的错误类型：

```
range_error
overflow_error
underflow_error
```

每个类分别有自己的构造函数，构造函数的参数用于虚函数 what()返回的字符串。

下溢（underflow）错误会出现在浮点数计算中。通常，存在浮点类型可以表示的最小非零值，计算结果比该值小，会导致下溢错误。整型和浮点型可能发生上溢错误，当计算结果超出某种类型能够表示的最大数量级时，会发生上溢错误。尤其计算结果可能不在函数允许的范围内，如果不发生上溢或下溢错误，此时可以使用 range_error 异常。

通常，logic_error 系列异常表明存在可以修复的问题，而 runtime_error 系列异常表明存在无法避免的问题。这些错误具有相同的常规特征，区别在于：不同的类名能够分别处理每种异常。同时，继承关系能处理些问题。并且捕获异常的形式是多种多样的。下面举例说明：

```
try{
//...
}
catch(out_of_bounds?& oe)
{
//....
}
catch(logic_error& oe)
{
//....
}
catch(exception& oe)
{
//....
}
```

如果上述形式还不能满足开发者的需要，程序开发者需要从 logic_error 类或 runtime_error 类派生新类，以确保异常类归纳入同一个继承层次结构中。

2. bad_alloc 异常和 new

对于处理使用 new 时可能出现的内存分配问题，C++提供了两种可供选择的方式：使 new 在无法满足内存请求时返回空指针；使 new 引发 bad_alloc 类型异常。头文件<new>中包含了 bad_alloc 的类型声明，从异常类 exception 派生而来。在具体实现时，根据实际情况选取一种方式即可。也可以使用编译器开关或其他方法，让开发者选择自己喜欢的方式。下面给出一个最简单的例题，说明异常捕捉后，程序将显示 what()方法返回的消息，之后终止运行。如果没有捕捉到异常，程序将继续执行，并查看返回值是否为空指针。

3. 示例

针对上述内容，下面给出一个关于处理异常的简单例题（例 8-1）。尽管在 C 语言中不希望使用 goto 语句，但为了便于测试各种异常类别，还是使用了两次 goto 语句。

例 8-1

```
#include <iostream>
#include <exception>
using namespace std;
void main()
{
logic_error a("Exception: logic_error.");                    //定义错误对象
runtime_error b("Exception: runtime_error.");
domain_error c("Exception: domain_error.");
invalid_argument d("Exception: invalid_argument.");
length_error e("Exception: length_error.");
range_error f("Exception: range_error.");
overflow_error g("Exception: overflow_error.");
underflow_error h("Exception: underflow_error.");
out_of_range i("Exception: out_of_range.");
int switch_K;
cout<<"1. Exception: logic_error."<<endl;                    //输出提示信息
cout<<"2. Exception: runtime_error."<<endl;
cout<<"3. Exception: domain_error."<<endl;
cout<<"4. Exception: invalid_argument.."<<endl;
cout<<"5. Exception: length_error."<<endl;
cout<<"6. Exception: range_error."<<endl;
cout<<"7. Exception: overflow_error."<<endl;
cout<<"8. Exception: underflow_error."<<endl;
cout<<"9. Exception: Out_of_Range."<<endl;
cout<<"0. Exit."<<endl;
loop: cout<<".....Please Input 1-9: ....."<<endl;
try{
cin>>switch_K;                                               //输入选项
if(switch_K==0)
{
goto loop2;
}
switch(switch_K)
{
case 1:
throw a;                                                     //抛出异常（错误）
break;
case 2:
throw b;
break;
case 3:
throw c;
break;
case 4:
throw d;
break;
case 5:
throw e;
break;
```

```
case 6:
throw f;
break;
case 7:
throw g;
break;
case 8:
throw h;
break;
case 9:
throw i;
break;
}
}
catch(logic_error& e )                                    //捕获错误或异常
{
cout<<e.what()<<endl;
}
catch(runtime_error& e)
{
cout<<e.what()<<endl;
}
catch(domain_error& e)
{
cout<<e.what()<<endl;
}
catch(invalid_argument& e)
{
cout<<e.what()<<endl;
}
catch(length_error& e)
{
cout<<e.what()<<endl;
}
catch(range_error& e)
{
cout<<e.what()<<endl;
}
catch(overflow_error& e)
{
cout<<e.what()<<endl;
}
catch(underflow_error& e)
{
cout<<e.what()<<endl;
}
catch(out_of_range& e)
{
cout<<e.what()<<endl;
}
goto loop;
```

```
loop2: return;
}
```

程序的执行结果如下：

```
1. Exception: logic_error.
2. Exception: runtime_error.
3. Exception: domain_error.
4. Exception: invalid_argument..
5. Exception: length_error.
6. Exception: range_error.
7. Exception: overflow_error.
8. Exception: underflow_error.
9. Exception: Out_of_Range.
0. Exit.
.....Please Input 1-9: .....
1
Exception: logic_error.
.....Please Input 1-9: .....
2
Exception: runtime_error.
.....Please Input 1-9: .....
3
Exception: domain_error.
.....Please Input 1-9: .....
4
Exception: invalid_argument.
.....Please Input 1-9: .....
5
Exception: length_error.
.....Please Input 1-9: .....
6
Exception: range_error.
.....Please Input 1-9: .....
7
Exception: overflow_error.
.....Please Input 1-9: .....
8
Exception: underflow_error.
.....Please Input 1-9: .....
9
Exception: out_of_range.
.....Please Input 1-9: .....
0
Press any key to continue
```

总结：在例 8-1 中，测试了 STL 中所包含的各种异常派生类。通过反复地调用 throw 和 catch，尝试了各种错误类别的抛出和捕获。

8.2.2 调用 abort()

异常发生时（例如：除数为 0），处理方式之一是：调用 abort()函数。函数 abort()的原型位于头文件 cstdlib 中，典型实现是向标准错误流发送消息 abormal program termination，然后终止程序。之

后函数的返回值会通知操作系统处理失败。abort()是否要刷新文件缓冲区取决于实现。有时，如果程序员愿意的话，也可以使用函数 exit()。函数 exit()完成刷新缓冲区的功能，但不显示消息。

函数 abort()的原型如下：

```
void abort( void );
```

函数 exit()的原型如下：

```
void exit( int status );
```

函数 abort()和函数 exit()一旦被执行，程序即终止。在 Visual C++6.0 环境中，使用函数 abort()时会弹出异常退出对话框，而函数 exit()则不会。下面用例 8-2 来加以说明。

例 8-2

```
#include <iostream>
#include <cmath>
using namespace std;
double fun(double x, double y)
{
double result=0.0;
if(fabs(y)<=1e-4)
{
cout<<"y may equal to 0.0."<<endl;
abort();
}
else
result=x/y;
return result;
}
void main()
{
double x,y,z;
cout<<"Input number: x and y."<<endl;
while(cin>>x>>y)
{
if(x=='q'||y=='q')
break;
z=fun(x,y);
cout<<"fun(x,y) is : "<<z<<endl;
cout<<"Please input next set of numbers :"<<endl;
}
cout<<"Bye."<<endl;
return;
}
```

程序的执行结果如下：

```
Input number: x and y.
1
2
fun(x,y) is : 0.5
Please input next set of numbers :
3
5
```

```
fun(x,y) is : 0.6
Please input next set of numbers :
7
5
fun(x,y) is : 1.4
Please input next set of numbers :
8
1
fun(x,y) is : 8
Please input next set of numbers :
9
0
y may equal to 0.0.
```

之后弹出如图 8-1 所示的对话框。

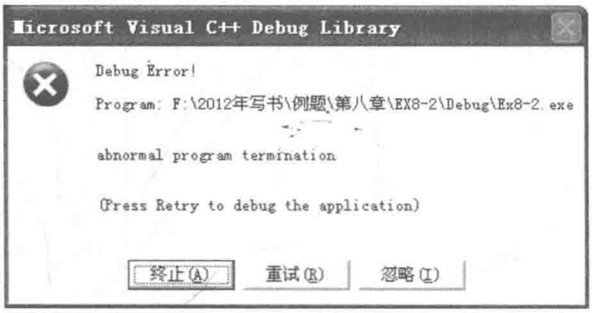


图 8-1　例 8-2 执行时弹出的异常对话框

替换例 8-2 中的代码，将函数 abort()替换为 exit()，则程序不会弹出上述的异常对话框。类似例 8-2 中除数为零的错误，依靠程序员编写程序来避免是不现实的。

8.2.3　堆栈解退

假如 try 块没有直接调用引发异常的函数，而调用对引发异常的函数进行调用的函数，则程序流程将从引发异常的函数跳转到包含 try 块和处理程序的函数。这些涉及堆栈解退。

C++程序通常是通过函数调用来实现的。C++通常通过把信息放在堆栈中来处理函数调用。程序将调用函数的指令地址放到堆栈中。被调用函数执行完毕之后，程序将使用该地址来确定从何处开始。函数调用将函数参数放到堆栈中。在堆栈中，函数参数被视为自动变量（auto）。如果被调用的函数创建新的自动变量，则这些变量也被填入堆栈中。如果被调用的函数调用另一个函数，后者的信息将被添加至堆栈中，依此类推。当函数结束时，程序流程将转至该函数被调用时存储的地址处，堆栈顶端的元素被释放。函数通常都返回到调用它的函数，同时每个函数都在结束时释放其自动变量。如果自动变量是类对象，则类的析构函数将被调用。

假定函数出现异常而终止，则程序将释放堆栈中的内存，但不会释放堆栈内第一个返回地址，而是继续释放堆栈，直至寻找到 try 块中返回的地址。之后，控制权将转到块尾的异常处理程序，而不是函数调用后面的第一条语句。该过程称为堆栈解退。

引发该机制的一个重要特性是：和函数返回一样，对于堆栈中的自动类对象，类的析构函数将被调用。函数返回仅仅处理该函数放在堆栈中的对象，而 throw 语句则处理 try 块和 throw 块之间整个函数调用序列放在堆栈中的对象。如果没有堆栈解退的特性，引发异常后对于中间函数调用放在

堆栈中的自动类对象，析构函数不会被调用。

8.2.4　错误代码

通常还有一种办法比异常更加灵活——使用函数的返回值来指出问题。例如：ostream 类的 get()成员函数通常返回下一个输入字符的 ASCII 码，到达文件尾时，将返回特殊值 EOF。这种方法通过告知调用程序该调用是否成功，可使得程序采取异常终止之外的其他措施。通过依靠程序员检查函数的返回值，虽然灵活，但广大程序员并不经常这样做。例如：为了保证程序的短小精悍，有些程序不会检查 new()产生的指针是否为空，还有，不会检查 cout 是否顺利地处理了输出。

在某处存储返回条件的方法是使用全局变量。可能的函数在出现问题时将该全局变量设置为特定的值，而调用程序可以检查变量。传统的 C 语言数学库使用的方法即是全局变量，其名称为 errno。当然，需要确保其他函数没有将该全局变量用于其他目的。

8.2.5　异常的迷失

异常被引发之后，在两种情况下会导致问题。

（1）如果异常是在包含异常规范的函数中引发的，必须与规范列表中的某种异常匹配，否则称为意外异常。在默认情况下，将导致程序异常终止。

（2）如果异常不在函数中引发，那么该异常必须被捕获。如果没有被捕获，则异常被称为未捕获异常。默认情况下，会导致程序异常终止。

对于意外异常和未捕获异常的反应，则可以根据情况由程序员自行设定。

1. 未捕获异常

未捕获异常不会导致程序立刻异常终止。相反，程序将调用函数 terminate()。在默认情况下，terminate()调用 abort()函数。指定 terminate()应调用的函数，实现修改 terminate()的行为。下面介绍如何调用函数 set_terminate()和 terminate()。这两个函数通常在头文件<exception>中声明。

```
typedef void (*terminate_handler)();
terminate_handler  set_terminate(terminate_handler f) throw()
void terminate();
```

其中，typedef 使 terminate_handler 成为一种类型的名称：指向没有参数和返回值的函数的指针。函数 set_terminate()的参数必须是：不带任何参数且返回类型为 void 的函数名称或函数地址。执行时会返回参数的地址。如果调用 set_terminate()函数很多次，则 terminate()将调用最后一次 set_terminate()设置的函数。

假定希望未捕获的异常会使程序打印一条消息，之后调用 exit()，将退出状态值设置为 5。在使用时必须包含头文件<exception>。

设计一个上述功能的函数 myQuit()，代码如下：

```
void  myQuit()
{
cout<<"Terminating test because of  uncaught exception."<<endl;
exit(5);
}
```

在 main()函数的起始部分，设置 set_terminate(myQuit)。一旦程序中某些异常被触发，并没有被捕获，程序会调用 terminate()，而函数 terminate()会触发调用函数 myQuit()。下面用例 8-3 来加以说明。

例 8-3

```
#include <exception>
#include <iostream>
using namespace std;
void myQuit()
{
cout<<"Terminating test because of uncaught exception."<<endl;
exit(5);
}
void main()
{
terminate_handler oldHandler =set_terminate(myQuit);
int switch_K;
logic_error a("Exception: logic_error.");
runtime_error b("Exception: runtime_error.");
cout<<".....Please Input 0 for exit.      ....."<<endl;
loop:cout<<".....Please Input 1 or else number: ....."<<endl;
try{
cin>>switch_K;
if(switch_K==0)
{
goto loop2;
}
switch(switch_K)
{
case 1:
throw a;
break;
default:
throw b;
break;
}
}
catch(logic_error& e )
{
cout<<e.what()<<endl;
}
catch(...)
{
terminate();
}
goto loop;
loop2: return;
}
```

程序的执行结果如下：

```
.....Please Input 0 for exit.      .....
.....Please Input 1 or else number: .....
1
Exception: logic_error.
.....Please Input 1 or else number: .....
```

```
2
Terminating test because of uncaught exception.
```

2. unexpected()处理

作为程序员，应该明确哪些异常是必须要捕获的。在默认情况下，未捕获的异常将导致程序异常终止。

通常原则上异常规范需要包含函数调用的其他函数引发的异常。例如：假设函数 A 调用函数 B，函数 B 会触发某个对象异常；则在异常规范中，均应包含该对象。

如果程序引发了异常规范中没有的异常，此时行为与未捕获的异常极其相似。如果发生意外异常，程序会调用 unexpected()函数。函数 unexpected()会调用函数 terminate()，而函数 terminate()默认情况下会调用 abort()函数。和 set_terminate()相对应，STL 提供了函数 set_unexpected()用于修改函数 unexpected()的行为。函数 unexpected()和 set_unexpected()同样是包含在头文件<exception>中的。

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler _Pnew ) throw( );
void unexpected( );
```

和函数 set_terminate()不同，函数 set_unexpected()的参数是严格受限制的。通常，unexpected_handler 函数可以有以下行为：

（1）调用 terminate()、abort()或 exit()，从而终止程序；

（2）触发异常。引发异常的结果取决于 unexpected_handler 函数所引发的异常及引发异常的函数的异常规范。一般包括以下几种情况：

① 如果新引发的异常与原来的异常规范匹配，则程序会开始进行正常处理，即寻找与新引发的异常匹配的 catch 块。基本上此方法将用于预期的异常取代意外异常。

② 新引发的异常与原有的异常规范不匹配，且异常规范中不包括 std::bad_exception 类型，则程序还是会调用 terminate()函数。bad_exception 是从 exception 派生而来的，声明在头文件<exception>中。

③ 新引发的异常与原有异常规范不匹配，原来的异常规范中包含 std::bad_exceptipon 类型，不匹配的异常会被 std::bad_exception 异常取代。

简而言之，需要捕获所有的异常，可以用如下方法处理：

（1）必须包含头文件<exception>；

（2）设计一个替代函数，将意外的异常转换为 bad_exception 异常，该函数的原型如下：

```
void myunexpected()
{
throw std::bad_exception();
}
```

使用 throw 并且不指定异常将导致重新引发原来的异常。若异常规范中包含此类型，该异常将被 bad_exception 对象取代。

在函数 main()的开始位置，将意外异常操作指定为调用自定义的 myunexpected()。最后，将 bad_exception 类型包括在异常规范中，并添加以下 catch 块。

```
catch(bad_exception& e)
{
...
}
```

这种异常处理如例 8-4 所示。

例 8-4

```
#include <exception>
#include <iostream>
using namespace std;
void myUnexpected()
{
cout<<"myUnexpected() is done."<<endl;
throw bad_exception();
exit(5);          //不再执行
}
void main()
{
terminate_handler oldHandler =set_unexpected(myUnexpected);
int switch_K;
runtime_error a("Exception: runtime_error.");
bad_exception b("Exception: bad_exception.");
cout<<".....Please Input 0 for exit.     ....."<<endl;
loop:cout<<".....Please Input 1 or 2: ....."<<endl;
try{
cin>>switch_K;
if(!switch_K)
{
goto loop2;
}
switch(switch_K)
{
case 1:
throw a;
break;
case 2:
unexpected();
break;
}
}
catch(runtime_error& e )
{
cout<<"catch: "<<e.what()<<endl;
}
catch(bad_exception& e)
{
cout<<"catch: "<<e.what()<<endl;
exit(0);
}
goto loop;
loop2: return;
}
```

程序的执行结果如下：

```
.....Please Input 0 for exit. .....
```

```
.....Please Input 1 or 2: .....
1
catch: Exception: runtime_error.
.....Please Input 1 or 2: .....
2
myUnexpected() is done.
catch: bad exception
```

8.2.6 异常处理的局限性

前面讲述了异常处理的重要性和有效性。但异常处理也具有局限性。前面的讨论显然说明，应该在程序设计时即加入异常处理功能，而不是以后添加。因为，这样做会增加程序代码，降低程序的运行速度。异常规范不适用于模板，模板函数引发的异常可能随特定的具体化而异。异常和动态内存分配不能总是协调工作。例如：

```
void test(int n)
{
string msg("I was trapped in endless loop.");
…
if(oh_no)
{
throw exception();
}
…..
return;
}
```

在上面的函数中，一旦执行了 throw 函数，即终止了函数 test()。string 型变量仍然使其析构函数被调用，得益于堆栈解退，内存被正确的管理。

但是还会有其他的情况，如例 8-5 所示。

例 8-5

```
void test2(int n)
{
   double *ar=new double[n];
   ......
   if(oh_no)
      {
        throw exception();
      }
      ….
      delete[]  ar;
      return;
}
```

在例 8-5 中，解退堆栈时，需要删除堆栈中的变量 ar。函数提前终止意味着函数末尾的 delete[] 语句无法执行。指针消失，但该指针指向的内存块未被释放，并且不能被访问。简而言之，这块内存泄漏了。

这种简单的内存泄漏是可以避免的。可以在引发异常的函数中捕获该异常，在 catch 块中包含一些清理代码，然后重新引发异常。举例如下：

```
void test2_i(int n)
{
```

```
    double *ar=new double[n];
    ......
    try{
    if(oh_no)
       {
         throw exception();
         }
    }
    catch(exception & e)
    {
    delete [] ar;
    throw;
    }
    ......
    delete[]  ar;
    return;
}
```

异常处理对于某些项目极为重要。但同时也增加了编程的工作量，增大了程序，降低了程序的速度。编译器对异常的支持及用户的经验还没达到成熟的程度，应有节制地使用该特性。

提示：在 STL 中，异常处理的复杂程度越来越高。主要原因在于文档没有对异常处理进行解释或解释不恰当。任何熟练使用现代操作系统的人都遇到过未处理的异常导致的错误和问题。这些错误是程序员通常面临的异常艰难的战役。需要不断了解库的复杂性，熟悉各种异常的引发原因及处理措施。

尤其对于程序员新手，理解库中的异常处理比学习 C 语言更难。要开发出优秀的软件，必须花时间了解库和类中的复杂内容，就像必须花时间学习 C++本身一样。通过库文档和源代码了解到的异常和错误处理细节会使程序员受益。

8.3 处理异常详述

异常处理的基本思想是简化程序的错误代码，为程序健壮性提供一个标准检测机制。前面章节已经讲述过：C++使用 throw 关键字来产生异常，try 关键字用来检测的程序块，catch 关键字用来填写异常处理的代码；异常可以由一个确定类或派生类的对象产生；C++能释放堆栈，并可清除堆栈中所有的对象。

本节将详细地讲述异常的处理、异常的声明及异常的访问。

对于可预料但不可避免的程序错误，不能任其发生而无所作为，要将消极的等待变为积极预防，还要将预防的处理内容归纳整理，分门别类地做成类。不能仅凭编程经验这种个体性、偶然性的做法，而是要凭借人人均掌握的规范，实现圆满的处理。通常规范是指在函数处理过程中布设陷阱，异常一旦触发，必然会被异常处理收容，统一归口处理。

前面已经讲过，异常处理有以下 3 种方式：

（1）在出现异常时，直接调用 abort()或者 exit()；

（2）通过函数的返回值来判断异常；如果函数有多个返回值，会浪费不必要的判断多个返回值的时间开销；

（3）通过函数 try()-catch()的结构化异常处理而完成。

8.3.1　异常处理的实现机制

异常处理的实现离不开 3 个关键字：throw，try()和 catch()。抛掷异常是在特定的函数中，try 块应该包含该函数，catch 块根据不同的情况进行相应的处理。其形式大概如下：

```
function()
{
throw 表达式;
}
try{
...
function();
...
}
catch(异常类型声明)
{
...
}
catch(异常类型声明)
{
...
}
```

前面章节给出了很多较详细的例子，此处再给出一例，见例 8-6。

例 8-6

```
#include <iostream>
#include <exception>
using namespace std;
float mydiv(float x,float y)
{
if(y==0.0)
throw y;
return x/y;
}
void main()
{
try{
   cout<<"5/2="<<mydiv(5.0,2.0)<<endl;
   cout<<"8/0="<<mydiv(8.0,0.0)<<endl;
   cout<<"7/1="<<mydiv(7.0,1.0)<<endl;
}
catch(float)
{
  cout<<"exception of dividing zero.\n"<<endl;
}
cout<<"test ok!"<<endl;
}
```

程序的执行结果如下：

```
5/2=2.5
exception of dividing zero.
```

```
test ok!
Press any key to continue
```

8.3.2 异常处理语句的语法

关键字 throw、try、catch 分工明确，各司其职。throw 负责发现异常，并抛掷异常对象；如果放在函数声明中又被称为异常接口声明。try 设置一个侦错范围，又称为保护段，其实是划定一个跳跃的边界。catch 负责处理捕获的异常，所抛掷的异常对象并非建立在函数栈上，而是建在专用的异常栈上，可以跨越函数而传递给上层。注意以下条目：

- 每个 catch()相当于一段函数代码；
- 每个 throw 相当于一个函数调用；
- 每个 try 块至少跟一个 catch()；
- 每个程序可以设置个数不定的 try、throw 和 catch。只有逻辑上的呼应，而无数量上的对应关系，且不受所在函数模块限制；
- 异常抛掷点往往距异常捕获点很远，可以不在同层模块中；
- 甚至有的 throw 不明确，可能来自于系统函数或标准库中；
- try 块可以并列，可以嵌套。

8.3.3 异常处理不唤醒

在保护段执行期间发生异常，则立即抛掷，由相应的 catch 子句捕获处理。抛掷—捕获间的代码被越过，不会被执行。程序从 try 块后跟随的最后一个 catch 块子句后面的语句继续执行。

异常处理通俗地说是将检测与处理分离，以便各司其职，灵活搭配工作。检测与处理之间的联系依靠“异常类型”。异常通过 throw 创建一个异常对象并抛掷。可能抛出异常的程序段嵌在 try 块中，按正常的顺序执行到达 try 语句时，执行 try 块内的保护段。如果在保护段执行期间没有引起异常，在 try 块之后的 catch 子句不被执行。程序会从 try 块后最后一个 catch 子句后面的语句继续执行。catch 子句按其在 try 块后出现的顺序被匹配之后，catch 块将被捕获并处理异常。

如果没有找到匹配的处理器，运行函数 terminate()会被调用，默认功能是调用 abort()函数终止程序。

8.3.4 函数声明

关键字 throw 还可以用于函数声明，前面章节已经有过介绍。例如：

```
void fun() throw(int);
void fun() throw();
void fun()
```

在函数声明时，带有 throw 关键字，在函数定义时必须同样出现。例如：

```
void fun() throw(int)
{

}
```

8.3.5 使用异常

什么时候使用异常？这是很难回答的问题。

首先，不能使用异常完全代替返回值，因为返回值的含义不一定仅是成功或失败，还可能是一个选择的状态。

其次，在特定情况下，异常能否发挥它的优点，该优点正好不能使用其他技术实现。

异常还有一些规则，现在介绍给各位读者。

（1）异常机制只能用于处理错误；

（2）通常不要使用 goto 语句或 switch 语句跳转至 try 块或 catch 块内；

（3）不要显式地抛出 NULL；

（4）如果函数声明时，指定了具体的异常类型，只能抛出指定类型的异常；

（5）如果函数声明时指定了异常类型，在编译单元中该函数的声明必须有同样的指定；

（6）异常只能在初始化之后且程序结束之前抛出；

（7）throw 语句本身不允许引发新的异常；

（8）空的 throw 语句只能出现在 catch 语句块中。

（9）所有的流程中显式地抛出异常应该有一个类型兼容的处理程序；

（10）至少需要有一个程序处理所有其他针对处理的异常；

（11）若 try-catch 语句块中有多个处理程序，或派生类和部分或全部基类的 function-try-block 块有多个处理程序，处理程序的顺序应该是先派生类后基类；

（12）如果 try-catch 语句块或者 function-try-block 块有多个处理程序时，catch(...)处理程序（捕获所有异常）应该放在最后。

（13）若异常对象为类的对象时，应该通过引用来捕获；

（14）若构造函数和析构函数是 function-try-block 结构的，在 catch 处理程序中不能引用该类或其基类的非静态成员；

（15）析构函数退出之后，不允许还有未处理的异常。

8.4　异常的特殊处理函数

前面章节已经简述了一些和异常处理相关的 STL 函数。本节详细讲述函数 terminate()、unexpected()和 uncaught_exception()的使用方法。

异常处理机制主要依赖两个函数：terminate()和 unexpected()。而处理错误主要依赖于异常处理机制本身。

1. 函数 terminate()

在下述情况下，异常处理必须被放弃，而采取较弱的错误处理方法。

（1）当异常处理机制调用的用户函数包括一个 uncaught_exception 时，在抛出完整的信息之后，异常被捕获之前的时间段内；

（2）当异常处理机制不能发现抛出的异常句柄时，堆栈中退出的对象的析构函数存在一个异常时；

（3）当函数被执行时，而此函数使用 atexit 注册过；

（4）当一个抛出的表达式尝试重新抛出异常并且没有异常被处理时；

（5）当 unexpected 函数抛出一个异常，而该异常不被前面的异常说明允许，并且 std::bad_exception 不包括在异常描述中；类 bad_exception 的声明如下：

```
class bad_exception : public exception
{
};
```

（6）当默认的 unexpected_handler 被调用时。

在上述条件下，函数 terminate()可以被终止。若没有匹配的句柄被发现，在函数被调用之前，不确定堆栈中的对象是否退出。在所有的其他条件下，堆栈中的对象在函数 terminate()调用之前，不应该从堆栈中退出。当确定退栈过程最终会引起调用 terminate()时，制订措施不允许过早的完成堆栈的退出。

2. unexpected()函数

如果一个函数带有一个异常描述，抛出异常时，异常描述没有被列表，则函数 void unexpected()被迅速调用。函数 unexpected()不应该返回，但它能抛出一个异常。如果它抛出新的异常，异常描述是必要的。当抛出或重新抛出一个异常时，异常描述不允许以下情况发生：如果异常描述不包括类 bad_exception，函数 terminate()会被调用，否则被抛出的异常是可以被自定义的异常类对象替代并继续搜索另一个句柄的，在调用函数过程中，其异常描述是违背的。

异常描述保证仅仅明确列出的异常被抛出。如果异常描述包括类型 std::bad_exception，则不会被列出的任何异常，在函数 unexpected()中将被 std::bad_exception 替代。

类 std::bad_exception 的声明如下：

```
class bad_exception  : public exception
 {
};
```

3. 函数 uncaught_exception()

函数 uncaught_exception()的函数声明如下：

```
bool  uncaught_exception( );
```

前面章节已经讲过，此函数可以放在析构函数中。

函数 uncaught_exception()返回 true，被抛出的对象被评估之后，直到相匹配的句柄完成异常声明的初始化。它包括了退栈的过程。如果异常被重复抛出，重新抛出异常之处，uncaught_exception()返回 true，直到重新抛出的异常被再次捕获。

8.5 本章小结

本章主要讲述了 4 部分内容。第 1 部分讲述了异常的概念和基本思想，第 2 部分对异常类和几个重要问题进行了简要阐述，第 3 部分对如何处理异常进行详细阐述，第 4 部分讲述了异常处理的特殊函数。通过这些内容的描述，尽最大可能对 C++语言和 C++ STL 中的异常处理实现了较全面的阐述。

提示：本章的学习重点应该是关于异常处理的概念和异常处理的方法、异常类的学习及对重要问题的阐述。通过对这些内容的深刻理解，并逐步掌握异常处理的几个关键函数，经过长时间的实践锻炼，最终必能熟练掌握异常处理的内容和方法。

第 9 章　通用工具

C++标准库的通用工具由短小精悍的类和函数构成，执行最一般化的功能。头文件<utility>是很小的头文件。本章对于通用工具库主要涉及比较、对组、动态内存管理、日期和时间、通用型别、数值极值等各方面的功能模块和函数。这部分内容在 ISO-IEC-14882-2003_C++中的第 20 款（clause）中有明确描述，还有部分辅助函数在头文件<algorithm>中声明，但按 STL 的要求，这些函数不能称为算法。

9.1　通用工具库简介

本节主要描述模板参数的要求。主要讲述在模板中类型的使用以及内存配置器的性能和要求。本节内容主要分 7 个部分：相等比较，小于比较，复制构造，默认构造，配置器要求，运算符和对组。其中“对组”是新概念。

9.1.1　相等比较

许多模板类均定义了“==”运算符。

在表 9-1 中，T 是一种类型，该类型是由 C++模板提供的。表中的 a、b、c 是类型 T 的数值。

表 9-1　相等比较运算符的说明

表 达 式	返 回 类 型	性能和要求
a= =b	convertible to bool	= = 是一种相等关系，即它满足以下性质：
		—For all a,　a= =a
		—If a = = b, then b = = a
		If a = = b and b = = c, then a= = c

9.1.2　小于比较

在关于“小于”比较的表 9-2 中，T 是由 C++程序开发人员提供的一种模板类型，a 和 b 是类型 T 的值。

表 9-2　小于比较运算符说明

表 达 式	返 回 类 型	性 能 要 求
a < b	convertible to bool	< 是一种严格的弱序关系

和“小于”操作相关的还有!=、>、<=、>=4 个比较操作符。利用操作符“==”和“<”完成的。这几个函数均定义在<utiliy>中。虽然运算是多个，但仅需定义“<” 和“==”，即可使用这些运算符。在使用时还需要添加 using namespace std::rel_ops。这些运算符均定义在 std 的次命名空间 rel_ops 中。如此安排，主要是防止用户定义的全局命名空间中的同类型操作待发生冲突。使用命名空间 std::rel_ops 之后，新的比较运算符就轻松到手了，无须使用复杂的搜寻规则来引用它们。

运算符!=、>、<=、>=的声明如下：

```
namespace std{
namespace rel_ops{
template<class T> bool operator !=(const T& , const T&);
template<class T> bool operator>(const T&, const T&);
template<class T> bool operator <=(const T& , const T&);
template<class T> bool operator>=(const T&, const T&);
  }
}
```

C++ STL 还可以使用对组（pair）进行比较运算，其声明形式如下：

```
template <class Type1, class Type2> bool operator!=(
const  pair<Type1, Type2>&  _Left ,  const  pair<Type1, Type2>&  _right
);
```

参看例 9-1，对“!=”运算符进行举例说明。其余运算符仅需将程序中的运算符号替代即可。

例 9-1

```
#include <utility>
#include <iomanip>
#include <iostream>
int main( )
{
  using namespace std;
  pair <int, double> p1, p2, p3;
  p1 = make_pair ( 10, 1.11e-1 );                    //创建对组
  p2 = make_pair ( 1000, 1.11e-3 );
  p3 = make_pair ( 10, 1.11e-1 );                    //创建对组
  cout.precision ( 3 );                              //设置输出精度
  cout << "The pair p1 is: ( " << p1.first << ", "
      << p1.second << " )." << endl;
  cout << "The pair p2 is: ( " << p2.first << ", "
      << p2.second << " )." << endl;
  cout << "The pair p3 is: ( " << p3.first << ", "
      << p3.second << " )." << endl << endl;
  if ( p1 != p2 )
    cout << "The pairs p1 and p2 are not equal." << endl;
  else
    cout << "The pairs p1 and p2 are equal." << endl;
  if ( p1 != p3 )
```

```
        cout << "The pairs p1 and p3 are not equal." << endl;
    else
        cout << "The pairs p1 and p3 are equal." << endl;
}
```

程序运行结果如下：

```
The pair p1 is: (10, 0.111 ).
The pair p2 is: (1000, 0.0011 ).
The pair p3 is: (100, 0.0111 ).
The pairs p1 and p2 are not equal.
The pairs p1 and p3 are not equal.
```

部分版本采用两个不同的参数型别来定义上述模板（template）。

举例如下：

```
namespace std{
    template<class T1, class T2>
      inline bool operator!=(const  T1& x , const T2& y)        //运算符!=()
        {
return !(x==y);
}
…
}
```

此时，虽然两个操作数的型别不同，但它们之间可以比较。但这不是 C++标准库的所支持的。读者可以像例 9-2 所示那样，用不同的数据类型数据来比较一下。

例 9-2

```
#include <iostream>
#include <utility>
using namespace std;
void main()
{
int x=0, y=0;
float z=0,w=0;
x=1;
if(x>y)
{
cout<<"x is bigger than  y. "<<endl;
}
else
{
cout<<"x is not bigger than y. "<<endl;
}
y=2;
if((x,y)!=(y,z))
{
cout<<"(x,y)!=(y,z)."<<endl;
}
else
{
cout<<"(x,y)==(y,z)."<<endl;
}
}
```

程序输出结果如下：

```
x is bigger than  y.
(x,y)!=(y,z).
```

9.1.3 复制构造

在表 9-3 中，T 是 C++程序提供的类型模板，t 是类型 T 的值。并且，u 是常量类型 T 的值。

表 9-3 可复制构造的性能

表 达 式	返 回 类 型	性 能 要 求
T(t)		t 等效于 T(t)
T(u)		u 等效于 T(u)
t.~T		
&t	T*	表明 t 的地址
&u	const T*	表明 u 的地址

9.1.4 默认构造

默认构造器不是必需的。某种容器类的成员函数表明签名特定化了其默认构造器作为一个默认的参数。如果所有签名中只有一个使用默认参数被调用，T()应该是一个定义良好的表达式。

9.1.5 配置器要求

STL 库描述了配置器要求的标准集，其中包含了内存配置模型的所有对象。该信息包括指针类型知识，对象大小类型，区别的类型，还有内存配置和存储单元分配等。就配置器而言，所有的容器均被参数化。表 9-4 描述了通过内存配置实现类型控制的要求。

表 9-4 描述性可变定义

变 量	定 义
T , U	任何类型（any type）
X	一个配置器类（和数据类型 T 相关）
Y	相应的配置器类（和数据类型 U 相关）
t	常量 T&型的值
a, a1, a2	类型 X& 的值
b	类型 Y 的值
p	类型 X:: pointer 的值，包含在可调用 a1.allocate 中，此处 a1==a
q	X::const_pointer 类型的值，从数据类型 q 转换而来
r	类型 X::reference 从*p 转换而来
s	类型 X::const_reference 的值，从表达式*q 获取或从值 r 转换而来
u	类型值 Y::const_pointer 通过调用 Y::allocate 获得，或为 0
n	类型 X::size_type 的值

表 9-5 描述了内存配置器类型的性能和要求。

表 9-5　内存配置器类型的性能和要求

表　达　式	返　回　类　型	断言/提示预/后-条件处理
X::pointer	Pointer to T	
X::const_pointer	Pointer to const T	
X::reference	T&	
X::const_reference	T const&	
X::value_type	Identical to T	
X::size_type	unsigned integral type	一个类型，在内存配置模型中可以代表最大对象的大小
X::difference_type	signed integral type	在内存配置模型中，该类型可以代表任意两个指针之间的差
typename X::template rebind<U>::other	Y	对于所有的 U（包括 T），Y::template rebind<T>::other 是 X
a.address(r)	X::pointer	
a.address(s)	X::const_pointer	
a.allocate(n)	X::pointer	内存可配置 n 个类型为 T 的对象，但对象不会被构造
a.allocate(n,u)		
a.deallocate(p,n)	not used	调用之前，由 p 确定的区域中，所有的 n 个 T 型对象指出，应该会被破坏。n 应该匹配由 allocate 配置获取的内存大小，并且不抛出异常
a.max_size()	X::size_type	由 X::allocate()可配置的最大内存值
a1= = a2	bool	如果每个对象分配的内存可以使用其他对象被重新分配，返回 true
a1 != a2	bool	等同于 !(a1==a2)
X()		创建默认的实例提示：析构器被假定
X a(b)		post:Y(a) = = b
a.construct(p,t)	not used	效果：new((void*)p) T(t)
a.destroy(p)	not used	效果：((T*)p)->~T()

说明：

（1）表中的成员类模板 rebind 是有效的类定义模板：如果名称 Allocator 和某配置器（SomeAllocator<T>）模板绑定，配置类的成员函数 rebind<U>::other 和 SomeAllocator<U>是同一类型；

（2）在国际标准中描述的容器，被允许假设，除表 9-5 之外，配置器模板参数满足下面的两个额外要求。

- 定配置器类型的所有实例被要求互相变化，并且总是相互相等的；
- 类型定义 members pointer、const_pointer、size_type 和 difference_type 均被要求称为 T* 、 T const*、size_t 和 ptrdiff_t 等。

开发人员（实现者）鼓励提供库，该库可以接受配置器。该配置器可以包括更普通的内存模型，并支持“不等”实例。在这些实现者中，任何要求通过容器强迫内存配置器。在表 9-5 中，配置器实例进行非等于比较时，并且容器的语义学和算法是可以自定义实现的。

9.1.6　运算符

为避免多余的运算符（!=）定义，而不是运算符（==）和运算符“>”、“ <=”和“>=”，而不

是运算符“<”。库提供以下运算符的声明：

```
template <class T> bool operator!=(const T&x , const T& y );
```

要求：类型 T 是可以平等比较的。

返回值：!(x= = y)

```
template <class T> bool operator > (const T&x , const T& y );
```

要求：类型 T 是小于可比较的。

返回值：y < x

```
template <class T> bool operator <=(const T&x , const T& y );
```

要求：类型 T 是小于等于可比较的。

返回值：!(y<x)

```
template <class T> bool operator >=(const T&x , const T& y );
```

要求：类型 T 是小于等于可比较的。

返回值：!（x<y）

在该库中，无论什么时候提供这些运算符“!=”、“>”、“>=”或“<=”的声明，要求和语义学没有清楚的提供，在本节中要求和语义学含义被明确化了。

9.1.7 对组(pairs)

“对组”是一个新概念。在 C++的标准模板库中，很多模板容器的实现都使用了对组。本小节主要讲述对组（pairs）的概念，便捷函数 make_pair()的使用方法。之后给出了一个使用对组的简单例题。

1. 对组的概念

通俗地讲，需要将两个数视为一个单元数值的场合，就需要用到对组 pair。类 pair 可以将两个值视为一个单位，恰恰满足了这个需要。C++标准程序库内多处使用该类。尤其是容器类别 map 和 multimap，均是使用 pairs 实现管理其键值和实值的成对元素。map 和 multimap 在前面的章节已经介绍。任何函数需要返回两个数值时，也可以使用对组 pair。在头文件<utility>中，pair 的声明如下：

```
template<class Type1, class Type2>  struct pair
  {
  typedef Type1 first_type;
  typedef Type2 second_type
  Type1 first;
  Type2 second;
  pair( );
  pair( const Type1& __Val1, const Type2& __Val2 );
  template<class Other1, class Other2> pair( const pair<Other1, Other2>& _Right );
  };
```

为方便使用 pair 类型，可以构造明确的模板类 pair<T, U>对象。可以让默认的构造函数提供默认初始值完成该工作。如果需要获取 pair 对象（例：x）的第一个成员，可以使用 first；如果需要获取第二个成员，需要使用 second。使用成员模板构造函数，可以用另一个模板类 pair<V, W>的对象构造模板类 pair<T, U>的对象。相应成员提供初始值。

对组 pair 被定义成结构体（struct），而非类（class）。所有成员都是 public，可以直接存取 pair 中的个别值。结构体中默认的构造函数会生成一个 pair()，以两个“被该 default 构造函数个别初始化”的值作为初始值。根据语言规则，基本型别的默认构造函数可以适当地初始化。例如：

```
std::pair<int, float> p;  //结构体 pair 的第一个和第二个元素被初始化为 0
```

在结构体 pair 的对象 p 初始化时，会调用类型 int()和 float()来初始化 p。这两个构造函数均返回零值。使用模板类的 copy 构造函数，因为构造过程中可能需要隐式型别转换。如果 pair 对象被复制，调用的是由系统隐式生成的 copy 构造函数。

pair 对象之间还可以实现比较。为比较两个 pair 对象，C++标准程序库提供常用的操作符。若两个 pair 对象内的所有元素均相等，则两个 pair 对象即视为相等。pair 对象“==”运算符的声明如下：

```
name std{
   template <class T1, class T2>
   bool operator ==(const pair<T1, T2>& x , const pair<T1, T2>& y)
   {
    return  x.first== y.first && x.second == y.second;
    }
   }
```

两个 pair 对象互相比较时，第一元素具有较高的优先级。如果两个 pairs 的第一元素不相等，其比较结果就成为比较行为的结果。若第一元素相等，才继续比较第二元素，并将比较结果作为整体比较结果。

```
namespace std{
   template <class T1, class T2>
   bool operator  <  (const pair<T1, T2>& x, const pair<T1,T2>& y)
      {
      return x.first < y.first||(!(y.first<x.first)&&x.second<y.second);
      }
   }
```

其他的比较运算符和运算符“<”的声明方法是相同的。

2. 便捷函数 make_pair()

模板（template）函数 make_pair()使程序员无须写出型别，即可生成一个 pair 对象。函数 make_pair()的声明如下：

```
namespace std{
  template <class T1, class T2>
  pair<T1, T2> make_pair(const T1& x , const T2& y )
    {
     return pair<T1, T2> (x, y);
    }
  }
```

使用函数 make_pair()即可迅速地创建一个 pair 对象，而不一定非要使用 pair 结构创建该结构的对象，下面在例 9-3 中详细说明。

例 9-3

```
std::make_pair(42,'@');
std:: pair<int, char>(42, '@');
```

尤其当必须接受 pair 型对象作为函数参数时，使用 make_pair()尤为重要。按例 9-3 举例如下：

```
//函数声明
void func(std::pair<int, const char*>);
  //函数定义
void main()
{
...
func(make_pair(42,"hello"));
   func(make_pair(41,"bye"));
...
}
```

由以上内容可知，函数 make_pair()使得“将两个值作为一个 pair 参数来传递”，形式上更简单、更容易。即使两个值的型别不能准确地符合要求，也在 template 构造函数提供的支持下顺利工作。尤其使用 map 和 multimap 时，pair 对象更是具有绝对的明确的型别。按例 9-3 举例如下：

```
std::pair<int, float>(42, 2.22);
```

上式产生的效果和

```
std::make_pair(42, 2.22)
```

是不一致的。

后者所产生的 pair，第二个元素的型别是 double。使用重载函数或类模板 template，更确切的型别非常重要。为提高效率，程序员可能会同时提供针对 float 和 double 的函数或类模板 template。

像上述情况，确切的数据类型型别尤显重要。

3. pair 简例

C++标准程序库大量运用 pair 结构对象。像前面章节介绍的 map 容器和 multimap 容器的元素型别更是使用 pair 型别，即键值（key/value）。关于 maps 和 multimaps 的通常描述，在前面章节已经介绍。C++标准程序库中凡是“必须返回两个值”的函数，均可利用 pair 对象。下面给出最简单的例题（例 9-4），说明 pair 类型的使用方法。

注意：如果 map 容器的元素类型是 pair，使用 map 容器排序有多种方法：（1）在 map 声明时，即定义排序规则；（2）将 map 中的元素转换至 vector 容器中，使用算法 sort 排序。第一种方法是自动排序，数据输入之后，不再保留原有的输入顺序。第二种方法在数据输入完毕之后，能够保持原有的输入顺序，在此基础上进行排序，看似更加合理一些。

例 9-4

```
#include <iostream>
#include <map>
#include <utility>
#include <fstream>
#include <cstdio>
#include <iomanip>
#include <algorithm>
#include <functional>
#include <vector>
using namespace std;
typedef pair<int, float> mypair;
void out(mypair& p)
{
cout<<"Index: "<<right<<p.first<<" ,";
cout<<std::fixed<<setprecision(2)<<setw(8)<<right<<p.second<<" ;"<<endl;
```

```
}
void print(map<int,float>& m)
{
map<int,float>::iterator it;
for(it=m.begin();it!=m.end();it++)
{
mypair p=(mypair)(*it);
out(p);
}
}
void printV(vector<mypair>& v)
{
vector<mypair>::iterator it;
for(it=v.begin();it!=v.end();it++)
{
mypair p=(mypair)(*it);
out(p);
}
}
void In_Original(string filename,map<int,float>& m)
{
if(filename.empty())
{
cout<<"filename is error, please input again."<<endl;
exit(0);
}
m.clear();
ifstream ifn;
ifn.open(filename.c_str(),ios_base::in);
if(ifn.fail())
{
cout<<"file can't be opened."<<endl;
exit(0);
}
else
{
while(!ifn.eof())
{
//mypair temp_pair;
char temp[128];
ifn.getline(temp,127);
int id;float  val;
sscanf(temp,"%d,%f",&id,&val);
//temp_pair.first=id;
//temp_pair.second=val;
//m.insert(temp_pair);
m.insert(make_pair(id,val));
}
}
}
bool mysort(mypair &m, mypair &n)
{
    return m.second < n.second;
}
bool mysortGreater(mypair &m, mypair &n)
{
return m.second > n.second;
```

```
}
void main()
{
map<int, float> m,m1;
m.clear();
m1.clear();
map<int,float>::iterator it;
vector<mypair> v;
vector<mypair>::iterator itV;
In_Original("data.txt",m);
print(m);
for(it=m.begin();it!=m.end();it++)
{
mypair temp=(mypair)(*it);
v.push_back(make_pair(temp.first,(float)temp.second));
}
cout<<"Output vector: "<<endl;
printV(v);
sort(v.begin(),v.end(),mysort);  // mysort);
cout<<"Output vector sorted by less."<<endl;
printV(v);
//暂存在 vector 中
sort(v.begin(),v.end(),mysortGreater);
cout<<"Output vector sorted by greater."<<endl;
printV(v);
//暂存在 vector 中
}
```

程序的执行结果如下：

```
Index: 1 ,   98.50 ;
Index: 2 ,   99.00 ;
Index: 3 ,  100.00 ;
Index: 4 ,   87.00 ;
Index: 5 ,   95.00 ;
Index: 6 ,   94.00 ;
Index: 7 ,   96.00 ;
Index: 8 ,   84.50 ;
Output vector:
Index: 1 ,   98.50 ;
Index: 2 ,   99.00 ;
Index: 3 ,  100.00 ;
Index: 4 ,   87.00 ;
Index: 5 ,   95.00 ;
Index: 6 ,   94.00 ;
Index: 7 ,   96.00 ;
Index: 8 ,   84.50 ;
Output vector sorted by less.
Index: 8 ,   84.50 ;
Index: 4 ,   87.00 ;
Index: 6 ,   94.00 ;
Index: 5 ,   95.00 ;
Index: 7 ,   96.00 ;
Index: 1 ,   98.50 ;
Index: 2 ,   99.00 ;
Index: 3 ,  100.00 ;
Output vector sorted by greater.
Index: 3 ,  100.00 ;
```

```
Index: 2 ,   99.00 ;
Index: 1 ,   98.50 ;
Index: 7 ,   96.00 ;
Index: 5 ,   95.00 ;
Index: 6 ,   94.00 ;
Index: 4 ,   87.00 ;
Index: 8 ,   84.50 ;
Press any key to continue
```

提示：本节讲述了 7 部分内容：相等比较，小于比较，复制构造，默认构造，配置器要求，运算符和对组。在前面的章节已经涉及了对组（pair）的内容，本章的第 9.1.7 节又详细讲解了对组的概念以及常用的对组使用方法。希望读者能够认真阅读本节，并能有所收获。

9.2 动态内存管理

对于广大程序员而言，内存处理是最麻烦的，也是最头疼的。在大学计算机语言教学课程中大多不会详细阐述。在 STL 中，头文件<memory>以非常杰出的方式为容器中的元素分配存储空间，同时也为某些算法执行期间产生的临时对象提供机制。在 C++标准中，提供了内存分配器(allocator)，从此以前常用的指针被分配器取代。头文件<memory>的主要部分是模板类，由它产生容器中所有的默认分配器。该模板类的多数版本由更新的版本替代。最近几年，为支持分配器而需要的语言特性开始广泛应用，实际需要使用分配器的经验非常有限。

9.2.1 默认配置器

在名称空间 std 中，声明了一系列和内存管理相关的模板类、模板函数、缓冲区模板类、迭代器和算法，代码如下：

```
namespace std{
template <class T> class allocator;                  //分配器类
template <>class allocator<void>;                    //分配器类
template<class T, class U> bool operator==(const allocator<T>& , const allocator<U>&)
throw();
//运算符==
template<class T, class U>bool operator!=(const allocator<T>&, const allocator<U>&)
throw();
//运算符!=
template<class OutputIterator, class T> class raw_storage_iterator;
//类 raw_storage_iterator
template<class T> pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);//模板函数
template<class T> void return_temporary_buffer(T* p);                   //模板函数
template<class InputIterator, class ForwardIterator> ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last, ForawrdIterator
result);
//模板函数 uninitialized_copy
template<class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
// 模板函数 uninitialized_fill
template<class ForwardIterator, class Size, class T>
void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
//模板函数 uninitialized_fill_n
```

```
    template<class X> class auto_ptr;
}
```

上述源代码中，包含了和内存管理相关的各种模板类和模板函数。

其中 allocator 是模板类，operator= =()和运算符 operator!=()属于模板函数。类 raw_storage_iterator 是内存管理的迭代器，模板函数 get_temporary_buffer()用于返回临时内存的指针。模板函数 return_temporary_buffer()也用于返回临时缓冲区指针。模板函数 uninitialized_copy()用于从指定的源区域复制对象至未初始化的目标范围内；uninitialized_fill()是模板函数，用于根据数据类型对数据进行数值填充；uninitialized_fill_n()是模板函数，和模板函数 uninitialized_fill()相比较，差别仅仅在于填充数据的个数。

头文件<memory>中，模板类 allocator 的声明如下：

```
template <class T> class allocator
{
public:
  typedef  size_t  size_type;
  typedef  ptrdiff_t  difference_type;
  typedef  T*     pointer;
      typedef  const  T*  const_pointer;
  typedef  T&    reference;
  typedef  const  T& const_reference;
}
```

模板类 allocator 的成员函数主要包括：

（1）pointer address(reference x) const，函数返回值为 x 的引用。

（2）const_pointer address(const_reference x) const，函数返回值为 x 的引用。

（3）pointer allocator (size_type n , allocator<void>::const_pointer hint=0)，此函数使用运算符 new(size_t)。hint 或者等于 0，或者从之前成员函数 allocate 获取并没有传递给 deallocate。值 hint 可能被用于改进实现过程的性能。函数返回值是指向数组初始元素的指针，数组大小为 size n* sizeof(T)，T 为对象的类型。并且存储空间可以通过调用函数 new()获得，但函数调用时是不确定的。使用 hint 是不确定的。如果存储空间不能够获取，函数会抛出异常 bad_alloc。

（4）void deallocate(pointer p, size_type n)，指针 p 应该是一个指针，从函数 allocate()获得的。n 应该等于第一个参数指针 p 之后的元素的截取个数。函数的功能是释放 n 个对象的内存，释放内存的起始位置由指针 p 决定。函数 delete()也可以释放指针指向的内存块，但在调用此函数时，是不确定的。

（5）size_type max_size() const throw()，返回值是函数 allocate()开辟内存时指定的内存大小（即函数 allocate()的第一个参数）。

（6）void construct（pointer p, const_reference val），功能：在特定地址构造特定类型的对象，使用特定的数值进行初始化。

（7）void destroy(pointer p)，返回值是((T*)p)->~T()。

下面介绍两个全局的运算符“==”和“！=”。运算符“==”的声明如下：

```
    template  <class  T1,  class  T2>  bool  operator  ==(const  allocator<T1>&  ,  const
allocator<T2>&)throw();
```

运算符“！=”的声明如下：

```
    template<class T1, class T2> bool operator!=(const allocator <T1>&, const
allocator<T2>&)throw();
```

9.2.2 raw storage iterator

raw_storage_iterator 用于使算法可以存储其结果至非初始化的内存。标准的模板参数 OutputIterator 被要求，其指针 operator*必须返回一个对象，目的是运算符 operator& 被定义并返回一个 T 型指针，并且也被要求满足一个输出型迭代器的性能。raw_storage_iterator 型的类声明如下：

```
    namespace std{
    template <class OutputIterator, class T>
        class raw_storage_iterator:public iterator<output_iterator_tag, void, void, void,
void>{
        public:
    explicit raw_storage_iterator(OutputIterator  x);
    raw_storage_iterator<OutputIterator, T>& operator*();
    raw_ storage_iterator<OutputIterator, T>& operator=(const T& element);
    raw_ storage_iterator<OutputIterator, T>& operator++(const T& element);
        raw_ storage_iterator<OutputIterator, T>& operator++(int);
        };
    }
    raw_storage_iterator(OutputIterator x);
```

使用参数 x 初始化迭代器的值。

```
raw_storage_iterator<OutputIterator, T>& operator*();
```

返回：*this 指针。

```
raw_storage_iterator<OutputIterator, T>& operator=(const T& element);
```

运算符赋值符号“=”使用参数 element 构造一个值，并返回迭代器的引用。

```
raw_storage_iterator<OutputIterator, T>& operator++();
```

功能：使迭代器前进，并返回迭代器新值的引用。

```
raw_storage_iterator<OutputIterator, T>& operator++(int);
```

功能：和上一个运算符函数不同，函数的功能使迭代器前进并返回迭代器原来的值。

9.2.3 temporary buffers（临时缓冲区）

```
template <class T> pair<T* , ptrdiff_t> get_temporary_buffer(ptrdiff_t  n);
```

功能描述：获取存储器指针，该内存必须足以存储 n 个相邻的 T 型对象。

返回值：包含缓冲区地址的“对”和“容量”，如果存储器不能获取，函数会返回 1 对零值。

```
template<class T> void return_temporary_buffer(T* p);
```

功能描述：释放指针 p 指向的内存。

要求：缓冲区应该是最初由函数 get_temporary_buffer()分配的内存。

9.2.4 特定算法

所有的迭代器要求使它们的运算符 operator*返回一个对象，运算符 operator&被定义，并返回一个 T 型指针。在下列算法中，迭代器可用于作为标准的模板参数。在算法 uninitialized_copy()中，标

准的模板参数 InputIterator 要求满足输入迭代器的性能。在所有的下述算法中，标准的模板参数 ForwardIterator 用于满足前向迭代器的性能，也满足可变迭代器的要求，并且要求具有性质为：在增加、指定、比较、废除有效的迭代器时，不抛出异常。在下述算法中，如果抛出异常，也没有作用。

未初始化复制（uninitialized_copy()），函数的声明形式如下：

```
template <class InputIterator, class ForwardIterator>
ForwardIterator
        Uninitialized_copy(InputIterator  first, InputIterator  last,  ForwardIterator
result );
```

功能描述：函数 uninitialized_copy()相当于以下语句：

```
for(; first!=last; ++result, ++first)
new(static_cast<void*>(&*result))
typename iterator_traits<ForwardIterator>::value_type(*first);
```

函数返回值：result。

未初始化填充（uninitialized_fill()），函数的声明形式如下：

```
template <class ForwardIterator , class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

功能描述：函数 uninitialized_fill()的功能相当于以下代码：

```
for(; first!=last; ++first)
new(static_cast<void*>(&*first))
typename iterator_traits<ForwardIterator>::value_type(x);
```

未初始化填充 n 个对象（uninitialized_fill_n()），函数的声明形式如下：

```
template <class ForwardIterator, class Size, class T>
void uninitialized_fill_n(Forwarditerator first, Size n, const T& x);
```

功能描述：函数的功能相当于以下代码：

```
for(;n--;++first)
new(static_cast<void*>(&*first))
typename iterator_traits<ForwardIterator>::value_type(x);
```

9.2.5 C 函数库中的内存管理函数

前面主要讲述了 C++ STL 中的内存管理。本小节主要讲述 C 语言中原有的内存管理函数。C 语言原有的内存函数在 C++中同样适用，这些内存函数的声明在头文件<cstdlib>中。这些函数包括：calloc()，malloc()，free()和 realloc()。头文件<cstdlib>和 C 语言中的头文件 stdlib.h 基本是一致的，不同之处在于：函数 calloc()malloc()和 realloc()不是通过调用函数 new()尝试分配内存空间的。函数 free()不是调用 delete()释放内存空间的。

头文件<cstring>中包含了几个内存管理有关的函数：memchr()、memcmp()、memcpy()、memmove()、memset()。该头文件中还声明了数据类型 size_t 和宏 NULL。除了函数 memchr()之外，其余函数的功能没有发生变化。函数 memchr()的声明如下：

```
const void* memchr(const void* s, int c, size_t n )
 void* memchr(  void* s, int c , size_t n);
```

如果函数调用成功，函数返回字符串 s 中第一个字符 c 的位置指针；如果函数调用失败，函数返回 NULL。

9.3 堆的内存分配

C++程序从存储器的全局存储区分配和释放动态存储块；全局存储区有时称为自由存储区（free store），更通用的称呼是堆（heap）。运用运算符 new 从堆中分配存储器，用运算符 delete 把存储器返回给堆。

9.3.1 new 和 delete 运算符

new 运算符和数据类型、类、结构或数组的名字一起使用，会为新建项目分配存储器并返回存储器位置，程序可以把返回的地址赋予指针。delete 把之前分配的存储器返回给堆，操作数必须是以前分配的存储器地址。存储器返回之后可以被 new 运算符重新分配。下面举例（例 9-5）说明：

例 9-5

```
#include <memory>
#include <iostream>
using namespace std;
struct Date{
int month;
int day;
int year;
};
void main()
{
Date* birthday=new Date;
birthday->month=6;
birthday->day=24;
birthday->year=2012;
cout<<"This day is : "<<birthday->month<<"/"<<birthday->day<<"/"<<birthday->year;
cout<<endl;
delete birthday;
}
```

程序的执行结果如下：

```
This day is : 6/24/2012
Press any key to continue
```

上述程序使用 new 分配内存空间，之后进行初始化，在显示相关内容之后，使用 delete 释放该内存空间。

9.3.2 分配固定维数的数组

使用 new 和 delete 为数组获取和释放内存空间，见例 9-6。

例 9-6

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

```
void main()
{
int* birthday=new int[3];
birthday[0]=8;
birthday[1]=19;
birthday[2]=2012;
cout<<"Today is : "<<birthday[0]<<"/"<<birthday[1]<<"/"<<birthday[2]<<endl;
delete[] birthday;
return;
}
```

程序的执行结果如下：

```
Today is : 8/19/2012
Press any key to continue
```

在例 9-6 中，delete 之后还添加了括号[]。这表示将要删除的存储空间是数组。程序员在删除所有动态分配的数组时，都必须使用此种表示方法。尤其当数组中的元素属于自定义类型对象时，尤为重要。

9.3.3 分配动态内存数组

在例 9-6 中，new 运算符接受数据类型和数组维数。在程序中，还可以提供一个变化的维数，new 运算符将根据变量值精确分配存储器。例如：数组的大小可以从屏幕上输入，由用户自定义，见例 9-7。

例 9-7

```
#include <iostream>
#include <cstdlib>
void main()
{
std::cout<<"Enter the array size: ";
int size;
std::cin>>size;
int* array=new int[size];
for(int i=0;i<size;i++)
{
array[i]=rand();
}
for(i=0;i<size;i++)
{
std::cout<<"\n"<<array[i];
}
std::cout<<std::endl;
delete[] array;
return;
}
```

程序的执行结果如下：

```
Enter the array size: 5

41
```

```
18467
6334
26500
19169
Press any key to continue
```

在上述例题中，运行程序时首先需要键入数组的大小。new 运算符使用键入值计算要分配的存储缓冲区大小。new 运算符将 size 值与数组类型的大小相乘，确定从堆中分配多少存储器。程序使用 new 运算符创建数组，在数组中填入随机数，显示数组的每个元素，最后使用 delete 运算符删除数组。

9.3.4　处理堆耗尽

使用 new 时，要充分考虑对存储器的耗尽问题。程序开发时，多数假设堆不会耗尽。如果程序对堆的要求非常大，程序将不可能将堆耗尽。系统提供的存储器不能满足需要时，C++的 new 运算符会抛出运行异常。一旦程序未能捕获异常，系统便会异常中止程序。在大多数情况下，用户并不知道中止的原因。另外，如果堆耗尽是由程序的错误导致的，而程序员又没有事先采取措施来阻止程序异常中止，则无法知道程序中止的详细情况。大多数程序都未就此问题采取措施。

通过将 try 块和 catch 块放在 main()函数中，至少可以将异常截获。

9.4　辅助功能

通用工具库还包含了大量的辅助功能：数值极限、最大最小值、两值交换、辅助性比较等。这些功能看似不复杂，其实用处非常大。例如：数值极限、两值交换均是使用非常广泛的功能。读者应该深入理解这些功能背后的含义。

9.4.1　数值极限

通常，数值型别的极值是与平台相关的特性。C++标准程序库通过 template numeric_limits 提供极值，取代传统的 C 语言所采用的预处理器常数。整数常数定义于<climits>和<limit.h>，浮点常数定义于<cfloat>和<float.h>。极值的概念有两个优点：首先提供了较好的型别安全性；其次程序员可以借此自定义模板用于核定极值。

C++标准中规定了各种型别必须保证的最小精度。如果能够注意并运用该极值，则较容易写出与平台无关的程序。各种数据类型的最小字节长度见表 9-6。

表 9-6　各种数据类型的最小字节长度

型　　别	最小字节长度
char	1 byte
short int	2 bytes
int	2 bytes
long int	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

STL 提供了类模板 class numeric_limits< >。使用模板，通常是为了对所有型别写出通用的解决方案。此外，在必要时刻能以模板 template 为每个型别提供共同的接口。不仅提供通用的模板 template，还提供其特化（specialization）版本。类模板 numeric_limits 即是该技术的典型例子。

（1）通用性的模板（template），为型别提供默认极值。

```
namespace std{
template <class T> class numeric_limits{
public:
static const bool is_specialized=false;
......
....};
}
```

通用性 template 将成员 is_specialized 设置为 false。对型别 T 而言，无所谓极值的存在。

（2）各具体型别的极值，由特化版本（specialization）提供：

```
namespace std{
template <> class numeric_limits<int>{
   public:
  static const bool is_specialized = true;
  static T min() throw(){
     return -2147483648;
          }
    static T max() throw(){
  return 2147483647;
    }
    static const int digits=31;
...
    }
}
```

通常 is_specialized 设置为 true，所有其他成员都根据特定型别的具体极值加以设定。通用性的类模板 numeric_limits 及其特化版本均放在头文件<limits>中。C++ STL 的特化版本中，涵盖了所有数值基本型别，包括：

```
bool
char
signed char
unsigned char
wchar_t
short
unsigned short
int
unsigned int
long
unsigned long
float
double
long double
```

模板类 numeric_limits 的所有成员及其意义列于表 9-7 中。表中的第三列对应相应的 C 常数，这些常数包含在头文件<climits>，<limits.h>，<cfloat>，<float.h>中。

表 9-7　模板类 numeric_limits 的成员说明

成　员	意　义	对应的 C 常数
is_specialized	用于判断类型型别在类模板 numeric_limits 中是否被明确定义	
is_signed	用于判断该类型型别是否有符号表达式	
is_integer	判断该类型型别是否为整型	
is_exact	判断该计算过程中该型别是否会产生舍入误差	
is_bounded	用于判断集合中元素个数是否是有界的	
is_modulo	测试该类型是否具有模表述	
is_iec559	测试数据类型是否遵从 IEC559	
min()	最小值。对于浮点数而言，是标准化的值，只有当（is_bounded||!is_signed）为真时才有意义	INT_MIN ， FLT_MIN ，CHAR_MIN，....
max()	最大值。当 is_bounded 成立时才有意义	INT_MAX，FLT_MAX，...
digits	字符和整数：不带正负号之位个数； 浮点数：尾数中之 radix 位个数	CHAR_BIT, FLT_MANT_DIG,...
digits10	十进制数个数（只有当 is_bound 成立时才有意义）	FLT_DIG
radix	获取数据类型的底数。（整数：表示式的底数（base），几乎总是 2；浮点数：指数表示式的底数（base））	
min_exponent	底数 radix 的最小负整数指数	FLT_MIN_EXP，...
max_exponent	底数 radix 的最大正整数指数	FLT_MAX_EXP，...
min_exponent10	底数 10 的最小负整数指数	FLT_MIN_10_EXP，...
max_exponent10	底数 10 的最大正整数指数	FLT_MAX_10_EXP，...
epsilon()	1 与最接近 1（大于 1 的）数值之间的差距，即精确度	FLT_EPSILON
round_style	浮点数舍入成整数的风格	
round_error	该类型的最大舍入误差（根据 ISO/IEC 10967-1 标准）	
has_infinity	测试有否具有“正无穷大”表述	
infinity()	“正无穷大”表述	
has_quiet_NaN	测试某类型是否具有非数值，非信号表述。	
quiet_NaN()	返回某类型的非数值表述。	
has_signalling_NaN	测试该型别具有发出信号的“非数值”表述式	
signaling_NaN	如果可以，返回某信号的表述，而不是“数值”	
has _denorm()	测试某型别是否允许非标准化数值	
has_denorm_loss	测试是否能检测出精度的缺失，作为一个非标准化缺失而不是一个不精确的结果	
denorm_min()	返回最小的非零非标准化正值	
traps	用于测试是否捕获算法异常的报告	
tinyness_before	在舍入处理之前，作为标准化的数值，测试某类型是否可以决断：数值由于太小而不能表述	

注 1：表格中“意义”一栏中的诸多解释，可能会有不合理之处。著者能力有限，请读者谅解。

下面以一个简单的实例（例 9-8）（摘自《C++标准程序库》（侯捷译））来说明。读者可以复制在自己的计算机上，看看你的计算机所支持的各种数据类型的最大数值是多少。

例 9-8

```
#include <iostream>
```

```
#include <limits>
#include <string>
using namespace std;
void main()
        {
        cout<<boolalpha;
        cout<<"max(short): "<<numeric_limits<short>::max()<<endl;
        cout<<"max(int): "<<numeric_limits<int>::max()<<endl;
        cout<<"max(long): "<<numeric_limits<long>::max()<<endl;
        cout<<endl;
        cout<<"max(float): "<<numeric_limits<float>::max()<<endl;
        cout<<"max(double): "<<numeric_limits<double>::max()<<endl;
        cout<<"max(long double): "<<numeric_limits<long double>::max()<<endl;
        cout<<endl;

        cout<<"is_signed(char): "<<numeric_limits<char>::is_signed<<endl;
        cout<<"is_specialized(string): "<<numeric_limits<string>::is_specialized<<endl;
        cout<<endl;
        }
```

程序的执行结果如下：

```
max(short): 32767
max(int): 2147483647
max(long): 2147483647

max(float): 3.40282e+038
max(double): 1.79769e+308
max(long double): 1.79769e+308

is_signed(char): 1
is_specialized(string): 0
```

程序的执行结果如下的最后一行表示型别 string 没有定义数值极限，因为 string 并非数值型别。对任何型别可以进行询问：是否定义了极值。

9.4.2 最大最小值（较大较小值）

算法库中包含了 3 个辅助函数。一个用于挑选两者中的较大者，另一个用于挑选两者中的较小者，第 3 个函数用于交换两个数值。本小节讲述最大值和最小值算法，随后讲述两值交换算法。函数（或算法）min()和 max()声明于头文件<algorithm>中。两个算法的声明如下：

```
template<class Type>
   const Type& min( const Type& _Left,  const Type& _Right  );
template<class Type, class Pr>
   const Type& min( const Type& _Left,  const Type& _Right,  BinaryPredicate _Comp );
template<class Type>
   const Type& max( const Type& _Left,  const Type& _Right );
template<class Type, class Pr>
   const Type& max( const Type& _Left,  const Type& _Right,  BinaryPredicate _Comp );
```

当两值相等时，通常会返回第一个值。上述函数的声明中，每个函数均有两个版本，即包含“比较准则”。作为“比较准则”的那个参数应该是个函数或仿函数（functor），接受两个参数并进行比

较。在某个指定规则下，判断第一参数是否小于第二次参数，并返回判断结果。

9.4.3　两值交换

函数 swap()用于交换两对象的值。其泛型化版本定义于<algorithm>。

```
namespace std{
   template<class T> inline void swap(T& a , T& b)
      {
         T tmp(a);
         a=b;
         b=tmp;
      }
   }
```

运用该函数，可以交换任意两个变量。

```
std::swap(x, y);
```

只有当 swap()所依赖的 copy()构造操作和 assignment 操作行为存在时，算法（函数）的调用才有效。swap()的最大优势在于，通过模板特化或函数重载，为更复杂的型别提供特殊的实作版本。即可以交换对象内部成员，不必反复赋值，大大节省时间。标准程序库中所有容器及 strings 均使用了这项技术。

前面讲述算法的章节已经讲述了 swap()的使用，调用算法 swap()不通过反复赋值操作来交换两容器的值，提高了效率。对于自定义型别，确实能够改善算法效率。此时需要义不容辞地提供 swap 的特化版本，即自定义的交换算法，见例 9-9。

例 9-9

```
#include <iostream>
#include <algorithm>
#include <memory>
#include <exception>
//using namespace std;
class MyContrainer{                  //自定义容器类
public:
int* elem;                           //容器的存储空间(数组)
int num_elem;                        //容器中存储元素的个数
public:
MyContrainer()                       //构造函数
{
try{
elem=new int[128];                   //开辟部分空间
}
catch(exception& e)                  //捕获异常
{
std::cout<<"exception."<<e.what()<<std::endl;
exit(0);
}
}
void myswap(MyContrainer& mc)        //自定义交换算法
{
std::swap(elem,mc.elem);
```

```
std::swap(num_elem,mc.num_elem);
}
void myassign(int* mc_elem,int num)                           //自定义 assign 算法
{
memcpy(elem,mc_elem,num*sizeof(int));
num_elem=num;
}
~MyContrainer()                                               //析构函数
{
delete[] elem;
}
};
inline void myswap(MyContrainer& c1, MyContrainer& c2)        //全局函数声明
{
c1.myswap(c2);
}
void print (MyContrainer& c1)                                 //输出容器对象
{
int num=c1.num_elem;
for(int i=0;i<num;i++)
{
std::cout<<" "<<*(c1.elem+i);
}
std::cout<<std::endl;
}
void main()
{
int array_one[10]={1,2,3,4,5,6,7,8,9,0};                      //数组 1
int array_two[10]={11,12,13,14,15,16,17,18,19,20};            //数组 2
MyContrainer mc1,mc2;
mc1.myassign(array_one,10);                                   //初始化容器对象 mc1
std::cout<<"the original mc1: "<<std::endl;
print(mc1);                                                   //输出容器的对象
mc2.myassign(array_two,10);                                   //初始化容器对象 mc2
std::cout<<"the original mc2: "<<std::endl;
print(mc2);                                                   //输出容器的对象
mc1.myswap(mc2);                                              //交换算法
std::cout<<"the original mc1(swaped): "<<std::endl;
print(mc1);                                                   //输出容器的对象
std::cout<<"the original mc2(swaped): "<<std::endl;
print(mc2);                                                   //输出容器的对象
myswap(mc1,mc2);                                              //交换算法
std::cout<<"the original mc1(swaped twice.): "<<std::endl;
print(mc1);                                                   //输出容器的对象
std::cout<<"the original mc2(swaped twice.): "<<std::endl;
print(mc2);                                                   //输出容器的对象
}
```

程序的执行结果如下：

```
the original mc1:
 1 2 3 4 5 6 7 8 9 0
the original mc2:
```

```
 11 12 13 14 15 16 17 18 19 20
the original mc1(swaped):
 11 12 13 14 15 16 17 18 19 20
the original mc2(swaped):
 1 2 3 4 5 6 7 8 9 0
the original mc1(swaped twice.):
 1 2 3 4 5 6 7 8 9 0
the original mc2(swaped twice.):
 11 12 13 14 15 16 17 18 19 20
```

提示：在阅读本例题时，注意指针的用法和自定义函数 myassign()。本例题主要是通过自定义容器和自定义交换算法，实现两个对象的交换。

9.4.4　辅助性比较

前面讲述了"小于比较"运算符。本小节重新讲述!=，>，<=，>= 4 个操作符。这 4 个操作符均是利用操作符==和<完成的。这 4 个操作符声明于头文件<unitility>中。

```
namespace std{
    namespace rel_ops{
    template <class T> inline bool operator != (const T& x, const T& y)
        {
            return  !(x==y);
        }
    template <class T> inline bool operator> (const T& x, const T& y)
        {
            return  y<x ;
        }
    template <class T> inline bool operator<= (const T& x, const T& y)
        {
            return  !(y<x) ;
        }
    template <class T> inline bool operator>= (const T& x, const T& y)
        {
            return  !(x<y) ;
        }
    }
}
```

其实，仅需定义<和==操作符，皆可使用上述操作符。使用时需要添加语句：using namespace std::rel_ops，上述 4 个比较操作符即自动获得了定义。命名空间 rel_ops 属于 std 的次命名空间。为防止和用户定义的全局命名空间中的同类型操作符发生冲突，使用名称空间 rel_ops 时，一定要记住使用如下语句：using namespace std::rel_ops。某些实例化版本采用两个不同的参数型别来定义上述模板 template。即：

```
namespace std{
    template <class T1, class T2> inline bool operator!=(const T1& x, const T2&y)
        {
            return !(x==y);
        }
            ......
    }
```

9.4.5 头文件 cstdlib 和 cstddef 简介

头文件<cstddef>和<cstdlib>与其原有的 C 版本对应，在 C++程序中经常用到。其在 C 语言中的对应头文件分别为<stddef.h>和<stdlib.h>。

1. <stddef>中的定义

头文件<stddef>的定义项主要包括：NULL，size_t，ptrdiff_t 和 offsetof。NULL 通常用于表明一个不指向任何对象的指针，其实就是 0。C 语言中的 NULL 通常定义为(void*)0。在 C++中其实并不正确，NULL 的型别必须是整数型别，否则无法将 NULL 赋值给指针。在 C++中没有定义从 void*到任何其他型别的自动转换。NULL 同时定义于头文件<cstdio>，<cstdlib>，<cstring>，<ctime>，<cwchar>，<clocale>内。

- size_t 是一种无正负号的型别，用于表示大小（例如：元素个数）。
- ptrdiff_t 是一种带有正负号的型别，用于表示指针之间的距离。
- offsetof 表示一个成员在结构体（struct）或联合体（union）中的偏移量。

2. <cstdlib>中的定义

头文件<cstdlib>中包含了最重要的一些定义，主要包括：exit()，EXIT_SUCCESS，EXIT_FAILURE，abort()，atexit(void (*function)())。常数 EXIT_SUCCESS 和 EXIT_FAILURE 均用于函数 exit()的参数，也可以用于 main()的返回值。

经由 atexit()设定的函数，在程序正常退出时会依据设定的相反次序被一一调用起来。无论是通过 exit()或者 main()退出，不传递任何参数。

函数 exit()和 abort()可用于在任意点终止程序的运行，无须返回 main()。函数 exit()会销毁所有的 static 对象，并将所有缓冲区清空（flush）。关闭所有 IO 通道，之后终止程序。如果函数 atexit()设定的函数抛出异常，会调用 terminate()。

函数 abort()会立刻终止函数，不做任何清理工作。

这两个函数不会销毁局部对象，因为堆栈 unwinding 不被执行。为确保所有的局部对象的析构函数均获得调用，应该运用异常或正常返回机制，之后再由 main() 离开。

9.5 日期和时间

在学习本节之前，首先提醒读者。时间是很复杂的概念。本节涉及的概念主要包括：GMT（格林尼治标准时间），UTC（世界标准时），日历。要分清楚这些概念，才能学好本节。

STL 库中还包含一些和日期、时间相关联的函数及算法。这些函数包含在头文件<ctime>中。在头文件<ctime>中包含 1 个宏：NULL；3 个类型：size_t、clock_t 和 time_t；1 个结构体 tm；9 个函数：asctime()，clock()，difftime()，localtime()，strftime()，ctime()，gmtime()，mktime()，time()。本小节主要讲述这些内容的使用方法。

9.5.1 3 个类型

3 个类型主要包括：size_t，clock_t，time_t。

size_t 应该是 unsigned int。在前面已经讲过 size_t 是与机器相关的 unsigned 类型，其大小足以保证存储内存中对象的大小。size_t 的引入主要是为了跨平台，增强了程序在平台上的可移植性。

类型 size_t 在头文件 time.h 中的声明如下：

```
#ifndef _SIZE_T_DEFINED
typedef unsigned int size_t;
#define _SIZE_T_DEFINED
#endif
```

clock_t 通常代表长整型。在头文件<time.h>中，其声明如下：

```
#ifndef _CLOCK_T_DEFINED
typedef long clock_t;
#define _CLOCK_T_DEFINED
#endif
```

time_t 通常也是长整型。在头文件<time.h>中，其声明如下：

```
#ifndef _TIME_T_DEFINED
typedef long time_t;        /* time value */
#define _TIME_T_DEFINED     /* avoid multiple def's of time_t */
#endif
```

9.5.2　结构体 tm

结构体 tm 在头文件<time.h>中的声明如下：

```
#ifndef _TM_DEFINED
struct tm {
       int tm_sec;     /* seconds after the minute - [0,59] */
       int tm_min;     /* minutes after the hour - [0,59] */
       int tm_hour;    /* hours since midnight - [0,23] */
       int tm_mday;    /* day of the month - [1,31] */
       int tm_mon;     /* months since January - [0,11] */
       int tm_year;    /* years since 1900 */
       int tm_wday;    /* days since Sunday - [0,6] */
       int tm_yday;    /* days since January 1 - [0,365] */
       int tm_isdst;   /* daylight savings time flag */
       };
#define _TM_DEFINED
#endif
```

结构体 tm 用于存储时刻或者计时器。其中成员 tm_sec 用于存储时刻中的“秒”，成员 tm_min 用于存储时刻中的“分”，成员 tm_hour 用于存储时刻中的“小时”。成员 tm_mday 用于存储时间中的“日期”，成员 tm_mon 用于存储时间中的“月份”，成员 tm_year 用于存储时间中的“年份”。成员 tm_wday 用于存储时间中的“星期几”，成员 tm_yday 用于存储时间中的日期（1～365），成员 tm_isdst 用于存储时间标志（白天或黑夜）。

9.5.3　相关时间函数

本小节详细讲述和时间有关的 9 个函数及其使用方法。

```
char * __cdecl asctime(const struct tm *);
char * __cdecl ctime(const time_t *);
clock_t __cdecl clock(void);
double __cdecl difftime(time_t, time_t);
struct tm * __cdecl gmtime(const time_t *);
struct tm * __cdecl localtime(const time_t *);
```

```
time_t __cdecl mktime(struct tm *);
time_t __cdecl time(time_t * );
size_t __cdecl strftime(char *, size_t, const char *, const struct tm *);
```

（1）asctime()

函数 asctime()用于将指定的时间以字符串（英文简写）形式输出。

（2）ctime()

函数 ctime()用于将指定的时间以字符串形式输出，并遵从本地时区设置。

（3）clock()

函数 clock()的返回值是硬件嘀嗒数，需要换算成秒或者毫秒，通常需要除以 CLK_TCK 或者 CLOCKS_PER_SEC。例如：在 Visual C++6.0 环境下，这两个宏的值均为 1 000，表示硬件嘀嗒 1 000 次所耗时间是 1 秒。因此需要计算进程所耗时间时，可以连续调用两次 clock()，所获得两个时间之差即为所耗时间。如果需要折算成秒，用硬件嘀嗒数除以 1 000 即可，如果不满 1 秒，可以计算出更精确的时间：十分之几秒、百分之几秒、千分之几秒等。

（4）difftime()

函数 difftime()返回两个 time_t 型参数（时刻）之间的时间差。

（5）gmtime()

函数 gmtime(const time_t*)用于将参数传递的时刻转换为格林尼治时间。返回值保存在 tm 型结构体中。tm 结构体中的成员定义在上文中已经阐述。格林尼治标准时间指的是位于英国伦敦郊区的皇家格林尼治天文台的标准时间，因为本初子午线被定义在通过该处的经线。自 1924 年 2 月 5 日开始，格林尼治天文台每隔一小时会向全世界发放调时信息。1999 年 12 月 28 日，一种新的时间系统——格林尼治电子时间正式诞生，为全球电子商务提供一个时间标准。原有的格林尼治时间（GMT）仍保留，作为 21 世纪的世界标准时间。

（6）localtime()

函数 localtime()的功能：把从 1970 年 1 月 1 日零时零分到当前时间系统所偏移的秒数时间转换为日历时间（本地时区时间）。函数返回的是 time_t 结构体，结构体中包含了本地时区的时间。

（7）mktime()

函数 mktime()的功能：将当前的日历时间转换为从 1970 年 1 月 1 日零时零分起至今的 UTC 时间经过的秒数。UTC 时间的中文名称为协调世界时，又称世界统一时间、世界标准时间、国际协调时间，简称 UTC，英文全称为“Universal Time Coordinated”。

（8）time()

函数 time()的功能：获取当前系统（电脑）时间，返回的结果保存在 time_t 型结构体中，其实是一个长整数，其数值表示从 1970 年 1 月 1 日零时零分至当前时间的秒数。可使用函数 localtime()将该时间转换为本地（本时区）时间，并转成结构体 tm 类型，该类型的数据成员分别表示年、月、日、时、分、秒。

（9）strftime()

函数 strftime()的功能：根据区域设置格式化本地时间/日期，函数的功能将时间格式化，或者说格式化一个时间字符串。函数 strftime()的操作类似于 sprintf()。识别以%开始的格式命令集合，格式化输出结果保存在一个字符串中。格式化命令说明串 strDest 中各种日期和时间信息的确切表示方法。格式化串中的其他字符原样放进串中。格式化命令包含很多种，并且是区分大小写的，如表 9-8 所示。

表 9-8　格式化命令说明

格式命令	说　明
%a	星期几的简写
%A	星期几的全称
%b	月份的简写
%B	月份的全称
%c	标准的日期的时间串
%C	年份的后两位数字
%d	十进制表示的每月的第几天
%D	月/天/年
%e	在两字符域中，十进制表示的每月的第几天
%F	年-月-日
%g	年份的后两位数字，使用基于周的年
%G	年份，使用基于周的年
%h	简写的月份名
%H	24 小时制的小时
%I	12 小时制的小时
%j	十进制表示的每年的第几天
%m	十进制表示的月份
%M	十时制表示的分钟数
%n	新行符
%p	本地的 AM 或 PM 的等价显示
%r	12 小时的时间
%R	显示小时和分钟：hh:mm
%S	十进制的秒数
%t	水平制表符
%T	显示时分秒：hh:mm:ss
%u	每周的第几天，星期一为第一天 （值从 0～6，星期一为 0）
%U	每年的第几周，把星期日作为第一天（值从 0～53）
%V	每年的第几周，使用基于周的年
%w	十进制表示的星期几（值从 0～6，星期天为 0）
%W	每年的第几周，把星期一作为第一天（值从 0～53）
%x	标准的日期串
%X	标准的时间串
%y	不带世纪的十进制年份（值从 0～99）
%Y	带世纪部分的十制年份
%z,%Z	时区名称，如果不能得到时区名称则返回空字符
%%	百分号

函数参数说明如下：

```
size_t __cdecl strftime(char *, size_t, const char *, const struct tm *);
```

函数的第 1 个参数是目标缓冲区，第 2 个参数是缓冲区大小，第 3 个参数是输出格式，第 4 个参数是时间参数。

9.5.4 时间示例

例 9-10 对上述的函数分别举例，对各个函数的使用方法进行详尽的演示。

例 9-10

```
#include <ctime>
#include <iostream>
using namespace std;
void main()
{
char tempbuf[128];
struct tm* mytime;
time_t myclock;
clock_t start, finish;
time_t  tstart,tfinish,tcurr;
double duration;
long  count = 600000000L;

time(&myclock);                                  //获取时间
mytime=localtime(&myclock);
cout<<"Usage of asctime() : " <<endl;
cout<<"asctime():  Current Date and time : "<<asctime(mytime)<<endl;

cout<<"Usage of ctime() : " <<endl;
cout<<"ctime():    Current Date and time : "<<ctime(&myclock)<<endl;
cout<<endl;

cout<<"Usage of clock() : "<<endl;
start=clock();                                   //获取时钟
while(count--);
finish=clock();
duration=(double)(finish-start)/CLOCKS_PER_SEC;
cout<<"clock():    Consume "<<duration<<" second. "<<endl;
cout<<endl;

cout<<"Usage of difftime() : "<<endl;
count=600000000L;
time(&tstart);
while(count--);
time(&tfinish);
duration=(tfinish-tstart);                       //获取延时
cout<<"time():     Consume "<<duration<<" second. "<<endl;
cout<<endl;

cout<<"Usage of gmtime()  (GMT) : "<<endl;
time(&tcurr);
```

```
    mytime=gmtime(&tcurr);                        //获取时间
    cout<<"The current time : "<<(mytime->tm_year+1900)<<" ,"<<(mytime->tm_mon+1)<<" ,"
<<mytime->tm_mday<<" , ";
    cout<<mytime->tm_hour<<" : "<<mytime->tm_min<<" : "<<mytime->tm_sec<<" , ";
    cout<<(mytime->tm_isdst?"Night.":"Daylight.")<<" , "<<mytime->tm_wday<<" , "
<<mytime-> tm_yday<<" th."<<endl;
    cout<<endl;

    cout<<"Usage of localtime() : "<<endl;
    time(&tcurr);
    mytime=localtime(&tcurr);                                 //获取当地时间
    cout<<"The  current  time  :  "<<(mytime->tm_year+1900)<<"  ,"<<(mytime->tm_mon+1)
<<" ,"<<mytime->tm_mday<<" , ";
    cout<<mytime->tm_hour<<" : "<<mytime->tm_min<<" : "<<mytime->tm_sec<<" , ";
    cout<<(mytime->tm_isdst?"Night.":"Daylight.")<<" , "<<mytime->tm_wday<<" , "
<<mytime-> tm_yday<<" th."<<endl;
    cout<<endl;

    cout<<"Usage of mktime() : "<<endl;
    tcurr=mktime(mytime);                                     //创建时间
    cout<<"The seconds from 1970/1/1/0:0:0.  :  "<<tcurr <<" seconds. "<<"  "<<endl;

    cout<<"Usage of time() : "<<endl;
    time(&tcurr);
    cout<<"time transformed by localtime() : "<<asctime(localtime(&tcurr))<<endl;
    cout<<"time transformed by gmtime() : "<<asctime(gmtime(&tcurr))<<endl;

    cout<<"Usage of strftime() : "<<endl;
    mytime=localtime(&tcurr);                                 //获取本地时间
    strftime(tempbuf,128,"Today is %A, day %d of %B in the year %Y.\n",mytime);
    //字符串和时间转换
    cout<<tempbuf<<endl;
  }
```

程序的执行结果如下：

```
Usage of asctime() :
asctime(): Current Date and time : Tue Aug 21 14:14:05 2012

Usage of ctime() :
ctime():   Current Date and time : Tue Aug 21 14:14:05 2012

Usage of clock() :
clock():   Consume 1.796 second.
Usage of difftime() :
time():    Consume 1 second.
Usage of gmtime()  (GMT) :
The current time : 2012 ,8 ,21 , 6 : 14 : 8 , Daylight. , 2 , 233 th.
Usage of localtime()  :
The current time : 2012 ,8 ,21 , 14 : 14 : 8 , Daylight. , 2 , 233 th.
Usage of mktime() :
The seconds from 1970/1/1/0:0:0.  : 1345529648 seconds.
Usage of time() :
```

```
time transformed by localtime() : Tue Aug 21 14:14:08 2012

time transformed by gmtime() : Tue Aug 21 06:14:08 2012

Usage of strftime() :
Today is Tuesday, day 21 of August in the year 2012.
```

提示：无论哪一种计算机语言，无论什么计算机系统，时间概念基本是相似的。掌握本节的一系列时间函数首先要弄清楚各种时间标准。对各个函数的输入参数和输出参数有清楚的认识，才能熟练掌握上述时间函数，才能准确使用计算机系统提供的时间。

9.6 模板类 auto_ptr

使用模板类 auto_ptr，需要包含头文件<memory>。模板类 auto_ptr 使用函数 new()将指针存储在一个对象中，当需要破坏此指针时，使用 delete()函数。模板 auto_ptr_ref 保持了对类 auto_ptr 的一个引用。该引用用于允许 auto_ptr 类型对象的传递和从函数中传递出来。代码如下：

```
    namespace std{
    template<class Y>struct auto_ptr_ref{};
    template<class X> class auto_ptr{
    public:
typedef  X  element_type;
explicit auto_ptr(X* p=0) throw();
auto_ptr(auto_ptr&) throw();
template<class Y>auto_ptr(auto_ptr<Y>&)throw();
        auto_ptr& operator=(auto_ptr&) throw();
        template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
        auto_ptr& operator=(auto_ptr_ref<X>  r) throw();
        ~auto_ptr()  throw();
        X& operator*() const throw();
        X* operator->() const throw();
        X* get() const throw();
        X* release() throw();
        void reset(X* p=0) throw();

        auto_ptr(auto_ptr_ref<X>) throw();
        template<class Y> operator auto_ptr_ref<Y>() throw();
        template<class Y> operator auto_ptr<Y>() throw();
        };
    }
```

类 auto_ptr 提供了一种语义学上的严格的所有权。类 auto_ptr 拥有的对象仅仅存储了一个指针而已。复制一个类 auto_ptr 的备份时，指针和所有权均被复制到目的地。如果多个 auto_ptr 类在同一时间拥有同一个对象，程序执行时，结果是不确定的。

提示：类 auto_ptr 的使用包括提供临时异常安全，用于动态分配内存，传递动态内存的所有权给函数，并返回之前为函数动态分配的内存。类 auto_ptr 不能满足标准库容器中的元素要求的“构造复制性”和“可设置性”；在不确定的情况下，不能满足使用 auto_ptr 类型结果初始化标准库容器。

9.6.1　auto_ptr 类构造函数

```
explicit auto_ptr(X* p=0) throw()
```

在函数声明中，*this 保留指针 p。

```
auto_ptr(auto_ptr& a) throw();
```

功能描述：会调用 a.release()。*this 代表从 a.release()返回的指针。

```
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

要求：Y*可以被隐式地转换为 X*。作用是调用 a.release()，*this 代表从 a.release()返回的指针。

```
auto_ptr& operator=(auto_ptr& a) throw();
```

要求：表达式 delete get()应该被很好地执行。执行过程中，会同时执行 reset(a.release())。函数返回值是*this。

```
template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
```

函数 Y*可以被隐式转换为 X*。同样表达式 delete get()应该被很好地执行。执行时会隐式执行。

```
reset(a.release())。函数返回值为*this。
~auto_ptr() throw()
```

要求：表达式 delete get()被很好地执行。函数的作用是删除 get()返回的指针，即 delete get()。

9.6.2　类 auto_ptr 的成员及转换

类 auto_ptr 只有 5 个成员。2 个运算符成员：*，->。3 个函数成员：get()，release()和 reset()。类 auto_ptr 的转换有 4 种形式：构造函数形式，quto_ptr_ref()，auto_ptr()和 operator=()。

1．成员

```
X& operator*() const throw();
```

要求 get()!=0，返回值为函数 get()返回的指针。

```
X* operator->() const throw();
```

函数返回值：函数 get()返回的指针。

```
X* get() const throw();
```

函数返回*this 指针。

```
X* release ( ) throw ( );
```

函数返回 get()的返回值。之后*this 指针即变成空指针。

```
void reset(X* p=0) throw()
```

函数的功能是如果 get()的返回值不等于指针 p，那么需要删除 get()的返回值，相应的等效代码为：

```
if (get()!=p )
delete get()
```

函数执行之后，指针*this 保存了指针 p。

2．转换

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

函数的执行效果：调用 p.release()，其中 p 为 r 内保存的指针。

```
template<class Y>operator  quto_ptr_ref<Y>() throw();
```

函数返回值为类型为 auto_ptr_ref 的变量，变量的值为*this。

```
template<class Y> operator auto_ptr<Y>() throw();
```

函数的作用是调用 release()函数。函数返回值是一个 auto_ptr<Y>，其中包含函数 release()返回的指针。

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw();
```

函数的作用是调用 reset(p.release())，p 为 r 中包含的引用。函数返回值为指针 this。

9.6.3 使用 auto_ptr 类

auto_ptr 类是一个模板类，用于管理动态内存分配的方法。下面举例说明：

```
void remodel(string &  str)
{
string* ps=new string(str);
...
str=ps;
return;
}
```

认真阅读会发现其中的缺陷。当调用时，该函数将分配堆中的内存，但不会收回，因此会导致内存泄漏。

解决的办法是在 return 语句之前添加 delete ps。一旦忘记，则不能实现最佳。编程者可能在不经意间删除或注释掉某些代码，导致出现很多问题。下面用例 9-11 来加以说明：

例 9-11

```
void remodel(string &  str)
{
string* ps=new string(str);
...
If(weird_thing())
throw exception();                    //抛出异常
str=*ps;
delete ps;
return;
}
```

在上述代码中，一旦发生异常，函数 delete 将不被执行，同样会发生内存泄漏的问题。异常抛出后，throw()后面的代码不会被执行。

内存泄漏的原因：当函数 remodel()执行中止时，函数中的局部变量会从堆中删除，指针 ps 占用的内存会被释放，但 ps 指向的内存不被释放。尤其对于基本类型，没有提供额外的服务，使程序自动释放 ps 指向的内存。对于类，可以通过析构函数机制来提供。上述代码中 ps 的问题在于：其仅仅是常规指针，不是类对象。如果是类的对象，在对象过期时其析构函数会删除其指向的内存。类 auto_ptr 恰恰提供上述的需求。

模板类 auto_ptr 定义了类似指针的对象，可以将 new 获得（直接或间接）的地址赋给该对象。

当 auto_ptr 对象过期时，其析构函数将使用 delete 来释放内存。将 new 返回的地址赋给 auto_ptr 对象时，不必记住释放的内存。在 auto_ptr 对象过期时，内存将自动释放。需要创建 auto_ptr 对象时，必须包含头文件 memory，该文件包括了 auto_ptr 模板。使用通常的模板句法实例化所需类型的指针。类模板 auto_ptr 的声明如下：

```
template<class Type>  class auto_ptr {
 public:
     typedef Type element_type;
   explicit auto_ptr(Type *_Ptr = 0) throw();          //不抛出异常
   auto_ptr(auto_ptr<Type>& _Right) throw();          //不抛出异常
       template<class Other> operator auto_ptr<Other>() throw();  //不抛出异常
   template<class Other> auto_ptr<Type>& operator=(auto_ptr<Other>& _Right) throw();
//不抛出异常
   template<class Other> auto_ptr(auto_ptr<Other>& _Right);        //
   auto_ptr<Type>& operator=(auto_ptr<Type>& _Right);              //
   ~auto_ptr();                                          //
       Type& operator*() const throw();                  //
   Type *operator->()const throw();                      //
   Type *get() const throw();                            //
   Type *release()throw();                               //
   void reset(Type *_Ptr = 0);                           //
};
```

在上述代码中，throw()意味着构造函数不引发异常。在请求 X 类型的 auto_ptr 会获得指向 X 类型的 auto_ptr。

```
auto_ptr <double > pd (new double);
auto_ptr<string> ps (new string);
```

new double 是 new 返回的指针，指向新分配的内存块。它是 auto_ptr<double>构造函数的参数，即它是原型中形参 p 的实参。同样，new string 也是构造函数的实参。因此，要转换 remodel()函数，需要 3 个步骤：

（1）包含头文件<memory>；

（2）将指向 string 的指针替换为指向 string 的 auto_ptr 对象；

（3）删除 delete 语句。

以上的函数 remodel()现在可以修改为以下代码：

```
#include <memory>
void remodel(string & str)
{
auto_ptr<string> ps(new string (str));
...
if(weird_thing())
    throw exception();
    str=*ps;
    //delete  ps:  No LONGER NEEDED
return;
}
```

注意：auto_ptr 的构造函数是显式的，意味着不存在指针到 auto_ptr 对象的隐式类型转换。

```
auto_ptr<double>  pd;
```

```
double *p_reg= new double;
pd= p_reg;
pd=auto_ptr<double> (p_reg);
auto_ptr<double>  pauto=p_reg;
auto_ptr<double>  pauto(p_reg);
```

模板使程序员能将 auto_ptr 对象初始化为常规指针。

auto_ptr 是智能指针，类似于指针。auto_ptr 作为类，特性比指针更多、更丰富。auto_ptr 类被定义为在多方面与常规指针类似。auto_ptr 类型的指针可以直接赋值给同类型的 auto_ptr，但会引起一个问题。

值得大家记住的是，虽然模板库（STL）中定义了 auto_ptr，但 auto_ptr 不是万能的。

```
auto_ptr<int> pi(new int [200]);
```

对于 new 和 new[]，必须相应地使用 delete 和 delete[]。auto_ptr 类模板使用的是 delete()函数，不是 delete[]。因此，只能和 new 一起配对使用，不能和 new[]配对使用。不适用于动态数组的 auto_ptr 等同物。复制头文件<memory>中的 auto_ptr 模板，将其重命名为 auto_arr_ptr，然后将其修改，使之使用 delete[]，而不是使用 delete()。希望添加对[]操作符的支持。

```
string vacation ("I wandered lonely as a cloud. ");
auto_ptr<string> pvac(&vacation);
```

提示： 只能对 new 分配的内存使用 auto_ptr 对象，不要对由 new[]分配的或通过声明变量分配的内存使用它。

还有一个常见的问题。

```
auto_ptr<string> ps (new string ( "I reigned lonely as a cloud." ));
auto_ptr<string> vocation;
vocation = ps;
```

如果 ps 和 vocation 是常规指针，则两个指针将指向同一个 string 对象。其中一个是另一个的复制。在 ps 和 vocation 均为过期时，程序将试图删除同一个对象两次，这个问题是要避免的。

- 定义赋值操作符，使之执行深复制。两个指针会指向不同的对象，其中的一个对象是另一个对象的复制。
- 建立所有权概念，对于特定的对象，只能有一个智能指针拥有它。智能指针的构造函数只能删除该智能指针拥有的对象，并使赋值操作转让所有权。用于 auto_ptr 的策略。
- 创建智能最高的指针，跟踪引用特定对象的智能指针针数，称为引用计数。例如：赋值时，计数将加 1；指针过期时，计数将减 1。仅当最后一个指针过期时，delete 才被调用。同样的方法也适用于复制构造函数。

详见例 9-12。

例 9-12

```
#include <memory>
#include <iostream>
#include <string>
using namespace std;
void main()
{
auto_ptr<string> str(new string("Goose eggs."));        //使用 auto_ptr 创建对象
```

```
    auto_ptr<string> str2(str);
    auto_ptr<string> str3(new string("Chicken runs."));

    cout<<str.get()<<endl;                                        //调用成员 get()
    cout<<str2.get()<<endl;
    cout<<str3.get()<<endl;

    cout<<"str: "<<*str<<endl;
    cout<<"str2: "<<*str2<<endl;
    cout<<"str3: "<<*str3<<endl;
}
```

程序的执行结果如下：

```
00382D38
00382D38
00382F68
str: Goose eggs.
str2: Goose eggs.
str3: Chicken runs.
Press any key to continue
```

9.7　本章小结

本章主要讲述了 6 部分内容。第 1 节简要介绍了通用工具库的情况，重点介绍了“对组”的概念。第 2 节讲述动态内存管理，重点讲述了特定算法和内存管理函数。第 3 节讲述堆的内存分配，读者要认真理解。第 4 节讲述辅助功能，例如：数值极限、最大最小值、两值交换等。第 5 节讲述日期和时间的相关内容，较详尽地讲述了时间相关的各种知识，对各个时间函数进行举例说明。第 6 节讲述模板类 auto_ptr，充分讲述了模板类 auto_ptr 的优点。

通过对本章的学习，读者可以了解通用工具库中的内容，对 STL 的 unitility 库有所了解，在使用这些功能时，不至于一无所知，尤其是“对组”、时间、模板类 auto_ptr 等内容对广大程序员帮助很大。

第 10 章 语言支持

本章将介绍在 C++程序执行过程中的功能签名 implicit 和隐含产生的各种类型对象。同时也介绍和这些隐含签名和定义相关类型的头文件。随后的各节将介绍通常类型的定义，这些定义贯穿整个 STL，并被广泛使用。还将介绍预定义类型的特性，函数如何支持 C++程序的启动和终止，如何支持动态内存管理，如何支持动态类型识别，如何支持异常处理，如何支持其他的运行库等，详见表 10-1。

表 10-1 语言支持库汇总

名　称	头　文　件
Types	<cstddef>
Implementation properties	<limits>、<climits>、<cfloat>
Start and termination	<cstdlib>
Dynamic memory management	<new>
Exception handling	<exception>
Other runtime support	<cstdarg>,<csetjmp>,<ctime>, <csignal>,<cstdlib>

10.1 类型

在头文件<cstddef>中，类型定义主要包括：NULL，offsetof，ptrdiff_t，size_t。头文件中的内容和标准 C 库头文件<stddef.h>的内容大致相同，差别在于：在国际标准中，宏 NULL 是一个 C++空指针常量。同样在国际标准中，宏 offsetof 接受严格的一系列类型参数。类型 type 应该是一个 POD 结构或一个 POD union。使用宏 offsetof 的后果是会导致静态数据成员或静态函数成员不能准确定义。

10.2 执行属性

在头文件<limits>、<climits>和<cfloat>中，提供了依赖于基本类型的执行特性。数值限制组件提供了一个 C++程序，包含许多基本类型的执行属性。特殊化或专门化应该提供给每一个基本类型，无论是浮点型还是整型（包括布尔类型）。对于所有的数值极限的特化，成员 is_specialized 应该是 true。

10.2.1　类模板 numeric_limits 及其成员

在数值极限模板 numeric_limits 中，对于所有声明的静态常量成员，在特化时应该定义它们的值，在整型常量表达式中，它们是可以使用的。非基本类型，像复数类型 complex<T>，不应该拥有特化。在头文件<limits>中，头文件的声明如下：

```
namespace std{
template <class T> class numeric_limits;
enum float_round style;
enum float_denorm style;
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<>class numeric_limits<signed char>;
template<>class numeric_limits<unsigned char>;
template<>class numeric_limits<wchar_t>;
template<>class numeric_limits<short>;
template<>class numeric_limits<int>;
    template<>class numeric_limits<long>;
    template<>class numeric_limits<unsigned short>;
    template<>class numeric_limits<unsigned int>;
    template<>class numeric_limits<unsigned long>;
    template<>class numeric_limits<float>;
    template<class> numeric_limits<double>;
    template<class>numeric_limits<long double>;
}
```

模板类 numeric_limits 的声明如下：

```
namespace std{
template<class T> class numeric_limits{
public:
static_const bool is_specialized =false;
static T min() throw();
static T max() throw();
static const int digits=0;
static const int digits10=0;
static const bool is_signed=false;
static const bool is_integer=false;
static const bool is_exact=false;
static const int radix=0;
static T epsilon() throw();
static T round_error() throw();
static const int min_exponent=0;
static const int min_exponent10=0;
static const int max_exponent=0;
static const int max_exponent10=0;
static const  bool  has_infinity=false;
static const  bool  has_quiet_NaN =false;
static const  bool  has_signaling_NaN=false;
static const float_denorm_style has_denorm=denorm_absent;
static const bool has_denorm_loss =false;
static T infinity() throw();
static T quiet_NaN() throw();
static T signaling NaN() throw();
static T denorm_min() throw();
static const bool is_iec559=false;
static const bool is_bounded=false;
```

```
    static const bool is_modulo=false;
    static const bool traps=false;
    static const bool tininess_before =false;
    static const float_round_style round_style=round_toward_zero;
    }
}
```

类模板中成员 is_specialized（是特化类型）使区分基本类型（具有特化性质）和非数量（非标量）类型成为可能。除了 0 或 false 值之外，默认的数值极限模板类 numeric_limits<T>应该包含所有的成员。

数值极限类的各成员下面逐一讲解，如表 10-2 所示。

表 10-2 数据极限类及功能说明

数值极限类	功 能 说 明
static T min() throw();	函数的功能是求解最小有限值。对于非标准化的浮点类型，返回规范化的最小正值。使用此函数时，该类型对象的成员 is_bounded!=false，或 is_bounded=false 并且 is_signed=false
static T max() throw();	函数的功能是求解最大有限值。对于所有的特化对象，只有当该对象的成员 is_bounded!=false 时，本函数才有意义
static const int digits;	成员 digits 代表能够表示的基数位数。对于整数类型，代表非负位的个数。对于浮点类型，基数的个数包含在尾数中
static const int digits10;	表示不会丢失精度的 10 进制数的位数。只有当对象的成员 is_bounded!=false 时，才有意义
static const bool is_signed;	如果是有符号数，值为 true；对于所有的特化对象，均是有意义的
static const bool is_integer;	如果类型是整数，值为 true；对于所有特化对象均是有意义的
static const bool is_exact;	如果类型使用一个精确的表示，值为 true。所有的类型均是严格的，但并不是所有的严格类型均为整数类型。例如：合理的定点数表示也是严格准确的，但并不是整数
static cons tint radix;	对于浮点类型，代表指数表达式的基数或底。对于整数类型，代表表达式的底。对于所有的特化对象，均是有意义的
static T epsilon() throw();	函数 epsilon()返回值代表 1 和可表示的大于 1 的最小值之间的误差。对于所有的浮点数类型，均是有意义的
static T round_error() throw();	函数 round_error()可以获取最大的舍入误差
static const int min_exponent;	函数返回一个最小负整数，幂的基数是成员 radix，但该整数小于最小的规范化的浮点数。对于所有的浮点类型均是有意义的
static const int min_exponent10;	代表最小的负整数，10 是幂的底数或基数，幂的值在规范化的浮点数范围内。对于所有的浮点类型均是有效的
static const int max_exponent;	代表最大的正整数指数，该正数的基数或底是 radix
static const int max_exponent10;	代表最大的正指数（幂）整数，该数的底或基数为 10，幂系数是可表达的有限浮点数
static const bool has_infinity;	如果表达式是正的无穷，值为 true。对于所有的浮点数均是有效的
static const bool has_quiet_NaN;	如果类型是一个关于 quiet 的（非信号量，不是成员）表达式，值为 true。对于所有的浮点类型，是有意义的。对于所有的特化对象，如果其成员 is_iec559 不等于 false，has_quiet_NaN 的值应该是 true
static const bool has_signaling_NaN;	如果类型是一个表达式，该表达式可以发出信号“Not a Number.”。成员 has_signaling_NaN 为真（true）。对于所有的浮点类型均是有意义的。对于所有的特化对象，当其成员 is_iec559 不等于 false 时，has_signaling_NaN 的值为 true
static const float_denorm_style has_denorm;	如果类型允许可表示的数值，成员 has_denorm 的值为 denorm_present；如果类型不允许可表示的值，成员 has_denorm 的值是 denorm_absent；并且程序在编译时，如果不能确定该类型是否是可表示的数值，成员 has_denorm 的值为 denorm_indeterminate。对于所有的浮点类型，has_denorm 均是有意义的
static const bool has_denorm_loss;	如果作为一个非规范化的 loss，loss 的准确度可以被检测到，而不是一个非精确的结果，此时 has_denorm_loss 的值为真

续表

数值极限类	功 能 说 明
static T infinity() throw();	如果函数调用成功，返回一个有效的数值时，该数值为正的无穷大。对于所有的特化对象，当其成员 has_infinity!=false 时，均是有意义的；当 is_iec559!=false 时，该特化也是必需的
static T quiet_NaN throw();	如果函数正确调用的话，函数返回值是一个非信号量，即"Not a Numer"。对于所有的特化对象，当其成员 has_quiet_NaN 不等于 false 时，均是有意义的，函数返回值为 true。当其成员 is_iec559!=false 时，特化对象也是必需的
static T signaling_NaN() throw();	如果函数调用成功的话，函数返回值是一个非信号量的表达式。只有当对象的成员 has_signaling_NaN 不等于 false 时，该特化对象才有意义，函数返回值为 true。当对象的成员 is_iec559!=false 时，特化对象才是必需的
static T denorm_min() throw();	函数返回正的最小规格化值。对于所有的浮点类型均是有意义的。当特化对象的成员 has_denorm 等于 false 时，函数返回正的最小规格化值
static const bool is_iec559;	当且仅当数值类型遵从 IEC559 标准时，对象的成员 is_iec559 为 true。对于所有的浮点类型均是有意义的
static const bool is_bounded;	如果数值集合是可以表达的，并且类型是有限的，对象的成员 is_bounded 为 true。所有的嵌入式类型均是有界的，成员是 false 对于那些任意的准确度类型。对于所有的特化对象均是有意义的
static const bool is_modulo;	如果类型是"模"表达式，is_modulo 为 true；否则为 false。通常，对于浮点类型，其值为 false；对于无符号整数，其值为 true；对于大多数机器，有符号整数的值也是 true。对于所有的特化对象均是有意义的
static const bool traps;	对于该类型，如果捕获被执行，traps 的值为 true。对于所有的特化对象均是有意义的
static const bool tinyness_before;	如果在舍入之前，"极小"被检测，tinyness_before 的值为 true。对于所有的浮点类型均是有意义的
static const float_round_style round_style;	成员 round_style 代表舍入类型。对于整数类型的数值，成员 round_style 的值为 round_towars_zero

10.2.2 float_round_style 和 float_denorm_style

float_round_style 类型是一个枚举类型。其声明形式如下：

```
namespace std{
enum float_round_style{
round_indeterminate= -1;
round_toward_zero = 0;
round_to_nearest = 1;
round_toward_infinity =2;
   round_toward_neg_infinity=3
   };
}
```

根据数值的具体情况，浮点数算法的舍入模式归纳情况如下：

```
-- round_indeterminate           如果舍入类型是不确定的
-- round_toward_zero             如果舍入类型为 0
-- round_to_nearest              如果舍入类型是最能够被接近的可描述的数值
-- round_toward_neg_infinity     如果舍入类型是趋向于负无穷的情况
```

类型 float_denorm_style 同样也是一个枚举类型，其声明如下：

```
namespace std{
      enum float_denorm_style{
      denorm_indeterminate= -1;
      denorm_absent = 0;
      denorm_present = 1;
      }
}
```

存在或缺乏非规范化可以归纳为下面几个方面：

- --denorm_indeterminate　不能确定类型是否允许非规范化的取值；
- --denorm_absent　如果类型不允许非规格化的数值；
- --denorm_present　如果类型允许非规格化的取值。

10.2.3　数值极限的特化

当数值极限类被特化时，所有的模板类成员应该被提供。然而，在某种条件下，许多值才有意义。任何没有意义的值被设置为 0 或 false。举例说明如下：

```
namespace std{
   template<> class numeric_limits<float>{
   public:
   static const bool is_specialized=true;
   inline static float min() throw() {return  1.1754}
   inline static float max() throw(){return  3.40282}
   static const int digits =24;
   static const  int  digits10=6;
   static  const  bool  is_signed =true;
   static  const  bool  is_integer=false;
   static  const  bool  is_exact=false;
   static  const  int  radix=2;
   static  const  float  epsilon()  throw(){return  1.192;}
   static  const  float  round_error()  throw(){return  0.5F;}
   static  const  int  min_exponent=  -125;
   static  const  int  min_exponent10= -37;
   static  const  int  max_exponent = 128;
   static  const  int  max_exponent10= 38;
   static  const  bool  has_infinity = true;
   static  const  bool  has_quiet_NaN = true;
   static  const  bool  has_signaling_NaN = true;
   static  const  float_denorm_style  has_denorm = denorm_absent;
   static  const  bool  has_denorm_loss = false;
   inline  static  float  infinity()  thow()  {return  …;}
   inline  static  float  quiet_NaN()  throw(){return …;}
   inline  static  float  signaling_NaN()  throw()  {return  …;}
   inline  static  float  denorm_min()  throw(){return  …;}
   static  const  bool  is_iec559=true;
   static  const  bool  is_bounded =true;
   static  const  bool  is_modulo=false;
   static  const  bool  traps  = tue;
   static  const  bool  tinyness_before  =true;
   static  const  float_round_style  round_style=round_to_nearest;
   }
}
```

10.2.4　库函数

在头文件<climits>中，其内容和标准 C 库头文件<limit.h>是一致的。在头文件<climits>中，内容主要包括以下宏定义：CHAR_BIT，INT_MAX，LONG_MIN，SCHAR_MIN，UCHAR_MAX，USHRT_MAX，CHAR_MAX，INT_MIN，MB_LEN_MAX，SHRT_MAX，UINT_MAX，CHAR_MIN，

LONG_MAX，SCHAR_MAX，SHRT_MIN，ULONG_MAX。

C++语言或 STL 中的头文件<cfloat>中，宏定义主要包括：DLB_DIG，DBL_EPSILON，DBL_MANT_DIG，DBL_MAX，DBL_MAX_10_EXP，DBL_MAX_EXP，DBL_MIN，DBL_MIN_10_EXP，DBL_MIN_EXP，FLT_DIG，FLT_EPSILON，FLT_MANT_DIG，FLT_MAX，FLT_MAX_10_EXP，FLT_MAX_EXP，FLT_MIN，FLT_MIN_10_EXP，FLT_MIN_EXP，FLT_RADIX，FLT_ROUNDS，LDBL_DIG，LDBL_EPSILON，LDBL_MANT_DIG，LDBL_MAX，LDBL_MAX_10_EXP，LDBL_MAX_EXP，LDBL_MIN，LDBL_MIN_10_EXP，LDBL_MIN_EXP。

头文件中的内容和标准 C 库头文件<float.h>中的内容是一致的。

10.2.5　应用举例

例 10-1 演示了各种类型的大小和程序运行所在系统的极限值等，程序虽然有点儿长，但结构非常简单，就是一段代码对应一段数值，读者一一对应即可。当你拿不准某个定义在系统中的大小和极值时，选取其中一段，在系统中运行即可。

例 10-1

```
    #include <iostream>
    #include <limits>
    using namespace std;
    void main()
    {
  //denorm_min  ()
  cout<<"The smallest nonzero denormalized value(float). "<<"object is: "<<numeric_limits
<float>::denorm_min()<<endl;
   cout<<"The  smallest  nonzero  denormalized  value(float).  "<<"object  is:
"<<numeric_ limits<int>::denorm_min()<<endl;
   cout<<"The smallest nonzero denormalized value(float)."<<"object is:<<numeric_
limits<double>::denorm_min()<<endl;
  //舍入为零
   cout<<"The smallest nonzero denormalized value(float). "<<"object is:<<numeric_
limits<float>::denorm_min()/2<<endl;
   cout<<"The smallest nonzero denormalized value(float). "<<"object is: "<<numeric_
limits<int>::denorm_min()/2<<endl;
   cout<<"The smallest nonzero denormalized value(float). "<<"object is:<<numeric_
limits<double>::denorm_min()/2
    <<endl;
  //digits
    cout<<"digits(float): "<<numeric_limits<float>::digits<<endl;
    cout<<"digits(double): "<<numeric_limits<double>::digits<<endl;
    cout<<"digits(int): "<<numeric_limits<int>::digits<<endl;
    cout<<"digits(_int64): "<<numeric_limits<_int64>::digits<<endl;
  //epsilon
    cout<<"epsilon(float): "<<numeric_limits<float>::epsilon()<<endl;
    cout<<"epsilon(double): "<<numeric_limits<double>::epsilon()<<endl;
    cout<<"epsilon(int): "<<numeric_limits<int>::epsilon()<<endl;
    cout<<"epsilon(long double): "<<numeric_limits<long double>::epsilon()<<endl;
  //has_denorm
    cout<<"has_denorm(float): "<<numeric_limits<float>::has_denorm<<endl;
    cout<<"has_denorm(double): "<<numeric_limits<double>::has_denorm<<endl;
```

```
        cout<<"has_denorm(int): "<<numeric_limits<long int>::has_denorm<<endl;
    //has_denorm_loss
        cout<<"has_denorm_loss(float): "<<numeric_limits<float>::has_denorm_loss<<endl;
        cout<<"has_denorm_loss(double):
"<<numeric_limits<double>::has_denorm_loss<<endl;
        cout<<"has_denorm_loss(int): "<<numeric_limits<long int>::has_denorm_loss<<endl;
    //has_infinity
        cout << "Whether float objects have infinity: "<< numeric_limits<float>::has_infinity<<
endl;
        cout << "Whether double objects have infinity: "<< numeric_limits<double>::has_infinity<<
endl;
        cout << "Whether long int objects have infinity: " << numeric_limits<long int>::has_
infinity<< endl;
    //has_quiet_NaN
        cout << "Whether float objects have quiet_NaN: "<< numeric_limits<float>::has_quiet_
NaN << endl;
        cout << "Whether double objects have quiet_NaN: "<< numeric_limits<double>::has_
quiet_NaN << endl;
        cout << "Whether long int objects have quiet_NaN: " << numeric_limits<long int>::has_
quiet_NaN << endl;
    //has_signaling_NaN
    cout << "Whether float objects have a signaling_NaN: "<< numeric_limits<float>: :has_
signaling_NaN << endl;
    cout << "Whether double objects have a signaling_NaN: "<< numeric_limits<double>::
has_signaling_NaN << endl;
    cout << "Whether long int objects have a signaling_NaN: "<< numeric_limits<long int>::has_
signaling_NaN<< endl;
    //infinity()
        cout << "The infinity for type float is: "<< numeric_limits<float>::infinity( ) <<endl;
        cout << "The infinity for type double is: "<< numeric_limits<double>::infinity( )
<<endl;
        cout << "The infinity for type long double is: "<< numeric_limits<long
double>::infinity( ) <<endl;
    //is_bounded
    cout << "Whether float objects have bounded set (是否是有界的): "<< numeric_limits <float>::
is_bounded<< endl;
    cout << "Whether double objects have bounded set (是否是有界的) : "<< numeric_ limits
<double>::is_bounded<< endl;
    cout << "Whether long int objects have bounded set (是否是有界的) : "<< numeric_limits<long
int>::is_bounded<< endl;
    cout << "Whether unsigned char objects have bounded set (是否是有界的) : "<< numeric_
limits<unsigned char>
    ::is_bounded<< endl;
    //is_exact
    cout << "Whether free  of rounding errors(float): "<< numeric_limits<float>::is_exact<<
endl;
    cout << "Whether free  of rounding errors(float): "<< numeric_limits<double>::is_exact<<
endl;
    cout << "Whether free  of rounding errors(float): "<< numeric_limits<long int>::is_exact
<< endl;
    cout << "Whether free  of rounding errors(float): "<< numeric_limits<unsigned char>::is_exact
<< endl;
```

```
    //is_iec559
    cout << "Whether conform to iec559 standards (float)(是否遵从iec559标准.): "<< numeric_
limits<float>::is_iec559
    << endl;
       cout << "Whether conform to iec559 standards (double)(是否遵从iec559标准.): "<<numeric_
limits<float>
          ::is_iec559<< endl;
       cout << "Whether conform to iec559 standards (int)(是否遵从iec559标准.): "<< numeric_
limits<float>::is_iec559
          << endl;
       cout << "Whether conform to iec559 standards (unsigned char)(是否遵从iec559标准.):
"<<numeric_limits<float>
          ::is_iec559<< endl;
    //is_integer
       cout << "Whether has an integral representation: "<< numeric_limits<float>::
is_integer<< endl;
       cout << "Whether has an integral representation: "<< numeric_limits<double>::
is_integer<< endl;
       cout << "Whether has an integral representation: " << numeric_limits<int>::is_integer
<< endl;
       cout << "Whether has an integral representation: "<< numeric_limits<unsigned
char>::is_integer<< endl;
    //is_modulo
       cout << "Whether has a modulo representation: "<< numeric_limits<float>::is_modulo<<
endl;
       cout << "Whether has a modulo representation: "<< numeric_limits<double>::is_modulo<<
endl;
       cout << "Whether has a modulo representation: "<< numeric_limits<signed char>::is_
modulo<< endl;
       cout << "Whether has a modulo representation: "<< numeric_limits<unsigned char>::is_
modulo<< endl;
    //is_signed
       cout << "Whether be a signed representation: "<< numeric_limits<float>::is_signed
<< endl;
       cout << "Whether be a signed representation: "<< numeric_limits<double>::is_signed
<< endl;
       cout << "Whether be a signed representation: " << numeric_limits<signed char>::is_
signed << endl;
       cout << "Whether be a signed representation: "<< numeric_limits<unsigned char>::is_
signed << endl;
    //is_specialized
       cout << "Whether be an explicit "<< "specialization in the class: "<< numeric_
limits<float>::is_specialized
          << endl;
       cout << "Whether be an explicit " << "specialization in the class: " << numeric_
limits<float*>::is_specialized
          << endl;
       cout << "Whether be an explicit "<<"specialization in the class: "<<numeric
_limits<int>::is_specialized<< endl;
       cout << "Whether be an explicit "<< "specialization in the class: " << numeric_
limits<int*>::is_specialized<< endl;
    //max
```

```
        cout << "The maximum value for type float is:  "<< numeric_limits<float>::max( )<<
endl;
        cout << "The maximum value for type double is:  "<< numeric_limits<double>::max( )<<
endl;
        cout << "The maximum value for type float is:  "<< numeric_limits<int>::max( )<< endl;
        cout << "The maximum value for type short int is:  " << numeric_limits<short
int>::max( )<< endl;
    //max_exponent
        cout<<"The default radix is : "<<numeric_limits<int>::radix<<endl;
        cout << "The maximum radix-based exponent for type float is:  "<< numeric_ limits
<float>::max_exponent<< endl;
        cout << "The maximum radix-based exponent for type double is:  "<< numeric_ limits
<double>::max_exponent
        << endl;
        cout << "The maximum radix-based exponent for type long double is:  " << numeric_
limits<long double>
            ::max_exponent<< endl;
        cout << "The maximum radix-based exponent for type int is:  " << numeric_limits<int>::
max_exponent<< endl;
    //max_exponent10
        cout<<"The default radix is : "<<numeric_limits<int>::radix<<endl;
        cout << "The maximum base 10 exponent for type float is:  "<< numeric_ limits<float>::
max_exponent10 << endl;
        cout << "The maximum base 10 exponent for type double is:  "<< numeric_ limits<double>::
max_exponent10 \
            << endl;
        cout << "The maximum base 10 exponent for type long double is:  "<< numeric
_limits<longdouble>::
            max_exponent10 << endl;
    //min
    cout << "The minimum value for type float is:  "<< numeric_limits<float>::min( )<< endl;
    cout << "The minimum value for type double is:  "<< numeric_limits<double>::min( )<<
endl;
    cout << "The minimum value for type float is:  "<< numeric_limits<int>::min( )<< endl;
    cout << "The minimum value for type short int is:  "<< numeric_limits<short int>::min( )<<
endl;
    //min_exponent
    cout << "The minimum radix-based exponent for type float is:  "<< numeric_limits
<int>::min_exponent<< endl;
    cout << "The minimum radix-based exponent for type float is:  "<< numeric_limits
<float>::min_exponent<< endl;
    cout << "The minimum radix-based exponent for type double is:  "<< numeric_
limits<double>::min_exponent<< endl;
    cout << "The minimum radix-based exponent for type long double is: "<< numeric_limits<long
double>::min_exponent
            << endl;
    //min_exponent10
    cout << "The minimum base 10 exponent for type float is: "<< numeric_limits<int> ::
min_exponent10<< endl;
    cout << "The minimum base 10 exponent for type float is: "<< numeric_limits<float> ::
min_exponent10 << endl;
    cout << "The minimum base 10 exponent for type double is: "<< numeric_limits<double> ::
```

```
min_exponent10<< endl;
    cout << "The minimum base 10 exponent for type long double is:  "<< numeric_limits<long
double>::min_exponent10
          << endl;
    //quiet_NaN
      cout << "The quiet NaN for type float is:  "<< numeric_limits<float>::quiet_NaN( )<<
endl;
      cout << "The quiet NaN for type int is:  " << numeric_limits<int>::quiet_NaN( )<< endl;
      cout << "The quiet NaN for type long double is:  "<< numeric_limits<double>::
quiet_NaN( )<< endl;
    //radix
      cout << "The base for type float is:  "<< numeric_limits<float>::radix << endl;
      cout << "The base for type int is:  " << numeric_limits<int>::radix << endl;
      cout << "The base for type long double is:  " << numeric_limits<long double>::radix
<< endl;
    //round_error
      cout << "The maximum rounding error for type float is:  "<< numeric_ limits
<float>::round_error( )<< endl;
      cout << "The maximum rounding error for type int is:  " << numeric_ limits<int>::
round_error( )<< endl;
     cout << "The maximum rounding error for type double is:  "<< numeric_ limits
<double>::round_error( )<< endl;
    //round_style
      cout << "The rounding style for a double type is: " << numeric_limits<double>::
round_style << endl;
      cout << "The rounding style for a double type is now: " << numeric_
limits<float>::round_style << endl;
      cout << "The rounding style for an int type is: " << numeric_limits<int>::round_style
<< endl;
    //signaling_NaN
      cout << "The signaling NaN for type float is:  "<< numeric_limits<float> ::signaling_NaN
( )<< endl;
      cout << "The signaling NaN for type int is:  "<< numeric_limits<int>::signaling_NaN
( )<< endl;
      cout << "The signaling NaN for type double is:  "<< numeric_limits<double> ::signaling_NaN
( )<< endl;
    //tinyness_before
       cout << "Whether float types can detect tinyness before rounding: "<< numeric_
limits<float>::tinyness_before<< endl;
       cout << "Whether double types can detect tinyness before rounding: "<< numeric_
limits<double>::tinyness_before
          << endl;
       cout << "Whether long int types can detect tinyness before rounding: "<< numeric_
limits<long int>::tinyness_before
          << endl;
       cout << "Whether unsigned char types can detect tinyness before rounding: "<< numeric_
limits<unsigned char>::
          tinyness_before<< endl;
    //traps
       cout << "Whether float types have implemented trapping: "<< numeric_
limits<float>::traps<< endl;
       cout << "Whether double types have implemented trapping: "<< numeric_
```

```
limits<double>::traps<< endl;
        cout << "Whether long int types have implemented trapping: "<< numeric_limits<long
int>::traps<< endl;
    cout << "Whether unsigned char types have implemented trapping: "<< numeric_limits
<unsigned char>::traps<< endl;
    }
```

程序的执行结果如下：

```
    The smallest nonzero denormalized value(float). object is: 1.4013e-045
    The smallest nonzero denormalized value(float). object is: 0
    The smallest nonzero denormalized value(float). object is: 4.94066e-324
    The smallest nonzero denormalized value(float). object is: 0
    The smallest nonzero denormalized value(float). object is: 0
    The smallest nonzero denormalized value(float). object is: 0
    digits(float): 24
    digits(double): 53
    digits(int): 31
    digits(_int64): 0
    epsilon(float): 1.19209e-007
    epsilon(double): 2.22045e-016
    epsilon(int): 0
    epsilon(long double): 2.22045e-016
    has_denorm(float): 1
    has_denorm(double): 1
    has_denorm(int): 0
    has_denorm_loss(float): 1
    has_denorm_loss(double): 1
    has_denorm_loss(int): 0
    Whether float objects have infinity: 1
    Whether double objects have infinity: 1
    Whether long int objects have infinity: 0
    Whether float objects have quiet_NaN: 1
    Whether double objects have quiet_NaN: 1
    Whether long int objects have quiet_NaN: 0
    Whether float objects have a signaling_NaN: 1
    Whether double objects have a signaling_NaN: 1
    Whether long int objects have a signaling_NaN: 0
    The infinity for type float is: 1.#INF
    The infinity for type double is: 1.#INF
    The infinity for type long double is: 1.#INF
    Whether float objects have bounded set (是否是有界的): 1
    Whether double objects have bounded set (是否是有界的) : 1
    Whether long int objects have bounded set (是否是有界的) : 1
    Whether unsigned char objects have bounded set (是否是有界的) : 1
    Whether free  of rounding errors(float): 0
    Whether free  of rounding errors(float): 0
    Whether free  of rounding errors(float): 1
    Whether free  of rounding errors(float): 1
    Whether conform to iec559 standards (float)(是否遵从 iec559 标准.): 1
    Whether conform to iec559 standards (double)(是否遵从 iec559 标准.): 1
    Whether conform to iec559 standards (int)(是否遵从 iec559 标准.): 1
    Whether conform to iec559 standards (unsigned char)(是否遵从 iec559 标准.): 1
    Whether has an integral representation: 0
    Whether has an integral representation: 0
    Whether has an integral representation: 1
    Whether has an integral representation: 1
```

```
Whether has a modulo representation: 0
Whether has a modulo representation: 0
Whether has a modulo representation: 1
Whether has a modulo representation: 1
Whether be a signed representation: 1
Whether be a signed representation: 1
Whether be a signed representation: 1
Whether be a signed representation: 0
Whether be an explicit specialization in the class: 1
Whether be an explicit specialization in the class: 0
Whether be an explicit specialization in the class: 1
Whether be an explicit specialization in the class: 0
The maximum value for type float is:  3.40282e+038
The maximum value for type double is:  1.79769e+308
The maximum value for type float is:  2147483647
The maximum value for type short int is:  32767
The default radix is : 2
The maximum radix-based exponent for type float is:  128
The maximum radix-based exponent for type double is:  1024
The maximum radix-based exponent for type long double is:  1024
The maximum radix-based exponent for type int is:  0
The default radix is : 2
The maximum base 10 exponent for type float is:  38
The maximum base 10 exponent for type double is:  308
The maximum base 10 exponent for type long double is:  308
The minimum value for type float is:  1.17549e-038
The minimum value for type double is:  2.22507e-308
The minimum value for type float is:  -2147483648
The minimum value for type short int is:  -32768
The minimum radix-based exponent for type float is:  0
The minimum radix-based exponent for type float is:  -125
The minimum radix-based exponent for type double is:  -1021
The minimum radix-based exponent for type long double is:  -1021
The minimum base 10 exponent for type float is: 0
The minimum base 10 exponent for type float is:  -37
The minimum base 10 exponent for type double is:  -307
The minimum base 10 exponent for type long double is:  -307
The quiet NaN for type float is:  -1.#IND
The quiet NaN for type int is:  0
The quiet NaN for type long double is:  -1.#IND
The base for type float is:  2
The base for type int is:  2
The base for type long double is:  2
The maximum rounding error for type float is:  0.5
The maximum rounding error for type int is:  0
The maximum rounding error for type double is:  0.5
The rounding style for a double type is: 1
The rounding style for a double type is now: 1
The rounding style for an int type is: 0
The signaling NaN for type float is:  -1.#INF
The signaling NaN for type int is:  0
The signaling NaN for type double is:  -1.#INF
Whether float types can detect tinyness before rounding: 1
Whether double types can detect tinyness before rounding: 1
Whether long int types can detect tinyness before rounding: 0
Whether unsigned char types can detect tinyness before rounding: 0
Whether float types have implemented trapping: 1
```

```
Whether double types have implemented trapping: 1
Whether long int types have implemented trapping: 0
Whether unsigned char types have implemented trapping: 0
```

10.3 程序的启动和终止

在前面的章节已经讲述了程序的启动和终止。本节头文件<cstdlib>中，包含宏 EXIT_FAILURE 和 EXIT_SUCCESS。还包含 3 个函数：abort()、atexit()、exit()。这 3 个函数在前面的第 8 章已经简单讲述过，在本节将详细地讲述其使用方法。

在头文件<stdlib.h>中，其内容和头文件<cstdlib>大致相似。

1. 函数 abort()

在国际标准中，函数 abort()具有额外的行为。对于自动静态存储对象，不调用对象的析构器，并且不调用函数 atexit()。函数 atexit()的声明形式如下：

```
extern  "C"  int  atexit (void(*f)(void));
extern  "C"  int  atexit(void(*f)(void));
```

功能：函数 atexit()将注册指针 f 指向的函数，并且指针 f 指向的函数不带参数。当执行被函数 atexit()注册的函数时，由于该指针 f 指向的函数不提供句柄，并且无法抛出异常， 一旦指针 f 指向的函数不能被有效控制，函数 terminate()将被调用。

局限性：在使用 atexit()时，代表函数的指针 f 至少可以支持注册 32 种函数。

返回值：如果指针 f 指向的函数注册成功，函数 atexit()返回 0，否则函数 atexit()返回非零值。

2. exit(int status)

在国际标准中，函数 exit()也可具有额外的功能。

（1）首先，具有静态存储类型的对象被破坏，通过函数 atexit()注册的函数被调用。具有静态存储类型的非局部类型对象被破坏，破坏顺序是按其构造器被执行顺序的逆序执行的。当调用 exit()时，自动存储类型的对象不会被破坏。被函数 atexit()注册的函数是按其最初注册顺序的逆序执行的。

除此之外，任何之前注册的函数调用之后，其他函数才会被调用。在静态存储类型的非局部型对象被初始化之前，使用函数 atexit()注册的函数是不会被调用的，直到该对象的析构过程结束。在非局部型的静态存储类型对象初始化之后，该对象的析构过程开始之前，使用函数 atexit()注册的任意函数，才会被调用。任意函数调用局部的静态类型对象的析构器时，如果该析构器是在该对象的构造器的末尾被函数 atexit()注册的，该局部静态类型对象才会被破坏。

（2）其次，具有不可写入的缓冲区数据被刷新时，所有的开放 C 流被关闭，并且所有使用函数 tmpfile()创建的临时文件，均被删除。

（3）最后，函数将返回至调用该函数的环境。如果参数 status 是 0 或宏 EXIT_SUCCESS，函数返回的状态为“成功终止程序”。如果参数是 EXIT_FAILURE，函数返回的状态的形式为“未成功终止”。否则，返回的状态是由实际环境决定的。函数 exit()绝不会返回到它的调用者，而是直接终止程序。

10.4 动态内存管理

头文件<new>定义了几种函数，这些函数负责程序中动态内存的分配。同时，头文件也定义了

负责报告内存管理错误的部件。在头文件<new>中，包含以下内容：

```
namespace std {
class  bad_alloc;
struct  nothrow_t{ };
extern const nothrow_t nothrow;
typedef void(*new_handler)();
new_handler  set_new_handler(new_handler  new_p) throw();
}
void*  operator new(std::size_t  size) throw (std::bad_alloc);
void*  operator new(std::size_t  size, const std::nothrow_t& )  throw();
void*  operator delete(void* ptr) throw();
void*  operator delete(void* ptr, const std::nothrow_t&) throw();
void*  operator new[](std::size_t size) throw(std::bad_alloc);
void*  operator new[](std::size_t size, const std::nothrow_t&) throw();
void*  operator delete[](void* ptr) throw();
void*  operator delete[](void* ptr, const std::nothrow_t&) throw();
void*  operator new(std::size_t size, void*ptr) throw();
void*  operator new[](std::size_t size, void* ptr) throw();
void*  operator delete(void* ptr, void*) throw();
void*  operator delete[](void* ptr, void*) throw();
```

10.4.1　内存的分配和释放

首先介绍操作符 new 和 delete。

1．单个对象的形式

操作符 new 的声明形式如下：

```
void*  operator  new(std:: size_t  size) throw(std::bad_alloc);
```

使用 new 表达式实现分配内存空间，参数 size 代表分配内存的字节大小。该内存代表了对象的大小。

在 C++程序中，可能定义任意函数，该函数的命名标志可以取代 C++标准库中定义的默认版本。如果分配内存成功，new 操作符可以返回一个非空指针；否则会抛出 bad_alloc 类型的异常。

操作符 new 还具有一些默认的行为法则。

（1）执行循环。在循环中，函数通常首先尝试分配必要的内存。是否尝试调用 malloc 函数，也不是特定的。

（2）如果尝试成功，会返回指针，该指针指向分配的内存。否则，如果函数 set_handler()的最后一个参数是一个空指针的话，会抛出异常 bad_alloc。

（3）要不然，函数会调用当前的句柄（new_handler）。如果被调用的函数返回，循环会重复执行。

（4）当尝试分配需要的内存成功时，或者被调用的函数句柄（new_handler）没有正常返回时，循环会中止。

```
void*  operator new(std::size_t size, const std::nothrow_t&) throw();
```

和第一种 new 的用法相似。唯一不同的是：当上述形式的 new 被调用时，C++程序会提供一个空指针作为错误标志，而不是抛出 bad_alloc 类型的异常。

同样，任意一个 C++程序会定义一个函数，该函数可以替换 C++标准库中的默认版本。

当内存分配适当时，会返回一个非空指针。否则，会返回一个空指针。操作符 new 的 nothrow 版本返回的指针和常规版本返回的指针是一样的。函数的替代版本同样需要满足这个要求。

同样，第二种 new 操作符也会执行一些默认行为法则。

（1）执行一个循环。在循环执行过程中，函数首先尝试分配必要的内存。该尝试涉及调用标准 C 库函数 malloc()。

（2）如果尝试成功，返回指向被分配内存的指针。否则，如果函数 set_new_handler()的最后一个参数是空指针，函数 new 会返回空指针。

（3）要不然，函数调用当前的 new_handler。如果被调用的函数返回，循环会继续重复执行。

（4）当尝试分配必要的内存成功时，或者当被调用的 new_handler 代表的函数没有返回时，循环会中止。如果通过抛出异常的形式，被调用的 new_handler 中止，函数 new()会返回一个空指针。

举例如下：

```
T* p1=new T;                        //如果分配内存失败，会抛出bad_alloc类型的异常
T* p2=new(nothrow) T;               //如果分配内存失败，会返回0
```

操作符 delete 的两种使用形式：

```
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

功能描述：释放内存的功能是通过 delete 表达式被调用的，目的是使指针 ptr 的值无效。任意一个 C++程序可能会定义一个函数，该函数签名可以取代 C++标准库中的默认版本。之前调用函数 new 时，指针 ptr 获取到相应的内存指针或空指针。同样，函数 delete 也存在以下默认属性。

（1）对于空指针 ptr，不采取任何措施；

（2）当 ptr 指针是非空数值时，其数值应该是函数 new 被调用时赋予的值，通过对其调用函数 delete，使该指针无效。对于这样的非空指针，delete 函数执行之后，会收回之前使用默认 new 函数开辟的内存。

提示：对于 new 函数或其他的函数（calloc、malloc、realloc 等）开辟的内存，并不能确定在什么条件下才能回收这些内存。

2. 数组形式（array forms）

```
void* operator  new[](std::size_t  size) throw(std::nad_alloc);
```

功能描述：当调用分配内存的函数，开辟数组形式的内存时，仍然使用 new 表达式。内存大小是 new 的参数 size，数组的大小应该小于或等于 new 的参数 size。同样，在 C++程序中可以定义函数，该函数的签名可以替换 C++标准库中的默认版本。函数返回值是指针类型。

```
void* operator new[ ](std::size_t  size, const std::nothrow_t&) throw();
```

功能描述：和上述内容相同，除了在调用 new 函数失败时，返回空指针之外，并不抛出 bad_alloc 类型的异常。而 new[ ]函数调用失败之后，会抛出 bad_alloc 类型的异常。任意一个 C++程序可以定义一个函数，该函数的签名可以替换 C++标准库中的默认版本（同名）。

两种 new[ ]函数的使用方法是相似的，函数返回值类型是一致的。

```
void operator delete[](void* ptr)  throw();
void operator delete[](void* ptr, const std::nothrow_t&)  throw();
```

功能描述：数组形式的内存释放函数（delete[ ]）在被调用时，主要目的是使指针 ptr 无效。

对于 C++程序，在程序中定义的函数可以替换 C++标准库中的同名默认版本。函数 delete[]的参数 ptr 允许的值可能是空指针，或者 ptr 的值是之前调用 new[]获得的内存指针。同样，在使用函数 delete 时，有几条需要注意：

（1）如果指针 ptr 的值是空指针，函数 delete 不执行任何措施；

（2）如果指针 ptr 的值是其他数值，其值应该是之前调用 new[]函数时获取的内存指针。对于一个非空值的指针 ptr，将回收其之前被分配的内存；

（3）头文件<cstdlib>中声明的内存分配函数主要包括 calloc、malloc 或 realloc()。调用这些函数会分配相应的内存，无论在什么条件下，使用这些函数可以重新分配 delete[]回收的内存。

3. Placement forms

这些函数是被保留的，在 C++程序中可能没有定义一些函数，这些函数可以替代 C++标准程序库中函数的默认版本。这些提供没有应用至这些保留的 new 和 delete 的 placement 形式中。

形式一：

```
void* operator new (std::size_t size, void* ptr) throw(();
```

函数返回指针 ptr。函数的目的是执行其他的行为。

举例如下：

```
void* place=operator new(sizeof(something));
something* p= new (place) something();
```

形式二：

```
void* operator new[](std::size_t size, void* ptr) throw();
```

函数返回值为 ptr。函数的目的是执行其他的行为。

形式三：

```
void* operator delete(void* ptr, void*) throw()
```

同样，其目的是不执行任何行为。通过采用抛出异常的形式，在 new 表达式初始化时，被调用的默认函数会被终止。

形式四：

```
void* operator delete[]( void* ptr , void* ) throw()
```

不执行任何行为。通过采用抛出异常的形式，在 new 表达式初始化时，被调用的默认函数会被终止。

10.4.2 内存分配错误

类 bad_alloc 的声明形式如下：

```
namespace std{
class bad_alloc: public exception{
public:
bad_alloc() throw();
bad_alloc(const bad_alloc&) throw();
bad_alloc& operator=(const bad_alloc&) throw();
virtual ~bad_alloc() throw();
virtual const char* what() const throw();
```

```
    }
}
```

类 bad_alloc 定义诸多异常对象的类型。这些异常对象是在分配内存失败时，被抛出的异常类型对象。下面分别介绍类中的各个函数。

1. bad_alloc() throw()

使用类 bad_alloc 构造一个对象，函数 bad_alloc()恰好是类的构造器。

调用函数 what()时，会导致被构造的新对象被定义。

2. bad_alloc(const bad_alloc&) throw()

```
bad_alloc (const bad_alloc&) throw();
bad_alloc& operator=(const bad_alloc&) throw();
```

这两个函数的功能是实现完成一个类 bad_alloc 的备份。

3. virtual const char* what() const throw()

函数返回值是一个 NTBS 。

4. typedef new_handler

函数 new_hanlder 产生一个新的句柄。新产生的 new_handler 类型的句柄应该执行下述行为：

（1）尽量开辟更多的内存，并返回该内存；

（2）抛出 bad_alloc 类型的异常，或者类 bad_alloc 的派生类类型的异常；

（3）调用 abort()或者 exit()。

5. set_new_hanlder

函数声明如下：

```
new_handler  set_new_handler(new_handler  new_p)  throw();
```

函数描述：设置参数 new_p 指定的函数作为当前的 new_handler。

函数返回值：第一次调用时，函数返回 0；函数返回值是之前设置的 new_handler 类型的句柄。

10.4.3 应用举例

下面的代码阐述了使用 C++的 new 和 placement new 来创建类的实例，以及使用 delete 和显示调用类的析构函数来释放内存，在示例的类的实现中同时打印出了该类被实例化后的内存地址（this 指针值），也简单演示了 std::bad_alloc 的用法。

针对 10.4.1 节和 10.4.2 节的内容，使用例 10-2 和例 10-3 具体说明上述相关内容的使用方法。

例 10-2

```
#include <iostream>
#include <new>
using namespace std;
class Myclass{
public:
int imember;
public:
Myclass()
{
cout<<"Construction MyClass. "<<this<<endl;
```

```
}
~Myclass()
{
imember=0;
cout<<"Destructing MyClass. "<<this<<endl;
}
};
void __cdecl myfun()
{
cout<<"handler test."<<endl;
throw bad_alloc();
return;
}

void main()
{
//1
Myclass* fpM=new Myclass;
delete fpM;
char x[sizeof(Myclass)];
Myclass* fpM2=new(&x[0]) Myclass;
fpM2->~Myclass();
cout<<"The address of x[0] is : "<<(void*)&x[0]<<endl;
Myclass* fpM3=new Myclass;
delete fpM3;
//2
char* int_p;
try{
//int* int_p=new int[99999];
int_p=new char[~size_t(0)/2];
   delete[] int_p;
}
catch(bad_alloc &e)
{
cout<<e.what()<<endl;
}
//3
bad_alloc ba;
cout<<ba.what()<<endl;
}
```

程序的执行结果如下：

```
Construction MyClass. 00393710
Destructing MyClass. 00393710
Construction MyClass. 0012FF48
Destructing MyClass. 0012FF48
The address of x[0] is : 0012FF48
Construction MyClass. 00393710
Destructing MyClass. 00393710
bad allocation
bad allocation
```

例 10-3

```
#include<new>
#include<iostream>
using namespace std;
void __cdecl newhandler( )
```

```
{
   cout << "The new_handler is called:" << endl;
   throw bad_alloc( );
   return;
}
int main( )
{
   set_new_handler (newhandler);
   try
   {
      while ( 1 )
      {
         new int[5000000];
         cout << "Allocating 5000000 ints." << endl;
      }
   }
   catch ( exception e )
   {
      cout << e.what( ) << " xxx" << endl;
   }
   cin.get();
}
```

程序的执行结果如下：

```
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
Allocating 5000000 ints.
… …
Allocating 5000000 ints.
Allocating 5000000 ints.
The new_handler is called:
bad allocation xxx
```

10.5 类型标识符

本节主要讲述了 3 个类：type_info 类、bad_cast 类和 bad_typeid 类。这 3 个类分别负责：类型信息，错误抛出，错误类型 id。在头文件<typeinfo>中，定义和类型信息相关的一种类型。同时，也定义了用于报告动态类型识别错误的两种类型。

10.5.1 类 type_info

```
namespace std{
class type_info{
public:
virtual  ~type_info();
bool  operator==(const type_info& rhs) const;
bool  operator!=(const type_info& rhs) const;
bool  before(const type_info& rhs) const;
const char* name() const;
```

```
private:
type_info(const type_info&  rhs);
type_info& operator=(const type_info& rhs);
    };
}
```

类 type_info 描述类型信息。类的对象有效地存储了一个类型的指针以及适合两个类型比较（或排序）的加密数值。类型的名称、编码规则和排序顺序均是不确定的，并且在程序间存在差异。

```
bool  operator = =(const type_info& rhs) const
```

功能描述：使用 rhs 和当前对象相比较。如果两个数值是统一数据类型，函数返回 true。

```
bool  operator!=(const type_info& rhs) const;
```

函数返回值是：!(*this = = rhs)。

```
bool  before (const type_info& rhs) const;
```

功能描述：使用参数 rhs 和当前对象进行比较。

函数返回值：如果当前对象在排序序列中位于 rhs 之前，返回 true。

```
const char* name() const;
```

函数返回值：一个已定义的 NTBS。

提示：信息可能是以空（null）为终止符的多字节字串，适合作为一个宽字符串（wstring）转换和显示。

```
type_info(const type_info& rhs);
type_info& operator=(const type_info& rhs);
```

函数的功能是实现类 type_info 对象的复制。

提示：由于类 type_info 的可复制的构造器和可赋值的操作符是私有成员，该类型的对象是不能被复制的。

使用 type_info 类时，是比较麻烦的，因为其构造函数是私有成员。

10.5.2　类 bad_cast

类 bad_cast 的声明如下：

```
    namespace std{
    class bad_cast : public exception{
    public:
    bad_cast() throw();
    bad_cast(const bad_cast&) throw();
    bad_cast& operator=(const bad_cast&) throw();
    virtual ~bad_cast() throw();
    virtual const char* what() const throw();
    }
}
```

类 bad_cast 定义异常对象的类型，这些异常对象在实现过程中会被抛出，用于汇报一个无效的 dynamic-cast 表达式的执行错误。

函数 bad_cast() throw()是该类的构造函数。构造器可以构造一个该类的对象。使用新构造的对

象调用函数 what()时，其结果是预先定义的。

构造函数还有一种形式：

```
bad_cast(const bad_cast& ) throw();
bad_cast& operator=(const bad_cast& ) throw();
```

功能描述：创建类 bad_cast 的对象或者该类对象的复制。

```
virtual const char* what() const throw();
```

函数返回值是已定义的 NTBS。提示：返回值的提示信息是以空（null）为结束标志的多字节字符串，也适合用于宽字节字符串 wstring 的转换和显示。

10.5.3 类 bad_typeid

类 bad_typeid 的声明如下：

```
    namespace  std{
    class bad_typeid : public excetion{
    public:
    bad_typeid()  throw();
    bad_typeid(const  bad_typeid&) throw();
    bad_typeid& operator = (const bad_typeid& ) throw();
    virtual ~bad_typeid() throw();
    virtual const char* what() const throw();
    }
}
```

类 bad_typeid 定义了异常对象的类型。在程序执行时，如果在 typeid 类型的表达式中发生空指针现象，这些异常对象会被以异常的形式抛出。

构造函数 bad_typeid() throw()，函数的功能是构造一个 bad_typeid 类型的对象。

提示：对于新生成的对象，调用函数 what()的结果是预先定义的。

```
bad_typeid(const  bad_typeid& ) throw();
bad_typeid& operator = (const bad_typeid& ) throw();
```

功能描述：实现复制类 bad_typeid 的对象。

```
virtual const char* what()  const  throw();
```

函数返回值是已定义的 NTBS。

提示：返回值信息是一个以空（null）为结束符的多字节字符串，也适合作为宽字节字符串的转换和显示。

10.5.4 操作符 typeid

typeid 操作符使程序能够询问某表达式的类型。

typeid 操作符的表达式为：

```
typeid(e);
```

e 是任意表达式或者是类型名。

如果表达式的类型是“类”类型，并且该类包含一个或多个虚函数，则表达式的动态类型可能不同于它的静态编译时的类型。例如：如果表达式对基类指针引用，则该表达式的静态编译时类型

是基类类型；但如果指针实际指向派生类对象，则 typeid 操作符将指出表达式的类型是派生类型。

typeid 操作符可以与任何类型的表达式一起使用。内置类型的表达式以及常量都可以用作 typeid 操作符的操作数。如果操作数不是“类”类型或者是没有虚函数的类，则 typeid 操作符指出操作数的静态类型；如果操作数是定义至少一个虚函数的“类”类型，则在运行时计算类型。

typeid 操作符的结果为 type_info 的标准库类型的对象引用，后面将更详细地讨论这个类型。使用 type_info 类，必须包含库头文件<typeinfo>。

对于 typeid 最常见的用途是比较两个表达式的类型，或者将表达式的类型与特定类型相比较。

10.5.5　操作符 dynamic_cast

使用 dynamic_cast 操作符将基类类型对象的引用或指针转换为同一继承层次中其他类型的引用或指针。与 dynamic_cast 一起使用的指针必须是有效的（它必须是 0 或指向一个对象）。

与其他强制类型转换不同，dynamic_cast 涉及运行时类型检查。如果绑定到引用或指针的对象不是目标类型的对象，则 dynamic_cast 失败。如果转换到指针类型的 dynamic_cast 失败，则 dynamic_cast 的结果是 0 值。如果转换到引用类型的 dynamic_cast 失败，则抛出一个 bad_cast 类型的异常。

dynamic_cast 操作符一次执行两个操作。首先验证被请求的转换是否有效，只有转换有效，操作符才实际进行转换。一般而言，引用或指针所绑定的对象的类型在编译时是未知的，基类的指针可以赋值为指向派生类对象，同样，基类的引用可以用派生类对象初始化，因此 dynamic_cast 操作符执行的验证必须在运行时进行。

使用 dynamic_cast 可将基类指针转换为派生类指针，可以使用 dynamic_cast 将基类引用转换为派生类引用，这种 dynamic_cast 的操作形式如下：

```
dynamic_cast<Type& >(val)
```

Type 是转换的目标类型，而 val 是基类类型的对象。仅当 val 实际引用一个 Type 类型对象时，或者 val 是一个 Type 派生类的对象时，dynamic_cast 操作才将操作数 val 转换为希望的 Type&类型。如果不存在空引用，不可能对引用使用用于指针强制类型转换的检查策略，相反当转换失败时，抛出一个 std::bad_cast 异常，该异常在库头文件 typeinfo 中定义。

10.5.6　应用举例

本节使用例 10-4 来阐述 10.5 节相关内容的使用方法。

下面的例子展示了 typeid 的用法，使用该内置操作符可以获取实例的类型 std::type_info，调用其成员函数 name()可打印出类的名字信息；其次该例子还展示了 dynamic_cast 的用法，代码中类 Ellipse 继承自类 Circle，当使用者试图从基类 cast 到子类时，会抛出异常。

例 10-4

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Circle {
public:
virtual void myfunc() const {}
};
```

```
class Ellipse: public Circle {
public:
virtual void myEllipse() const
 {
};
};
void main()
{
//typeid
int ti_obj=20;
string tstr="I am male.";
double td=32.56;
float tfl=21.075;
const type_info& ti=typeid(ti_obj);
const type_info& tif=typeid(tstr);
const type_info& tid=typeid(td);
const type_info& tifl=typeid(tfl);
cout<<ti.name()<<" ;"<<ti.raw_name()<<endl;
cout<<tif.name()<<endl;
cout<<tid.name()<<endl;
cout<<tifl.name()<<endl;
//dynamic_cast  bad_cast
   Circle instance;
   Circle& ref_shape = instance;
   try {
      Ellipse& ref_circle = dynamic_cast<Ellipse&>(ref_shape);
   }
   catch (bad_cast) {
      cout << "Caught: bad_cast exception. A Shape is a Circle.\n";
   }
//bad_typeid
   Circle* p=NULL;
   try{
   cout<<typeid(*p).name()<<endl;
   }
   catch(bad_typeid)
   {
   cout<<"Object is NULL."<<endl;
   }
   cout<<"//..."<<endl;
   Circle* pp=new Circle();
   try{
   cout<<typeid(*pp).name()<<endl;
   }
   catch(bad_typeid)
   {
   cout<<"Object is NULL."<<endl;
   }
}
```

程序的执行结果如下：

```
int ;.H
```

```
class std::basic_string<char,struct std::char_traits<char>,class std::allocator<
char> >
double
float
Caught: bad_cast exception. A Shape is a Circle.
Object is NULL.
//...
class Circle
请按任意键继续...
```

10.6　异常处理

前面的章节已经详细地讲述了异常处理。本节对和语言支持相关的部分内容详加讲述。在 C++ 程序中，需要包含头文件<exception>。在头文件<exception>中，定义了几个类型和函数，这些类型和函数均是和异常处理相关的。头文件<exception>中包含以下内容：

```
namespace std{
   class  exception;
   class  bad_exception;
   typedef  void  (*unexpected_handler)();
   unexpected_handler  set_unexpected(unexpected_handler  f) throw();
   void unexpected();
   typedef void(*terminate_handler)();
   terminate_handler  set_terminate(terminate_handler  f) throw();
   void terminate();
   bool  uncaught_exception()  throw();
   }
```

10.6.1　异常类（class exception）

类 exception 的声明如下：

```
namespace  std {
class exception{
public:
exception() throw();
   exception(const char * const &message);
   exception(const char * const &message, int);
exception(const  exception& ) throw();
exception& operator=(const  exception& ) throw();
virtual ~exception() throw();
virtual const char* what() const throw();
   }
}
```

类 exception 定义了异常对象类型的基类。在程序执行过程中，如果发生并汇报检测到的相关的错误，这些异常对象会被作为异常抛出。

构造函数如下：

```
exception() throw();
```

构造函数 exception() 可以用于实现构造类 exception 的对象，但不能抛出任何异常。

```
exception ( const exception& ) throw();
exception& operator = (const exception& ) throw();
```

上述两个函数实现对类 exception 对象的复制或备份。

提示： 尤其是赋值之后，调用 what()的返回结果是预先定义的。

```
virtual ~exception()  throw();
```

函数描述：破坏类 exception 的对象，释放内存。

提示： 不抛出异常。

```
virtual const char* what() const throw();
```

函数返回值是 NTBS。

提示： 返回的信息是以空（null）为结束符的多字节字符串。也适合宽字节字符的转换和显示。函数的返回值保持有效，直到捕获的异常对象被破坏或者异常对象的非常量成员函数被调用。

10.6.2 violating exception-specifications

1. 类 bad_exception

类 bad_exception 的声明如下：

```
namespace std{
class bad_exception: public exception{
public:
bad_exception() throw();
bad_exception(const bad_exception& ) throw();
bad_exception& operator = (const bad_exception& ) throw();
virtual ~bad_exception() throw();
virtual const char* what() const throw();
    }
}
```

类 bad_exception 定义被抛出的异常对象的类型。
类的构造函数如下：

```
bad_exception()  throw();
```

函数的功能描述：构造类 bad_exception 的对象。

提示： 对于新构造的对象，调用函数 what()时，结果是已经定义的。

```
bad_exception(const  bad_exception& ) thow();
bad_exception& operator= (const bad_exception& ) throw();
```

函数描述：复制类 bad_exception 的对象，构造并产生新的对象。

```
virtual const char* what()  const  throw();
```

函数返回值是已定义的 NTBS。返回的信息是以空（null）为结束符的多字节字符串，也适用于宽字节的转换和显示。

2. 类型 unexpected_handler

```
typedef void (*unexpected_handler) ();
```

当函数尝试抛出一个异常时（该异常并未在异常说明中列出），句柄函数会通过函数 unexpected()调用。

一个非期望的句柄（unexpected_handler）不会被返回。默认的 unexpected_handler 调用会终止。

3. set_unexpected

```
unexpected_handler set_unexpected (unexpected_handler  f)  throw();
```

函数的功能描述：创建使用指针 f 指定的函数，作为当前的 unexpected_handler 句柄。值得注意的是，f 不应该是空指针，函数返回值是之前的 unexpected_handler 型句柄。

4. unexpected

函数声明如下：

```
void unexpected()
```

当函数存在时，通过异常调用，也可以在程序中直接调用。在 ISO_IEC-14882-2003_C++中，此段英文原文应该是：Called by the implementation when a function exits via an exception not allowed by its exception specification。May also be called directly by the program。

在评估 throw 表达式时，快速地、有效地调用 unexpected_handler 类型函数。如果在执行过程中被调用，或者调用当前 unexpected_handler，也可以在程序中直接调用。

10.6.3　abnormal termination

本小节主要讲述用于处理异常的 3 个函数：terminate_handler()，set_terminate()和 terminate()。其中，函数 terminate_handler()是用于处理句柄的；函数 set_handler()是用于设置句柄的；terminate()可以直接终止程序。

1. type terminate_handler

```
typedef void (*terminate_handler)();
```

该句柄型函数被调用是通过调用函数 terminate()实现的，并终止异常处理的过程。

任一 terminate_handler 会终止程序的执行，而并不返回至函数的调用者。

在贯彻执行默认的 terminate_handler 时，会调用函数 abort()。

2. set_terminate

```
terminate_handler set_terminate(terminate_handler  f)  throw();
```

函数功能描述：设置参数 f 指定的函数作为当前句柄函数，用于终止异常处理过程。值得一提的是，f 不能是空指针。函数返回值是之前的 terminate_handler 型句柄。

3. terminate

```
void terminate()
```

在执行过程中，异常句柄由于其他原因必须被离弃时，此函数会被调用。也可以在程序中直接调用。

评估 throw 表达式之后，有效地、快速地调用 terminate_handler 类型函数。如果在执行过程中

被调用，或调用当前 terminate_handler 类型函数时，也可以被程序直接调用。

10.6.4 未捕获异常（uncaught_exception）

```
bool  uncaught_exception() throw();
```

函数 uncaught_exception 的返回值：返回 true 的条件是：完成对 throw 表达式评估之后，直至完成异常声明的初始化，或由于 throw()而输入 unexpected()；或输入 terminate()之后而没有显式地调用 terminate()。

提示：当函数 uncaught_exception()返回值为 true 时，抛出一个异常可以导致调用函数 terminate()。

10.6.5 应用举例

本节使用例 10-5 阐述 10.6 节相关内容的使用方法。

例 10-5

```
#include <iostream>
#include <exception>
using namespace std;
void Tfunction( )
{
cout << "1. I'll be back." << endl;
}
void termfunction( )
{
   cout << "2. I'll be back." << endl;
   exit(1);
}
class Test
{
public:
   Test( std::string msg ) : m_msg( msg )
   {
  std::cout << "In Test::Test()" << std::endl;
   }
   ~Test( )
   {
  std::cout << "In Test::~Test()" << "      std::uncaught_exception() = "<< std::uncaught
exception( )<< std::endl;
   }
private:
std::string m_msg;
};

void main()
{
exception ep("overflow.");
cout<<ep.what()<<" 1."<<endl;
try{
throw ep;
}
catch(exception& e)
{
cout<<e.what()<<" 2."<<endl;
```

```
}
int id=cin.get();
if (id==49)
{
unexpected_handler oldHand = set_unexpected( Tfunction );
unexpected();
}
else if(id==50)
{
terminate_handler oldHand_t = set_terminate(termfunction);
throw bad_alloc( );
//terminate();
}
//
else if(id==51)
{
Test t1( "outside try block" );
  try
  {
 Test t2( "inside try block" );
 throw 1;
  }
  catch (...)
  {
  }
}
}
```

程序的执行结果如下。

结果 1：

```
overflow. 1.
overflow. 2.
1
1. I'll be back.

abnormal program termination
Press any key to continue
```

结果 2：

```
overflow. 1.
overflow. 2.
2
2. I'll be back.
Press any key to continue
```

结果 3：

```
overflow. 1.
overflow. 2.
3
In Test::Test()
In Test::Test()
In Test::~Test()     std::uncaught_exception() = 0
In Test::~Test()     std::uncaught_exception() = 0
Press any key to continue
```

10.7 其他运行支持

本节主要讲述语言支持库中一些不常用的类、宏和函数。主要涉及 C 标准库的一些宏，程序跳转支持，时间处理支持，信号处理支持，环境支持等。这一部分内容比较生僻，但非常实用。

10.7.1 概述

头文件<cstdarg>(variable argument)、<csetjmp>(nonlocal jumps)、<ctime>(system clock clock(),time())、<csignal>(signal handling)和<cstdlib>(runtime environment getenv(), system())。

在表 10-3 中，给出的宏和函数可用于访问列表 list。

表 10-3　头文件<cstdarg>概要

Type	Name(s)
Macros:	va_arg，va_end，va_start
Type:	va_list

在表 10-4 中，给出的函数和宏主要用于在程序中实现跳转功能。

在表 10-5 中，给出的宏和函数主要用于处理时间（时钟和时刻）。

表 10-4　头文件<csetjmp>概要

Type	Name(s)
Macros:	setjmp
Type:	jmp_buf
Function:	longjmp

表 10-5　头文件<ctime>概要

Type	Name(s)
Macros:	CLOCKS_PER_SEC
Type:	clock_t
Function:	clock

在表 10-6 中，给出的宏和函数主要用于信号处理。

表 10-6　头文件<csignal>概要

Type	Name(s)
Macros:	SIGABRT ,SIGILL, SIGSEGV,SIG_DFL,SIG_IGN, SIGFPE,SIGINT,SIGTERM,SIG_ERR
Type:	sig_atomic_t
Function:	raise, signal

在表 10-7 中，给出两个环境处理函数。

表 10-7　头文件<cstdlib>概要

Type	Name(s)
Functions:	getenv , system

上述头文件和标准 C 库的头文件<stdarg.h>，<setjmp.h>，<time.h>，<signal.h>和<stdlib.h>基本上是一致的，仅包括以下几点差异：

（1）va_start、va_end、va_arg3 个函数主要应用于访问列表（list）的变量参数。函数 va_arg()返回当前的参数。函数 va_list 用于创建一个参数类型（type）指针 ap；va_start()用于初始化指针 ap，第二个变量必须是…之前的变量名；函数 va_arg(last, ap)第一次被调用时，会返回 last 之后的变长参数表中的第一个参数，并修改 ap 指向下一个元素，当再一次调用 va_arg()时，函数会返回一个参数，可以将 ap 向前移动一步。va_arg 除了需要 ap 参数指针之外，还需要数据类型 type，用于决定下一步的步长应该迈多大。函数 va_end()是和函数 va_start()相对应的。

ISO C 为宏函数 va_start()设置第二个参数，该宏函数在头文件<stdarg.h>中声明，严格地说这和国际标准是不同的。在函数定义的变量参数表中，参数 parmN 是最右边参数标志。如果参数 parmN 被声明，伴随着一个函数、数组或引用类型，或伴随一个类型（此类型并不适合），传递参数时，如果没有参数导致参数类型不合适， 此类行为是不确定的。

（2）函数 setjmp()和 longjmp()分别担任局部标号和 goto 的功能。在国际标准中，函数 longjmp(jmp_buf, int val)具有更受限制的行为。函数 longjmp 的作用是使进程返回至 setjmp 处执行，其第二个参数表示下一次调用 setjmp()时的返回值。有时通过抛出异常，转移控制到程序中的另一个目标点，如果任何自动类型的对象会被破坏，然后调用函数 longjmp(jbuf, val)在抛出地点，在该抛出点会转移控制到同一个点具有不确定的行为。

（3）函数 clock()在前面的章节已经有所介绍，此处仅在例题中举例说明。

（4）信号处理函数 raise()和 signal()可以在 Windows 环境中使用。函数 raise()用于抛出一个信号，signal()用于设置中断信号的处理函数（句柄函数）。

（5）函数 getenv()用于获取当前环境的值；函数 system()用于执行系统命令。

C 和 C++语言的通用子集一般包括：所有的声明、定义和表达式。这些可能在 C++程序中很好地出现，同样也遵从 C 程序。任一 POF（plain old function）是一个函数。该函数应用普通的子集的仅有特性，并不直接或间接地使用任意函数，该函数不是一个 POF。所有的信号句柄应该具有 C 链接。任一 POF 不会产生不确定的行为，该 POF 在 C 程序中可以作为信号句柄使用。在 C++程序中，作为信号句柄的任一其他函数的行为是可以确定性地被执行的。

10.7.2 应用举例

本节使用例 10-6 阐述 10.7 节内容的使用方法。

例 10-6 首先展示了如何处理变长参数，也就是 cstdarg 库中的 va 宏的用法。接下来使用 setjump 和 longjump 实现了长跳转（相对函数内部的 goto 跳转而言）。最后展示了如何通过 getenv 函数来获取系统环境变量。

例 10-6

```
#include <iostream>
#include <cstdarg>
#include <csetjmp>
#include <ctime>
#include <csignal>
#include <cstdlib>
#include <string>
using namespace std;
//va_list   char*
const int N=5;
void va_test(int i1,...)
    {
        va_list ptr;
        va_start(ptr,i1);
        cout<<i1<<" ;";
        int n_i=N;
        //逐一取出元素
        while(n_i--)
```

```
        {
        cout<<va_arg(ptr,int)<<" ;";      //依次取出一个整数
        }
        cout<<endl;
        va_end(ptr);
    }
void jmp_test()
    {
        int num=-1;
        jmp_buf  env;
        int state=setjmp(env);              //只有第一次调用 setjmp 时,才返回
        if(state==0)
        {
        cout<<"输入数字:";
        cin>>num;
        if(num!=0)
        {
        longjmp(env,1);                     //下一次调用 setjmp 时, setjmp 的返回值为
        }
        }
        else
        {
        cout<<"Error. "<<endl;
        }
    }
void mysleep(int sec_wait)
    {
        clock_t goal;
        long int wait=sec_wait*CLOCKS_PER_SEC;
        goal=wait+clock();
        while(goal>clock())
        {
        };
        cout<<"Testing wait "<<sec_wait<<" second. "<<endl;
    }
void time_test()
        {
        clock_t ct;
        ct=clock();
        }
void S_Handler(int signal)
        {
        printf("Signal dealing...\n");
    }
void test_Signal()
{
    typedef void (*SignalHandlerPointer)(int);
    SignalHandlerPointer previousHandler;
    previousHandler = signal(SIGABRT, S_Handler);
    raise(SIGABRT);
    }
void test_getenv()
```

```
    {
    cout<<"Lib information: "<<endl;
    char *libvar;
    libvar = getenv( "LIB" ); // C4996
    if( libvar != NULL )
       //printf( "Original LIB variable is: %s\n", libvar );
    cout<<libvar<<endl;
    char* oldlib;
    _putenv("LIB=c:\\;d:\\");
    oldlib=getenv("LIB");
    cout<<oldlib<<endl;
    string s1="LIB=";
    string s2=string(libvar);
    string s3=s1+s2;
    ////s1=s1+s2;
    _putenv(s3.data());
    oldlib=getenv("LIB");
    cout<<oldlib<<endl;
    }
void test_system()
{
    cout<<endl;
    cout<<"system dir e:\\ "<<endl;
    system("dir e:\\");
    }
void main()
    {
    va_test(12,34,56,31,76,98);
    jmp_test();
    mysleep(5);
    try{
    test_Signal();
    }
    catch(...)
    {
    cout<<"signal test."<<endl;
    }
    test_getenv();
    test_system();
}
```

程序的执行结果如下：

```
12 ;34 ;56 ;31 ;76 ;98 ;
输入数字:1
Error.
Testing wait 5 second.
Signal dealing...
Lib information:
C:\Program Files\Microsoft Visual Studio\VC98\mfc\lib;C:\Program Files\Microsoft
 Visual Studio\VC98\lib
c:\;d:\
C:\Program Files\Microsoft Visual Studio\VC98\mfc\lib;C:\Program Files\Microsoft
```

```
 Visual Studio\VC98\lib

system dir e:\
驱动器 E 中的卷是 资料备份
卷的序列号是 F019-A881

 e:\ 的目录

2012-08-09  11:29    <DIR>          Downloads
2011-07-06  14:08         2,514,432 ICBCSetupIntegration.msi
2011-11-29  16:11    <DIR>          一些资料
2012-04-11  10:40    <DIR>          体育锻炼
2012-08-23  10:48    <DIR>          办公文档
2012-07-23  21:05               105 回龙观上网信息.txt
2012-07-23  17:48    <DIR>          宽带我世界
2012-09-07  18:34    <DIR>          效能日记本 1-31
2012-04-26  21:41    <DIR>          测试卡数据
2012-06-19  12:27    <DIR>          游戏
2012-05-30  08:35    <DIR>          计算备份 2012-5-30
2012-09-03  14:29    <DIR>          软件备份
2012-05-11  14:03    <DIR>          道教典籍
2012-06-14  14:48    <DIR>          邮箱
2009-04-24  18:15    <DIR>          雍正王朝
               2 个文件      2,514,537 字节
              13 个目录     65,328,611,328 可用字节
请按任意键继续...
```

10.8 本章小结

本章主要讲述和语言支持相关的内容。主要包括：类型，执行属性，程序的启动和终止，动态内存管理，类型标识符，异常处理和其他运行支持。对上述的每个内容均采用最简单的例题进行举例说明。

提示：本章涉及的内容大多数并不常用，读者应该认真阅读本章的例题。对每个例题做到理解透彻，才能掌握本章讲述的内容。

第 11 章 检测库详解

本章主要讲述 C++程序中非常重要的部分——检测和报告错误。本章的各节将详细地描述：如何报告几种异常条件、文档程序的断言、全局变量用于错误编码等。涉及的头文件主要有<stdexcept>、<cassert>和<cerrno>。

11.1 异常类 Exception

在前面的章节已经多次讲述过异常类，本节将继续讲述异常类中和检测相关的内容。标准 C++库提供诸多的类，用于在 C++程序中报告某些确定的错误。这些错误的模型反映（折射）了这些类，错误被分成两个大类：逻辑错误和运行错误。逻辑错误的特性是：这些错误的原因是程序的内部逻辑错误。从理论上来讲，这些错误是可以预防的。相反，运行错误的原因通常是一些超出程序范围的事件。这些错误不容易提前预知。在头文件<stdexcept>中定义了几种类型的预定义的异常，用于报告 C++程序中的错误。这些异常通常和继承相关，前面已经讲过，头文件<stdexcept>的概要如下：

```
namespace std{
class logic_error;                          //逻辑错误
   class domain_error;                      //域错误
   class invalid_argument;                  //无效参数
   class length_error;                      //长度错误
   class out_of_range;                      //超出范围
   class runtime_error;                     //运行错误
   class range_error;                       //范围错误
   class overflow_error;                    //上溢出错误
   class underflow_error;                   //下溢出错误
}
```

11.1.1 类 logic_error

类 logic_error 的声明如下：

```
class logic_error : public exception {
   public:
      explicit logic_error(const string& message);
      virtual const char *what();
};
```

类 logic_error 定义了异常对象的类型，这些异常对象被抛出用于报告程序执行过程中发生的错误。例如：逻辑条件或类折射不变量偏离等。该类仅包括了一个构造函数和一个 what()函数。构造函数的参数是 what()函数的返回值。其关系为：what()的返回值应该等于 message.data()，或者等于 message.c_str()。下面用例 11-1 来加以说明。

例 11-1

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;
void main()
{
   try
   {
     throw logic_error( "logic error: 2012-9-11. " );        //抛出逻辑
   }
   catch ( exception &ep )
   {
     cerr << "Caught: " << ep.what( ) << endl;
     cerr << "Type name: " << typeid( ep ).name( ) << endl;
   };
}
```

程序的执行结果如下：

```
Caught: logic error: 2012-9-11.
Type name: class std::logic_error
```

11.1.2 类 domain_error

类 domain_error 的声明如下：

```
   namespace std{
   class domain_error: public logic_error{
      public:
          explicit domain_error(const string& what_arg );
}
}
```

类 domain_error 是类 logic_error 的派生类。因此，同时也继承了类 logic_error 的 what()函数。类 domain 的作用是定义了异常对象的类型，便于在程序执行时报告“域”错误。类的构造函数如下：

```
domain_error(const string& what_arg);
```

构造函数的主要功能是：构造一个 domain 类的对象。构造函数的参数 what_arg 的内容和函数 what()的返回值是一致的。下面用例 11-2 来加以说明。

例 11-2

```
#include <iostream>
//#include <stdexcept>
using namespace std;
void main()
{
   try
   {
     throw domain_error( "Your domain is in error!" );        //抛出逻辑
```

```
    }
    catch (domain_error &e)
    {
      cerr << "Caught: " << e.what( ) << endl;                  //输出错误名称
      cerr << "Type name: " << typeid(e).name( ) << endl;      //输出错误类
    };
}
```

上述代码在 Visual C++6.0 环境中执行时，程序在执行过程中被异常终止，并弹出异常对话框。在 Visual C++2008 环境中执行时，程序能够正常运行。程序的执行结果是：

```
Caught: Your domain is in error!
Type name: class std::domain_error
```

11.1.3　类 invalid_argument

同样类 invalid_argument 也是类 logic_error 的派生类。类的声明如下：

```
namespace std{
    class invalid_argument:public logic_error{
    public:
        explicit invalid_argument(const string& what_arg);
    };
}
```

类 invalid_argument 定义异常对象的一种类型，该异常对象被抛出时用于汇报一个无效参数错误。类的构造函数如下：

```
invalid_argument(const string& what_arg);
```

该构造函数的作用是构造一个类 invalid_argument 的对象。构造函数的参数 what_arg 的内容是函数 what()的返回值。下面用例 11-3 来加以说明。

例 11-3

```
#include <bitset>
#include <iostream>

using namespace std;

int main( )
{
   try
   {
     bitset< 32 > bitset( string( "11001010101100001b100101010110000") );
   }
   catch ( exception &e )
   {
     cerr << "Caught " << e.what( ) << endl;
     cerr << "Type name: " << typeid( e ).name( ) << endl;
   };
}
```

程序的执行结果如下：

```
Caught invalid bitset<N> char
Type name: class std::invalid_argument
```

上述例题之所以出现异常，主要是因为 bitset 结构主要使用 long int 作为 bit array，通常不会超

越 16 位。定义时即使超过了 16 位，访问位段时一旦超越了这个范围也会出错。第 3 章讲述位段时所给的例题，位段长短均为 16 位。这是读者需要注意的一个细节。

11.1.4 类 length_error

类 length_error 的声明如下：

```
class length_error : public logic_error {
public:
      explicit length_error(const string& message);
};
```

类 length_error 定义了异常对象的类型，在程序执行时，这些异常被抛出用于报告长度错误。所谓长度错误是指对象的长度超越了其最大允许的大小。类 length_error 的构造函数如下：

```
length_error(const string& what_arg)
```

该构造函数的作用是构造一个 length_error 类型的对象。构造函数的参数的内容是该对象成员函数 what()的返回值，即 strcmp(what(), what_arg.c_str())==0。下面用例 11-4 来加以说明。

例 11-4

```
#include <iostream>
#include <vector>
using namespace std;
void print(vector<int> v)
{
   vector<int>::iterator it;
   for(it=v.begin();it!=v.end();it++)
      cout<<" "<<*it<<" ;";
   cout<<endl;
}
void test_one()
{
   int a[]={1,2};
   vector<int> vi;
   vi.assign(a,a+2);
   try{
      vi.resize(vi.max_size()+1);
   }
   catch(length_error& le)
   {
      cout<<le.what()<<endl;
      cout<<typeid(le).name()<<endl;
   }
}
void test_two()
{
   try{
      length_error le("my custom error.");
      throw le;
   }
   catch(length_error& le)
   {
      cout<<le.what()<<endl;
      cout<<typeid(le).name()<<endl;
   }
```

```
}
void main()
{
    int ind=0;
    for(;;)
    {
        cout<<"Please Input number(1-3, 3:exit):"<<endl;
        cin>>ind;
        switch (ind)
        {
        case 1:
            test_one();
            break;
        case 2:
            test_two();
            break;
        case 3:
            exit(0);
        }
    }

}
```

程序的执行结果如下：

```
Please Input number(1-3, 3:exit):
1
vector<T> too long
class std::length_error
Please Input number(1-3, 3:exit):
2
my custom error.
class std::length_error
Please Input number(1-3, 3:exit):
3
请按任意键继续. . .
```

11.1.5 类 out_of_range

类 out_of_range 的声明如下：

```
class out_of_range : public logic_error {
public:
      explicit out_of_range(const string& message);
};
```

类 out_of_range 定义了异常对象的类型，该对象在程序执行过程中被抛出，用于报告参数值错误（该参数值不在期望的范围内）。

类的构造函数如下：

```
out_of_range(const string& what_arg);
```

该构造函数的作用是构造一个 out_of_range 类的对象。构造函数的参数是函数 what()的返回值。下面用例 11-5 来加以说明。

例 11-5

```
#include <iostream>
```

```
#include <vector>
#include <stdexcept>
using namespace std;
void print(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
        cout<<*it<<" ;";
    cout<<endl;
}
void main()
{
    vector<int> vi;
    int array[]={1,2,3,4,5};
    vi.assign(array,array+5);                    //初始化 vector 向量
    print(vi);                                   //输出向量
    try{
        int var=vi.at(-1);                       //获取某个元素
        int var2=vi.at(0);
    }
    catch(out_of_range& oe)                      //捕获异常
    {
        cout<<oe.what()<<endl;                   //输出异常的信息
        cout<<typeid(oe).name()<<endl;           //获取异常类的名称
    }
}
```

程序的执行结果如下：

```
1 ;2 ;3 ;4 ;5 ;
invalid vector<T> subscript
class std::out_of_range
```

11.1.6 类 runtime_error

类 runtime_error 的声明如下：

```
class runtime_error : public exception {
    public:
      explicit runtime_error(const string& message);
      virtual const char *what();
};
```

类 runtime_error 定义了异常对象的类型。这些异常对象在程序执行时被抛出，用于报告检测到的错误。类的构造函数如下：

```
runtime_error(const string& what_arg);
```

构造函数的功能是构造类 runtime_error 的对象。构造函数的参数 what_arg 和 what()函数的返回值是一致的。通常发生 runtime_error 类错误时，电脑会发生较明显的变化，例如明显的缓慢。当信息栏被关闭之后，程序一般会自动关闭或失去响应，甚至导致电脑重新启动。runtime_error 错误经常是由内存带来的；也有可能是由病毒或其他恶意软件带来的。

常见的运行错误代码如下：

```
1).Illegal function call Program error.
2).Overflow Program error.
3).Out of memory.
```

```
4).Subscript out of range Program error.
5).Duplicate definition Program error.
6).Division by zero Problem with a math formula in the program or the programs code.
7).Type Mismatch.
8) Out of string space Program error.
9).No Resume Program error.
10).Resume without error Program error.
11).Out of stack space
12).Sub or Function not defined Program error.
13).Error in loading DLL.
14) Bad file name or number Program error.
15).File not found File required by the program to run is not found.
16).Bad file mode Program error.
17).File already open Program or file associated with program is being used and program
does not have
 access to use it
18).File already exists Program error.
19).Disk full The disk, for example.
20).Input past end of file Program error.
… …
```

类 runtime_error 的使用详见例 11-6。

例 11-6

```
#include <iostream>
#include <string>
#include <stdexcept>
#include <cstdlib>
using namespace std;
int main()
{
    try{
        float in;
        runtime_error re("Input float error.");  //定义异常
        cin>>in;
        if(!cin)
            throw re;                             //抛出异常
    }
    catch (runtime_error& re)
    {
        cout << "Error type:" << re.what() << endl;
        cout<<"Class type:"<<typeid(re).name()<<endl;
    }
}
```

程序的执行结果如下：

```
1
Error type:Input float error.
Class type:class std::runtime_error
```

11.1.7　类 range_error

类 range_error 的声明如下：

```
namespace std{
class range_error: public runtime_error{
public:
```

```
        explicit range_error(const  string&  what_arg);
    };
}
```

类 range_error 定义了异常对象的类型。在程序执行时，尤其是内部计算时，这些异常对象被抛出用于报告范围类型的错误。

类 range_error 的构造函数如下：

```
range_error(const string& what_arg);
```

构造函数的功能：可以构造一个类 range_error 的对象。构造函数的参数是函数 what() 的返回值。下面用例 11-7 来加以说明。

例 11-7

```
#include <iostream>
#include <stdexcept>
using namespace std;
void main()
{
    range_error re("Error type: range_error.");
    try
    {
        locale loc( "test" );
    }
    catch(...)
    {
        cout<<re.what()<<endl;
        cout<<typeid(re).name()<<endl;
    }
}
```

程序的执行结果如下：

```
Error type: range_error.
class std::range_error
```

11.1.8 类 overflow_error

类 overflow_error 的声明如下：

```
namespace_std{
class overflow_error: public runtime_error{
public:
        explicit overflow_error(const  string& what_arg);
    };
}
```

类 overflow_error 定义了异常对象的类型。这些异常对象会在发生算术运算溢出错误时被抛出。类的构造函数如下：

```
overflow_error(const string& what_arg);
```

类 overflow_error 的构造函数的参数和 what()的返回值是一致的。下面用例 11-8 来加以说明，此例使用 Visual 2008 调试通过。

例 11-8（摘自 msdn2008 中类 overflow_error 例题）

```
#include <iostream>
#include <stdexcept>
```

```
#include <bitset>
using namespace std;
void main()
{
    try
    {
        bitset< 33 > bitset;                              //定义位段
        bitset[32] = 1;
        bitset[0] = 1;
        cout<<" "<<bitset<<endl;
        unsigned long x = bitset.to_ulong();              //位段转换成无符号长整型
        cout<<" "<<x<<" "<<endl;

    }
    catch ( exception &e )
    {
        cerr << "Caught : " << e.what() << endl;
        cerr << "Type : " << typeid(e).name( ) << endl;
    };
}
```

程序的执行结果如下：

```
100000000000000000000000000000001
Caught : bitset<N> overflow
Type : class std::overflow_error
```

11.1.9　类 underflow_error（下溢出）

类 underflow_error 的声明如下：

```
namespace std{
    class underflow_error : public runtime_error{
  public:
    explicit underflow_error(const string& what_arg);
    };
}
```

类 underflow_error 定义了异常对象的类型。程序执行过程中，一旦发生算术 underflow 错误，这些异常对象将被抛出。

类的构造函数如下：

```
underflow_error(const string&  what_arg);
```

构造函数的功能是构造一个 underflow_error 类的对象。构造函数的参数是函数 what()的返回值。下面用例 11-9 来加以说明。

例 11-9

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

void main()
  {
      try{
      underflow_error ue("under_flow_error.");
      throw ue;
```

```
        //int vb=vi.at(-1);
        // cout<<va<<endl;
        }
        catch(exception& e)
        {
            cout<<e.what()<<endl;
            cout<<typeid(e).name()<<endl;
    }
}
```

程序的执行结果如下：

```
under_flow_error.
class std::underflow_error
```

11.2 assertions（断言）

断言宏常用于 C++程序中。宏 assert 主要应用于调试程序，在程序运行时该宏会计算括号内的表达式，如果表达式为 false，程序将报告错误，并终止执行。若表达式不是 false，则继续执行该语句后面的语句。宏 assertion 会判断程序中是否出现明显非法数据，若出现了则终止程序以免导致严重后果，便于查找错误。使用函数 assert()必须包含头文件<assert.h>。函数 assert()的功能：评估其参数的值（非零和零），当参数 expression 为 false 时，程序会打印出检测到的信息，并终止（abort）程序的运行。

举例说明如下：

```
#include <assert.h>
void assert(int expression);
```

使用 assert()的缺点是会影响程序的性能，增加额外的开销。函数 assert()并不受宏 DEBUG 的限制。无论是程序的 debug 版本还是 release 版本，函数 assert()均起作用，函数 assert()能够及时地查出编码的错误，很多错误不是完全在编译时发生的，而是运行时发生的。不能仅仅依赖 assert()，更主要的是使用 exception 机制来检测运行时的错误，并相应地采取处理措施。

使用函数 assert()需要注意以下几个问题：

（1）使用断言捕捉不应发生的非法情况。不混淆非法情况与错误情况之间的区别，后者必然存在的并且是一定要进行处理；

（2）使用断言时，应对函数的参数进行确认；

（3）编写函数时，要反复考查，对设定的假定需要使用断言进行检查；

（4）通常程序员都熟悉部分防错性的程序设计，记住此风格会隐瞒错误。当进行防错性编程时，若不可能发生的事情却发生了，需要使用断言进行报警；

（5）每个断言需要详细、清楚；

（6）使用不同的算法对程序结果进行确认；

（7）在错误发生之前，即使用初始检查程序。

在 Visual Studio 的 msdn 中指出，宏函数 assert()典型地用于识别逻辑错误。在程序开发过程中，当程序运行不正确时，参数 expression 会转变为错误 false。当程序调试结束时，并不需要调整源文件，仅使用宏 NDEBUG，断言检测可以无效。NDEBUG 可以使用命令行选项/D 被定义，或使用语句#define。如果 NDEBUG 使用#define 被定义之后，#define 语句必须出现在头文件 assert.h 之前。当函数 assert()的参数 expression 为 false 时，函数 assert()会打印出检测信息并调用函数 abort()来终止

程序的执行。如果函数的参数 expression 为 true 时，不会采取任何措施。检测信息包括错误的表达式、源文件的名字、assert()函数所在语句的行号等。

在 Visual C++ 2005 版本中，检测信息是使用宽字符打印的。这样，函数 assert()可以按期望的情况工作，即使表达式是 unicode characters 的形式。

检测信息的目的仅仅在于应用程序的类型，该应用会调用进程。控制台程序总是接收来自于 stderr 的信息。在基于窗口的应用程序中，函数 assert()会调用窗口 MesageBox()，目的是创建一个信息框来显示伴随 OK 按钮的信息对话框。当用户单击 OK 按钮时，程序迅速调用 abort()，退出运行。

当应用程序链接 debug 版本的运行库时，宏断言函数 assert()会使用 3 个按钮创建一个消息框：Abort()、Retry()和 Ingore()。如果用户单击 Abort()，程序会迅速退出。如果用户单击 Retry()，调试器会被调用，并且如果 just-in-time 调试有效的话，用户可以调试程序。如果用户单击按钮 Ignore，断言函数 assert()继续其正常执行：使用 OK 按钮创建信息框。

提示：单击 Ignore 按钮时，如果错误条件存在，会导致不确定的行为。

未获取调试过程中的更多信息，可以查看“CRT（C 运行库（run time library））调试技术的文档”。

下面用例 11-10 来加以说明函数 assert()的使用方法。

例 11-10

```
#include <iostream>
#include <assert.h>
#include <string>
using namespace std;
void ana_string(char* str)
    {
        assert(str!=NULL);          //断言
        assert(str!="\0");
        assert(strlen(str)>2);
    }
void main()
    {
        char test1[]="abc";
        char *test2=NULL;
        char test3[]="";
        printf("string: %s. \n",test1);
        fflush(stdout);
        ana_string(test1);
        //
        printf("string: %s \n",test2);
        fflush(stdout);
        ana_string(test2);
        //
        printf("string: %s \n",test3);
        fflush(stdout);
        ana_string(test3);
        //
    }
```

例 11-10 的执行效果见图 11-1。

图 11-1　例 11-10 的执行效果

11.3　错误编码

在头文件<cerrno>中，包含了多个错误编码。下面一一介绍 C++ STL 中关于错误编码的内容。变量 errno 的值是常量，在不同的错误条件下或事件中，诸多的错误代码被赋值给变量 errno。在头文件 errno.h 中，包含了诸多变量 errno 值的定义。然而，在 32 位 Window 操作系统中，并不是所有的定义都在头文件 errno.h 中给出。头文件<errno.h>中的值同时也保持了同 UNIX 家族操作系统的兼容性。

在 32 位 Windows 操作系统中，变量 errno 的值是 XENIX 系统中 errno 值的子集。这样，errno 的值并不是必须和实际的错误代码保持一致。错误代码是通过系统调用从 Windows 操作系统获取的。为实现访问实际的操作系统错误代码，使用 doserrno 变量，该变量中包含了这些值。表 11-1 中的错误代码值是通常被支持的。

表 11-1　错误代码值及其说明

错误代码值	说　　明
ECHILD	没有大量的处理过程
EAGAIN	没有进一步的处理。尝试创建一个新的过程，如果尝试失败主要是因为没有进一步的进程缝隙，或没有足够的内存，或达到了最大嵌套级别
E2BIG	参数列表太长
EACCES	不被认可。文件设置不被允许特殊的访问。错误意味着任何方式访问文件的尝试是不和文件的属性矛盾的
EBADF	坏的文件代码。主要有两个可能的原因：特定的文件描述符是无效值或没有指向已打开的文件
EDEADLOCK	资源的死锁会发生。对于一个数学函数的参数，是指没有在合适的范围内
EDOM	数学参数
EEXIST	文件存在。当尝试创建一个文件时，该文件已经存在。例如：当实现 open 调用时，_O_CREAT 和_O_EXCL 标志是特定的，但命名的文件已经存在
EILSEQ	字节的非法序列。例如：在一个 MBCS 字符串中
EINVAL	无效参数。函数的参数之一是无效值。例如：当定位文件时，最初的给定值处于文件头之前
EMFILE	太多被打开的文件。不能创建更多的文件描述符，也不能再打开更多的文件
ENOENT	没有该文件和路径。指定的文件或路径不存在或不能被发现。当指定的文件不存在，或文件路径不能说明一个已存在的路径时，该信息会发生

续表

错误代码值	说　　明
ENOEXEC	可执行文件格式错误。尝试执行一个文件时，该文件是不可执行的，或具有无效可执行文件格式
ENOMEM	没有足够的核。当尝试操作符时，没有足够的内存。例如：当无足够的内存用于执行子进程时，此信息会发生；或要求调用 getcwd 却不能得到满足时
ENOSPC	设备没有剩余空间。设备没有足够的空间用于写操作。例如：磁盘已满
ERANGE	结果值太大。数学函数的参数值太大，导致部分或全部失去意义。当函数的参数远大于其期望值时，此错误也可以发生在其他类型的函数中
EXDEV	交叉设备链接。当尝试移动文件到其他存储设备时会发生此类错误提示
STRUNCATE	字符串复制或其他相关事物导致一个截取的字符串

11.4　本章小结

本章主要讲述 3 部分内容。总体来说，内容较少。第 1 节详细介绍了各种异常类 Exception，也给出了相应的例题；第 2 节讲述了 assertions（断言）；第 3 节讲述了基于 Windows 操作系统的错误编码。

第 12 章 国际化库详解

随着 C++语言在全球的推广和普及，国际化越来越成为软件开发的重要议题。为适应国际化的潮流，C++标准库提供了编写国际化程序代码支持机制。支持机制主要影响 I/O 的使用和字符串的处理。本章将介绍此机制。Dietmar Kuhi 是 C++标准库中有关 I/O 和国际化方面的专家，国际化的内容大部分由其负责。

12.1 国际化问题和国际化元素

C++标准程序库面对“区域”的转换问题，除着眼于某些特定转换外，还提供了一个普通方案。例如：为支持 16 位亚洲字符，该普通方案不限定字符型别。对于程序国际化而言，以下两个方面非常重要：

（1）不同的字符集具有不同的性质。面对字符数多于 256 个的字符集，char 型别显然不够用；

（2）使用者希望程序能够针对国家和文化传统间的差异，进行必要的转化（例如：日期、格式、货币书写格式、布尔值表示法等）。

对于上述两个问题，C++标准程序库均提供了相关解决方案。

其实，所谓国际化的问题最根本莫过于各种问题的本地化。顾名思义，区域（locale）表示的通常是在全世界不同区域具有不同的信息表达方式，对于不同的文化背景的计算机用户需要使用不同的信息显示方式。例如：各个国家的日期显示格式是不同的。即使在使用同一种国际语言的国家之间，信息的表达方式仍然存在较大差异。程序开发者如果试图在全球范围推广其软件产品，必须意识到不同区域的差异。

国际化问题的主要思路是以 locale 对象代表可扩充的面貌集合，以此进行地区转换工作。locale 在 C 语言中已经有所应用，在 C++标准中国际化问题被更加泛型化，设计得更有弹性。事实上，C++的 locale 机制可以根据用户环境或偏好进行各种专用定制。例如，可以对它进行扩展，使其处理度量衡系统、时区、纸张规格等问题。

大部分国际化相关机制仅需程序员付出最低程度的努力，根本不需要任何额外付出。程序员使用 locale 对象直接进行格式化、校勘、字符分类等工作。C++标准程序库所支持的某些方面并没有为其自身所用，程序员必须手工调用相应的函数才能运用。C++标准库中没有任何 stream 函数可以用来格式化时间、日期或货币表示式。最突出的代表就是 strings 和 streams 使用了国际化机制的另

一个概念：字符特性。二者均定义出能够把不同字符集区分开来的某些基本特性和操作。国际化机制的相关类是最近几年才被引入 C++标准中的。尽管整体方案极其灵活，但想真正达到完善还需要做一些工作。例如：字符串校勘的函数直接使用型别 const chart*的迭代器。charT 代表某种字符型别。尽管 basic_string<charT>通常会使用此型别作为迭代器型别，但并未进行任何的明文保证。

对于国际化元素，最简单的常识：当应用程序推广至国际市场时，最重要的问题是字符串输出。通常包括 7 种信息的输出。

（1）字符：使用简单 ASCII 码表示一种语言。ASCII 码字符集仅能表示最多 256 个字符，但其他某些语言需要更大的字符集。其他语言的表示问题导致 3 种类型字符编码的产生：单字节 ASCII 字符、多字节字符和宽字符。

（2）字符排序：对字符进行排序的过程称为排序。不同语言应用不同的排序规则。例如：在西班牙文中双符号字母“11”被看成单个字符；而在英语中“11”被看成两个“1”字符。因此西班牙文的排序方法与英语不能相同。

（3）字符分类：每种语言均含有字母、数字、标点和其他类型符号的字符编码。字符所属的组就是该字符的分类。

（4）数字：不同国家表示数字的方法各不相同。例如：美国风格的数字 34 785 000.75，如果按德国风格应写为 34.785.000,75。

（5）货币：不同的国家使用不同类型的货币，货币符号也不相同。美国货币符号是美元符号“$”，德国的货币符号是“DM”。另外，货币符号在货币值中的位置也不相同。美元符号放在数量值前，例如＄45.98；而 DM 符号放在数量值后，例如 12.45 DM。

（6）时间和日期：不同国家显示时间和日期的方式不同。

（7）大小写：不同语言在文本大小和转换规则上有所不同。

如果英语、西班牙语、法语或其他任何一种语言是地球上唯一的语言，编写国际化的应用程序就会很容易。但实际上语言的种类成千上万。在国际化程序中，文本应用一种适当的语言显示。通常需要把文本放在资源文件中，与执行文件分离，可以根据需要翻译文本。

C++的 STL 库提供管理国家化元素的基本方法。对于每种区域的本地语言，在大多数情况下必须手工翻译。

12.2　多种字符编码

国际化或本地化所关注的主要问题是如何处理不同的字符编码。尤其是亚洲地区，同一字符集存在不同的编码。例如：汉字存在多种编码方式（BIG5 码和 GB 码等）。甚至存在单一字符编码超过 8 个位的情况。面对上述这些问题，需要使用新的思想和新的函数进行文本处理。

12.2.1　宽字符和多字节文本

面对多于 256 个字符的字符集，有两种不同的解决方案：多字节表示法和宽字符表示法。

（1）在多字节表示法中，字符所用的字节个数是可变的。一个字节字符后面可以跟随 3 个字节，例如日本的表意文字或中国的象形文字。

（2）宽字符表示法，字符所用的字节数目恒定，与所表示的字符无关。典型的个数是 2 或 4 个字节。其实，这和那些只使用 1 个字节的表示法没有区别。

多字节表示法比宽字节表示法更紧凑，前者通常用于在程序外部储存数据，后者由于比较容易处理固定大小的字符，通常在程序内部被使用。

与 ISO C 相似，ISO C++使用 wchar_t 表示宽字符。不过，在 C++中 wchar_t 被扶正为关键词，不是区区一个型别定义。可以使用该型别对所有函数进行重载。

在多字节字符串中，当某单个字节代表一个字符时，它也有可能是字符的一部分。例如：对于 GB 汉字的表示，1 个字符仅表示半个汉字字符。尤其在遍历多字节字符串时，每个字节的意义均由当时的“转换”状态决定。根据转换状态，一个字节表示一个字符。通常多字节字符串会具有某些预定的初始转换状态。例如：在初始转换状态下，可能每个字节均代表 ISO 拉丁字符，直到遭遇“Escape”字符为止。紧跟在“Escape”字符之后的字符会用于指示新的转换状态：意味着随后的字节会被理解为阿拉伯字符，直到遇到下一个“Escape”字符为止。

类模板 codecvt<>用于在多种字符编码方案之间转换。该类主要为模板类 basic_filebuf<>提供服务。在数据内部表述和外部表述时，实现转换。C++标准没有对多字节字符编码进行任何的假定，仅仅支持“转换状态”的概念。类模板 codecvt<>的成员可以运用一个参数来储存字符串的任意状态。模板 codecvt<>还支持一个函数，允许以某个字符序列将转换状态还原为初始状态。

12.2.2 字符特性

对于多种字符集，多种表述方式给 string 和 I/O 的处理带来了多样性。字符串 string 和 stream 类可以根据内建型别完成对象的初始化。内建型别的接口不能够改变，需要引进一个独立的类别，即特性类别（trait class）。“如何处理和表述相关的各个面貌”的相关细节统统放入该类别中，是特性类别的概念表述。string 和 stream classes 都将 traits class 作为 template 参数；此参数默认为 char_traits，并以 string 或 stream 获取模板的参数作为参数。

```
namespace std{
     template <class chart , class traits = char_traits<charT>,
     class Allocator=allocator<charT>>
     class basic_string;
  }
namespace std{
     template <class charT, class traits = char_traits<charT> >
        class basic_istream;                    //模板类 basic_istream
     template<class chart, class traits= char_traits<charT>>
        class basic_ostream;                    //模板类 basic_ostream
     …
  }
```

字符的特性（character traits）的型别为 char_traits<>。该型别在<string>中定义，以字符型别为参数：

```
namespace std{
     emplate <class charT>
     struct char_traits{
     …
     }
}
```

特性类定义了字符的型别的所有基本特性，并以静态组件的形式定义出实作 strings 和 stream 时不可或缺的相关操作。

字符串和字符序列的处理函数，其存在价值完全是为了优化效率。其实完全可以运用“只处理单个字符”的函数实作出来。例如：copy()可运用 assign()实作出来，而在处理字符串时，copy()可能会采用更有效率的实作技术。

注意： 函数中使用的计数是确切的计数值，不是最大计数值。即所说序列的终止字符将被忽略。

最后一组函数专门对 end-of-file 字符进行特殊处理。该字符通过人工字符扩充字符集，用以指示某种特殊的处理。对某些表述而言，字符型别不足以容纳该特殊字符，因为 EOF 字符必须使用字符集中一般字符之外的值。C 语言建立一种转换模式：令字符函数返回一个 int，而不是一个 char。该技巧被 C++语言继承并得到发展。字符特性型别定义了 char_type 型别用于代表所有字符型别，定义 int_type 用以代表“所有的字符，外加 EOF”的型别。函数 to_char_type()、to_int_type()、not_eof()、eq_int_type()定义了相应的转换和比较操作。对于某些特殊字符，char_type 和 int_type 可能是相同的。char_type 中并非所有的数值都必须拿来表示字符，一定会有多余的值拿来代表 EOF。

pos_type 和 off_type 用于定义文件位置和偏移距离。类模板 char_trait 的成员见表 12-1。

表 12-1　字符特性类 char_trait

类　成　员	意　　　　义
char_type	字符型别（即 char_traits 的模板（template）参数）
int_type	足以容纳“附加 end-of-file 值”的型别
pos_type	此型别用以表现“stream 内的位置”
off_type	此型别用以表现“stream 内的两个位置之间的间距”
state_type	此型别用以表现“multibytes stream”当前的状态
assign(c1,c2)	将字符 c2 赋值给 c1
eq(c1, c2)	判断字符 c1 和 c2 是否相等
lt(c1,c2)	判断字符 c1 是否小于字符 c2
length(s)	返回字符串 s 的长度
compare(s1,s2,n)	比较字符串 s1 和 s2 的前 n 个字符
copy(s1,s2,n)	将字符串 s2 的前 n 个字符复制到 s1
move(s1,s2,n)	将字符串 s2 的前 n 个字符复制到 s1，s1 和 s2 可以重叠
assign(s,n,c)	将字符 c 赋值给字符串 s 的前 n 个字符
find(s,n,c)	在字符串 s 中搜寻第一个与 c 相等的字符，返回一个指针指向它。如果 s 的前 n 个字符中没有找到字符 c，函数返回 0
eof()	返回 end-of-file 值
to_int_type(c)	将字符 c 转换成“型别为 int_type”的相应值
to_char_type(i)	将型别为 int_type 的 i 转换为字符（如果转换 EOF，会导致未定义行为）
not_eof(i)	如果 i 不是 EOF，返回 i；如果 i 是 EOF，返回一个由实作版本所定义的、不同于 EOF 的值
eq_int_type(i1,i2)	检查字符 i1 和 i2 的 int_type 对应值是否相等（字符可以是 EOF）

C++标准程序将 char_traits<>型别针对 char 和 wchar_t 进行了特化设计：

```
namespace std{
    template<> struct char_traits<char>;
```

```
template<>struct char_traits<wchar_t>;
}
```

针对 char 而设计的特化版本，通常在实作过程中会运用定义于<cstring> 或<string.h>内的 C 全局函数。具体处理方法我们会在接下来的 12.2.3 小节细细详述。

12.2.3 特殊字符国际化

处理特殊字符也是一个比较难的问题。例如：换行符号或字符串终止符号的国际化。class basic_ ios 的成员函数 widen()、narrow()可用于解决这个问题：例如：对于流类型 stream，可以使用换行符编码如下：

```
stream strm;
strm.widen('\n');        //转换成宽字符
```

同样，字符终止符号可以编码如下：

```
strm.widen('\0');
```

函数 widen()和 narrow()在应用时使用了一个 locale 对象。准确地说是该对象的 ctype facet。该 facet 用于将所有字符“在 char 和其他表现形式之间”进行转换。locale 类型对象可以将 char 型别的字符 c 转换为一个 char_type 型别的对象。例如：

```
std::use_facet<std::ctype<char_type>>loc.widen(c);
Locales 及其 facets 的使用细节之后会讲述。
```

12.3 类 locale

在 C++中，std::locale 类的对象代表一个区域的表示。locale 对象是一种容器，容器中包含若干称为“刻面”的对象，刻面代表前面讲述的国际化元素。类 locale 对象含有若干个处理国际化元素的刻面，例如：日期时间格式、货币符号和数字格式等。刻面不仅提供区域表示的国际化元素信息，还为国际服务提供接口。

12.3.1 类 locale 概述

类 locale 实现一种类型安全的、多态的“刻面”集合，可以使用刻面类型检索。换一种说法，一种刻面具有双重角色：某种意义上，刻面正好是类的接口，同时也是 locale 刻面集合的索引。访问 locale 类型刻面可以通过两个函数模板：use_facet<>和 has_facet<>。

```
template <class chart , class traits> basic_ostream<chart, traits>&
        operator <<(basic_ostream<chart, traits>& s, Date d)
{
    typename basic_ostream<chart , traits>::sentry cerberos(s);
    if(cerberos)
    {
    ios_base::iostate  err=0;
    tm  tmbuf;                                              //定义 tm 的对象
    d.extract(tmbuf);
    use_facet<time_put<charT , ostreambuf_iterator<charT, traits>>>(
    s.getloc()).put(s,s,s.fill(), err , &tmbuf, 'x');          //获取本地时间,
    s.setstate(err);
```

```
    }
    return s;
}
```

在调用 use_facet<Facet>(loc)时，类型参数选择一种刻面，要使所有的名称类型的成员均有效。在 locale 中，Facet（刻面）不存在，该刻面会抛出标准异常 bad_cast。一个 C++程序可以检查出是否类 locale 使用函数模板 has_facet<Facet>实现一个特殊的刻面。类 locale 对象中，用户定义的刻面可能被装备并使用，就像许多标准刻面一样。

一种成员操作符模板 operator()(basic_string<C，T，A>&, basic_string<C，T，A>&) 被提供，以致于一种类 locale 对象可以用于作为一个可预测参数，对于标准集合，对于校核字符串。常规的全局接口界面被提供用于传统的 ctype 函数，类似 isdigit()和 isspace()，以至于给定一个 locale 对象，一个 C++程序可以调用 issapce(c, loc)。这容易升级现存的抽出符。

一旦一种刻面引用从一个 locale 对象获取，通过调用 use_facet<>，引用保留可用性。一些 locale 对象提到刻面，并且来自于成员函数的结果会被期望，并重复使用。

在连续地调用一个 locale 类的 facet 成员函数时，在一次调用中一个输入/输出流的插入器或抽出器或流缓冲区成员函数，返回的结果应该是一致的。类 locale 对象构建时，来自于一个名称字符串（“POSIX”），或来自于两个被命名的 locale 对象，拥有一个名称，所有其他的是不可能的。类 locale 命名可能用于“相等”比较，无命名的 locale 仅能等于它本身（即其备份）。对于一个无命名的 locale 对象，成员函数 name()返回类型为字符串 string。

对于真正的国际化，仅仅翻译“文字所带信息”通常是不够的。各种不同的数值、货币、日期等的规格都必须遵守。除此之外，用于操作字母的函数，应根据字符进行编码，确保正确处理特定语言中所有字母的字符。根据 POSIX 和 X/Open 标准，C 程序可以使用函数设定一个 locale。改变 locale 会对 isupper()和 toupper()之类的字符分类/操作函数以及 printf()之类的 I/O 函数产生影响。

说明：然而 C 的解决方案毕竟包括诸多限制。由于 locale 是全局属性，同时使用一个以上的 locale，即使不是不可能，也得花费很大的精力。locale 不能扩展，仅能提供“由编译器选择供应”的设施。若某个必须遵守的国家协议未被 C locale 支持，则它只有改弦更张。程序员不可能为支持特殊文化而定义新的 locale。

C++标准程序库利用面向对象方式解决了上述问题。首先，locale 相关的细节被封装在型别为 locale 的对象中。这样在同一时刻运用多个 locale 即可解决诸多问题。与 locale 相依的各种操作，将运用相应的 locale 对象。每一个 locale 对象安装到每个流中，流的各成员函数会利用该对象迎合相应规格。

经典的 C locale 赋值到标准输入通道中。所谓经典的 C locale，其实是没有任何 locale 的，使用最初的 C 形式来格式化数字、日期、字符分类等工作。

std::locale::classic() 获取经典的 locale 的对应对象。

表达式 std::locale(“C”)和 std::locale::classic()的效果相同。

该表达式根据给定的名字产生一个 locale 对象。“C”是特殊名称，实际上是各个 C++实际操作版本唯一必须支持的名称。C++标准并未强制要求支持其他的 locale，通常各个 C++实作版本都会支持其他 locales。表达式 cout.imbue(locale(“de_DE”))可以实现将 locale de_DE 赋值到标准输出通道中。唯有当系统支持该 locale，该动作才会成功。若所希望构造的 locale 名字不能被 C++实作版本识别出来，将会抛出一个 runtime_error 异常。

若一切顺利的话，输入时将按照经典 C 规格，输出时会按照德国规格。这样即可以按英文格式读取浮点数（德文是使用逗号作为小数点的）。

通常，除非需要按照某个固定格式来读写数据，否则程序不会预先定义一个特定的 locale 类，而利用环境变量 LANG 确定相应的 locale。另一种可能是读取一个 locale 名称时，需要运用它。

下面的例子展示了如何使用 locale 库，代码首先声明了一个 name 为“german”的 locale（loc1），然后调用 locale::global()来设置当前的 locale 为 loc1，并返回原来的 locale 设置（loc2），打印 loc1 和 loc2 的 name，即为新的 locale 和旧的 locale 的 name。最后将环境恢复为默认的 locale，并打印之前的 locale 的 name。

例 12-1

```
#include <iostream>
#include <locale>
#include <string>
using namespace std;
int main()
{
   locale loc1( "german" );          //本例题是讲述使用 locale 的
   locale loc2 = locale::global( loc1 );
   cout << loc1.name( ) << "." << endl;
   cout << loc2.name( ) << "." << endl;
   locale loc3=locale::global(std::locale(""));
   cout<<loc3.name()<<endl;
}
```

程序的执行结果如下：

```
German_Germany.1252.
C.
German_Germany.1252
```

如果程序遵循各地的习惯，应该使用相应的 locale 对象。类 locale 的静态成员 global()可以安装一个全局的 locale 对象。该对象可以用来作为某函数的 locale 对象参数的默认参数。如果 global()函数所设定的 locale 对象是有名称的，则 C 的 locale 相关函数也会做出相应的响应；若该对象没有名称，则 C 函数的执行结果由实作版本决定。

对于后继被执行的 C 函数会做出相应的注册操作，即那些 C 函数所受到的影响，以下调用操作所带来的影响相同。

```
std::setlocale(LC_ALL , " ");
```

通过设定全局 locale，不能替换已储存于对象内的 locale，只能改变由默认构造函数所产生的 locale 对象。stream 对象存储的 locale 对象不会被 locale::global()替换。若希望已经存在的 stream 使用某个特定的 locale，必须使用 imbue()函数通知该流（stream）。

当默认构造一个 locale 对象时，全局 locale 即可发挥作用。产生的新的 locale 即是全局 locale 的副本。下面 3 行代码是 3 个标准的 streams 安装默认的 locale 类对象的实例。

```
std::cin.imbue(std::locale());                //转换输入通道
std::cout.imbue(std::locale());
std::cerr.imbue(std::locale());
```

在 C++ STL 中使用 locale，牢记 C++ locale 机制和 C locale 机制之间仅有松散的耦合关系。一

个具名的 C++ locale 对象被设为全局 locale，则全局 C locale 会很被动。通常，不能假设 C 和 C++ 函数所操作的 locale 保持一致。

12.3.2 类 locale 的 facet

国内约定俗成的具体项目被划分为数个不同的面貌，分别由相应的对象处理。处理“国际化议题中的某一特定面貌”的对象，即称为 facet。locale 对象是扮演 facet 的容器。若存取 locale 的某个面貌，可以用相应的 facet 型别作为索引。如果 facet 作为模板参数，明白传给模板的函数 use_facet()，可以取用特定的 facet。

```
std::use_facet<std::numpunct<char>>(loc);
```

获取 locale loc 所管辖的“字符型别为 char”的 facet numpunct。每个 facet 均由“定义特定服务”的类别定义。若 facet numpunct 为格式化数值和布尔值提供服务，下述表达式返回用以表达 true 的字符串。

```
std::use_facet<std::numpunct<char>>(loc).truename();
```

表 12-2 描述了 C++标准程序库中预先定义的 facet。每个 facet 都和类别相关联。这些类别在 locale 的某些构造函数中用来组合其他的 locale，并产生新的 locale。

表 12-2 C++标准程序库预先定义的 facet 型别

分　类	facet 型别	用　　于
numeric	num_get<>()	数值输入
	num_put<>()	数值输出
	numpunct<>()	数值 I/O 中用到的符号
time	time_get<>()	时间和日期的输入
	time_put<>()	时间和日期的输出
monetary	money_get<>()	货币输入
	money_put<>()	货币输出
	moneypunct<>()	货币 I/O 中用到的符号
ctype	ctype<>()	字符信息（toupper(), isupper()）
	codecvt<>()	在不同字符编码之间进行转换
collate	collate<>()	字符串校勘（collation）
messages	messages<>()	获取字符信息

1. 标准刻面

程序员可以创建刻面，C++定义了 7 种标准刻面。这 7 种刻面包括：

（1）代码转换：此种刻面可以处理不同字符表示间的转换。例如：从多字节字符转换成宽字符。

（2）排序：此类刻面处理字符排序，用于字符串排序。

（3）分类：这类刻面处理字符的分类。例如设置字符的大小写，或判断某个字符是否属于语言的标点符号组。

（4）数字：此类刻面处理数字的格式化。

（5）货币：此类刻面处理货币的格式化。

（6）时间和日期：此类刻面处理时间和日期的格式化。

（7）消息：此类刻面处理多种消息类别，使应用程序能够翻译简单的响应，例如："是"和"否"。

2. 默认和全局表示

在前文中多次使用过区域表示。每个 C++程序均在默认的区域表示下运行，即美国的 ASCII 码表示。

C++库创建了该区域的表示对象，即 std::locale::classic。std::locale::classic 会像其他区域表示一样，决定库函数的显示信息方式。

全局区域表示是当前有效的区域表示。若没有在程序中修改区域表示，则默认和全局区域表示是一样的。全局区域表示可以修改，但默认区域表示不能修改。

3. 细述 locale 类

C++locale 是不变的刻面（facet）容器，定义于<locale>头文件内。

```
namespace std{
class locale{
public:
  static const locale& classic();
  static locale global(const locale&);
  class facet;
  class id;
  typedef int category;
  static const category none, numeric, time, monetary, ctype, collate, message, all;
  locale() throw();
  explicit locale(const char* name);
  locale (const locale& loc) throw();
  locale(const locale& loc, const char* name, category);
  template<class Facet> locale(const locale& loc, Facet* fp);
  locale(const locale& loc, const locale& loc2, category);
  const locale& operator=(const locale& loc) throw();
  template <class Facet> locale combine(const locale& loc );
  ~locale() throw();
  basic_string<char> name() const;
  bool operator==(const locale& loc) const;
  bool operator!=(const locale& loc) const;
  template <class chart, class Traits, class Allocator> bool operator(
  {
    const basic_string<charT, traits, Allocator >& s1;
    const basic_string<charT, Traits, Allocator>& s2) const;
 }
  template <calss Facet> const Facet& use_facet(const locale&);
  template<class Facet> bool has_facet(const locale&) throw();
  }
}
```

对于类 locale，其存取方式是比较特殊的。locale 内含 facet，是以 facet 型别为索引进行存取的。由于每个 facet 均有不同的界面，用于不同的目的，所以期望 locale 的存取函数返回对应索引的型别。以 facet 型别作为索引的好处是可以拥有型别安全的接口。

locale 是不可变的，即存储于 locale 之内的 facet 不能被改变。现有的 locale 和 facet 组合起来，产生新的 locale 变体。locale 类的构造函数存在多种形式，详见表 12-3。

表 12-3　locale 类的构造

表达式	功能
locale()	产生一个当前全局的 locale 副本
locale(name)	根据名称产生出一个 locale
locale(loc)	产生 locale loc 的副本
locale(loc1,loc2,cat)	产生 locale loc1 的副本，类别 cat 中所有的 facet 将被 locale loc2 的 facet 替换
locale(loc,name,cat)	此动作等同于 locale(loc, locale(name),cat)
locale(loc,fp)	产生 locale loc 的副本，并安装 fp 所指的 facet
loc1=loc2	将 loc2 赋值给 loc1
loc1.template combine<F>(loc2)	产生 locale loc1 的副本，并将 loc2 中型别为 F 的 facet 装入

几乎所有的构造函数均产生某个副本。复制 locale 是很廉价的操作，基本上只是设定一个指针并将引用计数累加 1 而已。产生一个变化过的 locale 的成本相当大，必须调整 locale 内每个 facet 的引用计数值。虽然 C++标准没有对此类操作的效率做出任何保证，但所有的实作版本大概都优先照顾 locale 的复制效率。

表 12-3 中的两个构造函数需要 locale 名称。除了“C”这个名称，其名称都没有标准规定。不过 C++标准要求每个 C++标准库附带的说明文件中必须列出可接受的名称。基本上，可以假设大部分实作版本均会接收前面章节所列名称。

成员函数 combine()运用最近才能被实作出的编译器特性：一个“带有显式指定之 template 参数”的成员函数模板。该模板 template 参数并非由另一个参数隐式推导而来。两个函数用于存取 locale 对象中的 facet，也采用相同的技术。不同之处在于：这两个函数是全局的 template 函数，不必加模板关键词 template。

函数 use_facet()会返回一个引用（reference），指向 facet。其型别是显式传递的模板 template 参数型别。若传入的 locale 不包含相应的 facet，则函数抛出 bad_cast 异常。函数 has_facet()可以用于检查某个 locale 中是否包含特定的 facet。见表 12-4。

表 12-4　刻面（facet）的存取

表达式	功能
has_facet<F>(loc)	若 locale loc 中有一个型别为 F 的 facet，则返回 true
Use_facet<F>(loc)	若返回一个 reference，指向 locale loc 中型别为 F 的刻面（facet）

类 locale 的其他操作见表 12-5。若某个类 locale 对象是从某个名称或若干个具名的 locale 构造而来的，名称将被保存起来。若参与构造的 locale 包含两个以上，则 C++标准并未构造出的名称给予保证。若一个 locale 是另一个 locale 的副本，或两个 locale 名字相同，则两个 locale 被认为是相同的。locale 的名称反映出被用来创建“具名 facet”的名称。运用该体制，两个 locale 若由具体的相同名称 facet 参与构造，生成的新名称也相同。C++标准本质上要求“由相同名称的 facet 组合而成”的 locale 均相同，才可能精心选出支持上述“相等性”思维。

表 12-5　Locale 的操作函数

表达式	功能
loc.name	返回 locale　loc 的名称字符串
loc1==loc2	若 loc1 和 loc2 是相同的 locale，返回 true

续表

表　达　式	功　　　　能
loc1!=loc2	若 loc1 和 loc2 是不同的 locale，返回 true
loc(str1, str2)	返回布尔值，显示字符串 str1 和 str2 的比较效果
locale::classic()	返回 locale("C")
locale::global(loc)	将 loc 安装为全局 locale，并返回上一个全局 locale

小括号操作符使我们得以运用 locale 对象作为字符串比较工具。此操作符运用 collate facet，按顺序比较参数所传递的字符串，即判断 locale 对象掌控时第一个字符串是否小于第二个字符串。这是仿函数的行为。采用 locale 对象作为 STL 算法所需的字符串排序规则。

```
std::vector<std::string> v;
std::sort(v.begin(), v.end(), locale("de_DE"));
```

12.3.3　区域表示和混合区域表示

由于默认区域表示不能修改，所以要根据需要创建区域的表示对象。std::locale 类提供 4 种创建区域表示对象的方法。

```
std::locale  localName();
std::locale  localeName(localeCode);
std::locale  localeName(locale1, locale2, cat);
std::locale  localeName(locale1, localeCode, cat);
```

第 1 种根据全局区域表示创建一个 locale 对象。第 2 种根据 localeCode 指明的区域表示代码字符串来创建 locale 对象。第 3 种根据名为 locale1 的 locale 对象创建一个副本，并用 locale2 对象中 cat 指明的刻面代替 locale1 的刻面。第 4 种与第 3 种类似，不同之处在于它的第 2 个参数是区域表示代码字符串 localeCode，指明刻面的所属区域。见表 12-6。

表 12-6　区域表示代码

语　　言	名　　　　称
中文	"chinese"
中文（简体）	"Chinese-simplified" 或 "chs"
中文（繁体）	"Chinese-traditional" 或 "cht"
捷克语	"czech" 或 "csy"
丹麦语	"danish" 或 "dan"
荷兰语	"dutch" 或 "nld"
荷兰语（比利时）	"belgian"，"dutch-belgian"或 "nlb"
英语（默认）	"english"
英语（澳大利亚）	"australian"，"ena" 或 "english-aus"
英语（加拿大）	"canadian"、"enc" 或 "English-can"
英语（新西兰）	"english-nz"或 "enz"
英语（英国）	"english-uk"，"eng" 或 "uk"
英语（美国）	"american"，"american english"，"american-english"，"english-american"，"english-us"，"english-usa"，"enu"，"us" 或 "usa"

续表

语　　　言	名　　　称
芬兰语	"finish" 或 "fin"
法语（默认）	"french" 或 "fra"
法语（比利时）	"french-belgian" 或 "frb"
法语（加拿大）	"french-canadian" 或 "frc"
法语（瑞士）	"french-swiss" 或 "frs"
德语（默认）	"german" 或 "deu"
德语（澳大利亚）	"german-austrian" 或 "dea"
德语（瑞士）	"german-swiss"，"des" 或 "swiss"
希腊语	"greek" 或 "ell"
匈牙利语	"hungarian" 或 "hun"
冰岛语	"icelandic" 或 "isl"
意大利语（默认）	"italian" 或 "ita"
意大利语（瑞士）	"italian-swiss" 或 "its"
日语	"japanese" 或 "jpn"
朝鲜语	"kor" 或 "korean"
挪威语（博克马尔语）	"norwegian-bokmal" 或 "nor"
波兰语	"polish" 或 "plk"
葡萄牙语（默认）	"portuguese" 或 "ptg"
葡萄牙语（巴西）	"portuguese-brazilian" 或 "ptb"
俄语（默认）	"russian" 或 "rus"
斯洛伐克语	"slovak" 或 "sky"
西班牙语（默认）	"spanish" 或 "esp"
西班牙语（墨西哥）	"Spanish-mexican" 或 "esm"
西班牙语（现代）	"Spanish-modern" 或 "esn"
瑞典语	"Swedish"或 "sve"
土耳其语	"turkish"或 "trk"

例 12-2 给出了创建和使用区域表示的过程，目的是按英语、法语、德语的格式显示日期。此句含义是：当程序调用仅有一个空字符串实参的 locale 构造函数时，返回的 locale 对象根据系统的用户设置被设置为本地区域表示。用户通常可以通过设置系统变量（LANG）来设置本地区域表示。

例 12-2

```
#include <iostream>
#include <locale>
#include <time.h>
using namespace std;
void main()
{
    char dataStr[81];
    time_t curtime;
    struct tm* tmTime;
```

```
    time(&curtime);                              //获取当前时间
    tmTime=gmtime(&curtime);
    strftime(dataStr,80,"%#x",tmTime);           //转换时间至 tm 结构
    std::locale native("");
    std::locale::global(native);                 //转换时间为字符串格式
    std::cout<<"Native Date: "<<std::endl;
    std::cout<<dataStr<<std::endl<<std::endl;
    std::locale german("german");                //更换国家
    std::locale::global(german);
    strftime(dataStr,80,"%#x",tmTime);
    std::cout<<"German Date: "<<std::endl;
    std::cout<<dataStr<<std::endl<<std::endl;
//
    std::locale chinese("chinese");              //更换国家
    std::locale::global(chinese);
    strftime(dataStr,80,"%#x",tmTime);
    std::cout<<"Chinese Date: "<<std::endl;
    std::cout<<dataStr<<std::endl<<std::endl;
//
    std::locale french("French");
    std::locale::global(french);
    strftime(dataStr,80,"%#x",tmTime);
    std::cout<<"French Date: "<<std::endl;
    std::cout<<dataStr<<std::endl<<std::endl;
}
```

程序的执行结果如下：

```
Native Date:
Friday, September 21, 2012
German Date:
Freitag, 21. September 2012
Chinese Date:
2012年09月21日
French Date:
vendredi 21 septembre 2012
French Date:
vendredi 21 septembre 2012
```

在上述代码中，代码行 std::locale native(“”); 是值得仔细体会的。

通过把一个区域表示的刻面修改成另一个区域表示的相应刻面，可以创建混合区域表示。下面用例 12-3 来加以说明。

例 12-3

```
#include <iostream>
#include <locale>
#include <clocale>
#include <ctime>
int main()
{
    char dataStr[81];
    time_t curtime;
    struct tm* tmtime;
    time(&curtime);
```

```
    tmtime=gmtime(&curtime);                          //获取当前时间
    strftime(dataStr,80,"%#x",tmtime);
    std::locale french(std::locale("french"),std::locale("american"),LC_TIME);
    std::locale::global(french);                      //设置国家
    strftime(dataStr,80,"%#x",tmtime);                //转换字符串
    std::cout<<"French Date: "<<std::endl;            //输出
    std::cout<<dataStr<<std::endl;
}
```

程序的执行结果如下：

```
French Date:
Friday, September 21, 2012
```

本例在调用 locale 类的构造函数时，新建的 locale 对象是法语区域表示，但该对象的时间刻面被替换为美国版，导致该区域表示的日期具有美国英语格式，其他刻面仍支持法语格式。LC_TIME 是变量。见表 12-7。

表 12-7 区域表示的种类值

值	说 明
LC_ALL	设置所有种类
LC_COLLATE	设置与排序函数有关的区域表示种类
LC_CTYPE	设置与字符分类函数有关的区域表示种类
LC_MONETARY	设置影响货币格式的区域表示种类
LC_NUMERIC	设置影响数字格式的区域表示种类
LC_TIME	设置影响时间和日期格式的区域表示种类

12.3.4 流和区域

在多数情况下，应用程序可以使用流简化区域表示，尤其在应用程序同时支持多种区域表示时。由于可以把区域表示灌输进一个流，此后这个流可以根据区域表示的刻面格式化数据。可以为程序需要支持的所有区域表示创建和灌输多个流。通过调用流对象的 imbue 成员函数，可以实现把某种区域表示灌输到流中。本章多次提到 imbue()成员函数。现举例说明，请参考例 12-4。

例 12-4

```
#include <iostream>
#include <locale>
using namespace std;
void main()
{
    locale native("");
    cout.imbue(native);                //使用 imbue
    cout<<"native Number: "<<std::endl;
    cout<<10999.82<<endl;
    locale dutch("dutch");
    cout.imbue(dutch);                 //使用 imbue
    cout<<"Dutch Number: "<<endl;
    cout<<10999.82<<endl;
    locale french("french");
    cout.imbue(french);
    cout<<"French Number: "<<endl;
```

```
    cout<<10999.82<<endl;
}
```

程序的执行结果如下：

```
native Number:
10999.8
Dutch Number:
10999,8
French Number:
10999,8
```

12.3.5 刻面的处理

由于 locale 对象的刻面提供用于本地化软件的服务，需要一种访问和使用该服务的方法。提供重要支持的模板函数包括两个：has_facet 和 use_facet。在给定的区域表示中存在指明的刻面，模板函数 has_facet 返回 true 值。例如：若要判断德语区域表示是否支持 ctype 刻面，编写以下代码：

```
std::locale german("german");
bool OK=std::has_facet<std::ctype<char>>(german);
```

每种类型的刻面均定义了函数，程序可以调用函数处理表示用到的数据。使用 use_facet 调用刻面对象的成员函数。介绍刻面函数超出了本章的内容安排。假定需要调用 ctype 类型刻面的 toupper 函数，在德语区域表示规则中把字符串转化成大写形式。举例如下：

```
string test="abcdefghijklmnopqrstuvwxyz";
char* first=test.begin();
char* last=test.end();
std::use_facet<std::ctype<char>>(german).toupper(first, last);
```

此处的 std::has_facet 和 std::use_facet，在调用某些实际参数时以前并未出现。

模板函数 has_facet 的使用方法如下：

```
template<class Facet> bool has_facet( const locale& _Loc );
```

参数 Loc 是现存 facet 的 locale 对象。如果 locale 类的对象具有被测试过的刻面，函数返回值为 true；否则，将返回 false。

说明：模板函数是非常有用的，尤其在一个 locale 中，检查非强制的刻面是否被列出。在模板函数 use_facet 被调用之前，避免抛出异常。

模板函数 use_facet 的使用方法：

```
template<class Facet> const Facet& use_facet( const locale& _Loc );
```

参数_Loc 是 locale 类型的常量，包含被引用的刻面类型。函数返回值是参数_Loc 的引用。函数使用说明：只要保留的任一版本 locale 存在，通过模板函数返回的刻面引用即保留了有效性。如果没有这样的刻面对象被列出在参数 locale 中，函数会抛出一个 bad_cast 类型的异常。

下面的例子阐述了模板函数 std::has_facet 和 std::use_facet 的用法。代码中首先声明了一个 name 为“german”的 locale，并替换为当前的系统 locale；接下来判断 std::ctype 是否可用，如果可用调用大写转换的方法将一段字符串转换为大写。同样对另外两个 name 为“German_Germany”和“English_Australia”的 locale 也进行了类似的操作。

例 12-5

```
#include <iostream>
#include <locale>
#include <string>
using namespace std;
void main()
{
    locale german("german");
    locale::global(german);
    bool OK=std::has_facet<std::ctype<char>>(german);
    if(!OK)
    {
        cout<<"Can't perform the conversion.";
    }
    else
    {
      cout<<"Can perform the conversion.\n";
    }
    char test[]="abcdefghijk";
    char* first=test;
    char* last=first+sizeof(test);
    cout<<"Original String: "<<endl;
    cout<<first<<endl;
    use_facet<ctype<char>>(german).toupper(first,last);                //变成大写
    cout<<"Converted String: "<<endl;
    cout<<first<<endl;
    locale loc1("German_Germany"), loc2("English_Australia");
    bool r1=use_facet<ctype<char>>(loc1).is(ctype_base::alpha,'a');//函数is()用于判断
    bool r2=use_facet<ctype<char>>(loc2).is(ctype_base::alpha,'!');
    if(r1)
    {
        cout<<"is alphabetic."<<endl;
    }
    else
    {
        cout<<"Is not alphabetic."<<endl;
    }
    if(r2)
    {
        cout<<"Is alphabetic."<<endl;
    }
    else
    {
        cout<<"Is not alphabetic."<<endl;
    }
}
```

程序的执行结果如下：

```
Can perform the conversion.
Original String:
abcdefghijk
Converted String:
ABCDEFGHIJK
is alphabetic.
Is not alphabetic.
```

12.4 标准 locale 的分类

每一个标准种类包括：刻面（facets）族。数据实例的格式化或分解，对于标准的使用或用户的流操作符“<<”和“>>”，可以作为成员 put()和 get()。每一个这样的成员函数均采用一个 ios_base& 参数，其成员 flags()、precision()和 width()用于规格化相应的数据。那些需要使用其他刻面的函数调用其成员 getloc()，以返回通道性质的 locale 对象。格式化刻面使用字符参数“fill”以完成特定的必需的宽度。

函数 put()成员不提供错误报告（任何 Outputiterator 类型参数必须被抽取，从已返回的迭代器中）。函数 get()成员采用一个 ios_base::iostate& 的参数，该参数的值可以忽略，但为防止描述错误要设置失败位（ios_base::failbit）。

12.4.1 类 ctype

类 ctype_base 的声明：

```
namespace std{
class ctype_base{
   public:
          enum mask{
  space=1<<0, print=1<<1, cntrl=1<<2, upper=1<<3, lower=1<<4, alpha=1<<5, digit=1<6,
  punct=1<<7, xdigit=1<<8, alnum=alpha|digit, graph=alnum|punct
          };
      };
}
```

类型 mask 是位掩码类型。

1. 模板类 ctype

模板类 ctype 的声明如下：

```
template <class charT> class ctype: public locale::facet, public ctype_base{
    public:
        typedef charT  char_type;
        explicit ctype(size_t refs=0);
    bool  is(mask  m,  charT  c )  const;
    const  charT*  is(const charT*  low,  const charT*  high,  mask* vec) const;
    const  charT*  scan_is(mask  m ,  const  charT* low, const charT* high) const;
    const  charT*  scan_not(mask  m ,  const  charT* low, const charT* high) const;
    charT  toupper(charT  c)  const;
    const  charT* toupper (charT*  low, const charT*  high ) const;
    charT  tolower(charT  c) const;
    const  charT* tolower(charT*  low, const charT*  high) const;
    charT  widen(char  c) const;
    const  char*  widen(const char* low, const char* high,  charT*  to) const;
    char   narrow(charT c,  char dfault) const;
    const  charT* narrow(const charT* low, const charT* , char dfault, char* to ) const;
    static  locale::id  id;
protected:
    ~ctype();
    virtual  bool  do_is(mask  m,  charT  c) const;
```

```
    virtual const charT* do_is(const charT* low, const charT* high, mask* vec ) const;
    virtual const charT* do_scan_is(mask m, const charT* low, const charT* high ) const;
    virtual const charT* do_scan_not(mask m, const charT* low, const charT* high ) const;
    virtual  charT  do_toupper(charT ) const;
    virtual const charT*  do_toupper(charT* low, const charT* high ) const;
    virtual  charT do_tolower(charT ) const;
    virtual  const charT* do_tolower(charT*  low,  const charT*  high)const;
    virtual  charT  do_widen(char) const;
    virtual  const char* do_widen(const char* low, const char* high, charT* dest ) const;
    virtual  char do_narrow(charT, char dfault ) const;
    virtual  const  charT*  do_narrow(const  charT*  low,
             const charT* high, char dfault, char* dest )const;
};
```

类 ctype 包装了 C 库头文件<cctype>的特性，流 istream 成员被要求使用 ctype<>，在输入分析时用于字符的分类。在表 12-2 中，即 ctype<char>和 ctype<wchar_t>的实例化过程中，字符分类适应本地字符集的实例过程。

下面逐一介绍类 ctype 的成员函数。

2. 类 ctype 的成员函数

（1）函数 is()

```
bool  is(mask  m ,  charT  c) const;
const charT* is(const  charT*  low, const  charT*  high,  mask* vec ) const;
```

函数返回值：do_is(m,c)或 do_is(low,high,vec)。这两个函数用于测试“单个字符是否具有特殊的属性或对每个字符进行分类，并将该字符存储在一个数组之中”。参数 m 是为字符设置的掩码；参数 c 是需要测试属性的字符；参数 low 和 high 指定一个范围，判断被测试字符是否在这个范围内；参数 vec 是指向存储字符掩码的数组的起始位置，这些掩码值描述了存储的每个字符的属性。第一个成员函数返回一个布尔量的值，如果被测试的字符具有掩码描述的属性，返回 true；如果不具备掩码规定的属性，返回 false。第二个成员函数返回一个指针，指向划定属性的范围的最后一个字符。

掩码值用于划分字符的分类，使用 STL 提供的类 ctype。类 ctype 是从 ctype 派生而来的。第一个 is()函数可以接受掩码规定的表达式，该表达式的第一个字符必须满足掩码的要求，掩码值是通过逻辑位操作实现的。

（2）scan_is()

```
const  charT*  scan_is(mask* m,  const charT* low,  const charT* high ) const;
```

函数返回值：do_scan_is(m, low, high)。函数 scan_is()用于定位和掩码匹配的第一个字符。参数 m、low 和 high 的意义和函数 is()相同。函数的返回值是一个指针，在指定范围内，指向和掩码匹配的第一个字符；如果该值不存在，函数返回 high。

（3）scan_not()

```
const charT* scan_not(mask  m, const charT* low, const charT* high) const;
```

函数返回值是 do_scan_not(m, low, high)。函数 scan_not()的功能和函数 scan_is()相反，用于定位和掩码不匹配的第一个字符。

（4）toupper()

```
charT  toupper(charT) const;
const charT* toupper(charT* low, const charT* high ) const;
```

函数返回值 do_toupper(c)或 do_toupper(low, high)。函数的功能是：转换单个字符或某个范围的字符成大写形式。第一个函数的参数代表需要被转换的字符；第二个函数的参数 low,high 用于限定需要转换的字符的范围[low, high]。

第一种形式的函数 toupper()返回其参数的大写形式；如果没有大写形式存在，将返回参数本身。第二种形式的函数 toupper()返回一个常量指针，该指针指向被转换的字符范围的最后一个字符。

（5）tolower()

```
charT tolower (charT  c) const;
const  charT*  tolower(charT*  low,  const  charT*  high ) const;
```

函数返回值是 do_tolower(c)或 do_tolower(low, high)。参数 c 是需要被转换成小写形式的字符，参数 low 和 high 用于划定需要转换成小写形式的字符范围[low, high]。第一种形式函数的返回值是参数 c 的小写形式，如果参数 c 没有小写形式存在，函数返回参数 c 本身；第二种形式函数返回一个常量指针，该指针指向范围[low, high]的最后一个字符。

（6）widen()

```
charT  widen(char c)const;
const  char*  widen(const  char*  low,  const char*  high, charT* to )const;
```

函数返回值：do_widen(c)或 do_widen(low, high, to)。第一种形式的函数返回本地类型字符 char 对应的 CharType 类型的字符；第二种形式的函数返回一个指向目标范围的指针，该指针类型为 CharType，该类型是通过 locale 将本地字符类型 char 转换而来的。

函数的功能：将本地字符集 char 类型的字符转换为相应的 CharType 类型字符，CharType 类型在 locale 类中使用。该方法是潜在的不安全的，就像该方法依赖调用者判断原有的值是正确的一样。参数 c 是本地字符集中的字符，该字符会被转换；参数 low 是一个指针，该指针指向需要被转换的字符范围的第一个字符；参数 high 也是一个指针，该指针指向规定范围([low, high])的最后一个字符后面的字符。参数 to 是一个指针，该指针指向目标范围中 CharType 类型的第一个字符，该范围中存储了需要被转换的字符。

提示：CharType 用于在 locale 类中表示字符类型。

（7）narrow()

```
char  narrow(charT  c,  char dfault) const;
const charT*  narrow(const  charT* low,  const charT* , char dfault, char* to )const;
```

函数返回值：do_narrow(c, dfault)或 do_narrow(low, high, dfault, to)。

函数的功能：将 locale 类中的 CharType 类型字符转换成相应的本地字符集的 char 类型的字符。同样，此方法也是不安全的，依赖于调用者判断其旧值是正确的。参数 c 是类 locale 中的 CharType 类型的数据，是用来被转换的；参数 dfault 参数默认为‘\0’，是该函数的默认值；参数 low 和 high 用于限定一个字符集的范围，该范围的字符会进行 narrow 转换；参数 to 是指向存储被转换字符的目标范围第一个字符的指针。

函数的返回值返回本地 char 类型的字符，该字符是和 CharType 参数 dfault 相对应的。第二种形式的函数返回一个指向本地字符集的目标范围指针，本地字符集是 CharType 类型的字符转换而来的。

第一种形式的函数返回 do_narrow(c, dfault)；第二种形式的函数返回 do_narrow(low,high, dfault, to)。仅仅基础的字符集被保证拥有独一无二的反向映像。对于最基本的源字符，表达式是成立的。

```
narrow(widen( c ), 0)==c
```

提示：任何 ctype 类的优先于 narrow()的方法同样也优先于 ctype 类的 narrow_s 方法。

3．类 ctype 的虚函数成员

（1）do_is()

```
bool do_is(mask m, charT c ) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;
```

上述两个函数用于分类一个字符或字符序列。对于每个参数，要识别掩码参数 vec 的值。第二种形式识别范围[low, high]中每个字符的掩码的值，并将其放入向量[p-low]中。第一种形式的函数返回表达式（(M&m)!=0）的结果；如果字符具备指定的特性，函数返回 true，第二种形式函数返回 high。

此函数被调用时，其功能是判断单个字符是否具有特定的属性；或对某范围内或数组内的每个字符进行分类。

如果被测试的字符具有特定的（由 mask 确定）属性，函数返回 true；否则函数返回 false。第二种形式的函数返回一个数组，该数组中容纳指定范围内字符的掩码特性。

（2）do_scan_is()

```
const charT* do_scan_is (mask m, const charT* low, const charT* high)const;
```

函数的作用是在一个缓冲区中定位一个字符，该字符满足分类（掩码）特性 m。函数的返回值是范围[low, high]中满足特定掩码特性的第一个指针，以至于函数 is(m, *p)会返回 true，否则返回 high。如果没有相应的值存在，函数返回 high。

（3）do_scan_not

```
const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
```

函数的作用是定位缓冲区中不满足条件的第一个字符。如果没有相应的值存在，函数返回 high。

保护成员函数返回范围[low, high]中最小的指针，该指针满足函数 do_is()的返回值为 false。

（4）do_toupper()

```
charT do_toupper (charT c) const;
const charT* do_toupper(charT* low, const charT* high ) const;
```

转换单个字符或多个字符，获取其大写形式。如果字符相应的大写形式存在，函数的第二种形式会取代指定范围[low, high]中的每个字符；否则返回该字符本身。如果参数 c 的大写形式存在的话，第一种形式的函数会返回参数 c 相应的大写字符；如果参数的大写形式不存在，返回字符本身。第二种形式会返回指针 high。

（5）do_tolower()

```
charT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high )const;
```

这两个函数的作用是将单个字符或多个字符转换成小写形式。第二种形式会转换范围[low, high]中的每个字符成为其小写形式，如果该字符不存在小写形式，会返回字符本身。第一种形式的函数，如果参数 c 的小写形式存在的话，会返回相应的小写形式；如果不存在会返回参数本身。第二种函数会返回参数 high。

（6）do_widen()

```
charT  do_widen(char  c) const;
const  char*  do_widen(const  char*  low,  const char* high, charT* dest) const;
```

此函数的作用是转换本地字符集中的 char 类型字符为相应的类 locale 中的 CharType 类型。本方法存在潜在的不安全性，它依赖于调用者判断该值是正确的。任何具名的 ctype 类刻面和有效的 ctype_base::mask 的值 M（is(M, c)||!ctw.is(M, do_widen(c))）为真。对于字符的第二种形式的转换，其结果保存在缓冲区的 dest 中。函数返回值是变化后的值，第二种形式返回 high。

（7）do_narrow()

```
char  do_narrow(charT  c, char dfault )const;
const  charT*  do_narrow(const  charT*  low,  const  charT*  high,  char  dfault,  char*
dest )const;
```

函数的作用是将 charT 类型的单个字符或多个字符转换为相应的 char 类型值。对于任何基本源字符集中的 char 类型字符 c，其变换具有以下性质：

```
do_widen(do_narrow(c,0))==c
```

对于任何具名的 ctype 类型，其刻面和掩码 ctype_base::mask 的值 M 之间表达式（下述）的值为真。

```
(is(M,c)||!ctc.is(M, do_narrow(c, dfault))
```

除非 do_narrow 返回 dfault。另外，对于任何数字式字符，表达式(do_narrow(c, dfault)-‘0’)评估为该字符的数字值。第二种形式转换范围[low, high]中每个字符，并将转换之后的结果放至缓冲区 dest 中。第一种形式返回转换之后的值，如果没有匹配的值会返回默认值 dfault；第二种形式返回 high。

4．举例

下面利用例 12-6 来说明类 ctype 的具体使用方法。

例 12-6

```
#include <iostream>
#include <locale>
using namespace std;
void main()
{
//widen
    locale loc("English");
    char* str="Hello everyone!";
    wchar_t s2[16];
    bool r = (use_facet<ctype<wchar_t> > ( loc).widen(str, str + strlen(str), &s2[0] ) != 0);
  // C4996
    s2[strlen(str)] = '\0';
    cout << str << endl;
    wcout << &s2[0] << endl;
    ctype<wchar_t>::char_type charT;
    charT = use_facet<ctype<char> > ( loc ).widen( 'a' );  //转换成宽字符
    cout<<charT<<" : "<<(char)charT<<endl;
//do_is  or  is
     locale loc1("German_Germany"), loc2("English_Australia");
```

```
    if(use_facet<ctype<char>>(loc1).is(ctype_base::alpha,'a'))
    {
      cout<<"The character 'a' in locale loc1 is alphabetic."<<endl;
    }
    else
    {
      cout<<"The character 'a' in locale loc1 is not alphabetic."<<endl;
    }
    if(use_facet<ctype<char>>(loc2).is(ctype_base::alpha,'!'))
    {
      cout<<"the character '!' in locale loc2 is alphabetic."<<endl;
    }
    else
    {
      cout<<"The character '!' in loc2 is not alphabetic."<<endl;
    }
    char* str2="Hello, my name is John!";
    ctype<char>::mask maskarray[30];
    use_facet<ctype<char>>(loc2).is(str2,str2+strlen(str2),maskarray); //用于判断
    for (int i = 0; i < strlen(str2); i++)
    {
  cout << str2[i] << ":  "<<maskarray[i]<<"  "<< (maskarray[i] & ctype_base::alpha ?
"alpha" : "not alpha")<<endl;
    };
    cout<<endl;
  //do_narrow and narrow
    locale loc3 ( "english" );
    wchar_t *str3 = L"\x0392fhello everyone";
    char str4 [16];
  bool r3 = (use_facet<ctype<wchar_t> > ( loc3 ).narrow( str3, str3 + wcslen(str3), 'X',
&str4[0] ) != 0); str4[wcslen(str3)] = '\0';
    wcout << str3 << endl;
    cout << &str4[0] << endl;
  //scan_is  and scan_not
    locale loc4 ( "German_Germany" );
    char *str5 = "Hello, my name is John!";
  const char* i = use_facet<ctype<char> > ( loc4 ).scan_is( ctype_base::punct, &str5[0],
&str5[strlen(&str5[0])-1] );
    cout << "The first punctuation is \"" << *i << "\" at position: "
       << i - str5 << endl;
  //
    i = use_facet<ctype<char> > ( loc4 ).scan_not( ctype_base::alpha, &str5[0],
&str5[strlen(&str5[0])-1] );
    cout << "First nonalpha character is \"" << *i << "\" at position: "
       << i - str5 << endl;
  //tolower
    locale loc5 ( "German_Germany" );
    char str6[] = "HELLO, MY NAME IS John!";
    use_facet<ctype<char> > ( loc5 ).tolower( &str6[0], &str6[strlen(&str6[0])-1] );
  //变成小写
    cout << "The lowercase string is: " << str6 << endl;
  //toupper
    locale loc6 ( "German_Germany" );
    char str7[] = "Hello, my name is John!";
```

```
    use_facet<ctype<char> > ( loc6 ).toupper( &str7[0], &str7[strlen(&str7[0])-1] );
//变成大写
    cout << "The uppercase string is: " << str7 << endl;
}
```

程序的执行结果如下：

```
Hello everyone!
Hello everyone!
97 : a
The character 'a' in locale loc1 is alphabetic.
The character '!' in loc2 is not alphabetic.
H:  769  alpha
e:  898  alpha
l:  770  alpha
l:  770  alpha
o:  770  alpha
,:  528  not alpha
 :  584  not alpha
m:  770  alpha
y:  770  alpha
 :  584  not alpha
n:  770  alpha
a:  898  alpha
m:  770  alpha
e:  898  alpha
 :  584  not alpha
i:  770  alpha
s:  770  alpha
 :  584  not alpha
J:  769  alpha
o:  770  alpha
h:  770  alpha
n:  770  alpha
!:  528  not alpha

Xhello everyone
The first punctuation is "," at position: 5
First nonalpha character is "," at position: 5
The lowercase string is: hello, my name is john!
The uppercase string is: HELLO, MY NAME IS JOHN!
```

5. ctype_byname 类

ctype_byname 类的声明如下（注意第二行黑体代码部分）：

```
namespace std{
    template<class charT> class ctype_byname : public ctype<charT>{
    public:
        typedef ctype<charT>::mask  mask;
        explicit  ctype_byname(const char* , size_t  refs=0);
    protected:
        ~ctype_byname();
         virtual  bool  do_is(mask  m ,  charT  c) const;
         virtual  const  charT* do_is(const charT*  low,  const charT* high,  mask* vec)const;
```

```
        virtual const charT* do_scan_is(mask m, const charT* low, const charT* high)const;
        virtual const charT* do_scan_not(mask m, const charT* low, const charT* high)const;
        virtual charT* do_toupper(charT)const;
        virtual const charT* do_toupper(charT* low, const charT* high )const;
        virtual charT* do_tolower(charT)const;
        virtual const charT* do_tolower (charT* low, const charT* high )const;
        virtual charT do_widen(char)const;
        virtual const char* do_widen (const char* low, const char* high, charT* dest )const;
        virtual char do_narrow(charT,char dfault )const;
        virtual const charT* do_narrow (const charT* low, const charT* high, char dfault,char* dest )const;
        };
}
```

6. ctype 类的实例化举例

ctype 类的实例化在本小节描述。主要实例化 char 类型。举例如下：

```
namespace std{
   template<>class ctype<char>: public locale::facet, pubic ctype_base{
public:
        typedef char char_type;
        explicit ctype(const mask* tab=0, bool del=false, size_t refs=0);
        bool is(mask m, char c)const;
        const char* is(const char* low, const char* high, mask* vec)const;
        const char* scan_is(mask m , const char* low, const char* high)const;
        const char* scan_not(mask m , const char* low, const char* high)const;
        char toupper(char c) const;
        const char* toupper(char* low, const char* high) const;
        char tolower(char c) const;
        const char* tolower (char* low, const char* high) const;
        char widen(char c) const;
        const char* widen(const char* low, const char* high, char* to) const;
        char narrow(char c) const;
        const char* narrow(const char* low, const char* high, char* to) const;
        static locale::id id;
        static const size_t table_size=IMPLEMENTATION_DEFINED;
protected:
        const mask* table() const throw();
        static const mask* classic_table() throw();
        ~ctype();
         virtual char do_toupper(char c) const;
         virtual const char* do_toupper(char* low, const char* high) const;
         virtual char do_tolower(char c)const;
         virtual const char* do_ tolower (char* low, const char* high) const;
         virtual char do_widen(char c) const;
         virtual const char* do_widen(const char* low, const char* high, char* to) const;
         virtual char do_narrow(char c) const;
         virtual const char* do_narrow(const char* low, const char* high, char* to) const;
    };
}
```

一个类 ctype 的实例化特例被提供以至于类型 char 的成员函数可以被在线实施。可实施的成员 table_size 的值至少等于 256。

（1）ctype<char>析构器

```
~ctype();
```

析构器的作用：如果析构器的第一个参数是非零的，并且析构器的第二个参数是“true”，析构器的作用是执行以下语句：

```
delete[ ] table();
```

（2）ctype<char>成员函数

下面介绍 ctype<char>的成员函数。对于非符号字符型的值 v，此处(v>=table_size)，table()[v]被假定拥有一个可实施的值（对于每个这样的值，存在可能的差异），且没有执行数组搜索。

```
explicit  ctype(const mask* tbl=0, bool del=false, size_t refs=0);
```

前提条件：tbl 或者为 0，或者是 table_size 的数组，该数组至少包含 table_size 个元素。函数的作用是传递参数 refs 给其基类的构造器。

```
bool  is(mask  m,  char  c) const;
const  char*  is(const  char*  low,  const char* high,  mask* vec)const ;
```

函数的作用：第一种形式函数会返回 table()[(unsigned char) c]& m，第二种形式返回参数 high。

```
const char* scan_is(mask m, const char* low, const char* high)const;
```

函数返回值是指定范围[low, high]中的最小元素，以至于 table()[unsigned char* p]&m 的值为 true。

```
const char* scan_not(mask m ,  const char* low, const char* high)const;
```

函数返回值是指定范围[low, high]中的最小值，以至于 table()[unsigned char* p]&m 的值为 false。

```
char   toupper(char  c) const;
const  char*  toupper(char*  low, const char* high) const;
```

函数返回值是 do_toupper(c)或 do_tolower(low, high)。

```
char   tolower(char  c) const;
const  char*  tolower (char*  low, const char* high) const;
```

函数返回值是 do_tolower(c)或 do_tolower(low, high)。

```
char widen(char c)const;
  const char*  widen(const  char*  low,  const  char*  high,  char* to) const;
```

函数的返回值是 do_widen(c)或 do_widen(low, high, to)。

```
char narrow(char c, char /*dfault*/) const;
const char* narrow(const char* low, const char* high, char/*dfault*/, char* to) const;
```

函数的返回值是 do_narrow(c)或 do_narrow(low, high, to)。

```
const mask* table() const throw()
```

函数返回值：如果构造器的第一个参数是非零，函数返回值是该参数；否则，返回 classic_table()。

（3）ctype<char>的静态成员

```
static const  mask* classic_table() throw();
```

函数返回值是数组初始元素的指针，数组的大小为 table_size，该指针代表了 locale 类的字

符类别。

（4）ctype<char>的虚函数

```
char  do_toupper(char)  const;
const char* do_toupper(char* low, const char* high) const;
char   do_tolower(char)const;
const  char* do_tolower(char* low , const char* high)const;
virtual char  do_widen(char c)const;
virtual const char* do_widen(const char* low, const char* high, char*  to) const;
virtual  char  do_narrow(char  c,  char  dfault)const;
virtual const char* do_narrow(const char* low, const char* high, char dfault, char* to)const;
```

上述这些函数和 ctype 模板中与其同名的函数具有同样的功能。

7. 类 ctype_byname<char>描述

类 ctype_byname<char>的声明如下：

```
namespace std{
template <> class ctype_byname<char> : public ctype<char>{
public:
   explicit  ctype_byname(const char*, size_t refs=0);
protected:
   ~ctype_byname();
   virtual char  do_toupper(char  c) const;
   virtual const  char*  do_toupper(char*  low, const char* high)const;
   virtual char  do_tolower(char  c)const;
   virtual const char*  do_tolower(char* low, const char* high)const;
   virtual char  do_widen(char  c)const;
   virtual const  char*  do_widen(const  char*  low, const char* high, char* to);
   virtual char  do_narrow(char  c,  char  dfault)const;
   virtual const  char*  do_ narrow (const  char*  low, const char* high, char* to);
   }
}
```

8. 模板类 codecvt

模板类 codecvt 的声明如下：

```
namespace  std{
    class codecvt_base{
       public:
    enum result {ok, partial, error, noconv};
   };
   template<class internT, class externT, class stateT>class codecvt: public locale::facet,
public codecvt_base{
   public:
    typedef internT  intern_type;
    typedef externT  extern_type;
    typedef stateT  state_type;
    explicit codecvt(size_t refs=0);
    result  out(stateT& stste, const internT* from, const internT* from_end,
         const   internT*&   from_next,   internT*   to,   internT*   to_limit,   internT*&
to_next)const;
         int encoding() const throw();
         bool always_noconv() const throw();
```

```
        int length(stateT&, const externT* from, const externT* end, size_t  max )const;
        int  max_length() const throw();
        static locale::id  id;
    protected:
        ~codecvt();
    virtual result  do_out(stateT& state, const internT* from, const internT* from_end, const
internT*&
        from_next, externT* to, externT* to_limit, externT* to_next)const;
    virtual result do_in(stateT& state, const extent* from, const externT* from_end, const
externT*& from_next,
        internT* to, internT* to_limit, internT*& to_next) const;
    virtual result do_unshift(stateT& state, externT* to, externT* to_limit, externT*& to_next)const;
    virtual int do_encoding() const throw();
    virtual bool do_always_noconv() const throw();
    virtual int do_length(stateT&, const externT* from, const externT* end, size_t max)const;
    virtual int do_max_length() const throw();
      };
    }
```

类 codecvt<internT, externT, stateT>是用来转换代码集的。例如：从宽字符到多字节字符；或者宽字符编码之间的转换（例如从 unicode 码到 EUC 码的转换）。

stateT 类型参数是中间类型，是两种类型在相互转换时需要的中间状态类型，需要选择成对的代码集，并且两个代码集之间互相匹配。

类的实例化通常有 codecvt<wchar_t, char, mbstate_t>和 codecvt<char, char, mbstate_t>，用于转换本地字符集。实例化类 codecvt<char, char, misstate_t>实施一种退化的转变；它不实行根本的变换。实例化类 codecvt<wchar_t, char, mbstate_t>实现在本地字符集的窄字符集和宽字符集之间进行转换。mbstate 的实例化实现在两种编码之间的转换。另一种编码可以使用自定义类型 stateT 实现转换。stateT 类型的对象可以包含任何状态，这些状态有利于在特定的 do_in 或 do_out 成员之间通信。

（1）codecvt 类成员

```
    result  out(stateT& state, const internT* from, const internT* from_end, const internT*&
from_next,
    externT* to, externT* to_limit, externT*& to_next) const;
```

函数返回值是 do_out(state, from, from_end, from_next, to, to_limit, to_next)。转换内部类型 CharType 的序列为外部类型 Bytes 的序列。

参数 state：代表一种转换状态，是在调用成员函数之间的中间状态。

参数 from：该指针指向被转换序列的开始。

参数 from_end：该指针指向被转换序列的尾部。

参数 from_next：前一个 CharType 类型被转换之后，指向其后第一个没有转换的 CharType 类型数据。

参数 to：该指针指向已转换序列的开始位置。

参数 to_limit：该指针指向已转换序列的尾部。

参数 to_next：该指针指向已转换数据之后的第一个未转换的 Byte 类型数据。

```
    result  unshift(stateT& state, externT* to, externT* to_limit, externT*& to_next)const;
```

函数返回值是 do_unshift(state, to, to_limit, to_next)。在转换过程中，为了完成序列中最后一个

Byte 类型字符的转换，要提供必需的 Byte 类型数据。参数 state 是中间过渡状态数据类型；参数 to 是指针，该指针指向目的范围的第一个位置；参数 to_limit 也是指针，该指针指向目的范围的最后一个位置；参数 to_next 是指针，该指针指向目的序列中第一个未转换的元素。

```
    result in(stateT& state, const externT* from, const externT* from_end, const externT*&
from_next,
    internT* to, internT* to_limit, internT*& to_next )const;
```

函数返回值是 do_in(state, from, from_end, from_next, to, to_limit, to_next)。函数的作用是将一种外部的 Bytes 类型的序列表示转换成为一种内部的 CharType 类型的序列。参数 state 是中间过渡状态；参数 from 是指针，指向被转换序列的开始位置；参数 from_end 是指针，该指针指向被转换序列的尾部；参数 from_next 是指针，指向已转换序列尾部之后第一个未转换字符；参数 to 是指针，该指针指向已转换序列的起始位置；参数 to_limit 是指针，该指针指向已转换序列的尾部；参数 to_next 是指针，该指针指向目标序列中已转换的 CharType 序列的第一个未被转换的字符。

```
    int encoding() const throw();
```

函数返回 do_encoding()。函数的作用：如果字节流的编码方式是状态独立的，函数判断 Byte 和 CharType 之间的转换比率是否是常量，如果是常量，决定比率参数 ratio 的数值。函数的返回值如果是正值，该值是 Byte 类型字符的常量数量，这些 Byte 类型字符用于产生 CharType 类型的字符。函数返回值通常有 3 种形式，见表 12-8。

表 12-8　函数 encoding 的返回值

返　回　值	条　　　　件
-1	如果 extern_type 类型序列的编码是状态独立的
0	如果编码涉及的序列长度是可变的
N	如果编码涉及的序列长度为 N

```
    bool always_noconv() const throw();
```

函数返回值是 do_always_noconv()。函数 always_noconv()用于判断是否没有转换需要执行。函数的返回值是逻辑量，如果没有转换被执行，函数返回 true；如果至少有一个转换被执行，函数返回 false。

```
    int length(stateT& state, const externT* from, const externT* from_end, size_t max )const;
```

函数返回值是 do_length(state, from, from_end, max)。length()函数的作用：判断有多少 Byte 类型的字节数，并返回该字节的数目。参数 state 是调用函数时要保持的中间转换状态；参数 from 是指针，指向外部序列的起始位置；参数 from_end 是指针，指向外部序列的尾部；参数 max 代表函数可以返回的 Byte 类型的最大数值。

（2）类的虚函数

```
    result  do_out(stateT& state, const internT* from, const internT* from_end, const internT*&
from_next,
        externT* to, externT* to_limit, externT*& to_next) const;
    result  do_in(stateT& state, const externT* from, const externT* from_end, const externT*&
from_next,
        internT* to, internT* to_limit, internT*& to_next) const;
```

前提条件：(from<=from_end&&to<to_end)被预先定义，且其值为 true。state 是预先初始化的，

它在序列的起始位置，否则等于序列中转换字符的结果。

函数的作用：转变范围[from , from_end]中的字符，将其放置于目标序列 to 中。转换不少于 from_end-from 个元素，并将存储不少于 to_limit-to 个目标元素。面对一个字符不能转换的情况时，函数将中止；函数总是保留指针 from_next 和 to_next，指向已成功转换的最后一个元素。函数将返回的 Noconv，internT 和 externT 是同一类型，并且被转换的序列和输入序列[from, fro_next]是一致的。指针 to_next 被设定和指针 to 相等，state 的值是不变的，并且范围[to, to_limit]中的值是不变的。

提示：当参数 state 未指定时，函数才能运行。

函数返回值有以下几种选项（见表 12-9）：

表 12-9　函数 do_out 和函数 do_in 的返回值

值	意　义	值	意　义
ok	完成转换	error	在范围[from, from_end]中存在 1 个字符不能转换
partial	部分源字符被转换	noconv	internT 和 externT 是同一类型，输入序列和被转换序列是一致的

返回值 partial 说明或者目标序列不能吸收所有的目标元素，或者另一个目标元素产生之前，附加的源元素是必要的。

```
    result do_unshift(stateT& state, externT* to, externT* to_limit, externT*&
to_next)const;
```

函数的作用：当前状态 stateT 等于 state 时，将起始位置为 to 的字符放置入序列中。存储不少于 to_limit-to 个目标元素。函数总是保留 to_next 指针，该指针指向被存储的最后一个元素的前面。函数的返回值也是一个枚举变量，其枚举值见表 12-10。

表 12-10　函数 do_unshift 的返回值

值	意　义	值	意　义
ok	完成序列	error	state 具有一个无效的值
partial	更多的字符需要用于完成终止	noconv	对于 state_type 类型，不需要终止

```
    codecvt<char, char, mbstate_t>返回 noconv。
        int do_encoding() const throw();
```

当 externT 类型序列编码是状态独立的，函数返回-1；否则常量数目的 externT 字符需要产生一个内部字符；如果数目不是常量，返回 0。对于所有的有效参数，如果 do_in()和 do_out()返回 noconv，函数返回值为真；codecvt<char, char, mbstate_t>返回 true。

```
    int do_length(stateT& state, const externT* from, const externT* from_end, size_t
max)const;
```

前提条件：(from<=from_end)预先定义，其值为 true。state 被初始化之后，序列的起始或许等于序列中被转换字符的结果。当参数 state 不确定时，函数调用 do_in()，do_in 的参数 to 指向包含至少 max 个元素的缓冲区。

如果 from_next 是范围[from, from_end] 中的最大值，则范围[from, from_next]中的值表示 internT 类型的最大的或次大的有效完整字符。

```
    int do_max_length()const throw();
```

返回 do_length 返回的最大值，并且 stateT 的值为 state。

实例化函数 codecvt<char, char, mbstate_t>::do_max_length()会返回 1。

下面使用例 12-7 说明类模板 codecvt 的使用方法。

例 12-7

```
#define _INTL
#include <locale>
#include <iostream>
#include <wchar.h>
#define LEN 90
using namespace std;
void main()
{
    char pszExt[LEN+1];
    wchar_t *pwszInt = L"This is the wchar_t string to be converted.";
    memset( &pszExt[0], 0, ( sizeof( char ) )*( LEN+1 ) );
    char* pszNext;
    const wchar_t* pwszNext;
    mbstate_t state;
    locale loc("C");//English_Britain"); //German_Germany
    int res = use_facet<codecvt<wchar_t,char, mbstate_t>>(loc).out( state,pwszInt,&
pwszInt[wcslen(pwszInt)], pwszNext ,pszExt, &pszExt[wcslen(pwszInt)],pszNext);    //输出
    pszExt[wcslen(pwszInt)] = 0;
    cout<<pszExt<<endl;
    char* pszExt2 = "This is the string to be converted!";
    wchar_t pwszInt2 [LEN+1];
    memset(&pwszInt2[0], 0, (sizeof(wchar_t))*(LEN+1));
    const char* pszNext2;
    wchar_t* pwszNext2;
    mbstate_t state2 = {0};

    res = use_facet<codecvt<wchar_t, char, mbstate_t>>(loc).in( state,pszExt2, &pszExt2
[strlen(pszExt2)], pszNext2,pwszInt2, &pwszInt2[strlen(pszExt2)], pwszNext2 );      //输入
    pwszInt2[strlen(pszExt2)] = 0;
    wcout<<pwszInt2<<endl;
    locale loc2 ( "German_Germany" );
    res = use_facet<codecvt<char, char, mbstate_t>>(loc2).encoding();        //编码
    cout<<"Ratio: "<<res<<endl;
    locale loc3 ( "German_Germany" );
    bool resB = use_facet<codecvt<char, char, mbstate_t>>(loc).always_noconv();
    if (resB)
        cout << "No conversion is needed." << endl;
    else
        cout << "At least one conversion is required." << endl;
    char* pszExt3 = "This is the string whose length is to be measured!";
    mbstate_t state3 = {0};
    locale loc4("C");//English_Britain");//German_Germany
    res = use_facet<codecvt<wchar_t, char, mbstate_t>>(loc).length(state3,pszExt3,
&pszExt3[strlen(pszExt3)], LEN );                                                 //长度
    wcout << "The length of the string is: "<<res<<" ."<<endl;
}
```

程序的执行结果如下：

```
This is the wchar_t string to be converted.
This is the string to be converted!
Ratio: 1
No conversion is needed.
The length of the string is: 50 .
```

9. 模板类 codecvt_byname

```
namespace std{
    template <class internT, class externT, class stateT>
    class codecvt_byname: public codecvt<internT, externT, stateT>{
    public:
        explicit  codecvt_byname(const char*, size_t refs=0);
    protected:
        ~codecvt_byname();
        virtual result do_out(stateT& state, const internT* from, internT* from_end, const 
internT*& from_next,
            externT* to, externT* to_limit, externT*& to_next) const;
        virtual result do_in(stateT& state, const externT* from, externT* from_end, const 
externT*& from_next,
            internT* to, internT* to_limit,internT*& to_next) const;
        virtual  result  do_unshift(stateT&  state,  externT*  to,  externT*  to_limit, 
externT*& to_next) const;
        virtual int do_encoding() const  throw();
        virtual bool do_always_noconv() const throw();
       virtual int  do_length(stateT&, const externT* from, const externT* end, size_t max) const;
        virtual result do_unshift(stateT& state, externT* to, externT* to_limit, externT* 
to_next) const;
        virtual int do_max_length() const throw();
    };
}
```

12.4.2 数值类的 locale 类

类 num_get<>和 num_put<>处理数值格式和数值分解。虚函数被提供用于几种数值类型。实施过程中会代理较小类型为较大的类型。所有特例化的成员函数仅仅应用于实例化类。这些实例化类可以是 num_get<char>，num_get<wchar_t>，num_get<C, InputIterator>，num_put<char>，num_put<wchar_t>和 num_put<C, OutputIterator>。这些实例化类涉及了 ios_base&类型的参数，用于格式化特例和它的激化 locale 类。刻面 numpunct<>用于识别所有的数值类标点优先，同时刻面 ctype<>用于字符分类。

对于标准流的抽出器和插入器成员，使用 num_get<>和 num_put<>成员函数格式化或分解数值。

1. 模板类 num_get

```
namespace std{
template<class charT, class InputIterator= istreambuf_iterator<charT> >
    class num_get: public locale::facet{
public:
    typedef  charT  char_type;
    typedef  InputIterator  iter_type;
    explicit  num_get(size_t  refs=0);
```

```
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, bool& v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, long& v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, unsigned
short & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, unsigned int
& v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, unsigned
long & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, float & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, double & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err,
long double & v);
    iter_type get(iter_type in, iter_type end, ios_base&, ios_base::iostate& err, void*& v);
       static locale::id id;
    protected:
    ~num_get();
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, bool&
v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, long&
v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err,
unsigned short& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err,
unsigned int& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err,
unsigned long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err, float& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate*
err,double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err,long
double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&, ios_base::iostate* err,void*&
v) const;
       };
    }
```

刻面 num_get 用于分解输入序列中的数值数据。

模板类 num_get 中包含了部分普通成员函数 get()，还包含了一部分虚函数 do_get()。对于虚函数 do_get()，实现从输入流 in 中读取字符，并按照 str.flag()阐释这些字符，使用刻面 use_facet<ctype<chart>>(loc)和 use_facet<numpunct<charT>>(loc)，此处 loc 是 str.getloc()的返回值，如果产生错误的话，val 是不变的，否则其将被设置为结果数值。参数的运行细节按以下 3 个阶段发生：

阶段 1：判断一个特殊的转换；

阶段 2：从流 in 中抽出字符，并判断相应的字符值；

阶段 3：存储结果。

各阶段的细节如下：

阶段 1：初始化本地变量。

```
fmtflags  flags=str.flags();
fmtflags  basefield = (flags& ios_base::basefiled);
fmtflags  uppercase=(flags& ios_base::uppercase);
```

```
fmtflags boolalpha=(flags& ios_base::boolalpha);
```

对于转换整数类型 inter，函数决定整形转换，汇总见表 12-11。

表 12-11　整形转换

类型	Stdio equivalent	类型	Stdio equivalent
basefield==oct	%o	signed integral type	%d
basefield==hex	%X	Unsigned integral type	%u
basefield==0	%i		

当转换浮点类型时，使用标识符“%g”。当转换空指针(void*)时，需要使用标识符“%p”。如果需要的话，还可以使用长度标识符，见表 12-12。

表 12-12　长度调整标志

类　型	长度调整符	类　型	长度调整符
short	h	unsigned long	l
unsigned short	h	double	l
long	l	long double	L

阶段 2：如果 in==end，阶段 2 将终止。否则 charT 类型会从流 in 中抽取，并初始化给本地变量。

```
    char_type  ct=*in;
    char  c=src[find(atoms, atoms+sizeof(src)-1,ct)-atoms]
    if(ct==use_facet<numpunct<charT>>(loc).thousands_sep()&&use_facet<numpunct<charT>>(loc).gr
ouping()
    .length()!=0)
```

值 src 和 atoms 被定义：

```
    static const char src[]="0123456789abcdefABCDEF+-";
    char_type  atoms[sizeof(src)];
    use_facet<ctype<charT>>(loc).widen(src, src+sizeof(src),atoms);
```

如果参数 discard 是 true，字符的位置需要被记忆，否则字符可以被忽略。如果 discard 是 false，需要检查字符 c 是否被允许作为第一阶段中符号转换的输入流中的下一个字符。如果为真，该字符被收集。

如果字符或者被抛弃或者被收集，输入流使用“++”可以前进并进行处理，例如：++in。

阶段 3：阶段 2 的处理结果可能是以下两种：

（1）在阶段 2 中，一个字符序列被收集，并被转换成该类型的值。该值被存储在 val 中，并且 ios_base::goodbit 被存储在变量 err 中。

（2）阶段 2 收集的字符序列会引起扫描，并汇报输入错误。之后 ios_base::failbit 被设置给变量 err。数字分组被检查。即抛弃分隔符的位置需要被检查，并和 use_facet<numpunct<charT>>(loc).grouping()保持一致。如果它们不能够一致，ios_base::failbit 的值将被设置给变量 err。

任何情况下，如果阶段 2 处理条件（in==end）被终止，之后 err|=ios_base::eofbit 被执行。

```
    iter_type  do_get(iter_type  in, iter_type  end, ios_base& str, ios_base::iostate& err,
bool& val)const;
```

如果(str.flag()&&ios_base::boolalpha)==0，输入操作将继续，输入的值被存入 val 中，按照以下

步骤，值被判断：如果被存储的值是 0，　将存储 false；如果被存储的值是 1，将存储 true。否则，语句 err|=ios_base::failbit 被执行，并不存储任何值。

否则目标序列被判断，似乎是通过调用刻面的函数 falsename()和函数 truename()。输入型迭代器 in 同 end 比较，必要时仅相差一个字符。当且仅当目标序列被单独匹配时，val 被设置给相应的值。范围[in, end]中的连续字符可以获取，在目标序列中并不按位置匹配。迭代器 in 总是指向成功匹配的最后一个元素之前的某个位置。如果 val 被设置，然后 err 被设置为 str.goodbit 的值；或被设置为 str.eofbit 的值。当寻找另一个匹配的字符时，可以发现(in==end)。如果 val 没有被设置，然后 err 被设置为 str.failbit 的值。如果失败的原因是(in==end)，或者将(str.failbit|str.eofbit)的值设置给 err。

函数返回指针 in。

2. 模板类 num_put

模板类 num_put 的声明如下：

```
namespace std{
    template <class charT, class OutputIterator=ostreambuf_iterator<charT> >
        class num_put: public locale::facet{
        public:
            typedef  charT   char_type;
            typedef  OutputIterator  iter_type;
            explicit  num_put(size_t refs=0 );
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill, bool v) const;
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill, long v)const;
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill, unsigned long v)const;
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill,double v)const;
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill,long  double v)const;
            iter_type  put(iter_type  s,  ios_base& f, char_type  fill,const void*  v)const;
        static locale::id  id;
    protected:
        ~num_put();
        virtual  iter_type  do_put(iter_type, ios_base&, char_type fill, bool v)const;
        virtual  iter_type  do_put(iter_type, ios_base&, char_type fill, long v)const;
        virtual iter_type do_put(iter_type, ios_base&, char_type fill,unsigned long v)const;
        virtual iter_type do_put(iter_type, ios_base&, char_type fill,double v)const;
        virtual iter_type do_put(iter_type, ios_base&, char_type fill,long double v)const;
        virtual iter_type do_put(iter_type, ios_base&, char_type fill,const void* v)const;
    };
}
```

刻面 num_put 用于格式化数值数据成为一个字符串序列（像一个输出流）。虚函数 do_put()的作用是：写字符至序列 out 中，格式化 val。下面的语句实现本地类对象变量的初始化：

```
locale  loc=str.getloc();
```

函数的运行细节发生时一般包括以下 4 个阶段：

阶段 1：判断一个 printf 的转换符标志 spec，并决定字符的输出。假定当前的 locale 对象属于 C 类型 locale。

阶段 2：调整表示方式。转换阶段 1 决定的每一个字符称为宽字符形式。其值被函数 getloc()返回。

```
use_facet<numpunct<charT>>(str.getloc());
```

阶段 3：判断何处是无用的。

阶段 4：插入序列至流 out 中。

对于每个阶段的详细描述如下：

阶段 1：阶段 1 的第一个行为是判断转换的符号标志。描述该判断的表使用下述本地变量。

```
fmtflags  flags=str.flags();
fmtflags  basefield=(flags&(ios_base::basefield));
fmtflags  uppercase=(flags&(ios_base::uppercase));
fmtflags  floatfield=(flags&(ios_base::floatfield));
fmtflags  showpos=(flags&(ios_base::showpos));
fmtflags  showbase=(flags&(ios_base::showbase));
```

在阶段 1 中，所有使用的表均是有序的。即第一行的条件为 true 时才应用。当没有代码行起作用时，第一行代码无条件执行。对于从整型到字符型的转换，函数判断整型转换符，见表 12-13。

表 12-13　整型数值转换字符型

State	stdio 等效
basefield==ios::base::oct	%o
(basefield==ios::base::hex)&&!uppercase	%x
(basefield==ios::base::hex)	%X
for a signed integral type	%d
For an unsigned integral type	%u

对于浮点类型的转换，函数决定使用浮点转换符号标志，见表 12-14。

表 12-14　浮点类型转换字符型

State	stdio 等效
floatfield==ios::base::fixed	%f
(floatfield==ios::sctentific)&&!uppercase	%e
(floatfield==ios::base::scientific)	%E
!uppercase	%g
otherwise	%G

对于转换过程中添加数据长度的标志，见表 12-15。

表 12-15　格式中的长度标志符

type	length modifier	type	length modifier
long	l	long double	L
unsigned long	l	otherwise	none

转换符具有以下可选择的预先附属资格，见表 12-16。

表 12-16　数值转换

Type	State	stdio equivalent
an integral type	flags&showpos	+
	flags&showbase	#
an floating-point type	flags&showpos	+
	flags&showpoint	#

为转换一个浮点类型，如果(flags&fixed)!=0 或如果 str.precision()>0，那么 str.precision()是指定的数值。对于空指针的转换，可以使用转换符%p。

阶段 1：最后的表达式包括 char's，通过调用 printf(s, val)可以被打印。此处，s 是由上述决定的转换符。

阶段 2：除了十进制小数点（.）之外，使用刻面 use_facet<ctype<chart>>(loc).widen(c)，任何字符 c 均可被转换为一个 charT。通过 numpunct<charT>punct=use_facet<numpunct<charT>>(str.getloc())，本地变量 punct 可以被初始化。对于整型类型，punct.thousands_sep()字符可以被插入序列中，其值由函数 punct.do_grouping()获取。十进制小数点符号（.）可以使用函数 punct.decimal_point()替换。

阶段 3：本地变量可以被初始化，例如：

```
fmtflags adjustfield=(flags&(ios_base::adjustfield));
```

任何位置的衬垫可以由表 12-17 确定。

表 12-17　填充衬垫

State	Location
adjustfield==ios_base::left	pad after
adjustfield==ios_base::right	pad before
adjustfield==internal and a sign occurs in the representation	pad after the sign
adjust field==internal and representation after stage1 began with 0x or 0X	pad after x or X
otherwise	pad before

如果 str.width()是非零的，并且阶段 2 之后的序列中 charT 的数目小于 str.width()；然后足够的 fill 字符被添加至序列中，序列的长度要自动添补至规定长度 str.width()。

阶段 4：阶段 3 末尾形成的 charT 类型的序列通过以下代码可以输出：

```
*out++=c;
```

函数 put 的使用方法：

```
iter_type  put(iter_type out, ios_base& str, char_type fill, bool val)const;
```

函数的作用：如果(str.flags()&ios_base::boolalpha)==0，　然后执行以下语句：

```
out=do_out(out, str, fill, (int)val);
```

否则：

```
const numpunct<charT>& np=use_facet<numpunct<charT>>(loc);
string_type  s= val ? np.truename():np.falsename();
```

并且将诸多 s 中的字符插入流 out.out 中。

例 12-8 主要用于说明数值类的使用方法。

例 12-8

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

```
void main()
{
    locale loc( "german_germany" );
    basic_stringstream<char> psz, psz2;
    psz << "-1000,56";
    cout<<psz.str()<<endl;
    ios_base::iostate st = 0;
    long double fVal;
    cout << use_facet <numpunct <char> >(loc).thousands_sep( ) << endl; //分隔符
    psz.imbue( loc );
use_facet <num_get <char> >(loc).get( basic_istream<char>::_Iter( psz.rdbuf()),basic_istream
<char>::
    _Iter(0), psz, st, fVal );                                      //使用 get()函数
    if ( st & ios_base::failbit )
        cout << "money_get() FAILED" << endl;
    else
        cout << "money_get()= "<< fVal << endl;
    st = 0;
    cout <<"The thousands separator is:"<<use_facet<numpunct<char>>(loc).thousands_sep()<<endl;
    psz2.imbue( loc );                                              //使用 imbue
    use_facet <num_put<char>>(loc).put(basic_ostream<char>::_Iter(psz2.rdbuf()),psz2,
' ', fVal=1000.67);
//使用 put 函数
    if (st & ios_base::failbit)
        cout<<"num_put( ) FAILED"<<endl;
    else
        cout<<"num_put( ) = "<<psz2.rdbuf()->str()<<endl;
}
```

程序的执行结果如下：

```
-1000,56
.
money_get()= -1000.56
The thousands separator is: .
num_put( ) = 1.000,67
```

下面给出例 12-9，用于显示函数 imbue()的使用方法。

例 12-9

```
#include <iostream>
#include <locale>
int main( )
{
    using namespace std;
    cout.imbue( locale( "french_france" ) );           //使用 imbue()
    double x = 123.123456;
    cout << x << endl;
    locale native("");
    cout.imbue( native);
    double  y=25.68;
    cout<<y<<endl;
}
```

程序的执行结果如下：

```
123,123
25.68
```

12.4.3　刻面 numeric punctuation

刻面 numpunct 主要用于描述程序开发过程中的标点符号。其成员函数主要包括：构造函数，decimal_point()，thousands_sep()，grouping()，truename()，falsename()和 id。

1. 模板类 numpunct

```
namespace std{
template<class  charT>  class numpunct : public locale::facet{
public:
    typedef  charT;
    typedef  basic_string<charT> string_type;
    explicit  numpunct(size_t  refs=0);
    char_type  decimal_point()  const;
    char_type  thousands_sep()  const;
    string  grouping()   const;
    string_type   truename()   const;
    string_type   falsename()  const;
    static  locale::id  id;
protected:
     ~numpunct();
     virtual  char_type  do_decimal_point()  const;
    virtual char_type   do_thousands_sep() const;
    virtual string  do_grouping()  const;
    virtual string_type  do_truename()  const;
    virtual  string_type  do_false()  const;
    };
}
```

模板 numpunct<>指定了数值的标点符号。实例化类 numpunct<wchar_t>和 numpunct<char>，提供典型的“C”数值格式，即他们包含的信息等效于包含在 locale 类中和那些使用 widen()获取的宽字符。

数值格式的语法如下所示：digit 代表参数 fmtflags 确定的 radix 集合；使用刻面 ctype<charT>确定的空格（whitespace）；并且 thousands-sep 和 decimal-point 是相应的 numpunct<charT>成员的相应结果。整型值具有以下格式：

```
integer ::=[sign]units
sign   ::=plusminus [whitespace]
plusminus ::='+'|'-'
units ::=digits[thousands-sep  units]
digit ::=digit[digits]
```

浮点类型的值具有以下特点：

```
    floatval ::=[sign] units [decimal-point[digits]][e[sign]digits]|[sign]decimal-point
digits [e[sign]digits]
    e ::='e'|'E'
```

thousands-seps 决定的数字的数目是由 do_grouping()决定的。对于数值分解，如果数字部分不包

含 thousands-separators，则没有分组常量被应用。

numpunct 类的成员：

```
char_type  decimal_point()  const;
```

函数返回值是 do_decimal_point()。函数返回一个元素，该元素作为十进制小数点。

```
char_type  thousands_sep() const;
```

函数返回值是一个实例化 locale 类元素，就像作为一个 thousands separator。

```
string_type  truename() const;
string_type  falsename() const;
```

函数的返回值是一个字符串，作为一个其值为 true 的文本表达式。

```
char_type  do_decimal_point() const;
```

函数的返回值是一个字符，用作十进制基数分隔符。必要的实例化返回 ‘.’ 或 L ‘.’。

```
char_type  do_thousands_sep() const;
```

函数返回一个字符，用作数字组的分隔符。必要的实例化返回 ‘,’ 或 L ‘,’。

```
string  do_grouping()
```

函数返回一个字符串（basic_string<char>vec）。该字符串可作为一个整型值的向量。向量的每个元素代表数字数目。规则是：判断任何十进制小数点左边有多少个数字被划分为组。起始位置 0 是最右边的组。如果 vec.size()<=i，数字是同一组。如果(i<0)||vec[i]<=0||vec[i]==CHAR_MAX)，数字组的大小是无限制的。

必要的实例化返回空字符串，表示没有分组。

```
string_type  do_truename()  const;
string_type  do_falsename()  const;
```

函数返回值是字符串表达式，其内容为“true”和“false”两种形式，或 L“true”和 L“false”。

2. 类模板 numpunct_byname

```
namespace std{
template<class chart> class numpunct_byname: public numpunct<chart>{
public:
    typedef  charT  char_type;
    typedef  basic_string<charT>  string_type;
    explicit  numpunct_byname(const char*, size_t refs=0);
protected:
    ~numpunct_byname();
    virtual char_type  do_decimal_point() const;
    virtual char_type  do_thousands_sep() const;
    virtual  string  do_grouping()  const;
    virtual  string_type  do_truename()  const;
    virtual  string_type  do_falsename()  const;
    };
}
```

例 12-10 用于说明 numeric punctuation 的使用方法。

例 12-10

```
#include <locale>
#include <iostream>
#include <sstream>
using namespace std;
int main( )
  {
    locale loc( "English" );
    const numpunct < char> &npunct = use_facet <numpunct <char> >(loc);   //创建刻面
    cout << loc.name() << " truename() is called. "<< npunct.truename() << endl;
    cout << loc.name() << " falsename() is called. "<< npunct.falsename() << endl;
    const numpunct <char> &npunct2 =use_facet <numpunct <char> >(loc);
    //cout<<npunct.grouping()<<endl;
    for (unsigned int i=0;i< npunct.grouping( ).length();i++)
    {
    cout << loc.name() << " international grouping:\n the "<< i <<"th group to the left of the 
radix character "
       << "is of size " << (int)(npunct.grouping ()[i])<< endl;  //输出 locale 的名称
    }
    cout << loc.name() << " decimal point "<<npunct.decimal_point() << endl;
    cout << loc.name( ) << " thousands separator "<< npunct.thousands_sep() << endl;
}
```

程序的执行结果如下：

```
English_United States.1252 truename() is called. true
English_United States.1252 falsename() is called. false
English_United States.1252 international grouping:
 the 0th group to the left of the radix character is of size 3
English_United States.1252 decimal point .
English_United States.1252 thousands separator ,
```

12.4.4　类 collate

模板类 collate 的声明如下：

```
namespace  std{
    template <class charT> class collate: public locale::facet{
   public:
       typedef  charT  char_type;
       typedef  basic_string<charT>string_type;
       explicit collate(size_t refs=0);
       int compare(const charT* low1, const charT* high1, const charT* low2, const charT*
high2)const;
       string_type transform(const charT* low, const charT* high)const;
       long hash(const charT*  low,  const  charT* high)const;
       static  locale::id  id;
   protected:
       ~collate();
       virtual  int do_compare(const  charT* low1,  const charT*  high1,
  const charT*  low2,  const charT*  high2) const;
       virtual string_type  do_transform(const  charT* low, const charT* high) const;
       virtual long do_hash(const  charT* low,  const  charT*  high) const;
   };
}
```

类 collate<charT>提供使用 collation 和 hashing 处理字符串的特性。本地成员函数模板 operator()使用 collate 刻面，目的是允许一个 locale 对象直接作为处理字符串的标准算法和容器的预测参数。实例化类 collate<char>和 collate<wchar_t>可以应用于字典式（lexicographic）排序。

1．collate 类的成员

```
int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2)
const;
```

函数的作用是：按照各自刻面定义的规则，比较两个字符串是否相等。参数(low1,high1)是参与比较的第一个序列；参数(low2, high2)是参与比较的第二个序列。函数返回值一般有 3 种可能：-1, 0, 1。如果第一个序列小于第二个序列，函数返回-1；如果第一个序列等于第二个序列，函数返回 0；如果第一个序列大于第二个序列，函数返回 1。

```
string_type transform(const charT* low, const charT* high) const;
```

函数的作用：一个字符序列从 locale 类对象到字符串 string 的转换可能被用于字典式和同一 locale 类的另一字符序列的比较。参数 low 和参数 high 代表已转换序列的起始字符。函数的返回值包含了被转换的字符序列。

```
long hash(const charT* low, const charT* high) const;
```

函数 hash()会按照刻面的规则，判断序列的 hash 值。参数 low 和参数 high 代表输入序列的起始位置和结束位置。

2．collate 类的虚函数

```
int do_compare(const charT* low1, const charT* high1, const charT* low2, const
charT* high2)const;
```

如果第一个字符串大于第二个字符串，函数返回 1；如果小于，函数返回-1；如果相等，函数返回 0。实例化类 collate<char>和 collate<wchar_t>可以实施字典式排序。

```
string_type do_transform(const charT* low, const charT* high) const;
```

函数返回值类型是 basic_string<charT>。比较方式是字典式比较。

```
long do_hash(const charT* low, const charT* high) const;
```

函数返回值是整型值，其值等于调用 hash()的结果。

3．模板类 collate_byname

```
namespace std{
template <class charT>class collate_byname: public collate<charT>{
public:
        typedef basic_string<charT>string_type;
        explicit collate_byname(const char* , size_t refs=0);
protected:
        ~collate_byname();
virtual int do_compare(const charT* low1, const charT* high1, const charT* low2, const
chart* high2)const;
virtual string_type do_transform(const charT* low, const charT* high) const;
virtual long do_hash(const charT* low, const charT* high) const;
    }
}
```

例 12-11 展示了 std::collate 的用法。代码首先创建了 name 为“German_germany”的 locale，并定义了两个_TCHAR 的字符串，然后使用 std::collate 的 compare 函数来比较两个字符串的大小。

例 12-11

```
#include <locale>
#include <iostream>
#include <tchar.h>
using namespace std;
void main() {
    locale loc ( "German_germany" );          //创建 locale 对象
    _TCHAR * s1 = _T("Das ist wei\x00dfzz.");
    _TCHAR * s2 = _T("Das ist weizzz.");
    int result1 = use_facet<collate<_TCHAR>>(loc).compare(s1, &s1[_tcslen(s1)-1 ], s2,
&s2[_tcslen(s2)-1]);
//比较
    cout << result1 << endl;
    locale loc2 ("C");
    int result2 = use_facet<collate<_TCHAR>>(loc2).compare (s1, &s1[_tcslen(s1)-1 ],
s2,&s2[_tcslen(s2)-1]);
    cout << result2 << endl;
    long r1 = use_facet<collate<_TCHAR>>(loc).hash (s1, &s1[_tcslen(s1)-1]);    //hash
    long r2 = use_facet<collate<_TCHAR>> (loc).hash (s2, &s2[_tcslen(s2)-1]);
    cout << r1 << " " << r2 << endl;
    s2 = _T("zzz abc.");
    collate<_TCHAR>::string_type r11;
                              // OK for typedef?
    r11 = use_facet< collate<_TCHAR> > (loc).
transform  (s2, &s2[_tcslen( s2 )-1 ]);
                              //转换
    cout << r11<< endl;
    basic_string<_TCHAR> r22 = use_facet<collate
<_TCHAR>> (loc).transform (s2, &s2[_tcslen (s2)-1]);
    cout << r22<< endl;  }
```

例 12-11 的执行效果见图 12-1。

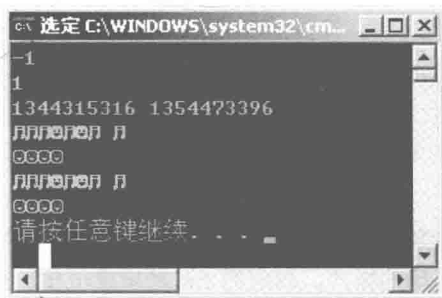


图 12-1　例 12-11 的执行效果

12.4.5　time 类（category）

模板 time_get<chart, InputIterator>和 time_put<chart, OutputIterator>提供了日期和时间的格式和分解方法。所有的类 time_get 和类 time_put 的成员函数均可应用于实例化。它们的成员使用他们的 ios_base&，ios_base::iostate&和 fill 参数，使用 ctype 类决定格式的细节。

1．类模板 time_get<>

```
namespace  std{
public:
enum dateorder{no_order, dmy, mdy, ymd, ydm};
template <class charT, class inputIterator=istreambuf_iterator<charT> >
    class  time_get:public  locale::facet, public time_base{
public:
    typedef  charT   char_type;
    typedef  InputIterator  iter_type;
```

```
        explicit  time_get(size_t refs=0);
        dateorder  date_order()  const{return do_date_order();}
        iter_type  get_time(iter_type  s, iter_type  end,  ios_base& f, ios_base::iostate& err, tm*
t)const;
        iter_type   get_date(iter_type   s,  iter_type   end,   ios_base& f,  ios_base::iostate&
err, tm* t) const;
        iter_type   get_weekday(iter_type   s,  iter_type   end,   ios_base& f,  ios_base::
iostate& err, tm* t) const;
        iter_type   get_monthname(iter_type   s,  iter_type   end,   ios_base& f,  ios_base::
iostate& err, tm* t) const;
        iter_type  get_year(iter_type  s, iter_type  end,  ios_base& f, ios_base::iostate& err, tm* t)
const;
           static  locale::id  id;
    protected:
        ~time_get();
        virtual  dateorder do_date_order() const;
        virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&, ios_base::
iostate& err, tm* t)const;
        virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&, ios_base::
iostate& err, tm* t)const;
        virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&, ios_base::
iostate& err, tm* t)const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&, ios_base::
iostate& err, tm* t)const;
        virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&, ios_base::
iostate& err, tm* t)const;
        }
    }
```

time_get()用于分解字符序列，抽出时间和日期的部件，并放入 tm 结构中。每一个 get 成员函数分解一种格式作为形成的相应的格式标识符，用于函数 time_put<>::put()中。如果被分解的序列匹配正确的格式，相应的 tm 结构成员参数被设置成用于产生序列的值；否则或者汇报一个错误，或者设定一个不确定的值。

（1）time_get<>的成员

```
    dateorder  date_order() const;
```

函数的作用是使用刻面返回日期的排序方式。

```
    iter_type get_time(iter_type  s,  iter_type  end,  ios_base& str, ios_base::iostate&
err, tm* t)const;
```

函数的作用是将表示时间的字符串放入 tm 结构中。

```
    iter_type  get_date(iter_type  s,  iter_type  end, ios_base& str, ios_base::iostate&
err, tm* t)const;
```

函数的作用是将表示日期的字符串放入 tm 结构中。

```
    iter_type     get_weekday(iter_type     s,     iter_type     end,   ios_base&    str,
ios_base::iostate& err, tm* t)const;
```

函数的作用是将表示星期的字符串放入 tm 结构中。

```
    iter_type get_year(iter_type  s,  iter_type  end, ios_base& str, ios_base::iostate& err,
tm* t)const;
```

函数的作用是将表示年的字符串放入 tm 结构中。

（2）time_get<>的虚函数

```
    dateorder  do_date_order(); const;
```

函数返回一个枚举值，该枚举值表明表达日期的组件的优先顺序。该日期格式主要包括：日期，月，年。

```
    iter_type do_get_time(iter_type s, iter_type end, ios_base* str, ios_base::iostate& err,
tm*t) const;
```

函数的作用：从[s, end]范围内获取字符串，并转换至 tm 结构中。函数返回迭代器指针，该指针指向最后一个字符前面。

```
    iter_type do_get_date(iter_type s, iter_type  end, ios_base& str, ios_base::iostate&
err, tm* t)const;
```

函数的作用：从指定的缓冲区抽取日期字符串，并将其内容转换至 tm 结构中。

```
    iter_type  do_get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate&
err, tm* t) const;
    iter_type  do_get_monthname(iter_type  s, iter_type end, ios_base& str, ios_base::
iostate& err, tm* t)const;
```

函数的作用是从缓冲区中抽取合适的日期字符串，并转换至 tm 结构中。

```
    iter_type  do_get_year(iter_type  s, iter_type end, ios_base& str, ios_base::iostate&
err, tm* t)const;
```

函数的作用是从缓冲区抽取合适的字符，并转换至 tm 结构中。

2．类模板 time_get_byname

类模板 time_get_byname 的声明如下：

```
    namespace  std{
       template<class charT, class InputIterator =istreambuf_iterator<charT>>
          class time_get_byname: public time_get<charT, inputIterator >{
    public:
       typedef  time_base::dateorder  dateorder;
       typedef  InputIterator  iter_type;
       explicit time_get_byname(const char*  size_t  refs=0);
    protected:
       ~time_get_byname();
       virtual  dateorder  do_date_order() const;
    virtual    iter_type     do_get_time(iter_type      s,    iter_type       end,       ios_base&,
ios_base::iostate& err,  tm* t)const;
    virtual iter_type  do_get_date(iter_type  s, iter_type  end,  ios_base&, ios_base::iostate&
err, tm* t)const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end,  ios_base&, ios_base::iostate& err,
       tm* t)const;
    virtual iter_type  do_get_monthname(iter_type  s, iter_type  end, ios_base&, ios_base::
iostate& err,
       tm* t)const;
```

```
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&, ios_base::iostate&
err, tm* t)const;
    }
    }
```

3. 类模板 time_put

```
    namespace std{
    template<class charT, class OutputIterator=ostreambuf_iterator<charT>>
       class time_put: public locale::facet{
       public:
          typedef charT char_type;
          typedef OutputIterator iter_type;
          explicit time_put(size_t refs=0);
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb, const charT*
pattern, const charT*
          pat_end)const;
          static locale::id id;
       protected:
          ~time_put();
       virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t, char format,
char modifier)const;
       };
    }
```

（1）time_put 类成员

```
    iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t, const charT*
pattern, const charT* pat_end) const;
```

函数的作用：第一种形式的函数，通过[pattern, pat_end]确定的序列，识别格式字符。格式字符序列的每一个字符被写入序列 s 中，并且识别出的每一个格式序列，均导致调用 do_put()。格式化元素和其他字符在顺序输出时，均是交叉存取的。按格式显示序列时，会转换字符 c 为 char 类型的值，类似调用 narrow()。此处 ct 是一个对于 ctype<charT>的引用，此引用是通过 str.getloc()获取的。每个序列的第一个字符等于“%”，并且格式标志符是通过函数 strftime()定义的。如果没有可变字符输出的话，mod 是 0。对于每个已识别的有效格式序列，需要调用 do_put(s, str, fill, t, spec, mod)。

第二种形式的 put()函数是通过调用 do_put(s,str,fill, t, format, modifier)来实现的。

提示：参数 fill 可以用于已定义的格式，或通过派生形式。一个空格字符是合理的默认填充符。函数返回值是一个迭代器，该迭代器指向最后一个字符之后。

（2）time_put 虚函数

```
    iter_type do_put(iter_type s, ios_base& , char_type fill, const tm* t, char format,
char modifier)const;
```

函数的作用：将参数 t 的内容按格式化参数 format 处理，并将之放在输出序列 s 中。格式是通过 format 和 modifier 来控制的。函数返回的迭代器指向最后一个字符之后。

4. 类模板 time_put_byname

类模板 time_put_byname 的声明如下：

```
    namespace std{
```

```
template <class charT, class OutputIterator=ostreambuf_iterator<charT> >
   class time_put_byname:pubic time_put<charT, OutputIterator>
{
public:
   typedef charT  char_type;
   typedef  OutputIterator  iter_type;
   explicit time_put_byname(const char* , size_t  refs=0);
protected:
   ~time_put_byname();
virtual iter_type do_put(iter_type  s,  ios_base& , char_type, const tm* t, char format,
char modifier) const;
   };
}
```

例 12-12 用来说明模板类 time category 中的 3 个类 time_base、time_get 和 time_put 的使用方法。

例 12-12

```
#include <locale>
#include <iostream>
#include <sstream>
#include <time.h>
using namespace std;
void po( char *p )
{
   locale loc( p );
   time_get <char>::dateorder order = use_facet <time_get <char> >( loc ).date_order ( );
                                                                     //获取日期格式
   cout << loc.name( );                                              //输出
   switch (order){
   case time_base::dmy: cout << "(day, month, year)" << endl;
      break;
   case time_base::mdy: cout << "(month, day, year)" << endl;
      break;
   case time_base::ydm: cout << "(year, day, month)" << endl;
      break;
   case time_base::ymd: cout << "(year, month, day)"<< endl;
      break;
   case time_base::no_order: cout << "(no_order)"<< endl;
      break;
   }
}
int main( )
{
   locale loc;
   basic_stringstream< char > pszGetF, pszPutF;
   ios_base::iostate st = 0;
   struct tm t;
   memset(&t, 0, sizeof(struct tm));
   pszGetF << "July 4, 2000";
   pszGetF.imbue( loc );                                             //使用 imbue
   basic_istream<char>::_Iter i = use_facet <time_get<char> >
      (loc).get_date(basic_istream<char>::_Iter(pszGetF.rdbuf( ) ),
      basic_istream<char>::_Iter(0), pszGetF, st, &t);               //获取日期
```

```
        if ( st & ios_base::failbit )
            cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
        else
            cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") ="          //输出日期
                << "\ntm_sec: " << t.tm_sec
                << "\ntm_min: " << t.tm_min
                << "\ntm_hour: " << t.tm_hour
                << "\ntm_mday: " << t.tm_mday
                << "\ntm_mon: " << (t.tm_mon+1)
                << "\ntm_year: " << t.tm_year
                << "\ntm_wday: " << t.tm_wday
                << "\ntm_yday: " << t.tm_yday
                << "\ntm_isdst: " << t.tm_isdst
                << endl;
        pszGetF.clear();
        pszGetF << "2000";
    i =use_facet<time_get<char>>(loc).get_year(basic_istream<char>::_Iter(pszGetF.rdbuf
()), basic_istream<char>
    ::_Iter(0), pszGetF, st, &t);                          //获取 年
        f (st & ios_base::failbit)
            cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: "
<< *i << endl;
        else
            cout << "time_get::get_year("<< pszGetF.rdbuf( )->str( )<< ") ="<< "\ntm_year:
" << t.tm_year<< endl;
        pszGetF << "July";
        pszGetF.imbue( loc );
    i = use_facet <time_get <char> >(loc).get_monthname(basic_istream<char>::_ Iter(pszGetF.
rdbuf( )), basic_ istream
    <char>::_Iter(0), pszGetF, st, &t);
        if (st & ios_base::failbit)
            cout << "time_get("<< pszGetF.rdbuf( )->str( )<< ") FAILED on char: " << *i << endl;
        else
            cout << (t.tm_mon+1)  <<"月"<< endl;
    //time
        basic_stringstream<char> pszGetF2;
        st = 0;                                                //非常重要的参数
        pszGetF << "11:13:20";
        pszGetF.imbue( loc );
    i = use_facet<time_get <char>>(loc).get_time(basic_istream<char>::_Iter(pszGetF.rdbuf()),
basic_ istream<char>
    ::_Iter(0), pszGetF2, st, &t);                         //获取时间
        if (st & ios_base::failbit)
            cout << "time_get::get_time("<< pszGetF2.rdbuf( )->str( )<< ") FAILED on char:
" << *i << endl;
        else
    cout << "time_get::get_time("<< pszGetF2.rdbuf( )->str( )<< ") ="<< "\ntm_sec: " <<
t.tm_sec<< "\ntm_min: "
    << t.tm_min<< "\ntm_hour: " << t.tm_hour<< endl;    //获取时间
        basic_stringstream<char> pszGetF3;
        pszGetF3 << "Tuesday";
        pszGetF3.imbue(loc);
    i = use_facet<time_get<char> >(loc).get_weekday(basic_istream<char>::_Iter(pszGetF3. rdbuf( )),
```

```
    basic_istream<char>::_Iter(0), pszGetF3, st, &t);          //获取周
        if (st & ios_base::failbit)
            cout << "time_get::get_time("<< pszGetF3.rdbuf( )->str( )<< ") FAILED on char:
" << *i << endl;
        else
            cout << "time_get::get_time("<< pszGetF3.rdbuf( )->str( )<< ") ="
            << "\ntm_weekday: " << t.tm_wday<< endl;
        po( "C" );
        po( "german" );
        po( "English_Britain" );
    }
```

程序的执行结果如下：

```
time_get(July 4, 2000) =
tm_sec: 0
tm_min: 0
tm_hour: 0
tm_mday: 4
tm_mon: 7
tm_year: 100
tm_wday: 0
tm_yday: 0
tm_isdst: 0
time_get::get_year(July 4, 20002000) =
tm_year: 100
7月
time_get::get_time() =
tm_sec: 20
tm_min: 13
tm_hour: 11
time_get::get_time(Tuesday) =
tm_weekday: 2
C(year, month, day)
German_Germany.1252(day, month, year)
English_United Kingdom.1252(day, month, year)
```

12.4.6　C 库 locale

头文件<clocale>中包含了以下内容。这些内容和头文件<locale.h>中的内容是一致的，见表 12-18。

表 12-18　头文件<clocale 概要>

Type	Name(s)		
Macros	LC_ALL	LC_COLLATE	LC_CTYPE
	LC_MONETARY	LC_NUMERIC	LC_TIME
	NULL		
Struct	lconv		
Functions	localeconv	setlocale	

12.5 本章小结

本章内容比较多。第 1 节简单描述了国际化问题和国际化元素；第 2 节讲述了多种字符编码；第 3 节讲述了类 locale，并深入分析了 locale 的刻面、区域等概念；第 4 节主要讲述标准 locale 的分类，涉及多种类，包括 ctype，numeric，collate，time，money，messages 等内容。第 5 节又针对上述提到的内容，分别详细讲述了各种刻面的使用。

第 13 章　仿函数

前面章节讲述算法时，读者会注意到很多 STL 算法使用了仿函数（函数对象）。仿函数也称为函数符。简单来说，函数符是以函数方式与括号()结合使用的任意对象。在 STL 的标准中，仿函数的英文名称是 FunctionObjects，即函数对象。更通俗地说，仿函数是将函数作为参数传递的使用方式，例如：逻辑谓词、算术运算、抽取信息等。

13.1　仿函数的概述

本章是本书的精华，而本节是本章的根本。仿函数（也称为函数符，function objection）的概念是学习仿函数的基础和根本。本节主要讲述了仿函数的概念和仿函数的作用。

13.1.1　仿函数的概念

通常可以这样理解，所谓仿函数是一个定义了 operator()的对象。仿函数可以视为一般函数，不同之处在于：仿函数的功能是在其成员函数 operator()中实现的。虽然仿函数的定义形式比较复杂，但仿函数的优点有 3 个方面：

（1）仿函数比一般函数更灵巧，主要是因为仿函数拥有状态（state）。对于仿函数，可以拥有两个状态不同的实体。而通常的普通函数是不能实现该目的的。

（2）每个仿函数都有其型别。因此仿函数的型别可以作为模板参数，实现指定某种行为的目的。容器型别不会和具体仿函数有关，而仅仅是该型别的仿函数均可使用。

（3）仿函数要比函数指针的执行速度快得多。在 C++规范中，函数调用通常使用指针，当需要调用函数时，只需要函数的地址（名称）即可。地址调用方法的缺陷是效率非常低，为提高效率需要使用仿函数的形式。定义的仿函数被调用时，是使用运算符 operator()的。通过对运算符进行自定义，能显著提高效率。

仿函数均是以类似函数的形式被调用的。仿函数均包含成员函数 operator()，这是仿函数的共同点。仿函数的基类是：

```
template<class _A, class _R> struct unary_function
   {  typedef _A Argument_Type;
      typedef _R Result_Type;
   };
```

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

这两个类的用途是为参数提供标准的名字，规定返回值类型，使程序员可以从这两个基类派生出相应的自己的类。

unary_function 是空的基类，类没有任何的成员函数和成员变量，仅仅包含型别信息。该基类是为了让 Adaptable Unary Function models 的定义更方便。Adaptable Unary Function 的 models 必须包含嵌套的型别声明，继承 unary_function 是获得嵌套型别的最简单办法。

同样，类 binary_function 也是一个空类，该类的存在使 Adaptable Binay Function models 的定义更方便。同样，Adaptable Binary Function 的任何 models 也必须包含嵌套型别声明。继承基类 binary_function 即获得这些类型定义的方便法门。

以上两个基类均声明于头文件<functional>中。

学习仿函数，要了解 3 个概念：生成器、一元函数和二元函数。

（1）生成器是不用参数就可以调用的仿函数，即无参数仿函数；

（2）一元函数是用一个参数就可以调用的仿函数；

（3）二元函数是用两个参数才可以调用的仿函数。

例如：对于算法 for_each()，其调用的仿函数应该是一元函数。

上述概念还有其他说法：

返回 bool 值的一元函数是 Predicate；

返回 bool 值的二元函数是 binary predicate（二元的 Predicate）。

部分 STL 的函数需要一元参数或二元参数。例如：算法 sort()需要使用二元的函数作为其调用的仿函数。

此外，仿函数还有型别，即仿函数的返回值。无参数的仿函数具有一个返回型别；Unary Function 具有两个返回型别；Binary Function 具有 3 个返回型别。仿函数被定义为“类”，可以拥有嵌套型别。Unary Function 和 Binary Function 两个类中仅仅具有一些类型声明（typedef）。

STL 还把仿函数和自适应仿函数区分开来。通常，一个仿函数会有一个引数型别与一个返回型别，程序不晓得那些型别的具体名称。自适应仿函数会指明引数与返回型别为何会内含嵌套的类型定义，程序中指定并使用那些型别。若型别 F1 是自适应一元仿函数（adaptable Unary Function）的 model，那么需要定义 F1::argument_type 和 F1：：result_type；若型别 F2 是自适应的二元仿函数（adaptable Binary Function），需要定义 F2::first_argument_type、F2::second_argument_type 和 F2::result_type。STL 的两个基类 unary_function 和 binary_function，可以使定义自适应的一元和二元仿函数简便得多。

13.1.2 仿函数的作用

程序开发人员在使用仿函数时，通常有 4 种使用形式：

（1）作为排序规则；

（2）拥有内部状态；

（3）算法 for_each()的返回值；

（4）作为判断式。

1. 仿函数作为排序规则

仿函数可以将已序的数据放入容器中。使用通常的运算符 operator<有时不能对这些元素排序，需要自定义特别的规则来实现排序的目的。仿函数的作用就在于此。详见例 13-1。

例 13-1

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <fstream>
using namespace std;
class person                                          //对象类
{
public:
    string firstname()
    {
        return fname;
    };
    string lastname()
    {
        return lname;
    };
public:
    string fname;
    string lname;
};
class person_sort_rule                                //排序规则类
{
public:
    bool operator()(person p1, person p2)
    {
        return (p1.lastname()<p2.lastname());
    };
};
string getfirstname(string buffer)                    //从缓冲区获取 firstname
{
    string r;
    size_t index=buffer.find(',');
    if (index!=-1)
        r=buffer.substr(0,index);
    else
        r="-1";
    return r;
}
string getlastname(string buffer)                     //从缓冲区获取 lastname
{
    string r;
    size_t index=buffer.find(',');
    if(index!=-1)
```

```
        r=buffer.substr(index+1,buffer.length()-1);
    else
        r="-1";
    return r;
}
void main()
{
    typedef set<person,person_sort_rule>personSet;
    personSet myset;
    myset.clear();
    string myfname,mylname;
    ifstream myfile;
    myfile.clear();                                          //删除该文件流的所有标志
    string filename;
    string buffer;
    cout<<"Please input the filename: "<<endl;
    cin>>filename;
    cout<<"Inputed filename : "<<filename<<endl;
    myfile.open(filename.c_str(),ios_base::in);    //打开文件
    if(myfile.is_open())
    {
        while(getline(myfile,buffer)&&buffer.size()>0)
        {
            myfname=getfirstname(buffer);
            mylname=getlastname(buffer);
            cout<<myfname<<", "<<mylname<<endl;
            if(myfname!="-1"&&mylname!="-1")            //将数据添加至 set 中
            {
                person temp;
                temp.fname=myfname;
                temp.lname=mylname;
                myset.insert(temp);
            }
        }
}
    myfile.clear();
    myfile.close();
    personSet::iterator it;
    for(it=myset.begin();it!=myset.end();it++)     //输出数据
    {
        cout<<(*it).fname<<" , "<<(*it).lname<<endl;
    }
}
```

程序的执行结果如下：

```
Please input the filename:
test.txt
Inputed filename : test.txt
Abra,  tom
kelinton, gulan
Crack, pack
grat,  jack
```

```
koom,  wert
grat ,  jack
Abra ,  tom
koom ,  wert
kelinton , gulan
Crack , pack
```

在例 13-1 中，使用了仿函数类 person_sort_rule。该类定义了函数 operator()。该函数对输入的数据的 lastname 字段进行比较。所有元素以此作为排序规则，进行排序。如果以一般的函数作为排序规则就比较困难。由于 set 具有自动排序性，这是不能避免的。但程序员可以设计自己的排序规则。

2. 仿函数可以拥有内部状态

仿函数还可以使程序在同一时刻拥有多个状态。函数传递数值时也有两种方式：通过值传递；通过引用传递。仿函数采用第一种方式：值传递（passed by value）。算法并不会改变随参数而来的仿函数的状态。仿函数参数采用“传递值”方式的好处是可以传递常量或临时表达式；缺陷是无法改变仿函数的状态。算法可以改变仿函数的状态，但无法存取或改变其最终状态，改变的仅仅是仿函数的副本。有时需要存取最终状态，关键是怎样从算法中获取结果。

从使用仿函数的算法中获取结果或反馈通常有两种方法：

（1）以引用的方式传递仿函数。若希望能够以引用的方式传递仿函数，需要在调用算法时，明确表示仿函数型别是引用型别。

（2）使用 for_each()算法的回返值。

下面以例 13-2 来说明此知识点。

例 13-2

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(list<int>& lt)
{
    list<int>::iterator it;
    for(it=lt.begin();it!=lt.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
class sequence
{
private:
    int value;
public:
    sequence(int ivalue):value(ivalue)
    {
    }
    int operator()()
    {
```

```
        return value++;
    }
};
void main()
{
    list<int> col;
    generate_n(back_inserter(col),9,sequence(1));
    print(col);
    generate(++col.begin(),--col.end(),sequence(42));
    print(col);
}
```

程序的执行结果如下：

```
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,
1 , 42 , 43 , 44 , 45 , 46 , 47 , 48 , 9 ,
```

本例是以仿函数产生一个整数序列。当仿函数的运算符 operator()被调用时，会返回整数值并累加 1。对于初始值，可通过构造函数的参数指定。在例 13-2 中还使用了两个函数 generate()和 generate_n()。两个函数的作用是：产生数值用以写入群集内。在使用仿函数 sequence()时，均使用了初始参数，该初始参数可作为产生序列的初始值。尤其是算法 generator_n()，会连续 n 次改写元素值，产生 n 个元素。举例说明如下：

```
sequence(42);
```

它会产生以 42 为起始值的整数序列。

通过修改 operator()，可以产生更加复杂的序列。

例如：

```
int operator()()
{
    //return value++;
    return ((value++)*2+1);
}
```

程序的执行结果如下：

```
3 , 5 , 7 , 9 , 11 , 13 , 15 , 17 , 19 ,
3 , 85 , 87 , 89 , 91 , 93 , 95 , 97 , 19 ,
```

为实现以引用的方式传递函数，可修改例 13-2 为例 13-3。

例 13-3

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
void print(list<int>& lt)
{
    list<int>::iterator it;
    for(it=lt.begin();it!=lt.end();it++)
    {
        cout<<*it<<",";
    }
    cout<<endl;
}
```

```
class sequence
{
private:
    int value;
public:
    sequence(int ivalue):value(ivalue)
{
    }
    int operator()()
    {
        return value++;
    }
};
void main()
{
    list<int> col;
    sequence seq(1);
    generate_n(back_inserter(col),4,seq);
    print(col);
    generate_n(back_inserter(col),4,sequence(42));
    print(col);
    generate_n(back_inserter(col),4,seq);
    print(col);
    generate_n(back_inserter(col),4,seq);
    print(col);
}
```

程序的执行结果如下：

```
1 , 2 , 3 , 4 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 , 1 , 2 , 3 , 4 ,
1 , 2 , 3 , 4 , 42 , 43 , 44 , 45 , 1 , 2 , 3 , 4 , 1 , 2 , 3 , 4 ,
```

3. 算法 for_each()的返回值

若使用算法 for_each()，不必实作仿函数的“引用计数版本”来存取最终状态。for_each()算法有其独门绝技，而其他算法是没有的，即可以返回其仿函数。通过算法 for_each()的回返值可以获取仿函数的状态，见例 13-4。

例 13-4

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class MeanV{
private:
    long num;
    long sum;
public:
    MeanV():num(0),sum(0)
    {

    }
    void operator()(int elem)
    {
        num++;
```

```
        sum+=elem;
    }
    double value()
    {
        return static_cast<double>(sum*1.0/num);
    }
};
void myprint(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
void main()
{
    vector<int>col;
    int array[8]={1,2,3,4,5,6,7,8};
    col.assign(array,array+8);
    myprint(col);
    MeanV mv=for_each(col.begin(),col.end(),MeanV());
    cout<<"Mean Value: "<<mv.value()<<endl;
}
```

程序的执行结果如下：

```
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 ,
Mean Value: 4.5
```

在语句 for_each(col.begin(),col.end(),MeanV())中，MeanV()产生一个反函数用于记录元素数量，并计算所有元素的总和。此仿函数传递给算法 for_each()，后者会针对容器内每个元素，调用仿函数。而返回的仿函数被赋值给 mv。调用仿函数的成员函数 value()，即可获取相应的平均值。

4. 判断式可返回仿函数的状态

所谓判断式，即返回布尔值的一个函数或仿函数。对于 STL 来说，并不是所有返回布尔值的函数都是合法的判断式。这样会导致很多意想不到的行为。详见例 13-5。

例 13-5

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
class Nth{
private:
    int nth;
    int count;
public:
    Nth(int n):nth(n),count(0)
    {
    }
bool operator()(int)
    {
```

```
        return ++count==nth;
        }
    };
    void myout(list<int>& lt)
    {
        list<int>::iterator it;
        for(it=lt.begin();it!=lt.end();it++)
        {
            cout<<*it<<" , ";
        }
        cout<<endl;
    }
    void main()
    {
        list<int> col;
        int array[9]={1,2,3,4,5,6,7,8,9};
        for(int i=1;i<9;i++)
            col.push_back(i);
        //col.assign(array,array+9);
        myout(col);
        list<int>::iterator pos;
        pos=remove_if(col.begin(),col.end(),Nth(3));
        col.erase(pos,col.end());
        myout(col);
    }
```

程序的执行结果如下：

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

例 13-5 中使用了一个仿函数 Nth()，当被重复调用时会返回 true。将其传递给算法 remove_if()时，此算法会将所有符合条件的元素删除。

13.2 预定义仿函数

STL 定义了许多基本仿函数。例如：其执行将两个值相加、比较两个值等操作。提供这些函数对象的是支持将函数作为参数的 STL 函数。常见的预定义仿函数见表 13-1。

表 13-1　预定义仿函数

仿　函　数	效　　果
negate<type>()	-param
plus<type>()	param1+param2
minus<type>()	param1-param2
multiplies<type>	param1*param2
divide<type>	param1/param2
modulus<type>()	param1%param2
equal_to<type>()	param1==param2
not_equal_to<type>()	param1!=param2
less<type>()	param1<param2

续表

仿 函 数	效 果
greater<type>()	param1>param2
less_equal<type()>	param1<param2
greater_equal<type>()	param1>param2
logical_not<type>()	!param
logical_and<type>()	Param1&&param2
logical_or<type>()	param1||param2

当使用仿函数对对象进行排序或比较时，通常都以 less<>作为预设的准则，所以 less<>经常被使用。预设的排序操作经常按升幂排序（element<nextElement）。

使用这些预定义的仿函数，必须包含头文件<functional>。为了实现对“国际化字符串”进行比较，C++标准程序库还提供了可以作为字符串排序准则的仿函数。

表 13-1 中的这些预定义仿函数均可在程序编写时任意使用。表中的仿函数可分为 3 类：算术运算、关系运算、逻辑运算。每类的仿函数既可以作为有名的仿函数使用，也可以作为无名的仿函数传递给其他函数。

13.3 辅助用仿函数

仿函数的组合能力很重要，可以从一些软件组件构造出另一些组件。最简单的仿函数可以构造出非常复杂的仿函数。一般而言，所有函数行为均可借由仿函数的组合而实现。C++标准程序库不能提供足够的配接器来达此境界。就理论而言，组合型配接器功能会更强大，用途会更加广泛。

（1）f(g(elem))

这是一元组合函数的最一般形式。一元判断式被嵌套调用，g()的执行结果作为 f()的参数。整个表达式的操作类似一个一元判断式

（2）f(g(lelem1, elem2))

两个元素 elem1 和 elem2 作为参数传递给二元判断式 g()。其结果作为参数传给一元判断式 f()。整个表达式类似一个二元判断式。

（3）f(g(elem), h(elem))

参数 elem 作为参数被传递给两个不同的一元判断式 g()和 h()，两者的结果由二元判断式 f()处理。此形式以某种方法将单一参数“注射”至组合函数中，整个表达式类似于一个一元判断式。

（4）f(g(elem1)，h(elem2))

此处参数 elem1 和 elem2 作为唯一参数传递给两个不同的一元判断式 g()和 h()，两个结果共同被二元判断式 f()处理。整个表达式类似二元判断式。

上述形式的配接器没有被写入标准中，也没有标准的名称。SGI STL 的实作版本中定义两个名称，目前 C++社群正寻找上述类型所有配接器的通用表达方式。上述几种形式见表 13-2。

表 13-2　组合型仿函数配接器的可能表现方式

功　　能	本 书 名 称	SGI STL 的名称
f(g(elem))	compose_f_gx	compose1
f(g(elem1, elem2))	compose_f_gxy	
f(g(elem), h(elem))	compose_f_gx_hx	compose2
f(g(elem1), h(elem2))	compose_f_gx_hy	

13.3.1　一元组合函数配接器

最基本的组合型函数配接器，是 SGI STL 实例版本的一部分。本小节首先以 compose_f_gx 进行嵌套计算来演示组合型仿函数的使用方法。最简单也是最基本的组合型仿函数配接器是将某一元运算结果作为另一个一元运算的输入。仅仅是嵌套调用两个一元仿函数。详见例 13-6。

例 13-6

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
template <class op1, class op2>class compose_f_gx_t: public std::unary_function<typename
      op2::argument_type, typename op1::result_type>
{
private:
   op1 myop1;
   op2 myop2;
public:
   compose_f_gx_t(const op1& o1, const op2& o2):myop1(o1),myop2(o2)
   {
   }
   typename op1::result_type operator() (const typename op2::argument_type& x) const
   {
      return myop1(myop2(x));
   }
};
template <class op1, class op2>inline compose_f_gx_t<op1,op2>compose_f_gx(const op1& o1,
   const op2& o2)
{
   return compose_f_gx_t<op1,op2>(o1,o2);
}
void myprint(vector<int>& vt)
{
   vector<int>::iterator it;
   for(it=vt.begin();it!=vt.end();it++)
      cout<<*it<<" , ";
   cout<<endl;
}
void main()
{
```

```
    vector<int>vt;
    int array[9]={1,2,3,4,5,6,7,8,9};
    vt.assign(array,array+9);
    myprint(vt);
        transform(vt.begin(),vt.end(),ostream_iterator<int>(cout," "),
            compose_f_gx(bind2nd(multiplies<int>(),5),bind2nd(plus<int>(),10)));
    cout<<endl;
}
```

程序的执行结果如下：

```
1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,
55 60 65 70 75 80 85 90 95
```

在例 13-6 中，当执行语句 transform()时，参数 compose_f_gx 的第二个参数优先被执行。即对于向量序列 vt，其中的每个元素均先增加 10，之后“和”再乘以 5。例如：对于第一个元素“1”，运算规则是：（1+10）*5=55。

下面以 compose_f_gx_hx 组合两个“运作规则”为例，使用例 13-7 来说明该规则的使用方法。

这可能是最重要的一个组合型函数配接器，它允许将两个准则加以逻辑组合，形成单一准则。在表 13-2 中，SGI STL 实作版本被称为 compose2。详见例 13-7。

例 13-7

```
#include <functional>
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
template <class OP1, class OP2, class OP3>class compose_f_gx_hx_t
            :public std::unary_function<typename OP2::argument_type,
             typename OP1::result_type>
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1& o1, const OP2& o2, const OP3& o3)
            :myop1(o1),myop2(o2),myop3(o3)
    {

    }
    typename OP1::result_type operator()(const typename OP2::argument_type& x)
    {
        return myop1(myop2(x),myop3(x));
    }
};
template <class OP1, class OP2, class OP3>inline compose_f_gx_hx_t<OP1,OP2,OP3>
    compose_f_gx_hx(const OP1& o1, const OP2& o2, const OP3& o3)
{
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}
void myprint(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
```

```
        cout<<*it<<" , ";
    cout<<endl;
}
void main()
{
    vector<int> vt;
    int array[9]={1,2,3,4,5,6,7,8,9};
    vt.assign(array,array+9);
    myprint(vt);
    vector<int>::iterator pos;
    pos=remove_if(vt.begin(),vt.end(),
  compose_f_gx_hx(logical_and<bool>(),bind2nd(greater<int>(),4),bind2nd(less<int>
(),7)));
    vt.erase(pos,vt.end());
    myprint(vt);
}
```

程序的执行结果如下：

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

在例 13-7 中，算法 remove_if()中第 3 个参数是由仿函数 compose_f_gx_hx()来实现的。此仿函数 compose_f_gx_hx()包含 3 个参数。对于例 13-7，该仿函数确定的条件是：大于 4 并且小于 7 的元素。所以，程序执行完毕之后，输出向量 vt，仅剩余 7 个元素了。向量中的元素“5”和“6”被删除。

13.3.2　二元组合函数配接器

二元组合函数配接器可以将两个一元运算的结果加以处理。例 13-8 是一种较简单的实例。

例 13-8

```
#include <functional>
#include <iostream>
#include <algorithm>
#include <string>
#include <cctype>
using namespace  std;
template <class OP1, class OP2, class OP3>
class compose_f_gx_hx_t:public std::binary_function<
        typename OP2::argument_type,
        typename OP3::argument_type,
        typename OP1::result_type>
{
private:
 OP1 myop1;
 OP2 myop2;
 OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1& o1, const OP2& o2, const OP3& o3):
         myop1(o1),myop2(o2),myop3(o3)
    {
    }
    typename OP1::result_type operator()(const typename OP2::argument_type& x,
        const typename OP3::argument_type& y)const
    {
        return myop1(myop2(x),myop3(y));
    }
```

```
};
template <class OP1, class OP2, class OP3>inline compose_f_gx_hx_t<OP1,OP2,OP3>
    compose_f_gx_hy(const OP1& o1, const OP2& o2, const OP3& o3)
{
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}
void main()
{
    string s("Internationalization");
    string sub("Nation");
    string ::iterator pos;
pos=search(s.begin(),s.end(),sub.begin(),sub.end(),compose_f_gx_hy(equal_to<int>(),
        ptr_fun(::toupper),ptr_fun(::toupper)));
    if(pos!=s.end())
    {
        cout<<"\""<<sub<<"\" is part of \""<<s<<"\""<<endl;
    }
}
```

程序的执行结果如下：

```
"Nation" is part of "Internationalization"
```

上述例题完成了在给定字符串中搜索子串的功能。

13.4 关系仿函数

预定义的关系函数对象包括：等于，不等于，大于，大于等于，小于和小于等于。在 C/C++中，大量使用关系运算符，而本节主要讲述关系仿函数，它可以是一元参数也可以是二元参数。对于常用的关系运算，使用仿函数关系运算即可将其应用于处理容器元素的过程中。

13.4.1 等于（equal_to<type>()）

类 eaual_to<type>是一种自适应一元 Predicate（Adaptable Binary Predicate）。该仿函数可以用于验证某条件的真伪。若 f 是类 equal_to<T>的对象，并且 x 与 y 为型别 T 之值，仅仅在 x==y 时，f(x,y)才可以返回 true。

仿函数 equal_to 使用时必须包含头文件<functional>。参数 type 是仿函数参数的型别。其基类为

```
binary_function<T,T,bool>
```

仿函数类 equal_to 包含了 5 个成员：

```
equal_to::first_argument_type
```

代表第一个参数的型别：T。

```
equal_to::second_argument_type
```

代表第二个参数的型别：T。

```
equal_to::result_type
```

代表执行结果的型别：bool。

```
bool equal_to::operator()(const T& x, const T& y) const
```

函数调用操作符。返回值为 x==y。

```
equal_to::equal_to()
```

代表仿函数的默认构造器。

使用示例：

```
equal_to<string> stringEqual;
sres=stringequal(sval1, sval2);
ires=count_if(svec.begin(), svec.end(), equal_to<string>(), sval1);
```

下面给出最简单的例 13-9，用以说明仿函数 equal_to 的使用方法。

例 13-9

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
void main()
{
    const int N=10;
    int A[N]={1,3,0,2,5,9,0,0,6,0};
    partition(A,A+N,bind2nd(equal_to<int>(),0));                //划分
    copy(A,A+N,ostream_iterator<int>(cout," "));                //复制
    cout<<endl;
}
```

程序的执行结果如下：

```
0 0 0 0 5 9 2 3 6 1
```

例 13-9 的作用是将序列中等于零的元素放置在序列的前面。

13.4.2　不等于（not_equal_to<type>()）

仿函数类 not_equal_to<T>也是一种 Adaptable Binary Predicate，此仿函数可用于验证某条件之真伪。若 f 是类 not_eaual_to<T>的一个对象，并且 x 和 y 均为型别 T 的值，仅仅在 x!=y 时，函数 f(x,y)才会返回 true。

仿函数 equal_to<T>的参数型别和基类与仿函数 equal_to 近似相同。该类也同样包含 5 个成员。这 5 个成员分别是：

```
not_equal_to::first_argument_type
```

代表第一个参数的型别。

```
not_equal_to::second_argument_type
```

代表第二个参数的型别。

```
not_equal_to::result_type
```

代表结果的型别：bool。

```
bool not_equal_to::operator()(const T& x, const T& y)const
```

代表函数调用的操作符。返回值为：x!=y。

```
not_equal_to::not_equal_to()
```

代表该仿函数的默认构造器。

例 13-10 展示了 not_equal_to 的具体用法。代码中首先声明并初始化了一个 std::list<int>实例，然后使用 not_equal_to<int>类作为参数调用 bind2nd 函数，从而生成新的函数对象，调用该函数对象的值不为 0，最后使用 find_if 模板函数找到第一个不为 0 的数字，并调用 std::copy 到 std::cout 中。

例 13-10

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <list>
using namespace std;
void main()
{
   const int N=9;
   int A[N]={0,0,0,0,1,2,4,8,0};
   list<int> L(A,A+N);
   list<int>::iterator i=find_if(L.begin(),L.end(),bind2nd(not_equal_to<int>(),0)) ;
//查找
   cout<<"Elements after initial zeros(if any): ";
   copy(i,L.end(),ostream_iterator<int>(cout," "));
//复制
   cout<<endl;
}
```

程序的执行结果如下：

```
Elements after initial zeros(if any): 1 2 4 8 0
```

13.4.3 小于（less<type>()）

仿函数 less<type>是一种 adaptable binary predicate ，该仿函数可以用来验证某条件的真伪。若 f 是仿函数的一个对象，并且 x 和 y 均是型别 type 的值，只有当 x<y 时，f(x,y)才可以返回 true。

STL 的许多仿函数和算法均需要比较函数。例如：sort、set、map。less 是典型的默认值。此仿函数的基类是 binary_function<type, type, bool>。仿函数类 less 包含了 5 个成员：

```
less::first_argument_type
```

代表第一个参数的型别：type。

```
less::second_argument_type
```

代表第二个参数的型别：type。

```
less::result_type
```

代表执行结果的型别：bool。

```
bool less::operator()(const T& x, const T& y) const
```

代表函数调用的操作符，其返回值为：（x<y）的值。

```
less::less()
```

代表默认的构造函数。

例 13-11 展示了如何使用 std::less 仿函数。代码首先初始化一个 int 数组，然后调用 std::partition 模板函数来将该数组找到小于 0 的部分，最后输出。

例 13-11

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
void main()
{
    const int N=10;
    int A[N]={1,-3,-7,2,5,-9,-2,1,6,-8};
    partition(A,A+N,bind2nd(less<int>(),0));        //划分
    copy(A,A+N,ostream_iterator<int>(cout," "));
    cout<<endl;
}
```

程序的执行结果如下：

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

13.4.4 大于（greater<type>()）

仿函数 greater<type>是一种自适应二元参数 Predicate，可以用其来验证条件之真伪。若 f 是仿函数的对象，并且 x 与 y 均是型别 type 的值，只有当 x>y 时，f(x,y)才能返回 true。此仿函数同样包含 5 个类成员：

```
greater::first_argument_type
```

代表第一个参数的型别：type。

```
greater::second_argument_type
```

代表第二个参数的型别：type。

```
greater::result_type
```

代表执行结果的型别：bool。

```
bool greater::operator()(const T& x, const T& y) const
```

代表函数调用的操作符。其返回值为 x>y。

```
greater::greater()
```

代表仿函数的默认构造函数。

13.4.5 大于等于（greater_equal）和小于等于（less_equal）

仿函数“大于等于”（greater_equal<type>()）和“小于等于”（less_equal<type>()）和以上的仿函数基本相似。不再细述。

例 13-12 展示了如何使用 std::greater 仿函数和 std::less_equal 仿函数。代码中首先声明并初始化一个 std::vector<int>，然后以 std::greater<int>为仿函数调用 std::sort 对数组进行排序并输出；接下来以 std::less_equal<int>为仿函数调用 std::remove_copy_if 将小于等于-7 的元素删除并输出；接下来找到大于等于 6 的元素并输出。

例 13-12

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

```
#include <algorithm>
#include <vector>
using namespace std;
void main()
{
    const int N=10;
    int A[N]={1,-3,-7,2,5,-9,-2,1,6,-8};
    vector<int>V(A,A+N);
    vector<int>v2;
    sort(V.begin(),V.end(),greater<int>());    //排序
    copy(V.begin(),V.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    remove_copy_if(V.begin(),V.end(),back_inserter(v2),bind2nd(less_equal<int>(),-7));
    //大于等于
    cout<<"Element remained : ";
    copy(v2.begin(),v2.end(),ostream_iterator<int>(cout," "));
    cout<<endl;
    vector<int>::iterator i=find_if(V.begin(),V.end(),bind2nd(greater_equal<int>(),6));
    cout<<*i<<endl;
}
```

程序的执行结果如下：

```
6 5 2 1 1 -2 -3 -7 -8 -9
Element remained : 6 5 2 1 1 -2 -3
6
```

13.5 逻辑仿函数

在讲述逻辑仿函数之前，首先介绍“谓词”的概念。根据传统语言学理论，汉语的谓词包括动词、形容词和一部分指代词。由此可见，谓词是用来描述或判定客体性质、特征或者客体之间关系的词项。

13.5.1 谓词

返回 bool 类型的仿函数称为谓词。前面讲述的关系仿函数和本节讲述的逻辑仿函数，都属于谓词。在头文件<functional>中，包含以下两个仿函数：

```
template <class T> struct logical_not: public unary_function<T bool>{
 bool operator()(const T& x)const
    {
        return !x;
    }
}
template <class T> struct less: public binary_function<T, T, bool>{
 bool operator()(const T& x, const T& y) const
    {
        return x<y;
    }
}
```

一元谓词和二元谓词在算法中是非常有用的。尤其是在比较两个序列时，寻找某个序列中的第一个不小于在另一个序列里的对应元素的元素。

```
void f(vector<int> & vi, list<int>& li)
    {
        typedef list<int>::iterator LI;
        typedef  vector<int>::iterator VI;
        pair<VI,LI> pl=mismatch(vi.begin(), vi.end(), li.begin(),less<int>());
        ...
    }
```

算法 mismatch()会反复地将其二元谓词作用于各对应元素，直到比较失败为止。使用 less<int>()，而不是使用 less<int>，因为此处需要的是一个对象，而不是型别（数据类型）。若需要的不是第一个不小于另一个序列中的对应元素的元素，寻找的是第一个比对应元素小的元素，可以使用 mismatch 寻找第一个不小于的对。

标准库提供的常用谓词见表 13-3。

表 13-3　常用谓词

谓　词		
equal_to	二元	arg1==arg2
not_equal_to	二元	arg1!=arg2
greater	二元	arg1>arg2
less	二元	arg1<arg2
greater_equal	二元	arg1>=arg2
less_equal	二元	arg1<=arg2
logical_and	二元	arg1&&arg2
logical_or	二元	arg1||arg2
logical_not	一元	!arg

除了标准库提供的常用谓词之外，程序员还可以自己编写自定义谓词。自定义谓词对于使用标准库和标准算法是非常有利的。尤其是在使用标准库算法时，自定义的数据结构类型标准算法有时无法识别，此时只好使用自定义的仿函数解决问题。

13.5.2　逻辑仿函数

逻辑仿函数支持逻辑与、逻辑或、逻辑非。逻辑运算和算术运算非常相似，执行的莫过于“或”和“与”等操作。逻辑操作本身并不重要，其主要用途在于和仿函数适配器相结合，使用仿函数时便可以执行逻辑操作。

1. 逻辑与

```
logical_and<T>
```

仿函数 logical_and<T>是一种自适应二元 Predicate。可以用来表示验证条件的真伪。若 f 是仿函数类 logical_and<T>的对象，并且 x 与 y 均是型别 T 的数值，而 T 可转换为 bool，仅仅当 x 和 y 都为 true 时，f(x,y)才会返回 true。

仿函数逻辑“与”在使用时需要包含头文件<functional>。型别 T 还可以是 bool 类型。该仿函数的基类是 binary_function<T, T, bool>。

该仿函数包含 5 个成员：

```
logical_and::first_argument_type
```

代表第一个参数的型别。

```
logical_and::second_argument_type
```

代表第二个参数的型别。

```
logical_and::result_type
```

代表执行结果的型别：bool。

```
bool logical_and::operator()(const T& x, const T& y )const
```

代表函数调用的操作符，返回值为：x&&y。

```
logical_and::logical_and()
```

代表仿函数的构造函数。

例 13-13 展示了如何使用仿函数 std::logic_and<T>的用法。代码中定义了仿函数 compose_f_gx_hx_t，继承自 std::unary_function，其构造函数是输入的 3 个函数 op1,op2,op3，然后实现 op1(op2(x),op3(x)) 的函数组合。代码中首先使用 std::generate_n 产生了 1 000 个小于 1 000 的随机数，并插入 list 中；接下来调用 std::find_if 实现查找>=1 并<=100 的数，返回迭代器，最后输出该数。

例 13-13

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <list>
using namespace  std;
template <class OP1, class OP2, class OP3>class compose_f_gx_hx_t
:public std::unary_function<typename OP2::argument_type,
typename OP1::result_type>
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1& o1, const OP2& o2, const OP3& o3)
        :myop1(o1),myop2(o2),myop3(o3)
    {
    }
    typename OP1::result_type operator()(const typename OP2::argument_type& x)
    {
        return myop1(myop2(x),myop3(x));
    }
};
template <class OP1, class OP2, class OP3>inline compose_f_gx_hx_t<OP1,OP2,OP3>
compose_f_gx_hx(const OP1& o1, const OP2& o2, const OP3& o3)
{
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}
void myprint(list<int>& l)                          //输出列表 list
{
    list<int>::iterator it;
    for(it=l.begin();it!=l.end();it++)
```

```
        cout<<*it<<" , ";
    cout<<endl;
}

void main()
{
    list<int> L;
    generate_n(back_inserter(L),1000,rand);    //产生随机数
    //myprint(L);
    list<int>::iterator i=find_if(L.begin(),L.end(),compose_f_gx_hx(logical_and
<bool>(),
                                                bind2nd(greater_equal<int>(),1),
                                                bind2nd(less_equal<int>(),100)));
                                                //查找
    assert(i==L.end()||(*i>1&&*i<=100);        //断言
    cout<<*i<<endl;
}
```

程序的执行结果如下：

```
41
```

2. 逻辑或

仿函数逻辑“或”是一种自适应二元 Predicate。该仿函数可用于验证某条件的真伪。若 f 是类 logical_or<T>的对象，并且 x 和 y 均是型别 T 的数值，型别 T 还可以是 bool 类型。只有当 x 或 y 两者之中至少有一个为 true 时，才会返回 true。仿函数“逻辑或”使用时，同样需要包含头文件<functional>。该仿函数的基类是 binary_function<T , T , bool>。该仿函数包含 5 个成员函数：

```
logical_or::first_argument_type
```

代表第一个参数的型别。

```
logical_or::second_argument_type
```

代表第二个参数的型别。

```
logical_or::result_type
```

代表执行结果的型别：bool。

```
bool logical_or::operator()(const T&x, const T&y) const
```

代表函数调用的操作符。其返回值为：x||y。

```
logical_or::logical_or()
```

代表仿函数的构造函数。

例 13-14 与上面的例子有类似之处，其中定义的 compose 仿函数同上例。不同的是这段代码展现了如何在字符串中寻找空格或者换行符号，使用 std::logical_or 仿函数来实现或的功能。

例 13-14

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <list>
#include <assert.h>
```

```
using namespace  std;
template <class OP1, class OP2, class OP3>class compose_f_gx_hx_t
:public std::unary_function<typename OP2::argument_type,
typename OP1::result_type>
{
private:
    OP1 myop1;
    OP2 myop2;
    OP3 myop3;
public:
    compose_f_gx_hx_t(const OP1& o1, const OP2& o2, const OP3& o3)
        :myop1(o1),myop2(o2),myop3(o3)
    {
    }
    typename OP1::result_type operator()(const typename OP2::argument_type& x)
    {
        return myop1(myop2(x),myop3(x));
    }
};
template <class OP1, class OP2, class OP3>inline compose_f_gx_hx_t<OP1,OP2,OP3>
compose_f_gx_hx(const OP1& o1, const OP2& o2, const OP3& o3)
{
    return compose_f_gx_hx_t<OP1,OP2,OP3>(o1,o2,o3);
}
void myprint(list<int>& l)
{
    list<int>::iterator it;
    for(it=l.begin();it!=l.end();it++)
        cout<<*it<<" , ";
 cout<<endl;
}
void main()
{
char str[]="The first line\nThe second line";
int len=strlen(str);
const char* wptr=find_if(str,str+len,compose_f_gx_hx(logical_or<bool>(),
                                bind2nd(equal_to<char>(),' '),
                                bind2nd(equal_to<char>(),'\n')));
    assert(wptr==str+len||*wptr==' '||*wptr=='\n');
    cout<<wptr<<endl;
}
```

程序的执行结果如下：

```
first line
The second line
```

3. 逻辑非

仿函数逻辑非：

```
logical_not<T>
```

仿函数类 logical_not<T>是一种自适应 Predicate。可以用于验证某条件的真伪。仿函数“逻辑非”仅仅需要一个参数。若 f 是仿函数“逻辑非”（logical_not<T>）的对象，并且 x 是型别 T 的数值；T 可以是作为 bool 类型的，仅仅当 x 为 false 时，f(x)才会返回 true。

仿函数“逻辑非”的基类是：

```
unary_function<T,bool>
```

仿函数类“逻辑非”包含 4 个成员：

```
logic_not::argument_type
```

代表参数的型别 T。

```
logic_not::result_type
```

代表结果的型别：bool。

```
bool logic_not::operator()(const T& x) const
```

代表函数调用的操作符，返回值为!x。

```
logical_not::logical_not()
```

代表仿函数的构造函数。

例 13-15 展示了 std::logical_not 的用法，实现了反转一个 vector<bool>的功能。代码首先初始化一个随机 bool 值的 vector，然后调用 std::transform 在该 vector 上进行变换，具体的变化规则是 std::logicl_not，也就是在取反。

例 13-15

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
#include <assert.h>
using namespace std;
void myprint(vector<bool>& vb)
{
    vector<bool>::iterator it;
    for(it=vb.begin();it!=vb.end();it++)
        cout<<*it<<" , ";
    cout<<endl;
}
void main(int argc, char* argv[])
{
    const int N=10;
    vector<bool> v1;
    for(int i=0;i<N;++i)
    {
        v1.push_back(rand()>(RAND_MAX/2));
    }
    cout<<"vector<bool> v1 :"<<endl;
    myprint(v1);
    vector<bool> v2;
    transform(v1.begin(),v1.end(),back_inserter(v2),logical_not<bool>()); //转换
    cout<<"vector<bool> v2 :"<<endl;
    myprint(v2);
    for(int i=0; i<N; ++i)
        assert(v1[i]==!v2[i]);
}
```

程序的执行结果如下：

```
vector<bool> v1 :
0 , 1 , 0 , 1 , 1 , 0 , 0 , 1 , 1 , 1 ,
vector<bool> v2 :
1 , 0 , 1 , 0 , 0 , 1 , 1 , 0 , 0 , 0 ,
```

13.6 算术仿函数

预定义的算术仿函数对象包含：加（plus）、减（minus）、乘（multiplies）、除（divide）、求余（modulus）和取反（negate），见表 13-4。调用的操作符是和 type 相关联的实例。对于一个 class 类型，若提供该操作符的重载实例，即可调用该实例。尤其在处理数值类时，把标准的算数函数作为函数使用是非常有用的。

表 13-4　算术运算仿函数

算术运算仿函数		
plus	二元	arg1+arg2
minus	二元	arg1−arg2
multiplies	二元	arg1*arg2
divide	二元	arg1/arg2
modulus	二元	arg1%arg2
negate	一元	−arg

13.6.1 加减乘除运算仿函数（plus）

仿函数 plus<T>是自适应二元仿函数。若 f 是仿函数的对象，并且 x 和 y 均为型别 T 的值，则 f(x,y) 会返回 x+y。该仿函数使用时需要包含头文件<functional>。仿函数的基类是 binary_function<T,T,T>。

仿函数类 plus<T>包含 5 个成员：

```
plus::first_argument_type
```

代表仿函数第一个参数的型别 T。

```
plus::second_argument_type
```

代表仿函数第二个参数的型别 T。

```
plus::result_type
```

代表仿函数的执行结果型别 T。

```
T plus::operator()(const T& x, const T& y) const
```

代表函数调用的操作符。执行结果是：x+y。

```
plus::plus()
```

代表函数调用的默认构造函数。

减法仿函数（minus<T>()）、乘法仿函数（multipies<T>()）、除法仿函数（divides<T>()）和加法仿函数的情况大体相同。

例 13-16 展示了如何使用加法仿函数 std::minus<T>，乘法 std::multipies<T>和除法 std::divides<T>。

代码首先随机生成了 vector v1, v2，然后依次进行 v1+v2，v1*v2，分别用到了加法和乘法仿函数。接下来使用乘法和除法仿函数进行了新的转换操作，并输出结果，代码比较简单。

例 13-16

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

using namespace std;
void myprint(vector<double>& vd)
{
    vector<double>::iterator it;
    for(it=vd.begin();it!=vd.end();it++)
        cout<<*it<<" , ";
    cout<<endl;
}
void main(int argc, char* argv[])
{
    int N=5;
    vector<double> v1(N);
    vector<double> v2(N);
    vector<double> v3(N);
    generate(v1.begin(),v1.end(),rand);
    fill(v2.begin(),v2.end(),-RAND_MAX/2.0);
    cout<<"vector v1: "<<endl;
    myprint(v1);
    cout<<"vector v2:"<<endl;
    myprint(v2);
    transform(v1.begin(),v1.end(),v2.begin(),v3.begin(),plus<double>());   //转换
    cout<<"vector v3(plus):"<<endl;
    myprint(v3);
    transform(v1.begin(),v1.end(),v2.begin(),v3.begin(),minus<double>());  //转换
    cout<<"vector v3(minus):"<<endl;
    myprint(v3);
    N=8;
    vector<double> v4(N);
    for(int i=0;i<N;++i)
        v4[i]=i+1;
    partial_sum(v4.begin(),v4.end(),v4.begin(),multiplies<double>());  //部分求和
    copy(v4.begin(),v4.end(),ostream_iterator<double>(cout,"\n"));       //复制
    cout<<endl;
    cout<<"vector v4(multiplies):"<<endl;
    myprint(v4);
    N=6;
    vector<double>v5(N);
    generate(v5.begin(),v5.end(),rand);
    transform(v5.begin(),v5.end(),v5.begin(),bind2nd(divides<double>(),155.0));
    //转换
    cout<<"vector v5(divides):"<<endl;
    myprint(v5);
}
```

例 13-16 的执行效果见图 13-1。

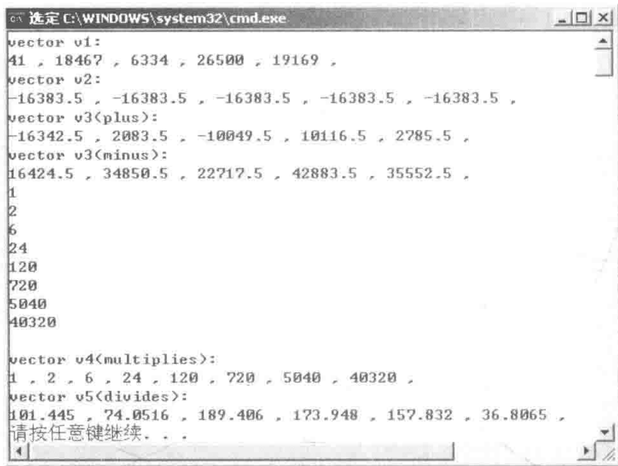


图 13-1 例 13-16 的执行效果

13.6.2 “求余”仿函数和“求反”仿函数

1. 求余仿函数（modulus<T>）

仿函数 modulus<T>是一种自适应二元仿函数（adaptable binary function）。若 f 是仿函数的一个对象，并且 x 和 y 均属于型别 T，则 f(x,y)会返回 x%y。求余仿函数包含 5 个成员：

```
modulus::first_argument_type
```

代表仿函数第一个参数的型别。

```
modulus::second_argument_type
```

代表仿函数第二个参数的型别。

```
modulus::result_type
```

代表执行结果的型别 T。

```
T modulus::operator()(const T& x, const T& y)const
```

函数调用的操作符。执行结果是 x%y。

```
modulus::modulus()
```

代表仿函数的构造函数。

2. 求反仿函数（negate<T>）

仿函数 negate<T>是自适应二元仿函数。是单参数的仿函数。若 f 是仿函数的对象，并且 x 属于型别 T，则 f(x)会返回（-x）。求反仿函数的基类是 unary_function<T,T>。

该仿函数类包括 4 个成员函数：

negate::argument_type 代表参数的型别 T。

negate::result_type 代表执行结果的型别 T。

T negate::operator()(const T& x)const 代表函数调用时的操作符。执行结果是(-x)。

negate::negate()代表仿函数的构造函数。

例 13-17 展示了求余仿函数和求反仿函数的用法。代码首先随机生成了一个 vector，然后对 vector 进行了对 10 求余操作并打印，以及求反操作，并打印结果。

例 13-17

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
void myprint(vector<int>& vd)
{
    vector<int>::iterator it;
    for(it=vd.begin();it!=vd.end();it++)
        cout<<*it<<" , ";
    cout<<endl;
}
void main(int argc, char* argv[])
{
    const int N=6;
    vector<int> V(N),V2(N),V3(N);
    generate(V.begin(),V.end(),rand);
    cout<<"vector v: "<<endl;
    myprint(V);
    transform(V.begin(),V.end(),V2.begin(),bind2nd(modulus<int>(),10));    //求余
    cout<<"vector v2: "<<endl;
    myprint(V2);
    transform(V.begin(),V.end(),V3.begin(),negate<int>());                 //求反
    cout<<"vector v3: "<<endl;
    myprint(V3);
}
```

程序的执行结果如下：

```
vector v:
41 , 18467 , 6334 , 26500 , 19169 , 15724 ,
vector v2:
1 , 7 , 4 , 0 , 9 , 4 ,
vector v3:
-41 , -18467 , -6334 , -26500 , -19169 , -15724 ,
```

13.7 其他类型的仿函数

除了上述最常见、最便宜的诸多仿函数之外，C++ STL 还提供了一些比较特殊的仿函数。这些仿函数主要有 7 个。它们分别是：

- identity <T>
- project1st <Arg1, Arg2>
- project2st <Arg1, Arg2>
- select1st <pair>
- select2nd <pair>
- hash <T>
- subtractive_rng <T>

13.7.1 证和映射

基本的“证”仿函数是 identity，其他的均是一般化形式。C++标准没有涵盖 identity 和映射两种操作行为，但由于它们非常普遍地被使用，多数 C++版本中也添加了它们的定义。

1. 仿函数 identity

```
identity<T>
```

类 identity 是一种一元仿函数。用于表示“证”仿函数。使用时需要一个参数，返回的是未经任何变化的原参数。

例 13-18 中的代码展示了如何使用 identity 仿函数，代码中的 id 是 identity<int>对象，等同于 int 类型。

例 13-18

```
#include <iostream>
#include <functional>
#include <assert.h>
using namespace std;
void main(int argc, char* argv[])
{
    int x=137;
    identity<int> id;
    assert(x==id(x));
    cout<<id(x)<<endl;
}
```

程序的执行结果如下：

```
137
```

提示： 在 Visual Studio2008 环境下，为确保程序能够顺利通过，需要将头文件 yvals.h 中的宏 _HAS_TRADITIONAL_STL 的值修改为 1，编译程序才能顺利通过。13.7 节和 13.8 节中的内容均为相同处理。

2. 仿函数 project1st 和 project2nd

仿函数 project1st 的声明形式为：

```
project1st<Arg1, Arg2>
```

仿函数 project2nd 的声明形式为：

```
project2nd<Arg1, Arg2>
```

这两个仿函数接受两个参数。project1st 返回第一个参数并忽略第二参数；project2nd 返回第二个参数并忽略第一个参数。这两个仿函数的基类均为 binary_function<Arg1, Arg2, Arg2>。这两个仿函数类均包括 4 个成员：

```
first_argument_type
```

第一个参数的类型

```
second_argument_type
```

第二个参数的类型

```
result_type
```

返回值的类型，也就是 Arg1 的类型

```
Arg1 operator()(const Arg1& x, const Arg2& y)const/  Arg2 operator()(const Arg1& x, const Arg2& y)const
```

函数调用，返回值是 x

```
project1st() / project2nd()
```

project1st 和 project2nd()类似。

上述两个仿函数在使用时需要包含头文件<functional>。

例 13-19 展示了如何使用 project1st 和 project2nd 仿函数。代码首先随机生成 1 个 vector，然后对该 vector 进行变换，规则是忽略第一个参数或者第二个参数，然后生成新的 vector，并打印。

例 13-19

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
#include <assert.h>
using namespace std;
void myprint(vector<int>& vi)
{
    vector<int>::iterator it;
    for(it=vi.begin();it!=vi.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
void main(int argc, char* argv[])
{
    vector<int>v1(5,137);
    vector<char*>v2(5,(char*)0);
    vector<int>result(5);
    transform(v1.begin(),v1.end(),v2.begin(),result.begin(),project1st<int, char*>());
    assert(equal(v1.begin(),v1.end(),result.begin()));
    myprint(result);
    transform(v2.begin(),v2.end(),v1.begin(),result.begin(),project2nd<char*,int>());
    assert(equal(v1.begin(),v1.end(),result.begin()));
    myprint(result);
}
```

程序的执行结果如下：

```
137 , 137 , 137 , 137 , 137 ,
137 , 137 , 137 , 137 , 137 ,
```

提示：在 Visual Studio2008 环境下，为确保程序能够顺利通过，需要将头文件 yvals.h 中的宏 _HAS_TRADITIONAL_STL 的值修改为 1，编译程序才能顺利通过。

3．仿函数 select1st 和 select2nd

仿函数 select1st<pair>和 select2nd<pair>可以接受单一参数，参数是 pair 类型。仿函数 select1st 返回该 pair 的第一个元素；仿函数 select2nd 返回该 pair 的第二个元素。这两个仿函数也定义于头文件<functional>中。这两个仿函数的基类是：

```
unary_function<pair, typename pair::first_type>
```

这两个仿函数类均包括 4 个成员：

```
argument_type
```

参数类型

```
result_type
```

返回值类型

```
const typename pair::first_type&select1st::operator()(const pair& p)const
```

函数调用

```
select1st()/select2nd()
```

调用方式

例 13-20 展示了如何使用 select1st 和 select2nd 的用法。代码首先定义了一个 std::map,key 为 int 类型，value 为 double 类型，使用 tranform 结合 select1st、select2nd 即可抽取 key 和 value 出来，并打印到标准输出。

例 13-20

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <map>
using namespace std;
void main(int argc, char* argv[])
{
    map<int, double> M;
    M[1]=0.3;
    M[47]=0.8;
    M[33]=0.1;
    cout<<"抽取键值: ";
    transform(M.begin(),M.end(),ostream_iterator<int>(cout," "),
        select1st<map<int,double>::value_type>());
    cout<<endl;
    cout<<"抽取实值: ";
    transform(M.begin(),M.end(),ostream_iterator<double>(cout," "),
        select2nd<map<int,double>::value_type>());
    cout<<endl;
}
```

程序的执行结果如下：

```
抽取键值: 1 33 47
抽取实值: 0.3 0.1 0.8
```

13.7.2　仿函数 hash 和 subtractive_rng

1. 仿函数 hash

仿函数 hash<T>是一种 hash 函数。STL 中所有的 hashed Associative Containers 均使用其作为默认的 hash 函数。

仿函数模板 template hash<T>仅针对 template 参数型别为 char*，const char*，string 和 integer 而定义的。若搭配不同的参数型别，需要提供自定义的 template specialization，或提供新的 hash 函数类。

使用仿函数 hash 需要包含头文件<hash_set>和<hash_map>。

仿函数没有基类。

该类的成员：

```
size_t hash::operator()(const T& x)const
```

运算符函数会返回参数 x 的 hash 值。

2. 仿函数 subtractive_rng

仿函数 subtractive_rng<T>是一种随机数发生器（Random Number Generator）。使用减去法可以产生拟真乱数。仿函数是一种一元函数。通常参数可以是无符号整型数，函数可以返回一个小于该数值的无正负号整数。若连续调用同一个 subtractive_rng 对象，会产生一个拟真乱数序列。

在《泛型编程与 STL》一书中指出，subtractive_rng 产生的数列完全是可决定的，由两个不同的 subtractive_rng 对象所产生的数列则互不相干。subtractive_rng 产生的数值取决于其种子以及之前被调用的次数。

仿函数的基类是：

```
unary_function<unsigned int , unsigned int>
```

仿函数类包含 6 个成员：

```
argument_type 代表参数型别。
result_type 代表执行结果的型别 unsigned int。
subtractive_rng(unsigned int seed)仿函数类的构造器。
subtractive_rng()默认构造器。
unsigned int operator()(unsigned int seed)重新设定乱数产生器，使其内部状态恰好与“以 seed 值构造之”的方式相同。
```

13.8　适配器

STL 标准库提供了一组函数适配器，用来特殊化或扩展一元和二元函数对象。适配器是特殊的类，可以被分成 4 类：

1. 绑定器（binder）

绑定器 binder 通过把二元函数对象的一个实参绑定到一个特殊值上，将其转换成一元函数对象。C++标准库提供了两种预定义的绑定器适配器：bind1st 和 bind2nd。bind1st 把值绑定到二元函数对象的第一个实参上；bind2nd 把值绑定在第二个实参上。绑定器非常有用，使用也非常灵活。

2．取反器（negator）

取反器（negator）是将函数对象的值反转的函数适配器。标准库提供了两个预定义的取反适配器：not1 和 not2。仿函数 not1 翻转一元预定义函数对象的真值；not2 翻转二元谓词函数的真值。取反器的定义和实现遵循成员函数适配器的模式。

3．成员函数适配器

使成员函数可以被用作算法的参数。当算法需要调用一个标准的操作或自定义操作时，成员函数的使用可以是非常简便的。例如：

```
void draw_all(list<Shape*>& c)
{
    for_each(c.begin(), c.end(), &Shape::draw);
}
```

4．函数指针适配器

使函数指针可以被用作算法的参数。头文件<functional>中使用了两个适配器，使函数指针可以和标准算法一起使用。

本节讲述时，将上述 4 种类型分成两组：成员函数适配器和其他适配器。

13.8.1　成员函数适配器

所谓成员函数适配器是一些小型的仿函数类。它们能够将成员函数作为仿函数来调用。每个适配器均需要一个型别为 T*或 T&的参数。通过该参数调用 T 类的成员函数。由此可知，成员函数是面向对象编程与泛型编程之间的桥梁。

无论是标准库中还是程序员自定义类的成员函数，其数量是巨大的。通常构造仿函数适配器是通过辅助函数，而不是通过构造函数。使用最普遍的两个辅助函数是 mem_fun 和 mem_fun_ref。

1．函数 mem_fun_t()和 const_mem_fun_t()

类 mem_fun_t 是一种成员函数适配器，和所有的仿函数具有同样的特点：拥有一个 operator()函数，使得 mem_fun_t()得以被一般的函数调用语法调用。通常情况下，mem_fun_t 的 operator()接受型别 T*的参数。

和其他的适配器一样，直接使用类 mem_fun_t 的构造函数是不方便的，但可以使用辅助函数 mem_fun()来代替。

使用仿函数 mem_fun 时，需要包含头文件<functional>。仿函数的基类如下：

```
unary_function<T*, R>
```

仿函数 const_mem_fun 和仿函数 mem_fun 大体相似。

例 13-21 展示了如何使用成员函数适配器 mem_fun。代码中首先定义了基类 B，以及派生类 D1 和 D2，并重写基类的 print 纯虚方法。在 main 函数中，初始化 4 个 B*，然后调用 for_each 遍历 vector 来利用多态机制分别调用相应的 print 方法；然后生成了一个二维数组，最后调用 transform 来获取这个二维数组中每个一维数组的大小。

例 13-21

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

```
#include <algorithm>
using namespace std;
struct B{
    virtual void print()=0;
};
struct D1:public B{
    void print(){
        cout<<"I'm a D1."<<endl;
    }
};
struct D2:public B{
  void print(){
        cout<<"I'm a D2."<<endl;
    }
};
void main(int argc, char* argv[])
{
    vector<B*> v;
    v.push_back(new D1);
    v.push_back(new D2);
    v.push_back(new D2);
    v.push_back(new D1);
    for_each(v.begin(),v.end(),mem_fun(&B::print));
    vector<vector<int>*> v2;
    v2.push_back(new vector<int>(5));
    v2.push_back(new vector<int>(3));
    v2.push_back(new vector<int>(4));
    transform(v2.begin(),v2.end(),ostream_iterator<int>(cout," "),mem_fun(&vector
<int>::size));
    cout<<endl;
}
```

程序的执行结果如下：

```
I'm a D1.
I'm a D2.
I'm a D2.
I'm a D1.
5 3 4
```

2．仿函数 mem_fun_ref_t 和 const_mem_fun_ref_t

仿函数 mem_fun_ref_t<R, X>是一种成员函数适配器。若 X 是类，具有成员函数 RX::f()，那么仿函数 mem_fun_ref_t<R,X>是一个仿函数适配器，从而可以方便地以一般函数的形式来调用 f()。该仿函数同样需要使用头文件<functional>。在实际使用过程中，使用辅助函数 mem+fun_ref()。

mem_fun_ref_t 和 const_mem_fun_ref_t 大体相似。

例 13-22 与例 13-21 类似，不同之处在于 mem_fun_ref 的参数与 mem_fun 的参数，前者使用的是类，而后者使用的是对象。代码中与上例中代码类似，不再过多叙述。

例 13-22

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

```
#include <algorithm>
using namespace std;
struct B{
   virtual void print()=0;
};
struct D1:public B{
   void print(){
      cout<<"I'm a D1."<<endl;
   }
};
struct D2:public B{
   void print(){
   cout<<"I'm a D2."<<endl;
   }
};
void main(int argc, char* argv[])
{
   vector<D1> v;
   v.push_back(D1());
   v.push_back(D1());
   for_each(v.begin(),v.end(),mem_fun_ref(&B::print));
   vector<vector<int>> v2;
   v2.push_back(vector<int>(2));
   v2.push_back(vector<int>(7));
   v2.push_back(vector<int>(3));
   transform(v2.begin(),v2.end(),ostream_iterator<int>(cout," "),mem_fun_ref
(&vector<int>::size));
   cout<<endl;
}
```

程序的执行结果如下：

```
I'm a D1.
I'm a D1.
2 7 3
```

3. 仿函数 mem_fun1_t 和 const_mem_fun1_t

仿函数 mem_fun1_t<R,X,A>是一种成员函数。若 X 是类，其成员函数 RX::f(A)，那么 mem_fun1_t<R, X, A>即是一种仿函数适配器，从而可以像一般函数一样来调用函数 f()。仿函数 mem_fun1_t 需要两个参数，一个参数是 X*，第二个参数是 A。由于仿函数 mem_fun1_t 的构造器不便于使用，通常以辅助函数 mem_fun()代替。

mem_fun1_t 和 const_mem_fun1_t 大体相似。

例 13-23 展示了如何使用 mem_fun。代码使用类似例 13-22 的原理，首先定义 operation 虚基类，square 和 Negate 子类实现 eval 方法；纯虚基类 B，以及子类 D 实现 f 方法。代码中定义了 operation* 的 vector，然后调用 transform 和 mem_fun 对该 vector 进行变换调用 eval 方法和 B::f 方法。

例 13-23

```
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>
using namespace std;
struct operation{
   virtual double eval(double)=0;
```

```
};
struct square:public operation{
    double eval(double x){
    return x*x;
    };
};
struct Negate:public operation{
    double eval(double x)
    {
    return -x;
    }
};
struct B{
    virtual int f(int x)const =0;
};
struct D:public B{
    int val;
    D(int x):val(x)
    {

    };
    int f(int x)const
    {
    return val+x;
    };
};
void main(int argc, char* argv[])
{
    vector<operation*>operations;
    vector<double>operands;
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operations.push_back(new square);
    operands.push_back(1);
    operands.push_back(2);
    operands.push_back(3);
    operands.push_back(4);
    operands.push_back(5);
    transform(operations.begin(),operations.end(),operands.begin(),
        ostream_iterator<double>(cout,"\n"),
        mem_fun(&operation::eval));
    cout<<endl;
    vector<B*>v;
    v.push_back(new D(3));
    v.push_back(new D(4));
    v.push_back(new D(5));
    int A[3]={7,8,9};
    transform(v.begin(),v.end(),A,ostream_iterator<int>(cout," "),mem_fun(&B::f));
    cout<<endl;
}
```

例 13-23 的执行效果见图 13-2。

图 13-2　例 13-23 的执行效果

4．仿函数 mem_fun1_ref_t 和 const_mem_fun1_ref_t

仿函数 mem_fun1_ref_t<R, X, A>是一种成员函数适配器。若 X 是类，具有成员函数 RX::f(A)；那么 mem_fun1_ref_t<R,X,A>是一个仿函数适配器。从而可以用一般函数的方式来调用 f。作为仿函数，mem_fun1_ref_t 需要两个参数。第一个参数的型别为 X&，第二个参数的型别为 A。通常使用辅助函数 mem_fun_ref 代替使用其构造器的方式。

仿函数 mem_fun1_ref_t 和 const_mem_fun1_ref_t 大体相似。

例 13-24 和 13-25 展示了 mem_fun_ref 的两种用法。原理与上面的实例类似，不再过多叙述。

例 13-24

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
void myprint(vector<vector<int>>& vv)
{
    vector<vector<int>>::iterator it;
    vector<int>::iterator itv;
    for(it=vv.begin();it!=vv.end();it++)
    {
        for(itv=it->begin();itv!=it->end();itv++)
        cout<<*itv<<" , ";
        cout<<endl;
    }
    cout<<endl;
}
void main(int argc, char* argv[])
{
    int A1[5]={1,2,3,4,5};
    int A2[5]={1,1,2,3,5};
    int A3[5]={1,4,1,5,9};
    int A4[5]={1,5,7,11,13};
    vector<vector<int>> v;
    v.push_back(vector<int>(A1,A1+5));
    v.push_back(vector<int>(A2,A2+5));
    v.push_back(vector<int>(A3,A3+5));
    v.push_back(vector<int>(A4,A4+5));
    myprint(v);
    int indices[4]={0,2,4,4};
    int& (vector<int>::*extract)(vector<int>::size_type);
    extract=&vector<int>::operator[];
    transform(v.begin(),v.end(),indices,ostream_iterator<int>(cout," "),mem_fun_ref
(extract));
    cout<<endl;
```

```
}
```

程序的执行结果如下：

```
1 , 2 , 3 , 4 , 5 ,
1 , 1 , 2 , 3 , 5 ,
1 , 4 , 1 , 5 , 9 ,
1 , 5 , 7 , 11 , 13 ,

1 2 9 13
```

例 13-25

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <vector>
using namespace std;
struct B{
    virtual int f(int x)const=0;
};
struct D :public B{
    int val;
    D(int x) : val(x)
    {

    }
    int f(int x)const
    {
        return val+x;
    }
};
void main(int argc, char* argv[])
{
    vector<D> v;
    v.push_back(D(3));
    v.push_back(D(4));
    v.push_back(D(5));
    int A[3]={7,8,9};
    transform(v.begin(),v.end(),A,ostream_iterator<int>(cout," "),mem_fun_ref(&B::f));
    cout<<endl;
}
```

程序的执行结果如下：

```
10 12 14
```

13.8.2　其他适配器

本小节主要讲述 8 种适配器：binder1st，binder2nd， pointer_to_unary_function，unary_negate，pointer_to_binary_function，binary_negate，unary_compose，binary_compose。将以上 8 种适配器分成 4 组：binder1st 和 binder2nd；pointer_to_unary_function 和 pointer_to_binary_function；unary_negate 和 binary_negate；unary_compose 和 binary_compose。

1．仿函数 binder1st 和 binder2nd

仿函数 binder1st()是一种仿函数适配器，可用于自适应二元函数转换成自适应一元函数。若 f 是仿函数 binder1st<BinaryFun>的对象，则 f(x)返回 F(c,x)，其中 F 是 BinaryFun 的对象，c 是常量。F 和 c 都被当成参数，传递给 binder1st 的构造函数。

通常使用仿函数 binder1st，不是使用其构造函数，而是使用其辅助函数 binder1st。双参函数的第一个参数可以设置成某个常量，从而形成单参函数。

使用仿函数 binder1st 时，需要包含头文件<functional>。该仿函数的基类是：

```
unary_function<typename BinaryFun::second_argument_type, typename BinaryFun::result_type>
```

仿函数 binder1st 的声明：

```
template<class Operation>class binder1st : public unary_function
        < typename Operation::second_argument_type, typename Operation::result_type>
    {
    public:
        typedef typename Operation::argument_type argument_type;
        typedef typename Operation::result_type result_type;
    binder1st( const Operation & _Func, const typename Operation::first_argument_type&
_Left );
        result_type operator()( const argument_type& _Right ) const;
        result_type operator()( const argument_type& _Right ) const;
    protected:
        Operation op;
        typename Operation::first_argument_type value;
    };
```

仿函数类 binder1st 具有 4 个成员：

（1）argument_type

代表参数型别：BinaryFun::second_argument_type。

（2）result_type

代表参数型别：BinaryFun::result_type。

（3）result_type operator()(const argument_type& x)const

代表函数调用的操作符。返回 F(c,x)，其中 F 和 c 是 binder1st 对象构造时的参数。

（4）binder1st(const BinaryFun& F, typename BinaryFun::first_argument_type c)

代表辅助函数，用于产生一个 binder1st 对象。若 F 的型别是 BinaryFun，则 binder1st（F,c）不但等价于 binder1st<BinaryFun>(F,c)，并且更方便。型别 T 必须转换为 BinaryFun 的第一参数型别。

同样，仿函数 binder2nd()也是一种仿函数适配器。可以用于将自适应二元函数转换成自适应一元函数。若 f 是仿函数类 binder2nd<BinaryFun>的对象，那么 f(x)会返回 F(x,c)，其中 F 为 BinaryFun 对象，c 是常量。F 和 c 均被作为参数，传给 binder2nd 的构造器。从直观上来讲，通过将双参函数的第二函数设定为常量，形成单参函数。

使用仿函数 binder2nd 也需要包含头文件<functional>。

函数的基类是：

```
unary_function<typename> BinaryFun::first_argument_type, typename BinaryFun::result_type>
```

仿函数 binder2nd 的声明：

```
template<class Operation>  class binder2nd : public unary_function
```

```
        < typename Operation::first_argument_type, typename Operation::result_type>
    {
    public:
        typedef typename Operation::argument_type argument_type;
        typedef typename Operation::result_type result_type;
    binder2nd( const Operation& _Func, const typename Operation::second_argument_type&
_Right );
        result_type operator()( const argument_type& _Left ) const;
        result_type operator()( argument_type& _Left ) const;
    protected:
        Operation op;
        typename Operation::second_argument_type value;
    };
```

类 binder2nd 和类 binder1st()相似，也包含 4 个成员。

由以上内容可知，这两个函数的区别在于：

函数的第二个参数是不同的。函数 binder1st 的第二个参数是“_Left”；函数 binder2nd 的第二个参数是“_Right”。

在处理谓词时，这两个函数也非常有意义。可以将二元谓词转变成一元谓词。常用于某范围内的数值与特定值之间的比较。

例 13-26 说明了如何使用 bind1st 仿函数。代码中首先定义了一个 vector，值为 0 到 9 对 10 求余。接着对该 vector 找到第一个不为 0 的元素，然后输出。接着使用了 bind2nd，实现查找大于-1 的值，并输出到标准输出。

例 13-26

```
#include <iostream>
#include <functional>
#include <algorithm>
#include <list>
using namespace std;
void myprint(list<int> lt)
{
    list<int>::iterator it;
    for(it=lt.begin();it!=lt.end();it++)
    cout<<*it<<" , ";
    cout<<endl;
}
void main(int argc, char* argv[])
{
    list<int> L;
    int i=0;
    for (i=0;i<10;i++ )
    {
    L.push_back(rand()%3);
    }
    myprint(L);
    list<int>::iterator first_nonzero=find_if(L.begin(),L.end(),bind1st(not_equal_to
<int>(),0));
    cout<<*first_nonzero<<endl;
    L.clear();
    for(i=0;i<10;i++)
    {
```

```
    L.push_back(rand()%4-3);
    }
    myprint(L);
    list<int>::iterator two_pos=find_if(L.begin(),L.end(),bind2nd(greater<int>(),-1));
    cout<<*two_pos<<endl;
    //return 0;
}
```

程序的执行结果如下：

```
2 , 2 , 1 , 1 , 2 , 1 , 0 , 0 , 1 , 2 ,
2
-2 , -2 , -2 , 0 , -2 , 0 , 0 , -1 , 0 , -3 ,
0
```

例 13-27 中的代码展示了如何使用 bind1st 和 bind2nd。代码中首先初始化一个 list，然后统计小于等于 8 的个数，以及大于 8 的个数，并输出到标准输出。

例 13-27

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

void main(int argc, char* argv[])
{
        int iarray[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    list<int> aList(iarray, iarray + 10);
    int k = 0;
    k = count_if(aList.begin(), aList.end(), bind1st(greater<int>(), 8));   //统计
    cout << "bind1st  test: " << k << endl;
    k = count_if(aList.begin(), aList.end(), bind2nd(greater<int>(), 8));
    cout << "bind2nd  test: " << k << endl;
}
```

程序的执行结果如下：

```
bind1st  test: 7
bind2nd  test: 2
```

在上述例子中，两个仿函数的作用是不同的。binder1st 用于统计小于 8 的数值的个数（8>q）；而 binder2nd 用于统计大于 8 的数值的个数。

通俗地讲：对于某范围内的每个元素用字符 q 表示，存在比较表达式 a>b，程序中给定的值为 val，如果使用函数 binder1st，内部的表达式应该是 val>q；如果使用函数 binder2nd，内部表达式是 q>val。

2. 仿函数 pointer_to_unary_function 和仿函数 pointer_to_binary_function

仿函数 pointer_to_unary_function 的声明如下：

```
    template<class Arg,class Result>class pointer_to_unary_function:public unary_function
<Arg,Result>
      {
      public:
         explicit pointer_to_unary_function( Result (*_pfunc)(Arg) );
```

```
    Result operator()( Arg _Left ) const;
};
```

仿函数 pointer_to_unary_function 是一种仿函数适配器，允许将函数指针 result(*f)(Arg)视为一个自适应一元函数。值得说明的是：表示某函数的函数指针应该是一个相当不错的一元仿函数，可以传入任何的 STL 算法中。在多数情况下，不会直接使用仿函数的构造器，而是使用仿函数的辅助函数 ptr_fun()。

举例如下：

```
transform(first, last, first, fabs);
```

上述语句可以将范围[first, last]中的元素求取绝对值，并使用该绝对值序列将原序列取代。

仿函数 pointer_to_binary_function 也是一种仿函数适配器，允许将函数指针 Result(*f)(Arg1, Arg2)视为一个自适应的二元函数。若 F 是一个仿函数 pointer_to_binary_function<Arg1, Arg2,result>对象，以型别 Result(*)(Arg1, Arg2)的函数指针 f 作为初值，那么 F(x,y)会调用函数 f(x,y)。

举例如下：

```
list<char*<::iterator item;
item=find_if(L.beigin(), L.end(), not1(binder2nd(pte_fun(strcmp),"OK")));
```

仿函数 pointer_to_binary_function，也经常使用辅助函数 ptr_fun()。

例 13-28 中的代码展示了如何使用 ptr_fun，以及如何使用 transform 算法。代码中首先定义了字符串数组，然后对该数组实行了 find_if 操作，找到不等于“pearly”的字符串，并输出到标准输出。

例 13-28

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

int main( )
{
   using namespace std;
   vector <char*> v1;
   vector <char*>::iterator Iter1, RIter;

   v1.push_back ( "Open" );
   v1.push_back ( "up" );
   v1.push_back ( "the" );
   v1.push_back ( "pearly" );
   v1.push_back ( "gates" );

   cout << "Original sequence contains: " ;
   for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
      cout << *Iter1 << " ";
   cout << endl;

   // To search the sequence for "pearly"
   // use a pointer_to_function conversion
   RIter = find_if(v1.begin(), v1.end(), not1 (bind2nd (ptr_fun ( strcmp ), "pearly" ) ) );

   if ( RIter != v1.end( ) )
   {
```

```
        cout << "The search for 'pearly' was successful.\n";
        cout << "The next character string is: "
        << *++RIter << "." << endl;
    }
}
```

程序的执行结果如下：

```
Original sequence contains: Open up the pearly gates
The search for 'pearly' was successful.
The next character string is: gates.
```

3. 仿函数 unary_negate 和 binary_negate

无论是 not1 还是 not2，其返回结果均取“非”运算。

仿函数 unary_negate 是自适应 Predicate，用于表示其他自适应 Predicate 的逻辑负值。若 f 是一个该仿函数的对象，构造时以 pred 作为其底部的仿函数，f(x)会返回!pred(x)。通常使用辅助函数 not1()实现该仿函数的功能，例如：not1(pred)。使用该辅助函数时，需要包含头文件<functional>。

仿函数 binary_negate 也是自适应 Predicate，用于表示其他自适应二元 Predicate 的逻辑负值。若 f 是二元负<Predicate>对象，构造时以 pred 作为其底层的仿函数，那么 f(x,y)会返回!pred(x,y)。此仿函数的辅助函数是 not2()。使用该辅助函数时，同样需要包含头文件<functional>。

提示：not1()可以用于对单个参数表达式逻辑值的取反；not2()可以用于对两个参数表达式逻辑值的取反。

例 13-29 展示了如何使用 not1。代码后弦定义了一个整数数组，然后使用 count_if 来遍历数组获取大于 10 的元素个数并输出；接着使用 not1 函数获取不大于 10 的元素个数并输出。

例 13-29

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>
using namespace std;
void myprint(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
void main(int argc, char* argv[])
{
    vector<int> v1;
    vector<int>::iterator Iter;
    int i;
    for (i = 0; i <= 6; i++)
    {
    v1.push_back(5 * i);
    }
```

```
    cout<<"vector v1:"<<endl;
    myprint(v1);
    vector<int>::iterator::difference_type res1;
    res1 = count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10));
                                                    //调用仿函数 greater<int>()
    cout << "v1 中大于的元素个数: "<< res1 << "." << endl;
    vector<int>::iterator::difference_type res2;
    res2 = count_if(v1.begin(), v1.end(),not1(bind2nd(greater<int>(), 10)));
                                                    //调用仿函数
    cout << "v1 中'不'大于的元素个数: "
        << res2 << "." << endl;
}
```

程序的执行结果如下：

```
vector v1:
0 , 5 , 10 , 15 , 20 , 25 , 30 ,
v1 中大于 10 的元素个数: 4.
v1 中'不'大于 10 的元素个数: 3.
```

例 13-30 展示了如何使用 not2。代码中首先定义了一个 int 型的 vector，然后对该 vector 调用排序函数 sort，首先是使用了默认的规则也就是升序排序。接着使用 not2 和 less_equal 实现了降序，并数组内容。

例 13-30

```
#include <vector>
#include <algorithm>
#include <functional>
#include <cstdlib>
#include <iostream>
using namespace std;
void myprint(vector<int>& v)
{
    vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
int main( )
{
    vector <int> v1;
    vector <int>::iterator Iter1;

    int  i;
    v1.push_back( 6262 );
    v1.push_back( 6262 );              //程序会发出异常
    for ( i = 0 ; i < 4 ; i++ )
    {
     v1.push_back( rand( ) );
    }
```

```
    cout << "Original vector v1:"<<endl ;
    myprint(v1);
    sort( v1.begin( ), v1.end( ) );
    cout << "Sorted vector v1:" ;
    myprint(v1);
   sort( v1.begin(),v1.end(),not2(less_equal<int>()));
   //调用仿函数 not2 和 less_equal<int>
    cout << "Resorted vector v1:" ;
    myprint(v1);
}
```

程序的执行结果如下：

```
Original vector v1:
6262 , 41 , 18467 , 6334 , 26500 ,
Sorted vector v1:41 , 6262 , 6334 , 18467 , 26500 ,
Resorted vector v1:26500 , 18467 , 6334 , 6262 , 41 ,
```

4. 仿函数 unary_compose 和 binary_compose

在 13.3 节中讲述辅助用仿函数时，在表 13-2 中已经列出了两个辅助用函数 compose1 和 compose2。此处再略加介绍。

仿函数 unsry_compose 的原型如下：

```
unary_compose<Fun1, Fun2>
```

仿函数类 unary_compose 是一种仿函数适配器。若 f 和 g 均是自适应一元函数，并且 g 的返回型别可以转换为 f 的参数型别，那么 unary_compose 可用于产生一个仿函数对象 h，从而 h(x)=f(g(x))。此种行为称为函数合成。函数合成是代数学的重要概念，可以使用单纯的仿函数任意构造出复杂的仿函数。仿函数的辅助函数是 compose1()。

仿函数 binary_compose 也是一种仿函数适配器。若 f 是自适应二元函数，g1 和 g2 均为自适应一元函数，并且 g1 和 g2 的返回型别均可转换为 f 的参数型别，仿函数 binary_compose 可用于产生一个仿函数 h，从而 h(x)=f(g1(x), g2(x))。仿函数的辅助函数是 compose2()。

使用上述两个仿函数需要包含头文件<functional>。

例 13-31 综合使用了多个仿函数，尤其是 compose1 和 compose2 函数。代码中首先定义了 vector angleA 并初始化，然后使用 transform 仿函数来初始化 sinesS，具体规则是使用嵌套的 compose1 来组合 sin 函数以及使用 bind2nd 来组合 multiple 乘法仿函数（*pi/180），实现了-sin（pi*x）/180 然后打印并输出。接着使用 compose2，实现了计算 sin(x)/(x+5)。

例 13-31

```
#include <vector>
#include <functional>
#include <algorithm>
#include <list>
#include <iostream>
#include <assert.h>
#include <cmath>
using namespace std;
template<typename T> void myprint(vector<T>& vd)
{
   vector<T>::iterator it;
```

```
    for(it=vd.begin();it!=vd.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
template<typename T> void myprintL(list<T>& l)
{
    list<T>::iterator it;
    for(it=l.begin();it!=l.end();it++)
    {
        cout<<*it<<" , ";
    }
    cout<<endl;
}
void main(int argc, char* argv[])
{
    cout<<"Test compose1()  and compose2() :"<<endl;
    double angleA[5]={60,45,90,-30,-180};
    double sinesS[5]={0,0,0,0,0};
    vector<double> angle;
    vector<double> sines;
    sines.assign(sinesS,sinesS+5);
    angle.assign(angleA,angleA+5);
    cout<<"Original Vector sines: "<<endl;
    myprint(sines);
    cout<<"Original Vector angle: "<<endl;
    myprint(angle);
//
    const double pi=3.1415926;
    assert(sines.size()>=angle.size());
    transform(angle.begin(),angle.end(),sines.begin(),
        ompose1(negate<double>(),
        compose1(ptr_fun(sin),
        bind2nd(multiplies<double>(),pi/180.0))));
    cout<<" Vector sines: "<<endl;
    myprint(sines);
//
    int larray[6]={1,2,6,15,10,3};
    double la[6]={0.1,0.1,0.1,0.1,0.1,0.1};
    list<int> li(larray,larray+6);
    list<double> li2(la,la+6);
    cout<<"Original list li :"<<endl;
    myprintL(li);
    cout<<"Original list li2 :"<<endl;
    myprintL(li2);
    list<int>::iterator itl;
    itl=find_if(li.begin(),li.end(),compose2(logical_and<bool>(),
                                    bind2nd(greater_equal<int>(),1),
                                    bind2nd(less_equal<int>(),10)));
    assert(itl==li.end()||(*itl>=*itl<=10));
    cout<<"范围[1,10]之内的第一个元素: "<<endl;
```

```
    cout<<*itl<<endl;
    transform(li.begin(),li.end(),li2.begin(),
        compose2(divides<double>(),
                ptr_fun(sin),
                bind2nd(plus<double>(),0)));//计算 sin(x)/(x+5)
    cout<<"list li2 :"<<endl;
    myprintL(li2);
}
```

程序的执行结果如下：

```
Test compose1()  and compose2() :
Original Vector sines:
0 , 0 , 0 , 0 , 0 ,
Original Vector angle:
60 , 45 , 90 , -30 , -180 ,
Vector sines:
-0.866025 , -0.707107 , -1 , 0.5 , 5.35898e-008 ,
Original list li :
1 , 2 , 6 , 15 , 10 , 3 ,
Original list li2 :
0.1 , 0.1 , 0.1 , 0.1 , 0.1 , 0.1 ,
```

范围[1,10]之内的第一个元素：

```
1
list li2 :
0.841471 , 0.454649 , -0.0465692 , 0.0433525 , -0.0544021 , 0.04704 ,
```

13.9 本章小结

本章全面介绍了 C++ STL 的仿函数内容。本章可以说是全书的精华部分。读者应该认真学习，并深刻体会。本章对仿函数的概念进行了阐述，按用途对仿函数做了简要地概述。之后按具体功能的不同，对 STL 中的预定义仿函数做了全面介绍，主要涉及：关系仿函数，逻辑仿函数，算术仿函数，证和映射仿函数，hash 和 subtractive_rng 仿函数。13.8 节重点讲述了 STL 的各种适配器。

本章内容是全书的重点，也是难点，更是精华。希望读者能反复阅读。

第 14 章 配置器

空间配置器（allocator）简称配置器。空间配置器代表一种特定的内存模型，并提供一种抽象概念，便于将内存的申请转变为对内存的直接调用。

配置器主要用于将算法和容器的实现隔离于物理存储细节之外。每个配置器均提供了一套分配与释放存储的标准方式、一套用于指针类型和引用类型的标准名字。

配置器是一种纯粹的抽象（概念）。C++ STL 提供了一个标准配置器，主要目的是为程序员提供更好的服务。程序员还可以根据自己的需要，设计和提供自定义的分配器。

标准器容器和算法均通过分配器提供的功能获取和访问存储。通过提供新的分配器，为标准容器提供新的不同的存储使用方式。

14.1 使用配置器

对于程序员而言，使用不同的配置器应该没有任何困难。在使用时仅需要将配置器作为一个模板参数。

例如：使用特殊的配置器 myAlloc，用于产生特殊的容器和 strings。

```
    std::vector<int, myAlloc<int>> v;
    std::map<int, float, less<int>,myAlloc<std::pair<cons tint, float>>> m;
    std::basic_string<char, std::char_traits<char>m myAlloc<char>> s;
```

还可以自定义一些配置器，例如：

```
    typedef std::basic_string<char, std::char_traits<char>, myAlloc<char>> xstring;
    typedef std::map<xstring, xstring, less<xstring>, myAlloc<std::pair<const xstring,
xstring>>>xmap;
    xmap map;
```

使用非标准的配置器分配的对象，表面上没有任何的问题。但是，不同的分配器分配的元素是不能混淆的，否则会造成未定义行为。运算符 operator==可以用来比较两个配置器是否使用了相同的内存模型。如果返回 true，表示一个配置器的存储空间可以由另一个配置器收回。“以配置器为模板参数”的型别，会提供一个 get_allocator()函数，由此可以获得对应的配置器。例如：

```
    if(mymap.get_allocator()==s.get_allocator())
    {
```

```
… …
}
```

对于广大程序员而言，使用配置器可以实例出容器和其他组件。配置器提供一个接口，包括：分配、生成、销毁和回收对象。通过配置器，容器和算法的元素存储方式才得以被参数化。

假如，实例出运用共享内存的配置器，或允许 map 的元素存储在持久性内存中。

配置器类的最简单的内存分配操作见表 14-1。

表 14-1　最简单的内存分配操作

表达式	效果
allocate(int　num)	为 num 个元素分配内存
construct(p)	将 p 所指的元素初始化
destroy	销毁 p 所指的元素
deallocate(p, num)	回收 p 所指的可容纳 num 个元素的内存空间

例如，最简单的 vector 实例方案。配置器被作为模板参数或构造器的函数传递给 vector，并保存在容器内部。容器 vector 的声明如下：

```
template < class Type, class Allocator = allocator<Type> > class vector
```

在容器类 vector 中，存在配置器类型的相关声明，其构造函数也包含了配置器的内容。

```
typedef  typename  Allocator::reference   reference;
typedef  typename  Allocator::const_reference  const_reference;
typedef  Allocator  allocator_type;
typedef  typename  Allocator::pointer   pointer;
typedef  typename  Allocator::const_pointer   const_pointer;
......
explicit vector(const Allocator&=Alocator());
explicit vector(size_type n, const T& value =T(), const Allocator& =allocator());
......
```

上述第二个构造函数中，参数 n 代表元素个数，参数 value 代表数值，配置器是由函数 allocator() 产生的。

对于一些未初始化的内存，STL 也提供了 3 个函数：uninitialized_fill()，uninitialized_fill_n()，uninitialized_copy()。这三个函数相对而言使用较方便。

```
uninitialized_fill(beg , end , val) : 以数值 val 初始化范围[beg , end]中的元素;
uninitialized_fill_n(beg , num , val): 以数值 val 初始化范围[beg, end]中的 num 个元素;
uninitialized_copy(beg, end, mem) : 以范围[beg, end]中的各个元素初始化 mem 为起始地址的各个元素。
```

例如，可以在构造器中添加以下代码：

```
int  size= n;
elems=alloc.allocate(num);            //elems 代表某种类型的指针
uninitialized_fill_n(elems, n , val);
......
```

还有对于原始存储区，其迭代器也是比较复杂的问题。C++ STL 提供了 raw_storgae_iterator 类，用于在未初始化的内存中访问，并初始化。这样，使用该类定义的指针，即可使用任何算法。例如：

```
copy(x.begin() , x.end() , raw_storage_iterator<T* , T>(elems));
```

模板 raw_storage_iterator 的第一个参数 T*需要一个对应元素型别的 output 迭代器；第二个模板

参数必须是元素型别。

对于临时缓冲区，程序开发过程中可能存在以下两个经常使用的函数：

```
get_temporary_buffer()
return_temporary_buffer()
```

这两个函数可以用来处理部分未初始化的内存，便于函数的短暂需求。

函数 get_temporary_buffer()返回的内存容量可能会比预期的少一些，返回值是 pair 类型，内含所获的内存地址和内存容量。例如：

```
pair<mytype*, std::ptrdiff_t> p= get_temporary_buffer<mytype>(num);
```

如果 pair 的 second 值为 0，那么表示分配内存失败，即分配的容量为 0；还有可能 second 会小于 num，表示分配的内存不够。pair 的 first 包含被分配内存的地址，如果不为 0 的话，可以返回 first 的值，即返回被分配的内存地址。

但是函数 get_temporay_buffer()和 return_temporary_buffer()已经很少使用了。

14.2　C++标准程序库的默认配置器（标准配置器）

在 C++ STL 的头文件<memory>中，标准配置器的声明如下：

```
namespace std{
   template<class T> class allocator{
   public:
      typedef  size_t size_type;
      typedef  ptrdiff_t  difference_type;
      typedef  T*   pointer;
      typedef  const  T*  const_pointer;
      typedef  T&  reference;
      typedef  const  T&  const_reference;
      typedef  T    value_type;
      template<class U> struct rebind{
         typedef allocator<U> other;
      }
      pointer address(reference value) const;
      const_pointer address(const_reference value) const;
      allocator() throw();
      allocator(const allocator&) throw();
      template<class U> allocator(const allocator<U>&) throw();
      ~allocator() throw();
      size_type  max_size() const throw();
      pointer allocator(size_type num, allocator<void>::const_pointer hint=0);
      void construct(pointer p, const T& value);
      void destroy(pointer p);
      void deallocator(pointer p, size_type num);
   };
}
```

标准配置器（默认配置器）使用全局的函数 operator new 和 operator delete 分配和回收内存。函数 allocate()有可能会抛出 bad_alloc 类型的异常。标准配置器是可以优化的。new 函数和 delete 函数被调用的确切时机很难预测。配置器中包含一个较特殊的模板结构定义：rebind。模板 template 结构

使配置器间接为其他型别分配空间。例如：

```
Allocator::rebind<T2>::other
```

如果需要实例化一个容器，必须为“非元素型别”的对象分配空间，rebind<>可以派上大用场。通常，需要使用配置器为数组分配空间，元素型别为指针。

函数 allocate(num)可以为 n 个对象分配空间，该空间可以由对应的 deallocate(p,n)释放。函数 deallocate()以元素的个数 num 作为参数，在维持最少量的有关被分配存储的信息条件下，尽可能地优化分配器。同时，配置器可以要求程序员在调用 deallocate()的时候能提供正确的 num 值。函数 deallocate()与运算符 delete()的区别在于：参数必须“非 0”。

默认的 allocator 使用 operator new(size_t)获取存储，使用 delete(void*)操作释放它。这意味着存储耗尽时，要调用 new_handler()，会抛出 std::bad_alloc 的异常。

函数 allocate()不需要每次都调用低级的分配系统。更好的策略是使配置器管理关于空间的自由表，从而在最小的时间开销下提交存储。allocate()的可选参数 hint 完全依赖于实现。在局部性特别重要的系统中为分配器提供帮助。

allocator<void>::pointer 类型被作为通用指针类型，在标准容器中都使用 void*。用户在调用 allocator()时，均存在两种合理选择：

（1）没有提示；

（2）用一个指针对象的指针作为提示，该对象经常与新对象一起使用。

实用配置器的目的是：在分配容器的内存时，不必直接和原始存储打交道。

需要说明的是：allocator 的操作均是基于 pointer 和 reference 这两个类型定义表述的，使用户可能提供其他的替代类型而访问存储。在 C++语言中，不可能定义出完美的引用类型。语言和库的实现需要使用大量的类型定义（typedef）来支持那些常规的基本数据类型。

每个程序可以有一个配置器，提供对持续性存储器的访问。另外，配置器可以像“长”指针一样，访问超出常规指针的寻址范围的主存。

常规用户为分配器提供一种不寻常的指针类型，用于服务特殊的用途。对引用无法做与此等价的事情。配置器的设计很容易处理使用模板参数描述的类型对象。多数容器的实现需要其他类型的对象。对于 rebind()类型，是使一个配置器能分配任意类型的对象。

14.3 自定义配置器

实现一个配置器最主要的是：分配和回收存储空间。在原有的标准配置器的基础上，修改配置器的几个成员函数即可，像 max_size()、allocator()、deallocator()。程序员可以将自己在内存分配方面的策略体现在这 3 个函数内，像重新运用内存、使用共享内存，或将内存映射到面向对象的数据库内。

容器的实现往往需要多次分配和释放对象。对于 allocator()的朴素实现，意味着对运算符 new 的大量使用。运算符 new 的效率并不是很高。如果使用固定大小存储块的存储池，可以使配置器的分配效率比常规的 new 更高。

标准库对于标准容器所使用的配置器有限制：允许标准容器的实现将分配器类型的对象认为是等价的。尤其是在访问两个序列的操作（swap）时，更是需要检查被处理的对象是否拥有相同的分配器。

14.4　配置类的详细讨论

根据 C++标准规定，配置器需要提供以下的型别定义和操作函数。若配置器被标准容器使用的话，还需要一些特殊要求，否则要求会少一些。

14.4.1　型别

型别名称	说　　明
allocator::value_type	代表元素型别，相当于类模板中的 T。例如：allocator<T>
allocator::size_type	一个无正负号的整数型别，表示应用程序模型中最大对象的大小
allocator::different_type	一个有正负号的整数型别，表示分配模型中两个指针的差距。如果配合标准容器使用，必须等于 ptrdiff_t
allocator::pointer	一个指向元素的指针型别。为配合标准容器的使用，此型别要等同于 allocator<T>中的 T*
allocator::const_pointer	一个"指向元素"的常数指针型别。若配合标准容器使用，此型别需要等同于 allocator<T>中的 const T*
allocator::reference	一个指向元素的 reference 型别。此型别要等同于 allocator<T>中的 T&
allocator::const_reference	一个指向元素的 const reference 型别。此型别要等同于 allocator <T> 中的 const T&
allocator::rebind	是一个模板结构体，使配置器间接为其他型别分配空间

14.4.2　配置类的成员函数

构造函数 allocator::allocator()：默认的构造函数，用于产生一个配置器对象。

构造函数 allocator::allocator(const allocator& a)：复制构造函数，产生一个配置器副本，使原有的配置器分配的空间可以通过另一个配置器回收。

析构函数~allocator::allocator()：用于销毁配置器对象。

```
pointer allocator::address(reference value)
const_pointer allocator::address(const_reference value)
```

这两个函数会返回非常数指针，指向一个非常数值；第二个参数返回一个常数指针，指向一个常数值。

size_type allocator::max_size()：返回"对 allocate()有意义、用于分配空间"的最大许可值。

```
pointer allocator::allocate(size_type num),
pointer allocator::allocate(size_type num, allocator<void>::const_pointer hint):
```

两种形式均返回一块内存空间，可容纳 num 个型别为 T 的元素。元素不会被构造或初始化。第二个参数是可有可无的，真实意义由实作的版本具体决定，可以用于辅助提升效能。

```
void allocator::deallocate(pointer p, size_type num):
```

用于释放空间，指针 p 所指的空间是由同一个配置器以 allocate()分配的。p 不能是 NULL 或 0。区块内的元素必须已经被析构。

void allocator::construct(pointer p, const T& value)：以参数 value 作为 p 所指的那个元素的初值，相当于 new((void*)p)T(value)。

void allocator::destroy(pointer p)：销毁 p 所指向的对象，但不回收空间；仅仅单纯地调用对象的析构函数，相当于((T*)p)->~T()。

bool operator==(const allocator&a1, const allocator& a2)：如果配置器 a1 和 a2 是可以互换的，会

返回 true。如果某个配置器分配的空间由另一个配置器收回，称为两者可以互换。若配合标准容器使用，对于不同型别产生的配置器必须可以互换。函数应当返回 true。

bool operator!=(const allocator& a1, const allocator& a2) ：若配置器 a1 和 a2 不是可以互换的，就返回 true。相当于：!（a1==a2）。若配合标准容器使用，对相同型别产生的配置器彼此必须可以互换。函数应当始终返回 false。

14.4.3 广义配置器

配置器 allocator 是有关通过模板参数向容器传递信息的一种简化变形。容器中每个元素均通过容器的分配器进行分配。若允许两个同类型的 list 采用不同的分配器，splice()无法通过重新链接的方式实现。将 splice()定义成基于元素复制，以防出现偶发的情况。如果允许配置器完全通用，一个配置器能为任意类型的分配元素的 rebind 机制需要做得更加精细一些。标准配置器被假定为不在每个对象里存放数据，标准的实现可以利用该情况。

反对配置器在对象内存放信息的限制实际不是很严重。大部分配置器不需要在对象内存放数据。配置器可以保存有关配置器类型的数据。若需要各种数据，可以使用各种配置器类型。

配置器不允许在每个对象里存放数据的限制是强制性的。由于标准库在运行时间和空间效率上要求比较严格，当标准库在效率方面的限制不是很重要时，可以采用配置器技术。配置器还可以携带某一类信息，该信息是从公共基类继承而来的。

配置器可以为容器提供一种控制，使容器的行为类似持续性存储器工作的高速缓存存储器，或者提供容器与其他对象之间的关联。

按照此方式，任何服务都能以常规容器操作透明的形式提供。最好是将有关数据存储的问题和数据使用的问题分开，后者不属于某个广义的配置器，但可通过单独的模板参数提供。

14.4.4 动态存储

动态存储用于实现 new 和 delete 运算符的功能，在头文件<new>声明中包含空异常描述：

```
class bad_alloc: public exception{/*…*/}
struct nothrow_t{};
extern const nothrow_t nothrow;
typedef void (*new_handler)();
new_handler set_new_handle(new_handler new_p) throw();
void* operator new(size_t) throw(bad_alloc);
void  operator delete(void* ) throw();
void* operator new(size_t , const nothrow_t&) throw();
void  operator delete(void* , const nothrow_t&) throw();
void* operator new[](size_t) throw(bad_alloc);
void  operator delete[](void*) throw();
void*  operator new[](size_t , const nothrow_t & ) throw();
void operator delete[](void* , const nothrow_t&) throw();
void* operator new(size_t, void* p) throw(){return p};
void operator delete(void* p , void* ) throw(){};
void* operator new[](size_t, void* p) throw(){return p};
void operator delete[](void* p , void* ) throw(){};
```

operator new()或 operator new[]()不能通过抛出 std::bad_alloc 异常发出存储耗尽的信号，在分配失败时会返回 0。new 表达式将检测由带有空异常描述的分配函数返回的值。若返回值是 0，不会调

用构造函数并返回值 0。带有 nothrow 的配置器将返回 0 指明分配失败，而不抛出 bad_alloc。

举例如下：

```
int* p=new int[1000];                    //如果分配错误，会抛出异常
int* q= new(nothrow) int [1000];         //不会抛出异常
```

如果 q==0，那么代表分配内存失败。

14.4.5 C 风格的分配

在头文件<cstdlib>中，可以找到以下语句：

```
void*  malloc(size_t  s);           //分配 s 个字节
void*  calloc(size_t n, size_t s); //分配 n 乘 s 个字节，初始化为 0
void  free(void*  p);               //释放由 malloc()或 calloc()分配的空间
void*  realloc(void* p, size_t  s);//将 p 所指向的数组的大小变为 s，不能实现就分配 s 个字节，将
                                    //p 所指数组复制过去并释放 p
```

这些函数现在已经很少使用，应尽量避免。尽量使用 new、delete 和标准容器。上述函数处理的都是未初始化的存储。尤其是 free()不会对它所释放的存储调用析构函数。实现 new 和 delete 有可能会使用这些函数，但不保证这样做。若需要使用 realloc()，并依赖于标准容器，通常会更简单有效。

标准库 STL 还提供了一些函数，其目的在于高效地完成字节操作。C 语言是通过 char*指针访问无类型的字节的，这些函数均在<cstring>中。所有的 void*指针在这些函数中作为 char*指针处理：

```
void* memcpy(void* p, const void* q, size_t n);
void* memmove(void* p, const void* q, size_t n);
```

和函数 strcpy()一致，函数从 q 向 p 复制 n 个字节并返回 p。使用 memmove()复制的区域可以有重叠，而 memcpy()是假定区域不能重叠的。

14.5 未初始化的内存

本节主要讲述 3 个辅助函数，这 3 个辅助函数均应用于未初始化内存之上。

（1）uninitialized_fill(ForwardIterator beg, ForwardIterator end, const T& value)

以数值 value 初始化范围[beg, end]区间内的元素；函数要么成功，要么没有影响。通常的实作形式为：

```
namespace std{
template <class ForwIter, class T>
   void uninitialized_fill(ForwIter  beg, ForwIter end,  const T& value)
{
typedef typename iterator_traits<ForwIter>::value_type VT;
ForwIter save(beg);
try{
       for(; beg!=end; ++beg)
{
           new(static_cast<void*> (&*beg))VT(value);
}
}
   catch(...)
{
```

```
        for(; save!=beg;++save)
    {
        save->~VT();
    }
    throw();
      }
    }
}
```

（2）uninitialized_fill_n()

函数原型为：

```
void uninitialized_fill_n(ForwardIterator beg, size  num, const T& value);
```

以数值 value 初始化从 beg 开始的 num 个元素。同样，函数要么成功，要么没有影响。函数通常实作为：

```
namespace std{
    template <class ForwIter, class Size, class T>
    void uninitialized_fill_n(ForwIter beg, Size num, const T& value)
    {
            typedef typename iterator_traits<ForwIter>::values_type VT;
            ForwIter save(beg);
            try{
            for(; num--;++beg)
    {
                save->~VT();
                }
    throw();
        }
    }
}
```

（3）uninitialized_copy()

函数原型为：

```
ForwardIterator uninitialized_copy(InputIterator  sourcebeg,
InputIterator  sourceend,
ForwardIterator   destbeg)
```

以[sourcebeg, sourceend]区间内的元素为根据，将 destbeg 起始的元素加以初始化。函数要么成功，要么没有影响。函数通常实作为：

```
namespace  std{
template <class InputIter, class ForwIter>
ForwIter uninitialized_copy(InputIter beg, InputIter end, ForwIter dest)
    {
        typedef typename iterator_traits<ForwIter>::value_type VT;
        ForwIter save(dest);
        try{
            for(;beg!=end;++beg,++dest)
    {
                new(static_cast<void*>(&*dest)VT(*beg));
    }
                return dest;
                }
        catch(...)
            {
```

```
            for(;save!=dest;++save)
          {
              save->~VT();
}
          }
       }
}
```

14.6 配置器举例

本小节使用例 14-1，用来说明配置的使用方法。

例 14-1

```
#include <iostream>
#include <memory>
#include <vector>
#include <algorithm>
using namespace std;
template<typename T>void myprint(vector<T>& vv)                    //输出 vector
{
 vector<T>::iterator it;
 for(it=vv.begin();it!=vv.end();it++)
 {
       cout<<*it<<" , ";
 }
  cout<<endl;
}
void myprintA(allocator<int>::pointer vv,size_t num)               //输出配置器
{
 int i;
 for(i=0;i<num;i++)
 {
       cout<<vv[i]<<" , ";
 }
 cout<<endl;
}
void main(int argc, char* argv[])
{
 cout<<"allocator test."<<endl;
 vector<int> v1;
 vector<int>::iterator v1It;
 vector<int>::allocator_type v1A;
 allocator<int>::const_pointer  cv1P;
 allocator<int>::pointer  v1p;
 allocator<int> AL;

 int i;
 for(i=1;i<=8;i++)
 {
       v1.push_back(i);                                            //初始化 vector
 }
 cout<<"The Original Vector v1: "<<endl;
 myprint(v1);
 int kk=6;
 cv1P=v1A.address(*find(v1.begin(),v1.end(),kk));                  //使用 address
```

```
  cout<<"find an element by address. "<<endl;
  cout<<*cv1P<<endl;
 //
  v1p=AL.allocate(10);                                          //分配内存
  int ar[6]={0,9,8,7,6,5};
  for(i=0;i<6;i++)
        v1p[i]=ar[i];
  cout<<"array  v1p:"<<endl;
  myprintA(v1p,6);
  AL.deallocate(v1p,10);                                        //释放内存
 //
  kk=6;
  int kb=12;
  v1p=v1A.address(*find(v1.begin(),v1.end(),kk));
  v1A.destroy(v1p);
  v1A.construct(v1p,kb);                                        //修改数据
  cout<<"destroy and construct :"<<endl;
  myprint(v1);
 //
  allocator<int>::size_type size;
  size=v1A.max_size();                                          //max_size
  cout<<"max_size : "<<size<<endl;
 }
```

例 14-1 的执行效果见图 14-1

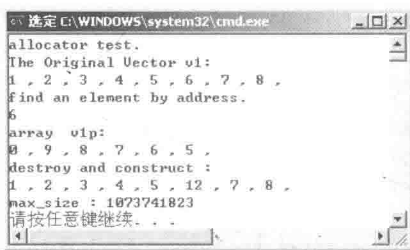


图 14-1　例 14-1 的执行效果

14.7　本章小结

本章主要讲述了内存配置器。首先简要描述了配置器的使用，之后描述了 C++ STL 的默认配置器和自定义配置器的使用方法。还详细讨论了配置类、未初始化内存等内容。并在 14.5 节给出了一个例题。

参 考 文 献

［1］ ISO_IEC-14882.programming languages C++ second edition

［2］ Stephen Prata. C++ Primer Plus 中文版 . 5 版. 孙建春，等译.北京：人民邮电出版社，2005.

［3］ Stanley B Lippman, Josee Lajoie. C++ Primer 中文版.3 版. 潘爱民，等译.北京：中国电力出版社，2002.

［4］ P.J. PLAUGER, ALEXANDER A. STEPANOV，MENG LEE. C++ STL 中文版. 王昕，译.北京：中国电力出版社，2002.

［5］ Stanley B Lippman, etc. C++ Primer 中文版. 4 版. 李师贤，等译.北京：人民邮电出版社，2010.

［6］彭木根，等. C++ STL 程序员开发指南. 北京：中国铁道出版社，2003.

［7］ Bjarne Stroustrup. C++程序设计语言特别版[M]. 裘宗燕，译.北京：机械工业出版社，2002.

［8］ Matthew H. Austern. 泛型编程与 STL[M]. 侯捷，译.北京：中国电力出版社，2003.

［9］ AI Stevens, or. Dobb’s Journal. C++大学自学教程.7 版. 林瑶，等译.北京：电子工业出版社，2003.

［10］Nicolai M. Josuttis. C++标准程序库自修教程与参考手册[M]. 侯捷，等译.武汉：华中科技大学出版社，2002.

［11］侯捷.STL 源码剖析[M]. 武汉：华中科技大学出版社，2002.

［12］萧黎，等. Turbo C 2.0 运行库函数源程序与参考大全[M]. 北京：中国科学院希望高级电脑技术公司，1990.