第一篇 C++基础

如果说有一种语言定义了当今编程的实质，那么它就是 C++。它是高性能软件开发的杰出语言。它的语法已经成为专业编程语言的标准，并且它的设计思想在计算界引起深刻的反响。

Java 和 C#语言都是从 C++继承而来的。简而言之，要成为专业的编程人员就意味着要深刻理解 C++。它是现代编程的基础。

本篇旨在介绍 C++，包括它的历史，它的设计思想，以及几个最重要的特性。学习一门编程语言最困难的事情就是所有的元素都不是单独孤立存在的。相反，构成语言的各个部分是相互协作，一起工作的。这种相关性使得我们很难讨论 C++的一个方面而不去考虑其它方面。为了克服这个困难，本篇对几个 C++特性进行了简单的介绍，包括 C++程序的通用形式，一些基本的控制语句，以及运算符。本篇不会涉及过多的细节，更注重 C++程序中通用的概念。

必备技能 1.1 C++历史简介

C++是从 C 语言发展而来的。这一点不难理解，因为 C++是构筑的 C 语言的基础之上的。然而，C++是 C 语言的超集。C++扩展并增强了 C 语言，支持面向对象的编程（这点在本篇的后面会进行描述）。C++同时对 C 语言做了改进，包括扩展了例行程序库集。然而大部分的 C++特性是直接从 C 继承而来的。因此，为了充分理解和欣赏 C++，我们必须深入了解 C 语言是如何运作的。

C: 现代编程时代的开始

C 语言的发明定义了现代编程时代的开始。它的影响不应该被低估，因为它从根本上改变了人们考虑和实现程序的方法。它的设计思想和语法已经影响到了每一个主流的编程语言。C 语言是计算发展的主要的，革命性的推动力之一。

C 语言由 Dennis Ritchie 在 DEC PDP-11 电脑上，在 UNIX 操作系统下发明并实现的。C 语言是从一种古老的叫做 BCPL 的语言发展而来。BCPL 是由 Martin Richards 开发的。BCPL 语言对由 Ken Thompson 发明的 B 语言产生着深刻的影响，而 B 语言最终在 20 世纪 70 年代发展成 C 语言。

在 C 语言发明之前，计算机语言被设计出来或者是为了进行学术计算，或者是为官方的委员们所使用。而 C 却不同。它是一群真实的程序员设计、实现并开发出来的。它反映了这些人完成编程工作的方法。它的特性是这群实际使用这个语言的人们反复推敲，打磨，测试的结果。因此，C 语言吸引了众多的拥护者，并迅速成为全世界编程人员的选择。

C 语言的发展经历了 20 世纪 60 年代的结构化编程的革命。在此之前，大型程序是难以书写的，因为程序的逻辑趋向于退化成“意大利面条式的代码”，充斥着难以理解的，混乱的

转跳，函数调用和返回。结构化编程通过增加很好的控制语句，带有局部变量的子程序段和其它的一些改进解决了这个问题。结构化编程使得编写巨大的程序成为了可能。尽管还有别的结构化的编程语言，例如 Pascal，C 却是第一个功能强大，富于表达，能书写出优美代码的结构化语言。它语法简单易用，并且它的设计思想是程序员掌控一切，而不是语言本身掌控一切，这就使得 C 语言很快拥有了众多的拥护者。我们现在来看这点可能有点难以理解，但是 C 当时确实为编程者带来了他们渴望已久的新鲜空气。因此，C 语言很快就在 20 世纪 80 年代变成了使用最广泛的结构化编程语言。

我们需要 C++

经过前文的描述，你可能会问，那为什么还要发明 C++呢？既然 C 是很优秀的编程语言，我们为什么还需要别的编程语言呢？这个问题的答案非常复杂。纵观编程技术的发展历史，程序复杂度的增加驱使我们需要更好的方式来管理我们的程序。C++就应运而生了。为了更好地理解增长的程序复杂度与计算机编程语言发展之间的关系，我们需要简单回顾一下计算机编程技术发展的历史。当计算机刚被发明出来的时候，人们使用计算机前面的面板，通过拨动开关来发送二进制的机器指令。这种方式在计算机程序只有几百行的时候还可以工作。随着计算机程序的增大，人们发明了汇编语言，通过使用符号代替机器指令，以便程序员可以处理更大的，更复杂的程序。第一个被广泛使用的计算机语言是 FORTRAN。FORTRAN 语言在起初给人的印象是非常深刻的，当时几乎没有语言能实现编写整洁，易于理解的程序。20 世纪 60 年代，结构化编程诞生了，这正是诸如 C 语言一样的语言所鼓励的编程方法。通过结构化的编程方法，很轻松的编写大型程序第一次成为了可能。然而，即使是使用结构化的编程方法，一旦一个项目到达了一定的规模，其复杂度也就超过了程序员所能管理的范围。在 20 世纪 70 年代，很多项目几乎都处于这种境地。为了解决这种问题，出现了一种新的编程方法：面向对象编程。通过使用面向对象编程，程序员可以处理更大的，更复杂的程序。而 C 语言是不支持这种面向对象编程方法的。于是，人们对面向对象的 C 的渴望就直接导致了 C++的诞生。可见，自从计算机发明以来，编程的方式已经发生了巨大的变化。

最后一点，尽管 C 是世界上最受欢迎的专业编程语言之一，也有复杂的程序是 C 不能完成的。一旦一个程序的规模达到了一定的大小，其复杂度就会增加，以至于很难从整体上对其进行把握。C++的目的就是突破这种障碍，帮助编程人员理解并管理更大，更复杂的程序。

C++的诞生

C++由 Bjarne Stroustrup 于 1979 年在位于新泽西州 Murray Hill 的贝尔实验室成功发明。起初它的名字叫“带有类的 C”，后来在 1983 年更名为 C++。Stroustrup 在 C 的基础上构建了 C++，因此 C++包括 C 的所有特性和优点。它还继承了 C 的理念：程序员而不是程序掌控一切。有一点必须明确，Stroustrup 并没有创建一个全新的编程语言。相反，它增强

了已经高度成功的语言。大多数 Stroustrup 为 C 增加的特性都是为了支持面向对象的编程。从本质上来讲，C++就是支持面向对象的 C。通过在 C 的基础上构建 C++，就实现了到面向对象编程的平滑过渡。C 程序员不必重新学习一门新的语言，只需要学习那些新增的特性，就能收获面向对象编程带来的好处。在设计 C++语言的时候，Stroustrup 清楚地知道在增加支持面向对象编程的特性时，保持原有 C 的特性，包括 C 的高效，灵活和 C 的设计理念是非常重要的。幸运的是，他的目标实现了。C++在提供了面向对象的编程优点同时，还保留了 C 的灵活。尽管发明 C++的初衷是为了辅助管理那些大型的程序，但它绝不仅限于此。实际上，C++的面向对象特性可以被有效地引用到实际上任何程序中。C++可以广泛地被用来开发诸如编辑器，数据库，个人文件系统，网络工具，通信程序等，这些都非常常见。由于 C++保留了 C 的高效性，大量的高性能系统软件都是用 C++开发的。同样，C++也经常被用来开发 windows 程序。

C++的发展

C++被发明后，经过了三次大的修订，每次修订都对语言自身做了增加和改动。第一次和第二修订分别是在 1985 年和 1990 年。第三次修订发生在 C++标准化的过程中。几年前（现在来看，应该是十几年前了），人们开始进行 C++的标准化工作。那时，建立了由美国国家标准研究所（ANSI)和国际标准组织（ISO）合作的标准化组织组。建议标准的第一次草案是在 1994 年 1 月 25 日完成的。在这份草案中，ANSI/ISO 联合委员会保留了 Stroustrup 当初定义的特性，并增加了一些新的特性。总的来说，这份最初的草案反映了当时 C++的情况。在此之后不久，发生了一件事情，促使了联合委员会大大地扩展了该标准：由 Alexander Stepanov 提出的标准模板库的创建。标准模板库是一套我们可以用之处理数据的通用程序的集合。通用模板库很强大，也很简洁优雅。但是它很巨大。在第一次草案之后，委员会曾经投票来决议是否在标准 C++中增加标准模板库。标准模板库的增加使得 C++大大超出了起初定义的范围。对标准模板库和其它一些特性的增加使得 C++标准化的步伐减慢了许多。完全可以说 C++的标准化工作比人们期望的时间要长许多。在整个过程中，C++中加入了许多新的特性，并做了许多小的改动。实际上，由该联合委员会制定的 C++比 Stroustrup 当初设计的 C++要复杂很多。最终的草案在 1997 年 12 月 14 日通过，ANSI/ISO 标准 C++在 1998 年成为现实。这就是人们通常说的标准 C++。本书描述的都是标准的 C++。本书描述的 C++是所有主流 C++编译器，包括微软的 visual C++都支持的 C++。因此本书中的代码和信息是完全可移植的。

必备技能 1.2：C++与 Java 和 C#的关系

除了 C++之外，还有两个重要的现代编程语言：Java 和 C#。Java 是有 Sun Microsystems 公司开发的，而 C#则是由微软公司开发的。由于人们有时会对 C++与 Java 和 C#的关系产生一些混淆，这里有必要对此介绍一下。C++是 Java 和 C#之父。尽管 Java 和 C#都是在

C++的基础上对语言的特性进行了一些增加，删除和改动，但是总体上来说它们三者的语法是几乎相同的。进一步来说，C++所采用的对象模型和 Java，C#的都是相似的。最后，三者给人的总体感觉也是非常相近的。这就意味着，一旦学会了 C++，就能很轻易地学习 Java 和 C#。 反之亦然，如果你懂 Java 或者 C#学习 C++也是很简单的。这就是为什么 Java，C#和 C++都是用相同的语法和对象模型了，这也是大量有经验的 C++程序员能顺地过渡到是 Java 或者 C#的原因。它们之间的区别在于各自设计针对的计算环境不同。C++是针对指定类型的 CPU 和操作系统而设计的高性能的语言。例如：如果你想写在 windows 操作系统下，因特尔奔腾系列的 CPU 上运行的程序，那么 C++是最好的选择。

专家答疑

问：

Java 和 C#都实现了跨平台和可移植的编程，C++为什么不能了？

答：

Java 和 C#之所以能实现跨平台，可移植的编程，而 C++不能是因为它们的编译器生成的目标代码不同。就 C++而言，编译器的输出是机器代码，这是 CPU 可以直接执行的。因此它是紧密和指定的 CPU 以及操作系统相关的。如果想让 C++程序在不同的系统上运行，则需要针对该目标系统进行代码的重新编译。为了让 C++程序可以在不同的环境上运行，就需要生成不同的可执行版本。Java 和 C#是通过把代码编译成伪码，一种中间语言。就 Java 而言，这种伪码是在运行时系统上运行的，这就是 Java 虚拟机。对 C#而言，这就是 CLR（公共语言运行时）。因此，Java 语言的程序可以在任何有 java 虚拟机的环境下运行，C#的程序可以在任何实现了 CLR 的环境下运行。因为 Java 和 C#的运行时系统处于程序和 CPU 之间，和 C++相比，这就引起了多余的开销。这就是为什么，对等情况下，C++程序比 Java 和 C#程序运行快的原因了。Java 和 C#的开发是为了满足互联网上在线程序的统一编程需求。（C#的设计也是用来简化软件构件的开发）。互联网上连接的是许多不同的 CPU 和操作系统。因此跨平台和可移植性就成了最重要的着眼点。第一个着眼于这个问题的语言就是 Java。Java 语言编写的程序可以在很多不同的环境下运行。因此，Java 程序可以在互联网上自由运行。然而这样做的代价就是牺牲了效率，Java 程序的执行要比 C++程序慢许多。同样的事情也发生在 C#身上。最终分析，如果你想开发高性能软件，就是用 C++。如果你想开发高度可移植的软件，就是用 Java 或者 C#。最后一点：请记住，C++，java 和 C#是用来解决不同问题的。这里并没有那个语言好，那个语言不好的问题，而是那个语言更适合用来完成我们手头工作的问题。

练习：

1. C++语言是从什么语言发展而来的？
2. C++语言产生的主要原因是什么？

3. C++语言是 Java 和 C#语言之父，对吗？

答案：

1. C++是从 C 语言发展而来的。
2. 程序复杂性的不断增加是产生 C++语言的主要原因。
3. 正确。

必备技能 1.3：面向对象的编程

C++的中心是围绕着面向对象的编程。正如前面介绍的那样，面向对象的编程是促进 C++产生的主要因素。正是因为这一点，在学习编写简单的 C++程序之前，理解面向对象编程的基本思想是非常重要的。

面向对象的编程很好地利用了结构化编程的思想，并增加了一些新的概念，能够更好地组织程序。通常情况下，有两种方式来组织程序：以代码为中心或者以数据为中心。通过结构化的编程技术，程序通常是以代码为中心来组织的。这种方法可以被认为是“代码作用于数据”。

面向对象的编程则是以数据为中心。程序以数据为中心进行组织，主要的原则就是“数据控制代码”。在面向对象的语言中，我们定义数据和允许作用于这些数据的例行程序段。因此，数据的类型明确定义了可以作用在这些数据上的操作。

为了支持面向对象编程的原则，所有的面向对象语言，包括 C++，都有三个显著的特点：封装，多态和继承。让我们一个一个来学习。

封装

封装是一种编程机制，它把数据和处理数据的代码捆绑在一起，这可以防止外部程序错误地访问数据和代码。在面向对象的语言中，数据和代码可以通过黑盒子的方式绑定起来，盒子内部是全部必要的数据和代码。当数据和代码以这种方式绑定的时候，此时我们就创建了一个对象。换句话说，一个对象就是支持封装的设备。

专家答疑

问：

我听说过子程序段的学名为方法，那么方法和函数是同一回事情吗？

答：

通常来说，是的。方法在 Java 里面使用很广泛。C++则中叫做函数，Java 中则被称作方法。C#程序员同样使用方法这个术语。因为它已经广泛地被采用了，所以有时候在 C++里面，函数有时候也被称作方法。

对于一个对象来说，代码或者数据或者两者都有可能是私有的或者公有的。私有的代码或者数据只能是该对象的其它部分可以访问和感知的。也就是说，私有的代码和或者数据是

不能被对象之外的程序段访问的。当代码或者数据是公有的，虽然它是被定义在对象中的，但是程序的其它部分都是可以访问它的。通常情况下，对象的公有部分是用来提供对对象私有元素的可控的访问接口的。

C++中最基本的封装单元就是类。一个类定义了对象的通用形式。它同时定义了数据和作用于这些数据的代码。C++使用类来构建对象。对象是类的实例。因此，一个类实际上是构建对象的工厂。

类中的数据和代码称为类的成员。更详细一点来说，成员变量，也叫做实例变量，是类定义的数据。成员函数是作用于这些数据的操作。函数在 C++中是针对子程序的术语。

多态

多态，字面意思就是多种形态的意思。它是一种机制，可以允许一个接口来访问类的通用动作。一个简单的多态性的例子就是汽车的方向盘。方向盘（接口）都是一样的，不管实际中使用了怎样的转向机制。也就是说，不管你的汽车使用的是手工的转向盘，还是动力的转向盘，或者是齿轮转向盘，方向盘的工作效果都是一样的：就是把方向盘往左转，车就会左转，不管使用的是什么样子的转向盘。这种统一接口的好处就是一旦我们知道了如何掌控方向盘，我们就可以开什么类型的车。

同样的原理也可以应用在编程上。比如，考虑一个栈，程序可能需要针对三种不同数据类型的栈。一个是针对整型数的，一个是针对浮点型数据的，一个是针对字符的。在这种情况下，栈的实现算法是一致的，只是存储的数据类型是不一样的。在非面向对象的语言中，我们必须创建三个不同的栈的程序，每一个都是用不同的名字。然而，在 C++，通过多态，我们就可以创建一个通用的栈，三种不同的数据类型都可以存储。这样，一旦知道了栈是如何使用的，我们就等于知道了三种不同数据类型的栈的用法。

通常，多态的概念经常被表述如下：一个接口，多种方法。意思是说可以通过对一组相关的活动定义一个通用的接口。多态性通过一个接口来阐明动作的通用性，可以帮助我们减少程序的复杂性。编译器针对应用的情况来选择具体应该是那个动作（方法）。程序员不必手工来做这种选择。我们只需要记住并使用这个通用的接口就可以了。

继承

继承是一个对象可以获取到另外一个对象的属性的过程。继承非常重要，因为它支持了层次化的分类。仔细想想，大多数的知识都是通过层次化的分类来管理的。例如：红色的甜苹果是属于苹果分类的，而苹果分类又是水果分类的一种，水果又是一个更大的类：食物的一种。也就是说，食物有一些特质，比如可以食用，有营养等等，逻辑上来说它的子类水果也是有的。除了这些特质，水果还有特殊的特质，比如多汁，香甜等等，正是这些特质使得水果和其它的食物有所区分。苹果类定义了苹果特有的特质，比如生长在树上，非热带的等等。一个红色香甜的苹果应该继承了它的父类的所有特质，同时也定义了一些使得它和别的

苹果区分开来的特质。

如果没有层次化的分类，每一个对象都必须显示地定义自己的所有特性。通过使用继承，一个对象只需要定义那些使得它在整个类中显得独特的那些特质。它可以从父类那里继承所有通用的特质。因此正是继承机制使得一个对象成为一个通用类的实例成为可能。

专家答疑

问：

文中说到面向对象的编程是一种有效的进行大程序管理的方法。但是面向对象的编程似乎会增加相对小程序的开销。就 C++而言，这点是真是假？

答：

假。理解 C++的关键一点是使用 C++可以编写面向对象的程序，但是并不是说必须写面向对象的程序。这一点是 C++和 Java、C#的一个很重要的不同点。Java 和 C#采用严格的对象模型，所以每个程序，不管其多小，都是面向对象的。C++也提供了该选项。更重要的是，在运行的时候，C++的面向对象特性是透明的，因此，如果存在的话，增加的开销也是很少的。

练习：

1. 说出面向对象编程的原则。
2. C++中，封装的最基本的单元是什么？
3. C++中，子程序通常被称作什么？

答案：

1. 封装，多态和继承是面向对象编程的原则。
2. C++中类是封装的基本单元。
3. C++中，子程序通常被成为函数。

必备技能 1.4：第一个简单的程序

现在我们开始编程了。我们通过编译和运行下面的一个简短的 C++程序开始：

view plain

```
1. /*
2.     这是一个简单的 C++程序.
3.     文件名称为 Sample.cpp.
4. */
5. #include <iostream>
6. using namespace std;
7. // C++程序都是从 main()函数开始的
8. int main()
```

```
9.  {
10.     cout << "C++ is power programming.";
11.     return 0;
12. }
```

我们将按照下面的三个步骤来进行

1.键入程序

2.编译程序

3.运行程序

在开始之前，让我们先复习两个术语：源代码和目标代码。源代码就是人类可读的程序，它是存储在文本文件中的。目标代码是由编译器生成的程序的可执行形式。

键入程序

本书中出现的程序都可以从 Osborne 的网站上获取到：www.osborne.com 。然而，如果你想手工键入这些程序也是可以的。手工键入这些程序通常可以帮助我们记忆这些关键的概念。如果你选择手工键入这些程序，你必须有一个文本编辑器，而不是一个字处理器。字处理器通常存储带有格式的文本信息。这些信息会使得 C++编译器不能正常工作。如果你使用的是 windows 平台，那么你可以使用记事本程序，或者其它你喜欢的程序编辑器。

存储源代码的文件名称从技术上来说是可以任意取的。然而，C++程序文件通常采用.cpp 的扩展名。因此，你可以任意命名 C++的程序文件，但是它的扩展名应该是.cpp。针对这里的第一个例子，源文件取名为 Sample.cpp。本书中的其它程序，你可以自行选择文件名。

编译程序

怎样编译 Sample.cpp 取决于你的编译器和你采用了什么样子的编译选项。更进一步来说，许多编译器，例如你可以自由下载的微软的 Visual C++ Express Edition，都提供两种不同的编译方式：命令行编译和集成开发环境。因此，这里不可能给出一个通用的编译 C++程序的操作。你必须查看与你的编译器相关的操作指南。

如果你采用的是 Visual C++，那么最简单的编译和运行本书中的程序的方式就是采用 VC 中提供的命令行。例如,使用 Visual C++来编译 Sampler.cpp，你可以采用这样的命令行：

C:/...>cl -GX Sample.cpp

其中-GX 是增强编译选项。在使用 Visual C++命令行编译器之前，你必须先执行 visual C++提供的批处理文件 VCVARS32.BAT。Visual Studio 同时也提供了一个很方便使用命令行的方式：可以通过任务栏的 开始|程序|Microsoft Visual Studio 菜单中的工具列表中的 Visual Studio Command Prompt 菜单来激活命令行。

C++编译器的输出是一个可以执行的目标代码。在 windows 环境下，可执行文件的名字和源码文件的名字相同，但是扩展名为.exe。因此，Sample.cpp 的可执行文件为 Sample.exe。

运行该程序

C++程序编译好之后就可以运行了。既然 C++编译器的输出为可执行的目标代码，那么运行这个程序，只需要在命令行提示符下键入这个程序的名字即可。例如,运行 Sample.exe 时，采用如下的命令行

C:/...>Sample

运行时，程序将显示下面的输出：

C++ is power programming.

如果你使用的是集成开发环境，那么你可以从菜单中选择 Run 来运行这个程序。运行的方式也和编译器相关，这点请参考你使用的编译器的操作指南。针对本书中的程序，通常从命令行来编译和运行会简单一些。

最后一点：本书中的程序都是基于控制台模式的，而不是基于 windows 模式的。也就是说，这些程序是运行在命令行提示符下的。C++对 windows 编程也是很内行的。实际上，它是 windows 开发时最常用的语言。然而，本书中所有程序都没有使用 windows 的图形化用户界面。这一点的原因也很好理解：windows 编程本质上是很巨大和复杂的，即使是创建一个最小的 windows 框架程序也需要 50 至 70 行代码，而编写能够展示 C++特性的 windows 程序也需要几百行代码。相比之下，基于控制台的程序更短小，通常都是采用基于控制台的程序来学习的。一旦你掌握了 C++，你就能够很轻松地把学习的知识应用到 windows 编程上了。

逐行剖析第一个简单的程序

尽管 Sample.cpp 相当短小，但是它却涵盖了几个 C++中常用的关键特性。让我们近距离地研究一下这个程序的每个部分。程序以

```
/*
    这是一个简单的 C++程序.
    文件名称为 Sample.cpp.
*/
```

开始，这点是很常见的。就像其它大多数编程语言一样，C++允许程序员对程序的源码进行注释。注释的内容会被编译器忽略掉。注释的目的是给所有读源码的人描述或者解释该程序。就本例子来讲，注释解释了改程序的功能。在更复杂的程序中，注释可以被用来解释程序的每个部分是为什么这样做，以及如何做才能以完成预期的功能。换句说，你可以对程序的功能提供一种详细的描述。

在 C++中，有两种注释的方式。刚才看到的叫做多行注释。这种类型的注释以/*开始， 以*/结束。任何处于这两个注释符号之间的内容都会被编译器忽略掉。多行注释可以是一行或

者多行的。第二种注释会在后续的程序中看到。

接下来的代码是：

#include <iostream>

C++语言定义了几个头文件，这些头文件中定义了或者必要的或者有用的信息。该程序就需要头文件中的 iostream 来支持 C++的输入/输出系统。该头文件是由编译器提供的。程序中通过#include 来包含头文件。在本书后续章节中，我们会学习到更多的关于头文件和为什么要包含头文件的知识。

接下来的代码是：

using namespace std;

这行告诉编译器使用 std 命名空间。命名空间是相对较新的增加到 C++里面的特性。本书后续会对命名空间进行详细的讨论，这里给出一个简单的描述。一个命名空间创建了一个声明域，其中可以放置各种程序元素。在一个命名空间中声明的元素和另外一个命名空间中声明的元素是相互分开的。命名空间可以帮助我们组织大型程序。using 语句通知编译器我们要使用 std 这个命名空间。这是声明了整个标准 C++库的命名空间。通过使用 std 这个命名空间，我们访问标准库。（因为命名空间相对比较新，旧的编译器可能不支持。如果你使用的是旧的编译器，请参见附录 B，其中描述了简单的应急措施）

程序的下一行是：

// C++程序都是从 main()函数开始的

这一行中使用到了 C++中的第二种注释方式：单行注释。单行注释以//开始，在一行的末尾结束。通常情况下，针对较多的，详细的注释使用多行注释，简单的注释一般使用单行注释。当然，这个也完全是个人编程风格的问题。

接下来的一行代码，正如上面的注释一样，是程序执行的入口：

int main()

所有的 C++程序都是由一个或者多个函数组成。正如前面解释的那样，一个函数就是一个子程序。每一个 C++函数都必须有个名字，任何 C++程序都必须有一个 main()函数，就像本例子中的一样。main()函数是程序执行的入口，通常也是程序结束的地方。（从技术上来讲，一个 C++程序通过调用 main()函数而开始，大多数情况下，当 main()函数返回的时候，程序也就结束了）。该行中的一对括号表示 main()函数代码的开始。main()函数前面的 int 表明了该函数的返回值为 int 类型。后面会介绍到，C++支持几种内建的数据类型，其中就有 int。它代表的是一个整型数。

接下的一行是：

cout<<"C++ is power programming.";

这是一条控制台输出语句。它输出：C++ is power programming. 到计算机显示器上。输出是通过使用输出运算符<<来完成的。<<运算符把它右侧的表达式输出到左侧的设备上。cout 是一个预先定义好的标识符，它代表了控制台的输出，通常就是指计算机的显示器。因此，

这句程序就是把信息输出到计算机的显示器上。请注意：这句程序是以分号结束的。实际上，所有的 C++语句都是以分号结束的。信息“C++ is power programing.”是一个字符串。在 C++中，一个字符串就是由双引号引起来的一个字符的序列。在 C++中，字符串是会经常用到的。

下一行是：

return 0;

该行代码终止了 main()函数，并且返回 0 给调用进程（调用进程通常就是操作系统）。对于大多数的操作系统来说，返回值 0 就意味着程序正常终止。其它返回值则代表程序因为产生了错误而终止。return 是 C++的关键字之一，用它来表示函数的返回值。所有的程序在正常终止的情况下（也就是没有出现错误的情况下）都应该返回 0。

程序最后的大括号表示程序到此结束。

处理语法错误

根据以前的编程经验，我们知道，编程时很容易出现键入错误的情况的。庆幸的是，如果我们在键入程序的时候输入错误了，编译器在编译的时候会报告语法错误信息的。大多数的 C++编译器都试图正确理解我们编写的程序。正是由于这个原因，编译器报告的错误并不一定都反映了问题出现的原因。例如，在前面的程序中，如果我们遗忘了 main()后面的大括号，编译器则会报告说 cout 语句有语法错误。当编译器报告语法错误信息的时候，我们就应该检查最新写的那几行代码，看看是否有错误。

专家答疑

问：

除了错误信息以外，编译器还会提供告警信息。告警和错误有什么区别，我应该关注那类信息？

答：

除了报告致命的错误信息以外，大多是的编译器都会上报几种告警信息。错误信息代表的是程序中出现了明确的错误，比如忘记键入分号等。告警信息则是指出现了从技术上讲是正确的，但是值得怀疑的地方。由程序员来决定此处是否正确。

告警可以用来报告诸如使用了已经被废弃的特性或者无效的构成等。通常来讲，我们可以选择那些告警是可见的。本书中的程序是符合标准 C++的，因此只要输入无误，是不会出现令人烦恼的告警信息的。

针对本书中的示例程序，你可能想使用编译器缺省的错误报告。然而，你可以查阅自己编译器的相关文档以明确编译器提供哪些选项。很多编译器都可以帮助我们检查出潜在的错误，不至于引起大麻烦。了解编译器的错误报告机制是很值得的。

练习：

1.C++程序是从哪里开始执行的？

2.cout 是做什么用的？

3.#include<iostream>是用来做什么的？

答案：

1.C++程序从 main 函数开始执行。

2.cout 是预先定义好的标识符，它连接到控制台输出。

3.它用来包含头文件<iostream>，可以支持输入/输出。

必备技能 1.5：第二个简单的程序

变量是程序最基本的构成元素。一个变量就是一个被命名的，可以被赋值的内存位置。进一步来说，变量的取值在程序的执行过程中是可以改变的。也就是说变量的内容是可变的，而不是固定的。

下面的程序创建了一个变量叫做 length，赋值 7，并在显示器上输出信息“The length is 7”。

view plain

```
1. //using a variable.
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6.     int length;     // 这行是声明一个变量
7.     length = 7;     // 给变量赋值为 7
8.     cout << "The length is ";
9.     cout << length;     // 这行输出 7，也就是输出变量 length 的值
10.    return 0;
11. }
```

正如前文所提到的，C++程序的命名是可以任意的。因此，键入这个程序后，你可以选择一个自己喜欢的文件名字来保存该程序。当然，文件可以取名为 VarDemo.cpp。

这个程序引入了两个概念。第一，语句

int length; // 这行是声明一个变量

声明了一个变量，叫做 length，它的类型为整型。在 C++中，所有的变量都必须在使用之前进行声明。而且，变量所能存储的值的类型也必须指定，这叫做变量的类型。所以上面的代码声明的变量 length 是用来存储整型数的。整型数的取值范围为-32768 到 32767。在 C++中，欲声明一个整型的变量，就在它的名称前加上 int。后面会学习到 C++支持更多的内置的变量类型。（还可以自己创建自己的数据类型）

接下来的一行中用到了第二个新的特性：

Length=7; // 给变量赋值为 7

正如注释所解释的那样，这句代码给变量 length 赋值为 7。在 C++中，赋值运算符就是一个单等号。它把右侧的取值拷贝到左侧的变量中。在该赋值语句之后，变量 length 的取值将为 7。

接下来的语句输出了变量 length 的值：

cout << lenght; // 这行输出 7，也就是输出变量 length 的值

通常来讲，如果我们想要输出一个变量的取值，只要把它放置在 cout 语句中<<的右侧即可。针对例子中的情况，因为 length 的取值为 7，因此输出的值就会是 7。在继续学习下面的内

容之前，我们可以尝试给 length 赋于其它的值，并观察输出的结果。

必备技能 1.6：使用运算符

和大多数其它的语言一样，C++支持全部的算术运算符，以便能够处理程序中的数字。这其中就包括：

+ 加法

- 加法

* 乘法

/ 除法

在 C++中，这些运算符的用法和代数中的用法是一样的。

下面的程序使用*运算符来根据长度和宽度计算矩形的面积。

view plain

```
1. //使用运算符
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6.     int length ; // 这行定义了一个变量
7.     int width ; // 这行定义了另外的一个变量
8.     int area;   // 这行也是定义变量
9.     length = 7; // 给变量 length 赋值为 7
10.    width = 5; // 给变量 width 赋值为 5
11.    area = length*width; //计算面积，把 length 和 width 的乘积赋值给 area
12.    cout<<"The area is ";
13.    cout<<area; //输出 35
14.    return 0;
15. }
```

这个程序声明了三个变量：length，width 和 area。给变量 length 赋值为 7，width 赋值为 5。然后计算他们的乘积，并把结果赋值给变量 area。这个程序的输出如下：

The area is 35

在这个程序中实际没有必要声明变量 area。可以按照下面的示范重写一下刚才的程序：

view plain

```
// 计算面积程序的简化版本.
#include <iostream>
using namespace std;
int main ()
{
    int length; //声明一个变量
    int width;  // 声明另外的一个变量
    length =7;//给 lenght 赋值为 7
    width=5; //给 width 赋值为 5a
    cout << "The are is ";
    cout << length * width;//输出 35
    return 0;
}
```

在这段代码中，面积是在 cout 语句中通过把 legth 和 width 做乘法运算而得到的，然后把结果输出到显示屏幕上。

在继续学习之前，我们还应该指出，在同一个声明变量的语句中，我们是可以声明两个或者多个变量的。只需要用逗号把它们分开即可。例如，可以通过下面的方式声明三个变量 length,width,area：

int length,width,area; //使用一条语句声明全部的变量

在专业的代码中，通过一条语句声明两个或者多个变量是非常普遍的。

练习：

1.变量在使用前是否必须先声明？

2.如何给变量 min 赋值 0？

3.在一条声明语句中是否可以声明多个变量？

答案：

1.是的，C++中的变量在使用前都必须声明。

2.min=0;

3.是的。在一条声明语句中可以声明多个变量。

必备技能 1.7：从键盘读取输入

前面的程序中使用的都是显示指定的数据。例如，计算面积的程序计算的是长为 7，宽为 5 的矩形的面积，矩形的尺寸本身就是程序的一部分。然而，不管矩形的尺寸是多少，其面积的计算方法都是一样的。因此，如果能够提示用户从键盘输入矩形的尺寸，然后计算矩形的面积的话，那么这个程序将够有用一些。

我们可以使用>>运算符来使用户从键盘输入数据到程序中。这就是C++中的输入运算符。通常使用下面的形式从键盘获取数据

cin>> var;

其中，cin 是另外一个预先定义好的标识符。它代表控制台输入，这是 C++自动支持的。缺省情况下，cin 是和键盘绑定的，它也可以被重定向到其它的设备上。var 代表接收输入的变量。

下面重写计算面积的程序，允许用户输入矩形的尺寸：

view plain

```
/*
    用来计算矩形面积的交互式程序
*/
#include <iostream>
using namespace std;
int main()
{
    int lenght; // 声明一个变量
    int width;  // 声明另外一个变量
    cout << "Enter the length:";
    cin >> length; //从键盘输入长度
    cout << "Enter the width:";
   cin >> width; //从键盘输入宽度
    cout << "The area is ";
    cout << length * width; //输出面积
    return 0;
}
```

运行的结果可能如下：

Enter the length: 8

Enter the widht: 3

The area is 24

请注意下面的几行：

cout << "Enter the length:"

cin >> length; //从键盘输入长度

cout 语句提示用户输入数据。cin 语句读取用户输入的数据，并把值存储在变量 length 中。于是，用户键入的数值（就本例中的程序，用户必须输入一个整型数）就被存放在了>>右侧的变量中（本例中就是 length）。在执行完毕 cin 语句后，变量 length 存放的就是矩形的长度（如果用户键入的是非数字，变量 lenght 的值将会是 0）。提示用户输入宽度和从键盘读取矩形长度的语句工作原理是一样的。

一些输出选项

到目前为止，我们一直使用的都是 cout 的最简单的形式。然而，cout 可用来输出更复杂的语句。下面是两个有用的技巧。首先，我们可以使用一个 cout 语句输出多条信息。比如，在计算面积的程序中，我们使用下面的两行来输出面积：

```
cout << "The area is ";
cout << length * width;
```

这两行代码可以使用下面更方便的语句来表达：

```
cout << "The ares is " << length * width;
```

这种方式在同一个 cout 语句中使用了两个输出运算符，这将在输出字符串"Ther area is "后接着输出面积。通常情况下，我们可以在一条输出语句中连接多个输出运算符，每个输出项单独使用一个运算符即可。

第二个技巧，到目前为止我们还没有换行输出，也就是回车。但是不久我们就需要这样输出了。在字符串中我们使用"/n"表示换行，试试下面的程序就可以看到"/n"的效果了。

view plain

```
/*
    这个程序演示了/n 的用法，可以输出换行
*/
#include <iostream>
using namespace std;
int main()
{
    cout << "one/n";
    cout << "two/n";
    cout << "three";
    cout << "four";
    return 0;
}
```

这段程序的输出如下：

```
one
two
threefour
```

换行字符可以被放置在字符串中的任意位置，并不一定是放置在最后。在理解了换行字符的作用后，就可以自己写程序看看结果了。

练习：

1.C++中的输入运算符是哪个？

2.缺省情况下，cin 是和那个设备进行绑定的？

3./n 代表什么？

答案

1.输入运算符是>>

2.c 缺省地是和键盘绑定。

3./n 代表换行字符。

其它的数据类型

在前面的程序中，我们使用的都是 int 类型的变量。int 类型的变量只能用来存储整型数。当需要存储小数的时候，int 类型的变量就不能使用了。比如，一个 int 类型的变量可以用来存储值 18，但是不能用来存储值 18.3。幸运的是，int 类型只是 C++中定义的几种数据类型之一。C++定义了两种浮点类型来表示小数：float 和 double 类型，它们分别代表单精度和双精度的小数。其中，double 可能是最常用的了。

可以采用类似下面的方式来声明一个 double 类型的变量：

```
double result;
```

这里，result 是变量的名称，它的类型是 double 类型的。因为它的类型是浮点类型，因此可以被用来存放诸如 88.56 或者-107.03 之类的数据。

为了更好地理理解 int 和 double 类型的区别，可以试一试下面的程序:

view plain

```
/*
这个程序展示了 int 类型和 double 类型的区别
*/
#include <iostream>
using namespace std;
int main()
{
    int ivar; //声明一个整形的变量
    double dvar; //声明一个浮点型的变量
    ivar = 100;//给 ivar 赋值 100
    dvar =100.0;//给 dvar 赋值 100.0
    cout<< "Original value of ivar: " << ivar << "/n";
    cout<< "Original value of dvar: "<< dvar << "/n";
    cout<< "/n"; //打印一个空行
    //各自都除以 3
    ivar=ivar/3;
```

```
17.     dvar=dvar/3.0
18.     cout << "ivar after division: " << ivar << "/n";
19.     cout << "dvar after division: " <<dvar <<"/n"
20.     return 0;
21. }
```

程序的输出如下：

Original value of ivar: 100

Original value of dvar: 100

ivar after division: 33

dvar after division: 33.3333

从上面的例子可以看出，当 ivar 除以 3 的时候，得到的结果是 33，小数部分丢失了。而 dvar 除以 3 后，小数部分是被保留的。

该程序中还有一个新的需要注意的地方：

cout << "/n" //输出一个空行

这句将输出一个空行。可以在需要任何需要输出空行的地方使用该语句。

专家解答

问：

为什么 C++中用两种不同的数据类型来分别表示整型数和浮点数了？也就是说，为什么不是所有的数据都仅仅使用一个类型了？

答：

C++提供了不同的类型是为了程序的效率更高。比如，整形数的运算要比浮点数的运算快。因此，如果不需要小数，就没有必要引入 float 或者 double 类型带来的开销。还有就是，各种类型的数据在存储的时候所需的内存的大小也是不一样的。通过支持不同的数据类型，C++使得我们可以最好地利用系统的资源。最后，一些算法是需要一些特定类型的数据的。C++提供了大量的内置类型以提供最好的灵活性。

项目 1-1 英尺到米的转换

虽然前面的几个程序展示了 C++语言的几个重要特性，但是它们并不实用。尽管到目前为止我们对 C++的学习并不多，但是我们仍然可以利用所学习的知识来创建一个实用的程序。在该项目中，我们创建一个程序来把英尺转换成米。该程序提示用户输入英尺数据，然后显示出转换后的米的数据。

一米大约等于 3.28 英尺，因此我们需要使用浮点数。为了完成转换，程序中声明了两个浮点类型的变量，一个用来存储英尺的数据，一个用来存储米数据。

步骤：

1. 创建一个 C++文件，命名为 FtoM.cpp。（注意：在 C++中文件的名字是可以任意的，你可以取任何自己喜欢的名字。）

2. 文件以下面的内容开始。这些内容解释了该程序的功能，包含了 iostream 头文件，指明了 std 命名空间。

view plain

```
/*
    Project 1-1  该程序实现从英尺到米的转换
    程序文件命名为 FtoM.cpp
*/
#include <iostream>
using namespace std;
```

3.main()函数中声明变量 f 和 m：

view plain

```
int main()
{
    double f; //存储英尺的长度
    double m; // 存储转换后的米的数据
```

4.增加输入英尺数据的代码：

view plain

```
cout << "Enter the legth in feet:";
cint>>f; //读取英尺数据
```

5.增加进行转换和输出的代码：

view plain

```
m= f / 3.28;//转换成米
cout << f << " feet is " << m << " meters.";
```

6.以下面的代码结束程序：

view plain

```
return 0;
```

7. 程序完成后应该是下面的这个样子

view plain

```
/*
    Project 1-1  该程序实现从英尺到米的转换
    程序文件命名为 FtoM.cpp
*/
#include <iostream>
using namespace std;
int main()
{
    double f; // holds the length in feet
    double m; // holds the conversion to meters
    cout << "Enter the length in feet: ";
    cin  >> f; // read the number of feet
    m = f / 3.28; // convert to meters
    cout << f << " feet is " << m << " meters.";
    return 0;
}
```

8. 编译并运行该程序，举例的输出结果如下：

Enter the length in feet: 5 5 feet is 1.52439 meters.

9. 可以尝试着输入其它的数据，也可以尝试修改成把米转换成英尺。

练习：

1.在 C++中整型类型的关键字是什么？

2.double 是什么意思？

3.怎样才能输出新的一行？

答案：

1.整形数据类型为 int

2.double 是用来表示双精度的浮点数的。

3.输出新的一行，使用\n.

必备技能 1.8：两条控制语句

在一个函数内部，语句是按照从上到下的顺序执行。然而，我们可以通过使用 C++支持的各种程序控制语句来改变程序的顺序执行方式。尽管我们会在后文中仔细研究控制语句，在这里我们还是要简单介绍两种控制语句，我们可以使用它们来写几个简单的示例程序。

if 语句

我们可以使用 C++的条件语句选择性地执行程序的部分代码：if 语句。C++中的 if 语句

和其它程序语言中的 if 语句是很类似的。例如，在语法上 C++中的 if 语句和 C，Java，C# 中的都是一样的。它的最简单的形式如下：

if (condition) statement;

这里的 condition 是一个待评估的条件表达式，它的值或者是 true（真）或者是 false（假）。在 C++中，true 就是非零的数值，而 false 就是 0。如果 condition 为真，则 statement 部分将会被执行。如果 condition 为假，statement 部分将不会被执行。例如，下面的代码片段将在显示器上显示出片语：10 is less than 11，因为 10 是小于 11 的。

if(10 < 11) cout << " 10 is less than 11";

再考虑下面的代码片段：

if (10 > 11) cout << " this dose not display";

在上面的片段中，10 并不是大于 11 的，所以 cout 语句并不会被执行。当然，if 语句中的条件表达式中不一定都必须使用常量。它们也可以是变量。

C++中定义了全部的关系运算符，它们可以被用在条件表达式中。如下：

运算符 含义

< 小于

<= 小于或者等于

> 大于

>= 大于或者等于

== 等于

!= 不等于

注意，等于是两个等号。'

下面是一个示例程序，它展示了 if 语句的用法：

view plain

```
1. // Demonstrate the if
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6.     int a,b,c;
7.     a = 2;
8.     b = 3;
9.     if ( a < b ) cout << " a is less than b /n";  //这里是一条 if 语句
10.    //下面的语句将不能显示任何东西
11.    if ( a == b ) cout << " you won't see this/n"
12.    cout << /n;
13.
14.    c = a - b; //c 的值为-1
15.    cout << " c contains -1/n";
```

```
16.     if ( c >= 0 ) cout << "c is no-negative/n";
17.     if ( c < 0 ) cout << " c is negative/n";
18.     cout <<"/n";
19.
20.     c = b- a; // c 的值现在为 1
21.     cout << " c contains 1/n";
22.     if ( c>= 0 ) cout << "c is non-negativa/n";
23.     if ( c < 0 ) cout << "c is negative/n";
24.     return 0;
25. }
```

这个程序的输出如下：

a is less than b

c contains -1

c is negative

c contains 1

c is non-negative

for 循环

通过循环，我们可以反复执行一系列的语句。C++提供了多种不同的方式来支持循环。这里我们将要看到的是 for 循环。如果你很熟悉 Java 或者 C#，你会很高兴地看到，C++中的 for 循环和这两个语言中的 for 循环是一样的。for 循环最简单的形式如下：

for (initialization；condition；increment） 语句）

这里 initialization 用来设置循环控制变量的初始值。condition 是每次重复执行的时候需要进行的检查条件，只要条件为真，循环就一直继续。increment 是一个表达式，它表示每次循环的时候，循环的控制变量时如何递增的。

下面的程序演示了 for 循环的使用方法。它在屏幕上输出数字 1 到 100。

view plain

```
1.  // A program that illustrates the for loop.
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      int count;
7.      for ( count = 1; count<=100;count=count+1)
8.          cout << count <<" ";
9.      return 0;
10. }
```

在上面的循环中，count 初始化为 1。循环每执行一次，条件 count <= 100 就会被检查一次。如果条件为真，则输出 count 的取值，并且 count 增加 1。当 count 的值大于 100 的时候，检测的条件就为假了，循环终止了。在专业的代码中，是永远不会看到类似下面的语句的：

count = count + 1;

因为 C++中包含了一个特殊的自增运算符来更有效的完成上面的功能。这个自增运算符就是++，也就是两个连续的加号。自增运算符使得运算数的值增加 1。前面的 for 语句通常会被写成下面的形式：

for(count=1; count <= 100; count++) cout << count << " ";

本书的后续章节将都会采用这种书写的形式。

C++中同时提供了自减的运算符：--，它使得运算数的值减一。

练习：

1.if 语句是用来做什么的？

2.for 语句是用来做什么的？

3.C++中有哪些关系运算符？

答案：

1.if 语句是 C++中的条件语句。

2.for 语句是用来做循环用的。

3.C++中的关系运算符有：==，!=, <，>，<=,>=

必备技能 1.9 ： 使用代码块

C++中一个基本的元素就是代码块。一个代码块由两个或者多个语句组成，由一对花括号括起来的。代码块可以作为一个逻辑单元出现在任何单条语句可以出现的地方。例如，代码块可以被使用在 if 语句或者 for 循环中：

view plain

```
1. if ( w < h )
2. {
3.     v = w * h;
4.     w = 0;
5. }
```

这里，如果 w 小于 h，则代码块中的两条语句都会被执行。因此，代码块中的两条语句形成了一个逻辑单元，要么两条语句都会被执行，要么两条语句都不会被执行。凡是需要把两条或者多条语句在逻辑上进行关联的地方，我们都可以使用代码块。代码块使得许多算法实现

起来更加清晰和有效。

下面的程序中使用了代码块来防止除数为 0。

view plain

```
1.  //演示代码块的示例程序
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      double result,n,d;
7.      cout << "Enter value: ";
8.      cin >> n;
9.      cout << "Enter divisor: "
10.     cin >> d;
11.     //the target of this if is a block
12.     if ( d!= 0 )
13.     {
14.         cout << "d does not equal zero so division is OK" << "\n";
15.         result = n / d;
16.         cout << n << " / " << d << " is " << result;
17.     }
18.     return 0;
19. }
```

一个示例的输出结果如下：

Enter value: 10

Enter divisor: 2

d does not equal zero so division is OK

10 / 2 is 5

在上面的这个例子中，if 语句的目标不是一条语句，而是一个代码块。如果 if 语句的控制条件为真，代码块中的三条语句都会被执行。可以尝试输入除数为 0，并观察程序的输出结果。此时，代码块中的三条语句都不会被执行。

在本书的后面的的章节中，我们会看到代码块还有别的属性和用法。然而，代码块存在的主要原因是它能够创建在逻辑上不可分割的独立单元。

专家答疑

问：

使用代码块是否会降低运行时的效率？也就是说代码块中的大括号是否会在代码执行的过程中消耗运行时间？

答：

不会的。代码块在运行的时候并不会增加任何的开销。实际上，由于它能简化一些特性算法的编码，使用代码块通常会增加运行的速度和效率。

分号和位置

在 C++中，分号代表着语句的结束。也即是说，每一个单独的语句都必须以分号结束。通过前面的介绍，我们知道，代码块是在逻辑上关联的一组语句的集合，它由一对花括号括起来。代码块不是以分号作为结束的。既然代码块是一组语句的集合，其中每条语句都是以分号结束，所以代码块不是以分号结束是有意义的。代码块的结束是通过}来标记的。

C++中，并不是一行的结束就代表着一条语句的结束，只有分号才能代表语句的结束。所以，我们把语句写在了哪一行并不是很重要。例如 ，在 C++中

```
x = y;
y = y + 1;
cout << x << " " << y;
```

和下面的代码是一样的

```
x = y; y = y + 1; cout << x << " " << y;
```

更进一步，语句的单独元素也可以被放置在不同的行中。例如，下面的代码也是可以接受的:

```
cout << "This is a long line. The sum is : " << a + b + c +
        d + e + f;
```

采用上面的方式避免一行中写过多的代码可以增加代码的可阅读性。

缩进

在前面的例子中，可以看到一些语句采用了缩进的格式。C++是一个很自由的语言，也就是说，在一行中，把相关的代码放置在了行的什么地方并不要紧。然而，人们已经形成了常用的和可以接受的缩进风格，方便阅读程序。本书将遵循这样的缩进风格。并建议读者也这样做。在这种风格中，针对每个{ 后面的代码都需要缩进一个级别，在}之后代码的缩进格式需要向前提升一个级别。还有一些语句将采用另外的缩进方式，本书后续会进行描述。

练习

1.如何创建代码块？它是用来做什么用的？

2.在 C++中语句块是以_____结束的。

3.所有的 C++语句都必须在同一行开始和结束，对吗？

答案：

1.代码块是以{开始，以}结束的。它是用来创建逻辑单元的。

2.分号。

3.错误。

项目 1-2 生成一张从英尺到米的转换表

这个项目将显示一张从英尺到米的转换表，其中用到了 for 循环，if 语句和代码块。表格的起始为一英尺，终止为 100 英尺。每输出 10 英尺的转换表后，输出一个空行，这个是通过使用变量 counter 来实现的。边玲 counter 是用来表示行数的，请注意它的用法。

步骤：

1.创建一个新的文件，命名为 FtoMTable.cpp。

2.文件中键入下面的程序：

view plain

```
#include <iostream>
    using namespace std;
    int main()
    {
        double f; // 用来存储英尺长度
        double m; //用来存储转换后的米的值
        int counter;
        counter = 0; // 用于统计行数，变量初始化为 0
        for ( f =1.0; f <= 100; f++)
        {
            m = f /3.28; //英尺转换为米
            cout << f << " feet is " << m << " meters.\n";
            //循环每次都要把行数增加 1
            counter++;
            //每 10 行输出一个空行
            //如果行数为 10 了，则需要输出一个空行
            if ( counter == 10 )
            {
                cout << "\n"; //输出空行
                counter = 0;  //重置行数计数器
            }
        }
        return 0;
    }
```

3.注意我们是如何使用 counter 变量来实现每 10 行输出一个空行的。在 for 循环外该变量被初始化为 0。在 for 循环体内部，每做一次英尺到米的转换，counter 就增加 1。当 counter 增加到 10 的时候，就输出一个空行，然后重置 counter 的值为 0，重新开始上面的过程。

4.编译并运行上面的程序。下面是输出的一部分。注意程序输出的小数部分并不是很整齐。

```
1 feet is 0.304878 meters.
2 feet is 0.609756 meters.
```

3 feet is 0.914634 meters.
4 feet is 1.21951 meters.
5 feet is 1.52439 meters.
6 feet is 1.82927 meters.
7 feet is 2.13415 meters.
8 feet is 2.43902 meters.
9 feet is 2.7439 meters.
10 feet is 3.04878 meters.

11 feet is 3.35366 meters.
12 feet is 3.65854 meters.
13 feet is 3.96341 meters.
14 feet is 4.26829 meters.
15 feet is 4.57317 meters.
16 feet is 4.87805 meters.
17 feet is 5.18293 meters.
18 feet is 5.4878 meters.
19 feet is 5.79268 meters.
20 feet is 6.09756 meters.

21 feet is 6.40244 meters.
22 feet is 6.70732 meters.
23 feet is 7.0122 meters.
24 feet is 7.31707 meters.
25 feet is 7.62195 meters.
26 feet is 7.92683 meters.
27 feet is 8.23171 meters.
28 feet is 8.53659 meters.
29 feet is 8.84146 meters.
30 feet is 9.14634 meters.

31 feet is 9.45122 meters.
32 feet is 9.7561 meters.
33 feet is 10.061 meters.
34 feet is 10.3659 meters.

35 feet is 10.6707 meters.

36 feet is 10.9756 meters.

37 feet is 11.2805 meters.

38 feet is 11.5854 meters.

39 feet is 11.8902 meters.

40 feet is 12.1951 meters.

5.可以自己尝试修改上面的程序为每 25 行输出一个空行。

必备技能 1.10：引入函数

C++程序通常是由多个叫做函数的代码块构成的。尽管我们会在第五章对函数进行详细的介绍，这里也有必要进行一个简单的介绍。函数的定义如下：一个包含了一条或者多条C++语句的子程序。

每个函数都有一个名字，通过名字来调用该函数。调用函数的时候，需要在自己的程序的源码中指定函数的名字，后面紧跟着函数的参数。例如，有个函数名字为 MyFunc。如下代码展示了如何调用该函数：

```
MyFunc();
```

当调用一个函数的时候，程序的控制就转移到了被调用函数中，函数中的代码将被执行。当函数中的代码执行完毕后，程序的控制又转回给函数的调用者。因此，函数是为一个程序的其它部分完成相应功能的。

有些函数需要一个或者更多的参数，我们在调用的时候需要传入这些参数给函数。因此，一个参数就是我们传入到函数中的值。在调用函数的时候，参数是放置在括号中的。例如，如果函数 MyFunc()需要一个整型数作为参数，那么下面的代码就是在调用函数的时候传入了参数 2：

```
MyFunc(2);
```

当函数需要两个或者更多参数的时候，这些参数之间用逗号来分隔。在本书中，术语参数列表指的就是用逗号分隔的参数。注意，并不是所有的函数都需要参数。如果函数不需要参数，则括号中为空。

函数可以为调用者返回一个值。并不是所有的函数都需要返回值的，但大多数都是需要返回值的。函数的返回值可以被赋值给调用者中的一个变量，这是通过把对函数的调用放置在赋值语句的右边来实现的。例如，如果函数 MyFunc()返回一个值，那么可以通过下面的方式来调用它：

```
x=MyFunc(2);
```

首先，函数 MyFunc()被调用。当它返回的时候，它的返回值被赋值给 x。还可以在表到式中调用函数。例如，

```
x=MyFunc(2)+10;
```

在这中情况下，函数 MyFunc()的返回值加上 10 后被赋值给 x。通常情况下，当在语句中遇到函数的名字，该函数就会自动被调用，以便获取它的返回值。

复习一下，一个参数就是调用函数的时候传入的值。返回值是函数返回给调用者的值。下面是一个简短的小程序，它展示了如何调用函数。其中使用了 C++中内置的函数，叫做 abs()来计算一个数的绝对值。abs()函数需要一个参数，把这个参数转换成它的绝对值，并返回结果。

view plain

```
1. // 使用 abs() 函数
2. #include <iostream>
3. #include <cstdlib>
4. using namespace std;
5. int main()
6. {
7.     int result;
8.     resutl = abs(-10); //调用函数 abs(),将其返回值赋值给变量 result
9.     cout << result;
10.    return 0;
11. }
```

这里，传入到 abs()函数中的值为-10。abs()函数在调用的时候接收传入的参数-10，并返回 -10 的绝对值，也就是 10 给调用者。这个指被赋值给变量 result。因此，该程序在屏幕上显示 10。

关于上面这个程序需要注意的另外一个地方：它包含了头文件 cstdlib。这是 abs()函数需要的头文件。一旦我们使用了内置的函数，我们必须包含它的头文件。

通常情况下，我们的程序需要两种类型的函数。第一种就是我们自己写的函数，main()就是其中之一。后面，我们会学习到如何自己编写其它的函数。我们会看到，一个真实的 C++程序实际上包含了许多自己编写的函数。

第二种就是编译器提供的函数。前面例子中的 abs()函数就是一个这样的函数。我们自己写的程序通常是即包含自己编写的函数也包含编译器提供的函数。

当在本文中提及函数的时候，本书已经也即将继续使用 C++程序中的传统表达函数的方式：就是函数名字后面跟一对括号。例如有个函数的名字为 getval,在书写的时候会书写成 getval()。本书中的这种写法是为了把变量的名字和函数的名字区分开来。

C++库

正如前面解释的那样，abs()函数是 C++编译器提供的。这个函数是在 C++的标准库中提供的。C++标准库还提供了许多其它的函数。本书中的程序都会用到库函数。

C++的标准函数库中定义了大量的函数。这些函数完成了很多必要的功能，包括输入输出操作，数学运算和字符串处理。当使用库函数的时候，C++编译器自动把这些函数的目标

代码链接到我们自己程序的目标代码中去。

因为 C++标准库很庞大，它已经包含了许多我们在自己程序中需要的函数。这些函数就像盖房子时候的砖块，我们仅仅需要把它们装配起来就可以了。我们可以仔细阅读编译器的函数库的文档，看看编译器都提供了那些有用的函数。我们会惊奇地发现这些函数是如此的多样。如果我们自己编写了一个自己反复使用的函数，这个函数也可以存储在库中。

除了提供库函数外，每个 C++编译器还提供了类库，这个就是面向对象的库。然而，在我们学习了类和对象以后，我们才能使用类库。

练习

1. 什么是函数？
2. 函数是通过使用它的名字来调用的，对吗？
3. 什么是 C++标准函数库？

答案

1. 一个函数就是包含了一条或者多条语句的子程序。
2. 正确
3. C++标准库就是所有 C++编译器提供的函数的集合。

必备技能 1.11：C++中的关键字

标准 C++目前定义了 63 个关键字，如表格 1-1 所示。它们和正式的语法一起形成了 C++语言。早期的 C++版本定义了 overload 关键字，但是现在他已经被废弃了。请记住 C++是一个大小写敏感的语言，并且要求所有的关键字都必须是小写的。

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespac
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void

volatile	wchar_t	while	

表格 1-1 C++中的关键字

必备技能 1.12：标识符

在 C++中，一个标识符可以是一个函数、变量或者任何其它用户自定义项目的名字。标识符可以是一个到几个字符的长度。变量的名字可以是以字母表中任意字符开始或者以下划线开始。后面接着字母，数字或者一个下划线。使用下划线可以增强变量名字的可读性，诸如变量 line_count。大写字母和小写字母是有区分的；也就是说，在 C++中，myvar 和 MyVar 是两个不同的名字。还有一条重要的标识符的规则：任何关键字都不能作为标识符的名字。另外预先定义好的标识符，例如 cout 是不受这个限制的。下面是一些合法的标识符实例

Test	x	y2	MaxIncr
up	_top	my_var	simpleInterest23

记住，标识符不能以数字开头。因此 98OK 是无效的标识符。好的编程实践告诉我们，标识符的名字应该能够反映该标识符的含义或者用法。

练习

1. 哪个是 C++中的关键字，for，For 还是 FOR？
2. 一个 C++中的标识符可以含有那种类型的字符？
3. 标识符 index21 和 Index21 是否是同一个标识符？

答案

1. for 是 C++中的关键字。在 C++中所有的关键字都是小写的。
2. 一个 C++中的标识符可以含有字母，数字和下划线。
3. 不是。C++是大小写敏感的语言。

复习题

1. 有人说 C++是现代编程的核心。请解释这句话。
2. C++编译器编译出计算机可以直接执行的代码，对吗？
3. 面向对象编程的三个主要原则是什么？
4. C++程序从哪里开始执行？
5. 什么是头文件？
6. <iostream>是什么？下面的代码是用来做什么的？

```
#include <iostream>
```

7. 什么是命名空间？
8. 什么是变量？
9. 面那个/些变量的命名是不合法的？

a. count

b. _count

c. count27

d. 67count

e. if

10. 如何创建单行的注释？如何创建多行的注释？

11. if 语句的常用形式是怎样的？for 循环的常用形式是怎样的？

12. 如何创建代码块？

13. 在月球上，重力大约为地球的 17%。编写一个程序，显示一张地球的磅重量对应的在月球上的重量。表格从 1 开始到 100 结束，每 25 磅输出一个空行。

14. 木星上的一年（就是木星围绕太阳旋转一周需要的时间）大约为地球上的 12 年。编写一个程序把木星年转换成地球年，由用户指定木星年，允许出现小数的年。

15. 当调用函数的时候，程序控制会发生什么变化？

16. 编写一个程序，用来计算用户输入的 5 个数据的绝对值的平均值，显示其结果。

第二篇 数据类型和运算符简介

编程语言的核心就是它的数据类型和运算符。这些元素定义了语言的极限和语言可以完成的功能。正如我们所期望的那样，C++支持大量的数据类型和运算符，这使得它在很多领域都是很合适的编程语言。数据类型和运算符的内容很丰富。这里，我们将以 C++中最基本的数据类型和最常用的运算符开始学习。我们还会进一步研究变量和表达式。

为什么数据类型如此重要

数据类型如此重要是因为它决定了可以使用的运算符和可以存储的数值的范围。C++中定义了几种数据类型，每种都有各自的特点。由于数据类型不一样，所有的变量在使用之前都必须进行声明。变量的声明包括指定变量的类型。编译器需要这个信息来生成正确的代码。在 C++中没有所谓的“没有类型”的变量。

数据类型如此重要的第二个原因是因为这几种基本的类型是和计算机操作的基本对象紧密相关的：字节和字。因此 C++允许我们操作的数据类型是和 CPU 直接操作的类型一样的。这也是为什么 C++能够编写出高效的、系统级的代码的原因之一。

基本技能 2.1: C++中的数据类型

C++提供的内置数据类型是对应于整型，字符，浮点和布尔类型的值。这也是程序中通常存储和处理数据的方式。在本书后面的章节中会看到，C++允许我们构建更复杂的类型，比如类，结构，枚举，但是它们完全是由内置的类型所构成。

C++类型系统的核心就是下面的 7 种基本的数据类型

类型	意义
char	Character 字符
wchar_t	Wide Character 宽位字符
int	Integer 整型数
float	Floating Point 单精度浮点数
double	Double Floating Point 双精度浮点数
bool	Boolean 布尔类型
void	Valueless 空类型

C++允许一些基本数据类型可以被修饰符修饰，修饰符放置在类型的前面。修饰符改变了基本类型的含义，使得它能满足不同的需要。数据类型的修饰符如下：

signed

unsigend

long

short

修饰符 signed、unsigned、long、short 都可以用于修饰 int。修饰符 signed 和 unsigned 可以用来修饰 char 类型。类型 double 可以被修饰符 long 来修饰。表格 2-1 显示了所有有效的基本类型和修饰符的组合。这张表同时还给出了 ANSI/ISO C++中定义的每种类型的最小取值范围。

类型	最小取值范围
char	-127 到 127
unsigned char	0 到 255
signed char	-127 到 127
int	-32767 到 32767
unsigned int	0 到 65535
signed int	和 int 的一样
short int	-32767 到 32767
unsigned short int	0 到 65535
signed short int	和 short int 的一样
long int	-2147483647 到 2147483647
signed long int	和 long int 的一样
unsigned long int	0 到 4294967295
float	1E-37 到 1E+37 6 位精度
double	1E-37 到 1E+37 10 位精度
long double	1E-37 到 1E+37 10 位精度

表格 2-1 ANSI/ISO C++标准中定义的所有数字的数据类型和它们的最小取值范围

注意上表中的最小取值范围仅仅是最小的取值范围。C++编译器是允许对这些最小的取值范围进行扩展的，事实上大部分编译器都进行了扩展。因此 C++数据类型的取值范围是和实现相关的。比如，在使用二进制补码的计算机上（几乎所有的计算机都是使用二进制补码的），一个整型数的取值范围至少是 -32768 到 32767。但是无论在什么环境下，short int

的取值范围都是 int 类型的子域，而 int 类型的取值是 long int 的子域。针对 float，double 和 long double 也是一样的。这里子域的意思是说范围小于或者相等。因此，int 和 long int 是可以有相同的取值范围的，但是 int 类型的取值范围不能大于 long int 类型的。

既然 C++明确的只是数据类型必须支持的最小范围，我们应该阅读自己使用的编译器的文档，以便明确实际支持的范围是多少。例如，表格 2-2 显示了在 32 位环境下 C++中数据类型的典型位宽，windows XP 就是这样的环境。让我们了仔细看看吧。

类型	位宽	典型的取值范围
char	8	-128 到 127
unsigned char	8	0 到 255
signed char	8	-128 到 127
int	32	-2147483648 到 2147483647
unsigned int	32	0 到 4294967295
signed int	32	-2147483648 到 2147483647
short int	16	-32768 到 32767
unsigned short int	16	0 到 65535
signed short int	16	-32768 到 32767
long int	32	和 int 的一样
signed long int	32	和 singned int 一样
unsigned long int	32	和 unsigned int 一样
float	32	1.8E-38 到 3.4E+38
double	32	2.2E-308 到 1.8E+308
long double	64	2.2E-308 到 1.8E+308
bool	N/A	true 或者 false
wchat_t	16	0 到 65535

表格 2-2 在 32 位环境下 C++中数据类型的典型的位宽和取值范围

整型数

正如我们在第一篇中所学习到的那样，int 类型的变量存储的是不需要小数部分的整型数。这种类型的变量通常被用来做 for 循环的控制变量以及用于条件语句中，还有就是用来计数。因为其中不含有小数部分，因此整型数的运算要比浮点数的运算快很多。

因为整型数对于编程来说非常重要，所以 C++中定义了多种类型的整型类型。就像在表格 2-1 中的那样，有短整型，正规的整型和长整型三种类型。更重要的是每种类型都有有符号和无符号之分。有符号的整型可以存储正数和负数。缺省情况下，整型数都是有符号的。因此在 int 前面加上 signed 通常是多余的，但是这种写法是合法的。无符号的整型数只能存储正数。在需要创建无符号的整型数的时候，就需要使用 unsigned 修饰符。

有符号和无符号整型数的区别在于整型数的最高位的解释是不一样的。针对有符号的整型数编译器生成的代码会把最高位解释成整型数的符号位。如果这个符号位是 0，则表示这个数是一个正数；如果为 1，则表示这个数是个负数。负数通常总是采用补码的方式来表示。

在这种表示方法中，数据的所有比特位（不包含符号位）都是取反的结果，然后加上 1 的结果。最后，符号位被置为 1。

有符号的整数对于许多的算法来说是很重要的，但是它的取值范围只能是无符号整数取值范围的一半。例如，假设是 16 位的整型数 32767：

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

对于有符号的整型数来说，如果最高位被设置为 1，则这个数被解释为-1（采用二进制的补码方式）。然而，如果我们把它声明为无符号的整型数，那么当最高位被设为 1 的时候，它的值将为 65535。

为了理解 C++中有符号和无符号的区别，可以看看下面的程序：

view plain

```
#include <iostream>
/* 这个程序演示了无符号和有符号类型数据的差别 */
using namespace std;
int main ()
{
    short int i;                //一个有符号的短整型变量
    unsigned short int j;       //一个无符号的短整型变量
    j = 60000;                  //60000 是在无符号的整型变量的表示范围内的。
    i = j;                      //60000 超出了有符号的短整型的表示范围。
                                //因此，当把他赋值给 i 的时候，它被解释成一个负数了。
    cout << i << " " <<j;
    return 0;
}
```

上面这个程序的输出如下：

-5536 60000

由于作为一个短的无符号的整型数 60000 被解释成短的有符号的整型数后为-5536，因此会出现上面的结果。

C++允许在声明 unsigned ,short 或者 long 整型数的时候采用简写。我们可以仅使用 unsigned,short, or long 这几个字，而不用谢 int。int 关键字是暗含的。例如，下面的两条语句都声明了无符号的整型变量：

unsigned x;

unsigned int y;

字符类型

char 类型的变量可以保存 8 位的 ASCII 码的字符，如 A，z 或者 G，或者任意其它的 8 位的量。在明确指定字符的时候，需要用单引号把字符扩起来。例如给变量 ch 赋值为 X 的

语句如下：

char ch；

ch = 'X';

我们可以使用 cout 语句来输出一个 char 类型变量的值。例如，下面的代码输出变量 ch 的值：

cout << "This is ch : " << ch;

输出的结果为：

This is ch : X

char 类型可以被 signed 或者 unsigned 来修饰。从技术上来讲，不管 char 类型是有符号的还是无符号的，缺省情况下，这都是由编译器的实现而决定的。然而，对于大多数编译器来说，char 类型都是有符号的。在这种情况下，char 前面用 signed 来修饰就显得有些冗余。就本书来讲，我们假定 char 类型是有符号的。

char 类型的变量除了可以用来存储 ASCII 码的字符集外，还可以用来存储别的值。它还可以被用作表示小范围-128 到 127 之间的整数。当程序中不需要比较大的整数的时候，可用 char 类型来代替 int 类型。例如，下面的程序使用 char 类型的变量来控制循环，在屏幕上输出字母表。

view plain

```
1. // This program displays the alphabet
2. // 这个程序用来显示字母表
3. #include <iostream>
4. using namespace std;
5. int main()
6. {
7.     char letter;
8.     //使用一个 char 类型的变量来控制 for 循环
9.     for ( letter = 'A'; letter <='Z'; letter++ )
10.    {
11.        cout << letter ;
12.    }
13.    return 0;
14. }
```

上面程序中的 for 循环之所以能够正确的工作是因为在计算机中字母 A 使用 65 这个值来表示的，从字母 A 到字母 Z，数值是一次递增的。因此 letter 的初始化值为'A'，每循环一次，letter 增加一。因此在第一次循环后，字母就变成了 'B'。

wchar_t 类型存储的字符集是大字符集的一部分。众所周知，许多人类的语言，例如中文都定义了大字符集，超出了 8bit 的 char 的表示范围。C++中增加了 wchar_t 类型就是为

了适应这种情况。然而，在本书中，我们将不使用 wchar_t 类型。如果针对国际化的程序进行裁剪的时候就需要用到 wchar_t 类型了。

练习

1. 七种基本的类型是哪些 ？
2. 有符号和无符号数据之间的区别是什么？
3. 一个 char 类型的变量可以用来表示比较小的整型数吗？

答案

1. 七种基本的类型是：char, wchar_t, int, flot, double, bool 和 void。
2. 一个有符号的整型数可以存储正数和负数。一个无符号的整型数只能存储正数。
3. 是的。

专家答疑

问：

为什么 C++中只是定义了内置类型的最小取值范围，而不是明确规定这些范围了？

答：

C++中没有明确规定内置类型的最小取值范围，可以使得各种编译器针对执行环境进行数据类型的优化。这也是 C++为什么能够创建高性能的软件的部分原因。ANSI/ISO 制定的 C++标准中只是简单描述了内置类型必须满足一定的要求。例如，int 的范围大小可以根据代码的执行环境的要求不同而不同。因此，在 32 为环境下，一个 int 类型就是 32 比特。在 16 位的环境下，一个 int 类型将是 16 比特的宽度。这时没有必要要求 16 位编译器实现 32 为的 int 表示范围，这样做也会降低性能的。当然，C++标准中确实指定了所有环境下都可用的内置类型的最小范围。因此，如果我们编写的程序中的数据都没有超出这些最小的表示范围，那么我们的程序是可以移植到其它环境下的。最后一点：每个 C++编译器都是在 <climits>头文件中明确基本类型的表示范围的。

浮点类型

在程序中需要表示小数的时候或者需要表示非常大或者非常小的数值的时候可以使用 float 和 double 类型。float 和 double 类型的区别在于各自表示的最大和最小数值的尺度各不相同。一个 double 类型可以存储的数值大概比 flot 类型要大 10 倍。两者中，double 是最常用的。其中的一个原因是因为 C++函数库中的大部分数学函数都是用 double 类型。例如 sqrt()函数就返回一个 double 的值：入参的平方根。下面的程序根据输入的直角三角形的两直角边的长度来计算斜边长度，其中就用到了 sqrt()函数：

view plain

```
1. /*
2.      use the pythagorean theorem to find
3.      the length of the hypotenuse given
4.      the lengths of the two opposing sides
5.      使用勾股定理根据直角三角形的两直角边的长度
```

```
    计算斜边的长度
*/
#include <iostream>
#include <cmath>  //sqrt()函数需要改头文件
using namespace std;
int main()
{
    double x,y,z;
    x = 5.0;
    y = 4.0;

    z = sqrt( x * x + y * y ); // sqrt()函数是C++书序库函数中的一个
    cout << "Hypotenuse is " << z;
    return 0;
}
```

程序的输出为：

Hypotenuse is 6.40312

需要说明的另外一点：因为 sqrt()函数是 C++标准函数库中的，它需要标准头文件 <cmath>,程序中包含了该头文件。

long double 类型可以表示更大或者更小的数值。它在科学计算程序中使用的比较多。例如，在分析天文学数据的时候可能就非常有用了。

布尔类型

布尔类型是近期才增加到 C++中的。它用来存储布尔值，也就是 true 或者 false。C++中定义了两个布尔类型的常量，true 和 false，布尔类型只能取这两种值。在继续学习之前，我们需要知道 C++中是如何定义 true 和 false 的，这点很重要。C++中的一个基本的概念就是非零的值被解释为 true，零被解释为 false。这种概念完全和布尔类型是一致的，因为在布尔表达式中，C++自动地把非零值转换为 true，零值转换为 false。反之亦然，在非布尔表达式中，true 被解释为 1，false 被解释为 0。这种零、非零与布尔值之间的转换关系是非常重要的，特别是在控制语句中，这点我们在第三篇中会看到。下面的程序展示了布尔类型的用法：

view plain

```
// Demonstrate bool values.
// 展示布尔值的用法
#include <iostream>
using namespace std;
int main()
{
    bool b;
    b = false;
```

```
9.      cout << "b is " << b << '\n';
10.     b = true;
11.     cout << "b is " << b << '\n';
12.     //布尔值可以用户 if 语句中的控制
13.     if ( b ) cout << "This is executed.\n";
14.
15.     b = false;
16.     if ( b ) cout << "This is not executed.\n";
17.     //输出关系运算符的结果是 true 或者是 false
18.     cout << "10 > 9 is  " << ( 10 > 9 ) << '\n';
19.     return 0;
20. }
```

上面这段程序的输出如下：

b is 0

b is 1

This is executed.

10 > 9 is 1

这个程序中有三点有趣的现象需要注意。第一，正如我们所看到的，当使用 cout 输出布尔值的时候，输出的结果是 0 或者 1。在本书的后面还会看到，有一个输出选择可以使得输出结果为"false"或者"true"。第二，在 if 语句中，可用布尔值来进行控制，没有必要使用诸如下面的代码：

if (b == true) ...

第三，关系运算符的输出结果，例如< 的结果，是布尔值。这也是为什么表达式 10>9 输出的值为 1。更进一步，10>9 需要用括号括起来，这个是非常必要的，因为<<运算符的优先级别高于>运算符。

void 类型

void 类型用来表示“无类型”的表达式。这一点看起来似乎很奇怪。我们将在本书的后面讨论如何使用 void 类型。

练习

1. float 和 double 类型的主要区别是什么？
2. 布尔变量可以取什么值？0 值会被转换成什么样的布尔值？
3. 什么是 void 类型

答案

1. float 和 double 类型的主要区别在于它们存储数值尺度不同。
2. 布尔类型变量的取值或是 true 或者 false。0 值转换成布尔为 false。
3. void 代表的是“无类型”。

项目 2-1 与火星对话

火星距离地球的最近点大约有 34000000 英里。假设我们想与火星上的某个人进行交谈，那么从无线电信号离开地球到信号到底火星的时间延迟是多少了？该项目就是创建一个程序来回答该问题。无线电信号是以光速传播的，大约为 186000 英里/秒钟。因此为了计算时间延迟，我们需要用距离除以光速。程序分别以秒和分钟为单位输出时间延迟。

步骤:

1. 创建一个新的文件叫做 Mars.cpp。
2. 在计算时延的过程中，我们需要使用浮点数。为什么了？因为时间差可能是含有小数的。下面是程序中使用的变量：

```
double distance;
double lightspeed;
double delay;
double delay_in_min;
```

3. 按照如下的代码初始化变量 distance 和 lightspeed:

```
distance = 34000000.0; // 34，000，000 英里
lightspeed = 186000.0; // 186，000 英里每秒钟
```

4. 计算时延时用距离除以光速。这样得到的结果是以秒钟为单位的。把得到的结果赋值给变量 delay 并显示之。这步如下所示：

```
delay = distance / lightspeed;
cout << "Time delay when talking to Mars: " << delay << " second . /n"
```

5. 时延的秒数除以 60 得到以分钟为单位的时延。使用下面的代码显示出结果。

```
delay_in_min = dely / 60.0;
cout << "This is " << delay_in_min << " minutes.";
```

6. 整个程序如下：

```
/*
Project 2-1 Talking to Mars
*/
#include <iostream>
using namespace std;
int main()
{
   double distance;
   double lightspeed;
   double delay;
   double delay_in_min;
```

```
    distance = 34000000.0; // 34,000,000 miles
    lightspeed = 186000.0; // 186,000 per second
    delay = distance / lightspeed;
    cout << "Time delay when talking to Mars: " << delay << " seconds./n";
     delay_in_min = delay / 60.0;
     cout << "This is " << delay_in_min << " minutes.";
     return 0;
}
```

7. 编译并运行这个程序。输出结果如下：

   Time delay when talking to Mars: 182.796 seconds.

   This is 3.04659 minutes.

8. 也可以自己写程序计算与火星双向对话时的时间延迟。

必备技能 2.2 :实字

实字是指固定的，人类可以阅读的，不可改变的值。例如数值 101 就是一个整形实字。实字通常也被称作是常量。实字的使用方法很简单直观，前面的几个实例程序中都用到了实字。现在是时候正式介绍一下实字了。

C++中的实字可以是任何的基本数据类型。实字的表示方式取决于它的类型。正如前面所看到的那样，字符实字是用单引号括起来的一个字符。比如：'a'和'%'都是字符型的实字。

整型实字的就是不带小数的数值。例如，10 和-100 都是整型实字。浮点型实字表示的时候小数点后面要跟小数部分。例如，11.123 就是一个浮点类型的实字。C++中允许使用科学计数法来表示浮点类型的实字。

所有的实字都有类型的，这就引入了一个问题。我们都知道，整型类型有分为长整形，短整型和无符号的整型。浮点类型也分为三种：单精度的浮点，双精度的浮点和长形双精度浮点数。那么编译器如何确定一个实字的类型了？比如，123.23 是单精度的浮点数还是双精度的浮点数了？这个问题的答案有两部分构成。第一，C++编译器自动地采用缺省的类型；第二，我们可以根据需要明确指定实字的类型。

缺省情况下，对于整型实字，C++编译器会采用最小的可以兼容的数据类型来存放这个实字，最小的从 int 类型开始。因此，假设整形数为 16 位，那么 10 缺省的就是 int 类型，但是 103000 就是 long 类型了。尽管 char 类型也可以用来表示 10，但是编译器不会这样做，因为这意味着跨越了类型的边界。

缺省情况下，浮点类型实字的类型是 double 的。因此，123.23 的类型就是 double。

实际上，对于我们初学者来说，编译器的缺省类型对我们来说是相当够用的。只有在编译器缺省取值不满足要求的情况下，我们可以通过明确指定后缀来表示数字型实字的类型。

对于浮点类型来说，如果数字的后面跟了 F，这个数值就会被认为是 float 类型的。如果后面跟的是 L，那么他就是 long double 类型的。对于整型类型来说，后跟 U 表示 unsigned，L 表示 long。在指定一个 unsigned long 类型的时候，后缀 U 和 L 都要使用。下面给出了一些示例：

数据类型	常量举例
int	1 123 2100 -234
long int	3500L -34L
unsigned int	10000U 987U 40000U
unsigned long	12323UL 900000UL
float	123.23F 4.3e-3F
double	23.23 123123.33 -0.9876324
long double	1001.2L

16 进制和 8 进制实字

有时候在程序中使用 16 进制或者 8 进制比使用 10 进制更方便一些。8 进制就是基于 8 的数字系统，它使用数字 0 到 7。在 10 进制中，数字 10 和 8 进制中的 8 是一样的，即数字 10（一零）如果是 10 进制就代表 10，如果是 8 进制就代表 8。16 进制系统中使用数字 0 到 9 加上字母 A 到 F 来表示 10，11，12，13，14 和 15。例如，16 进制数 10 就是 10 进制中的 16。C++中我们可以指定整形实字是 16 进制的或者是 8 进制的。16 进制的数值以 0x 开始。8 进制的实字以 0 开始。下面是一些例子：

hex = 0xFF;//10 进制中的 255

oct = 011;//10 进制中的 9

字符串实字

C++除了支持前面提到的内置类型的实字外，还支持另外一种实字：字符串。一个字符串就是用双引号引起来的字符的集合。比如，“this is a test"就是一个字符串。在前面的示例程序中我们在 count 语句中实际上已经使用到了字符串实字。有一点必须注意：尽管 C++允许我们定义字符串常量，但是 C++中并没有内置的字符串这种类型。而在后面，我们会看到 C++中把字符串作为字符的数组来支持的（C++在类库中提供了 string 类型来支持字符串）。

专家答疑

问：

我们知道了如何指定一个字符实字。那么 wchar_t 类型的实字也是用同样的方式来指定的吗？

答：

不是的。一个宽位字符常量（也就是我们说的 wchar_t 类型）前面需要加上一个字符 L。例如

```
wchar_t wc;
wc = L'A';
```

上面的代码为 wc 赋值了一个宽位的常量值 A。在日常的编程中，我们不会用到宽位字符的。但是如果你需要编写多过语言的程序，可能就要用到宽位字符了。

转义字符

单引号引起来的字符常量大多数都是可打印的，但是也有部分字符，比如回车，当在使用文本编辑器的时候就会出现问题。另外，一些其它的字符，比如单引号和双引号在 C++中都是有特殊含义的，我们不能直接使用它。基于上述的原因，C++提供了转义字符，有时候也被称作为反斜杠字符常量，如表格 2-3 所示，我们可以在程序中使用这些转义字符。可以看到，我们之前使用的/n 就是一个转义字符。

代码	含义
\b	退格
\f	换页
\n	新的一行（换行)
\r	回车
\t	水平制表符
\"	双引号
\'	单引号
\\	反斜杠
\v	垂直制表符
\a	告警
\?	问号
\N	8 进制常量
\xN	16 进制常量

表 2-3　转义字符

专家答疑

问：

由一个字符组成的字符串是否和一个字符常量一样？比如，"k"和'k'一样吗?

答：

不。我们不能把字符和字符串混淆。一个字符实字代表的是一个 char 类型的单一字母。只有一个字符的字符串依然是字符串，不是字符。经字符串是有字符构成的，但是他们是不同的类型。

下面的程序展示了一个转义字符：

```
#include <iostream>
```

```
using namespace std;
int main()
{
   cout << "one\ttwo\tthree\n" ;
   cout << "123\b\b45" ;  //\b\b 将会退格掉 2 和 3
   return  0;
}
```

程序的输入如下：

```
one     two     three
145
```

上面的程序中，第一个 cout 语句使用了制表符来确定 tow 和 three 的位置。第二个 cout 语句中显示输出了一个 123，然后两个退格就删除了 2 和 3，最后输出了字符 4 和 5。

练习：

1. 缺省情况下，实字 10 的类型是什么？10.0 呢？
2. 怎么指定 100 为 long int 类型？怎么指定 100 为 unsigned int 类型？
3. /b 是什么意思？

答案：

1. 10 是 int 类型，10.0 是 double 类型。
2. 指定 100 为 long int：100L；指定 100 为 unsigned int 为：100U
3. \b 退格的转义字符。

必备技能 2.3 再谈变量

在第一章中已经介绍过了变量。本节中我们再仔细研究一下变量。正如前面学到的那样，我们通过如下的格式来声明一个变量：

```
type var-name;
```

即：类型 变量名;

我们可以通过上面的形式来声明任何合法类型的变量。当我们创建变量的时候，我们创建的是他的类型的一个实例。因此，一个变量的能力是有他的类型所决定的。例如，一个用于存储布尔值的布尔变量，就不能用于存储浮点数。更一步来说，变量的类型在变量的生存周期中是不能改变的。比如，一个 int 类型的变量不能变成一个 double 类型的变量。

变量的初始化

我们可以在声明变量的时候就给变量赋值。这个是通过在变量名后面跟上等号和数值来完成的。这叫做变量的初始化。通常是通过下面的形式来完成的：

```
type var = value;
```

这里 value 就是在变量 var 创建的时候需要赋予的值。

下面是一些变量初始化的例子：

```
int count = 10; //初始化 count 为 10
char ch = 'x';   // ch  的初始化值为字母 X
float f = 1.2f;   // f  的初始化值为 1.2
```

当使用逗号来一次声明两个或者多个变量的时候，也可以给其中的一个或者多个或者全部进行初始化。例如：

```
int a,b=8,c=19,d; //b 和 c  都进行了初始化
```

在这种情况下，只有 b 和 c 进行了初始化。

动态初始化

前面的示例中我们都是使用常量对变量进行初始化，C++允许我们动态地对变量进行初始化，那就是在变量声明的时候使用任何有效的表达式来对变量进行初始化。例如,下面的小程序是根据圆柱体底面边境和高度计算体积的。

```
//演示动态初始化
#include <iostream>
using namespace std;
int main()
{
    double radius = 4.0,height = 5.0;
    //动态初始化
    double volume = 3.14156 * radius * radius * height;
    cout << "Volume is " << volume;


    return 0;
}
```

上面的代码中有三个局部变量-radius,height,和 volume。前两个 radius 和 heights 是通过常量来初始化的。然后 volume 是通过圆柱体的体积来动态进行初始化的。这里需要强调的是，初始化表达式中可以使用任何合法的元素，包括调用函数，其他变量或者实字。

运算符

C++中提供了丰富的运算符。一个运算符就是一个标记，用来告诉编译器进行指定的数学或者逻辑运算。C++中有四种基本的运算符：算术运算符，位运算符，关系运算符和逻辑运算符。C++中还有其他的集中运算符可以处理一定的特殊情况。本章将研究算术运算符，关系运算符和逻辑运算符。同时，我们还会研究赋值运算符。位运算符和其他的特殊运算符将在后面的章节中进行研究。

必备技能 2.4 :算术运算符

C++中定义了下面的算术运算符：

运算符	含义
+	加法
-	减法（同时也是一元的负号）
*	乘法
/	除法
%	取模
++	自增
--	自减

其中+，-，*，/运算和几何中的含义是一样的。这些运算符都可以用于与 C++内置类型的数值型数据。它们还可以作用于 char 类型的数据。

取模运算%用来求取除以某个正数的余数。回忆一下，当/运算符应用到整形数的时候，其结果的余数会被截断。例如,整数 10 除以整数 3 的结果是 3。我们可以使用%运算符来获取这中除法的余数。例如,10%3=1。在 C++中，%只能用于整数的操作数。它不能用于浮点数的操作数。

下面的程序演示了取模运算：

```
//演示取模运算
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    x=10;
    y=3;
    cout<<x<<"/"<<y<<" is "<<x/y<<" with a remainder of "<<x%y<<"/n";
    x=1;
    y=2;
    cout<<x<<"/"<<y<<" is "<<x/y<<"/n"<<x << "%" <<y << " is " << x%y <<"/n";
    return 0;
}
```

程序的输出结果是：

```
10 / 3 is 3 with a remainder of 1
1 / 2 is 0
1 % 2 is 1
```

自增和自减运算符

在第一章中我们介绍过++和--分别是自增和自减运算符。他们有一些特殊的性质，这使得这两个运算符很有意思。让我们先回忆一下自增和自减运算符的到底是干什么的.

自增运算符是给他的操作数加 1，自减运算符是个他的操作数减一。因此，

x=x+1;

和

x++;

是一样的。

x=x-1;

和

--x;

是一样的。

自增和自减运算符都既可以是前运算符也可以是后缀运算符。例如：

x=x+1;

可以写成

x++;

或者

++x；

在上面的这个例子中，前缀和后缀运算没有区别。然而，当自增和自减运算符作为一个更大的表达式的一部分的时候，作为前缀和后缀就有了重要的区别。当自增和自减运算符位于操作数的前面，C++会先进行自增和自减运算，然后才是取出变量的值参与表达式的其它部分进行运算。如果自增或者自减运算符位于操作数的后面，C++会先取操作数的值进行运算，然后才进行自增和自减运算。考虑下面的情况：

x=10; y=++x;

在这种情况下，y 的值将会是 11。然而，如果代码如下：

x=10,y=x++;

则 y 的值将会是 10。两种情况下，X 的值都会变成 11。区别就在于 X 是什么时候变成 11 的。这种可以控制变量在什么时候进行自增和自减的方式是很有优势的。

算术运算符的优先级如下：

最高优先级　　++ --

-(一元的负号)

* / %

最低优先级　　+ -

只有在同一优先级的运算符，编译器才会从左到右进行运算。当然，我们可以使用括号来改

变运算的先后顺序。C++中的括号实际上和其他计算机语言中的一样：或者是用来进行强制运算的，或者是用来执行一系列运算的，或者是用来提高运算级别的。

专家答疑

问：

自增运算符++和 C++这个名字有关系吗？

答：

是的。我们都知道，C++是构建于 C 之上的。C++在 C 上做了很多增强，其中大部分都是支持面向对象的编程的。因此，C++代表着对 C 的增加和改进。对 C 这个名字后面增加++也是一个很好的解释 C++的方式。

斯特劳施特鲁普期初把 C++叫做“带有类的 C”，但是在 Rick Mascitti 的建议下，他把名字改成了 C++。这个新的语言注定是要成功的，C++名字的修改实际上确保了他在历史中的地位，因为这个名字是一个所有 C 程序员都能立刻认识的名字。

必备技能 2.5 关系和逻辑运算符

术语关系运算符和逻辑运算符中，关系是指值和其它的值可能存在的关系，逻辑是指真值和假值关联起来的方式。关系运算符产生的是真值或者假值，因此它们通常和逻辑运算符一起连用。它们也经常被放在一起来讨论的。

关系和逻辑运算符如表格 2-4 所示。注意：在 C++中不相等是用!=来表示的，相等是用==来表示的，两个等号。在 C++中，关系或者逻辑表达式的值是布尔类型的。因此，关系或者逻辑表达式的结果要么是 true 要么是 false。

注意：在比较老的编译器中，关系或者逻辑表达式的值为整数 0 或者 1。这种差别只是在学术中差别，因为 C++自动地把 true 转换成 1，把 false 转换成 0，反之亦然。

关系运算符的操作数几乎可以使任何类型，只要他们能够进行有意义的比较。逻辑运算符的操作数必须能产生真值 true 或者假值 false。既然任何的非 0 值都是 true，0 值是 false，这就意味着逻辑运算符可以被用于任何可以产生 0 值或者非 0 值的表达式。因此，任何表达式都可以用作逻辑运算符的操作数，结果为 void 类型的除外。

关系运算符

关系运算符	
运算符	含义
>	大于
>=	大于或者等于
<	小于
<=	小于或者等于
==	等于

!=	不等于
逻辑运算符	
运算符	含义
&&	AND 与
||	OR 或
!	NOT

表格 2-4 C++中的关系和逻辑运算符

逻辑运算符是用来支持基本的逻辑运算 AND（与），OR（或），NOT（非），其真值表如下：

p	q	p AND q	p OR q	NOT p
false	false	false	false	true
false	true	false	true	true
true	true	true	true	false
true	false	false	true	false

下面的程序演示了几种关系运算符和逻辑运算符。

```
#include <iostream>
using namespace std;
int main()
{
    int i, j;
    bool b1, b2;
    i = 10;
    j = 11;
    if(i < j) cout << "i < j/n";

    if(i <= j) cout << "i <= j/n";
    if(i != j) cout << "i != j/n";
    if(i == j) cout << "this won't execute/n";
    if(i >= j) cout << "this won't execute/n";
    if(i > j) cout << "this won't execute/n";

    b1 = true;
    b2 = false;
    if(b1 && b2) cout << "this won't execute/n";
    if(!(b1 && b2)) cout << "!(b1 && b2) is true/n";
```

```
    if(b1 || b2) cout << "b1 || b2 is true/n";
    return 0;
}
```

程序的输出如下：

```
i < j
i <= j
i != j
!(b1 && b2) is true
b1 || b2 is true
```

关系运算符和逻辑运算符的优先级都低于算术运算符。这就意味着表达式 10>1+12 和 10>(1+12)是等效的，其结果自然是 false 了。

我们可以使用逻辑运算符吧任意多个关系运算连接起来。例如，下面的例子中是吧三个关系运算用逻辑运算符连接起来了。

```
var > 15 || !(10 < count ) && 3 <= item
```

下面的表格列出了关系运算符和逻辑运算符的相对优先级：

最高优先级	!
	> >= < <=
	== !=
	&&
最低优先级	||

项目 2-2 构建异或关系的逻辑运算

C++中没有定义异或关系的逻辑运算符，也就是 XOR。异或运算是一种二进制的运算，只有当两个运算数中只有一个为 true 的时候其结果才是 true。其真值表如下：

p	q	p XOR q
false	false	false
false	true	true
true	false	true
true	true	false

一些程序员认为缺失了异或运算是 C++的一个缺憾。也有人认为没有引入异或运算是 C++的简化设计，避免了冗余的特性。他们认为使用 C++中提供的其他三个逻辑运算符就很容易构造出异或运算。

在本项目中，我们将使用&&，||，和!运算符来构建异或运算。

步骤：

1. 创建一个新的文件叫做 XOR.cpp
2. 假定有两个 bool 类型变量 p 和 q，那么异或运算可以通过下面的方式来构造：

(p||q)&&!(p&&q)

我们来仔细研究一下这个表达式。首先，p 和 q 做或运算，如果其值为 true，说明其中至少有一个为真。接着，p 和 q 做与的运算，其值只有两者都为 true 的时候为 true。然后对其结果做非运算，!(p&&q)的值只有 p 或者 q 或者两者都是假的时候。最后，这个结果和(p||q)相与。因此，整个表达式只有在其中一个操作数为 true 的情况下才为 true。

3. 下面是完整的程序。他演示了四种异或运算的结果。

```
/* Project 2-2 Create an XOR using the C++ logical operators. */
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   bool p, q;
   p = true;
   q = true;
   cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "/n";
   p = false;
   q = true;
   cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "/n";

    p = true;
    q = false;
    cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "/n";

    p = false;
    q = false;
    cout << p << " XOR " << q << " is " << ( (p || q) && !(p && q) ) << "/n";

return 0;
}
```

4. 编译并运行这个程序。输出结果如下：

```
1 XOR 1 is 0
0 XOR 1 is 1
1 XOR 0 is 1
0 XOR 0 is 0
```

5. 注意异或运算表达式最外层的括号。由于运算符优先级的关系，它们是必须的。<<运算符的优先级是高于逻辑运算符的。为了证实这一点，可以尝试去掉最外层的括号，然后试图编译这个程序。结果就是有编译错误。

练习：

1. %运算符是用来做什么的？他可以被应用那种类型的数据上？
2. 如何声明一个 int 类型的名字叫做 index 的变量，他的初始值为 10？
3. 关系或者逻辑运算符的结果是什么类型？

答案：

1. % 是取模运算符。他返回整除的余数，他可以被应用于整形数上。
2. int index = 10;
3. 关系和逻辑运算表达式的结果是 bool 类型的。

必备技能 2.6：赋值运算符

我们从一开始就是用了赋值运算符。现在应该正式地认识一下它了。赋值运算符就是一个单的=（等号）。C++中的赋值运算符和其它计算机语言中的赋值运算符是一样的。它的通用形式如下：

变量 = 表达式;

这里就是把表达式的值赋值给变量。赋值运算符有一个很有意思的特性：那就是可以创建赋值的链形式。例如下面的代码片段:

```
int x, y, z;
x = y = z = 100; //设置 x，y，z 的值都是 100
```

上面的代码片段使用了一条语句来给给 x，y，z 三个变量都赋值 100。这样是可行的，因为赋值运算的值就是右边表达式的值。因此表达式 z=100 的值为 100，然后把它赋值给 y，在把 y 的值赋值给 x。采用这种连续的赋值方式可以很容易地给一组变量赋上相同的值。

必备技能 2.7:复合赋值

C++中提供了一个特殊的复合赋值运算符，它能简化特定的赋值表达式。我们看个例子吧。赋值语句如下：

```
x = x + 10;
```

用复合赋值可以写成：

```
x += 10;
```

其中的运算符对 += 告诉编译器给 X 赋值为 X 的值加上 10。下面是另外的一个例子。语句:

```
x = x -100;
```

和

x -= 100；

是等价的。两条语句都是给 X 赋值为 X 的值减去 100。对于大部分的二目运算符来说都有对应的复合赋值运算符。也就是说下面的形式：

var = var op experesion;

可以被写成：

var op= expression;

由于复合赋值运算符在书写上都比对应的非复合赋值运算符简单，因此复合赋值运算符有时候也被称为赋值运算符的简写。

复合赋值运算符带了两个好处：其一，他们更紧凑一些。其二，他们可以产生更有效的可执行代码。基于以上的原因，我们在专业的 C++程序中会经常看到复合赋值运算符。

必备技能 2.8：赋值时的类型转换

当一种类型的变量和另外一种类型的变量混合使用的时候，就会发生类型的转换。在一个赋值语句中，类型转换的规则很简单：赋值语句右边的值被转换成左边的值。正如下面的代码展示的那样：

```
int x;
char ch;
float f;
ch = x;   /* line 1 */
x = f     /* line 2 */
f = ch;   /* line 3 */
f = x;    /* line 4 */
```

在 line 1 中，整形变量 x 的高位被截断，剩下的低 8 位赋值给 ch。如果 x 的值在-128 到 127 之间，ch 和 x 将有着相同的取值。否则，ch 的值只能体现 x 的低 8 位的值。在 line 2 中，x 将被赋值为 f 的整数部分。在 line 3 中，f 将把 ch 中存储的 8 比特整形数转换成相等的浮点数形式。line 4 中的情况也是如此，除了是把一个整形数转换为一个浮点数。

当转换时从整形数到字符类型或者是从长整形到整形的时候，对应数字的高位将被移除。在很多的 32 位环境中，这意味着如果是从整形数到字符的转换，将要丢失 24 个比特；如果是从整形数到短整形数的转换，将丢失 16 比特。当由浮点数转换为整形数的时候，小数部分将会被丢弃。如果目标类型不足够大来保存结果，其结果将会和原来的值相差甚远。

值得注意的地方：C++会自动地对内置的类型进行相互转换，但其结果并不一定是我们想要的。所以在表达式中使用不同类型的数据的时候要特别注意。

表达式

运算符，变量，和实字都是表达式的组成部分。或许我们已经从其它语言的编程经验中

或者是从代数中知道了表达式的基本形式。然而，我们在这里还是有必要讨论一下表达式的几个特性。

必备技能 2.9：表达式中的类型转换

当我们在表达式中使用了不同类型的常量和变量的时候，它们会被转换成同一种类型。首先，所有的 char 类型和 short int 会被自动转换为 int 类型。这个过程叫做整型提升。接着，所有的运算数都会被转换成“最大”的类型，这个叫做类型的提升。这种提升是基于“按照工序组合"的方式进行的。例如：如果一个运算数是 int 类型的，另外一个是 long int 类型的，那么这个 int 类型的数据就会被提升为 long int 类型。或者，如果一个运算数是 double 类型的，另外一个就会被提升为 double 类型。这意味着类似于从 char 到 double 类型的转换时完全有效的。一旦进行了提升，每一对操作数就会有相同的类型，其结果的类型也就是两个操作数的类型一样了。

bool 类型的转换

正如前面所提到的那样，如果在表达式中使用了 bool 类型，其值会被自动的转换成 0 或者 1。当一个整数被转换成 bool 类型的时候，0 被转换成 false，非 0 就会被转换为 true。尽管 bool 类型是近期才添加到 C++语言中的，这种自动的转换对老的代码来说实际上是没有任何影响的。更进一步来说，这种自动的转换是和之前对 true 和 false 定义为非 0 和 0 是一致的。

必备技能 2.10：强制类型转换

我们还可以把一个表达式的值强行地转换为某种特定的类型，这叫做强制类型转换。强制的类型转换是一种显示的类型转换。C++定义了五种强制的类型转换。其中四种在转换的时候是允许有多细节的控制的，我们将在后面学习到对象以后再介绍。剩下一种是我们现在就可以学习的。它就是 C++中最常用的强制类型转换。它可以把任意的类型转换成别的类型。它也是早期版本的 C++唯一支持的转换。他的通用形式如下：

(type）expression

这里，type 代表的是目标类型，是我们要把 expression 转换成 type 的类型。例如：如果我们希望表达式 x/2 是按照浮点数的方式来运算的，就可以这样写：

```
(float) x / 2;
```

强制类型转换被认为是一种运算符。作为一种运算符，它是单目的，和其它单目的运算符有着相同的优先级。

有时候，强制类型转换是非常有用的。例如，我们可能用到整数来做循环控制，但是还需要对其按照 float 类型类进行运算，正如下面的程序所展示的一样：

```
// Demonstrate a cast
```

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for (i=1; i <= 10; ++i )
        cout << i << " / 2 is: " << (float) i / 2 << '\n';
    return 0;
}
```

程序的输出结果如下：

```
1 / 2 is: 0.5
2 / 2 is: 1
3 / 2 is: 1.5
4 / 2 is: 2
5 / 2 is: 2.5
6 / 2 is: 3
7 / 2 is: 3.5
8 / 2 is: 4
9 / 2 is: 4.5
10 / 2 is: 5
```

如果上面的程序中没有强制的类型转换(float)，就只能进行整数的除法了。在这里，强制类型转换确保了我们想要的答案的小数部分。

必备技能 2.11: 关于空格和括号

C++中的表达式中可以含有 tab 键和空格键，来使得程序更加具有可读性。例如，下面的两个表达式是相等的，但是第二个更具有易读性。

```
x=10/y*(127/x);
x = 10 / y * (127 / x);
```

括号增加了被括起来的操作的优先级，就像代数里面的一样。在代码中使用了多余的或者附加的括号并不会导致错误或者降低表达式的执行效率。为了自己也为了后来可能阅读你代码的人，我们鼓励使用括号来使得计算的顺序更加清晰。例如，下面的那个表达式更容易读懂了？

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

项目 2-3 计算贷款的定期还款额

在这个项目中，我们将要创建一个程序用来计算贷款的定期还款额度，比如买车的贷款。指定本金，贷款的时间长度，每年偿还的次数，以及贷款利率，程序就会计算出每次应该偿还的额度。因为这里涉及到小数的运算，我们需要使用浮点类型的数据来进行计算。既然 double 类型是最常用的浮点类型，在这个项目中我们就是用 double 类型的浮点数。这个程序将使用到另外的一个 C++库函数：pow（）。计算定期还款金额的公式如下：

$$\text{Payment} = \frac{IntRate * (Principal/PayPerYear)}{1 - ((\frac{IntRate}{PayPerYear}) + 1)^{PayPerYear*NumYEars}}$$

这里 IntRate 代表利率，Principal 代表本金，PayPerYear 代表每年偿还贷款的次数，NumYears 代表贷款的年限。

注意在上面的公式中使用到了幂运算，在程序中我们将使用 pow()函数来完成这个功能。下面的代码显示了我们应该怎么使用这个函数：

```
result = pow ( base, exp);
```

pow()函数返回 base 的 exp 次幂。传入到 pow()中的参数是 double 类型的，返回值也是 double 类型的。

步骤：

1. 创建一个新的文件命名为 RegPay.cpp。
2. 下面是程序中将使用到的变量：

```
double Principal; //原始的本金
double IntRate;    //利率，例如，0.075
double PayPerYear; //每年偿还的次数
double NumYears; //偿还的年限
double Payment;     //每次偿还的数额
double number, denom; //  临时的变量
double b,e; //底数，指数
```

注意上面在变量声明后的注释，用于描述变量的作用，这样有助于别人阅读我们的程序，也能很容易地明白各个变量的作用。虽然本书的大部分小程序都没有包含这样的细节，但是这种做法确实很好，特别是当程序变得越来越大，越来越复杂的时候。

3. 添加下面的代码，其中包括输入贷款的信息：

```
count << "Enter principal:";
cin >> Principal;
count << "Enter interest rate(i.e.,0.075): ";
cin >> IntRate;
```

```
count << "Enter number of years: ";
cin >> NumYears;
```

4. 添加下面的代码来进行运算：

```
number = IntRate * Principal / PayPerYEar;
e = -(PayPerYear * NumYears );
b = (IntRage / PayPerYears )+1
denom = 1 - pow(b,e);
Payment = number / denom;
```

5. 最后，输出还款额：

```
cout << "Payment is " << Payment;
```

6. 完整的 RegPay.cpp 程序如下：

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
   double Principal; // original principal
   double IntRate; // interest rate, such as 0.075
   double PayPerYear; // number of payments per year
   double NumYears; // number of years
   double Payment; // the regular payment
   double numer, denom; // temporary work variables
   double b, e; // base and exponent for call to pow()

   cout << "Enter principal: ";
   cin >> Principal;

   cout << "Enter interest rate (i.e., 0.075): ";
   cin >> IntRate;

   cout << "Enter number of payments per year: ";
   cin >> PayPerYear;
   cout << "Enter number of years: ";
   cin >> NumYears; numer = IntRate * Principal / PayPerYear;
```

```
    e = -(PayPerYear * NumYears);
    b = (IntRate / PayPerYear) + 1;
    denom = 1 - pow(b, e);
    Payment = numer / denom;
    cout << "Payment is " << Payment;
    return 0;
}
```

程序可能的输出结果如下：

```
Enter principal: 10000
Enter interest rate (i.e., 0.075): 0.075
Enter number of payments per year: 12
Enter number of years: 5
Payment is 200.379
```

我们可以自己修改程序，输出共计需要支付的利息。

复习题

1. C++中都支持那些类型的整数？
2. 缺省情况下，12.2 的类型是什么？
3. 布尔类型的变量可以取那些值？
4. 什么是长整形数据类型(long integer)?
5. 什么样的转义字符代表一个 tab 键？什么样的转义字符代表产生哔哔的响声？
6. 一个字符串是用双引号括起来的，对吗？
7. 什么是十六进制数？
8. 写出在声明变量时进行初始化的通用格式。
9. %是做什么用的？ 他可以应用于浮点数吗？
10. 解释前缀自增运算符和后缀自增运算符的区别？
11. 下面那些是 C++中的逻辑运算符？
    a. &&
    b. ##
    c. ||
    d. $$
    e. !
12. 如何重写：

```
x = x + 12;
```

13. 什么是强制类型转换

14. 写一个程序，列出 1 到 100 之间的素数。

第三篇 程序控制语句

本章中我们将讨论控制程序执行顺序的语句。有三重程序控制语句：选择语句，包括 if 和 switch 语句；循环语句，包括 for,，while 和 do-while；跳转语句，包括 break，continue，return 和 goto。除了 return 语句我们会在本书的后面章节介绍外，其它的流程控制语句我们都会在本章进行讨论，包括前面我们意见接触过的 if 和 for 语句。

必备技能 3.1：if 语句

在第一章中我们就介绍过 if 语句，现在我们来进一步研究一下 if 语句。if 语句的完整形式如下：

```
if (表达式) 语句；
else 语句；
```

上面的形式中，if 和 else 的目标语句都是单条的。其中的 else 语句是可选的。if 和 else 的目标语句也可以是多条语句构成的代码块。使用代码块作为目标语句的形式如下：

```
if(表达式)
{
    语句序列
}
else
{
    语句序列
}
```

如果 if 语句中的条件表达式的值为 true（真），则会执行目标代码（块）；否则，如果存在 else 分支，则会执行 else 的目标代码（块）。if 和 else 的目标代码（块）是不可能都执行的。其中用来控制 if 语句的条件表达式可以是任何类型的有效的 C++表达式，其结果为 true 或者 false。

下面的程序通过一个简单的“猜数字”的游戏演示了 if 语句的用法。程序生成一个随机数，然后提示用户输入自猜想的数字。如果用户猜对了，则打印信息"**Right**。程序中还用到了另外的一个 C++库函数，rand()。该函数用来产生一个随机的整型数。使用这个函数需要引入<cstdlib>头文件。

```
//猜数字
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main()
{
    int magic;
    int guess;
    magic = rand(); //  随机地生成一个数字
    cout << "Enter your guess: ";
    cin >> guess;
    //如果猜中了，则打印信息
    if ( guess == magic ) cout << "**Right**";
    return 0;
}
```

上面的程序中，使用了 if 语句来判断用户是否猜中了数字。如果猜中了，则在屏幕上输出“**Right**”信息。我们改写一下上面的程序，如下。程序中使用了 else 分支来处理用户没有猜中的情况，输出“...Sorry, you're wrong.”

```
//猜数字
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int magic;
    int guess;
    magic = rand(); //  随机地生成一个数字
    cout << "Enter your guess: ";
    cin >> guess;
    //如果猜中了，则打印信息
    if ( guess == magic ) cout << "**Right**";
    else cout << "...Sorry, you're wrong.";
    return 0;
}
```

条件表达式

有时候，C++新手可能会对“任何有效的 C++表达式都可以用于 if 语句的条件表达式”这一点不是很理解。也就是说 if 语句中的条件表达式不局限于涉及关系和逻辑运算符的那些表达

式或者是产生布尔值的那些表达式。if 语句的条件表达式只要是能产生 true（真）或者 false（假）值的表达式就可以。回忆一下，我们在前面的章节中学过，0 值会被自动地转换为 false，所有的非 0 值会被自动的转换为 true。因此，任何能够产生 0 值或者非 0 值的表达式都是可以用作 if 语句的条件表达式的。例如，下面的程序从键盘读入两个整型数，然后输出它们的商。为了避免 0 做除数，程序中使用了 if 语句，条件表达式就是 b。

```
//使用一个整型数来作为 if 语句的控制条件
#include <iostream>
using namespace std;
int main()
{
    int a,b;
    cout << "Enter numerator: ";
    cin >> a;
    cout << "Enter denominator: ";
    cin >> b;
    if(b) cout << "Result: " << a/b << "\n";
    else cout << "Cannot divide by zero.\n";
    return 0;
}
```

程序的输出如下：

```
Enter numerator: 12
Enter denominator: 2
Result: 6
Enter numerator: 12
Enter denominator: 0
Cannot divide by zero.
```

注意在上面的程序中使用使用了 if(b)来检查 b 是否为 0。这样写实正确的，因为当 b 是 0 值的时候，if 的控制条件就是 false，else 分支得到执行。否则，控制条件就是 true（非 0 值），于是便进行了除法运算。这里，没有必要写出如下的程序：

```
if ( b== 0) cout << "Cannot divide by zero./n";
```

上面的写法通常会被认为是不好的编程风格。这种写法显得冗余，并且效率较低。

嵌套的 if

嵌套的 if 是指一个 if 语句作为另外的 if 语句或者 else 语句的目标语句。实际编程中，嵌套的 if 是很普遍的。需要注意的是：在嵌套的 if 语句中，else 分支总是对应的是和该 else 分支在同一代码块中的没有 else 分支的那个最近的 if 语句。下面就是一个例子：

```
if (i)
{
    if (j) result = 1;
    if (k) result = 2;
    else result = 3; //  和 if(k)对应
}
else result = 4; //  和 if(i)对应
```

正如上面的注释那样，最后的那个 else 对应的不是 if(j)，尽管它是距离最近的没有 else 分支的 if，这是因为它们不在同一个代码块中。所以最后一个 else 是和 if(i)对应的。花括弧中的最后一个 else 是和 if(k)相对应的，因为 if(k)是距离它最近的，并且在同一个代码块中。

我们可以使用嵌套的 if 来修改一下前面的猜数字程序。下面的程序中增加了针对用户猜错时候的提示信息。

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int magic;
    int guess;
    magic = rand();
    cout << "Enter your guess: ";
    cin >> guess;
    if ( guess == magic )
    {
        cout << "**Right**";
        cout << magic << " is the magic number.\n";
    }
    else
    {
        cout << "...Sorry, you're wrong.";
        //下面使用了嵌套的 if
        if ( guess > magic ) cout << "Your guess is too hight.\n";
        else cout << " Your guess is too low.\n";
    }
```

```
    return 0;
}
```

阶梯状的 if-else-if

常见的使用嵌套的 if 语句的形式就是阶梯状的 if-else-if 结构。如下：

```
if (条件表达式)
    语句;
else if (  条件表达式)
    语句;
else if (  条件表达式)
语句;
.
.
.
else
语句;
```

在这种形式中，条件表达式会从上到下被依次进行检查。只要遇到一个条件为真值，相对应的语句就会被执行，其它的代码将被跳过。如果所有的条件表达式都不是真值，则执行最后的一个 else 对应的目标语句。最后的一个 else 语句通常都是用来处理缺省情况的。也就是说，只有所有的 if 检测都失败了，才会执行最后的 else 分支的语句。如果没有最后的 else 分支，并且所有的 if 检测都是失败的，则什么都不做。

下面的程序演示了阶梯状 if-else-if 的用法：

```
//演示阶梯状 if-else-if 的用法
#include <iostream>
using namespace std;
int main()
{
    int x;

    for ( x = 0; x < 6; x++)
    {
        if ( x == 1) cout << "x is one\n";
        else if ( x== 2 ) cout << "x is tow\n";
        else if ( x== 3 ) cout << "x is three\n";
        else if ( x== 4 ) cout << "x is four\n";
```

```
        else cout << "x is not between 1 and 4\n";
    }
    return 0;
}
```

上面程序的输入如下：

```
x is not between 1 and 4
x is one
x is tow
x is three
x is four
x is not between 1 and 4
```

从中我们可以看出，缺省的 else 分支只有在前面所有的 if 分支检测都失败的情况下才会被执行。

练习：

1. if 语句中的条件控制语句必须使用关系运算符吗。正确还是错误？
2. else 是和什么样的 if 相对应的？
3. 什么是阶梯状的 if-else-if 结构？

答案：

1. 错误
2. else 总是对应于和它位于同一代码块中并且没有 else 分支的最近的那个 if。
3. 阶梯状的 if-else-if 如下：

```
if (条件表达式)
   语句;
else if ( 条件表达式)
   语句;
else if ( 条件表达式)
    语句;
.
.
.
else
    语句;
```

必备技能 3.2：switch 语句

C++中的第二个用来选择的语句就是 switch 语句。switch 语句提供多个分支的选择，因此它可以使程序在多个选项中进行选择。尽管一系列的嵌套 if 语句也可以实现多分支的检测，但是在多数情况下，switch 语句会更有效。当进行匹配的时候，表达式的取值会与一些列的常量值进行连续地检测。当匹配时，和该匹配相关联的语句序列就会被执行。switch 语句的通用形式如下：

```
switch (表达式)
{
   case  常量 1:
      语句序列
      break;
   case  常量 2:
      语句序列
      break;
   case  常量 3:
      语句序列
      break;
   .
   .
   .
    default :
      语句序列
}
```

其中的表达式的值必须是字符或者是整数值（浮点数是不允许的）。通常情况下，控制 switch 语句的表达式只是一个简单的变量。case 分支中的常数必须是整数或者是字符常量。

其中的 default 语句只有在没有找到匹配的时候才会执行。default 语句是可选的。如果没有 default 语句，并且也没有找到匹配的 case，那么什么都不会做。如果找到了匹配的 case 分支，那么和该 case 分支相关联的语句就会被执行，直到遇到一个 break 语句，或者直到 case 语句的结束或者是到 default 语句，或者是到 switch 语句的结尾。

关于 switch 语句有四点需要明确注意：

1. switch 语句和 if 语句的不同之处是 switch 语句只能进行相等的条件检测，而 if 语句则可以进行其它更多的条件检测。

2. 在 switch 中，不能有两个 case 分支的常量取值是一样的。当然，嵌套的 switch 语句中，内外两个 switch 中的 case 分支的常量可以有相同的取值。

3. switch 语句通常比 if 嵌套更高效。

4. 和每个 case 相关联的语句序列并不是一个代码块，然而整个 switch 语句却是一个

代码块。这一点的重要性会随着我们对 C++的学习而越来越明显。

下面的程序演示了 switch 的用法。它要求用户输入一个 1 到 3 之间的数字，然后显示一个和该数字相关的言语。如果输入的是其它的数字，则显示错误信息。

view plain

```
/*
    A simple proverb generator that
    demonstrates the switch.
*/
#include <iostream>
using namespace std;

int main()
{
    int num;
    cout << "Enter a number form 1 to 3:";
    cin >> num;

    switch(num)
    {
        case 1:
            cout << " A rolling stone gathers no moss.\n";
            break;
        case 2:
            cout << " A bird in hand in worth two in the bush. \n";
            break;
        case 3:
            cout << " A fool and his money are soon parted.\n";
            break;
        default:
            cout << " You must enter either 1,2,or 3.\n";
    }

    return 0;
}
```

下面是两个示例运行结果：

Enter a number form 1 to 3:1

A rolling stone gathers no moss.

Enter a number form 1 to 3:5

You must enter either 1,2,or 3.

从技术层面来讲，break 语句也是可选的，尽管大部分的应用程序中的 switch 语句中都会使用 break 语句。当在一个 switch 中的 case 相关联的语句中出现 break 的时候，程序的流程就会跳出整个 switch 语句，进而执行 switch 外的下一条语句。然而，一个 case 相关联的语句中没有以 break 来结束，那么所有在匹配上的 case 分支中的语句以及下面的语句都会被执行，直到遇到一个 break 语句或者是到了 switch 语句的结束。举例来说，仔细研究下面的程序，猜猜看会在屏幕上显示什么？

view plain

```
/* A switch without break statements.
*/
#include <iostream>
using namespace std;

int main()
{
    int i;
    for ( i = 0; i < 5; i++)
    {
        switch ( i )
        {
          case 0: cout << " less than 1\n";
          case 1: cout << " less than 2\n";
          case 2: cout << " less than 3\n";
          case 3: cout << " less than 4\n";
          case 4: cout << " less than 5\n";
        }

        cout << '\n';
    }
     return 0;
}
```

这个程序的输出如下：

less than 1

less than 2

less than 3

less than 4

less than 5

less than 2

less than 3

less than 4

less than 5

less than 3

less than 4

less than 5

less than 4

less than 5

less than 5

正如这个程序演示的那样，case 中如果没有 break 语句，程序会执行到下一个 case 中去。当时我们是可以有空的 case 的，正如下面的程序段一样：

```
switch(i)
{
   case 1:
   case 2:
   case 3:
      cout << " i is less than 4":
      break;
   case 4:
      count << " i is 4 ";
      break;
}
```

在上面的这个程序段中，如果 i 的值为 1，2 或者 3，就会输出下面的信息： i is less than 4；如果 i 的值是 4 就会输出 i is 4。像这个程序中的那样，“堆叠”的 case 分支在多个 case 分支公用相同的代码的时候非常常见。

嵌套的 switch 语句

在程序中很有可能用到一个 switch 语句嵌套在另外一个 switch 语句中。即使是内外层的 case 分支中的常量是相同的，这也不会引起冲突。例如，下面的代码段是可以接受的：

view plain

```
switch(ch1)
    {
        case 'A': cout << " This A is part of outer switch";
            switch(ch2)
            {
                case 'A':
                    cout << "This A is part of inner switch";
                    break;
```

```
9.                 case 'B':
10.                     //.......
11.             }
12.             break;
13.         case 'B':
14.             //.....
15.     }
```

练习：

1. switch 的控制表达式必须是什么类型？
2. 当 switch 表达式匹配上了一个 case 的常量，会发生什么？
3. 如果一个 case 相关的语句序列中没有用 break 来终止，会怎样？

答案：

1. switch 的控制表达式必须是整数类型或者字符类型。
2. 当 switch 表达式匹配上了一个 case 的常量，那么和该 case 相关联的语句就会被执行。

3. 一个 case 相关联的语句中没有以 break 来结束，那么所有在匹配上的 case 分支中的语句以及下面的语句都会被执行，直到遇到一个 break 语句或者是到了 switch 语句的结束。

专家解答：

问：当遇到多个分支选择的时候，我们应该如何确定是应该采用 if-else-if 的阶梯式结构了还是应该采用 switch 语句了？

答：通常情况下，在控制选择过程的条件不是简单的的值的情况下，我们使用 if-else-if 的阶梯式结构。比如，考虑下面的 if-else-if 阶梯结构：

```
if ( x < 10 ) //...
else if ( y > 0 ) //...
else if ( !done )///....
```

这样的结构不能用 switch 语句来代替，因为三个不同的条件中包含了不同的变量，而且变量的类型还不一样。什么样的变量可以用来控制 switch 语句了？同样，在检测浮点数的值的时候或者别的不能在 switch 中用的类型的时候，我们需要使用 if-else-if 的阶梯结构。

必备技能 3.3：for 循环

在第一章节中我们就用到了 for 循环的简单形式。但是 for 循环的强大和灵活绝对会让我们感到吃惊。让我们还是先来回顾一下 for 循环最基本的最传统的形式吧。

重复执行一条语句的简单的 for 循环如下：

for (初始化; 表达式; 自增) 单条语句;

重复执行一段代码的 for 循环的通用形式如下：

```
for ( 初始化; 表达式; 自增)
{
    语句序列
}
```

上面的初始化通常是一个给循环控制变量赋值的语句，这个变量像一个计数器一样可以控制循环的次数。其中的表达式是一个条件表达式，它决定了是否需要重复执行循环体中的代码。其中的自增语句则表达了每次循环后控制变量的变化是多少。请注意上面这三个主要的部分必须是用分号间隔的。只要表达式部分检测的结果为 true，循环就一直进行。一旦表达式的检测结果为 false 了，就会退出循环，程序会继续执行 for 循环后面的语句。

下面的程序使用了一个 for 循环来打印 1 到 99 之间的数字的方根。请注意，在这个程序中，循环的控制变量是 num。

```
// Show square roots of 1 to 99
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int num;
    double sq_root;

    for ( num = 0; num < 100; num++)
    {
       sq_root = sqrt ((double) num);
       cout << num << " " << sq_root <<'\n';
    }

    return 0;
}
```

在这个程序中使用到了标准的库函数 sqrt()。正如在第二篇中介绍的那样，函数 sqrt()返回他的参数的方根。它的参数必须是 double 类型的，函数的返回值也是 double 类型的。这里头文件<cmath>是必须的。

for 循环可以以增加的方式进行，也可以以减小的方式进行，其实对控制变量的增加量也是可以取任意值的。例如，下面的程序打印数字 50 到-50，控制变量时减少 10 。

```
// A negatively runing for loop
#include <iostream>
using namespace std;
int main()
{
    int i;
    for ( i = 50; i >= -50;   i = i -10 ) cout << i << ' ';


    return 0;
}
```

输出的结果如下：

50 40 30 20 10 0 -10 -20 -30 -40 -50

非常重要的一点是其中的条件表达式的检测总是在循环之前进行的。这就是说 for 循环的循环体有可能不会被执行，如果条件表达的检测一开始就是 false。下面就是一个例子：

```
 for ( count = 10; count < 5; count++ )
       cout << count; //这条语句是不会被执行的。
```

专家解答：

问：C++中是否支持除了 sqrt（）之外的其它的数学函数？

答：是的！除了 sqrt（）之外，C++支持广阔的数学库函数。例如：sin()，cos()，tan()，log(), ceil(),还有 floor（），这些只是其中的一部分。如果你主要进行和数学相关的编程，那么你可以研究一下 C++的数学库。所有的 C++编译器都支持这些库函数。你可以在编译器的文档中找到这些函数的描述。它们都需要头文件<cmath>。

for 循环的一些变化

for 循环是 C++语言中最通用的语句之一。因为它提供了非常灵活的变化。例如，我们可以使用多个循环控制变量。考虑下面的代码片段：

```
for ( x= 0, y = 10; x<=y ;++x, --y ) //这里使用了多个循环控制变量
    cout << x << ' ' << y << '\n';
```

这里在两个变量的初始化和自增表达式中间都用了逗号间隔。这样做是为了告诉编译器有两个初始化语句和两个自增语句。在 C++中，逗号是一种运算符，它的意思是“做这个和这个”。它是在 for 循环中经常使用的。我们可以有任意多的初始化和自增语句，但是实际上使用多于两个或者三个以上的初始化和自增语句是非常令人难懂的。

条件控制语句可以是任意的有效的 C++表达式。它也可是不必含有循环控制变量。在下一个例子中，循环回一直执行，直到 rand（）函数产生一个大于 20，000 的数字。

```
/* Loop until a random number that is
```

```
   greater than 20,000
*/
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    int i;
    int r;
    r = rand();
    for ( i = 0; r <= 20000; i ++) //  这里的条件表达式没有使用循环控制变量
        r = rand();

    cout << "Number is   " << r << ". It was generated on try " << i << ".";

    return 0;
}
```

上面程序可能的输出如下：

```
Number is   26500. It was generated on try 3.
```

在上面的每次循环中，通过调用函数 rand()生成一个新的随机数。当生成的随机数大于 20,000 的时候，循环的条件就变成了 false， 循环就终止了。

for 循环还可以缺少某些部分

C++中的 for 循环和其它计算机语言相比还有一点需要注意，那就是循环的定义可以缺少某些部分。例如，我们想些一个循环，直到用户从键盘键入 123，那么可以这样写：

```
// A for loop with no increment.
#include <iostream>
using namespace std;
int main()
{
    int x;
    for ( x= 0; x !=123 ; ) //这里缺少了自增表达式
    {
        cout << "Enter a number: ";
        cin >> x;
    }
```

```
    return 0;
}
```

在这个程序中，for 循环定义的自增表达式部分是空缺的。这就是说，每次执行完循环体后，接着去检查看 x 的值是不是等于 123，而不是循环体执行后，执行自增运算，然后才去检查 x 的值。如果我们从键盘键入 123，那么 for 的控制条件就变成了 false，循环也就结束了。for 循环中如果没有自增表达式，那么控制变量是不会被修改的。

还有一种变化的形式如下，就是把把初始化部分放置在 for 循环的外部。

```
x= 0;
for ( ; x < 10 ; ) //这里缺少了自增表达式
{
   cout<< x <<' ';
   ++x;
}
```

这里，for 的初始化部分为空，x 是在进入循环之前进行的初始化。通常只有在初始化值是通过一个复杂的计算过程得到的，而这个过程由于比较复杂而不适合放置在 for 语句中。请注意：这个例子中自增的部分也被移植到了循环体的里面。

无限循环

我们可以使用下面的形式来创建一个无限的循环，也就是永远都不会结束的循环。

```
for(;;)
{
   //...
}
```

这个循环会一直执行下去。尽管有诸如操作系统的命令处理器之类的程序需要这样的无限循环，但是大多数的“无限循环”实际上都是有特殊的要求的。本篇章的最后，我们可以看到如何终止这样的循环。（提示：通过使用 break 来终止这样的循环。）

无循环体的循环

在 C++中，for 循环的循环体可以是空的。这是因为空语句在语法上是正确的。无循环体的循环通常是有用的。例如，下面的程序就是使用了一个无循环体的循环来计算 1 到 10 之间的数字之和。

```
// The body of a for loop can be empty.
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
int main()
{
    int i;
    int sum = 0;
    for ( i = 1; i <= 10 ; sum += i++ ) ; //没有循环体的 for 循环
    cout << "Sum is " << sum;
    return 0;
}
```

程序的输出如下：

```
Sum is 55
```

注意上面的累加运算完全是在 for 语句中完成，for 循环无循环体。需要特别注意自增运算表达式：

```
sum += i++;
```

在 for 循环中声明循环控制变量

通常，控制 for 循环的变量仅仅是为了 for 循环而使用的，别的地方不需要使用它。在这种情况下，我们可以在 for 循环的初始化部分声明这个变量。例如，下面的程序计算了 1 到 5 的累积和与阶乘。控制变量 i 是在 for 循环中声明的。

```
// Declare loop control variable inside the for
#include <iostream>
using namespace std;
int main()
{
    int sum = 0;
    int fact = 1;
    //计算到的阶乘
    for ( int i = 1; i <= 5 ; i++ )
    {
       sum += i; // i  在循环中是可用的。
       fact *= i;
    }
    //但是在循环外，i 是不可用的。
    cout << " Sum is " << sum << "\n";
    cout << " Factorial is " << fact;
    return 0;
}
```

程序的输出如下：

Sum is 15

Factorial is 120

我们在 for 循环中声明变量的时候，有一点需要注意，就是这个变量只能在 for 循环中使用。即，以程序的语言来说，这个变量的范围仅限于这个 for 循环。for 循环外，这个变量就不存在了。因此，在上面的例子中， i 在 for'循环外面是不可用的。如果在程序的其它地方需要使用到这个循环控制变量，我们就不能再 for 循环中声明这个变量。

注意：

for 循环中定义的变量是否在其它地方可以使用是随着时间有所变化的。起初，for 循环中声明的变量在 for 循环之后是可以使用的，但是这点在 C++标准化的过程中有所改变。时至今日，ANSI/ISO 标准 C++把变量的范围限制到了 for 循环内。然而一些编译器则不是这样的。大家需要各自检查自己的编译器是否有这个限制。

在继续学习之前，我们可以自行实验一下，自己写一些 for 循环的变体。你会发现，for 循环是非常有吸引力的。

练习：

1. for 循环是否允许缺少某些部分？
2. 如何使用 for 来创建一个无限循环？
3. for 语句中声明的变量的范围是什么？

必备技能 3.4：while 循环

另外一种循环是 while 循环。While 循环常用的形式是：

while(表达式) 语句;

上面的语句可以是一条语句，也可是一段语句。其中表达式定义了控制循环的条件。它可以是任何有效的表达式。while 循环中的语句或者语句序列在该条件为真 true 的情况下会一直被执行。当条件变成 false 的时候，退出该循环，继而转向执行循环后面的语句。

下面的这个短小的非常有趣的程序就演示了 while 循环的使用。实际上，所有的计算机程序都支除了 ASCII 码外扩展的字符集。扩展字符集通常含有一些诸如外国语言标识和科学标记所用的特殊字符。ASCII 字符集使用的是小于 128 的值。扩展字符集从 128 开始直到 255。下面的程序打印了所有 32 到 255 之间的字符。当运行这个程序的时候，你很有可能看到一些非常有趣的字符。

```
/*
    This program displays all printable characters,
    including the extended character set, if one exists
*/
```

```
#include <iostream>
using namespace std;
int main()
{
    unsigned char ch;
    ch = 32;
    while (ch )
    {
        cout << ch;
        ch++;
    }
    return 0;
}
```

仔细阅读上面的程序，你可能会发现，在这个程序中为什么只使用了一个字符来控制 while 循环了？既然 ch 是一个无符号的字符，它就只能取值 0 到 255。当他的值为 255 的时候，自增后那就得值就会回归到 0 了。因此检查 ch 是不是 0 是一个很便捷的进行循环控制的方法。

和 for 循环一样，while 循环也是在循环开始时先进行控制表达式的检测，这意味着循环体中的语句很有可能会不被执行。这就省去了在循环之前需要进行一个单独的检测。下面的程序演示了 while 循环的这个特性。它打印一行句号。其中句号的数量就是用户键入的数值。程序不允许该行的长度超过 80。其中对允许的句号的数量的判断是在循环的条件表达式中进行的，而不是在其之外进行的。

```
#include <iostream>
using namespace std;
int main()
{
    int len;

    cout << "Enter length ( 1 to 79 ):";
    cin >> len;
    while ( len > 0 && len < 80 )
    {
        cout << '.';
        len--;
    }
```

return 0;

}

如果 len 超过了允许的范围，那么 while 循环的循环体一次都不会被执行。否则，循环一直执行到 len 的值变为 0。这里在 while 循环体就不在需要别的语句了。下面也是一个例子：

while (rand () != 100)

这个循环一直进行到 rand()生成的随机数等于 100。

必备技能 3.5：do-while 循环

C++中的最后一个循环是 do-while 循环。不像 for 和 while 循环，它们都是在执行循环之前先进行控制条件的检测，do-while 循环是在循环体执行之后才进行循环控制条件的检测。这就是说，do-while 循环至少要执行一次。它的通用形式如下：

do

{

语句;

}while(条件);

在循环中只有一条语句的情况下，上面的一对括号并不是必要的，但是通常还是要这样写以便提高程序的可读性。do-while 循环一直执行，只要条件表达式的值为真(true)。

下面的程序会一直循环直到用户键入 100 这个数字：

view plain

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    do
    {
        cout << " Enter a number ( 100 to stop ):";
        cin >> num;
    }while( num != 100 ) ;
    return 0;
}
```

使用 do-while 循环，我们可以进一步改善一下前面的“猜数字”程序。这次，这个程序会一直循环直到用户猜中了数字。

view plain

```
//Magic Number program; 3rd improvement.
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main()
{
    int magic; // magic number
    int guess; // user's guess
    magic = rand();
    do
    {
        cout << "Enter your guess:";
        cin >> guess;
        if ( guess == magic )
        {
            cout << "**Right**";
            cout << magic << " is the magic number. \n";
        }
        else
        {
            cout << "...Sorry, your are wrong.\n";
            if ( guess > magic )
                cout << " Your guess is too high. \n";
            else
                cout << " Your guess is too low. \n";
        }
    }while( guess != magic );
    return 0;
}
```

程序运行的可能结果如下：

Enter your guess:10

...Sorry, your are wrong.

Your guess is too low.

Enter your guess:100

...Sorry, your are wrong.

Your guess is too high.

Enter your guess:50

...Sorry, your are wrong.

Your guess is too high.

Enter your guess:41

Right41 is the magic number.

最后一点：和 for 以及 while 循环一样，do-while 循环体可以为空，但这种情况在实际中很少用到。

练习：

1. while 和 do-while 循环的主要区别是什么？

2. While 循环的循环体可以为空吗？

答案：

1. While 循环中的条件检测是在循环开始前进行的，而 do-while 的循环检测是在循环之后才进行的，因此 do-while 循环至少会执行一次。

2. 是的，while 循环的循环体可以为空。其实 C++中的任何其他循环的循环体都可以为空。

专家答疑：

问：考虑到 C++中内置的灵活性，在选择使用循环的时候，应该有什么样的准则？也就是说，怎么来选择合适的循环了？

答：在明确知道循环次数的情况下通常使用 for 循环；在循环体至少需要被执行一次的情况下使用 do-while 循环。在通常不能明确知道循环需要执行的次数的情况下使用 while 循环。

项目 3-2 对项目 3-1 的帮助系统进行改进

该项目对 3-1 中的帮助系统进行扩展。增加了针对 for，while 以及 do-while 循环的帮助信息。同时，增加了对用户输入的有效性检查。程序一直循环，直到用户输入了有效的菜单选项。其中使用到了 do-while 循环来完成该功能。一般情况下都是使用 do-while 循环来处理菜单选择的。因为我们总是希望循环体至少要被执行一次。

步骤

1.复制 Help.cpp,生成新的文件，重命名为 Help2.cpp。

2.修改显示菜单并等待用户选择的代码：

view plain

```
do
{
    cout << "Help on : \n";
    cout << "    1.if\n";
    cout << "    2. switch\n";
    cout << "    3. for\n";
    cout << "    4. while\n";
    cout << "    5. do-while\n";
    cout << "Choose one :";

    cin >> choice;

}while(choice < '1' || choice > '5');
```

3. 在 switch 语句中增加对 for,while 以及 do-while 的帮助信息，如下：

view plain

```
switch(choice)
    {
    case '1':
        {
            cout << "The if:\n\n";
            cout << "if(condition) statement;\n";
            cout << "else statement;";
            break;
        }
    case '2':
        {
            cout << "The switch:\n\n";
            cout << "switch(expression)\n";
            cout << "    case constant:\n";
            cout << "        statement sequence\n";
            cout << "        break;\n";
            cout << "    //...;";
            break;
        }
    case '3':
        {
            cout << "The for:\n\n";
            cout << "for(init, condition, increment)\n";
            cout << "    statement;\n";
            break;
        }
    case '4':
        {
            cout << "The while:\n\n";
            cout << "while(condition) statement;\n";
            break;
        }
    case '5':
        {
            cout << "The do-while:\n\n";
            cout << "do\n";
            cout << "{\n";
            cout << "    statement;\n";
            cout << "}while(condition);\n";
```

```
            break;
        }
    }
```

注意：在这个版本中没有 default 分支。由于在菜单选择的循环中已经能够确保用户键入的是有效的选择了，因此就没有必要增加对无效输入的处理了。

4.完整的程序如下：

view plain

```
/*
 项目 3-2
 使用 do-while 循环来处理菜单选择，对原来的帮助系统进行改进
*/

int main()
{
    char choice;

    do
    {
        cout << "Help on : \n";
        cout << "    1.if\n";
        cout << "    2. switch\n";
        cout << "    3. for\n";
        cout << "    4. while\n";
        cout << "    5. do-while\n";
        cout << "Choose one :";

        cin >> choice;

    }while(choice < '1' || choice > '5');

    cout << "\n";

    switch(choice)
    {
    case '1':
        {
            cout << "The if:\n\n";
            cout << "if(condition) statement;\n";
            cout << "else statement;";
            break;
        }
    case '2':
```

```
36.         {
37.             cout << "The switch:\n\n";
38.             cout << "switch(expression)\n";
39.             cout << "    case constant:\n";
40.             cout << "        statement sequence\n";
41.             cout << "        break;\n";
42.             cout << "    //...;";
43.             break;
44.         }
45.     case '3':
46.         {
47.             cout << "The for:\n\n";
48.             cout << "for(init, condition, increment)\n";
49.             cout << "    statement;\n";
50.             break;
51.         }
52.     case '4':
53.         {
54.             cout << "The while:\n\n";
55.             cout << "while(condition) statement;\n";
56.             break;
57.         }
58.     case '5':
59.         {
60.             cout << "The do-while:\n\n";
61.             cout << "do\n";
62.             cout << "{\n";
63.             cout << "    statement;\n";
64.             cout << "}while(condition);\n";
65.             break;
66.         }
67.     }
68.
69.     return 0;
70. }
```

专家答疑：

问：前面的程序中看到过在 for 循环的初始化部分声明变量，请问，这种控制变量是否可以在 for 循环之外的其他地方进行声明？

答：是的，可以的。在 C++中，我们是可以在 if 的条件表达式，switch 的条件表达式，while 循环的条件表达式以及 for 循环的初始化部分进行变量声明的。在这些地方声明的变量的作用域仅限于由这些语句控制的代码块中。

之前我们看到过在 for 循环中声明变量。下面就是一个在 if 语句的条件表达式中声明变量的示例：

view plain

```
1.  if ( int  x = 20)
2.  {
3.      xx = x- y;
4.      if ( x > 10 )
5.      {
6.          y = 0;
7.      }
8.  }
```

这里在 if 的条件表达式中声明了变量 x 并给其赋值为 20。既然这是一个真值，if 语句中的目标代码就会被执行。由于在 if 语句中声明的变量的作用域只局限于 if 语句的代码块中，因此，在 if 语句的代码块之外是不能感知到变量 x 的。

正如我们在讨论 for 循环的时候提到的：当我们在一个控制语句中声明变量的时候，改变量的作用域是仅限于该控制语句的代码块中还是在其之后也可以使用取决于编译器。关于这点在编码前我们应该根据自己的编译器进行确认。当然，ANSI/ISO C++规定其作用域仅局限于该控制语句的代码块中。

大部分的程程序员不会在包含 for 在内的控制语句中声明变量。实际上，在除了 for 之外的其他控制语句中声明变量是一种有争议的做法，通常被认为是一种不好的编程风格。

必备技能 3.6：使用 break 来退出循环

使用 break 可以忽略对循环条件的检测，强制性地退出一个循环。当在循环体中遇到 break 语句的时候，循环就会立即终止，继而执行循环后面的语句。下面就是一个示例：

```
#include <iostream>
using namespace std;
int main()
{
    int t;
    //下面的循环只能从 0 执行 10，而不会执行到 100；
    for ( t = 0;t < 100; t++ )
    {
       if ( t == 10 ) break;
       cout << t << ' ';
    }
    return 0;
}
```

程序的输出如下：

0 1 2 3 4 5 6 7 8 9

正如输出的结果所展示的那样，这个程序只是打印了数字 0 到 9，而不是打印到了 100，这是因为使用了 break 语句是的循环提前结束了。

当在使用嵌套的循环的时候，一个 break 语句只能使得程序的执行从最内部的循环中退出。下面就是一个示例：

```
#include <iostream>
using namespace std;
int main()
{
    int t, count;
    for ( t = 0; t < 10; t++ )
    {
        count = 1;
        for (;;)
        {
            cout << count << ' ';
            count ++;
            if ( count == 10 ) break;
        }
        cout << '\n';
    }

    return 0;
}
```

程序的输出如下：

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

可以看出上面的程序共计打印数字 1 到 9 十次。每次打印的时候，在内部循环中都会遇到 break 语句，程序的控制就从内部循环转移到了外部的循环。注意上面的内部循环本是一个无限循环，通过使用 break 来使其终止。

break 语句可以使用在任何的循环体中。通常，在遇到特殊，需要立刻终止循环的时候使用 break 语句，就像上面的例子展示的那样。

还有一点需要注意：switch 中的 break 语句只会影响 switch 语句，而不会影响 switch 所在的循环语句。

必备技能 3.7：使用 continue

实际中也有可能需要提前终止某一次循环，绕过循环体中的正常控制结构。这个功能是通过使用 continue 语句来完成的。continue 语句强制性地执行下一次循环，而跳过从它到控制循环的条件表达式之间的语句。例如，下面的程序打印了 0 到 100 之间的偶数。

```
#include <iostream>
using namespace std;
int main()
{
   int x;
   for( x = 0; x <= 100; x++ )
   {
      if ( x % 2 ) continue;
      cout << x << ' ';
   }
   return 0;
}
```

在上面的程序中，只有偶数才会被打印出来，因为奇数会导致 continue 语句被执行，从而引起本次循环提前结束，也就越过了 cout 语句。　注意%运算符是用来计算一个整数做除法的余数的。因此，当 x 是奇数的时候，对 2 做除法的余数就是 1，也就是真值。当 x 是偶数的时候，对 2 做乘除法的余数是 0，也就是假值。

在 while 和 do-while 循环中，continue 语句会使得控制直接转向到条件表达式，然后继续执行循环。在 for 循环的情况下，循环的自增部分是会被执行的，再执行条件表达式的检测，然后继续循环。

练习：

1. 当在循环体中执行了 break 的时候会怎样？

2. continue 是用来做什么用的？

3. 假设一个循环体中含有 switch，那么 switch 中的 break 会导致循环终止吗？

答案：

1. 当在循环中执行了 break 的时候，就会终止循环，继而执行循环后面的语句。

2. continue 是用来提前终止一次循环的，继而开始下一次的循环，如果循环条件满足。

3. switch 中的 break 不会到时循环终止。

项目 3-3 完善 C++帮助系统

下面我们对之前的 C++帮助信息进行完善：增加针对 break，continue 以及 goto 语句的帮助信息。同时，还允许用户多次查看不同语句的帮助信息。用户可以反复键入菜单选择，直到 q 表示退出程序。

步骤

1.复制 Help2.cpp，重新命名为 Help3.cpp。

2.把整个程序用一个无限循环的 for 语句包裹起来。退出程序时可以使用 break 语句。由于整个程序都是由这个 for 循环的循环体，因此 break 会导致程序退出。

3.修改显示菜单，如下：

view plain

```
1.  do
2.      {
3.          cout << "Help on : \n";
4.          cout << "    1.if\n";
5.          cout << "    2. switch\n";
6.          cout << "    3. for\n";
7.          cout << "    4. while\n";
8.          cout << "    5. do-while\n";
9.          cout << "    6. break\n";
10.         cout << "    7. continue\n";
11.         cout << "    8. goto\n";
12.         cout << "Choose one( q to quit) :";
13.
14.         cin >> choice;
15.
16.     }while(choice < '1' || choice > '8' && choice != 'q');
```

上面显示的菜单中将包含 break，continue 以及 goto 语句。同时接收一个 q 作为退出程序的选项。

4.对原来程序中的 switch 语句进行扩展。增加针对 break，continue 和 goto 的帮助信息。如下：

view plain

```
case '6':
    {
        cout << "The break:\n\n";
        cout << "break;\n";
        break;
    }
case '7':
    {
        cout << "The continue:\n\n";
        cout << "continue;\n";
        break;
    }
case '8':
    {
        cout << "The goto:\n\n";
        cout << "goto label;\n";
        break;
    }
```

5.完整的程序如下：

view plain

```
/*
    项目 3-3
    改进的帮助系统。可以处理多次输入
*/

int main()
{
    char choice;

    for (;;)
    {
        do
        {
            cout << "Help on : \n";
            cout << "    1.if\n";
            cout << "    2.switch\n";
            cout << "    3.for\n";
```

```
18.             cout << "    4.while\n";
19.             cout << "    5.do-while\n";
20.             cout << "    6.break\n";
21.             cout << "    7.continue\n";
22.             cout << "    8.goto\n";
23.             cout << "Choose one( q to quit) :";
24.
25.             cin >> choice;
26.
27.         }while(choice < '1' || choice > '8' && choice != 'q');
28.
29.         if ( choice == 'q')
30.         {
31.             break;
32.         }
33.
34.         cout << "\n";
35.
36.         switch(choice)
37.         {
38.         case '1':
39.             {
40.                 cout << "The if:\n\n";
41.                 cout << "if(condition) statement;\n";
42.                 cout << "else statement;\n";
43.                 break;
44.             }
45.         case '2':
46.             {
47.                 cout << "The switch:\n\n";
48.                 cout << "switch(expression)\n";
49.                 cout << "    case constant:\n";
50.                 cout << "        statement sequence\n";
51.                 cout << "        break;\n";
52.                 cout << "    //...;\n";
53.                 break;
54.             }
55.         case '3':
56.             {
57.                 cout << "The for:\n\n";
58.                 cout << "for(init, condition, increment)\n";
59.                 cout << "    statement;\n";
60.                 break;
61.             }
```

```
62.          case '4':
63.              {
64.                  cout << "The while:\n\n";
65.                  cout << "while(condition) statement;\n";

66.                  break;
67.              }
68.          case '5':
69.              {
70.                  cout << "The do-while:\n\n";
71.                  cout << "do\n";
72.                  cout << "{\n";
73.                  cout << "    statement;\n";
74.                  cout << "}while(condition);\n";
75.                  break;
76.              }
77.          case '6':
78.              {
79.                  cout << "The break:\n\n";
80.                  cout << "break;\n";
81.                  break;
82.              }
83.          case '7':
84.              {
85.                  cout << "The continue:\n\n";
86.                  cout << "continue;\n";
87.                  break;
88.              }
89.          case '8':
90.              {
91.                  cout << "The goto:\n\n";
92.                  cout << "goto label;\n";
93.                  break;
94.              }
95.          }
96.
97.          cout << "\n";
98.
99.      }
100.
101.
102.      return 0;
103.  }
```

6. 一个可能的运行结果如下：

```
Help on :
    1.if
    2.switch
    3.for
    4.while
    5.do-while
    6.break
    7.continue
    8.goto
Choose one( q to quit) :1

The if:

if(condition) statement;
else statement;

Help on :
    1.if
    2.switch
    3.for
    4.while
    5.do-while
    6.break
    7.continue
    8.goto
Choose one( q to quit) :6

The break:

break;

Help on :
    1.if
    2.switch
    3.for
    4.while
    5.do-while
    6.break
    7.continue
    8.goto
Choose one( q to quit) :q
```

必备技能 3.8：嵌套的循环

正如我们在前面的示例中看到的那样，循环是可以嵌套的。嵌套的循环可以用来解决很多问题，也是编程中很重要的一部分。因此，在结束 C++中的循环语句之前，我们应该再来看一个嵌套循环的例子。下面的程序就是使用了嵌套的循环来找出 2 到 100 之间的数字的因子。

```
/*
  使用嵌套的循环来找出到之间的数字的因子
*/
#include <iostream>
using namespace std;
int main()
{
    for( int i = 2; i <= 100;i++ )
    {
        cout << " Factors of " << i << ":";
        for ( int j = 2; j < i; j++ )
        {
            if ( ( i %  j ) == 0 )
            {
                cout << j << " ";
            }
        }
        cout << "\n";
    }
    return 0;
}
```

下面是程序输出结果的一部分：

```
 Factors of 2:
 Factors of 3:
 Factors of 4:2
 Factors of 5:
 Factors of 6:2 3
 Factors of 7:
 Factors of 8:2 4
 Factors of 9:3
```

```
Factors of 10:2 5
Factors of 11:
Factors of 12:2 3 4 6
Factors of 13:
Factors of 14:2 7
Factors of 15:3 5
Factors of 16:2 4 8
Factors of 17:
Factors of 18:2 3 6 9
Factors of 19:
Factors of 20:2 4 5 10
```

在这个程序中，外层的循环是从 2 到 100。内层的循环连续地检查从 2 到 i 的所有数据，如果其能被 i 整除，就输出其值。

必备技能 3.9：使用 goto 语句

goto 是 C++中的无条件的转移语句。因此，遇到 goto 语句的时候，程序的流程直接转移到 goto 语句中指明的地方。这个语句在很多年前曾被程序员们冷落，因为它使得程序看起来很杂乱无章。然后，如今我们依然会偶尔使用它，甚至在某些情况下使用 goto 会更有效。本书并不会对使用 goto 语句的正当性进行批判。但是，应该明确在编程中并不存在非使用 goto 语句不可的情况， 我们不需要使用 goto 语句来使我们的程序变得复杂难懂。然而，在一些特定的情况下，巧妙的使用 goto 语句会使编程变的更加方便。因此，除了在本章以外，本书的其它章节都不会出现 goto 语句。大多数程序员关于 goto 语句考虑更多的是它会使得程序变得很杂乱，近乎不可阅读。然而，某些情况下使用 goto 语句会使得程序变得更加清晰明朗，而不是使其杂乱无章。

goto 语句需要使用一个标签来进行操作。一个标签就是一个有效的 C++标识符，其后面紧跟一个分号。更重要的是，标签必须和 goto 语句在一个函数中。例如，从 1 到 100 的循环，使用 goto 和标签来写的话，如下所示：

```
x = 1;
loop1:
  x++;
  if ( x < 100 ) goto loop1;  //  程序的执行转跳到 loop1
```

一个很适合使用 goto 语句的情况就是从嵌套的程序中退出。例如下面的代码段：

```
for(...)
{
   for ( ... )
```

```
    {
        while ( ... )
        {
            if (...)
                goto stop;
        }
    }
}
stop:
    cout << "Error in program./n"
```

如果上面的程序段中不使用 goto 语句，那么我们就要进行一系列的检查操作了。一个 break 语句是不能完成这个任务的，因为它只能让程序的执行退出最内层的循环。

复习题

1. 编写一个程序：从键盘读入字符，直到键入的字符为$为止。程序对键入的句号进行统计计数，在程序的最后输出统计的总数。
2. 在 switch 结构中，代码序列是否可以从一个 case 运行到下一个 case 分支?为什么？
3. 写出 if-else-if 阶梯形式的通用形式？
4. 考虑下面的代码段：

```
 if ( x < 10 )
    if ( y > 100 )
    {
        if ( !done ) x = z;
        else y = z;
    }
   else cout << " error";
```

最后的 else 是和那个 if 相关联的？

5. 写出用 for 循环实现从 1000 到 0 计数，每次递减 2。
6. 下面的代码是否有效?

```
for ( int i = 0; i < num; i ++)
    sum += i;


count = i;
```

7. 请解释 break 是干什么用的？
8. 在下面的代码段中，break 语句执行后，会输出什么？

```
for ( i = 0; i< 10; i ++)
{
    while ( runing )
    {
        if ( x< y ) break;
    }
    cout << "after while\n";
}
cout << " After for\n";
```

9. 下面的代码段输出什么？

```
for ( i = 0; i< 10; i ++)
{
    cout << i << " ";
    if ( ! (i%2) ) continue;
        cout << "\n";
}
```

10. for 循环中的自增表达式并不一定是每次改变一个固定的取值。实际上，循环控制变量可以以任意的方式进行修改。写一个程序，使用 for 循环来生成并显示级数 1,2,4,8,16,32 等等。

11. 小写字母的 ASCII 码和大写字母的 ASCII 码相差 32。因此，小写字母减去 32 就变成大写字母了。编写一个程序，从键盘读入字符。程序把所有的小写字母转换成大写字母，并把所有的大写字母转换成小写字母，并输出结果。程序不对其它的字符进行转换。程序在用户键入一个句号的时候结束。程序在最后输出大小写转换的次数。

第四篇 数组、字符串和指针

本章我们将讨论数组，字符串和指针。这三者看起来是相互独立的，其实不然。在 C++中，在 C++中它们是相关关联的。对其中一个得了解会帮助我们对另外两个的认识。

数组就是一组有着相同类型的变量的集合，它们共有一个名称。数组可以使一维的或者是多维的，但一维数组是最常用的。数组提供了一种创建一组相关变量的便捷方式。

最常用的数组可能就是字符数组了，因为我们可以用它来存储字符串。C++语言中没有内置的字符串类型。因此，字符串是用过使用字符数组来实现的。这种字符串的实现方式比起语言自身明确定义字符串类型的方式显得更加自由和灵活。

指针是一种取值为内存地址的对象。通常，指针被用来访问另外一个对象的值，而另外的那个对象往往就是一个数组。实际上，指针和数组直接的联系比我们想象的还要紧密。

必备技能 4.1：一维数组

一维数组就是一组相关的变量的列表。这种列表在编程中会经常用到。例如，我们可以使用一维数组来存储网络上被激活的用户的账号；可以使用数组来存储一个棒球队的击球得分率。当我们需要计算一组数值的平均值的时候，通常也是使用数组来存储这些数值。数组是现代编程的基础。

声明一个一维数组的通用形式如下：

类型 名称[大小]；

其中的类型表示数组的基本类型。数组的基本类型决定了构成数组的每个元素的类型。大小指明了数组中可以存储的元素的数量。例如，下面的语句声明了一个名称为 sample 的可以存储 10 个整型数的数组：

```
int sample[10];
```

我们可以通过索引来访问数组中单独的一个元素。索引描述了一个元素在数组中的位置。在 C++中，所有数组的第一个元素的索引都是 0。在前面的例子中，sample 数组有 10 个元素，那么索引的取值就是 0 到 9。我们就是通过这种索引的方式来访问数组中的元素的。书写的方式为：我们在数组名称后面的方括号中写上要访问元素的索引值。所以，数组中的第一个元素可以表示为 sample[0]，最后一个元素可以表示为 sample[9]。例如，下面的程序为数组 sample 装载数值 0 到 9。

```
#include <iostream>
using namespace std;
int main()
{
    int sample[10]; //数组可以存储个元素
   int t;
   //为数组装载数据
   for ( t = 0; t < 10; ++t)
    sample[t]=t;
   //输出数组的值
   for( t= 0; t < 10; ++t)
    cout << "This is sample[" << t << "]" << sample[t] << "\n";
   return 0;
   }
```

程序的输出如下：

```
This is sample[0]0
This is sample[1]1
This is sample[2]2
```

This is sample[3]3

This is sample[4]4

This is sample[5]5

This is sample[6]6

This is sample[7]7

This is sample[8]8

This is sample[9]9

在 C++中，所有数组都是有连续的内存空间构成。（也就说，所有数组中的元素在内存位置上是紧密相连的。）其中最低的地址对应的就是第一个元素，最高的地址对应的就是最后一个元素。例如，下面的代码段运行后：

```
int num[5];
int i;
for ( i = 0; i < 5; i++) nums[i]=i;
```

数组 nums 的存储形式如下：

num[0]	num[1]	num[2]	num[3]	num[4]
0	1	2	3	4

数组之所以在编程中会被经常用到就是因为它允许我们方便地对一组相关的变量进行处理。下面的示例程序中，我们创建了一个含有 10 个元素的数组，并给每个元素赋值。程序然后计算这些数值的平均值，找出其中的最大值和最小值。

```
/*
   计算一组数值的平均值；找出其中的最大值和最小值
*/
#include <iostream>
using namespace std;
int main()
{
    int i,avg, min_val,max_val;
    int nums[10];
    nums[0]=10;
    nums[1]=18;
    nums[2]=75;
    nums[3]=0;
    nums[4]=1;
    nums[5]=56;
```

```
    nums[6]=100;
    nums[7]=12;
    nums[8]=-19;
    nums[9]=88;
    //计算平均值
    avg = 0;
    for ( i = 0; i < 10; i++ )
      avg += nums[i];
    avg /= 10;
    cout << "Average is " << avg << '\n';

    //找出最大值和最小值
    min_val = max_val = nums[0];
    for ( i = 1; i < 10; i++ )
    {
        if ( nums[i] < min_val )
            min_val = nums[i];
        if ( nums[i] > max_val )
            max_val = nums[i];
    }
    cout << "Minimum value: " << min_val << "\n";
    cout << "Maximum value: " << max_val << "\n";
    return 0;
}
```

程序输出如下：

```
Average is 34
Minimum value: -19
Maximum value: 100
```

请注意程序中是如何通过循环来访问数组中的元素的。正如程序所展示的那样，for 循环的控制变量被用来作为数组的索引。当我们使用数组的时候，类似这样的循环是非常普遍的。

我们应该注意一点：在 C++中我们不能把一个数组赋值给另外的一个数组。例如，下面的代码片段是错误的：

```
int a[10], b[10];
//...
```

```
a = b; //错误
```

为了把一个数组中的值赋值给另外一个数组，我们必须对数组中的元素逐个进行赋值。如下：

```
for ( i = 0 ; i < 10 ; i++) a[i]=b[i];
```

不进行边界检测

C++不对数组的边界进行检测。这就是说 C++不会阻止我们对数组的越界访问。换句话说，我们可以对大小为 N 的数组通过大于 N 值的索引来进行访问，而不会引起任何的编译或者运行错误信息。但是这样做通常都会给程序带来灾难性的后果。例如，尽管在下面的代码片段中，我们越界访问了数组 crash，但是编译器依然可以编译并运行这段程序，而不产生任何的错误报告信息。

```
int crash[10], i;
for(i=0; i<100; i++)
    crash[i]=i;
```

此时，尽管数组 crash 中只有 10 个元素，但是循环会进行 100 次。这将导致本不属 crash 数组的内存也被改写。

专家答疑:

问：既然对数组的越界访问可能带来灾难性的后果，为什么 C++在进行数组操作的时候不进行边界检测呢？

答：当初设计 C++的目的是为了专业的程序员能够创建快速的，最高效的的代码。因此，运行时只进行少许的检测。因为这种检查会降低程序的执行速度。实际上，C++期望我们自己，也就是程序员首先来对数组越界访问的问题负责。如果需要，我们自己可以增加相应的检查代码。另外，如果有必要的话，我们还可以自己定义数组类型，来完成对数组越界的检测。

如果程序中对数组进行了越界的访问，被用于其它目的的内存空间，比如用来存储其它变量可能会被改写。如果是对数组进行了越界的读取，那么读取到的无效数据可能会使程序崩溃。换句话说，我们程序员应该确保数组有足够大的空间以便存储我们的数据，也要在必要的时候对数组的边界进行检测。

练习：

1. 什么是一维数组？
2. 数组中第一个元素的索引时 0，正确吗？
3. C++中提供了对数组边界的检查吗？

必备技能 4.2：二维数组

C++支持创建多维数组。最简单的多维数组就是二维数组。一个二维数组在本质上是由

一维数组构成的列表。我们采用下面的方式来声明一个大小为 10，20 的二维数组 twoD：

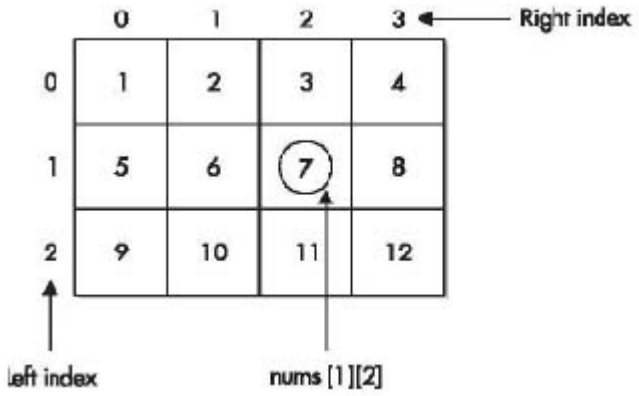
int twoD[10][20];

请注意上面的声明方式。不像别的计算机语言，它们使用逗号来间隔数组的维数。C++中是把每维的大小单独放置在自己的方括号中。和访问一维数组类似，我们使用多个索引来访问多维数组中的元素，每一维的索引单独放置在自己的方括号中。例如，访问数组 twoD 中位于 3,5 处的元素，我们使用 twoD[3][5]这样的写法。

在下面的示例程序中，我们为一个二维数组装载数据 1 到 12。

```
#include <iostream>
using namespace std;
int main()
{
 int t, i, nums[3][4];
 for ( t = 0; t < 3; t++ )
 {
  for ( i = 0; i < 4; i++ )
  {
   nums[t][i] = ( t * 4 ) + i + 1;
   cout << nums[t][i] << ' ';
  }
  cout << "\n";
 }
 return 0;
}
```

在上面的这个例子中，nums[0][0]的值将是 1，nums[0][1]的值将是 2，nums[0][2]的值将是 3，以此类推。nums[2][3]的值将是 12。从概念上来讲，数组的形式如下：



二维数组是以行-列的矩阵的形式存储的。其中的第一个索引表示的是行，第二个索引表示的是列。这就意味着，当我们以这种二维数组被存储在内存中的方式来访问二位数组的时候，右索引（列）的变化要快于左索引（行）。

我们应该明确所有数组元素的存储都是在编译的时候决定的。用于存储数组的空间在数组的生存期都是有用的。就二维数组而言，我们可以通过下面的公式来计算存储它所需的空间大小，单位为字节。

所需的字节数 = 行数 × 列数 × 数组类型的字节数

这里，我们假设一个整型数占用 4 个字节，那么维数为 10，5 的一个整型的二维数组占用的空间的大小就是 10×5×4 个字节。

必备技能 4.3：多维数组

C++允许我们创建多于二维的数组。下面是多维数组声明的通用形式：

类型 名称[大小 1] [大小 2] [大小 3]... [大小 N];

例如，下面的声明就创建了一个 4×10×3 的整型数组：

```
int mutidim[4][10][3];
```

多于三位的数组通常由于需要存储空间较大的原因而很少用到。请记住，分配给数组的存储空间在数组的整个生存周期都是有用的。当我们使用多维数组的时候，可能需要大量的内存空间。比如，维数为 10,6,9,4 的一个四维的字符数组将占用 10×6×9×4=2160 字节空间。如果数组的每一维度都增加到原来的 10 倍，也就是增加到 100×60×90×40，那么所需要的存储空间将增加到 21,600,000 字节！正如我们所看到的那样，大型的多维数组会导致程序的其它部分缺少内存空间。因此，含有多个多于二维或者三维数组的程序很快就会导致内存不足。

练习：

1. 多维数组中的每一维的大小都只单独在自己的方括号中指定的，对吗？
2. 写出如何声明一个名称为 list 的二维整型数组，其大小为 4×9？
3. 结合上面的数组 list，写出如何访问位于 2,3 的元素？

项目 4-1 对数组进行排序

一维数组以基于索引的线性列表来组织其中的数据，这是一种很好的可用于排序的数据结构。在这个项目中，我们将学习一种简单的对数组进行排序的方法。或许你已经知道有多种不同的排序算法。快速排序，筛选排序和希尔排序只是其中的三种。然而，最有名的，也是最简单、最容易理解的排序算法叫做冒泡排序。实际上，冒牌排序算法并不是十分的高效。当使用大数组的时候，它的性能是不可接受的。但是针对小数组来讲，我们倒是可以使用这种排序算法。

步骤：

1. 创建一个文件，命名为 Bubble.cpp

2. 冒泡排序的名字得于这种算法的排序操作。这种算法使用反复的比较，根据比较的结果，交换数组中相邻的元素。在整个过程中，小的数值向数组的一端移动，大的数值向数组的另一端移动。这个过程就像是水槽中的气泡一样。冒泡排序通过多次遍历数组，必要的时候交换乱序的元素来达到排序的目的。为了确保数组中的元素最终是有序，需要遍历数组的次数要等于或者小于数组中元素的数量。

下面是一段进行冒泡排序的代码。被排序的数组叫做 nums。

```
//使用冒泡排序
for ( a = 1; a < size ; a++ )
{
   for ( b = size - 1; b >= a; b-- )
   {
      if ( nums[b-1] > nums[b] ) //如果乱序，则交换元素
      {
         t = nums[b-1];
         nums[b-1] = nums[b];
         nums[b] = t;
      }
   }
}
```

排序使用了两个循环，内部的循环对数组中相邻的两个元素进行检查，查找出乱序的元素，然后交换之。每次这样遍历一遍数组，最小的元素就被移动到了正确的位置。外部的循环保证了这种检查交换的过程一直进行直到整个数组中的元素都被正确排序了。

3. 下面是完整的 Bubble.cpp 的程序：

```
/*
   项目 4-1
   演示冒泡排序
*/
#include <iostream>
#include <cstdlib>
using namespace std;
int main ( )
{
   int nums[10];
```

```
int a,b,t;
int size;
size = 10; //需要排序的元素的数量
//用随机的数值来初始化数组
for ( t = 0; t < size; t++ )
{
   nums[t] = rand();
}
//显示排序前的原始的数值
cout << "Original array is :\n";
for ( t= 0; t < size; t++ )
{
   cout << nums[t] << ' ';
}
cout << '\n';
//使用冒泡排序
for ( a = 1; a < size ; a++ )
{
   for ( b = size - 1; b >= a; b-- )
   {
      if ( nums[b-1] > nums[b] ) //如果乱序，则交换元素
      {
         t = nums[b-1];
         nums[b-1] = nums[b];
         nums[b] = t;
      }
   }
}

//显示出排序后的数组
cout << "\nSorted array is:\n";
for ( t = 0; t < size ; t++ )
{
   cout << nums[t] << ' ';
}
```

```
    return 0;
}
```

程序的输出结果如下：

```
Original array is :
16807 282475249 1622650073 984943658 1144108930 470211272 101027544
1457850878 1458777923 2007237709
Sorted array is:
16807 101027544 282475249 470211272 984943658 1144108930 1457850878
1458777923 1622650073 2007237709
```

尽管冒泡排序对于小的数组很有效，但是对于大数组则效率很低下。最好的通用的排序算法是快速排序。但是由于快速排序中使用到了我们还没有学习的 C++特性。还有，C++标准库中已经包含了函数 qsort()来实现快速排序。为了使用这个函数，我们还需要学习更多的 C++知识。

必备技能 4.4: 字符串

到目前为止，一维数组最常用的就是用来创建字符串。C++支持两种类型的字符串。第一种，也是最常用的一种，就是以 0 结束的字符串，它是一个一维数组，其中用 0 表示字符串结束。因此，一个以 0 结束的字符串包含了组成这个字符串的字符和紧跟在最后面的 0（字符'/0')。这种以 0 结束的字符串由于提供了很高的效率和对字符串的详细操作而得到广泛的使用。当 C++程序员在谈论到字符串的时候，他（她）通常指的就是这种以 0 结束的字符串。第二种字符串指的是 C++中定义的字符串类，它是 C++类库中的一部分。所以，字符串不是 C++内置的数据类型。虽然字符串类提供了面向对象的字符串处理方法，但是它的是使用没有以 0 结束的字符串的使用广泛。这里我们只研究以 0 结束的字符串。

字符串的基本知识

当使用数组来定义以 0 结束的字符串的时候，数组的大小需要比字符串中字符的数量大 1。例如，如果我们想定义一个数组表示的字符串，它含有 10 个字符，那么应该这样写：

```
char str[11];
```

之所以定义数组的大小是 11 而不是 10 是为了给字符串的末尾的 0 保留空间。正如在前面学习到的那样，C++运行程序员定义字符串常量。一个字符串常量就是用双引号引起来的字符的序列。下面是一些示例：

“hello there” “I like C++” “Mars” “”

在定义字符串常量的时候不需要手工在字符串的末尾添加结束标记 0。这个有 C++编译器来完成。因此，字符串”Mars”在内存中的存储情况如下：

M	a	r	s	0

上面示例的最后一个字符串为“”。这种字符串被称为空字符串。它只包含了字符串的结束标识，并没有其它的字符。空字符串是有用的，它代表的是一个不含有任何字符的字符串。

从键盘读取一个字符串

从键盘读取一个字符串的最简单的方式就是使用 cin 语句和字符数组。例如，下面的程序读入一个用户输入的字符串：

```
#include <iostream>
using namespace std;
int main ()
{
    char str[90];
    cout << "Enter a string:";
    cin   >> str ;  //使用 cin 从键盘读入一个字符串
    cout << "Here is your string:";
    cout << str;
    return 0;
}
```

程序运行的结果可能如下：

```
Enter a string:testing
Here is your string:testing
```

尽管上面的程序从技术上来讲是正确的，但是它并不总是能按照我们的预期来工作。为了明确为什么，我们可以再次运行这个程序，键入字符串“This is a test”。看到的结果如下：

```
Enter a string:This is a test
Here is your string:This
```

当程序输出字符串的时候，只是输出了“This”，而不是整个句子。这是因为 C++中的输入输出系统在读取字符串的时会在遇到第一个空白的时候就停止。空白可以是空格，制表符和换行符。

一个用于解决空白带来的问题的方法就是使用 C++的库函数 gets()。gets()的常用的使用方法如下：

gets(数组名)；

在读取字符串的时候，调用 gets()方法，传入一个数组名作为参数，不用传入任何索引。gets()函数调用结束后，指定的数组中就包含了从键盘读入的字符串。gets()函数会一直读取字符，包括空白，直到用户键入一个回车键。使用这个函数，需要引入头文件<cstdio>。

用 gets()方法获取含有空格的字符串，重写上面的程序如下：

```
#include <iostream>
#include <cstdio>
using namespace std;
int main ()
{
    char str[90];
    cout << "Enter a string:";
    gets(str) ;  //使用 cin 从键盘读入一个字符串
    cout << "Here is your string:";
    cout << str;
    return 0;
}
```

输出结果如下：

```
Enter a string:This is a test
Here is your string:This is a test
```

从上面的结果中可以看出，空格也被读入到了字符串中。还有一点需要注意，就是在 cout 语句中直接使用了 str。通常，用于存储字符串的字符数组的名字可以出现在任何字符串常量可以出现的地方。

必须注意：cin 和 gets()都不对数组的边界进行检查。因此，如果用户键入的字符串的长度大于了数组的大小，就会造成写数组越界。在后面的章节，我们会学习到另外的可以避免这种情况发生的方法。

练习：

1. 什么是 null-terminated 字符串？
2. 为了存储一个含有 8 个字符的字符串，字符数组的大小应该是多少？
3. 使用什么样的函数可以从键盘读取到含有空格的字符串？

必备技能 4.5: 一些字符串相关的库函数

C++支持很多处理字符串的函数。最常用的有：

```
strcpy()
strcat()
strcmp()
strlen()
```

这些处理字符串的函数使用同一个头文件，<cstring>。下面我们就学习一下这些函数的使用方法。

strcpy

该函数的常用形式如下：

strcpy(目的，源)；

这个函数把源字符串的内容复制到目的字符串中。注意，用于目的字符串的数组的大小必须足够大，以便存储源字符串中的内容。如果目的字符串的数组大小不够大，则会导致数组写越界，这将有可能导致程序崩溃。

strcat

该函数的常用形式如下：

strcat(字符串 1，字符串 2);

函数把字符串 2 的内容追加到字符串 1 的尾部。字符串 2 保持不变。使用这个函数的时候，也需要保证用于存储字符串 1 的数组要足够大，能够保存得下字符串 1 和字符串 2 的内容。

strcmp

该函数的常用形式如下：

strcmp(字符串 1，字符串 2);

该函数对两个字符串进行比较。如果两个字符串的内容相同，则返回 0；如果字符串 1 根据字典序大于字符串 2，则返回一个正数；如果字符串 1 根据字典序小于字符串 2 则返回一个负数。

使用这个函数需要注意的是如果两个字符串相同，函数的返回值为假。因此，如果需要在两个字符串相同的时候作某些事情，就必须在判断语句中使用！。例如，下面代码片段中控制 if 语句的条件只有在 str 的内容和“C++”相同的时候才为真。

```
if ( !strcmp(str, “C++”)) cout << “str is C++”;
```

strlen

该函数的通用形式如下：

strlen(字符串);

函数返回字符串的长度。

使用字符串函数的示例:

```
//演示字符串函数的使用
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
int main ()
{
    char s1[80], s2[80];
```

```
    strcpy(s1, "C++");
    strcpy(s2, " is power programmimg.");
    cout << "lengths:" << strlen(s1);
    cout << ' ' << strlen(s2) << '\n';
    if (!strcmp(s1,s2))
       cout << "The strings are equal\n";
    else cout << "not equal\n";
    strcat(s1,s2);
    cout << s1 <<'\n';
    strcpy(s2,s1);
    cout << s1 << " and " << s2 << "\n";
    if (!strcmp(s1,s2))
       cout << "s1 and s2 are now the same.\n";
    return 0;
}
```

程序的输出结果如下：

```
lengths:3 22
not equal
C++ is power programmimg.
C++ is power programmimg. and C++ is power programmimg.
s1 and s2 are now the same.
```

使用字符串结束标识

字符串以 0 结尾这个特点可以被用来简化程序。例如，下面的程序把一个字符串转换为大写的。

```
//把一个字符串转换为大写的
#include <iostream>
#include <cstring>
#include <cctype>
using namespace std;
int main ()
{
   char str[80];
   int i ;
   strcpy(str, "This is a test");
```

```
  for ( i = 0; str[i]; i++ )
  {
    str[i] = toupper(str[i]);
  }
  cout << str;
  return 0;
}
```

程序的输出结果如下：

THIS IS A TEST

上面的这个程序中用到了库函数 toupper()。这个函数返回字符参数的大写字符。使用该函数，需要引入头文件<cctype>

需要注意的是 for 循环的控制条件为控制变量索引的字符数组的元素。这样作是合理的，因为真值就是任何非零的值。请记住，所有的字符值都是非零的，只有字符串的结束标志是 0。因此，这个循环会一直进行，直到遇到字符串的结束标志，也就是 str[i]为零的时候。因为所用的用户标记字符结束的 0 都是在字符传的最末尾，所以上面的循环能够正确第完成我们预期的任务。在专业的 C++代码中，我们会看到很多这种使用字符串结束标记的例子。

专家答疑

问：除了 toupper()之外，C++还支持别的字符处理函数吗？

答：是的。C++标准库包含了几个其他的处理字符的函数。例如与 toupper()相对应的 tolower()。它返回一个字符对应的小写的字符。我们还可以使用函数 isupper()来判断一个字符是不是大写的。如果字符是大写的，这个函数返回真值。还有 islower()函数，如果字符是小写的，函数返回真值。还有其他的函数，如 isalpha(), isdigit(), isspace()和 ispunct()。这些函数的参数都是一个字符，函数返回字符的类型。例如， isalpha()函数，如果字符是字母表中的字符，则函数返回真值。

练习：

1. strcat 函数是用来作什么的？
2. 当用 strcmp 函数比较两个相同的字符串的时候，函数返回什么？
3. 写出如何获取名为 mystr 的字符串的长度?

必备技能 4.6: 数组的初始化

在 C++中我们可以对数组进行初始化。数组的初始化和其它变量的初始化有些类似，如下：

数组类型数组名[数组大小] ＝ {值列表};

其中，值列表中值的类型应该和数组的类型相兼容。列表中所有的值用逗号隔开。列表中的

第一个数值会被放置在数组的第一位置上，第二个数值被放置在数组的第二个位置上，以此类推。注意，在{}后面需要紧跟一个分号。

在下面的示例中，我们用数字 1 到 10 来初始化一个 10 个整型数的数组。

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

这意味着，i[0]的值为 1，i[9]的值为 10。用于存储字符串的字符数组可以采用如下的简易的初始化方式：

```
char  数组名[数组大小] = "string";
```

例如，下面的代码片段用字符串“C++”来初始化数组 str：

```
char str[4]=”C++”;
```

这种写法和下面的写法是等价的：

```
char str[4]={ 'C', '+', '+', '\n'};
```

由于在 C++中字符串必须是以 0 结尾的，因此在声明用于存储字符串的字符数组的时候，我们必须保证数组足够大。这也是为什么上面的 str 数组的大小是 4 个字符，而不是“C++”中的三个字符。当使用字符串常量的时候，编译器会自动添加 0 作为字符串的结束标记的。

多维数组的声明和一维数组的声明是类似的。例如，下面的程序中，用数字 1 到 10 和它们的平方来初始化数组 sqrs：

```
int sqrs[10][2] =
{
        1,1,
        2,4,
        3,9,
        4,16,
        5,25,
        6,36,
        7,49,
        8,64,
        9,81,
        10,100
};
```

表 4－1 展示了数组 sqrs 在内存中的存储影像。

	0	1	右索引
0	1	1	
1	2	4	
2	3	9	

3	4	16	
4	5	25	
5	6	36	
6	7	49	
7	8	64	
8	9	81	
9	10	100	
左索引			

表 4-1 sqrs 数组的初始化

当初始化多维数组的时候，还可以使用花括号来把每一维的数据括起来。这被称为子集分组。例如，上面的初始化 sqrs 数组的程序也可以写成：

```
int sqrs[10][2] =
{
        {1,1},
        {2,4},
        {3,9},
        {4,16},
        {5,25},
        {6,36},
        {7,49},
        {8,64},
        {9,81},
        {10,100}
};
```

当使用子集分组的方式来初始化数组的时候，如果没有提供足够的初始化数据，则剩余的元素都会被自动初始化为 0。

下面的程序使用 sqrs 数组来查找用户键入的一个数字的平方。首先在数组中查找这个数字，然后打印出对应的平方值。

view plain

```
1. #include <iostream>
2. #include <cstring>
3. #include <cctype>
4.
5. using namespace std;
6.
7. int main ()
```

```
{
    int i,j;
    int sqrs[10][2] =
    {
        {1,1},
        {2,4},
        {3,9},
        {4,16},
        {5,25},
        {6,36},
        {7,49},
        {8,64},
        {9,81},
        {10,100}
    };

    cout << "Enter a number between 1 and 10: ";
    cin >> i;

    //查找 i
    for ( j = 0; j < 10 ; j++ )
    {
        if ( sqrs[j][0] == i )
            break;
    }

    cout << "The square of " << i << " is ";

    cout << sqrs[j][1];

    return 0;
}
```

程序的运行结果如下：

Enter a number between 1 and 10: 4

The square of 4 is 16

未指定大小的数组的初始化

当需要在声明时初始化一个数组，我们还可以使用 C++提供的自动计算数组大小的功能，只要不指明数组的大小即可。实际上，编译器通过计算初始化数据的个数来创建一个足够大的数组来保存这些初始的数据。例如：

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

这些的写法就是创建了一个名字为 nums，大小为 4 的初始数据为 1，2，3，4 的数组。

因为没有明确指定数组的大小，这样的数组被称为未指定大小的数组。这样的数组是非常用的。例如，假想我们需要使用初始化数组的方式来定义一个网址的列表，如下：

```
char e1[16] = "www.osborne.com";
char e2[16] = "www.weather.com";
char e3[15] = "www.amazon.com";
```

如果手工去计算每个网址中含有的字符的多少来确定数组的大小，这将是一件非常累人的事情。而且，这样很容易计数错误而导致数组的大小设置不当。如果让编译器来计算数组的大小，如下的写法则会更好：

```
char e1[] = "www.osborne.com";
char e2[] = "www.weather.com";
char e3[] = "www.amazon.com";
```

这样写除了可以免去累人的手工计数外，这种未指定大小的数组的初始化还可以让我们随便修改字符串的内容，而不用考虑数组的大小是否也需要重新定义。

这种未指定大小的数组的初始化并不局限于针对一维数组。对于多维数组来讲，最左端的维的可以为空。（其它维的大小必须明确指定，以便数组中的元素都可以被正确的索引。）使用这种未指定大小的数组初始化的方法，我们可以构建长度不同的表，由编译器自动为它们分配存储空间。例如，sqrs 数组就可以采用未指定大小的数组的初始化的方式来进行初始化：

```
int sqrs[][2] = {1,1,2,4,3,9,4,16,5,25,6,36,7,49,8,64,9,81,10,100};
```

这种方法和明确指定数组大小的方式相比较的优点就是通过修改初始化数组列表，这张表的长度可以变长或者变短而不用改变数组维的大小。

必备技能 4.7：字符串数组

一种特殊的二维数组就是字符串数组。在程序中使用字符串构成的数组并不罕见。比如在需要访问数据库的程序中，输入进程需要使用字符串数组来对用户键入的命令进行有效性验证。我们使用二维的字符数组来创建字符串数组。其中的左索引用来表示字符串的数量，右索引用来表示字符串的最大长度。包括字符串结束标志。例如，下面的代码片段声明了 30 个字符串的数组，每个字符串的最大长度维 79，在加上一个结束表示共计为 80。

```
char str_array[30][80];
```

对其中每一个单独字符串的访问是很简单的：我们只需要指定左索引即可。例如，下面 gets() 函数就用到了 str_array 的第三个字符串：

```
gets(str_array[2]);
```

访问第三个字符串的每个字符，我们可以使用如下的语句：

```
cout << str_array[2][3];
```

上面的语句把第三个字符串的第四个字符打印出来。

下面的程序通过实现一个简单的电话本功能来演示字符串数组的使用。二维数组 numbers 用来存储姓名和电话号码。查找电话号码时，输入姓名，程序会输出对应的电话号码。

```
//一个简单的电话本程序
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    int i;
    char str[80];
    char numbers[10][80] =
    {
        "Tom","555-3322",
        "Mary","555-8976",
        "Jon","555-1037",
        "Rachel","555-1400",
        "Sherry","5558873"
    };
    cout << "Enter name: ";
    cin >> str;
    for ( i = 0; i < 10 ; i+=2 )
    {
        if (!strcmp(str, numbers[i] ))
        {
            cout << "Number is " << numbers [i+1] << "\n";
            break;
        }
    }
    if ( i == 10 )
    {
         cout << "Not found.\n";
    }
```

```
    return 0;
}
```

可能的输出结果如下：

```
Enter name: Jon
Number is 555-1037
```

注意在上面的 for 循环中循环控制变量的变化是每次递增 2。这样作是非常有必要的，因为数组中的姓名和电话号码是交替出现的。

练习：

1. 写出如何用 1，2，3，4 来初始化一个有四个元素的整型数组。
2. 重写下面的初始化: char str[6]={'H', 'e', 'l', 'l', 'o', '\0'};
3. 用未指定大小数组的方式重新下面的代码：int nums[4]={44,55,66,77};

必备技能 4.8：指针

指针是 C++中最强大的一个特性之一。同时它也是最容易出问题的一个特性。尽管在使用指针的时候很容易出错，指针仍是 C++中的一个关键部分。例如，指针使得 C++可以支持链表和动态内存分配。它还提供了一种由函数来改变参数的内容的方法。然而，这些和其它的指针的用法将在本书的后续章节中进行讨论。本章中，我们将学习指针的基本知识和一些简单的使用方法。

什么是指针？

指针就是含有内存地址的对象。通常情况下，这个地址是另外一个对象的位置，比如，可以是另外一个变量的位置。举例说明一下，如果 x 含有 y 的地址，那么就说 x 指向 y。

指针变量必须按照下面的方式进行声明：

```
类型 *变量名；
```

这里，类型是指针基本类型。这个类型决定了指针指向数据的类型。变量名就是指针变量的名称。例如，声明 ip 为一个指向 int 数据的指针的语句如下：

```
int *ip;
```

既然 ip 的基本类型为整型 int，它就可以指向任何整型的数据。下面声明了一个 float 类型的指针：

```
float *fp;
```

fp 的基本类型为 float，它就可以指向任何类型为 float 的数据。通常情况下，只需要在声明变量的语句中在变量的前面加上*就可以把这个变量变成一个指针。

必备技能 4.9: 指针运算符

在使用指针的时候，有两个特殊的运算符要用到：*和&。其中&是一个单目运算符，它返回操作数的内存地址。（单目运算符只需要一个操作数参与运算。）

例如：

ptr = &total;

上面的语句把变量 total 的地址赋值给 ptr。这个地址就是变量 total 在内存中的位置。它和变量 total 的取值没有任何关系。取地址运算&可以看做是返回某某变量的地址。因此上面的赋值语句可以表述成“为 ptr 赋值为 total 的地址“。为了更好的理解这种赋值运算，我们假设变量 total 在内存中的地址为 100。那么经过这个赋值语句后，ptr 的值就是 100。

第二个运算符是*，它是对&运算符的补充。它也是一个单目的运算符。它返回运算数的值对应的内存地址的值。我们继续使用前面的那个例子，如果 ptr 的值为变量 total 的内存地址，那么

val = *ptr;

将把 total 变量的值赋值给 val。例如，如果 total 变量的值为 3200，那么 val 的值就是 3200，因为 3200 就是存储在位置为 100 的内存中的值，而 ptr 的值就是 100。所以*运算符可以看做是返回某个内存地址上的值。因此，上面的这个语句可以表述为“把 ptr 指向的内存地址的值赋值给 val”。

下面的程序演示了对上面的操作：

```
#include <iostream>
using namespace std;
int main()
{
    int total;
    int *ptr;
    int val;
    total = 3200; //给变量 total 赋值为 3200
    ptr = &total; //给 ptr 赋值为变量 total 的地址
    //把指针 ptr 指向的变量的值赋值给 val，也就是把 ptr 的值作为一个内存地址
    //然后把这个内存地址中的值赋值给 val
    val = *ptr ;
    cout << "Total is:" << val << '\n';
    return 0;
}
```

需要注意的是，乘法的运算符和取某个内存地址的值的运算符是相同的。这点对于 C++初学者来说可以会引起混淆。这两者之间没有任何的关系。请记住，&和*运算符的优先级是高于任何算术运算符的，而它们和单目运算符负的(-)的优先级是相同的。

使用指针的方法通常被称作是间接的方式，因为我们是通过一个变量间接地访问另外一个变量。

练习：

1. 什么是指针？

2. 写出如何声明一个名为 valPrt 的指向一个 long int 类型的指针？

3. 和指针相关的两个运算符*和&的作用是什么？

指针的基本类型很重要

在前面的讨论中，我们看到，我们可以通过间接的使用指针来把 total 变量的值赋值给 val 变量。 你也许会考虑到一个很重要的问题：C++是如何知道需要从 ptr 所指向的地址中赋值多少个字节的值到 val 变量中呢？或者，更通用一点的问题：在涉及指针的赋值时，C++编译器是如何决定需要复制的字节的多少呢？针对前面的例子，因为 ptr 是一个整型数指针，所以需要从 ptr 执行的地址开始复制 4 个字节数据到 val 中（假设整型数的大小为 32 位）。然而，如果 ptr 被定义成 double 类型的指针，则需要复制 8 个字节的数据到 val 中。

指针变量总是指向相应的正确类型的数据，这一点至关重要。例如，当我们声明一个 int 类型的指针的时候，C++编译器会认为它所指向的任何数据都是一个 int 类型的数据。如果它所指向的数据不是 int 类型，通常很快就会出现问题。例如，下面的写法是错误的：

```
int * p;
double f;
//…
p=&f;  //这种写法是错误的。
```

上面的代码片段是无效的，因为我们不能把一个 double 类型的指针赋值给一个 int 类型的指针。也就是说，&f 产生的是一个 double 类型的指针，但是 p 是一个 int 类型的指针。这两种类型是不兼容的。（实际上，编译器会在这条语句这里报告编译错误，说明是不兼容赋值。）

尽管在指针相互赋值的时候，两个指针的类型必须是兼容的，但是我们可以使用强制类型转换来突破这个限制。例如，下面的代码种计数上来讲是正确的：

```
int * p;
double f;
//…
p=(int*)&f;
```

强制的类型转换把 double 类型的指针转换成了一个 int 类型的指针。然而，这种使用强制类型的做法是有争议的。因为指针的基本类型决定了编译器如何对待它所指向的数据。在这种情况下，尽管 p 实际上是指向了一个 double 类型的数据，但是编译器会认为 p 指向的是一个 int 类型的数据，因为 p 的基本类型位 int 类型。为了更好地理解这点，考虑下面的程序：

```
//下面的程序将不能正确工作。
```

```
#include <iostream>
using namespace std;
int main()
{
    double x,y;
    int *p;
    x = 123.33;
    p = (int *)&x; //强制地把 double 类型的指针转换位 int 类型的指针
    y = *p;  // 会发生什么情况了？
    cout << y;
    return 0;
}
```

下面是程序的输出。（你可能会看到不同的输出结果。）

```
-1.20259e+09
```

这个数值显然不是 123.33！这是为什么了？在程序中，p 是一个 int 类型的指针， 它被赋值为一个 double 类型的数据的地址。因此，当通过 p 给 y 赋值的时候，y 只是被复制了 4 个字节的数据（而不是 double 类型的 8 个字节），因为 p 是一个 int 类型的指针。因此，cout 语句输出的数据不是 123.33 而是一个垃圾数据。

通过指针来赋值

我们可以在赋值语句的左边使用指针来把一个内存地址赋值给该指针。假设 p 是一个 int 类型的指针，下面的赋值语句将把 101 赋值给 p 所指向的位置。

```
*p=101;
```

上面的这个赋值语句可以表述为：“把 101 赋值给 p 虽指向的位置。“如果需要对 p 执行的位置的值进行自增或者自减我们可以使用下面的语句：

```
(*p)++;
```

其中括号是有必要的，因为*运算符的优先级是低于++运算符的。

下面的程序演示了使用指针来进行赋值：

```
#include <iostream>
using namespace std;
int main ()
{
    int *p, num;
    p = &num;
    *p = 100; //通过使用 p 来给 num 赋值为 100
    cout << num << ' ';
```

```
    (*p)++;   // 通过 p 来实现 num 的自增
    cout << num << ' ';
    (*p)--;   // 通过 p 来实现 num 的自减
    cout << num << ' ';
    return 0;
}
```

程序的输出如下：

100 101 100

必备技能 4.10:指针表达式

在大部分的 C++表达式中都可以使用指针。然而，指针的使用还是有一些规则的。还需要注意在使用指针的时候我们可能需要使用括号来把表达式的部分括起来，以便得到我们预期的结果。

指针用于算术表达式

只有四种可以用于指针的算术运算符：++,--,+和－。为了理解指针用于算术表达式会发生什么，我们假设 p1 是一个 int 类型的指针，p1 的值为 2000（也就是说它指向的地址为 2000）。假设我们使用的是 32 的整型类型，那么在表达式

p1++；

之后，p1 的值就会是 2004，而不是 2001！这是因为，每次 p1 自增的时候，它就会指向下一个 int 数据。同样的原理也使用于自减。例如，假定 p1 的值为 2000，那么表达式

p1--；

将使得 p1 的值变为 1996。

把上面示例中的原则推广开来，应用于指针的算术运算符有如下的规则：每次指针自增的时候，它会指向下一个基本类型的内存地址。每次指针自减的时候，它会指向上一个基本类型的内存地址。如果指针的类型为 char 类型，那么指针的自增和自减就和平时理解的自增和自减是一致的，这是因为 char 类型正好是一个字节。然而，任何其它类型的指针的自增和自减都是增加或者减少了基本类型的长度大小。

上面的规则不仅仅局限于自增或者自减。我们还可以对指针加上一个正数或者减去一个正数。表达式

p1=p1+9;

会使得 p1 指向后面的第九个基本类型的位置。

尽管我们不能把两个指针加起来，但是我们可以对两个指针进行相减（前提是它们的基本类型是一样的）。减法的结果就是两个指针的位置之间的基本类型数据的数量。

除了可以把指针加上一个正数或者把指针和一个正数相减，以及两个指针之间的相减之外，指针不能用于其他的算术运算。例如，我们不能对一个指针加上一个 float 或者 double 类型

的数据，也不能种指针中减去一个 float 或者 double 类型的数据。

为了更加清楚的看到指针的算术运算的结果，可以执行下面的程序。下面的程序创建了一个 int 类型的指针 i 和一个 double 类型的指针 f，然后分别给这两个这指针加上 0-9 的数字，并打印出结果。仔细观察输出地址的变化和指针基本类型的关系。（在 32 位的编译器下，i 每次增加 4，f 每次增加 8）注意在 cout 语句中使用指针会直接输出它的地址，输出的格式地址格式是 CPU 和环境可用的地址格式。

```
#include <iostream>
using namespace std;
int main ()
{
    int *i,j[10];
    double *f, g[10];
    int x;
    i = j;
    f = g;
    for ( x = 0; x < 10; x++ )
    {
       cout << i+x << ' ' << f+x << '\n';
    }
    return 0;
}
```

程序的输出结果如下（你还可能看到其他的输出结果）：

```
0xbffff69c 0xbffff648
0xbffff6a0 0xbffff650
0xbffff6a4 0xbffff658
0xbffff6a8 0xbffff660
0xbffff6ac 0xbffff668
0xbffff6b0 0xbffff670
0xbffff6b4 0xbffff678
0xbffff6b8 0xbffff680
0xbffff6bc 0xbffff688
0xbffff6c0 0xbffff690
```

指针的比较

我们可以用关系运算符，例如==,<和>来比较指针。通常情况下，为了保证两个指针相比较的结果是有意义的，这两个指针必须是有关系的。例如，两者指针是指向同一个数组的元

素的。（在工程 4-2 中会看到这样的例子。）然而还有一种指针的比较：任何指针都可以可以空指针进行比较，空指针就是 0。

必备技能 4.11：指针和数组

在 C++中，指针和数组有着紧密的联系。实际上，指针和数组经常是可以互换的。考虑下面的代码片段：

```
char str[80];
char *p1;
p1=str;
```

这里，str 是一个 80 个字符大小的数组，p1 是一个字符指针。然而，第三行的代码就很有意思了。该行中，p1 被赋值为数组 str 中的第一个元素的地址。（也就是说在这个语句之后，p1 就指向 str[0]了。）为什么了？这是因为在 C++中使用数组名，而不使用索引会生成一个指向该数组的第一个元素的指针。因此，赋值语句

```
p1=str;
```

就是把 str[0]的地址赋值给了 p1。理解这一点是非常重要的：当在一个表达式中使用了没有索引的数组名字的时候。它将生成一个指向数组第一个元素的指针。既然如此，在这个赋值语句之后，p1 就是指向 str 的开始，那么我们就可以使用 p1 来访问数组中的元素。例如，我们需要访问数组中的第五个元素，我们可以使用

```
str[4]
```

或者

```
*(p1+4)
```

这两中写法都能获取到数组中的第五个元素。请记住，数组的起始索引是 0，所以，当使用索引 4 的时候，我们访问到的是数组的第五个元素。因为 p1 当前的值是指向数组的第一个元素的，所以给指针 p1 加上 4 得到的也是第五个元素的地址 。

p1+4 必须用挂号括起来，这是因为*运算符的优先级别高于+运算符。如果没有括号，那么这个表达式将先是得到 p1 指向的地址的值（也就是数组的第一个元素），然后在给这个值加一。实际上，在 C++中有两种用来访问数组元素的方法：有指针参与的算术运算和对数组的索引。这点非常重要，因为指针参与的算术运算有时候比对数组的索引要要快很多，特别是在访问一个元素之间有严格顺序的数组的时候。由于在编程的时候，我们经常需要考虑速度的问题，所以用指针来访问数组元素在编程中就变得非常普遍。同样，有时候使用指针可以使得代码变得更紧凑。

下面的示例程序演示了使用数组的索引和指针的算术运算来访问数组元素的不同之处。这里我们创建两个版本的程序来完成字符串中大小写转换的功能。第一个是使用数组的索引来完成，第二个是使用指针的算术运算来完成。第一个版本的程序如下：

```
#include <iostream>
```

```
#include <cstring>
using namespace std;
int main ()
{
    int i;
    char str[80] = "This Is A Test";
    cout << "Original string: " << str << "\n";
    for ( i = 0; str[i]; i++)
    {
       if ( isupper( str[i] ) )
       {
          str[i] = tolower( str[i] );
       }
       else if ( islower(str[i]) )
       {
          str[i] = toupper( str[i] );
       }
    }
    cout << "Inverted-case string: " << str;
    return 0;
}
```

上面程序的输出如下：

```
Original string: This Is A Test
Inverted-case string: tHIS iS a test
```

注意上面的程序中使用到了 isupper()和 islower()两个库函数来判断字母的大小写。函数 isupper()在参数的字母是大写的时候返回真值；islower 函数在参数的字母是小写的时候返回真值。在 for 循环中，通过了索引来访问数组中的元素，检查每个字母的大小写。for 循环直到遍历到字符串的结束标识时结束。因为 0 就是假值，循环就此结束。

下面是用指针的算术运算来重写上面的程序：

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
```

```
    char *p;
    char str[80] = "This is A test";
    cout << "Original string: " << str << "\n";
    p = str; //给 p 赋值为数组的首地址
    while(*p)
    {
       if ( isupper(*p) )
       {
          *p = tolower( *p );
       }
       else if ( islower( *p ) )
       {
          *p = toupper( *p );
       }
       p++;
    }
    cout << "Inverted-case string : " << str ;
    return 0;
}
```

在这个版本中，p 被赋值为数组 str 的首地址。在 while 循环中，检查 p 指针指向位置的字母的大小写，并进行转换后，对 p 进行自增。循环直到遇到字符串的结束标识。由于一些 c++编译器生成代码的方式的不同，上面的这两种方式在性能和效率上也不尽相同。通常来说，使用对数组的索引需要更多的机器指令。因此，在专业的 C++代码中，使用上面的第二个版本，也就是通过指针来访问数组中的元素的方式更为普遍。然而，作为 C++的初学者，我们可以随意使用数组的索引这种方法，直到我们灵活地掌握了指针的用法。

对指针进行索引

正如我们前面看到的那样，我们可以通过指针的算术运算来访问数组中的元素。反之亦然！在 C++中我们还可以通过对指针进行索引，把它像数组一样使用。下面就是使用了这种方式的字母大小写进行转换的第三个版本：

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
    int i;
```

```
    char *p;
    char str[80] = "This is A test";
    cout << "Original string: " << str << "\n";
    p = str; //给 p 赋值为数组的首地址
    //下面通过对 p 的索引来访问数组中的元素
    for ( i = 0; p[i] ; i++ )
    {
       if ( isupper( p[i] ) )
       {
          p[i] = tolower( p[i] );   //把 p 当作数组来使用
       }
       else if ( islower( p[i] ) )
       {
           p[i] = toupper( p[i] );
       }
    }
    cout << "Inverted-case string : " << str ;
    return 0;
}
```

上面的这个程序创建了一个指针 p，然后给它赋值为数组的首地址。在 for 循环的内部，我们把 p 当作数组来使用，通过索引来访问数组中的元素，这点在 C++中时完全有效的。语句 p[i]在功能上和*(p+i)是相同的。这个程序就更深一步地演示出了指针和数组的关系。

练习：

1. 能否通过指针来访问数组？
2. 是否可以对指针也进行索引，把它看做和数组一样了？
3. 只使用一个数组的名称，而没有使用索引，其结果值如何？

专家答疑：

问：指针和数组是否可以互换？

答：正如前面的几个程序那样，在大多数情况下指针和数组是强相关的，并且是可以互换的。例如，当一个指针指向一个数组的首地址的时候，我们就可以使用指针的算术运算或者类似于数组索引的方式来访问数组中的元素。然而，指针和数组并不是完全可以互换的。例如下面的程序片段：

```
int nums[10];
int i;
for ( i= 0; i< 10 ; i++)
```

```
{
    *nums =i;   //这样写是合法的。
    nums++; //  这样写是错误的，因为 nums 是不能被修改的。
}
```

这里，nums 是一个整型数的数组。正如注释所描述的那样，对 nums 使用*运算是完全可以接受的，然后修改 nums 的值却是非法的。因为 nums 是一个指向数组首地址的常量。因此，不能对其进行自增的运算。通常来说，使用数组的名称，而没有使用索引确实是产生一个指向数组首地址的指针，但是数组名的值不能被修改。

尽管，使用数组名产生的是一个指针常量，但是它是可以被用于到指针风格的表达式中的，只要不去修改它的值。例如，下面的代码给 num[3]赋值为 100 的语句是有效的：

```
*(num+3) = 100; //  这样写是合法的，因为没有修改 num 的值。
```

字符串常量

你可能对下面代码中所涉及到的 C++处理字符串常量的方式还不是很清楚：

```
cout << strlen ("Xadalu");
```

问题的答案在于：当编译器遇到一个字符串常量的时候，它就把这个字符串加入到程序的字符串表中，并生成一个指针指向它。因此，"Xadalu"就会产生一个指向该字符串的指针。下面的程序是完有效的，它打印出短语 Pointers and power to C++.:

```
#include <iostream>
using namespace std;
int main ()
{
    char *ptr;
    ptr = "Pointers add power to C++.\n";//p 被赋值为字符串常量的地址
    cout << ptr;
    return 0;
}
```

在这个程序中，构成整个字符串常量的字符会被存储在字符串表中，ptr 被赋值为指向该表中这个字符串的指针的值。

既然在使用字符串常量的时候，会自动地生成一个指向字符串的指针，你可能会想到使用这点来修改字符串表中的常量。这可不是什么好主意，因为很多的 C++编译器会对字符串表进行优化，一个字符串常量可能会在程序的多出被用到。因此，修改这个字符串会导致难以预料的后果！

项目 4-2 字符串倒置

我们曾经在前面提到过只有当两个指针指向同一个对象的时候，指针的比较才是有意义

的。然后我们又学习了指针和数组之间的关系，这样我们就可以用指针的比较来简化某些算法了。在下面的项目中，我们就将看到这样的一个例子。下面的这个程序完成对存储在数组中的字符串进行倒置的功能。它不是把一个字符串从尾部到首部复制到另外一个字符串中，而是在存储该字符串的数组上完成该字符串的倒置。这个程序使用了两个指针变量来完成这个功能。一个指针初始化为字符串的首地址，另外一个指针初始化为字符串的最后一个字符的地址。通过建立一个循环，交换这两个指针指向位置的字符来完成字符串的倒置。只要起始指针小于结束的指针，循环就一直进行。当起始指针的值等于或者大于结束指针的值的时候，就完成了对字符串的倒置，循环也就结束了。

步骤：

1. 创建一个名称为 StrRev.cpp 的文件

2. 在文件中增加下面的代码

```
/*
 工程 4-2
 原地对一个字符串进行倒置
*/
#include <iostream>
#include <cstring>
using namespace   std;
int main
{
    char str[] = "this is a test";
    char *start, *end;
    int len;
    char t;
```

我们要进行倒置的字符串保存在 str 中。我们将使用指针 start, end 来访问该字符串。

3. 增加下面的几行代码，完成原始字符串的显示，获取字符串的长度，并完成 start，end 两个指针的初始化。

```
    cout << "Original: " << str << "/n";
    len = strlen(str);
    start = str;
    end = &str[len-1];
```

注意，end 指针指向的是字符串中的最后一个字符，而不是字符串的结束标记。

4. 增加下面的代码完成字符串的倒置

```
while ( start < end )
{
```

```
    // swap chars
    t = *start;
    *start = *end;
    *end = t;
    //移动指针
    start++;
    end--;
}
```

上面代码的工作原理：只要 start 指针指向的内存位置小于 end 指针指向的位置，循环就一直继续。在循环内部，把 start 和 end 指针指向的位置的字符进行交换。start 指针是通过自增来改变其值的，end 指针是通过自减来改变其值的。当 end 指针大于或者等于 start 指针的值的时候，字符串的倒置就完成了。因为 start 和 end 是指向同一个数组的，所以它们之间的比较有意义的。

5. 下面是完整的程序

```
/*
 工程 4-2
 原地对一个字符串进行倒置
*/
#include <iostream>
#include <cstring>
using namespace   std;
int main( )
{
    char str[] = "this is a test";
    char *start, *end;
    int len;
    char t;
    cout << "Original: " << str << "\n";
    len = strlen(str);
    start = str;
    end = &str[len-1];
    while ( start < end )
    {
        // swap chars
        t = *start;
```

```
        *start = *end;
        *end = t;

        //移动指针
        start++;
        end--;
    }
    cout << "Reversed:" << str << "\n";
    return 0;
}
```

上面程序的输出如下：

```
Original: this is a test
Reversed:tset a si siht
```

指针数组

像其它类型的数据一样，我们也可以把指针用数组的方式组织起来。例如，下面的语句声明了 10 个整型指针的数组：

```
int *pi[10];
```

这里，pi 是 10 个整型指针的数组。把一个变量 var 的地址赋值给数组的第三个指针元素的代码如下：

```
int var；
pi[2]=&var;
```

请记住，pi 是一个指针数组，数组中可以保存的值都是整型值的地址，而不是整型值本身。可以用下面的语句来获取变量的值：

```
*pi[2]
```

和其他的数组一样，指针数组也可以进行初始化。最常用的一个对指针数组进行初始化的方法就是为其装载指向字符串的指针。下面就是一个使用二维的字符型指针数组来实现一个小字典的程序：

```
// 使用一个二维的指针数组来实现一个小字典
#include <iostream>
#include <cstring>
using namespace   std;
int main ()
{
    char * dictionary [][2] =
    {
```

```
        "pencil", "A writing instrument.",
        "keyboard", "An input device.",
        "rifle","A shoulde-fired firearm",
        "airplan", "A fixed-wing aircraft.",
        "network", "An interconnnected group of computers."
        "","";
    };
    char word[80];
    int i;
    cout << "Enter word:";
    cin >> word;
    for ( i = 0; *dictionary[i][0]; i++ )
    {
        if ( !strcmp ( dictionary[i][0], word ) )
        {
            cout << dictionary[i][1] << "\n";
            break;
        }
    }
    if ( !*dictionary[i][0] )
    {
        cout << word << " not found.\n";
    }
    return 0;
}
```

程序的输出可能如下：

```
Enter word:network
An interconnnected group of computers.
```

在创建数组 dictionay 的时候，我们用一组单词和对它们的解释来初始化这个数组。回忆前面讲过 C++把程序中所有的字符串常量都存储在字符串表中，数组中只需要保存指向这些字符串的指针即可。这个程序把 word 和字典中的字符串进行比较。如果相同，则显示出对其的解释。如果没有匹配到相同的单词，则打印错误信息。

请注意，字典最后是以两个空字符串结尾的。*运算获取指针指向位置的字符。如果这个字符是 null，那么这个表达式的值就是假值，循环也就结束了。相反，表达式的值就为真值，循环继续进行。

空指针的约定

在声明一个指针之后，还没有给它赋值之前，指针的值是任意的。如果在给指针赋值之前就使用指针，就很有可能导致程序崩溃。然而，我们没有办法来确保不使用未经初始化的指针。但是 C++程序员可以采用一种方法来帮助减少这种错误的发生。大家约定俗成的，如果一个指针的值为 null（0），那就认为这个指针不指向任何东西。因此，如果所有未使用的指针的值都是 null，并且我们在程序中避免使用 null 值的指针，那么就可以避免地使用未经初始化的指针的错误。这是一个我们都应该遵循的很好的实践。

任何类型的指针都声明的时候可以初始化为 null。例如，下面的代码片段初始化 p 为 null:

```
float * p = null; // p 现在是一个空的指针
```

我们可以使用 if 语句来检查空指针：

```
if ( p ) // p 为非空指针的时候，判断成立
if ( !p ) //p 为空指针的时候，判断成立。
```

练习：

1. 程序中使用到的字符串常量是存储在_____ 。

2. 下面的语句创建的是什么？

```
float * fpa[18];
```

3. 按照约定俗成的规矩，空指针意味着没有被使用的指针，对吗？

必备技能 4.12：多重指针

指向指针的指针就是多重指针。如图 4-2 所示，就普通的指针而言，指针的值是另外一个值的地址。而对指向指针的指针来说，第一个指针的取值是第二个指针的地址，第二个指针才指向变量的存储位置。

多重指针可以扩展到任何想要的多次指向。但是，通常情况下我们都是不会用到比指向指针的指针更复杂的情况了。过多次数的指针指向会导致程序难以理解并且很容易导致程序出错。

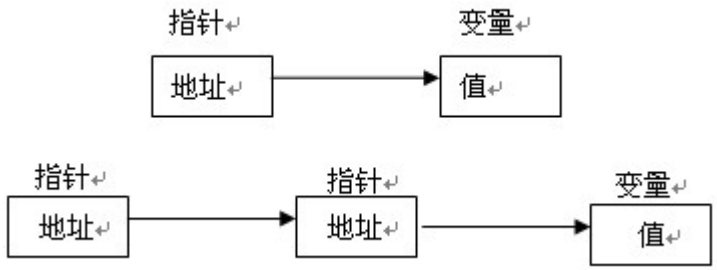


图 4-2 单次和多次的指针指向

指向指针的指针必须按照这样方式进行声明，就是需要在变量的名称前面再增加一个星号。例如，下面的代码告知编译器 balance 是一个指向 int 类型指针的指针：

```
int ** balance;
```

变量 balance 不是一个指向整型数类型的指针，而是一个指向 int 类型的指针的指针，理解这一点非常重要。当通过指向指针的指针来访问目标值的时候，就需要两次使用*运算符，正如下面的程序所示：

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

int main()
{
    int x, *p, **q;
    x = 10;
    p = &x;
    q = &p;
    cout << **q; //打印出 x 的值，通过 q 来访问 x 的值，这里需要两次使用*运算符
    return 0;
}
```

在上面的这段程序中，p 被声明为一个指向整型数的指针，q 被声明为一个指向整形指针的指针。cout 语句将在屏幕上打印出数字 10。

专家答疑

问：从前面的学习中可以看出指针的功能是非常强大的。然而指针的错误使用很容易给程序带来灾难性的后果。你有什么建议来避免指针的错误使用？

答：首先，我们必须确保指针变量在使用前是被初始化的。也就是说，我们要确保在使用指针之前，指针已经实际地指向了某个对象。其次，我们要确保指针指向对象的类型和指针的类型是一致的。再次，不要对空指针进行操作。回忆一下，空指针实际上是没有指向任何对象的指针。最后，不要仅仅为了编译通过而对指针进行强制类型的转换。通常，指针类型不匹配的错误意味着我们程序员错误地把一件事情当做了另外的事情。指针类型的强制转换只有在特殊的场合才会用到。

复习题

1. 写出如何声明一个可以容纳 31 个元素的名称为 hightemps 的类型为 short int 的数组？

2. 在 C++中，所有的数组索引的开始都是___。

3. 写出搜索含有 10 个整型数数组中重复数值的程序，并输出这些重复的数据。

4. 什么是 null-terminated 字符串？

5. 编写一个程序，提示用户输入两个字符串，然后忽略大小写比较两个字符串是否相等。

也就是说”ok”和”OK”是相等的。

6. 在使用 strcat()函数的时候，接收字符串的数组的大小必须是多少？

7. 对于多维数组，我们如何指定索引？

8. 写出如何用 5，66，88 来初始化一个名为 nums 的整型数组？

9. 未指定大小的数组会给我们带来什么好处？

10. 什么是指针？和指针相关的两个运算符是什么？

11. 指针是否可以像数组那么被索引？数组是否可以通过指针来访问？

12. 编写一个程序用来统计一个字符串中大写字母的个数，并输出结果。

13. 我们如何称呼一个指针指向另外一个指针的这种用法？

14. 在 C++中空指针有什么意义？

第五篇 函数简介

第五章 函数

本章我们将深入讨论函数。函数是 C++程序的基本构成单元，深刻的理解函数是我们成为一个成功的 C++程序员的基础。本章中，我们就要学习到如何创建函数。除此之外，我们还将学习函数的参数传递，函数的返回值，局部和全局变量，函数的原型以及递归。

函数的基础知识

一个函数就是含有一条或者多条 C++语句，用来完成一项特定任务的子程序。到目前为止，我们所编写的所有程序都至少用到了一个函数:main()。函数之所以被称为是 C++程序的基本构成单元是因为 C++程序都是一系列函数的集合。程序中的所有动作都是可以在函数中找到的。因此，一个函数就包含了我们通常认为的程序的可执行部分。尽管本书中所使用到的例子都是非常简单的，只有一个函数那就是 main()，大多数的程序都含有多个函数。实际上，大型的商用程序通常会含有成千的函数。

必备技能 5.1：函数的通用形式

所有的 C++函数都有如下的通用形式：

返回值类型 函数名称(参数列表){函数体};

其中，返回值类型表明了这个函数返回的数据的类型。它可以是任何有效的类型，但不能是数组。如果函数没有返回值，它的返回类型必须是 void 类型。函数名称指明了这个函数的名字。它可以是任何程序中目前还没有使用到的有效的标识符。参数列表是一系列由逗号间隔开的类型和标识符对。参数实际上是在调用函数的时候用来接收传入到函数中的值的变量。如果一个函数不需要参数，那么参数列表就为空。函数体由一对花括号括起来。函数体是由一组定义了函数功能的 C++语句构成。函数在遇到函数体右括号的时候终止，并返回到调用的地方。

必备技能 5.2：创建一个函数

创建函数是一个非常简单的过程。既然所有的函数都有着相同的形式，那么它们在结构上应该都和我们一直使用的 main()函数是类似的。下面让我们从一个简单的示例程序开始。这个程序中还有两个函数：main()和 myfunc()。在运行下面的程序之前，请仔细研究下面的程序，看看自己能否想出下面的程序会在屏幕上输出什么结果。

```
#include <iostream>
using namespace std;
void myfunc(); //声明函数的原型
int main()
{
    cout << " In main() \n" ;
    myfunc(); //调用函数 myfunc()
    cout << " Back in main()\n";
    return 0;
}
//下面是函数 myfunc()的定义
void myfunc()
{
    cout << " Inside myfunc()\n";
}
```

上面这个程序的工作方式是这样的：首先，程序从 main()函数开始，它执行第一个 cout 语句。接着，main()函数调用函数 myfunc()。注意上面的函数调用是这样完成的：在函数的名称后面紧跟一对括号。在上面的程序中对函数 myfunc()的调用是一个单一的语句，因此后面紧跟一个分号。接下来，函数 myfunc()执行它的第一个 cout 语句，然后在遇到函数体结束的右括号的时候返回到 main()函数中。程序的执行继续从对 myfunc()函数的调用下面的代码开始，那又是一个 cout 语句，在遇到 return 语句时，main()函数就终止了。程序的输出如下：

```
 In main()
 Inside myfunc()
 Back in main()
```

上面的函数 myfunc()被调用的方式以及函数返回的方式是一种对所有函数都适用的方式。通常来讲，调用一个函数的时候，只需要在函数名称后面跟上一对括号就可以了。当调用一个函数的时候，程序的执行就会转跳到这个函数中，然后一直执行到遇到函数体的右括号。当函数结束的时候，程序的执行就会返回到调用该函数的地方的下一条语句处。

必备技能 5.3：使用参数

函数还可以使用参数。传递给函数的参数叫做实参(arguments)。参数是一种把信息带入到函数中的一种方式。

当我们创建一个需要一个或者多个参数的函数的时候，也必须声明用于接收这些参数的变量。这些变量叫做函数的形参(parameters)。下面就是一个示例，其中定义了一个名称为box()的函数。它用来计算一个盒子的体积并输出结果。它需要三个参数。

```
void box(int length, int width, int height)
{
    cout << "volume of box is " << length * width * height << "\n";
}
```

这样以来，每次在调用函数 box()的时候，它就会把传入的三个参数 length,width,height 相乘，计算出体积。注意上面的示例中声明参数的方式。参数之间是用逗号相间隔的，并且参数被放置在函数名称后面的一对括号中。这种方式对所有使用参数的函数都是适用的。

这样，在调用 box() 函数的时候就需要指定三个参数。如下：

```
box(7,20,4);
box(50,3,2);
box(8,6,9);
```

其中位于大括号中的数值就是传递给函数 box()的参数。这些值会被复制到对应的参数中。所以在第一个调用函数 box()的语句中，7 会被复制到 length 中，20 会被复制到 width 中，4 会被复制到 height 中。在第二个调用函数 box()的语句中 50 被复制到变量 leight 中，3 被复制到变量 width 中，2 被复制到变量 height 中。在第三个调用函数 box()的语句中，8 被复制到变量 lenght 中，6 被复制到变量 width 中，9 被复制到变量 height 中。

下面的程序演示了 box ()函数调用的方法：

```
#include <iostream>
using namespace std;
void box (int length, int width, int height);   //声明函数 box()的原型
int main()
{
    box( 7,20,4 );
    box( 50,3,2 );
    box( 8,6,9 );
    return 0;
}
void box(int length, int width, int height)
{
```

```
    cout << "volume of box is " << length * width * height << "\n";
}
```

上面程序的输出如下：

```
volume of box is 560
volume of box is 300
volume of box is 432
```

请记住，术语实参(argument)指的是在调用函数的时候传入的值；用于接收这些值的变量叫做形参（parameter）。实际上，需要使用参数的函数被称为参数化的函数。

练习：

1. 当在程序中调用函数的时候，程序的执行顺序有什么变化？
2. 实参（argument）和形参（parameter）的区别是什么？
3. 如果一个函数需要使用参数，那么应该如何声明这个函数？

必备技能 5.4：使用 return 语句

在前面的示例中，函数在遇到函数体结束的右括号时才会返回到调用该函数的地方。对于大多数的函数来说，这种方式是可以接受的。但并不是所有的函数都是这样的。通常，我们需要精确地控制在什么时候函数以何种方式来返回到调用的地方。为此，我们需要使用 return 语句。

return 语句有两种形式：一种是返回一个值；另外一种则什么都不返回。我们将从什么都不返回的这种形式开始学些 return 语句。如果一个函数的返回值类型为 void 类型，则在该函数中可以使用不返回任何值的 return 语句。这种 return 语句的形式如下：

```
return;
```

当在函数中遇到 return 语句的时候，程序的执行就会立刻返回到调用该函数的地方。函数中 return 语句之后的所有代码都会被忽略掉。例如，研究一下下面的程序：

```
#include <iostream>
using namespace std;
void f();
int main()
{
    cout << "Befor call\n";
    f();
    cout << "After call\n";
    return 0;
}
```

```
//使用 return 语句的无返回值的函数
void f()
{
    cout << "Inside f()\n";
    return; //返回到调用该函数的地方
    cout << "This won't display\n";
}
```

程序的输出如下：

```
Befor call
Inside f()
After call
```

正如程序的输出那样，函数 f()在执行到 return 语句的时候就返回了，所以第二个 cout 语句是不会被执行的。

下面例子中的 return 相比来说更加实际一些。其中的 power()函数输出一个整型数的正数次幂。如果指数是负数，那么 return 语句就会终止该函数的执行，避免了企图进行计算。

```
#include <iostream>
using namespace std;
void power(int base, int exp);
int main()
{
    power(10,2);
    power(10,-2);
    return 0;
}
void power(int base, int exp)
{
    int i;
    if ( exp < 0 ) return ; //避免了企图计算指数为负数的幂
    i= 1;
    for (; exp; exp--) i = base * i;
   cout << "The answer is : " << i;
}
```

程序的输出如下：

```
The answer is : 100
```

当 exp 为负数的时候，函数 power()就直接返回了，其后面的代码就被忽略了。

一个函数中可以含有多个 return 语句。只要遇到其中的一个，函数就立刻返回。例如，下面的代码段是有效的：

```
void f()
{
    //...
    switch(c)
    {
        case 'a': return;
        case 'b': //.....
        case 'c': return;
    }
    if ( count < 100 ) return;
}
```

返回值

函数还可以给调用者返回一个值。因此返回值是一种函数把信息带出来的方式。带返回值的 return 语句的形式如下：

return 值；

其中的值就是要被返回的数值。这种形式的 return 语句只能用于那些返回类型不是 void 的函数。

一个需要返回值的函数必须指明返回值的类型。返回类型必须和 return 语句中返回值的类型是相兼容的。如果不兼容，编译的时候就会出现错误。函数可以返回任意有效的 C++ 的数据类型，但是函数不能返回数组。

为了演示函数返回值的过程，我们按照下面的方式重写函数 box()。在这个版本中，函数 box()返回计算得到的体积。注意，其中把函数放置到一个赋值语句的右边表示把函数的返回值赋值给左侧的变量。

```
#include <iostream>
using namespace std;
int box(int length, int width, int height); //返回计算得到的体积
int main()
{
    int answer;
    answer = box(10,11,3);
    cout << "The volumn is " << answer;
```

```
    return 0;
}
//函数返回一个值
int box(int length, int width, int height)
{
    return length * width * height;
}
```

程序的输出如下：

```
The volumn is 330
```

在这个实例中，函数 box()中使用了 return 语句返回 length * width * height，然后这个返回值被赋值给 answer。也就是说，在调用的时候，return 语句返回的值就变成了 box()的值。

既然 box()函数现在是返回一个值的，那么它的返回类型就不再是 void 的了。（请记住，void 只有在函数不返回任何值的时候才使用。）而是被声明成返回一个类型为 int 的值。请注意，放置在函数名称之前的返回类型在函数的原型和函数的定义中都是要有的。

当然，int 类型并不是唯一的函数可以返回的类型。正如前面讲到的那样，函数可以返回除了数组之外的任意类型。例如，下面的程序中，我们重写了 box() 函数，它的参数为 double 类型，返回类型也是 double 类型。

```
#include <iostream>
using namespace std;
//使用 double 数据类型
double box(double length, double width, double height);
int main()
{
    double answer;
    answer = box(10.1,11.2,3.3 );
    cout << "The volume is " << answer;
    return 0;
}
//这个版本中，box 函数使用 double 数据类型
double box(double length, double width, double height )
{
    return length * width * height;
}
```

程序的输出如下：

```
The volume is 373.296
```

还有一点：如果在一个非空的返回值的函数中，由于遇到了函数体结束的右括号而结束，则返回值是不确定的。根据 C++语言的语法，无返回值的函数并不一定需要 return 语句，函数在遇到函数体结束的右括号的时候就自动返回。然而，如果一个函数如果是有返回值的，那么这种遇到函数体结束的右括号而返回的值则是垃圾数据。当然，好的编程实践告诉我们，在任何非空返回值的函数中，我们都要使用 return 语句来明确返回值。

必备技能 5.5：在表达式中使用函数

在前面的示例中，box()函数的返回值被赋值给了一个变量，然后通过使用 cout 语句输出了变量的值。这样做虽然是正确的，但是我们可以在 cout 语句中直接使用函数的返回值来重写这个程序，这样程序的效率会更高。例如，前面程序中的 main()函数可以按照下面的方式被重写：

```
int main()
{
    //直接使用 box()函数的返回值
    cout << "The volume is " << box(10.1, 11.2, 3.3);

    return 0;
}
```

在执行上面的 cout 语句的时候，box()函数会被自动调用并获取到它的返回值，然后输出。没有必要先把函数的值赋值给一个变量。

通常来说，有返回值的函数可以在任何表达式中使用。当计算表达式值的时候，函数会被自动调用，以便获取函数的值。例如，下面的程序把三个盒子的体积相累加，然后输出平均体积。

```
#include <iostream>
using namespace std;
//使用 double 类型的数据
double box(double length, double width, double height);
int main()
{
    double sum;
    sum = box(10.1,11.2,3.3) + box(5.5, 6.6, 7.7) + box(4.0, 5.0, 8.0);

    cout << "The sum of the volumes is " << sum << "\n";
    cout << "The average volume is " << sum / 3.0 << "\n";
```

```
    return 0;
}
double box(double length, double width, double height)
{
    return length * width * height;
}
```

程序的输出结果如下：

```
The sum of the volumes is 812.806
The average volume is 270.935
```

练习：

1. 写出 return 语句的两种形式。
2. void 类型的函数是否可以返回数值？
3. 函数是否可以作为表达式中的一部分？

作用域

到目前为止，我们一直都在使用变量，但是却没有讨论过在什么地方可以声明变量，如何确定变量的生存期，以及程序的哪些部分可以访问这些变量。这些属性是由 C++中的作用域规则来确定的。

通常情况下，语言中的作用域规则确定了对象的可见性以及对象的生存周期。

尽管 C++中定义了很好的作用域系统，但是基础的只有两个：局部的和全局的。在这两个范围中，我们都可以声明变量。在本小节中，我们会看到在局部范围中声明的变量和在全局范围中声明的变量有什么区别，它们之间又有着什么联系？

必备技能 5.6：局部作用域

局部作用域是由块创建的。（回忆前面讲过的块是由一对花括号括起来的）。因此，每次我们创建代码块的时候也就创建了一个新的作用域。我们可以在任何代码块中声明变量。在代码块中声明的变量我们称之为局部变量。

局部变量只能被和它在位于同一个代码块中的代码所访问。换句话说，局部变量在其代码块之外是不会被感知的。因此，代码块之外的语句是不能访问代码块之内的对象的。从本质上来说，当我们声明了一个局部变量的时候，我们就是将这个变量本地化了，避免了对它的未经授权的访问或者修改。实际上，作用域规则提供了封装的基础。

关于局部变量有一个很重要的点需要明确：它们只有当其所在的代码段的代码被执行的时候才存在。局部变量在其所在的代码块中被声明的时候便开始存在，它们在程序的执行离开所在代码块的时候被销毁。由于局部变量在离开其所在代码块的时候被销毁，它的值也就没有意义了。最常用的用于声明变量的代码块就是函数。每个函数都定义了一个代码块。这个代码块以函数体的左花括号开始，以函数体的右花括号结束。函数的代码和数据是函数私

有的。除了调用该函数之外，别的函数中的任何语句都是不能访问它们的。（goto 语句也是不能从一个函数中转跳到另外一个函数的中间的。）

函数的函数体对程序的其它部分来说是隐藏的，它不能影响程序的其它部分，也不受程序其它部分的影响。因此一个函数的功能是完全和别的函数相独立的。换句话说，一个函数中的语句和数据是不会影响另外一个函数中的数据或者代码的，因为两个函数的作用域是不相同的。因为每个函数都定义了自己的一个作用域，所以在一个函数中声明的变量对另外函数的变量时没有任何影响的，即使变量的名字是相同的。

例如下面的程序：

view plain

```
#include <iostream>
using namespace std;
void f1();
int main()
{
    int val = 10; // 变量 val 的作用域仅限于函数 main
    cout << " Val in main(): " << val << '\n';
    f1();
    cout << " Val in main(): " << val << '\n';
    return 0;
}
void f1()
{
    int val = 88;
    cout << " val in f1() " << val << "\n";
}
```

程序的输出如下：

val in main(): 10

val in f1(): 88

val in main(): 10

上面的程序中名称为 val 的变量被声明了两次，一次是在 main()函数中，一次是在 f1()函数中。main()函数中的 val 和 f1()函数中的 val 没有任何的关系。这是因为每个 val 只能在自己所在的函数中被感知。正如程序的输出那样，即使在 f1()函数中声明的变量 val 被赋值为 88，main()函数的 val 的值仍为 10。

由于局部变量是在进入其所在的代码块的时候被创建，退出其所在代码块的时候被销毁，所以局部变量在函数被多次调用之间是不会保留其值的。这一点对于理解函数调用是非常重要的。当调用一个函数的时候，其中的局部变量就会被创建，当函数返回的时候，这些局部

变量就会被销毁。这就意味着，在函数多次被调用之间，这些局部变量的值是不会被保留的。

如果局部变量在声明的时候进行了初始化，那么这个变量在每次进入其所在代码块的时候都会被初始化。如下程序所示：

view plain

```
/*
    局部变量的初始化在进入其所在代码块的时候每次都会进行
*/
#include <iostream>
using namespace std;
void f();
int main()
{
    for( int i = 0; i < 3; i++) f();
    return 0 ;
}
//num 变量在每次调用函数 f()的时候都会被初始化
void f()
{
    int num = 99;
    cout << num << "/n";
    num++;// 这句代码并不会影响下一次函数 f()被调用是 num 的值。
}
```

程序的输出如下：

99

99

99

没有被初始化的局部变量的值是未知的，直到这个变量被赋值。

局部变量的声明可以在任何代码块中进行

通常，我们会把函数中所需要的变量在函数一开始的时候就进行声明。这样做可以让代码的阅读者很容易地看清楚都需要哪些变量。然而，函数一开始并不是可以声明变量的唯一地方。局部变量可以在任何代码块的任何地方进行声明。这就意味着，只有进入了相应的代码块，变量才会被创建，当退出相应代码块的时候，变量就会被销毁。更进一步来说，变量所在代码块之外的任何代码都是不能访问这个变量的。为了理解这点，我们可以看看下面的程序：

view plain

```
1. //代码块中的局部变量
2. #include <iostream>
3. using namespace std;
4. int main()
5. {
6.     int x = 19 ; // x 可以被 main() 函数中的所有代码感知
7.     if ( x == 19 )
8.     {
9.         int y = 20; // 变量 y 的作用域仅限于 if 的代码块
10.        cout << " x + y is  " << x + y << "\n";
11.    }
12.    // y = 100; 错误！在这里是感知不到 y 的
13.    return 0;
14. }
```

上面的程序中，变量 x 是在 main()函数开始的时候就声明的，所以 main()函数中其后的所有代码都是可以访问这个变量的。在 if 的代码块中声明了变量 y。由于代码块就定义了作用域，因此 y 只是对与其在同一代码块中的代码所感知。这也是在 if 代码块之外的语句 y = 100; 被注释掉的原因。如果我们去掉该语句前面的注释符，编译时就会报错，就是因为 y 在 if 代码块之外是不被感知的。在 if 的代码块中，x 是可以被感知的，这是因为 if 代码块也是位于 main()函数所定义的代码块中的。

尽管局部变量通常都是在代码块开始的时候进行声明，但这不是必须的。局部变量可以在代码块中的任何地方进行声明，只要是在它被使用之前。例如，下面的程序时完全正确的：

view plain

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     cout << "Enter a number ";
6.     int a; // 这里声明变量是完全可以的
7.     cin >> a;
8.     cout << "Enter a second number ";
9.     int b; // 这里声明变量是完全可以的
10.    cin >> b;
11.    cout << "Product: " << a * b << '\n';
12.
13.    return 0;
14. }
```

在这个示例中，变量 a,b 并没有在函数一开始就声明了，而是在需要的时候才进行声明。实际上，大部分程序员都是在函数一开始的地方进行变量的声明。这只是代码风格的问题。

名称隐藏

当在一个内部代码块中声明了一个与外部代码块中同名的变量的时候，内部代码块中的这个变量就使得外部的这个变量被隐藏了。如下面的程序所示：

view plain

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    int j;
    i = 10;
    j = 100;
    if ( j > 0 )
    {
        // 在 if 的代码块中，这个 i 使得 main()函数一开始声明的 i 被隐藏了。
        //也就是说 if 代码块中出现的 i 都是这里的 i 而不是 main()函数一开始的声明的那个 i。
        int i;
        i = j / 2;
        cout << " inner i :" << i << "\n";
    }
    cout << " Outer i: " << i << '\n';
    return 0;
}
```

上面程序的输出如下：

inner i :50

Outer i: 10

上面程序中 if 代码块中声明的变量 i 就使得外部代码块中的 i 被隐藏了。对内部代码块中 i 的修改不会影响外部代码块中的 i。跟进一步来说，在 if 的代码块之外，其内部的 i 就不可见了，而外部的 i 又恢复可见了。

函数的形参

函数形参的作用域就局限于函数体中。因此，它们对于函数来说是局部变量。形参除了是用于接收传入实参的值之外，它们的用法完全和局部变量一样。

专家答疑：

问：关键字 auto 是用来干什么的？我听说是用来声明局部变量的，是这样吗？

答：C++的关键字中是有 auto 这个关键字的。它可以被用来声明局部变量。然而，既然所有的局部变量缺省情况下都是 auto 的，所以这个关键字实际上很少使用。本书中的所有示例程序也是不用这个关键字的。然而，如果你决定要使用这个关键字，只要把它放置在声明变量时类型的前面就可以了,如下：

auto char ch;

重复一次，auto 关键字是可选的，本书的其它地方是不会使用它的。

必备技能 5.7：全局作用域

既然局部变量只有在它们所在的函数中才是可见的，那么你也许会问：如何创建一个可以供多个函数共同使用的变量了？答案就是把变量声明在全局作用域中。全局作用域就是一个声明变量的区域，它的范围是覆盖所有函数的局部作用域的。在全局作用域中声明一个变量就创建了一个全局变量。

全局变量在程序的任何部分都是可见的。它可以被程序的任何代码使用，它们的值在程序的整个生命周期都是保留的。因此，它的作用域就是整个程序。我们可以在函数的外面来声明全局变量。由于它们是全局的，任何表达式都可以使用它们，而不管这些表达式是位于哪个函数中。

下面的程序演示了全局变量的用法。变量 count 被声明在了所有函数的外部，也是在 main()函数的外部。实际上它的声明可以被放置在任何地方，只要不是放置在某个函数的内部即可。请记住：由于我们必须在使用变量之前就对变量进行声明，所以我们最好是把全局变量的声明放置在程序的最顶部。

view plain

```
1.  //使用全局变量
2.  #include <iostream>
3.  using namespace std;
4.  void func1();
5.  void func2();
6.  int count; //全局变量
7.  int main()
8.  {
9.      int i; // 局部变量
10.     for( i = 0; i < 10 ; i++)
```

```
11.     {
12.         count = i * 2;
13.         func1();
14.     }
15.     return 0;
16. }
17. void func1()
18. {
19.     cout << "count : " << count; //全局变量 count
20.     cout << '\n';
21.     func2();
22. }
23. void func2()
24. {
25.     int count; // 局部变量
26.     for ( count = 0; count < 3; count++ ) cout << '.';
27. }
```

程序的输出如下：

count : 0

...count : 2

...count : 4

...count : 6

...count : 8

...count : 10

...count : 12

...count : 14

...count : 16

...count : 18

...

仔细研究上面的代码，显然，在函数 main()和 func1()中使用的 count 变量都是全局变量。然而在函数 func2()中则使用的是局部变量 count，而不是全局变量。这一点非常重要：如果全局变量和局部变量有相同的变量名称，则在声明了局部变量的函数中对这个相同变量名称的使用指的都是该局部变量，而不是全局变量；对这个局部变量的引用也不会影响全局变量。因此，同名的局部变量使得全局变量被隐藏了。

局变量的初始化是在程序开始的时候进行的。如果全局变量在声明的时候有明确指定初始化值，则全局变量的初值就是这个值；否则，它的初值就是 0.

全局变量是被存储在代码区域之外的一个固定区域。当我们需要多个函数使用同一个数据的时候，或者当我们需要一个变量在程序的整个运行阶段都保有其值的时候，全局变量就

显得很有用了。但是，基于下面的三个原因，我们应该尽量避免使用不必要的全局变量：

1. 全局变量在程序运行的整个阶段都是占有内存空间的，而不仅是在需要的时候。

2. 在使用局部变量就可以满足要求的地方使用全局变量会降低函数的独立性，因为这样做使得函数依赖于自身之外的东西了。

3. 使用大量的全局变量会导致程序出现未明的的错误和预料之外的副作用。在开发大型程序的过程中，一个主要的问题就是在程序的其它地方对变量的错误修改。如果我们在程序中使用了过多的全局变量，这样的事情就有可能发生。

练习：

1. 局部变量和全局变量的主要区别是什么？

2. 局部变量是否可以在代码块中的任何地方进行声明？

3. 局部变量是否会在声明它的函数被多次调用之间保有其值？

必备技能 5.8：给函数传入指针和数组

在前面的示例程序中，我们传入到函数中的参数都是简单的数据，比如整型数，或者浮点数之类的。实际上，有时候我们需要把指针或者数组作为参数传入到函数中的。尽管给函数传入指针或者数组的方式是很简单的，但是其中有些特殊的事项需要注意。

传入指针

为了给函数传入指针作为其参数，我们必须把函数的参数类型声明为指针的类型。下面就是一个示例：

view plain

```
1. //给函数传入指针作为参数
2. #include <iostream>
3. using namespace std;
4. void f( int *j); //函数原型中需要指针作为参数
5. int main()
6. {
7.     int i;
8.     int *p;
9.     p = &i;
10.    f(p);
11.    cout << i; //i 的值现在是了
12.    return 0;
13. }
14. //f()函数是接收一个指针作为参数
15. void f(int *j)
16. {
17.    *j = 100;
18. }
```

仔细研究上面的程序。我们可以看到，函数 f()接收一个参数：一个整型数的指针。在 main()函数中，p 被赋值为变量 i 的地址。接下来,调用函数 f()的时候传入参数 p。当函数 f()中用形参 j 来接收传入的 p 的值后，j 也就指向了 main()函数中的变量 i。因此，赋值语句

*j = 100;

就会使得 i 的值变为 100。可见，函数 f()会把 100 赋值给调用时传入的任何地址的变量。

在上面的程序中，实际上我们没有必使用指针变量 p。我们在调用函数 f()的时候只需要在变量 i 的前面加上取地址符号&，传入到函数中即可。如下所示：

view plain

```
1.  //给函数传入指针作为参数
2.  #include <iostream>
3.  using namespace std;
4.  void f( int *j); //函数原型中需要指针作为参数
5.  int main()
6.  {
7.      int i;
8.      f(&i);
9.      cout << i; //i 的值现在是了
10.     return 0;
11. }
12. //f()函数是接收一个指针作为参数
13. void f(int *j)
14. {
15.     *j = 100;
16. }
```

关于给函数传入指针有很重要的一点需要理解：在函数中对指针参数的操作就是对指针指向变量的操作。因此，函数可以改变指针参数指向对象的值。

传入数组

当我们给函数传入数组的时候，我们实际上传入的是数组中第一个元素的地址，而不是整个数组的一个副本。（回忆我们在前面说过，不带索引的数组名称就是一个指向数组中第一个元素的指针。）这就意味着函数参数的声明必须是可以与之相兼容的类型。有三种方式来声明一个函数需要接收一个数组指针。首先，函数的参数可以被声明为与调用时传入的数组类型和大小一致的数组。如下：

view plain

```
1.  #include <iostream>
2.  using namespace std;
3.  void display ( int num[10] );
4.  int main()
5.  {
```

```
6.      int t[10], i;
7.      for ( i = 0; i < 10; i++ ) t[i] = i;
8.      display (t); //给函数传入一个数组
9.      return 0;
10. }
11. //打印数组中的数字
12. //参数声明为一个指定大小的数组
13. void display (int num[10] )
14. {
15.     int i;
16.
17.     for ( i = 0; i < 10; i++) cout << num[i] << ' ';
18. }
```

在上面的程序中，尽管参数 num 被声明为一个含有 10 个元素的整型数数组，C++编译器会自动地把它转化为一个整型指针。这样做是有必要的，因为函数的参数实际上是不能接收整个数组的。既然，传入的只是一个指针，那就得有个指针参数来接收它。

第二种方式就是把函数的参数声明为不指定大小的数组，如下：

view plain

```
1.  void display (int num[] )
2.  {
3.      int i;
4.
5.      for ( i = 0; i < 10; i++) cout << num[i] << ' ';
6.  }
```

在这里，num 被声明为一个没有指定大小的数组。由于在 C++中是不进行数组边界检查的，所以数组的大小就和参数没有什么关系（但不是和整个程序没有关系）。C++编译器也同样会把这种声明方式转化成整型指针。

最后一种方式就是把参数声明为指针。这也是在专业的 C++程序中最常用的一种方式。如下：

view plain

```
1.  void display (int *num)
2.  {
3.      int i;
4.
5.      for ( i = 0; i < 10; i++) cout << num[i] << ' ';
6.  }
```

之所以可以这样做的原因就是任何指针都可以通过[]来进行索引，就像数组一样。上述的三种方式都会产生相同的结果，那就是指针。

当我们给一个函数传入数组的时候，我们实际上传入的是数组的地址，记住这一点是非常重要的。这就意味着函数中对形参的操作或者修改实际上都是对调用时传入数组的操作和修改。例如，在下面的程序中，函数 cube()把数组中的数值转换为其值的立方，调用 cube()函数的时候传入数组地址作为第一个参数，数组的大小最为第二个参数。

view plain

```
1. //调用函数来修改数组的元素值
2. #include <iostream>
3. using namespace std;
4. void cube( int *n, int num);
5. int main()
6. {
7.     int i, nums[10];
8.     for( i = 0; i < 10; i++)
9.     {
10.        nums[i] = i+1;
11.    }
12.    cout << "Original contents: ";
13.    for ( i = 0; i < 10; i++)
14.    {
15.        cout << nums[i] << ' ';
16.    }
17.    cout << '/n';
18.    cube(nums, 10);
19.    cout << "Altered contents: ";
20.    for ( i = 0; i < 10; i++)
21.    {
22.        cout << nums[i] << ' ';
23.    }
24.
25.    return 0;
26. }
27. void cube(int *n, int num)
28. {
29.    while(num)
30.    {
31.        *n = *n * *n * *n;
32.        num--;
33.        n++;
34.    }
35. }
```

程序的输出如下：

Original contents: 1 2 3 4 5 6 7 8 9 10

Altered contents: 1 8 27 64 125 216 343 512 729 1000

正如我们看到的那样，在调用了函数 cube()以后，main()函数中数组 nums 的元素值都变成了原来值的立方。也就是说，数组 nums 中元素的数值都被 cube()函数中的语句修改了，这就是因为传入的是指针。

传入字符串

由于字符串就是一个使用了 0 作为结尾的字符数组，所以当我们传入字符串到函数中的时候，实际传入的是数组的首地址，也就是传入的是类型为 char *的指针。例如下面的程序中定义了函数 strInvertCase()，用来对字符串中字母的大小写进行转换。

view plain

```
1. //给函数传入字符串
2. #include <iostream>
3. #include <cstring>
4. #include <cctype>
5. using namespace std;
6. void strInvertCase(char *str);
7. int main()
8. {
9.     char str[80];
10.    strcpy(str,"This Is A Test");
11.    strInvertCase(str);
12.    cout << str; //输出转换后的结果字符串
13.    return 0;
14.}
15.//转换字符串中字符的大小写
16.void strInvertCase(char *str)
17.{
18.    while(*str)
19.    {
20.        //进行大小写的转换
21.        if ( isupper(*str) ) *str = tolower(*str);
22.        else if ( islower(*str) ) *str = toupper(*str);
23.        str++;//指针移动的下一个字符处。
24.    }
25.}
```

程序的输出如下：

tHIS iS a tEST

练习：

1. 如何声明一个无返回值的函数，它的名称为 count，它有一个整型指针参数。
2. 当给函数传入的是一个指针的时候，函数是否可以修改指针指向对象的内容？
3. 是否可以把数组传入到函数中？请解释为什么？

必备技能 5.9：返回指针

函数还可以返回指针。函数返回指针和函数返回其它类型的数据是一样的，没有什么特殊的地方。但是由于指针是 C++中最容易让人迷惑的地方之一，所以在这里，我们有必要对函数返回指针进行讨论。

返回指针的时候，函数的返回类型也必须声明为指针。例如，下面的语句就声明了函数 f()返回整型数指针：

int *f();

如果一个函数的返回值是指针类型，那么在函数中也必须返回一个指针。（正如其它返回值类型的函数一样，返回值必须和返回类型是兼容的。）

下面的程序演示了函数返回指针。函数 get_subStr()在一个字符串中查找子串，并返一个指向第一个匹配字串的指针。如果没有匹配的字串，则返回空指针。例如，如果字符串为“I like C++”，而查找的字串是”like”，那么函数返回的就是指向”like”的指针。

view plain

```
1.  //函数返回指针
2.  #include <iostream>
3.  using namespace std;
4.  char *get_subStr(char *sub, char *str);
5.  int main()
6.  {
7.      char *subStr;
8.      subStr = get_subStr("three","one tow three four");
9.      cout << "substring found: " << subStr;
10.     return 0;
11. }
12. //返回指向字串的指针或者空指针
13. char *get_subStr( char *sub, char *str)
14. {
15.     int t;
16.     char *p, *p2,*start;
17.     for ( t = 0; str[t]; t++)
18.     {
19.         p = &str[t];
```

```
20.         start = p;
21.         p2 = sub;
22.         while(*p2 && *p2==*p)
23.         {
24.             p++;
25.             p2++;
26.         }
27.         //如果找到了匹配的字串
28.         if ( !*p2)
29.         {
30.             return start;
31.         }
32.     }
33.
34.     return 0;//没有找到字串
35. }
```

上面程序的输出如下:

substring found: three four

main()函数

我们都知道 main()函数是一个非常特殊的函数，因为它是我们程序执行时第一个被调用的函数。它标记着我们程序的开始。不像那些程序总是从代码的最顶部开始执行的编程语言那样，C++程序的开始都是从调用 main()函数开始，而不管 main()函数是在代码的什么位置。(然而把 main()函数最为程序中的第一个函数是很常见的，这样可以方便地找到程序的入口。)

一个程序中只能有一个 main()函数。如果一个程序中还有多个 main()函数，则程序将无法知道应该应该从哪个 main()函数开始执行。实际上，大部分的编译器都是能检查并报告这种错误的。正如前面所提到的那样，由于 main()函数是 C++中预先定义好的，所以我们不必声明其原型。

必备技能：5.10：通过命令行给 main()函数传递参数

有时候我们需要在运行一个程序的时候就为其传入一些信息。通常，这是通过命令行给 main()函数传递参数来完成的。命令行参数就是在系统命令行上紧跟在程序名称后面的信息。（windows 上，Run 命令同样使用的是命令行参数）。例如，我们可以采用下面的命令来从命令行编译一个 C++程序：

cl prog-name

其中，prog-name 就是我们想要编译程序的名称。程序的名字被作为命令行参数传递给了 C++编译器。

C++中定义了 main()函数有两个内置的，也是可选的参数。它们是：argc 和 argv。它们是用来接收命令行参数的。它们也是 C++中为 main()函数定义的唯一一种参数。然而，由于操作系统的不同，main()函数还有可能有别的参数，这点需要查阅编译器的相关文档。下面让我们仔细研究一下 argc 和 argv。

注意：从技术上来讲，命令行参数的名称可以是任意的，我们可以使用任何自己喜欢的名称。然而，多年来，人们一直都是用 argc 和 argv 这两个名称，建议我们最好也都是用这两个名称，这样代码的阅读者就能方便地确认到它们是用来接收命令行参数的。

argc 变量是一个整型数，它用来存放命令行参数的数量。它的取值至少是 1，这是因为程序的名称就是作为第一个参数的。

argv 参数是一个指向字符指针数组的指针。argv 数组中的每个指针都指向一个命令行的字符串形式的参数。指针 argv[0]指向程序的名称；argv[1]指向第一个参数，argv[2]指向第二个参数；以此类推。所有的参数都是以字符串的形式传递到 main()函数中的，所以数值的参数必须由我们自己的程序转换成内部所需的形式。

正确地声明 argv 参数也是很重要的。最常用的形式如下：

char *argv[];

我们可以通过对 argv 进行索引而访问到每个参数。下面的程序演示了如何使用命令行参数。程序输出了所有命令行参数。

view plain

```
1. //打印命令行参数
2. #include <iostream>
3. using namespace std;
4. int main(int argc, char *argv[] )
5. {
6.     for ( int i = 1; i < argc; i++ )
7.     {
8.         cout << argv[i] << "\n";
9.     }
10.    return 0;
11. }
```

我们假设上面的程序名称为 ComLine，则从命令行执行它的情况会是如下：

C:/>ComLine one two three

one

two

three

C++并没有规定命令行参数的含义，因为这个是随着操作系统的不同而不同的。然而，最普遍的一个传统是这样的：每个命令行参数必须有空格或者制表符间隔。常用的逗号，分

号等都不是有效的参数分隔符。如下：

one,two,and three

是由四个字符串组成的，但是作为命令行参数的时候却是两个字符串。这是因为逗号不是有效的间隔符。

如果我们确实需要传递还有空格的字符串参数，那么我们必须把这个字符串用引号引起来。如下，下面的方式将被认为是一个命令行参数：

“this is one argument”

请记住，这里提供的示例有着很广泛的使用范围。但是并不一定全部都适用于你所使用的系统。

通常，我们使用 argc 和 argv 来为程序获取初始选项或者初始值，例如一个文件名称。在 C++中，命令行参数的数量是可以和你操作系统支持的数量一样多的。使用命令行参数会使得我们的程序显得非常专业，也方便在批处理程序中使用我们的程序。

通过命令行参数传入数值型的参数

当我们通过命令行参数把数值型数据传入到程序中的时候，数值将会被以字符串的形式接收。我们的程序需要使用 C++标准库函数中的函数来把这些字符串转换为二进制的，内部格式的数据。下面给出了处理这种情况时最常用的三个函数：

atof	把字符串转换为一个 double 类型的数据，返回其结果
atol	把字符串转换为一个 long int 类型的数据，返回其结果
atoi	把字符串转换为一个 int 类型的数据，返回其结果

上面的每个函数都是接收一个含有数值的字符串作为参数，它们使用的头文件是<cstdlib>。

下面的程序演示了如何把数值型的命令行参数转换为响应的二进制数值。程序计算命令行参数中跟在程序名称后面的两个数值的和。其中使用到了函数 atof()来把数值参数转换为相应的内部表示方式。

view plain

```
1.  //这个程序把命令行的两个数值型参数相加，并输出结果
2.  #include <iostream>
3.  #include <cstdlib>
4.  using namespace std;
5.  int main( int argc, char *argv[] )
6.  {
7.      double a,b;
8.      if (argc!=3)
9.      {
10.         cout << "Usage: add num num\n";
11.         return 1;
12.     }
13.     a = atof(argv[1]);
```

```
14.     b = atof(argv[2]);
15.
16.     cout << a+b;
17.     return 0;
18. }
```

假设程序的名称为 add，那么使用命令行来计算两个数的和如下：

C:/>add 100.2 231

必备技能 5.11：函数原型

在本章开始的时候我们对函数的原型进行了简单的讨论。现在我们要对函数的原型进行更深入的讨论。在 C++中，函数在使用之前是必须声明的。通常，这个是通过函数原型来完成的。函数原型中说明了函数的三部分内容：

1. 函数的返回类型
2. 函数参数的类型
3. 函数参数的数量

函数原型使得编译器可以进行以下三个重要的操作：

1. 函数原型告诉编译器在调用函数的时候应该生成什么样子的代码。编译器必须针对不同的返回类型进行不同的处理。
2. 函数原型使得 C++能够报告任何在调用函数时实参和形参之间的非法的类型转换。
3. 函数的原型使得编译器能够检测出调用函数时传入实参的数量和函数原型需要的参数数量的差别。

函数原型的通用形式如下。除了不需要函数体外，它和函数的定义是一样的。

类型 函数名称(类型 1 参数 1,类型 2 参数 2,... ...,类型 N 参数 N);

在函数原型中，参数名称是可选项。然而，明确地写上参数名称确实可以使得编译器在出现错误的时候通过参数名称来确定任何的类型不匹配，因此我们最好还是写上参数的名称。

为了能更好地理解函数原型的作用，我们可以看看下面的程序。如果我们编译一下下面的程序，就会看到编译错误。这是因为程序中函数 sqr_it()需要的是一个整型指针，而程序在调用这个函数的时候传入的是一个整型数。（不存在从整型数到整形指针的自动转换。）

```
/*
    该程序中使用了函数原型来强制进行强类型检查
*/
void sqr_it(int *i);
int main()
{
    int x;
```

```
    x= 10;
    sqr_it(x); //错误，参数类型不匹配。
    return 0;
}
void sqr_it(int *i)
{
    *i = *i * *i;
}
```

如果函数的定义在函数的使用之前，那么函数的定义也可以起到函数原型的作用。例如，下面的程序是完全合法的：

```
//如果函数的定义在函数的使用之前，那么定义也可以起到函数原型的作用。
#include <iostream>
using namespace std;
//用来判断一个数字是不是偶数
bool isEven(int num)
{
    if ( !(num%2) ) return true; //偶数
    return false;
}
int main()
{
    if(isEven(4)) cout << "4 is even\n";
    if(isEven(3)) cout << "this won't display";
    return 0;
}
```

在这里，由于函数 isEven()的定义是在 main()函数之前，所以函数的定义就可以起到函数原型的作用，因此也不需要单独的函数原型。

通常，和确保函数定义都是在函数调用之前相比，声明一下程序中用到的函数的原型显得更简单。特别针对大型程序来说，有时候是很难确定函数之间调用关系。更进一步来说，还存在两个函数相互调用的情况，这时我们必须使用函数的原型。

包含函数原型的头文件

在本书的前面，我们学习过了标准的 C++头文件。我们知道这些头文件中含有程序所需要的信息。这种解释只能说是部分是正确的，因为这种解释并不全面。C++的头文件中含有标准库中的函数原型。（其中还含有这些函数需要用到的不同的值和定义。）和我们自己写的函数一样，标准库函数在使用之前也必须声明其原型。我们可以查阅编译器的库函数文档

来明确使用库函数是所需要的头文件。在库函数文档中除了有对函数的描述之外，我们还可以看到使用这个函数是必须包含的头文件。

练习：

1. 什么是函数的原型？函数的原型有什么作用？
2. 除了 main()函数之外，其它所有的函数都必须声明其原型吗？
3. 当我们使用标准库函数的时候，为什么必须包含其头文件？

必备技能 5.12：递归

本章中我们需要最后讨论的就是递归。递归有时也被称为循环定义。递归就是用自身来定义自己。在编程中，递归就是函数自己调用自己。自己调用自己的函数我们就称之为递归函数。

典型的递归函数就是 factr()， 它用来计算一个整数的阶乘。N 的阶乘就是从 1 到 N 的所有数的乘积。例如，3 的阶乘就是 1×2×3，或者说是 6。下面的程序用到了 factr()函数的递归形式和循环形式。

/

view plain

```
1. /该程序演示了递归的用法
2. #include <iostream>
3. using namespace std;
4. int factr(int n);
5. int fact(int n);
6. int main()
7. {
8.     //使用递归
9.     cout << "4 factorial is " << factr(4);
10.    cout << "\n";
11.    //使用循环
12.    cout << "4 factorial is " << fact(4);
13.    cout << "\n";
14.    return 0;
15.}
16.//递归
17.int factr(int n)
18.{
19.    int answer;
20.    if ( n== 1) return (1);
21.    answer = factr(n-1) * n; //递归调用函数 factr()
22.    return answer;
23.}
24.//循环
```

```
int fact ( int n )
{
    int t, answer;
    answer =1;
    for ( t = 1; t <= n ; t++)
    {
        answer = answer * t;
    }
    return (answer);
}
```

采用循环方式计算阶乘的 fact()函数比较简单。其中使用了一个从 1 开始的循环，每次都对乘积的结果再乘以当前的循环数。

采用递归方式的函数 factr()有点复杂。当函数 factr()接收的参数为 1 的时候，函数就返回 1；否则，就返回 factr(n-1)与 n 的乘积。这种方式一直进行到 n 等于 1 的时候，函数调用才依次返回。例如，当计算 2 的阶乘的时候，第一次调用 factr()的时候会导致再次调用该函数，传入参数 1。此时函数返回 1，然后这个 1 和 2 相乘得到阶乘为 2。我们可以在函数 factr()中增加 cout 语句，这样就可以看到每次 factr()被调用的级别和其返回的值是多少。

但函数调用自己的时候，局部变量和形参都会被重新分配存储空间（通常是分配在系统的栈空间上），函数会从头开始使用这些新的变量来进行计算。当每次函数返回的时候，函数用的局部变量和形参变量就会被从栈空间中移除，程序返回到调用处的下一条指令处继续执行。递归就像“望远镜”一样可以进行伸缩。必须明确，大部分的递归程序实际上并不能减少代码量。另外，递归函数的执行要比对等的循环方式的函数速度要慢一些。这是因为函数的参数和局部变量都是在系统的栈空间中进行保存的，每调用一次都会进行一次对这些变量的拷贝，这样就有可能导致栈空间被耗尽。如果这点确实发生了，那么数据就会被破坏。然而，在递归层次不是很深的情况下，我们没有必要担心这一点。

使用递归的一个显著特点就是和对等的循环方式来比，一些算法采用递归则会更清晰和简单。例如，快速排序算法就很难用循环的方式来实现。另外，一些问题特别是和人工智能相关的问题采用的都是递归。

在编写递归程序的时候，我们必须写上一个 if 这里的条件语句，来迫使函数不再递归调用就可以返回。如果不提供这样的条件语句，一旦函数被调用就永远不会返回。这也是一个常见的错误。当我们编写带有递归函数的程序的时候，我们可以有意地在其中加入 cout 语句以便观察程序的执行，一点发现错误就应该离开终止程序。下面是另外一个使用递归函数 reverse()的程序，它把字符串参数逆序打印出来。

view plain

```
#include <iostream>
using namespace std;
void reverse( char *s);
```

```
4.  int main()
5.  {
6.      char str[] = "this is a test";
7.      reverse(str);
8.      return 0;
9.  }
10. //逆序打印字符串
11. void reverse(char *s)
12. {
13.     if (*s)
14.         reverse(s+1);
15.     else
16.         return;
17.     cout << *s;
18. }
```

函数 reverse()先是检查传入的指针是否指向字符串的结束标志。如果不是，则递归调用 reverse()，传入指向下一个字符的指针。最终当传入的是指向字符串结束标志的指针的时候，函数调用开始依次返回，字符串就被逆序打印出来。

最后一点：递归对于初学者来说会比较难。虽然你可能目前对递归有点迷惑，但是请不要放弃。随着时间的推移，我们会越来越熟悉递归的。

项目 5-1 快速排序

在第四章中，我们学习了一种简单的排序方法叫做冒泡排序法。当时我们也谈到过有更好的排序方法。本章中我们就将学习一种最好的排序方法：快速排序法。快速排序法是由 C.A.R.Hoare 发明的。它是目前最好的通用的排序算法。在第四章中我们之所以没有使用快速排序法是因为其最好的实现需要使用到递归。本章中我们编写的快速排序算法将是针对一个字符数组进行排序，但是其排序的逻辑是可以应用于任何类型的对象的。

快速排序的思想是基于分段的。过程如下：选择一个称之为比较数的值，然后把数组分成两部分。所有大于或者等于比较数的值放置在数组的一端；小于比较数的值放置在数组的另一端。然后对两端再分别重复上面的过程，直到整个数组变成有序的。例如，数组 fedacb，比较数为 d，那么第一次进行快速排序后的结果将为 bcadef。然后对其两个部分 bca 和 def 分别重复上面的过程。从中可以看出这实际上是一个递归的过程。实践中最简洁的快速排序法的实现也就是使用递归。

比较数的选择可以有两种方式：随机的选取或者是对数组中一小部分数字取其平均值。最好的排序是选择数组值中的中间值作为比较数。但是，对于大多数的数据集合来说，找出这个中间值可不容易。最差的排序是选择数是数组中的最大值或者最小值这种极端的情况。

即使是在上面的极端情况下，快速排序法仍然能够正确的进行排序。本章中我们编写的

快速排序算法是采用中间位置的元素作为比较数。

还有一点，C++的库函数中已经含有了一个进行快速排序的函数 qsort()。我们可以把它和我们下面自己编写的快速排序函数进行比较。

步骤：

1. 创建一个名称为 QSDemo.cpp 的文件
2. 快速排序将通过一对函数来实现。其中函数 quicksort()为用户提供了一个方便的接口，其中调用到了实际进行排序的函数 qs()。首先我们来实现 quicksort()函数，如下：

view plain

```
1.  //函数实际上调用 qs()函数来完成排序功能
2.  void quicksort(char *items, int len)
3.  {
4.      qs(items, 0, len -1);
5.  }
```

其中，items 指向需要排序的数组，len 指明了该数组的大小。在下面我们会看到函数 qs()需要一个初始的分段，这里在函数 quicksort()中提供了这个初始的分段。这样做的好处是我们只需要调用函数 quicksort()来进行排序，它只需要一个指向数组的指针参数和一个表示数组大小的参数，共计两个参数。由这个函数把数组的开始和结束索引作为参数传递给 qs()函数完成排序。

3. 添加完成实际排序功能的函数 qs()的实现，如下：

view plain

```
1.  //递归函数完成对字符数组的快速排序
2.      void qs(char *items, int left, int right)
3.      {
4.          int i,j;
5.          char x, y;
6.
7.          i = left;
8.          j = right;
9.          x = items[(left + right) / 2];
10.
11.         do
12.         {
13.             while(items[i] < x && ( i < right) ) i++;
14.             while(items[j] > x && ( j > left ) ) j--;
15.
16.             if ( i <=j )
```

```
            {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++;
                j--;
            }

        }while(i <= j );

        if ( left < j ) qs(items,left, j);
        if ( i < right ) qs(items, i,right);
    }
```

调用这个函数的时候需要传入分段的边界索引。变量 left 表示分段的左边界。变量 right 表示分段的右边界。第一次调用该函数的时候，分段就是整个数组。每次递归调用的时候分段就会变得越来越小。

4. 进行快速排序的时候，我们只需要调用函数 quicksort()即可，传入数组的名称和数组的长度。当该函数返回后，数组中的元素就是有序的了。请记住，我们编写的这个快速排序函数是针对字符数组的。但是，排序的逻辑是适用于任何类型的数组的。

5. 完整的程序如下：

view plain

```
/* 采用快速排序法完成对字符数组的排序*/
#include <iostream>
#include <cstring>

using namespace std;

void qs(char *items, int left, int right);
void quicksort(char *items, int len);

int main()
{
    char str[] ="jfmckldoelazlkper";

    cout << "Original order: " << str << "\n";

    quicksort(str, strlen(str) );

```

```
18.         cout << "Sorted order: " << str << "\n";
19.
20.         return 0;
21.
22.     }
23.
24.     //递归函数完成对字符数组的快速排序
25.     void qs(char *items, int left, int right)
26.     {
27.         int i,j;
28.         char x, y;
29.
30.         i = left;
31.         j = right;
32.         x = items[(left + right) / 2];
33.
34.         do
35.         {
36.             while(items[i] < x && ( i < right) ) i++;
37.             while(items[j] > x && ( j > left ) ) j--;
38.
39.             if ( i <=j )
40.             {
41.                 y = items[i];
42.                 items[i] = items[j];
43.                 items[j] = y;
44.                 i++;
45.                 j--;
46.             }
47.
48.         }while(i <= j );
49.
50.         if ( left < j ) qs(items,left, j);
51.         if ( i < right ) qs(items, i,right);
52.     }
53.
54.     //函数实际上调用 qs()函数来完成排序功能
55.     void quicksort(char *items, int len)
56.     {
57.         qs(items, 0, len -1);
58.     }
```

程序的输出如下：

Original order: jfmckldoelazlkper

Sorted order: acdeefjkkllmoprz

问专家：

问：我听说有“默认是 int 类型”的规则，它是什么意思，适用于 C++吗?

答：在最早的 C 语言中以及早期的 C++版本中，如果没有指明类型，那么类型就是整型。例如，在老的代码中，下面的函数是有效的，其返回值为整型：

```
f() //缺省地函数返回整型
{
    int x;
      // ...
    return x;
}
```

这里没有指明函数的返回值，则函数 f()的返回值采用缺省的整型。然而，现在的 C++是不支持这种“默认是 int 类型的”规则。尽管大多数的编译器都是支持对该规则的编译，以便实现向前兼容，但是我们还是应该明确指定自己所编写的每个函数的返回类型。由于老代码中依然会存在这种“默认是 int 类型”的写法，所以当我们在对老代码进行转换的时候需要牢记这点。

复习题：

1. 写出函数的一般形式。

2. 编写一个函数，其名称为 hypot()，用来计算直角三角形斜边的长度，给定两个直角边的长度；并在程序中使用这个函数。该问题需要使用标准函数库中的函数 sqrt()。它用来计算参数的平方根。其原型如下：

```
double sqrt(double val);
```

这个函数需要使用头文件<cmath>

3. 函数是否可以返回指针？是否可以返回数组？

4. 自己编写函数实现标准库中 strlen()的功能，其名称为 mystrlen()。并在程序中演示它的用法。

5. 在函数中声明的局部变量是否会在函数多次被调用之间保留其值？

6. 说出全局变量的一个好处？说出全局变量的一个弊端？

7. 编写一个函数，其名称为 byThrees()，用来返回一些列的数字，每次返回的值都比前一次返回的值大 3.。序列从 0 开始。也就是说 5 次调用函数，其返回的结果应该分别为 0,3,6,9 和 12。同时，编写一个函数名称为 reset()，用来使得函数 byThrees()返回的数值重新从 0 开始。演示在程序中如何使用这两个函数。提示：需要使用全局变量。

8. 编写一个程序，运行时需要从命令行输入密码。程序不做实质性的事情，只是报告输入

的密码是否正确。

9. 函数原型可以避免函数在被调用的时候传入不正确的参数个数。对吗？

10. 编写一个递归程序用来打印数字 1 到 10，并在程序中演示如何使用这个函数。

第六篇 进一步了解函数

在本章中，我们将更深入地研究函数。我们将讨论和函数相关的重要的三个 C++特性：引用，函数的重载和缺省参数。这些特性大大地扩展了函数的能力。引用是一种隐式的指针。函数的重载使得一个函数可以有多种实现，每种实现完成一种单独的任务。函数的重载是C++支持多态性的一种方式。函数缺省的参数是指在调用函数的时候没有指明具体的参数，则函数自动采用缺省参数。我们将从参数传入到函数中的两种方式和这两种方式的实现开始讨论。理解函数参数的传递方式对于理解引用是有必要的。

必备技能 6.1：参数传递的两种方式

通常，计算机语言中有两种方式来传递参数给子程序。第一种叫做传值。这种方式是指把实参的值复制给子程序的形参。因此，子程序中对于形参的修改不会影响传入的实参的值。

第二种参数传递的方式叫做传引用。这种方式是指实参的地址被复制给子程序的形参。在子程序中使用的是地址来访问调用时传入的实际参数。这就意味着，在子程序中对形参的修改会影响传入的实参的值。

必备技能 6.2：C++中参数的传递方式

缺省情况下，C++使用的是传值的方式。这就意味着函数中的代码对形参值的改变不会影响调用时传入的实参的值。在本书中，到目前为止的所有程序都是传值的方式传递参数的。例如，我们可以看一下下面程序中的函数 reciprocal():

view plain

```
1. <pre name="code" class="cpp">// 传值的方式中，对形参值的改变不会影响实参的值
2. #include <iostream>
3. using namespace std;
4. double reciprocal(double x);
5. int main()
6. {
7.     double t = 10.0;
8.     cout << "Reciprocal of 10.0 is " << reciprocal(t) << '/n';
9.     cout << "Value of t is still: " << t << "/n";
10.    return 0;
11. }
12. // 返回一个数的倒数
13. double reciprocal(double x )
14. {
15.    x = 1 / x; //这里对 x 值的修改不会影响传入的 t 的值
```

```
16.     return x;
17. }
```

在函数 reciprocal()中我们修改了局部变量 x 的值。在 main()函数中传入的实参 t 的值是不受影响的，在调用了 reciproacl()函数后，其值还是 10。

必备技能 6.3：使用指针来实现传引用

虽然在 C++中缺省的方式是值传递，但是我们可以通过传递参数的地址来实现传引用。通过这种方式我们可以在函数之外修改函数中变量的值。我们在前面讨论传递指针到函数中的时候看到过这样的例子。传递指针到函数中其实是和传递其它类型的数据是一样的。当然，需要我们把函数的形参声明为指针的类型。

为了更加深刻理解为什么传递指针可以手动地实现传引用的功能，我们可以研究一下下面的函数 swap()。这个函数用来交换两个指针指向变量的值。下面就是一种实现方法：

view plain

```
1. void swap( int *x,int *y)
2. {
3.     int temp;
4.     temp = *x; //把 x 地址中的值保存在 temp 中
5.     *x = *y;    //把 y 地址中的值复制给 x 地址
6.     *y = temp;  //把 temp 的值赋值给 y 地址
7. }
```

swap()函数的参数是两个指针：x 和 y。通过使用这两个参数来实现传入的指针指向地址的值的交换。在这里需要注意*x,*y 指的是由 x 和 y 指向的变量，因此表达式

*x = *y;

是把 y 指向的变量的值赋值给 x 指针指向的变量。这样一来，当函数执行完毕的时候，调用该函数时传入的变量的内容就会被交换了。

既然 swap()函数需要的是两个指针，我们必须在调用函数 swap()的时候传入想要交换值的两个变量的地址。正确的调用方法如下面的程序所示：

//该程序演示使用指针来实现交换值的函数 swap()

view plain

```
1. #include <iostream>
2. using namespace std;
3. //声明函数 swap()使用指针作为参数
4. void swap( int *x,int *y);
5. int main()
6. {
```

```
7.      int i, j;
8.      i = 10;
9.      j = 20;
10.     cout <<"Initial values of i and j: ";
11.     cout << i << ' ' << j << '\n';
12.     swap(&j,&i);
13.
14.     cout <<  "Swapped values of i and j : ";
15.     cout << i << ' ' << j << '\n';
16.     return 0;
17. }
```

在 main()函数中，变量 i，j 分别被赋值为 10，20。然后使用了 i 和 j 的地址来调用函数 swap()。其中的单目运算符&得到的是变量的地址。因此，我们传递给 swap()函数的是变量 i,j 的地址而不是值。当函数 swap()返回的时候，变量 i 和 j 的值就被交换了。程序的输出结果如下：

Initial values of i and j: 10 20

Swapped values of i and j : 20 10

练习：

1. 什么是值传递？
2. 什么是引用传递？
3. C++中缺省的参数传递方式是那种方式？

必备技能 6.4：引用参数

前面我们看到可以通过使用指针的方式来手动地实现传引用，但是这种方式显得有些笨拙。首先，这种方式强制我们所有的操作都是基于指针的。其次，这种方式要求我们必须铭记在传参数的时候需要传入变量的地址。然而，C++中有一种方式可以自动地告诉编译器在调用某些函数的时候使用的是传引用而不是缺省的传值。这是通过引用参数来实现的。当我们使用引用参数的时候，参数的地址而不是值自动地被传入到函数中。在被调用的函数中，对引用参数的操作会被 C++编译器自动感知为对地址的操作，因此我们不需要使用和指针相关的那些运算符。

声明函数的参数为引用参数的时候，只需要在参数的前面加上&即可。对于引用参数的操作会影响实参的值，而不是影响引用参数。

为了更好地理解引用参数，我们可以看看下面简单的程序。在下面的程序中，函数 f()使用一个 int 类型的引用参数。

view plain

```
1.  // 使用引用参数
2.  #include <iostream>
```

```
3.  using namespace std;
4.  void f ( int &i ); //这里 i 是引用参数
5.  int main()
6.  {
7.      int val = 1;
8.      cout << "Old value for val: " << val << '\n';
9.      f(val);
10.     cout << "New value for val: " << val << '\n';
11.     return 0;
12. }
13. void f(int &i)
14. {
15.     i = 10; //这句将导致传入的参数的值被修改
16. }
```

程序的输出如下：

Old value for val: 1

New value for val: 10

我们需要特别注意函数 f()的定义：

view plain

```
1.  void f(int &i)
2.  {
3.      i = 10; //这句将导致传入的参数的值被修改
4.  }
```

注意上面对 i 的声明。变量 i 的前面多了一个&，它就会使得变量 i 变为一个引用参数。（在函数的声明中也需要这样写。）在函数的内部，语句

i = 10;

不是给 i 赋值 10，而是给被 i 引用的变量（实际上就是 val）赋值为 10。还需要注意的是，这条语句中没有使用与指针相关的*运算符。当我们使用引用参数的时候，C++编译器能自动地知道它是一个地址。实际上，在代码中写上*会导致错误。

由于 i 被声明为是一个引用参数，编译器会在调用 f()函数的时候把实参的地址传递给 f()。因此 main()函数中是把 val 的地址，而不是值传递给了 f()。这里我们没有必要在 val 的前面增加&运算符了（实际上这样做会导致编译错误）。既然 f()函数接收到的是以引用的方式接收到 val 的地址，它就可以修改变量 val 的值。

为了充分展示实际中引用参数的用法以及带来的好处，我们可以采用引用参数的形式重写一下 swap()函数。请特别留意下面的 swap()函数的声明和调用。

view plain

```cpp
//使用引用参数来实现 swap()函数
#include <iostream>
using namespace std;
//声明 swap()函数使用引用参数
void swap(int &x, int &y);
int main()
{
    int i,j;
    i = 10;
    j = 20;
    cout << "Initial values of i and j : ";
    cout << i << ' ' << j << '\n';
    swap(j, i);
    cout << "Swapped values of i and j : ";
    cout << i << ' ' << j << '\n';

    return 0;
}
/*
    下面采用引用参数的方式，而不是传值的方式来实现 swap()函数。
    因此，swap()函数能够完成对传入参数的值的交换
*/
void swap(int &x, int &y)
{
    int temp ;
    //使用引用来完成参数值的交换。
    temp = x;
    x = y;
    y = temp;
}
```

这个程序的输出和上一个版本的输出结果是一样的。再次提醒，因为 x，y 都是引用参数，所以我们在交换值的时候不需要使用*运算符。请记住：在调用函数 swap()的时候，编译器会自动地传入实参的地址。

我们复习一下前面的内容。当创建了引用参数的时候，参数会自动地表示对传入实参的引用。进一步来说，调用该函数的时候没有必要使用&运算符。同样，在该函数的内部，我们直接使用引用参数，而不用使用*运算符。在函数中所有对于引用参数的操作都会自动转换为对传入实参的操作。最后，当我们在给引用参数赋值的时候，我们实际上是给它指向的对象进行赋值。

最后一点，C 语言是不支持引用的。因此，在 C 语言中使用指针是实现传引用的唯一方

式，就像我们在 swap()函数的第一个版本的那样。当需要把 C 代码转换为 C++代码的时候，我们可能需要把这种代码转换为引用参数的方式。

专家答疑

问：

在一些 C++的代码中，我看到一些变量在声明的时候，&是和变量的类型结合在一起的，而不是和变量自身结合在一起的，如下所示：

```
int& i;
```

而不是:

```
int &i;
```

这两种方式有区别吗？

答：

简而言之，是没有区别的。比如，我们也可以采用如下的方式来声明 swap()函数的原型:

```
void swap( int& x, int& y);
```

正如我们所看到的那样，&是紧跟在 int 的后面，而不是紧密地写在了 x 的前面。更有甚者，一些程序员也会把声明指针的*紧密跟在类型的后面，而不是写在变量名称的前面，如下：

```
float* p;
```

这种写法反映出了 C++程序员试图创建一个单独的引用或者指针类型的良好愿望。然而这种写法存在一个问题。那就是，根据 C++的语法，&和*是不能作用于一系列变量的。例如下面的声明方式声明的是一个 int 类型的指针，而不是两个：

```
int* a, b;
```

这里，b 是一个 int 类型的变量，而不是一个 int 类型的指针变量。这是因为，根据 C++语法，当在声明中使用*或者&得时候，它们是和其后的单个变量相关联的，而不是和前面的类型相关联的。

需要明确的是，就 C++编译器而言，我们到底是写 int *p 还是 int* p 并不重要。因此，如果你个人喜欢把*和&紧密地和类型写在一起，而不是和变量的名称写在一起的话，你完全可以这样做的。然而，为了避免混淆，本书中我们将始终把*和&写在它们所修饰的变量名称的前面，而不是紧跟在类型的后面。

练习：

1. 如何声明引用参数？
2. 当调用使用引用参数函数的时候，我们必须在传入参数的前面使用&吗？

必备技能 6.5：返回引用

在 C++中，函数可以返回一个引用。函数返回引用有好几种用法，其中一些我们会在本书的后面讲到。这里我们涉及其中的一部分。

如果一个函数返回的是一个引用，那么它实际上暗含地返回了一个指向返回值的指针。这就出现了一个令人乍舌的情况：函数可以出现在一个赋值语句的左侧！例如下面的程序：

view plain

```
1. //函数返回引用
2. #include <iostream>
3. using namespace std;
4. double &f(); //函数 f()返回引用
5. double val = 100.0;
6. int main()
7. {
8.     double x ;
9.     cout << f() << '\n'; //输出 val 的值
10.    x = f(); //把 val 的值赋值给 x
11.    cout << x << '\n'; //输出 x 的值
12.    f() = 99.1; //修改 val 的值
13.    cout << f() << '\n'; //输出 val 的值
14.    return 0;
15. }
16. double &f()
17. {
18.    return val; //返回全局变量 val 的引用
19. }
```

程序的输出如下：

100

100

99.1

我们仔细研究一下上面的程序。在程序的开始，函数 f()被声明为返回一个 double 类型的引用，全局变量 val 被初始化为 100。在 main()函数中，采用了下面的语句打印 val 的初始值:

cout << f() << '\n'; //输出 val 的值

当调用函数 f()的时候，它返回 val 的引用：

return val; //返回全局变量 val 的引用

该句自动地返回了对 val 的引用，而不是 val 的值。然后我们使用这个引用在 cout 语句中输出其值。

在下面的一行：

x = f();//把 val 的值赋值给 x

通过函数 f()返回的引用把 val 的值赋值给 x。程序中最有意思的一行就是下面的代码：

f() = 99.1; //修改 val 的值

这句将使得变量 val 的值被修改为 99.1。原因如下：既然函数 f()返回的是对 val 的引用，那么该引用就是赋值的对象。这样一来，99.1 就被间接地通过函数 f()返回的引用赋值给了 val。

下面是另外一个使用返回引用的示例程序：

view plain

```
1. //返回数组元素的引用
2. #include <iostream>
3. using namespace std;
4. double &change_it(int i); //函数返回引用
5. double vals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
6. int main()
7. {
8.     int i;
9.
10.    cout << "Here are the original values: ";
11.    for ( i = 0; i < 5; i++)
12.    {
13.        cout << vals[i] << ' ';
14.    }
15.    cout << '\n';
16.    change_it(1) = 5298.23; //修改第二个元素的值
17.    change_it(3) = -98.8;   //修改第四个元素的值
18.    cout << "There are the changed values: ";
19.    for ( i = 0; i < 5; i++)
20.    {
21.        cout << vals[i] << ' ';
22.    }
23.    cout << '\n';
24.
25.    return 0;
26. }
27. double &change_it(int i)
28. {
29.    return vals[i]; //返回对第 i 个元素的引用
30. }
```

程序通过使用返回引用的函数修改了数组的第二个和第四个元素的值。输出如下：

Here are the original values: 1.1 2.2 3.3 4.4 5.5

There are the changed values: 1.1 5298.23 3.3 -98.8 5.5

让我们来看看这一点是如何实现的。

函数 change_it 被声明为返回 double 类型引用。更为明确地说，它返回的是对数组 vals 的第 i 个元素的引用。在 main()函数中，这个引用被用在赋值语句中来给元素赋值。

当返回引用的时候，必须注意返回的对象不能超越它的作用范围。例如下面的程序：

view plain

```
1.  //错误！不能返回一个对局部变量的引用
2.  int &f()
3.  {
4.      int i = 10;
5.      return i; //错误！当函数 f()返回后，i 就超出了它的作用域
6.  }
```

在上面的函数 f()中，当函数返回的时候，局部变量 i 就超出了它的作用域。因此，返回的对于 i 的引用将是没有定义的。实际上，一些编译器正是出于上述的原因而不会把函数 f()编译成如代码所写的那样。然而，这种问题是间接引起的，我们必须谨慎处理应该返回对哪些对象的引用。

必备技能 6.6：单独的引用

尽管 C++中引入引用的主要目的是为了支持引用参数的传递方式和函数返回引用，但是我们同样可以声明单独的引用变量。但是必须说明，这种非参数的单独引用很少被使用。这是因为这种单独的引用会破坏程序的结构，让人容易产生混淆。因此，我们在这里也只是简单介绍一下单独引用。

单独的引用是必须指向某些对象的。因此，单独的引用变量必须在声明的时候就进行初始化。通常情况下，这就意味着，它会被初始化为之前声明的变量的地址。引用变量一旦被初始化之后，就可以出现在任何它所引用的变量可以出现的地方。实际上，两者之间是没有区别的。例如下面的程序。

view plain

```
1.  //使用单独的引用
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      int j,k;
7.      int &i = j; //单独的引用
8.      j = 10;
9.      cout << j << " " << i;
```

```
10.     k =121;
11.     i = k; //把 k 的值赋值给 j，而不是 k 的地址
12.     cout << "\n" << j; //将输出
13.     return 0;
14. }
```

程序的输出如下：

10 10

121

引用变量指向的地址是固定的，不能被改变的。因此，当计算语句

i = k;

的值的时候，是把 k 的值通过引用变量 i 赋值给 j 的，而不是 k 的地址。

正如前面描述的那样，使用单独的引用变量并不是什么好主意，因为使用单独的引用变量并不是必须的，而且它会使得程序变得很混乱。同一个变量有两个名字的确是一件很容易让人混淆的事情。

使用引用时的一些限制

使用引用的时候需要注意下面的限制：

不能对引用引用变量。

不能创建引用变量的数组。

不能创建指向引用变量的指针。也就是说不能对引用变量使用&运算符。

练习:

1. 函数是否可以返回引用?
2. 什么是单独的引用？
3. 我们是否可以创建引用的引用？

必备技能 6.7：函数重载

在本小节中，我们将学习到 C++中一个令人兴奋的特性：函数重载。在 C++中，只要函数的参数声明不同，两个或者多个函数是可以使用同一个名称的。这时，我们就说这两或者多个函数被重载了。这样的过程就是函数重载。函数重载是 C++支持多态性的一种方式。

通常情况下，为了重载一个函数，我们只需要声明一个函数的另外一种形式即可，剩余的事情有编译器来处理。函数重载必须遵循一个重要的原则：重载函数的参数类型或者参数数量不能相同。它们要么是参数的个数不相同，要么是参数的类型不相同。（由于函数的返回值不能提供足够的信息以便区分应该调用哪个版本的函数，所以只有返回值不同的函数不能重载。）当然，重载的函数是可以在返回值上不相同的。当我们调用重载的函数的时候，参数能够匹配上的那个版本的函数会被调用。

我们以下面的这个短小的程序开始学习函数重载：

view plain

```
1. //重载一个函数三次
2. #include <iostream>
3. using namespace std;
4. //针对每一次的重载，都需要声明函数的原型
5. void f(int i );          //整型参数
6. void f(int i, int j );   //需要两个整型参数
7. void f(double k);        //浮点型参数
8.
9. int main()
10. {
11.     f(10);       //调用函数 f(int)
12.     f(10,20);    //调用函数 f(int,int)
13.     f(12.23);    //调用函数 f(double)
14.     return 0;
15. }
16.
17. //下面是三个 f()函数的不同的实现
18. void f(int i)
19. {
20.     cout << "In f(int), i is " << i << '\n';
21. }
22. void f(int i, int j)
23. {
24.     cout << "In f(int), i is " << i;
25.     cout << ", j is " << j << '\n';
26. }
27. void f(double k )
28. {
29.     cout << "In f(double), k is " << k << '\n';
30. }
```

上面程序的输出如下：

In f(int), i is 10

In f(int), i is 10, j is 20

In f(double), k is 12.23

从中可以看出，函数 f()被重载了三次。第一次，需要一个整型数作为参数；第二次需要两个整型数作为参数；第三次需要一个 double 类型的参数。因为三个版本所需要的参数的列表是不一样的，编译器就能够根据调用时传入参数的类型来决定应该调用哪个版本的函数。为了能更好地理解重载的价值，我们参考函数 neg()，用来返回参数的相反数。例如，

当调用时传入参数-10，neg()就返回 10；当传入参数 9，neg()就返回-9。在没有函数重载的情况下，如果我们想创建求相反数的函数来分别实现对 int、double 和 long 三种类型的参数求相反数，那么我们就需要三个不同的函数，每个函数的名称都不一样，例如，ineg(),lneg()和 fneg()。然而，通过使用函数重载，我们就只需要一个函数名称 neg()即可，用它代表所有能返回参数相反数的函数。因此，重载支持了多态性概念“一个接口，多种实现”。下面的程序展示了这一点：

view plain

```
1. //创建函数 neg()的不同版本
2. #include <iostream>
3. using namespace std;
4. int neg(int n);          //neg() for int
5. double neg(double n );   //neg() for double
6. long neg(long n );       //neg() for long
7. int main()
8. {
9.     cout << "neg(-10): " << neg(-10) << "\n";
10.    cout << "neg(9L): " << neg(9L) << "\n";
11.    cout << "neg(11.23): " << neg(11.23) << "\n";
12.    return 0;
13. }
14. //neg() for int
15. int neg(int n )
16. {
17.    return -n;
18. }
19. //neg() for double
20. double neg(double n )
21. {
22.    return -n;
23. }
24. //neg() for long
25. long neg(long n )
26. {
27.    return -n;
28. }
```

上面程序的输出结果如下：

neg(-10): 10

neg(9L): -9

neg(11.23): -11.23

在上面的程序中，我们创建了三个相似但却不同的函数，它们的名称都是 neg(),每一

个都是返回其参数的相反数。在调用该函数的时候，编译器能根据参数的类型确定应该调用哪个函数。

重载的价值就在于它使得我们可以使用同一个名称来访问一组相关的函数。因此，名称 neg 就代表了函数完成的通用功能。由编译器根据调用时传入参数的情况来确定应该调用哪个版本的函数。我们，也就是程序员只需要记住这个通用的名称即可。因此，通过使用多态性，我们需要记住的东西就由三个变成了一个。尽管，上面的程序十分简单，但是如果我们将这个概念扩展开来，我们就能看到重载是如何帮助我们管理更大型程序的。

函数重载的另外一个好处：就是一些功能针对不同的数据类型其实现不同，但是确可以使用相同的函数名称。例如，假设我们想实现计算两个值中较小值的功能。这是针对不同的数据类型，其实现有可能不同，但是我们可以使用同一个函数名称 min()。当我们比较的是两个整型数的时候，函数 min()返回其中数值较小的那个。当我们比较的是两个字符的时候，min()函数就返回最先出现在字母表中的那个字符，不考虑大小写。在 ASCII 码序列中，大写字母的表示之要比小写字母的表示值小 32。因此，当按照字母顺序排序的时候，忽略大小写是十分有用的。当我们比较的是两个指针的时候，我们可以让 min()函数比较两个指针指向的值，并返回指向较小值的那个指针。下面的程序就实现了上述的三个版本的 min()函数：

view plain

```
1. //min()函数的不同实现
2. #include <iostream>
3. using namespace std;
4. int min(int a, int b); //整型参数的min()函数
5. char min(char a, char b); //字符型参数的min()函数
6. int* min(int* a, int* b); //指针型参数的min()函数
7. int main()
8. {
9.     int i = 10, j =22;
10.    cout << "min('X','a'): " << min('X','a') << "\n";
11.    cout << "min(9, 3): " << min(9,3) << "\n";
12.    cout << "*min(&i,&j): " << *min(&i, &j) << "\n";
13.    return 0;
14. }
15. //整型参数的min()函数，返回较小的值
16. int min(int a, int b)
17. {
18.    if ( a < b) return a;
19.    else return b;
20. }
21. //字符型参数的min()函数，比较字母的时候忽略大小写
22. char min( char a, char b)
23. {
```

```
24.     if (tolower(a) < tolower(b) ) return a;
25.     else return b;
26. }
27. //指针型参数的 min()函数，比较指针指向值的大小，返回指向较小值的指针
28. int * min(int* a, int *b)
29. {
30.     if ( *a < *b ) return a;
31.     else return b;
32. }
```

上面程序的输出如下：

min('X','a'): a

min(9, 3): 3

*min(&i,&j): 10

当我们重载函数的时候，函数的每个版本都可以实现任何我们想完成的功能。也就是说，没有规则要求我们重载的函数必须是相关联的。然而，从格式上来讲，函数重载就暗含着这些函数是相关联的。因此，当函数是不相关的时候，我们就不应该使用函数重载，而让它们使用相同的函数名称。例如，我们可以使用 sqrt()这个名称来创建两个函数，一个返回整型数的平方，另一个则返回一个 double 类型数的平方根。这两个功能显然是不同的，这样的用法就违背了函数重载的本意。（实际上，这种编程方式是一种极其不好的编程风格！）实践中，我们只应该对那些紧密相关的函数进行重载。

自动类型转换和重载

回忆一下在第二篇中，我们学习到 C++提供了一定的自动类型转换功能。这种自动类型转换同样可用于重载函数的参数。例如下面的程序：

view plain

```
1.  /*
2.      自动类型转换可以影响函数重载的结果
3.  */
4.  #include <iostream>
5.  using namespace std;
6.  void f(int x);
7.  void f(double x);
8.  int main()
9.  {
10.     int i = 10;
11.     double d = 10.1;
12.     short s =99;
13.     float r = 11.5F;
14.     f(i); //调用函数 f(int)
15.     f(d); //调用函数 f(double)
```

```
    f(s); //调用函数 f(int) 自动类型转换
    f(r); //调用函数 f(double) 自动类型转换
    return 0;
}
void f( int x)
{
    cout <<"Inside f(int): " << x << "/n";
}
void f( double x)
{
    cout <<"Inside f(double): " << x << "/n";
}
```

上面程序的输出结果为：

Inside f(int): 10

Inside f(double): 10.1

Inside f(int): 99

Inside f(double): 11.5

在上面的例子中，我们只是定义了两个版本的 f()函数：一个需要一个 int 类型的参数，另一个需要的是 double 类型的参数。然而，我们还是可以给函数 f()传入一个 short 类型或者 float 类型的参数。这里，C++自动地把 short 转换为 int 类型，因此调用的是函数 f(int)；而把 float 转换为 double 类型，因此调用的是函数 f(double)。

有一点必须明确，那就是自动类型转换只有在实参和函数的形参不能完全匹配的情况下才进行。例如，我们针对上面的程序增加一个需要 short 类型参数的的 f()函数：

view plain

```
/*
    自动类型转换可以影响函数重载的结果
*/
#include <iostream>
using namespace std;
void f(int x);
void f(short x); //增加一个需要 short 类型参数的函数 f()
void f(double x);
int main()
{
    int i = 10;
    double d = 10.1;
    short s =99;
    float r = 11.4F;
    f(i); //调用函数 f(int)
```

```
16.     f(d); //调用函数 f(double)
17.     f(s); //调用函数 f(short)
18.     f(r); //调用函数 f(double) 自动类型转换
19.     return 0;
20. }
21. void f( int x)
22. {
23.     cout <<"Inside f(int): " << x << "/n";
24. }
25. void f( short x)
26. {
27.     cout <<"Inside f(short): " << x << "/n";
28. }
29. void f( double x)
30. {
31.     cout <<"Inside f(double): " << x << "/n";
32. }
```

现在运行程序，输出如下：

Inside f(int): 10

Inside f(double): 10.1

Inside f(short): 99

Inside f(double): 11.4

在这个版本中，由于存在一个函数 f()它需要一个 short 类型的参数，所以当我们调用函数 f()时，如果传入的参数是 short 类型，就会调用需要 short 类型参数的这个函数 f(short)，而不会进行自动类型转换。

练习：

1. 进行函数重载必须满足什么条件？
2. 为什么重载函数必须完成的是相关的功能？
3. 函数的返回值是否参数函数重载的解析？

项目 6-1 重载输出函数

在下面的项目中，我们将创建一组重载的函数，用于把不同类型的数据输出到显示器上。尽管使用 cout 是相当便捷的，但是下面的程序提供了另外一种输出方式。这种输出方式或许对一些程序员来时是很有吸引力的。实际上，Java 和 C#中使用的都是输出函数，而不是输出运算符。重载输出函数后，我们既可以使用输出函数也可以使用输出运算符，那种方便我们就可以使用哪种方式。另外，我们还可以对输出函数进行裁剪，以满足自己特殊的需要。例如，我们可以输出布尔类型变量的值为“true”或者“false”，而不是 1 或者 0。

我们将创建两套函数：println()和 print()。println()函数把参数的值输出，并输出一个换行。

print()函数输出参数的值，但是不输出换行。如下:

print(1);

println('X');

print("Function overloading is powerful.");

print(18.22);

将产生如下的输出：

1X

Function overloading is powerful.18.22

我们在程序中会针对 bool,char,int,long,char *,double 类型来对 print()和 println()函数进行重载。当然，自己也可以增加针对别的类型的重载。

步骤：

1. 创建一个文件，命名为 Print.cpp
2. 以下面的代码开始：

view plain

```
/*
    项目-1
    重载函数 println()和 print()，用于显示各种类型的数据
*/

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

3. 如下，增加函数 print()和 println()的原型：

view plain

```
//下面的这些函数将输出一个换行
 void println(bool b);
 void println(int i);
 void println(long i);
 void println(char ch);
 void println(char * str);
 void println(double d);

 //下面的这些函数将不会输出换行
void print(bool b);
```

```
11.  void print(int i);
12.  void print(long i);
13.  void print(char ch);
14.  void print(char * str);
15.  void print(double d);
```

4. 如下，实现 println()函数

view plain

```
1.  //下面是 println()函数的实现
2.   void println(bool b)
3.   {
4.       if(b) cout <<"true\n";
5.       else cout << "false\n";
6.   }
7.   void println(int i)
8.   {
9.       cout << i << "\n";
10.  }
11.  void println(long i)
12.  {
13.      cout << i << "\n";
14.  }
15.  void println(char ch)
16.  {
17.      cout << ch << "\n";
18.  }
19.  void println(char * str)
20.  {
21.      cout << str << "\n";
22.  }
23.  void println(double d)
24.  {
25.      cout << d << "\n";
26.  }
```

注意上面的实现在每次输出之后增加了输出换行。同时，println(bool)函数输出的是“true”或者“false”。可见，我们可以很方便地自行定义输出格式以便满足自己的需要。

5. print()函数的实现如下：

view plain

```
//下面是print()函数的实现
 void print(bool b)
 {
     if(b) cout <<"true";
     else cout << "false";
 }
 void print(int i)
 {
     cout << i ;
 }
 void print(long i)
 {
     cout << i;
 }
 void print(char ch)
 {
     cout << ch;
 }
 void print(char * str)
 {
     cout << str;
 }
 void print(double d)
 {
     cout << d;
 }
```

这些函数除了不输出换行外，其它的都和 println()函数一样。因此，连续的输出将会是在同一行。

6. 完整的程序如下：

view plain

```
/*
     项目-1
     重载函数println()和print()，用于显示各种类型的数据
 */

```

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

//下面的这些函数将输出一个换行
void println(bool b);
void println(int i);
void println(long i);
void println(char ch);
void println(char * str);
void println(double d);

//下面的这些函数将不会输出换行
void print(bool b);
void print(int i);
void print(long i);
void print(char ch);
void print(char * str);
void print(double d);

int main()
{
    println(true);
    println(10);
    println("This is a test");
    println('x');
    println(99L);
    println(123.23);

    print("Here are some values: ");
    print(false);
    print(' ' );
    print(100000L);
    print(' ');
    print(100.01);

    println("Done!");

    return 0;

}


//下面是println()函数的实现
```

```
50. void println(bool b)
51. {
52.     if(b) cout <<"true\n";
53.     else cout << "false\n";
54. }
55. void println(int i)
56. {
57.     cout << i << "\n";
58. }
59. void println(long i)
60. {
61.     cout << i << "\n";
62. }
63. void println(char ch)
64. {
65.     cout << ch << "\n";
66. }
67. void println(char * str)
68. {
69.     cout << str << "\n";
70. }
71. void println(double d)
72. {
73.     cout << d << "\n";
74. }
75.
76. //下面是print()函数的实现
77. void print(bool b)
78. {
79.     if(b) cout <<"true";
80.     else cout << "false";
81. }
82. void print(int i)
83. {
84.     cout << i ;
85. }
86. void print(long i)
87. {
88.     cout << i;
89. }
90. void print(char ch)
91. {
92.     cout << ch;
93. }
```

```
94. void print(char * str)
95. {
96.     cout << str;
97. }
98. void print(double d)
99. {
100.     cout << d;
101. }
```

程序的输出如下：

true

10

This is a test

x

99

123.23

Here are some values: false 100000 100.01Done!

必备技能 6.8：缺省的函数参数

接下来我们要讨论的和函数相关的一个特性就是缺省参数。在 C++中，我们可以指定函数参数的缺省值。如果在调用给函数的时候，没有传入对应的参数，则函数使用缺省的参数值。缺省的函数参数可以用来简化对复杂函数的调用。有时，这种做法可以被看做是函数重载的一种紧缩形式。

在句法上，指明函数的缺省参数和对变量进行初始化是一样的。在下面的示例中，声明了需要两个整型参数的函数，第一个参数的缺省值为 0，第二个参数的缺省值为 100。

void myfunc(int x= 0, int y = 100);

这样就会有下面的三种调用该函数的方式：

myfunc(1,2). //明确地传入两个参数

myfunc(10); //只传入 x 的值， y 采用缺省的值

myfunc(); //x，y 都采用缺省的值

第一种调用的方式传入 1 给 x，传入 2 给 y。第二种调用方式传入 10 给 x，y 采用的是缺省的值 100。第三种调用方式 x 和 y 都采用缺省的值。正如下面的程序所示的那样：

view plain

```
1. //函数的缺省参数
2. #include <iostream>
3. using namespace std;
```

```
4.  void myfunc(int x = 0, int y = 100);  //函数的两个参数都指定了缺省值
5.  int main()
6.  {
7.      myfunc(1,2);
8.      myfunc(10);
9.      myfunc();
10.     return 0;
11. }
12. void myfunc(int x, int y )
13. {
14.     cout << "x: " << x << ", y: " << y << "\n";
15. }
```

上面程序的输出为：

x: 1, y: 2

x: 10, y: 100

x: 0, y: 100

当我们在使用缺省参数值的函数的时候，缺省的值只能被声明一次，而且必须是在文件中第一次声明该函数的时候就指明缺省的参数值。在上面的程序中，缺省参数就是在声明myfunc()函数原型的时候指定的。如果试图在函数的实现中指定一个新的缺省值（即使指定的是相同的值），编译器就会报告错误信息，并停止对程序的编译。

尽管函数的缺省参数只能被定义一次，但是我们完全可以为重载函数的不同版本指定不同的缺省值。也即是说，重载的函数的不同版本可以有不同的缺省值。

还有重要的一点是所有有缺省值的参数必须是放置在没有缺省值的参数的右边。例如，下面的函数原型是错误的：

void f(int a = 1, int b);

一旦我们在函数原型中书写了一个有缺省值的参数，那么在其后面我们就不能书写不带缺省值的参数了。也就是说下面的写法是错误的：

int myfunc(float t, char * str, int i = 10, int j); //这种写法是错误的

由于上面的参数 i 是有缺省值的，所以其后面的 j 也必须是有缺省值的。

C++中引入参数缺省值的一个原因就是它使得程序员可以方便地管理大型的程序。为了处理不同的情况，通常一个函数需要的参数要比它的常用用法使用的参数要多。因此，当使用了有缺省值的函数参数时，我们只需要记住并指明针对我们的情况有意义的参数即可，而不是需要记住通用形式的所有参数。

缺省参数与重载

缺省参数的一种应用就是可以被看做是函数重载的一种紧缩形势。为了能充分理解这一点，我们假设要创建两个完成标准函数 strcat()功能的函数。其中一个就是像 strcat()那样，把一个字符串的全部内容连接到另外一个字符串的尾部。另外一个则需要第三个参数来指明

需要连接的字符的数量。也就是说，第二个函数将把一个字符串中指定数量的字符连接在第二个字符串的尾部，而不是把第一个字符串的全部链接在第二个字符串的尾部。假设函数的名称为 mystrcat()，则它们的原型如下：

void mystrcat(char * s1, char * s2, int len);

void mystrcat(char * s1, char * s2);

其中第一个版本是把 s2 的 len 个字符连接在 s1 的后面。第二个版本是把 s2 字符串的全部连接到 s1 字符串的后面，就像 strcat()函数那样。

尽管实现上面的两个版本的函数是没有错误的，但是我们有着更为方便的方式。那就是使用缺省参数。通过使用缺省参数，我们只需要一个版本的 mystrcat()函数就能完成上述的两种功能。下面是程序的实现：

view plain

```
// 自己编写的 strcat 功能的函数
#include <iostream>
using namespace std;
void mystrcat(char * s1, char * s2, int len = 0);  //len 的缺省值为
int main()
{
    char str1[80] = "This is a test";
    char str2[80] = "0123456789";
    mystrcat(str1,str2,5); //只需要连接 5 个字符
    cout << str1 << "\n";
    strcpy(str1, "this is a test"); //重置字符串 str1
    mystrcat(str1,str2); //把整个字符串进行连接
    cout << str1 << "\n";
    return 0;
}
//自己定义的 strcat 功能的函数
void mystrcat(char * s1, char * s2, int len)
{
    //找 s1 的尾部
    while(*s1) s1++;
    if (len == 0 ) len = strlen(s2);
    while(*s2 && len )
    {
        *s1 = *s2;  //复制字符
        s1++;
        s2++;
        len--;
    }
    *s1 ='\0';  //标记是 s1 结束
}
```

上面程序的输出如下：

This is a test01234

this is a test0123456789

正如上面的程序所展示的那样，mystrcat()函数把 s1 字符串的 len 个字符连接在 s2 字符串的后面。然而，当 len 为 0 的时候，也就是取缺省值的时候，mystrcat()函数是把字符串 s2 整个连接到 s1 的后面。（也就是说，当 len 为 0 的时候，mystrcat()函数和标准的库函数 strcat()完成的功能是一样的。）

通过使用缺省参数，就可以把两个操作合并到一个函数里面。就像示例程序展示的那样，缺省参数有时提供的是函数重载的一种紧缩形式。

必备技能 6.9：函数重载与二义性

在结束本篇章之前，我们还有必要讨论一种 C++特有的错误：二义性。二义性是指存在这样的情况：编译器不能决定应该调用重载函数中的那个。当这种情况出现的时候，我们就说程序有二义性。具有二义性的语句是一种错误，含有二义性的程序是不能被编译的。

到目前为止，导致二义性的主要原因就是 C++中的自动类型转换。C++中的自动类型转换总是试图把调用函数时传入的参数转换成函数需要的类型。下面就是一个例子：

int myfunc(double d);

//…

cout << myfunc('c'); //这是正确的，这里会进行自动类型转换

正如上面的注释写的那样，上面的代码是正确的。因为 C++中的自动类型转换会把字符'c'转换成一个等价的 double 类型。实际上，在 C++中，类似于这样的自动类型转换大多数都是允许的。虽然，自动类型转换很便捷，它却是导致二义性的主要因素。看看下面的程序：

view plain

```
1. //重载引起的二义性
2. #include <iostream>
3. using namespace std;
4. float myfunc(float i);
5. double myfunc(double i);
6. int main()
7. {
8.     //无二义性，调用的是函数 myfunc(double)
9.     cout << myfunc(10.1) << " ";
10.    //有二义性
11.    cout << myfunc(10); //错误！，因为编译器不能判断到底应该调用哪个 myfunc()
   函数
12.    return 0;
13. }
14. float myfunc(float i)
```

```
{
    return i;
}
double myfunc(double i)
{
    return -i;
}
```

这里对函数 myfunc()进行了重载，它可以使用 float 或者 double 类型的参数。在代码 myfunc(10.1)中，传入的参数是 10.1，由于 C++中所有的浮点类型的常量自动地都是 double 类型的，所以调用的是 myfunc(double)，这里没有二义性。然而代码 myfunc(10)是有二义性的，因为编译器不能确定是调用函数 myfunc(double)还是 myfunc(float)。整型数 10 转换成 double 类型或者 float 类型都是有效的。编译器在编译的时候会报告这种二义性错误，并停止对程序的编译。

从上面的程序可以看出，产生二义性错误的原因并不是对函数 myfunc()的重载，而是由于在调用函数的时候传入了不能确定类型的参数。

下面是另外一个由于 C++中自动类型转换而导致的二义性的程序：

//另外一个二义性程序

view plain

```
#include <iostream>
using namespace std;
char myfunc(unsigned char ch);
char myfunc(char ch);
int main()
{
    cout << myfunc('c');        // 这里调用的是函数 myfunc(char)
    cout << myfunc(88) << " "; // 错误，有二义性
    return 0;
}
char myfunc(unsigned char ch)
{
    return ch-1;
}
char myfunc(char ch)
{
    return ch+1;
}
```

在 C++中，unsigned char 和 char 类型本质上是没有二义性的。它们是两种不相同的类型。然而当调用函数 myfunc()的时候，由于传入的是整型数 88，编译器就不能确定到底应该调

用哪个版本的 myfunc()函数了。也就是说，整型数 88 应该被转换为 unsigned char 还是 char 呢？两种转换都是合法的。

另外一种导致程序二义性的原因可能是使用了缺省函数参数。看看下面的程序：

view plain

```
#include <iostream>
using namespace std;
int myfunc(int i);
int myfunc(int i, int j = 1);
int main()
{
    cout << myfunc(4,5) << " "; //无二义性
    cout << myfunc(10);         //错误！有二义性
    return 0;
}
int myfunc(int i)
{
    return i;
}
```

在第一次调用函数 myfunc()的时候，指明了两个参数，因此是没有二义性的，调用的是函数 myfunc(int,int)。然而在第二次调用 myfunc()的时候就产生了二义性，因为编译器不能确定到底是调用 myfunc(int)还是调用有缺省参数的 myfunc(int,int)。

在继续学习 C++的过程中，我们有可能会遇到二义性错误。不幸的是，在还没有程序专业的 C++程序员之前，我们会发现很容易就会出现二义性的错误。

习题：

1. 传递参数给子程序的两种方式什么？
2. 在 C++中什么是引用？如何创建引用参数？
3. 有如下的代码：

   int f(char &c,int * i);

   //…

   char ch =’x’; int i= 10;

   写出如何调用函数 f()，传入参数 ch 和 i。
4. 编写一个名称为 round 的函数，用来计算与一个 double 类型参数值最接近的整型数。

第七篇 更多数据类型和运算符

本章我们又回到数据类型和运算符的话题上。除了目前我们使用过的那些数据类型之外，C++还支持几种别的数据类型，其中就包括对我们已经学过的数据类型的修饰符，还有包括

枚举和类型定义的别的数据类型。C++中还提供了几种附加的运算符，这些运算符极大地扩展了编程任务的范围。这其中就包括位运算符，移位运算符号和 sizeof 运算符。

必备技能 7.1:const 和 volatile 修饰字

C++中有两种可以影响变量访问或者修修改方式的限定字。它们就是 const 和 volatile。通常，它们被称为 CV 限定字。

const

被 const 修饰的变量在程序执行期间其值是不能被修改的。因此，一个被 const 修饰的“变量”实际上并不是真正意义上的变量。但是我们还是可以给一个被 const 修饰的变量进行初始化的。例如：

const int max_users = 9;

就创建了一个名称为 max_users 的变量，其值为 9。它可以像其他变量一样应用于任何的表达式。但是程序不能修改其值。

const 最常用的就是被用来创建一个有名称的常量。通常，程序中出于不同的目的需要使用同一个值。例如，程序中可以还有多个大小一样的数组。在这种情况下，我们就可以使用一个被 const 修饰的变量来指明数组的大小。这种方式的优点就是如果以后需要修改数组的大小，我们只需要修改这个被 const 修饰的变量的值，然后对程序进行重新编译即可。而不需要在每个数组声明的地方都对其大小进行修改，并且这很容易引入错误。下面的程序演示了 const 的用法。

view plain

```
#include <iostream>
using namespace std;
//创建一个常量,名称为 num_employees,其值为
const int num_employees = 100;
int main()
{
    int empNums[num_employees];
    double salary[num_employees];
    char *names[num_employees];
    //...
    return 0;
}
```

在上面的程序中，如果我们需要为数组指定新的大小，我们是需要修改声明变量 num_employees 的初始化值，然后重新编译即可。三个数组的大小就都会自动地变为新的值。

const 的另外一个重要的用法就是防止通过指针来修改对象。例如，有时候我们需要避免在一个函数中对指针参数指向的对象进行修改。为了达到这个目的，我们只需要把指针参数

声明为 cont 就可以了。这样就可以防止在函数中通过指针修改对象了。也即是说，如果指针参数前面用 const 进行修饰了，那么函数中的语句就不能通过该指针修改对象了。例如下面程序中的函数 negate()返回的是其指针参数指向对象的相反数。这里就用到了 const 修饰指针参数，从而避免了在函数中通过指针修改对象的值。

view plain

```
1. //使用常量指针参数
2. #include <iostream>
3. using namespace std;
4. int negate(const int *val);
5. int main()
6. {
7.     int result;
8.     int v = 10;
9.     result = negate(&v);
10.    cout << v << " negated is " << result;
11.    cout << "\n";
12.    return 0;
13. }
14. int negate(const int *val)
15. {
16.    return -*val;
17. }
```

由于在上面的函数中 val 被声明为是一个常量指针，所以函数 negate()中就不能修改指针 val 指向对象的值。所以程序可以正常编译并运行。然而，如果我们把 nagate()函数修改成下面的形式，则会在编译阶段就报告错误：

view plain

```
1. //This won't work!
2. int negate(const int *val)
3. {
4.     *val = -*val;
5.     return *val;
6. }
```

这里，程序企图修改指针 val 指向的对象的值。这样做是不可以的，因为 val 被声明为是 const 的。

const 修饰符同样可以作用于引用参数，来防止函数修改引用参数的值。例如，下面版本中的 negate()函数是错误的，因为它企图修改 val 引用的变量的值。

view plain

```
1. //This won't work!
2. int negate(const int &val)
3. {
4.     val = -val;  //错误，val 引用对象不能被修改
5.     return val;
6. }
```

volatile

修饰字 volatile 告诉编译器，被其修饰的变量可能会被程序中没有明确指定的方式被修改。例如，在定时中断处理程序中，我们可以传入一个全局变量的地址，每次定时中断后，处理程序就会更新该全局变量的值。在这种情况下，该全局变量的值就会在我们程序没有明确赋值的情况下被修改。变量被外部修改之所以对 C++来说是特别重要的，是因为 C++编译器经常会基于“如果变量没有出现在赋值运算符的左侧，则变量的值就没有改变”这种假设来对表达式进行优化。然而，如果超出了程序范围之外的因素如果导致变量的值被修改，则这种优化就会导致问题。为了避免这种问题的出现，我们必须在声明变量的时候使用 volatile 修饰字。如下：

volatile int current_users;

由于变量被 volatile 所修饰，每次我们在引用 current_users 的时候，它的值都会被重新获取。

练习：

1. 被 const 修饰的变量的值是否在程序中可以被修改？
2. 如果一个变量的值会被程序之外的事件处理所修改，我们应该如何声明这个变量？

存储类型说明字

C++中有 5 种存储类型说明字，它们是：

auto

extern

register

static

mutable

这些说名字高速编译器如何存储变量。存储类型说名字是放置在变量声明的最前面的。其中的 mutable 说名字只能用于类对象，关于这点我们将在本书的后面进行介绍。

本章中，我们只对其它的说名字进行介绍。

auto

auto 说明字用于声明一个局部变量。然而，它很少被用到。这是因为局部变量缺省地就是 auto 的。在程序中我们基本上看不到这个说明字。这个说明字是从 C 语言中继承而来的。

必备技能 7.2：extern

尽管到目前为止我们所编写的程序都是很短小的。然而，实际上，计算机程序要比我们编写的程序大很多。随着程序文件的增大，编译所需要的时间也会随之增大，甚至会达到让人们感到生气的程度。当这样的情况发生的时候，我们就应该把程序拆分中几个响度独立的文件。这样一来，对其中一个文件的修改不至于导致需要对所有文件都进行编译。实际上，我们只需要重新编译那个别修改的文件，然和和其它已经存在的目标文件一起进行链接即可。这种把程序拆分成多个文件的方法针对大型程序来说可以节省很多的时间。关键字 extern 就是可以支持这种方法的。下面我们就自己研究一下 extern。

当一个程序是有两个或者多个文件组成的时候，每个文件都必须知道程序中所使用到的全局变量的名称和类型。然而，我们是不能再每个文件中都对同一个全局变量都尽兴声明的。这是因为程序中只能还有一个同名的全局变量。因此，如果我们在每个文件中都对同一个全局变量进行声明，当链接器对这些文件进行链接的时候就会出现错误。链接器会发现有重复的全局变量，并终止对程序的链接。针对这种情况的处理方式就是在一个文件中队所有的全局变量进行声明，而在其它的文件中使用 extern 来进行声明。如下所示：

文件 1：

view plain

```
int x, y;
chat ch;
int main()
{
    //...
}
void func1()
{
    x = 123;
}
```

文件 2：

view plain

```
extern int x, y;
extern char ch;
void func22()
{
    x = y/10;
}
void func23()
{
    y = 10;
}
```

在文件 1 中声明了变量 x，y 和 ch。在文件 2 中我们只是复制了全局变量的声明，并在前面加上了 extern 关键字。extern 说明字使得一个模块可以感知到这些变量，而不是创建这些变量。换句话说，extern'关键字告知编译器这些全局变量的类型和名称，而不用为它们再次分配存储空间。当链接器把这两个模块链接起来的时候，所以对于外部变量的引用都是可以被正确解析的。

目前我们还没有对变量的声明和变量的定义进行区分，然而在这里两者的区分是至关重要的。变量的声明就是声明变量的类型和名称；变量的定义则是给变量分配存储空间的。在大多数的情况下，变量的声明同时也是变量的定义。然而，通过在变量名称前面加上 extern 说名字的时候，我们就可以声明一个变量而不定义它（引者注：也就是不为它分配存储空间）。

extern 还有一个变种的用法，就是进行链接说明。它是被用作高速编译器如何编译函数的指令。缺省情况下，函数是作为 C++样式的函数进行编译的，但是链接说明字可以让编译器以别的语言的形式来链接函数。通用的链接说明形式如下：

extern "language" 函数原型

其中的 language 就代表预期的语言。例如，下面的语句将是的函数 mycfunc()将会以 C 语言的函数来进行链接：

extern "c" void mycfunc();

所有的 C++编译器都支持 C 和 C++的链接。一些编译器还可能支持 FORTAN,Pascal, 或者 BASIC 的链接。采用下面的链接说明形式，我们一次可以指定多个函数：

extern "language" { 多个函数原型 }

大多数情况下，我们是不需要进行链接说明的。

必备技能 7.3：静态变量

静态变量在其函数或者文件中是永久性的变量。它们和全局变量不同的是，在其在所在的函数或者文件之外，它们是不被感知的。static 对全局变量和局部变量的影响是不一样的，因此我们将分别讨论这两种情况。

静态的局部变量

当 static 修饰局部变量的时候，变量将被分配一个永久性的存储空间，这点和全局变量时一样的。这样一来，静态的局部变量在函数被多次调用之间是能保持其职的。（也就是说，它的值不会在函数返回的时候丢失，这点不像普通的局部变量。)静态的局部变量和全局变量的主要区别就是：静态的局部变量只能在声明他的代码块中被感知。

声明静态局部变量的方式就是在它类型的前面加上 static 关键字。例如，下面的语句声明变量 count 为一个静态的变量：

static int count;

我们也可以在声明静态变量的时候对其进行初始化。例如，下面的语句就是初始化静态变量

count 为 200：

static int count = 200;

静态变量只会在程序开始执行的时候进行一次初始化，而不是每次在进入他所在的代码块的时候都进行初始化。

对于那些在被多次调用之间必须保持变量值的函数来说，静态局部变量时至关重要的。假设此时我们不使用静态的局部变量，那么我们就只能使用全局变量了，使用全局变量会产生一些副作用的。

下面的程序就是静态局部变量的一个示例。他用来计算用户输入的数值的平均值。

view plain

```
1. //计算用户键入数值的平均值
2. #include <iostream>
3. using namespace std;
4. int running_arg(int i);
5. int main()
6. {
7.  int num;
8.  do
9.  {
10.  cout << "Enter numbers ( -1 to quit): ";
11.  cin >> num;
12.  if ( num != -1 )
13.  {
14.   cout << "Running average is : " << running_arg(num);
15.  }
16.  cout << "\n";
17. }while(num > -1);
18. return 0;
19.}
20.int running_arg(int i)
21.{
22. //由于 sum 和 count 都是静态的局部变量，所以函数
23. //running_arg()被多次调用之间，它们的值是会被保留的。
24. static int sum = 0, count = 0;
25.
26. sum = sum + i;
27. count++;
28. return sum / count;
29.}
```

这里，局部变量 sum 和 count 被声明为静态的，并初始化为 0 值。请注意，静态变量的初始化只会进行一次，而不是每次进入函数中都要进行初始化。函数 running_avg()用来计算

用户当前输入的数值的平均值。由于 sum 和 count 都是静态的变量，它们会在函数被多次调用之间保有其值，因此程序可以正确的工作。为了证明上面的 static 关键字是必要的，我们可以尝试去掉这个关键字，看看程序是否能正确地工作。其结果就是程序不能正常工作了，这是因为每次在函数返回的时候 sum 的都会丢失了。

静态的全局变量

当我们在声明全局变量的时候，如果加了 static 关键字，就是要告诉编译器这个全局变量只能被它所在的文件中的代码所感知。这就意味着，尽管这个变量是全局变量，但是其它文件中的代码是不能感知到这个变量的，更不能修改这个变量的值。这样就消除了全局变量的作用。因此，对于那些极少数的，局部变量不能完成所需功能的情况下，我们就可以创建一个小的文件，其中只包含那些需要使用全局静态变量的函数，对其进行单独的编译后就可使用了，而不用考虑全局变量带来的副作用了。在这里，我们重写了上面的用来计算平均值的程序，作为演示静态全局变量的例子。我们把程序分为两个文件，还增加了函数 reset() 用来对对平均值进行清零。

文件 1:

view plain

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

int running_avg(int num);
void reset();
int main()
{
 int num;
 do
 {
  cout << "Enter numbers ( -1 to quit, -2 to rest ): ";
  cin >> num;
  if ( num == -2)
  {
   reset();
   continue;
  }
  if ( num != -1 )
  {
   cout << "Running average is: " << running_avg(num);
  }
  cout << "\n";
 }while(num != -1);
 return 0;
}
```

文件 2:

view plain

```
1. static int sum = 0, count = 0; //静态的全局变量只有在声明其的文件中才能被感
   知
2. int running_avg(int i )
3. {
4.  sum = sum + i;
5.  count++;
6.  return sum / count ;
7. }
8. void reset()
9. {
10.  sum = 0;
11.  count = 0;
12.}
```

这里，sum 和 count 都是全局的静态变量，它们仅限于文件 2 中。因此它们可以被文件 2 中的 running_arg()和 reset()函数访问，但是不能被其它文件中的代码访问。这也使得通过 reset()函数可以重置它们的值，以便对下一组的数据进行平均值的计算。然而，除了文件 2 之外的函数都是不能访问这两个变量的。例如，当我试图在文件 1 中访问 sum 和 count 这两个变量的时候，编译器就会报告错误。

复习一下：局部的静态变量只能被在它被声明的待模块或者函数中被感知；全局的静态变量也只能被其所在的文件所感知。本质上，static 修饰字使得变量在需要的范围里可以被感知，避免了可能的副作用。static 类型的变量使得我们程序员能够把程序的一部分对其它部分隐藏起来。这对于大型程序的管理来说是一个很大的帮助。

问专家：

问：我听说有些 C++程序员不使用静态的全局变量，是这样吗？

答：尽管静态的全局变量目前是有用的，并且也在 C++代码中存在着广泛的应用，但是 C++标准并不鼓励这样做。它们推荐使用另外的一种用于对全局变量进行访问权限控制的机制：命名空间。本书的后续章节会对这点进行介绍。然而，C 程序员会大量使用这样的静态全局变量，这是因为 C 语言不支持命名空间。正是由于这些原因，我们还是会在很长的一段时间内看到静态全局变量的出现。

必备技能 7.4：寄存器变量

也许最常用的存储类型描述字就是 register（寄存器）了。regitster 描述字告诉编译器应该以尽可能快的访问方式来存储该变量。通常这意味着这个变量要么是存放在 CPU 的寄存器中要么是存放在缓存中。或许我们都知道，访问 CPU 中寄存器或者是缓存的速度要比访问内存的速度快很多。因此，访问寄存器中的变量的速度也要比访问随机存储器中变量的速

度快很多。由于访问变量的速度和整个程序的执行速度有着密切的关系，慎重地选择使用寄存器变量是一项很重要的编程技巧。

从技术上来说，register 描述字仅仅是告诉编译器需要把变量存储在寄存器中，但是编译器可以忽略这种要求。这样做的原因也是很容易理解的：寄存器的数量是有限的，而且寄存器的数量也和硬件的相关，因此，如果计算机的快速存储资源已经用完了，则变量就会被以正常的方式来存储。通常情况下，这样做是不会有什么问题的，当然此时使用寄存器存储方式的优点也就没有了。通常我们可以对至少两个变量来进行优化以提高速度。既然，能够用于快速访问的变量的数量是有限的，谨慎选择哪些变量应该用 regitster 来修饰就变得非常重要了。（只有正确地选择了用 register 修饰的变量，才能达到提高速度的目的）。一般情况下，一个变量被访问的频率越高，把这个变量设置为寄存器变量带来的速度优化效果也就越明显。正是基于这个原因，用于控制循环的变量或者是在循环中被访问的变量通常都是可以考虑被设置为寄存器变量的。

下面的示例程序中就使用了寄存器变量来提高函数 summation()的效率。该函数用于计算一个数组中所有数值的和。在下面的程序中，我们假设只需要对两个变量进行速度的优化即可。

view plain

```
1. //演示 register 修饰字
2. #include <iostream>
3. using namespace std;
4. int summation(int nums[], int n);
5. int main()
6. {
7.     int vals[] = { 1, 2, 3, 4, 5 };
8.     int result;
9.     result = summation(vals, 5);
10.    cout << "Summation is " << result;
11.    return 0;
12.}
13.//返回整型数组中数值的和
14.int summation(int nums[],int n )
15.{
16.    register int i;
17.    register int sum = 0;
18.    for( i = 0; i < n; i++ )
19.    {
20.        sum = sum + nums[i];
21.    }
22.    return sum;
23.}
```

其中，用于循环控制的变量 i 和用于在循环中访问的变量 sum 都被指定为寄存器变量。由于两者都是在循环中使用的变量，所以把它们限定为寄存器变量就可以达到预期的提高速度的目的。在这个程序中，我们假设了只能有两个变量被优化成是寄存器变量，因此我们只是对 i 和 sum 进行 register 的限定，而没有对 n 和 nums 进行寄存器变量的限定。然而，在允许更多变量可以被优化的情况下，我们也可以把 n 和 nums 变量设置成是寄存器变量来进一步提高程序的性能。

练习：

1. 一个静态的局部变量在函数被多次调用之间____其值。
2. 我们使用 extern 来声明一个可以不用定义的变量，对吗？
3. 那个描述字要求编译器对变量的存储进行优化以提高程序的执行速度？

问专家：

问：我在程序中的变量前增加了 register 描述字，但是我没有看到明显的程序执行速度的提高，这是为什么？

答：由于编译技术的不断进步，大部分的编译器都会自动对程序进行优化。因此，在很多情况下，在变量声明的前面增加 register 描述字是不会提高程序执行速度的，这是因为这些变量在没有增加 register 之前已经被优化了。然而，在某些情况下，使用 register 描述字还是有用的，因为这样实际上是告诉编译器我们认为对这些变量的优化是很重要的。特别是在函数中使用了大量变量的情况下，这样做是非常重要的，因为并不是所有的变量都会被优化。因此，尽管编译技术在不断提高，register 描述字还是有着重要的作用。

基本技能 7.5：枚举

在 C++中我们可以定义一组命名的整型常量列表。这样的列表称之为枚举。这些常量可以出现在任何整型数可以出现的地方。枚举的定义是通过使用关键字 enum 来完成的，其通用的形式如下：

enum 类型名称 { 值列表 }变量列表;

上面的类型名称就是枚举的类型名称了。其中的值列表是用逗号间隔开的代表枚举值的名称。其中的变量列表是可选项，这是因为我们可以使用上面的类型名称来在后续进行变量的声明。

下面的代码段就定义了一个名称为 transport 的枚举类型，还定义了两个 transport 类型的变量叫做 t1 和 t2：

```
enum transport { car, truck, airplane, train, boat } t1,t2;
```

一旦我们定义了一个枚举类型，我们就可以使用它的类型名称来声明变量了。例如，下面的语句就声明了一个名称为 how 的 transport 枚举类型的变量：

```
transport how;
```

上面的语句还可以写成：

enum transport how;

然而这里使用 enum 是多余的。由于在 C 语言中上面的第二种形式是必须的，所以我们在一些程序中还是可以看到这样的形式的。

基于上面的声明的变量，下面的语句给变量 how 赋值为 airplane：

how=airplane;

理解枚举的关键点就是枚举中的每一个标识符都代表的是一个整型值。因此，它们可以出现在任何的整型表达式中。在枚举没有进行初始化的情况下，枚举中的第一个标识符的值是 0，第二个标识符的值是 1，以此类推。因此

cout << car <<' ' << train;

将输出 0 3.

尽管枚举常量会被自动地转换为整型数，而整型数是不能自动地转换为枚举常量的。例如，下面的语句是错误的：

how = 1; //Error

上面的语句会导致编译时的错误，这是因为不存在从整型数到 transport 枚举类型的自动转换。但是可以通过强制转换的方式来完成这种转换，例如：

how = (transport)1; // 这种方式是符合语法的，但是这是一种很不好的编程风格

上面的代码将给变量 how 复制为 truck，这是因为 transport 枚举中 1 的是和 truck 相关的。正如上面代码中的注释的那样，尽管这种写法是正确的，但是在非特殊的环境下，这种写法是很不好的。

我们还可以通过初始化的方式来制定枚举类型中常量的值。这是通过在常量标识符的后面跟上等号和一个整型数来完成的。当我们使用初始化的时候，被初始化之后的常量的值就是前面常量的值加一。例如，下面的语句给 airplane 初始化为 10：

enum tranport { car, truck, airplane = 10, train, boat };

这样一来，枚举 transport 中标识符的值如下：

car	0
truck	1
airplane	10
train	11
boat	12

一个常见的错误就是企图把枚举中的标识符作为字符串来进行输入或者输出。实际并不是这样的。例如，下面的程序不会像期望的那样：

//下面的代码并不会输出 trian

how = train;

cout << how;

请记住标识符 train 只是一个整型数的名称，它不是一个字符串。因此，上面的代码输出的将是 train 标识符的值，而不是字符串“train”。实际上，要把枚举中的标识符作为字符串使出或者输入是一个很繁琐的过程。例如，下面的代码就是用于输出标识符字符串的，也就是说输出 how 中的交通工具的类型：

view plain

```
switch(how)
{
    case car:
     cout << "Automobile";
     break;
    case truck:
     cout << " Truck";
     break;
    case airplane:
     cout << "Airplane";
     break;
    case train:
     cout << "Train";
     break;
    case boat:
     cout << "Boat";
     break;
}
```

有时，我们可以定义一组字符串的数组，并使用枚举值作为字符串数组的所引来获取枚举值对应的字符串。例如，下面的程序就打印出三种交通工具的类型；

view plain

```
//演示枚举的使用
#include <iostream>
using namespace std;
enum transport
{
    car,
    truck,
    airplane,
    train,
    boat
};
char name[][20]=
```

```
{
    "Automobile",
    "Truck",
    "Airplan",
    "Train",
    "Boat"
};
int main()
{
    transport how;
    how = car;
    cout << name[how] << "\n";
    how = airplane;
    cout << name[how] << "\n";
    how = train;
    cout << name[how] << "\n";
    return 0;
}
```

程序的输出如下：

Automobile

Airplan

Train

上述陈述中的把枚举值转换成字符串的方法可以适用于一切没有使用初始化的枚举类型。为了能正确地对字符串数组进行所以，枚举常量的值必须是冲 0 开始，并且保持严格的升序排列，后一个都是比前面的一个大 1。正是由于枚举值到字符串的转换是需要人工完成的，所以枚举类型通常只用于不需要这种转换的情况。例如，我们常见到的就是使用枚举来定义编译器的符号表。

必备技能 7.6： 类型定义

在 C++中我们可以使用 typedef 关键字来定义新的类型名。当我们使用 typedef 的时候，我们并不是创建了一种新的数据类型，而是为已经存在的类型定义了一个新的名称。这有助于使提高代码的可移植性；我们只需要修改 typedef 语句就可以了。同样，对标准的数据类型取一个具有描述性的名称可以提高代码的可阅读性。typedef 语句的通用形式如下：

typedef 类型 名称;

其中的类型可以是任意有效的数据类型，其中的名称就是这个类型的新的名称。新的名称是对原来类型名称的补充，而不是替换了原来的类型名称。

例如，我们可以为 float 类型创建一个新的名称：

typedef float balance；

上面的语句告诉编译器把 balance 看做是 float 的另外一个名称。接下来我们就可以使用 balance 来创建一个 float 的变量了：

balance over_due;

在这里，over_due 就是一个浮点型变量，类型为 balance，这只是浮点数的另外一个名称而已。

练习：

1. 枚举是一组___类型常量的列表。
2. 枚举值是从什么值开始的？
3. 如何为 long int 类型定义另外的一个名称为 BigInt？

必备技能 7.7：位运算符

C++语言的设计考虑到了要支持对计算机硬件的访问，其中很重要的一点就是直接操作字节或者字中的比特位。C++中的位运算符就能实现这一点。位运算指的是对 C++字符类型和整型类型对应的字节或者字中的位进行检测，设置或者移位的操作。位运算符不能使用于 bool，float，double，long double，void 等其它复杂的数据类型。在很多系统级别的编程中，位运算是很重要的，特别是在需要查询或者设置设备状态的时候。表格 7-1 列出了位运算符，其中对每一个为运算符都进行解释.

位运算符	含义
&	位与
|	位或
^	位异或
~	非
>>	右移位
<<	左移位

表格 7-1 位运算符

位与，位或，位异或和非

位与，位或和非运算与其对应的逻辑运算的真值表是相同的，不同之处在于它们是逐位进行运算的。位异或运算符的真值表如下：

p	q	p^q
0	0	0
1	0	1
1	1	0
0	1	1

正如上面的真值表所示的那样，异或运算的结果只有在两个运算数中只有一个为 1 的时候，其结果才为 1，否则运算结果就都是 0。

位与最常用的就是用来算来把一个数中的某些位置 0。这是由于只要两个运算数中对应比特位中只要有一个数为 0，其运算结果就是 0。如下：

```
 1 1 0 1 0 0 1 1
&1 0 1 0 1 0 1 0
----------------------
 1 0 0 0 0 0 1 0
```

下面的程序通过把第六比特位的值设置为 0 来实现从小写字母到大写字母的转换。正如

在 ASCII 字符集中定义的那样，小写字母的值比大写字母的值大 32。因此，把小写字母转换为大写字母的时候只需要关闭第六个比特位即可，正如程序演示的那样：

view plain

```
1.  //使用位与运算来实现小写字母到大写字母的转换
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      char ch;
7.
8.      for ( int i = 0; i < 10; i++ )
9.      {
10.         ch = 'a' + i;
11.         cout << ch;
12.         //下面的这条语句就是把第六比特位关闭，也就是设置为.
13.         ch = ch & 223; // ch 就变成大写的字母了
14.         cout << ch << " ";
15.     }
16.     cout << "\n";
17.     return 0;
18. }
```

程序的输出如下：

aA bB cC dD eE fF gG hH iI jJ

其中在使用位与运算符的语句中用到了值 223，其二进制的形式为 1101 1111。这样一来，位与运算的结果就会使得第六比特位的值变为 0，而其它比特位的值都保持不变。

当我们需要判断某个比特位是打开还是关闭的时候，使用位与运算符此时也是非常有效地。例如，下面的语句就可以检查第四个比特位是否被打开：

if (status & 8) cout << " bit 4 is on";

这里使用到了数字 8，这是因为它的二进制形式为 0000 1000。也就是说 8 的二进制中只有第四位的值是 1，其它都是 0。因此，if 语句中的表达式只有在 status 的第四比特位也是 1 的时候才为真。这个特性的一个有趣的应用就是下面的 show_binary()函数。它以二进制的形式输出参数的值。本章节的最后，我们会用到这个函数的输出来检查其它位运算符的结果。

view plain

```
1.  //显示一个字节中的比特位的值
2.  void show_binary(unsigned int u )
3.  {
4.      int t;
5.
```

```
    for ( t = 128; t > 0; t = t/2 )
    {
        if ( u & t ) cout << "1 ";
        else cout << "0 ";
    }
    cout << "\n";
}
```

函数 show_binary()通过使用位与运算符连续地从高位到低位逐个检测每个比特位的值。如果该位的值是打开的，则输出 1，否则输出 0。

位或运算和位与恰好相反，它可以用来打开比特位。任意一个运算数中的 1 值都可以使得运算结果中对应的比特位的值为 1。例如，

1 1 0 1 0 0 1 1

| 1 0 1 0 1 0 1 0

---

1 1 1 1 1 0 1 1

我们可以使用位或运算来把大写字母转换为小写字母，如下：

view plain

```
//使用位或运算来实现大写字母到小写字母的转换
#include <iostream>
using namespace std;
int main()
{
    char ch;

    for ( int i = 0; i < 10; i++ )
    {
        ch = 'A' + i;
        cout << ch;
        //下面的这条语句将第六比特位打开，也就是设置为
        ch = ch | 32; // ch 就变成小写的字母了
        cout << ch << " ";
    }
    cout << "\n";
    return 0;
}
```

程序的输出如下：

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

当第六比特位被设置为 1 的时候，每一个大写字母就被转换成了对应的小写字母了。

位运算中的异或，也常被缩写成 XOR。异或只有在两个运算数对应比特位上的值不相同的时候，其结果对应位才是 1，其它情况，结果对应比特位都是 0。如下：

```
  0 1 1 1 1 1 1 1
^ 1 0 1 1 1 0 0 1
-----------------
  1 1 0 0 0 1 1 0
```

XOR 有一个很有意思的特性：当一个值 X 和另外一个值 Y 进行 XOR 运算后，其结果再次和 Y 做 XOR 运算就可以得到 X 的值。我们可以利用这一点来对消息进行编码。也就是如果:

```
R1 = X ^ Y;
R2 = R1 ^ Y;
```

那么 R2 的值和 X 的值是一样的。也就是连续两次使用 XOR 与同一个数进行运算将得到最初的那个运算数。我们可以使用这个特性来创建一个简单的加密程序：其中使用一个整型数作为关键字来通过 XOR 运算对字符消息进行加密和解密。加密的时候第一次使用 XOR 运算来产密文；解密时再次使用 XOR 运算产生明文。下面的程序就使用上述方法来完成加密和解密的：

view plain

```
1.  //使用 XOR 来对消息进行加密和解密
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      char msg[] = "This is a test";
7.      char key = 88;
8.      cout << "Original message: " << msg << "\n";
9.      for ( int i = 0; i < strlen(msg); i++)
10.     {
11.         msg [i] = msg[ i ] ^ key ;  //生成密文
12.     }
13.     cout << "Encoded message: " << msg << "\n";
14.     for ( int i = 0; i < strlen(msg); i++)
15.     {
16.         msg [i] = msg[ i ] ^ key ;  //生成明文
17.     }
18.     cout << "Decoded message: " << msg << "\n";
19.     return 0;
20. }
```

程序的输出如下：

Original message: This is a test

Encoded message: 01+x1+x9x,=+,

Decoded message: This is a test

正如上面的程序展示的那样，使用相同的 key 值进行两次 XOR 运算得到的就是解密后的消息。

单目运算符非则是对操作数中的所有比特位取反。例如，假设 A 的二进制位 1001 0110，那么~A 的结果就是 0110 1001。下面的程序通过显示一个数字及其它的逐位取反的结果来展示非运算，其中使用到了前面的 show_binary()函数。

view plain

```
#include <iostream>
using namespace std;
void show_binary(unsigned int u);
int main()
{
    unsigned u;
    cout << "Enter a number between 0 and 255:  ";
    cin >> u;
    cout << "Here's the number in binary: ";
    show_binary(u);
    cout << "Here is the complement of the number: ";
    show_binary(~u);
    return 0;
}
//显示一个字节中的比特位的值
void show_binary(unsigned int u )
{
    int t;

    for ( t = 128; t > 0; t = t/2 )
    {
        if ( u & t ) cout << "1 ";
        else cout << "0 ";
    }
    cout << "\n";
}
```

程序输出结果如下：

Enter a number between 0 and 255 99

Here's the number in binary: 0 1 1 0 0 0 1 1

Here is the complement of the number: 1 0 0 1 1 1 0 0

由于&，|，^，~是直接作用于运算数的每一个比特位上的。因此，位运算符通常不像关系和逻辑运算符那样应用于条件表达式中。例如，如果 x 等于 7，那么 x&&8 的结果是 true，然而 x&8 的结果则是 false。

必备技能 7.8 移位运算符

移位运算符>>和<<把变量的比特位按照指定的数量进行右移和左移。右移运算符的通用形式如下：

变量 >> 移动的位数

左移运算符的通用形式如下：

变量 << 移动的位数

其中的移动的位数表示的是需要向右或者向左移动的比特位的次数。每进行一次左移，就会把变量的所有比特位向左边移动一个比特位，并在最右边的比特位上填充 0。每进行一次右移，就会把变量的所有比特位向右边移动一个比特位，并在最左边的比特位上填充 0。然而，如果变量是一个有符号的整型数，并且是个负数，那么每右移一次就会在最左端填充 1，这样可以在移位的过程中保留数值的符号。另外，移位运算不是循环移位的。也就是说被移出去的那一位不会回复到另一端的开始的。

移位运算符只能用于整型类型，例如 int,char,long int,short int。它们不能应用于浮点类型。

移位运算符在对诸如数/模转换器之类外部输入设备的输入进行解码或者是读取状态信息的时候是非常有用的。我们还可以使用移位运算来进行快速的乘法或者除法运算。左移移位相当于是对这个数乘以 2，右移一位相当于是对这个数除以 2。

下面的程序演示了一位运算的效果：

view plain

```
1.  // 演示移位
2.  #include <iostream>
3.  using namespace std;
4.  void show_binary(unsigned int u );
5.  int main()
6.  {
7.      int i = 1, t;
8.
9.      //左移
10.     for ( t = 0; t < 8; t++)
11.     {
12.         show_binary(i);
13.         i = i << 1;
14.     }
15.     cout << "/n";
```

```
16.     //右移
17.     for ( t= 0; t< 8; t++)
18.     {
19.         i = i >> 1;
20.         show_binary(i);
21.     }
22.     return 0;
23. }
24. //显示一个字节中的比特位的值
25. void show_binary(unsigned int u )
26. {
27.     int t;
28.
29.     for ( t = 128; t > 0; t = t/2 )
30.     {
31.         if ( u & t ) cout << "1 ";
32.         else cout << "0 ";
33.     }
34.     cout << "\n";
35. }
```

程序的输出如下：

0 0 0 0 0 0 0 1

0 0 0 0 0 0 1 0

0 0 0 0 0 1 0 0

0 0 0 0 1 0 0 0

0 0 0 1 0 0 0 0

0 0 1 0 0 0 0 0

0 1 0 0 0 0 0 0

1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0

0 0 1 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 1 0 0 0

0 0 0 0 0 1 0 0

0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 1

练习：

1。 位与，位或，非和异或的位运算符是什么？

2。 位运算符是基于比特位进行运算的，对吗？

3。 有一个整型数 x，写出如何把 x 左移两个比特位？

必备技能 7.9：？运算符

C++中提供了一个非常令人兴奋的运算符，它就是？运算符。这个运算符通常用来替换如下格式的 if-else 语句：

if (条件) 变量 = 表达式 1; else 变量 = 表达式 2;

这里，变量的取值取决于 if 中的条件表达式的值。

这里的？运算符被称为是三目运算符，这是因为它需要三个运算数。它的通用形式如下：

表达式 1 ？表达式 2 : 表达式 3;

也就是说需要三个表达式来参与运算。请注意其中分号的用法。

?表达式的值是这样计算的：首先对表达式 1 的值进行判断。如果其值为 true，则计算表达式 2 的值，并且表达式 2 的值就是整个?表达式的值；如果表达式 1 的值为 false，则计算表达式 3 的值，并且表达式 3 的值就是整个表达式的值。例如下面的表达式就是给变量 absval 赋值为 val 的绝对值：

absval = val < 0 ? -val : val;

这里，如果 val 的值是 0 或者是大于 0 的，absval 就会被赋值为 val 的值。如果 val 的值是负数，那么 absval 就会被赋值为其的相反数。对应的 if-else 语句如下：

if (val < 0) absval = -val; else absval = val;

下面是另外一个?运算符的例子。下面的程序完成两个数字相除，但是除数不能为零。

view plain

```
1. /* 下面的程序使用了?运算符来避免除数为零*/
2. #include <iostream>
3. using namespace std;
4. int div_zero();
5. int main()
6. {
7.     int i, j, result;
8.
9.     cout << "Enter dividend and divisor:";
10.    cin >> i >> j;
11.
12.    //下面的语句中使用?运算符来避免除数为零
13.    result = j ? i/j : div_zero();
14.    cout << "Result: " << result;
15.
16.    return 0 ;
17. }
```

```
18.
19. int div_zero()
20. {
21.     cout << "Cannot divide by zero.\n";
22.     return 0;
23. }
```

在上面的程序中，如果 j 是非零的数，那么就会进行 i/j 的运算，并把结果赋值给 result。否则，错误处理函数 div_zero()就会被调用，并给 result 赋值为 0。

必备技能 7.10：逗号运算符

另外一个很有意思的运算符就是逗号运算符。我们在 for 循环中已经看到过几个关于逗号运算符的例子了。当时我们是使用逗号运算符来进行多个变量的初始化或者多个自增语句的。然而，逗号运算符是可以作为任何表达式的一部分的。它用于把多个表达连接起来。用逗号进行间隔的表达式列表的值就是其中最右边的表达式的值，其它表达式的值都会被丢弃。这就意味着最右边的表达式的值就是整个逗号间隔的表达式的值，例如：

var = (cout = 19, incr = 10, count+1);

上面的语句中先是给 cout 赋值为 19，然后给 incr 赋值为 10，然后拿 count 的值加上 1，最后把整个逗号表达式的值赋值给变量 var，也就是把值 20 赋值给变量 var。其中的括号是有必要的，因为逗号运算符的优先级别是低于赋值运算符的。

运行一下下面的程序，逗号运算符的作用就更加清楚了：

view plain

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.      int i, j;
6.      j = 10;
7.      i = ( j++, j+100, 999+j );
8.      cout << i;
9.      return 0;
10. }
```

上面的程序将在屏幕上打印"1010"。这是因为：j 一开始的值为 10；然后 j 自增到 11；然后再把 j 和 100 相加；最后把 j（j 的值仍为 11）和 999 相加；这样最终的结果就是 1010。

实际上，逗号的作用是进行一系列的运算。当在赋值语句的右侧出现逗号表达式的时候，就是把这一些列运算中的最后一个表达式的值赋值给左侧的变量。我们可以把逗号运算符看做是英语中的单词"and"，那么逗号表达式的含义就是：“做这个，并且做这个，并且再做这

个”

练习

1. 考虑如下的表达式：

x = 10 > 11 ? 1: 0;

计算完毕后 x 的值是多少？

2.?运算符被称为是三目运算符是因为它需要_____个运算数。

3.逗号是用来做什么的？

多重赋值

C++中提供了一种给多个变量赋相同值的方法：在一个语句中使用多重赋值。例如，下面的代码段给 count,incr 和 index 都赋值为 10

count = incr = index = 10;

在专业的程序中，我们会经常看到这样的给多个变量赋值为同一常量值的写法。

必备技能 7.11：复合赋值

C++提供了一种特殊的复合赋值运算符。它能够简化特定类型的赋值语句。例如：

x = x + 10;

采用复合赋值运算符可以写为：

x += 10;

其中的运算符对+=告诉编译器给变量 x 赋值为 x 的值加上 10。复合赋值运算符对应于 C++中的所有二目运算符。其通用形式如下：

变量 运算符= 表达式；

下面是另外的例子：

x = x -100;

和

x -= 100 ;

是等效的。

由于在书写上比较简单，所以有时候复合赋值运算符也被称为是赋值运算符的缩写。在专业的 C++程序中，这样的缩写是很常见的，所以我们必须熟悉这种写法。

必备技能 7.12：使用 sizeof

在实际编程中，有时需要知道某种类型数据占用的空间的大小。由于 C++中内置数据类型的大小是与计算环境相关的，所以我们不可能提前知道所有情况下变量占用空间的大小。为了解决这个问题，C++提供了一个编译时的运算符 sizeof。它的通用形式如下：

sizeof(类型);

sizeof 变量 ;

第一种形式返回的是指定数据类型占用空间的大小；第二种形式返回的是指定变量占用空间的大小，单位都是字节。从上面的形式我们可以看出，如果需要知道指定数据类型的大小，则需要在 sizeof 后面使用括号把数据类型扩起来；如果想要知道一个变量占用的空间

大小，则可以不用括号。当然，使用括号也是没有错误的。

下面的程序演示 sizeof 的用法。在 32 位环境下，程序的输出为 1,4,4 和 8。

view plain

```
1.  //演示 sizeof 的用法
2.  #include <iostream>
3.  using namespace std;
4.  int main()
5.  {
6.      char ch;
7.      int i;
8.      cout << sizeof ch << ' ';
9.      cout << sizeof i << ' ';
10.     cout << sizeof(float) << ' ';
11.     cout << sizeof(double) << ' ';
12.     return 0;
13. }
```

sizeof 可以应用于任何的数据类型。例如，当 sizeof 作用于一个数组的时候，返回的是数组占用的空间的字节数。如下面的代码段：

```
int array[4];
cout << sizeof(array) << ' ';
```

假设 int 类型的大小为 4 个字节，那么上面的代码段将输出 16。也就是 4 个元素，每个元素占用 4 个字节，一共占用 16 个字节的空间。

正如我们在前面提到的那样，sizeof 是一个编译时的运算符。所有的用于计算变量或者数据类型大小的必要信息在编译时都是可知的。sizeof 运算符主要是用来帮助我们生成可移植的代码。请记住：既然 C++中定义的类型的空间大小取决于它们的实现，所以在编程的时候程序员假设它们的大小为某个值就是一种不要的编程习惯了。

练习：

1. 写出如何使用一个赋值语句来给变量 t1,t2,t3 赋值为 10。
2. 如何重写下面的代码：

```
x = x + 100;
```

3. sizeof 运算符在_____时返回变量或者类型的空间大小。

小结

表格 7-2 列出了 C++中运算符的优先级，顺序为从高到低。大多数运算符的结合性都是从左至右的。单目运算符，赋值运算符和？运算符的结合性是从右向左的。下面的表格中列出的部分运算符是我们目前还没有学习到的，其中许多都是在面向对象的编程中用到的。

优先级	运算符

最高优先级	() 组运算 [] 访问数组 -> 通过指针访问成员 :: 范围运算符 . 对象的成员
	! 逻辑非 ~ 逐位取反 ++ 前缀自增 -- 前缀自减 - 负号 * 去引用 & 取地址 sizeof 返回字节大小 new delete typeid type-casts
	. * -> *
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
	= += -= *= /=

	%= >>= <<= &= ^= |=
	,

表 7-2 C++中运算符的优先级

本章练习

1. 如何声明一个 int 类型的变量，其名称为 test，它的初始化值为 100，程序不能修改其值。

2. 限定字 volatile 告诉编译器一个变量有可能被该程序之外的因素所修改，对吗？

3. 在含有多个文件的工程中，我们可以用什么样的修饰字来使得一个文件中可以感知到在别的文件中声明的变量？

4. 静态的局部变量有什么重要的特性？

5. 编写一个程序，其中有一个函数叫做 counter()，用来对函数被调用的次数进行统计，并返回该次数。

6.考虑下面的代码段，其中那个变量最好被限定为是寄存器变量？

view plain

```
int myfunc()
    {
          int x;
          int y;
          int z;
          z = 10;
          y = 0;

          for ( x = z; x< 15; x++ )
          {
             y += x;
          }
          return y;
    }
```

7. 运算符&和&&的区别是什么？

8. 下面的语句是什么意思？

x *=10；

9. 扩展函数 show_binary()，使其能显示出一个 unsigned int 类型数据的全部比特位，而不仅仅是显示出高 8 位。

第一篇 类和对象

到目前为止，我们所编写的程序都没有使用到任何 C++的面向对象的能力。也就是说，前面写的程序都是结构化的程序，不是面向对象的程序。编写面向对象的程序就需要用到类。类是 C++封装的基本单位。类是用来创建对象的。类和对象是 C++的基础。本书的后面篇章将全部用来讨论类和对象。

类的基础知识

我们先从术语类和对象开始学习。类是一种定义了对象通用形式的模板。类中明确定义了数据和代码。C++中使用类的详细说明来创建对象的。对象是类的实例。因此，类实际上是创建对象的工厂。有一点必须明确：类是逻辑上的抽象。只有当类的对象被创建了，内存中才会有该类的一个物理表示。

当定义类的时候，需要声明类中含有的数据以及作用于这些数据上的代码。一些简单的类可以只含有数据或者代码，但实际应用中的类通常都是既含有数据又含有函数代码的。

必备技能 8.1：类的通用形式

类的创建是通过使用关键字 class 来完成的。用于声明类的通用形式如下：

```
class 类名称
{
    私有数据和函数
public：
    共有数据和函数
}对象列表；
```

其中的类名称就是这个类的名称。这个名称将变成一种新的类型的名称，可用于创建类的对象。我们也可以在类声明后面的对象列表中直接创建类的对象，这是可选的。一旦声明了一个类以后，我们就可以在需要的时候创建这个类的对象了。

类中可以含有私有的或者是共有的成员。缺省情况下，类中的成员都是私有的。这就意味着，这些成员只能被类的其它成员访问，而程序的别的部分是不能访问这些成员的。这是一种获得封装的方式——通过私有成员的方式可以严格控制对某些特性项的访问。

为了使得类的成员是公有的（也就是说，程序的其它部分可以访问它），我们必须把这些程序声明在 public 关键字之后。所有声明在 public 关键字之后的变量或者函数都是程序其它部分可以访问的。典型的情况就是通过共有的函数来访问私有的成员。注意，public 关键字之后必须跟一个分号。

尽管句法上没有规定，但是一个设计得体的类仅定义一个逻辑实体。例如，一个用于存储姓名和电话号码的类通常是不需要存储股票行情，平均降雨量，太远黑子周期或者别的不相关的信息的。这里想要说的是：设计得当的类是把逻辑上相关的信息归为一组的。把其它不相关的信息也放入的这个类中将会使得你的程序结构非常混乱。

让我们复习一下：类创建了一种新的数据类型，我们可以用它来创建对象。

另外，类创建了用于定义其成员间关系的逻辑框架。当我们声明一个类的变量的时候，我们创建的就是一个类的对象。对象是有实际物理空间的，它是类的实例。也就是说，对象是占用存储空间的，而类型定义是不占用存储空间的。

必备技能 8.2：定义类并创建对象

我们将通过编写一个来学习类的用法。该类封装了与车辆相关的信息，例如，轿车、货车和汽车。这个类名称为 Vehicle，它将包含和车辆相关的三个信息：载客量，可承载的油量以及每加仑油能跑的英里数。

这个类的最初版本如下。其中定义了三个实例变量：passengers，fuelcap 和 mpg。请注意，其中没有任何的函数。所以，它是一个只含有数据的类。

view plain

```
1. class Vehicle
2. {
3. public:
4.     int passengers;   //可以乘坐乘客的人数
5.     int fuelcap;      //可以承载的油量，加仑
6.     int mpg;          //每加仑油能跑的英里数
7. };
```

它给我们展示了声明实例变量的方式。声明实例变量的通用形式如下：

类型 变量名称；

其中的类型指明了变量的类型；变量名称就是变量的名字。可见，实例变量的声明和其它变量的声明是一样。针对 Vehicle 这个类来说，它的实例变量前面有 public 关键字。正如我们在前面解释过的那样，这就使得我们在 Vehicle 类之外也是可以访问这三个实例变量的。

一个类的定义就创建了一种新的数据类型。针对上面的类，这个新的数据类型就是 Vehicle。我们可以使用这个名称来创建 Vehicle 类型的对象。因此，每个 Vehicle 类型的对象都将含有三个实例变量的副本：passangers，fuelcap，mpg。访问这些实例变量的时候我们需要使用点号(.)运算符。点号运算符把对象的名称和成员的名称连接起来。点号运算符的通用形似如下：

对象.成员

因此，对象是在点号的左侧的，成员是在点号的右侧的。例如，给对象 minivan 的 fuelcap 成员赋值为 16，代码如下：

view plain

```
1. minivan.fuelcap=16;
```

一般情况下，我们可以使用点号运算符来访问实例变量和函数。下面是使用 Vehicle 类的完整程序：

view plain

```
1. #include <iostream>
2. using namespace std;
3.
4. class Vehicle
5. {
6. public:
7.     int passengers; //可以乘坐乘客的人数
8.      int fuelcap;     //可以承载的油量，加仑
9.      int mpg;         //每加仑油能跑的英里数
10.};
11.
12.
13.
14.int main()
15.{
16.    Vehicle minivan; //创建一个 Vehicle 类型的对象
17.     int range;
18.
19.
20.
21.    //给 minivan 的字段赋值,注意我们使用点号来访问对象的成员
22.     minivan.passengers = 7;
23.    minivan.fuelcap = 16;
24.    minivan.mpg = 21;
25.
26.    range = minivan.fuelcap * minivan.mpg;
27.
28.    cout << "Minivan can carry " << minivan.passengers <<
29.        " with a range of " << range << "\n";
30.
31.
32.
```

```
33.     return 0;
34. }
```

运行上面的程序，输出结果如下：

Minivan can carry 7 with a range of 336

在我们继续学习之前，我们复习一个重要的原则：每一个对象是含有一份类中定义的实例变量的拷贝。因而，一个对象的变量的内容是可以和另外的对象的变量的内容不一样的。除了是相同类型的对象之外，这两个对象之间没有任何别的联系。例如，有两 Vehicle 的对象，每一个都含有 passengers，fuelcap 和 mpg 的拷贝，它们的值也可以完全不一样。下面的程序就演示了这种情况：

view plain

```
1. #include <iostream>
2. using namespace std;
3.
4. class Vehicle
5. {
6. public:
7.     int passengers; //可以乘坐乘客的人数
8.     int fuelcap;    //可以承载的油量，加仑
9.     int mpg;        //每加仑油能跑的英里数
10. };
11.
12.
13.
14. int main()
15. {
16.     //创建两个对象，每个对象都含有自己的实例变量的拷贝
17.     Vehicle minivan; //创建一个 Vehicle 类型的对象
18.     Vehicle sportcar; //创建了另一个对象
19.
20.     int range1, range2;
21.
22.
23.
24.     //给 minivan 的字段赋值
25.     minivan.passengers = 7;
26.     minivan.fuelcap = 16;
27.     minivan.mpg = 21;
28.
29.
30.
```

```
31.     //给 sportcar 的各个字段赋值
32.     sportcar.passengers = 2;
33.     sportcar.fuelcap = 14;
34.     sportcar.mpg = 12;
35.
36.
37.     range1 = minivan.fuelcap * minivan.mpg;
38.     range2 = sportcar.fuelcap * sportcar.mpg;
39.
40.     cout << "Minivan can carry " << minivan.passengers <<
41.         " with a range of " << range1 << "\n";
42.
43.     cout << "Sportcar can carry " << sportcar.passengers <<
44.         " with a range of " << range2 << "\n";
45.
46.
47.
48.     return 0;
49. }
```

程序的输出如下：

Minivan can carry 7 with a range of 336

Sportcar can carry 2 with a range of 168

正如我们看到的那样，minivan 的数据和 sportcar 的数据时完全分开的。图 8-1 描述了这种情况：

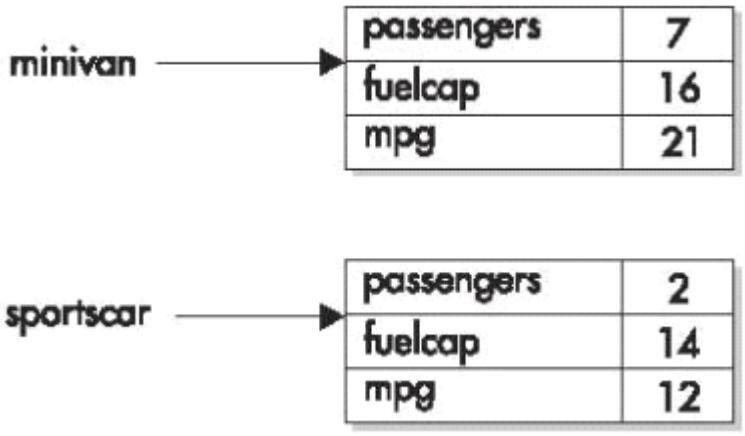


图 8-1 不同对象的实例变量之间是相互对立的

练习

1. 一个类可以含那两部分的内容？
2. 通过对象来访问类的成员的时候需要什么运算符？
3. 每一个对象都含有类的____的副本。

必备技能 8.3：为类增加成员函数

到目前为止，Vehicle 类中只包含有数据，而没有函数。尽管只含有数据的类是完全可以使用的，但是但多数的类都是含有成员函数的。通常来说，成员函数都是用来处理类中定义的数据的，绝大多数情况下，也对外提供了访问这些数据的接口。所以，我们程序的其它部分都是通过类的成员函数来和类交互的。

我们将通过为 Vehcile 类增加一个成员函数来学习类成员函数的使用方法。回忆一下在前面程序的 main()函数中，我们是通过把汽车每燃烧一加仑油能跑的里程数和汽车能装载的油量相乘来计算汽车最多能跑的里程数的。虽然前面的程序是完全正确的，但是这种处理方式却不是最好的。汽车最多能跑的里程数最好是通过 Vehicle 类自身来完成。这样做的原因是很容易理解的：汽车最多能跑的里程数取决于它的油箱的容量和它的油耗，而这两个量都是封装在 Vehicle 类中的。通过为 Vehicle 类增加一个函数来计算汽车最多能跑的里程数就使得程序具有了面向对象的结构。

给类 Vehicle 增加一个函数是通过在类的声明中指明函数的原型来完成的。例如，下面这个版本的 Vehicle 类中就指明了一个成员函数 range()，用来返回汽车能跑的最多里程数：

view plain

```
1.  class Vehicle
2.  {
3.     public:
4.          int passengers; //载客人数
5.          int fuelcaps;   //油箱的容量
6.          int mpg;        //每加仑油能跑的英里数
7.          int range();    //返回汽车最多能跑的里程数
8.  }
```

由于类中的成员函数，比如上面的 range()函数，是在定义类的时候指明了原型，所以不再需要在别的地方再声明它的原型。

在实现类的成员函数的时候，我们必须通过在成员函数的前面使用类的名字来限定的方式来告诉编译器要实现的是哪个类的成员函数。例如，下面就是一种实现 range()函数的方式：

//实现 vehicle 类的 range 函数

view plain

```
1.  int Vehicle::range()
2.  {
3.      return mpg * fuelcap;
4.  }
```

注意上面的代码中用::运算符把 Vehicle 类和函数 rang()隔开了。它叫做域解析运算符，用来告诉编译器这个函数属于哪个类。就上面的例子来说，::运算符把 Vehicle 类和 range()函数关联起来，表明 range()函数是 Vehicle 类的函数。多个不同名字的类是可以有相同名字的成员函数名字的。由于域解析运算符的存在，编译器就能知道哪个函数是哪个类的。
上面 range()函数只有一句代码：

view plain

```
1.  return mpg * fuelcap;
```

这个语句返回汽车油箱容量和每加仑油能跑的里程数相乘得到的最大里程数。既然 Vehicle 类的每一个对象都有一份 fuelcap 和 mpg 的副本，当调用函数 range()的时候，实际上使用到的就是每一个对象自己的 fuelcap 和 mpg 变量的值。
在上面的函数 range()中，实例变量 fuelcap 和 mpg 都是被直接引用的，而没有在其前面使用对象的名称和使用点号运算符。当一个成员函数使用其类中定义的实例变量的时候，就可以直接引用，而不需要明确指明对象和使用点号运算符。这点应该是很容易理解的。成员函数都是通过类的对象来使用的。一旦这种成员函数的调用发生了，其对象就是可知的。因此在类的成员函数中不再需要重复地指明对象了。这就是说，range()函数中的 fuelcap 和 mpg 指的就是调用函数 range()的时候指定的对象中含有的实例变量的值。当然，Vehicle 类之外的代码必须是通过一个对象加上点号运算符来使用 range()函数的。
调用类的成员函数的时候必须指明具体的对象。这就有两种方式。第一，类的成员函数可以被类之外的代码所调用。此时，我们就必须使用对象的名字和点号运算符。例如，下面的代码调用 range()函数得到的就是 minivan 的最大里程数：

view plain

```
1.  range = minivan.range();
```

上面的调用表示 range()函数中的计算用到的是 minivan 这个对象中的实例变量副本的值，因此，得到的就是 minivan 的最大里程数。
另外一种情况就是在一个类的成员函数中调用同类的另外一个成员函数。此时，我们可以直接调用该成员函数，而不需要使用点号运算符。这是因为编译器此时能知道需要操作的对象。只有当成员函数在它所在的类的范围之外被调用的时候才需要使用对象的名称加上点号运算符。
下面的程序在演示 range()函数的同时，也给我们演示了上面说到的细节：

view plain

```
1. //使用了 Vehicle 类的一个程序
2. #include <iostream>
3. using namespace std;
4. //声明 Vehicle 类
5. class Vehicle
6. {
7. public:
8.     int passengers; // 载客人数
9.     int fuelcap;    // 油箱容量
10.    int mpg;        // 每加仑油能跑的里程数
11.    int range();    //计算并返回汽车能跑的最大里程数，此处为函数声明
12. };
13. //下面是 range() 函数的实现
14. int Vehicle::range()
15. {
16.    return mpg * fuelcap;
17. };
18. int main()
19. {
20.    Vehicle minivan;   // 生成 Vehicle 类的一个对象
21.    Vehicle sportscar; // 生成 Vehicle 类的一个对象
22.    int range1, range2;
23.    //给 minivan 对性的字段赋值
24.    minivan.passengers = 7;
25.    minivan.fuelcap = 16;
26.    minivan.mpg = 21;
27.    //给 minivan 对性的字段赋值
28.    sportscar.passengers = 2;
29.    sportscar.fuelcap = 14;
30.    sportscar.mpg = 12;
31.    //计算汽车加满油后能跑的最大里程数
32.    range1 = minivan.range();
33.    range2 = sportscar.range();
34.    cout << "Minivan can carry " << minivan.passengers << " with a ran
   ge of " << range1 << "\n";
35.    cout << "Sportscar can carry " << sportscar.passengers << " with a
    range of " << range2 << "\n";
36.
37.    return 0;
38. };
```

上面程序的输出如下：

Minivan can carry 7 with a range of 336

Sportscar can carry 2 with a range of 168

练习：

1. 运算符::被称为什么运算符？

2. 运算符::是用来做什么的？

如果在一个类之外使用这个类的成员函数，则必须指明操作的对象并使用点号运算符。对吗？

工程 8-1 创建一个帮助类

如果我们使用一句话来总结类的本质，那就是：类封装了功能。通常，类只是构建大型程序的基本元素。因此每个类都应该是一个功能明确且单一的逻辑体，而且类要尽可能的小，小的不能再小了！也就是说含有冗余功能的类不够明确并且也破坏了代码的结构。但是含有太多过小功能的多个类有显得过于凌乱。那么如何来平衡了？正是这一点使得编程成为了一种艺术！幸运的是，随着程序员编程经验的不断增加，这种艺术性的工作会变得越来越简单。

下面我们就改写第三篇章中的工程 3-3，将其修改为一个帮助类。通过这个例子来丰富大家的编程经验。这样做有什么好处了？首先，整个帮助类定义了一个逻辑单元：简单地显示出 C++中控制语句的语法。因此其功能的定义是相当紧凑和单一的。其次，这种封装帮助功能成一个类的方式是非常让人赏心悦目的。无论我们需要在什么时候提供相关的帮助系统给用户，我们只需要提供它的一个实例即可。最后，由于帮助功能被封装成一个类了，必要的时候我们可以对其进行升级或者修改，而这些完全不会给使用这个类的用户带来任何的副作用。

步骤：

1. 创建一个新的文件，命名为 HelpClass.cpp。为了节省时间，我们可以把项目 3-3 中的代码复制过来。

2. 为了把帮助系统封装成一个类，我们必须先明确地定义帮助系统都由什么组成。例如，在 Help3.cpp 中，有代码来显示菜单，有代码来提示用户进行选择，有代码来检查用户的输入是否有效，还有代码来显示用户选择的信息。程序循环直到用户键入 p。仔细考虑一下就会发现，显示菜单，检查用户的输入并显示帮助信息给用户都是帮助类中不可缺少的组成部分；而如何获取用户的输入以及是否需要重复循环都不是帮助类的组成部分。因此，我们的类中必须包含显示帮助信息，显示帮助菜单，检查用户选择是否有效的功能。这些函数我们分别称之为：helpon()，showmenu()和 isvalid()。

3. 按照下面的代码来声明 Help 类：

```
//封装了帮助功能的类
class Help
{
public:
```

```
    void helpon(char what);
    void showmenu();
    bool isvalid(char ch);
};
```

请注意，这个类是一个只含有成员函数的类，我们此时不需要实例变量。正如我们在前面解释的那样，只含有数据或者只含有函数的类是完全合法的。

4. 按照下面代码所示来创建 helpon()函数：

```
//显示帮助信息
void Help::helpon(char what)
{
  switch(what)
  {
    case '1':
    {
      cout << "The if :\n\n";
      cout << "if (conditaion) statement;\n";
      cout << "else statement;\n";
      break;
    }
    case '2':
    {
      cout << "The switch:\n\n";
      cout << "switch(expression)\n";
      cout << "{\n";
      cout << "   case constant:\n";
      cout << "      statement sequence\n";
      cout << "      break;n";
      cout << "   \\...\n";
      cout << "}\n";
      break;
    }
    case '3':
    {
      cout << "The for:\n\n";
      cout << "for(init;condition;increment)\n";
```

```
    cout << "{\n";
    cout << "    statement\n";
    cout << "}\n";
    break;
}
case '4':
{
    cout << "The while:\n\n";
    cout << "while(condition) statement;\n";
    break;
}
case '5':
{
    cout << "The do-while:\n\n";
    cout << " do\n";
    cout << " {\n";
    cout << "    statement;\n";
    cout << " }while(condition)\n";
    break;
}
case '6':
{
    cout << "The break:\n\n";
    cout << "break;\n";
    break;
}
case '7':
{
    cout << "The continue:\n\n";
    cout << "continue;\n";
    break;
}
case '8':
{
    cout << "The goto:\n\n";
```

```
            cout << "goto label;\n";
            break;
        }
    }
}
```

5. 按照下面的代码创建 showmenu()函数：

```
//显示帮助菜单
void Help::showmenu()
{
    cout << "Help on:\n";
    cout << "   1. if\n";
    cout << "   2. switch\n";
    cout << "   3. for\n";
    cout << "   4. while\n";
    cout << "   5. do-while\n";
    cout << "   6. break\n";
    cout << "   7. continue\n";
    cout << "   8. goto\n";
    cout << "Choose one (q to quit):";
}
```

6. 按照下面的代码创建 isvalid()函数：

```
//如果选择是有效的，则返回 true
bool Help::isvalid(char ch)
{
    if ( ch < '1' || ch > '8' && ch != 'q' )
        return false;
    else
        return true;
}
```

7. 重写其中的 main()函数。完成后的最终代码如下：

```
/* 工程 8-1
   把工程-3 转换为一个 Help 类
*/
```

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

//封装了帮助功能的类
class Help
{
public:
   void helpon(char what);
   void showmenu();
   bool isvalid(char ch);
};
//显示帮助信息
void Help::helpon(char what)
{
   switch(what)
   {
      case '1':
      {
         cout << "The if :\n\n";
         cout << "if (conditaion) statement;\n";
         cout << "else statement;\n";
         break;
      }
      case '2':
      {
         cout << "The switch:\n\n";
         cout << "switch(expression)\n";
         cout << "{\n";
         cout << "   case constant:\n";
         cout << "      statement sequence\n";
         cout << "      break;\n";
         cout << "   \\...\n";
         cout << "}\n";
         break;
      }
```

```
case '3':
{
   cout << "The for:\n\n";
   cout << "for(init;condition;increment)\n";
   cout << "{\n";
   cout << "    statement\n";
   cout << "}\n";
   break;
}
case '4':
{
   cout << "The while:\n\n";
   cout << "while(condition) statement;\n";
   break;
}
case '5':
{
   cout << "The do-while:\n\n";
   cout << " do\n";
   cout << " {\n";
   cout << "    statement;\n";
   cout << " }while(condition)\n";
   break;
}
case '6':
{
   cout << "The break:\n\n";
   cout << "break;\n";
   break;
}
case '7':
{
   cout << "The continue:\n\n";
   cout << "continue;\n";
   break;
```

```
        }
        case '8':
        {
            cout << "The goto:\n\n";
            cout << "goto label;\n";
            break;
        }
    }
}

//显示帮助菜单
void Help::showmenu()
{
    cout << "Help on:\n";
    cout << "   1. if\n";
    cout << "   2. switch\n";
    cout << "   3. for\n";
    cout << "   4. while\n";
    cout << "   5. do-while\n";
    cout << "   6. break\n";
    cout << "   7. continue\n";
    cout << "   8. goto\n";
    cout << "Choose one (q to quit):";
}

//如果选择是有效的，则返回 true
bool Help::isvalid(char ch)
{
    if ( ch < '1' || ch > '8' && ch != 'q' )
        return false;
    else
        return true;
}

int main()
```

```
{
    char choice;
    Help hlpob; //创建一个 Help 类的对象

    //使用 Help 对象来显示信息
    for(;;)
    {
        do
        {
            hlpob.showmenu();
            cin >> choice;
        }while(!hlpob.isvalid(choice));

        if ( choice == 'q' )
            break;

        hlpob.helpon(choice);
    }

    return 0;
}
```

我们运行上面的程序，发现它的功能和前面的第三篇中的工程 3-3 是一样的。这样封装成类的方法的好处就是我们现在有了一个可以复用的帮助系统构件。我们可以在任何需要的地方使用它。

必备技能 8.4：构造函数和析构函数

在前面的例子中，我们可以看到 Vechile 类的对象的实例变量必须通过如下的代码段来手工地进行初始化：

```
minivan.passengers = 7;
minivan.fuelcap = 16;
minivan.mpg = 21;
```

在专业的 C++代码中，这样的方式是不会被用到的。因为这样除了容易出错之外还可能造成忘记了给某个成员变量进行初始化。其实有一个更简单的方式来完成这样的事情：构造函数。

构造函数在对象被创建的时候就对其进行初始化。它的名字和类的名字是一样的，在语法上和函数很相似。然而，构造函数没有明确的返回值类型。构造函数的通行形式如下：

```
class-name()
{
  //constructor code
}
```

通常情况下，我们使用构造函数来给实例变量进行初始化，或者是在其中进行一些别的工作来创建一个完整的对象。

和构造函数相对应的就是析构函数。在大多数的情况下，当一个对象被销毁的时候，我们需要做一些或者一些列的善后工作。局部对象在进入相应的代码段的时候被创建，并在离开其所在代码段的时候被销毁。全局的对象会在程序结束的时候被销毁。这里就有很多因素导致我们需要析构函数。例如，在销毁对象的时候，需要释放之前为其分配的内存空间；或者关闭之前有它打开的文件。在C++中正是由析构函数来做这些事情的。析构函数的名字和构造函数的名字是一样的，但是需要在前面加上~。和构造函数一样，析构函数也是没有返回类型的。

下面就是一个使用了构造函数和析构函数的简单的例子：

view plain

```
1. //一个简单的演示构造函数和析构函数的程序
2. #include <iostream>
3. using namespace std;
4.
5. class MyClass
6. {
7. public:
8.     int x;
9.
10.     //声明构造函数和析构函数
11.     MyClass();
12.     ~MyClass();
13. };
14.
15. //实现构造函数
16. MyClass::MyClass()
17. {
18.     x = 10;
19. }
20.
21. //实现析构函数
```

```
MyClass::~MyClass()
{
    cout << "Destructing ... \n";
}

int main()
{
    MyClass ob1;
    MyClass ob2;

    cout << ob1.x << " " << ob2.x << "\n";

    return 0;
}
```

上面程序的输出如下：

10 10

Destructing ...

Destructing ...

在上面的这个例子中，MyClass 类的构造函数就是：

view plain

```
//实现构造函数
MyClass::MyClass()
{
    x = 10;
}
```

注意上面的构造函数是被声明为共有的。这是因为 MyClass 类的构造函数是在该类之外被调用的。在构造函数中给 MyClass 类的实例变量 x 赋值为 10。这个构造函数在 MyClass 类的对象被创建的时候被调用。例如，在下面的代码行中：

MyClass ob1;

构造函数 MyClass()就会被针对对象 ob1 而调用，也就是给 ob1 的实例变量 x 赋值为 10。同样对 ob2 来说也是这样的。在 ob2 被创建后，其 x 的值也是 10。

MyClass 类的析构函数如下：

view plain

```
//实现析构函数
MyClass::~MyClass()
{
    cout << "Destructing ... \n";
```

```
5. }
```

这个析构函数只是简单的显示了一条信息。但是在真实的程序中，析构函数会被用来释放有该对象使用的一个或者多个资源的（例如，文件句柄或者内存空间）。

练习:

1. 什么是构造函数，它在什么时候会被执行？
2. 构造函数是否有返回值类型？
3. 析构函数什么时候会被调用？

必备技能 8.5：带有参数的构造函数

在前面的例子中，我们用到的是没有参数的构造函数。在某些情况下，这样做是可以的。但是大多数情况下我们都需要带有一个或者多个参数的构造函数。为类的构造函数增加参数和为普通的函数增加参数的方式是一样的：只需要在函数名称后面的括号中声明这些参数即可。例如，下面就是类 MyClass 的一个带有参数的构造函数：

view plain

```
1. MyClass::MyClass(int i )
2. {
3.     x = i;
4. }
```

那么如何为类的构造函数传入参数了？我们只需要在生成类的对象的时候把传入的值和该对象关联即可。C++提供了两种方式来完成这件事情。第一种方式如下：

view plain

```
1. MyClass ob1 = MyClass(101);
```

上面的代码创建了 MyClass 的一个对象，并为其传入 101 这个值。然而这种方式实际中用的比较少。因为第二种方式更简洁明了。其第二种方式就是把一个或者多个参数放置在对象名称的后面，并用括号括起来。例如，用第二种方法完成上面功能的写法：

MyClass ob1(101);

这是我们最常使用的方式。因此我们可以总结出给类的构造函数传递参数的一般形式为：

类类型 变量名称（参数列表);

其中的参数列表就是传递给构造函数的用逗号间隔开的一系列参数。

注意：从技术上来讲，上面提到的两种方式是有细小差别的。这点我们在后面会讨论到。然而，这些些微的差别不影响我们目前编写的程序，或者说对于我们写的大部分程序来说是没用影响。

下面就是一个完成的演示带有参数的构造函数的程序：

view plain

```
1. //带有参数的构造函数
2. #include <iostream>
3. using namespace std;
4. class MyClass
5. {
6. public:
7.     int x;
8.
9.     //声明类的构造函数和析构函数
10.    MyClass(int i); //构造函数
11.    ~MyClass(); //析构函数
12. };
13. //实现类的构造函数
14. MyClass::MyClass(int i)
15. {
16.    x = i;
17. }
18. //实现类的析构函数
19. MyClass::~MyClass()
20. {
21.    cout << "Destructing object whose x value is " << x << "\n";
22. }
23. int main()
24. {
25.    //给构造函数传入参数
26.    MyClass t1(5);
27.    MyClass t2(19);
28.
29.    cout << t1.x << " " << t2.x << "\n";
30.
31.    return 0;
32. }
```

上面程序的输出如下：

5 19

```
Destructing object whose x value is 19
Destructing object whose x value is 5
```

在这个版本的程序中，MyClass 函数的构造函数需要一个参数 i。这个 i 被用来对实例变量 x 进行初始化。因此，当

```
MyClass ob1(5);
```

执行后，就是把 5 传递给了 i，然后把这个值赋值给 x。

和构造函数不同，析构函数不能含有参数。这点很容易理解：我们没有方式能够传递参数给即将被销毁的对象。如果我们确实需要在销毁对象之前为其传入一些数据（这种情况是非常罕见的），我们就应该为此创建一个特殊的变量，并在销毁对象之前，设置一下这个变量即可。

为 Vehicle 类的增加构造函数

到此，我们就可以通过给我们前面学过的 Vehicle 类增加一个带有参数的构造函数来进一步改进这个类了。通过带有参数的构造函数，可以在对象被创建的时候就对其变量 passengers，uelcap 和 mpg 进行初始化。请特别注意我们是如何创建 Vehicle 类的对象的。

view plain

```
1.  //为 Vehicle 类增加构造函数
2.  #include <iostream>
3.  using namespace std;
4.  //声明 Vehicle 类
5.  class Vehicle
6.  {
7.  public:
8.      int passengers; // 载客人数
9.      int fuelcap;    // 油箱容量
10.     int mpg;        // 每加仑油能跑的里程数
11.     //构造函数
12.     Vehicle(int p, int f, int m);
13.     int range();    //计算并返回汽车能跑的最大里程数，此处为函数声明
14. };
15. //构造函数的实现
16. Vehicle::Vehicle(int p, int f, int m)
17. {
18.     passengers = p;
19.     fuelcap = f;
20.     mpg = m;
21. }
22. //range()函数的实现
23. int Vehicle::range()
24. {
25.     return mpg * fuelcap;
```

```
};
int main()
{
    //通过构造函数为类的对象传递信息
    Vehicle minivan(7,16,21);
    Vehicle sportscar(2,14,12);
    int range1, range2;
    //计算汽车加满油后能跑的最大里程数
    range1 = minivan.range();
    range2 = sportscar.range();
    cout << "Minivan can carry " << minivan.passengers << " with a ran
ge of " << range1 << "\n";
    cout << "Sportscar can carry " << sportscar.passengers << " with a
 range of " << range2 << "\n";

    return 0;
};
```

在上面的程序中，对象 minvan 和 sportscar 都是在创建的时候通过构造函数来初始化的。每个对象的初始化值都是通过构造函数的参数来传递的。例如：

Vehicle minivan(7,16,21);

就是把 7，16，21 传递给 Vehicle 的构造函数。因此，minivan 这个对象的 passengers，fuelcap，mpg 的就分别是 7,16 和 21。因此上面程序的输出和先前程序的输出时一样的。

另外一种进行初始的方式

如果一个构造函数只是需要一个参数，此时我们可以采用另外一种方式来对其进行初始化。参见下面的程序：

view plain

```
//另外一种进行初始化的方式
#include <iostream>
using namespace std;
class MyClass
{
public:
    int x;

    //声明类的构造函数和析构函数
    MyClass(int i); //构造函数
    ~MyClass(); //析构函数
};
//实现类的构造函数
```

```
14. MyClass::MyClass(int i)
15. {
16.     x = i;
17. }
18. //实现类的析构函数
19. MyClass::~MyClass()
20. {
21.     cout << "Destructing object whose x value is " << x << "\n";
22. }
23. int main()
24. {
25.     MyClass ob = 5;
26.
27.     cout << ob.x << "\n";
28.
29.     return 0;
30. }
```

在这里，MyClass 类的构造函数只需要一个参数。请特别注意在 main()函数中是如何声明 ob 这个对象的：

view plain

```
1.  MyClass ob = 5;
```

在这种初始化方式下，5 会自动地被传递给 MyClass()构造函数的参数 i。编译器对这种初始化的方式就像是和

view plain

```
1.  MyClass ob(5);
```

一样的。

通常，在任何构造函数只有需要一个参数的情况下，我们都可以或者使用 ob(x)或者使用 ob=x 的方式来对对象进行初始化。这是因为我们创建的构造函数如果只需要一个参数，就意味着我们阴隐式地创建了一种从参数的类型到该类类型的转换。

练习

1. 假设有一个类 Test，写出如何为其声明接受一个名称为 count 的参数的构造函数？
2. 下面的语句可以怎么被重写：

```
Test ob = Test(10);
```

3. 上面第二个问题中的声明可以怎样被重写？

必备技能 8.6：内联函数

在我们继续探讨类之前，有必要先对一个很重要的题外话进行讨论一下。那就是内联函数。尽管它并不属于面向对象的编程范畴，但它是 C++中一个非常有用的特性，也是经常会在类的定义中被用到的一个特性。一个内联函数会在引用它的地方被展开，而不是像普通函数那样是被调用。有两种方式可以创建一个内联函数。其一就是使用 inline 修饰符。

例如，想要创建一个名称为 f，返回值为一个 int 类型的，不需要参数的函数，可以采用如下的方式来声明它：

```
inline int f()
{
  //...
}
```

其中 inline 修饰符是要放置在声明函数的其它要素之前的。

内联函数的存在是出于效率的考虑。每一次进行函数调用的时候，计算机必须执行一些列的指令来完成函数的调用。其中就包括参数的入栈。在很多情况下，完成这些操作需要多个 CPU 周期。然而，当一个函数被以内联的方式展开的时候，就不会存在这样的开销。因此，程序运行的总速度也会提高。但是，如果内联函数中的代码非常多，那么程序的总大小也将会变得很大。正式由于这样的原因，最好的内联函数就是那些非常短小的函数。大多数的大型函数不应该是内联函数，而应该是作为普通的函数。

下面的程序就展示了内联函数的用法：

view plain

```
1. //内联函数的示例
2.
3. #include <iostream>
4. using namespace std;
5.
6. class c1
7. {
8.     int i; // 缺省为私有的数据成员
9. public:
10.     int get_i();
11.     void put_i(int j);
12.
13. };
14.
15. inline int c1::get_i()
16. {
17.     return i;
18. }
19.
```

```
20. inline void c1::put_i(int j)
21. {
22.     i = j;
23. }
24.
25. int main()
26. {
27.     c1 s;
28.     s.put_i(10);
29.     cout << s.get_i();
30.
31.     return 0;
32. }
```

有一个技术点必需明确：inline 只是一种要求，而不是一种命令。也就是说 inline 修饰符只是要求编译器生成内联的代码。但是在很多情况下，编译器是不会按照 inline 的要求来生成内联代码的。下面列举了一些这样的情况:

●·如果一个函数含有循环，switch 或者 goto 语句，有些编译器则不会生成内联代码。

●·递归函数不能作为内联函数

●·含有静态变量的函数通常是不允许作为内联函数的。

请记住：内联的限制要求和编译器有密切的关系。因此我们最好查阅一下自己使用的编译器的文档，以确定其对内联函数有那些具体的要求。

在类的内部创建内联函数

第二种创建内联函数的方法就是在定义类的代码中来定义函数的实现编码。任何在类的定义中定义的函数都会被自动地被认为是内联函数。此时就不需要在函数的声明前面使用关键字 inline 了。例如，前面的程序可以重写为：

view plain

```
1.  //内联函数的示例
2.
3.  #include <iostream>
4.  using namespace std;
5.
6.  class c1
7.  {
8.      int i; // 缺省为私有的数据成员
9.  public:
10.     int get_i() { return i; };// 在类的定义中定义函数
11.     void put_i(int j) { i = j; };
12.
```

```
};


int main()
{
    c1 s;
    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

请注意上面程序中函数代码的组织方式。对于那些非常简短的函数，上面的代码组织方式反映出了典型的 C++风格。然而，我们也可以这样编写代码：

view plain

```
class c1
{
    int i; // 缺省为私有的数据成员
public:
int get_i()
{
    return i;
};// 在类的定义中定义函数
void put_i(int j)
{
     i = j;
};

};
```

一些比较短小的函数，例如上面程序中的这些函数，通常都会在类的定义中来定义。在使用类的时候，使用内联函数是一种非常常见的现象。这是因为在类中通常需要提供一个公有的函数来提供对私有数据的访问。这种函数被称为访问函数。由于大多数 C++程序员都会在类中定义访问函数和其它一些短小的函数，我们在本书后面的示例程序中也会采用这种传统。这也是我们应该使用的一种方式。

下面的程序中，我们重新编写了 Vehicle 类。其构造函数和析构函数，以及 range()函数都是在类中定义的。变量 passengers，fuelcap 和 mpg 都被声明成了私有的数据。我们为其增加了访问函数来获取他们的值。

view plain

```
1. //内联的构造函数，析构函数和 range()函数
2.
3. #include <iostream>
4. using namespace std;
5.
6. //声明 Vehicle 类
7. class Vehicle
8. {
9.     //私有数据
10.     int passengers;
11.     int fuelcap;
12.     int mpg;
13.
14. public:
15.     //构造函数
16.     Vehicle(int p, int f, int m )
17.     {
18.         passengers = p;
19.         fuelcap =f;
20.         mpg = m;
21.     }
22.
23.     //计算并返回最大行程
24.     int range()
25.     {
26.         return mpg * fuelcap;
27.     }
28.
29.     //访问函数
30.     int get_passengers() { return passengers ; }
31.     int get_fuelcap() { return fuelcap; }
32.     int get_mpg() {return mpg; }
33. };
34.
35. int main()
36. {
37.     //给 Vehicle 的构造函数传递参数
38.     Vehicle minivan(7,16,21);
39.     Vehicle sportscar(2,14,12);
40.
41.     int range1, range2;
42.
```

```
43.     //计算装满油后最大能行驶的里程数
44.     range1 = minivan.range();
45.     range2 = sportscar.range();
46.
47.     cout << "Minivan can carry " << minivan.get_passengers()
48.          << " withe a range of " << range1 << "\n";
49.     cout << "sportscar can carry " << sportscar.get_passengers()
50.          << " withe a range of " << range2 << "\n";
51.
52.     return 0;
53. }
```

由于成员变量现在都是私有的，我们就必须在 main()函数中使用访问函数 get_passengers()来获取一辆车的载客人数。

练习：

1. inline 是用来做什么用的？
2. 是否可以在类的声明中定义内联函数？
3. 什么是访问函数？
4.

专家答疑

问：是否可以在一个类中声明另外的一个类？ 也就是说类的声明是否可以嵌套？

答：是的。在一个类中定义另外的一个类是可以的，这样就是创建了一个嵌套类。由于类的声明实际上定义了一个作用范围，因此嵌套的类只有在该类的范围内才是有效的。坦白来讲，由于 C++的其它一些丰富特性和灵活性，例如后面我们会讨论到的继承性，需要创建嵌套类的情况实际上不存在。

必备技能 8.7：对象数组

我们可以像创建其它数据类型的数组那样创建对象数组。例如，下面的代码就创建了 MyClass 类对象的数组。我们可以用索引来访问构成数组元素的对象。

view plain

```
1.  //创建对象构成的数组
2.  #include <iostream>
3.  using namespace std;
4.
5.  class MyClass
6.  {
```

```
7.      int x;
8.  public:
9.      void set_x(int i)
10.     {
11.         x = i;
12.     }
13.     int get_x()
14.     {
15.         return x;
16.     }
17. };
18.
19. int main()
20. {
21.     MyClass obs[4]; //生成对象数组
22.     int i;
23.
24.     for ( i = 0; i< 4; i++ )
25.     {
26.         obs[i].set_x(i);
27.     }
28.
29.     for ( i = 0; i< 4; i++ )
30.     {
31.         cout <<"obs[" << i << "].getx():" << obs[i].get_x() << "\n";

32.     }
33.
34.     return 0;
35. }
```

上面程序的输出如下：

obs[0].getx():0

obs[1].getx():1

obs[2].getx():2

obs[3].getx():3

基本技能 8.8：初始化对象数组

如果类的构造函数是带有参数的，那么该类的对象数组是可以被初始化的。例如，下面的程序中 MyClass 类的构造函数就是带有参数的，obs 就是被初始化了的对象数组。

view plain

```cpp
//对象数组的初始化
#include <iostream>
using namespace std;
class MyClass
{
    int x;
public:
    MyClass(int i)
    {
        x = i;
    }

    int get_x()
    {
        return x;
    }
};
int main()
{
    MyClass obs[4] = { -1, -2, -3,-4 };
    int i;

    for( i = 0; i < 4; i++)
    {
        cout << "obs[" << i << "].get_X():" << obs[i].get_x() << "\n";

    }

    return 0;
}
```

在上面的程序中，值-1 到-4 被传递给 MyClass 类的构造函数。程序的输出如下：

obs[0].get_X():-1

obs[1].get_X():-2

obs[2].get_X():-3

obs[3].get_X():-4

实际上，上面进行初始化的语句是下面形式的简写：

MyClass obs[4] = { MyClass(-1), MyClass(-2), MyClass(-3), MyClass(-4) };

正如我们在前面提到的那样，当构造函数只需要一个参数的时候就意味着从参数的类型到该类类型的一个隐式转换。上面较长形式的写法仅仅是直接调用了类的构造函数。

当类的构造函数需要多于一个参数的时候，此时类对象数组的初始化就必须采用上面的较长形式的写法。 例如：

view plain

```
#include <iostream>
using namespace std;
class MyClass
{
    int x, y;
public:
    MyClass( int i, int j)
    {
        x = i;
        y = j;
    }

    int get_x()
    {
        return x;
    }

    int get_y()
    {
        return y;
    }
};
int main()
{
    MyClass obs[4][2] =
    {
        MyClass(1,2), MyClass(3,4),
        MyClass(5,6), MyClass(7,8),
        MyClass(9,10), MyClass(11,12),
        MyClass(13,14), MyClass(15,16),


    };

    int i;

    for( i = 0; i < 4; i++ )
    {
        cout << obs[i][0].get_x() << ' ';
        cout << obs[i][0].get_y() << '\n';
```

```
41.         cout << obs[i][1].get_x() << ' ';
42.         cout << obs[i][1].get_y() << '\n';
43.     }
44.
45.     return 0;
46. }
```

在这个例子中， MyClass 类的构造函数需要两个参数。在 main()函数中，我们声明了类对象数组，并通过直接调用构造函数的方式来对数组进行初始化。当对对象数组进行初始化的时候，我们总是可以使用上面的这种较长形式的写法，即使当构造函数只需要一个参数的时。只是相对而言，在只需要一个参数的时候，较短形式的写法更加便捷一些。上面程序的输出如下：

1 2

3 4

5 6

7 8

9 10

11 12

13 14

15 16

必备技能 8.9：指向对象的指针

我们可以通过直接使用的方式来使用对象，也可以通过使用指向对象指针的方式来使用对象。当使用指向对象的指针来访问对象的某些元素的时候，我们必须使用箭头运算符：->。这个运算符是通过在减号后面跟上一个大于号组合而成的。

声明一个对象指针的方法和前面学习过的声明其它数据类型指针的方法是一样的。下面的程序就是创建了一个简单的类，叫做 P_example，然后定义了该类的一个对象 ob，再定义了一个类型为 P_example 的指针。程序展示了如何直接访问 ob 这个对象以及如何使用指针来访问这个对象。

view plain

```
1. //一个使用了对象指针的简单示例
2. #include <iostream>
3. using namespace std;
4. class P_example
5. {
6.     int num;
7. public:
```

```
8.      void set_num( int val)
9.      {
10.         num = val;
11.     }
12.     void show_num()
13.     {
14.         cout << num << "\n";
15.     }
16. };
17. int main()
18. {
19.     P_example ob, *p; //声明一个对象和一个指向它的指针
20.
21.     ob.set_num(1); //直接通过对象调用方法
22.     ob.show_num();
23.
24.     p =&ob;         //把 ob 的地址赋值给 p
25.     p->set_num(20); //通过指向对象的指针调用方法
26.     p->show_num(); //注意箭头运算符的使用
27.
28.     return 0;
29. }
```

注意：对象 ob 的地址的获取是通过使用&运算符来取得的。这点和获取其它任何类型变量的地址是一样的。

我们都知道，当对指针进行自增或者自减运算的时候，指针总是增加或减少到下一个该类型元素的地址处。这点对于对象来说也是一样的，即增加到或者减少到下一个对象的地址处。为了演示这点，我们修改前面的程序：ob 为有两个 P_example 类对象的数组。请注意其中 P 是如何通过自增和自减来访问数组中的两个元素的。

view plain

```
1.  //对象指针的自增和自减
2.  #include <iostream>
3.  using namespace std;
4.  class P_example
5.  {
6.      int num;
7.  public:
8.      void set_num(int val)
9.      {
10.         num = val;
11.     }
12.     void show_num()
```

```
13.     {
14.         cout << num << "\n";
15.     }
16. };
17. int main()
18. {
19.     P_example ob[2], *p;
20.
21.     ob[0].set_num(10); //直接访问对象
22.     ob[1].set_num(20);
23.
24.     p = &ob[0]; //取得第一个元素的地址
25.     p->show_num(); //使用指针来显示 ob[0]的值
26.     p++; // 前进到下一个对象处
27.     p->show_num(); //使用指针来显示 ob[1]的值
28.
29.     p--; //退回到第一个元素的地址处
30.     p->show_num(); //再次显示 ob[0]的值
31.
32.     return 0;
33.
34. }
```

上面程序的输出为：

10

20

10

在本书后面的学习中我们会看到，对象指针在 C++的一个重要特性中起着关键的作用：多态性。

练习：

1. 对象数组是否可以进行初始化？

2. 针对指向对象的指针，使用什么样的运算符来访问其成员？

对象引用

对象的引用和对其他类型数据的引用是一样的，其中没有特殊的限制和指令。

复习题：

1. 对象和类有什么区别？

2. 声明一个类需要使用哪个关键字？

3. 每个类的对象都有自己的一个什么的拷贝？

4. 写出如何声明一个名称为 Test，含有两个私有的 int 变量，一个是 count，一个是 max 的类？

5. 类的构造函数的名字是什么？析构函数呢？

6. 有如下类的声明：

view plain

```
class Sample
{
    int i;
public :
    Sample(int x)
    {
        i = x;
    }
};
```

写出如何声明一个 Sample 的对象，并给 i 的初始化值为 10

7. 当在声明一个类的时候定义了一个成员函数，编译器会对该函数做怎么样的优化？

8. 写出一个类，名称为 Triangle。它有两个私有的实例变量用来存储直角三角形的底和高。通过构造函数来设置这两个值。并定义两个函数，一个是 hypot()，用于返回斜边的长度；第二个是 area()，用于计算三角形的面积。

9. 扩展 Help 类，使其保存一个整形的 ID 号码，用于区别该类的不同用户。在销毁 Help 类对象的时候，显示出该 ID 号码。并编写一个获取该 ID 号码的 getID()函数。

第九篇 进一步了解类

本章中我们继续讨论前面第八章中学习到的类。本章我们将学习到和类相关的几个特性，包括重载，为函数传递对象参数以及函数返回对象。本章中还将讨论一种特殊的构造函数：拷贝构造函数。这种构造函数在我们需要复制对象的时候会用到。接下来讨论友元函数，结构体和联合体，以及 th is 关键字。最后我们讨论 C + + 令人激动的特性之一：运算符重载。

必备技能 9.1 ：重载构造函数

尽管构造函数是完成一种特定功能的函数，但是它在其它方面和普通的函数没有两样。它们也可以被重载。对构造函数进行重载，只需要另外定义不同的形式即可。例如，在下面的程序中就定义了三个构造函数。

view plain

```
//重载构造函数
#include <iostream>
```

```
using namespace std;
class Sample
{
public:
    int x;
    int y;
    //重载缺省的构造函数
    Sample()
    {
        x = y = 0;
    }
    //一个参数的构造函数
    Sample (int i)
    {
        x = y = i;
    }
    //两个参数的构造函数
    Sample( int i, int j)
    {
        x = i;
        y = j;
    }
};
int main()
{
    Sample t;          //调用缺省的构造函数
    Sample t1(5);      //调用 Sample(int)
    Sample t2(9, 10); //调用 Sample(int , int );
    cout << "t.x: " << t.x << ", t.y: " << t.y << "\n";
    cout << "t1.x: " << t1.x << ", t1.y: " << t1.y << "\n";
    cout << "t2.x: " << t2.x << ", t2.y: " << t2.y << "\n";
}
```

上面程序的输出如下：

t.x: 0, t.y: 0

t1.x: 5, t1.y: 5

t2.x: 9, t2.y: 10

在上面的程序中，我们创建了三个构造函数。第一个是不需要参数的构造函数，其中对 x 和 y 都是用 0 来初始化。这个构造函数就变成了缺省的构造函数，替代了 C++自

动提供的缺省构造函数。第二个构造函数需要一个参数，并把参数的值赋值给 x 和 y。第三个构造函数需要两个参数，分别赋值给 x 和 y。

重载构造函数有以下的几个好处。第一，为类增加了灵活性，允许我们通过不同的方式来创建类的对象。第二，针对一个给定的任务，重载构造函数为该类的用户能够以一种自然的方式来构造类的对象提供了便利。第三，通过定义缺省的构造函数和带有参数的构造函数，我们可以创建未初始化的对象，也可以创建初始化的对象。

必备技能 9.2 ：对象之间相互赋值

如果两个对象是同一个类型的（也就是说，两个对象是同一个类的对象），那么就可以把一个对象赋值给另外的一个对象。两个类只在物理构成上相似是不行的，它们的类名必须相同才行。缺省情况下，当把一个对象赋值给另外一个对象的时候，执行的是从一个对象到另外一个对象的逐位拷贝。赋值之后，两个对象在值上将是相同的，但是还是两个区分独立的对象。下面的程序演示了对象之间的赋值。

view plain

```
1. //对象之间的赋值
2. #include <iostream>
3. using namespace std;
4. class Test
5. {
6.     int a, b;
7. public:
8.     void setab(int i, int j)
9.     {
10.         a = i;
11.         b = j;
12.     }
13.     void showab()
14.     {
15.         cout << "a is " << a << '\n';
16.         cout << "b is " << b << '\n';
17.     }
18. };
19. int main()
20. {
21.     Test ob1, ob2;
22.     ob1.setab(10, 20);
23.     ob2.setab(0, 0);
24.     cout << "ob1 before assignment:\n";
25.     ob1.showab();
26.     cout << "ob2 before assignment:\n";
```

```
27.     ob2.showab();
28.     cout << "\n";
29.     ob2 = ob1; //把 ob1 赋值给 ob2
30.     cout << "ob1 after assignment:\n";
31.     ob1.showab( );
32.     cout << "ob2 after assignment:\n";
33.     ob2.showab( );
34.     cout << "\n";
35.     ob1.setab(-1, -1);  //修改 ob1 的值
36.     cout << "ob1 after changing ob1:\n";
37.     ob1.showab( );
38.     cout << "ob2 after changing ob1:\n";
39.     ob2.showab( );
40.     cout << "\n";
41.
42. }
```

上面程序的输出如下：

ob1 before assignment:

a is 10

b is 20

ob2 before assignment:

a is 0

b is 0

ob1 after assignment:

a is 10

b is 20

ob2 after assignment:

a is 10

b is 20

ob1 after changing ob1:

a is -1

b is -1

ob2 after changing ob1:

a is 10

b is 20

正如上面程序演示的那样，把一个对象赋值给另外一个对象会使得两个对象的值一样。但是这两个对象依然是完全独立的两个对象。因此，后续对其中一个对象值的修改不会影响

到另一个对象。这样也有可能带来副作用。例如，当对象 A 含有一个指向其他对象 B 的指针的时候，该对象 A 的副本也将含有指向 B 的指针。这样以来，对对象 B 的修改将会同时影响到对象 A 和 A 的副本。在这样的情况下，我们需要通过自己定义赋值运算符，从而忽略这种缺省逐位拷贝的操作。这点我们将在本章的后面进行讨论。

必备技能 9.3 ：为函数传递对象作为参数

对象可以像其它类型的数据一样被作为参数传递给函数。对象传递给函数的时候采用的是 C++ 传统的值传递方式。这就意味着是把对象的一个副本，而不是对象本身传递给函数的。因此，在函数内部对参数所做的修改都不会影响到传递到函数中的实参。下面的程序就演示了这一点：

view plain

```
1. //传递对象作为函数的参数
2. #include <iostream>
3. using namespace std;
4. class MyClass
5. {
6.     int val;
7. public:
8.     MyClass(int i)
9.     {
10.        val = i;
11.    }
12.    int getval()
13.    {
14.        return val;
15.    }
16.    void setval(int i)
17.    {
18.        val = i;
19.    }
20. };
21. void display ( MyClass ob )
22. {
23.    cout << ob.getval() << '\n';
24. }
25. void change(MyClass ob)
26. {
27.    ob.setval(100) ; //对传入的实参没有作用
28.    cout << "Value of ob inside change(): ";
29.    display( ob );
30. }
```

```
int main()
{
    MyClass a(10);
    cout << "Value of before calling change(): ";
    display( a );
    change( a );
    cout <<"Value of a after calling change():";
    display( a );
    return 0;
}
```

上面程序的输出如下：

Value of before calling change(): 10

Value of ob inside change(): 100

Value of a after calling change():10

正如程序的输出所示，在函数 change()里面对 ob 的修改不会影响到 main()函数中的对象 a。

构造函数，析构函数和传递对象

尽管上述的传递对象到函数中的方法是非常直观的，但是考虑到构造函数和析构函数，这样的传递方式会导致一些意想不到的问题。为了理解为什么会产生这种问题，我们看看下面的代码：

view plain

```
//构造函数，析构函数和传递对象
#include <iostream>
using namespace std;
class MyClass
{
    int val;
public:
    MyClass(int i)
    {
        val = i;
        cout << "Inside constructor\n";
    }
    ~MyClass()
    {
        cout << "Destructing\n";
    }
    int getval()
    {
        return val;
```

```
    }
};
void display (MyClass ob)
{
    cout << ob.getval() << '\n';
}
int main()
{
    MyClass a(10);
    cout << "Before calling display().\n";
    display(a);
    cout << "After display() returns.\n";
    return 0;
}
```

上面程序的输出如下：

Inside constructor

Before calling display().

10

Destructing

After display() returns.

Destructing

注意上面输出的结果中有两个 Desctructing。

从上面的输出结果中我们可以看出，只有一次调用了构造函数，但是有两次调用了析构函数。为什么会这样呢？

当传递对象作为函数的参数的时候，会生成一个该对象的副本。这个副本就是传递给函数的参数。这就意味着有新的对象生成了。当函数结束的时候，作为参数的副本就要被销毁。这就引入了两个重要的问题：第一，在生成副本的时候，是否会调用构造函数？第二，当销毁副本的时候，是否调用析构函数？这两个问题的答案乍看上去会让我们大吃一惊！

在传递对象作为函数参数的时候执行的是逐位拷贝的操作，这样做的原因很容易理解。由于构造函数是用来对对象的某些方面进行初始化的，所以它不能被用在对一个已经存在对象的复制操作上来。因为这样的操作会修改对象的值。当我们传递一个对象作为参数的时候，我们想要使用的是该对象当前的状态，而不是它的初始状态。

然而，当函数结束的时候，作为实参副本的对象也要被销毁，需要调用析构函数。这样做是非常必要的，因为对象已经超出了其作用域。这样就是上面的程序为什么会两次调用析构函数的原因了。其中第一次就是当函数 display()的参数超出其作用域的时候被调用；第二次是在 main()中，当程序结束，对象 a 被销毁的时候。

小结一下：当对象的一个副本被创建出来作为函数的实参的时候，正常的构造函数是不会被调用的，而调用的是缺省的拷贝构造函数来进行逐位拷贝。然而，当该副本被销毁的时候（通常就是在超出其作用域的时候）则会调用析构函数。

传递对象的引用

另外一种传递对象到函数中的方法就是传递引用。此时是把对该对象的引用传递到了函数中，函数直接操作的是作为实参的对象。因此，在函数中对形参所做的修改都会影响到传递到函数中的实参。但是传递对象的引用到函数中并不适用于所有的场合。然而在适合的场合中，这样做有两大好处。第一，由于此时传递的只是对象的地址，而不是整个对象，传递对象的引用比传递对象更快速和更有效。第二，当传递对象的引用的时候，不会生成新的对象，因此也不需要浪费时间来对临时的对象调用构造函数或者析构函数。

下面的程序就演示了传递对象的引用：

view plain

```
1. //构造函数，析构函数和传递对象的引用
2. #include <iostream>
3. using namespace std;
4. class MyClass
5. {
6.     int val;
7. public:
8.     MyClass(int i)
9.     {
10.        val = i;
11.        cout << "Inside constructor\n";
12.    }
13.    ~MyClass()
14.    {
15.        cout << "Destructing\n";
16.    }
17.    int getval()
18.    {
19.        return val;
20.    }
21.    void setval(int i)
22.    {
23.        val = i;
24.    }
25. };
26. void display (MyClass &ob)
27. {
28.    cout << ob.getval() << '\n';
29. }
```

```
30. void change (MyClass &ob)
31. {
32.     ob.setval(100);
33. }
34. int main()
35. {
36.     MyClass a(10);
37.     cout << "Before calling display().\n";
38.     display(a);
39.     cout << "After display() returns.\n";
40.     change( a );
41.     cout << "After calling change().\n";
42.     display( a );
43.     return 0;
44. }
```

程序的输出如下：

Inside constructor

Before calling display().

10

After display() returns.

After calling change().

100

Destructing

在上面的这个程序中，函数 display()和 change()都是采用引用参数。因此实参的地址，而不是实参的一个副本，被传递到函数中了。这样函数是直接对实参进行操作。例如，当调用函数 change()的时候，传递的是引用。因此，在函数 change()中对形参的修改会影响到 main()函数中传递给它的对象 a。还要注意上面只调用了一次构造函数和析构函数。这是因为只有一个对象被创建和销毁。程序中没有临时的对象。

传递对象时的潜在问题

尽管从理论上来说，正常的值传递方式对于传递对象来说可以起到保护实参的目的。但是这种方式还是有可能对作为参数的对象造成影响，甚至是破坏。例如，当一个对象在生成的时候被分配了一些系统资源，比如内存，在该对象被销毁的时候，这些资源会被释放。这样以来，在函数中它的副本就会在析构函数被调用的时候释放了这些资源。这样就会造成问题。这是因为原始的对象依然还在使用这些资源。这种情况通常会导致原始的对象遭到了破坏。

解决这种问题的一个方法就是传递对象的引用，正如在前面程序中看到的那样。此时，不会生成对象的副本，因此也就不会在函数结束的时候调用析构函数。正如前面解释的那样，传递对象的引用还可以加快函数调用的速度。因此此时只需要传递对象的地址即可。然而，

传递对象的引用不是适用于所有的情况。庆幸的是，还有一个更通用的解决方法：我们可以创建自己的拷贝构造函数。这样做使得我们可以自己决定对象的拷贝应该如何进行，这样也就避免了上面描述的问题。然而，在讨论拷贝构造函数之前，让我们先看看另外一个可以从拷贝构造函数获益的情形。

必备技能 9.4 ：函数返回对象

和对象可以被作为参数传递给函数一样，函数还可以返回对象。在需要函数返回对象的时候，要声明函数的返回类型为对象的类类型，然后在函数中使用普通的 return 语句来返回该类的对象。下面的程序中，类的成员函数 mkBigger()用于返回一个 val 值为调用该函数对象的 val 值 2 倍的对象。

view plain

```
1. //函数返回对象
2. #include <iostream>
3. using namespace std;
4. class MyClass
5. {
6.     int val;
7. public:
8.     //普通的构造函数
9.     MyClass (int i)
10.    {
11.        val = i;
12.        cout << "Inside constructor\n";
13.    }
14.    ~MyClass()
15.    {
16.        cout << "Destructing\n";
17.    }
18.    int getval()
19.    {
20.        return val;
21.    }
22.    //返回对象
23.    MyClass mkBigger()
24.    {
25.        MyClass o(val *2);
26.        return o;
27.    }
28. };
29. void display(MyClass ob)
30. {
```

```
31.     cout << ob.getval() << '\n';
32. }
33. int main()
34. {
35.     cout <<"Before constructing a. \n";
36.     MyClass a(10);
37.     cout <<"After constructing a. \n";
38.     cout <<"Before call to display().\n";
39.     display(a);
40.     cout <<"After call to display().\n";
41.     cout <<"Before call to mkBigger().\n";
42.     a = a.mkBigger();
43.     cout <<"after mkBigger() returns. \n";
44.     cout <<"Before second call to display().\n";
45.     display(a);
46.     cout <<"After second call to display().\n";
47.     return 0;
48. }
```

上面程序的输出如下：

Before constructing a.

Inside constructor

After constructing a.

Before call to display().

10

Destructing

After call to display().

Before call to mkBigger().

Inside constructor

Destructing

Destructing

after mkBigger() returns.

Before second call to display().

20

Destructing

After second call to display().

Destructing

在这个示例程序中，函数 mkBigger()创建了一个局部的对象叫做 o。它的 val 值是调用该函数的对象的 val 值的 2 倍。这个对象随后被返回，赋值给 main()函数中的对象

a。然后 o 被销毁，这就导致输出第一个"Destructing"信息被输出。但是第二次对析构函数的调用又是怎么回事呢？

当函数返回一个对象的时候，临时的对象会被自动创建。这个临时对象持有返回的值。函数返回的正是这个临时对象。在这个对象被返回后，它就会被销毁。这也正是在输出"After mkBigger() returns"之前输出了第二个"Desctructing"的原因。这个临时对象被销毁了。

和为函数传递对象一样，函数返回对象也存在这潜在的问题。这个临时对象的销毁可能引来同样的我们不希望的副作用。例如，如果返回对象的析构函数中释放了某些资源（诸如内存之类的），那么用该临时对象赋值的对象将持有的是无效的资源。对于这个问题的解决就要涉及到拷贝构造函数了。这个我们会在下面的小节中进行讨论。

最后一点：函数也是可以返回对对象的引用的。但是此时必须注意被返回引用的对象不能在函数结束的时候就超出了作用域。

练习：

1. 构造函数不能被重载，对吗？
2. 当为函数传递一个对象的时候，会生成该对象的一个副本。该副本会在函数返回的时候被销毁吗？
3. 当函数返回一个对象的时候，一个持有返回值的链式对象会被创建，对吗？

必备技能 9.5 ：创建并使用拷贝构造函数

正如我们在前面看到的那样，当一个对象被传递给函数或者从函数返回的时候，都会生成该对象的副本。缺省情况下，该副本是对原来对象的逐位拷贝而生成的。这种缺省的操作方式在大多数情况下是可以接受的。但是在那些不能被接受的情况下，我们可以通过为类定义拷贝构造函数来明确指定应该如何进行该类对象的复制。拷贝构造函数是一种特殊的重载的构造函数。它通常在生成对象副本的时候被自动调用。

让我们先来看看为什么需要明确定义拷贝构造函数呢？缺省情况下，当把一个对象传递给一个函数的时候，生成该函数参数的操作是对原来的对象进行逐位拷贝。然而，在某些情况下，这种和原对象完全一样的副本不是我们想要的。例如，如果原对象用到了一些资源，比如一个打开的文件，那么这样生成的副本也和原对象一样使用同样的资源。因此，如果副本对资源做了改动，那么对于原对象来说这些资源也被改动了！

更进一步来说，当函数结束的时候，副本会被销毁，这将导致析构函数被调用，进而导致原对象所需要的资源被释放。

当函数返回对象的时候同样的情况也会发生。编译器会自动生成一个临时对象来保存函数返回对象的值。（这个是自动完成的，超出了我们程序的控制）。这个临时对象在函数一旦返回给调用函数之后就会超出作用域，导致对象的析构函数被调用。然而，析构函数就有可能销毁了调用者想使用的一些资源。这就会导致问题的发生。

导致这些问题的核心就是在生成对象副本的时候进行的是逐位拷贝的操作。为了避免这些问题的发生，我们需要明确定义在生成对象副本的时候应该如何操作，以避免那些意外的副作用。我们就是通过拷贝构造函数来实现这点的。

在我们继续讨论拷贝构造函数的用法之前，我们必须知道，在C++中定义了两种方式来完成把一个对象的值赋给另外一个对象。第一个就是赋值，第二个就是初始化。初始化在以下三种情况下会发生：

●·当我们使用一个对象来明确地初始化另外一个对象的时候，比如在声明对象的时候。

●·当需要生成对象的副本，传给一个函数的时候。

●·当需要创建临时对象的时候（大多数情况下，就是作为返回值）

拷贝构造函数只能用于对象的初始化。它不能应用于赋值。

拷贝构造函数的通用形式如下：

```
类名(const 类名 &ob)
{
  //body of constructor
}
```

其中，obj是对一个对象的引用，用来对另外的一个对象进行初始化。例如，假设一个类叫做MyClass，并且y是这个类的一个对象，那么下面的语句将会调用MyClass类的拷贝构造函数：

```
MyClass x = y;//用y显示地来初始化x
func1(y); //y作为函数的实参
y=func2();// y用来接受函数的返回值
```

在上面的前两种情况中，会把对y对象的引用传递够拷贝构造函数。第三种情况会把对函数func2()返回对象的引用传递给拷贝构造函数。因此，当一个对象被作为参数，或者由函数返回，或者在初始化的时候被用到，使用的都是拷贝构造函数来生成对象的副本。

请牢记，当把一个对象赋值给另外对象的时候，拷贝构造函数是不会被调用到的。例如，下面的代码不会调用拷贝构造函数：

```
MyClass x;
MyClass y;
x = y;// 这里不会用到拷贝构造函数
```

再次说明一下，赋值操作是由赋值运算符来处理的，而不是拷贝构造函数。

下面的程序演示了拷贝构造函数的使用：

view plain

```
1. /* 当为函数出入对象的时候会调用拷贝构造函数 */
2. #include <iostream>
3. using namespace std;
4. class MyClass
```

```
{
    int val;
    int copynumber;
public:
    //普通的构造函数
    MyClass(int i)
    {
        val = i;
        copynumber = 0;
        cout << " Inside normal constructor. \n";
    }
    //拷贝构造函数
    MyClass (const MyClass &o)
    {
        val = o.val;
        copynumber = o.copynumber + 1;
        cout << " Inside copy constructor. \n";
    }
    ~MyClass ()
    {
        if ( copynumber == 0)
        {
            cout << "Destruction original. \n";
        }
        else
        {
            cout << "Destructing copy " << copynumber << "\n";
        }
    }
    int getval()
    {
        return val;
    }
};
void display(MyClass ob)
{
    cout << ob.getval() << "\n";
}
int main()
{
    MyClass a(10);
    display(a);
    return 0;
}
```

上面程序的输出如下：

Inside normal constructor.

Inside copy constructor.

10

Destructing copy 1

Destruction original.

上面过程的执行过程是这样的：当在 main()函数中创建对象 a 的时候，通过普通的构造函数为其 copynumber 赋值为 0。接着，a 被传递给了函数 display()。此时会调用拷贝构造函数来生成一个 a 的副本。在此过程中，拷贝构造函数增加了 copynumber 的值。当函数 display()返回的时候，对象 ob 就超出其作用域了。这导致析构函数会被调用。最后，当 main()函数返回的时候，a 也超出其作用域了。

我们还可以尝试对上面的程序进行修改。例如，创建一个返回 MyClass 类对象的函数，观察拷贝构造函数是在何时被调用的。

练习：

1. 缺省的拷贝构造函数是如何生成对象的副本的？
2. 当把一个对象赋值给另外的一个对象时会调用拷贝构造函数，对吗？
3. 为什么我们需要明确定义类的拷贝构造函数呢？

必备技能 9.6 ：友元函数

通常情况下，只有类的成员才能够访问类的私有成员。然而，我们可以通过声明一个函数为类的友元函数来允许这个非成员函数访问类的私有成员。这个是通过在类的共有成员区引入该函数的原型，并在前面加上 friend 关键字来完成的。例如，下面的代码段中，frnd()就被声明成是类 MyClass 的一个友元函数：

view plain

```
class MyClass
{
    //
public:
    friend void frnd(MyClass ob);
    //
};
```

可以看出，关键字 friend 是位于最前的。一个函数可以是多个类的友元函数。下面就是一个简短的示例程序，其中由友元函数来判断 MyClass 类的两个私有字段是否有公约数：

view plain

```
1. //友元函数示例
2. #include <iostream>
3. using namespace std;
4.
5. class MyClass
6. {
7.     int a, b;
8. public:
9.     MyClass ( int i, int j )
10.     {
11.         a = i;
12.         b = j;
13.     }
14.
15.     friend int comDenom(MyClass x); // 友元函数
16.
17.
18.
19. };
20. //注意：comDenom 并不是类的成员函数
21. int comDenom(MyClass x )
22. {
23.     /*由于 comDenom 是友元函数，所以它可以直接访问类的成员 a 和 b */
24.     int max = x.a < x.b ? x.a : x.b;
25.
26.     for ( int i = 2; i <= max; i++)
27.     {
28.         if ( (x.a % i == 0 ) && (x.b % i == 0 ) )
29.         {
30.             return i;
31.         }
32.     }
33.
34.     return 0;
35. }
36.
37. int main()
38. {
39.     MyClass n(18,111);
```

```
40.
41.     if (comDenom((n)))
42.     {
43.         cout <<"Common denominator is " << comDenom(n) << "\n";
44.     }
45.     else
46.     {
47.         cout << "No common denominator. \n";
48.     }
49.
50.     return 0;
51. }
```

在这个示例中，函数 comDenom()并不是类 MyClass 的成员函数。然而，它却可以完全访问类 MyClass 的私有成员。更明确地说，它可以完全访问 x.a 和 x.b。还需要注意的是 comDenom()的调用和普通函数的调用是一样的。前面不需要使用类的名称（其实，前面是不能使用类的名称的）。一般情况下，友元函数会被传入一个或者多个对象来作为友元函数的参数，就像上面的 comDenom()一样。

从上面的示例程序来看，把函数 comDenom()作为类 MyClass 的友元函数和作为类的成员函数相比并没有什么明显的好处。实际中，在有些情况下，友元函数是相当有用的。其一，友元函数可以被用来重载某些运算符，这点我们在本章的后面会进行讨论。其二，友元函数简化了某些输出输出函数的创建，这点我们在第十一章中会进行讨论。

使用友元函数的第三种好处就是：在某些情况下，两个或者多个类的成员在程序的某些部分是相互有关联的。例如，假设有两个类 Cube 和 Cylinder 用来定义立方体和圆柱体的一些特性，其中就有一个是颜色。为了能对立方体和圆柱体的颜色进行方便地比较，我们可以定义了一个友元函数来比较两个对象的颜色。如果两个对象的颜色相同，函数返回 true，否则就返回 false。下面的程序演示了这种想法：

view plain

```
1.  //两个或者多个类可以共享同一个友元函数
2.
3.  #include <iostream>
4.  using namespace std;
5.
6.  class Cylinder; // 前置声明
7.
8.  enum colors
9.  {
10.     red,
11.     green,
```

```
12.     yellow
13. };
14.
15. class Cube
16. {
17.     colors color;
18. public:
19.     Cube(colors c)
20.     {
21.         color = c;
22.     }
23.
24.     friend bool sameColor(Cube x, Cylinder y);
25.     //....
26. };
27. class Cylinder
28. {
29.     colors color;
30. public:
31.     Cylinder(colors c)
32.     {
33.         color = c;
34.     }
35.
36.     friend bool sameColor(Cube x, Cylinder y);
37.     //....
38. };
39.
40. bool sameColor(Cube x, Cylinder y)
41. {
42.     if (x.color == y.color )
43.     {
44.         return true;
45.     }
46.     else
47.     {
48.         return false;
49.     }
50. }
51.
52. int main()
53. {
54.     Cube cube1(red);
55.     Cube cube2(green);
```

```
56.     Cylinder cyl(green);
57.
58.     if (sameColor(cube1, cyl))
59.     {
60.         cout << "cube1 and cyl are the same color.\n";
61.     }
62.     else
63.     {
64.         cout << "cube1 and cyl are the different color.\n";
65.     }
66.
67.     if (sameColor(cube2, cyl))
68.     {
69.         cout << "cube2 and cyl are the same color.\n";
70.     }
71.     else
72.     {
73.         cout << "cube2 and cyl are the different color.\n";
74.     }
75.
76.     return 0;
77. }
```

上面程序的输出如下：

cube1 and cyl are the different color.

cube2 and cyl are the same color.

注意：在上面的这个程序中我们使用到了类Cylinder 的前置声明（也叫做前置引用）。之所以这样做是因为在Cube 类中的函数 sameColor()引用到了该类，而之前，该并没有声明该类。创建类的前置声明就是采用上面示例程序中的写法。

一个类的友元函数还可以是另外类的成员函数。例如，下面的程序是对上述程序的重写，其中 sameColor()是作为类Cube 的成员函数出现的。注意：这里在声明 sameColor()作为 Cylinder 的友元函数的时候需要使用到作用域解析运算符。

view plain

```
1.  /*一个函数可以是一个类的成员函数，同时又是另外一个类的友元函数 */
2.
3.  #include <iostream>
4.  using namespace std;
5.
6.  class Cylinder; //前置声明
```

```
7.
8.  enum colors
9.  {
10.     red,
11.     green,
12.     yellow
13. };
14.
15. class Cube
16. {
17.     colors color;
18. public:
19.     Cube(colors c)
20.     {
21.         color = c;
22.     }
23.
24.     bool sameColor (Cylinder y);
25.
26. };
27.
28. class Cylinder
29. {
30.     colors color;
31. public:
32.     Cylinder(colors c)
33.     {
34.         color = c;
35.     }
36.
37. // Cube::sameColor()是 Cylinder 类的友元函数
38.     friend bool Cube::sameColor(Cylinder y);
39. };
40.
41.
42. bool Cube::sameColor(Cylinder y)
43. {
44.     if (color == y.color )
45.     {
46.         return true;
47.     }
48.     else
49.     {
50.         return false;
```

```
51.     }
52. }
53.
54. int main()
55. {
56.     Cube cube1(red);
57.     Cube cube2(green);
58.     Cylinder cyl(green);
59.
60.     if (cube1.sameColor(cyl))
61.     {
62.         cout << "cube1 and cyl are the same color.\n";
63.     }
64.     else
65.     {
66.         cout << "cube1 and cyl are the different color. \n";
67.     }
68.
69.     if (cube2.sameColor(cyl))
70.     {
71.         cout << "cube2 and cyl are the same color. \n";
72.     }
73.     else
74.     {
75.         cout << "cube2 and cyl are the different color. \n";
76.     }
77. }
```

在这个程序中，由于 sameColor()是 Cube 的成员函数，所以在调用的时候必须是通过一个 Cube 类的对象来调用。这也就意味着在函数中是可以直接访问 Cube 类的成员变量 color 的。因此，我们只需要为其传入 Cylinder 类的对象即可。

练习

1. 什么是友元函数？声明友元函数时用到那个关键字?
2. 通过对象来使用友元函数时要使用点号（.）运算符？
3. 友元函数是否可以是另外一个类的成员函数？

必备技能 9.7 ：结构体和联合体

除了 class 关键字外，C++中还有两个关键字可以用来创建类类型。第一个可以用来创建结构体，第二个可用来创建联合体。下面分别进行讨论。

结构体

结构体是从 C 语言中继承而来的，声明时使用 struct 关键字。结构体在语法上和类很相似，都可以用来创建类类型。在 C 语言中，结构体只能含有数据成员，但是在 C++中这种限制被突破了。在 C++中结构体本质上是另外来指定类的一种方法。实际上，类和结构体的唯一区别只在于：缺省情况下，结构体的成员是公有的，而类的成员是私有的。在其它方面结构体和类都是相同的。

下面就是一个使用结构体的示例程序：

view plain

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

struct Test
{
    int get_i()
    {
        return i;
    }

    void put_i(int j)
    {
        i = j;
    }
private:
    int i;
};

int main()
{
    Test s;
    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

上面这个简短的程序中定义了一个叫做 Test 的结构体。它的成员 get_i()和 put_i()都是公有的，而成员 i 则是私有的。注意：要使用关键字 private 来指定结构体的私有成员。

下面的程序展示了使用类来实现对等功能：

view plain

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

class Test
{
    int i; //private by default
public:
    int get_i()
    {
        return i;
    }

    void put_i(int j)
    {
        i = j;
    }
};

int main()
{
    Test s;
    s.put_i(10);
    cout << s.get_i();

    return 0;
}
```

专家答疑

问：既然结构体和类几乎是相同的，那么C++中为什么会两者都有呢？

答：从表面上来看，既然结构体和类有着实际上相同的能力，那么两者在C++中同时存在就是一种冗余。许多C++的新手也会疑惑为什么会存在这种很明显的重复呢？实际上，有认为或者class或者struct关键字是不必要这种想法的人并不少。

这种现象存在的原因是为了保证C++对C的兼容。这样以来，标准C语言中的结构体在C++中就是完全有效的。在C语音中是没有公有和私有概念的。所有结构体的成员都缺省地是共有的。这也是为什么在C++中结构体的成员缺省情况下都是公有的而不是私有的。而关键字class是专门被设计出来用来支持封装的，这样它的成员缺省情况下都应该是私有的才是合理的。这样以来，为了避免和C语言中这个问题的不兼容，就没有对结构体的缺省情况进行修改，而是增加了新的关键字。然而，长远地来看，把结

构体和类区分开来还有一个重要的原因。由于类在句法上是和结构体相区别的单独实体，因此它可自由的进化和发展，而不用考虑必须和 C 中的结构体兼容的问题。由于结构体和类是相互独立的，那么 C++以后的发展方向就不会受到必须和 C 中结构体保持兼容这个问题的阻碍。

更重要的是：C++程序员可以使用类来定义含有成员函数的对象的通用形式，可以使用结构体来定义那些只含有数据成员的传统的对象。有时候会采用缩写 POD 来描述这种不含有成员函数的结构体。POD 代表的就是“plain old data.”(简单的老式数据)

联合体

联合体就是由两个或者多个不同的变量同享的内存区域。通过使用 union 关键字可以创建联合体。 联合体的声明和结构体的声明有些类似，示例如下：

view plain

```
union utype
{
    short int i;
    char ch;
};
```

上面的示例代码中定义了一个联合体，其中短整形变量 i 和字符型变量 ch 共享同一个内存位置。有一点必须明确：这个联合体是不能同时即含有一个整形值，又含有一个字符值的。这是因为 i 和 ch 是相互重叠的。在程序中，任何时候我们都可以对存储在这个联合体中的数值活着作为整形数或者字符来处理。因此，联合体使得我们可以以多种方式来看待同一块数据。

我们可以通过在联合体声明的后面紧跟变量的名字来声明联合体变量，或者是使用单独的声明语句。例如，想要声明一个 utype 类型的联合体变量 u_var，我们可以这样写：

utype u_var;

对于变量 u_var 来说，短整形的 i 和字符型的 ch 共享同一个内存位置。（当然，i 要占用两个字节，而 ch 只占用一个字节）图 9-1 展示了 i 和 ch 是如何共享这同一块内存地址的。

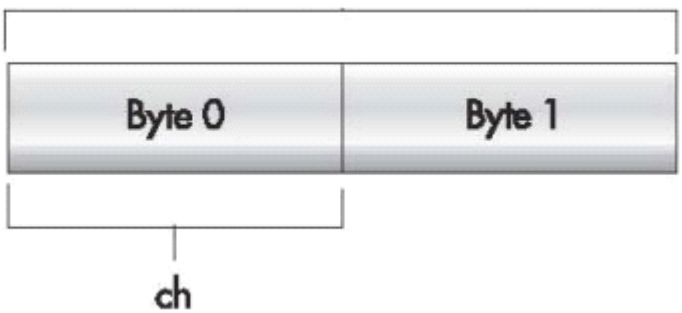


图 9-1 i 和 ch 共享内存

在 C++中，联合体本质上就是类，只是它的所有元素都存储在同一个位置。实际上，联合体定义了一种类类型。联合体可以含有构造函数和析构函数以及其它的成员函数。由于联合体是从 C 语言中继承而来的，因此缺省情况下，它的所有成员都是公有的，而不是私有的。

下面的程序中就使用了联合体来显示构成短整形的高字节和低字节的字符值（我们假定短整形是两个字节）：

view plain

```
1. //演示联合体
2.
3. #include <iostream>
4. using namespace std;
5.
6. union u_type
7. {
8.     u_type(short int a)
9.     {
10.        i = a;
11.    }
12.    u_type ( char x, char y )
13.    {
14.        ch[0] = x;
15.        ch[1] = y;
16.    }
17.
18.    void showChars()
19.    {
20.        cout << ch[0] << " ";
21.        cout << ch[1] << "\n";
22.    }
23.
24.    //联合体的数据成员
25.    short int i;
26.    char ch[2];
27. };
28.
29. int main()
30. {
31.    u_type u(1000);
32.    u_type u2('X','Y');
33.
34.    //联合体中的数据可以被看作是短整形类型，也可以被看作是两个字符
```

```
35.     cout << " u as interger: ";
36.     cout << u.i << "\n";
37.     cout << " u as chars: ";
38.     u.showChars();
39.
40.     cout << " u2 as interger: ";
41.     cout << u2.i << "\n";
42.     cout << " u2 as chars: ";
43.     u2.showChars();
44. }
```

上面程序的输出如下：

u as interger: 1000

u as chars: ·

u2 as interger: 22872

u2 as chars: X Y

正如程序输出的那样，使用 u_type 联合体，我们可以把同一个数据以两种方式来看待。

和结构体一样，联合体也是 C++从语言 C 中继承而来的。然而在 C 语言中，联合体只能含有数据成员，而不能含有成员函数和构造函数。在 C++中，联合体被扩展到了和类几乎有相同的能力。虽然 C++为联合体提供了更多的功能，但是我们不一定非要使用到这些功能。通常情况下，联合体都是仅含有数据的。然而，在必要的时候，我们是可以为联合体增加函数来封装对其数据的处理。

在使用 C++的联合体的时候有几个限制必须注意。其中大多是都和我们在本书后面要讨论的 C++特性相关，这里仅是为了叙述的完整性而提一下。首先，联合体不能从类继承而来；其次，联合体也不能被继承。联合体不能含有虚的成员函数，也不能含有静态数据。联合体中也不能有引用成员，也不能含有任何重载了=运算符的成员。最后，有显示构造函数和析构函数的对象也不能作为联合体的成员。

匿名联合体

在 C++中有一种特殊的联合体叫做匿名联合体。匿名联合体没有类型名称，也不能声明联合体类型的变量。实际上，匿名联合体告诉编译器它的成员变量是共享相同的内存位置的。然而你，这些联合体的成员变量时可以被直接饮用的，而不需要点号运算符。例如下面的程序：

view plain

```
1.  //匿名联合体
2.  #include <iostream>
```

```
3.  using namespace std;
4.
5.  int main()
6.  {
7.      //定义匿名联合体
8.      union
9.      {
10.         long l;
11.         double d;
12.         char s[4];
13.     };
14.
15.     //现在就可以直接引用联合体的数据成员了
16.     l = 100000;
17.     cout << l << " ";
18.     d = 123.2342;
19.     cout << d << " ";
20.     strcpy(s, "hi");
21.     cout << s;
22.
23.     return 0;
24. }
```

正如我们所看到的那样，联合体的元素可以和普通的局部变量一样被直接引用。这个程序中演示的对匿名联合体使用方法就是我们实际中的使用方法。更进一步来说，尽管这些成员变量时声明在联合体中的，但是他们和其它的局部变量有着相同的作用域。这也就意味着，这些匿名联合体成员的名称不能和同一个作用域中其它标识符的名称冲突。

所以针对于联合体的限制也都适用于匿名联合体。除此之外，还有：第一，匿名联合体只能含有数据成员，而不能含有成员函数；第二，匿名联合体中不能含有私有的或者保护的成员；最后，全局的匿名联合体必须指定为是静态的。

必备技能 9.8：th is 关键字

在我们学习运算符重载之前，有必要先来讨论一下 C++中的另外一个关键字：this。当每次调用成员函数的时候，会自动地传入一个指针，叫做 this 指针。该指针指向调用该成员函数的对象。

正如我们在前面学习到的那样，成员函数是可以直接访问类的私有数据的。例如下面的程序：

view plain

```
class Test
{
    int i;
    void f()
    {
        //....
    };
};
```

在函数 f()中，我们可以使用下面的语句来为 i 赋值 10：

i = 10;

实际上，上述语句是下面语句的简写形式：

this->i = 10;

为了更好的理解这点，请参看下面这个简短的程序：

view plain

```
//使用 this 关键字
#include <iostream>
using namespace std;
class Test
{
    int i;
public:
    void load_i(int val)
    {
        this->i = val; //等同于 i = val;
    };
    int get_i()
    {
        return this->i;  //等同于 return i;
    }
};
int main()
{
    Test o;
    o.load_i(100);
    cout << o.get_i();
    return 0;
}
```

上面的程序将显示数字 100。当然，上述例子并不能展示 this 指针的重要性，也不会有人以这种方式来使用 this。但是，不久我们就会看到为什么 this 指针在 C++中很重要了。

练习：

1. 结构体是否可以函数成员函数
2. 联合体有什么特点？
3. this 指针代表什么？

必备技能 9.9 ：运算符重载

在本章的剩余部分，我们将讨论 C++中最令人激动也是最强大的特性之一:运算符重载。在 C++中，和我们创建的类相关的运算符可以被重载。运算符重载带来的好处就是：我们可以无缝地把心的数据类型集成到我们的程序环境中。

当进行运算符重载的时候，我们就是在针对一个特此的类来定义该运算符的含义。例如，链表类就可以使用+运算符来为链表中增加元素。栈类可以使用+运算符来把一个对象压入栈顶。

还有别的类可能以完全不同的另外的方式来使用+运算符。当一个运算符被重载以后，它并没有失去原有的含义。而是我们针对相关的类定义了一个新的运算符。因此重载+运算符来处理链表并不意味着+运算符对于整型数的含义也发生了变化。

运算符的重载和函数的重载是密切相关的。重载运算符的时候，我们必须针对和他相关的类来定义该运算符的含义。我们是通过创建一个运算符函数来进行运算符重载的。通用的形式如下：

```
类型 类名::operator#(参数列表)
{
    //操作实现
}
```

这里，我们要用需要被重载的运算符来替换其中的#；其中的类型就是指定运算返回的类型。尽管这里的返回值可以是任意我们选择的类型，但是一般情况下，返回值的类型就是和我们重载运算符相关的类的类型。这样方便在复合表达式中使用该运算符。其中的参数列表由几个因素来决定。这点我们会在下面进行讨论。

运算符函数可以是类的成员函数，也可以不是类的成员函数。非类成员函数的运算符函数通常都是类的友元函数。但是两种运算符函数重载的方式还是有区别的。下面将逐一进行描述。

注意：由于 C++中定义了很多的运算符，所以运算符重载涉及的范围是很广泛的。在本书中，我们不可能全部都覆盖到。更多关于运算符重载的信息，可以参见另外的 C++书籍：Osborne/McGraw-Hill.出版的《C++ 全书》（英文名为：C++:The complete reference）

必备技能 9.10 ：以成员函数方式进行运算符的重载

我们从一个简单的程序开始来讨论成员运算符函数。下面的程序中创建了一个名为 ThreeD 的类，用来维护一个对象在三维空间中的坐标。程序中针对 ThreeD 这个类，对+和=运算符进行了重载。我们可以仔细研究一下：

view plain

```
1. //为 ThreeD 类定义+和=运算符
2. #include <iostream>
3. using namespace std;
4. class ThreeD
5. {
6.     int x, y, z; //三维坐标
7. public:
8.     ThreeD()
9.     {
10.        x = y = z = 0;
11.    }
12.
13.    ThreeD(int i, int j, int k )
14.    {
15.        x = i;
16.        y = j;
17.        z = k;
18.    }
19.
20.    ThreeD operator+(ThreeD op2);
21.    ThreeD operator=(ThreeD op2);
22.    void show();
23. };
24. //重载+运算符
25. ThreeD ThreeD::operator +(ThreeD op2)
26. {
27.    ThreeD temp;
28.
29.    temp.x = x+ op2.x;
30.    temp.y = y+ op2.y;
31.    temp.z = z+ op2.z;
32.
33.    return temp; //返回一个新的对象，参数对象保持不变
34. }
35. //重载赋值运算符
36. ThreeD ThreeD::operator =(ThreeD op2)
37. {
```

```
38.     x = op2.x; //这里的等号是整数的赋值，保持原来的含义
39.     y = op2.y; //这里的等号是整数的赋值，保持原来的含义
40.     z = op2.z; //这里的等号是整数的赋值，保持原来的含义
41.
42.     return *this; //返回被修改的对象
43. }
44. //显示 X,Y,Z 坐标
45. void ThreeD::show()
46. {
47.     cout << x << " ";
48.     cout << y << " ";
49.     cout << z << "\n";
50. }
51. int main()
52. {
53.     ThreeD a(1,2,3), b(10,10,10),c;
54.
55.     cout << "Original value of a: ";
56.     a.show();
57.     cout << "Original value of b: ";
58.     b.show();
59.
60.     cout << "\n";
61.
62.     c = a+b; //把 a 和 b 相加
63.     cout << "Value of c after c = a + b: ";
64.     c.show();
65.
66.     cout << "\n";
67.
68.     c = a + b + c; //把 a,b 和 c 相加
69.     cout << "Value of c after c = a + b + c: ";
70.     c.show();
71.
72.     cout << "\n";
73.
74.     c = b = a; //多重赋值
75.     cout << "Value of c after c = b = a: ";
76.     c.show();
77.     cout << "Value of b after c = b = a: ";
78.     b.show();
79.
80.     return 0;
81.
```

```
82. }
```

程序的输出如下：

```
Original value of a: 1 2 3
Original value of b: 10 10 10
Value of c after c = a + b: 11 12 13
Value of c after c = a + b + c: 22 24 26
Value of c after c = b = a: 1 2 3
Value of b after c = b = a: 1 2 3
```

仔细阅读上面的程序，我们会惊奇地发现两个运算符函数都只有一个参数，尽管它们都是二目运算符。这种明显的不统一是因为当二目运算符被作为类的成员函数重载的时候，只需要为其显示地传入一个参数即可。另外的一个参数是通过隐式的 this 指针传递的。因此，下面的代码：

```
temp.x = x + op2.x;
```

中 x 指的就是 this->x，也就是调用这个运算符函数的对象的 x。所有运算符左侧的对象就是调用该运算函数的对象。运算符右侧的对象是被作为参数传入到运算符函数中的对象。

一般来说，当我们重载单目运算符函数的时候是不需要为其传入参数的。只有在使用双目运算符函数的时候才需要为其传入参数。（不能重载三目运算符 ?.）在这两种情况下，调用该运算符函数的对象都是通过 this 指针来传入到函数中的。

让我们仔细研究一下上面的程序，以便弄清楚重载运算符函数是如何工作的。我们从+运算符的重载开始。当两个 ThreeD 的对象通过+运算符来运算的时候，它们各自的坐标大小被加起来，正如程序展示的那样。注意，这个运算不会修改两个运算数的值。而是由运算符函数返回一个包含了运算结果的对象。为了能够理解为什么+运算符不能修改两个对象的值，我们可以考虑一下标准的算数中的+运算符，它是这样使用的：10 + 12。结果是 22，但是 10 和 12 都没有被修改。尽管没有规定来限制+运算符不能修改运算数的值，但是保持被重载运算符的含义和它的普通含义一致是一种很好的习惯。

还要注意上面的+运算符函数返回的是 ThreeD 类的对象。虽然它可以返回任意有效的 C++类型，但是返回一个 ThreeD 类的对象使得+运算符可以用于复合表达式，例如 a+b+c。这里，a+b 的结果是一个 ThreeD 的对象，然后再和 c 相加。如果 a+b 的结果是别的类型，那么这个表达式将不能正常工作。

和+运算符相比，赋值运算的确修改了一个参数的值。这也是赋值运算的本质。既然=运算符是由它左侧的对象来调用的，那么该运算符修改的也就正是左侧对象的值了。通常情况下，重载的赋值运算的返回值就是其左侧被修改后的对象。这点是为了和传统的=运算保持一致。例如，这样就可以写出下面的语句：

```
a = b = c;
```

赋值运算符必须返回 this 指向的对象，也就是赋值运算符左侧的对象。这样做是为了可以编写链式的赋值语句。赋值运算符是 this 指针的一个重要用武之地。

在前面的程序中，其实是没有必要对赋值运算进行重载的。这是因为 C++提供的缺省的赋值运算就可以完成这样的功能。（正如我们在前面学习过的，缺省的赋值操作就是进行逐位的拷贝。）这里对赋值运算进行重载只是为了展示重载的正确过程。一般来说，我们只有在缺省的赋值操作不能满足我们要求的时候才进行重载。由于缺省的赋值操作对 ThreeD 这个类来说是足够的，在后面的示例程序中我们就不再对其进行重载了。

顺序很重要

在进行双目运算符重载的时候，我们要记住，运算数的顺序很重要。不同的顺序产生不同的结果。例如，A+B 中的两个运算数是可以交换的，但是 A-B 则不能。因此，当实现运算数不可交换的运算符的重载的时候，我们必须清楚哪个运算数在左侧，哪个在右侧。例如，下面是针对 ThreeD 实现减法运算：

view plain

```
1. //重载减法
2. ThreeD ThreeD::operator -(ThreeD op2)
3. {
4.     ThreeD temp;
5.
6.     temp.x = x -op2.x;
7.     temp.y = y -op2.y;
8.     temp.z = z -op2.z;
9.
10.    return temp;
11. }
```

请记住，调用运算符函数的对象是运算符左侧的运算数。右侧的是被作为参数显示传递给运算符函数的。

必备技能 9.11：以非成员函数的方式重载运算符函数

除了前面讨论的采用类的成员函数的方式重载运算符外，我们还可以以非成员函数的方式来重载运算符，那就是通过类的友元函数。我们在前面已经讨论到友元函数没有 this 指针。因此，当使用友元函数的方式来重载运算符的时候，对于双目运算符来说两个运算数都是要显示地通过参数的方式传递给运算符函数的；对于单目运算符来说，那一个参数也是要显示地通过参数的方式传递给运算符函数的。但是有以下的几个运算符是我们不能通过友元函数的方式来进行重载的：=，()，[]和->。

下面的程序展示了如果使用友元函数的方式来实现对 ThreeD 类的+运算符的重载：

view plain

```cpp
//使用友元函数实现运算符的重载
#include <iostream>
using namespace std;
class ThreeD
{
    int x, y, z;
public:
    ThreeD()
    {
        x = y = z;
    }
    ThreeD (int i, int j , int k)
    {
        x = i;
        y = j;
        z = k;
    }
    friend ThreeD operator+(ThreeD op1, ThreeD op2);
    void show();
};
// operator+()是友元函数
ThreeD operator+(ThreeD op1, ThreeD op2)
{
    ThreeD temp;
    temp.x  = op1.x + op2.x;
    temp.y  = op1.y + op2.y;
    temp.z  = op1.z + op2.z;
    return temp;
}
// 显示 x, y, z 坐标
void ThreeD::show()
{
    cout << x << ",";
    cout << y << ",";
    cout << z << "\n";
}
int main()
{
    ThreeD a(1,2,3), b(10,10,10), c;
    cout << "Original valud of a: ";
    a.show();
    cout << "Original value of b: ";
    b.show();
    cout <<"\n";
```

```
45.     c = a + b; // 把 a 和 b 相加
46.     cout << "Value of c after c = a +b : ";
47.     c.show();
48.     cout << "\n";
49.     c = a + b + c; // 把 a，b，c 相加
50.     cout << "Value of c after c = a + b + c: ";
51.     c.show();
52.     cout << "\n";
53.     c = b = a; // 多重赋值
54.     cout << "Value of c after c = b = a : ";
55.     c.show();
56.     cout << "Value of b after c = b = a: ";
57.     b.show();
58.     return 0;
59. }
```

上面程序的输出为：

Original valud of a: 1,2,3

Original value of b: 10,10,10

Value of c after c = a +b : 11,12,13

Value of c after c = a + b + c: 22,24,26

Value of c after c = b = a : 1,2,3

Value of b after c = b = a: 1,2,3

从上面的程序中可以看出，重载的+运算符的两个运算数都是通过参数传递的方式显示地传入到 operator+()函数中的。左侧的参数传递给参数 op1，右侧的参数传递给了参数 op2。

在许多情况下，使用友元函数的方式来重载运算符和使用类的成员函数的方式相比并不能带来明显的好处。但是，有一种情形使用友元函数会非常有效：当我们希望内置类型对象出现在双目运算符的左侧的时候。这是为什么呢？我们都知道，调用成员函数的对象是通过 this 指针传入到函数中的。针对双目运算符的情况，这个对象就是运算符左侧的运算数。所以只要运算符左侧的运算数明确定义了这种运算符的含义就是可以的。例如，有对象 T，假定我们已经为其定义了赋值运算符和+运算符的操作，那么下面的语句就是有效的：

T = T + 10; //有效的语句

这是因为对象 T 是出现在运算符的左侧的，由它来调用+运算符函数来完成加法的运算。然而，下面的语句则是无效的：

T = 10 + T; //无效的语句

上面这个语句的问题就在于出现在+运算符左侧的是一个整形数，一种内置的数据类型，其没有明确定义如何和对象 T 的类型进行加法运算。解决这个问题的方法就是使用两个友元函数来重载+运算。这样以来，重载的两个函数会显式地传入两个运算数，可以像其它重载的函数那样基于参数的类型来被调用。这两个友元函数中的一个用来处理 T 类型的对象和整数相加的操作，另外一个用来处理整数和 T 类型的对象的加法。这样以来，通过使用友元函数的方式就使得内置类型的数据也可以出现在运算符的左侧及右侧。下面的程序就演示了这种技术。其中定义了针对 ThreeD 类定义了两版本的+运算。它们都是完成整数和 ThreeD 对象相加的功能。其中整数是可以出现在运算符的左侧及右侧的。

view plain

```
1. //重载加号运算，实现 整数+对象 和 对象+ 整数
2. #include <iostream>
3. using namespace std;
4. class ThreeD
5. {
6.     int x, y ,z; //三维坐标
7. public:
8.     ThreeD()
9.     {
10.         x = y = z = 0;
11.     }
12.     ThreeD ( int i, int j , int k )
13.     {
14.         x = i;
15.         y = j;
16.         z = k;
17.     }
18.     friend ThreeD operator+(ThreeD op1, int op2); // 对象 + 整型数
19.     friend ThreeD operator+(int op1, ThreeD op2); // 整型数 + 对象
20.     void show();
21. };
22. //实现 ThreeD + 整型数
23. ThreeD operator+(ThreeD op1, int op2)
24. {
25.     ThreeD temp;
26.     temp.x = op1.x + op2;
27.     temp.y = op1.y + op2;
28.     temp.z = op1.z + op2;
29.     return temp;
30. }
31. //实现 整型数 + ThreeD
32. ThreeD operator+(int op1, ThreeD op2)
33. {
```

```
34.     ThreeD temp;
35.
36.     temp.x = op1 + op2.x;
37.     temp.y = op1 + op2.y;
38.     temp.z = op1 + op2.z;
39.     return temp;
40. }
41. //显示 x, y, z 坐标
42. void ThreeD::show()
43. {
44.     cout << x << ", ";
45.     cout << y << ", ";
46.     cout << z << "\n ";
47. }
48. int main()
49. {
50.     ThreeD a(1,2,3), b;
51.     cout << "Origianl value of a: ";
52.     a.show();
53.     cout << "\n";
54.     b = a + 10; //对象 + 整型数
55.     cout << "Value of b after b = a + 10 : ";
56.     b.show();
57.     cout << "\n";
58.     b = 10 + a; // 整型数 + 对象
59.     cout << "Value of b after b = 10 + a : ";
60.     b.show();
61.     return 0;
62.
63. }
```

上面程序的输出如下：

Origianl value of a: 1, 2, 3

Value of b after b = a + 10 : 11, 12, 13

Value of b after b = 10 + a : 11, 12, 13

由于在上面的程序中个，我们对+()进行了两次重载，这样就可以处理整型数和 ThreeD 对象相加的两种情况了。

使用友元函数来重载单目运算符

我们还可以使用友元函数来实现对单目运算符的重载。然而，如果我们使用友元函数来重载++或者--运算符，我们必须把运算数以引用的方式传递到对应的运算符函数中。由于引用是一种隐式的指针，那么在函数中对形参的修改都会影响到传入的实参。因此采用传递引用的方式可以实现对象的自增或者自减运算。当我们使用友元函数来重

载自增运算符的时候，其前缀形式需要一个参数，也就是运算数；而后缀形式则需要两个参数。其中第二个参数是一个整型数，但是这个参数不会被使用。下面的代码段就展示了如何使用友元函数来对前缀和后缀的自增运算进行重载：

view plain

```
1. //采用友元函数的方式重载前缀自增运算符，其中使用到传递引用*/
2. ThreeD operator++(ThreeD &op1)
3. {
4.     op1.x++;
5.     op1.y++;
6.     op1.z++;
7.     return op1;
8. }
9. //采用友元函数的方式重载后缀自增运算符，其中使用到传递引用
10. ThreeD operator++(ThreeD &op1, int notused)
11. {
12.     ThreeD temp = op1;
13.     op1.x++;
14.     op1.y++;
15.     op1.z++;
16.     return temp;
17. }
```

练习：

1. 以非成员函数的方式重载双目运算符函数，需要几个参数？

2. 当使用非成员函数的方式重载++运算符的时候，应该以什么样的方式进行参数传递？

3. 使用友元函数的方式重载运算符函数的一个好处就是可以让内置类型的数据出现在运算符的左侧，对吗？

关于运算符重载的小提示和一些限制

针对类定义的重载运算符可以和该运算符作用于内置数据类型含义没有什么关系。例如，引用于 cout 和 cin 的<<和>>运算符就和该运算符作用于整型数的含义没有什么关系。然而，为了程序的结构化和代码的可阅读性，重载的运算符应该尽可能地反映出该操作的原本含义。例如，ThreeD 类的+运算符就和整型数的+运算符在概念上相近。如果我们针对别的类定义+运算来完成||运算符的功能也是可以的。只是这样作并没有什么好处。这里想要说明的就是尽管我们可以在重载运算符的时候随意地定义其功能，但是出于代码清晰度的考虑，我们最好还是在定义它的新功能时，保持其和原始功能相关。

专家答疑

问：当对关系运算符进行重载的时候，有什么需要特别注意的地方吗？

答：重载诸如==或者<之类的关系运算符的方法和我们学习的方法一样简单的。但是，有一个小问题需要注意。众所周知，重载的运算符函数通常都是返回相关类的对象。然而，重载的关系运算符函数却应该返回 true 或者 false。这点是和普通的关系运算符保持一致的。这样可以使得重载的关系运算符函数也可以在条件表达式中使用。这点对于逻辑运算符的重载也是一样的。

下面的程序演示了如果针对 ThreeD 类来重载==运算符。通过这个例子展示关系运算符重载的实现。

view plain

```
1. bool ThreeD::operator ==(ThreeD op2)
2. {
3.     if ( (x == op2.x )&&(y == op2.y )&&( z == op2.z))
4.     {
5.         return true;
6.     }
7.     else
8.     {
9.         return false;
10.    }
11. }
```

经过上面的对==运算符的重载后，下面的代码就是完全有效的：

```
ThreeD a(1,2,3), b(2,2,2) ;
if ( a == b)
  cout << " a equals b \n";
else
  cout << "a does not equal b \n";
```

运算符的重载是有一些限制的。第一，不能修改运算符的优先级。第二，尽管运算数可以被忽略，但是运算数的数量也是不能修改的。最后，除了函数调用运算符外，运算符函数不支持缺省参数。

几乎所有的 C++运算符都是可以被重载的。这也包括特殊的运算符，例如数组索引运算符[]，函数调用运算符()以及->运算符。但是下面的运算符是不能被重载的：

. :: .* ?

其中上面的.*是一种很特殊的运算符，它的使用方法不在本书中进行 讨论。

工程 9-1 创建集合类

运算符重载使得我们可以创建与 C++编程环境完全兼容的类。例如， 通过定义必要的运算符函数，我们可以像使用内置数据类型那样使用类。我们可以对该类的对象使用运算符，还可以在表达式中使用该类的对象。下面我们通过创建一个名为 Set 的类，定义 Set 类型来演示如何创建与 C++编程环境想兼容的类。

在开始之前，我们必须先明确这里说的 Set 的具体含义。在这个工程中，Set（集合）被定义是一系列互不相同的元素的集合。也就是说，在任何一个给定的 Set(集合）中没有两个相同的元素。集合中元素的顺序是无关紧要的。因此集合{A,B,C}和{A,C,B}是同一个集合。集合还可以为空。

集合支持很多的操作。本工程中实现了集合的下列操作：

●·为集合增加元素

●·从集合中删除一个元素

●·集合的并集

●·集合的差集

其中，增加元素到集合中和从集合中删除元素操作的含义是很明显的。这里需要对并集和差集进行解释。两个集合的并集就是含有这两个集合全部元素的集合。（当然，其中不含有重复的元素）。我们将使用+运算来完成并集的运算。

集合的差集是仅含有第一个集合中那些不属于第二个集合的元素构成的集合。我们将使用-运算来实现集合的差集。例如，有两个集合 S1 和 S2, 把 S2 从 S1 中删除后的集合构成 S3 的语句如下：

S3 = S1 - S2;

如果 S1 和 S2 中的元素是完全相同的，则 S3 集合为空的集合。类 Set 中还有一个成员函数为 isMember(),用来判断一个元素是否属于该集合。当然，还有一些别的和集合相关的运算。其中一些会在练习题中找到，其余的我们可以尝试自己编写。

为了简单起见，Set 类存储的是字符。但是用于存储其它类型元素的类在实现原理上都是相同的。

步骤：

1. 创建一个新文件，名称为 Set.cpp.
2. 通过声明 Set 类，如下：

view plain

```
const int MaxSize = 100;
class Set
{
    int len; //元素的数量
    char members[MaxSize]; //采用数组来存储集合中的元素
    /* find（）函数是私有的，因为在 Set 类之外是不使用的*/
    int find(char ch);  //查找一个元素
public:
```

```
9.      //构造一个空的集合
10.     Set()
11.     {
12.         len = 0;
13.     };
14.
15.     //返回集合中元素的数量
16.     int getLength()
17.     {
18.         return len;
19.     }
20.
21.     void showset(); // 显示出集合中的元素
22.     bool isMember( char ch ); // 检查是否是集合中的元素
23.
24.     Set operator +(char ch); // 为集合增加元素
25.     Set operator -(char ch); // 删除一个元素
26.
27.     Set operator +(Set ob2); //求集合的并集
28.     Set operator -(Set ob2); //求集合的差集
29. };
```

集合中采用数组 members 来存储元素。集合中元素的数量保存在 len 变量中。集合中最多可以含有 MaxSize 个元素，也就是 100 个元素。（如果需要处理更大的集合，可以自行修改这个值。）

Set 类的构造函数生成一个空的集合，即不含有任何元素。我们没有必要再创建别的构造函数或者显示地定义其拷贝构造函数了，因为缺省的逐位拷贝的操作是满足我们要求的。函数 getLength()返回 len 的值，也就是集合中实际的元素的数量。

3. 从私有函数 find()开始定义其成员函数，如下：

view plain

```
1.  /* 返回指定元素 ch 的索引;
2.     如果没有找到，则返回-1 */
3.  int Set::find(char ch)
4.  {
5.      int i;
6.      for ( i = 0; i < len ; i++)
7.      {
8.          if ( ch == members[i] )
9.          {
10.             return i;
11.         }
```

```
    }

    return -1;
}
```

上面的函数用来判断传入的字符 ch 是否是集合中的元素。如果在集合中找到了该元素，函数返回其索引；否则返回-1。我们在类 Set 范围之外不使用这个函数，因此它是私有的。正如我们在前面讲过的那样，成员函数可以是仅供该类使用的私有函数。私有函数只能由该类中的其他成员函数来调用。

4. 为类增加 showset()函数，如下：

view plain

```
//显示集合中的元素
void Set::showset()
{
    cout << " ( ";
    for( int i = 0; i < len ; i++)
    {
        cout << members[i] << " ";
    }

    cout << ")";
}
```

这个函数显示出集合的内容。

5. 为类增加 isMember()函数，用来判断一个字符是否在集合中。如下：

view plain

```
/* 如果指定的字符时集合中的元素则返回 true;
   否则返回 false */
bool Set::isMember(char ch)
{
    if ( find(ch) != -1 )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

这个函数中调用了 find()来判断 ch 是否是集合中的元素。如果是，isMember()函数返回 true；否则返回 false。

6. 下面开始为 Set 类增加运算符。从加法运算符开始。对 Set 类的+运算符进行重载，实现往集合中增加一个元素，如下。

view plain

```
1. //为集合中增加一个元素
2. Set Set::operator +(char ch)
3. {
4.     Set newset;
5.
6.     if ( len == MaxSize )
7.     {
8.         cout << "Set is full . /n";
9.         return *this;
10.    }
11.
12.    newset = *this;//复制当前的集合
13.
14.    //检查 ch 是否已经在集合中了
15.    if(find(ch) == -1 )
16.    {
17.        //如果不在集合中，则增加该元素到集合中
18.        newset.members[newset.len] = ch;
19.        newset.len++;
20.    }
21.
22.    return newset; //返回数据更新后的集合
23. }
```

这个函数我们需要仔细研究一下。首先，生成了一个新的集合。这个集合将用来存储原集合中的元素再加上元素 ch。在把 ch 增加到集合中之前，先检查是否有足够的空间来存储增加的字符。如果有足够的空间可用于存储新增的字符，则把原始的集合赋值给新的集合。接着调用函数 find()来判断 ch 是否已经在集合中了。如果不在，则把 ch 增加到新的集合中，并让 len 自增。在各种情况下，函数都是返回 newset 这个新的集合。因此，原始的集合在该函数中没有被修改，保持不变。

7. 重载-运算，以实现从集合中删除一个元素，如下：

view plain

```
// 从集合中删除一个元素
Set Set::operator -(char ch)
{
    Set newset;

    int i = find(ch); //如果 ch 不在集合中，则 i 为-1

    //把其它的元素复制到新的集合中
    for ( int j = 0; j < len ; j ++)
    {
        if ( j != i )
        {
            newset = newset + members[j];
        }
    }

    return newset;
}
```

函数一开始创建了一个空的集合。然后调用 find()函数来取得 ch 在原始集合中的索引。find()函数在 ch 不是集合中元素的时候返回-1。接着，通过循环把原始集合中除了索引等于 find()返回值之外的所有元素都加入到新的集合中。这样，新的集合中就含有原始集合中除了 ch 以外的全部元素了。如果 ch 不是原始集合中的元素，新的集合和原始集合是相等的。

8. 重载+和−，来求集合的并集和差集，如下：

view plain

```
//求并集
Set Set::operator +(Set ob2)
{
     Set newset = * this; //拷贝第一个集合

     //把第二个集合中不属于第一个集合的元素拷贝到新的集合中
     for ( int i = 0; i < ob2.len; i++)
     {
         newset = newset + ob2.members[i];
     }

     return newset; //返回并集
}

//求差集
Set Set::operator -(Set ob2)
{
```

```
18.
19.     Set newset = * this; //拷贝第一个集合
20.
21.     //从中减去第二个集合中的元素
22.     for ( int i = 0; i < ob2.len; i++)
23.     {
24.         newset = newset - ob2.members[i];
25.     }
26.
27.     return newset; //返回差集
28. }
```

我们可以看到，在这两个函数中我们用到了前面定义的+和-运算来辅助完成求并集和差集的操作。就求并集的操作来说，先生成了一个含有第一个集合所有元素的新集合。然后把第二个集合中的元素增加到了新的集合中。由于+运算符只会把不在集合中的元素加入到集合中，因此得到的新的集合就是不含有重复元素的两个集合的并集。Set 类的求差集运算符则是把第二个集合中的元素从第一个中删除掉。

9. 下面是 Set 了以及 main()函数构成的演示 Set 类使用方法的完整程序：

view plain

```
1. /* 构成-1
2.    元素为字符的集合类
3. */
4. #include <iostream>
5. using namespace std;
6.
7. const int MaxSize = 100;
8. class Set
9. {
10.     int len; //元素的数量
11.     char members[MaxSize]; //采用数组来存储集合中的元素
12.     /* find（）函数是私有的，因为在 Set 类之外是不使用的*/
13.     int find(char ch);  //查找一个元素
14. public:
15.     //构造一个空的集合
16.     Set()
17.     {
18.         len = 0;
19.     };
20.
21.     //返回集合中元素的数量
22.     int getLength()
```

```
    {
        return len;
    }

    void showset(); // 显示出集合中的元素
    bool isMember( char ch ); // 检查是否是集合中的元素

    Set operator +(char ch); // 为集合增加元素
    Set operator -(char ch); // 删除一个元素

    Set operator +(Set ob2); //求集合的并集
    Set operator -(Set ob2); //求集合的差集

};


/* 返回指定元素 ch 的索引;
   如果没有找到，则返回-1 */
int Set::find(char ch)
{
    int i;
    for ( i = 0; i < len ; i++)
    {
        if ( ch == members[i] )
        {
            return i;
        }
    }

    return -1;
}

//显示集合中的元素
void Set::showset()
{

    cout << " ( ";
    for( int i = 0; i < len ; i++)
    {
        cout << members[i] << " ";
    }

    cout << ")";
}

```

```
/* 如果指定的字符时集合中的元素则返回 true;
   否则返回 false */

bool Set::isMember(char ch)
{
    if ( find(ch) != -1 )
    {
        return true;
    }
    else
    {
        return false;
    }
}

//为集合中增加一个元素
Set Set::operator +(char ch)
{
    Set newset;

    if ( len == MaxSize )
    {
        cout << "Set is full . /n";
        return *this;
    }

    newset = *this;//复制当前的集合

    //检查 ch 是否已经在集合中了
    if(find(ch) == -1 )
    {
        //如果不在集合中，则增加该元素到集合中
        newset.members[newset.len] = ch;
        newset.len++;
    }

    return newset; //返回数据更新后的集合
}

// 从集合中删除一个元素
Set Set::operator -(char ch)
{
    Set newset;

```

```
60.     int i = find(ch); //如果 ch 不在集合中，则 i 为-1
61.
62.     //把其它的元素复制到新的集合中
63.     for ( int j = 0; j < len ; j ++)
64.     {
65.         if ( j != i )
66.         {
67.             newset = newset + members[j];
68.         }
69.     }
70.
71.     return newset;
72. }
73.
74. //求并集
75. Set Set::operator +(Set ob2)
76. {
77.      Set newset = * this; //拷贝第一个集合
78.
79.      //把第二个集合中不属于第一个集合的元素拷贝到新的集合中
80.      for ( int i = 0; i < ob2.len; i++)
81.      {
82.          newset = newset + ob2.members[i];
83.      }
84.
85.      return newset; //返回并集
86. }
87.
88. //求差集
89. Set Set::operator -(Set ob2)
90. {
91.
92.     Set newset = * this; //拷贝第一个集合
93.
94.     //从中减去第二个集合中的元素
95.     for ( int i = 0; i < ob2.len; i++)
96.     {
97.         newset = newset - ob2.members[i];
98.     }
99.
100.     return newset; //返回差集
101. }
102.
103. //展示集合类 Set 的使用
```

```
104. int main()
105. {
106.     //构建空的集合
107.     Set s1;
108.     Set s2;
109.     Set s3;
110.
111.     s1 = s1 + 'A';
112.     s1 = s1 + 'B';
113.     s1 = s1 + 'C';
114.
115.     cout << "s1 after adding A B C: ";
116.     s1.showset();
117.
118.     cout << "/n/n";
119.
120.     cout << "Testing for membership using isMember()./n";
121.     if ( s1.isMember('B'))
122.     {
123.         cout << "B is a member of s1./n";
124.     }
125.     else
126.     {
127.         cout << "B is not a member of s1./n";
128.     }
129.
130.     if ( s1.isMember('T'))
131.     {
132.         cout << "T is a member of s1./n";
133.     }
134.     else
135.     {
136.         cout << "T is not a member of s1./n";
137.     }
138.
139.     cout << "/n";
140.
141.     s1 = s1 - 'B';
142.     cout << "s1 after s1 = s1 - 'B' ";
143.     s1.showset();
144.
145.     cout << "/n";
146.
147.     s1 = s1 - 'A';
```

```
148.     cout << "s1 after s1 = s1 - 'A' ";
149.     s1.showset();
150.
151.     cout << "/n";
152.
153.     s1 = s1 - 'C';
154.     cout << "s1 after s1 = s1 - 'C' ";
155.     s1.showset();
156.
157.     cout << "/n/n";
158.
159.     s1 = s1 + 'A';
160.     s1 = s1 + 'B';
161.     s1 = s1 + 'C';
162.     cout << "s1 after adding A B C: ";
163.     s1.showset();
164.
165.     cout << "/n/n";
166.
167.     s2 = s2 + 'A';
168.     s2 = s2 + 'X';
169.     s2 = s2 + 'W';
170.     cout << "s2 after adding A X W: ";
171.     s2.showset();
172.
173.     cout << "/n/n";
174.
175.     s3 = s1 + s2;
176.     cout << "s3 after s3 = s1 + s2: ";
177.     s3.showset();
178.
179.     cout << "/n/n";
180.
181.     s3 = s3 - s1;
182.     cout << "s3 after s3 - s1 : ";
183.     s3.showset();
184.
185.     cout << "/n/n";
186.
187.
188.     s2 = s2 - s2; /*清空 s2 */
189.     cout << "s2 after s2 = s2 - s2: ";
190.     s2.showset();
191.
```

```
192.     cout << "/n/n";
193.
194.     s2 = s2 + 'C';
195.     s2 = s2 + 'B';
196.     s2 = s2 + 'A';
197.
198.     cout << "s2 after adding C B A: ";
199.     s2.showset();
200.
201.     return 0;
202. }
```

上面程序的输出结果如下：

```
s1 after adding A B C:  ( A B C )

Testing for membership using isMember().
B is a member of s1.
T is not a member of s1.

s1 after s1 = s1 - 'B'  ( A C )
s1 after s1 = s1 - 'A'  ( C )
s1 after s1 = s1 - 'C'  ( )

s1 after adding A B C:  ( A B C )

s2 after adding A X W:  ( A X W )

s3 after s3 = s1 + s2:  ( A B C X W )

s3 after s3 - s1 :  ( X W )

s2 after s2 = s2 - s2:  ( )

s2 after adding C B A:  ( C B A )
```

练习

1. 什么是拷贝构造函数？它在什么时候被调用？写出拷贝构造函数的通用形式。

2. 请解释一下当函数返回一个对象的时候发生了什么？特别解释析构函数是在什么时候被调用的。
3. 有如下代码段：

view plain

```
class T
{
    int i,j;
public:
    int sum()
    {
        return i + j;
    }
}
```

请使用 this 指针重写上面的代码。

4. 什么是结构体？ 什么是联合体？
5. 在类的成员函数中，*this 指的什么？
6. 什么是友元函数？
7. 写出重载二目运算符函数的通用形式。
8. 怎样做才能实现涉及类类型和内置类型的运算？
9. ？运算符可否被重载？运算符的优先级别是否可以被改变？
10. 就工程 9-1 中的 Set 类，定义< 和 >运算用来判断一个集合是其子集或者是超集。其中，
    < 运算符在其左侧的集合是其右侧集合的子集的时候返回 true；否则返回 false。
    > 运算在其左侧集合是其右侧集合的超级的时候返回 true；否则返回 false。
11. 为 Set 类定义&运算，用于计算两个集合的交集。
12. 就工程 9-1 中的 Set 类，为其增加其它的运算。比如，增加|运算，用于计算两个集合的对称差分。对称差分是由两个集合中不属于他们交集的那些元素构成的集合。

第十篇 继承、虚函数和多态性

本章中我们将讨论和面向对象编程直接相关的三个 C++特性：继承，虚函数和多态。

继承是允许一个类继承另外一个类特征的特性。使用继承，我们通过一个类来定义一组相关对象的通用特征。这个类可以被另外一些更具体的类继承，每个类都可以增加其特有的属性。而虚函数正是以类的继承为基础的。虚函数支撑了多态性，也就是"同一个接口，多种实现方式"的面向对象的思想。

必备技能 10.1：继承的基础知识

在 C++中，被继承的类称为基类。继承别的类的类叫做派生类。因此可以说派生类是基类的特殊化。派生类继承了基类中的所有成员，并增加了自己所特有的成员。继承是由基类和派生类共同作用实现的。这是通过在声明派生类的时候指明其基类来完成的。下面我们通过一个简短的示例程序来讨论和继承相关的几个特性。

下面的程序创建了一个基类叫做 TwoDShape，用于存储一个二维对象的宽带和高度，还定义了一个派生类叫做 Triangle。特别注意下面类 Triangle 的声明方式。

view plain

```
1. // 一个简单的类的继承示例
2. #include <iostream>
3. #include <csting>
4. #include using namespace std;
5. //二维形状类 class TwoDShape
6. {
7. public:
8.     double width;
9.     double height;
10.    void showDim()
11.    {
12.        cout << "Width and height are " << width << " and " << height
   << "\n";
13.    }
14. };
15.
16.
17. //Triangle 类是从 TwoDShape 类中继承而来的
18. class Triangle : public TwoDShape
19. {
20. public:
21.    char style[20];
22.    double area()
23.    {
24.        return width * height / 2;
25.    }
26.
27.    void showStyle()
28.    {
29.        cout << "Triangle is " << style << "\n";
30.    }
31. };
32.
33.
```

```
34. int main()
35. {
36.     Triangle t1;
37.     Triangle t2;
38.
39.     t1.width = 4.0;
40.     t1.height = 4.0;
41.     strcpy(t1.style,"isosceles");
42.
43.
44.     t2.width = 8.0;
45.     t2.height = 12.0;
46.     strcpy(t2.style,"right");
47.
48.
49.     cout << "Info for t1:\n";
50.     t1.showStyle();
51.     t1.showDim();
52.     cout << "\n";
53.
54.
55.     cout << "Info for t2:\n";
56.     t2.showStyle();
57.     t2.showDim();
58.     cout << "Area is " << t2.area() << "\n";
59.
60.     return 0;
61. }
```

上面程序的输出结果如下：

```
Info for t1:Triangle is isosceles
Width and height are 4 and 4
Info for t2:Triangle is right
Width and height are 8 and 12Area is  48
```

其中，TwoDShape 类定义了二维形状的通用属性，比如正方形，长方形，三角形等等。而 Triangle（三角形）类则是 TwoDShape 类的具体的例子。三角形类包含了 TwoDShape 这个类的所有成员，并且还增加了字段 style，函数 area() 和 showStyle()。关于三角形类型的描述存储在 style 字段中，area() 函数用于计算并返回三角形的面积，showStyle() 函数用于显示三角形的类型。

下面的代码行就显示了 Triangle 类是怎么继承 TwoDShape 类的：

```
class Triangle : public TwoDShape
```

其中，TwoDShape 是 Triangle 类所继承的类，也就是基类。正如这个示例程序演示的那样，类继承的语法是很简单的，也是很方便使用的。

由于 Triangle 类包含了它的基类 TwoDShape 的所有成员,因此它就可以在 area()函数中访问 width 和 height。同样，在 main()函数中，t1,t2 都可以直接访问 width 和 height，就像它们是 Triangle 的一部分一样。图 10-1 从概念上刻画了 TwoDShape 和 Triangle 类的直接关系。

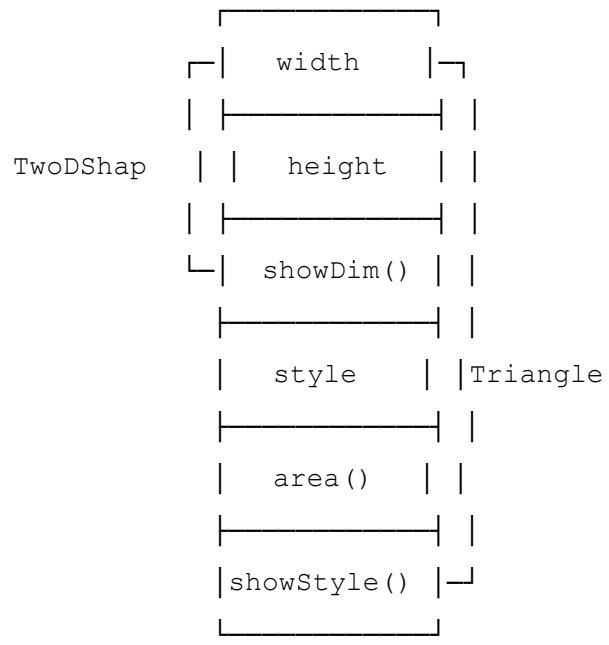


图 10-1 Triangle 类的构成

另外的一点：尽管 TwoDShape 是 Triangle 类的基类，它仍然是一个完全对立的类。作为一个派生类的基类并不是说这个类就不能单独使用了。

声明继承关系的通用形式如下：

```
class 派生类：访问限定符 基类
{
    // 派生类的定义体
}
```

其中的访问限定符是可选的。然而，如果需要明确指定，只能是 public, private 或者是 protected。我们将在本章的后面对它们进行更多的讨论。到目前为止，我们所有的继承关系都将使用 public。使用 public 的继承关系意味着基类中所有的公有的成员在派生类中依然是公有的。

继承的一个主要好处就是一旦我们创建了一组对象所共有特性的基类后，我们就可以创建任意数量的具体特殊性质的派生类。每个派生类都可以精确地根据分类来进行剪裁。例如，下面就是另外一个从 TwoDShape 类继承的类 Rectangle，它封装了对矩形的操作：

view plain

```
class Rectangle:public TwoDShape
{
public:
    bool isSquare()
    {
        if ( width == height )
        {
            return true;
        }
        else
        {
            return false;
        }
     }

    double area()
    {
         return width * height;
    }
};
```

这里的 Rectangle 类包含了 TwoDShape 类的全部成员，并增加了函数 isSquare()，用来判断该矩形是不是正方形。同时，增加了函数 area()，用来计算矩形的面积。

继承与成员的访问权限

正如我们在第八篇中学习到的那样，类的成员经常会被声明为是私有的，以避免未经授权的使用或者篡改。继承机制并没有推翻私有访问权限的限制。然而，尽管派生类中包含了基类的所有成员，但是它不能访问基类中的那些私有成员。例如，如果在 TwoDShape 类中 width 和 height 都是私有的，如下代码所示，那么 Triangle 类就是不能访问它们的。

view plain

```
//二维形状类
class TwoDShape
{
    // width 和 height 是私有的
    double width;
    double height;

    public: void showDim()
    {
```

```
10.         cout << "Width and height are " << width << " and " << height
    << "\n";
11.     }
12. };
13. //Triangle 类是从 TwoDShape 类中继承而来的
14. class Triangle : public TwoDShape
15. {
16. public:
17.     char style[20];
18.     double area()
19.     {
20.         return width * height / 2; //错误，Triangle 类不能访问基类的私有成
    员
21.     }
22.     void showStyle()
23.     {
24.          cout << "Triangle is " << style << "\n";
25.     }
26. };
```

上面中 Triangle 类将不能被成功编译。这是因为其中的 area()函数中引用了基类的私有数据成员 width 和 height，这是违反规则的。由于 width 和 height 成员现在是私有的，只有 TwoDShape 类本身可以访问它们，其派生类是无权访问它们的。

乍看起来，我们可能会认为派生类不能访问基类的私有数据成员貌似是一种过分严格的限制，因为在很多情况下，这将使得我们无法访问私有的成员。幸运的是，实际情况并非如此，C++提供了几种不同的方法来解决这个问题。其中之一就是使用 protected（保护）成员，这点我们将在下面的小节中进行讨论。还有一种方法就是提供公有的函数来实现对私有数据的访问。正如我们在前面篇章中看到的那样，C++程序员通常会通过公有的函数来实现对私有数据成员的访问。提供访问私有数据成员的函数被称为是访问函数。下面代码中，我们就采用了访问函数的方式来提供对 TwoDShape 类的私有数据成员的访问：

view plain

```
1. // 一个简单的类的继承示例
2. #include <iostream>
3. #include <cstring>
4. #include using namespace std;
5.
6. //二维形状类
7. class TwoDShape
8. {
```

```
9.      double width;
10.     double height;
11.     public: void showDim()
12.     {
13.         cout << "Width and height are " << width << " and " << height
    << "\n";
14.     }
15.     double getWidth()
16.     {
17.         return width;
18.     };
19.
20.     double getHeight()
21.     {
22.          return height;
23.     };
24.     void setWidth(double w )
25.     {
26.         width = w ;
27.     };
28.
29.     void setHeight(double h )
30.     {
31.          height = h;
32.     };
33. };
34. //Triangle 类是从 TwoDShape 类中继承而来的
35. class Triangle : public TwoDShape
36. {
37. public:
38.     char style[20];
39.     double area()
40.     {
41.         //return width * height / 2;
42.         return getWidth() * getHeight() / 2;
43.     }
44.
45.     void showStyle()
46.     {
47.         cout << "Triangle is " << style << "\n";
48.     }
49. };
50. int main()
51. {
```

```
52.     Triangle t1;
53.     Triangle t2;
54.
55.     //t1.width = 4.0;
56.     //t1.height = 4.0;
57.     t1.setWidth(4.0);
58.     t1.setHeight(4.0);
59.     strcpy(t1.style,"isosceles");
60.
61.     //t2.width = 8.0;
62.     //t2.height = 12.0;
63.     t2.setWidth(8.0);
64.     t2.setHeight(12.0);
65.     strcpy(t2.style,"right");
66.
67.     cout << "Info for t1:\n"; t1.showStyle(); t1.showDim();
68.     cout << "\n";
69.     cout << "Info for t2:\n"; t2.showStyle(); t2.showDim();
70.     cout << "Area is " << t2.area() << "\n";
71.
72.     return 0;
73. }
```

练习

1. 基类是通过怎样的方式被派生类继承的？
2. 派生类中是否包含了基类中的成员？
3. 派生类是否可以访问基类中私有的成员？

必备技能 10.2：对基类成员的访问控制

正如我们在前面所讨论到的那样，当一个类继承了另外一个类后，基类的所有成员也都会变成派生类的成员。然而，在派生类中，基类成员的可访问性是由派生类继承基类的方式来决定的，也就是由继承的访问限定符来决定的。派生类继承基类时的访问限定符只能是 public，private 或者 protected。当我们没有指定访问限定符的时候，如果基类是一个类，那么缺省的访问限定符就是 private。如果基类是一个结构体，则缺省的访问限定符是 public。下面，我们就来讨论一下 public 和 private 继承方式的区别。protected 的继承方式我们将在下节中进行讨论。

当派生类以公有的(public)的方式继承基类的时候，基类的所有公有成员就会变成是派生类的公有成员。然而，基类的私有成员则永远会是基类的私有成员，派生类是不能访问的。

例如，在下面的程序中，B 的公有成员变成了 D 的共有成员，因此可以被程序的其它部分访问。

view plain

```
1. //演示公有继承
2. #include <iostream>
3. using namespace std;
4. class B
5. {
6.     int i,j;
7. public:
8.     void set( int a, int b )
9.     {
10.         i = a;
11.         j = b;
12.     }
13.     void show()
14.     {
15.         cout <<  i << " " << j << "\n";
16.     }
17. };
18. class D : public B
19. {
20.     int k;
21. public:
22.     D(int x)
23.     {
24.          k = x;
25.     }
26.     void showk()
27.     {
28.         cout << k << "\n";
29.         // i = 10 ; //错误！i 在 B 类中是私有的成员，派生类是不能访问的。
30.     }
31. };
32. int main()
33. {
34.     D ob(3);
35.
36.     ob.set(1,2); //访问基类的共有成员
37.     ob.show();//访问基类的共有成员
38.     ob.showk();//访问派生类的共有成员
39.     return 0;
40. }
```

既然 set()和 show()函数在基类 B 中都是公有的，因此他们是可以在 main()函数中 D 类的对象上被调用的。i 和 j 在基类 B 中都是私有的，这是下面的代码行

// i = 10 ; //错误！i 在 B 类中是私有的成员，派生类是不能访问的。

被注释掉的原因。D 类是不能访问 B 的私有成员的。

和公有继承想对应的就是私有继承。当以私有的方式继承基类的的时候，基类中所有的公有成员都在派生类中变成了私有成员。例如，下面的程序不能成功编译，就是因为 set（）和 show()现在对于 D 类来说都是私有的，因此不能在 main()函数中被调用。

view plain

```
1. //演示私有的继承方式，改程序不能被成功编译
2. #include <iostream>
3. using namespace std;
4. class B
5. {
6.     int i,j;
7. public:
8.     void set( int a, int b )
9.     {
10.        i = a;
11.        j = b;
12.    }
13.    void show()
14.    {
15.        cout <<  i << " " << j << "\n";
16.    }
17. };
18. class D : private B
19. {
20.    int k;
21. public:
22.    D(int x)
23.    {
24.         k = x;
25.    }
26.    void showk()
27.    {
28.        cout << k << "\n";
29.        // i = 10 ; //错误！i 在 B 类中是私有的成员，派生类是不能访问的。
30.    }
31. };
32. int main()
33. {
```

```
34.     D ob(3);
35.
36.     ob.set(1,2); //访问基类的共有成员
37.     ob.show();//访问基类的共有成员
38.     ob.showk();//访问派生类的共有成员
39.     return 0;
40. }
```

小结：当基类以 private 的方式被继承的时候，其所有的共有成员都变成了派生类的私有成员。这就意味着，它们可以被派生类的成员调用，但是不能被程序的其它部分调用。

专家答疑

问：我知道在 Java 编程中使用父类和子类概念，这两个概念在 C++中有什么含义？

答：Java 中的父类就是 C++中的基类。Java 中的子类就是 C++中的派生类。通常我们在这两个语言中都能听到这两种术语。但是在本书中，我们将使用标准的 C++术语。另外，在 C#中也是使用基类和派生类的术语。

必备技能 10.3：使用保护的成员

正如我们所讨论的那样，基类的私有成员在派生类中是不能被访问的。这样一来，如果派生类想要访问基类的成员，似乎只能把这些成员在基类中声明为公有的了。但是，这样以来，程序中的其它代码也是可以访问这些公有成员的，这不是我们想要的。幸运的是，C++允许我们创建保护的成员，从而没有必要把派生类访问的成员在基类中声明为公有的。保护的成员在类的继承关系中是公有的，但是对于继承关系之外的代码来说又是私有的。

保护成员的创建是通过使用 protected 修饰符来完成的。当类的一个成员被声明为 protected 的时候，该成员就是私有的，只有一种情况例外。就是当该保护成员被继承的时候。此时，派生类是可以访问基类的保护成员的。因此，通过使用保护成员，我们就可以创建对于自身来说是私有的，但是派生类又可以访问的类的成员。protected 修饰符也可以用于结构体。

下面的程序演示了保护成员：

view plain

```
1.  //演示保护成员
2.  #include <iostream>
3.  using namespace std;
4.
5.  class B
6.  {
7.  protected:
8.      int i,j;
9.  public:
```

```
10.     void set( int a, int b )
11.     {
12.         i = a;
13.         j = b;
14.     }
15.
16.     void show()
17.     {
18.         cout <<  i << " " << j << "\n";
19.     }
20. };
21.
22. class D : public B
23. {
24.     int k;
25. public:
26.     //D 可以访问 B 类中的 i 和 j。这是因为他们是保护的，而非私有的。
27.     void setk()
28.     {
29.          k = i * j;
30.     }
31.
32.     void showk()
33.     {
34.         cout << k << "\n";
35.     }
36.
37. };
38.
39. int main()
40. {
41.     D ob;
42.
43.     ob.set(1,2); //访问基类的共有成员
44.     ob.show();//访问基类的共有成员
45.
46.     ob.setk();
47.     ob.showk();//访问派生类的公有成员
48.
49.     return 0;
50. }
```

其中由于 D 公有地继承了 B 类，并且 i 和 j 在 B 类中被声明为保护成员，所以 D 的函数 setk()就可以访问它们。如果 i 和 j 被声明为是 private 的，那么 D 就不能访问它们，程序也就不能成功编译了。

当派生类以公有的方式继承基类的时候，基类中的保护成员在派生类中也是保护成员。当基类被以私有的方式继承的时候，基类的保护成员在派生类中就变成了私有的成员了。

protected 修饰符在声明类的时候可以出现在任何位置，但是通常它是被写在类的私有成员之后，而在共有成员之前。因此，最常用的完整的类的声明形式如下：

```
class 类名
{
private：
    //私有成员,缺 省就是私有的。
protected：
    //保护成员
public：
    //共有成员
};
```

当然，其中的 protected 是一个可选项。

除了上面的类的保护成员外，关键字 protected 也可以用于基类被继承时候的访问限定符。当基类以保护的方式被继承的时候，其所有的公有和保护成员在派生类中都变成了保护成员。例如：

```
class D : protected B
{
  ///.....
};
```

那么，B 中所有的非私有成员在 D 中都变成了保护的。

练习：

1. 当基类以 private 的方式被继承的时候，它的公有成员在派生类中都成了私有的成员。对吗？
2. 基类中的私有成员可否通过继承的方式在派生类中变成共有的？
3. 在继承关系中，使用哪个访问限定符可以使得基类的成员可以被访问，而不是私有的不能访问。

专家答疑

问：能对 public，protected 和 private 进行一个总结吗？

答：当类的成员被声明为共有的，它是可以被程序中任何的代码访问的。当类的成员被声明私有的，它只能被该类本身的成员访问。派生类不能访问基类中的私有成员。当一个成员被

声明为保护的时候，它只能被自身类及其子类访问。所以，保护成员可以被继承但是在继承关系中则为私有的。

当基类以公有的方式被继承的时候，它的公有成员在派生类中依然是公有的，它的保护成员在派生类依然是保护的。当基类被以保护的方式继承的时候，它的公有和保护成员则在派生类中都变成了保护的。当基类被以私有的方式继承的时候，它的公有和保护成员在派生类中都是私有的。无论什么情况，基类中私有的成员都永远是该基类私有的。

构造函数与继承

在继承关系中，基类和派生类有可能都有自己的构造函数，这就存在一个很重要的问题：到底调用哪个构造函数来生成派生类的对象了，基类的构造函数，还是派生类的构造函数，或者两者都调用？这个问题的答案是这样的：基类的构造函数负责构建该对象的基类部分，派生类的构造函数负责构建该对象的派生类部分。这样做是合理的，这是因为基类是不能感知并访问派生类的成员的。因此它们各自部分的构造必须是分开的。前面的示例程序中采用了 C++提供的缺省的构造函数，因此没有什么问题。但是，实际应用中大部分的类都会定义构造函数。这里我们就针对这种情况进行讨论。

当只有派生类中定义了构造函数的时候，处理起来很简单：直接生成派生类的对象。其中基类的部分自动调用缺省的构造函数。例如，下面是 Triangle 类的另外一个版本，这个版本中，我们为 Triangle 类定义了构造函数。同时，还把 style 成员声明为私有的，它在构造函数中被赋值。

view plain

```
// 为 Triangle 类定义构造函数

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

//二维对象类
class TwoDShape
{
    double width;
    double height;
public:
    void showDim()
    {
        cout << "Width and height are "
            << width  << " and " << height << "\n";
    }

    //访问函数
    double getWidth() { return width; };
```

```
21.     double getHeight() { return height; };
22.     void setWidth(double w ) { width = w ; };
23.     void setHeight(double h ) { height = h; };
24. };
25.
26. //Triangle 类是从 TwoDShape 类中继承而来的
27. class Triangle : public TwoDShape
28. {
29.     char style[20];
30.
31. public:
32.
33.     // 构造函数
34.     Triangle( char *str, double w, double h)
35.     {
36.         //初始化 TwoDShape 的部分
37.         setWidth(w);
38.         setHeight(h);
39.
40.         //对 Triangle 类特有的部分进行初始化
41.         strcpy(style,str);
42.     }
43.
44.     double area()
45.     {
46.         return getWidth() * getHeight() / 2;
47.     }
48.
49.     void showStyle()
50.     {
51.         cout << "Triangle is " << style << "\n";
52.     }
53. };
```

其中，Triangle 类的构造函数对从 TwoDShape 类中继承来的以及自己增加的成员都进行了初始化。

当基类和派生类中都定义了构造函数的时候，情况就有点复杂，因为此时基类和派生类的构造函数都是要被执行的。

必备技能 10.4：调用基类的构造函数

当基类中定义了构造函数的时候，派生类必须显式地调用该构造函数来对对象的基类部分进行初始化。派生类通过使用扩展的派生类构造函数的声明方法来调用基类的构造函数。这种扩展的形式如下：

派生类的构造函数(参数列表): 基类的构造函数(参数列表);

{

//派生类构造函数体

}

其中，基类的构造函数就是派生类继承的基类的名称。注意，在上面的形式中使用了一个冒号来把基类和派生类的构造函数分开。如果一个派生类继承了多个基类，那么这些基类的构造函数都是通过冒号来分隔的。

下面的程序演示了如何把参数传入到基类的构造函数中。其中为 TwoDShape 类定义了一个构造函数用来完成对 width 和 height 属性的初始化。

view plain

```
1. // 为 TwoDShape 类增加构造函数
2. #include <iostream>
3. #include <cstring>
4. using namespace std;
5. //二维对象类
6. class TwoDShape
7. {
8.     double width;
9.     double height;
10.public:
11.    //构造函数
12.    TwoDShape(double w,double h)
13.    {
14.         width = w;
15.         height = h;
16.    }
17.    void showDim()
18.    {
19.        cout << "Width and height are "
20.             << width  << " and " << height << "\n";
21.    }
22.    //访问函数
23.    double getWidth() { return width; };
24.    double getHeight() { return height; };
25.    void setWidth(double w ) { width = w ; };
26.    void setHeight(double h ) { height = h; };
27.};
```

```
28. //Triangle 类是从 TwoDShape 类中继承而来的
29. class Triangle : public TwoDShape
30. {
31.     char style[20];
32. public:
33.     // 构造函数
34.     Triangle( char *str, double w, double h):TwoDShape(w,h)
35.     {
36.         strcpy(style,str);
37.     }
38.     double area()
39.     {
40.         return getWidth() * getHeight() / 2;
41.     }
42.     void showStyle()
43.     {
44.         cout << "Triangle is " << style << "\n";
45.     }
46. };
47. int main()
48. {
49.     Triangle t1("isosceles",4.0,4.0);
50.     Triangle t2("right", 8.0, 12.0);
51.     cout << "Info for t1:\n";
52.     t1.showStyle();
53.     t1.showDim();
54.     cout << "Area is  " << t1.area() << "\n";
55.     cout << "\n";
56.     cout << "Info for t2:\n";
57.     t2.showStyle();
58.     t2.showDim();
59.     cout << "Area is  " << t2.area() << "\n";
60.     return 0;
61. }
```

其中，Triangle()通过调用了 TwoDShape 的构造函数并把 w 和 h 的值传入其中来完成对 width 和 height 的初始化，而不用自己再对其进行初始化了。它只需要对自己特有的值进行初始化即可，那就是 style。这就使得 TwoDShape 类可以自由地按照自己的方式来构建自己的子对象。进一步来讲，TwoDShape 类可以自由增加不为派生类所感知的功能，避免了既存代码的修改。

基类中定义的任何形式的构造函数都是可以被派生类的构造函数调用的。真正被调用的是那个和传入参数相匹配的构造函数。例如，下面的代码中队 TwoDShape 和 Triangle 类机进行了扩展，增加了构造函数：

view plain

```
// 为 TwoDShape 类增加构造函数
#include <iostream>
#include <cstring>
using namespace std;
//二维对象类
class TwoDShape
{
    double width;
    double height;
public:
    //缺省的构造函数
    TwoDShape()
    {
        width = height = 0;
    }
    //构造函数
    TwoDShape(double w,double h)
    {
         width = w;
         height = h;
    }
    //构造高度和宽度相等的对象
    TwoDShape(double x)
    {
        width = height = x;
    }
    void showDim()
    {
        cout << "Width and height are "
            << width  << " and " << height << "\n";
    }
    //访问函数
    double getWidth() { return width; };
    double getHeight() { return height; };
    void setWidth(double w ) { width = w ; };
    void setHeight(double h ) { height = h; };
};
//Triangle 类是从 TwoDShape 类中继承而来的
class Triangle : public TwoDShape
```

```
{
    char style[20];
public:
    //缺省的构造函数。这将自动调用 TwoDShape 的缺省构造函数
    Triangle()
    {
        strcpy(style,"unknown");
    }
    //有三个参数的构造函数
    Triangle( char *str, double w, double h):TwoDShape(w,h)
    {
        strcpy(style,str);
    }
    //构建一个等腰三角形
    Triangle(double x):TwoDShape(x)
    {
        strcpy(style,"isoscoles");
    }
    double area()
    {
        return getWidth() * getHeight() / 2;
    }
    void showStyle()
    {
        cout << "Triangle is " << style << "\n";
    }
};
int main()
{
    Triangle t1;
    Triangle t2("right", 8.0, 12.0);
    Triangle t3(4.0);
    t1 = t2;
    cout << "Info for t1:\n";
    t1.showStyle();
    t1.showDim();
    cout << "Area is  " << t1.area() << "\n";
    cout << "\n";
    cout << "Info for t2:\n";
    t2.showStyle();
    t2.showDim();
    cout << "Area is  " << t2.area() << "\n";
    cout << "\n";
    cout << "Info for t2:\n";
```

```
84.     t3.showStyle();
85.     t3.showDim();
86.     cout << "Area is  " << t3.area() << "\n";
87.     cout << "\n";
88.     return 0;
89. }
```

上面程序的输出结果如下：

```
Info for t1:
Triangle is right
Width and height are 8 and 12
Area is  48
Info for t2:
Triangle is right
Width and height are 8 and 12
Area is  48
Info for t2:
Triangle is isoscoles
Width and height are 4 and 4
Area is  8
```

练习:

1. 派生类如何执行基类的构造函数？
2. 是否可以为基类的构造函数传入参数？
3. 哪个构造函数负责完成派生类对象中基类部分的初始化，是基类定义的构造函数还是派生类中定义的构造函数？

工程 10-1 扩展 Vehicle 类

在该工程中，我们针对第八章中的 Vehicle 类创建一个子类。

回忆一下，Vechile 类封装了汽车的信息，包括载客人数，邮箱容量以及每加仑油能跑的里程数。我们可以以 Vechile 类作为基础来创建更多更具体的类。例如，卡车就是一种 Vechile。卡车的一个重要属性就是它的载重量。因此我们可以采用集成 Vehicle 类的方式来创建 Truck 类，只需要增加一个实例变量来表示它的载重量。在该工程中，我们就将创建这样的 Truck 类。其中，我们会把 Vehicle 的实例变量声明为 private，而增加访问函数来获取它们的值。

步骤：

1. 创建一个文件，名称 TruckDemo.cpp，把第八章中最后一个版本的 Vehicle 的实现拷贝过来。

2. 创建 Trucnk 类，如下：

view plain

```
//在 Vechile 类的基础上增加特有的属性
class Truck : public Vehicle
{
public:
    int cargocap; //载重量，单位：磅
    //构造函数
    Truck( int p , int f, int m, int c ) : Vehicle(p,f,m)
    {
        cargocap = c;
    }
    //载重量的访问函数
    int get_cargocap()
    {
        return cargocap;
    }
};
```

这里，Truck 类继承了 Vehicle 类，并增加了 cargaocap 成员。因此，Truck 类中包含了通用类 Vechile 中的全部元素。增加的只是 Trunck 特有的元素。

3. 完整的程序如下：

view plain

```
#include <iostream>
using namespace std;
//声明 Vehicle 类
class Vehicle
{
    int passengers; // 载客人数
    int fuelcap;    // 油箱容量
    int mpg;        // 每加仑油能跑的里程数
public:
    //构造函数
    Vehicle(int p, int f, int m)
    {
        passengers = p;
        fuelcap = f;
        mpg = m;
    }
```

```
    int range()     //计算并返回汽车能跑的最大里程数，此处为函数声明
    {
        return mpg * fuelcap;
    }
    //访问函数
    int get_passangers() { return passengers; };
    int get_fuelcap() { return fuelcap; };
    int get_mpg() { return mpg; };
};
//在 Vechile 类的基础上增加特有的属性
class Truck : public Vehicle
{
public:
    int cargocap; //载重量，单位：磅
    //构造函数
    Truck( int p , int f, int m, int c ) : Vehicle(p,f,m)
    {
        cargocap = c;
    }
    //载重量的访问函数
    int get_cargocap()
    {
        return cargocap;
    }
};
int main()
{
    //生成 Trunck 类的对象
    Truck semi(2,200,7,4400);
    Truck pickup(3,28,15,2000);
    int dist = 252;
    cout <<"Semi can carry " << semi.get_cargocap() << " pounds.\n";
    cout << "It has a range of " << semi.range() << " miles.\n";
   cout << "To go " << dist << " miles semi needs "
    << dist/semi.get_mpg() << " gallons of fuel.\n\n";
    cout <<"Pickup can carry " << pickup.get_cargocap() << " pounds.\n
  ";
    cout << "It has a range of " << pickup.range() << " miles.\n";
    cout << "To go " << dist << " miles pickup needs "
    << dist/pickup.get_mpg() << " gallons of fuel.\n";
    return 0;
}
```

4. 上面程序的输出如下：

```
Semi can carry 4400 pounds.

It has a range of 1400 miles.

To go 252 miles semi needs 36 gallons of fuel.

Pickup can carry 2000 pounds.

It has a range of 420 miles.

To go 252 miles pickup needs 16 gallons of fuel.
```

5. 我们还可以从 Vehicle 类派生出许多别的类。比如，下面的代码框架就创建了一个越野车的类，它存储了车辆距离地面的净距离：

```
class off-road : public Vechile

{

    int groundClearance; // 距离地面净距离，单位英寸

public:

    // ...

};
```

重要的一点是，一点我们创建了具有对象通用属性的基类，我们就可以继承该类从而创建特殊的类。每个派生类只是简单地增加自己特有的属性即可，这就是继承的本质。

必备技能 10.5：多层继承

到目前为止，我们使用到的继承关系只是一个基类和一个派生类。然而，我们是可以创建任意多层的继承关系的。正如在前面所提到的那样，把一个派生类作为另外一个类的基类是完全可以接受的。例如，有三个类 A，B 和 C。C 可是是从 B 继承而来的，同时 B 可以是从 A 继承而来的。当发生这样的情况的时候，每个派生类都继承了它所有基类的所有属性。针对刚才的例子，C 就继承了 A 和 B 的全部属性。

为了理解多层继承的作用，我们来研究下面的程序。其中，Triangle 类被作为基类来创建派生类 ColoreTriangle。ColorTriangle 就拥有了 Triangle 类和 TwoDShape 类的全部属性。

view plain

```
1. //多层继承
2. #include <iostream>
3. #include <cstring>
4. using namespace std;
5.
6. //二维对象类
7. class TwoDShape
8. {
9.     double width;
10.    double height;
11.public:
12.    //缺省的构造函数
13.    TwoDShape()
14.    {
15.        width = height = 0;
16.    }
17.
18.    //构造函数
19.    TwoDShape(double w,double h)
20.    {
21.         width = w;
22.         height = h;
23.    }
24.
25.    //构造高度和宽度相等的对象
26.    TwoDShape(double x)
27.    {
28.        width = height = x;
29.    }
30.
31.    void showDim()
32.    {
33.        cout << "Width and height are "
34.             << width  << " and " << height << "\n";
35.    }
36.
37.    //访问函数
38.    double getWidth() { return width; };
39.    double getHeight() { return height; };
40.    void setWidth(double w ) { width = w ; };
```

```
41.     void setHeight(double h ) { height = h; };
42. };
43.
44. //Triangle 类是从 TwoDShape 类中继承而来的
45. class Triangle : public TwoDShape
46. {
47.     char style[20];
48.
49. public:
50.
51.     //缺省的构造函数。这将自动调用 TwoDShape 的缺省构造函数
52.     Triangle()
53.     {
54.         strcpy(style,"unknown");
55.     }
56.
57.     //有三个参数的构造函数
58.     Triangle( char *str, double w, double h):TwoDShape(w,h)
59.     {
60.         strcpy(style,str);
61.     }
62.
63.     //构建一个等腰三角形
64.     Triangle(double x):TwoDShape(x)
65.     {
66.         strcpy(style,"isoscoles");
67.     }
68.
69.     double area()
70.     {
71.         return getWidth() * getHeight() / 2;
72.     }
73.
74.     void showStyle()
75.     {
76.         cout << "Triangle is " << style << "\n";
77.     }
78. };
79.
80. //对 Triangle 进行扩展
81. class ColorTriangle : public Triangle
82. {
83.     char color[20];
84. public:
```

```
85.     ColorTriangle(char * clr, char *style, double w, double h):Triangl
    e(style, w, h)
86.     {
87.         strcpy(color,clr);
88.     }
89.
90.     //显示颜色
91.     void showColor()
92.     {
93.         cout << "Color is  " << color << "\n";
94.     }
95. };
96.
97. int main()
98. {
99.     ColorTriangle t1("Blue", "Right", 8.0, 12.0);
100.     ColorTriangle t2("Red", "isosceles", 2.0, 2.0);
101.
102.     cout<<"Info for t1: \n";
103.     t1.showStyle();
104.     t1.showDim();
105.     t1.showColor();
106.     cout << "Area is  " << t1.area() << "\n";
107.
108.     cout << "\n";
109.
110.     cout<<"Info for t2: \n";
111.     t2.showStyle();
112.     t2.showDim();
113.     t2.showColor();
114.     cout << "Area is  " << t2.area() << "\n";
115.
116.     return 0 ;
117. }
```

上面程序的输出如下：

```
Info for t1:
Triangle is Right
Width and height are 8 and 12
Color is  Blue
Area is  48
```

```
Info for t2:
Triangle is isosceles
Width and height are 2 and 2
Color is  Red
Area is  2
```

作为继承关系中的派生类，ColorTriangle 类拥有了其基类 Triangle 和 TwoDShape 类的属性，并增加了自己需要的信息，也是特有的信息。这也是继承的一大好处，使得代码可以重用。

这个例子中我们看看到另外一点。那就是在继承关系中，如果基类的构造函数需要参数，那么所有派生类的构造函数就必须为其传入这些参数，而不管派生类是否需要自己的参数。

必备技能 10.6：继承多个基类

在 C++中，一个派生类是可以继承两个或者多个基类的。如下的程序中，D 就继承了 B1 和 B2 两个类。

view plain

```
1. //演示继承多个基类
2. #include <iostream>
3. using namespace std;
4.
5. class B1
6. {
7. protected:
8.     int x;
9. public:
10.     void showx()
11.     {
12.         cout << x << "\n";
13.     }
14. };
15.
16. class B2
17. {
18. protected:
19.     int y;
20. public:
21.     void showy()
22.     {
23.         cout << y << "\n";
24.     }
25. };
```

```
26.
27. //类 D 继承了多个基类
28. class D : public B1, public B2
29. {
30. public:
31.     //D 是可以访问 x 和 y 的。因为它们在基类中是保护成员。
32.     void set(int i, int j)
33.     {
34.         x = i;
35.         y = j;
36.     }
37. };
38.
39. int main()
40. {
41.     D ob;
42.     ob.set(10,20); //调用 D 提供的函数
43.     ob.showx(); //显示从 B1 继承来的成员值
44.     ob.showy();//显示从 B1 继承来的成员值
45.
46.     return 0;
47. }
```

正如在这里例子中看到的那样，我们使用逗号来把继承的多个基类分割。另外，还要确保每个基类前面都有访问限定符。

必备技能 10.7：构造函数和析构函数在什么时候被执行？

基类可能含构造函数或者析构函数，派生类也可能含有构造函数或者析构函数，或者两者都有构造函数或者析构函数，此时理解这些构造函数和析构函数的执行顺序就是非常重要的了。具体来说，当一个派生类的对象被生成的时候，基类和派生类的构造函数是以什么样的顺序被调用的呢？当这个对象被销毁的时候，析构函数又是以什么样的顺序被调用的了。为了回答这些问题，我们来研究一下下面的这个程序：

view plain

```
1. #include <iostream>
2. using namespace std;
3. class B
4. {
5. public:
6.     B()
7.     {
```

```
8.          cout << "Constructing base portion.\n";
9.      }
10.     ~B()
11.     {
12.         cout << "Desctructing base portion.\n";
13.     }
14. };
15. class D : public B
16. {
17. public:
18.     D()
19.     {
20.         cout << "Constructing derived portion.\n";
21.     }
22.     ~D()
23.     {
24.         cout << "Destructing derived portion.\n";
25.     }
26. };
27. int main()
28. {
29.     D ob;
30.     //do nothing
31.     return 0;
32. }
```

正如上面程序中的注释那样，这个程序只是简单的生成了一个名称为 ob 的对象，它是 D 类的一个对象，然后就销毁了该对象。执行该程序时，输出如下：

Constructing base portion.

Constructing derived portion.

Destructing derived portion.

Desctructing base portion.

如程序输出的那样，首先是 B 类的构造函数被执行，接着执行 D 类的构造函数。再接着，D 的析构函数被调用，最后是 B 的析构函数被调用。

从上面程序的经验来看，我们可以总结如下：当派生类的对象被创建的时候，首先调用的是基类的够战术，然后调用派生类的构造函数。当派生类的对象被销毁的时候，派生类的析构函数先被调用，然后基类的析构函数才被调用。换句话说，构造函数的调用顺序和继承关系保持一致，而析构函数的调用则恰好相反。在多层次的继承关系中，同样的规则也适用：构造函数的调用顺序和继承关系保持一致，而析构函数则恰好相反。当一个类继承了多余一

个基类的时候，构造函数的调用顺序则是按照基类继承列表中从左到右的顺序进行的，而析构函数的调用顺序则是相反的，从右到左进行的。

练习：

1. 一个派生类可否作为另外一个派生类的基类？
2. 在类的继承关系中，构造函数的调用顺序如何？
3. 在类的继承关系中，析构函数的调用顺序如何？

专家答疑

问：为什么构造函数的调用顺序是以继承的关系为准，而析构函数则是以相反的顺序被调用？

答：如果我们仔细考虑一下，构造函数是按照继承的顺序来来调用的，这样是有道理的。因为基类是感知不到派生类的。基类所要进行的初始化操作是和派生类要进行的操作完全分开的，而且还有可能是派生类进行初始化的前提。因此，基类的构造函数必须首先被调用。同样的道理，析构函数是以与继承关系相反的顺序被调用也是有道理的。既然基类是派生类的基础，基类的销毁就意味着派生类的销毁。因此，派生类的析构函数必须在派生类的对象被完全销毁前就被调用。

必备技能 10.8：指向派生类对象的指针

在继续讨论虚函数和多态性之前，我们有必要讨论一下关于指针的一个重要方面。指向派生类（对象）的指针和指向基类（对象）的指针不像其它类型的指针那样没有什么关系，而是有关系的。通常来讲，一个类型的指针是不能指向另外一种类型的对象的。然而，基类指针和派生类对象则是个例外。在 C++中，一个基类的指针是可以指向任何继承了该类的派生类的对象的。例如，有类 B 和 D，其中 D 是从 B 继承而来的。那么任何 B 类型的指针都是可以用来指向 D 类型的对象的。因此，下面的代码段是有效的：

```
B *p; //B 类型的指针
B B_ob;  //B 类的对象
D D_ob; // D 类的对象
p = &B_ob; //p 指向 B 类的对象
p = &D_ob;  //p 指向 D 类的对象，而 D 类继承了 B 类
```

基类的指针如果指向的是派生类的对象，那么通过该指针则只能访问该对性从基类中继承的那部分。因此，针对上面的例子，我们可以通过 p 指针来访问 D_ob 对象从 B 类中继承的那部分，而不能通过 p 访问到 D 类对象特有的那部分。

还有一点需要注意：尽管基类的指针是可以指向派生类的对象的；但是反之则不行。也就是说，不能通过派生类的指针来访问基类的对象。

我们都知道，指针的加法和减法都是和它的基本类型相关的。因此，当一个基类的指针指向一个派生类的对象的时候，该指针自加或者自减后就不是指向下一个或者前一个派生类

的对象了，而是指向的是下一个或者前一个基类的对象。所以，我们可以认为，当基类的指针指向派生类对象的时候，对其进行增加或者减小都是错误的做法。

基类指针可以指向任意的它的派生类的对象，这一点非常重要，也是C++中的一个基础点。正如我们将要学校到的，这种灵活性是C++实现运行时多态性的关键。

对派生类型的引用

和前面描述的指针类似，基类的引用可以被用来对一个派生类的对象进行引用。这点经常被用在函数的参数中。基类引用参数可以接收基类对象作为实参，也可以接收任何基类的派生类的对象作为实参。

必备技能 10.9：虚函数和多态性

C++中对多态性的支持都是基于继承和基类指针的，而实际上实现了多态性的是虚函数。本章的剩余部分将讨论这个重要的特性。

虚函数的基础

虚函数是在基类中被声明为 virtual，而在一个或者多个的派生类中被重新定义的函数。因此，每个派生类都可以拥有自己的与众不同的实现版本。

真正使得虚函数变得有意思的事情发生在当我们使用基类的指针来调用虚函数的时候。当我们通过基类的指针来调用虚函数的时候，C++根据该指针指向对象的实际类型来决定应该调用该函数的那个实现版本。这种决定是在运行时发生的。因此，当指针指向不同对象的时候，就会执行虚函数的不同实现版本。换句话说，正是指针指向对象的类项，而不是指针本身的类型决定了应该被执行的虚函数的实现版本。因此，如果一个基类中含有一个虚函数，并且有两个或者更多的派生类继承了该基类，当通过该基类指针指向不同派生类对象的时候，执行的就会是该函数的不同版本。当我们使用基类的引用来调用该函数的时候，同样的情况也会发生。

我们通过在基类中声明函数的时候使用 virtual 关键字来实现虚函数的声明。当在派生类中重新定义该虚函数的时候，关键字 virtual 就没有必要在重复使用了，尽管这样做也不会引起错误。

含有虚函数的类被称为是多态性的类。这个术语同样也适用于继承了含有虚函数的类的派生类。

下面的程序就对虚函数进行了演示：

view plain

```
1. //一个用于演示虚函数的简短的程序
2. #include <iostream>
3. using namespace std;
4. class B
5. {
6. public:
```

```
7.      virtual void who() //声明为虚函数
8.      {
9.          cout << "Base\n";
10.     }
11. };
12. class D1 : public B
13. {
14. public:
15.     void who()//重新定义基类中的虚函数
16.     {
17.         cout << "First derivation\n";
18.     }
19. };
20. class D2 : public B
21. {
22. public:
23.     void who()//重新定义基类中的虚函数
24.     {
25.         cout << "Second derivation\n";
26.     }
27. };
28. int main()
29. {
30.     B base_obj;
31.     B *p;
32.     D1 D1_obj;
33.     D2 D2_obj;
34.     p = &base_obj;
35.     p->who();
36.     p = &D1_obj;
37.     p->who();
38.     p = &D2_obj;
39.     p->who();
40.     return 0;
41.
42. }
```

上面程序的输出如下：

Base

First derivation

Second derivation

下面我们就仔细研究一下上面的程序，看看它是如何工作的。

正如我们看到的那样，在类 B 中，函数 who()被声明为虚函数。这就意味着该函数可以在派生类中被重新定义。在两个派生类 D1 和 D2 中也各自实现了对 who()函数的重新定义。在 main()函数中，声明了 4 个变量。其中 base_obj 是 B 类的对象；p 是一个 B 类型的指针；D1_obj 和 D2_obj 分别是派生类 D1 和 D2 的对象。接着给 p 赋值为 base_obj 对象的地址，然后调用 who()函数。由于 who()函数是虚函数，C++在运行时才能根据 p 指向对象的类型来判断到底应该调用那个版本的 who()。在这里，p 指向的是一个 B 类型的对象，因此调用的也就是 B 类中的 who()函数了。接着，给 p 赋值为 D1_obj 的地址，这样做是可以，回忆一下我们在前面说过基类指针是可以指向任何派生类的对象的。然后在调用 who()函数，C++再次检查 p 指针指向对象的类项，进而判断需要调用那个版本的 who()函数。既然现在 p 指向对象的类型为 D1，那么调用的也自然就是 D1 类中的 who()函数了。同样的道理，当给 p 赋值为 D2_obj 的地址的时候，后面调用的就是 D2 类中的 who()函数了。

总结：当我们通过基类的指针来调用虚函数的时候，真正被执行的函数版本取决于运行时指针指向对象的类型。

尽管通常情况下虚函数都是通过基类指针来被调用的，但是它也可以像普通的函数那样被调用，即使用标准的点号（.）运算符。这也就是说，在上面的示例程序中，下面的语句是完全正确的：

D1_obj.who();

然而，以这样的方式调用虚函数就忽略了它的多态属性。只有当通过基类的指针或者引用来访问虚函数的时候才能够获得它的多态性。

咋看起来，在派生类中对虚函数的重新定义貌似是一种特殊形式的函数重载。其实不然。实际上，这完全是两种不同的过程。首先，函数重载要求必须在函数参数类型或者数量上不同，而对虚函数的重新定义则要求参数的类型和数量必须完全一样。如果函数的原型不一样，那么就会被认为是函数的重载，而不是对虚函数的重新定义，也就没有了虚函数的性质。其次，虚函数必须是成员函数，而不能是类的友元函数。但是虚函数是可以成为别的类的友元函数的。最后，类的析构函数可以是虚函数，而构造函数则不能。

由于上述的限制和普通函数重载与重新定义虚函数的差别，人们使用术语重写来描述这种对虚函数的重新定义。

虚函数是会被继承的

一旦一个函数被声明是虚函数，那么它将始终都是虚函数而不管它所在的基类被继承了多少层。例如，如果 D2 类继承了 D1 类而不是 B 类，如我们下面的程序所示，那么 who()函数依然是虚函数：

view plain

```
1. //D2 类是继承 D1 类的，而不是 B 类
2. class D2 : public D1
3. {
```

```
4. public:
5.     void who()//重新定义基类中的虚函数
6.     {
7.         cout << "Second derivation\n";
8.     }
9. };
```

当派生类中没有对基类中的虚函数进行重写的时候，那么将会使用基类中的虚函数。例如，在下面的程序中，D2 类就没有对 who()函数进行重写：

view plain

```
1. //一个用于演示虚函数的简短的程序
2. #include <iostream>
3. using namespace std;
4. class B
5. {
6. public:
7.     virtual void who() //声明为虚函数
8.     {
9.         cout << "Base\n";
10.    }
11.};
12.class D1 : public B
13.{
14.public:
15.    void who()//重新定义基类中的虚函数
16.    {
17.        cout << "First derivation\n";
18.    }
19.};
20.class D2 : public B
21.{
22.public:
23.   // 没有重写 who()函数
24.};
25.int main()
26.{
27.    B base_obj;
28.    B *p;
29.    D1 D1_obj;
30.    D2 D2_obj;
31.    p = &base_obj;
```

```
32.     p->who();
33.     p = &D1_obj;
34.     p->who();
35.     p = &D2_obj;
36.     p->who();
37.     return 0;
38.
39. }
```

上面程序的输出如下：

```
Base
First derivation
Base
```

由于 D2 类中没有对 who()函数进行重写，使用的就是 B 类中定义的 who()的版本。

请牢记虚函数的特性是继承的。因此，如果在上面的程序中， D2 类继承的是 D1 类而不是 B 类，那么当使用 D2 类的对象调用 who()函数的时候，实际调用的就是 D1 类中的 who()函数，而不是 B 类中的 who()函数。这是因为 D1 类是距离 D2 类最近的类。

为什么要引入虚函数？

正如我们在前面陈述过的那样，虚函数结合继承就支持了 C++中的运行时多态性。多态性是面向对象编程中的核心，因为它允许通用来类定义它所有的派生类所共有的函数，也允许了派生类根据自己的需要来明确定义部分或者全部虚函数的实现。有时候也会这样来描述这种思想：基类中规定了派生类对象需要提供的接口，但是由派生类来具体实现这些接口。这也是我们为什么总是会用“一个接口，多种实现”来描述多态性的原因了。

成功应用多态性的重要一点就是需要理解基类和派生类形成了一种层次关系。这种关系从基类到派生类代表着通用性的逐渐降低。如果设计合理，基类可以提供派生类可以直接使用的全部元素，同样也定义了派生类必须自己实现的函数。这就使得派生类能在保持统一接口的基础上，还能增加自己特有的函数，从而增加了灵活性。也即是说，既然接口是由基类来定义的，那么任何由它派生出来的派生类都将享有相同的接口。这样以来，虚函数的使用就使得用基类来定义所有派生类的通用接口成为了可能。

此时，我们可能会问，为什么保持接口统一，而实现不同这么重要呢？答案又回到了驱动面向编程发展的原因上：这样可以帮助程序员处理日益增加大的复杂的程序。例如，如果我们的程序设计的正确，那么我们就知道从基类派生出来的所有派生类的对象都是使用相同的接口来访问的，尽管不同派生类对接口的实现可能不同。这就意味着，我们只需要处理同一个接口即可，而不是需要处理多个不同的接口。另外，派生类还可以自由地使用基类的部分或者全部的功能，而不需要我们再次编写这些功能。

这种接口与实现的分离也使得我们可以创建第三方提供的类库。如果这些类库实现正确，我们就可以使用类库中提供的公共接口来定义满足自己特殊需要的派生类。例如，微软的基础类以及新的.NET 框架窗体类都是支持 windows 编程的。通过使用这些类，我们的程序就可以继承 window 编程的大部分功能。我们只需要增加自己程序特有的部分即可。这点在编写大型复杂程序的时候会带来很大的好处。

使用虚函数

为了能够更好的理解虚函数的强大，我们会在 TwoDShape 类中使用它。在前面的示例中，每一个从 TwoDShape 类派生出来的类中都定义了 area()函数。这就暗示着我们最好把 area()函数在 TwoDShape 类中定义为虚函数，允许每个派生类重写它即可。派生类根据自己封装的二维形状来定义面积应该如何计算。下面的程序就是这样做的，其中还为 TwoDShape 类增加了一个字段叫做 name。(这样做是为了方便程序演示。)

view plain

```
1. //使用虚函数和多态性
2. #include <iostream>
3. #include <cstring>
4. using namespace std;
5. //二维形状类
6. class TwoDShape
7. {
8.     //私有成员
9.     double width;
10.    double height;
11.    //增加一个 name 字段
12.    char name[20];
13.public:
14.    //缺省的构造函数
15.    TwoDShape()
16.    {
17.        width = height = 0.0;
18.        strcpy(name,"unknown");
19.    }
20.    //构造函数
21.    TwoDShape(double w, double h, char *n)
22.    {
23.        width = w;
24.        height = h;
25.        strcpy(name,n);
26.    }
27.    //创建宽度和高度相等的对象
28.    TwoDShape(double x, char *n)
29.    {
```

```
        width = height = x;
        strcpy(name,n);
    }
    void showDim()
    {
        cout << "Widht and height are: "
             << width << " and  " << height;
    }
    //访问函数
    double getWidth() { return width; };
    double getHeight() { return height; };
    void setWidth(double w ) { width = w; };
    void setHeight(double h ) { height = h; };
    char *getName() { return name; };
    //虚函数 area()
    virtual double area()
    {
        cout << "Error: area() must be overridden.\n";
        return 0.0;
    }
};
//Triangle 类继承了 TwoDShape 类
class Triangle : public TwoDShape
{
    char style[20]; // 私有成员
public:
    //缺省的构造函数。这将自动调用 TwoDShape 类的缺省构造函数
    Triangle()
    {
        strcpy(style, "unknown");
    }
    //带有三个参数的构造函数
    Triangle(char *str, double w, double h):TwoDShape(w,h,"triangle")

    {
        strcpy(style,str);
    }
    //生成等腰三角形
    Triangle(double x):TwoDShape(x,"triangle")
    {
        strcpy(style,"isosceles");
    }
    //重写 area()函数
    double area()
```

```
73.     {
74.         return getWidth() * getHeight() / 2;
75.     }
76.     void showStyle()
77.     {
78.         cout << "Triangle is " << style << "\n";
79.     }
80. };
81. //Rectangle 类继承 TwoDShape 类
82. class Rectangle : public  TwoDShape
83. {
84. public:
85.     //生成矩形
86.     Rectangle(double w, double h) : TwoDShape(w,h,"rectangle")
87.     {
88.     };
89.     //生成正方形
90.     Rectangle(double x) : TwoDShape(x,"rectangle")
91.     {
92.     };
93.     bool isSquare()
94.     {
95.         if ( getWidth() == getHeight() )
96.         {
97.             return true;
98.         }
99.         else
100.         {
101.             return false;
102.         }
103.     };
104.     //重写 area()函数
105.     double area()
106.     {
107.         return getWidth() * getHeight();
108.     }
109. };
110. int main()
111. {
112.     //声明一个指向 TowDShape 对象的指针数组
113.     TwoDShape * shapes[5];
114.     shapes[0] = &Triangle("right",8.0, 12.0);
115.     shapes[1] = &Rectangle(10);
116.     shapes[2] = &Rectangle(10, 4);
```

```
117.     shapes[3] = &Triangle(7.0);
118.     shapes[4] = &TwoDShape(10,20,"generic");
119.     for ( int i = 0; i < 5; i++)
120.     {
121.         cout << "Object is " << shapes[i]->getName() << "\n";
122.         cout << "Area is " << shapes[i]->area() << "\n";
123.         cout << "/n";
124.     }
125.     return 0;
126. }
```

上面程序的输出如下:

```
Object is triangle
Area is 48
Object is rectangle
Area is 100
Object is rectangle
Area is 40
Object is triangle
Area is 24.5
Object is generic
Error: area() must be overridden.
Area is 0
```

我们来仔细研究一下上面的这个程序。首先，在类 TowDShape 中，area()函数被声明为虚函数，由 Triangle 和 Rectangle 类重写。在 TwoDShape 类的 area()函数只是起到了占位的作用，并输出信息告诉用户该函数必须由派生类来进行重写。派生类都是根据自身封装的形状来重写 area()函数。如果我们还需要实现椭圆形这个类，也需要重写 area()来计算椭圆形的面积。

这个程序为我们演示了一个重要的特性。注意在 main()函数中 shapes 是被声明为 TwoDShape 类的指针数组。然而，给这个数组中元素赋值的是 Triangle，Rectangle 以及 TwoDShape 类的对象地址。这样写是正确的，这是因为基类指针是可以指向派生类的对象的。尽管这个示例程序相当简单，但是它为我们演示了继承和虚函数结合的强大功能。基类指针指向对象的类型是在运行是才判断的，并且执行相应的动作的。如果一个对象是由 TwoDShape 派生出来的，那么我们就可以调用 area()函数来计算其面积。计算面积这个操作始终是保持一样的，而不管我们用到的是什么形状。

练习：

1. 什么是虚函数？

2. 为什么虚函数如此重要？

当通过基类指针来调用虚函数的时候，实际执行的是该函数的那个版本？

必备技能 10.10：纯虚函数和抽象类

有时候我们想定义一个所有派生类都能共享的基类，它只为派生类提供通用的形式，而由派生类来完成实现的具体细节。这样的类决定了派生类必须实现其中的函数，而基类自身却不提供一个或者多个这样的函数的实现。当基类不能用来提供有意义的函数实现方法的时候，我们就需要这样做。前面示例程序中的 TwoDShape 类中的 area()函数就属于这种情况。在 TwoDShape 类中 area()函数的定义只是起到了占位的作用。它不能用来计算并显示出任何对象的面积。

由于我们自己是可以创建类库的，在基类中存在没有明确意义的函数是非常普遍的。我们可以有两种方式来对这种情况进行处理。第一，就是像我们在前面的示例程序中的那样，让其输出一个警告信息。这种处理方式在某些情况下是有用的，比如程序的调试，但这种方式不总是正确的。基类中可能含有派生类必须重写的函数，以便这样的函数在派生类中有明确的含义。就拿前面的 Triangle 类来说，如果我们不为其定义 area()函数，则继承的 area()函数对 Triangle 来说是没有意义的。在这种情况下，我们需要一种机制来确保派生类的确对某些必要的函数进行了重写。在 C++中，针对这个问题的解决方法就是纯虚函数。

纯虚函数就是在基类中声明的函数，但是该函数在基类中并没有定义。这样一来，所有该类的派生类都必须定义该函数的自己的版本，而不能直接使用基类提供的函数。声明纯虚函数的通用形式如下：

virtual 类型 函数名称(参数列表) = 0;

其中的类型就是函数返回值的类型，函数名称为该函数的名字。我们可使用纯虚函数来对 TwoDShape 类进行改进。由于对于一个没有定义的二维图形来说，计算其面积是没有什么意义的，下面的程序中就把 TwoDShape 类中的 area()函数定义为纯虚函数。这就意味着，它的所有派生类都必须重写 area()这个函数。

view plain

```
1.  //使用纯虚函数
2.  #include <iostream>
3.  #include <cstring>
4.  using namespace std;
5.  //二维形状类
6.  class TwoDShape
7.  {
8.      //私有成员
9.      double width;
10.     double height;
```

```
    //增加一个 name 字段
    char name[20];
public:
    //缺省的构造函数
    TwoDShape()
    {
        width = height = 0.0;
        strcpy(name,"unknown");
    }
    //构造函数
    TwoDShape(double w, double h, char *n)
    {
        width = w;
        height = h;
        strcpy(name,n);
    }
    //创建宽度和高度相等的对象
    TwoDShape(double x, char *n)
    {
        width = height = x;
        strcpy(name,n);
    }
    void showDim()
    {
        cout << "Widht and height are: "
             << width << " and  " << height;
    }
    //访问函数
    double getWidth() { return width; };
    double getHeight() { return height; };
    void setWidth(double w ) { width = w; };
    void setHeight(double h ) { height = h; };
    char *getName() { return name; };
    //纯虚函数 area()
    virtual double area() = 0;
};
//Triangle 类继承了 TwoDShape 类
class Triangle : public TwoDShape
{
    char style[20]; // 私有成员
public:
    //缺省的构造函数。这将自动调用 TwoDShape 类的缺省构造函数
    Triangle()
    {
```

```
55.         strcpy(style, "unknown");
56.     }
57.     //带有三个参数的构造函数
58.     Triangle(char *str, double w, double h):TwoDShape(w,h,"triangle")

59.     {
60.         strcpy(style,str);
61.     }
62.     //生成等腰三角形
63.     Triangle(double x):TwoDShape(x,"triangle")
64.     {
65.         strcpy(style,"isosceles");
66.     }
67.     //重写 area()函数
68.     double area()
69.     {
70.         return getWidth() * getHeight() / 2;
71.     }
72.     void showStyle()
73.     {
74.         cout << "Triangle is " << style << "\n";
75.     }
76. };
77. //Rectangle 类继承 TwoDShape 类
78. class Rectangle : public  TwoDShape
79. {
80. public:
81.     //生成矩形
82.     Rectangle(double w, double h) : TwoDShape(w,h,"rectangle")
83.     {
84.     };
85.     //生成正方形
86.     Rectangle(double x) : TwoDShape(x,"rectangle")
87.     {
88.     };
89.     bool isSquare()
90.     {
91.         if ( getWidth() == getHeight() )
92.         {
93.             return true;
94.         }
95.         else
96.         {
97.             return false;
```

```
98.         }
99.     };
100.     //重写 area()函数
101.     double area()
102.     {
103.         return getWidth() * getHeight();
104.     }
105. };
106. int main()
107. {
108.     //声明一个指向 TowDShape 对象的指针数组
109.     TwoDShape * shapes[4];
110.     shapes[0] = &Triangle("right",8.0, 12.0);
111.     shapes[1] = &Rectangle(10);
112.     shapes[2] = &Rectangle(10, 4);
113.     shapes[3] = &Triangle(7.0);
114.     for ( int i = 0; i < 4; i++)
115.     {
116.         cout << "Object is " << shapes[i]->getName() << "\n";
117.         cout << "Area is " << shapes[i]->area() << "\n";
118.         cout << "\n";
119.     }
120.     return 0;
121. }
```

如果一个类中至少含有一个纯虚函数，那么我们就说这个类是抽象类。抽象类的一个重要特性就是：不能创建这种类的对象。为了验证这一点，我们可以把上面程序中 Triangle 类中对 area()函数的重写删掉。此时当我们试图创建 Triangle 类的对象的时候就会出现错误。实际上，抽象类只能作为派生类的基类来使用。抽象类之所以不能实例化的原因就在于其中有一个或者多个函数没有定义。正是由于这个原因，在上面的程序中数组 shapes 的大小被缩减成了 4，并且不再创建通用的 TwoDShape 类的对象了。正如这个程序展示的那样，尽管此时基类是抽象类，但是我们还是可以使用它作为类型来声明指针的。这个指针可以被用来指向派生类的对象。

复习题

1. 被继承的类叫做_____类。继承别的类的类叫做_____类。
2. 基类是否可以访问派生类中的成员？派生类是否可以访问基类中的成员？
3. 创建 TwoDShape 类的一个派生类叫做 Circle。在 area()函数中计算圆形的面积。
4. 如何阻止派生类访问基类中的成员？
5. 写出派生类的构造函数调用基类构造函数的通用形式。
6. 有如下的继承关系：

```
class Alpha { .... };
class Beta : public Alpha { ..... };
class Gamma : public Beta { ..... };
```

当创建 Gamma 类的对象的时候上述类中的构造函数是按照什么样的顺序被调用的？

7. 保护成员的访问权限如何？
8. 基类的指针是可以指向派生类的对象的。解释一下为什么这点对于函数重写来讲是非常重要的。
9. 什么是纯虚函数？什么是抽象类？
10. 抽象类是否可以实例化？
11. 解释一下为什么纯虚函数有助于实现“一个接口，多种实现”的多态性。

第十一篇 C++ IO 系统

我们从本书的一开始就使用到了 C++中的输入输出系统。当时我们并没有对其进行正式的讨论。由于输入输出系统是基于类的层次关系，所以在没有对类和继承进行讨论之前是不可能对其理论进行讨论的。现在我们可以对 C++的输入输出系统进行仔细的讨论了。C++中的输入输出系统是非常庞大的，在这里我们不可能对全部的类、函数和相关特性进行讨论，但是在本章中我们将对最重要和最常用的部分进行讨论。本章特别对<<和>>运算符的重载进行了讨论，以便于我们能对自己编写的类对象进行输入和输出。本章还讨论了如何进行格式化的输出以及如何处理输入输出操作。本章最后对文件的输入和输出进行讨论。

旧版本与新版本的 C++输入输出系统

目前在 C++中提供了两个版本的面向对象的输入输出库可以使用：基于最初的 C++详细说明的老版本以及由标准 C++定义的新版本。老版本的库由头文件<iostream.h>支持。新的版本由文件<iostream>支持。对程序员来说，这两个库大部分是相同的。这是因为新的版本从本质上来说是对老版本的更新和改进。实际上，两个版本之间的主要差别在于库的实现上，而不是在他们的使用方法上。

从程序员的角度来看，老版本和新版本的 C++输入输出库有两个主要的区别。第一，新的输入输出库中包含了一些新增的特性和定义了一些新的数据类型。因此，从本质上来讲新版本实际上是老版本的超级。几乎所有的使用了老版本库的程序都可以在使用新版本库的时候被正确编译，而不需要实际的修改。第二，老版本的输入输出库是在全局命名空间中的，而新版本的库是位于 std 命名空间中的。由于老版本的输入输出库已经被废弃了，本书将只对新版本的输入输出库进行讨论，但是其中大部分也是适用于老版本的库的。

必备技能 11.1 C++流

理解 C++的输入输出库的最重要的一点就是对流的操作。流是一种抽象的概念，它或者用来生产信息，或者用来消费信息。在 C++的输入输出系统中，流是和物理设备相关联的。所有的流都是以相同的方式工作的，即使这些流相关的物理设备互不相同。由于所有的流都

是以相同的方式工作的，相同的输入输出操作和函数实际上可以工作在任何设备上。例如，我们使用的在屏幕上输出的方式是同样可以被用来写磁盘或者输出到打印机上的。

以最通用的形式来说，流是一种对文件的逻辑接口。C++中定义的“文件”可以是磁盘，屏幕，键盘，端口，磁带上的文件等等。尽管文件的形式和能力各不相同，但是所有的流都是相同的。这样做对我们程序员来说带来的好处就是：所有的硬件设备看起来都是一样的。流提供了兼容的接口。

流是通过打开操作和文件相关联的，并通过关闭操作和文件解除关联。

流分为两种：文本的和二进制的。文本的流处理的是字符。当我们使用文本流的时候，会发生一些字符转换。例如，当输出换行字符的时候，它可能会被转换成回车换行序列。这样一来发送给流的数据和写到文件中的数据就没有了一一对应的关系了。二进制流可以处理任何类型的数据。其中不会发生字符转换，并且发送给流的数据和实际写到文件中的数据是一一对应的。

还需要理解的一个概念就是当前位置。当前位置就是在一个文件中下一次访问即将发生的位置。例如，有一个文件是 100 字节长的，已经读取了一半，下一次读取操作将发生在第 50 个字节处，这就是当前位置。

总结一下，在 C++中，输入输出操作是通过一个逻辑的接口来进行的，它就是流。所有的流都有相同的属性，并且所有的流都是通过相同的输入输出函数来进行操作的，而不管它们关联的是什么样的文件类型。文件就是含有数据的物理实体。尽管文件有所不同，但是流都是一样的。（当然，有些设备并不能支持全部的操作，比如不支持随机访问操作等，那么与其相关联的流也就会不支持这样的操作。）

C++中预先定义的流

C++中预先定义了一些流。这些流在我们的程序开始执行的时候就会被自动地打开。它们是 cin，cout，cerr 和 clog。我们都知道，cin 是和标准输入相关的流，cout 是和标准输出相关的流。cerr 是和标准输出相关联的，clog 也是。它们的区别是 clog 流采用了缓冲机制，而 cerr 则没有。这就意味着，任何发送到 cerr 的数据都会被立刻输出，而发送到 clog 的数据只有当缓冲区满的时候才会被输出。一般来说，cerr 和 clog 流是用来进行程序调试和输出错误信息的。C++还会打开宽位字符的标准流：wcin，wcout，wcerr 和 wclog。这种流的存在时为了支持诸如汉语这样的使用了大字符集的语言。在本书中我们不使用这些流。缺省情况下，C++标准流是和控制台相关的，但是我们的程序是可以把它们重定向到别的设备或者文件上的。它们还可以由操作系统来进行重定向。

必备技能 11.2：C++中的流类

正如我们在第一篇中学习到的那样，C++中对输入输出系统的支持都是在<iostream>中提供的。在该头文件中定义了一些相当复杂的类层次的集合，它们用来支持输入和输出的操作。输入输出类是从一个系统模版类开始的。我们将在地 12 篇中学习到，模版定义了类的

通用形式，而不用具体指明这些类操作的数据的类型。标准 C++从这些模版中定义了两个具体的和输入输出库相关的类：其中一个是针对 8 位字符集的，一个是针对宽位字符集的。这些类和其它的类一样，我们不需要熟悉模版相关的知识也能很好地使用这些输入输出的功能。

C++中的输入输出系统是基于两个相关但是又有区别的模版类的。一个是从低级的输出输出类继承而来的，叫做 basic_streambuf。这个类提供基础的低级的输入和输出操作，并提供了对整个 C++输入输出系统的支持。我们没有必要直接使用 basic_streambuf 这个类，除非我们要进行高级的输入输出编程。类层次中我们经常用到的是从 basic_ios 派生的类。这是一个高级的输入输出类。它提供了格式化，错误检查以及输入输出流的状态信息。basic_ios 是多个派生类的基类，包括 basic_istream、asic_iostream 和 basic_iostream。它们可以分别北用来创建用于输入、输出、输入和输出的流。

正如我们解释的那样，输入输出库中创建了两个版本的输出输出类层次：一个是针对 8 位的字符集的；一个是针对宽位字符集的。本书中我们只讨论针对 8 位字符集的类，这是因为到目前位置，它是最常用到的。下面是模版类名称和基于字符的类名称的对应表：

模版类名称	对应的基于字符的类名称
basic_streambuf	streambuf
basic_ios	ios
basic_istream	istream
basic_ostream	ostream
basic_iostream	iostream
basic_fstream	fstream
basic_ifstream	ifstream
basic_ofstream	ofstream

我们将在本书的后面使用到上面的基于字符的类名。这些类也是我们在程序中要用到的类，也是在老版本的输入输出库中用到的名称。这也是为什么老版本的和新版本的输入输出库在源码级上是兼容的。

最后一点：类 ios 包含了许多成员函数和变量用来控制或者件事流的基本操作。我们将经常引用这些成员函数或者变量。牢记，只要我们在程序中包含了<iostream>，我们就可以访问这个重要的类了。

练习：

1. 什么是流？什么是文件？
2. 和标准输出相关联的类是哪个类？
3. C++中的输入输出是由一系列成熟的类层次来支持的，对吗？

必备技能 11.3：重载输入输出运算符

在前面的篇章中，当我们需要对和类相关的数据进行输入或者输出的时候，我们会使用成员函数来专门输入或者是输出类的数据。这种方式本身并没有什么不对的地方，但是 C++为我们提供了更好的方式来对类的数据进行输入或者输出：重载>>或者<<运算符。

在 C++中，输出运算符<<也被称为是插入运算符，因为它是被用来把数据插入到流中的。类似的，输入运算符>>也被称为是提取运算符，因为它是被用来从流中提取数据的。

在<iostream>中，插入运算符和提取运算已经被针对所有的内置数据类型进行了重载。下面我们来看看如何针对自己创建的类来定义这两个运算符。

创建插入运算

作为第一个简单的示例，我们来为 ThreeD 类创建插入运算符：

view plain

```
class ThreeD
{
public:
    int x, y,z;
    ThreeD(int a, int b, int c)
    {
        x = a; y = b; z = c;
    }
};
```

下面通过重载来实现对 ThreeD 对象插入运算。下面就是一种实现方法：

view plain

```
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";

    return stream; //返回该流

};
```

由于上面函数中的许多特性对所有的插入函数都是适用的，所以我们将仔细讨论一下上面的这个函数。首先，函数的返回值是一个 ostream 类对象的引用。这样做是必须的，是为了能在复合输入输出语句中可以和多个该类型的插入运算复合使用。接下来我们看到这个函数需要两个参数。其中第一个是出现在<<运算符左侧的流的引用。第二个是出现在<<运算

符右侧的对象。（如果我们愿意，第二个参数也可以是对某个对象的引用。）在函数的实现体中，把 ThreeD 类的对象的三个数值都进行了输出，并且返回了流对象。

下面是一个简单的用来演示插入运算的程序：

view plain

```
1. //演示自定义的插入运算符
2.
3. #include <iostream>
4. using namespace std;
5.
6.
7. class ThreeD
8. {
9. public:
10.     int x, y,z;
11.     ThreeD(int a, int b, int c)
12.     {
13.         x = a; y = b; z = c;
14.     }
15. };
16.
17. //显示 X，Y，Z 坐标-----ThreeD 类的插入运算
18. ostream &operator<<(ostream &stream, ThreeD obj)
19. {
20.     stream << obj.x << ", ";
21.     stream << obj.y << ", ";
22.     stream << obj.z << "\n";
23.
24.     return stream; //返回该流
25.
26. };
27.
28. int main()
29. {
30.     ThreeD a(1,2,3), b(3,4,5), c(5,6,7);
31.     cout << a << b << c;
32.
33.     return 0;
34. }
```

上面的程序输入如下：

1，2，3

3, 4, 5

5, 6, 7

如果我们删掉上面程序中针对 ThreeD 类的处理，就剩下了插入函数的框架如下：

```
ostream &operator<<(ostream &stream, class_type obj)
{
  //详细的和类相关的处理代码

  return stream; //返回该流
};
```

其中 obj 也可以采用传递引用的方式。

尽管在插入函数中具体要做些什么完全是由我们程序员决定的。但是，让插入函数进行合理的输出才是好的编程习惯。这里唯一确定的就是插入函数返回的是流。

使用友元函数来进行插入运算的重载

在上面示例程序中，被重载的插入运算函数并不是 ThreeD 类的成员函数。实际上，插入函数和提取函数都是不能作为类的成员函数的。其原因在于，如果一个运算符是类的成员，那么运算符左侧的操作数就应该是该类的对象。这点是不可改变的。然而，当对插入运算进行重载的时候，其左侧的运算数是流，而其右侧的运算数才是类的对象。因此，重载的插入运算只能是作为类的非成员函数。

上面说到的插入运算只能是以非成员函数的方式进行重载这一事实就引发了一个严重的问题：重载的插入运算怎么能访问类的私有成员了？在上面的示例程序中，变量 x，y，z 都是公有的，这样插入函数是可以访问它们的。但是面向对象的编程中重要的一点就是信息隐藏。因此上述强制地把所有数据都声明是共有的做法显然和这点相悖。这个问题的解决就要用到友元函数：插入函数作为类的友元函数。作为它所定义的类的友元函数，插入函数可以访问到类的私有数据。下面重写了前面的示例程序和 ThreeD 类。其中把对插入函数的重载是作为类的友元函数来实现的：

view plain

```
1. //演示自定义的插入运算符
2.
3. #include <iostream>
4. using namespace std;
5.
6.
7. class ThreeD
8. {
9.     //私有的数据
10.    int x, y,z;
11.public:
```

```
12.     ThreeD(int a, int b, int c)
13.     {
14.         x = a; y = b; z = c;
15.     }
16.
17.     //插入运算符作为该类的友元函数
18.     friend ostream &operator<<(ostream & stream, ThreeD obj);
19. };
20.
21. //显示 X，Y，Z 坐标-----ThreeD 类的插入运算
22. ostream &operator<<(ostream &stream, ThreeD obj)
23. {
24.     stream << obj.x << ", ";
25.     stream << obj.y << ", ";
26.     stream << obj.z << "\n";
27.
28.     return stream; //返回该流
29.
30. };
31.
32. int main()
33. {
34.     ThreeD a(1,2,3), b(3,4,5), c(5,6,7);
35.     cout << a << b << c;
36.
37.     return 0;
38. }
```

注意，其中的 x，y，z 现在都是类的私有成员，但是在插入函数中仍然可以被直接访问。这样把插入运算（和提取运算）作为它所定义的类的友元函数就保持了面向对象编程的封装原则。

重载提取运算

提取运算的重载和插入运算的重载是相同的。例如，下面的提取函数就把三维坐标输入到了 ThreeD 的对象中。注意，其中有提示用户进行输入。

view plain

```
1. istream &operator>>(istream &stream, ThreeD &obj)
2. {
3.     cout << "Enter x, y, z values:\n";
4.     stream >> obj.x >> obj.y >> obj.z;
5.
```

```
    return stream;
}
```

下面的程序演示了针对 ThreeD 类对象的提取运算：

view plain

```
//演示自定义的插入运算符

#include <iostream>
using namespace std;


class ThreeD
{
    //私有的数据
    int x, y,z;
public:
    ThreeD(int a, int b, int c)
    {
        x = a; y = b; z = c;
    }

    //插入运算符作为该类的友元函数
    friend ostream &operator<<(ostream &stream, ThreeD obj);
    friend istream &operator>>(istream &stream, ThreeD &obj);
};

//显示 X，Y，Z 坐标-----ThreeD 类的插入运算
ostream &operator<<(ostream &stream, ThreeD obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";

    return stream; //返回该流

};

//输入三维坐标
istream &operator>>(istream &stream, ThreeD &obj)
{
    cout << "Enter x, y, z values:\n";
```

```
37.     stream >> obj.x >> obj.y >> obj.z;
38.
39.     return stream;
40. }
41.
42. int main()
43. {
44.     ThreeD a(1,2,3);
45.
46.     cout << a;
47.
48.     cin >> a;
49.     cout <<a ;
50.
51.     return 0;
52. }
```

上面程序的运行如下：

1, 2, 3

Enter x, y, z values:

5 6 7

5, 6, 7

和插入运算类似，提取运算函数也是不能作为操作对象类的成员函数的。它只能是友元函数或者是独立的函数。

除了必须返回一个对 istream 对象的引用外，我们可以在提取运算函数中做任何自己想做的事情。但是为了程序的结构清晰，我们最好还是只让它完成输入操作即可。

格式化的输入输出

到目前为止，信息的输入或者是输出都是按照 C++输入输出系统提供的缺省格式进行的。然而，我们是可以通过下面两种方式之一来控制数据的格式的：第一就是使用 ios 类的成员函数。第二种方式就是使用一种特殊的函数叫做控制器。

练习：

1. 什么是插入运算？
2. 什么是提取运算？
3. 为什么在插入运算或者提取运算的中经常要用到友元函数？

基本技能 11.4：使用 ios 的成员函数进行格式化输入和输出

每一个流都有与之相关联的一套格式化标记，用来控制流对信息的格式化方式。类 ios 中定义了位图形式的枚举类型 fmtflags，其中定义了下面我们要使用的值。（从技术的角度来讲，这些值是定义在 ios_base 类中的，它是 ios 类的基类。）

这些值可以被用来设置或者清除格式化标记。一些旧的编译器中可能没有定义 fmtflags 这个枚举类型。此时，这些格式化标记将被以长整形数的形式进行编码。

当标记 skipws 被设置后，在对流进行输入的时候空白字符（空格，tab 符以及换行符）符将会被忽略。当标记 skipws 被清除后，这些空白字符将不会被忽略。

当标记 left 被设置后，输出将按照左对齐的方式进行。当标记 right 被设置的时候，输出将按照右对齐的方式进行。如果这两个标记都没有被设置，输出则按照右对齐的方式进行。

当标记 internal 被设置后，数值的符号将会采用左对齐的方式，而数值采用右对齐的方式进行输出。

缺省情况下，数字都是以 10 进制的格式输出的。然而我们是可以修改基数的。通过设置 oct 标记可以使得数值以八进制的形式输出；设置 hex 标记可以使得数值以 16 进制的格式进行输出。再次设置 dec 标记则可以恢复到缺省的 10 进制输出格式。

设置 showbase 标记则会在输出数值的时候也输出基数。比如，十六进制的 1F 在以 16 进制的格式输出的时候就是 0x1F。

缺省情况下，当以科学计数法的形式输出数值的时候，字符 e 是以小写的形式出现的。同样当以十六进制的格式输出数值的时候 x 也是以小写的形式出现的。当设置了 uppercase 标记后，这些字母都会以大写的形式出现。

设置 showpos 标志后，在输出整数的时候会在前面增加+号。设置 showpoint 标志后，在输出浮点数的时候不管是否需要都会输出十进制的小数点部分。

设置了 scientific 标志后，浮点数将会以科学技术法的格式进行输出。设置了 fixed 标志后，将固定的输出小数点后面的小数部分(小数部分输出的位数由设置的精度来确定)。如果这两个标记都没有被设置，编译器将自己选择一个合适的输出格式。

当设置了 unitbuf 标记后，输入输出缓冲区将在每一次插入操作之后都立刻把缓冲区中的数据进行输入输出。当 boolalpha 标记被设置后，则在输出布尔值的时候会输出 true 或者 false，输入的时候也可以输入 true 或 false。

最后，标记 scientific 和 fixed 则可以通过 floatfield 标记来一次性设置。

设置和清除格式化标记

我们可以使用函数 setf（）来设置一个标志。这个函数是 ios 的成员函数，它的常用形式如下：

```
fmtflags setf(fmtflags flags);
```

该函数打开由 flags 所指示的标记并返回先前的格式标记。例如，需要打开 showbase（该标记用于输出时显示整数字段基数的前缀）标记时，我们可以使用如下语句：

```
stream.setf(ios::showbase);
```

其中，stream 就是我们想要设置其格式标记的流对象。注意，这里使用到了 iso::来修饰 showbase。这是因为 showbase 是由 ios 类定义的一个枚举类型的常量，在引用这个常量的时候我们必须使用 ios 来进行限定。这种原则适用于所有的格式化标记。

下面的程序演示了如何使用 setf()函数来同时打开 showpos（该标记用于输出时在非负的数字字段前加上一个+符号）和 scientific(用科学计数法的方式来显示浮点数)标记：

view plain

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

int main()
{
    //打开 showpos 和 scientific 标记
    cout.setf(ios::showpos);
    cout.setf(ios::scientific);

    cout<< 123 << "  " << 123.23 << "";


    return 0;
}
```

上面程序的输出如下：

+123 +1.232300e+002

我们还可以在一次调用 setf()函数的时候把多个标记用 OR 运算符连接起来，一次性进行设置。例如，通过下面的语句中的把 showpos 和 scientific 进行一次性设置：

cout.setf(ios::showpos | ios::scientific);

需要关闭格式标记的时候可以使用 unsetf()函数。它的原型如下：

void unsetf(fmtflags flags); 由 flags 所指定的所有标记将会被关闭。

有时我们需要查询当前的格式标记设置情况，此时我们可以使用 flags()函数来获取当前的标记状态。它的原型如下：fmtflags flags();

这个函数返回与当前相关流的标记的当前值。flags()函数还有一种用法如下：fmtflags flags(fmtflags flags); 这种用法根据 flags 参数设置相关流的格式化参数，并返回之前的格式化标记值。

下面的程序演示了如何使用 flags()和 unsetf()函数：

view plain

```
//演示 flags()和 unsetf()函数的用法
```

```
2.
3.  #include "stdafx.h"
4.  #include <iostream>
5.  using namespace std;
6.
7.  int main()
8.  {
9.      ios::fmtflags f;
10.
11.     f = cout.flags();//获取当前的格式标记值
12.
13.     if ( f&ios::showpos)
14.     {
15.         cout << "showpos is set for count.\n";
16.     }
17.     else
18.     {
19.        cout << "showpos is cleared for cout.\n";
20.     }
21.
22.
23.     cout << "\nSetting showpos for cout.\n";
24.     cout.setf(ios::showpos);//设置 showpos 标记
25.     f = cout.flags();
26.     if ( f & ios.showpos )
27.     {
28.         cout << "showpos is set for count.\n";
29.     }
30.     else
31.     {
32.         cout << "showpos is cleared for cout.\n";
33.     }
34.
35.
36.     cout <<"/nClearing showpos for cout.\n";
37.     cout.unsetf(ios::showpos);
38.     f = cout.flags();
39.     if ( f & ios.showpos )
40.     {
41.     cout << "showpos is set for count.\n";
42.     }
43.     else
44.     {
45.         cout << "showpos is cleared for cout.\n";
```

```
46.     }
47.
48.     return 0;
49. }
```

上面程序的输出结果如下：

showpos is cleared for cout.

Setting showpos for cout.

showpos is set for count.

Clearing showpos for cout.

showpos is cleared for cout.

在上面的程序中，在使用类型 fmtflags 声明变量 f 的时候，类型前面使用了 ios::进行限定。这样做是必要的，这是因为 fmtflags 是由 ios 类定义的类型。通常情况下，当我们使用一个由类定义的类型名称或者枚举常量的时候，我们必须使用该类的名称来限定它。

设置宽度，精度和填充字符

除了上面谈到的格式化标记外，ios 类还提供了另外的三个函数，可以用来设置附加的格式控制信息：流的宽度，精度和填充字符。这三个函数分别是：width(), precision()和 fill()。我们将逐一进行讨论。

缺省情况下，当输出一个值的时候，它只占用对应数量的字符空间。然而，我们可以通过使用 width()函数来指定它所占用的最小宽度。该函数的原型如下：

```
streamsize width(streamsize w);
```

其中，w 指定了新的输出宽度，之前的输出宽度由函数返回。在某些实现中，流宽度的值必须在每次输出前都被设置。否则就使用缺省的宽度值。其中的 streamsize 是由编译器定义的整形类型。

在设置了流的最小宽度以后，如果当一个值输出时所占用的空间小于该指定的最小宽度，多余的空间就会被用当前填充字符进行填充（在缺省情况下填充字符就是空格）。如果一个值的输出空间超出了指定的最小宽度，则使用需要的真实空间数量进行输出，而不会对输出结果进行截断。

当以科学计算法输出浮点数的时候，我们可以使用 precision()函数来设置小数点后面的小数位数。该函数的原型如下：

```
streamsize precision(streamsize p);
```

其中，p 就是指定的精度，函数返回之前的精度值。缺省的精度为 6。在一些实现中，必须在每次输出浮点数之前都调用该函数来设置精度，否则将使用缺省精度进行输出。

缺省情况，如果输出的时候需要进行填充，则采用空格进行填充。我们可以通过使用函数 fill()来指定填充字符。它的原型如下：

```
char fill(char ch);
```

调用该函数后，字符 ch 将成为新的填充字符，函数返回之前的填充字符。

下面的程序演示了上面三个函数的使用方法：

view plain

```
1. //演示 width(),precision()和 fill()函数的用法
2.
3. #include "stdafx.h"
4. #include <iostream>
5. using namespace std;
6.
7. int main()
8. {
9.      cout.setf(ios::showpos);
10.    cout.setf(ios::scientific);
11.    cout << 123 << "    " << 123.23 <<  "\n";
12.
13.
14.     cout.precision(2); //十进制小数点后两位数
15.    cout.widen(10); //10 个字符的宽度
16.    cout << 123 << "    ";
17.    cout.widen(10); //10 个字符的宽度
18.    cout << 123.23 << "\n";
19.
20.
21.    cout.fill('#');//设置填充字符为#
22.    cout.width(10);//10 个字符的宽度
23.    cout << 123 << "    ";
24.    cout.width(10);//10 个字符的宽度
25.    cout << 123.23;
26.
27.
28.    return 0;
29. }
```

上面程序的输出结果如下:

+123 +1.232300e+002

+123 +1.23e+002

######+123   +1.23e+002

正如我们说过的那样，由于一些实现上的差异，在每次输出之前都必需重新设置输出宽度。这也是上面程序中为什么反复调用 width()函数的原因。这三个函数还有重载的形式，可以用来获取而不是修改当前的设置。其形式如下：

```
char fill();
streamsize width();
streamsize precision();
```

练习

1. boolalpha 是用来做什么的？
2. setf()是用来做什么的？
3. 我们是用什么函数来设置填充字符？

基本技能 11.5：使用控制器来进行格式化输入和输出

C++的输入和输出系统提供了第二种可用于修改流格式参数的方法。这种方法使用到了特殊的函数，叫做控制器。这种控制器可用于输入/输出表达式中。表格 11-1 列出了标准的控制器。在使用需要参数的控制器的时候我们必须在程序中包括<iomanip>。

控制器	引入的目的	输入/输出
boolalpha	打开 boolalpha 标记开关	输入/输出
dec	打开 dec 标记开关	输入/输出
endl	输出一个换行字符并强制流输出数据	输出
ends	输出一个空字符('/0')	输出
fixed	打开 fixed 标记开关	输出
flush	强制输出流中的所有数据	输出
hex	打开 hex 标记开关	输入/输出
internal	打开 internal 标记开关	输出
left	打开 left 标记开关	输出
noboolalpha	关闭 boolalpha 标记开关	输入/输出
noshowbase	关闭 showbase 标记开关	输出
noshowpoint	关闭 showpoint 标记开关	输出
noshowpos	关闭 showpos 标记开关	输出
noskipws	关闭 skipsw 标记开关	输入
nounitbuf	关闭 unitbuf 标记开关	输出
nouppercase	关闭 uppercase 标记开关	输出
oct	打开 oct 标记	输入/输出
resetosflags(fmtflags f)	关闭 f 参数中指定的标记	输入/输出

right	打开 right 标记	输出
scientific	打开 scientific 标记	输出
setbase(int base)	设置整数的基数	输入/输出
setfill(int ch)	设置填充字符	输出
setiosflags(fmtflags f)	打开 f 参数中指定的标记开关	输入/输出
setprescision(int p)	设置小数的精度	输出
setw(int w)	设置宽度	输出
showbase()	打开 showbase 标记开关	输出
showpoint()	打开 showpoint 标记开关	输出
showpos	打开 showpos 标记开关	输出
skipws	打开 skipws 标记开关	输入
unitbuf	打开 unitbuf 标记开关	输出
uppercase	打开 uppercase 标记开关	输出
ws	跳过前导的空白字符	输入

表格 11-1 C++中的输入/输出控制器

这些格式控制器通常是作为较大的输入/输出表达式的一部分来使用的。下面的程序就演示了如何使用上面提到的控制器来对输出格式进行控制：

view plain

```
1. //演示输入/输出控制器的使用
2. #include <iostream>
3. #include <iomanip>
4. using namespace std;
5. int main()
6. {
7.     cout << setprecision(2) << 1000.234 << endl;
8.     cout << setw(20) << "Hello there.";
9.     return 0;
10. }
```

上面程序的输出如下：

1e+003

Hello there.

注意在上面程序中控制器是如何在输入输出的链式表达式中使用的。另外，当一个控制器不需要参数的时候，例如示例程序中的 endl，其后面是不需要括号的。

下面的程序演示了使用 setiosflags()来设置科学计数和现实符号标记

view plain

```
1. //演示 setiosflags()
2. #include<iostream>
3. #include <iomanip>
```

```
using namespace std;
int main()
{
    cout << setiosflags(ios::showpos)
    << setiosflags(ios::scientific)
    << 123 << "  " << 123.23;
    return 0;
}
```

接下来的程序演示了如何在输入字符串的时候使用 ws 来跳过前导空白：

view plain

```
//跳过前导的空白
#include <iostream>
using namespace std;
int main()
{
    char s[80];
    cin >> ws >> s;
    cout << s;
    return 0;
}
```

基本技能 11.6：创建自己的控制器函数

C++允许我们创建自己的控制器函数。控制器函数分为两种：带参数的和不带参数的。自行定义带参数的控制器函数涉及到的知识超出了本书所要涉及的内容，我们在此不做讨论。创建不带参数的控制器函数则是比较简单的，将在下面进行讨论。

所有不带参数的控制器输出函数都有着下面的结构：

```
ostream &manip_name(ostream &stream)
{
    // 自己的代码
    return stream;
}
```

其中，manip_name 就是控制器的名字。有重要的一点需要理解：尽管上面的形式中控制器参数为一个指向流的指针，但是在输出表达式中使用该控制器的时候是不需要指定任何参数的。

下面的程序中，我们创建了一个名为 setup()的控制器，用来打开左对齐的标记开关，设置宽度为 10，并且设置填充字符为美元符号。

view plain

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

//自定义控制器
ostream &setup(ostream & stream )
{
    stream.setf(ios::left);
    stream << setw(10) << setfill('');
    return stream;
}

int main()
{
    cout << 10 << " " << setup << 10;
    return 0;
}
```

自定义控制器在下面两种情况下会显得非常用用：第一，在我们需要对某种设备进行输入/输出操作，但是系统原先定义的控制器不能满足我们的要求，例如绘图仪。此时创建自己的控制器会使得输出到绘图仪的操作更加方便。第二，我们有可能在输入/或者输出的时候多次重复一段相同的格式控制语句，那么我们就可以把这种操作合并为一个简单的控制器，如下面的程序演示的那样。

例如，在下面的程序中，我们创建了一个名称为 prompt()控制器。它用于显示提示信息，然后配置输入格式为 16 进制。

view plain

```cpp
//创建一个收入控制器
#include <iostream>
#include <iomanip>

using namespace std;

istream &prompt(istream &stream)
{
    cin >> hex;
    cout << "Enter number using hex format:";

    return stream;
}
```

```
15.
16.
17. int main()
18. {
19.     int i;
20.     cin >> prompt >> i;
21.     cout << i;
22.
23.
24.     return 0;
25. }
```

必须注意的是：自定义的控制器必须返回一个流对象；否则，这个控制器将不能在输入或者输出的链式表达式中使用。

文件输入/输出

我们可以使用 C++的输入/输出系统来进行文件的输入/输出。此时，我们需要在程序中包含头文件<fstream>，其中定义了几个非常重要的类和值。

练习

1. endl 是用来做什么的？
2. ws 是用来做什么的？
3. 输入/输出控制器通常是被用作更大的输入/输出表达式的一部分，对吗？

基本技能 11.7: 打开和关闭文件

在 C++中，是通过把文件和流进行关联来打开文件的。正如我们知道的那样，有三种类型的流：输入流，输出流，以及输入/输出流。我们通过声明 ifstream 类的流对象来生成一个输入流；通过声明 ofstream 类的流对象来生成输出流；同时需要进行输入和输出的流必须声明为 fstream 类。例如，下面的代码段创建了一个输入流，一个输出流和一个能同时进行输入和输出的流。

ifstream in; //输入流

ofstream out; //输出流

fstream both; //既可以进行输入也可以进行输出的流

一旦我们创建了一个流，我们就可以使用 open()方法来把它和一个文件关联起来。这个方法是这三种流都有的。它们的原型如下：

void ifstream::open(const char *filename,

ios::openmode mode = ios::in);

void ofstream::open(const char *finename,

ios::openmode mode = ios::out|ios::truc);

void fstream::open(const char *filename,

ios::openmode mode = ios::in|ios::out);

其中，filename 就是文件的名称，其中是可以包含路径的。mode 的值决定了文件是以什么方式被打开的。它的取值必须至少是下面的一个。这些值都是 ios 类（通过他们的基类 iso_base）定义的枚举值。

ios::app

ios::ate

ios::binary

ios::in

ios::out

ios::trunk

我们可以通过或(|)运算来把它们合并起来。其中 ios::app 会使得针对该文件的所有输出都会被追加到文件的末尾。这个值只适用于可用于输出的文件。ios::ate 会使得在打开文件的时候查询到文件的末尾;尽管如此，我们还是可以在文件的任意位置进行输入或者输出操作的。

ios::in 指定文件可用于输入。

ios::out 指定文件可用于输出。

ios::binary 指定文件以二进制格式打开。缺省情况下，所有文件都是以文本的模式被打开的。在文本模式打开文件的时候会进行不同的字符转换。例如，控制字符 CRLF 会被转换为换行。然而，当以二进制格式打开文件的时候，这样的转换是不会发生的。必须注意的是:不管是格式化的文本还是原始数据文件都可以是以二进制或者文本方式打开的;唯一的区别就在于是否进行字符转换。

ios::trunc 会使得先前存在的同名文件被销毁，文件长度会被消减为 0。当我们使用 ofstream 输出流的时候，任何先前存在的同名文件都会自动地被销毁，长度消减为 0。

下面的代码段就以文本方式打开一个文件进行输出：

ofstream mystream;

mystream.open(“test”);

由于 open()方法的参数 mode 缺省地就是和流的类型是相对应，通常情况下我们没有必要去指明它的值，就像上面的例子那样。（可能会有些编译器并没有实现把 fstream::open()方法参数 mode 缺省值设置为 in|out，此时就需要显示指定了。）

如果 open()方法打开文件失败了，流对象会在布尔表达式中被当做 false 值。我们可以利用这一点来使用类似下面的语句来对打开文件操作是否成功进行确认：

view plain

```
1.  if(!mystream)
2.  {
3.      cout << “Cannot open file!.\n”;
```

```
4.      //错误处理
5.  }
```

通常情况下，我们在访问文件前总是应该对 open()方法是否成功进行检查。我们还可以使用 is_open()方法来完成这个功能。它也是 fstream，ifstream 和 ofstream 类的成员，原型如下：

bool is_open();

如果流和打开的文件成功关联了，则返回 true，否则返回 false。例如，下面的语句用于检查 mystream 流是否是打开的：

view plain

```
1.  if(!mystream.is_open())
2.  {
3.      cout <<"File is not open.\n";
4.      //…
5.  }
```

尽管使用 open()方法来打开一个文件是完全正确的，但是大多数情况下我们不这样做。这是因为 ifstream,ofstream 和 fstream 都有自动打开文件的构造函数。这些构造函数的参数及其缺省值都是和 open()方法一致的。因此，最常用的打开文件的语句如下：

ifstream mystream("text");//为输入而打开文件

如果由于某些原因导致文件不能被打开，相关流变量的值也会被当做是 false。关闭文件的时候我们使用 close()方法。例如，我们使用下面的语句关闭和 mystream 相关联的文件：

mystream.close();

close()方法不需要任何参数，也不返回任何值。

基本技能 11.8：读写文本文件

最简单的读取文本文件或者写入到文本文件的方法就是使用<<和>>操作符。例如，下面的程序写入一个整型数，浮点数和字符串到一个名称为 test 的文件中：

view plain

```
1.  //写入文件
2.  #include <iostream>
3.  #include <fstream>
4.  using namespace std;
5.  int main()
6.  {
7.      ofstream out("test");
8.      if (!out)
9.      {
10.         cout << "Cannot open file.\n";
```

```
11.             return 1;
12.     }
13.     out << 10 <<" " << 123.23 << "\n";
14.     out << "This is a short text file.";
15.     out.close();
16.     return 0;
17. }
```

下面的程序则从上面程序写入的文件中读出一个整型数，一个浮点数，一个字符和一个字符串:

view plain

```
1.  //读取文件
2.  #include <iostream>
3.  #include <fstream>
4.  using namespace std;
5.  int main()
6.  {
7.      char ch;
8.      int i;
9.      float f;
10.     char str[80];
11.     ifstream in("test");
12.     if( !in)
13.     {
14.         cout <<"Cannot open file." ;
15.         return 1;
16.     }
17.     in >> i;
18.     in >>f;
19.     in >> ch;
20.     in >>str;
21.     cout <<  i  << " " << f << " " << ch <<"\n";
22.     cout << str;
23.     in.close();
24.     return 0;
25. }
```

必须明确的是当我们使用>>操作符来读取文本文件的时候，是会进行一定的字符转换的。例如，空白字符将会被忽略。如果我们不想进行这样的字符转换，我们必须以二进制方式打

开文件。另外，还要明确的是，当使用>>读取字符串的时候，输入会在遇到第一个空白字符的时候就停止了。

专家答疑：

问题：正如你在篇章 1 中讲到的那样，C++是 C 的超集。那我知道在 C 中定义了自己的输入/输出系统。这个系统在 C++中可以使用吗？如果可以使用，我们应该在 C++程序中使用这个系统吗？

解答：对于第一个问题的答案是肯定的。C 语言中的输入/输出系统在 C++中也是可以使用的。对于第二个问题的答案则不完全是否定的。C 语言中的输入/输出系统不是面向对象的。然而这个系统现在还被广泛地使用着，它的效率高，开销小。因此，在一些专业性较强的程序中，选择 C 语言的输入/输出系统则是不错的选择。关于 C 语言中的输入/输出系统，更多的信息可以参见我的饿另外一本书《C++完全参考手册》

练习：

1. 那个类可用来创建一个输入文件？
2. 那个函数可用来打开一个文件？
3. 我们是否可以使用<<和>>来对文件进行输出和输入呢？

基本技能 11.9：非格式化的二进制文件的输入/输出

尽管文本文件的读写是比较简单的，但这并不是处理文件的最有效的方式。另外，有时候我们是需要存储非格式化的二进制数据的，而不是文本数据。我们将在下面描述实现这些功能的函数。

当我们对一个文件进行二进制操作的时候，必须确保它是使用 ips::binary 模式被打开的。尽管这些非格式化的文件操作函数对于文本模式的文件也是起作用的，但是其中是会进行字符转换的。字符转换与我们进行二进制文件操作的目的是相背的。

通常情况下，有两种方式来从二进制文件中读取或者是向二进制文件中写入数据的。第一种方法就是使用 put()函数向文件中写入一字节，使用 get()从文件中读入一个字节；第二种方法就是使用块函数进行输入输出：read()和 write()。下面逐一进行讨论、

使用 get()和 put()

get()函数由很多种形式，但是最常用的是如下的形式，put()函数也是一样：

```
istream & get(char *ch);
ostream &put(char ch);
```

其中，get()函数从相关联的流中读取一个字符，并把值赋给变量 ch。这个函数返回相关流的引用。如果没有到达文件的结尾，该函数返回非空值。put()函数是把 ch 的值写入到相关的流中并返回对这个流的引用。

下面的程序将会把任意文件的内容显示在屏幕上。其中使用到了 get()函数：

view plain

```
1. //使用 get()来显示一个文件的内容
2. #include <iostream>
3. #include <fstream>
4.
5. using namespace std;
6.
7. int main(int argc, char * argv[])
8. {
9.     char ch;
10.
11.     if (argc != 2)
12.     {
13.         cout <<"Usage: PR<filename> \n";
14.         return 1;
15.     }
16.
17.     ifstream in(argv[1], ios::in | ios::binary);
18.     if (!in)
19.     {
20.         cout << "Cannot open file.\n";
21.         return 1;
22.     }
23.
24.     while( in)//in 的值将会是 false，如果遇到了文件末尾
25.     {
26.         in.get(ch); //in
27.         if(in)
28.         {
29.             cout << ch;
30.         }
31. }
32.     in.close();
33.
34.     return 0;
35. }
```

请注意其中的 while 循环。当读取到文件末尾的时候 in 在布尔表达式中的值将会是 false，这样 while 循环就结束了。

其实还有一种更简便的从文件中读取字符并显示该字符的循环方式，如下：

view plain

```
while(in.get(ch))
{
    cout<<ch;
}
```

之所以这样写也是可以的，是因为 get()函数返回的是 in 的流，当遇到文件结尾的时候 in 将会被认为是 false。下面的程序使用 put()函数来吧一个字符串写入到文件中。

view plain

```
//使用 put()函数来向文件写入数据
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char * p ="hello there.\n";

    ofstream out("test", ios::out | ios::binary);
    if (!out)
    {
        cout << "Cannot open file.\n";
        return 1;
    }

    //写入字符到文件中，知道遇到字符串结束标记
    while(*p)
    {
        out.put(*p++); //使用 put 把字符串写入到文件中，将不会进行字符转换。
    }
    out.close();

    return 0;
}
```

执行该程序后，文件 test 的内容将是字符串”hello there.”后接着一个换行字符。这种方式不会进行字符转换。

读取和写入块数据

我们使用 read()和 write()来读取和写入块数据。它们的原型如下：

istream &read(char *buf, streamsize num);

ostream &write(const char *buf, streamsize num);

其中的 read()函数从相关联的流中读取 num 指示的字节大小的数据，并把这些数据存储在 buf 指示的空间中。write()函数则是把从 buf 开始的大小为 num 的空间中的数据写入到相关联的流中。正如我们在前面提到的那样，streamsize 是由 C++库定义的原型为整型数的类型，它的表示范围足以表示任何一种输入输出操作可以传输的字节的数量。

下面的程序先把一组整型数写入文件，然后再从文件中出入：

view plain

```
1.  //使用 read()和 write()函数
2.
3.  #include <iostream>
4.  #include <fstream>
5.  using namespace std;
6.
7.  int main()
8.  {
9.      int n[5] = {1,2,3,4,5};
10.     register int i;
11.     ofstream out("test",ios::out|ios::binary);
12.     if(!out)
13.     {
14.         cout <<"Cannot open file.\n" ;
15.         return 1;
16.     }
17.     out.write((char*)&n,sizeof(n)); //写块数据到文件中
18.     out.close();
19.
20.     for( i = 0;i < 5; i++) //清除数组中的数据
21.     {
22.         n[i] = 0;
23.     }
24.
25.     ifstream in("test",ios::in|ios::binary);
26.     if(!in)
27.     {
28.         cout << "Cannot pen file.\n";
29.         return 1;
30.     }
```

```
31.     in.read((char*)&n, sizeof(n));//从文件中读取块数据
32.
33.     for( i = 0; i< 5; i++)
34.     {
35.         cout << n[i] << " ";
36.     }
37.
38.     in.close();
39.
40.     return 0;
41.
42. }
```

请注意，在上面的程序中我们在调用函数 read()和 write()的时候都进行了强制的类型转换。如果一个缓冲区不是以字符数组的形式定义的，那么这样做时完全有必要的。

如果在读取 num 指定字节的时候遇到了文件结尾，则函数会停止从流中读取数据，并把当前读取到的可用的字节数的数据放入到缓冲区中。我们可以通过调用 gcount()来查询实际上读取到得字符的数量。它的原型如下：

streamsize gcount();

该函数返回的是最后一次进行输入操作的字符的数量。

练习

1. 如果想进行二进制的读写，则文件应该以什么模式进行打开？
2. get()和 put()函数是用来做什么用的？
3. 哪个函数可以被用来读取块数据？

基本技能 11.10：更多的输入/输出函数

除了前面介绍的这些输入/输出函数之外，C++中还定义了其他的可用于输入/输出的函数。其中一些也是很有用的。下面我们就对其中部分进行介绍。

get()函数的不同版本

除了前面介绍的基本用法外，get()函数还有其他不同的重载形式。其中最常用的三种重载的形式如下：

istream &get(char *buf, streamsize num);

istream &get(char *buf, streamsize num, char delim);

int get();

第一种形式：读取字符并存储到大小为 num 个字节的 buf 指示的空间中，直到读取的字符数量为 num-1 个，或者遇到了一行的结束，或者遇到了文件结束。其中 buf 指示空间中的内容将是以’/0’结束的字符串。如果在输入的流中遇到换行符，则该字符依然保留在流中。

第二种形式：读取字符并存储到大小为 num 个字节的 buf 指示的空间中，直到读取的字符数量为 num-1 个，或者遇到了由 delim 指定的字符，或者遇到了文件结束。buf 指示的空间内容将是以’/0’结束的字符。如果是遇到了 delim 指定的字符，则这个字符会保留在流中。

第三种形式：被重载的第三种形式返回流中的下一个字符。遇到文件结束的时候返回 EOF。EOF 是在<iostream>定义的。

get()函数一个很好的用法就是用来读取包含了空白字符的字符串。我们都知道当使用>>来读取字符串的时候，遇到空白字符就会停止读取操作。这就使得>>不能完成读取含有空白字符的字符串。然而，我们可以使用 get(buf,num)来解决这个问题，如下：

view plain

```
1. //使用 get()函数来读取含有空白字符的字符串
2. #include <iostream>
3. #include <fstream>
4. using namespace std;
5. int main()
6. {
7.     char str[80];
8.     cout << "Enter your name:";
9.     cin.get(str,79);
10.    cout << str << "\n";
11.    return 0;
12.}
```

其中，这里使用的 get()函数缺省的界定符就是换行符。这样就使得 get()函数可以像标准的 gets()函数一样工作了。

getline()

另外一个用于输入的函数就是 getline()。他是输入流的成员函数，其原型如下：

istream &getline(char *buf, streamsize num);

istream &getline(char *buf, streamsize num, char delim);

第一种形式：读取字符并存储在 buf 指示的空间中，直到读取了 num-1 个字符，或者遇到了换行字符，或者遇到文件结束。buf 指示的空间的内容将是以’/0’结束的字符串。如果在读取的过程中遇到了换行字符，该字符会被摘出，不会放入 buf 空间中。

第二种形式：读取字符并存储在 buf 指示的空间中，直到读取了 num-1 个字符或者遇到了 delim 指示的字符，或者遇到了文件结束。buf 指示的空间内容将是以'/0'结束的字符串。如果在读取的过程中遇到了 delim 指定的字符，则该字符会被摘出，不会被放置在 buf 指示的空间中。

从上面的描述可以看出，这两个版本的 getline()实际上和 get(buf,num)以及 get(buf,num,delim)的功能是一样的：都是从输入流中读取字符，并把字符放置在 buf 指示的空间中，直到读取字符的数量为 num-1 或者是遇到由 delim 指定的界定符。唯一的区别就在于 getline()函数会从输入流中读取并摘出界定符，而 get()函数不会。

判断是否遇到了文件结尾

我们可以通过使用成员函数 eof()来判断是否遇到了文件结尾，其原型如下：

bool eof();

当遇到文件结尾的时候，函数返回真值，否则返回假值。

peek()和 putback()

我们可以通过使用 peek()函数来窥探一下输入流中的下一个字符是什么而不用把他从流中移除。其函数原型如下：

int peek();

peek()函数返回输入流中的下一个字符；如果遇到文件结束则返回 EOF。返回的字符存储在返回值的低字节中。我们可以通过使用 putback()函数把从流中读取到得最后一个字符重新放回到流中（原文如下：

You can return the last character read from a stream to that stream by using putback()。这种描述容易让人误认为该函数是用来把读取的最后一个字符重新放回到输入的流中。其实是可以把任意的由参数 c 指示的字符放回到流中的。)。其函数原型如下：

istream &putback(char c);

其中 c 是读取的最后一个字符（c 是指定的任意字符）。

flush()

在进行输出的时候，数据不会立刻被写入到与流相关联的物理设备中去的；而是被存储在一个内部的缓冲区中，直到缓冲区满了才会被写入到设备中。然而，我们可以使用 flush()函数来强迫性地把数据写入到物理设备中。其函数原型如下：

ostream &flush();

调用该函数在很多恶劣的环境下是很合理的。（比如在经常出现电力短缺的情况下）。

注意：关闭文件或者终止程序也会强制性地把缓冲区的数据写入到相关联的设备中去。

工程 11-1 一个简单的文件比较工具

通过这个工程，我们创建一个简单而实用的文件比较工具。程序通过对两个文件进行读取和比较的操作来判断文件是否相同。如果找到了一处不相同的地方，则两个文件内容不同。

如果同时遇到两个文件的结束，并且没有发现不同之处，则说明两个文件的内容是完全相同的。

步骤：

1. 创建一个名称为 CompFiles.cpp 的文件。
2. 在该文件中增加如下的代码：

view plain

```
#include <iostream>
#include <fstream>
using namespace std;
int main( int argc, char *argv[])
{
    register int i;
    unsigned char buf1[1024], buf2[1024];
    if ( argc != 3)
    {
        cout <<"Usage: comfiles <file1> <file2>\n";
        return 1;
    }
```

注意：两个文件的名称是通过命令行指定的。

3. 增加以二进制方式打开两个文件的代码。如下：

view plain

```
ifstream f1(argv[1], ios::in | ios::binary);
    if ( !f1 )
    {
        cout << "Cannot open fist file.\n";
        return 1;
    }
    ifstream f2(argv[2], ios::in | ios::binary);
    if ( !f2 )
    {
        cout << "Cannot open second file.\n";
        return 1;
}
```

我们以二进制格式打开文件是为了避免文本方式打开时有可能进行的格式字符转换操作。

4. 增加进行实际比较的代码，如下：

view plain

```
do
    {
        f1.read((char*)buf1, sizeof(buf1));
        f2.read((char*)buf2, sizeof(buf2));
        if(f1.gcount() != f2.gcount())
        {
            cout << "Files are of differsing sizes.\n";
            f1.close();
            f2.close();
            return 0;
        }
        for ( i = 0; i < f1.gcount(); i++)
        {
            if (buf1[i] != buf2[i])
            {
                cout << "Files differ.\n";
                f1.close();
                f2.close();
                return 0;
            }
        }
    } while (!f1.eof() && !f2.eof());
    cout << "Files are the same.\n";
```

上面的这段代码使用 read()函数每次都从两个文件中读取等量的数据，然后对缓冲区中的数据进行比较。如果发现不同之处，则关闭两个文件，并在屏幕上显示“Files differ.”，然后程序结束。否则，继续从两个文件中读取数据到缓冲区中，并进行比较，直到遇到一个文件结束。由于在遇到文件结束的时候很有可能读取到得数据没有能够填满缓冲区，在程序中我们用到了 gcount()来精确判断缓冲区中数据的多少。如果其中一个缓冲区中的数据数量小于另外一个缓冲区，那么在遇到一个文件结束的时候两次 gcount()返回的值将会不相等，此时程序会显示“Files are of differing sizes.”。

5. 最后，如果两个文件的内容相同，此时当遇到一个文件结束的时候，另外一个文件的读取也会结束。通过调用 eof()函数我们可以确定这一点。如果两个文件的内容和和长度都相等，则这两个文件是相同的。

   最后，程序关闭两个文件，如下：

view plain

```
f1.close();
f2.close();
return 0;
}
```

6. 完整的 FileComp.cpp 程序如下：

view plain

```
/*
    工程 11-1
    创建文件比较工具
*/
#include <iostream>
#include <fstream>
using namespace std;
int main( int argc, char *argv[])
{
    register int i;
    unsigned char buf1[1024], buf2[1024];
    if ( argc != 3)
    {
        cout <<"Usage: comfiles <file1> <file2>\n";
        return 1;
    }
    ifstream f1(argv[1], ios::in | ios::binary);
    if ( !f1 )
    {
        cout << "Cannot open fist file.\n";
        return 1;
    }
    ifstream f2(argv[2], ios::in | ios::binary);
    if ( !f2 )
    {
        cout << "Cannot open second file.\n";
        return 1;
    }
    cout << "Comparing files...\n";
    do
    {
        f1.read((char*)buf1, sizeof(buf1));
        f2.read((char*)buf2, sizeof(buf2));
        if(f1.gcount() != f2.gcount())
```

```
35.         {
36.             cout << "Files are of differsing sizes.\n";
37.             f1.close();
38.             f2.close();
39.             return 0;
40.         }
41.         for ( i = 0; i < f1.gcount(); i++)
42.         {
43.             if (buf1[i] != buf2[i])
44.             {
45.                 cout << "Files differ.\n";
46.                 f1.close();
47.                 f2.close();
48.                 return 0;
49.             }
50.         }
51.     } while (!f1.eof() && !f2.eof());
52.     cout << "Files are the same.\n";
53.     f1.close();
54.     f1.close();
55.     return 0;
56.
57.
58. }
```

7. 运行程序的时候，我们可以复制 CompFile.cpp 的内容到一个文本文件 temp.txt 中。然后在命令行键入:CompFiles CompFiles.cpp temp.txt
   程序会输出两个文件是相同的。我们还可以把 CompFiles.cpp 和别的文件进行比较，比如可以和别的章节中的程序文件进行比较，可以看到程序显示文件是不同的。
8. 我们还可以自己增强上面的程序。比如，增加选择，用于在比较文件的时候或略字符的大小写；还可以增加选项用来显示两个文件是在什么地方开始不相同的。

基本技能 11.11：随机访问文件

到目前为止，我们一直都是按照顺序的方式来读取文件。实际上我们是可以以随机的方式来对文件进行访问的。在 C++的输入/输出系统中，我们可以使用 seekg()和 seekp()函数来随机访问文件。他们最常用的形式如下：

```
istream &seekg(off_type offset, seekdir origin);
ostream &seekp(off_type offset, seekdir origin);
```

其中：off_type 是由 ios 定义的整形类型。它的取值范围足以用于表示有效的偏移量。seekdir 是一个可以取下列值得枚举类型：

值	含义
ios::beg	文件的开始
ios::cur	当前位置
ios::end	文件的末尾

C++的输入/输出系统了管理了两个和文件相关联的指针。一个就是读取文件的指针，它表明了下一次读取的位置；还有一个就是写入文件的指针，它指示了下一次写入操作的位置。每进行一次输入或者输出操作，相应的指针就会自动的向前进行移动。通过使用 seekg()和 seekp()这两个函数，我们就可以移动这两个指针，实现对文件的随即访问。

函数 seekg()把相关联的文件的读取指针从指定的起点移动 offset 个字节；seekp()函数把相关文件的写入指针从指定的起点移动 offset 个字节。

通常来说，随即访问进行输入输出仅用于二进制文件的操作。这是因为文本文件读取时的字符转换有可能导致指针的位置和文件实际内容中的位置不能正确对应。

下面的程序就演示了函数 seekp()的用法。程序运行时需要我们在命令行输入文件的名称以及我们期望的文件被修改的位置，程序将在这个位置写入一个 X 字符。请注意其中的文件必须是以读写的方式打开。

view plain

```
1.  //文件的随机访问
2.
3.  #include <iostream>
4.  #include <fstream>
5.  #include <cstdlib>
6.
7.  using namespace std;
8.
9.  int main(int argc, char * argv[])
10. {
11.     if(argc != 3)
12.     {
13.         cout << "Usage: CHANGE<filename> <byte> \n";
14.         return 1;
15.     }
16.
17.     fstream out(argv[1], ios::in | ios::out | ios::binary);
18.     if ( !out )
19.     {
20.         cout << "Cannot ope file.\n";
21.         return 1;
22.     }
```

```
23.
24.     out.seekp(atoi(argv[2]), ios::beg);
25.
26.     out.put('X');
27.     out.close();
28.
29.     return 0;
30.
31. }
```

接下来的程序中使用到了 seekg()函数。程序显示从用户指定的位置开始到文件结束的内容。

view plain

```
1.  //从指定的位置开始显示文件的内容
2.  #include <iostream>
3.  #include <fstream>
4.  #include <cstdlib>
5.
6.  using namespace std;
7.
8.  int main(int argc, char * argv[])
9.  {
10.     char ch;
11.
12.     if ( argc != 3)
13.     {
14.         cout << "Usage : NAME <file name> <starting location>\n";
15.         return 1;
16.     }
17.
18.     ifstream  in(argv[1], ios::in|ios::binary);
19.     if (!in)
20.     {
21.         cout << "Cannot open file.\n";
22.         return 1;
23.     }
24.
25.     in.seekg(atoi(argv[2]),ios::beg);
26.
27.     while(in.get(ch))
28.     {
```

```
29.         cout << ch;
30.     }
31.
32.     return 0;
33. }
```

我们还可以通过下面的两个函数来获取两个指针的当前位置：

```
pos_type tellg();
pos_type tellp();
```

其中的 pos_type 是有 ios 定义的能够涵盖上述两个函数中任意一个返回值的最大值的类型。

seekg()和 seekp()函数还有如下的重载形式：

```
istream & seekg(pos_type positon);
ostream &seekp(pos_type position);
```

可以把文件指针移动到指定的位置。

练习

1. 那个函数可以用来判断是否遇到了文件结束？
2. getline()函数是用来做什么的？
3. 那些函数可用于对文件进行随机的访问？

基本技能 11.12：检查输入输出的状态

C++的输入输出系统针对每一个输入输出操作都维护其状态信息。输入输出流的状态信息是通过一个 iostate 类型的对象来描述的。这个对象是一个枚举类型，取值的含义如下：

名称	含义
ios::goodbit	没有错误发生
ios::eofbit	当遇到文件结尾的时候，被置为 1；否则置为 0
ios::failbit	当有可能发生非致命性的输入输出错误的时候被置为 1；否则置为 0
ios::badbit	当发生致命性输入输出错误的时候被置为 1,；否则被置为 0

我们有两种方式来获取输入输出的状态信息。第一，就是调用 rdstate()函数。他的原型如下：

```
iostate rdstate();
```

函数返回当前的错误标记。从上面的 iostate 类型的取值列表中我们可以猜测的出来，rdstate()函数在没有错误出现的情况下返回的是 ios::goodbit；否则返回的是错误对应的值。

另外一种判断是否发生了某种错误的方法就是使用 ios 的一个或者多个成员函数：

```
bool bad();
bool eof();
```

```
bool fail();
bool good();
```

其中的 bool eof()我们之前已经讨论过了。函数 bad()在 badbit 标志位被设置的时候返回 true。fail()函数在 failbit 被设置的时候返回 true。good()函数在没有错误的情况下返回 true。其他情况下他们都返回 false。

一旦发生了错误，我们应该在程序继续运行之前清除这些错误。此时我们可以使用函数 clear()，他的原型如下：

```
void clear(iostate flags = ios::goodbit);
```

当其中的参数 flags 为缺省的 ios::goodbit 的时候，所有的错误标记都会被清除。否则，我们可以把 flags 标记设置为自己需要清除的错误标记。

在继续进行下面的内容之前，我们可以在之前的程序中加入这里提到的这些状态报告函数来对我们程序的错误检查力能进行扩展。

本章复习题

1. 系统预先定义好的四个流是什么？
2. C++中是否同时定义了 8 比特的和宽位的字符流？
3. 写出重载插入运算符的通用形式 ？
4. ios::scientific 是用来做什么的？
5. width()是用来做什么的？
6. 输入/输出控制器是用在输入/输出表达式中的，对吗？
7. 写出如何以文本方式打开一个文件进行输入？
8. 写出如何以文本方式打开一个文件进行输出？
9. ios::binary 是用来做什么的？
10. 当遇到文件结尾的时候，流变量将被当做是 false 值，对吗？
11. 假定一个文件和一个名为 strm 的输入流相关联，写出如何读取文件的内容直到文件结束？
12. 编写一个用于复制文件的程序。程序允许用户在命令行输入原文件名称和目的文件名称。确保程序既可以复制文本文件也可以复制二进制文件。
13. 编写一个程序用来合并两个文本文件。程序允许用户以文件内容出现的先后顺序输入两个文件的名称。同时允许用户输入合并后文件的名称。也就是说，如果程序的名称为 merge，那么用于把 MyFile1.txt 和 MyFile2.txt 合并成 Target.txt.的命令行应该为：

    ```
    merge MyFile1.txt MyFile2.txt Target.txt
    ```

14. 写出如何使用 seekg()函数来定位到一个名称为 Mystrm 流的第 300 个字节处。

第十二篇 异常、模板和其他高级主题

到目前为止我们已经学习了很多关于 C++的知识。在本书的最后一章中，我们将学习到几个重要的 C++高级特性。它们是：异常处理，模板，动态分配和命名空间。本章还将学习运行时类型标识和强制转换运算符。请记住，C++是一个强大的、成熟的、专业的编程语言。在作为初级教材的本书中我们不可能把所有的高级特性，特殊的技术和编程技巧都一一涵盖。但是，学习完本章后，我们就已经掌握了 C++语言的核心，并能够开始编写真实的程序了。

必备技能 12.1：异常处理

异常就是运行时出现的错误。使用 C++中提供的异常处理机制，我们可以以一种结构化的，可控的方式来处理运行时的错误。当我们使用异常处理机制时，在出现异常时候程序会自动唤起异常处理的子程序。这样的最大好处就是能把一切之前我们必须手动完成的错误处理程序自动化。

异常处理的基础

C++中的异常处理是基于三个关键字的：try,catch 和 throw。更通俗地来说，我们把需要进行监视的代码块放置在 try 的代码块中，以便监视其异常。如果在 try 的代码块中出现了异常，异常就会被抛出。我们使用 catch 子句来捕获抛出的异常，并对其进行处理。下面我们将对这些进行详细的说明。

我们要监视的以便处理其异常的代码必须放置在 try 代码块中。try 代码块中的代码所抛出的异常会被其后的 catch 子句所捕获。try 和 catch 的常用形式如下：

```
try
{
   // try 的代码块
}
catch(类型 1 参数 arg)
{
    //catch 代码块
}
catch(类型 2 参数 arg)
{
    //catch 代码块
}
//...
catch(类型 N 参数 arg)
{
    //catch 代码块
}
```

try 中的代码块就是我们要监视的，以便处理其异常的代码段。这段代码可以很短，比如可以是一个函数中的几条语句；也可以是整个 main()函数（这样一来就是整个程序都被监视了。）。

当程序抛出异常的时候，对应的 catch 子句就会捕获这个异常，然后处理这个异常。一个 try 子句可以有多个 catch 子句和其相匹配。try 中语句抛出异常的类型决定了那个 catch 子句会被使用，其中的语句会被执行。当捕获一个异常的时候参数 arg 就会接收到异常的值。任何类型的数据都可以被接收，这也包括我们自己创建的类。

throw 语句的通用形式如下：

```
throw exception;
```

该语句就生成一个 execption 指定的异常。如果要捕获这个异常，那么 trow 语句就必须是在一个 try 语句的代码块中，或者是在 try 代码块中调用的函数中。

如果抛出了一个异常而没有对应的 catch 子句，程序就会异常终止。也就是说，程序会以一种我们不能控制的方式而结束。因此，我们应该对抛出的所有异常都进行捕获。

下面的简单程序展示了 C++中的异常处理机制：

view plain

```
1.  //一个简单的异常处理程序
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main()
6.  {
7.      cout << "start\n";
8.
9.      try
10.     {
11.         //try 代码块的开始
12.         cout << "Inside try block\n";
13.         throw 99;
14.         cout << "This will not execute";
15.     }
16.     catch ( int i )//捕获异常
17.     {
18.         cout << "Caught an exception -- value is : ";
19.         cout << i << "\n";
20.     }
21.
22.     cout << "end";
23.
24.     return 0;
25. }
```

上面程序的输出如下：

```
start
Inside try block
Caught an exception -- value is : 99
end
```

仔细研究上面的代码，我们可以看到其中的 try 子句中有三条语句，catch 子句捕获了一个整型数的异常。try 代码块中的三条语句只执行了两条：第一个 cout 语句和 throw 语句。一旦有异常抛出，程序的执行就会转向 catch 子句，try 代码块就结束了。也就是说，catch 子句不是被调用的，而是程序的执行转跳到 catch 子句的。因此 thow 语句后面的 catch 是永远不会被执行的。

一般来说，catch 子句中的语句都是用来对错误进行补救的。如果异常是可以被修正的，程序会继续执行 catch 子句后面的语句。否则，程序应该以一种可控的方式结束。

正如前面我们讲到的那样，抛出异常的类型必须和 catch 子句中异常的类型是相匹配的。例如，在前面的例子中，如果我们把 catch 子句中异常的类型修改为 double，那么这个异常就不会被捕获，程序就会异常终止。如下所示：

view plain

```
1.  //一个简单的异常处理程序
2.  #include <iostream>
3.  using namespace std;
4.
5.  int main()
6.  {
7.      cout << "start\n";
8.
9.      try
10.     {
11.         //try 代码块的开始
12.         cout << "Inside try block\n";
13.         throw 99;
14.         cout << "This will not execute";
15.     }
16.     catch ( double i )//捕获异常
17.     {
18.         cout << "Caught an exception -- value is : ";
19.         cout << i << "\n";
20.     }
21.
22.     cout << "end";
```

```
23.
24.     return 0;
25. }
```

由于程序抛出的整型数异常不会被 catch 子句捕获，程序的输出将如下：

start

Inside try block

This application has requested the Runtime to terminate it in an unusual way.

Please contact the application's support team for more information.

try 代码块中调用的函数抛出的异常也可以由 try 来处理。例如，下面的程序是正确的：

view plain

```
1. #include <iostream>
2. using namespace std;
3.
4.
5. void Xtest(int test)
6. {
7.     cout << "Inside Xtest, test is: " << test << "\n";
8.     if ( test ) throw test;
9. }
10.
11. int main()
12. {
13.     cout << "start\n";
14.
15.     try //监视的代码段
16.     {
17.         cout << "Inside try block\n";
18.         Xtest(0);
19.         Xtest(1);
20.         Xtest(2);
21.     }
22.     catch( int i ) //捕获异常
23.     {
24.         cout << "Caught an exception -- value is: ";
25.         cout << i << "\n";
26.     }
27.
```

```
28.     cout << "end";
29.
30. return 0;
31. }
```

上面程序的输出如下：

start

Inside try block

Inside Xtest, test is: 0

Inside Xtest, test is: 1

Caught an exception -- value is: 1

end

正如程序的输出那样，在函数 Xtest()中抛出的异常在函数 main()中被捕获了。try 子句可以被放置在一个函数中。当每次调用这个函数的时候，函数中用来处理异常的代码就会被重置。如下面的程序那样：

view plain

```
1.  //try 代码块可以局部化到一个函数中
2.  #include <iostream>
3.  using namespace std;
4.
5.  //当每次进入到函数的时候，其中的 try 和 catch 都会被重置
6.  void Xhandler( int test)
7.  {

14.     try
15.     {
16.         if (test) throw test;
17.     }
18.     catch ( int i )
19.     {
20.         cout << "Caught One! Ex.#: " << i << "\n";
21.     }
22. }
23.
24. int main()
25. {
26.     cout << "start\n";
27.
28.     Xhandler(1);
```

```
29.     Xhandler(2);
30.     Xhandler(0);
31.     Xhandler(3);
32.
33.     cout << "end";
34.
35.     return 0;
36. }
```

程序的输出如下：

```
start
Caught One! Ex.#: 1
Caught One! Ex.#: 2
Caught One! Ex.#: 3
end
```

在这个实例程序中，共计抛出了三个异常。在每次抛出异常之后，函数就都退出了。当再次调用这个函数的时候，异常处理就被重置了。try 代码块也是在每次进入的时候会被重置。因此，如果 try 代码块是作为循环的一部分，那么每次循环的时候 try 代码块就会被重置。

练习：

1. 在 C++语言中，什么是异常？
2. 异常处理时基于那三个关键字的？
3. 异常的捕获是基于它们的类型。对吗？

使用多个 catch 语句

正如我们前面讲过的那样，我们可以把多个 catch 语句和一个 try 关联起来。实际上，这样做是很普遍的。然而，每个 catch 子句捕获的异常类型必须是不相同的。例如，下面的程序捕获的就是整型数和字符型指针的异常。

view plain

```
18. //使用多个 catch 语句
19. #include <iostream>
20. using namespace std;
21.
22. //不同类型的异常都可以被捕获
23. void Xhandler( int test )
24. {
25.     try
26.     {
27.         if ( test ) throw test; //抛出一个 int 类型的异常
```

```
28.         else throw "Value is zero"; //抛出一个字符指针异常
29.     }
30.     catch(int i)
31.     {
32.         cout << "Caught One Ex.#: " << i << '\n';
33.     }
34.     catch(char *str)
35.     {
36.         cout << "Caught a string: ";
37.         cout << str << "\n";
38.     }
39. }
40.
41. int main()
42. {
43.     cout << "start\n";
44.
45.     Xhandler(1);
46.     Xhandler(2);
47.     Xhandler(0);
48.     Xhandler(3);
49.
50.     cout << "end";
51.
52.     return 0;
53. }
```

通常来讲，catch 表达式是根据它们在程序中出现的先后位置来检查的。只有异常类型匹配的那一个 catch 子句的代码块才会被执行。其它的 catch 子句都会被忽略。

捕获基类的异常

关于多个 catch 语句有一个和基类相关的重要问题：捕获基类异常的 catch 子句是能够捕获到子类的异常的。因此，当我们需要同时捕获基类和子类的异常的时候，我们需要把捕获子类的 catch 子句放置捕获基类异常的 catch 子句前面。如果不这样做，捕获基类异常的 catch 子句会捕获到子类的异常。例如下面的程序：

view plain

```
14. //捕获子类的异常。这个程序是错误的
15. #include <iostream>
16. using namespace std;
17.
18. class B
```

```
{
};

class D: public B
{
};

int main()
{
    D derived;

    try
    {
        throw derived;
    }
    catch(B b)
    {
        cout << "Caught a base calls.\n";
    }
    catch (D d)
    {
        cout <<"This won't execute.\n";
    }

    return 0;
}
```

这里，由于 derived 是一个基类为 B 的子类对象，所以它将被第一个 catch 子句所捕获，第二个 catch 子句是永远也不会被执行的。有些编译器会对这样的程序产生告警。还有的编译器则会报告编译错误，从而停止编译。总之，要修改这样的错误或者告警，只要调整一下 catch 子句的顺序即可。

捕获所有的异常

在某些情况下，我们需要异常处理程序捕获所有的异常，而不是捕获某种特定类型的异常。我们可以采用下面的形式来实现这样的功能：

catch(...) { //处理所有的异常}

这里，省略号表示所有的异常。下面的程序演示了这种用法：

view plain

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

```
24.
25. void Xhandler(int test)
26. {
27.     try
28.     {
29.         if ( test == 0) throw test; //抛出一个整型数的异常
30.         if ( test == 1) throw 'a';  //抛出一个字符类型的异常
31.         if ( test == 2) throw 123.33;//抛出一个浮点类型的异常
32.     }
33.     catch(...)//捕获所有的异常
34.     {
35.         cout << "Caught one!\n";
36.     }
37. }
38.
39. int main()
40. {
41.     cout << "start\n";
42.
43.     Xhandler(0);
44.     Xhandler(1);
45.     Xhandler(2);
46.
47.     cout << "end";
48.
49.     return 0;
50. }
```

程序的输出如下：

start

Caught one!

Caught one!

Caught one!

end

Xhandler()函数抛出了三种类型的异常：整型的，字符型的和浮点类型的。所有类型的异常都可以通过语句 catch(...)来捕获了。

这种 catch 的最典型的一种用法就是把这个 catch 放置在一个 try 的众多的 catch 子句中的最后一个。这样，就提供了一种缺省的或者是针对所有类型异常的处理方法。这种方式也

是一种处理那些我们不想明确如何处理的异常的好方法。同样这种方式可以处理因为我们没有捕获某个异常而导致程序异常终止的情况。

在函数中明确抛出异常

我们可以指定函数会抛出的异常的类型，也可以限制函数抛出某种异常。只需要在函数定义的后面加上 throw 子句就可以了。这种 throw 子句的基本形式如下：

```
ret-type func-name(arg-list) throw(type-list)
{
    //...
}
```

其中，只有在 throw 后面的由逗号间隔开来的数据类型的异常才可以由该函数爆出。如果函数抛出其他类型的异常则会导致程序异常终止。如果我们不希望函数抛出任何异常，我们可以保持这个列表为空。

注意：在编写本书的时候， Visual C++实际上还不能阻止我们在函数中抛出不在异常列表类型中的异常的做法。这是一种不标准的行为。我们仍然可以指定 throw 子句，但是该子句只是提供了一些函数习惯的信息而已。（译者注：言下之意，不参与是否能进行正确编译的判断）。

下面的程序就展示了如何指定函数抛出各种类型的异常：

view plain

```
//指定函数抛出异常的类型


#include <iostream>
using namespace std;

//下面的函数只能抛出整形，字符型和双精度浮点型的异常
void Xhandler(int test) throw(int,char, double) //throw 子句指明了函数能抛出的异常的类型
{
    if ( test == 0 ) throw test;
    if ( test == 1 ) throw 'a';
    if ( test == 2 ) throw 123.23;
}

int main()
{
    cout << "start\n";
    try
    {
        Xhandler(0); //同样，我们也可以传入 1 和 2
```

```
223.     }
224.     catch (int i)
225.     {
226.     cout << "Caught int\n";
227.     }
228.     catch (char c)
229.     {
230.         cout << "Caught char\n";
231.     }
232.     catch (double  d)
233.     {
234.         cout << "Caught double\n";
235.     }
236.
237.     cout << "end";
238.
239.     return 0;
240. }
```

在上面的这个程序中，Xhandler()函数只能抛出整形，字符型和双精度类型的异常。如果该函数企图抛出其他类型的异常，程序会异常终止。为了掩饰这一点，我们可以把上面函数抛出 int 类型的异常删除掉，然后重试该程序，就会看到程序异常终止。（就面前 Visual C++来说，程序不会异常终止，这点我们在前面有提到过。）

非常重要的一点：我们可以限制函数给调用他的 try 代码块抛出的异常的类型；但是函数中的一个 try 块是可以抛出任意类型的异常的，只要这些异常在同一个函数中被捕获。这点限制仅局限于函数需要把异常抛出到函数本身之外。

再一次抛出异常

我们可以在异常处理程序中再次抛出同一个异常。这种做法将使得当前的异常被传递给更外层的 try/catch 序列块来处理。这种情况最常用于对同一个异常进行多次处理的情况。例如，一个异常处理程序只能处理异常的某一个方面，而别的处理程序可以处理其他的方面。异常只能在 catch 代码块中被重新抛出或者是由在该段代码块中调用的函数重新抛出。当异常被重新抛出的时候，它是不会被同一个 catch 语句所捕获的，而是会被传播到更外层的 catch 语句的。下面的程序就演示了重新抛出异常。其中重新抛出的是一个 char*类型的异常。//演//演示重新抛出一个异常

view plain

```
14. #include <iostream>
15. using namespace std;
16.
17. void Xhandler()
```

```
{
    try
    {
        throw "hello"; //抛出一个字符指针类型的异常
    }
    catch (char* e)
    {
        cout<< "Caught char * inside Xhandler\n";
        throw; //重新抛出该字符指针类型的异常到函数的外部
    }

}

int main()
{
    cout << "start \n";

    try
    {
        Xhandler();
    }
    catch (char* e)
    {
        cout << "Caught char * inside main\n";
    }

    cout << "End";

    return 0;
}
```

上面程序的输出如下；

start

Caught char * inside Xhandler

Caught char * inside main

End

练习：

1. 怎么才能捕获所有的异常？
2. 如何指定可以被抛出到函数体外的异常的类型？
3. 如何将一个异常重新抛出？

专家答疑：

问：

通过上面的学习我们可以看出函数有两种方式可以来报告错误：一个是通过返回错误码，另外一个就是通过抛出异常。请问，通常情况下，我们应该使用哪种方式呢？

答：

你说的是正确的。通常有两种方式来报告错误：一个是通过函数返回错误码，一个是通过抛出异常。现如今，大多数的专家更倾向于使用异常机制而不是返回错误码的方法。例如，在 Java 和 C#中都是依靠异常机制来报告错误的，例如报告打开文件的错误或者是算术运算溢出的错误。由于 C++是从 C 发展而来，他融合了错误码方式和异常机制两者来报告错误，因此，许多 C++库函数还都是通过返回错误码的方式来报告错误的。然而，在新编写的代码中，我们应该考虑使用异常处理机制来报告错误。这也是现代的编程模式

模板

模板是 C++中最高级和最强大的特性之一。模板并不是在最初的关于 C++的说明中就有的，而是后来才被加入到 C++中的。如今所有的 C++编译器都是支持模板特性的。模板能够帮助我们对编程中最难以琢磨的东西进行归档：增加代码的复用性。

通过使用模板，我们可以创建通用的函数和类。在这些通用的函数和类中，其要操作的数据的类型是作为参数被传入的。因此，我们可以针对多种类型的数据只编写一次代码，一个函数或者类；而不用针对不同数据类型编写不同版本的代码。下面我们就将学习通用函数和类。

基本技能 12.2：通用函数

通用的函数定义了针对不同数据类型的一套通用的操作，其操作对象的数据类型是作为一个参数被传入的。通过使用一个通用的函数，一个简单的通用的过程就可以被应用到不同的数据类型上去。或许我们都知道，许多算法在逻辑上都是通用的，而不管算法所操作的数据类型是否相同。例如，快速排序算法就是这样，不管我们是针对一组整形数据或者是针对一组浮点型数据进行排序。区别只在于被排序对象的类型不一样。通过定义通用函数，我们实际上定义的是算法的本质，使得算法独立于数据类型。一旦我们完成了通用函数的定义，编译器进行编译时会根据调用该函数时候的实际数据类型自动生成正确的代码。从本质上来说，一旦我们创建了一个通用的函数，我们就等于创建了能自动对自己进行重载的函数。

通用函数是通过关键字 template 模板来创建的。模板这个词恰如其分的表达了他在 C++中的作用：被用来创建描述要做什么的函数样板，而由编译器在必要的时候补充细节。定义通用函数的一般形式为：

```
template〈class Ttype〉返回值类型 函数名称（参数列表）
{
    函数体
```

}

其中的 Ttype 是数据类型的占位符。这个类型将被用来定义函数操作中声明的数据的类型。编译器在生成特定函数版本的时候会自动地使用实际的数据类型来替换 Ttype。虽然我们在上述形式中使用的是 class 关键字来指定通用函数声明中的通用类型，我们也是可以使用 typename 这个关键字的。

下面的实例中，我们创建了一个通用的用于交换两个变量值的函数。由于交换两个值的过程和对应的变量的类型没有关系，所以把这个过程编写成通过函数是非常明智的选择。

view plain

```
1. //函数模板示例
2. #include <iostream>
3. using namespace std;
4.
5. template<class X> void swapargs(X &a, X&b)
6. {
7.     X temp;
8.     temp = a;
9.     a = b;
10.    b = temp;
11.}
12.
13.int main()
14.{
15.    int i = 10, j = 20;
16.    float x = 10.1f, y = 23.2f;
17.    char a = 'x', b = 'z';
18.
19.    cout <<"Original i,j:" << i <<' ' << j << '\n';
20.    cout <<"Original x,y:" << x <<' ' << y << '\n';
21.    cout <<"Original a,b:" << a <<' ' << b << '\n';
22.
23.    //编译器会自动根据传入参数的类型创建不同
24.    //版本的 swapargs()函数
25.    swapargs(i, j);
26.    swapargs(x, y);
27.    swapargs(a, b);
28.
29.    cout <<"Swapped i,j:" << i <<' ' << j << '\n';
30.    cout <<"Swapped x,y:" << x <<' ' << y << '\n';
31.    cout <<"Swapped a,b:" << a <<' ' << b << '\n';
32.
33.    return 0;
34.}
```

我们将对上面的程序进行仔细的研究。首先是以第一行：

template<class X> void swapargs(X &a, X&b)

这句告诉编译器两件事情：即将创建模板并且是通用函数类型的模板。其中 X 就是作为占位符的通用数据类型，也就是交换的数据的类型。在 main()函数中，我们在调用函数 swapargs()函数的时候分别三次传入了不同的数据类型的变量作为参数：整型，单精度的浮点数以及字符类型。由于 swapargs()函数是通用函数，编译器就自动地针对上述三种类型创建不同版本的函数，分别用来交换整型变量的值，单精度浮点型变量的值以及字符类型变量的值。这样一来同一个通用的交换函数就可以被用来针对不同数据类型的数据进行交换操作。

这里涉及到了和模板相关的几个重要的术语。首先，通用函数又叫做模板函数。本书中将交替使用这两个相同含义的术语。当编译器创建了通用函数的某一个特定的版本函数的时候，我们就说通用函数被特定化了，也被叫做是由模板函数生成特定的函数。这种生成的过程被称为实例化。换句话说，由模板函数生成的函数实际上是模板函数的实例化。

有两个通用数据类型的函数

在使用模板函数的时候我们还可以指定两个或者更多的通用数据类型。他们之间用逗号间隔。例如，下面的程序就创建了有两个通用数据类型的函数：

view plain

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

//模板函数需要两个通用的数据类型
template <class Type1, class Type2>
void myFunc(Type1 x, Type2 y)
{
    cout << x << ' ' << y << "\n";
}

int main()
{
    myFunc(10, "hi");
    myFunc(0.23, 10L);

    return 0;
}
```

在上面的程序中，有两个数据类型占位符 Type1 和 Type2。在进行编译的时候需要生成模板函数 myFunc()的特定实例，编译器分别用整形、字符指针和双精度浮点型、长整形替换模板中的数据类型占位符。

显式重载通用函数

我们在前面提到过，模板函数会在需要的时候自动地被重载。这里需要指出的是我们也是可以显式地对其进行重载。这个过程通常被称为显式地实例化。如果我们对一个通用函数进行了重载，那么重载的函数将覆盖模板函数对应的数据类型的版本。例如下面的程序。我们重写编写了前面的用于交换两个参数值得示例程序：

view plain

```
1. //显式地生成模板函数的实例
2. #include <iostream>
3. using namespace std;
4.
5. template <class X>
6.     void swapargs(X &a, X &b)
7. {
8.     X temp;
9.     temp =a;
10.    a = b;
11.    cout << "Inside template swapargs.\n";
12.
13.}
14.
15.//下面的这个函数就覆盖了模板函数针对整形类型的实例
16.void swapargs(int &a,int &b)
17.{
18.    int temp;
19.    temp = a;
20.    a = b;
21.    b = temp;
22.    cout << "Inside swapargs int specialization./n";
23.}
24.
25.int main()
26.{
27.    int i = 10, j = 20;
28.    float x = 10.1f, y = 23.3f;
29.    char a = 'x', b = 'z';
30.
31.    cout << "Original i, j:" << i << ' ' << j << '\n';
32.    cout << "Original x, y:" << x << ' ' << y << '\n';
33.    cout << "Original a, b:" << a << ' ' << b << '\n';
34.
35.    swapargs(i,j); //调用的是显式地指定了参数类型的函数
36.    swapargs(x,y); //调用模板函数
```

```
37.     swapargs(a,b); //调用模板函数
38.
39.     cout << "Swapped i, j:" << i << ' ' << j << '\n';
40.     cout << "Swapped x, y:" << x << ' ' << y << '\n';
41.     cout << "Swapped a, b:" << a << ' ' << b << '\n';
42.
43.     return 0;
44. }
```

上面程序的输出结果如下：

```
Original i, j:10 20
Original x, y:10.1 23.3
Original a, b:x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j:20 10
Swapped x, y:23.3 23.3
Swapped a, b:z z
```

正如程序中注释描述的那样，swapargs(i, j);调用的是显式地指定了参数类型为 int 的这个函数。这个函数覆盖了模板函数 swapargs()模板函数针对 int 类型的实例化函数。

后来，人们引入了另外一种语法来描述这种显示实例化的函数。这种新的语法中使用 到 template 关键字。例如，上述程序中针对 int 类型的显示实例化就可用下面的程序完成：

view plain

```
1. template<> void swapargs<int>(int &a,int &b)
2. {
3.     int temp;
4.     temp = a;
5.     a = b;
6.     b = temp;
7.     cout << "Inside swapargs int specialization.\n";
8. }
```

从中我们可以看出，这种新的语法中使用 template<>来描述要对模板函数进行实例化。具体实例化的数据类型被放在了函数名称后面的尖括号中。这种形式可用来针对任何模板函数的任何类型进行实例化。两种描述方式没有孰优孰劣之分。长期来看，后一种方式也许是一种更好的方式。

显示地对模板函数进行实例化似的我们可以针对特殊的情况进行处理。一般来说，如果针对不同的数据类型我们需要使用不同的函数实现，此时最好是使用函数重载还不是模板函数。

基本技能 3：通用的类

除了上面的通用函数以外，我们还可以创建通用的类。创建通用类的实质是定义了这个通用类使用到的算法。算法中实际使用到的数据的类型是在对该类进行实例化的时候通过参数的方式传入的。

当一个类提供的逻辑可以通用化时，使用通用类此时会非常显得非常有用。例如，用来处理元素类型为整形数的队列的算法同样适用于元素类型为字符的队列；用来处理元素为电子邮件地址的链表的算法也同样适用于元素为汽车配件信息的链表。一旦定义了一个通用的类，该类就能处理任何类型的数据。例如上面提到的队列和链表。在我们对该通用类进行实例化的时候，编译器会自动地根据我们传入的数据类型来生成正确的数据对象。

通用的声明通用类的方式如下：

```
template<class Ttype>class 通用类名称
{
  //类的实现
}
```

其中的 Ttype 就是类型占位符。当我们使用通用类来生成对象的时候就需要指定该类型。我们还可以为通用类定义多个类型占位符。只要把他们写在尖括号中，用逗号隔开即可。

一旦创建了一个通用的类，我们就可以使用如下的方式来生成特定的对象：

通用类名称<类型>对象名;

其中的类型就是这个类所要操作的数据的类型。类的成员函数由编译器自动生成，我们不必显示地指定这些成员函数。

下面就是一个简单的通用类的示例程序：

view plain

```
1.  //一个简单的通用类
2.  #include<iostream>
3.  using namespace std;
4.
5.  //声明一个通用的类，其中的 T 是通用的数据类型
6.  template<class T> class MyClass
7.  {
8.     T x,y;
9.  public:
10.    MyClass(T a, T b)
11.    {
12.        x = a;
```

```
13.         y = b;
14.     }
15.
16.     T div()
17.     {
18.         return x/y;
19.     }
20. };
21.
22. int main()
23. {
24.     //创建 double 类型的 MyClass 的对象
25.     MyClass<double> d_ob(10.0,3.0); //创建通用类的一个特定的实例
26.     cout <<"double division: " << d_ob.div() <<"\n";
27.
28.     //创建 int 类型的 MyClass 的对象
29.     MyClass<int> i_ob(10,3);
30.     cout <<"integer division: " << i_ob.div() <<"\n";
31.
32.     return 0;
33. };
```

上面程序的输出如下：

double division: 3.33333

integer division: 3

正如程序输出的那样，针对 double 类型的对象进行的是浮点数的除法运算；而针对整型数的对象则进行的是整型数的除法。

当我们声明通用类 MyClass 的对象的时候，编译器自动地针对传入的数据类型生成了 div()函数，x,y 的类型也由编译器自动根据传入参数的类型来进行声明。在上面的这个示例程序中，我们声明了两个不同类型的对象。第一个，d_ob 操作的是 double 类型的数据，这就意味着 x，y 是 double 类型的，并且除法运算的结果也是 double 类型的。第二个，i_ob 操作的是 int 类型的，因此 x，y 都是 int 类型的，除法运算的结果也是 int 类型的。注意其中的声明方式：

MyClass<double> d_ob(10.0,3.0);

MyClass<int> i_ob(10,3);

所期望的数据类是被放置在通用类名称后面的尖括号中。通过在声明对象的时候传入不同的类型，我们就可以修改 MyClass 类的操作对象的类型。

通用的类也可以有多个通用的数据类型。只要在模板声明时把这多个类型放置在尖括号中，并用逗号隔开即可。例如，下面的示例程序中的通用类就需要两个通用数据类型：

view plain

```
12. //这个示例程序中用到了两个通用数据类型
13. #include<iostream>
14. using namespace std;
15.
16. template<class T1, class T2> class MyClass
17. {
18.     T1 i;
19.     T2 j;
20. public:
21.     MyClass(T1 a, T2 b)
22.     {
23.         i = a;
24.         j = b;
25.     }
26.     void show()
27.     {
28.         cout << i <<' ' << j << "\n";
29.     }
30. };
31.
32. int main()
33. {
34.     MyClass<int,double> ob1(10,0.23);
35.     MyClass<char,char *> ob2('X',"This is a test");
36.
37.     ob1.show();//输出整型数，双精度浮点数的值
38.     ob2.show();//输出字符，字符串的值
39.
40. };
```

上面程序的输出结果如下：

10 0.23

X This is a test

上面的程序中声明了两种类型的对象：ob1 对象使用的是整型和双精度浮点数类型；ob2 对象使用的是字符和字符指针类型。针对这两种情况，编译器自动地生成正确的数据和函数来适应这两个对象。

显示地生成特定的类

和前面学习过的通用函数一样，我们也可以显示地指定通用类的特定实例。我们可以通过类似于显示生成通用函数的特定实例那样使用 template<>的形式来创建通用类的特定实例。例如：

view plain

```cpp
//显示指定通用类的实例
#include<iostream>
using namespace std;

template<class T> class MyClass
{
    T x;
public:
    MyClass(T a)
    {
        cout <<"Inside generic MyClass.\n";
        x = a;
    }

    T getX()
    {
        return x;
    }
};

//显示地生成通用类的实例
template<>class MyClass<int> //该类就是通用类 MyClass 的一个显示实例
{
    int x;
public:
    MyClass(int a)
    {
        cout <<"Inside MyClass<int> specialization.\n";
        x = a * a;
    }
    int getX()
    {
        return x;
    }
};

int main()
{
    MyClass<double> d(10.1);
    cout <<"double: " << d.getX() <<"\n";

    MyClass<int> i(5); //这里用到的是显示实例
    cout <<"int: " << i.getX() << "\n";

```

```
60.     return 0;
61.
62. }
```

上面程序的输出如下：

Inside generic MyClass.

double: 10.1

Inside MyClass<int> specialization.

int: 25

注意上面程序中显示指定通用类的实例的方式：

```
template<>class MyClass<int>
{
};
```

上面的写法告诉编译器将显示地生成针对 int 类型的 MyClass 通用类的实例。这种形式可用于针对任何通用类进行显示实例化。

显示地对通用类进行实例化使得我们可以针对个别的特殊的情况进行特殊处理，而其他情况下都是由编译器根据通用类自动生成相应代码。这无疑是对通用类的一种扩展。当然，如果我们发现我们需要处理的特殊情况很多，那此时我们就应该考虑使用通用类是否合适了。

练习：

1. 声明通用函数或者通用类的时候使用哪个关键字？
2. 通用函数是否可以被显示地重载？
3. 在使用通用类的时候，他的所有成员函数是否都会被自动地生成呢？

项目 12-1：创建通用的队列类

在项目 8-2 中，我们创建了一个元素类型为字符的队列。在该项目中我们把队列修改为通用的队列，其中的元素可以是任意类型的。把队列修改成通用的类是一个很好的主意，因为队列操作的逻辑与队列中元素的类型是相互对立的。不管队列中的元素是整型数，浮点数或者是自定义类的对象，队列的操作逻辑都是一样的。一旦我们定义了这样通用的队列类，我们就可以在任何需要的地方使用它（言下之意就是和数据类型无关）。

步骤:

1. 新建文件 GenericQ.cpp，把项目 8-2 中队列类的代码复制到其中。
2. 把队列的声明修改为模板，如下：

   ```
   template<classQType>classQueue
   ```

   这里，通用的数据类型占位符为 QType

3. 修改 q 的类型为 QType，如下：

QTypeq[maxQsize]; //用一个数组来存储队列中的元素

由于 q 现在是通用类型的数组，因此他可以用来存放声明队列时候的任意类型的数据

4. 把 put()函数参数的类型修改为 QType，如下：

view plain

```
void put(QTypedata)
{
    if (putloc ==size)
    {
        cout <<" --Queue is full.\n";
        return;
    }
    putloc++;
    q[putloc] =data;
}
```

5. 把 get()函数返回值的类型修改为 Qtype，如下：

view plain

```
QType get()
{
     if(getloc ==putloc)
     {
         cout <<" --Queue is empty.\n";
         return 0;
     }

     getloc++;
     return q[getloc];
}
```

6. 完整的程序如下：

view plain

```
/*
    项目-1
    通用的队列类
*/

#include<iostream>
using namespace std;

```

```
71. const int maxQsize = 100;
72. 
73. //创建通用的队列类
74. template<classQType>class Queue
75. {
76.     QType q[maxQsize];//用一个数组来存储队列中的元素
77.     int size;//队列所能存储的最大元素数量
78.     int putloc,getloc;//put，get 操作的对应的当前位置
79. public:
80.     //构建一个特定长度的队列
81.     Queue(int len)
82.     {
83.         //队列的长度必须小于最大值，并且为正数
84.         if (len>maxQsize)
85.         {
86.             len =maxQsize;
87.         }
88.         else if(len <= 0)
89.         {
90.             len = 1;
91.         }
92. 
93.         size =len;
94.         putloc =getloc = 0;
95.     }
96. 
97.     //数据进队列
98.     void put(QType data)
99.     {
100.         if (putloc ==size)
101.         {
102.             cout <<" --Queue is full.\n";
103.             return;
104.         }
105.         putloc++;
106.         q[putloc] =data;
107.     }
108. 
109.     //元素出队列
110.     QType get()
111.     {
112.         if(getloc ==putloc)
113.         {
114.             cout <<" --Queue is empty.\";
```

```
115.             return 0;
116.         }
117.
118.         getloc++;
119.         return q[getloc];
120.     }
121. };
122.
123. //演示通用队列类的使用
124. int main()
125. {
126.     Queue<int> iQa(10),iQb(10);//创建两用用于存放整型数的队列
127.     iQa.put(1);
128.     iQa.put(2);
129.     iQa.put(3);
130.
131.     iQb.put(10);
132.     iQb.put(20);
133.     iQb.put(30);
134.
135.     cout <<"Contents of integer queue iQa: ";
136.     for (int i = 0; i < 3; i++)
137.     {
138.         cout <<iQa.get() <<" ";
139.     }
140.     cout <<endl;
141.
142.     cout <<"Contents of integer queue iQb: ";
143.     for (int i = 0; i < 3; i++)
144.     {
145.         cout <<iQb.get () <<" ";
146.     }
147.     cout <<endl;
148.
149.     //创建两个元素类型为 double 的队列
150.     Queue<double> dQa(10),dQb(10);
151.     dQa.put(1.01);
152.     dQa.put(2.02);
153.     dQa.put(3.03);
154.
155.     dQb.put(10.01);
156.     dQb.put(20.02);
157.     dQb.put(30.03);
158.
```

```
159.    cout <<"Contents of double queue dQa: ";
160.    for (int i = 0; i < 3; i++)
161.    {
162.        cout <<dQa.get() <<" ";
163.    }
164.    cout <<endl;
165.
166.    cout <<"Contents of double queue dQb: ";
167.    for (int i = 0; i < 3; i++)
168.    {
169.        cout <<dQb.get() <<" ";
170.    }
171.    cout <<endl;
172.
173.    return 0;
174.
175. }
```

上面程序的输出如下：

Contents of integer queue iQa: 1 2 3

Contents of integer queue iQb: 10 20 30

Contents of double queue dQa: 1.01 2.02 3.03

Contents of double queue dQb: 10.01 20.02 30.03

7. 从上面的程序中我们可以看出，通用函数和通用类都是非常强大的工具。通用函数和通用类使得我们能够定义适用于任意数据类型的通用的对象，使得程序员从针对每种数据类型都需要单独实现的繁杂工作中解脱出来；而由编译器自动地为我们创建特定版本的代码，进而最大发挥代码的效能。

基本技能 12.4：动态内存分配

C++程序中有两种主要的方式用于在内存中存储数据。第一种就是通过使用变量。变量的存储空间是在编译时就分配好的，在程序运行时是不可改变的。第二种方式就是通过 C+的动态内存分配机制来进行内存分配。在这种方式中，数据存储空间是在程序需要的时候由系统从位于程序代码区和栈区之间的空闲区域中进行分配。这段区域被称为堆。图 12-1 从概念上展示了 C++程序在内存中的存储方式。

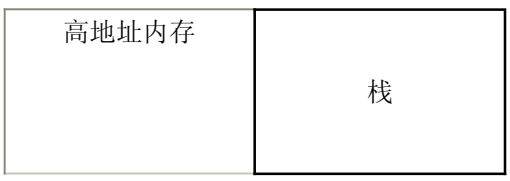


图 12-1 C++程序中存储空间使用示意图

动态内存分配是在程序运行时进行的。因此动态内存分配使得我们可以在程序运行的时候根据需要创建变量。根据需要，我们创建或多或少的变量。动态内存分配常常被用来支持诸如线性链表，二叉树或者小型数组。当然，我们可以根据需要自由地使用动态内存分配机制。动态内存分配机制几乎是现实的程序中的一个很重要的部分。

动态内存分配是从堆空间中进行的。我们都知道，在极端的情况下内存是有可能被耗尽的。因此，经过动态内存分配机制很灵活，但是堆取的空间也是有限的。

C++中提供了两个和动态内存分配相关的运算符：new 和 delete。new、运算符用于分配内存空间并返回指向该空间的首地址。delete 运算符用于释放先前有 new 分配的空间。new 和 delete 的常用形式如下：

p_var = new 类型;

delete p_var;

其中，p_var 是一个指针。指向一块足够大的可用于存储 type 类型数据的内存空间。

由于堆空间是有限，因此堆空间也会是会被耗尽的。如果没有足够的可用内存用于 new 来进行分配，那么 new 会分配失败，并抛出 bad_alloc_exception 异常。这种异常在头文件<new>中有定义。有程序来负责处理该异常并进行正确的处理。如果程序中没有处理这个异常，程序会异常终止。

标准的 C++有对 new 分配空间失败进行相应的描述。麻烦的问题是：一些老的编译器会以不同的方式实现 new 操作。当 C++刚被发明出来的时候，如果分配内存空间失败，new 会返回空指针。后来，人们对此进行了修改，才抛出异常。如果您使用的是老的编译器，请查阅编译器文档，看看其中是如何实现 new 操作的。

既然标准的 C++中描述到 new 操作失败会抛出异常，我们就以这种方式来编写相应的代码。如果您所使用的编译器使用的是不同的 new 操作的实现方式，你可能需要修改下面的程序。

下面的程序使用动态内存分配机制来分配空间用于存储一个整型数：

view plain

```
1. //演示 new 和 delete 运算符
2.
3. #include<iostream>
4. #include<new>
5.
6. using namespace std;
7.
8. int main()
9. {
10.    int *p;
11.    try
12.    {
13.        p =new int; //为整型数分配内存
14.    }
15.    catch (bad_alloc xa)
16.    {
17.        cout <<"Allocation Failure/n";
18.        return 1;
19.    }
20.
21.    *p = 100;
22.    cout <<"At " <<p <<" ";
23.    cout <<"is the value " << *p <<"/n";
24.
25.    delete p; //释放分配的内存
26.
27.    return 0;
28. }
```

在上面的程序中，我们给 p 赋值为一块足够大可用于存储整型数的堆空间中的一个地址。然后给这块内存赋值为 100，随后在屏幕上输出该内存空间的值，最后释放掉我们动态分配的内存空间。

delete 运算中的指针必须是先前使用 new 分配的有效的指针。在 delete 操作的时候如果使用的是其他类型的指针，则会导致严重的问题，例如系统崩溃，因为这种行为是没有定义的（就是说没有明确针对这种情况应该如何处理）。

初始化分配的内存

我们可以在分配内存的时候给内存初始化为已知的值。这是通过在 new 语句中的类型后面的括号中写上初始化数值来完成:

p_var = new 变量类型(初始化值);

当然，初始化值的类型必须和被分配的内存空间数据的类型相兼容。

下面的程序中，动态分配的整型数的初始值为 87：

view plain

```
1. //内存初始化
2.
3. #include<iostream>
4. #include<new>
5.
6. using namespace std;
7.
8. int main()
9. {
10.    int *p;
11.
12.    try
13.    {
14.        p =new int(87); //初始化值为
15.    }
16.    catch (bad_alloc xa)
17.    {
18.        cout <<"Allocation Failure/n"; //分配内存空间失败
19.    }
20.
21.    cout <<"At " <<p <<" ";
22.    cout <<"is the value " << *p <<"/n";
23.
24.    delete p;
25.
26.    return 0;
27. }
```

动态分配数组

我们可以使用下面的形式来动态为数组分配空间：

p_var = new array_type[size];

其中，size 指定的是数组中元素的数量。释放数组的空间时，使用下面的形式：

delete []p_var;

其中的[ ]意思为需要释放的是数组空间。例如下面的程序就为 10 个整型数分配数组空间：

view plain

```
1.  //动态为数组分配空间
2.
3.  #include<iostream>
4.  #include<new>
5.
6.  using namespace std;
7.
8.  int main()
9.  {
10.     int *p,i;
11.     try
12.     {
13.         p =new int[10]; //分配个整型数的数组空间
14.     }
15.     catch(bad_alloc xa)
16.     {
17.         cout <<"Allocation Failure/n";
18.         return 1;
19.     }
20.
21.     for(i = 0;i < 10;i++)
22.     {
23.         p[i] =i;
24.     }
25.
26.     for(i = 0;i < 10;i++)
27.     {
28.        cout <<p[i] <<" ";
29.     }
30.
31.     delete []p;
32.
33.     return 0;
34. }
```

请注意其中的 delete 语句。正如前面提到的那样，当我们使用 delete 释放 new 所分配的数组空间的时候，必须使用[ ]来表明是释放数组空间。（正如我们在下一章节中看到的那样，当我们动态分配对象的数组空间的时候，这一点显得尤为重要。）

关于动态分配数组空间的一个限制：可以不指定初始化的值。也就是说，在分配数组空间的时候可以不对其进行初始化。

专家答疑

问：我曾经看到过使用 malloc()和 free()进行动态内存分配的 C++程序，这两个函数是用来做什么的？

答：C 语言是不支持使用 new 和 delete 运算符的。相反，C 语言中使用 malloc 和 free 来进行动态内存分配。malloc()函数用来分配内存空间；free()用来释放空间。C++也是支持这两个函数的，所以我们在 C++代码中偶尔也能看到使用这两函数来分配内存空间。如果代码是从原始的 C 代码中升级过来的，这种情况就很可能发生。然而新写的代码中我们应该使用 new 和 delete 来进行内存空间的分配和释放。这是因为 new 和 delete 不仅提供了更为方便的内存分配和释放的方式，而且还因为使用这两个运算符能避免使用 malloc 和 free 时常出现的几种错误。还有一点：尽管 C++中没有明确规定不能在程序中把这两对函数进行混合使用，但是我们最好还是不要这样做。这样可以保持程序内存的分配和释放相互兼容。

动态分配对象

我们还可以使用 new 运算符来动态地分配对象。此时，new 运算会创建一个对象并返回指向该对象的指针。动态创建的对象和普通的对象是一样的：当我们创建该对象的时候会调用对象的构造函数；当我们释放该对象的时候，会调用其析构函数。

下面的程序中创建了一个 Rectangle 类，其中封装了长方形的宽度和高度。在 main()函数中，我们动态地创建了一个 Rectangle 类的对象。这个对象在程序结束的时候被释放。

view plain

```
1.  //动态分配对象
2.
3.  #include<iostream>
4.  #include<new>
5.
6.  using namespace std;
7.  class Rectangle
8.  {
9.      int width;
10.     int height;
11. public:
12.     Rectangle(int w,int h)
13.     {
14.         width =w;
15.         height =h;
```

```
16.         cout <<"Constructing " <<width <<" by "<<height <<" rectangle./
    n";
17.     }
18.
19.     ~Rectangle()
20.     {
21.         cout <<"Destructing " <<width <<" by " <<height <<" rectangle./
    n";
22.     }
23.
24.     int area()
25.     {
26.         return width*height;
27.     }
28. };
29.
30. int main()
31. {
32.     Rectangle *p;
33.     try
34.     {
35.         p =new Rectangle(10,8); //动态生成 Rectangle 对象，这里会调用
    Rectangle 的构造函数
36.     }
37.     catch (bad_alloc xa)
38.     {
39.         cout <<"Allocation Failure/n";
40.         return 1;
41.     }
42.
43.     cout <<"Area is " <<p->area();
44.     cout <<"/n";
45.     delete p;
46.     return 0;
47.
48. }
```

上面程序的输入如下：

Constructing 10 by 8 rectangle.

Area is 80

Destructing 10 by 8 rectangle.

请注意，在上面的程序中，对象的构造函数的参数是直接写在对象类型的后面的，这点和普通的初始化是一致的。另外，由于 p 是一个指向对象的指针，所以当调用 area()函数的时候要使用->箭头运算符而不是.点号运算符。

我们还可以动态分配对象的数组。但是此时有一点需要注意：由于 new 生成的数组不能对其中的元素进行显示初始化，所以对象的类中必须有不需要参数的构造函数。如果没有定义不需要参数的构造函数，C++编译器就会在分配对象数组的时候因为找不到匹配的构造函数而报告错误。

下面我们重写之前的程序，我们为 Rectangle 类增加了不需要参数的构造函数，这样在动态生成对象数组的时候这些对象可以被正确的初始化。我们还增加了 set()函数，用来设置长方形的尺寸。

view plain

```
1. //分配对象数组
2.
3. #include<iostream>
4. #include<new>
5.
6. usingnamespacestd;
7.
8. class Rectangle
9. {
10.    int width;
11.    int height;
12.public:
13.    Rectangle() //不需要参数的构造函数
14.    {
15.        width =height = 0;
16.        cout <<"Constructing " <<width <<" by " <<height <<" rectangle.
   /n";
17.    }
18.
19.    Rectangle(int w,int h)
20.    {
21.        width =w;
22.        height =h;
23.        cout <<"Constructing " <<width <<" by "<<height <<" rectangle./
   n";
24.    }
25.
26.    ~Rectangle()
27.    {
```

```
28.         cout <<"Destructing " <<width <<" by " <<height <<" rectangle./
    n";
29.     }
30.
31.     void set(int w,int h)
32.     {
33.         width =w;
34.         height =h;
35.     }
36.
37.     int area()
38.     {
39.         return width*height;
40.     }
41. };
42.
43. int main()
44. {
45.     Rectangle *p;
46.     try
47.     {
48.         p =new Rectangle[3]; //动态生成 Rectangle 对象，这里会调用 Rectangle
    的构造函数
49.     }
50.     catch (bad_alloc xa)
51.     {
52.         cout <<"Allocation Failure/n";
53.         return 1;
54.     }
55.
56.     p[0].set(3, 4);
57.     p[1].set(10, 8);
58.     p[2].set(5, 6);
59.
60.     for(inti = 0;i < 3; ++i)
61.     {
62.         cout <<"Area is " <<p[i].area() << endl;
63.     }
64.
65.     cout <<"/n";
66.
67.     delete []p; //此时会调用数组中每个对象的析构函数
68.
69.     return 0;
```

```
70.
71. }
```

上面程序的输出如下:

Constructing 0 by 0 rectangle.

Constructing 0 by 0 rectangle.

Constructing 0 by 0 rectangle.

Area is 12

Area is 80

Area is 30

Destructing 5 by 6 rectangle.

Destructing 10 by 8 rectangle.

Destructing 3 by 4 rectangle.

由于指针 p 是通过 delete []来释放的，数组中的每一个对象的析构函数都会被调用，正如程序的输出那样。还有，注意上面的 p 可以当做数组，通过索引来访问其中的每一个对象，然后使用.点号运算符来访问对象的成员。

练习

1. 那个运算符可用于分配内存空间？那个运算符可用于释放内存空间？
2. 如果分配满足指定要求的空间失败会怎么样？
3. 当内存在被分配时是否可以对其进行初始化？

基本技能 12.5：命名空间

我们曾经在第一章中对命名空间进行简单的介绍。这里我们将对命名空间进行深入的讨论。使用命名空间的目的是对标识符的名称进行本地化，以避免命名冲突。在 C++中，变量、函数和类都是大量存在的。如果没有命名空间，这些变量、函数、类的名称将都存在于全局命名空间中，会导致很多冲突。比如，如果我们在自己的程序中定义了一个函数 toupper()，这将重写标准库中的 toupper()函数，这是因为这两个函数都是位于全局命名空间中的。命名冲突还会发生在一个程序中使用两个或者更多的第三方库的情况中。此时，很有可能，其中一个库中的名称和另外一个库中的名称是相同的，这样就冲突了。这种情况会经常发生在类的名称上。比如，我们在自己的程序中定义了一个 Stack 类，而我们程序中使用的某个库中也可能定义了一个同名的类，此时名称就冲突了。

Namespace 关键字的出现就是针对这种问题的。由于这种机制对于声明于其中的名称都进行了本地化，就使得相同的名称可以在不同的上下文中使用，而不会引起名称的冲突。或许命名空

间最大的受益者就是 C++中的标准库了。在命名空间出现之前，整个 C++库都是定义在全局命名空间中的（这当然也是唯一的命名空间）。引入命名空间后，C++库就被定义到自己的名称空间中了，称之为 std。这样就减少了名称冲突的可能性。我们也可以在自己的程序中创建自己的命名空间，这样可以对我们认为可能导致冲突的名称进行本地化。这点在我们创建类或者是函数库的时候是特别重要的。

命名空间基础

namespace 关键字使得我们可以通过创建作用范围来对全局命名空间进行分隔。本质上来讲，一个命名空间就定义了一个范围。定义命名空间的基本形式如下：

namespace 名称{//声明}

在命名空间中定义的任何东西都局限于该命名空间内。

下面就是一个命名空间的例子，其中对一个实现简单递减计数器的类进行了本地化。在该命名空间中定义了计数器类用来实现计数；其中的 upperbound 和 lowerbound 用来表示计数器的上界和下界。

view plain

```
1.  //演示命名空间
2.  namespace CounterNameSpace
3.  {
4.      int upperbound;
5.      int lowerbound;
6.
7.      class counter
8.      {
9.          int count;
10.     public:
11.         counter(int n)
12.         {
13.             if (n <=upperbound )
14.             {
15.                 count =n;
16.             }
17.             else
18.             {
19.                 count =upperbound;
20.             }
21.         }
22.
23.         void reset(int n)
24.         {
25.             if (n <upperbound )
```

```
26.             {
27.                 count =n;
28.             }
29.         }
30.
31.         int run()
32.         {
33.             if (count >lowerbound)
34.             {
35.                 return count--;
36.             }
37.             else
38.             {
39.                 return lowerbound;
40.             }
41.         }
42.     };
43. }
```

其中的 upperbound，lowerbound 和类 counter 都是有命名空间 CounterNameSpace 定义范围的组成部分。

在命名空间中声明的标识符是可以被直接引用的，不需要任何的命名空间的修饰符。例如，在 CounterNameSapce 命名空间中，run()函数中就可以直接在语句中引用 lowerbound：

view plain

```
1.  if (count >lowerbound)
2.     {
3.         return count--;
4.     }
```

然而，既然命名空间定义了一个范围，那么我们在命名空间之外就需要使用范围解析运算符来引用命名空间中的对象。例如，在命名空间 CounterNameSpace 定义的范围之外给 upperbound 赋值为 10，就必须这样写：

CounterNameSpace::upperbound = 10;

或者在 CounterNameSpace 定义的范围之外想要声明一个 counter 类的对象就必须这样写：

CounterNameSpace::counter obj;

一般来讲，在命名空间之外想要访问命名空间内部的成员需要在成员前面加上命名空间和范围解析运算符。

下面的程序演示了如何使用 CounterNameSpace 这个命名空间：

view plain

```
1.  //演示命名空间
2.  #include<iostream>
3.  using namespace std;
4.  namespace CounterNameSpace
5.  {
6.    int upperbound;
7.    int lowerbound;
8.
9.    class counter
10.   {
11.     int count;
12.   public:
13.     counter(int n)
14.     {
15.       if (n <=upperbound )
16.       {
17.         count =n;
18.       }
19.       else
20.       {
21.         count =upperbound;
22.       }
23.     }
24.
25.     void reset(int n)
26.     {
27.       if (n <upperbound )
28.       {
29.         count =n;
30.       }
31.     }
32.
33.     int run()
34.     {
35.       if (count >lowerbound)
36.       {
37.         returncount--;
38.       }
39.       else
40.         return lowerbound;
41.     }
42.   };
43. }
```

```
44.
45.  int main()
46.  {
47.     CounterNameSpace::upperbound = 100;
48.     CounterNameSpace::lowerbound = 0;
49.
50.     CounterNameSpace::counter ob1(10);
51.
52.     inti;
53.     do
54.     {
55.        i =ob1.run();
56.        cout <<i <<" ";
57.
58.     }while (i > CounterNameSpace::lowerbound);
59.     cout <<endl;
60.
61.     CounterNameSpace::counterob2(20);
62.     do
63.     {
64.        i =ob2.run();
65.        cout <<i <<" ";
66.
67.     }while (i > CounterNameSpace::lowerbound);
68.     cout <<endl;
69.
70.     ob2.reset(100);
71.     do
72.     {
73.        i =ob2.run();
74.        cout <<i <<" ";
75.
76.     }while (i > CounterNameSpace::lowerbound);
77.     cout <<endl;
78.
79.     return 0;
80.  }
```

请注意：counter 类以及 upperbound 和 lowerbound 的引用都是在前面加上了 CounterNameSpace 修饰符。但是，一旦声明了 counter 类型的对象，就没有必须在对该对象的任何成员使用这种修饰符了。因此 ob1.run()是可以被直接调用的。其中的命名空间是可以被解析的。

相同的空间名称是可以被多次声明的,这种声明向相互补充的。这就使得命名空间可以被分割到几个文件中甚至是同一个文件的不同地方中。例如：

```
namespace NS
{
    int i;
}

//...

namespace NS
{
  int j;
}
```

其中命名空间 NS 被分割成两部分，但是两部分的内容却是位于同一命名空间中的。也就是 NS。最后一点：命名空间是可以嵌套的。也就是说可以在一个命名空间内部声明另外的命名空间。

using 关键字

如果在程序中需要多次引用某个命名空间的成员，那么按照之前的说法，我们每次都要使用范围解析符来指定该命名空间，这是一件很麻烦的事情。为了解决这个问题，人们引入了 using 关键字。using 语句通常有两种使用方式：

using namespace 命名空间名称;

using 命名空间名称::成员；

第一种形式中的命名空间名称就是我们要访问的命名空间。该命名空间中的所有成员都会被引入到当前范围中。也就是说，他们都变成当前命名空间的一部分了，使用的时候不再需要使用范围限定符了。第二种形式只是让指定的命名空间中的指定成员在当前范围中变为可见。我们用前面的 CounterNameSpace 来举例，下面的 using 语句和赋值语句都是有效的：

```
using CounterNameSpace::lowerbound; //只有 lowerbound 当前是可见的
lowerbound = 10; //这样写是合法的，因为 lowerbound 成员当前是可见的
using CounterNameSpace; //所有 CounterNameSpace 空间的成员当前都是可见的
upperbound = 100; //这样写是合法的，因为所有的 CounterNameSpace 成员目前都是可见的
```

下面是我们对之前的程序进行修改的结果：

view plain

```
1.  //使用 using
2.  #include<iostream>
3.
```

```
4.  using namespace std;
5.
6.  namespace CounterNameSpace
7.  {
8.    int upperbound;
9.    int lowerbound;
10.   class counter
11.   {
12.     int count;
13.   public:
14.     counter(int n)
15.     {
16.       if ( n < upperbound)
17.       {
18.         count = n;
19.       }
20.       else
21.       {
22.         count = upperbound;
23.       }
24.     }
25.
26.     void reset(int n )
27.     {
28.       if ( n <= upperbound )
29.       {
30.         count = n;
31.       }
32.     }
33.
34.     int run()
35.     {
36.       if ( count > lowerbound )
37.       {
38.         return count--;
39.       }
40.       else
41.       {
42.         return lowerbound;
43.       }
44.     }
45.   };
46. }
47.
```

```
48.
49.  int main()
50.  {
51.    //这里只是用 CounterNameSpace 中的 upperbound
52.    using CounterNameSpace::upperbound;
53.
54.    //此时对 upperbound 的访问就不需要使用范围限定符了
55.    upperbound = 100;
56.    //但是使用 lowerbound 的时候，还是需要使用范围限定符的
57.    CounterNameSpace::lowerbound = 0;
58.    CounterNameSpace::counter ob1(10);
59.    int i;
60.
61.    do
62.    {
63.       i = ob1.run();
64.       cout << i << " ";
65.    }while( i > CounterNameSpace::lowerbound);
66.    cout << endl;
67.
68.    //下面我们将使用整个 CounterNameSpace 的命名空间
69.    usingnamespace CounterNameSpace;
70.    counter ob2(20);
71.    do
72.    {
73.       i = ob2.run();
74.       cout << i << " ";
75.    }while( i > CounterNameSpace::lowerbound);
76.    cout << endl;
77.
78.    ob2.reset(100);
79.    lowerbound = 90;
80.    do
81.    {
82.       i = ob2.run();
83.       cout << i <<" ";
84.    }while( i > lowerbound);
85.
86.    return 0;
87.  }
```

上面的程序还为我们演示了重要的一点：当我们用 using 引入一个命名空间的时候，如果之前有引用过别的命名空间（或者同一个命名空间），则不会覆盖掉对之前的引入，而是对之

前引入内容的补充。也就是说，到最后，上述程序中的 std 和 CounterNameSpace 这两个命名空间都变成全局空间了。

没有名称的命名空间

有一种特殊的命名空间，叫做未命名的命名空间。这种没有名称的命名空间使得我们可以创建在一个文件范围里可用的命名空间。其一般形式如下：

```
namespace
{
  //声明
}
```

我们可以使用这种没有名称的命名空间创建只有在声明他的文件中才可见的标识符。也即是说，只有在声明这个命名空间的文件中，它的成员才是可见的，它的成员才是可以被直接使用的，不需要命名空间名称来修饰。对于其他文件，该命名空间是不可见的。我们在前面曾经提到过，把全局名称的作用域限制在声明他的文件的一种方式就是把它声明为静态的。尽管 C++是支持静态全局声明的，但是更好的方式就是使用这里的未命名的命名空间。

std 命名空间

标准 C++把自己的整个库定义在 std 命名空间中。这就是本书的大部分程序都有下面代码的原因：

```
using namespace std;
```

这样写是为了把 std 命名空间的成员都引入到当前的命名空间中，以便我们可以直接使用其中的函数和类，而不用每次都写上 std::。

当然，我们是可以显示地在每次使用其中成员的时候都指定 std::，只要我们喜欢。例如，我们可以显示地采用如下语句指定 cout：

```
std::cout << “显示使用 std::来指定 cout”;
```

如果我们的程序中只是少量地使用了 std 命名空间中的成员，或者是引入 std 命名空间可能导致命名空间的冲突的话，我们就没有必要使用 using namespace std;了。然而，如果在程序中我们要多次使用 std 命名空间的成员，则采用 using namespace std;的方式把 std 命名空间的成员都引入到当前命名空间中会显得方便很多，而不用每次都单独在使用的时候显示指定。

练习：

1. 什么是命名空间？创建命名空间时使用哪个关键字？
2. 命名空间是可追加的吗？
3. using 关键字是用来做什么的？

基本技能 12.6：静态的类成员

我们在前面的第七章节中学过使用 static 关键字来修饰局部和全局变量。除此之外，static 还有另外的用法：他可以被用来修饰类的成员，包括成员函数和数据成员。我们将在本小节对此进行一一学习。

静态的成员变量

当我们在类的成员变量前加上 static 关键字的时候，就是告诉编译器该变量是共该类的所有成员共享的，只存在一份，而不是每个对象都持有一份的。和普通的数据成员不同，只有一份的静态成员不是为类的每一个对象单独存在的。不管程序中创建了多少个该类的对象，静态数据成员只存在一份。也就是说所有类的对象使用的是同一个成员。所有的静态变量在没有指定初始化值得时候都会被初始化为 0。我们在类中声明静态数据成员的时候，我们紧紧只是声明了这个变量，而没有定义它。也就是说我们必须在类的外面为之提供全局的定义。这点是通过使用范围解析运算符来重新声明它而完成的。范围解析运算符前面的类表明该数据属于那个类。这也是为静态数据分配存储空间的过程。下面就是一个使用静态成员的示例程序：

view plain

```
1.  //使用静态变量
2.
3.  #include<iostream>
4.  using namespace std;
5.
6.  class ShareVar
7.  {
8.    static int num;//静态成员，由所有该类的对象共享
9.  public:
10.   void setNum(int i)
11.   {
12.     num =i;
13.   }
14.
15.   void shownum()
16.   {
17.     cout<<num<<" ";
18.   }
19.
20. };
21.
22. int ShareVar::num;//定义 ShareVar 的静态数据成员
23.
24. int main()
25. {
26.   ShareVar a,b;
27.
```

```
    a.shownum();//将输出 0
    b.shownum();//将输出 0

    a.setNum(10);//设置静态数据成员为 10
    a.shownum(); //将输出 10
    b.shownum(); //将输出 10

    return 0;

}
```

上面程序的输出如下：

```
0  0  10  10
```

在上面的程序中，我们对静态的整型变量 num 在类 ShareVar 中进行了声明，同时也作为全局变量进行了声明。正如我们在前面提到的那样，这样做时很有必要的，这是因为在类 ShareVar 中对 num 的声明并不代表为 num 变量分配空间；而是在作为全局变量声明的时候才分配空间的。同时由于作为全局变量声明的时候没有对其进行初始化，C++自动使用 0 来对其进行初始化。这也是第一次调用 shownun()函数的时候输出结果为 0 的原因。紧接着，我们使用 ShareVar 类的对象来对 num 设置为 10；然后调用 a 和 b 的 shownum()来显示他的值。由于 num 是有 a 和 b 共享的，所以两次 shownum()的结果都是 10。

当静态数据被声明为是共有的时候，我们可以通过类名来直接使用该变量；而不用使用该类的任何对象。当然也是可以通过类的对象来使用该变量的。例如：

view plain

```
//通过类名来方位类的静态变量

#include<iostream>
using namespace std;

class Test
{
public:
  static int num;
  void shownum()
  {
     cout<<num<<endl;
  }
};

intTest::num;

```

```
int main()
{
    Test a,b;

    //通过类名来使用静态变量
    Test::num = 100;

    a.shownum();//将输出 100
    b.shownum();//将输出 100

    //通过对象类使用该静态变量
    a.num = 200;

    a.shownum();//将输出 200
    b.shownum();//将输出 200

    return 0;
}
```

请注意在上面的程序中我们是如何使用类名来设置 num 的值：

```
Test::num = 100;
```

同样，我们也是可以通过对象来使用 num 的：

```
a.num = 200;
```

上面这两种方式都是有效的。

静态的成员函数

同样，我们也可以把类的成员函数声明为静态的。但是他的使用方式就有些变化。一个在类中声明的静态成员函数只能访问该类的静态数据成员。（当然，静态的成员函数是可以访问非静态的全局数据和函数的。）静态函数没有 this 指针。C++不允许出现虚的静态函数（也就是静态函数被 virtual 修饰）。同样，静态函数也不能被声明为是 const 或者是 volatile。我们可以通过该类的任何成员来调用类的静态成员函数，也可以通过类名和范围解析运算符来调用类的静态成员函数，不需要任何对象。例如下面的程序。其中定义了一个静态变量 count 用来对当前存在的类的对象进行计数。

view plain

```
//静态成员函数的示例程序

#include<iostream>
using namespace std;

```

```
6.  class Test
7.  {
8.    static int count;
9.  public:
10.   Test()
11.   {
12.     count++;
13.     cout<<"constructing object "<<count<<endl;
14.   }
15.
16.   ~Test()
17.   {
18.     cout<<"Destoring object "<<count<<endl;
19.     count--;
20.   }
21.
22. static int numObjects()//静态成员函数
23.   {
24.     return count;
25.   }
26. };
27.
28. intTest::count;
29.
30. int main()
31. {
32.   Test a,b,c;
33.
34.   cout<<"There are now "
35.     <<Test::numObjects()
36.     <<" in existence./n";
37.
38.   Test *p =new Test();
39.
40.   cout<<"After allocating a Test object, "
41.     <<"there are now "
42.     <<Test::numObjects()
43.     <<" in existence./n";
44.
45.   delete p;
46.   cout<<"After deleting an object, "
47.     <<"there are now "
48.     <<a.numObjects()
49.     <<" in existence./n";
```

```
50.
51.     return 0;
52. }
```

上面程序的输出如下：

```
constructing object 1
constructing object 2
constructing object 3
There are now 3 in existence.
constructing object 4
After allocating a Test object, there are now 4 in existence.
Destoring object 4
After deleting an object, there are now 3 in existence.
Destoring object 3
Destoring object 2
Destoring object 1
```

这个程序为我们演示了类的静态成员函数的调用方法。程序中的前两次对静态成员函数的调用都是通过类名加上范围解析运算符：

```
Test::numObjects();
```

在第三次调用的时候，使用的是和普通函数一样的方式：对象加点号。

基本技能 12.7：运行时的类型标识

运行时的类型信息是非多态性语言中如 c 或者传统的 BASIC 没有的一个特性，所以我们可能对此有些陌生。在这些非多态性的语言中，没有必要有运行时的类型信息。这是因为每个对象的类型在编译的时候就是确定的（也就是在编写程序的时候就是确定的）。然而，在具有多态性的语言，如 C++中，可能存在编译时对象的类型不可知的情况，也就是说只有到程序运行时对象的类型才确定的情况。我们都知道，C++是通过使用类的继承关系，虚函数以及基类指针来实现多态性的。一个基类的指针可以指向基类的任何对象，也可以执行任何该基类的派生类的对象。因此，在任意指定的时刻，我们并不总是能够提前知道基类指针指向对象的实际类型。此时只有在程序运行时使用运行时类型标识才能确定基类指针指向对象的实际类型。

我们可以使用 typeid()来获取一个对象的类型。此时我们必须引入头文件<typeinfo>。其常用的形式如下：

```
typeid(对象);
```

其中，对象就是要获取类型的那个对象。它可以是任意类型的，就包括了内置类型和我们自己定义的类类型。typeid 返回的是对一个描述了对象类型的 type_info 对象的引用。（听起来很绕口！就是说返回的是对一个对象的引用。这个对象的类型是 type_info 类型。返回的这个对象就描述了 typeid()中参数对象的类型）。

type_info 类定义例如如下的共有成员：

```
bool operator == (consttype_info&ob);
bool operator !=(consttype_info * ob);
bool before(consttype_info&ob);
const char * name();
```

其中对==和!=重载是用来对类型进行比较的。before()函数是用来判断调用者是否在参数对象之前。这种前后的关系是按照对类的整理顺序来讲的。（这个函数绝大多是都是内部使用的。它的返回值和类的继承关系已经层次图没有任何关系。）name()函数返回的是指向类型名称的指针。

下面是一个使用 typeid()的示例程序：

view plain

```
<span style="font-family:'Courier New';">//使用 typeid()函数的示例程序

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

class MyClass
{
  //...
};

int main()
{
  int i,j;
  float f;
  MyClass ob;


  //使用函数 typeid 来获取对象的运行时类型信息
  cout<<"The type of i is:"<<typeid(i).name();
  cout<<endl;
  cout<<"The type of f is:"<<typeid(f).name();
  cout<<endl;
  cout<<"The type of ob is:"<<typeid(ob).name();
  cout<<endl;
```

```
26.
27.    if(typeid(i) ==typeid(j))
28.    {
29.       cout<<"The types of i and j are the same/n";
30.    }
31.
32.    if(typeid(i) !=typeid(f))
33.    {
34.       cout<<"The types of i and f are not the same/n";
35.    }
36.
37.    return 0;
38.
39. }</span>
```

上面程序的输出如下：

```
The type of iis:int
The type of f is:float
The type of obis:classMyClass
The types of i and j are the same
The types of i and f are not the same
```

或许 typeid()函数主要是用在当我们使用具有多态性的基类指针的时候。此时，该函数会自动返回指针指向对象的实际类型，一个基类类型或者是派生类类型（请记住：基类指针是可以指向任意基类的对象和任意该基类的派生类的对象的）。因此，此时使用 typeid()函数我们就可以在运行时确定一个基类指针指向对象的类型。下面的程序就对此进行了演示：

view plain

```
1.  <span style="font-family:'Courier New';">//在具有多态性的层次关系中使用时 typeid()函数的示例程序
2.  #include<iostream>
3.  #include<typeinfo>
4.
5.  using namespace std;
6.
7.  class Base
8.  {
9.    virtual void f()//虚函数使得该基类具有多态性
10.   {
11.   }
```

```
12.     //...
13.  };
14.
15.  class Derived1:publicBase
16.  {
17.     //...
18.  };
19.
20.  class Derived2:publicBase
21.  {
22.     //...
23.  };
24.
25.  int main()
26.  {
27.     Base *p,baseob;
28.     Derived1 ob1;
29.     Derived2 ob2;
30.
31.     p = &baseob;
32.     cout<<"p is pointing to an object of type ";
33.     cout<<typeid(*p).name() <<endl;
34.
35.     p = &ob1;
36.     cout<<"p is pointing to an object of type ";
37.     cout<<typeid(*p).name() <<endl;
38.
39.     p = &ob2;
40.     cout<<"p is pointing to an object of type ";
41.     cout<<typeid(*p).name() <<endl;
42.
43.     return 0;
44.  }</span>
```

上面程序的输出如下：

```
p is pointing to an object of type class Base
p is pointing to an object of type class Derived1
p is pointing to an object of type class Derived2
```

正如程序输出的那样：当传入 `typeid()` 函数中的参数是一个具有多态性的基类指针的时候，在程序运行时指针指向的对象的实际类型会被返回。

当 typeid()函数作用于不具有多态性的基类指针的时候，它总是返回该基类的类型。也就是说不管该指针指向对象类型是基类对象还是派生类的对象。我们可以把上面程序中的虚函数注释掉，观察程序的运行结果。我们可以看到函数返回的类型总是基类的类型。
由于 typeid()通常总是和取指针指向对象的运算符*一起使用，就出现了针对空指针取其指向对象的一个特殊的异常：bad_typeid。也就是说此时 typeid()会抛出该异常。

对于具有多态性的类层次的引用和也指针一样。此时，当传入 typeid()的参数是对具有多态性的类的引用的时候，函数会返回对运行时实际对象的类型。此时就很有可能是一个派生类型。当我们通过引用来传递参数的时候就有可能使用到该特性。

typeid()函数的第二种形式：

typeid(类型名称)

例如：下面的形式是完全正确的：

```
cout<<typeid(int).name();
```

这种写法主要是用于通过类型来获取描述该类型的相关信息，以便可以用于进行类型比较。

练习：

1. 静态成员有什么特殊性？
2. typeid()是用来做什么的？
3. typeid()返回的是什么类型？

基本技能 12.8：类型转换运算符

C++中定义了五种类型转换运算符。其中的第一种就是我们在前面描述过的传统的类型转换。其余的四种类型转换运算符是前几年才加入到 C++中。它们是：dynamic_cast，const_cast，reinterpret_cast 和 static_cast。这些运算符使得我们在进行类型转换的时候可以获得更多的控制权。下面逐一对其进行简单的介绍。

dynamic_cast

或许后来增加的这四种类型转换运算符中最重要的就是 dynamic_cast（动态类型转）了。动态类型转换是在程序运行时的类型转换，并能对转换是否有效进行验证。如果在进行类型转换时发现转换是无效的，则动态类型转换会失败。其通用的形式如下：

dynamic_cast<目标类型>(表达式)

其中，目标类型就指定了转换后的类型；表达式就是需要转换的对象了。其中的目标类型必须是指针或者引用类型，并且表达式也必须是可以被评估为指针或者引用的表达式。因此，动态类型转换可以被用来把一种类型的指针转换为另外一种类型的指针，或者把这一种引用转换为另外一种类型的引用。

引入动态类型转换的目的是针对具有多态性的类的。假设有两个多态性的类 B 和 D；D 是从 B 继承而来的。动态类型转换总是能成功地把 D 类型的指针转换为 B 类型的指针。这是因为基类指针是可以指向任意的派生类的对象的。但是动态类型转换只有在 B 类的指针指向对象的实际类型是 D 类对象的时候才能把该 B 类指针转换为 D 类型的指针。通常，动态类型转换在被转换指针指向的是目标类型的对象或者是目标类型的派生类的对象时才能转换成功；在其它情况下转换都会失败。在转换失败的情况下，如果是进行指针类型的转换，那么转换的值就是 null；如果进行的是引用类型的转换，则会抛出 bad_cast 异常。

下面是一个简单的例子。假定 Base 是具有多态性的类；而 Berived 是从 Base 派生而来的类。

```
Base* bp, b_ob;
Derived * dp, d_ob;

bp = &d_ob; //基类指针指向派生类的对象

dp = dynamic_cast<Derived *> (bp)'//将基类的指针转换为派生类的指针

if(dp) cout << "Cast OK"
```

在这段代码中，由于基类指针 bp 指向对象的是派生类的对象，因此从基类指针到派生类指针的转换是成功的。因此，上述代码片段将显示 CastOK。但是在下面的代码片段中，由于 bp 指向的是基类对象，把基类对象转换为派生类对象是不合法的，因此类型转换会失败。

```
bp = &b_ob; //基类指针指向派生类的对象

dp = dynamic_cast<Derived *> (bp);//将基类的指针转换为派生类的指针

if (!dp) cout << "Cast Fails";
```

由于转换失败了，所以会输出 Cast Fails。

const_cast

const_cast 是用来对常量或者易变量进行类型转换的。其目标类型和源类型除了 const 或者 volatiole 属性可以不一样外，其类型一样的。const_cast 最常用的就是用来移除 const 属性。其通用的形式如下：

```
const_cast<类型>(表达式)
```

其中的类型就是指定的目标类型；表达式就是需要对其进行类型转换的表达式。必须强调的是使用 const_cast 来去除常量属性很有可能存在潜在的危险，因此在使用的时候需要格外的注意。

还有一点需要注意：只有 const_cast 能够移除对象的 const 属性。也就是说 dynamic_cast,static_cast 和 reinterpret_case 都不能修改对象的 const 属性的。

static_cast

静态类型转换进行的是非多态性的转换。它可用于任意的标准转换，也不会进行运行时检查。因此，静态类型转换实质上是传统的类型转换的替代方式。其通用形式如下：

static_cast<目标类型>(表达式)

其中的目标类型就表明了转换的目的类型；表达式就是需要对其转换的表达式。

reinterpret_cast

这个运算符把一种类型转换为另外一种从本质上完全不同的类型。例如，把指针转换为整型数或者把整型数转换为指针。它还可以被用来就继承关系上来说不兼容得指针类型的转换。其通用形式如下：

reinterpret_cast<目标类型>(表达式)

其中的目标类型就指明了转换的目的类型；表达式就是需要对其类型转换的表达式。

更深入 C++还有什么？

本书的目的是介绍 C++语言的核心元素.这些核心元素都是我们日常使用 C++编程时所需要的特性和技术。就目前介绍的知识来讲，我们已经能够编写真实的、专业的程序了。然而，C++语言是一个非常丰富的语言。它含有很多高级的特性，我们仍然需要掌握。这其中包括：

l 标准模板库(STL)

l 显示构造函数

l 转换函数

l 类的 const 成员函数和 mutable 关键字。

l asm 关键字

l 重载数组索引运算符[]，重载函数调用运算符()，重载动态分配和释放运算符 new 和 delete

在上述内容中，或许最重要的就是标准模板库了。STL 是一个模板类的库。它提供了现成的、可用于解决常用的数据存储问题的方案。例如，标准模板库中定义了通用的数据结构，如队列，栈，列表等，我们可以在程序中直接使用。

我们也许还需要学习一下 C++函数库。其中含有大量能简化我们程序的函数。

我建议大学阅读我的《C++: The Complete Reference》(《C++完全手册》)来进一步学习 C++语言。该书由 Osborne/McGraw-Hill, Berkeley,California 出版社出版。这本书除了涵盖了本书的全部内容外，还有更多别的东西。现在我们已经有充足的知识来阅读这本更加深入介绍 C++的书了。

第 12 章练习题

1. 说明 try,catch 和 throw 是如何协调工作来支持异常处理的?
2. 当需要同时捕获基类和派生类的异常时我们应该如何组织 catch 子句列表?
3. 写出如何声明一个返回值为 void 类型的函数 func()，该函数抛出 MyExcpt 类的异常。
4. 为项目 12-1 中的通用队列定义一个异常：在队列上溢和下溢的时候抛出该异常。并写出如何使用这个异常。
5. 什么是通用函数? 使用什么关键字来创建通用函数?
6. 为项目 5-1 创建通用的 quicksort()和 qs()函数。并演示如何使用之。
7. 使用下面定义的 Sample 类，并使用项目 12-1 中的通用队列来创建一个含有三个 Sample 类对象的队列。

view plain

```
class Sample
{
   int id;
public:
   Sample()
   {
      id = 0;
   }
   Sample(intx)
   {
      id =x;
   }

   void show()
   {
     cout<<id<<endl;
   }
}
```

8. 重写针对问题 7 的答案，用动态分配的方式生成其中的三个 Sample 类的对象。

9. 写出如何声明一个名称为 RobotMotion 的命名空间。

10. 哪个命名空间中囊括了 C++标准库？

11. 类的静态成员函数是否可以访问类的非静态数据？

12. 哪个运算符可以在运行时得到一个对象的类型？

13. 使用哪个运算符可以在运行时检测类型转换的有效性？

14. const_cast 是用来做什么用的？

15. 自己针对项目 12-1 进行重写：队列类放入到 QueueCode 命名空间中，并放入到自己的文件 Queue.cpp 中，然后修改 main()函数，使用 using 语句来引入 QueueCode 命名空间。

16. 请继续学习 C++。它是当前世界上最强大的计算机语言。掌握了它，我们就成为了众多编程者中的中坚分子。