

第一部分 C++ 基础

第一章 面向对象

1.1 简介

面向对象(Object-Oriented)已经成为90年代重要主题词之一。可用如下说法简单地表达面向对象的整体思路:先对特定问题建立通用的解决方案,然后再改制这些方案以适应特定需要。这样的设计方法内在的优点是:

- 能够直接地再利用他人已经设计并编写成功的程序,而不需要再作广泛的重新测试。
- 能够由基本而通用的解决方案派生出新的解决方案。
- 能够建立一个由模块组成的系统。

用实物作比拟会有助于弄清模块化系统的概念。比如一种组合型长椅,是按这样的结构设计制造:适合某种特定需要,其中的一块或几块既可以加到长椅的椅架上且可以从椅架上拿开,又不影响长椅本身的性能和设计上的美观。又比如一个模块化的房间,可以设计得既容易得到扩展又不影响房子的基本结构。模块化系统与上述所列的比拟很相似,是由包含系统基本功能的固定基组成的。这个固定基是待完善系统的构件,它容易扩展,新的功能可从这个基派生出来,无需进行广泛的重新测试和编码。

面向对象的系统的主要缺点(如果称之为缺点的话),就是无法正确地设计一个足够通用的解决方案来构成系统的基。因此,要建立一个强大而有用的面向对象的系统,使系统具有长期的可用性是至关重要的。

C++是一种面向对象的编程语言。但如果系统没有采用面向对象的技术,C++只能作为面向过程的语言来使用。为了将C++带入生活,读者必须考虑面向对象。在编写第一行程序之前,必须尽力地花时间去思考系统的设计、系统的基以及期望由系统基所派生出来的解决方案。如果没有考虑到面向对象和模块化,那C++这种工具的面向对象强大之处将变得毫无用处,因为没有去正确使用它。本章将教会读者在编程之前如何构思。

1.2 抽象

抽象是面向对象设计的核心思想。抽象的艺术在于抓住一个系统最核心的特性,这些特性是能够表述以后发生的任何事情 of 的构件。虽然抽象是一个简单易懂的概念,但读者会惊讶地发现,在设计一些系统的时候,使用这种方法的频度几乎微乎其微!本章提供了一个极复杂系统的极简单的设计。我们的目的是要说明怎样把一个极复杂的问题更好地分解成一些最简单的组成部分,又如何建立并修改一个已经用模块化风格分解产生的系统。

为简化起见,我们只考虑系统最通用的特性。

1.3 设计一个电讯系统

当您拿起话筒给别人打电话时，在您和另一方之间就建立了一个连接。简单说，这就是两点间的连接。您的目标就是要设计一个能获得这种连接的系统。

当然，实际生活中不会有这样简单的两点间的直接连接。所以我们要给系统增加一些复杂性。

例如，假设您要给尼古拉挂个电话，通话线路可能有几种不同方式：

- 您非常幸运地有一条直达热线接到尼古拉那里。在这种情况下，电话线路直接联接读者和尼古拉：

读者 -----> 尼古拉
(无问题)

- 您不得经过另外一个地点的线路才可最终与尼古拉相接通，即：

读者 -----> 太德 -----> 尼古拉

在这种情况下，您到太德那里有一条直达线，而太德到尼古拉那里又有一条直达线。为了将您和尼古拉连通，必须先经过太德。

- 如果太德的线路处于忙的状态又会怎样呢？那就不能通过它来与尼古拉相连。这就必须考虑其复杂性。读者的呼叫经由另外一条绕过太德的线路到达尼古拉那里。如：

读者 -----> 太德 -----> 尼古拉
|----- (忙) -----|
| |
|-----> 汤姆 -----> |

- 现在，看看最复杂的情况。在现实生活中，象我们刚才谈到的这些不同的连接（您，太德，汤姆，尼古拉），实际上可认为是物理的硬连接的机器。用某种方式对这些机器编程，使之能接受某种类型的电话呼叫，再将呼叫导向目的地。这种逻辑被编入机器（从现在起，我们称这些机器为设备），以便能理解那些对它们的呼叫的类型。然而，如果这些呼叫类型不同，那么设备就不知道如何处理，因而拒绝这些呼叫。从技术上讲，不同类型的设备具有理解不同类型协议的能力，如果一个电话呼叫在逻辑上不遵守这个设备能理解的协议，则该呼叫就会遭到拒绝。

这样，在为某一呼叫进行路由选择的时候，必须鉴别一下要通过设备的性质，以确保该设备确实有能力处理该呼叫并为之选择路径。

1.4 电讯系统的再思考

如果读者再花点时间去重读上一段落，会发现在设计该系统的时候需要涉及以下几个问题：

- 怎样获得可能毗连或非毗连的两点间的连接？
- 怎样获知不同设备的不同行为特及这些设备能确定连接的不同类型？
- 如果在正常情况下选择的逻辑连接不可用，那么如何将该呼叫导向另外的设备？

下一步的任务就要把眼光集中在为实现本系统而要明确表达的各种不同类型对象的基本特性上。我们试图简化上述问题的复杂性，简化结果如下：

- 两点间最简单的连接就是彼此间毗连的两点的连接，它们之间就可以直通。因而要确定的第一个对象是获得这种连接的本质。
- 所有的设备的都要求最终能实现一种主要功能：将一个呼叫向前传递给下一点。因而要确定的另一个对象是获得一个基本设备这一特征的本质。
- 反呼叫从一个设备导向另一个设备之前，第一个设备必须向下一个设备发送信息以询问将呼叫发给下一个点是否可行。下一个设备也必须给这个设备回送一个适当的应答信息。如果该设备可行，发送呼叫的设备就将这个呼叫传递给它。如果不可行，那么发送呼叫的设备就要试图找到另一条路径。这样，就可以归纳成一种机制，在这种机制下，上面所确定的对象彼此间能进行通讯。

刚才所罗列的简单概括可以构成整个系统的骨架。一旦一个系统设计得能够满足所列的要求，就有足够能力处理以后的复杂问题，在下一小节再详述。

1.5 在基本抽象基础上建立系统

现在总结一下分析结果。在上一小节里，我们列举了两种对象的功能。在C++中，对象的功能用一种称作类（Class）的形式来表达。用来构成其它类的类叫作基类（Base Class）。在上一小节中，我们认定了两种用来确定该系统骨架的基类。这两种基类将构成所有由它们派生出来的其它类。对这两种基类规定如下：

- 获取在毗邻两点间传递呼叫所需规则的类。称之为 Class A。
- 获取用作呼叫路由选择的最基本设备的功能的类。称之为 Class B。

此外，我们还认定了这些类的对象彼此间进行通讯的机制，通过这些机制，完成呼叫的完整路程。

有了这些基本构成，我们就可以使系统更复杂些，并看到所设计的这些基类怎样成功地处理这些情况。下面是该系统所要求的另外一些特性。

- 一个呼叫可以在非毗连的两点间漫游，这是经过了其它连接才链在一起的。

我们从 Class A 派生出一个类来完成这种功能。这个派生类继承了明确表达两点间一对一连接所要求的规则。但是，它包含有另外的方法，此方法中有明确表达更复杂的连接所要求的规则。

- 各种不同类型的设备都可以加入到该系统中，用不同的方式来为一个呼叫选择路径。然而，为呼叫进行路由选择的基本机制总是相同的。

我们通过从 Class B 中派生出一些类的办法来完成该特性。这些派生类将继承最基本设备的功能。然而，每一个派生类都应该增加一些独特的属性或方法来将该设备与其它设备区分开。类之间彼此通讯的机制也就被确定下来。

这就应该是在设计一个大系统时所用的基本思维模式。本系统将能足够灵活地处理完整的路由选择和改变设备的属性所需的变化规则。这些变化将简单地通过修改派生类的应用代码来实现。

每个设备的方法都封装（encapsulate）在它所属的类中。如果系统中加入了新的设备，而该设备没有明确描述的属性，那么它可以利用基类中说明的功能。这种特性通过虚函数（virtual function）来表达。一旦这个新设备的属性通过多态性（polymorphism）被确定下来，那么这些虚函数就可被废弃（override）。

现在我们强调一下几个关键的短语和词汇，它们是：对象，类，基类，派生类，虚函数以及多态性。

1.6 结 论

尽管这里对一个复杂系统作了极其简单的表达，但这正是我们所要阐明的观点。尽可能地将复杂问题简化。在初始设计阶段把精力集中在主要的特性和需求上，而不是陷入到不重要的细节中。一定要从系统的最根本处入手。一旦当读者能并用这种方法分析并获得所需系统，读者会觉得，在系统的强大的基上构造系统是一件多么容易的事。系统的复杂性往往掩盖了潜在的，用经建立和实现系统时的简单性。需要设计者花时间去努力地思考如何反复复杂问题分解成最简单的形式，一旦能这样做，就会觉得其它什么事情都可按部就班了。

另一方面，如果设计中有缺陷，那么就会感到，试图在这样的设计上构造系统会混乱不堪。一个好的系统总是遵循正确的规则；而一个设计得不好的系统最终将混乱得几乎无法收拾，通常的结果是完全重来。

尽管本章是预备性的，建议读者在学习第二章之前能把第一章重读一遍（最好要慢一些）。我们在第二章讨论语言本身。

第二章 C++ 及编程基础

2.1 简介

在编写程序之前，必须懂得程序如何存储以及如何在计算机上运行。本章将描述什么是程序，程序如何被计算机处理和执行，然后，我们将编写并执行一个简短的C++ 程序。

2.2 什么是程序设计？

■ 程序是用能够被计算机理解的一种语言编写的语句的集合。

程序的正确执行通常会通过计算机得到某种输出结果。

C++ 程序包含对预先写好的代码的引用，这些预先写好的代码以库 (libraries) 的形式存储，可通过特殊的语句把这些库包含到C++ 代码中。

■ 程序要经过编译器、链接器和装载器的处理。

这些个别的实体只不过是存储在计算机中的不同类型的软件，是专门用来处理程序代码的。所有这些处理的最终结果是产生能被计算机所认识并能被计算机运行的代码。

程序设计从开始到完成所需的步骤如下：

第一步 程序员进行计算机登录。

第二步 程序员用喜爱的系统的编辑器编辑源程序。

第三步 程序员把编辑好的源程序存入文件中，这个文件叫做源代码文件。

第四步 程序员用适当的命令指示计算机编译源代码文件。

用来编译程序的命令因编程语言的不同而不同。例如，用来编译 C++程序的命令与用来编译 BASIC 程序的命令不同。如果使用的是菜单驱动系统，那么编译和运行程序的处理只不过是适当的菜单选项上敲击一下。

我们再继续前面的步骤，这就是编译开始后将会发生什么事情。

第五步 编译器将源程序代码翻译成汇编语言格式，然后传给计算机汇编器。

第六步 汇编器将汇编语言代码翻译成可重定位的目标码，并将这些目标码传送给链接器。

第七步 链接器将程序引用的所有支持例程连接在一起。这些例程存在于运行库中或早已存在于被程序所引用的预编译程序中，然后链接器产生源代码的最终版本，我们把它叫做可执行代码。该代码被传给执行程序的装载器。

第八步 运行程序。

虽然，整个处理过程显得很长，但是这些步骤对于程序员来说大多数是透明的。需要程序员做的只是以下几个步骤：

第一步 产生一个包含源代码的文件。

第二步 输入编译命令，得到编译结果。

第三步 运行程序。

下面将要讨论源代码文件命名的约定，在保存源代码文件时，要遵守这些约定。

2.3 源代码文件的命名约定

不同的操作系统有各自不同的C++ 程序源文件命名规则, 应参照系统手册来决定。一般来讲, C++ 程序用“.C”作后缀(大写C; 如果后缀是小写字母 c, 编译器将认为该程序是一个C 语言程序)。

2.4 编译器

本书中所有的程序都是在 IBM PC 机的 MS Windows 环境下用 Turbo C++ 编译器(V3.1版)编译并运行通过的。

本书中所用的程序都是可移植的。实际上, 可移植性也正是使C++ 语言颇具实力的原因之一。这些程序可以用任何一种 C++ 的编译器编译并运行, 可以在 Windows 环境下, 也可以不在 Windows 环境下进行。

Borland C++ 的编译器用 .c 后缀来识别 C语言程序, 用 .cpp 后缀来识别 C++ 程序。在本书中采用同样的约定。若使用其它的操作系统或编译器, 那么只需参阅相关资料中的适用说明。一般说来, 程序能象在本系统上一样正确地编译并运行。代码和输出都会是相同的。尽管出错信息的措辞会因使用不同的编译器而变化, 但这些信息的含义是相同的。

2.5 程序的产生和执行

要运行一个程序, 必须先用程序员所喜爱(或不甚喜爱)的编译器将程序敲进去, 并保存在文件中。一旦文件建立之后, 就可以按照正在使用的特定工具的要求去编译、链接和运行。有些工具允许在操作系统命令行提示符下键入命令来完成上述步骤, 而另一些系统则允许通过菜单和对话框的形式来完成整个过程。本书所用的系统, 全部是由菜单驱动的, 使用户可在一个集成的环境里编辑、编译并运行程序。对于其它类型的系统只需根据系统资料提供的说明来编译并运行。

2.6 Borland C++ 3.1 编译器

下面的章节将逐步讲述 Borland 公司的 Turbo C++ V3.1 编译器提供的集成开发环境中的编辑、编译和运行过程。读者可以通过定位并单击鼠标左边按钮来选择选项, 或简单地用键盘上的方向键移动高亮光条到相关选项并按 Enter 键。本书采用后一种方法。选项也可以通过与之相关联的热键或敲击标明该命令的字母来选择。

集成开发环境 (IDE) 是 Borland 的程序员开发平台。对其作简短的介绍以开始我们的讨论。

2.7 Borland 的程序管理器

在使用编译器之前, 必须将它安装到 PC 上, 必须执行编译器提供的INSTALL程序将所有可用文件装入计算机。该程序要求 PC 机中已经装入了 Microsoft Windows, 当启动 Windows 后, 进入程序管理器。安装成功后, 先退出, 再重新进入 Windows 环境, Borland C++ 将自动建立一个新的程序管理器组, 组中包含与该平台相关联的一组图标。在以 Borland C++ 3.1 为标题的窗口中共有 20 个图标, 如图 2.1 所示。

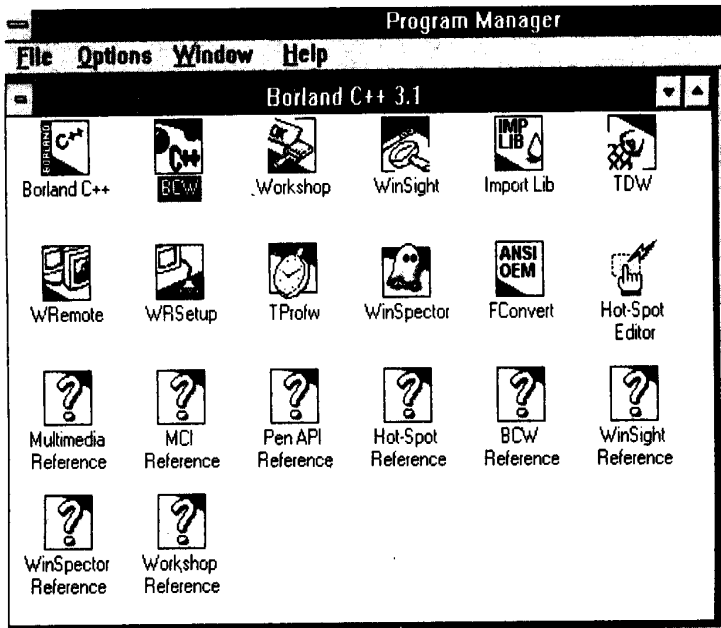


图 2.1 Borland C++ 3.1 图标组

我们只探讨用于 Borland C++ for Windows 编译器的菜单选项（左起第二个图标）。我们鼓励读者探索用于其余功能的图标。详细的说明，请参阅其它 C++ 资料。

在图 2.1 的窗口中，将光标移到标有 BCW 的图标（最上一行左起第二个图标）上，双击鼠标左按钮，就进入了 IDE 平台。这个环境包括窗口、菜单、对话框、加速条、卷滚条、编辑状态行。

2.8 IDE 平台

在 IDE 平台的顶部是菜单选项，后面将详细讨论。在菜单选项下面是加速条。加速条包含一组图标，这些图标为频繁使用的任务提供捷径。将鼠标箭头定位在其中一个图标上，都会在状态域中显示出该图标功能的简短描述。当鼠标箭头定位在？（帮助）图标（左起第一个）时，可见到图 2.2 的信息显示（Display Context Sensitive Help）。

窗口底部的箭头钮允许左右滚动，窗口右边的箭头钮允许上下滚动。

窗口本身可极大化（覆盖终端 屏幕的整个区域）；或极小化（变成一个图标），当单击窗口左上角的按钮时，从显示出的菜单中选择相关的项目即可实现极大化和极小化。

2.8.1 编辑窗口

再次观察编辑窗口时，就会注意到窗口顶部的标题行写着：

C: \ borlandc\bin\noname00.cpp.

编译器自动打开一个编辑会话窗口，并把要创建的 C++ 文件取名为 noname00.cpp。可以使用这个默认文件名，也可以通过 File 菜单选项改变该文件名。这一点我们后面会解释。

关于基础就讲这些。下面让我们浏览一下可显示各种选项的菜单条。

将鼠标箭头放在 File 选项（左起第一个菜单选项）上，单击它（或通过键盘上的方向键移动光标定位并敲回车键）。File 菜单条将显示出来。如图 2.3 所示。

可以用下面三种方式之一选择菜单条中的任一选项。

- 把鼠标移到要选择的菜单项上，然后单击鼠标。
- 如果菜单项中含有带下划线的字母，则可键入该字母。
例如要选 File 菜单条中的 New 菜单项，键入 N 即可。
- 如果存在热键（hotkey），可键入菜单项的热键。

例如对于 File 菜单条中的 Exit，可按住 Alt 键的同时敲 F4 键。

后面带有省略号（...）的每个菜单命令隐含着一个对话框，如果选择该命令，将激活对话框。该对话框叠加在当前窗口上。例如，如果读者选择 File 菜单条中的 Open 命令，Open a File（打开一个文件）对话框就显示出来。如图 2.4 所示。

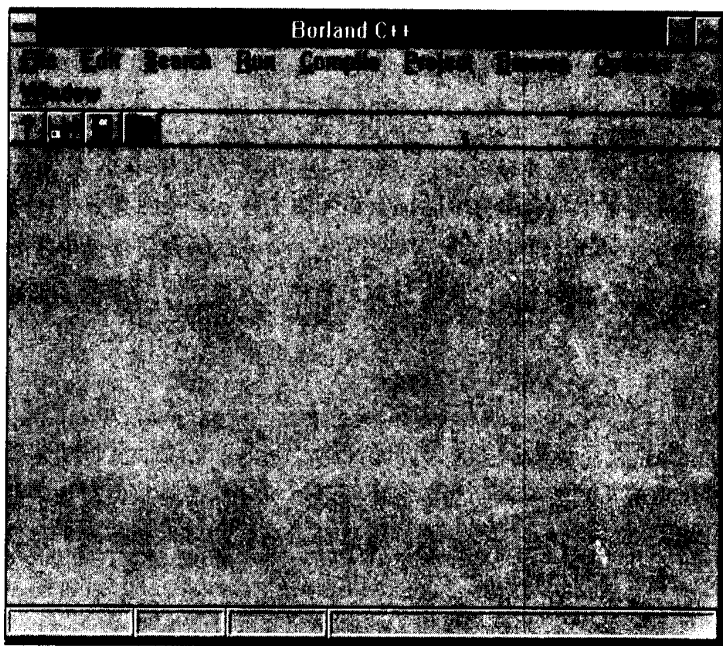


图2.2 Borland C++ 3.1编译器的 IDE 环境

对每个打开的对话框，键入适当的信息，然后单击 OK 按钮使系统接受所敲入的信息，或者单击 Cancel 按钮使系统废弃所键入的信息。单击 Help 按钮可以得到本窗口相关的帮助信息。

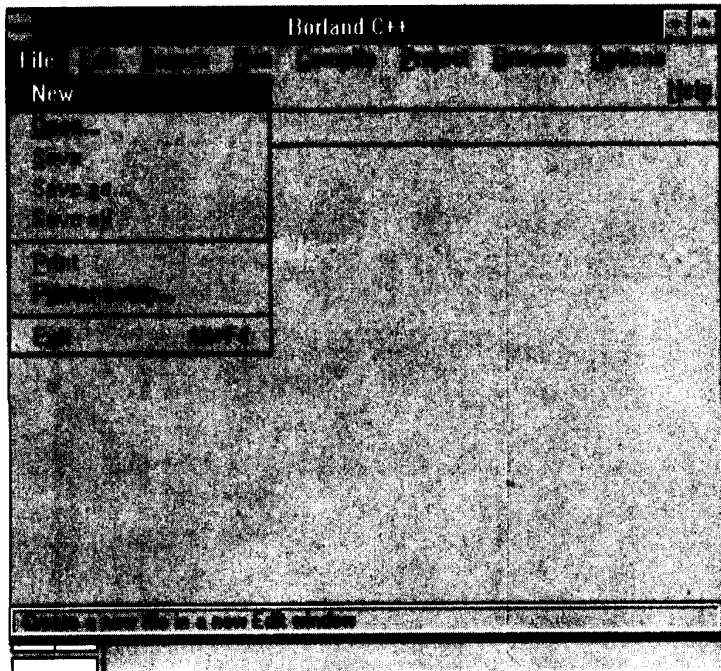


图2.3 IDE中的 File（文件）菜单条

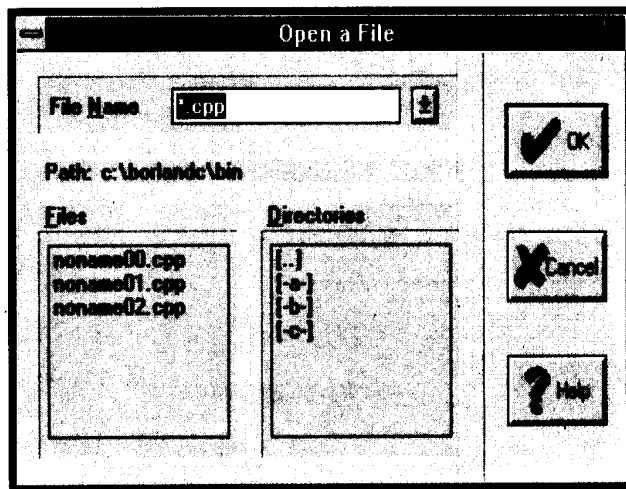


图2.4 Open a File（打开一个文件）对话框

图2.5～图2.13显示了所有可用菜单项的菜单条。

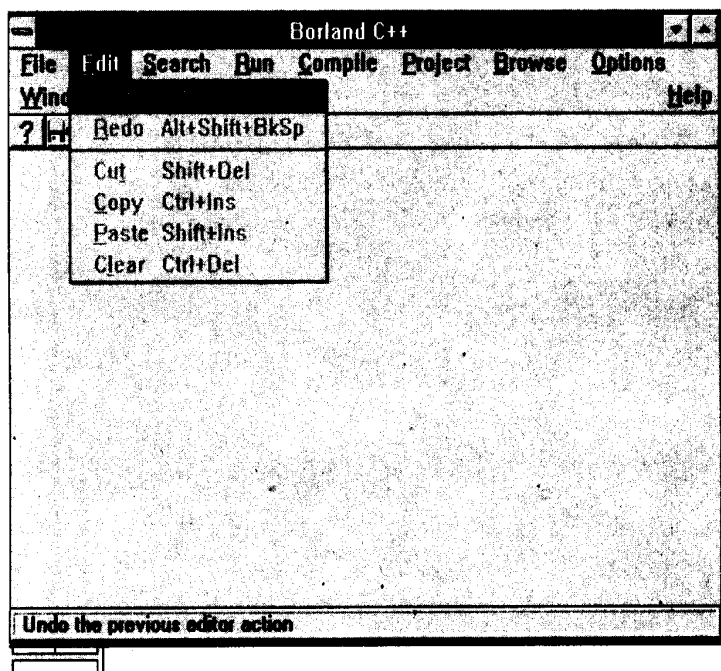


图2.5 IDE中的 Edit 菜单条

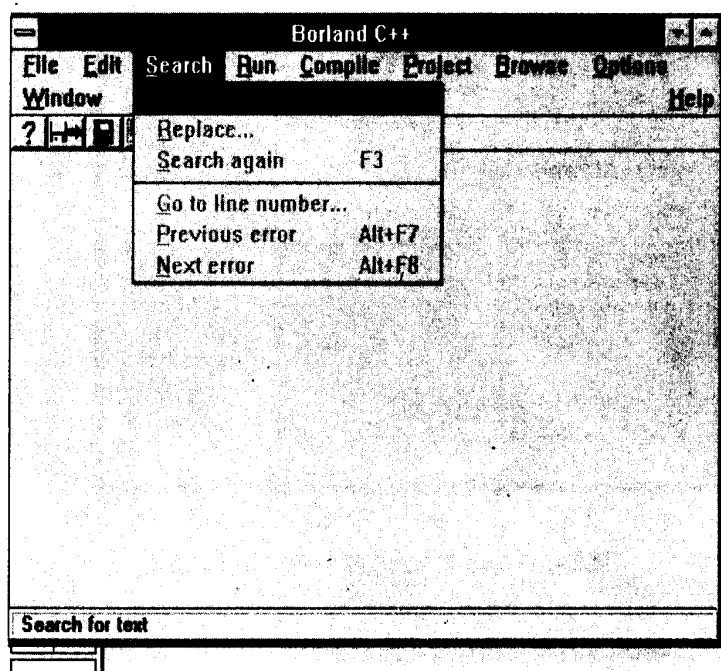


图2.6 IDE中的 Search 菜单条

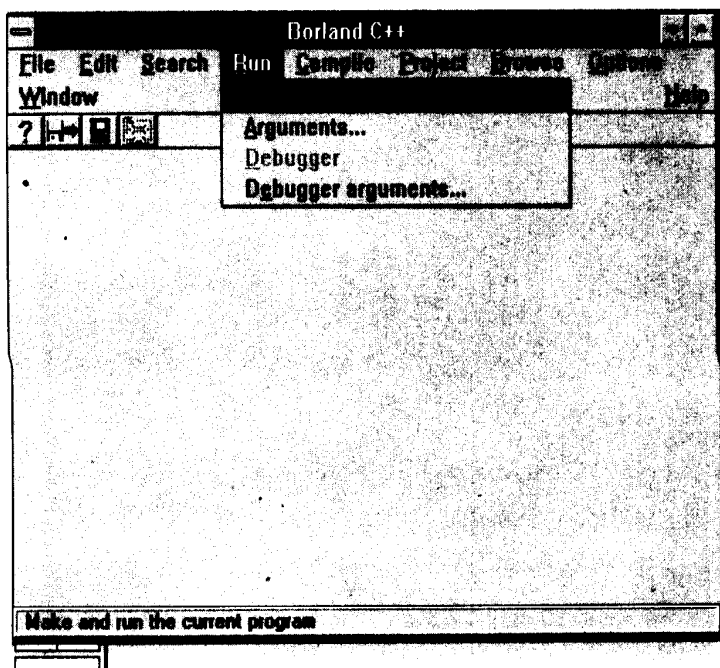


图2.7 IDE中的 Run 菜单条

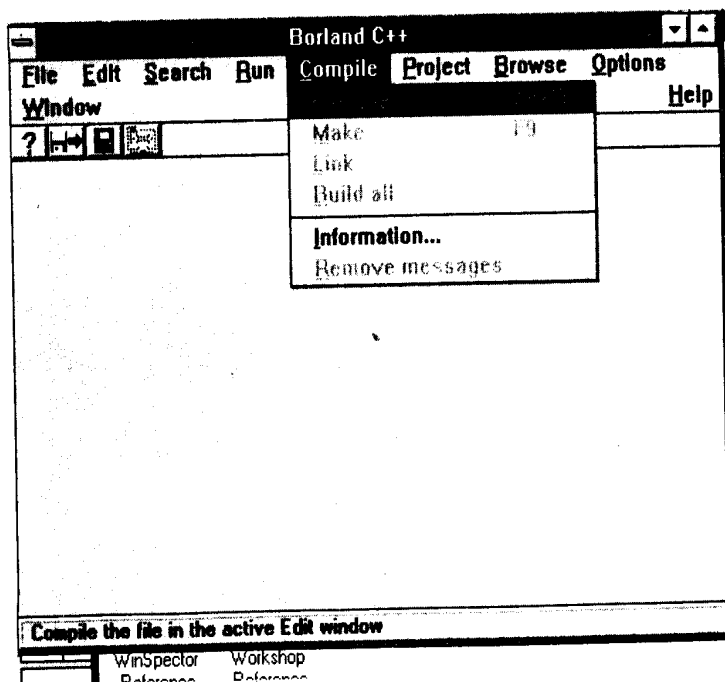


图2.8 IDE中的 Compile 菜单条

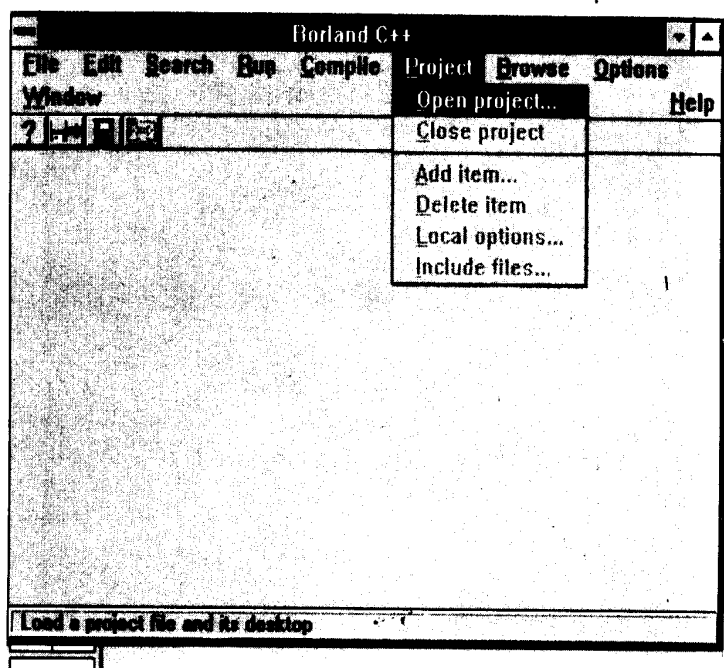


图2.9 IDE中的 Project 菜单条

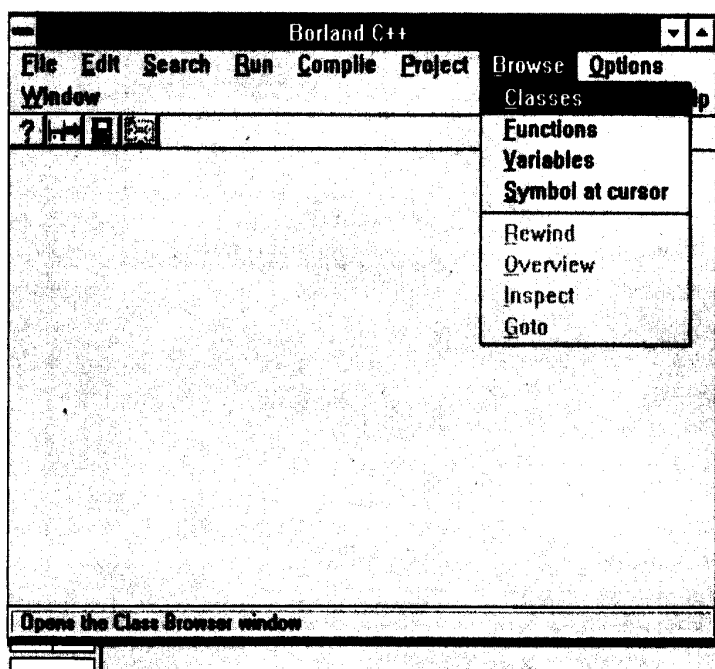


图2.10 IDE中的 Browse 菜单条

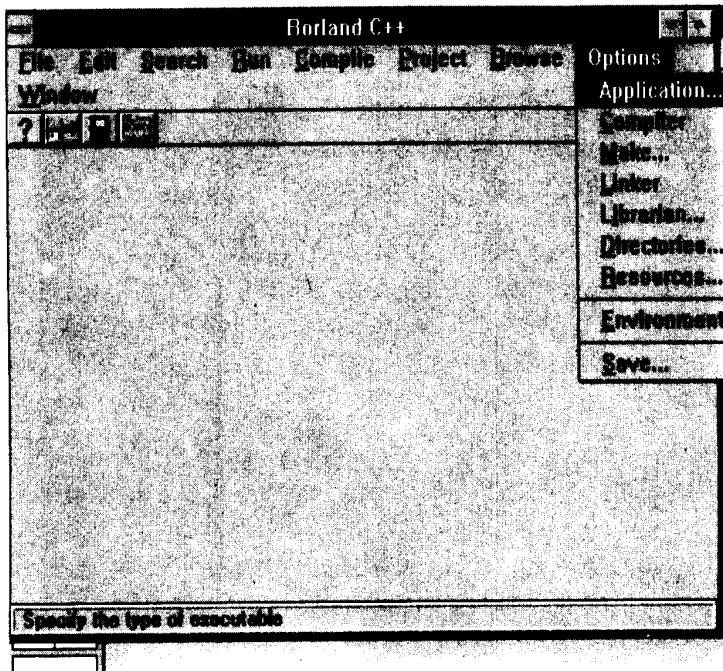


图2.11 IDE中的 Option 菜单条

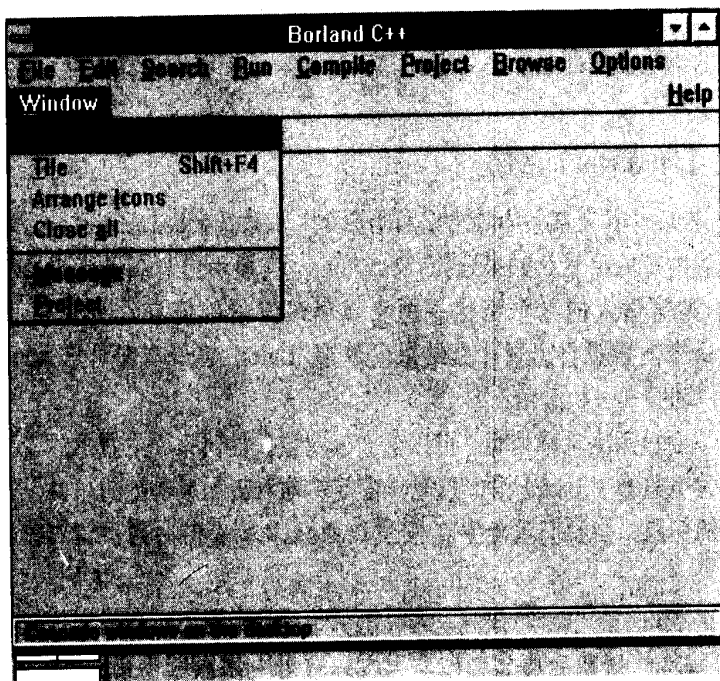


图2.12 IDE中的 Windows 菜单条

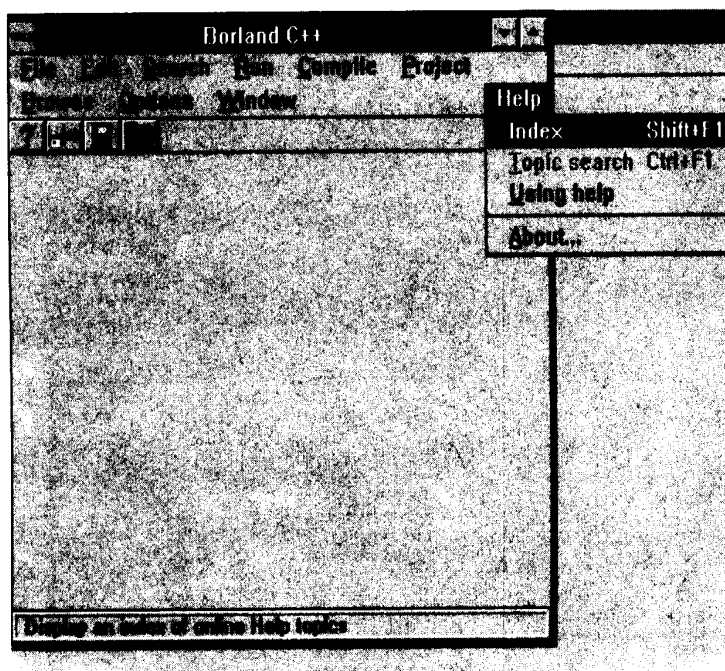


图2.13 IDE中的 Help 菜单条

对每个菜单条功能的解释不包括在本书的范围中，建议读者参考适当的系统手册以便得到更详细的说明。下面我们将要讲述在实际操作过程中如何获得联机 (On-Line) 帮助信息。

2.8.2 联机 (On-Line) 帮助

选择 Help 选项，其菜单条中将出现四个命令。如果选择 Using Help，就会显示出我们下面将要讲述内容的信息。如果选择 About...，将显示出对应于该产品的版权信息。Topic Search 则对输入的某个特定主题查找相关的帮助信息。首先我们选择 Index (Shift+F1) 命令，看看会出现什么结果。单击 Index 菜单项即可进入 Borland/Turbo C++ 帮助窗口，如图 2.14 所示。

窗口标题下面显示出主题列表，可以把光标定位在任何一个主题上。光标将由箭头型转变成指向该索引的手 (Hand) 形。这意味着简单地双击手形所指的项目，就会看到更多的关于该主题的信息。当我们双击窗口中的 Dialog Boxes 项目时，窗口显示信息将如图 2.15 所示。

单击 Help 窗口中的 Back 按钮，就可以从子序列中退出。单击 << 按钮会使文档向前滚动，单击 >> 按钮会使文档向后滚动。读者无须查阅手册，只需通过联机 (On-Line) 帮助即可浏览全部帮助信息。

我们现在简要地解释一下如何编辑、编译并运行一个典型的 C++ 程序。

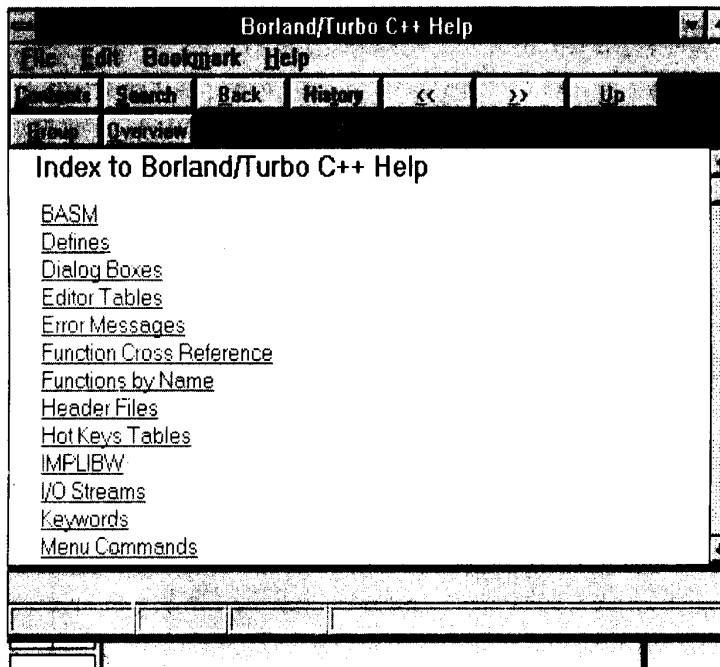


图2.14 Help 菜单条中的Index菜单子项

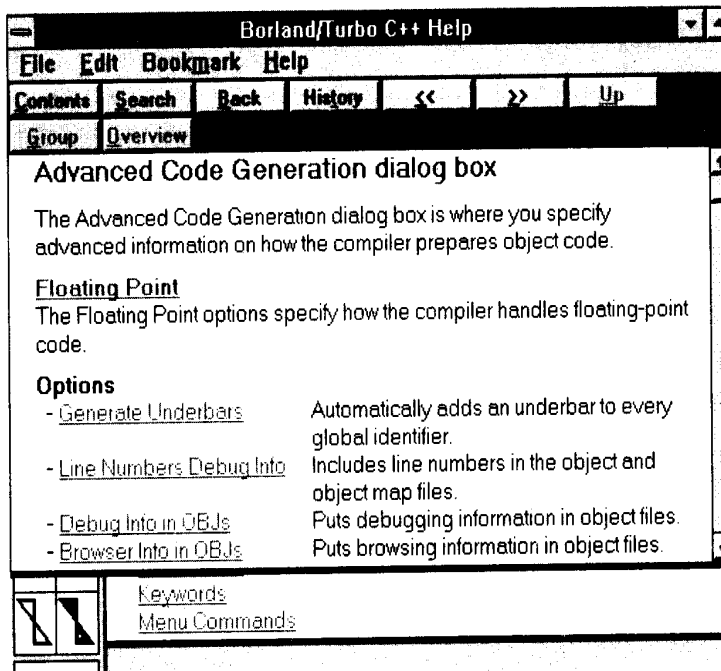


图2.15 对话框帮助信息

2.8.3 编辑、编译和运行过程

在主菜单中选择 File 选项, 就可以创建一个新文件或编辑一个已存在的文件。选择 Open 子项允许编辑一个已存在的文件。对话框如图 2.4 所示。

下面我们就要建立一个新文件。在 File 菜单的 New 子项上击点一下, 然后键入 C++ (或 C) 程序。Edit 菜单提供了一些特性, 可以更方便地编辑程序。多数情况下如果键入错了, 只需用方向键将光标移到需要修改的地方, 用回退键 (Backspace) 或删除键 (Delete) 删除字符然后重新键入。在键入程序时, 经常保存一下文件是个好习惯。

为了保存文件, 需要在 File 菜单条中选择 Save As... 命令, 单击该命令会出现保存文件对话框。如图 2.16 所示。

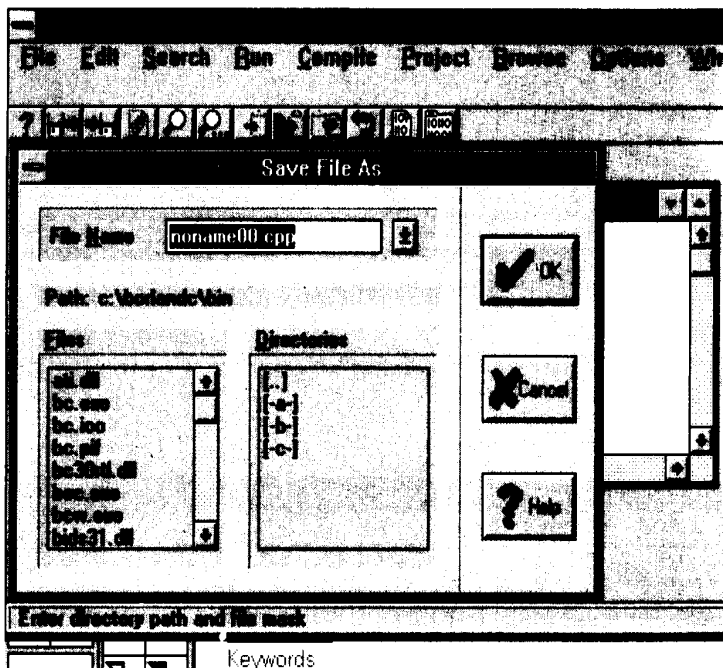


图 2.16 Save File As (保存文件) 对话框

单击要保存该文件的目录, 然后, 要么从显示的文件列表中选择文件名, 要么在 File-Name 域中键入一个新的文件名, 单击 OK 按钮, 文件就被保存起来了。

可以采用默认的文件名 (noname001.cpp), 或按前面所讲的方法键入文件名。但一定要确保 C 语言程序的后缀是 .c, C++ 语言程序的后缀是 .cpp。

文件存盘后, 可以从主菜单中选择 Compile (编译) 选项来进行编译。如果程序有错误, 编译器会如实地显示出来。只需重新编辑文件, 存盘, 再编译即可。

如果程序经过编译没有错误, 就可以从主菜单中选择 Run (运行) 选项。顺便提一下, 如果在存文件后直接选择 Run (运行) 选项, 会同时编译并执行程序。程序成功运行的输出结果将显示在新的窗口中, 该窗口会叠加在其它的窗口上。

表2.1 列出了一些常用的热键 (Hotkey)，这些热键能加快操作。

菜单条、对话框、窗口框架、热键等因编译器而异。但是选择项目和执行的过程与上面所述的不会有太大的差异。

表2·1 一些有用的热键

热键	结果
F1	帮助 (Help)
F2	保存文件 (Save)
F3	打开文件 (Open)
F4	运行程序 (Run)
F5	放大活动窗口 (Zoom active Windows)
F6	切换到下一窗口 (Go to next windows)
F10	切换到菜单条 (Go to menu bar)

2.9 非菜单系统下的编译过程

大多数 C++ 编译器典型的编译程序的命令是 CC。只要键入程序的名字即可运行，无需键入扩展名。例如：

CC source.cpp<编译名为source.cpp 的源代码程序>

source <运行该程序>

2.10 一个 C++ 程序

现在已经知道了如何使用菜单系统。让我们编写并运行一个简短的 C++ 程序。首先在编辑窗口键入下列源程序：

```
//test2-1.cpp
#include <iostream.h>
main()
{
    char name[20];
    cout<<"My name is :";
    cin>>name;
    cout<<name<<"\n";
}
```

在 File 菜单条中敲击 Save As 命令保存文件，Save File As 对话框将显示出来，单击要存放该文件的目录，然后在 File Name 域中键入 test2-1.cpp 并单击 OK 按钮，该文件就以 test2-1.cpp 为文件名保存起来。

现在，在主菜单中单击 Compile菜单项，如果是严格按照上面显示的程序键入的话，编译就不会有错误。

现在选择 Run菜单项开始运行，原来的屏幕上会叠加上一个新窗口，下面的内容会显示在窗口中：

My name is:

现在系统会等待输入，键入名字并按回车键。这样，刚刚键入的内容（名字）会再次显示，然后程序运行结束。

2.11 解释 C++ 实例程序

程序的第一行是：

```
//test2-1.cpp
```

// 用来表示注释，它可以用在程序的任意地方，// 后面的信息在编译时被忽略。

第二行是：

```
# include <iostream.h>
```

符号 # 后面紧接关键字 include，用来告诉编译器在当前程序编译时还要包括另一个文件。被包括的文件名是 iostream.h，文件名用尖括号括起来表示这个文件处在一个特定的目录中（这是在安装编译器时预定义的）。

iostream.h 含有 C++ 中的标准输入输出设备，换句话说，在需要执行输入或输出语句的每个程序中都必须包含这个文件。我们的程序既有输入又有输出，所以就包含了这个文件。

下一行：

```
main()
```

是每个 C++ 程序的入口，必须存在。程序体通常用左花括号和右花括号括起来，表示程序的打开和关闭。如下面的语句：

```
main()  
{  
    <程序体>  
}
```

下一行：

```
char name[20];
```

是程序所使用的一个变量的声明。char 表示这个变量是字符数据类型的。name 是变量名，长度为 20 个字节。这样，字符数据类型的变量的大小由括在方括号 [] 中的数字来表示。

下一行：

```
cout<<"My name is :";
```

调用了 cout。cout 用来向标准输出设备输出，<< 符号是输出操作符，用来输出紧接其后的变元。在这个例子中，输出一个字符串。在 iostream.h 中，cout 被定义为标准输出流。通常认为终端就是标准输出流。因此：

My name is :

就显示在屏幕上。

下一行:

```
cin>>name;
```

调用了 `cin`。`cin` 是标准输入流，也在 `iostream.h` 中定义。系统等待输入，所做的输入就存放到变量 `name` 中。

下一行:

```
cout<<name<<"\n";
```

向标准输出设备输出刚刚输入的内容。请注意该行尾的 `\n`，这是一个转义字符，其结果将导致自动回车。因此在标准输出设备上输出了 `name` 以后，光标就会自动跳到下一行。请注意，刚才输入读者的名字时，没有用转义字符。这是故意的，以使读者输入的名字紧接在提示信息的后面。

在本书后面的章节中将详细描述这个程序的所有特点。现在，我们只是使读者能够编写、编译并运行一个完整的C++程序。

程序中值得注意的事项:

- 包含文件必须存在。
- `main()` 以及其后用来打开、关闭的花括号 (`{}`) 必须存在。
- 程序中用到的所有变量都必须事先声明。
- 除了 `#include`、`main()` 和花括号之外的每条语句必须以分号作结束符。

2.12 小 结

本章讲述了如何编写、编译和运行一个程序。我们也讲述了 Windows 环境下的 borland C++ 3.1 菜单驱动系统及其 IDE 开发环境。读者现在已经知道在这个环境中如何输入程序，如何提交程序使之编译并运行。也知道了在需要时如何访问帮助信息文件。

在下一章，我们将讲述 C++ 程序所能处理的数值类型（叫做数据）以及用来操作这些数值的运算符。

第三章 数据类型、标识符和关键字

3.1 简介

由程序操纵的实际的值叫做数据 (data)，数据可以表达成多种不同的形式。数据由数字和字符组成。在 C++ 程序中，不同的数据类型以不同的方式存取和处理。这一章我们将介绍基本数据类型。

程序由单词样的单元和空白所组成。空白将单词样的单元分开。这些单词样的单元称为标识。标识与空白的组合构成了程序。本章还将描述编写 C++ 程序所用到的不同类型的标识。而关键字和操作符则是我们要着重讲述的两类专用标识。

3.2 数据类型

下面是 C++ 中的数据类型：

- 整数
- 字符型
- 浮点型

现在讨论这些数据类型的特点。

- **数据类型可以是常量或变量。**

常量数据在程序执行过程中不能改变。变量数据类型所包含的值则可按需而作改变。

C++ 有具体的表示法来表示这些不同数据的数据类型。

在 C++ 程序中，对于不同的数据类型，有的在使用之前需要声明，而有的则不需要声明。

- 常量数据类型在使用之前不必声明。
- 所有变量数据类型在使用之前必须声明。

3.3 标识符和关键字

- 标识符是在 C++ 程序中为常量、变量 (数据类型) 和其它实体选定的名字。

标识符的第一个字符必须是字母或下划线，随后的字符可以包含数字。字符可以是大写或小写形式。标识符名字所允许的长度因系统而异。我们的系统允许最长 32 个字符，而别的系统则可能有不同的长度。参阅具体的编译器的有关资料，这些资料将提供该编译器的具体规定。

- **数据类型通过关键字来声明。**

关键字是指不能由使用者自己声明的词；它们为 C++ 所保留。请注意，不能用保留字或关键字来命名标识符。

下面要解释每种数据类型，并介绍在 C++ 程序中用来声明这些数据类型的关键字。

3.4 整数

- 整数是指没有小数部分的数，可以为正，也可以为负。

正整数如 32, 45, 以及 1003 等可象我们日常生活中用到的那样来表示。0 当作正整数。负整数如 -90 和 -500, 需要在数字的前面加上短横线 (-)。

■ **关键字 integer 和 int 用来声明整数类型。**

下面是整数声明的例子:

```
int count, numbers;
```

3.4.1 短整数和长整数

为了便于理解短整数数和长整数数, 有必要讨论一下计算机如何存储数据。在计算机中任何信息都是以二进制形式存储的。二进制数只能用 0 或 1 来存储。每个 0 或 1 称作一位 (bit)。八位组成一个字节 (byte), 两个字节组成一个字 (word)。

整数能存储的最大值是由计算机给它分配的存储空间的大小决定的。而整数数所占的存储空间因不同的计算机而异。例如, 某种计算机可能用 16 位 (两个字节) 来存储一个整数数, 而另一种计算机则可能用 32 位 (四个字节) 来存储。

当一个整数存储在计算机中时, 左起第一位叫做第 0 位, 它用来表示该数是正数或负数。

■ **如果是负数, 则第 0 位被置为 1; 正数则第 0 位为 0。**

16 位计算机能存储的整数范围是 +32,767 ~ -32,768, 而 32 位计算机存储的整数范围则增加到 -2,147,483,648 ~ 2,147,483,647。

然而, 如果必须在 16 位计算机中以整数形式存储一个相当大的数, 则可以使用长整数。

■ **在整数声明前加上关键字 long, 即表示长整数。加上关键字 short, 则表示短整数数。**

下面是一个声明长整数的例子:

```
long int speed_of_light;
```

用长整数可能会降低程序的执行速度。

短整数数可用来加速程序的执行。在大多数计算机中, 存储短整数数与普通整数占用相同的空间。下面说明了如何声明短整数数。

```
short int planet_count;
```

■ **长整数和短整数声明都可以省略关键字 int, 如:**

```
long speed_of_light;
```

```
short planet_count;
```

3.4.2 无符号整数

■ **如果希望程序中的某个整数一直是正数, 则可以在该整数声明的前面加上关键字 unsigned。**

结果导致能存储的整数数的最大值增大了。第 0 位不再用作符号位。在 16 位计算机中以无符号整数形式存储时, 数据范围由 -32,768 ~ +32,767 变为 0 ~ +65,535。下面是几个例子:

```
unsigned large_number;
```

```
unsigned int speed;
unsigned long distance;
unsigned short mileage;
```

以上所有变量都是可接受的。

3.4.3 长整数和无符号常量

常量的值在整个程序运行期间不能改变。长整数常量比普通常量有着更大的变动范围。

■ 常整数常量在其后加上 l 或 L 来表示。如:

```
int 45000l;
int 50000L;
```

无符号常量象无符号整数一样进行存储。

■ 无符号整数在其后加 u 或 U 来表示。如:

```
int 60000u;
int 65535U;
```

3.5 字符型数据

■ 字符数据类型用来表示字母表 (alphabet)。

实际上，字符数据只是表示字符的数字值。这些数字值基于 ASCII (American Standard Code for Information Interchange) 标准。根据 ASCII 码标准，数值 65 代表大写字母 ‘A’，而 47 则代表小写字母 ‘a’。表 3.1 列出了全部 ASCII 码的十进制和八进制表示。您不必记住这些代码值。重要的是要理解每个 ASCII 码都与一个特定的数值相关联。

表 3.1 ASCII 代码表

十进制	八进制	符号	十进制	八进制	符号	十进制	八进制	符号
0	000	^@	43	053	+	86	126	V
1	001	^A	44	054	,	87	127	W
2	002	^B	45	055	-	88	130	X
3	003	^C	46	056	.	89	131	Y
4	004	^D	47	057	/	90	132	Z
5	005	^E	48	060	0	91	133	[
6	006	^F	49	061	1	92	134	\
7	007	^G	50	062	2	93	135	]
8	010	^H	51	063	3	94	136	^
9	011	^I	52	064	4	95	137	_
10	012	^J	53	065	5	96	140	`
11	013	^K	54	066	6	97	141	a

12	014	^L	55	067	7	98	142	b
13	015	^M	56	070	8	99	143	c
14	016	^N	57	071	9	100	144	d
15	017	^O	58	072	:	101	145	e
16	020	^P	59	073	;	102	146	f
17	021	^Q	60	074	<	103	147	g
18	022	^R	61	075	=	104	150	h
19	023	^S	62	076	>	105	151	i
20	024	^T	63	077	?	106	152	j
21	025	^U	64	100	@	107	153	k
22	026	^V	65	101	A	108	154	l
23	027	^W	66	102	B	109	155	m
24	030	^X	67	103	C	110	156	n
25	031	^Y	68	104	D	111	157	o
26	032	^Z	69	105	E	112	160	p
27	033	^[	70	106	F	113	161	q
28	034	^\	71	007	G	114	162	r
29	035	^]	72	110	H	115	163	s
30	036	^^	73	111	I	116	164	t
31	037	^_	74	112	J	117	165	u
32	040	[sp]	75	113	K	118	166	v
33	041	!	76	114	L	119	167	w
34	042	“	77	115	M	120	170	x
35	043	#	78	116	N	121	171	y
36	044	¥	79	117	O	122	172	z
37	045	%	80	120	P	123	173	{
38	046	&	81	121	Q	124	174	
39	047	‘	82	122	R	125	175	}
40	050	(	83	123	S	126	176	—
41	051	)	84	124	T	127	177	DEL
42	052	*	85	125	U			

看一下这张表，会发现有不少的特殊字符。如制表符，标点符号、控制字符（由 ^ 符号来表达）、甚至数字。其中数字不仅可以表达成数值型（通过关键字 int），还可以被规定为字符。

■ 字符数据通过关键字 char 来声明，如：

```
char string1, initial;
```

把一个字符作为数值时，要用单引号引起来，如：

```
char yes='y'; //字符常量 yes 初始化为 y
```

- 字符常量可由单引号引起来的多个字符所组成。这些常量可以是转义序列，可对于编译器具有特殊意义的字符的组合。

表 3.2 列出了可由我们的编译器识别的转义序列。

表 3.2 转义序列

转义序列	编译器动作
\a	蜂鸣
\b	回退键
\f	换页
\n	换行
\r	回车换行
\t	水平制表
\v	垂直制表
\\	反斜杠
\'	单引号
\"	双引号
\?	问号
\nnn	八进制位模式，nnn 是一个八进制数
\xnn	十六进制位模式，xnn 是一个十六进制数

用这些专用字符来向编译器表达特殊情况。其实在前面的几个程序中已遇到过 '\n' 的用法了。

- 这些专用序列都由一个或多个键码表示。然而，编译器将它们当作单一字符。

在本书后面的章节里，当我们开始学习怎样声明字符串时，这一特点将有更多意义。

可以把字符初始化成与它们的 ASCII 码等效的数值，如：

```
char A=65;
```

与下列语句的效果是一样的：

```
char tab='A';
```

3.6 浮点型

- 浮点型数据由一个整数部分、一个小数部分以及可选的后缀组成。

对该类数据可声明如下：

```
float half=0.5;
```

浮点型数据也可以包含一个指数数值，可声明如下：

```
float big-number=465.012e3;
```

一个浮点型的常量由 f 或 F 做后缀：

```
float constant_half=o.5f;
```

符点数通常以 32 位存储(假设是 16 位计算机)。

3.7 双精度数据类型

- 双精度数据类型类似于浮点型, 只是要占据两倍于浮点类型的变量或常量所要求的存储空间。

这种类型的常量也可以带有 l 或 L 做后缀, 以表示长型。

3.8 常量数据类型

- 修饰符 const 可以优先于任何数据类型的声明, 以指明这一变量的值不能被改变。这样, const 可用来做为规定一个常量的另一种方法。

下面是一个 const 声明:

```
const ten 10; // ten 永远等于 10
```

3.9 识别大小写的能力

- C++标识符是区分大小写的。

如下三个变量的声明:

```
int variable;  
int Variable;  
int VARIABLE;
```

说明了三个不同的变量。

下面用一个小例子来说明我们到此为止所讨论的内容。

```
//TEXT3.1.CPP  
//iostream.h 是 cout 语句所必需的  
#include <iostream.h>  
main()  
{  
    int float; //错误 --用关键字作名字非法  
    int sum; //合法的标识符名字  
    int Sum; //合法的标识符名字  
    float=2;  
    Sum=0;  
    sum=5;  
    Sum=sum+float;  
    cout << "Sum is " << Sum<<"\n";  
}
```

编译这段程序将得到几条错误信息。请注意我们只列出那些与这段程序有关的信息, 本书的其余部分也将保持这种惯例。通常, 其它错误只是针对实际问题的, 或者是因

为在实际问题中对于其上下文的含义是不正确的。下面是本程序显示的信息:

ERROR: Too many types in declaration

这是因为 float 是一个关键字, 不能用来作为标识符或保留字的名字。这一问题可通过把 float 改成不是关键字或保留字的名字, 象 var1 这样的就可以。下面是 text3-1.cpp 的重写:

```
//TEXT3-1.CPP
//iostream.h 对于 cout 是必需的
#include <iostream.h>
main()
{
    int var1; //合法的标识符名字
    int sum;  //合法的标识符名字
    int Sum;  //合法的标识符名字
    var1=2;
    Sum=0;
    sum=5;
    Sum=sum+var1;
    cout << "Sum is " << Sum<<"\n";
}
```

编译并执行上述代码得到正确输出结果:

Sum is 7

3.10 保留的关键字

下列单词不能用来作为标识符, 因为它们仅为 C++编译器保留使用:

asm	default	float	new	switch
auto	delete	for	operator	template
break	do	friend	overload	this
case	double	goto	private	typedef
catch	else	if	protected	union
char	enum	inline	public	unsigned
class	extern	int	register	virtual
const		long	return	volatile
continue			short	while
			sizeof	
			static	
			struct	

3.11 小 结

本章我们学会了如何声明基本数据类型,并且简短地介绍了这些数据类型在计算机中如何存储。表 3.3 总结了这些数据类型在 16 位计算机上的存储需求和数值范围。请记住存储规定会因不同的计算机而异。如果在一台 IBM 或 IBM 兼容 PC 机上工作,那么存储规定将与此大致相同。

表 3 · 3 数据类型及存储需求

数据类型	在 C++中声明	存储	允许值
整数	int	16 位	-32,768 至 +32,767
短整数	short int	16 位	-32,768 至 +32,767
长整数	long int	32 位	-2,147,483,648 至 +2,147,483,647
无符号整数	unsigned int	16 位	0 至 65,535
长整数常量	加后缀 l 或 U	32 位	-2,147,483,648 至 +2,147,483,647
无符号整形常量	加后缀 u 或 U	16 位	0 至 65,535
字符型	char	8 位	ASCII 码
浮点型	float	32 位	3.4×10^{-38} 至 $3.4 \times 10^{+38}$
双精度	double	64 位	1.7×10^{-308} 至 $1.7 \times 10^{+308}$
双精度长型	加后缀 l 或 L	80 位	3.4×10^{-4932} 至 $1.1 \times 10^{+4932}$

第四章 存储分类符和作用域

4.1 简介

上一章我们学会了 C++ 中的不同数据类型。在本章里，我们将讨论这些数据类型的存储分类符 (storage classes)。

- 存储分类符定义了变量在内存中的位置以及程序执行过程中变量的作用时间即生命周期。

与存储分类符紧密相关的是作用域 (scope)。

- 作用域是指程序中变量能够活动 (active) 的程序段。在此程序段中，变量可被存取。
- 变量的存储分类符可以通过声明来显式地 (explicitly) 定义，也可以通过上下文隐式地 (implicitly) 定义。

缺省时，存储分类符通过上下文声明，下面是C++的存储分类符：

```
automatic
static
external
register
```

- 存储分类符可以缺省。

如果一个变量的存储分类符未声明就意味着编译器认为它是通过上下文来声明的缺省的存储分类符。本章后面几节将详细讲述各种存储分类符。

4.2 自动变量声明

- 关键字auto用来声明自动变量。被声明的变量可以省略标识符auto。请看下列代码：

```
auto int a; //自动变量
char b; //自动变量(缺省形式)
```

- C++中，自动变量的作用域（变量的值在程序内部的活动范围）仅仅是该变量所在的程序块或该程序块内部的程序块。

下面是说明自动变量的作用域的例子：

```
//test4-1.cpp
#include <iostream.h>
main( )
{
    //i 和 j 是缺省的自动变量
    int i;
    int j;
```

```
cout<<" i in outer block is"<<i<<"\n";
cout<<" j in outer block is"<<j<<"\n";
cout<<" k in outer block is"<<k<<"\n";
{ //内部的程序块
    int k; //缺省的自动变量
    cout<<" i in inner block is"<<i<<"\n";
    cout<<" j in inner block is"<<j<<"\n";
    cout<<" k in inner block is"<<k<<"\n";
}
//外部的程序块
cout<<" i back in outer block is"<<i<<"\n";
cout<<" j back in outer block is"<<j<<"\n";
cout<<" k back in outer block is"<<k<<"\n";
} // main() 结束
```

编译这个程序会得到如下错误信息:

```
Error: Undefined symbol 'k' in function main()
```

```
Error: Undefined symbol 'k' in function main()
```

这是因为变量 `k` 缺省为自动变量, 它在内部的程序块中声明, 所以只能在内部的程序块中存在或存取。在外部的程序块中, 编译器不知道 `k` 的值 (在外部的程序块中, 超出了 `k` 的作用域)。因此产生了错误信息。这个错误很容易改正, 只要删除在内部的程序块中对 `k` 的声明, 而在外部的程序块中声明 `k` 即可。修改之后, 重新编译运行该程序会得到如下输出结果:

```
i in outer block is 8886
j in outer block is 18
k in outer block is 0
i in inner block is 8886
j in inner block is 18
k in inner block is 0
i back in outer block is 8886
j back in outer block is 18
k back in outer block is 0
```

出现在变量 i、j、k 中的值 8886、18、0 都是从哪里来的呢？在 C++ 中，自动变量如果没在它本身的作用域内初始化，那么该变量就可能包含随机的无用的信息值。假设变量 i、j、k 的初始声明修改如下：

```
.  
.br/>int i=1;  
int j=2;  
int k=3;
```

这时输出会变为：

```
i in outer block is 1  
j in outer block is 2  
k in outer block is 3  
i in inner block is 1  
j in inner block is 2  
k in inner block is 3  
i back in outer block is 1  
j back in outer block is 2  
k back in outer block is 3
```

4.3 静态变量声明

- 可在变量声明语句前面加上关键字 `static` 来表示静态变量。

静态变量象自动变量一样，也是局部变量。只在它所处的函数、程序块或子程序块内部有效，然而与自动变量有一个重大区别：

- 当静态变量离开作用域时，它的值仍然保持不变或者说它具有有粘性。

当程序又回到静态变量所在的函数时，程序处理的是上一次存储在静态变量中的值。

- 静态变量在没有明确的初始值时，会自动初始化为 0 或空（NULL）。

请看下面的程序：

```
//test4_2.cpp  
#include <iostream.h>  
  
//下面是函数原型  
void sum(void);  
  
main( )  
{
```

```
// j 声明为静态变量
static int j;
sum();
cout<< "Inside main(), j is " << j << "\n";
sum();
cout<< "Inside main(), j is " << j << "\n";
}
void num(void)
{
    static int j;
    j = j+1;
    cout<< "Inside sum() j is %d" << j << "\n";
}
```

编译运行该程序会得到如下输出结果:

```
Inside sum() j is 1
Inside main(), j is 0
Inside sum() j is 2
Inside main(), j is 0
```

j 在主函数 main() 和 sum() 中均被声明为一个静态变量。在 sum() 中, j 被加 1, 然后进程返回到主函数 main() 中, 我们看到这时的 j 是 0, 这是因为在主函数 main() 中的变量 j 已经超出了函数 sum() 的作用域, 静态变量被赋值为 0。因此, 在主函数 main() 中 0 作为 j 的值输出; 接着, 控制又回到 sum(), 这时 j 保持着原先的值 1 (因为 j 是静态变量), 现在 j 变为 2, 但在主函数 main() 中 j 的值仍然是 0。

在 C++ 中, 类 (class) 的静态成员也有与之相联系的特殊规则。当我们讲到类时再详细讨论。

4.4 外部变量声明

- 外部变量不同于静态变量和自动变量, 它的作用域是全局的而不是局部的。

外部变量在函数的内外都能存在, 并且对于同一文件内的所有函数均可用 (这叫做文件作用域, File Scope)。

- 外部变量的值象静态变量一样可以保持。
- 在没有明确的初始值时, 外部变量也会自动初始化为 0 或空 (NULL)。

请看下面的程序:

```
//test4-3.cpp
```

```
#include <iostream.h>

//声明 i 是全程外部变量
int i;
//下面是函数原型
void function-1(void);
void function-2(void);
main( )
{
    cout<< "i inside main() is" << I <<"\n";
    function-1();
}

void function-1(void)
{
    i = i+1;
    cout<< "i inside function-1() is" << I <<"\n";
    function-2();
}

void function-2(void)
{
    i = i+1;
    cout<< "i inside function-2() is" << I <<"\n";
}
```

编译运行该程序得到如下输出结果:

```
i inside main() is 0
i inside function-1() is 1
i inside function-2() is 2
```

在 main() 中 i 的输出值是 0, 因为 i 没有明确地初始化; 在函数 function-1() 中, i 的输出值是 1。i 在函数 function-1() 中不必声明, 因为 i 已经在主函数 main() 的外部声明为全局变量了。在函数 function-2() 中, i 的输出值又为 2, 因为函数 function-1() 中得到的 i 的值被保持下来了。

在函数内使用的外部变量可以用 extern 来声明。下面是对原始程序的修改, 编译器一样会接受, 输出结果不变。

```
void function-1(void)
{
    extern int i;
    .
    .
    .
}
```

```
void function-2(void)
{
    extern int i;
    .
    .
    .
}
```

必须遵守的规则是，一定要在引用外部变量的函数的外部和前面进行 `extern` 声明，并且必须放在同一个源代码文件内。对于下面的程序，编译器会产生错误信息：

```
//test4_4.cpp
#include <iostream.h>

//i声明为外部变量
int i;
main( )
{
    i=5;
    cout<<"i is"<<i<<"\n";
    cout<<"j is"<<j<<"\n";
}
//j在被引用之后声明
int j;
```

Error: Undefined symbol 'j' in function main()

解决这一问题只需把 `j` 的声明语句挪到 `i` 的声明语句的后面。另外一种修改的办法是保留程序最初的代码不变，在主函数 `main()` 中，用 `extern` 来声明 `i` 和 `j`。

```
.  
.   
int i;  
.   
.   
main( )  
{  
    extern int i;  
    extern int j;  
    .  
    .  
}  
int j;
```

编译运行该程序会得到如下输出结果:

```
i is 5  
j is 0
```

在后面的章节里,我们将会学到在 C++ 怎样对数据私有这个概念进行应用。我们建议读者只在绝对必要时才使用外部变量。因为其作用域是全程的,在整个程序运行期间都占用存储空间,因此就失去了数据的私有性。在一个相当长的程序中,对于不同时刻,不同函数中存储在这些全局变量中的值都很难跟踪。

4.5 寄存器变量声明

寄存器变量类似于自动变量,它也是局部变量,只在其作用域内有效。但是当变量声明前面加上关键字 `register` 时,就意味着请求编译器在有可用的寄存器时将这一变量存储在寄存器中。

- 寄存器变量的使用能有利于缩减程序的大小,改进程序的性能。因为对寄存器操作比对内存中的变量操作要快。

下面是合法的寄存器变量声明:

```
register int a; //寄存器变量声明  
register char b; //寄存器变量声明
```

4.6 变量期间

- 变量期间是指变量被分配存储空间的那段时间。

按照前面的说法

- 静态变量和外部变量在整个程序期间内分配内存。
 - 自动变量和寄存器变量只在它们所属的程序块或函数内部分配内存。
 - 可以通过使用 new 操作符和 delete 操作符来给变量动态地分配内存。
- 这些操作符将在以后的章节中讲到。我们现在提到它们是为了让您知道有这些操作。

4.7 小 结

本章我们讲述了存储分类符。表 4.1 说明了讲过的主要概念：

表 4.1 C++中的存储分类符

存储分类符	作用域	存储器分配	编译器默认初始值
auto	在声明它的程序块，子程序块或函数内部有效，值不保持	作用域内	无
static	在声明它的程序块，子程序块或函数内部有效，值保持	整个程序期间	0
extern	只要在所有的函数外部或上面声明，对于同一文件的所有函数全局可用，值保持	整个程序期间	0
register	在声明它的程序块，子程序块或函数内部有效，值不保持	作用域内	无

第五章 运算符、优先级和结合性

5.1 简介

■ 运算符是表示对数值进行一种运算的符号。

运算符可对操作数进行运算。操作数是一个数值，既可以是常量也可以是变量。

C++ 包含一个丰富的运算符集。有些运算符带有一个操作数，有些则带有两个操作数；有些运算符本身是专用的，而有些则是由其它运算符派生出来的；有些运算符甚至是重载 (Overloaded) 的。这意味着相同的运算符可能表示不只一种运算。编译器能从使用该操作的上下文关联中来理解要执行的恰当的运算。

很难找到一个程序能将所有的运算符都用上。本章我们只讨论算术运算符、赋值运算符、取模运算符、增量运算符和减量运算符等最基本的运算符。几乎所有的 C++ 程序都使用这些运算符。更特殊的运算符将要在后面的章节中讨论。

■ 优先级指在程序中运算执行的先后顺序。

一条 C++ 语句可能包含多个运算符，对它们所涉及的数据类型执行多重操作。而这些运算必须按一定的先后顺序来执行，否则同一语句就会产生不同的结果。正象使用括号来指定算术运算中的运算符优先级（括号内的运算在括号外的运算之前执行）一样，C++ 语句中执行的运算优先级由运算符本身指定。本章将讲述运算执行时的优先级顺序。

5.2 算术运算符、赋值运算符和取模运算符

其实在前面已经遇到过算术运算符和赋值运算符了。下面是一条使用加法 (+) 运算符的简单语句：

```
a=5+5;
```

该语句中有三个操作数：a，5 和 5，第一个操作数是一个变量，其余两个则是常量，最左边的操作数存放运算的结果。这样，a 就得到 5 和 5 作加法运算的结果，即 10，加法运算符用于将左右两边的操作数加起来。

下面是一段使用其余的数学运算符的简短程序：

```
//test5-1.cpp
#include <iostream.h>
main()
{
    int a,b,c,d;
    a=5+3; //加法运算符
    b=a-2; //减法运算符
    c=a+b; //加法运算符
    d=c/2; //除法运算符
    e=d*4; //乘法运算符
```

```
f=e%5; //模除运算符
cout<<"a is"<<a<<"\n"
cout<<"b is"<<b<<"\n"
cout<<"c is" <<c<<"\n"
cout<<"d is"<<d<<"\n"
cout<<"e is"<<e<<"\n"
cout<<"f is"<<f<<"\n"
}
```

编译并运行这段程序将得到如下输出结果:

```
a is 8
b is 6
c is 14
d is 7
e is 28
f is 3
```

大概除了最后一条输出结果外,都用不着解释。

■ 取模运算符用来得到除法的余数而不是商。

e 的内容(即 28)被 5 除。商是 5, 丢弃不用, 余数是 3, 余数放到变量 f 中。

5.3 增量和减量运算符

语句:

```
a=a+1;
```

也可以这样表达:

```
a++;
```

a++ 利用了增量运算符(++)。这两条语句的结果是一样的, 都是将 a 加 1。

同样, 语句

```
b=b-1;
```

也可以这样表达:

```
b--;
```

b--利用了减量运算符(--)。

在上面的例子中, 增量和减量运算符都放在后缀(即跟在操作数后面)的位置上。这些运算符也可以放在前缀位置上, 如:

```
++a;
```

```
--b;
```

前后缀运算的效果是不同的。如果用另外一个变量来存放运算结果, 这两种形式的差异就更容易理解了。假定 a 中存放 10, 语句:

```
c=a++;
```

将 10 存放到 c 中，然后 a 增加 1，最终结果将如下：

变量	内容
c	10
a	11

另一方面，语句

`c=++a;`

导致 a 先增加 1，然后将 a 的新值存放到 c 中，两个变量变成这样：

变量	内容
c	11
a	11

现在我们将前后两种语句用减量运算符来修改，假定这时 b 中存有 20：

`c=b--;`

结果就是：

变量	内容
c	20
b	19

用前缀形式：

`c=--b;`

得到如下结果：

变量	内容
c	19
b	19

5.4 复合运算符

按照前一节的解释，表达式：

`b=b+1;`

能表示成：

`b=b++;`

C++ 也允许我们使用下面的与之等效的表达式：

`b+=1;`

句法是这样的：

变量 运算符=表达式

这种句法类型可以适用于下列运算符：

=
+=
*=
-=
/=

%=

因此，表达式

```
a=a*5;
```

也可以表示为

```
a*=5;
```

该句法也适用于位运算符，这将在本章后面详细讲述。这些运算符是：

```
>>=
```

```
<<=
```

```
&=
```

```
^=
```

```
|=
```

5.5 逻辑与运算符

■ 逻辑与运算符用符号&&来表示。

■ 逻辑与运算符用来执行一个或多个条件的逻辑组合。

如果要对两个或多个条件进行求值，就用&&将它们结合起来。

例如，假定要对下列情况编码：

如果年龄小于或等于 55 岁，身体健康，则准予办理人寿保险。

如果年龄小于或等于 55 岁，身体不健康，则不予办理人寿保险。

如果年龄大于 55 岁，则不予办理人寿保险。

这些条件可以用 IF 语句来编码。（将在下一章详细讲述 IF 语句）。用逻辑与运算符来组合这些条件，得出需要的结果。下面是实现上述功能的程序：

（再者，如果您没能立即学会 IF 语句的全部语法，用不着担心，因为此种语句相当直观，我想您无论如何都能理解它的含义。）

```
//test5-2.cpp
#include <iostream.h>
main()
{
    char life_insurance[1];
    char health[1];
    int age=50;
    health='B';
    if(age<=55 && health == 'G')
        life_insurance='Y';
    if(age<=55 && health == 'B')
        life_insurance='N';
    if(age>55)
```

```
    life_insurance='N';  
    cout<<"Life insurance issued: "<<life_insurance<<"\n";  
}
```

程序输出:

Life insurance issued: N

本程序中, 声明了必要的变量, 然后把 health 初始化为 B, 表示 Bad; IF 语句执行了三次。第一条 IF 语句, 检查是否年龄小于或等于 55 岁, 并且身体健康。如果是的话, 则将 life_insurance 赋值为 Y(Yes); 第二条 IF 语句表示如果年龄小于或等于 55 岁, 并且身体不健康的话, 则将 life_insurance 赋值为 N(No), 现在您就应该能猜出第三条 IF 语句的含义。

请注意:

- && 运算符用来将两个条件组合在一起。
- && 运算符左边和右边的条件都为真时, 整个表达式的值才为真(记住, 一个非 0 值表示真, 0 表示假)。

5.6 逻辑或运算符

逻辑或运算符用符号 || 来表示。

|| 对它所结合的表达式求值, 同与运算相似。不同之处是:

- 任一条件为真, 表达式的值就为真, 条件都不为真时, 结果同与运算符。
- 前一节的程序用 || 运算符重写为:

```
//test5_3.cpp  
#include <iostream.h>  
  
main()  
{  
    char life_insurance[1];  
    char health[1];  
    int age=50;  
    health='B';  
    if(age<=55 && health == 'G')  
        life_insurance='Y';  
    if(age<=55 && health == 'B' || (age>55))  
        life_insurance='N';  
    cout<<"Life insurance issued: "<<life_insurance<<"\n";  
}
```

程序输出为:

life insurance issued: N

注意 || 运算符用来结合 IF 语句中的第二个条件和第三个条件, 如果这两个条件中有一个是真的 (|| 运算符左边的或是右边的), 整个表达式的值就为真。

5.7 位运算符

您已经知道, 八位构成一个字节, 两个字节构成一个字。16 位计算机通常用一个字节存储一个字符, 用一个字存储一个整数, 而浮点数则需要用两个字节来存储等等。无论在什么情况下, 数据总是用 0 和 1 的组合来存储的。这就是数据的机内表示。有时按位来操作数据是很必要的, 也就是说, 程序员可能希望通过改变存储在位、字节或字中的 0 和 1 来改变它们的值。这就是位操作的由来。

5.7.1 左移位和右移位运算符

其实您已经遇到过放入 (>>) 和取出 (<<) 运算符的用法了。

- <<运算符也可用来将一个字中的数据位左移指定的位数。
- >>运算符也可用来将一个字中的数据位右移指定的位数。

下面是说明其用法的程序:

```
//test5-4.cpp
#include <iostream.h>
main()
{
    int age=6;
    int new_age;
    new_age=age<<4;
    cout<<" new_age is: "<<new_age<<"\n";
}
```

程序输出:

age is 96

在程序开始, age 被初始化为 6, 请看另一条语句:

age<<4;

怎样将 age 中存放的数据左移四位呢? 在机器内部, 6 被表示为:

0000 0000 0000 0110

左移四位即:

0000 0000 0110

空位由 0 来填充, 即

0000 0000 0110 0000

这个数的结果是 96。

右移位运算符的操作很象左移位运算符。当然它是向右移位, 下面是根据 test5-4.cpp 改写的使用右移位运算符的程序:

```
//test5-5.cpp
#include <iostream.h>
main()
{
    int age=6;
    int new_age;
    new_age=age>>4;
    cout<<" new_age is: "<<new_age<<"\n";
}
```

再看一下数字 6 的机内表示:

0000 0000 0000 0110

被右移四位后:

... 0000 0000 0000

空位由 0 填充, 请注意, 只有需要右移位的变量是无符号数时(即总是正数), 用 0 作填充位才能得到保证。概括地说,

- 左移位运算符 (<<) 用来将数据位左移。
- 空白位用 0 填充。
- 右移位运算符 (>>) 用来将数据位右移。
- 空白位可能或不能用 0 填充, 只有当需要右移的变量是无符号数时, 才用 0 填充。有时用符号位来填充空白位。
- 如果位运算符右边的表达式是负数, 或者大于左边的表达式, 位操作的结果是无法预言的。

5.7.2 按位与运算符

- 按位与运算符用来得到两个整型操作数的逻辑乘积。
- 用符号 & 表示的按位与运算符来将两位“与”在一起。
- 如果被“与”的两位都是 1, 按位与的结果为 1, 否则结果为 0。

即

0 & 0 是 0

0 & 1 是 0

1 & 0 是 0

1 & 1 是 1

下面是一个具体的例子, 将 128 和 -97 按位与:

0000 0000 1000 0000

```
& 1111 1111 1001 1111
```

```
0000 0000 1000 0000
```

这样，128 与 -97 按位与产生的结果为 128。

这个例子用 C++ 表示如下：

```
//test5-6.cpp
#include <iostream.h>
main()
{
    int a=128;
    int b=-97;
    int c;
    c=a&b;
    cout<<" c is: "<<c<<"\n";
}
```

程序的输出结果是：

```
c is 128
```

5.7.3 按位或运算符

- 按位或运算符用来得到两个整型操作数的逻辑和。
- 用符号 | 表示两位的“与”。
- 如果两位都是 0，则按位或的结果为 0，否则结果为 1。

即：

0 | 0 是 0

0 | 1 是 1

1 | 0 是 1

1 | 1 是 1

下面我们用按位或运算符来运算前一节中出现的例子。

```
0000 0000 1000 0000
| 1111 1111 1001 1111
```

```
1111 1111 1001 1111
```

下面是 C++ 中的例子：

```
//test5.7.cpp
```

```
#include <iostream.h>
main()
{
    int a=128;
    int b=-97;
    int c;
    c=a | b;
    cout<<" c is: "<<c<<"\n";
}
```

输出为:

c is -97

5.7.4 按位异或运算符

- 按位异或运算符用来执行异或操作，通过 ^ 运算符来进行两位之间的异或。
在异或操作中:

0 ^ 0 是 0

0 ^ 1 是 1

1 ^ 0 是 1

1 ^ 1 是 0

下面对前面两节中的例子采用异或操作:

```
0000 0000 1000 0000
^ 1111 1111 1001 1111
```

```
1111 1111 0001 1111
```

用 C++ 程序表示:

```
//test5.8.cpp
#include <iostream.h>
main()
{
    int a=128;
    int b=-97;
    int c;
    c=a ^ b;
    cout<<" c is: "<<c<<"\n";
}
```

输出为:

c is -225

5.8 new 和 delete 运算符

new 和 delete 将在本书后面关于构造函数和析构函数的章节中作更详细的讲述。为了考虑其完整性，在这里只作简单介绍。

■ new 运算符用来从自由存储空间即堆 (heap) 中给不同的数据类型动态地分配内存。

自由存储空间是在程序运行时提供给程序的尚未分配的存储池。在用 delete 运算符释放存储空间之前，为变量分配连续的存储空间。

■ delete 运算符用来释放在 new 运算符分配的存储空间。

确切地说，delete 操作必须要带上一个由 new 操作返回的指针。指针这个概念还没讲过。现在，暂且把指针当作一个数值，这个数值代表内存中存储变量的特定位置。

如果分配成功，new 运算符返回指向内存位置的指针。如果分配不成功，则返回一个空指针 (null)。delete 运算符根据同样的指针来释放它所请求的内存空间。

并不强制使用 delete，因为程序结束时显然会自动释放内存。

5.9 sizeof 运算符

■ sizeof 运算符用来返回操作数所占存储空间大小。以字节为单位返回。

例如下面的语句：

```
char a[1];
sizeof(a);
```

将返回 1，这是 a 的大小。同样，下面的语句：

```
int j;
sizeof(j);
```

将返回 2，因为整数存储占用 2 个字节。

■ sizeof 用来返回任何有效数据类型的大小。

5.10 条件(即三元)运算符

■ 条件运算符是 if-else 语句的变体。

本运算符不常用，因为对 if-else 语句使用和理解起来要更直观。虽然 if-else 结构还没讲过 (后面的第六章将详细讲述，看了这里的伪代码之后，就会懂得它的含义了)。出于完整性的缘故，本章对这个运算符作些解释。

语法是这样的：

```
expr1 ? expr2 : expr3
```

它按如下步骤求值：如果 expr1 (表达式 1) 是真，返回值是 expr2；如果 expr1 是假，返回值是 expr3。

同样的条件用 if-else 结构表示如下：

```
if (expr1 is true)
    do expr2
else
    do expr3
```

下面是 C++ 中使用该运算符的典型语句:

```
a=b ? 2 : 4;
```

该语句可以这样解释:

如果 *b* 为真(即等于一个非 0 值), *a* 赋值为 2; 如果 *b* 为假(即等于 0), 则 *a* 赋值为 4, 然后将 *a* 赋值为 *b*。

5.11 优先级与结合性

- 优先级是指语句中运算符求值的顺序。
- 结合性是指在同一语句内相同类型的多个运算符的求值顺序。
- C++ 具有一个内置的规则集来支配运算符求值的顺序。

表 5.1 列出了 C++ 运算符的结合性和优先级。其中大多数运算符已经解释过了, 其余的将在以后的章节中解释。

- 同一行中的每组运算符的优先级顺序是由左向右的。

这样, `()` 和 `[]` 就具有相同的优先级, 这些运算符的优先级比下面各行的运算符优先级要高。如 `!` 和 `-`。但是, 如果 `()` 和 `[]` 都在同一条件语句中相遇, 那么 `()` 要在 `[]` 之前求值。

现在您是否会有这样的疑问: 如果同一运算符在同一语句中出现多次, 优先级会是什么样的呢? 请看下面的例子。

表 5.1 C++ 运算符的结合性和优先级

运算符	结合性
<code>() [] -> :: .</code>	由左向右
<code>! ~ - ++ -- & *</code>	由右向左
<code>sizeof new delete . * -> / %</code>	由左向右
<code>+ -</code>	由左向右
<code><< >></code>	由左向右
<code>< <= > >=</code>	由左向右
<code>== !=</code>	由左向右
<code>&</code>	由左向右
<code>^</code>	由左向右
<code> </code>	由左向右
<code>&&</code>	由左向右
<code> </code>	由左向右
<code>? conditional operator</code>	由右向左
<code>= += /= %= += -=</code>	由右向左
<code>&= ^= =</code>	由左向右

```
a=b+5+1;
```

是 *b* 先赋值为 5, 然后 *a* 再赋值为 *b*+1, 即 6 呢? 还是先把 6 这个值赋值给 *b* 然

后再把它放入 a 呢？在第一种情况下，b 里面存放的是 5，而 a 里存放的则是 6。您可能会感到惊奇：

■ C++ 并没有一个规则集来解决同一语句内同一运算符多次出现时的优先级的问題。

在这些情况下，总是使用括号来规定您想要的顺序，用这种办法就可避免无法预言的结果。这样，要表达 b 赋值为 5，a 赋值为 6 的结果就有下列代码：

```
a=(b=5)+1
```

第二种情况下的括号位置就变化如下：

```
a=b=(5+1);
```

5.12 小 结

本章我们讨论了运算符和优先级的结合性。优先级是指运算符在程序中的执行顺序。而结合性则是指同一语句内多个同类运算符的求值顺序。我们描述了算术运算符，赋值运算符，取模运算符，增量运算符，减量运算符，逻辑与和逻辑或运算符，按位运算符，new 运算符，delete 运算符，sizeof 运算符，以及条件运算符。同时我们也讲述了复合运算符的作用。

第六章 控制结构

6.1 简介

如果读者懂得了上一章的内容，那么就应该读一读本章了；如果还没学懂第五章，在继续学习本章之前，要把第五章再重读一遍。

我们刚才所说的就是一个条件逻辑的例子：用了一个 IF 语句。在一个计算机程序中要使用各种条件逻辑：如果条件为真，则做这个，否则就做那个。

■ 控制结构允许我们指定程序的逻辑即控制流程。

本章我们将讲述下列控制结构：

- if 和 if-else 语句
- while 语句
- do-while 语句
- for 语句
- switch-case 结构

6.2 IF 和 IF-ELSE语句

IF 语句是以最简单形式是按条件执行的。可以把这一逻辑翻译成下列伪代码：

```
if (this condition is true)
{
    do this ;
    and this ;
}
```

注意括号中要被检测的条件，如果条件为真，则执行花括号内的语句。如果只执行一条语句，就不需要花括号了。语句通常以分号结束。

如果条件不为真会怎样呢？将else 语句附加到 IF 语句上，即：

```
if (this condition is true)
{
    do this ;
    and this ;
}
else
{
    do this ;
    and this ;
}
```

我们怎么表述一个条件？这个条件又如何求出真值或假值呢？

■ 使用 C++ 的条件运算符来表达条件。

下面就列出条件列表及其含义。

表 6.1 C++的条件运算符

运算符	含义
==	左边的操作数等于右边的操作数
!=	左边的操作数不等于右边的操作数
<	左边的操作数小于右边的操作数
>	左边的操作数大于右边的操作数
<=	左边的操作数小于或等于右边的操作数
>=	左边的操作数大于或等于右边的操作数

下面是一个用伪代码表示的例子。

```
if (money>600)
    print " I'm loaded "
else
    print " I'm not loaded "
```

下面是表示这个例子的一段C++程序:

```
//test6-1.cpp
#include <iostream.h>
main( )
{
    int money=700;

    if (money>600)
        cout<< "I'm loaded\n";
    else
        cout<< "I'm not loaded\n";
}
```

编译并运行这段程序得到如下输出结果:

```
I'm loaded
```

这是因为在程序的开始 money 的值就初始化为700了。下面是执行多重语句的相似程序:

```
//test6-2.cpp
#include <iostream.h>
main( )
```

```
{
    int money=700;

    if (money>600)
    {
        cout<< "Pay gas bill";
        money=money-40;
        cout<< "Pay electric bill";
        money -=65;
        cout<< "Pay Visa account";
        money -=145;
        if (money>=600)
            cout<< "I'm still loaded\n";
        else
            cout<< "I'm not loaded anymore\n";
    }
    else
        cout<< "I'm not loaded\n";
}
```

编译并运行这个程序得到如下输出结果:

I'm not loaded anymore

这个例子说明了以下几种特性的用法:

- 我们在此语句中用习惯性的句法来使 money 的值减少:

money=money-40;

而简化的句法则是这样的:

money -=65;

- 我们使用花括号来括住 if 条件后面的一组语句。

```
if (money>600)
{
    cout<< "Pay gas bill";
    .
    .
    .
    cout<< "I'm not loaded anymore\n";
}
```

- 第二级的 if-else 控制结构包含在第一级中

```
if (money>600)
{
    .
    .
    .
    if (money>=600)
        cout<< "I'm still loaded\n";
    else
        cout<< "I'm not loaded anymore\n";
}
.
.
.
```

如果第二级的 if-else 控制结构有多条语句，也要括在第二组花括号中。读者可以将每个控制结构内部括起来。想括几个括几个，只是要保证每个结构都有一个逻辑上的开始和结束。

6.3 WHILE语句

■ While 语句在所引用的条件为真时，重复地执行语句。

下面是句法：

```
while(this condition is true)
{
    do this ;
    and this ;
}
```

与 if 语句应用的规则是相同的，while 条件也要通过关系运算符来表达。被测试的条件要用括号括起来；多条语句必须用花括号括起来；每个语句必须以分号结尾。

我们将上一节的例子用 While 语句代替 if 语句来修改一下，首先，用伪代码表示如下：

```
if (money>=700)
{
    while (money>200)
    {
        pay bill;
        reduce amount of money by bill;
    }
}
```

对于这个例子最好的编码方法是使用一个数组 (array), 数组中的每个元素代表已付款的账单。然而, 因为我们还没有进行到讲解数组的章节, 所以我们把每张应付帐单的钱数设定为一个常量, 如 ¥100, 程序如下:

```
//test6-3.cpp
#include <iostream.h>
main( )
{
    int money=700;

    if(money>=600)
    {
        while(money>200)
        {
            cout<<"Pay bill";
            money -= 100;
        }
        cout<<"Money left is : "<<money<<"\n";
    }
}
```

输出结果如下:

```
Pay bill
Pay bill
Pay bill
Pay bill
Money left is :200
```

注意:

- while 语句包括在 if 语句中。
- while 中的语句执行了四次, 直到被测试的条件为假。
共四次从700 中减去100, 结果 money 赋值为 200。在第五次循环中, while 条件得出假值, 这一点可由 if 语句中包含的 cout 语句执行的结果来表明。
我们由此得出非常重要的一点:
- 如果编写一个 while 语句, 切记一定要将某种测试结果能改变的条件包括进去。
否则 while 循环将永远执行下去。因为测试条件永不改变。

6.4 DO-WHILE语句

- do-while 语句是 while 语句的变种（尽管 do-while 语句并不那么流行，但大多数 C++ 程序员坚持用它）。

下面是句法：

```
do{  
    this;  
    and this;  
}  
while(this condition is true);
```

while 语句和 do-while 语句的唯一区别在于测试条件的位置不同。while 语句在开始处检测条件，而 do-while 语句则在执行完条件体后检测条件。这就说明：

- 如果在开始处条件测试为假，那么 while 语句将不能被执行；相反，do-while 语句总是至少执行一次。

do-while 循环的用法举例说明如下：

```
//test6_4.cpp  
#include <iostream.h>  
main()  
{  
    int money=700;  
  
    do{  
        cout<<"Pay bill";  
        money -= 100;  
    }  
    while(money>700)  
    cout<<"Money left is : "<<money<<"\n";  
}
```

输出结果是：

```
Pay bill  
Money left is: 600
```

注意：

- 在 if 循环的开始，money 的总数是 700。
- do-while 循环一执行，就从 700 中减去 100，现在 money 中存的是 600。
- do-while 循环的 while 条件失败。
再看看用 while 循环重写该程序会怎样。

```
//test6-5.cpp
#include <iostream.h>
main( )
{
    int money=700;

    while(money>700)
    {
        cout<<"Pay bill";
        money -= 100;
    }
    cout<<"Money left is : "<<money<<"\n";
}
```

输出结果是:

Money left is: 700

现在读者应该清楚这两种循环的区别了吧。

6.5 FOR语句

■ for 循环包含一个初始化部分，一个条件测试部分以及一个条件改变部分。

这大概是最流行的循环结构之一。for语句的句法看起来是这样的:

for (初始化计数器; 执行条件测试; 条件改变计数器)

```
{
    this;
    and this;
}
```

下面是个用英语表达的简单的例子:

While money is greater than 200, pay the bill.

下面是其伪代码表示, 用了for 循环:

```
for (money=700; money>200; money -=100)
{
    pay bill;
}
```

下面是等效的C++程序:

```
//test6-6.cpp
#include <iostream.h>
main( )
```

```
{
    for (money=700; money>200; money -=100)
    {
        cout<<"Pay bill";
    }
    cout<<"Money left is : "<<money<<"\n";
}
```

相应的输出结果是:

```
Pay bill
Pay bill
Pay bill
Pay bill
Money left is: 200
```

同 test6-3.cpp 中的 while 循环比较一下:

```
//test6-3.cpp
#include <iostream.h>
main()
{
    int money=700;

    if (money>=600)
    {
        while (money>200)
        {
            cout<<"pay bill";
            money -= 100;
        }
        cout<<"Money left is : "<<money<<"\n";
    }
}
```

下面是关于 test6-6.cpp 的解释:

■ 初始化部分在 for 语句自身内部执行。

for (money=700;)

■ 条件测试也在 for 循环内执行。

```
for ( ...; money>200; ... )
```

■ 改变条件的语句也是如此。

```
for ( ...; ...; money -=100 )
```

■ for语句的前两个参数以分号结束，这两个分号是不可缺少的。

6.5.1 无参数的 FOR循环

for语句的有趣之处是每个参数都一定是任何合法的C++语句。

■ 在for语句中，第一个参数不必是一个初始值，第二个参数也不一定是一个测试条件，第三个参数也不必是改变条件的。每一部分都可以是任意合法的C++语句，包括空语句在内。

下面的语句完全合法：

```
for( ; a=2; b=3)
{
    cout <<"Hi\n";
}
```

这个for结构的输出结果是：

```
Hi
Hi
Hi
.
.
.
```

将被无限地打印下去。这个 for 结构表示：无初始值；第二个参数将 a 赋值为 2，这是个非 0 值；第三个参数只是将 b 赋值为 3。这样看来，测试条件的值总是为真，因为它的值在第三个参数中没有改变。这样，这个 for 循环语句将无限地执行下去，直到通过操作系统发出一个中断为止。

甚至可以略去第二个参数，即：

```
for( ; ; b=3)
```

结果同样得到无限输出。

可以略去所有这三部分，即：

```
for( ; ; );
```

请注意，除了括号内的两个分号之外，括号外的分号结束符也是必要的。上述语句将导致程序暂停，直到发出一个中断。这是个什么都不做的永久循环。

6.5.2 在括号外条件改变

可以把条件的改变放到括号的外面，即：

```
//test6_6.cpp
```

```
#include <iostream.h>
main()
{
    for (money=700; money>200; )
    {
        cout<<"Pay bill";
        money -= 100;
    }
    cout<<"Money left is : "<<money<<"\n";
}
```

请注意第三个参数没有了, 取而代之的是在 for 循环体内部进行的条件改变。

6.6 SWITCH语句

■ switch 语句是一个能处理多重 if-else 语句的语句。

请考虑如下例子:

```
if you are in your first year of college
    you are a freshman
else if you are in your second year
    you are a sophomore
else if you are in your third year
    you are a junior
else if you are in your fourth year
    you are a senior
else
```

you 're not in college

可看到的是一连串的选择, 其实只有一项是真的 (理论上); 而 switch-case-default 结构则可以用来处理下述的多中选一的条件。句法如下:

```
switch(on integer expression)
{
    case constant 1:
    {
        do this;
        and this;
    }
    case constant 2:
    {
        do this;
        and this;
```

```
    }  
default:  
    {  
        do this;  
        and this;  
    }  
}
```

- 整数表达式是指任何一个能产生整数的合法的C++ 表达式，求得的值为真（非零）或为假（零）。

其中 case constant 1、case constant 2 等等，代表了标号。它们也是常量。

- 如果整数表达式的结果与 switch 结构中的某一标号匹配，那么就执行对应于这一标号的一组语句（多于两条的语句要用花括号括起来），连同以后的 case 和 default 语。
- 如果没有匹配，那么就跳过这段代码。
- 如果所有的case标号都不匹配，那么执行 default 语句。

现在我们回到在本小节开始处提到的那个例子，假设用数值1代表 freshman（新学生），2代表 sophomore（二年级学生），3代表 junior（三年级学生），4代表 senior（毕业生）。下面是本例的C++程序：

```
//test6-7.cpp  
#include <iostream.h>  
main()  
{  
    int number_of_years=3;  
  
    switch(number_of_years)  
    {  
        case 1:  
            cout<<"I'm a freshman\n";  
        case 2:  
            cout<<"I'm a sophomore\n";  
        case 3:  
            cout<<"I'm a junior\n";  
        case 4:  
            cout<<"I'm a senior\n";  
        default:  
            cout<<"I'm not in college\n";  
    }  
}
```

这个程序的输出结果如下:

```
I'm a junior
I'm a senior
I'm not in college
```

switch 的括号内包含的值是 3, 匹配这个常量 (case 3) 的case标号就找到了, 因此就执行这条 case 语句, 其后的所有语句也都执行了。

大多数情况下, 我们希望程序只执行属于匹配了的case 的语句。C++则使用 break 语句来解决这个问题。这样几乎所有的 switch-case 结构的代码都要包含一个break语句。

■ break语句导致立即从这个控制结构中退出。

我们用break语句重写例 test6-7. cpp

```
//test6-8. cpp
#include <iostream.h>
main()
{
    int number-of-years=3;

    switch(number-of-years)
    {
        case 1:
        {
            cout<<"I'm a freshman\n";
            break;
        }
        case 2:
        {
            cout<<"I'm a sophomore\n";
            break;
        }
        case 3:
        {
            cout<<"I'm a junior\n";
            break;
        }
    }
}
```

```
    case 4:
    {
        cout<<"I'm a senior\n";
        break;
    }
    default:
        cout<<"I'm not in college\n";
}
}
```

这个程序的输出结果如下:

I'm a junior

注意:

- default case 没有 break 语句, 因为这是最后一个case, 在这里不需要使用 break 语句。控制能从这个 switch 中退出来。
- 用花括号把每个 case 结构的代码括起来, 是因为现在每个case 都包含了多条语句。

如果没有匹配值, 那么就会执行其后的 default 语句。

6.6.1 case 结构的顺序

- case 结构不需要任何特定的顺序。

要记住的是: 如果没有用 break 语句, 那么该 case 下面的所有的 case 都会执行。

6.6.2 以字符作开关量

- 在整数表达式中, 可以用字符常量来代替整数。
这些常量也可以代替标号。当编译器对字符常量求值时会把它转换成对应的ASCII码。这个ASCII码无异于字符的整数表示。

下面是对上例使用了字符常量的例子:

```
//test6-9.cpp
#include <iostream.h>
main()
{
    int school_year='J';

    switch(school_year)
    {
```

```
case 'F':
{
    cout<<"I'm a freshman\n";
    break;
}
case 's':
{
    cout<<"I'm a sophomore\n";
    break;
}
case 'J':
{
    cout<<"I'm a junior\n";
    break;
}
case 'S':
{
    cout<<"I'm a senior\n";
    break;
}
default:
    cout<<"I'm not in college\n";
}
}
```

这个程序的输出结果如下:

I'm a senior

注意:

- 小写的 's' 用来代表 sophomore (二年级学生)
- 大写的 'S' 用来代表 senior (毕业生)
- 切记只有单个的常量值才可以作为标号来求值。
也就是说, 只有单个的字符才可用, 因此我们不能用字符串作开关量。

6.6.3 整数和字符混用

- 在一个switch结构中, 整数和字符常量可以随意相互混用。
- 字符常量可以转换成相应的与ASCII码等效的十进制数。
下面对这一特性的举例说明:

```
//test6-10.cpp
#include <iostream.h>
main()
{
    char match='1';

    switch(match)
    {
        case 1:
        {
            cout<<"I'm in case 1 \n";
            break;
        }
        case 49:
        {
            cout<<"I'm in case 49 \n";
            break;
        }
        case '2':
        {
            cout<<"I'm in case 2 \n";
            break;
        }

        default:
            cout<<"I'm default\n";
    }
}
```

在这个例子中，变量 `match` 赋值为字符常量 `'1'`。那么，`switch` 语句按这个字符常量来执行：

I'm in case 49

得到这个结果的原因在于 49 是字符常量 `'1'` 的 ASCII 码 的十进制等效数值。请注意：case 1 没有匹配是因为 1 是一个十进制数。

6.7 CONTINUE 和 GOTO 语句

在前一小节中我们讨论了 `break` 语句，我们在附带地讲一下 `continue` 和 `goto` 语句。它们在功能上是相似的。

■ `continue`语句用来从循环中中断出来，然后直接检测循环条件。

下面是一个简短的例子:

```
//test6-x.cpp
#include <iostream.h>
main( )
{
    int i=50;

    while(i<=80)
    {
        i += 10;
        if(i==70)
            continue;
        cout<<"i is"<<i<<"\n";
    }
}
```

这个程序的输出结果如下:

```
i is 60
i is 80
```

请注意语句

```
i is 70
```

没有输出。原因是这样的: 当 $i=70$ 时, 执行 `continue` 语句, 结果控制直接转到条件测试去了, 循环体中其后的语句就不能执行了。这样, 就永远不会执行到 $i=70$ 的 `cout` 语句。

下面我们讨论 `goto` 语句。

- 无论在哪儿碰到 `goto` 语句, 都允许从一处到另一处的无条件跳转。

下面是对上一个例子用了 `goto` 语句的例子:

```
//test6-x.cpp
#include <iostream.h>
main( )
{
    int i=50;

    while(i<=80)
    {
top-of-loop: i += 10;
```

```
        if (i==80)
            goto top-of-loop;
        cout<<"i is"<<i<<"\n";
    }
}
```

这个程序的输出结果如下:

```
i is 60
i is 80
```

请注意下列语句:

```
top-of-loop: i +=10;
```

top-of-loop是标号的名字, goto语句转到有标号作前缀的语句, 并继续执行。

6.8 小 结

本章我们讲述了下列控制结构:

- if-else语句
- while语句
- do-while语句
- for语句
- switch-case结构

所有这些控制结构都代表了在程序中实现条件逻辑的不同方法。

第七章 函 数

7.1 简 介

在本章中，我们将讨论在 C++ 程序中是如何将函数结合在一起的。

- 函数是独立的程序代码块，它的内部工作对于程序的其余部分是不可见的。

下面打一个比方有助于确切理解函数是什么。

7.2 一个比方

假设您对于竞争激烈的生活感到疲倦，决定用一个星期的时间到加勒比海上一个具有异国情调的小岛上休假。由于您太忙不能亲自安排一切行程，您就选择了一家旅行社来帮您做这些事情。该旅行社通过电话向有关机构预定飞机和旅馆住宿。航空代理处将机票发给旅行社。而旅馆代理处将必要的预定单据也发给了旅行社。然后旅行社把机票和这些单据邮给您本人，这样您所要求的一切就都办妥了！

本来您是有任务要执行的（取得机票和预定旅馆），但您把这些任务委派给旅行社了。旅行社再与适当的业务人员接洽来获得相关的机票和单据。实际上旅行社究竟是怎样完成他们的职责的对您来说是不可见的。而航空公司和旅馆职员究竟是怎样处理机票和预定单据的对旅行社也是不可见的。您所知道（以及可能关心）的一切只不过是您传递给旅行社的信息（您的旅行日期，您要去的岛屿），返回给您的东西（机票和单据），以及可能的副作用（您的旅行开销）。这些人中的每一个人与他人的合作都是通过一个独立执行函数来进行的。而每个函数是怎样执行的对于其外部的机构则是不可见的。

在本质上，这就是一个 C++ 函数的全部内容。

- 函数是由能完成特定任务的独立程序代码块组成，如有必要，可与其它函数合用来产生最终的输出。
- 函数内部工作对程序的其余部分是不可见的。

可以把数值从一个函数传递给另一个函数，这如同在您本人、旅行社以及旅馆/航空公司职员之间来回传递信息一样。现在您应该清楚函数的概念了吧。

所有的 C++ 程序都包含一个叫做 `main()` 的主函数。我们下面就讨论它。

7.3 MAIN() 和函数

- 所有的 C++ 程序都有一个集中的程序入口点，叫做 `main()`。

在 `main()` 内部，可以没有或有任意多个对其它函数的调用。下面是两个例子，可以说明刚刚讲过的内容。

```
//test7-1.cpp
```

```
#include <iostream,h>
main()
{
```

```
    cout<<"This is main \n";  
}
```

这个程序只包含一个函数，即 `main()`，现在看一看 `test7-2.cpp`。首先用自然语言来表达要做的事情。

将 `a` 初始化为 5

调用一个叫做 `call-1` 的函数

将等于 5 的那个值再加上 5

回到主函数

将新值存到另一个称作 `b` 的变量中，

下面是 C++ 语言表达：

```
//test7-2.cpp  
  
#include <iostream.h>  
int call-1(int); // 注意函数原型  
  
main()  
{  
    int a=5;  
    int b;  
  
    b=call-1(a);  
  
    cout<<"a is "<<a<<" b is "<<b<<"\n";  
}  
int call-1(a)  
{  
    a+=5;  
    return a;  
}
```

本程序说明了函数的许多重要特性，以及在 C++ 程序中如何使用。我们将逐行分析一下该程序并指出每行中有意义的特性。

本程序从声明 C++ 类库 `iostream.h` 开始，如果我们要用 `cout` 的话，这是必须的。

下一行是函数原型的声明。

■ 函数原型是用来指明要调用函数的名字。该函数将在以后的程序体中调用和定义。

■ 函数原型不仅指明函数的名字，也指明函数的参数和返回值类型。

传递给函数的值的数据类型（如果有的话）叫做参数（即括号里的内容），函数的返回值（如果有的话）的数据类型叫做返回值类型（即函数名左边的内容）。在这个例子中，函数原型声明为：

```
int call-1(int);
```

表示即将定义一个叫做 call-1() 的函数。它有一个整型参数，同时该函数也返回一个整型数据类型。

接下来的四行是：

```
main()
```

```
{
```

```
    int a=5;
```

```
    int b;
```

下一行需要多想一想：

```
    b=call-1(a);
```

编译器读到这一行时，它把控制从主函数 main() 传递给函数 call-1。我们继续看看 call-1。

```
int call-1(a)
```

```
{
```

```
    a+=5;
```

```
    return a;
```

```
}
```

■ 用来详细说明函数做什么的代码（即函数体）叫做函数定义（function definition）

注意函数定义中的函数名、返回类型和参数类型要与在程序开头遇到的函数原型相匹配。

```
int call-1(int); //函数原型
```

```
int call-1(a)    //函数定义
```

唯一不同之处是：用作为参数而传递给函数的实际变量名代替。但也要注意以下一点：

■ 函数原型以分号结尾，而位于函数定义头部的函数名却不以分号作结尾。

在这个函数中，a 的当前值加上 5，然后用语句

```
return a;
```

将该值返回到调用它的函数，本例中该函数则刚好是 main()。

再回到 main() 中，a 的新值存到 b 中，象这样：

```
b=call-1(a);
```

cout 用来打印出这两个变量的新值。

现在应该对函数的工作原理有了清楚的认识，下面我们将列出与一些函数和函数调用相关的规则。

7.4 必须要有函数原型

- 程序中定义的所有函数的原型必须事先声明。

看下面这段程序:

```
//test7-3.cpp
#include <iostream.h>

main()
{
    int a=5;
    int b;

    b=call_1(a);

    cout<<"a is "<<a<<" b is "<<b<<"\n";
}
int call_1(a) // 请注意要返回的类型
{
    a+=5;
    return a;
}
```

编译这段程序会产生一个错误, 这个错误是由于缺少函数原型而造成的。

7.5 函数原型必须与函数定义相一致

- 函数原型的函数名、参数、以及返回类型必须与其函数定义相一致。

我们改一下 test7-1.cpp, 来解释一下如果不遵守本规则会发生什么问题。

```
//test7-4.cpp
#include <iostream.h>
int call_1(int); // 请注意函数原型

main()
{
    int a=5;
    int b;

    b=call_1(a);
    cout<<"a is "<<a<<" b is "<<b<<"\n";
}
```

```
}  
float call-1(a) // 请注意要返回的类型  
{  
    a+=5;  
    return a;  
}
```

编译这段程序也会产生一个错误。这是因为在函数原型中声明函数的返回类型是整型，而在函数定义中却声明它为浮点型。

7.6 VOID 是合法的参数和返回类型

- 如果不需要给函数传递参数或不需要函数返回一个值，则可使用关键字 `void`。看下面这段程序：

```
//test7-5.cpp  
#include <iostream.h>  
void call-1(void) //注意关键字 void 的用法  
  
main()  
{  
    cout<<"This is main \n";  
    call-1;  
}  
void call-1(void)  
{  
    cout<<"This is call-1\n";  
}
```

编译运行这段程序我们得到结果：

```
This is main  
This is call-1
```

注意在函数定义中关键字 `void` 作为返回值和参数的用法。也要注意在实际调用函数时，略过这个关键字，而在函数定义中再现。

7.7 如果函数有返回值，则 RETURN 语句必须存在

- 如前所述，函数定义中的函数返回类型要与在原型中定义的类型的一致。
- 如果在原型中没有明确指定函数的返回类型，编译器假定 `int`（整型）是默认的返

回类型。

然而请注意：

- 如果不在函数定义中插入一个 `return` 语句, 那么函数就不能将正确的值传回 `main()` 中。

7.8 函数名不必唯一

这一点可能会令您吃惊, 但的确如此。

- 两个或多个不同的函数实际上可以使用同一函数名。在 C++ 中, 用一个术语来表示这一特性, 就是函数重载。

这是 C++ 的一个增强功能, 在与之相似的 C 语言中是不存在的。看下面的程序:

```
//test7-6.cpp

#include <iostream.h>

// 注意有同名的两个不同函数
void sum(int i);
void sum(float j);

main(void)
{
    int i=5;
    float j=5.5;

    sum(i); //调用 sum() 做整数加法
    sum(j); // 调用 sum() 做浮点数加法
}

void sum(int i)
{
    i+=5;
    cout<<"i is "<<i<<"\n";
}

void sum(float j)
{
    j+=5;
    cout<<"j is "<<j<<"\n";
}
```

编译并运行上段程序将得到如下结果:

```
i is 10
j is 10.5
```

这里所看到的只是函数重载的一种极简单的情况。名为 `sum()` 的函数被重载了。

- 在重载的函数中，同一函数被多次定义或实现。怎样正确地实现是靠编译器来完成的，而不是程序员。

编译器将函数定义中的参数类型与函数调用中的函数类型匹配起来，在函数调用时就执行参数类型匹配的那个函数。对于这一点就先讲这么多。

现在让我们修改一下这段程序，看看如果向这两个函数传递不同的数据类型会怎样：

```
//test7_7.cpp

#include <iostream.h>

void sum(int i);
void sum(float j);

main(void)
{
    char i='c';
    float j=3.14323;

    // 将 ASCII 码 c 传给 sum()
    // 将浮点型传给 sum()
    sum(i);
    sum(j);
}

void sum(int i)
{
    cout<<"Inside sum(int) \n";
}

void sum(float j)
{
    cout<<"Inside sum(float)\n";
}
```

该程序正确编译，输出结果如下：

```
Inside sum(int)
Inside sum(float)
```

现在 C++ 编译器每次怎样识别该引用哪个函数呢？这个处理过程很简单。首先，将重载函数定义中的参数与函数调用中定义的参数相比较，如果匹配，就执行这个函数，如果不匹配，则

■ 编译器将进行类型转换，并执行其参数与这个转换类型等效的那个函数。

在我们这个例子中，第一次调用了 `sum(int)`，是因为 ASCII 码字符 ‘c’ 被转换成整型数。因此与 `sum(int)` 的函数定义相匹配。

第二次调用时，值 3.14323 与 `float` 类型相匹配，所以调用了 `sum(float)`。

现在我们改一下这个程序，看一看向需要字符型的函数传递整数会怎样：

```
//test7_8.cpp

#include <iostream.h>

//sum(char) 要求一个字符型参数
//sum(float) 要求一个浮点型参数

void sum(char);
void sum(float);

main(void)
{
    int i=65;
    float j=6.5;

    // 将整型传给 sum()
    // 将浮点型传给 sum()
    sum(i);
    sum(j);
}

void sum(char i)
{
    cout<<"Inside sum(char) \n";
}

void sum(float j)
{
    cout<<"Inside sum(float) \n";
}
```

编译这个程序会导致下列错误信息:

Error: Ambiguity between 'sum(char)' and 'sum(float)'

产生这个错误信息的原因是编译器需要传递给 `sum(char)` 一个字符, 而不是整型, 然而又无法用另一种方法进行必要的类型转换。函数重载将在以后的章节中再次讨论。现在重要的是要使读者通过以上极简单函数重载的例子来理解这个概念。总结一下:

- 在C++中, 多个函数可以起同一个名字。
- 函数调用中的参数数据类型必须与函数原型和函数定义中的数据类型相匹配。
- 如果不匹配, 编译器将执行类型转换。如果数据类型还不能匹配, 就会产生错误信息。

7.9 多个参数是合法的

- 在调用函数时, 在函数名后面的参数叫实际参数。
- 被调用函数中的参数叫形式参数。

到现在为止, 我们举的所有例子都只传递一个参数给被调用的函数。然而,

- 可以向一个函数传递任意多的参数。

重要的是函数定义中参数的个数和类型必须总与其相关函数原型相匹配。如果没有十分合适的, C++ 就试着转换实参的类型, 使之与被调用的函数的形式参数相匹配。

下面是处理多个参数的例子。

```
//test7-9.cpp
#include <iostream.h>

void call-1(int, int, int);
main()
{
    int a=b=c=5;
    call-1(a, b, c);
}
void call-1(a, b, c)
{
    cout<< "a is "<< a << "\n";
    cout<< "b is "<< b << "\n";
    cout<< "c is "<< c << "\n"
}
```

编译运行这段简单的程序, 输出如下结果:

```
a is 5
b is 5
c is 5
```

程序和输出内容应该是很容易理解的。

7.9.1 const 修饰符

- 可以用修饰符 `const` 放在参数之前来标明该参数不可以改变。

应用了 `const` 的参数不可以赋值或以任何方式改变。

下面的程序举例说明其用法。

```
//test7-10.cpp
#include <iostream.h>

//常量参数传递给 function-1()
void function-1(const int l);

main(void)
{
    int i=1;

    // i 做为值参传递
    function-1(i);
}

// const 修饰符放在参数名前
void function-1(const int i)
{
    cout << " i is " << i << "\n";
}
```

编译并运行这段程序得出下列输出结果。

```
i is 1
```

至此，程序正确无疑。现在让通过我们改变 `i` 的值来修改程序。

```
//test7-11.cpp
#include <iostream.h>
void function-1(const int l);

main(void)
{
```

```
int i=1;
function-1(i);
}

void function-1(const int i)
{
    // 常量参数值递增
    i++;
    cout << " i is " << i << "\n";
}
```

编译该程序产生一条错误信息:

Error: Cannot modify a const object.

这是因为试图改变参数 *i* 的值, 而 *i* 早已经被规定为 `const` 类型了。

7.9.2 volatile 修饰符

`volatile` 修饰符在功能上与 `const` 是相反的。

- `volatile` 关键字可放在形式参数的前面, 以表明它们在程序的正常执行过程中是易变的。

这样声明参数是为了防止编译器将它们存储在寄存器中。让我们用关键字 `volatile` 来代替 `const` 修改一下 `test7-11.cpp` 的程序。

```
//test7-12.cpp
#include <iostream.h>

//请注意关键字 volatile 的用法
void function-1(volatile int i);

main(void)
{
    int i=1;
    function-1(i);
}

// 请注意关键字 volatile 的用法
void function-1(volatile int i)
{
    i++;
}
```

```
cout << " i is " << i << "\n";  
}
```

程序输出结果如下:

```
i is 2
```

7.9.3 参数表中可以包含默认的初始化参数

- 在C++中, 可以在调用函数的时候, 赋以参数指定的默认值。
看下段程序:

```
//test7-13.cpp
```

```
#include <iostream.h>  
void function_1(int i, int j);  
  
main(void)  
{  
    int i=1;  
    int j;  
  
    // j 在函数调用时被初始化  
    function_1(i, j=8 );  
}  
void function_1(int i, int j)  
{  
    cout << " i is " << i << "\n";  
    cout << " j is " << j << "\n";  
}
```

编译运行这段程序得到输出结果:

```
i is 1  
j is 8
```

这段程序有特殊的两点:

- 除了参数的数据类型以外, 参数名也可以在函数原型中声明。
- 可以在函数调用时初始化参数。
- 尽管参数可以在函数调用时初始化, 但也请注意它们不能在函数定义的第一行中初始化。

看下段程序:

```
//test7-14.cpp
#include <iostream.h>
void function-1(int i, int j);

main(void)
{
    int i=1;
    function-1(i );
}

void function-1(int i , int j=8 )
{
    cout << " i is " << i << "\n";
    cout << " j is " << j << "\n";
}
```

编译这段程序得出下列错误信息:

Error: Default argument value redeclared for parameter j;

同样, 只有最后一个或最后一组参数可以被初始化, 请看下段程序。

```
//test7-15.cpp

#include <iostream.h>

//注意多个参数的初始化
void function-1(int i = 1, int j, int k=2);

main(void)
{
    int i,k;
    int j=2;

    function-1(i, j, k );
}

void function-1(int i , int j, int k)
{
    cout << " i is " << i << "\n";
    cout << " j is " << j << "\n";
    cout << " k is " << k << "\n";
}
```

```
}
```

编译这段程序得到以下错误信息:

Error: Default value missing following parameter 'i'

这段程序不能正确编译, 是因为 C++ 关于默认参数还有另外一条规则:

- 参数表中只有最后边的一些参数可以初始化, 而变量 `i` 并不是最后几个中的一个。
- 现有让我们来修改 `test7_15.cpp`, 只将变量 `j` 和 `k` 初始化。看看发生了什么。

```
//test7-16.cpp
#include <iostream.h>

//只有 j 和 k 被初始化
void function_1(int i, int j=2, int k=3);

main(void)
{
    int i=1;
    int j, k;

    function_1(i);
}

void function_1(int i, int j, int k)
{
    cout << " i is " << i << "\n";
    cout << " j is " << j << "\n";
    cout << " k is " << k << "\n";
}
```

这段程序正常编译通过, 下面是输出结果:

```
i is 1
j is 2
k is 3
```

输出结果正如我们所料。

现在我们可以总结一下关于参数表中默认参数的应用规则。

- 参数可以在调用函数时初始化。
- 只有参数表中的最后几个参数可以赋值。

7.10 参数表中可以包含省略符

- 在 C++ 中，可以在函数的形式参数中放入省略符来表明该函数在不同的情况下引用不同的参数表来调用。

不对这些参数进行类型检查。如果在格式说明中没有正确规定参数的数据类型，那么结果是不确定的，因为没有进行类型检查。

参数数目可变实现起来是有一些复杂。必须要包含头文件 `stdarg.h`，此头文件中定义的宏是用来得到必要的输出的。因为本书只是一个入门，所以我们还没有真正讲述头文件和宏。我们就不讨论这些函数的具体实现的详细解释了。请参阅用户手册。现在我们只是想让读者知道有这一内容。

7.11 INLINE 函数

函数被调用时，伴随着大量的系统开销。在 C++ 中，可以在被定义的函数前面加上关键字 `inline` 来减少这种开销。

- 编译器编译产生用 `inline` 定义的函数代码，然后用这些代码去代入程序内每次调用函数的地方。这叫做内联扩展（`inline expansion`）。

因为函数已经被编译，在函数调用过程中带来的通常的开销也就避免了。请看下面这段程序：

```
//test7-17.cpp

#include <iostream.h>
int increment(int i);

//请注意关键字 inline
inline increment(int i)
{
    i++;
    return i;
}

main(void)
{
    int i=0;
    while ( i<3)
    {
        i=increment(i);
        cout<< "i is "<< i <<" \n";
    }
}
```

```
}
```

这段程序的输出结果是:

```
i is 1
i is 2
i is 3
```

现在让我们修改一下这段程序, 将内联函数的定义移到 `main()` 之后。

```
//test7_18.cpp
#include <iostream.h>
int increment(int i);

main(void)
{
    int i=0;
    while ( i<3)
    {
        i=increment(i);
        cout<<" i is "<< i <<" \n";
    }
}

// inline 函数定义放在函数调用之后
inline increment(int i)
{
    i++;
    return i;
}
```

编译这段程序会出现错误, 这是因为:

- 内联函数必须在调用之前定义。

由此可知, 内联函数:

- 能减少函数开销
- 必须先定义, 后调用。

7.12 递 归 函 数

- 在 C++ 中, 函数可以调用自身, 称为递归函数。

看下列函数:

```
//test7-19.cpp

#include <istream.h>
void decrement(int i);

main(void)
{
    int i=2;

    i=decrement( i );
    cout<<" i is " << i << "\n";
}

int decrement( i )
{
    cout<< "Inside decrement() \n";
    i--;
    if ( i>0 )
        // 函数调用自身
        decrement ( i );
    else
        return( i );
}
```

程序的输出如下:

```
Inside decrement()
Inside decrement()
i is 0
```

函数 `decrement()` 在测试条件的值大于 0 时 (为真值) 一直递归地调用自身。当 `i` 的值为 0 时, 程序控制返回到 `main()`。

7.13 小 结

在本章中, 我们学习了 C++ 中有关函数许多有趣的特性:

- 主函数 `main()` 给操作系统返回一个整型值。
- 函数原型是强制性的。
- 在函数原型中的返回类型、函数名、参数的数目和类型必须与函数实际调用和定义中的一致。

- 编译器尽可能地完成参数和返回值类型转换，尽量地创建相应的类型。
- 函数名可以重载，即可以起相同的名字。在程序中何时执行哪个函数由编译器决定，而不是由程序员来决定。
- `const` 修饰符通知编译器，这个类型的变量不能修改。
- `volatile` 修饰符通知编译器，这个类型的变量可能被修改。
- 在函数原型中可以给函数设置默认的初始值。
- 借助于参数省略符，可以用可变数目和可变类型的参数来调用函数。
- C++ 包含引用类型，这只是被赋值的变量的另一个名字。
- 引用提供了一个用地址而不是用值给函数传递参数的方便办法。
- C++ 允许定义内联函数，有助于减少函数调用时的开销。
- C++ 允许递归函数，即函数可以调用自身。

第八章 数 组

8.1 简 介

经常把数据组合在一起成为一个有序序列。例如，某个年级中学生的分数，家庭中所有成员的年龄以及某个机构中雇员的姓名。这些例子都是将数据组合在一个特定的类别中(姓名，分数，年龄)，组中的每个成员赋予相同或不同的值，例如这个班的学生分数可能是 85，92，97，88，家庭成员的年龄可能是 35 岁，38 岁，3 岁和 5 岁，机构中的雇员的名字可能是玛丽，约翰，莎丽。所有这些都可以用数组来表示。

- 数组是数据的集合，每个数据项都是有序序列的一部分。
- 每个数据项叫做数组的元素。

本章将描述数组的特性，以及在 C++ 程序中如何表示它们。

8.2 数 组 表 示

让我们来看一看怎样用数组来表示上一小节中的例子。

首先，让我们用数组表示班级的分数。把这个数组取名为 marks，假设这个班只有 4 个人，因此该数组将包含 4 个元素。数组中的数据是一个整数集合，这是个整数的数组即整型数组。该数组表示如下：

```
marks[0]=85;
marks[1]=92;
marks[2]=95;
marks[3]=88;
```

要注意的是：

- 方括号用来规定要处理的是数组中的哪一个元素。
- 从 0 开始计数，而不是从 1 开始。

上面的四个语句用来给数组的每个元素赋值，但在给数组赋值之前，一定要事先声明（记住，所有变量在使用之前都必须声明）。下面是数组的声明：

```
int marks[4];
```

注意：

- 数字 4 表示这个数组中有 4 个元素，从第 0 号元素开始，到第 3 号元素结束。

学习了下一章关于指针的问题后，原因就会显而易见了。现在我们只是给出一点启示。

- 数组的最后一个元素总是一个结束符，用 ‘\0’ 来表示。这个结束符告诉 C++ 编译器这就是数组的结尾。

这个字符实际上是不可见的，但对编译器很有意义，它是这个数组的第五个元素。

我们继续进行下一个例子。我们定义一个家庭成员的年龄数组，该数组表示如下：

```
int age[4]; //声明
```

```
int age[0]=35; //给每个元素赋值
int age[1]=38;
int age[2]=3;
int age[3]=5;
```

最后，我们看一下某机构的雇员姓名的数组。该数组被声明和赋值如下：

```
char names[3][6]; //声明
char names[0][6]="Mary";
char names[1][6]="John";
char names[2][6]="Sally";
```

这个数组看起来与我们先前定义的有如下不同，请注意：

- 这个数组带有两对方括号。

数组 `names` 实际上是个二维数组(相对于我们前面定义的一维数组)，在声明这个数组时，第一个方括号内的数字表示该数组内包含的元素总数。

- 二维数组中的第二对方括号中的数字表明该数组中的每个元素所包含的元素个数。

在这个例子中的二维数组包含了 3 个元素。其中的每个元素又是一个数组，元素是字符型的，赋予每个元素的最大长度是 6。这样数组内的每个姓名不能多于 5 个字符。

现在，我们就总结一下数组的特性：

- 数组可以是任何合法的数据类型。
- 数组可以是一维的，二维的，甚至维数更多，如三维或四维。因为多于二维的数组用得很少，所以我们只讲述二维以下的数组。
- 数组在使用前必须声明
- 数组可以赋值和初始化

8.3 数组在内存中如何存储

在继续学习之前，我们先要了解一下数组在计算机内部是怎样存放的。理解这一点很重要。有以下几点原因：

首先，您必须知道数组在内存中是如何存放的，以便在您使用指针来存取数组元素(这是大多数程序中的基本操作)时概念清晰。

其次，新程序员常犯的错误之一就是不能正确地估计数组的大小。在编译器中的结束符要么截去了数组中有意义的数值，要么覆盖掉程序的其它部分，因为编译器不能对数组的最后一个元素进行正确定位。

当声明数组时，编译器在内存中为之保留 X 个相连的地址。当数组元素赋值时，编译器把适当的值放在这些地址中。我们再回过来看第一个例子：

```
int marks[4];
```

假设编译器在内存中为之保留 4 个空间，内存的起始地址是 5000。已知存放一个整数要占用两个字节。下面这些空间将被保留：

内存地址	内容
5000	85
5002	92
5004	95
5006	88

注意在内存中每个元素的地址以 2 为单位递增。正如前面已经提到过的，这是因为计算机用两个字节的存储空间来存储一个整型数。如果声明的是一个双精度型的数组，则在内存中这样存放：

内存地址	内容
5000	85
5004	92
5008	95
5012	88

现在内存地址都是 4 的倍数。因为存放一个 double 数据类型需要 4 个字节。

字符型数组这样存储：

内存地址	内容
5000	'M'
5001	'a'
5002	'r'
5003	'y'
5004	'\0'

注意存放数组时，每个字符只需要一个字节。名为 name[] 的数组声明是这样的：

```
char name[3][6];
```

数字 6 使得编译器给数组中的每个元素都分配 6 个连续的内存空间，该数组这样存放：

5000	5001	5002	5003	5004	5005
'M'	'a'	'r'	'y'	'\0'	
5006	5007	5008	5009	5010	5011
'J'	'o'	'h'	'n'	'\0'	
5012	5013	5014	5015	5016	5017
'S'	'a'	'l'	'l'	'y'	'\0'

注意内存地址 5005 和 5011 的内容是空的。这是因为 name[] 的第一个元素和第二个元素都只有 4 个字符长，因此只占用 5 个连续的内存地址空间。然而数组声明时编译器受命为之保留 6 个字符空间。也要注意，每个字符串都以一个空字符(记作'\0')结束。

8.4 数组的其它特性

本节将通过一个简短的 C++ 程序来说明与数组有关的其它特性：

8.4.1 在声明时初始化数组

- 数组可以在声明时初始化。

看下面的程序

```
//test8_1.cpp
#include <iostream.h>
main()
{
    int marks[4]= {85,92,95,88};
    cout << "Second element is : " << marks[1] << "\n";
}
```

该程序的输出是:

Second element is 92

注意:

- 这个数组在声明时就初始化了。
- 用花括号来将初始化的值括起来。
- 用 marks[1] 来引用数组的第二个元素, 这是因为数组是从第 0 号元素开始计数的。

下面是另一段程序:

```
//test8_2.cpp
#include <iostream.h>
main()
{
    int marks[3][2] = {{1,2},{3,4},{5,6}};
    cout << "Second element is: " << marks[1][1] << "\n";
}
```

本程序的输出是:

4

只要仔细研究一下这个例子, 就应该能理解上述输出结果。

请注意:

- 本程序中的一个二维数组在声明时初始化。
- 二维数组的每个元素各用一对花括号括起来。
- 元素之间用逗号分开。
- 该语句以分号结束。

8.4.2 数组大小的说明可以省略

数组也可以声明如下:

```
//test8_3.cpp
#include <iostream.h>
main()
{
    int marks[] = {85,92,95,88};
    cout << "Second element is : " << marks[1] << "\n";
}
```

该程序输出:

Second element is 92

注意:

- 数组的大小没有指定。

只有当数组立即初始化时,才能这样声明。编译器检查数组初始化的元素数目,然后按这个大小分配存储空间。如果把 marks[] 初始化为8个元素,编译器就会认为数组 marks[] 由9个元素组成。

8.4.3 数组可以传递给函数

- 数组可以在函数之间来回传递。

回想一下,当向一个函数传递一个变量时,实际上传递的只是该变量的一个拷贝,而不是变量本身。请注意这个规则不适用于数组。

- 当数组传给函数时,处理的是数组的实际的值。

下面是一个例子:

```
//test8_5.cpp
#include <iostream.h>
main()
{
    void modify(int); // 函数原型
    int marks[] = {85,92,95,88};
    modify(marks);
    cout << "Second element is : " << marks[1] << "\n";
}
void modify (marks[])
{
    marks[1] -= 10;
```

```
    return;  
}
```

该程序输出:

Second element is 82

注意:

- 数组 `marks[]` 在 `main()` 中定义, 并没有在被调用的函数 `modify()` 中重新定义。
在被调用的函数中没有必要对数组重新定义, 因为这个数组在 `main()` 中声明时已经分配了存储单元。请注意这个规则用于二维数组时例外。如果要将二维数组作为参数传递函数, 那么它的第二个下标(即数组的每个元素的大小)也必须同时传递。
- 在 `main()` 中, 调用 `modify()` 时, 没有说明作为参数的数组的大小, 只传递了数组名。方括号也省略了。
- 在 C++ 中, 数组的 0 号元素可以简单地通过数组的名字来引用。

例如下面的语句:

```
marks=90;
```

结果是数组的第 0 号元素被赋值。数组的后续元素则可以简单地通过给数组名加上适当的元素号来存取。

```
marks+1=92;
```

将导致 `marks[1]` 中的内容被修改。

- 函数定义中, `marks[]` 的大小被省略。
- 数组 `marks[]` 的第二个元素被减去 10。
- 数组 `marks[]` 修改后的值被传输给 `main()`, 尽管它是在 `modify()` 中修改的。
- 这是因为传递给被调用的函数的是数组的实际内容而不是数组的拷贝。

8.5 小 结

本章描述了数组、数组的表示、以及数组在计算机中怎样存储。我们还学习了如何在声明时初始化数组, 以及在函数内部如何处理。

第九章 指 针

9.1 简 介

指针是 C++ 最强有力的特性之一。在使初学者感觉到最困难的几方面内容中，可以说是难中之最。本章我们将非常详细地解释指针。在后续的章节中，读者将体会到指针对于类、派生类、虚函数和抽象类等 C++ 的增强特性的实现是必不可少的。

9.2 一个比方

用比拟的方法是解释指针这个概念的最好的办法。假定你是克林顿家的朋友。克林顿家与肯尼迪家是朋友，你很想知道肯尼迪家住着多少人。但不知道他们住在哪里，而克林顿却知道。

假定克林顿住在 8364 房间，你可以通过如下步骤来获得必要的信息。

- 1 · 先去 8364 房间
- 2 · 获知肯尼迪家的房门号
- 3 · 该房门号是 9381
- 4 · 去 9381 房间
- 5 · 获知住在那里的人数

如果我们只用姓名和房门号来简化一下这一事件链，可重新写成这样：

地址	姓名	得到的信息
8364	克林顿	9381
9381	肯尼迪	5

我们注意到：很有必要通过克林顿家得到肯尼迪家的地址。也就是说，克林顿家指出了去肯尼迪家的路线。

现在我们用 C++ 来表示该图。假定在程序中 `clinton` 和 `konnelly` 定义为整型变量。在本例中，`clinton` 是指向 `kennedy` 的指针。

■ 指针是一个变量，它包含了另一个变量的地址，即间接地指向那个变量。

请看下列这些关系：

克林顿 (姓名)	9381 (内容)	肯尼迪 (姓名)	5 (内容)
	8364 (地址)	9381 (地址)	

在地址 8364 中，我们可以得到 9381 这个数；在地址 9381，我们得到 5，可用列表格式表示：

姓名	地址	内容
克林顿	8364	9381
肯尼迪	9381	5

数字 9381 是什么呢？它是存储在地址 8364 中的值，即地址 8364 中的内容。9381 也是变量 kennedy 的地址。那么数字 5 是什么呢？它是存储在地址 9381 中的值。前面讲过，9381 是变量 kennedy 的地址。数字 5 是存储在地址 9381 中的变量 kennedy 的内容。让我们用一些词来代替“内容”这一列中的数字。

姓名	地址	内容
克林顿	8364	肯尼迪的地址
肯尼迪	9381	存储在肯尼迪这个地址中的内容

如果地址这个词用 & 表示，而内容这个词用 * 表示的话，我们就可以重新写出这些关系：

姓名	地址	内容
克林顿	8364	&肯尼迪
肯尼迪	9381	*(&肯尼迪)

我们再前进一步，赋给这些名字以与之相符的值：

```
clinton=&kennedy; //第一行
```

```
kennedy=**(&kennedy); //第二行
```

第二行用 clinton 来代替：

```
kennedy=*(clinton); //重写第二行
```

前面说过，clinton 是指向 kennedy 的指针，这是个独特的数据类型，因此它的声明也很独特。指针声明如下：

```
int *clinton; //声明指针类型
```

上面的语句定义了一个名为 clinton 的指针，这个指针指向一个整型数。下面是表明这些关系的程序：

```
//test9-1.cpp
#include <iostream.h>
main()
{
    int *clinton;    // clinton 是指向一整型数的指针
    int kennedy;     // kennedy 不是指针
```

```
kennedy=5; // kennedy 中存放的是 5
clinton=&kennedy; // 使 clinton 指向 kennedy
cout<< "The value stored at kennedy is "<<kennedy<<"\n";
cout<< "The value stored at kennedy is "<< *(&kennedy) <<"\n";
cout<< "The value stored at kennedy is "<<*clinton<<"\n";
```

该程序的输出是:

The value stored at kennedy is 5

The value stored at kennedy is 5

The value stored at kennedy is 5

该程序说明了如下几点特性:

- 引用变量的名字时, 得到变量的内容。

这一点可以通过第一条 cout 语句来说明, 该语句打印变量 kennedy, 所以, kennedy 的内容被输出。

- 变量的内容还可以通过取其地址中的内容而得到。

第二个 cout 语句说明了指针地址访问。

- 可以间接打印变量的内容, 即通过间接访问指向它的指针来完成。

第三个 cout 语句说明了这一点。

要记住如下要点:

- 象其它类型一样, 指针类型也必须事先声明。
- 指针必须用来指向属于该种数据类型的内容

如果没有遵守这两点规则, 指针中就会包含无用的东西。再回到这个例子中, 将 clinton 定义成一个整型指针, 然后使它指向 kennedy 的地址, kennedy 的地址是可以通过在变量名前加上地址访问操作符 (&) 来获得的, 储存在 kennedy 地址中的内容通过间接操作符 * 来获得。

*(&kennedy) 或 *clinton

指针就讲这么多, 在继续学习之前, 必须要理解本章已讲述过的内容。花点时间再来复习一下前几页的内容。

9.3 内存地址

- 变量声明时, 就在计算机内存的某个地方为它分配了存储空间。

再看一看上节中讲过的程序:

```
//test9-1.cpp
#include <iostream.h>
main()
{
    int *clinton; // clinton 是指向一整数的指针
```

```
int kennedy; // kennedy 不是一个指针

kennedy=5; // kennedy 中存放的是 5
clinton=&kennedy; // clinton 指向 kennedy
cout<< "The value stored at kennedy is "<<kennedy<<"\n";
cout<< "The value stored at kennedy is "<< *(&kennedy) <<"\n";
cout<< "The value stored at kennedy is "<<*clinton<<"\n";
```

这里声明了一个指针 (clinton) 和一个整型变量, 基于前面所讲的, 把计算机中某一处的存储空间分配给了 clinton 和 kennedy。

简要提一下, 地址操作符 (&) 用来获取变量的地址。让我们给 test9-1.cpp 加一些代码, 看看这个地址是什么样的:

```
//test9-2.cpp
#include <iostream.h>
main()
{
    int *clinton; // clinton 是指向一整数的指针
    int kennedy; // kennedy 不是指针
    kennedy=5; // kennedy 中存放的是 5
    clinton=&kennedy; // clinton 指向 kennedy
    cout<< "The value stored at kennedy is "<<kennedy<<"\n";
    cout<< "The value stored at kennedy is "<< *(&kennedy) <<"\n";
    cout<< "The value stored at kennedy is "<<*clinton<<"\n";
    cout<<"&kennedy is " << &kennedy <<" \n";
    cout<<"clinton is "<<clinton<<" \n";
```

本程序的输出结果如下:

```
The value stored at kennedy is 5
The value stored at kennedy is 5
The value stored at kennedy is 5
&kennedy is 0x1b02
clinton is 0x1b02
```

以下图表可以有助于对所发生的事情看得清楚一些。

变量名	存放的内存地址	内容
Kennedy	0x1b02	5
clinton		0x1b02

数字 0x1b02 是一个十六进制表示的内存地址。kennedy 和 clinton 存放在内存的不同地址里。存在这些内存地址中的内容显示在第三列中。如您所见, 存在 kennedy 中的内容是 5; 而存在 clinton 中的内容只不过是 kennedy 在内存中的地址。这些地址用数字表示。

对指针和指针表示法作如下概括:

- 象其它数据类型一样, 指针也必须事先声明。数据类型名称, 如 int, char, double 等, 在声明中必须放在指针名的前面。
- 在声明中, 指针名的前面必须要有间接操作符 (*)。
- 指针必须指向具体的数据, 否则其内容将是无用的东西。
- 地址操作符 (&) 必须放在指针所指的变量名的前面。
- 指针适用于任何合法的数据类型。

9.4 数 组 指 针

在第八章, 我们描述了数组是怎样在连续的内存空间内存储的。下面这小段程序的输出结果就是数组在内存中被存放的位置。

```
//test9-3.cpp
#include <iostream.h>
main()
{
    int i;
    int array[4]={10,20,30,40};
    for (i=0; i<4; i++)
    {
        cout<<"array[" << i << "] is " << array[i]<<"\n";
        cout<<"and it is stored at " << &array[i]<<"\n";
    }
}
```

该程序的输出结果是:

```
array[0] is 10
and it is stored at 0x1ae0
array[1] is 20
and it is stored at 0x1ae2
array[3] is 30
and it is stored at 0x1ae4
array[4] is 40
and it is stored at 0x1ae6
```

每个变量都存储在两个字节的内存空间里。请注意一下用来输出数组中每个元素在内存中存储地址的标记法。我们要想得到整型、字符型或任何其它合法数据类型在内存中的地址都要这样做。同样地,

- 可通过使用间接操作符来获得存储在数组中的每个元素的对应地址中的内容。

下面这段修改后的程序说明了这一点。

```
//test9-4.cpp
#include <iostream.h>
main()
{
    int i;
    int array[4]={10,20,30,40};
    for (i=0; i<4; i++)
    {
        cout<<"array[" << i << "] is " << array[i]<<"\n";
        cout<<"and it is stored at " << &array[i]<<"\n";
        cout<<"using the indirection operator, it is "
        << *( &array[i])<<"\n";
    }
}
```

输出结果是

```
array[0] is 10
and it is stored at 0x1ae6
using the indirection operator, it is 10
array[1] is 20
and it is stored at 0x1af0
using the indirection operator, it is 20
array[2] is 30
and it is stored at 0x1af2
using the indirection operator, it is 30
array[3] is 40
and it is stored at 0x1af4
using the indirection operator, it is 40
```

请注意为了输出数组中每个元素的内容,我们既可使用常规标记法(array[0]),也可使用间接操作符(*(&array[0]))。

在前一章中,我们也曾提到过数组的名字与它的第 0 号元素是等效的。因此,

- 可以简单地给数组的名字加上正确的偏移量来引用数组中各个元素。

下面有一个例子:

```
//test9-5.cpp
#include <iostream.h>
main()
{
    int i;
    int array[4]={10, 20, 30, 40};
    for (i=0; i<4; i++)
    {
        cout<<"array is " <<array<<"\n";
        cout<<"array+1 is " <<array+1<<"\n";
        cout<<"array+2 is " <<array+2<<"\n";
        cout<<"array+3 is " <<array+3<<"\n";
    }
}
```

该程序的输出结果是:

```
array is 0x1af4
array+1 is 0x1af6
array+2 is 0x1af8
array+3 is 0x1afa
```

请注意这里输出了数组中的每个元素的地址。在这些地址中存储的内容可以通过使用间接操作符来得到。

```
//test9-6.cpp
#include <iostream.h>
main()
{
    int array[4]={10, 20, 30, 40};
    cout<<"array is " <<array<<"\n";
    cout<<"and its contents are" <<*(array)<<"\n";
    cout<<"array+1 is " <<array+1<<"\n";
    cout<<"and its contents are" <<*(array+1)<<"\n";
    cout<<"array+2 is " <<array+2<<"\n";
    cout<<"and its contents are" <<*(array+2)<<"\n";
    cout<<"array+3 is " <<array+3<<"\n";
    cout<<"and its contents are" <<*(array+3)<<"\n";
}
```

```
}
```

下面是输出的结果:

```
array is 0x1b20
and its contents are 10
array+1 is 0x1b22
and its contents are 20
array+2 is 0x1b24
and its contents are 30
array+3 is 0x1b26
and its contents are 40
```

小结:

■ 可以用如下两种方法中的任一种来获得数组中元素的地址:

```
&array[i]
```

或

```
array+i
```

■ 可以用如下两种方法中的任意一种来获得数组中每个元素的内容:

```
array[i]
```

或

```
*(array+i)
```

现在, 还有另外一种方法来得到数组中的每一个元素的内容和内存地址, 就是用指向该数组的指针来完成。下例就说明了这一实现过程。

```
//test9-7.cpp
#include <iostream.h>
main()
{
    int *ptr, i;
    int array[4]={10, 20, 30, 40};
    ptr=array;
    for (i=0; i<4; i++)
    {
        cout<<"array["<<i<<"] is " <<*ptr<<"\n";
        cout<<"and it is stored in memory location "<<array+i<<"\n";
        ptr++;
    }
}
```

该程序的输出结果是这样的:

```
arrar[0] is 10
and it is stored in memory location 0x1ae6
arrar[1] is 20
and it is stored in memory location 0x1ae8
arrar[3] is 30
and it is stored in memory location 0x1aea
arrar[4] is 40
and it is stored in memory location 0x1aec
```

要注意:

- 把 ptr 声明为一个整型指针。这是必须的, 用来指向一个整数数组。
- 将数组的名字赋值给 ptr, 可使 ptr 指向数组的第 0 号元素。
- 数组中的 0 号元素也叫做数组的基地址。
- 只需将 ptr 递增并打印 ptr 所指的内存地址中的内容就可以输出数组中的每个元素。

9.5 字符串指针

- 字符串就是由 ASCII 码字符所组成的数组。

字符串可表示如下:

```
char name[5]="Mary";
```

name[0] (数组中的基地址)中的内容是字母 M。地址 name[1]中的内容是字母 a, name[2]中的内容是字母 r, 而 name[3]中的内容是字母 y, name[4]中的内容是 null 结束符, 即 \0。这些字符存储在连续的内存地址中。可通过指向该数组的指针来操纵该数组的内容。下面这段程序就说明了这个问题。

```
//test9-8.cpp
#include <iostream.h>
main()
{
    char name[5] = "Mary";
    char *ptr_char; //这是字符指针
    //ptr_char 指向字符数组的基地址
    ptr_char = name;
    *(ptr_char+0)='G';
    cout<<"name is "<<name;
}
```

该程序的输出结果是

name is Gary

上述程序中, ptr_char 用来指向该数组的基地址。用了下面这个语句来修改存储在
该数组基地址中的内容:

```
* (ptr_char+0)='G';
```

也可如下表示:

```
* (ptr_char)='G';
```

请注意 G 包含在单引号中。这是因为我们只修改了该数组中的一个元素。如果用双
引号结果就使 G 表示成一个字符串, 也就是说它要以一个空 (null) 字符结尾, 并占用
两个字节的存储空间。

下例再次表示了上述概念。

```
//test9-9.cpp
#include <iostream.h>
main()
{
    char name[5] = "Mary";
    char *ptr_char; //这是字符指针
    //ptr_char 指向字符数组的基地址
    ptr_char = name;
    *(ptr_char+2)='n';
    cout<<"name is "<<name;
}
```

该程序的输出结果是这样的:

name is Many

对字符串和指针总结如下,

- 字符串是字符型的数组。
- 可以通过指向字符的指针来访问并操纵字符串。
- 可通过字符串的名字得到该变量的基地址。
- 可以通过在数组的名字上加上适当的偏移量来获得各个后续元素。
- 可通过对字符串的引用来得到字符串常量的基地址。

9.6 函数参数指针

在函数一章中, 我们曾提到可以向函数传送参数, 但是, 这样只是把参数的拷贝传
递给函数。在函数的内部对这些变量进行了操作, 那么, 实际变量的值也仍然保持不变。
然而,

- 如果将变量的指针作为参数传递, 那么该变量的实际内容就可以改变了。

下面用简例来说明:

```
//test9-10.cpp
#include <iostream.h>
char name[4] = "Mary";
char *ptr;
void modify(char *ptr); // 函数原型
main()
{
    ptr= name; //ptr 指向字符数组 name[] 的基地址
    modify(ptr);
    cout<<"name is "<<name<<"\n";
}
void modify(char *ptr)
{
    *(ptr+2) = 'n';
}
```

该程序的输出结果是:

name is Many

首先, ptr 指向了数组 name[] 的基地址。然后调用 modify() , 该指针就作为一个参数传递给函数 modify() 。在这个函数中, 修改了该指针所指向的数组中的第二个元素。name[] 的新内容回送到 main() 中。

总结一下,

- 函数不能直接地改变变量的内容, 但是却可以通过指向该变量的指针而间接地改变变量的内容。

9.7 指 针 运 算

- 指针可以递增、递减、比较。
- 指针可以作加、减运算。
- 如果两个指针指向相同类型的对象, 那么它们可以彼此相减。

当然, 这里假定正在进行逻辑上正确的运算。

不允许下列对指针的运算:

- 指针与常量相乘
- 指针被一常量来除
- 两个指针相乘
- 一个指针除以另一个指针
- 常量减去指针

■ 两个指针相加

表 9.1 总结了允许的和不允许的指针运算, 以及这些运算的执行结果。下面是个程序实例, 表 9.1 的内容以该例为基础。

```
//test9_11.cpp
#include <iostream.h>
main()
{
    int *ptr1,*ptr2;
    int a[2],b;
//ptr1 指向整型数组 a[]的第一个元素
//ptr2 指向整型数组 a[]的第二个元素
    ptr1 = &(a[0]);
    ptr2 = &(a[1]);
//假定存储 a[0]的内存地址是 1000。
//我们的计算机用两个字节来存储一个整数。
//因此, 存储 a[1]的内存地址是 1002。

    .
    .
    .
}
```

表 9.1 的最后一列表明“表达”那一列所示的表达式的执行结果。所有的结果都基于指针的初始值, 即 ptr1 的内容是 1000, ptr2 的内容是 1002。

表 9.1 指针操作结果

操作	允许	表达	结果
指针递增	允许	ptr1++; ++ptr1; ptr2++; ++ptr2;	ptr1=1002 ptr2=1004
指针递减	允许	ptr1--; --ptr1; ptr2--; --ptr2;	ptr1=998 ptr2=1000
指针比较	允许	if (ptr1>ptr2) for (; ptr1<ptr2; ptr1++) while (ptr1<ptr2)	false 将循环一次 true
指针加上 整数	允许	ptr1+=5; ptr2+=5;	ptr1=1005 ptr2=1007

指针减去 整数	允许	ptr1-=5; ptr2-=5;	ptr1=995 ptr2=997
两指针相减	允许	b=ptr1-ptr2	b=2
常量减去 指针	不允许	b=8-ptr1 b=10-ptr2	不能编译
两指针相加	不允许	b=ptr1+ptr2	不能编译
指针乘以 常量	不允许	b=ptr1*5	不能编译
指针除以 常量	不允许	b=ptr1/5	不能编译
指针相乘	不允许	b=ptr1*ptr2	不能编译
指针相除	不允许	b=ptr1/ptr2	不能编译

9.8 引用参数

本小节将讲述一种引用。引用可代替指针来传递给函数。下面讲述的内容可使您对引用有一些了解。

■ 引用只不过是变量的另一个名字。

请看下列代码:

```
//test9_12.cpp
#include <iostream.h>
main(void)
{
    //给 devil 赋值
    // satan 是 devil 的另一个名字
    int devil=666;
    int &satan= devil;
    cout<<"devil is "<<devil <<"\n";
    cout<<"satan is"<<satan <<"\n";
}
```

程序输出结果是:

```
devil is 666
satan is 666
```

变量 devil 是整型数,初始化为 666,然后作为引用类型的 satan 被赋值为等于 devil。当用 & 符放在一个名字之前就说明是引用。引用必须在声明的同时就被初始化。用一种稍微不同的说法重述即是, satan 只不过是 devil 的另一个名字。devil 初始化为 666,把引用类型 satan 赋值为等于 devil 以后, satan 中就包含了数值 666。请注

意该程序输出了 `satan` 的值，而不是输出了 `&satan` 的值。如果输出 `&satan` 将会得到该引用类型在内存中的地址。

让我们修改一下该引用的值，看看会发生什么。

```
//test9-13.cpp
#include <iostream.h>
main()
{
    int devil=666;
    int &satan= devil;
    cout<<"devil is "<<devil <<"\n";
    cout<<"satan is"<<satan <<"\n";
    //改变了 satan 的值
    satan=999;
    cout<<"devil is "<<devil <<"\n";
    cout<<"satan is"<<satan <<"\n";
}
```

该程序输出如下:

```
devil is 666
satan is 666
devil is 999
satan is 999
```

如上例所示，改变引用类型 `satan` 的值导致了赋予其值的那个变量（即 `devil`）值的变化。这一点是很有意义的。总之，`satan` 只是 `devil` 的另一个名字！

既然已懂得了什么是引用，让我们看看怎样把它用作函数的一个参数。

在函数中通过传值来把参数传递给函数。换句话说，不把变量本身传给函数，传递的只是变量的一个拷贝。对于一个程序员来说，如果想改变被调用的函数中的变量值，就得给函数传递指针，然后通过这个指针来操纵变量的内容。然而，

■ 在 C++ 中，既可以通过传值又可通过引用来传递参数。

请看看下面这段程序是怎样使用引用来代替指针而改变被调用函数中的 `i` 的内容。

```
//test9-14.cpp
#include <iostream.h>
int function_1(int j);
main()
{
    int i=66;
```

```
// j 是对整数类型的一个引用
int &j= i;
cout<<"j is "<<j<<"i is "<<i<<"\n";

//将引用传递给 i
i=function_1(j);
cout<<"j is "<<j<<"i is "<<i<<"\n";
}
int function_1(j)
{
    j=3;
    return j;
}
```

编译并运行程序将得到如下结果:

```
j is 66 i is 66
j is 3 i is 3
```

通过引用来传递参数 *j*，而不是由值来传递。因此这个参数就提供了对被引用的对象（即 *i*）的直接存取方法，并且不用指针就可修改被调用的函数中的参数中的内容。

让我们回顾一下引用这个概念:

- 引用只不过是赋值变量的另一个名字。引用要在赋值操作符的左侧加一个 **&** 符号，而且必须在定义的同时初始化。
- 引用提供了一种向函数传递参数的方便途径。它是用传递地址而不是传值的办法来实现的。在 C 中也存在这种情况。

9.9 小 结

本章描述了指针和引用。指针包含了变量的内存地址，它必须事先声明，然后初始化成指向某一具体数据。

我们学习了怎样声明和使用不同数据类型的指针。如数组和字符串。我们也学习了怎样用指针来改变函数内变量的实际值；并列出了所允许和不允许的指针运算。本章也讨论了引用，以及怎样用引用代替指针来修改函数中的变量。

第十章 结 构

10.1 简 介

在韦伯斯特百科大辞典中，是这样定义结构的：

“... 建筑结构或组织的模式；基本构成成份、元素、部件的分类排列...”

“... 任何事物都是由以某种形式排列在一起的基本部件组成；一个组织...”

在 C++ 中，用关键字 `structure` 来替换英语中的这个概念。下面我们来学习结构。

10.2 结构的概念

现实世界中，常把数据以某种特定的方式组合在一起。例如，把家庭定义成有一个丈夫，一个妻子和一个或几个小孩。一个班级则可以通过用名称以及该班内的男生和女生的人数来定义。公司的雇员可由姓名，头衔和薪水组成。

■ C++ 中允许以结构的形式来定义一组数据的集合。

■ 结构是一种复合数据类型。用一种混合模式将不同的信息块收集到一起，并赋予一个名字。

10.3 结 构 声 明

正如上一小节所述，可以把家庭定义成包含一个丈夫，一个妻子，一个或多个孩子。下面是这一数据的记录：

表 10.1 家庭结构

Husband	Wife	Children (个数)
John	Kathy	2
Kenneth	Sylvia	3
Peter	Ioyce	1

这一信息可以翻译成如下的 C++ 结构形式：

```
structure family
{
    char husband[10];
    char wife[10];
    int children;
};
```

这就是结构声明。

■ 关键字 `structure` (可以简写为 `struct`) 通知编译器要声明一个结构类型数据。该结构的名称即标记是 `family`。

■ 结构的内容叫做成员 (members)。

结构 `family` 的成员有 `husband`，`wife` 和 `children`。`husband` 和 `wife` 是字符数

据类型, 而 children 则是整型。下列是应用于该句法的规则:

- 结构声明必须以关键字 `structure` 或 `struct` 打头。
- 该关键字后必须跟有结构名即结构标记。
- 结构名后紧跟着一个左花括号。结构成员及其数据类型就包含在左括号及右括号之间。每个成员的数据类型以分号结束。
- 右花括号后面带有分号。

请注意:

- 结构声明仅仅定义了该结构的模板即它的格式。编译器在遇到一结构声明时并不为之预留内存空间。

那么编译器何时才给一个结构分配内存呢? 答案是当声明该结构类型的变量时才分配内存。下面是属于 `family` 结构类型的变量声明语句:

```
struct family Gerard, Cole, Chen;
```

这样, Gerard, Cole 和 Chen 就被定义为 `family` 结构类型。编译器现在就为这些结构分配内存。假设从内存地址 1001 开始。将分配如下:

名字	Husband	Wife	Children
Gerard	1001-1010	1011-1020	1021-1022
Cole	1023-1032	1033-1042	1043-1044
Chen	1045-1054	1055-1064	1065-1066

如表所列。给 husband 和 wife 成员各自分配 10 个字节。给 children 成员分配了两个字节的空间, 因为定义为整型, 在计算机中一个整数存放在两个字节中。

也请注意:

- 结构成员总是按照它们声明的顺序分配空间的。

让我们看另外两个结构声明的例子。

表 10 · 2 班级结构

班级名	男孩数	女孩数
Math	10	15
Computer science	13	10
Economics	7	9

这一信息可以翻译成如下的 C++ 结构形式:

```
structure class
{
    char class-name[];
    int number-of-boys;
    int number-of-girls;
```

```
};
```

该结构类型的变量可以声明如下:

```
struct class Math, Computer-Science, Economics;
```

表 10 · 3 雇员结构

雇员名	头衔	薪水
Berkowitz	高级分析员	85000
Carlston	经理	87000
Reis	程序员	60000

这一信息可以翻译成如下的 C++ 结构形式:

```
structure employee
{
    char name[];
    char title[];
    int salary;
}
```

该 employee 类型的结构可以声明如下:

```
struct employee Berkowitz, Carlston, Reis;
```

10.4 结构成员的赋值

■ 结构成员可以用赋值操作符来赋值。

可以用如下形式来引用结构成员。

```
structure-name.member-name
    结构名      成员名
```

下面的例子可以帮助您理解前面讲过的内容。

```
//test10-1.cpp
#include <iostream.h>
main()
{
    struct family
    {
        char husband[10];
        char wife[10];
        int children;
    };
    struct family Gerard = {"John", "Kathy", 2};
```

```
struct family Cole = {"Kenneth", "Sylvia", 3};
struct family Chen = {"Peter", "Joyce", 1};
Gerard.children=0;
Cole.children=2;
Chen.children=4;
cout<<" Gerard children:"<< Gerard.children<<"\n";
cout<<" Cole children:"<< Cole.children<<"\n";
cout<<" Chen children:"<< Chen.children<<"\n";
}
```

该程序的输出结果是:

```
Gerard children: 0
Cole children: 2
Chen children: 4
```

总结一下:

- 可以在结构变量名后面加上相应的结构成员名(而不是结构类型名称)来引用结构成员。
- 结构成员可以通过赋值操作符来赋值。

结构成员也可以按如下形式赋值:

```
struct family
{
    char husband[10];
    char wife[10];
    int children;
} Gerard={"John", "Kathy", 2},
  Cole={"Kenneth", "Sylvia", 3},
  Chen={"Peter", "Joyce", 1};
```

请注意:

- 结构变量名之间要用逗号分开。
- 语句要以分号结束

10.5 结 构 数 组

- 结构数组声明和其它类型的数组声明句法相同。

请看下面下面的程序:

```
//test10-2.cpp
#include <iostream.h>
main()
```

```
{
    struct employee
    {
        char employee-num;
        double salary;
    };
    struct employee a[3];    //a[]是一个结构数组
    a[0].employee-num=9901;
    a[0].salary=85000;
    a[1].employee-num=1026;
    a[1].salary=66000;
    a[2].employee-num=3908;
    a[2].salary=55000;
    cout<<"Employee Number    Salary\n";
    for(i=0; i<=2; i++)
    {
        cout<<a[i].employee-num<<"    "
            <<a[i].salary<<"\n";
    }
}
```

该程序的输出结果是:

Employee Number	Salary
9901	85000
1026	66000
3908	55000

我们注意到 salary 声明为 double , 这是很有必要的, 是为了能存储雇员的高薪。
这样,

■ 引用结构数组成员的句法是:

array-name[index].member-name

数组名 [下标]. 成员名

为结构数组分配的存储单元情况如下:

数组元素	Employee-num	salary
a[0]	1001-1002	1003-1006
a[1]	1007-1008	1009-1012
a[2]	1013-1014	1015-1018

10.6 结构变量指针

字符型、整型和其它数据类型都可以有指针，那么结构类型也可以有指针。指针可以用来指向任何合法的结构，可以由指针来操纵结构成员的内容。下面的例子与前一例子功能相同，不同的只是是用指针来打印每个结构成员的内容。

```
//test10-3.cpp
#include <iostream.h>
main()
{
    struct employee
    {
        char employee_num;
        double salary;
    };
    struct employee a[3]; //a[]是一个结构数组
    struct employee *ptr //ptr 是指向 employee 结构类型的指针
    a[0].employee_num=9901;
    a[0].salary=85000;
    a[1].employee_num=1026;
    a[1].salary=66000;
    a[2].employee_num=3908;
    a[2].salary=55000;
    cout<<"Employee Number    Salary\n";
    for(i=0; i<=2; i++)
    {
        ptr=&a[i];
        cout<<ptr->employee_num<<"    "
            << ptr->salary<<"\n";
    }
}
```

该程序的输出结果是:

Employee Number	Salary
9901	85000
1026	66000
3908	55000

把变量 `ptr` 声明为结构类型 `employee` 的指针。接着使它指向类型为 `employee` 的结构数组的第一个元素。最后, 用指向结构的指针来输出每个成员。表示形式:

`ptr->employee-num`

与

`a[0].employee-num`

产生同样的输出。

-->操作符用来指向或用来获取任何结构成员的值。这样,

■ 通过指针引用结构数组成员的句法是:

`array_name[index] -> member_name`

数组名 [下标] -> 成员名

■ 使用指针来引用结构的内容之前, 必须先赋值使指针指向该结构的地址。

还要讲述另外一种使用结构指针的方法。

修改一下前面的例子可产生同样的输出结果:

```
//test10-4.cpp
#include <iostream.h>
main()
{
    struct employee
    {
        char employee-num;
        double salary;
    };
    struct employee a[3]; //a[]是一个结构数组
    struct employee *ptr; //ptr是结构类型employee的指针
    a[0].employee-num=9901;
    a[0].salary=85000;
    a[1].employee-num=1026;
    a[1].salary=66000;
    a[2].employee-num=3908;
    a[2].salary=55000;
    cout<<"Employee Number    Salary\n";
    for (ptr=a; ptr<=a+2; ptr++)
    {
        cout<<ptr->employee-num<<"    "
            << ptr->salary<<"\n";
    }
}
```

输出结果同前例。

可以看到, ptr 指向数组 a 的 0 号元素。a 是个结构数组。前面讲过数组的名字实际上就是它的第一个元素的指针, 该操作在 for 语句中执行:

```
for(ptr=a; . . .)
```

for 语句的条件测试部分如下:

```
for(. . .; ptr<=a+2; . . .)
```

这个循环将执行 3 次。请回忆一下前面讲过的:

- 可以简单地通过给数组的名字加上正确的偏移量来引用数组中的每个元素。

在第一次循环中, ptr 指向数组中的第一个元素; 第二次循环时, 它指向第二个元素; 直到最后的第三次。指针在 for 语句中是递增的:

```
for(. . .; . . .; ptr++)
```

现在您应该很清楚这种语句的句法了。

10.7 指针作为结构成员

- 结构成员可以是任何合法的数据类型

字符型、整型、浮点型, 甚至任何数据类型的指针都可以作为结构成员。

下面的例子能有助于理解这一点。

```
//test10-5.cpp
#include <iostream.h>
main()
{
    struct course_work
    {
        char course1[20];
        char course2[20];
        struct course-work *ptr;
    };
    //声明 course-work 类型的结构数组
    course-work semester[3]=
        {{"Computer 101", "Math 101", &semester[1]},
        {"System 101", "Math 102", &semester[2]},
        {"Analysis 101", "Calculus", &semester[0]}};
    int i;
    for(i=0; i<=2; i++)
    {
        cout<<"Ptr points to : "<<semester[i].ptr<<"\n";
    }
}
```

该程序的输出结果是:

Ptr points to : 0x1ac6

Ptr points to : 0x1acc

Ptr points to : 0x1ac0

一个

在这个例子中, 指针 ptr 是这个结构本身的一个成员。声明了该结构的一个数组 semester。数组中除了最后一个元素之外, 每个 ptr 元素都赋值为指向下一个元素的地址, 最后一个元素的 ptr 指回第一个元素。这里所示的是一个典型的链表结构的设计。

也要注意 ptr 的内容是使用结构成员的标准句法来输出的:

```
cout<<"Ptr points to:"<<semester[i].ptr<<"\n";
```

总结一下:

- 象其它合法的数据类型一样, 指针也可以作为结构成员。
- 作为结构成员的指针, 与非结构成员的指针一样, 遵循同样的句法、声明和使用规则。
- 结构可以包含指向任何结构类型的指针 (不只是它所在的结构类型), 只要这些结构已经声明即可。

我们修改一下前面的程序来说明输出的是 ptr 所指的内容, 而不是 ptr 本身的内容。

```
//test10.6.cpp
#include <iostream.h>
main()
{
    struct course-work
    {
        char course1[20];
        char course2[20];
        struct course-work *ptr;
    };
    //声明 course-work 类型的结构数组
    course-work semester[3]=
        {{"Computer 101", "Math 101", &semester[1]},
        {"System 101", "Math 102", &semester[2]},
        {"Analysis 101", "Calculus", &semester[0]}};
    int i;
    for (i=0; i<=2; i++)
    {
        cout<<"contents being pointed to : "
            <<semester[i].ptr->course1<<"\n";
    }
}
```

```
}
```

该程序的输出结果是:

```
contents being pointed to :System 101
contents being pointed to :Analysis 101
contents being pointed to :Computer 101
```

请注意用在 cout 语句中的句法:

```
cout<<"contents being pointed to:"
    <<semester[i].ptr->course1<<"\n";
```

因为 ptr 是一个结构成员, 必须用这个句法来引用。指针所指的内容用 -->操作符来输出。紧接着这个操作符的是要显示其内容的结构成员名。

请再花点时间来消化一下刚才讲过的内容。这种句法在 C++中用得很多。

10.8 用结构作结构成员

在前一节中, 我们讲过指针可以是任何合法的数据类型。那么, 结构也是一完全合法的数据类型。因此,

■ 结构也可以作为其它结构中的独特成员。

请看下例:

```
//test10.7.cpp
#include <iostream.h>
main()
{
    struct children
    {
        char boy[10];
        char girl[10];
    };
    struct family
    {
        char husband[10];
        char wife[10];
        struct children Kenneth; //一个结构作为结构成员
    };
    struct children Kenneth={"Jeff","Sally"};
    struct family Krieger={"Gene","Sara","Jack","Jill"};
    cout<<"Krieger.Kenneth.boy: "
        <<Krieger.Kenneth.boy<<"\n";
```

```

    cout<<"Krieger.Kenneth.girl: "
        <<Krieger.Kenneth.girl<<"\n";
}

```

该程序的输出为:

```

Krieger.Kenneth.boy: Jack
Krieger.Kenneth.girl: Jill

```

本例说明, 结构可以作为另一个结构的成员。请注意引用这个结构的句法:

```
krieger.kenneth.boy
```

先是取外面的结构名, 接着是内联的下一个结构名, 如此下去。kenneth是krieger的一个结构成员。因此krieger在kenneth前面。如果children结构的成员之一也是一个结构, 假定其名为description, 那么, 该结构中的任何一个成员(比如height)都可以这样引用:

```
kenneth.krieger.Description.height
```

我们有把握得出如下结论:

- 结构可以作为其它结构的结构成员。
- 结构可以多级地嵌入, 只要在编译器或计算机级支持的范围内。
- 所有的结构, 无论是嵌入的还是非嵌入的, 都必须在程序中的某个地方定义。

10.9 在函数内修改结构变量的内容

结构或结构中的成员可以在函数之间来回传递。在函数内修改结构变量内容的规则与修改任一其它所有数据类型变量是一样的:

- 可以通过指针间接地修改结构成员的内容。

这上因为以参数的形式传递给函数只是变量的拷贝而不是实际的变量。可以把指针设定成指向结构成员的地址, 然后将该指针作为参数传给函数。

用下例来说明:

```

//test10-8.cpp
#include <iostream.h>
struct couple
{
    char man[];
    char woman[];
};
void break-up(struct couple *ptr);
main()
{
    struct couple Camastra = {"John", "Sara"};
}

```

```
struct couple *ptr;

ptr=& Camastra;
cout<<"Before break-up,Camstra couple was : \n";
cout<<Camstra.man<<" "<< Camstra.woman<<"\n";
break-up(ptr);
cout<<"After break-up,Camstra couple is : \n";
cout<<Camstra.man<<" "<< Camstra.woman<<"\n";
}

void break-up(struct couple *ptr)
{
    ptr->woman ='C';
}
```

该程序通过 breakup 函数说明了这种情况，输出结果是：

```
Before break-up,Camstra couple was :
John Sara
After break-up,Camstra couple was :
John Cara
```

也说明了如何在函数内通过指向结构成员的指针来修改该结构成员的内容。

10.10 小 结

在本章中，我们学习了结构的概念以及它在 C++ 中是怎样表示的。一个结构以某种固定的模式将不同的信息块组合到一起，并赋予一个名字。我们学习了如何给结构成员赋值。同时也学习了：

- 如何声明结构数组及其在计算机内如何存储。
- 如何使用结构变量指针。
- 结构指针如何作为结构成员。
- 如何内用指针在函数修改结构成员。

第二部分 C++增强特性

第十一章 类 机 制

11.1 简 介

从这一章开始，我们讲述使C++成为强大语言的那些特性。首先，我们讨论 C++ 的关键概念之一：类机制。读者不仅要学会如何声明、定义和使用类，而且，更重要的是，读者要知道用类来代替结构的原因，而且要知道使用类的究竟能带来什么好处。下一章，我们将讨论声明和定义类成员的具体规则。

首先，我们回顾一下上一章讲过的结构。

11.2 C++中的结构

上一章我们讨论过，结构是一种特殊的数据类型，它以某种固定的格式将其它合法的数据类型聚集到一起。下面是一个简单的结构声明：

```
struct family
{
    char *husband;
    char *wife;
    char *son;
    char *daughter;
};
```

```
struct family Anderson;
```

结构 family 由四个指向字符数组的指针变量组成，显然这些变量分别为 husband、wife、son 和 daughter。Anderson 是结构类型 family 的一个变量，也可以在声明时初始化Anderson 的成员，并通过指针来存取：

```
Anderson.husband="John Anderson";
Anderson.wife="Mary Anderson";
Anderson.son="Joey Anderson";
Anderson.daughter="Marla Anderson";
```

通过指针来存取Anderson的成员，可用如下程序来说明：

```
//test11-1.cpp
#include <iostream.h>
struct family
{
    char *husband;
```

```
char *wife;
char *son;
char *daughter;
};
// 声明 Anderson 为结构类型 family 的一个变量
//注意关键字 struct( 或 structure ) 可以省略
family Anderson=
    [{"John Anderson"}, {"Mary Anderson"},
    {"Joey Anderson"}, {"Marla Anderson"}];
//ptr 是 family类型的指针
family *ptr;

main(void)
{
    //ptr 指向 family 的第一个成员
    ptr=& Anderson;

    cout<<"husband is "<<ptr->husband<<"\n"
    <<"wife is "<<ptr->wife<<"\n"
    <<"son is "<<ptr->son<<"\n"
    <<"daughter is "<<ptr->daughter<<"\n";
}
```

编译运行该程序会得到如下输出结果:

```
husband is John Anderson
wife is Mary Anderson
son is Joey Anderson
daughter is Marla Anderson
```

在本程序中, 声明了一个名为 family 的结构, 定义 Anderson 为这个结构类型的变量, 并且给其中各个成员赋以初始化值。接着, 声明了这个类型的指针 ptr。在 main() 中, 指针 ptr 指向存储结构变量 Anderson 的内存地址。通过指向相应成员的指针 ptr, 由 cout 语句来打印输出 Anderson 的每个结构成员。

要注意, 在声明 Anderson 和 ptr 时, 关键字 struct(或structure)被省略掉了。在C++中, 这个关键字是可选的。但在声明结构原型时, 这个关键字则是必不可少的。

如果再看一下这个 main() 函数, 就会发现它分成了两个部分。第一部分逻辑上初始化为指针 ptr, 使之指向结构变量 Anderson。第二部分只是简单地输出每个成员的内容。因为是用函数把程序划分成各个逻辑分支的, 而每一分支都完成各自内在的任务, 修改一下 test11-1.cpp, 建立两个函数来实现这个功能。

```
//test11-2.cpp
#include <iostream.h>
struct family
{
    char *husband;
    char *wife;
    char *son;
    char *daughter;
};
family Anderson=
    {"John Anderson"}, {"Mary Anderson"},
    {"Joey Anderson"}, {"Marla Anderson"}};
//ptr 是 family类型的指针
family *ptr;
//下面是函数原型
family *initialize(family *ptr);
void output(family *ptr);

main(void)
{
    // ptr 指向 initialize()
    //将 ptr 传递给 output()
    ptr=initialize(ptr);
    output(ptr);
}

family *initialize(family *ptr)
{
    ptr=& Anderson;
    return ptr;
}

void output(family *ptr)
{
    cout<<"husband is "<<ptr->husband<<"\n"
    <<"wife is "<<ptr->wife<<"\n"
    <<"son is "<<ptr->son<<"\n"
    <<"daughter is "<<ptr->daughter<<"\n";
}
```

该程序的输出结果如下:

```
husband is John Anderson
wife is Mary Anderson
son is Joey Anderson
daughter is Marla Anderson
```

现在, 假定有由 15 个附加函数组成的 1000 行程序代码, 要加到这一原始程序中, 这些增加的函数由 `main()` 来调用。该程序就象下面的程序:

```
//test11.3.cpp
#include <iostream.h>
struct family
{
    char *husband;
    char *wife;
    char *son;
    char *daughter;
};
family Anderson;
family Anderson=
    {"John Anderson"}, {"Mary Anderson"},
    {"Joey Anderson"}, {"Marla Anderson"};
//ptr 是 family类型的指针
family *ptr;
//下面是函数原型
family *initialize(family *ptr);
void output(family *ptr);
//增加的15个函数的函数原型放到这里

main(void)
{
    //ptr接收到指向Anderson的指针
    ptr=initialize(ptr);
    //在这里调用那附加的15个函数
    output(ptr);
}

//函数initialize()的代码保持不变
```

```
family *initialize(family *ptr)
{
    ptr=& Anderson;
    return ptr;
}
```

//15个新增加的函数的代码放在这里

//函数output()的代码保持不变

```
void output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
    <<"wife is"<<ptr->wife<<"\n"
    <<"son is"<<ptr->son<<"\n"
    <<"daughter is"<<ptr->daughter<<"\n";
}
```

假定该程序的输出结果如下:

```
husband is Mark Davis
wife is Jennifer Davis
son is Michael Davis
daughter is Maria Davis
```

这个输出结果是由于在程序的某个过程中, 指针 ptr 错误地指向结构类型 family 的其它的变量(也许是 Davis), 它的结构成员已初始化为 Davis 的成员了。这对于作为程序员来说意味着要在这 1000 行语句的程序中, 查找究竟是在哪个函数中将指针 ptr 设为指向 family 类型的变量 Davis (而不是 Anderson) 的内存地址。可是这样的事情通常会使程序员既伤脑筋又浪费时间。C++ 就能有效地解决这一问题。

11.3 C++中的类机制

在C++ 中, 有一个类机制允许指定一个独立的对象集合, 该对象集合由组成该类的对象以及这些对象所允许的操作组成。让我们来修改一下这个程序的最初版本 (test11-2.cpp), 用类(class)来代替结构, 看一看类是如何运作的:

```
//test11-4.cpp
#include <iostream.h>
class family
{
    //注意关键字 private 的用法
```

```
//这个关键字后面是成员表
//ptr 在类 family 内部声明
private:
char *husband;
char *wife;
char *son;
char *daughter;
family *ptr;
//注意关键字 public 的用法
//这个关键字后面是成员函数
public:
void initialize (void);
void output (family *ptr);
};
//Anderson 是类 family 的对象
family Anderson;

main(void)
{
    //initialize() 被对象名限定
    //注意圆点 (.) 操作符
    Anderson.initialize();
}

//initialize() 被类名限定
void family :: initialize(void)
{
    //初始化对象 Anderson 的成员表
    //类成员由对象名来限定
    Anderson.ptr=&Anderson;
    Anderson.ptr->husband="John Anderson";
    Anderson.ptr->wife="Mary Anderson";
    Anderson.ptr->son="Joey Anderson";
    Anderson.ptr->daughter="Marla Anderson";
    Anderson.out (Anderson.ptr);
}

//output() 被类名限定
void family :: output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
```

```
<<"wife is"<<ptr->wife<<"\n"  
<<"son is"<<ptr->son<<"\n"  
<<"daughter is"<<ptr->daughter<<"\n";  
}
```

该程序产生如下输出结果:

```
husband is John Anderson  
wife is Mary Anderson  
son is Joey Anderson  
daughter is Marla Anderson
```

现在让我们一步步地分析这个程序,理解为什么要用类来代替结构,以及为什么要这样来声明类。

关键字 `class` 后面跟着的是类名,叫做 `family` (很象 C 中的结构名),接着是左花括号 (又很象 C 中的结构声明),这样就能生成一种独特的数据类型,一个名为 `family` 的类 (`class`)。

接着,是关键字 `private`,后面是一个冒号,然后是声明表。包含在类声明的左右花括号中的声明表就叫做成员表。关键字 `private` 是一个存取说明符,表明其后的声明的存取权限。该关键字规定了后面的变量只能由该类的成员函数 (或其友元,我们将在后面的章节里讨论友元的机制) 所使用。但什么是成员函数呢? 让我们在回答这个问题之前继续分析这一类声明。

在关键字 `private` 的后面是 `husband`、`wife`、`son`、`daughter` 的声明。然后还有一个 `family *ptr` 也是私有的, `husband`、`wife`、`son`、`daughter` 叫做类成员 (正象结构成员一样)。 `ptr` 声明为指向 `family` 类的指针。请注意,类名 `family` 前并没有关键字 `class`,正象在一个结构中声明结构的变量一样,不要求一定要在此结构的声明之前加上 `struct`。编译器自然会将 `family` 理解为类名,因为已经在类声明的开始处指定过了。

下面是关键字 `public`,后跟一个冒号,然后是函数声明表。这些函数叫做成员函数 (`member functions`),即方法 (`methods`)。由于这些函数声明为 `public`,因此它们既可以被成员函数所访问,也可以被非成员函数所访问;它们可以在整个程序范围内的任何地方被访问或传递以参数。

在 `main ()` 中,对成员函数 `initialize ()` 作了调用。请注意,该函数有一个前缀叫 `Anderson`,以及一个圆点 (`.`),再回顾一下结构的成员的引用方式。

```
struct family  
{  
    char *husband;  
    char *wife;  
    char *son;  
    char *daughter;
```

```
};  
struct family Anderson;  
Anderson.husband="John Anderson";  
Anderson.wife="Mary Anderson";  
Anderson.son="Joey Anderson";  
Anderson.daughter="Marla Anderson";
```

结构成员前面带有这种结构类型的变量名加上一个圆点操作符。

可以用同样的方法来引用类成员。initialize() 是类 family 的成员函数，而 Anderson 则是该类的一个对象。可象下面这样调用函数 initialize()：

```
Anderson.initialize();
```

现在看看 initialize() 的函数定义：

```
void family::initialize(void)  
{  
    .  
    .  
    .  
}
```

要注意函数名的前面是类名加上作用域分辨符 ::。回忆一下本书第一部分讲述的 C++ 是如何允许不同的函数拥有相同的函数名的。在我们的程序中很可能还有其它的函数以 initialize() 为名，然而编译器却能理解这里只不过是类 family 的一个成员函数，因为在它的前面已经标明了类的名称！换句话说，该函数被类名所限定。现在应该能理解为什么把 :: 称做作用域分辨符：它分辨了其后的函数名的作用域，使编译器理解其后的函数定义究竟是属于类还是属于结构，或者只是自含的一个独立实体。让我们继续分析下列代码：

```
Anderson.ptr=&Anderson;  
Anderson.ptr->husband="John Anderson";  
Anderson.ptr->wife="Mary Anderson";  
Anderson.ptr->son="Joey Anderson";  
Anderson.ptr->daughter="Marla Anderson";  
Anderson.output(Anderson.ptr);
```

类成员 ptr 赋值为类对象 Anderson 内存地址，然后初始化其余的成员。最后，调用成员函数 output()，而且该调用也用了类对象名来限定。

该程序以函数 output() 的定义结束。函数依然用类名来限定。返回的值和类型以及参数个数都与类说明相一致。

总结一下：

- 通过关键字 class 来声明类。
- 类象结构一样也有一个标记名。在本章的示例程序中，类名叫 family。
- 类由变量和 / 或函数的表所组成。

在类成员表中分别包含了指向字符数组 husband、wife、son 和 daughter 的指针。除此之外，把 ptr 定义为类 family 的指针。函数 initialize() 和 output() 在类成员表中是成员函数。

■ **关键字 class 不一定要放总在该类对象（实例）的变量声明的前面。**

在本示例中，Anderson 是类 family 的一个对象即实例。

■ **存取说明符，如 private 和 public 用来规定类中成员的存取权限。**

在本示例中，类成员 husband、wife、son、daughter 和 ptr 是私有的，类成员函数 initialize() 和 output() 是公共的。这就意味着程序中任何地方的任何合法参数都可以传递给 initialize() 和 output()。而私有的类成员只能由 initialize() 和 output() 来访问。重申一下，husband、wife、son、daughter 和 ptr 只能通过函数 initialize() 和 output() 来赋值或操纵。

■ **用类对象名加上圆点操作符来限定成员函数的调用。**

■ **用类名加上作用域分辨符来限定成员函数的定义。**

■ **成员函数的返回值类型，函数名以及参数个数和参数类型必须与类中的相关声明相一致。**

那么，为什么类总是使人迷惑呢？让我们回过头来看看 test11-3.cpp 中的方案。在该程序中，ptr 错误地指向了 Davis 的 family 结构内存地址。问题就在于程序员不得不在包含 15 个以上的函数的 1000 行的代码中仔细查找究竟 ptr 在何处被重新赋值了。但是如果在 test11-4.cpp 中再加上 1000 行代码，万一出现了什么错误，程序员总可以将查找范围仅限于两个函数中：initialize() 和 output()。这是因为其中含有错误值的变量被规定为类的私有成员。这就意味着它们只能由这两个函数来操纵。

因此：

■ **类机制允许将变量以及操纵这些变量的函数组合在一起作为一个单一的类型。**

■ **类也有助于将问题迅速定位，这一点是通过类声明中指定存取权限来实现的。**

此外，假设需要在以后修改程序的话，而且这些修改只与类 family 的成员有关，已知这些成员被声明为私有的，所要做的只是修改该类中的两个成员函数的逻辑。这类程序维护起来非常容易，可以说是举手之劳。

11.4 一些常见错误

现在读者已经理解了类的基本概念，那么我们对 test11-4.cpp 作些改动，再看看编译器对这些变化如何反应。修改一下 main() 的内容，用类名代替类的对象名来限定函数 initialize()。这里所有对原始程序的修改都用加双星号 (**) 的注释来表示。一定要注意这些变化。

```
//test11-5.cpp
#include <iostream.h>
class family
{
    //注意关键字private的用法
```

```
//这个关键字后面是成员表
//在类 family 内部声明 ptr
private:
char *husband;
char *wife;
char *son;
char *daughter;
family *ptr;
//注意关键字public的用法
//这个关键字后面是成员函数
public:
void initialize(void);
void output(family *ptr);
};
//Anderson 是类 family 的对象
family Anderson;

main(void)
{
    // ** initialize() 被类名所限定
    family.initialize();
}

void family::initialize(void)
{
    //初始化对象 Anderson 的成员表
    //类成员由对象名来限定
    Anderson.ptr=&Anderson;
    Anderson.ptr->husband="John Anderson";
    Anderson.ptr->wife="Mary Anderson";
    Anderson.ptr->son="Joey Anderson";
    Anderson.ptr->daughter="Marla Anderson";
    Anderson.output (Anderson.ptr);
}
// output() 由类名来限定
void family::output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
    <<"wife is"<<ptr->wife<<"\n"
```

```

    <<"son is"<<ptr->son<<"\n"
    <<"daughter is"<<ptr->daughter<<"\n";
}

```

编译这个程序，编译器会产生如下的错误信息：

Error: Use . or -> to call family::initialize()

对类声明可以写成：

```
family Anderson;
```

也可以这样的格式声明：

```
family Anderson,Davis,Samuel;
```

编译器需要知道操纵的是其中的哪一个对象或实例。这就是在函数 `main()` 中，被操纵的函数在调用时要在函数名前加上前缀的必要性。因此语句 `family::initialize()` 是不正确的。

总之：

■ 在调用类的成员函数时，必须在函数名前加上实例名作为前缀，而不是类名。

现在我们修改一下前一版本（`test11-4.cpp`），在函数 `initialize()` 中引入没有任何名称限定的类成员。要注意到，修改的部分用加双星号的注释来表示。

```

//test11-6.cpp
#include <iostream.h>
class family
{
    //注意关键字 private 的用法
    //这个关键字后面是成员表
    //ptr 的声明在类 family 内部
    private:
    char *husband;
    char *wife;
    char *son;
    char *daughter;
    family *ptr;
    //注意关键字 public 的用法
    //这个关键字后面是成员函数
    public:
    void initialize(void);
    void output(family *ptr);
};
//Anderson 是类 family 的对象

```

```

family Anderson;

main(void)
{
    //initialize() 被类的对象名限定
    //注意圆点(.)操作符
    Anderson.initialize();
}

void family::initialize(void)
{
    //初始化对象 Anderson 的成员表
    /** 类成员不用对象名来限定
ptr=&Anderson;
ptr->husband="John Anderson";
ptr->wife="Mary Anderson";
ptr->son="Joey Anderson";
ptr->daughter="Marla Anderson";
output(ptr);
}
//output() 由类名来限定
void family::output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
    <<"wife is"<<ptr->wife<<"\n"
    <<"son is"<<ptr->son<<"\n"
    <<"daughter is"<<ptr->daughter<<"\n";
}

```

该程序编译时，并未出现错误（读者会很惊讶），test11-6.cpp 的输出结果与前例相同。下面我们看看原因。

ptr 指向对象 Anderson 的内存地址，因此，每次用 ptr 初始化类成员的时候，就会作用于对象 Anderson 上。这就是指针 ptr 不需要对象名来限定的原因。编译器可以理解这一点。

我们再修改一下最初的程序，试着在 main() 中初始化指针 ptr。ptr 将作为一个实际参数传给 initialize()，类声明和函数定义也作相应的修改。

```

//test11-7.cpp
#include <iostream.h>
class family

```

```
{
//注意关键字 private 的用法
//这个关键字后面是成员表
private:
char *husband;
char *wife;
char *son;
char *daughter;
family *ptr;
//注意关键字 public 的用法
/**ptr作为参数传递给 initialize()
public:
void initialize(family *ptr);
void output(family *ptr);
};
//Anderson 是类 family 的对象
family Anderson;

main(void)
{
/**在main()中存取私有成员
Anderson.ptr=&Anderson;

/**给initialize()传递参数ptr
Anderson.initialize(Anderson.ptr);
}

/**initialize()接受 ptr作为参数
void family::initialize(family *ptr)
{
//初始化对象 Anderson 的成员表
//类成员不用对象名来限定
ptr->husband="John Anderson";
ptr->wife="Mary Anderson";
ptr->son="Joey Anderson";
ptr->daughter="Marla Anderson";
output(ptr);
}
//output() 由类名来限定
```

```
void family :: output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
    <<"wife is"<<ptr->wife<<"\n"
    <<"son is"<<ptr->son<<"\n"
    <<"daughter is"<<ptr->daughter<<"\n";
}
```

编译这个程序会产生如下的错误信息:

Error: family::ptr is not acceassable

出错原因很明显, 只需稍微想一下就会发现, 指针 ptr 是作为类 family 的私有成员来声明的。总之,

■ 类的私有成员只能由该类的成员函数来操纵。

在本章结束之前, 我们再来试着对最初的程序作更多的修改。这次, initialize() 和 output() 的函数原型都在类的外部声明。

```
//test11-8.cpp
#include <iostream.h>
class family
{
    //注意关键字 private 的用法
    //这个关键字后面是成员表
    private:
        char *husband;
        char *wife;
        char *son;
        char *daughter;
        family *ptr;
    //注意关键字 public 的用法
    //这个关键字后面是成员函数
    public:
        void initialize(void);
        void output(family *ptr);
};
//Anderson 是类 family 的对象
family Anderson;
/**下面是类成员函数的函数原型
void initialize(void);
```

```
void output(family *ptr);

main(void)
{
    //initialize() 被类的对象名限定
    Anderson.initialize();
}

//initialize() 被类名限定
void family::initialize(void)
{
    //初始化对象 Anderson 的成员表
    //类成员由对象名来限定
    ptr->husband="John Anderson";
    ptr->wife="Mary Anderson";
    ptr->son="Joey Anderson";
    ptr->daughter="Marla Anderson";
    output (Anderson. ptr);
}

//output() 由类名来限定
void family::output(family *ptr)
{
    cout<<"husband is"<<ptr->husband<<"\n"
    <<"wife is"<<ptr->wife<<"\n"
    <<"son is"<<ptr->son<<"\n"
    <<"daughter is"<<ptr->daughter<<"\n";
}
```

编译这个程序会产生如下的错误信息:

Error:

Class member 'initialize' declared outside its class

Class member 'output' declared outside its class

编译器把每个函数原型的返回值、函数名、参数类型及数量与类中的声明相匹配,因为在类定义的外部发现了类的成员函数的声明,所以产生了出错信息。

总之,

■ 成员函数只能在它所在的类内声明,不能在其它任何地方重新声明

如果有任一因素未能匹配,编译器都不会发出这一信息。例如,下列的声明就可以被接受。

```
void initialize(int i,int j);
```

```
void output(int k,int l);
```

然而，这样的声明会让编译器相信这些函数不属于任何类或结构。编译器期望找到函数 `initialize()` 和 `output()` 各自的定义。属于类的定义前面会带类名和作用域分辨符，象我们在前一程序中所示的说明。不属于这个类的函数定义，会简单地定义如下：

```
.  
.
  
void initialize(int i,int j)
{
    .
    .
}
  
void output(int k,char l)
{
    .
    .
}
```

11.5 小 结

我们总结一下本章所讲的内容：

- 类提供了这样一种机制，它将变量以及操纵这些变量的函数组合到一个单一的类型中。
- 类可以使问题快速定位，使程序容易维护。
- 类中的成员表包含了变量声明和可选的函数声明。
- 类的公共成员函数调用需要以类的对象名作为前缀。
- 类的成员函数定义需要以类名和作用域分辨符作为前缀。
- 类中的成员函数不能在类外重新声明。

第十二章 类作用域和类成员访问

12.1 简介

本章将讨论关于类、类成员和成员函数的具体规则。如果较好地理解了上一章所讲述的基本概念,那么学习这一章就轻而易举了。如果读者还没有适应过来,本章还会帮着达到这个目的。本章有些信息看似重复,但却有助于我们巩固上一章中介绍的概念。全面地理解类的概念能使我们认清其全部潜力和强大功能。

12.2 类声明

■ 类按如下格式声明:

```
class class-name
{
//以下是成员表
//成员函数也叫做方法
member-1;
member-2;
member-3;
member-function-1();
member-function-2();
};
```

■ 类的对象即实例按如下格式定义:

```
//instance-1是类class-name的一个实例
class-name instance-1;
```

12.3 类名作用域

■ 在类的作用域内类名必须唯一。

不能给两个不同类型的类取相同的名字。看下列程序:

```
//test12-1.cpp
#include <iostream.h>
class increment
{
//下面是公有成员
public:
int i;
int j;
int add-one(int i,int j);
};
```

```
class increment //错误!!!取相同的类名
{
    //下面是公有成员
public:
    int k;
    int sub-one(int k);
};
//var1 是类 increment 的一个实例
//var2 是类 increment 的一个实例
increment var1;
increment var2;
main(void)
{
    //x 和 y 是 main() 中的局部变量
    int x,y;
    //初始化对象 var1 的类成员
    var1.i=1;
    var1.j=2;
    x=var1.add-one(var1.i, var1.j);
    //初始化对象 var2 的类成员
    var2.k=5;
    y=var2.sub-one(var2.k);
    cout<<"x is "<<x<<"\n"
        <<"y is"<<y<<"\n";
}
//用类名来限定 add-one
int increment::add-one(int i, int j)
{
    int l;
    l=i+j;
    return l;
}
//用类名来限定 sub-one
int increment::sub-one(int k)
{
    k -=1;
    return k;
}
```

编译这个程序会导致如下错误:

Error: Multiple declaration for increment()

这个问题可以通过改变其中一个类名来解决, 象这样:

```
//test12-2.cpp
#include <iostream.h>
class increment
{
    //下面是公有成员
public:
    int i;
    int j;
    int add-one(int i,int j);
};
//请注意不同的类名
class decrement
{
    //下面是公有成员
public:
    int k;
    int sub-one(int k);
};
increment var1; //var1 是类 increment 的一个实例
decrement var2; //var2 是类 decrement 的一个实例
main(void)
{
    //x 和y是main()中的局部变量
    int x,y;
    //初始化对象var1的类成员
    var1.i=1;
    var1.j=2;
    x=var1.add-one(var1.i,var1.j);
    //初始化对象var2的类成员
    var2.k=5;
    y=var2.sub-one(var2.k);
    cout<<"x is "<<x<<"\n"
        <<"y is"<<y<<"\n";
}
```

```
//用类名来限定add_one
int increment::add_one(int i,int j)
{
    int l;
    l=i+j;
    return l;
}
//用类名来限定sub_one
int decrement::sub_one(int k)
{
    k -=1;
    return k;
}
```

编译并运行该程序的输出结果如下:

```
x is 3
y is 4
```

现在我们再看看下面这个程序, 两个类取相同名字:

```
//test12-3.cpp
#include <iostream.h>
class increment
{
    //下面是公有成员
public:
    int i;
    int j;
    int add_one(int i,int j);
};
//var1 是 类increment的一个实例
increment var1;
//下面是函数原型
int sub_one(int y);
main(void)
{
    //x 和y是main()中的局部变量
    int x,y;
    //初始化对象var1的类成员
```

```
var1.i=1;
var1.j=2;
x=var1.add_one(var1.i,var1.j);
y=sub_one(y);
cout<<"x is "<<x<<"\n"
    <<"y is"<<y<<"\n";
}
//用类名来限定add_one
int increment::add_one(int i,int j)
{
    int l;
    l=i+j;
    return l;
}
//不用类名来限定sub_one
int sub_one(int k)
{
    /**注意相同的类名
    class increment
    {
        public:
        int k;
    };
    //var2 是 类 increment 的一个对象
    increment var2;
    //初始化类 increment 的成员
    var2.k=5;
    return var2.k;
}
```

该程序的输出结果如下:

```
x is 3
y is 5
```

尽管两个类(increment)取名相同,然而这一版本的程序能够运行。原因是第二个类 increment 的声明在第一个类声明的作用域之外。那么类名的作用域是什么呢?是从类声明开始,到类所在的程序块结束的这一范围。总之,

■ 只要同名的两个类不在彼此的作用域内,编译器就会正确地识别。

在这个示例程序中, 类 `increment` 的第二个声明在函数 `sub_one()` 中, 它不在第一个类 `increment` 的作用域之内。

12.4 类成员的数据类型

■ 类成员表可以由任何合法的 C++ 数据类型组成。

可以包含通常的基本类型:

```
class primary
{
    int a; //整型
    char b; //字符型
    float c; //浮点型
    double d; //双精度浮点型
}
//class-1 是类 primary 的一个对象
primary class-1;
```

可以包含结构类型:

```
class structure-1
{
    //成员表中包含结构类型 family
    struct family;
}
//Anderson是类 structure-1 的对象
structure-1 Anderson;
```

可以包含任何合法类型的指针:

```
class pointer-1
{
    //ptr 是结构类型 family 的指针
    struct family;
    struct family *ptr;
}
//Anderson是类 pointer-1 的一个对象
pointer-1 Anderson;
```

甚至可以包含类, 然而一个类中包含的类必须在前面某处已作声明。
请看下列程序:

12.6 类成员的存取说明符

本小节没有什么新概念，主要是加强已学过的概念，并对前一节的程序作些扩展。

- 类成员可以是公共的 (public)、私有的 (private) 或保护的 (protected)。

在类声明中，在一个或多个类成员的声明的前边可带有某一个关键字。类成员通过这些存取说明符来获得其具体特征。

12.6.1 公共 (public) 存取

- 在C++中，struct (结构) 的所有成员缺省时存取说明都是 public (公共的)。

在下面的声明中：

```
struct family
{
char *husband;
char *wife;
char *son;
char *daughter;
};
//Anderson是结构类型family的变量
family Anderson;
```

在程序中的任何地方都可以存取结构成员 husband、wife、son 和 daughter。

- 类成员缺省的约定是 private (私有的)。

所有属 public (公共) 的成员都必须明确地指出。

```
class family
{
//下面是公共的类成员
public:
char *husband;
char *wife;
char *son;
char *daughter;
};
//Anderson 是类 family 的对象
family Anderson;
```

- 一个存取说明符在遇到另一个存取说明符之前，对其后的所有声明均有效。同一存取说明符在类中可以多次使用。

下列声明是合法的：

```
class family
{
//下面是公共的类成员
```

```
//变量variable-2是嵌套的类成员
class-2 variable-2;
};
//变量variable-1是类class-1的对象
class-1 variable-1;
class class-2
{
    public:
    int i;
};
main(void)
{
    variable-1. variable-2. i=12;
    cout <<"i is "<<variable-1. variable-2. i<<"\n";
}
```

编译该程序会产生如下出错信息:

Error: Type name expected

这是因为在类class-1中声明类 class-2, 而这时类 class-2 还没定义。

12.5 类成员的存取说明符 (Storage Specifiers)

- 在类成员表中的声明可以用除 auto、extern和register外的任何存储分类符来限定。

可以如下地声明一个类:

```
class invalid-1
{
    //下列是不合法的存取说明符:
    auto int a;
    extern char b;
    register int c;
}
//Variable是类 invalid-1的对象
invalid-1 Variable;
```

将使得编译器产生如下信息:

Error: Storage class 'auto' not allowed for a field

Error: Storage class 'extern' not allowed for a field

Error: Storage class 'register' not allowed for a field

12.5.1 静态类成员

请看下面的类声明:

```
class add-1
{
    int counter;
    int i;
    int add-number(int);
};
```

这是类 add-1 的声明, 其中包含两个整数和一个函数的声明。当声明了几个属于该类的变量后, 例如:

```
add-1 class-1, class-2, class-3;
```

类 add-1 的数据成员的拷贝赋给每个实例。也就是说, class-1, class-2 和 class-3 都类 add-1 的对象。

现在假设 对于所有变量 counter 的值都是相同的 (或许 counter 为一个循环计数器), 在程序运行期间, 该值将存在三个拷贝。而实际上, 只有一份拷贝才能效率最高。C++ 允许将关键字 static 放在指定类型声明的前面以代替三个拷贝。

static (静态) 说明了该声明类型的拷贝只有一份, 并且为该类的所有数据对象或实例所共享, 现在请看如下声明:

```
class add-1
{
    private:
        // 请注意关键字 static
        static int counter;
    public:
        int i;
        int add-number(int);
};
```

关键字 static 放在 counter 声明语句的前面。现在当声明这种类型的对象时:

```
add-1 class-1, class-2, class-3;
```

它们的静态成员 counter 在内存中的地址都是相同。这样效率就高多了, 因为该成员只有一份拷贝, 它的内容对于该类的每个实例都不变。

静态成员提供的优点可以总结如下:

- 减少对全程变量的需求。
- 很显然, 能在同一个类内实现逻辑上的数据共享。

```
Anderson.initialize();
}
//注意类名和作用域分辨符
void family::initialize(void)
{
    Anderson.ptr = &Anderson;
    ptr->husband = "John Anderson";
    ptr->wife = "Mary Anderson";
    ptr->son = "Joey Anderson";
    ptr->daughter = "Marla Anderson";
    output(ptr);
}
//注意无类名
void output(family *ptr)
{
    cout<<"husband is "<<ptr->husband << "\n"
        <<"wife is "<<ptr->wife << "\n"
        <<"son is "<<ptr->son << "\n"
        <<"daughter is "<<ptr->daughter << "\n"
}
```

在名为 family 的类声明中,我们发现 output() 被声明为友元函数。这意味着它不是类 family 的成员函数。然而,因为它是友元,所以可以存取该类的私有成员 husband、wife、son、daughter 和 ptr。本程序的输出结果如下:

```
husband is John Anderson
wife is Mary Anderson
son is Joey Anderson
daughter is Marla Anderson
```

■ 友元函数不受存取说明符的影响。

如果类 family 中的 output() 的声明作如下变动:

```
class family
{
//以下是私有成员
private:
char *husband[25];
char *wife[25];
char *son[25];
```

```
public:
char *husband;
char *wife;
//下面是私有的类成员
private:
char *son;
char *daughter;
//下面又是公共的类成员
public:
char *neice;
};
```

下面是说明公共类成员用法的简短程序:

```
//test12.6.cpp
#include <iostream.h>
void function_1(int i);
class add_numbers
{
    // 下面是公共类成员
public:
    int i;
    int add_sum(int);
}
// one 是类 add_numbers 的对象
add_number one;
main(void)
{
    one.i=1;
    function_1(one.i);
}
void function_1(int i)
{
    i += 5;
    cout<<"i is "<< i << "\n";
}
```

输出为:

I is 6

请注意, 函数 `function_1()` 不是 `add_number` 的成员函数。然而编译器允许它带参数 `i`, 并可以改变 `i` 的值, 这是因为 `i` 被声明为公共的 (`public`), 因此对它没有存取限制。

12.6.2 私有 (`private`) 存取

■ 在 C++ 中, 所有类成员缺省的存取说明都是私有的 (`private`)。

这样, 下面的声明:

```
class family
{
char *husband;
char *wife;
char *son;
char *daughter;
};
```

等效于:

```
class family
{
private:
char *husband;
char *wife;
char *son;
char *daughter;
};
```

■ 私有成员只能被它们所在类的成员函数存取 (也可以被类的友元 (`friend`) 函数存取, 这一点我们在本章后面将讨论)。

下面是 `test12-6.cpp` 的另一个版本, 在该程序中用 `private` 代替 `public` 来限定类成员。

```
//test12-7.cpp
#include <iostream.h>
void function_1(int i);
class add_numbers
{
// 缺省时为类的私有成员
int i;
int add_sum(int);
}
// one 是类add_numbers的对象
add_number one;
```

```
main(void)
{
    one.i=1;
    function-1(one.i);
}
void function-1(int i)
{
    i += 5;
    cout<<"i is "<< i << "\n";
}
```

编译该程序出现如下错误:

Error: add-numbers :: i is not accessible

这个错误是由于在 main() 中调用了函数 function-1() 而引起。在 function-1() 中传递了一个整型参数。然而, 被传递的恰巧是类 add-numbers 的私有成员, 函数 function-1() 对它没有存取权限, 因为该函数不是该类的成员函数 (或友元函数)。

让我们修改一下 test12-7.cpp。看一看如果给 function-1() 传递的是其它变量会怎么样, 我们将给它传递一个在 main() 中初始化过的变量 k

```
//test12-8.cpp
#include <iostream.h>
void function-1(int i);
class add-numbers
{
    // 缺省时为私有类成员
    int i;
    int add-sum(int);
}
// one 是类 add-numbers 的对象
add-number one;
main(void)
{
    //k 是main()中的局部变量
    //k 作为参数传递给function-1()
    int k=10;
    function-1(k);
}
```

```
void function_1(int k)
{
    k += 5;
    cout<<"k is "<< k << "\n";
}
```

该程序编译正确并输出如下结果:

```
k is 15
```

原因很简单, 变量 `k` 是在 `main()` 中声明的, 它不是类成员, 因此它没有存取限制。这样, 可以把它传递给同样也不是类成员的函数 `function_1()`。

现在让我们试一试如果将类成员 `i` 传递给类成员函数 `add_num()` 会怎样。

```
//test12-9.cpp
#include <iostream.h>
void function_1(int i);
class add_numbers
{
    // 缺省时为私有类成员
    int i;
    int add_sum(void);
}
// one 是类 add_numbers 的对象
add_number one;
main(void)
{
    //k 是main()中的局部变量
    //k 作为参数传递给function_1()
    int k=10;
    function_1(k);
    //调用add_num()
    one.add_num();
}
void function_1(int k)
{
    k += 5;
    cout<<"k is "<< k << "\n";
}
void add_numbers::add_num(void)
```

```
{
    one.i = 5;
    cout<<"i is "<<one.i<<"\n";
}
```

编译这段程序产生如下信息:

Error: add-numbers :: add-num is not accessible

产生这一信息的原因很明显, add-num() 是它所在的类的私有成员, 因此在main()中就不可访问。那么怎样才能访问 add-num() 呢? 通常(而且符合逻辑)将类中的数据成员声明为私有的, 而将成员函数声明为公用的。C++程序中的类声明大多数都以这种方式来进行。我们对 test12.9.cpp 作如下改动, 即可排除上述错误:

```
class add-numbers
{
//缺省说明的类成员是私有的
//类方法是公共的
int i;
public:
    void add-num(void);
};
```

该程序的输出如下:

```
k is 15
i is 5
```

12.6.3 保护(protected)存取

保护存取涉及到派生类的成员存取。但派生类还没有讲过, 将在讲到派生类时再详细讲述。

12.7 类成员函数

- 所有在类中声明的函数都叫做成员函数(member function)或方法(method)。
成员函数的作用域和存取规则与普通的类成员是一样的。

12.8 友元(FRIEND)函数

- 类成员函数名前带有关键字 friend 作前缀的, 就是友元函数。

下面是关于友元函数的声明:

```
class family
{
//以下是私有成员
private:
```

```
char *husband;
char *wife;
char *son;
char *daughter;
family *ptr;
//以下是公共成员
public:
void initialize(void);
/**请注意友元函数
friend void output(family *ptr);
};
```

这里声明了类 family，其中 husband, wife, son, daughter, ptr 是它的私有成员，成员函数 initialize() 是公共的，output() 函数用关键字 friend 作前缀，也就是说它是该类的友元函数。

■ 友元函数不是类的成员，然而它可存取该类的私有和全部受保护成员。

我们现在来说明程序中友元函数的用法，请看一段使用了上面这个类的程序：

```
test12_11.cpp
#include <iostream.h>
class family
{
private:
char *husband;
char *wife;
char *son;
char *daughter;
family *ptr;
public:
void initialize(void);
// 注意友元函数
friend void output(family *ptr);
}
// Anderson 是类family的对象
family Anderson;

// **注意output()的函数原型
void output(family *ptr);
main(void)
{
```

```
Anderson.initialize();
}
//注意类名和作用域分辨符
void family::initialize(void)
{
    Anderson.ptr = &Anderson;
    ptr->husband = "John Anderson";
    ptr->wife = "Mary Anderson";
    ptr->son = "Joey Anderson";
    ptr->daughter = "Marla Anderson";
    output(ptr);
}
//注意无类名
void output(family *ptr)
{
    cout<<"husband is "<<ptr->husband << "\n"
        <<"wife is "<<ptr->wife << "\n"
        <<"son is "<<ptr->son << "\n"
        <<"daughter is "<<ptr->daughter << "\n"
}
```

在名为 family 的类声明中, 我们发现 output() 被声明为友元函数。这意味着它不是类 family 的成员函数。然而, 因为它是友元, 所以可以存取该类的私有成员 husband、wife、son、daughter 和 ptr。本程序的输出结果如下:

```
husband is John Anderson
wife is Mary Anderson
son is Joey Anderson
daughter is Marla Anderson
```

■ 友元函数不受存取说明符的影响。

如果类 family 中的 output() 的声明作如下变动:

```
class family
{
//以下是私有成员
private:
    char *husband[25];
    char *wife[25];
    char *son[25];
```

```
char *daughter[25];
// 友元函数在私有成员表中
friend void output(family *ptr);
family *ptr;
//以下是公共成员
public:
void initialize(void);
};
```

它并不表示该友元函数是私有的。下面列出的是友元函数的全部要点。友元是类外面的函数。作为类的友元，可以存取友元类的私有的和保护了的成员，而且其函数原型也不必存在。在本例中故意包含了没有函数原型的友元函数，以证明在程序编译时不会产生出错信息。而假使它是一个类成员，那么就会有错误了。

■ 通过一个简单语句，类的所有成员都可以变为另一个类的友元。

下面的程序说明了这个概念，请看：

```
//test12-12.cpp
#include <iostream.h>
//** 注意不完全的类说明
class neighbor

class family
{
//类的所有成员都是类的友元
friend neighbor;
char *husband;
char *wife;
family *ptr;
// 下面是公共的类成员
public:
void initialize(void);
}
// Anderson 是类 family 的对象
family Anderson;
class neighbor
{
public:
void output(family *ptr);
};
```

```
// Davis 是类 neighbor 的对象
neighbor Davis;
main(void)
{
    Anderson.initialize();
}
void family::initialize(void)
{
    Anderson.ptr = &Anderson;
    ptr->husband = "John Anderson";
    ptr->wife = "Mary Anderson";
    //将ptr传递给友元函数output()
    Davis.output(Anderson.ptr);
}
void neighbor::output(family *ptr)
{
    cout<<"husband is "<<ptr->husband << "\n"
        <<"wife is "<<ptr->wife << "\n"
}
```

编译并运行这段程序会得到如下预期结果:

```
husband is John Anderson
wife is Mary Anderson
```

请注意类 neighbor 的不完整的声明。这样做是必需的。因为类 neighbor 在声明之前被引用了。请记住, 此时也不能声明类 neighbor 的对象, 除非在类 neighbor 的声明完整的前提下才可以。

函数 output() 是作为类 neighbor 的一个成员函数来声明的。类 neighbor 被声明为类 family 的友元类。因此, 类 neighbor 的所有函数也就变成了类 family 的友元函数, 甚至不必逐一在它们的声明前加上关键字。这就是友元函数 output() 能存取类 family 的私有成员 ptr 的原因。

12.9 内联 (INLINE) 函数

在本章结束之前, 还有一个问题需要讨论, 这就是内联函数。内联函数提供了一种减少函数调用时的系统开销的方法。

可以在类声明的内部定义成员函数。这一点可通过下例来说明:

```
//test12-13.cpp
#include <iostream.h>
```

```
class display
{
    int i;
    public:
    //output()是内联函数
    void output(void)
        {cout<<"i is "<<i<<"\n";}
};

// object_1 是类display的对象
display object_1;
main(void)
{
    object_1.output();
}
```

编译并执行这个程序得到如下输出结果:

```
i is 0
```

在类 display 的声明中, 定义了函数 output() , 这是一个隐式的内联函数定义的例子。

■ 在类声明中定义内联函数可以减少函数调用时的系统开销。

因为内联函数的代码将变成该类每个实例的一部分。小的函数是最适合作为类声明的内联函数的。

■ 内联函数也可以显式地声明。

下面是显式地声明内联函数的一个例子:

```
//test12-14.cpp
#include <iostream.h>
class display
{
    int i;
    public:
    //output()是在其它地方定义的内联函数
    void output(void)
};

//object_1 是类 display 的对象
display object_1;
//display::output() 被显式地定义为内联函数
```

```
inline void display::output(void)
{
    cout<<" i is "<< i <<"\n";
}
main(void)
{
    object_1.output();
}
```

运行该程序得到如下输出结果:

```
i is 0
```

内联函数 `output()` 的显式声明是在类声明外边进行的。并且要在前面加上关键字 `inline` 来限定。结果在每个调用 `output()` 函数的地方都会扩展这段内联代码。

■ 内联函数必须在引用之前定义

如果将函数 `output()` 的定义放在 `main()` 的后面, 这个程序就不能正确编译。

12.10 小 结

本章描述了类成员声明和定义的具体规定。

- 作用域内类名必须唯一。
- 类成员表可以由任何合法的C++数据类型组成。
- 静态类成员允许同一类的多个对象共享同一内存地址, 这样减少了全局变量。
- 结构成员缺省的存取说明是公共的 (`public`)。
- 类成员缺省的存取说明是私有的 (`private`)。
- 类成员有公共的 (`public`), 私有的 (`private`) 和保护 (`protected`) 三种存取特性。
- 类成员函数与普通类成员有着相同的作用域和存取规则。
- 友元 (`friend`) 函数可以存取类中的私有的和保护的成员。
- 内联 (`inline`) 函数可以减少函数调用时的系统开销, 小型数最适合在类内部声明为内联函数。

第十三章 派生类

13.1 简介

当一个小孩出生时,就从父亲和母亲那里继承了一定的特征。随着时间的推移,这个小孩就逐步显示出自己所独有的性格,因此就具有了从父母那里继承过来的以及由于其所处的生活环境而产生的特性的组合。

随着人们的成长,人生观也随之改变。我们的梦想、目标和雄心壮志都来源于在我们成长和相互影响的生活天地里的所学所用。

然后我们将自己的学问和成就传授给我们身边和关系密切的人,这就给了他们在我们的成就基础上建功立业的机会。

假想将人的一生都浓缩到一个胡桃核中,就会看清自己的一生究竟有多少是从父母那里继承而来的,有多少是从学校里学的,有多少是从社会生活中学到的,又有多少是通过自己的雄心和努力达到的。如果能将从父母那里继承来的特性组合到一个类中;学校学来的特性放到一个类中;从所处的社会结构中得到的特性再放到另外一个类中,那么就类似于一个派生类(derived class)。因为既继承了基类(父母、教育以及社会结构)的特性。并且又加入了一些属于个人的独特禀赋(雄心、成就以及更多的东西)。C++中的派生类就类似于上述简单的生活故事。派生类继承(inherit)了一个或多个基类的特性,然后又加入了属于它自己独有的特性。这种派生类又可以作为其它派生类的基类,如此类推。

以上也说明了类的等级(hierarchy)是怎样建立起来的,同时更说明了一个对象是怎样在另一个对象基础上建立起来的。这也正是使C++成为如此强大的语言的原因之一。

13.2 简单的C++应用

下面我们要用传统的结构化程序设计方法来开发一个简单的C++应用程序。该程序可以增加、查找、修改和删除文件中的记录。因为我们现在并不关心完成这几项任务的每个函数都是如何编码的。因此,在这些函数中只列出简单的输出语句来表明某一时刻在执行哪个函数。下面是该应用程序的代码:

```
/*test13.1.cpp*/
#include <stdio.h>
/*下面是函数原型*/
void add(void);
void find(void);
void modify(void);
void remove(void);
main()
{
    char string[2], option='\0';
```

```
while(option != 'q')
{
    cout<<"Please enter option: \n";
    cout<<"<a>dd, <f>ind, <m>odify, <d>elete or <q>uit : ";
    cin>>string;
    option=string[0];
    switch(option)
    { /*switch 语句开始*/
        case 'a':
            add();
            break;
        case 'f':
            find();
            break;
        case 'm':
            modify();
            break;
        case 'd':
            remove();
            break;
        case 'q':
            exit(0);
            break;
        default :
            printf("Invalid input ! Please re-enter. \n");
    } /* switch 结束 */
} /* while() 结束 */
return(0);
} /* main() 结束 */

void add(void)
{
    cout<<"Inside add() \n";
}

void find(void)
{
    cout<<"Inside find() \n";
}

void modify(void)
{
```

```
class sum
{
    int sum_1;
    double sum_2;
public:
    //注意函数名sum()被重载了
    sum(int i);
    sum(double k);
};

void main(void)
{
    /** 调用sum(int)
    sum object_1(10);
    /** 调用sum(double)
    sum object_2(10,5);
    cout<<"Inside main() \n";
}

sum::sum(int i)
{
    sum_1 = i;
    cout<<"Inside sum(int) \n";
}

sum::sum(double k)
{
    sum_2 = k;
    cout<<"Inside sum(double) \n";
}
```

编译并运行该程序会产生如下结果:

```
Inside sum(int)
Inside sum(double)
Inside main()
```

当声明 object_1 时,引用了构造函数sum(int),因为参数10 是整型(int)数。当声明object_2时,引用构造函数 sum(double)。

15.7 构造函数的调用顺序

下列程序演示了在处理基类和派生类时构造函数的调用顺序:

```
while(option != 'q')
{
    cout<<"Please enter option: \n";
    cout<<"<a>dd, <f>ind, <m>odify, <d>elete or <q>uit : ";
    cin>>string;
    option=string[0];
    switch(option)
    { /*switch 语句开始*/
        case 'a':
            add();
            break;
        case 'f':
            find();
            break;
        case 'm':
            modify();
            break;
        case 'd':
            remove();
            break;
        case 'q':
            exit(0);
            break;
        default :
            printf("Invalid input ! Please re-enter. \n");
    } /* switch 结束 */
} /* while() 结束 */
return(0);
} /* main() 结束 */

void add(void)
{
    cout<<"Inside add() \n";
}

void find(void)
{
    cout<<"Inside find() \n";
}

void modify(void)
{
```

```
        cout<< " Invalid input ! Please re-enter. \n";
    } /* switch 结束 */
} /* while() 结束 */
return(0);
} /* main() 结束 */
void add(void)
{
    cout<<"Inside add() \n";
}
void find(void)
{
    cout<<"Inside find() \n";
}
void modify(void)
{
    cout<<"Inside modify() \n";
}
void remove(void)
{
    cout<<"Inside delete() \n";
}
```

请注意必须包含头文件 <stdlib.h>, 这个头文件含有对函数 atoi() 的定义, 该函数用来将 ASCII 码字符串转换成整型数。在这个例子中, 对口令的输入必须转换成整数以后再进行正确性检查。根据输入的口令, 在 switch_case 结构中增加一些代码来对不同功能的访问进行限制。

现在我们用类来重写 test13_1.cpp。可由不同雇员执行的函数组合成不同的类。程序分成几个逻辑段。程序分段列出, 每当继续之前, 我们都先解释一下前面的小段程序。首先请看一下类的声明:

```
//test13-1.cpp
//输入输出必需 iostream.h
//getch() 必需 stdio.h
//atoi() 需要 stdlib.h
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
class group_1
{
```

```
public:
void find(void);
};
//everyone 是类 group-1的对象
group-1 everyone;
class group-2
{
public:
void add(void);
void modify(void);
};
//select-few 是类 group-2的对象
group-2 select-few;
class group-3
{
public:
void add(void);
void modify(void);
//注意函数名为 remove()
//因为 delete 是 C++ 的保留字
void remove(void);
};
//heath 是类 group-3的对象
group-3 heath;
```

这里声明了三个类，它们是 group_1，group_2，group_3。注意，group_2 和 group_3 中有两组同名的函数，即 add() 和 modify()，这样做是有必要的，因为我们只对组合在一起的一组函数感兴趣。这组函数可以由作为独立实体的一个组合来执行。C++ 允许我们使用类来实现。

下面是主程序：

```
main(void)
{
char string[2],option='\0';
int i,*intptr;
char *password,*charptr;
cout<<"Please enter password: ";
//获得用户输入
//转换成整数
charptr=gets(password);
```

```
i=atoi(charptr);

while(option != 'q')
{ /*while循环开始*/
cout<<"Please enter option:\n";
cout<<"<a>dd, <f>ind, <m>odify, <d>elete or <q>uit : ";
cin>>option;
switch(option)
{ //按 option 的内容选择 switch 入口
//有权限的雇员可以访问 add()
case 'a':
if (i==10 || i==20)
{
select_few.add();
}
else if (i==99)
{
heath.add();
}
else
cout << "Permission denied!\n";
break;
//所有的雇员都可以访问 find()
case 'f':
everyone.find();
break;
//有权限的雇员可以访问 modify()
case 'm':
if (i==10 || i==20)
{
select_few.modify();
}
else if (i==99)
{
heath.modify();
}
else
cout << "Permission denied!\n";
break;
```

```
//只有希斯先生可以访问remove()
case 'd':
    if (i==99)
        heath.remove();
    else
        cout << "Permission denied!\n";
    break;
case 'q':
    break;
default :
    cout<<"Invalid input ! Please re-enter. \n";
} /* switch 结束 */
} /* while() 结束 */
} /* main() 结束 */
```

在主函数 main() 中, 其内部逻辑与 test13_2.cpp 是一样的。然而, 在这里我们根据输入的口令不同而引用不同的类成员函数, 而不是象前一段程序那样引用相应函数的同一拷贝。下面再看一下函数的定义。

//下面是函数定义

```
void group_1::find(void)
{
    cout<<"Inside group1 find() \n";
}
void group_2::add(void)
{
    cout<<"Inside group2 add() \n";
}
void group_2::modify(void)
{
    cout<<"Inside group2 modify() \n";
}
```

//下面是 group-3 中的 add() 函数定义

```
void group_3::add(void)
{
    cout<<"Inside group3 add() \n";
}
```

//下面是 group-3 中的 modify() 函数定义

```
void group_3::modify(void)
```

```
{
    cout<<"Inside group3 modify() \n";
}
// 下面是 group-3 中 remove() 函数的定义
void group-3::remove(void)
{
    cout<<"Inside group3 remove() \n";
}
```

请注意, add() 和 modify() 有两种版本, 它们分别是类 group_2 和类 group_3 的成员函数。remove() 只有一个版本, 它只是类 group-3 的成员函数。输出语句已修改, 为了能够精确地说明某个时刻在执行哪个函数。

现在如果看一看两个函数每一个版本的代码, 就会发现是完全相同的。怎样才能既有理由把它们组合在相同的类中, 又能省略掉相同代码的多多个版本呢? 答案是使用派生类。

13.3 C++派生类

- C++允许从一个或多个基类中派生出一个类。派生类继承基类的所有成员。
 - 派生类可以改变继承过来的成员的存取权限, 方法是在基类中声明前面加上存取说明符。
 - 合法的存取说明符是 public 和 private。
- 看下面的程序代码:

```
class group-1
{
public:
    void find(void);
};
//everyone 是类 group-1 的对象
group-1 everyone;
//这是一个基类声明, 很容易理解。我们可以从中派生出类 group_2。
//请注意类 group-2 的声明
class group-2:public group-1
{
    //请注意这里省略了 find() 的声明
public:
    void add(void);
    void modify(void);
};
//select-few 是类 group-2 的对象
```

```
group-2 select-few;
```

现在要弄明白派生类是如何定义的。

借助如下语句，从类 `group_1` 中派生出类 `group_2`：

```
class group_2:public group_1
```

表中的第一个类名是派生类的类名，冒号 (:) 后面的类名是基类的类名。类 `group_2` 继承了基类 `group_1` 的所有成员。这些成员现在就属于类 `group_2` 的成员表中了。然而，基类中只有公共的和受保护的成员（受保护的成员在下一章详细讨论）能被派生类使用，而基类中的私有成员对于派生类来说则是不可存取的。

放在基类名前的存取说明符 `public` 用来限定在派生类中被继承的成员表的存取权限：

```
class group_2:public group_1
```

`public` 关键字表示基类的公共成员在派生类中也是公共成员。同样，基类的受保护成员在派生类中也是受保护成员。换句话说，

- **关键字 `public` 表示在派生类中被继承的成员的存取权限与其在基类中的存取权限相同。**

然而请记住，

- **基类的私有成员继续保持其私有的特性，因为它们不能被派生类存取。**

这一点很有意义。类的私有成员的真正意义就是要限制对这些成员的存取权限。如果派生类也继承了对这些成员的存取权限，那么也就失去了私有的意义。派生类通常用作其它对象的构造块。如果产生了十级派生类，而每个类的私有成员又能被下一级的类所存取，那么由类成员的限制而产生的优势也就不复存在了。

下面是派生类的声明：

```
/**请注意关键字private
```

```
class group_2: private group_1
```

```
{
    public:
    void add(void);
    void modify(void);
}
```

```
//select-few 是类 group_2 的对象
```

```
group_2 select-few;
```

同样也可如下表示：

```
class group_2: group_1
```

```
{
    public:
    void add(void);
    void modify(void);
};
```

```
group_2 select-few;
```

这是因为，对于派生类来说，缺省时的派生类别就是 `private`（私有的）。如果在基类的类名前用这个关键字来限定的话，那么基类中公共的和受保护的成员都将成为派生类中的私有成员。

现在我们修改 `test13_1.cpp` 中的派生类。代码再次以分段的形式表示，在进行下一段之前，先解释一下每一段的含义。首先是类声明：

```
// test13-2.cpp
//输入输出需要iostream.h
//getch()需要stdio.h
//atoi()需要stdlib.h
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
class group_1
{
public:
void find(void);
};
//everyone 是类 group_1 的对象
group_1 everyone;
class group_2
{
public:
void add(void);
void modify(void);
};
//select-few 是类 group_2 的对象
group_2 select-few;
class group_3: public group_2
{
public:
void remove(void);
};
//heath 是类 group_3 的对象
group_3 heath;
```

类 `group_3` 是类 `group_2` 的派生类，基类 `group_2` 被限定为公共的（`public`）。结果导致函数 `add()` 和 `modify()` 也成为类 `group_3` 的公共成员，现在请看一下主函数 `main()` 的代码。

```
main(void)
{
    char string[2], option='\0';
    int i, *intptr;
    char *password, *charptr;
    cout<<"Please enter password: ";
    charptr = gets(password);
    i=atoi(charptr);
    while(option != 'q')
    { /*while循环开始*/
        cout<<"Please enter option: \n";
        <<"<a>dd, <f>ind, <m>odify, <d>elete or <q>uit : ";
        cin>>option;
        switch(option)
        { //switch 语句开始
            //有权限的雇员可以访问add()
            case 'a':
                if (i==10 || i==20)
                {
                    select.few.add();
                }
                else if (i==99)
                {
                    heath.add();
                }
                else
                    cout << "Permission denied!\n";
                break;
            case 'f':
                everyone.find();
                break;
            //有权限的雇员才可以访问modify()
            case 'm':
                if (i==10 || i==20)
                {
                    select.few.modify();
                }
                else if (i==99)
                {
```

```
        heath.modify();
    }
    else
        cout << "Permission denied!\n";
    break;
case 'd':
    if(i==99)
        heath.remove();
    else
        cout << "Permission denied!\n";
    break;
case 'q':
    break;
default :
    cout<<"Invalid input ! Please re-enter. \n";
} /* switch 结束 */
} /* while() 结束 */
} /* main() 结束 */
```

这些代码与 test13_1.cpp 中的相同。下面再看一下函数的定义：

//下面是函数定义

```
void group_1::find(void)
{
    cout<<"Inside group1 find() \n";
}
void group_2::add(void)
{
    cout<<"Inside group2 add() \n";
}
void group_2::modify(void)
{
    cout<<"Inside group2 modify() \n";
}
void group_3::remove(void)
{
    cout<<"Inside group3 remove() \n";
}
```

请注意，每个函数只定义了一个版本。编译并运行这段程序会在屏幕上显示如下信息：

Please enter password.

我们假定操作者是有权删除主文件记录的雇员(同时该操作员也要承担合法记录也有可能被删除的责任)。

输入口令 99, 然后选择 f 来查找记录, 屏幕上会显示如下信息:

```
Inside group1 find()
please enter option
<a>dd <f>ind <m>odify <d>lete, or <q>uit
```

输入选择 a, 会显示下列信息:

```
Inside group2 add()
please enter option
<a>dd <f>ind <m>odify <d>lete, or <q>uit
```

输入选择 d, 产生信息:

```
Inside group3 remove()
please enter option
<a>dd <f>ind <m>odify <d>lete, or <q>uit
```

输入选择 q, 则退出程序

现在再看一下这段代码:

```
case 'a':
    // 一部分雇员可访问 add()
    if (i==10 || i==20)
    {
        select_few.add();
    }
    else if (i==99)
    {
        heath.add();
    }
```

在该段程序中, 如果口令为 99, 则执行 heath.add()。然而程序中并没有对 heath.add() 进行定义。编译器之所以能接受它, 是因为 heath 是类 group_3 的一个实例, 并且 group_3 是 group_2 的派生类, 因此它继承了其基类的所有公共成员。

■ 如果函数是从一个类中继承来的, 那么就不需要重新定义。

图 13.1 是对程序补充说明的逻辑图。

这是对象 1

add()
modify()

对象 2 由对象 1 派生出来。

这是对象 2

对象 1
add()和 modify()
remove()

图 13.1 对象的产生

如果把类 group_2 当作一个对象, 则类 group_3 包含这个对象, 还要加上一个与 group_2 中所有函数都不同的函数。如果没有 group_2, 就不会有 group_3 了。这就是一个对象可以用作另一个对象的构造块的很好说明。到现在为止, 文教对派生类这个概念应该有很清楚的认。

13.4 小 结

在本章中, 我们学习了为什么要从一个基类派生出另一个类, 以及如何派生出另一个类。读者应明白用类代替结构的原因。总的来说, 派生类的提供了类的优点, 应具有构造多级对象结构的能力。最上面一级的对象是最通用的对象。从它所派生出来的每个对象都包含被派生对象的特性, 然后又加上了一些其本身独有的特性。

从现在起, 在编程的时候应该记得考虑构造对象, 而不是书写过程循环, 然而这却是原先在编写 C 代码时不得不做的。派生类所能起到的作用到此已略见一斑了。

第十四章 派生类的存取权限

14.1 简介

本章将要讲解由公共基类和私有基类派生出来的类中的被继承成员的存取权限。但是首先要讨论一下保护的成员。

14.2 类中的保护的成员

请考虑如下的类声明：

```
class group_1
{
    int a; //由于缺省，因此 a 是私有的
protected:
    int b; //b 是保护的
public:
    void output1(void);
}
```

group_1 everyone;

group_1 是一个类，其中 a 是一个私有成员，b 是一个保护的成员，而函数 output1() 则是公共的。现在请看一个完整的程序。

```
//test14_1.cpp
#include <iostream.h>
class group-1
{
    //a 缺省为私有的
    int a;
    //b 是保护的
protected:
    int b;
    //output1() 是公共的
public:
    void output1(void);
};
group-1 everyone;
//group-2由group-1派生出来
class group-2:public group-1
{

```

```
//c 缺省为私有的
int c;
public:
    void output2(void);
};
group_2 a_few;
void main(void)
{
    everyone.output1();
}
void group_1::output1(void)
{
    a = 5;
    b = 10;
    cout<<"a is "<<a<<"b is "<<b<<"\n";
    a_few.output2();
}
void group_2::output2(void)
{
    c = 15;
    b += 10;
    cout<<"c is "<<c<<"b is "<<b<<"\n";
}
```

编译并运行这个程序将产生如下结果

```
a is 5 b is 10
c is 15 b is 20
```

类成员 a 和 b 分别是类 group_1 的私有的和公共的成员。于是毫无疑问，编译器允许通过本组的成员函数 output1() 来存取这些成员。保护的成员 b 则可通过 output1() 以及 output2() 来存取。output2() 是类 group_2 是公共成员。这样，

■ 派生类的成员可存取基类中保护的成员。

请注意在这里 group_2 是由 group_1 派生出来的。这就是说，它继承了 group_1 的所有成员。因此，当 group_2 中的成员函数存取 group_1 的类成员时，编译器不产生任何错误。（假设 group_1 中的类成员是公共的或私有的）。

让我们修改一下这段程序，试图通过 output2() 来存取私有成员 a。

```
//test14_2.cpp
#include <iostream.h>
```

```
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};

group_1 everyone;

//group-2 由 group-1 派生出来
class group_2: public group_1
{
    int c;
public:
    void output2(void);
};

group_2 a_few;

void main(void)
{
    everyone.output1();
}

void group_1::output1(void)
{
    a = 5;
    b = 10;
    cout<<"a is "<<a<<"b is "<<b<<"\n";
    a_few.output2();
}

void group_2::output2(void)
{
    c = 15;
    b += 10;
    //存取私有成员
    cout<<"a is "<<a<<"\n";
    cout<<"c is "<<c<<"b is "<<b<<"\n";
}
```

编译并运行这段程序结果产生如下错误信息:

Error: 'group_1::a' is not accessible

这段程序说明:

■ 派生类中的成员不可存取基类中的私有成员。

回忆一下我们在上一章曾谈到过这一点。要注意, 当在 `group1::output1()` 中访问 `a` 的时候, 将不会产生错误信息。

现在我们要再次修改这段程序, 看一看基类中的私有的和保护了的成员对于公共成员是否能自由存取。读者应该已经知道了该程序的结果。但我们还是要列出这段代码以加深已有的印象。

```
//test14-3.cpp
#include <iostream.h>
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};
group_1 everyone;
//group-2 由 group-1 派生出来
class group_2: public group_1
{
    int c;
public:
    void output2(void);
};
group_2 a_few;
void main(void)
{
    cout<<"a is "<<a<<" b is "<<b<<"\n";
    everyone.output1();
}
void group_1::output1(void)
{
    a = 5;
    b = 10;
    cout<<"a is "<<a<<" b is "<<b<<"\n";
```

```
a_few.output2();
}
void group_2::output2(void)
{
    c = 15;
    b += 10;
    //存取私有成员
    cout<<"a is "<<a<<"\n";
    cout<<"c is "<<c<<"b is " <<b<<"\n";
}
```

编译这段程序结果产生如下错误信息:

```
Error: Undefined symbol 'a'
       Undefined symbol 'b'
```

可以肯定地得出以下结论:

- 基类中私有的和保护了的成员只能由该类的成员的存取, 而基类中保护的成员则可以由该基类的派生类的成员存取。

14.3 公共派生类的存取权限

以下这小段程序说明了派生类对基类成员的存取规则:

```
//test14_4.cpp
#include <iostream.h>
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};
group_1 class1;
//group_2 由 group_1 派生出来
class group_2 : public group_1
{
public:
    void output2(void);
};
```

```
group_2 class2;
//group_3由group_2派生出来
class group_3 : public group_2
{
    public:
    void output3(void);
};
group_3 class3;
void main(void)
{
    class1.output1();
    class2.output2();
    class3.output3();
}
void group_1::output1(void)
{
    a = 5;
    b = 10;
    cout<<"a is "<<a<<"b is "<<b<<"\n";
}
void group_2::output2(void)
{
    cout<<"c is "<<c<<"b is "<<b<<"\n";
}
void group_3::output3(void)
{
    cout<<"c is "<<c<<"b is "<<b<<"\n";
}
```

编译这段程序产生下列错误信息:

```
Error: group_1::a is not accessible
      group_1::a is not accessible
```

请注意, 派生类 group_2 和派生类 group_3 中的成员函数对保护的成员 b 的存取没有产生错误信息。我们可以得出如下结论:

- 基类中的公共成员对于其派生类来说仍然是公共的, 因此, 可对公共成员任意存取。在派生类 group_2 和 group_3 中声明的函数 output2()和 output3()是公共成员, 于是, 可以在主函数 main()中调用。
- 基类中的保护的成员对于其派生类来说仍是保护的。

b 在基类 group_1 中是保护的成员。当派生类的成员函数存取该变量的时候, 编译器并没有产生错误信息。而如果试图在一个非成员函数中存取这一变量时, 那么编译器将发出错误信息。

■ 基类中的私有成员只为基类所私有, 不能被派生类中的成员函数存取。

14.4 私有派生类的存取权限

现在把 test14_4.cpp 程序中用 public 作前缀的基类改成用 private 作前缀。下面是程序代码。

```
//test14_5.cpp
#include <iostream.h>
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};
group_1 class1;
/**注意这里的声明为私有的
class group_2 : private group_1
{
    public:
    void output2(void);
};
group_2 class2;
/**注意这里的声明为私有的
class group_3 : public group_2
{
    public:
    void output3(void);
};
group_3 class3;
void main(void)
{
    class1.output1();
    class2.output2();
    class3.output3();
}
```

```
}  
void group_1::output1(void)  
{  
    a = 5;  
    b = 10;  
    cout<<"a is "<<a<<"b is "<<b<<"\n";  
}  
void group_2::output2(void)  
{  
    cout<<"c is "<<c<<"b is "<<b<<"\n";  
}  
void group_3::output3(void)  
{  
    cout<<"c is "<<c<<"b is "<<b<<"\n";  
}
```

编译这段程序产生下列这组相当有趣的错误信息:

```
Error: group_1::a is not accessible  
      group_1::a is not accessible  
      group_1::b is not accessible  
      group_1::b is not accessible
```

我们花点时间来分析一下这些错误信息。

第一条信息说明了在函数 `group_2::output2()` 中不可存取 `group_1::a`。这是很容易理解的。`a` 是类 `group_1` 中的私有成员, 并且对于它所处的类来说总保持其私有的特性, 它不能被派生类的成员所存取。

第二条信息说明了在函数 `group_3::output3()` 中不可存取 `group_1::b`。在原理上与第一条错误的原因是一样的。

第三条错误信息说明了在函数 `group3::output3()` 中不可存取 `group1::b`。这一条信息理解起来可能要多费一些时间。产生这一信息的原因在于 `b` 在其基类中是保护的成员, 而 `group_2` 是由 `group_1` 私有地派生出来, 因为:

■ 私有的派生结果是, 基类中的保护的成员在派生类中变成私有的成员。

于是, 现在 `b` 对于派生类 `group_2` 来说是私有的。因此, `group_3` 的成员函数是不可存取 `b` 的。那么为什么在 `group_2::output2()` 存取 `b` 时就没有得到这一错误信息呢? 这是因为 `b` 在 `group_1` 中是保护的成员, 因此对于派生类 `group_2` 的成员是可以存取的。然而, 正是因为 `group_2` 是私有地派生出来的, 所以 `b` 就变成 `group_2` 的私有成员, 因而对于以后再派生出来的任何类都是不可存取的。

既然读者理解了私有地派生出来的类对于基类中私有成员的存取规则, 那么让我们修改一下 `test14_5.cpp`, 看一看私有的派生类怎样影响公共的基类成员的存取权限。

```
//test14_6.cpp
#include <iostream.h>
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};
group_1 class1;
//group_2是私有的派生类
class group_2 : private group_1
{
public:
    void output2(void);
};
group_2 class2;
//group_3是私有的派生类
class group_3 : public group_2
{
public:
    void output3(void);
};
group_3 class3;
void main(void)
{
    class1.output1();
    class2.output2();
    class3.output3();
    //访问公共成员 output1()
    class2.output1();
    class3.output1();
}
void group_1::output1(void)
{
    a = 5;
    b = 10;
```

```

    cout<<"a is "<<a<<"b is "<<b<<"\n";
}
void group_2::output2(void)
{
    cout<<"a is "<<a<<"b is "<<b<<"\n";
}
void group_3::output3(void)
{
    cout<<"a is "<<a<<"b is "<<b<<"\n";
}

```

编译这段程序得到与上一段程序的前三条信息一样的错误信息，另外还有这两条信息：

```

Error: group_1::output1() is not accessible
      group_1::output1() is not accessible

```

这些信息是由于以下语句产生的：

```

class2.output1();
class3.output1();

```

产生这些错误信息的原因是：

■ 私有派生类的公共成员对于派生类来说也变成了私有的。

output1() 是类 group_1 的公共成员。group_2 是由 group_1 私有地派生出来的。于是，output1() 就变成了 group_2 的私有成员。class2 是类 group_2 的一个实例。语句 class2.output1() 是在 main() 中执行的。但是因为现在 output1() 是 group_2 的私有成员，只能由它所在的成员列表中的成员来存取，而 main() 的权限又不会进入该操作范围。要想解决这个问题，可把 main() 中的语句 class2.output() 移入 group_2::output2() 中去，即：

```

void group_2::output2(void)
{
    class2.output1();
    cout<<" a is"<<a<<"b is"<<b<<"\n";
}

```

然而，下列代码仍有问题。

```

void group_3::output3(void)
{
    class3.output1();
    cout<<" a is"<<a<<"b is"<<b<<"\n";
}

```

第一条语句是可接受的，因为 output2() 是类 group_2 中的成员函数，group_2 是从 group_1 中派生出来的，所以可以在 output2() 中存取 output1()。而第二条语句则不能被接

受, 因为 group_2 是由 group_1 私有地派生出来的; 于是公共成员 output1() 就变成了 group_2 的私有成员, 因而对于 group_3 就是不可存取的了。下面是一段能够正确执行的完整程序。在 output2() 和 output3() 中, 对变量 a 和 b 作输出的代码已经去掉了, 因为出于刚解释过的原因, a 和 b 是不能存取的。

```
//test14_7.cpp
#include <iostream.h>
class group_1
{
    int a;
protected:
    int b;
public:
    void output1(void);
};
group_1 class1;
/**注意group-1和group-2 的 private 说明符
class group_2:private group_1
{
    public:
    void output2(void);
};
group_2 class2;
/**注意这里的group-3是私有的派生类
class group_3:public group_2
{
    public:
    void output3(void);
};
group_3 class3;
void main(void)
{
    class1.output1();
    class2.output2();
    class3.output3();
}
void group_1::output1(void)
{
    a = 5;
```

```

b = 10;
cout<<"a is "<<a<<"b is "<<b<<"\n";
}
void group_2::output2(void)
{
    /*在 output2() 中访问 output1()
    cout<<"Inside output2() \n";
    class2.output1();
}
void group_3::output3(void)
{
    cout<<"Inside output3() \n";
}

```

另外一种解决办法只需公共地派生出 group_2 和 group_3 来，而不是私有地派生出它们来。究竟选择哪种办法，要取决来您希望的这些变量的显著性以及可能受影响的函数的显著性。

这里需要注意的重点是：

- 可用类成员列表以及基类的存取说明符来做到数据隐藏，这是设计模块化程序的关键所在。

好的程序应该由一些很容易识别的独立部分组成，因此会更灵活、更容易修改。如果编写者弄清楚每个变量的作用域，在使用存取说明符的基础上，就可以做到这一点。只是要事先花一些时间来规划出这此具有级联关系的类，决定出派生类对基类成员的存取权限，并且要真正理解为什么要这样做！

14.5 小 结

在本章中，我们描述了派生类中的基类成员的存取权限。表 9.1 总结了关于存取权限的主要概念。

表 14.1 基类和派生类的存取权限

基类的存取说明符	基类中的存取	派生类中的存取
public	公共	公共
	私有	不可存取
	保护的	保护的
private	公共	私有
	私有	不可存取
	保护的	私有

第十五章 构造函数和析构函数

15.1 简介

在本章中，我们将讨论 C++ 如何提供一种机制使得在产生一个类的同时将它初始化，而且又有相应的机制，当这些类不在作用域中即不再需要时又如何释放它们。

- 在创建一个对象时，C++ 可以通过构造函数自动地对该对象作初始化。
- 析构函数与构造函数是对立的。它提供了释放由构造函数所分配的对象空间的一种办法。

在开始讨论构造函数和析构函数之前，读者要理解如果不显式地去调用构造函数和析构函数，它们也能自动被调用来执行的这种机能。既然这样，显式地调用构造函数和析构函数的原因是当编译器没有象我们预期的那样去操作时，我们可用这种办法来分配或释放这些实例的内存空间。例如，如果一个类的对象没有正常地创建起来，就可能有必要释放当初分配给它的内存，然后再在内存的另一个位置上去重新创建。当阅读完本章其余部分时，原因就更明显了。

首先讨论构造函数。

15.2 构造函数简介

考虑一下下列程序：

```
//test15.1.cpp
#include <iostream.h>
class sum
{
public:
    int sum_1;
    int sum_2;
};
sum object_1,object_2;
void main(void)
{
    cout<<"Inside main() \n";
    cout<<"object_1.sum_1 is "<<object_1.sum_1<<"\n";
    cout<<"object_1.sum_2 is "<<object_1.sum_2<<"\n";
    cout<<"object_2.sum_1 is "<<object_2.sum_1<<"\n";
    cout<<"object_2.sum_2 is "<<object_2.sum_2<<"\n";
}
```

编译并运行这个程序产生如下输出:

```
Inside main()
object-1.sum-1 is 0
object-1.sum-2 is 0
object-2.sum-1 is 0
object-2.sum-2 is 0
```

sum1 和 sum2 都自动初始化为 0, 因为它们是在 main() 的外部声明的, 从未赋过任何值。但假定这些类成员已经包含了一些值。下面我们在类声明中初始化这些类成员。

```
//test15-2.cpp
#include <iostream.h>
class sum
{
    /**我们试着初始化类成员
public:
    int sum-1=1;
    int sum-2=2;
};
sum object-1,object-2;
void main(void)
{
    cout<<"Inside main() \n";
    cout<<"object-1.sum-1 is "<<object-1.sum-1<<"\n";
    cout<<"object-1.sum-2 is "<<object-1.sum-2<<"\n";
    cout<<"object-2.sum-1 is "<<object-2.sum-1<<"\n";
    cout<<"object-2.sum-2 is "<<object-2.sum-2<<"\n";
}
```

编译这个程序将产生如下出错信息:

Error: cannot initialize a class member here

因此,

- 类成员不能在声明的同时初始化。但 C++ 允许在对象创建时通过构造函数来初始化。

15.3 缺省的构造函数

- 构造函数是同它要初始化的类具有相同名字的函数。
- 构造函数可以定义成内联 (Inline) 函数的形式, 也可以在类声明的外部定义。
- 构造函数的函数原型和函数定义象普通的函数一样, 只是没有返回值。而且如果它们

没有被显式地调用, 编译器会将它们自动地初始化。

请看 test15-3.cpp

```
// test15-3.cpp
#include <iostream.h>
//函数 sum() 与类 sum 同名
class sum
{
public:
    int sum_1;
    int sum_2;
    sum();
};
sum object_1, object_2;
void main(void)
{
    cout<<"Inside main() \n";
    cout<<"object_1.sum_1 is "<<object_1.sum_1<<"\n";
    cout<<"object_1.sum_2 is "<<object_1.sum_2<<"\n";
    cout<<"object_2.sum_1 is "<<object_2.sum_1<<"\n";
    cout<<"object_2.sum_2 is "<<object_2.sum_2<<"\n";
}
//sum::sum() 是构造函数
sum::sum()
{
    cout<<"Inside sum() \n";
}
```

编译并运行这个程序将产生如下输出结果:

```
Inside sum()
Inside sum()
Inside main()
object_1.sum_1 is 0
object_1.sum_2 is 0
object_2.sum_1 is 0
object_2.sum_2 is 0
```

看一下类 sum 的声明, 就会发现有个类成员函数也叫做 sum()。这是个缺省的构造函数, 这样写是因为它没有参数(带参数的构造函数将在本章的后一部分讨论)。请注意以

下重要的几点:

■ **构造函数与它们所在的类同名, 并且没有返回类型。**

这个程序的输出结果演示了构造函数的一个有趣的特性。当看到输出内容 “ Inside sum()” 在 “ Inside main()” 之前时, 也许会很惊讶。原因是这样的, 类 sum 的对象产生的同时就调用了构造函数 sum()。稍有些不同的是:

■ **对象的构造函数在该类的对象声明的同时被调用。**

如果再看一下 test15-3.cpp, 会发现 object-1 和 object-2 是在 main() 的外部创建的。因此, “ Inside sum()” 在 “ Inside main()” 之前输出了。而且 “ Inside sum()” 输出了二次, 这也是因为创建了类 sum 的两个对象。如果要创建三个该类的对象, 则要调用 sum() 函数三次, 以此类推。

另一个有趣的特性是:

■ **每次声明对象时, 如果没在类声明中指定, 编译器都会隐式地调用缺省的构造函数。**

回想一下前几章的所有程序, 每次声明对象时, 都调用了缺省的构造函数, 文教甚至一直都不知道。

sum-1 和 sum-2 的值继续为 0, 这是因为这些类对象是在 main() 的外部声明的, 它们还没有被赋予其它的任何值。我们看一下如果在 main() 中声明 object-1 和 object-2 会怎么样。

```
//test15-4.cpp
#include <iostream.h>
//函数sum()与类sum同名
class sum
{
public:
int sum-1;
int sum-2;
sum();
};
void main(void)
{
//object-1和object-2在 main() 中声明
cout<<"Inside main() \n";
sum object-1, object-2;
cout<<"object-1.sum-1 is "<<object-1.sum-1<<"\n";
cout<<"object-1.sum-2 is "<<object-1.sum-2<<"\n";
cout<<"object-2.sum-1 is "<<object-2.sum-1<<"\n";
cout<<"object-2.sum-2 is "<<object-2.sum-2<<"\n";
}
//sum::sum()是构造函数
```

```
sum::sum ()
{
    cout<<"Inside sum() \n";
}
```

编译并运行这个程序会产生如下结果:

```
Inside main()
Inside sum()
Inside sum()
object_1.sum_1 is 0
object_1.sum_2 is 8815
object_2.sum_1 is 1
object_2.sum_2 is 0
```

这次, 类成员都显示了一些无用的值, 这是因为类 `sum` 的对象是在 `main()` 中创建的, 它们的作用域不是全程的, 因此未初始化为 0。类成员怎样才能被初始化为指定的值呢? 答案是:

- 可以通过使用参数或默认参数的方式来初始化类成员。

下面讨论参数。

15.4 带参数的构造函数

请看 `test15-5.cpp`

```
// test15-5.cpp
#include <iostream.h>
class sum
{
    // 现在sum_1 和sum_2是私有的
    int sum_1;
    int sum_2;
public:
    //sum() 没有返回类型和参数
    sum (int i, int j);
};

void main(void)
{
    /*object_1和object_2创建时带有参数
    sum object_1(10, 20), object_2(20, 30);
    cout<<"Inside main() \n";
```

```
}  
sum::sum (int i,int j)  
{  
    //请注意对参数初始化  
    sum-1 = i;  
    sum-2 = j;  
    cout<<"sum-1 is "<< sum-1 <<  
        "sum-2 is "<< sum-2 <<"\n";  
}
```

sum-1 和 sum-2 被声明为私有的。因为我们所感兴趣的是利用 C++的数据隐藏特性。接着, 声明带参数的构造函数 sum():

```
sum(int i,int j);
```

在 main() 中, 用值代替参数来声明 object-1 和 object-2:

```
sum object-1(10,20), object-2(20,30);
```

在 sum() 中, 两个私有的类成员 sum-1 和 sum-2 被初始化为参数 i 和 j:

```
sum-1=i;
```

```
sum-2=j;
```

编译并运行这个程序得到如下输出结果:

```
sum-1 is 10 sum-2 is 20
```

```
sum-1 is 20 sum-2 is 30
```

```
Inside main()
```

在本程序(甚至子程序)中也可能产生警告, 指出 object-1 和 object-2 在 main() 中从未使用过。但这些警告可以忽略。因为该程序只用来演示与构造函数有关的特性, 代码写成这样也就足够了。

请注意, 每个对象的类成员初始化的顺序要与它们声明的顺序相同。首先创建的是 object-1 的类成员 sum-1 和 sum-2, 然后分别初始化为 10 和 20, 接着, 创建 object-2 的类成员 sum-1 和 sum-2, 分别初始化为 20 和 30。

15.5 带默认参数的构造函数

可以用不同方式来初始化类成员。

- 可以在构造函数声明中指定默认参数。如果没有指定其它的值, 那么, 将用这些默认值来初始化类成员。

下列程序是对刚才所讲内容的说明:

```
//test15-6.cpp  
#include <iostream.h>  
class sum  
{
```

```
/* 请注意现在sum-1 和sum-2是私有的
int sum-1;
int sum-2;
public:
/*注意默认的参数
sum (int i,int j = 5);
};
/**object-1创建时只带一个参数
sum object-1(10), object-2(20,30);
void main(void)
{
    cout<<"Inside main() \n";
}
sum::sum (int i,int j)
{
    sum-1 = i;
    sum-2 = j;
    cout<<"sum-1 is "<< sum-1 <<
        "sum-2 is "<< sum-2 <<"\n";
}
```

该程序输出如下:

```
sum_1 is 10  sum_2 is 5
sum_1 is 20  sum_2 is 30
Inside main()
```

构造函数 sum 的声明是:

```
sum(int i,int j=5);
```

第二个参数初始化为一个默认值。现在请注意 object-1 和 object-2 的创建形式:

```
sum object-1(10), object-2(20,30);
```

object-1 的类成员 sum-1 被初始化为 10, sum-2 默认为 5, object-2 的类成员 sum-1 和 sum-2 的值就无需再解释了。

15.6 重载构造函数

■ 构造函数可以重载, 也就是说, 构造函数可以用相同的名字却带有不同类型的参数。

编译器可以根据参数的类型来引用正确的构造函数。请看 test15-7.cpp

```
//test15-7.cpp
#include <iostream.h>
```

```
class sum
{
    int sum_1;
    double sum_2;
public:
    //注意函数名sum()被重载了
    sum(int i);
    sum(double k);
};

void main(void)
{
    /** 调用sum(int)
    sum object_1(10);
    /** 调用sum(double)
    sum object_2(10,5);
    cout<<"Inside main() \n";
}

sum::sum(int i)
{
    sum_1 = i;
    cout<<"Inside sum(int) \n";
}

sum::sum(double k)
{
    sum_2 = k;
    cout<<"Inside sum(double) \n";
}
```

编译并运行该程序会产生如下结果:

```
Inside sum(int)
Inside sum(double)
Inside main()
```

当声明 object_1 时, 引用了构造函数sum(int), 因为参数10 是整型(int)数。当声明object_2时, 引用构造函数 sum(double) 。

15.7 构造函数的调用顺序

下列程序演示了在处理基类和派生类时构造函数的调用顺序:

```
//test15-8.cpp
#include <iostream.h>
class one
{
public:
    one();
};
class two
{
    int b;
public:
    two(int j);
};
main(void)
{
    one object_1();
    two object_2(20);
    cout<<"Inside main() \n";
}
one::one ()
{
    cout<<"Inside one \n";
}
two::two (int j)
{
    b = j;
    cout<<"Inside two b is " << b << "\n";
}
```

类 two 是由类 one 派生出来的。在 main() 中，声明了类 one 的一个对象，也声明了类 two 的一个对象。构造函数 one 是一个缺省的构造函数，因为它没有参数。构造函数 two 只传递一个参数，编译并运行该程序产生如下输出结果：

```
Inside one
Inside two b is 20
Inside main()
```

构造函数调用的顺序不需解释也很清楚了。但是如果只创建一个派生类的对象而并不创建基类的对象又会怎样呢？请看 test15-9.cpp:

```
//test15-9.cpp
#include <iostream.h>
class one
{
public:
    one();
};
//two由one派生出来
class two : public one
{
    int b;
public:
    two(int j);
};

main(void)
{
    /**我们先不创建基类的对象
    two object_2(20);
    cout<<"Inside main() \n";
}

one::one ()
{
    cout<<"Inside one \n";
}

two::two (int j)
{
    b = j;
    cout<<"Inside two b is " << b <<"\n";
}
```

编译并运行该程序将产生如下输出:

```
Inside one
Inside two b is 20
Inside main()
```

如您所见，基类的构造函数还是被调用了。这是必要的（而且是有意义的）。因为类 two 是由类 one 派生的，必须首先构造 one（怎么能从一个根本不存在的对象派生出东西来呢？必须首先创建基类的对象）。

■ 首先构造的是基类的构造函数，然后才是派生类的构造函数。

请看下列函数，它说明了多基类时构造函数的调用顺序。

```
//test15-10.cpp
#include <iostream.h>
class one
{
public:
    one();
};
class three
{
public:
    three();
};
/**注意类two由多个基类派生出来
class two:public one,public three
{
    int b;
public:
    two(int j);
};

main(void)
{
    two object_2(20);
    cout<<"Inside main() \n";
}

one::one ()
{
    cout<<"Inside one \n";
}
two::two (int j)
{
    b = j;
```

```
    cout<<"Inside two b is " << b <<"\n";
}
three::three ( )
{
    cout << "Inside three \n";
}
```

编译并执行这个程序产生如下输出结果:

```
Inside one
Inside three
Inside two b is 20
Inside main()
```

如您所见, 基类的构造函数调用的顺序与它们声明的顺序是相同的:

```
class two : public one, public three
```

15.8 带参数的基类构造函数

如果想要创建一个派生类的对象, 而它的基类又需要参数该怎么办呢? 请看下面的代码就可解决这一问题。

```
//test15_11.cpp
#include <iostream.h>
class one
{
    int a;
public:
    one(int i);
};
/**类two是由基类one派生出来
class two: public one
{
    int b;
public:
    two(int j);
};

main(void)
{
    two object_2(20);
```

```
    cout<<"Inside main() \n";
}

one::one (int i)
{
    a=i;
    cout<<"Inside one a is " << a << "\n";
}
two::two (int j):one(j)
{
    b = j;
    cout<<"Inside two b is " << b << "\n";
}
```

如您所见，基类 one 的构造函数需要一个参数：

```
one(int I);
```

类 two 由类 one 派生出来：

```
class two : public one
```

创建类 two 的对象时可带有一个所需的参数。

```
two object_2(20);
```

接着，是派生类的构造函数的一个有趣的定义：

```
two::two (int j):one(j)
{
    .
    .
}
```

这是将参数传递给基类的一种方法之一。派生类构造函数的定义遵循通常的句法。只是要附加一个冒号(:) (表示派生)，一个基类名以及参数列表。

编译并执行该程序会产生如下输出结果：

```
Inside one a is 20
Inside two b is 20
Inside main()
```

构造函数的调用顺序（先基类，然后派生类）保持不变。

■ 参数也可以通过其它不同方式传递给基类。参数表中可以有常量。

请看下列程序：

```
//test15-12.cpp
```

```
#include <iostream.h>
class one
{
    int a;
public:
    one(int i);
};
/**类two是由基类one派生出来的
class two:public one
{
    int b;
public:
    two(int j);
};

main(void)
{
    two object_2(20);
    cout<<"Inside main() \n";
}

one::one (int i)
{
    a=i;
    cout<<"Inside one a is " << a << "\n";
}
/** 注意 one() 的常量参数
two::two (int j):one(10)
{
    b = j;
    cout<<"Inside two b is " << b << "\n";
}
```

类 two 的构造函数的定义:

```
two::two (int j):one(10)
```

有一个常量作为参数传递给了类 one 。编译并运行这个程序产生如下输出:

```
Inside one a is 10
Inside two b is 20
Inside main()
```

该输出结果无需再解释了。

■ 全程参数也可以传递给基类的构造函数。

请看 test15-13.cpp:

```
//test15-13.cpp
#include <iostream.h>
class one
{
    int a;
public:
    one(int i);
};
class two:public one
{
    int b;
public:
    two(int j);
};
/**注意外部变量声明
int k;

main(void)
{
    /**初始化外部变量
    k = 5;
    two object_2(20);
    cout<<"Inside main() \n";
}

one::one (int i)
{
    a=i;
    cout<<"Inside one a is " << a << "\n";
}
two::two (int j):one(k)
{
    b = j;
    cout<<"Inside two b is " << b << "\n";
```

```
}
```

在main()外部, k 声明为全程变量, 在 main() 中, k 初始化为 5, 然后被当作参数传递给基类的构造函数。

```
two::two (int j):one(k)
```

编译并执行这个程序将会产生如下输出结果:

```
Inside one a is 5
```

```
Inside two b is 20
```

```
Inside main()
```

该输出结果就无需解释了。

15.9 析构函数

与构造函数功能相反的是析构函数。

- 析构函数用来释放在类创建时分配给它的存储空间。
 - 析构函数只是在函数名前附加一个否定号(~), 与构造函数同名。
- 请看下列示例程序, 它说明了构造函数和析构函数的用法:

```
//test15_14.cpp
#include <iostream.h>
class one
{
    /** one() 是构造函数
    /** ~one() 是析构函数
    public:
        one();
        ~one();
};
main(void)
{
    cout<<"Inside main() \n";
    one object_1;
    cout<<"Inside main() -after creating object_1 \n";
}
/** one() 是构造函数
one::one ()
{
    cout<<"Inside one \n";
}
```

```
/** ~one() 是析构函数
one::~~one()
{
    cout<<"Inside ~one \n";
}
```

类 one 中有两个成员函数, 即 one() 和 ~one()。one() 是缺省的构造函数, 而~one() 则是析构函数。编译并运行这个程序会产生如下输出结果:

```
Inside main()
Inside one
Inside main() -after creating object-1
Inside ~one
```

在本例中, 从 main() 中退出时自动调用析构函数 ~one()。象构造函数一样, 即使没有定义析构函数, 也会调用。

- 与构造函数相同的是, 析构函数没有返回类型即没有返回值。
- 与构造函数不同的是, 析构函数不接受任何参数。
- 当变量离开作用域时, 析构函数被隐式地调用 (前面的示例程序就是这样的)。

对于局部变量, 只要不在程序块作用域内就会被释放掉; 对于全程变量, 只有当程序结束时才会释放。

15.10 析构函数的调用顺序

- 析构函数的调用顺序与构造函数的调用顺序正好相反。

请看下列程序:

```
//test15-15.cpp
#include <iostream.h>
class one
{
public:
    one();
    ~one();
};
/**注意类two由基类one派生出来
class two:public one
{
public:
    two();
    ~two();
```

```
};

main(void)
{
    cout<<"Inside main() \n";
    one object_1;
    cout<<"Inside main() -after creating object-1 \n";
}

/** ~one() 是基类的构造函数
one::~one ( )
{
    cout<<"Inside one \n";
}
/** ~one() 是基类的析构函数
one::~~one ( )
{
    cout<<"Inside ~one \n";
}
/** ~two() 是派生类的构造函数
two::~~two ( )
{
    cout<<"Inside ~two \n";
}
/** ~two() 是派生类的析构函数
two::~~two ( )
{
    cout<<"Inside ~two \n";
}
```

在这里，类 two 从类 one 中派生出来，编译并执行该程序将产生如下输出结果：

```
Inside main()
Inside one
Inside two
Inside main() -after creating object-1
Inside ~one
Inside ~two
```

在 `main()` 中，创建了类 `two` 的一个对象。而类 `two` 是由类 `one` 派生的。因此，首先构造类 `one`，然后才是类 `two`。接着，程序结束并退出。在这里，我们首先看到派生类（类 `two`）的析构函数被调用，然后才调用基类（类 `one`）的析构函数。

■ 析构函数遵循与构造函数相反的调用顺序。从逻辑上讲也应如此。

15.11 小 结

在本章中，我们学习了关于构造函数和析构函数的许多有趣之处。

- 构造函数是与它所在的类同名的函数，用来在创建对象时初始化对象。
- 在对象声明的时候，如果在程序中没有显式地调用析构函数，编译器会自动地调用。
- 可以给构造函数传递默认的参数。
- 构造函数可以被重载，因为编译器能自动引用其正确的形式。
- 构造函数以先基类后派生类的逻辑顺序调用，即基类的构造函数先于派生类的构造函数。
- 析构函数功能与构造函数相反。它与构造函数同名，只是在函数名前加一个否定符（`~`）。
- 析构函数释放那些通过构造函数分配给被创建对象的存储空间。
- 从 `main()` 中退出时，如果没有显式地调用析构函数，则会自动地调用。
- 析构函数不接受任何参数。
- 析构函数的调用顺序与构造函数正好相反，即：派生类的析构函数在基类的析构函数之前被调用。

第十六章 虚函数与多态性

16.1 简介

从本书的第一章到现在，我们已经讲述了很多内容，在第一章里，我们介绍过虚函数的概念。那时，读者未能弄懂 C++ 的这一强大特性。本章我们还要重新介绍虚函数的概念，并做更详细的说明。我们还要介绍多态性（Polymorphism）这个术语，以及早期联编（early binding）和运行时联编（run-time binding）等术语。到本章结束时，读者就会适应这些术语了。这些术语是 C++ 程序员和一些 C++ 书籍中常用的。这些概念就象读者以前学过的一样，其实并不难理解。

16.2 回顾一下指针

在讲述虚函数之前，对指针作简短的复习。这一点很有必要，因为 C++ 中的运行时多态性是通过指向基类和派生类的指针来实现的。

正如我们已知的那样，指针包含了某一个类型数据的内存地址。例如，在某个程序里声明了一个整型变量：

```
int a;    //a是一个整型数据
```

然后，指向这个变量的指针声明如下：

```
int *ptr; //ptr是一个整型指针
```

接着，指针 ptr 被赋值为变量 a 的内存地址（或位置）。

```
ptr=&a;    // ptr 被赋值为变量 a 的内存地址
```

一旦这样做了，就可以通过指针 ptr 来间接地存取变量 a 的内容，下面的两条语句产生同样的输出结果：

```
*ptr=5;    //通过指针 ptr 来间接地存取内容，a 赋值为 5。
```

```
a=5;    //直接地存取内容，a 赋值为5。
```

可以通过在指针 ptr 的前面加上一个 * 操作符来获得 a 的内存地址。下面的语句说明了这一点：

```
cout<<"At location"<<ptr<<"is stored"<<*ptr;
```

上述语句将在计算机上输出如下结果：

```
At location XXXXXX is stored 5
```

这里的 XXXXXX 是 a 的内存地址。对 ptr 的第一次引用输出了 a 的地址，对 ptr 的第二次引用（带 * 号）输出了存储在这个地址内的内容。

16.3 类指针

- C++ 具有字符指针、浮点指针、双精度指针、数组指针等等，在 C++ 中，有结构指针，还有类指针。

- 通过指针来存取类成员的句法与结构类型是一样的。

看下列程序：

```
//test16-1.cpp
#include <iostream.h>
class one
{
public:
void output(void)
{
    cout<<"Inside output \n";
}
};
main(void)
{
    //object-1 是类 one 的一个对象
    //pointer 是类 one 的指针
    one object-1;
    one *pointer;
    //pointer 赋值为对象 object-1的地址
    //调用output()
    pointer=&object-1;
    pointer->output();
    return(0);
}
```

编译并执行这个程序会产生如下输出结果:

Inside output

这个程序首先声明了类 one, 函数 output() 是内联的 (inline) (它在类的内部定义)。在 main() 中, object-1 是类 one 的一个对象。接下来, pointer 声明为指向类 one 的指针。然后, 将 pointer 赋值为对象 object-1 的地址。再往下, 通过如下语句来调用类 one 的成员函数 output() :

```
pointer->output();
```

->操作符是通过指针存取类成员的一种方法。下列语句也可以得出相同的结果。

```
pointer.output();
```

在这个程序中, 我们使用->操作符, 因为箭头很直观地表示指向了某个东西。这正是本程序要实现的功能。

16.4 派生类指针

■ 在C++中, 可以有指向基类的指针和指向派生类的指针。

- 可以用基类指针来存取派生类的成员，只要被存取的派生类的成员是从它的基类继承来的。

读了下面的程序后，您就会更清晰了：

```
//test16.2.cpp
#include <iostream.h>
class one
{
public:
void output(void)
{
    cout<<"Inside output \n";
}
};
//two 是 one 的派生类
class two:public one
{
public:
void output2(void)
{
    cout<<"Inside output2 \n";
}
};

main(void)
{
    //object-1 是类 one 的一个对象
    //object-2 是类 two 的一个对象
    //pointer 声明为基类 one 的指针
    one object-1;
    two object-2;
    one *pointer;
    //pointer 赋值为对象 object-1 的地址
    //执行object-1::output()
    pointer=&object-1;
    pointer->output();
    //现在，pointer 赋值为对象 object-2 的地址
    //执行object-2::output()
    pointer=&object-2;
```

```
    pointer->output();  
    return(0);  
}
```

编译并执行这个程序会产生如下输出结果:

```
Inside output  
Inside output
```

在这个程序中,首先是类 one 的定义。然后定义了由类 one 公共地派生出来的类 two。类 one 有一个成员函数 output(),类 two 有两个成员函数: output() (从类 one 继承而来)和 output2() (类 two 所独有的)。

在 main() 中,声明了基类 one 的指针 pointer。然后,将 pointer 赋值为对象 object-1 的地址,而 object-1 则是类 one 的一个对象,通过如下语句来执行函数 output():

```
pointer->output();
```

接下来,将 pointer 赋值为派生类 two 的对象 object-2 的地址:

```
pointer=&object-2;
```

然后,再次调用 output():

```
pointer->output();
```

编译器不产生任何出错信息,因为属于 object-2 的函数 output() 是从基类 one 中继承来的,如果把 test16-2.cpp 修改成通过基类的指针来调用 output2() (output2() 是类 two 所独有的成员函数),就会出现错误。该程序如下:

```
//test16-3.cpp  
#include <iostream.h>  
class one  
{  
public:  
void output(void)  
{  
    cout<<"Inside output \n";  
}  
};  
//two 是 one 的派生类  
class two:public one  
{  
public:  
void output2(void)  
{
```

```
    cout<<"Inside output2 \n";
}
};
main(void)
{
    //object-1 是类 one 的一个对象
    //object-2 是类 two 的一个对象
    //pointer 是基类 one 的指针
    one object-1;
    two object-2;
    one *pointer;
    //pointer 赋值为对象 object-1 的地址
    //调用 object-1::output()
    pointer=&object-1;
    pointer->output();
    //现在, pointer 赋值为对象 object-2 的地址
    //执行 object-2::output()
    pointer=&object-2;
    pointer->output2();
    return(0);
}
```

编译并执行这个程序会产生如下出错信息:

Error : output2 is not a member of one

编译器所指出的错误是针对下列语句的:

```
pointer->output2();
```

这是因为 pointer 被声明为基类 one 的指针, 而 output2() 则是类 two 所独有的, 不是从类 one 继承来的。

顺便提一下, 可以把基类指针强制地转换成派生类指针, 再通过该指针按如下形式就可成功地执行 output2():

```
((two *)pointer)->output2();
```

为不使读者感到混乱, 在本章里对这类语句暂不作详细说明。

■ 可以通过派生类指针来存取那些不是从基类继承来的派生类元素。

现在我们看一下这个程序的另一版本:

```
//test16-4.cpp
#include <iostream.h>
class one
```

```
{
public:
void output(void)
{
    cout<<"Inside output \n";
}
};
//two 是 one 的派生类
class two:public one
{
public:
void output2(void)
{
    cout<<"Inside output2 \n";
}
};
main(void)
{
    //声明了基类one的一个对象
    //声明了派生类的一个对象object-2
    //声明了基类的指针
    //声明了派生类的指针
    one object-1;
    two object-2;
    one *pointer;
    two *pointer2;
    //pointer赋值为对象object-1的地址
    //调用 object-1::output()
    pointer=&object-1;
    pointer->output();
    //现在, pointer2 服值为对象object-2的地址
    //调用 object-2::output()
    pointer2=&object-2;
    pointer2->output2();
    return(0);
}
```

编译并执行这个程序会产生如下输出结果:

Inside output

Inside output2

16.5 虚函数

介绍了基类指针和派生类指针后，我们再讨论虚函数 (Virtual Function)。对弄明白怎样通过指针来存取类成员是很有必要的，因为虚函数的执行要借助于这种基本机制。

多态性 (Polymorphism) 这个术语究竟是什么意思呢？我们参考一下手边的韦伯斯特的百科全书大词典，摘录出如下解释：

“…动物或植物以不同形式或 颜色的多种存在形式…”

“…状态或条件的多种形式…”

同时，多态性还有如下涵义：

“…具有、呈现或经过的许多各种各样的形式…”

请花一分钟时间考虑一下第一条定义。以玫瑰为例，凡是玫瑰都差不多，但是有些是红的，有些是白的，而有些是粉的，或许将来某一天某个人培植出黑色的或其它具有罕见颜色的玫瑰来也未可知。

现在我们把玫瑰作为一个对象来考虑。我们知道，玫瑰之所以是玫瑰，正是因为它具有某种确定的形状，花瓣是软的，还具有与众不同的花束，而且枝干上有刺等等。现在 we 并不关心这株玫瑰是怎样出现的，或者谁培植的。但是姑且让我们将它的存在归因于大自然吧。大自然创造出的一枝花具有这种独特形状，并赋予使其之所以成为一枝玫瑰的其余的性状。一切就绪之后，假设大自然留出颜色这一特性来让其它实体操纵，大自然就要为该这一实体提供一种无色的玫瑰，或具有某种默认颜色（如红色）的玫瑰，并允许该实体确定一种确切的方法来为那朵玫瑰赋予特定的颜色。该实体将以其独特的方式来确定该玫瑰的颜色，并将这颜色返回给大自然。每当要为一朵玫瑰赋予一种新颜色时，大自然就送给该实体一朵具有默认颜色的玫瑰，而当颜色赋予完毕，玫瑰就被送回给大自然。

每次送出玫瑰的接口即互连方式是相同的：一朵没有颜色或具有默认颜色的玫瑰。从实体中的返回对象也总是相同的：一朵具有颜色的玫瑰。然而上色的方法是不确定的，即赋予每种颜色的方法是不同的。如果要生成一种绿玫瑰，那么就需要将黄色和蓝色混合，如果要生成一种紫玫瑰，那么就需要将红色和蓝色混合，等等。接口只有一个，但实现手段却有多种。按词典定义，可以说玫瑰是多态的，因为它可以以几种颜色的变体存在。更进一步说，这种一个接口、多种实现的概念就是理解C++多态性的关键。

现在，我们将在这种一个接口、多种实现概念的基础上，用另外一个例子和一些示例程序来帮助读者加深理解C++ 这种强大的面向对象的机制。

我们已经了解到一个小的旅行社专门包揽加勒比海的假期旅游生意。当该旅行社最初着手准备时，老板决定设计一个C++ 程序来列出每项包揽生意中所含的加勒比海的每个旅游景点独到的揽胜之处。指派一个精明的程序员来完成此项任务，他懂得利用类的优势来代替结构，所以决定构造一个类层次，其中的基类用来描述所有加勒比海岛屿的共同特性，然后每个小岛都由基类以及针对该岛屿特有的名胜进行描述的派生类来表达。

该程序设计得尽可能地灵活，以使变化的内容很容易就加进去。如果台风袭击了某个岛屿，或另一个岛屿发生地方局势紧张又该怎么办呢？很显然，这些岛屿就不得不从旅游包揽表中去掉。同时，随着公司的壮大，又可加入新的包揽业务。我们所需要的就是这样一个程序，该程序可以加入或删除一些内在相同（即列出的名胜）而实现不同（每个岛屿的不同的名胜综合）的函数。为满足这些需要，显然虚函数是最好的答案。下面是程序的内容：

```
//test16-5.cpp
#include <iostream.h>
//caribbean-isles will be main base class
class caribbean-isles
{
public:
/**请注意关键字 virtual
virtual void other_attractions(void)
{
    cout<<"!!COME VISIT THE CARIBBEAN ISLANDS!! \n";
    cout<<"White sandy beaches \n";
    cout<<"Crystal clear water \n";
    cout<<"No hassles - No worries - No cares \n\n";
}
};
```

我们将该程序的代码分解成为几个逻辑段，因为它比本书中其它大多数程序都长。程序开始就声明一个叫做 caribbean-isles 的基类。该类包含了一个叫做 other_attractions() 的公共的函数，定义这一函数要以关键字 virtual 做前缀，它是内联函数（并不一定非要这样做），用来输出那些在包揽生意表中所有加勒比海岛屿的共同特性，包含 White sandy beaches; crystal clear water, 以及 no hassles, worries or cares。

下面让我们继续这段代码：

```
//bahamas, grand_cayman 和 st_thomas 是由类caribbean-isles派生的
class bahamas:public caribbean-isles
{
public:
/**请注意同一函数名，无关键字virtual
void other_attractions(void)
{
    cout<<"!!BAHAMAS!! \n";
    cout<<"Fun-filled casinos \n";
```

```
    cout<<"Action-packed water sports \n";
    cout<<"Great rum punches \n\n";
}
};

class grand_cayman:public caribbean-isles
{
public:
    /**请注意同一函数名, 无关键字virtual
void other_attractions(void)
    {
        cout<<"!!GRAND CAYMAN!! \n";
        cout<<"Incredible scuba diving \n";
        cout<<"Big time game fishing \n";
        cout<<"Duty free shopping \n\n";
    }
};

class st_thomas : public caribbean-isles
{
public:
    /**请注意同一函数名, 无关键字virtual
void other_attractions(void)
    {
        cout<<"!!ST THOMAS!! \n";
        cout<<"Excellent shopping \n\n";
    }
};
```

接着, bahamas, grand_cayman 和 st_thomas 是由类caribbean-isles派生出来。这些岛屿继承了包含在类caribbean-isles中的一切特性, 然后又加入了它们自己的特性。要注意的是; 这里的每个类都包含了一个与基类中虚函数名称相同的成员函数 (other_attractions())。除此之外, 返回类型也是相同的 (void)。而且参数也相同 (也是void)。然而, 每个函数的实现都是不同的 (即每个方法中都具有一组不同的名胜)。现在请看看主函数main()的代码:

```
main(void)
{
    /**声明类caribbean-isles的对象islands和指针ptr
```

```
caribbean-isles islands, *ptr;
//声明派生类的对象
bahamas package-1;
grand-cayman package-2;
st-thomas package-3;
//ptr被设置为基类对象的地址
ptr=&islands;
//调用虚函数
//编译器知道该选择哪一个版本来执行
ptr->other_attractions();
//现在 ptr 被设置为派生类对象的地址
ptr=&package-1;
ptr->other_attractions();
ptr=&package-2;
ptr->other_attractions();
ptr=&package-3;
ptr->other_attractions();
return(0);
}
```

在main()中, 声明了基类和派生类的对象, 以及基类的指针。然后, ptr 被赋值为基类对象的地址, 并执行了函数 other_attractions()。但究竟会执行该函数的哪个版本呢? C++根据指针所指向的对象类型, 在运行时而不是在编译时决定究竟执行该函数的哪个版本。我们称之为后期联编 (late binding) 和运行时多态性 (run-time polymorphism)。这些术语现在就开始变得有意义了。

编译并执行这个程序会产生如下输出结果:

```
!!COME VISIT THE CARIBBEAN ISLANDS!!
```

```
White sandy beaches
```

```
Crystal clear water
```

```
No hassles - No worries - No cares
```

```
!!BAHAMAS!!
```

```
Fun-filled casinos
```

```
Action-packed water sports
```

```
Great rum punches
```

```
!!GRAND CAYMAN!!
```

```
Incredible scuba diving
```

Big time game fishing

Duty free shopping

!!ST THOMAS!!

Excellent shopping

现在我们来解释一下这个输出结果。

首先, ptr 指向对象 islands, 也就是类 caribbean-isles 的一个实例。于是, 语句:

```
ptr->other_attractions();
```

执行属于基类的 other_attractions() 那个版本。

接着, ptr 指向对象 package-1, 即类 bahamas 的一个实例。于是, 语句:

```
ptr->other_attractions();
```

执行属于派生类 bahamas 的 other_attractions() 版本。

读者自己一步步地分析程序的其余部分, 就会弄清执行的是哪一个版本, 以及为什么执行这个版本。

16.6 什么是虚函数

- 虚函数是指在基类中声明的, 以后又可在由此基类派生出来的一个或多个类中重新定义的函数 (不久就会看到, 不一定必须重新定义)。
- 根据指针所指的对象, 在运行时选择正确的函数来实现, 而不是在编译时选择。
- 在基类中, 虚函数要以关键字 virtual 为前缀, 这种函数在其派生类中可重定义。
- 函数名、返回类型以及函数参数必须与基类中的原型完全相同, 否则就会失去这些函数的虚特性。

正是基于这个原因, 我们不用重载 (overloaded) 这个术语来描述这些特殊的函数 (请回忆一下, 重载函数具有相同的名字, 但具有不同的参数)。相反, 该函数在派生类中是废弃 (overriden) 的。

- 通过指向基类的指针来访问这些函数, 做到运行时多态性。

基类中所定义的函数对于它的派生类来说都是公共的。它提供了派生类中这些函数的统一接口。实现这一函数的真正方法在派生类中定义。

16.7 什么是早期联编和后期联编

- 早期联编表明了程序编译时就可确定要发生事件的出现。

早期联编的例子是对标准重载函数的调用。实现标准的重载函数所需的信息是在程序编译时得知的。

- 后期联编表明了程序运行时才可确定要发生事件的出现。

后期联编的例子包括虚函数。实现虚函数所需的信息是在程序运行时而不是在编译时得知的。让我们更进一步地解释一下:

请看如下这个简短的程序:

```
//test16-6.cpp
main(void)
{
    int a=5; //第一条语句
    a +=2;   //第二条语句
}
```

编译并执行这个程序既不产生错误信息，又不产生输出结果；这个例子虽然简单，然而却说明了两个要点：

第一个语句在编译时执行，当test16-6.cpp编译时，编译器向变量a存入一个整数，存入的是数值5。

第二个语句在运行时执行，当test16-6.cpp运行时，变量a中的值加上了2。

为什么说虚函数是一种运行时的现象呢？现在已经知道，究竟要执行哪个函数是由当时指针所指的对象决定的。ptr 的值是在运行时修改的。因此，要执行函数的那个版本也是在运行时确定下来的。

16.8 小 结

在本章里，我们学习了什么是虚函数以及为什么要用虚函数。主要学习内容：

- 如何用基类和派生类指针来引用函数类成员。
- 基类指针可用来存取派生类成员，只要这些类成员是从基类中继承而来。
- 虚函数怎样使用基类指针存取不同类型的对象。早期联编和后期联编用来描述这一现象。

第十七章 虚函数与抽象类

17.1 简介

在前一章里，我们又讲述了虚函数的概念。解释了如何调用虚函数以及为什么要这样做。在这一章里，我们在这些概念的基础上，介绍一下纯虚函数与抽象类。

17.2 虚函数的灵活性

在第十六章里，曾经设计过一个 C++ 程序，在该程序中列出了旅行社所提供的假期旅游包揽表中的主要游览胜地。设计这个程序所用的方法非常灵活，以至于不需要对主程序做更多的修改就足以方便地处理好旅游包揽表中各个岛屿的增减。这一目标是通过使用虚函数达到的。为方便起见，在这里我们列出该程序的全部内容：

```
//test16-5.cpp
#include <iostream.h>
//caribbean-isles will be main base class
class caribbean-isles
{
public:
/**注意关键字 virtual
virtual void other_attractions(void)
{
    cout<<"!!COME VISIT THE CARIBBEAN ISLANDS!! \n";
    cout<<"White sandy beaches \n";
    cout<<"Crystal clear water \n";
    cout<<"No hassles - No worries - No cares \n\n";
}
};
//bahamas, grand_cayman 和 st-thomas 是由类caribbean-isles派生的
class bahamas:public caribbean-isles
{
public:
/**请注意同一函数名，无关键字 virtual
void other_attractions(void)
{
    cout<<"!!BAHAMAS!! \n";
    cout<<"Fun-filled casinos \n";
    cout<<"Action-packed water sports \n";
```

```
        cout<<"Great rum punches \n\n";
    }
};

class grand_cayman:public caribbean_isles
{
public:
    /**请注意同一函数名, 无关键字virtual
void other_attractions(void)
    {
        cout<<"!!GRAND CAYMAN!! \n";
        cout<<"Incredible scuba diving \n";
        cout<<"Big time game fishing \n";
        cout<<"Duty free shopping \n\n";
    }
};

class st_thomas:public caribbean_isles
{
public:
    /**请注意同一函数名, 无关键字 virtual
void other_attractions(void)
    {
        cout<<"!!ST THOMAS!! \n";
        cout<<"Excellent shopping \n\n";
    }
};

main(void)
{
    /**声明类 caribbean_isles 的对象 islands 和指针 ptr
    caribbean_isles islands,*ptr;
    //声明派生类的对象
    bahamas package-1;
    grand_cayman package-2;
    st_thomas package-3;
    //ptr被赋值为基类对象的地址
    ptr=&islands;
    //调用虚函数
    //编译器知道该选择哪一个版本来执行
    ptr->other_attractions();
```

```
//ptr被赋值为派生类对象的地址
ptr=&package-1;
ptr->other_attractions();
ptr=&package-2;
ptr->other_attractions();
ptr=&package-3;
ptr->other_attractions();
return(0);
}
```

现在我们假定要增加第四个旅游胜地，将原来的程序代码修改如下：

```
//test17-1.cpp
#include <iostream.h>
//caribbean-isles will be main base class
class caribbean-isles
{
public:
/**注意关键字 virtual
virtual void other_attractions(void)
{
    cout<<"!!COME VISIT THE CARIBBEAN ISLANDS!! \n";
    cout<<"White sandy beaches \n";
    cout<<"Crystal clear water \n";
    cout<<"No hassles - No worries - No cares \n\n";
}
};
//bahamas, grand.cayman 和 st-thomas 是由类caribbean-isles派生的
class bahamas:public caribbean-isles
{
public:
/**请注意同一函数名，无关键字virtual
void other_attractions(void)
{
    cout<<"!!BAHAMAS!! \n";
    cout<<"Fun-filled casinos \n";
    cout<<"Action-packed water sports \n";
    cout<<"Great rum punches \n\n";
}
}
```

```
};  
class grand_cayman : public caribbean_isles  
{  
public:  
/**请注意同一函数名, 无关键字 virtual  
void other_attractions(void)  
{  
    cout<<"!!GRAND CAYMAN!! \n";  
    cout<<"Incredible scuba diving \n";  
    cout<<"Big time game fishing \n";  
    cout<<"Duty free shopping \n\n";  
}  
};  
class st_thomas:public caribbean_isles  
{  
public:  
/**请注意同一函数名, 无关键字 virtual  
void other_attractions(void)  
{  
    cout<<"!!ST THOMAS!! \n";  
    cout<<"Excellent shopping \n\n";  
}  
};  
class st_lucia : public caribbean_isles  
{  
public:  
/**请注意同一函数名, 无关键字 virtual  
void other_attractions(void)  
{  
    cout<<"!!ST LUCIA!! \n";  
}  
};  
  
main(void)  
{  
    /**声明类 caribbean_isles 的对象 islands 和指针ptr  
    caribbean_isles islands,*ptr;  
    /**声明派生类的对象  
    bahamas package-1;
```

```
grand-cayman package-2;
st-thomas package-3;
/**类 st-lucia 的对象声明加到这里
st-lucia package-4;
//ptr 赋值为基类对象的地址
ptr=&islands;
//调用虚函数
//编译器知道该选择哪一个版本来执行
ptr->other_attractions();
//现在 ptr 分别赋值为派生类对象的地址
ptr=&package-1;
ptr->other_attractions();
ptr=&package-2;
ptr->other_attractions();
ptr=&package-3;
ptr->other_attractions();
/**类 st-lucia 的对象的指针的赋值加到这里
ptr=&package-4;
ptr->other_attractions();
return(0);
}
```

从上面程序可以看到，一个新的类 `st-lucia` 加到了程序中（这是包揽表中的最后一个名胜），该类中也包含了它自己的 `other_attractions()` 函数的版本。现在来看看主函数 `main()` 的代码。

下面这几行代码是对主函数 `main()` 仅有的一点改动，这些代码用来调用新的派生类函数：

```
st-lucia package-4;
和
ptr=&package-4;
ptr->other_attractions();
```

该程序的输出结果与前一章中 `test16_5.cpp` 的结果基本相同，只是多了一行：`!!ST LUCIA!!`。该程序表明了用这种办法把一个新函数并入到程序中是很容易的。如果想把某一虚函数从程序逻辑中去掉也是同样容易，只需在主函数中去掉对这一派生类的指针赋值就可以了。

17.3 虚函数特例

在加勒比海的中心有一个小岛，我们对它所知甚少，只知道这个离奇的小地方有着雄伟的山峦、白沙海滩以及惊险的海底悬崖。这个小岛名叫 `Saba`。

假定要将这个小岛并入到主要的游览胜地中，称之为mystery package（神秘岛）。然而，此时没有人非常清楚如何将这些游览胜地归类。那么怎样处理这种情况呢？这一点我们还没有讲到。

■ 如果虚函数在派生类中没有定义，将自动地按基类中的虚函数去执行。

或换一种说法，如果派生类没有对基类中的虚函数提供重载函数，那么当这个派生类的对象使用该虚函数时，将调用基类的虚函数。我们修改一下最初的程序来说明一下这种情况：

```
//test17-2.cpp
#include <iostream.h>
//caribbean-isles 是基类
class caribbean-isles
{
public:
//**我们将标题分开输出
void header(void)
{ cout<<"!!COME VISIT THE CARIBBEAN ISLANDS!! \n";}
virtual void other_attractions(void)
{
    cout<<"White sandy beaches \n";
    cout<<"Crystal clear water \n";
    cout<<"No hassles - No worries - No cares \n\n";
}
};
```

上述程序是对基类修改后的定义。把原来函数中的头一行输出与其余部分分开（分成两个函数）。没有把函数 header() 声明为virtual，因为没有这个必要。我们继续看这个程序：

```
//bahamas, grand-cayman 和 st-thomas 是 caribbean-isles 的派生类
//** 类saba也这样
class bahamas:public caribbean-isles
{
public:
//**请注意同一函数名，无关键字virtual
void other_attractions(void)
{
    cout<<"!!BAHAMAS!! \n";
    cout<<"Fun-filled casinos \n";
```

```
    cout<<"Action-packed water sports \n";
    cout<<"Great rum punches \n\n";
}
};
class grand_cayman:public caribbean_isles
{
public:
    /**请注意同一函数名, 无关键字virtual
    void other_attractions(void)
    {
        cout<<"!!GRAND CAYMAN!! \n";
        cout<<"Incredible scuba diving \n";
        cout<<"Big time game fishing \n";
        cout<<"Duty free shopping \n\n";
    }
};
class st_thomas:public caribbean_isles
{
public:
    /**请注意同一函数名, 无关键字virtual
    void other_attractions(void)
    {
        cout<<"!!ST THOMAS!! \n";
        cout<<"Excellent shopping \n\n";
    }
};
class saba:public caribbean_isles
{
public:
    void name(void) { cout<<"!!MYSTERY ISLAND SABA!! \n"; }
    /** 函数 other_attractions() 没有在这里定义
};
```

类 saba 也象其它类一样来定义。然而, 其成员函数 other_attractions() 的属性还未定下来。这些属性期望会在将来的某个时候知道。现在, 这个神秘岛只是继承了它的基类的属性。继续看一下main():

```
main(void)
{
    /**声明类caribbean_isles的对象islands和指针ptr
```

```
caribbean-isles islands,*ptr;
//声明派生类的对象
bahamas package-1;
grand-cayman package-2;
st-thomas package-3;
saba mystery-package;
//输出标题
islands.header();
//ptr赋值为基类对象的地址
ptr=&islands;
//调用虚函数
//编译器知道该选择哪一个版本来执行
ptr->other_attractions();
//现在 ptr 赋值为派生类对象的地址
ptr=&package-1;
ptr->other_attractions();
ptr=&package-2;
ptr->other_attractions();
ptr=&package-3;
ptr->other_attractions();
/**输出神秘岛的名字
/**ptr 赋值为派生类saba的对象mystery-package的地址
mystery-package.name();
ptr=&mystery-package;
/**调用 caribbean-isles::other_attractions()
ptr->other_attractions();
return(0);
}
```

类saba的一个对象命名为mystery-package, 列出其它岛屿的摘要说明之后, 就输出这个神秘岛的名字。指针 ptr 指向该类的对象, 接着, 调用函数 other_attractions()。下面是该程序的输出结果:

!!COME VISIT THE CARIBBEAN ISLANDS!!

White sandy beaches

Crystal clear water

No hassles - No worries - No cares

!!BAHAMAS!!

Fun-filled casinos

Action-packed water sports

Great rum punches

!!GRAND CAYMAN!!

Incredible scuba diving

Big time game fishing

Duty free shopping

!!ST THOMAS!!

Excellent shopping

!!MYSTERY ISLAND SABA!!

White sandy beaches

Crystal clear water

No hassles - No worries - No cares

上例说明，在派生类中不存在废弃的（overriding）函数时，基类中的虚函数就变为可以调用的默认函数。以后函数other_attractions() 还可能被新的派生类所定义，这时，当语句

```
ptr->other_attractions();
```

被调用时，

```
saba::other_attractions();
```

将代替caribbean_isles::other_attractions() 来执行。

17.4 纯虚函数与抽象类

纯虚函数（pure virtual function）与前一小节中的情况恰恰相反。有时，并没有定义基类中虚函数的属性。纯虚函数就是在这种情况下产生的。

- 纯虚函数仅仅用来为将来要从基类中派生的函数占据一个位置。
- 纯虚函数在基类中没有定义；它们被初始化为 0。
- 任何由纯虚函数的类派生出来的类都要自己提供该函数的具体实现。

纯虚函数的用法在 test17_3.cpp 中说明：

```
//test17-3.cpp
#include <iostream.h>
//caribbean_isles 是基类
class caribbean_isles
{
public:
//**标题分开输出
```

```
void header(void)
{ cout<<"!!COME VISIT THE CARIBBEAN ISLANDS!! \n";}
//other_attractions() 的属性或者不存在,
//或者有个简单的地名
//因此, 我们把它定义为纯虚函数
virtual void other_attractions(void)=0;
};
```

上面程序中, 基类中的这个虚函数other_attractions()被初始化为0。它是个纯虚函数。

■ 声明一个纯虚函数, 其结果是, 在它的每个派生类中都必须给出它们自己对该函数的实现。

我们继续看这个程序:

```
//bahamas, grand_cayman 和 st_thomas 是由类caribbean_isles派生的
class bahamas:public caribbean_isles
{
public:
/**请注意同一函数名, 无关键字virtual
void other_attractions(void)
{
    cout<<"!!BAHAMAS!! \n";
    cout<<"Fun-filled casinos \n";
    cout<<"Action-packed water sports \n";
    cout<<"Great rum punches \n\n";
}
};

class grand_cayman:public caribbean_isles
{
public:
/**请注意同一函数名, 无关键字virtual
void other_attractions(void)
{
    cout<<"!!GRAND CAYMAN!! \n";
    cout<<"Incredible scuba diving \n";
    cout<<"Big time game fishing \n";
    cout<<"Duty free shopping \n\n";
}
};

class st_thomas:public caribbean_isles
```

```
{
public:
/**请注意同一函数名，无关键字virtual
void other_attractions(void)
{
    cout<<"!!ST THOMAS!! \n";
    cout<<"Excellent shopping \n\n";
}
};
```

现在看一下main()的头几行:

```
main(void)
{
    /**只声明了类caribbean-isles的指针ptr
    /**我们不能创建一个带有纯虚函数的类的对象
    caribbean-isles *ptr;
    如您所见，没有声明该基类的对象。这是因为:
    ■ 含有纯虚函数的基类叫做抽象类 (abstract)。
    ■ 不能声明抽象类的对象。
    ■ 允许声明含有纯虚函数的类指针，这对实现运行时多态性是很有必要。
    下面是其余的程序:
```

```
//声明派生类的对象
    bahamas package-1;
    grand-cayman package-2;
    st_thomas package-3;
    //输出基类的 header() 函数
    //但要作为派生类的一个元素
    package.header();
    //ptr被赋值为基类对象的地址
    ptr=&islands;

    //调用虚函数
    //编译器知道该选择哪一个版本来执行
    ptr->other_attractions();
    ptr=&package-2;
    ptr->other_attractions();
    ptr=&package-3;
    ptr->other_attractions();
```

```
    return(0);  
}
```

对于其余的代码就没有什么变化了。如果我们忘记了在某个派生类中给出纯虚函数的具体实现，假定是 `st-thomas`，编译器将产生下列错误信息：

```
Error: pure function 'caribbean-isles:: other_attractions()' not overridden in  
'st-thomas'
```

因此，

- 在程序中，当需要派生类给出它们自己的关于基类虚函数的版本时，可以用纯虚函数来作为一种安全机制。

17.5 小 结

在本章里，我们围绕着虚函数进行了讨论。我们学习了：

- 根据当前的需要，非常容易地将新的虚函数加入到程序中或从程序中删掉过时的虚函数。
- 在派生类没有提供虚函数的具体实现的实例中，基类中的虚函数将被引用。
- 对于纯虚函数，程序员必须在派生类中给出该虚函数的具体实现。
- 包含纯虚函数的类叫抽象类。不能声明该类的对象，只能声明该类的指针。在抽象类中的虚函数用来为它的派生类占据一个位置。

第十八章 运算符重载

18.1 简介

在这一章里，我们要描述 C++ 的一种独特而有趣的特性：运算符重载 (operator overloading)。

■ 运算符重载允许改变运算符的含义，象 +、-、*、/ 以及其它运算符等等。

如果读者花点时间仔细考虑一下重载运算符的理由，就会发现这一工具多么强大。另一方面，不加选择地使用重载运算符会导致调试过程陷入混乱之中。在学习本章的其余部分时，也要记住这一点。

18.2 运算符重载很普通

下面有六行程序：

```
//test18-1.cpp
#include <iostream.h>
main(void)
{
    cout<<"Notice the use of the << operator!! \n";
}
```

编译并运行该程序将得到如下的输出结果：

Notice the use of the << operator!!

这个结果没什么神秘的。现在看一下test18-2.cpp。在这里，根据上下文之间的逻辑关系，运算符 << 被用作左移位运算符：

```
//test18-2.cpp
#include <iostream.h>
main(void)
{
    int a=2;
    cout<<"a is "<<a<<"\n";
    //将a左移12位
    a=a<<12;
    cout<<"a is "<<a<<"\n";
}
```

编译并运行该程序将得到如下的输出结果：

```
a is 2
a is 8192
```

在这个程序里，运算符<<的用法很有意思。在第一个例子中，它作为输出运算符，在第二个例子中的下列语句

```
a=a<<12;
```

把它作为左移位运算符。它把 a 左移12位；之后，a中所存储的就值由2变为8192，并把它输出。

刚才我们举的是一个运算符重载的例子。运算符重载是C++具有的一个特性，它允许改变运算符的含义。在第一个例子中，把运算符<<作为输出运算符来用，而在第二个例子中，既把它作为左移位运算符，又把它作为输出运算符来用。

运算符重载是个有趣的概念。但读者会惊讶地发现，其实运算符重载一直就存在着，只是没有注意到。例如两个整数相加用 + 运算符：

```
a=5+2;
```

两个浮点数相加也用这个相同的 + 运算符：

```
a=5.5+2.8;
```

可以使用相同的运算符对两种不同的数据类型实现相同的功能。运算符 + 被重载。在第一个例子中，用它来实现两个整型数的加法运算；而在第二个例子中，则用它来实现两个浮点型数的加法运算。

这里还有另外一个例子，运算符 * 用来实现两个数的乘法运算：

```
a=a*b;
```

而运算符 * 也可以用在指针声明中，用来声明一个指针类型：

```
int *ptr;
```

上述语句将 ptr 声明为一个整型指针。运算符 * 也可以用来间接地存取一个指针变量，以便能操作 ptr 所指的内存地址中所存的内容：

```
*ptr=5;
```

上述语句将存储在ptr所指的内存地址中的内容赋值为 5，这是运算符重载的另一个例子。

18.3 运算符重载的句法

下面是个简短的程序：

```
//test18-3.cpp
#include <iostream.h>
class assign
{
public:
    int a;
};
```

```
assign object_1, object_2;
main(void)
{
    object_1.a=5;
    object_2=object_1;
    cout<<"object_2.a is "<<object_2.a<<"\n";
}
```

编译并运行该程序将得到如下的输出结果:

```
object_2.a is 5
```

该程序只不过是简单地给类成员赋以整型值, 然后用 `cout` 语句用来在屏幕上显示这个值。再看下面的程序, 它说明了运算符重载。这次我们将要重载赋值运算符(=)。

```
//test18_4.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**请注意重载函数的句法
    void operator=(assign var1);
};
//这个重载运算符函数没有返回值(类型为void)
//它带有一个参数, 是类assign的对象
//它遵循普通的函数定义规则
void assign::operator=(assign var1)
{
    a=2+var1.a;
}
//object_1和object_2是类assign的对象
assign object_1, object_2;

main(void)
{
    //将对象object_1的类成员 a 赋值为 5
    object_1.a=5;
    //将对象object_2设置成与对象object_1相等
    object_2=object_1;
```

```
cout<<"object-2.a is "<<object-2.a<<"\n";  
}
```

编译并运行该程序将得到如下的输出结果:

```
object-2.a is 7
```

这个程序的输出很有意思。在正常情况下, object-2.a应该是 5, 不是 7, 因为 object-2 与对象 object-1 相等。object-1.a 在此之前被赋值为 5。这种异常输出的原因是运算符 = 被重载为左操作数等于右操作数加 2。这就是object-2.a 等于 7 而不是 5 的原因。我们仔细从头看一下这个程序, 弄清楚它做了哪些事。首先看一下类assign的声明:

```
class assign  
{  
public:  
    int a;  
    /**请注意重载函数的句法  
    void operator=(assign var1);  
};
```

请注意其中的成员函数 assign() 的句法:

```
void operator=(assign var1);
```

该语句说明了存在一个运算符重载函数。它是类assign的成员函数。被重载的运算符是赋值运算符 = 。这个运算符重载函数没有返回值 (类型为void); 它带有一个参数, 是类 assign 的对象。该函数也可以声明如下:

```
assign operator=(void);
```

这个语句表示该函数有一个返回值, 是类assign的对象; 它没有参数。该函数也可以这样声明:

```
int operator=(char);
```

或者使用任何合法的返回值或参数类型。在这个例子中, 该函数声明为:

```
void operator=(assign var1);
```

这个类成员函数的定义如下:

```
void assign::operator=(assign var1)  
{  
    a=2+var1.a;  
}
```

一般来说, 是将运算符右边的对象赋值给运算符左边的对象。在该函数中, 类成员a 属于对象object-2。该函数的定义与类声明中的函数原型一致。该函数没有返回值, 它是类 assign 的成员 (通过函数名 assign::operator=前面的作用域分辨符 :: 就可以看出来)。它重载了运算符 =, 并且带有一个类 assign 的对象作为参数。

函数本身只有一行。它接收类对象var1作为参数。并把它的成员var1.a加上 2, 接着赋值给类成员 a, 然后返回到main()中。

在 `main()` 中, 将对象 `object-1` 赋值给对象 `object-2`, 这是个合法的赋值, 因为两个对象属于同一个类。最后, 输出 `object-2.a` 的内容。

得到这个异常结果的原因是: 对于含有运算符重载成员函数的这个类来说, 其对象的操作使用了被重载过的运算符 `=`。下面让我们再认真地复习一下这个概念。

`object-1` 和 `object-2` 是类 `assign` 的对象。运算符重载函数 `=` 是类 `assign` 的一个类成员。运算符重载的规则如下:

■ 如果类对象的类型定义中定义有重载函数, 那么, 每次调用或使用这些运算符来对该类的对象进行操作时, 将用重载函数的定义取代通常的运算。

(也可以取代友元函数的定义, 这一点将在本章的后面讨论)

现在再来看一下 `main()` 的代码:

```
assign object-1, object-2;
main(void)
{
    //将对象object-1的类成员 a 赋值为 5
    object-1.a=5;
    //将对象object-2赋值为与对象object-1相等
    object-2=object-1;
    cout<<"object-2.a is "<<object-2.a<<"\n";
}
```

`object-1` 和 `object-2` 被声明为类 `assign` 的对象。在 `main()` 中, 类 `assign` 的成员 `a` 被赋值为5; 接着, 将对象 `object-1` 赋值给对象 `object-2`, 而运算符 `=` 在前面已经声明为类 `assign` 的成员函数, 因为 `object-1` 和 `object-2` 是同一类 (类 `assign`) 的对象, 编译器将用特定的重载运算符来实现这些代码。

在弄清楚函数本身所实现的逻辑之前, 应该首先注意一下第一条赋值语句:

```
object-1.a=5;
```

实际上赋给 `object-1.a` 的是5, 并未加2, 这是因为:

■ 重载运算符的左右两个操作数必须与运算符的重载函数 (或友元函数) 所拥有的类型相同, 重载运算符的代码才能执行。

在程序中, 右边的操作数是个常量, 不是类 `assign` 的对象成员, 因此, 采用常规的赋值操作来实现。这说明了当上述条件不满足时, 运算符保留它们的已有特性。(对于单目运算符, 即只对一个操作数来运算的运算符, 例如 `++` 和 `--`, 也可以通过运算符重载来操作属于同一个类的操作数。这一点将在本章的后面几节里讨论)。

现在再看一下被重载的运算符函数的代码:

```
void assign::operator=(assign var1)
{
    a=2+var1.a;
}
```

上述语句中, `assign` 接收了一个名为 `var1` 的参数。这是类 `assign` 的对象模板。将该对象的成员 `a` 加上2, 然后将结果赋值给第二个对象的成员 `a`。一直到现在读者一定还

是很疑惑，究竟是哪一个对象被加，又是哪一个对象被赋值呢？这个问题的答案是：运算符右边的操作数作为显式的（可以看见的）参数被传递给该重载函数。因此，在语句

```
object_2=object_1;
```

中，实际上代替var1作为参数传递给重载函数的是 object_1，运算符左边的对象被赋值。在函数内部的类成员 a 是属于object_2的（这种独特的机制将在不久以后讨论）。这就是object_2.a 的内容从 5 变成 7 的原因。

同时，object_1.a 的值保持不变，我们修改一下test18-4.cpp来说明这一点：

```
//test18-5.cpp
#include <iostream.h>
class assign
{
    public:
    int a;
    /**请注意重载函数的句法
    void operator=(assign var1);
};
//这个运算符重载函数没有返回值（类型为void）
//它带有一个参数，是类assign的对象
//它遵循常规的函数定义规则
void assign::operator=(assign var1)
{
    a=2+var1.a;
}
//object_1和object_2是类assign的对象
assign object_1,object_2;

main(void)
{
    //将对象object_1的类成员 a 赋值为 5
    object_1.a=5;
    //将对象object_2设置成与对象object_1相等
    object_2=object_1;
    cout<<"object_2.a is "<<object_2.a<<"\n";
    cout<<"object_1.a is "<<object_1.a<<"\n";
}
```

编译并运行该程序将得到如下的输出结果：

```
object_2.a is 7
```

object-2.a is 5

这个输出是正确的。运算符右边的操作数在整个处理过程中保持不变。

18.4 不要偏用运算符重载

现在，我们修改一下 test18-5.cpp，玩一点小小的技巧，偏用一下刚刚解释过的概念。请看下列代码：

```
//test18-6.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**请注意重载函数的句法
    void operator+(assign var1);
    };
//这个重载运算符函数没有返回值（类型为void）
//它带有一个参数，是类 assign 的对象
//它遵循常规的函数定义规则
void assign::operator+(assign var1)
{
    a=2+var1.a;
}
//object-1 和 object-2 是类 assign 的对象
assign object-1, object-2;
main(void)
{
    //将对象object-1的类成员 a 赋值为 5
    object-1.a=5;
    //运算符 '+' 实际上执行 '=' 的运算!!!!
    object-2+object-1;
    cout<<"object-2.a is "<<object-2.a<<"\n";
}
```

应该特别注意这两行代码：

```
void operator+(assign var1);
和
object-2+object-1;
```

在第一条语句中, 运算符+被重载。在第二条语句中, object-2与object-1相加, 而结果并未赋给其它任何地方。编译并运行该程序将得到如下的输出结果:

object-2.a is 5

如果运算符没有被重载的话, 语句

object-2+object-1;

将在编译时产生错误。而现在没有出错, 是因为语句

object-2+object-1;

与前例中的

object-2=object-1;

执行相同的操作。

象上述这种情况, 如果有些程序员不经意地改变了运算符的含义, 例如语句

object-2+object-1;

就会使其他人感到迷惑, 不容易理解。因此, 提一点忠告: 在重载运算符时, 不要太偏离该运算符的本来含义。例如不要把 + 运算符重载为 -运算符、 * 运算符重载为 / 运算符等等。在做这些事之前, 应该先想清楚。不要扰乱人们的正常思维, 当然更不要把自己搞糊涂。

18.5 重载的运算符只不过是函数调用

■ 重载的运算符函数的句法仅仅是函数调用的另一种形式。

请看下列代码, 函数名 operator() 被缩写成 op()。

```
//test18-7.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**请注意现在op()只是一个普通的函数
    /**并且它返回一个 assign 类型的对象
    assign op(assign var1);
};
assign object-1, object-2;
//这个op()只是一个普通的函数
//它返回一个assign类型的对象
//它带有一个参数, 是类assign的对象
assign assign::op(assign var1)
{
    a=2+var1.a;
    object-2=object-1;
```

```

    cout<<"object-2.a in op() function is "
        <<object-2.a<<"\n";
    return(var1);
}

main(void)
{
    //将对象 object-1 的类成员 a 赋值为 5
    object-1.a=5;
    //该语句先执行object-1.op()
    //然后将返回值赋给object-2
    object-2=object-1.op(object-1);
    cout<<"object-2.a is "<<object-2.a<<"\n";
}

```

编译并运行该程序将得到如下的输出结果:

```

object-2.a in op() function is 7
object-2.a is 7

```

函数 op() 现在只是类 assign 中的一个普通的成员函数, object-1 和 object-2 被声明为类assign的对象。在main()中, 下列语句

```
object-2=object-1.op(object-1);
```

调用函数 object-1.op() , 并把 object-1 作为参数传递给该函数。在函数内部, 将object-1的类成员加上2, 并通过语句

```
object-2=object-1;
```

赋值给object-2。

因此, 在op()中, object-2.a 变为7, 然而object-2.a的值则保持不变, 这个值返回到main()中。该值也赋值给了 object-2, 所以现在 object-2.a 就变成了5。

前面已讲过, 传递到函数中的变量的内容只能通过指针来操纵或修改(除非传递给函数的是数组或全局变量)。这是因为只是将参数的拷贝传递给了函数, 而不是实际的值本身。下面我们通过一个小程序来说明这一点:

```

//test18-8.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**请注意现在op()只是一个普通的函数
    assign op(assign *object-1);

```

```

};
assign object_1, object_2, *ptr;
//这个op() 只是一个普通的函数
//它返回一个assign类型的对象
//它带有一个参数, 是类assign的对象指针
assign assign::op(assign *ptr)
{
    //修改 ptr 所指的内容
    //ptr 指向了 object_1.a
    ptr->a=2+ptr->a;
    //将object_1返回
    return object_1;
}

main(void)
{
    //将对象object_1的类成员 a 赋值为 5
    object_1.a=5;
    //将ptr指向object_1的内存地址
    ptr=&object_1;
    //将 object_2 赋值为等于 object_1
    //该返回值是类 assign 的对象
    object_2=object_1.op(ptr);
    cout<<"object_2.a is "<<object_2.a<<"\n";
}

```

编译并运行该程序将得到如下的输出结果:

```
object_2.a is 7
```

这次, 把指向object_1的指针传递给函数op():

```
assign op(assign *object_1);
```

将 ptr 指向 object_1 的内存地址, 然后调用函数op()。

在函数 op() 内部, 通过 ptr 修改了object_1 的实际内容:

```
ptr->a=2+ptr->a;
```

object_1的成员通过运算符->来存取。同样也可以使用圆点(.)运算符, 如:

```
ptr.a=2+ptr.a;
```

将object_1修改后的内容传回到 main() 中, 这个返回值赋给了object_2。因此, object_2.a 输出为 7。

哪一个程序读起来更清晰、更自然呢? 如下使用指针的程序会有些技巧的问题, 下面列出两个程序中的第一个:

```
//test18-8.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**请注意现在op() 只是一个普通的函数
    assign op(assign *object-1);
    };
    assign object-1, object-2, *ptr;
    //这个op() 只是一个普通的函数
    //它返回一个 assign 类型的对象
    //它带有一个参数, 是类 assign 的对象指针
    assign assign::op(assign *ptr)
    {
        //修改 ptr 所指的内容
        //ptr 正指向 object-1.a
        ptr->a=2*ptr->a;
        //将object-1返回
        return object-1;
    }

    main(void)
    {
        //将对象 object-1的类成员 a 赋值为 5
        object-1.a=5;
        //将 ptr 指向 object-1 的内存地址
        ptr=&object-1;
        //将 op() 的返回值赋给 object-2
        object-2=object-1.op(ptr);
        cout<<"object-2.a is "<<object-2.a<<"\n";
    }
}
```

另一个是使用运算符重载的程序:

```
//test18-4.cpp
#include <iostream.h>
class assign
```

```
{
    public:
    int a;
    /**请注意重载的函数的句法
    void operator=(assign var1);
};
//这个重载的运算符函数没有返回值（类型为void）
//它带有一个参数，是类 assign 的对象
//它遵循普通的函数定义的规则
void assign::operator=(assign var1)
{
    a=2+var1.a;
}
//object_1和object_2是类assign的对象
assign object_1,object_2;

main(void)
{
    //将对象object_1的类成员 a 赋值为 5
    object_1.a=5;
    //将对象object_2设置成与对象object_1相等
    object_2=object_1;
    cout<<"object_2.a is "<<object_2.a<<"\n";
}
```

如果读者更喜欢第一种形式，那么可能还没能体会到调用重载运算符的句法是多么的轻松自然。我们还是建议先不要使用重载运算符这种方法，直到读者感觉到这些概念和句法用起来很舒适为止。

如果读者更喜欢第二种形式，那么证明已可使用 C++ 的另一种强大特性。

18.6 运算符重载的优点

- 运算符重载的主要优点之一就是：它允许您使用用于内部数据类型运算的运算符来对用户自定义的数据类型可使用用于系统内部数据类型的同样的运算符。

可以使用如下的表达式：

```
a=a+b;
a=a*b;
```

但是如果运算符没有重载，语句

```
object_1=object_1+object_2;
object_1=object_1*object_2;
```

就是非法的，因为 `object.1` 和 `object.2` 属于用户自定义的数据类型。运算符重载提供了一种非常自然的方法，以表达这种语句。再看另一个例子，用如下表达式直观地表达两个字符串的联接：

```
"This is a"+"concatenation of 2 strings. ";
```

如果没有运算符重载，这条语句将不能正确地编译。

18.7 运算符重载的缺点

■ 运算符重载的主要缺点就是有可能将运算符的含义隐藏在难以理解的代码中。

程序员可能搞不懂为什么象+这样的运算符没能执行所期望的功能。程序员因此而受到约束而感到不适应（即不能理解这些代码），以及由此引起诸多烦恼。

18.8 小 结

在本章里，我们解释了如何进行运算符重载以及运算符重载之所以成为一种强大工具或有点消极作用的原因。侧重地解释了：

- 以前已经使用过象 `<<` 和 `*` 这样的重载运算符，实际上运算符重载并不新鲜了。
- 这种机制可以用来改变C++所提供的大多数运算符的含义。
- 重载运算符函数必须是类成员函数或友元函数。
- 重载运算符函数仅仅是函数调用的另一种替代形式。
- 重载函数的句法如下：

```
type operator#(argument-list);
```

其中`type`表示函数的返回值类型，它可以是`void`类型或其它任何合法的数据类型；`#`表示被重载的运算符；`argument-list`表示参数表，这些参数将传递给重载函数。

- 重载运算符允许使用一种非常自然的表达方式来表达两个用户自定义数据类型之间的关系。
- 如果在不显眼的或令人费解的代码中使用重载运算符，可能会给程序员带来麻烦。

第十九章 运算符重载, this 和 friend

19.1 简介

在本章中, 我们将介绍名为 `this` 的指针 (`this` 是指针的名字)。这个指针是构成运算符重载这一机制所必需的组成部分。还要介绍友元函数的用法。友元函数可以用来代替运算符重载函数。

19.2 双目和单目重载运算符

双目运算符是指能操纵两个对象的运算符。例如, `+` 就是一个双目运算符, 因为为了实现此种运算, 必须将某个东西与另一个东西相加。单目运算符是指只有一个操纵对象的运算符, 例如, `++` 就是一个单目运算符; `something++` 的结果是将 `something` 的内容加一。

表 19.1 和 19.2 分别列出了不能被重载和能被重载的运算符。

表 19.1 不能被重载的运算符

运算符	功能
,	逗号
.	成员
->	类或结构指针
?=	三元运算符
sizeof	取字节大小

表 19.2 可重载的运算符

运算符	功能
++	增量运算
--	减量运算
!	取非运算
-	取补运算
+	加法运算
-	减法运算
*	乘法运算
/	除法运算
%	取模运算
()	函数调用
[]	数组描述
new	分配自由存储空间
delete	释放自由存储空间

=	赋值
+=	加并赋值
-=	减并赋值
*=	乘并赋值
/=	除并赋值
&	按位与
	按位或
^	按位异或
	逻辑或
&&	逻辑与
<	小于
<=	小于等于
>	大于
>=	大于等于
<<	左移位
>>	右移位
=	取或并赋值
^=	取异或并赋值
&=	取与并赋值
<<=	左移位并赋值
>>=	右移位并赋值
==	逻辑等于
!=	不等于

19.3 运算符重载的限制

- 不能构造自己的运算符，只能重载已存在的运算符。
- 运算符重载只能应用于类对象
- 不能改变原来的运算符的优先级和结合性。
- 不能将双目运算符重载成单目运算符。
- 不能将单目运算符重载成双目运算符。
- 运算符++和一重载后区分不了前缀或后缀的应用。
- 不能重载专用于指针的运算符。

19.4 运算符重载的表达句法

请注意前一小节中的第四和第五条限制。这些限制说明必须遵循与特定运算符相联系的句法格式。也就是说，不能改变它的基本模板。看下列程序：

```
//test19-1.cpp
```

```
#include <iostream.h>
class assign
{
public:
    int a;
    /**注意运算符重载函数句法:
    void operator/(assign var1);
    };
    //运算符重载函数返回的数据类型为 void(即无类型)
    //它带一个类 assign 的对象作为参数
    //它遵循普通函数定义的规则
    void assign::operator/(assign var1)
    {
        a=var1.a / 5;
    }
    //object-1 和 object-2 是类 assign 的对象
    assign object-1,object-2;
    void main(void)
    {
        //将对象 object-1 的类成员 a 赋值为15
        //将对象 object-2 的类成员 a 赋值为10
        object-1.a=15;
        object-2.a=10;
        //将除法运算符作为单目（一元）运算符来用
        object-2 / ;
        cout<<"object-1.a is "<<object-1.a<<"\n";
        cout<<"object-2.a is "<<object-2.a<<"\n";
    }
}
```

请注意语句:

```
object-2/;
```

可以看到, 这里试图使用本该是双目运算符的除法运算符对单个对象做运算。编译该程序将产生如下出错信息:

Error: Expression syntax

现在再看一看下面的程序, 该程序也重载了除法运算符, 但作用于两个对象而不是作用于单个对象。

```
//test19-2.cpp
#include <iostream.h>
```

```
class assign
{
public:
    int a;
    /**注意运算符重载函数句法:
    void operator/(assign var1);
    };
//运算符重载函数返回的数据类型为void(即无类型)
//它带一个类assign 的对象作为参数
//它遵循普通函数定义的规则
void assign::operator/(assign var1)
{
    a=var1.a / 5;
}
//object-1和object-2是类assign的对象
assign object-1,object-2;
main(void)
{
    object-1.a=15;
    object-2.a=10;
    cout<<"object-2.a is "<<object-2.a<<"\n";
    //将 object-2 除以 object-1
    object-2 / object-1;
    cout<<"object-1.a is "<<object-1.a<<"\n";
    cout<<"object-2.a is "<<object-2.a<<"\n";
}
```

编译并运行这个程序将产生如下输出结果:

```
object-2.a is 10
object-1.a is 15
object-2.a is 3
```

object-1.a 和 object-2.a 分别初始化为 15 和 10, 除法运算符 (/) 被重载, object-2被 object-1 除, 语句:

```
a=var1.a/5;
```

可被解释为如下代码:

```
object-2.a=object-1.a/5;
```

这个语句等价于

```
object-2.a=15/5;
```

所以 object-2.a 等于 3。输出语句应该是有作用的：表示 object-1.a 的内容保持不变，object-2.a 的内容被赋值为 3。

在上一章中，我们曾经提到过对含有重载运算符的语句的引用方法：

- 运算符右边的操作数作为显式的参数传递给要调用的重载运算符函数。
- 左边的操作数则隐式地传递给要调用的函数。

本章对这个概念作详细的说明，以便读者能理解这一机制如何运作。再看一下这个函数的定义：

```
void assign::operator/(assign var1)
{
    a=var1.a/5;
}
```

基于如上所述，var1 将包含赋给 object-1 的值，因为 object-1 在运算符的右边。这是传递给调用函数的显式参数。

指向左操作符的指针被隐式地传给 operator/()。因此，语句

```
a=var1.a/5;
```

中的变量 a 属于 object-2。指向 object-2 的指针被隐式地传递给被重载的函数。但这究竟是如何发生的呢？答案是通过一个指针。Stroustrup 先生把它叫做 this 指针。

19.5 this 指针

- 每次成员函数被调用时，都会给调用它的对象传递一个指针。这个指针的名字叫做 this 指针。
- 这个指针对于我们来说是不可见的，因为它自动地即隐式地传递。
- this 指针的句法类似于其它指针的句法。唯一的不同点就在于 this 指针不需要声明，因为它能自动地产生。

现在来证明一下指向重载运算符左边的操作数的指针是通过 this 指针隐含地传递给调用函数的。下面是 test19-2.cpp 的一个修改版本：

```
//test19-3.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**注意运算符重载函数句法:
    void operator/(assign var1);
    };
//运算符重载函数无返回值
//它带一个类 assign 的对象作为参数
```

//它遵循普通函数定义的规则

```
void assign::operator/(assign var1)
{
    cout<<"this->a is " << this->a <<"\n";
    a=var1.a / 5;
}
//object-1 和 object-2 是类 assign 的对象
assign object-1,object-2;

main(void)
{
    object-1.a=15;
    object-2.a=10;

    //将object-2除以object-1
    object-2 / object-1;

    cout<<"object-1.a is " << object-1.a <<"\n";
    cout<<"object-2.a is " << object-2.a <<"\n";
}
```

编译并运行该程序将产生如下输出结果:

```
this-a is 10
object-1.a is 15
object-2.a is 3
```

我们知道, this->a 的内容是 10, 是在 main() 中赋值给 object-2.a 的, 语句:

```
a=var1.a/5;
```

可被解释为如下代码:

```
object-2.a=object-1.a/5;
```

这个语句等价于

```
object-2.a=15/5;
```

所以 object-2.a 被赋值为 3。这个输出语句应该是很清楚的。

让我们来交换一下 main() 中的 object-1.a 和 object-2.a 的位置, 看一看 this 指针的值将会受到什么影响。下面是程序代码:

```
//test19-4.cpp
#include <iostream.h>
class assign
```

```
{
public:
int a;
/**注意运算符重载函数句法:
void operator/(assign var1);
};
//运算符重载函数返回类 assign 的对象
//它带一个类 assign 的对象作为参数
//它遵循普通函数定义的规则
void assign::operator/(assign var1)
{
    cout<<"this->a is " << this->a <<"\n";
    a = var1.a / 5;
}
//object-1 和 object-2 是类 assign的对象
assign object-1,object-2;

main(void)
{
    object-1.a=15;
    object-2.a=10;
    cout<<"object-2.a is "<<object-2.a<<"\n";
    //将 object-2 除以 object-1
    object-2 / object-1;
    cout<<"object-1.a is "<<object-1.a<<"\n";
    cout<<"object-2.a is "<<object-2.a<<"\n";
}
```

编译并运行该程序将产生如下输出结果:

```
object-2.a is 10
this->a is 15
object-1.a is 2
object-2.a is 10
```

语句

```
a=var1.a/5;
```

可被解释为如下代码:

```
object-1.a=object-2.a/5;
```

这个语句等价于

```
object-1.a=10/5;
```

所以 object-1.a 等于2, 而 object-2.a 保持不变。就不需再解释这个输出结果。

现在, 我们再巩固一下所学内容。下面的程序说明了运算符重载是怎样进行的。这次, * 运算符被重载:

```
//test19-5.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**注意运算符重载函数的返回值:
    int operator*(assign var1);
    };
//运算符重载函数返回的数据类型为 int
//它带一个类 assign 的对象作为参数
//它遵循普通函数定义的规则
int assign::operator*(assign var1)
{
    cout<<"this->a is " << this->a <<"\n";
    a = this->a * var1.a ;
    return a;
}
//object-1, object-2 和object-3 是类 assign 的对象
assign object-1,object-2,object-3;
void main(void)
{
    object-1.a=15;
    object-2.a=10;
    //object-3.a接收返回值
    object-3.a = object-2 * object-1;
    cout<<"object-1.a is "<<object-1.a<<"\n";
    cout<<"object-2.a is "<<object-2.a<<"\n";
    cout<<"object-3.a is "<<object-3.a<<"\n";
}
```

要注意的是，这一次重载函数返回一个整型值，而不是类 `assign` 的对象。在函数内部，返回的是 `a`，它是 `object-2` 的类成员。因为要返回一个整型值，所以在 `main()` 中作了一些修改，用一个整型变量来接收这个值。

编译并运行该程序将产生如下输出结果：

```
this->a is 10
object-1.a is 15
object-2.a is 150
object-3.a is 150
```

语句

```
object-3.a=object-2*object-1;
```

导致 `object-2` 和 `object-1` 相乘，`*` 运算符被重载，在重载函数中：

```
int assign::operator*(assign var1)
{
    cout<<"this->a is " << this->a <<"\n";
    a = this->a * var1.a ;
    return a;
}
```

`object-1` 被显式地传递，指向 `object-2` 的指针被隐式地传递。变量 `a` 和 `this->a` 属于 `object-2`（它们在被重载的运算符左边）。语句

```
a=this->a*var1.a;
```

可被解释为如下代码：

```
object-2.a=10*15;
```

这样，`object-1.a` 的值保持不变，而则 `object-2.a` 通过语句：

```
return a;
```

来返回，并将其值赋给 `object-3.a`；

```
object-3.a = object-2*object-1;
```

现在应该完全清楚这些输出结果了。

19.6 FRIEND函数

到此为止，成员函数只是用来说明运算符重载是如何运作的。

■ 包含重载运算符函数定义的类的友元函数也可以用来进行运算符重载。

如前所述，只能给重载双目运算符的成员函数显式地传递一个参数，并通过 `this` 指针可以隐式地传递另一个参数的指针。

■ 被重载的单目运算符函数定义不需要任何参数。

■ 在友元函数中，被重载的单目运算符函数带有一个参数，而双目运算符函数则带有两个参数。这两类运算符的参数都要显式地传递。

■ `this` 指针不能用于友元函数，因为它仅仅从类成员函数中返回类成员的内存地址。而友元函数则不在这一作用范围内。

请看程序 test19_6.cpp。它说明了重载运算符的友元函数用法。

```
//test19_6.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**注意友元函数
    friend int operator*(assign var-1,assign var-2);
    */;
//运算符重载函数返回的数据类型为整型数
//它带两个类 assign 的对象作为参数
//它遵循普通函数定义的规则
//operator* 是一个友元函数，因此不需作用域分辨符来限定
int operator*(assign var-1,assign var-2)
{
    var-2.a = var-1.a * var-2.a ;
    return var-2.a;
}
//object-1, object-2 和 object-3 是类 assign 的对象
assign object-1, object-2, object-3;
void main(void)
{
    object-1.a=15;
    object-2.a=10;
    //object-3.a接收返回值
    object-3.a = object-2 * object-1;
    cout<<"object-1.a is "<<object-1.a<<"\n";
    cout<<"object-2.a is "<<object-2.a<<"\n";
    cout<<"object-3.a is "<<object-3.a<<"\n";
}
```

再看一下函数定义:

```
int operator*(assign var-1,assign var-2)
{
    var-2.a=var-1.a*var-2.a;
    return var-2.a;
}
```

要注意到函数名没有用作用域分辨符来限定，这是有意义的。因为它是友元函数，无需作用域分辨符。但也要注意这两个参数都是显式传递的。

编译并运行该程序将产生如下输出结果：

```
object-1.a is 15
object-2.a is 10
object-3.a is 150
```

现在让我们再来看一下调用重载运算符函数的语句：

```
object-3.a=object-2*object-1;
```

在重载运算符 `*` 左边的对象 `object-2`，赋值为 `var-1`，右边的对象 `object-1`，赋值为 `var-2`。在该函数内部，对象本身的值并不改变，因为被操纵的是该对象的拷贝。返回值赋值给 `object-3.a`，因此 150 作为这个数据类型的值来输出。

如果要修改 `object-1` 或 `object-2` 的内容，则要以引用的形式来传递这些对象。下面的程序说明了如何通过引用而不是通过值的方式来传递参数。

```
//test19-7.cpp
#include <iostream.h>
class assign
{
public:
    int a;
    /**注意友元函数
    friend int operator*(assign var-1,assign var-2);
    */
};
//运算符重载函数返回值为整型数
//它带两个类 assign 的对象作为参数
//它遵循普通函数定义的规则
//operator* 是一个友元函数，因此不需作用域分辨符来限定
//var-2 作为一个参数来传递。
int operator*(assign var-1,assign var-2)
{
    var-2.a = var-1.a * 5 ;
    var-2.a = var-1.a * var2.a ;
    return var-2.a;
}
//object-1, object-2和object-3是类assign的对象
assign object-1,object-2,object-3;
```

```
void main(void)
{
    object_1.a=15;
    object_2.a=10;
    //object_3.a 接收返回值
    object_3.a = object_2 * object_1;
    cout<<"object_1.a is "<<object_1.a<<"\n";
    cout<<"object_2.a is "<<object_2.a<<"\n";
    cout<<"object_3.a is "<<object_3.a<<"\n";
}
```

请注意被重载的友元函数的函数原型:

```
int operator*(assign var_1,assign &var_2)
```

调用该函数时, 第一个参数通过值传递参数; 也就是说, 将这个变量的一个拷贝传递给调用函数。第二个参数则是通过引用来传递的(用&号置于变量名之前)。

编译并运行该程序将产生如下输出结果:

```
object_1.a is 500
object_2.a is 10
object_3.a is 500
```

语句

```
object_3.a=object_2*object_1;
```

使重载运算符函数被调用。在该函数中, var_1 被赋予 object_2 的值(10)。因为它在重载运算符的左边, var_2 被赋予 object_1 的值(15)。语句

```
var_2.a=var_1.a*5;
```

可解释为如下代码:

```
object_1.a=object_2.a*5;
```

object_1. 赋值为 $10*5$, 即等于50。

因为 object_1 是作为引用来传递的, 所以它的实际值也变成了50。语句

```
var_2.a=var_1.a*var_1.a;
```

可解释为如下代码:

```
object_1.a=object_2.a*object_1.a;
```

等价于

```
object_1.a=10*50;
```

即 object_1.a 等于500。现在这个输出结果的原因就变得很明了。

■ 友元函数使得重载运算符代码比较容易读, 因为所有的参数的传递都是亲眼可见的。

友元函数重载运算符时, 没有引用隐含的或隐藏的参数。不仅可以使用类成员来重载运算符, 也可以使用友元函数来重载运算符。不管选择哪一种, 都应记住将来其他人可能

接手您的代码。所以一定要确保您的重载函数代码容易被别人读懂，多加一些注释，千万不要扰乱别人的正常思维。

19.7 小 结

本章，我们描述了重载运算符的其它特性：

- 已有的大多数运算符是可重载的。
- 对于能被重载的运算符的引用范围有一些限制。
- 双目重载运算符左边对象指针通过 `this` 指针隐式地传递给调用函数。
- 重载运算符的右边对象作为参数显式地传递给调用函数。
- 双目重载运算符只传递一个参数。
- 单目重载运算符不传递任何参数。
- 也可以用友元函数来重载运算符，友元函数要求将所有的参数都显式地传递给调用函数。

第二十章 C++的预处理程序指令

20.1 简介

在本章中，我们将描述C++的预处理程序指令。

- 预处理程序指令实现文件包含，简单的字符串替换，宏扩展以及执行条件编译。有些指令也可以帮助我们调试源代码。首先让我们弄清楚什么是预处理程序。

20.2 C++的预处理程序

C++带有一个标准的函数库，这些函数包含在标准的C++编译器软件包中。

- 预处理程序是指在程序的源代码传递给编译器之前，处理该程序的源代码的程序。
- 预处理程序指令前面带有 # 符号，称作预处理程序控制行。

基于控制行，预处理程序完成一个或多个如下的功能：

- 包含文件

用下列指令来包含一个文件：

```
#include
```

- 替换字符串，扩展和 / 或取消宏定义。

上述两种功能由下列指令来实现：

```
#define
```

```
#undef
```

- 执行条件编译

通过如下几组控制行来实现条件编译：

```
#if
```

```
#else
```

```
#endif
```

```
#if
```

```
#elif
```

```
#endif
```

```
#if !defined
```

```
#endif
```

- 帮助调试

下列的控制行可以帮助调试：

```
#pragma
```

下面几节将具体描述每种指令。

20.3 INCLUDE (包含) 文件

我们以前介绍的所有示例程序都包含了文件 `<iostream.h>`。我们来看一看如果不包含该文件会怎么样。

```
//test20-1.cpp
void main(void)
{
    cout<<"Hi there! \n";
    printf("Hi there again! \n");
}
```

编译该程序会产生如下错误信息:

Error: Underfind Symbol 'cout'

Error: Function 'printf()' should have a prototype

很显然, 包含文件 `iostream.h` 非常有用。让我们将必要的文件包含进去再编译:

```
//test20-2.cpp
//输入输出需要iostream.h
//printf()需要stdio.h
#include <iostream.h>
#include <stdio.h>
void main(void)
{
    cout<<"Hi there! \n";
    printf("Hi there again! \n");
}
```

编译并运行该程序将产生如下输出结果:

Hi there!

Hi there again!

`#include` 指令使得在编译时把紧跟其后的文件名所含内容都包含进去。头文件 `iostream.h` 包含了 `cout` 的定义, `stdio.h` 包含了 `printf()` 的定义。

`#include` 指令有三种格式:

■ `#include <文件名>`

读者一定很熟悉这种格式, 文件名用尖括号 (`<>`) 括起来, 是指编译器要在当前工作目录以外的预先安装好的目录表中查找该文件。

■ `#include "文件名"`

这种格式会让编译器在当前工作目录中查找该文件。

■ #include “C:\目录名\文件名”

这种形式告诉编译器要在指定的路径内查找文件。如果该目录中没有该文件，则在标准的目录中查找。

下面的小程序用来说明指定目录名这种形式。

```
//test20-3.cpp
//输入输出需要iostream.h
//printf()需要stdio.h
#include <iostream.h>
#include <stdio.h>
#include "C:\WP51\fileone"

void main(void)
{
    cout<<"Hi there! \n";
    printf("Hi there again! \n");
    cout<<"a from fileone is "<< a <<"\n";
}
```

刚好在我们的计算机中有一个名为 wp51 的目录（实际上，这是我们所用的字处理器目录），文件 filename 中有如下内容：

```
int a=5;
```

正好，文件 filename 只有一行，正常情况下，被包含的文件可能很大。实际上，这些就是包含文件的全部用途了。包含文件有助于将一个大的源文件分成逻辑上易于管理的段落。编译并运行该程序将产生如下输出结果：

```
Hi there!
```

```
Hi there again!
```

```
a from fileone is 5
```

■ 虽然预处理程序将文件包含到当前源代码中，但诸如检查语法错误之类的事还是由编译器来负责完成的。

如果被包含的文件没能正确地编译，编译器也会产生出错信息。假设被包含的文件名叫 filetwo，其内容如下：

```
int a=5
```

请注意，其中丢失了分号（;）。我们看一下包含了该文件的程序：

```
//test20-4.cpp
#include <iostream.h>
#include <stdio.h>
#include "C:\WP51\filetwo"
```

```
void main(void)
{
    cout<<"Hi there! \n";
    printf("Hi there again! \n");
    cout<<"a from fileone is "<< a <<"\n";
}
```

编译该程序会产生五个错误，第一个就是：

Error: Declaration syntax error

这个出错信息会使人误解，我们不必列出其余的出错信息，因为它们也没有意义。从这里可以得出：

- 在把文件包含到源程序代码中之前，一定要确保它能被正确地编译；否则在调试时会带来麻烦。

20.4 简单的字符串替换

- 使用 `#define` 指令作简单的字符串替换。

下面的小程序说明该控制行的用法：

```
//test20-5.cpp
#include <iostream.h>
#define HELLO "Hi there! \n"
void main(void)
{
    cout<<HELLO;
}
```

编译并运行该程序将产生如下结果：

Hi there!

在每次遇到 `HELLO` 源代码时，它都被字符串 `Hi there\n` 所替换。语句：`cout<<Hello;` 被 `cout<<"Hi there!\n"` 所替换。

这就是为什么在屏幕上输出了 `Hi there!` 的原因。

在使用 `#define` 控制行时，大多数程序员使用大写字母命名。这是个值得遵循的好习惯，因为它能清晰地标识出在程序中被定义的那个变量。

20.5 不带参数的宏 (Macros)

- 宏扩展是字符串替换的一种形式。

在宏中也可以指定参数，这一点我们在下一节里讨论。我们先看一下下面的程序，它说明了一个简单的宏扩展。

```
//test20_6.cpp
#include <iostream.h>
#define SQUARE-TWO 2*2
void main(void)
{
    int a;
    a = SQUARE-TWO;
    cout<<"a is "<< a <<"\n";
}
```

在

```
#define SQUARE-TWO 2*2
```

这一行中，#define是控制行，SQUARE-TWO是宏名，2*2是宏的扩展，这是一个宏定义。编译并运行该程序会产生如下输出结果：

```
a is 4
```

因为语句

```
a = SQUARE-TWO;
```

可以扩展为

```
a=2*2;
```

所以输出结果当然应该是 a 等于4。

■ 宏扩展只在程序内的某个位置上定义了字符串替换时才有用。

假定在一个500行的程序中，2的平方出现了60次，这些代码都分散在整个程序之中。

如果您想把 2 乘以 2 改成 2 乘以 8，就需要将宏定义行修改如下：

```
#define SQUARE-TWO 2*8
```

其后所有对 SQUARE-TWO 的引用也都按此来变化。

20.6 带参数的宏

下面这段程序说明了带参数的宏的扩展。

```
//test20_7.cpp
#include <iostream.h>
#define ADD(X) (X+X)
void main(void)
{
    int b;
    b = ADD(4);
    cout<<"b is "<< b <<"\n";
}
```

控制行

```
#define ADD(X) (X+X)
```

被下列语句引用:

```
b = ADD(4);
```

它被扩展为

```
b=4+4;
```

编译并运行该程序将产生如下输出结果:

```
b is 8
```

再看一下这个控制行:

```
#define ADD(X) (X+X)
```

请注意, 在宏名和参数 X 之间没有空格, 如果在中间加一个空格的话, 参数就会被当作宏扩展的一部分。请看 test20-8.cpp, 它说明了宏扩展的这种形式。

```
//test20-8.cpp
#include <iostream.h>
//注意宏名和参数之间的空格
#define ADD (X) (X+X)
void main(void)
{
    int b;
    a = ADD(4);
    cout<<"b is "<< b <<"\n";
}
```

编译该程序会产生如下出错信息:

```
Error:Undefined symbol 'X'
```

当字符串被扩展时, 编译器不承认括号中的 4 是被取代的参数。在此, 空格作为宏名和宏扩展的分界符。

现在假设我们在控制行中忘了括括号。如:

```
#define ADD(X) X+X
```

这种省略可能潜在着危险, 我们通过下列程序来说明:

```
//test20-9.cpp
#include <iostream.h>
#define ADD(X) X*X
void main(void)
{
    int b;
    //用宏扩展的结果乘以5
```

```
a = ADD(4)*5;
cout<<"b is "<< b <<"\n";
}
```

编译并运行该程序会产生如下输出:

```
b is 24
```

语句

```
b =ADD(4)*5
```

可被扩展为

```
b=4+4*5;
```

因为乘法的优先级比加法高, b 将通过如下表达式来计算:

```
b=4+(4*5);
```

然而我们希望得到的是:

```
b=(4+4)*5;
```

因省略了括号, 表达式未按期望的情况来求值。所以一定要记住这一点。

20.7 取消宏定义

■ 在前面定义的宏可以用下面的控制行来取消:

```
#undef
```

该控制行不需要字符串替换, 请看 test20-10.cpp, 它说明了undef的用法。

```
//test20-10.cpp
#include <iostream.h>
#define FOUR 4
main(void)
{
    int a = FOUR;
    cout<<" a is "<< a <<"\n";
    #undef FOUR
    int b = FOUR;
    cout<<"b is "<< b <<"\n";
}
```

编译该程序将产生如下错误信息:

```
Error: Undefined symbol FOUR
```

该错误是由下列语句引起的:

```
int b=FOUR;
```

这是因为在这个语句之前, 取消了FOUR的定义, 编译器不再用FOUR来代替4。取消一个宏名是为了将这个宏名限制在必需它的这段代码中使用。

20.8 条件编译

■ 在源代码中遇到下列关键字时, 将发生条件编译:

`#if` - `#else` - `#endif`

`#if` - `#elif` - `#endif`

`#ifdef` - `#endif`

`#ifndef` - `#endif`

我们在下面的几小节里讨论这些关键字。

20.8.1 `#if` 和 `#endif`

关键字 `#if` 后接一个常量表达式为真值 (或非0值) 时, 才将 `#if` 和 `#endif` 之间的程序代码块包含到编译中。

程序 `test20_11.cpp` 说明了它的用法。

```
//test20_11.cpp
#include <iostream.h>
void main(void)
{
    const int a = 5;
    const int b = 0;
    #if (a)
        cout<<" a is TRUE , i.e., non-zero \n";
    #else
        cout<<" a is FALSE , i.e., zero \n";
    #endif
    #if (b)
        cout<<" b is TRUE , i.e., non-zero \n";
    #else
        cout<<" b is FALSE , i.e., zero \n";
    #endif
    cout<<"This code is outside the blocks \n";
}
```

编译并运行该程序将产生如下输出:

a is TRUE , i.e., non-zero

b is FALSE , i.e., zero

This code is outside the blocks

这个输出结果就无需再解释了。请注意, 测试条件中的常量表达式是必要的。如果没有, 将会出错。请看 `test20_12.cpp`

```
//test20-12.cpp
#include <iostream.h>
void main(void)
{
    //a 和 b 作为变量来声明
    int a = 5;
    int b = 0;
    #if (a)
        cout<<" a is TRUE , i.e., non-zero \n";
    #else
        cout<<" a is FALSE , i.e., zero \n";
    #endif
    #if (b)
        cout<<" b is TRUE , i.e., non-zero \n";
    #else
        cout<<" b is FALSE , i.e., zero \n";
    #endif
    cout<<"This code is outside the blocks \n";
}
```

编译该程序将产生如下出错信息:

Error: Constant experssion required

使用 #if , #else 和 #endif 与使用普通的 if, else, endif的控制结构的不同之处是, 对于前者的求值发生在程序编译和运行之前。不能在这种测试条件中放入变量, 因为变量的值在运行时是可以改变的。

20.8.2 #if, #elif和#endif

这些控制行与前面讲过的 #if, #else 和 #endif 控制行等效

```
main(void)
{
    #ifdef COMPILE
        cout<<"This code will be compiled \n";
    #endif
}
```

编译并运行该程序将产生如下输出结果:

This code will be compiled

在该例中, 定义了宏COMPILE; 因此在两个控制行之间的代码就被编译了。如果我们前面这个程序中的 #define 语句注释掉, 控制行之间的代码就不会被编译。请看一下test20_14.cpp:

```
//test20_14.cpp
#include <iostream.h>
//将宏定义COMPILE 作为注释
//#define COMPILE
void main(void)
{
    #ifdef COMPILE
        cout<<"This code will be compiled \n";
    #endif
}
```

编译该程序, 没有输出内容, 因为宏COMPILE没被定义。

■ 预处理程序的这种特性可以用来辅助程序的调试。

在这类宏中插入调试语句, 将这些语句设置成注释的和非注释的, 以便能弄清楚源代码中是否包含这些调试语句。

20.8.4 #ifndef 和 #endif

这些控制行与 #ifdef 和 #endif 语句正好相反。

■ 只有当宏名没被定义时, 在 #ifndef 和 #endif 之间的程序代码块才被包含到编译中。

下列程序说明了这一点:

```
//test20_15.cpp
#include <iostream.h>
#define COMPILE
```

```
void main(void)
{
    //如果宏COMPILE 没有定义，将编译下列代码
    #ifndef COMPILE
        cout<<"This code will be compiled \n";
    #endif
}
```

编译并运行该程序不会产生任何输出结果，这是因为COMPILE已经定义过了。如果将宏控制行作为注释或干脆删掉，就会有输出。

20.9 #PRAGMA

- 该控制行允许将不同的指令送给编译器，该指令与具体的实现有关。请参考具体的编译器资料对 #pragma 指令的说明。

20.10 小 结

在本章中，我们讨论了预处理程序及其指令。我们学习了：

- 包含文件

```
#include <fileone>
#include "fileone"
#include "c:\dir\fileone"
```

- 通过宏定义实现简单的字符串替换。

```
#define HELLO "hello"
```

该语句将导致在源程序代码中，每次遇到 HELLO，都用 hello 将它替换。

- 取消宏定义，以使字符串替换不再发生。

```
#undef HELLO
```

- 定义带参数的宏

```
#define square(X) (X*X)
```

该语句带一个参数，按指定的方式来扩展它。

- 通过下列控制行按条件编译：

```
#if, #else和#endif
```

只有当测试条件中的常量表达式为TRUE或非零值时，其块内的程序代码才被编译进去。

下面的指令与 #if, #else和#endif 等效：

```
#if, #elif和#endif
```

只有当指令 #ifdef 后面所带的宏名在此之前已被定义时，在其块内的程序代码才会被编译：

```
#ifdef 和 #endif
```

只有在宏名没有被定义时，下列控制行指令之间的代码才会被编译：

`#ifndef` 和 `#endif`

- 如何使用`#pragma`指令，该指令给编译器发送指令。该指令的实现依赖于具体的编译器。

附录 A 术语汇编

抽象类 (Abstract Classes) : 含有纯虚成员函数的基类。抽象类只能作为基类来使用, 并为其派生类所继承。

联编 (Binding) : 是指将一个过程的地址传递给调用者这一事件的发生过程。请参见早期联编和后期联编。

C++ : 一种由 Bjarne Stroustrup 在八十年代中期开发出来的面向对象的语言, 它继承了 C 语言。

类 (Classes) : 是对象的定义, 它包含了成员以及在类声明中指定的函数即方法对这些成员的存取权限。

构造函数 (Constructor) : 在对象产生时, 自动初始化对象的方法或函数。如果没有提供明显的定义, C++编译器就隐式地调用构造函数。

数据隐藏 (Data Hiding) : 通过将类成员声明为私有的 (private), 而使这些成员的数据不能被该类以外的所有函数所使用的这一机制。

析构函数 (Destructor) : 用来释放那些由构造函数分配给对象的内存单元的函数或方法。如果没有提供明显的定义, C++编译器就隐式地调用析构函数。

动态联编 (Dynamic Binding) : 又叫后期联编。在程序运行时所发生的联编事件。过程或函数的地址在运行时才定下来。后期联编使多态性成为可能。

早期联编 (Early Binding) : 在程序编译时所发生的联编事件。过程或函数的地址在编译/链接时就定了下来。

封装 (Encapsulation) : 将数据和允许操纵这些数据的函数包装在一个对象定义内部, 这一机制称为封装。

可扩展性 (Extensibility) : 面向对象的语言提供了一种特性。它允许在不知道 (或不需要) 源代码的情况下, 对已存在的代码进行扩展。代码扩展允许从已存在的对象中产生新的对象。

友元函数 (Friend Functions) : 在类中包含的却不是该类成员的一种函数。然而, 它完全可以访问该类中的所有私有成员和受保护的成员。

自由存储对象 (Free Store Objects) : 通过使用操作符 `new` 来分配自由存储空间即堆而产生的对象。自由存储空间是一个未分配的内存池, 它在程序运行时提供。

信息隐藏 (Information Hiding) : 见封装。

继承性 (Inheritance) : 一个类从另一个或多个其它类中取得所有的数据成员和函数定义 (不包括私有成员) 的这种特性。继承来的数据成员或函数不需重新描述。它允许从一个已存在的对象产生新的对象。结果在逻辑上导致了对象的等级关系。

内联函数 (Inline Functions) : 在类内定义的成员函数即方法。编译器遇到关键字 `inline` 时, 编译内联函数的代码。其结果是减少函数调用过程给程序带来的开销。

实例 (Instance) : 一个预定义对象的存在。

后期联编 (Late Binding) : 见动态联编。

成员函数 (Member Function) : 类中的方法。一个方法只不过是在类中声明的一个函数原型。该函数可以是内联的 (即在类中定义的), 也可以不是。

方法 (Method) : 见成员函数。

多继承性 (Multiple Inheritance) : 一个对象从两个或更多基类中继承数据和函数的一种特性。

对象 (Object) : 类的实例或事件。

面向对象 (Object Oriented) : 一个术语, 用来修饰任何支持使用对象的系统或语言, 它允许将单个的代码块聚合成分层级的类, 允许在不需要重定义的情况下的代码重用和代码扩展。

运算符重载 (或操作符重载 Operator Overloading) : 一种特性, 它允许大多数双目或单目运算符改变其原来的含义。

多态性 (Polymorphism) : 使用指向基类的指针来引用虚函数的一种特性。它允许多个对象在多级实现中根据自己的需要来定义方法。

私有性 (Private) : 一种特性, 它允许将数据和函数封装在一个单独的类中。在一个对象中, 数据和函数默认为私有的。参见公共性, 受保护性。

私有派生类 (Privately Derived Classes)：在从基类派生时，基类前用关键字 `private` 限定的那种派生类。它使得基类中公共和受保护成员成为派生类的私有成员，基类的私有成员继续保持它本身的私有性（只为基类私有）。

受保护性 (Protected)：一种特性，它允许函数只被封装在基类和它的派生类内，参见私有性和公共性。

公共性 (Public)：一种特性，它允许一个类的数据成员和函数被其派生类和其余的应用所访问。

公共派生类 (Publicly Derived Classes)：在从基类派生时，基类前用关键字 `public` 限定的那种派生类。它使得基类的公共的和受保护的成员成为派生类的公共的和受保护成员。基类的私有成员继续为基类私有。

可重用性 (Reusability)：见面向对象

This：不可见的指针标识符，它自动指向存在的对象，以及对象的每个实例。它被隐式地传给重载的运算符函数。

虚函数或虚方法 (Virtual Function or Method)：虚函数通常是一组名字相同但具体实现不同的函数。在某一时刻，根据所指向的特定对象地址来引用相应的函数。虚函数通过动态联编来实现。参见动态联编和多态性。

附录 B C++ 句法

格式说明:

- 冒号后面是对该句法的定义。
- 每个多重选项都各占一行。
- 可选元素包含在尖括号中 (<>)。

声明句法:

声明:

<声明说明符><声明列表>

声明说明符:

存储类说明符

类型说明符

函数说明符

说明列表:

声明

存储类说明符:

auto

register

static

extern

类型说明符:

类型名

类说明符

枚举说明符

const

volatile

函数说明符:

inline

virtual

friend

类型名:

类名

自定义类型名

char

short

int

long

signed

unsigned

float

double

void

类说明符:

class

struct

union

枚举说明符

enum <标识符>{枚举列表}

枚举列表:

标识符

标识符 = 常量表达式

表达式句法

表达式:

基本表达式

表达式 双目运算符 表达式

表达式列表

运算符:

=	*=	/=	%=	+=	-=	<<=	>>=
&=	^=	=	++	--	<<	>>	<
<=	>	>=	==	!=	&	^	
&&		?:					

基本表达式:

常量

字符串

this

::标识符

::操作符函数名

名字

名字:

标识符

操作符函数名

类名::标识符

类名::操作符函数名

类名::类名

类名::~类名

内存分配表达式:

<::> new <表达式列表> 类型说明符 <初始值>

内存释放表达式:

<::> delete 表达式

类声明句法

类说明符:

基本说明符 {成员列表}

基本说明符:

类名

virtual <存取说明符> 类名

friend 类名

成员列表:

成员声明 <成员列表>

存取说明符:

private

public

protected

成员声明:

<声明说明符><成员列表>

函数定义

成员列表:

成员声明 <成员列表>

存取说明符: <成员列表>

操作符函数名:

操作符 操作符

操作符:

+	-	*	/	%	^
&		~	!	=	<>
+=	-=	*=	/=	%=	^=
&=	=	<<	>>	<<=	>>=
==	!=	<=	>=	&&	
++	--	,-	>*-	>	()
[]	new	delete	sizeof		