

C++ 语言命令详解(第二版)



全书由两大部分及四个附录和一个词汇表构成, 第一部分介绍 C++ 的基本概念以及 C++ 程序设计方法, 这部分从 C++ 最基本的概念讲起, 覆盖最新 C++ 标准中的所有概念并重点介绍类及其有关的函数和运算符. 第二部分可以用于 C++ 编程的参考, 分别详细介绍数据类型, 运算符, 类型转换, 关键字, 预处理指令以及库函数和库类. 四个附录分别介绍 C/C++ 的区别, ANSI 及 C++ 成员的速查表. 本书最后有 C++ 术语及概念的词汇表. 本书可以用作学习 C++ 的标准教材, 也可成为高级程序员的有价值的参考书.

目 录



前言	(1)
第一部分 了解 C++	(1)
第一章 C++ 的功能	(3)
C++ 的起源	(4)
从 C 到 C++ 的转换	(4)
类:对象的组织形式	(5)
封装:方便的编程方式	(7)
多态:分散化控制	(8)
C++ 及其增强类型	(9)
函数重载	(9)
运算符重载	(10)
C++ :未来就在眼前	(10)
第二章 C++ 编程的基本特征	(11)
第一个 C++ 程序	(11)
添加数据声明	(12)
关于 #include	(14)
语句的功能	(16)
赋值	(16)
打印输出	(17)
获得输入	(18)
C++ 的特殊规定	(19)
注意分号!	(19)
赋值语句也是表达式	(20)
在程序中添加函数	(21)
函数的通用语法	(22)
函数例子	(23)
void 型函数	(23)

局部变量、全局变量以及其它变量	(24)
局部变量	(24)
全部变量	(25)
静态变量	(26)
外部变量	(27)
有趣的控制结构	(28)
if 语句	(28)
while 语句	(30)
加强对 C++ 运算符的印象	(30)
递增运算符和递减运算符	(31)
赋值运算符	(32)
位运算符、逻辑运算符以及移位运算符	(33)
位段:十分简洁的数据结构	(35)
第三章 指针、字符串及其它	(37)
更为快捷的数据传递方法	(37)
指针与通过引用进行数据传递	(38)
通过引用进行传递的步骤	(39)
两个通过引用进行传递的完整的例子	(40)
指针与数组	(41)
数组的基本知识	(42)
使用指针进行循环处理	(44)
C++ 字符串	(47)
指针与动态内存分配	(51)
使用 malloc 和 free(C 与 C++ 均支持)	(51)
使用 new 和 delete(C++ 特有)	(53)
第四章 输入、输出和 C++	(55)
流的概念	(55)
流操作符<<和>>	(57)
输入输出的格式	(59)
stdio.h 中基于行的输入	(60)
用 stdio.h 对文件进行输入输出	(63)
文件操作符和流操作符	(65)
争议:使用流还是不使用流	(67)

第五章 类	(69)
类的开发:一个更好的字符串类型	(69)
成员函数	(71)
将代码组织到文件中	(72)
分号符(;):一个备须注意的语法现象	(73)
对象	(74)
调用一个成员函数	(75)
成员函数	(78)
对象指针	(80)
使用私有数据的好处	(81)
动态内存分配实现	(84)
对象的生存期:构造函数及其它	(87)
内联函数	(89)
类的一种特殊情况:结构	(90)
类的远景	(90)
封装	(91)
类、对象和实例	(91)
类的重新使用及发布	(91)
第六章 构造函数	(93)
构造函数的重载	(93)
构造函数的两个例子	(94)
默认构造函数	(95)
复制构造函数和引用	(96)
引用:使用地址操作符(&)的一个新方式	(98)
编写复制构造函数	(100)
const 关键字	(101)
其他构造函数的例子	(102)
C++ 如何调用构造函数	(103)
总结:构造函数的重点	(104)
重载构造函数	(104)
默认的构造函数	(105)
复制构造函数	(105)
初始化和转换	(105)

第七章 类的运算(操作符重载)	(107)
基本表达式.....	(107)
编写加法(+)操作符函数.....	(108)
操作符函数的调用过程	(109)
还需注意的问题(其它加法函数)	(110)
友元的使用	(111)
赋值函数的编写.....	(113)
赋值函数的定义	(114)
this 指针及其用法	(115)
赋值操作里的引用类型(&)	(116)
编写类型转换函数.....	(116)
CStr 类的小结	(118)
另一个类操作符的实例.....	(121)
操作符重载进阶.....	(123)
操作符函数的命名	(123)
双操作数操作符	(123)
单操作数操作符	(124)
赋值操作符	(125)
不同类型对象之间的赋值	(125)
其它赋值操作符(+=、-=等)	(125)
自增和自减操作符	(126)
下标操作符([])	(127)
new 和 delete 操作符	(127)
函数调用操作符().....	(127)
语法规则小结.....	(128)
第八章 继承 C++ 的优越特性	(129)
由 CStr 类谈起:软件工程里的一个窘境	(129)
CStr 的派生类	(130)
派生类的语法表达式	(130)
编写新类的函数	(132)
函数重载和作用域的划分	(133)
继承的层次	(134)
使用继承和不使用继承的比较.....	(135)

Public、Private 和 Protected 所决定的访问权限	(138)
另一个实例:轿车类(Fast Cars)和继承关系树型图	(141)
基类构造函数	(144)
基类和指针	(145)
第九章 虚函数及其性质	(149)
关键字 virtual 的使用	(149)
虚函数的使用场合	(151)
菜单命令的实例	(152)
基类的声明和定义	(153)
菜单(Menu)对象的声明和定义	(153)
对象的使用	(155)
虚函数在应用上的优点	(157)
无实现函数(纯虚函数)	(158)
如何实现纯虚函数	(159)
第二部分 C++ 参考大全	(163)
轻松学习 C++	(165)
C++ 元素速查表	(183)
第十章 数据类型	(193)
整型数和浮点数	(194)
第十一章 运算符	(207)
赋值运算符	(216)
位运算符	(217)
逻辑运算符	(220)
取模运算符(%)	(221)
指针运算符	(221)
指针到成员(Pointer-to-Member)运算符	(222)
关系运算符	(224)
作用域标识符(;)	(225)
第十二章 类型转换操作符(cast)	(227)

第十三章 C++ 的关键字	(237)
第十四章 预处理器指令,宏和运算符	(281)
指令	(281)
预定义宏	(291)
预处理器运算符	(294)
第十五章 库函数	(297)
库函数简介	(297)
第十六章 I/O 库类与对象	(367)
I/O 库类概述	(367)
与 I/O 类的通信	(369)
扩展输出流移位符(<<)	(369)
扩展输入流移位符(>>)	(370)
I/O 操作符	(372)
I/O 标志符	(373)
C++ 的新类	(374)
C++ I/O 类和对象的总结	(375)
附录 A C 与 C++ 的区别	(394)
附录 B ANSI C++ 特征总结	(395)
新式头文件	(396)
ANSI 类型转换运算符	(396)
模板与异常处理	(397)
其它关键字	(397)
if 语句中变量的作用范围	(398)
具有枚举类型的函数的重载	(399)
嵌入类的前向引用	(400)
附录 C 标准异常	(401)
附录 D ASCII 字符代码	(403)
C++ 术语及概念词汇表	(405)

第一部分

了解 C++

第一章 C++ 的功能

第二章 C++ 编程的基本特征

第三章 指针、字符串以及其它

第四章 输入、输出和 C++

第五章 类

第六章 构造函数

第七章 类的运算（运算符重载）

第八章 继承 C++ 的优越特性

第九章 虚函数及其性质

第一部分从 C++ 的基础讲起。本部分共有九章内容，覆盖了 C++ 所有的基本语法，如数据类型及变量、运算符、控制、语句、函数以及面向对象的编程概念，它包括了类和对象、运算符重载、变量范围、虚函数、继承和多态。如果你刚刚开始学习编程，建议你从第一章开始看起；如果你是经验丰富的 C 程序员，则可以跳过头几章而从第四或第五章开始学起。

第 一 章

C++ 的功能

在计算机的黑暗年代,程序员是机器的奴隶。开发者不得不用二进制代码(用 1 和 0 表示)编写所有的指令。二进制代码是计算机的母语。随着时代的发展,新的编程语言为程序员提供了表达算法的更好方式。计算机语言的改进意味着程序员可以将注意力从计算机的内部结构转移到程序的目的上来。

面向对象的编程方式使得软件的发展向前迈了一大步。尽管它的功能可能被夸大了,但是面向对象的编程方法确实提高了程序员的效率。一般来说,在面向对象的编程方式出现之前,最明显的编程结构是代码与数据之间的分离。这种分离精确地反映了计算机的内部工作方式,不过这并不是描述世界的理想方式。

而面向对象的编程方式却类似于人类的大脑。人类的大脑是单独的脑细胞所形成的巨大组合。如果借用计算机术语,每个细胞就是一个对象,它具有自己的基础材料(数据)以及可以控制的行为(代码)。没有必要将脑细胞的这些不同部分分离开,也就是说没有必要区分哪里是数据哪里是代码。更没有必要区分哪里存放着是大脑的所有数据,哪里存放着大脑的所有代码。

这个类比说明了面向对象的含义:它将对象作为基本的单位从而取代了传统的方式。对象由状态信息以及行为组成,并且每个对象都能象大脑中的细胞一样发出信号并可以对刺激作出响应。

使用面向对象的语言编写的程序不能自动地将现实模型化。从这一点来看它并不比其它程序优越。一个成功的程序是认真思考、详尽计划以及勤奋的结果,而不是语言选择的结果。并且,如果你想要使用面向对象的方法(越来越多的系统软件要求这种方法),那么象 C++ 这样的面向对象的语言的许多特征可以相当方便并十分有效地帮你达到目的。

为什么要使用 C++ 呢? C++ 可能不是使用最广泛的面向对象的语言;更多的人使用 Visual Basic,不过 Visual Basic 是否是面向对象的还是有争议的。但是 C++ 的争议更多。很明显,越来越多的程序员将 C++ 视为最完整的面向对象的语

言。有人争辩说 Smalltalk 或 Eiffel 已经完整地实现了面向对象,而 C++ 可能设置了一些衡量其它语言的标准。对程序员来说,了解 C++ 就像几年前了解 C 那样重要。今天任何一个大的、重要的开发项目通常都是使用 C++ 来编写的。对开发管理员来说,C++ 是一门艺术。

对某些人来说,Java 是刚刚出现的未来语言,至少对 Internet 中的程序是这样。但是应该了解到,Java 有许多语法是从 C++ 中借用去的。事实上,如果先学习了 C++,Java 学起来将十分容易。

在任何情况下“为什么要使用 C++?”都是一个很有意义的问题。与 C 一样,答案很大程度上源于对功能和效率的双重需要。

C++ 的起源

C++ 是挪威编程人员的又一发明。在挪威先是出现了一个名叫 Simula 的语言,它也是最早使用类这个概念的语言(类是程序的单位,包含数据以及相关的函数)。类,像在本书中将要学到的那样,与对象这个概念紧密相关,类是一种对象类型。

Simula 用于开发事件模型。尽管事件驱动模型与面向对象的模型在概念上并不一致,但二者有许多共同之处。这就是为什么 Visual Basic、Windows、OS/2 Presentation Manager 以及许多其它事件驱动的结构是面向对象的或者至少是基于对象的原因。因为面向对象的基础是可以响应消息的独立实体,所以它是实现事件驱动系统的很自然的方式。

1978 年,在 Cambridge 大学的计算实验室,作为博士论文的一部分,Enter Bjarne Stroustrup 需要编写用于分布式计算机系统的模拟器。Stroustrup 发现使用 Simula 的类可以完整地描述网络中不同机器的接口。问题是对大规模的系统来说,Simula 的效率太低。他需要将 Simula 面向对象的特征与某种语言,如 C 语言的能力和效率结合起来。

C++ 就是这么产生的。Stroustrup 在 C 中使用了类并添加了 Simula 的数据类型创建了 C++。“具有类的 C”在成为今天使用的 C++ 标准之前走了几次弯路。不过 C++ 这种语言基本上仍然实现了 Stroustrup 最初将 C 与 Simula 的类结合的愿望。

从 C 到 C++ 的转换

可能你以前听说过,C++ 是“更好的 C”。这句话意味着 C++ 是 C 的超集。说这句话的人认为可以使用 C++ 编写所有用 C 编写的程序。另外,C++ 进行了一些改善,无须使用面向对象的方法就可以将这些改善添加到你的程序中。

上面陈述的第一部分只能说是大致正确。许多 C 程序无须修改就可以作为 C++ 程序进行编译,但是对某些程序来说,由于存在一些语法区别而无法通过编译。本书的一个目标就是尽可能清晰地指出这些区别。(对这个问题的完整回答,请参考附录 A)

C++ 重要的一点是它没有将面向对象强加在你身上。如果你是 C 程序员,应该清楚在转换到 C++ 时,只需注意 C++ 引入的为数不多的限制,就可以象从前一样正常工作。不必将任何东西重新编写为对象。只要花足够的时间,你就可以将 C++ 的新功能逐步添加到你的程序中去。

类:对象的组织形式

尽管你可能想要了解 C++ 的核心内容——面向对象,但是一开始,你还是需要花一些时间和精力来学习 C++ 的基本知识。从概念上讲,对象只不过是内嵌有与之有关的函数的数据结构。(类是对象的类型;关于类,本书以后会经常提到。)看起来好象 C++ 有相当多的运算符、关键字以及规则,并且它们大部分用于应付特殊情况。不过对象的概念本身却十分灵巧并相当简单。

传统程序的组织形式代表的是事物在计算机软件中的实现方式而不是代表事物在现实生活中实现方式。从技术上来说,计算机内存的所有内容都由数据构成,但是内存中存储的内容中最重要的部分是称为代码的特殊数据。代码由发送给处理器的指令构成:对两个数进行加、跳转到新地址等等。内存的其它部分由数据构成,正如人们所说的那样,数据是提供给用户或计算机使用的信息。

因为(与人类的大脑不同)计算机只有一个中央处理器,要由这个处理器来分别处理代码和数据,所以代码与数据在计算机内存中是截然分开的。除了偶尔跳到新的位置,中央处理器顺序地执行代码,由于这个原因,大段的代码不得不结合成代码段。传统程序的组织形式反映了这个事实。一般来说,代码可以使用函数图表来表示,由指令集构成的每个函数作为一个代码块来执行。数据由记录(结构)、数组以及表格组成。

尽管有这么多的结构,最终的程序仍然是函数的层次图表和与之分离的数据结构的集合。这种编程语言在这两个部分之间没有加入联系(如图 1-1)。

代码与数据的分离会产生问题吗?对简单的程序来说,这种分离不会有什么影响。但是程序员必须牢记哪个函数使用哪个数据结构,否则就会出现错误。

打开计算机,你就会明白计算机硬件本身并不适合传统的软件模式。计算机结构本身并没有反映代码与数据的分离!相反,计算机的主板是由一些独立的芯片组成,这些芯片通过电路在彼此之间发出或接收信号(如图 1-2)。

图 1-1
由传统程序
组织形式分
离代码与数
据

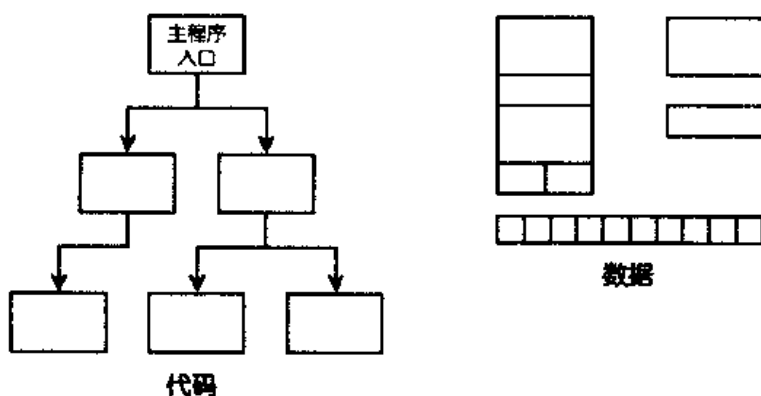
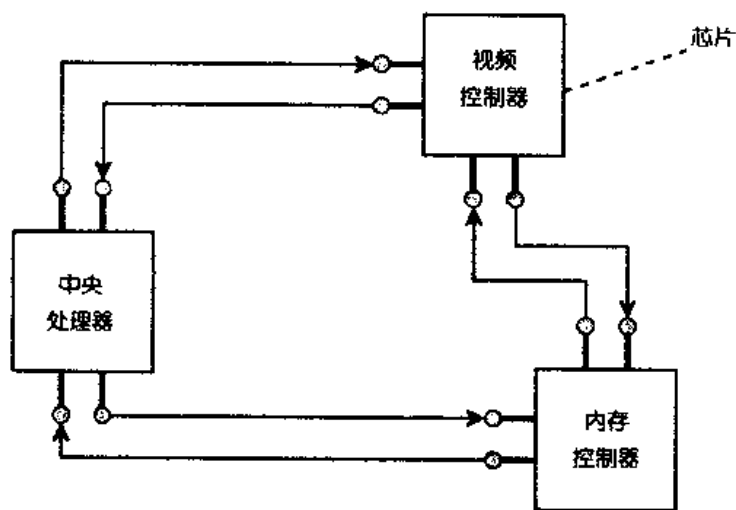


图 1-2
计算机箱内
部的活动



在面向对象的编程方式中,这些芯片变成了对象。按照 C++ 的术语,芯片(如奔腾)的制造及其模型是一个类;单独的一个芯片是一个对象。芯片的软件描述既不是纯代码的也不是纯数据的。与大脑细胞类似,一个芯片既有代码(行为)又有数据(状态信息)。

如果你开始编写一个描述计算机内部硬件的程序,你会发现对于程序的设计及编写来说,使用对象来表示每个芯片是相当有用的。以下就是面向对象的 C++ 与 C 截然不同的所在:组成程序的基本单元不是代码或数据,而是对象的类,而类则是包含有数据与代码的一种新类型(图 1-3)。

封装:方便的编程方式

图 1-3 中最重要的观点之一是黑匣子的概念。芯片仅通过特定的管脚进行通信;除了这些接口,每个芯片对其它的芯片来说是不可知的。没有任何一个芯片可以到达另一个芯片的内部并干涉其内部工作。另外,计算机的设计者也不需要知道芯片的内部电路,只要该芯片的输入输出如芯片制造者所说的那样工作正常即可。

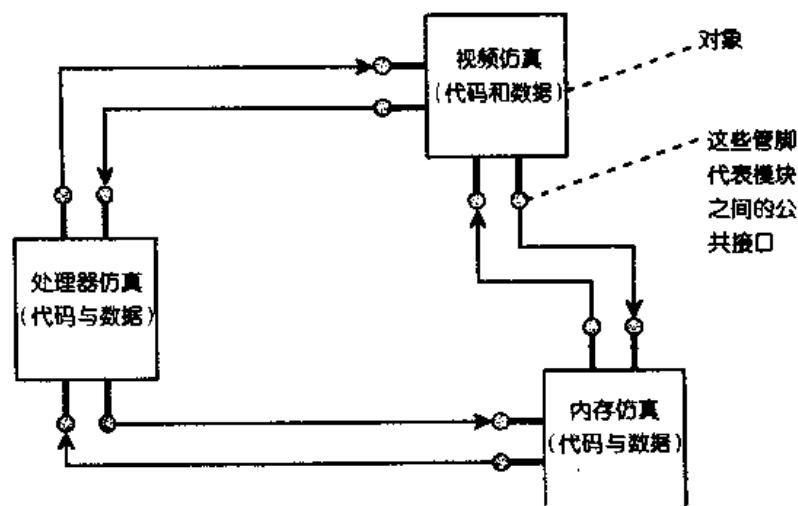


图 1-3
对象的
组织形式

黑匣子具有这种特征是必要的。由于每个芯片都有特定的状态指明芯片的运转情况,如果每个芯片都工作正常,系统就会正常工作。只要芯片使用同样的规定,就可以用新的芯片来代替旧的芯片而整个系统也会象以前一样运转正常。尽管这个新的芯片的内部结构与旧的芯片完全不同,你也无须担心。只要芯片与系统按照同样的方式相互作用,任何两个芯片都是可以互换的。

没有什么对互换性的要求比在软件开发中更大了。在一个典型的项目中,软件组要经常重写程序每个部分的内容。打比方说,为了查找错误、提高效率或者增加新功能,程序员要不断地用新的芯片代替旧的芯片。按照传统的编程方式,程序的各个部分没有完全明显分开,这样的程序常常会带来灾难性的后果。对程序任何部分的改变都可能会影响程序的其它部分。例如,Pete 现在的工作是使用 Camille 正在编写的程序的一些内容。但是一旦 Camille 对程序进行了某些修改,Pete 的所有假设就都是无效的,于是错误就发生了。

从某种程度来说,C 语言的功能也可以解决这个问题。可以在文件级别对不同模型(接口)之间的连接进行管理。例如,可以让 Pete 与 Camille 工作在不同的文件中并且在以后注明对所声明共享数据的限制。C 语言的某些功能(如关键字 `extern`

及 static)可以控制某个文件中的数据及函数是否对其它文件可见。

C++ 提供了在程序不同部分之间更好地共享数据及进行函数控制的方法。数据的保护部分不再限于文件级别,而是可以如你所愿——根据类任意扩大或缩小。在 C++ 里,可以在类的级别使用各种私有函数或数据。函数和数据都被视作成员。这些私有成员对程序的其它部分是不可见的;不能对它们进行访问。对象的公有成员构成了对象的接口。这些公有成员构成了对象外部可见的“管脚”。程序的其它部分可以引用这些成员,前提是使用这些成员没有发生改变。同时,你可以按照自己的意愿重新编写类的内部。

内部与接口之间明显的分离称为封装。封装意味着“保护某些事物的内部”。C++ 的封装方式不仅提供了更大范围的控制,而且使得在源代码中公有/私有的区别更加清晰。C++ 有助于表明哪一部分是类的接口,哪一部分是类的内部。

● 注意

在第一次学习 C++ 及面向对象时,你可能也使用对象及类的可互换性这些术语。但是记住以下这些区别很重要:类是一种类型而对象是这种类型的一个实现。在第五章将会更多地强调这个区别。

多态:分散化控制

面向对象的系统的一个好处是每个对象都尽可能地独立。我可以给一个对象发送一个通用的信号,该对象能够正确地响应。我不必了解该对象是如何产生这些响应,甚至不必知道对象的确切类型。

如果使用硬件术语,可以这样说,我能够发送一个通用的打印命令而无须提前知道打印机的制造过程以及它的模型。只要每个人都遵守同样的通信协议,我的打印命令将一直起作用。一般来说,该命令的功能是:打印文档。而点阵打印机对该命令的响应方式与激光打印机的响应方式完全不同。

面向对象这一点意味着主函数或主循环知道的越少越好。如何执行命令的决定应该在对象内部产生。这时,对主函数来说要做的只是发送一个通用信号。

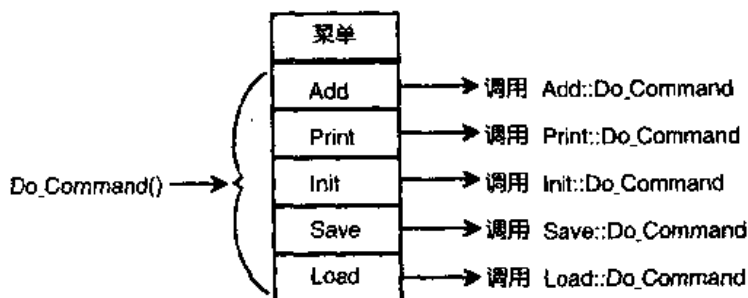
在第九章我将给出一个例子,用于说明程序是如何实现这一点的。该例子使用一组菜单命令。在用户选择一个命令时,主函数给菜单对象发送一个消息:Do_Command。按照命令的不同,程序按不同的方式进行响应(如图 1-4)。

因为程序没有被限制在特定的命令集合中,所以这种方式十分有效。主函数所要做的只是给相应的菜单对象发送一个 Do_Command 命令,这样程序就会作出正确的响应。使用这种方式,在将来开发新的菜单命令时无须重写主函数,而且也可以动

态地修改菜单项。在运行时可以增加或修改菜单,这种功能对传统方式来说是很难实现的。

图 1-4

多态的菜单
系统给菜单
对象发送一
个通用命令



而且,对于诸如 Macintosh 或 Microsoft Windows 等图形用户界面系统来说,这种设计是这些系统的关键部分。操作系统不可能预先知道所有的窗口的运行方式;否则,开发的应用程序将相当有限。另外,操作系统必须按照每个窗口的要求对通用信息(如“初始化显示器”)作出特定的响应。

通过使用回调函数,也可以在 C 中实现同样的功能。但是,回调函数并不规范。处理控制对象对消息的响应,C++ 使用了更加结构化的方法,包括继承层次和虚函数。在第八章和第九章中可以更多地了解这些术语。在实际应用中,重要的在于知道 C++ 的编程方式是更可靠,更方便的编程方式,并且这种方法是规范的。

C++ 及其增强类型

面向对象是 C++ 最吸引人的地方。但是 C++ 与 C 之间最根本的区别在于 C++ 包含了更多的关于类型的概念。而且也可以将面向对象看作是增强的类型之一;C++ 增强的类型不仅可以包含代码而且可以包含数据。

无论你是否使用对象,C++ 对类型的加强有助于程序的安全,增加了程序的可靠性。在 C 中,在一段代码中可能使用一个声明为 4 字节浮点型的变量,而在另一段代码中,可能将该变量作为 8 字节的变量来使用,这种情况很容易发生。这样做的结果是运行时会出现大量的错误。C++ 通过使用类型安全连接防止发生这种事情。类型安全连接阻止连接程序将不同类型的符号视作同一个变量。

重载是 C++ 广泛使用的另一个概念。重载使用类型信息来判断函数与运算符应该如何运行。它也反映了在 C++ 中类型的重要性。

函数重载

重载意味着重新使用某个名字。在 C++ 里,重载的最普通的例子就是函数重载,即编写同一个函数的不同版本。实际上,重载的函数可以完全不同,具有各自的函数定义而且可以是分别声明的。但是从程序员的角度来讲,函数重载是使用同一个名称实现不同功能的简便方式。

假如要编写函数 `GetNum` 的两个不同的版本。其中一个使用指向整数的指针作为参数,另一版本的参数是一个指向浮点数的指针。这两个版本具有独立的函数声明。`C++` 使用变量列表中的类型信息来区分这两个函数。在源代码中调用 `GetNum` 时,`C++` 将查看变量的数据类型并以此判断应该调用哪一个函数:

```
void GetNum(int* pn);  
void GetNum(double* pf);
```

运算符重载

运算符重载类似于函数重载。`C++` 在遇到运算符时(如 `+`、`-`、`/`、`*`)时,它通过检查运算符的参数类型来决定如何进行运算。也就是说,可以对任何类型使用运算符。事实上正是如此。在 `C++` 里,只要类型声明中定义了操作符的运算方式,就可以对任何类型使用加(`+`)、减(`-`)或者任何其它标准运算符。换句话说,可以编写一个对类对象使用加法(`+`)操作的函数。例如,在第五、六、七章中,创建了一个类 `CStr`,它使用加号(`+`)来连接字符串,类似于 `Visual Basic` 中加号的功能:

```
CStr name,first("Bernie"),last("Schwartz");  
  
name = first + last;
```

运算符重载可以用来创建类型(或者类),这些类型可以视为对 `C++` 的扩充。它们可以如 `int`(整型)或 `long`(长整型)一样成为程序的基本类型。就象对 `C++` 中那些基本类型使用运算符一样,也可以对创建的类型使用任何运算符。

C++ : 未来就在眼前

对未来作出预言纯粹是骗人的把戏。正如 St. Augustine 曾经说过的那样,最真实的只有现在,未来与过去只不过是幻影。

与 `Java` 及 `Visual Basic` 一样,`C++` 的未来是不可限量的,尽管它也与 `Java` 及 `Visual Basic` 一样有某些缺点。新开发的操作系统的结构越来越倾向于面向对象。如果想要使用这些系统,就必须将软件作为对象——可以响应消息的独立的单元——来学习。在这方面,`C++` 非常优秀。

除了这些,`C++` 在保留 `C` 的效率及对硬件的直接访问之外,它还纠正了 `C` 对类型的一些观点。`C++` 是未来而不是现在的系统编程语言。

尽管如此,应该说明的是,现在最好还是学一学 `C++`。初学者很可能为 `C++` 语言中大量的术语(如封装、虚函数、抽象)所迷惑。不过不要被这些术语羁绊,`C++` 不是宗教时代的咒语(尽管 `C++` 中总是使用一些莫名其妙的名词),相反,由于包含了从低级操作到灵巧的面向对象的数据描述,`C++` 无疑是一套十分有用的工具。如果有耐心一步一步地学习 `C++`,你就会很好地使用这些工具。

第 二 章

C++ 编程的基本特征

C 与 C++ 的核心特征与其它流行的计算机语言相比来说略微复杂一些——但不是十分复杂。首先,必须花些时间来理解分号与运算符的用法。但是这些复杂性也使得程序更有效,具有更大的灵活性,而且易于编写简洁的程序。

对 C 和 C++ 的初学者来说,最可怕的东西是指针——对数据的间接引用。由于指针十分重要,所以把它单独编成一章(参看第三章)。不过现在可以松口气,别这么早就让指针困扰你。

C 与 C++ 是有明显区别的两种语言,但是它们也有一些相同的基本结构。简单地说,本章只是介绍一下 C++, 指出 C 与 C++ 的不同之处。

第一个 C++ 程序

输出简单的字符串是对编程系统的第一个测试。使用 C++ 编写这样的程序尽管不象 BASIC 那样简单,但是也不是很繁:

```
#include <stdio.h>

void main() {
    printf("Can you C++ now?");
}
```

在输入这段程序时,你可能注意到几件事。首先,与诸如 FORTRAN 及 BASIC 等语言相比,C++ 使用了一些标点符号:分号(;)和大括号({})。编程时必须如实地输入这些符号。

但是,有时 C++ 提供其它语言所没有的自由。对任意一行来说,可以使用各种方法进行分隔,甚至可以在不同的行中加入成员变量。C++ 能够正确识别它们。

```
#include <stdio.h>
```

```
void  
main()  
{  
  
    printf  
        ("Can you C++ now?");  
}
```

因为 C++ 使用分号(;)来判断哪里是 printf 语句的结束,所以上面的程序可以正确地运行。C++ 通常忽略物理行的结束(在使用命令和注释的时候除外,以后将会讨论到这个问题)。一方面,在输入时要注意一些事情,另一方面,C++ 的语法又提供了 BASIC 所没有的自由。

● 注意

将上面的代码输入到源文件之后,需要使用编译器或开发环境对源文件进行编译。编译的过程就是将代码转换为可执行文件的过程。对于如何创建程序的信息,请参看编译器文档。

● C /C++

上面谈到的事情对 C 来讲都适用,除了下面这一点:在 C 中,main 前面的关键字 void 可以不用,而在 C++ 里,却是必须要求的。void 的意思是函数(main)没有返回值。后面还会更多地提到 void。

当然,可以用自己的字符串来代替“Can you C++ now?”下面是该程序的通用模板:

```
#include <stdio.h>  
void main() |    printf("enter - your - string - here"); |
```

在有 *enter - your - string - here* 的地方输入自己的字符串。

添加数据声明

在可以存储和控制消息时,程序才开始变得有用。这样的程序需要在某些地方存储数据变量。

C++ 的变量声明由紧跟类型名称的变量名称及分号组成。变量类型告诉编译器在该变量中存储何种类型。类型的例子有 int、short、long、float、double 等等。下面的例子声明了两种 int 型(int 中存储整型数,没有小数部分)的变量:

```
int variable_name1;  
int variable_name2;
```

C++ 的变量声明十分简单。它不涉及任何外来的关键字,如 Dim 或 var。

可以在同一行创建多个数据的声明。使用逗号将每个变量分隔开。下面的例子声明了三个 short 型变量(i,j,k)以及三个 float 型的变量(x,y,z)。这些类型分别是短整数和浮点数。

```
short i,j,k;  
float x,y,z;
```

C++ 中数据声明的基本语法有另外一个特点:可以在声明的时候对它们进行初始化。也就是说,可以在声明的时候给变量一个起始值,以后也可以改变变量的值。

使用等号后跟一个值来对变量进行初始化。例如:

```
int my_var = 0;      //my_var 初始化为 0  
int your_var = 1;    //your_var 初始化为 1  
int a,b = 10,c = 12; //b 和 c 进行了初始化,a 没有进行初始化
```

现在最好谈谈 C++ 中的注释。在 C++ 中,注释以双斜杠开始一直到行结尾,由二者之间的所有文本组成。编译器忽略注释中的任何东西。理论上讲,可以在注释中添加任何内容,但是人们通常使用注释说明程序是如何运行的。

● C / C++

C++ 也支持 C 中的注释开始和结束符号(分别是 /* 和 */)。尽管有些 C 的编译器也支持 C++ 的注释符号(//),但是并不是所有的编译器都支持。

在声明并初始化数据之后,就可以创建略微复杂的程序了:

```
#include <stdio.h>  
  
void main()  
{  
    int x = 1;  
    int y = 2;  
  
    printf("The sum of x+y is %d",x+y);  
}
```

printf 函数支持格式化输出。它可以使用数字变量如 x+y 并可以将数字与其

它字符一起打印出来。格式化字符 %d 的含义是“将下一个参数按十进制整数的形式打印出来”。其中, d 代表十进制整数。打印浮点值需要使用 %f。

最基本的变量类型是整型和浮点型。浮点数据可以有小数而整型数据不可以有小数。浮点数据更灵活些, 但整型数据也很有用。如果预先知道某个变量不包括小数, 就可以将其声明为整型。

在表 2-1 中归纳了 int、short、long、float 以及 double 类型的基本特征。本章后面将提到数据输入函数 scanf。

表 2-1 常用数据类型的特征

类型	数据种类	典型大小	printf 格式化 字符	scanf 格式化 字符
short	整型	2 字节	%d	%hd
long	整型	4 字节	%d	%ld
int	整型	2 或 4 字节	%d	%d
float	浮点型	4 字节	%f	%f
double	浮点型	8 字节	%f	%lf

● 注意

在 scanf 中也可以使用 i(整型)以及 d(十进制整型)。二者的区别将在本章的 scanf 部分“输入数据的获得”中提到。

数据类型的大小决定了数据的范围。例如, 由于使用 16 位二进制数位(2 字节)表示的有符号整型数的范围为 -32768~32767, 所以 short 可以表示的范围为 -32768~32767。long 的范围为 2 百万左右。关于数据类型及其范围的详细内容, 请参看第十章。

关于 #include

现在, 你可能奇怪: “为什么要使用 #include 呢?”

在 C++ 中, 除了 main 函数, 所有的其它函数在使用之前必须先声明。函数的声明是告诉编译器如何使用该函数, 如: 函数参数及返回值(如果有的话)的类型。无论是自己编写的函数还是如 printf 等标准库函数, 都需要进行声明。除了在程序中进行声明之外, 更为方便的方式是使用专门用于声明函数的特殊文件(称为头文件)。

文件 `stdio.h` 就对所有的标准输入输出函数,如 `printf` 进行了声明。

`#include` 是指令而不是语句。在源代码文件中,在对程序进行编译之前,先进行指令的编译。指令与普通的代码行不同:它们以镑号(`#`)开始,以行的结尾而不是以分号(`;`)为结束。另外,头文件必须从源文件的第一列开始。`#include` 指令告诉编译器到头文件中读取。

一旦熟悉了 C++ 的标准库,无须查找就可以列出你所需要的头文件。表 2-2 列出了包含最常用的库函数的头文件。

表 2-2 常用的 C++ 包含文件

头文件	使用方法	说明
<code>stdio.h</code>	<code>#include <stdio.h></code>	标准输入输出函数,包括进行文件操作的函数。
<code>iostream.h</code>	<code>#include <iostream.h></code>	流运算符(C++ 独有),代替 <code>printf</code> 和 <code>scanf</code> 。第四章解释了这些流运算符的用法。
<code>string.h</code>	<code>#include <string.h></code>	字符串操作函数;例如,将某字符串复制到另一字符串
<code>ctype.h</code>	<code>#include <ctype.h></code>	检测以及修改字符串中单个字符的函数。
<code>math.h</code>	<code>#include <math.h></code>	三角函数、对数函数、指数函数以及其它工程函数。
<code>malloc.h</code>	<code>#include <malloc.h></code>	从系统中动态释放和分配内存的函数。(C++ 也提供了用于此种目的的内嵌操作符 <code>new</code> 和 <code>delete</code> 。)

下面的程序总结了简单 C 程序使用的通用模板。除了 `main` 函数外,此模板不涉及任何函数,但是在本章的后面,将给这段程序添加函数。`#include` 指令应该出现在程序的所有内容之前;当然,这个规定也不是绝对的,遵循这样的规则无疑将使程序的编写十分方便。

```
include...directives
```

```
void main(){
```

data-declarations-and-other-statements

在 *data-declarations-and-other-statements* 的地方既可以包括前面讲到的数据声明也可以包括称为可执行语句的语句。可执行语句的类别很大,在下面的部分将分开来讲。

●— C /C++ —

在 C 中,所有的数据声明必须在其它语句之前出现。C++ 放松了这个限制。但是,无论在 C 中还是在 C++ 中,在使用变量之前必须先对变量进行声明。

语句的功能

在变量声明之后,就可以使用语句进行一些操作。这些操作通常使用或控制已经声明的这些变量。在 C++ 中,可以将变量声明与语句混合使用。(在 C 中则不行。)但是无论在 C 还是在 C++ 中,由于在使用变量之前必须先对其进行声明,所以最好还是先进行变量的声明再编写语句。

除了后面将要提到的控制结构及函数以外,可以使用的所有操作可以分为三类:

- 赋值(涉及计算时使用最多)
- 打印输出
- 获得输入

下面三部分将讨论如何在 C++ 中进行这些操作。

赋值

C++ 中的赋值语句类似于其它语言的赋值语句,特别是 BASIC 中的赋值语句。C++ 中的赋值语句与 BASIC 的赋值的主要不同之处在于 C++ 的语句是以分号作为结束的。

在将数据分配给某个变量之前,要确保进行了变量的声明:

```
int amount,a,b,c;
```

要将数值分配给上面的四个变量,在等号(=)的左边加上变量的名称。在等号的右边可以放置变量、常量(例如 1 或 -240)或复合表达式(例如 $2 * c$)。

```
a = 1;  
amount = -240;
```

```
b = 2 * c;  
amount = a + 10 * b * -1;
```

在 C++ 中,星号(*)表示乘法。

上面这些语句类似于数据声明时进行的初始化。事实上,在 C++ 中,除了初始化能够创建变量并可以给它赋值之外,其它的与赋值运算通常没有什么区别。

● — C / C++ —

在 C 语言中对变量的初始化的限制更为严格;在数据声明中,只能使用常量进行初始化。在 C++ 中,这个限制有所放松;可以使用任何有效的表达式进行初始化。

打印输出

可以使用 printf 函数来输出简单的字符串或显示包含数字值的格式化输出。例如,下面的程序输出两行字符,每行都显示某个变量的值。当然,必须首先包含 stdio.h 并在使用变量之前进行声明。

```
#include <stdio.h>  
  
void main() {  
    int date = 10, d2 = 15;  
    float temp = 45.0, t2 = 33.5;  
  
    printf("On Dec. %d, temperature was %f. \n", date, temp);  
    printf("On Jan. %d, temperature was %f. \n", d2, t2);  
  
}
```

注意,%d 与整数相对应,而%f 与浮点数相对应。上面这段程序的输出结果如下:

```
On Dec. 10, temperature was 45.000000.  
On Jan. 15, temperature was 33.500000.
```

上面的例子也说明了 C++ 中打印输出的另外一方面:在 C++ 字符串中,特殊字符 \n 的作用是在新的一行打印(这个字符的名称为换行符)。如果将这个字符从上面的例子中删去,则输出将如下所示。printf 函数不能自动换行。

```
On Dec. 10, temperature was 45.000000. On Jan. 15,
```

temperature was 33.500000.

换行符是 C++ 的转义字符之一。转义字符都是以反斜杠开始。其它的转义字符有 \t 及 \". 前者输出一个制表宽度, 后者输出双引号。要输出真正的反斜杠, 使用两个反斜杠即可: \\. 关于 C 及 C++ 中转义字符的完整列表, 请参看第十章。

● C / C++

C++ 使用 cin 和 cout 对象分别代替 printf 和 scanf。但是 C 中则没有这两个对象。在第四章再介绍这些对象。

获得输入

scanf 函数读取数据。尽管 scanf 与 printf 一样也使用格式化字符, 但是它们有所不同: 首先, scanf 的所有变量都是内存地址, 所以除非变量是一个指针, 否则就必须在变量前加入一个地址操作符(&)。其次, scanf 对类型的要求更为严格。long 型的变量必须使用相应的 %ld(long decimal) 符号, double 类型的变量必须使用相应的 %lf(long floating-point) 符号。

下面的程序说明如何进行输入提示、如何获得四个不同类型数据的输入。不能使用 scanf 输出提示语句: 必须使用 printf。在此程序中使用的格式化字符有 %d、%hd、%ld、%f、%lf。

```
#include <stdio.h>

void main() {

    int i;
    short sho;
    long lng;
    float flt;
    double dbl;

    printf("Enter a value for i:");
    scanf("%d", &i);

    printf("Enter a value for sho:");
    scanf("%hd", &sho);

    printf("Enter a value for lng:");
    scanf("%ld", &lng);
```

```
printf("Enter a value for f1t:");  
scanf("%f", &f1t);
```

```
printf("Enter a value for dbl:");  
scanf("%lf", &dbl);
```

```
}
```

可以使用%i来代替%d。%i只用于指定整数的格式,%d用于指定十进制整数的格式。除了%i允许用户使用特定的前缀(0或0x)来指定八进制或十六进制格式之外,它们有同样的效果。例如:

```
printf("Enter a value for i:");  
scanf("%i", &dbl);
```

```
printf("Enter a value for sho:");  
scanf("%hi", &sho);
```

C++ 的特殊规定

C++与C有一些特殊的规定。虽然使用起来并不困难,但是如果你习惯了其它语言的编程方式而不注意的话,就可能遇到麻烦。

注意分号!

第一次使用C或C++编程时,最常出现的错误是忘记输入分号(;).也可能输入了C++不能识别的分号。

分号的使用规则是这样的:分号用于终止分号前的语句。下面几种情况除外:

1. 语句由指令构成,如#include或#define。
2. 语句是一个复合语句。复合语句表明在后大括号(})之后没有输入分号,除非后大括号是类或变量的声明的结束。

图2-1中的简单程序说明了规则的使用方法以及两种错误纠正方法。

因为此处是指令,所以不能以分号作为结束。

前后大括号不能使用分号作为结束(除非是类声明的结束)。C++使用分号而不是物理行的结尾作为语句的结束。在语句很长,甚至超越了物理屏幕的时候,可以

将语句分为几行。例如：

```
printf("On %d%d%d, the temperature was %f. \n",
      date,
      month,
      year,
      temp);
```

图 2-1

在 C++ 中
使用分号

```
#include <stdio.h>
void main () {
    int i = 5, j, k = 1;

    while (i > 0) {
        k = k * i;
        i = i - 1;
    }

    printf("k is %d", k);
}
```

这一行是一条指令，因此它不以分号结束

大括号结束不必续以分号（在类声明中除外）

C++ 语法的另一个特点是可以将多个语句放在一行，就象下面的四个赋值语句那样。不过，要使用分号作为每个语句的结束标志，最后一个也不例外。

```
a=0;b=0;c=0;d=0;
```

下面的内容将使用另外一种更为简洁的方式编写上面的程序。

赋值语句也是表达式

C++ 语言中最基本的单元之一是表达式。一般来说，表达式最后都获得一个值。表达式可以是变量、常量、函数调用或者可以由小的表达式通过运算符（如 +、-、*、/ 等）连接起来的复合表达式。

有趣的是，在 C 及 C++ 中，赋值运算符的功能（=）与其它运算符的功能相同，赋值表达式也与其它类型的表达式类似。也就是说，可以使用一个简洁的语句将多个变量赋予相同的值：

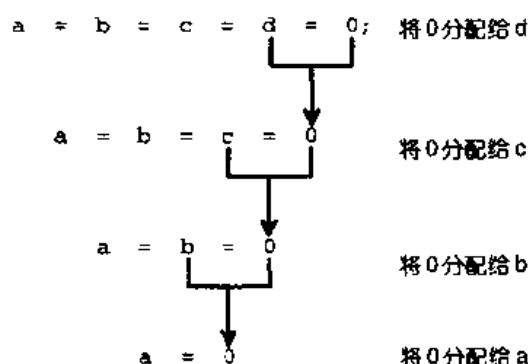
```
a=b=c=d=0;    //将变量初始化为 0。
```

赋值操作对所赋的值进行计算。如表达式 `d=0`，首先将 0 分配给 `d`，然后对 0 进行计算。在下一个表达式中又对这个值进行计算。赋值的顺序是从右向左进行，最右边的赋值首先进行。图 2-2 说明整个语句是如何进行每一次赋值操作的，在每一次赋值时，每次都使用 0 这个值。

需要注意的是,当赋值语句出现在 if 条件测试的内部时,它仍然是赋值语句,尽管看起来它象判断是否相等的测试语句。但是此时赋值语句计算的结果是真(true)还是假(false)。例如,在下面的程序中,赋值根据语句 $n=5$ 将计算 5 的值,即为真(只要是非零值)。

```
if(n=5)                                //错! 将 5 赋给 n
    printf("n is equal to 5. \n");      //总是执行
```

图 2-2
C++ 中的
多次赋值



无论 n 的值为多少,此处总是输出字符串。因为这里的 $n=5$ 是将 5 赋给 n 并返回 5。C++ 并不认为这里的 $n=5$ 的目的是判断二者的值是否相等。C 和 C++ 提供另外一种运算符(==)用于判断是否相等。该运算符进行比较并按照是否相等返回 true(1)或 false(0)。

```
if(n==5)                                //判断 n 是否等于 5
    printf("n is equal to 5. \n");      //如果 n 等于 5,则输出
```

在 C 及 C++ 的发展中,成千上万的程序员曾经辛辛苦苦地寻找程序中一个莫名其妙的错误,最后才发现是使用了赋值符(=)而不是相等判断符(==)。这个错误是学习 C 和 C++ 时最常遇到的错误。判断语句中的赋值符(=)通常都是错误的。其它的编程语言则不会出现这样的问题。C 以及 C++ 的随意性有点象搬起石头砸自己的脚。

在程序中添加函数

C++ 有一种子程序:函数。不返回值的函数应该声明为 void 型;返回值的函数的返回类型可以是 int 型、double 型或 float 型。这使得 C++ 的语句十分流畅,并且与 Visual Basic 不同的是,C++ 中没有 Sub 或 Function 等关键字。

以前你一定遇到过函数。函数可以没有参数或者有多个参数,这取决于是如何对它进行声明的,并返回一个在其它表达式中可以使用的值。图 2-3 说明了对 Pythagoras 函数的调用是如何进行的。

在图 2-3 中,表达式 `Pythagoras(3.0,4.0)` 产生对 Pythagoras 函数的调用,并将值 3.0 和 4.0 传给它的两个参数。函数使用 `return` 语句将控制交给调用者并返回值 5.0。

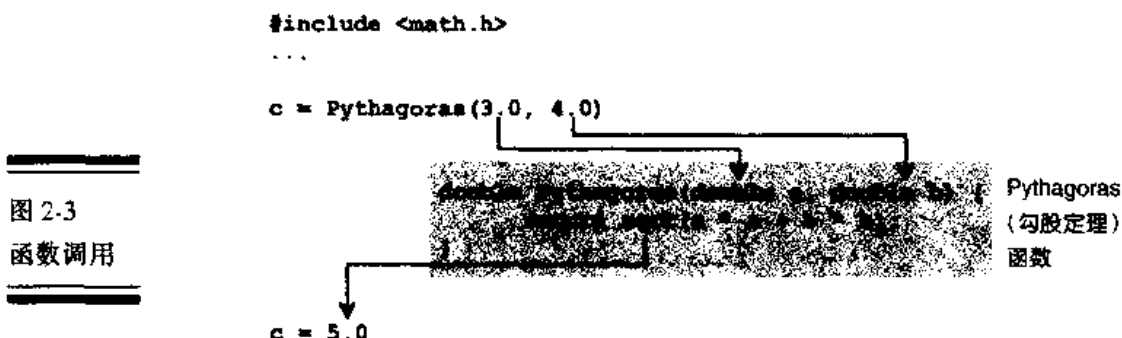


图 2-3
函数调用

函数的通用语法

使用函数时,C++ 程序的通用语法规则如图 2-4 中的模板所示。

在调用函数之前,必须先进行声明;`function_prototypes` 就是这个用处。函数原型将类型信息告诉编译器,以便编译器知道函数所需要的参数类型。函数原型的格式与函数定义的第一行类似:

```
return_type function_name(argument_list);
```

但是函数原型以分号(;)结束。而在函数定义的后大括号(})之后无须加一分号。因为函数原型以分号结尾而函数定义不是以分号结尾,所以很容易将它们区分开。

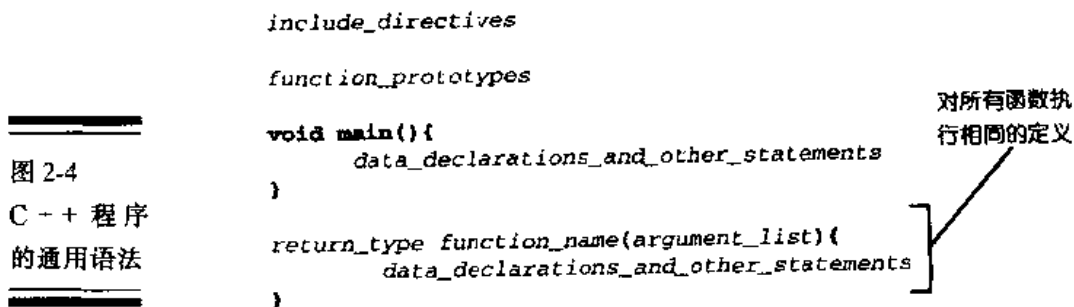


图 2-4
C++ 程序的
通用语法

● 技巧

函数原型与函数定义的标题头十分类似, 所以将函数定义的第一行进行复制就可以得到函数原型, 反之亦然。但是要相应地加入或删除分号。

函数例子

使用例子来说明函数的语法更容易让人理解。图 2-5 说明了函数语法的各个部分, 包括原型、函数调用以及函数定义。函数原型为函数调用做准备(告诉编译器如何进行类型检查), 函数调用某个执行函数而函数定义则告诉程序如何执行这个函数。

```
#include <stdio.h>
#include <math.h>

double Pythagoras(double a, double b);

void main() {
    double a, b, c;

    printf("Enter Angle 1: ");
    scanf("%lf", &a);
    printf("Enter Angle 2: ");
    scanf("%lf", &b);
    c = Pythagoras(a, b);
    printf("The hypotenuse is %f.", c);
}

double Pythagoras(double a, double b) {
    double c;

    c = sqrt(a * a + b * b);
    return c;
}
```

图 2-5
调用函数的
示例程序

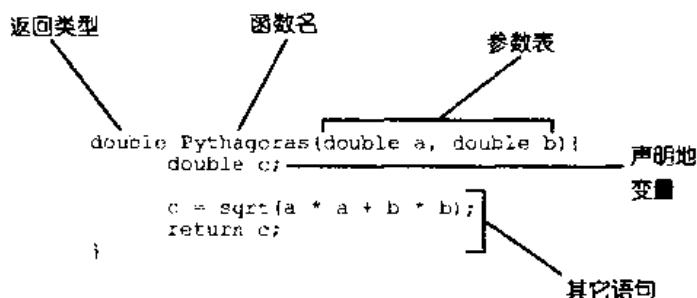
在这个例子中, 由于程序要使用 I/O 函数(`printf` 和 `scanf`)以及数学函数(`sqrt`), 所以必须包含两个头文件(`stdio.h` 以及 `math.h`)。 `sqrt` 计算一个数的平方根。这两个头文件提供这些函数的原型。

图 2-6 分析了 `Pythagoras` 函数定义的语法。返回值是 `double` 型的, 表明此函数要得到一个双精度浮点数。

void 型函数

如果函数不返回任何值, 它的返回类型为 `void` 型。与其它返回类型不同, `void` 型函数不需要 `return` 语句。函数可以使用 `return` 语句提前退出, 也可以让函数在函数定义的结尾自然终止。更多的信息, 请参阅第十三章。

图 2-6
函数定义语
法的分析



下面的例子调用了一个名为 `print_vars` 的函数。

```
#include <stdio.h>

void print_vars(int i1,int i2,int i3);

void main(){
    int a,b,c;

    a=b=c=1;
    print_vars(a,b,c);
    a=b=c=2;
    print_vars(a,b,c);
}

void print_vars(int i1,int i2,int i3){
    print("The value of param1 is %d \n",i1);
    print("The value of param2 is %d \n",i2);
    print("The value of param3 is %d \n",i3);
}
```

局部变量、全局变量以及其它变量

变量的属性之一是它的作用域,而变量的作用域又决定了变量在程序中的使用位置。C++ 中有四种作用域,都是从 C 中继承来的。它们是:局部变量(local)、全局变量(global)、静态变量(static)和外部变量(external)。除了这些作用域,C++ 又加入了类作用域,这使得可以将变量的作用范围限制在类的对象中。

局部变量

局部变量对函数定义来讲是一个私有成员。每个函数都可以有自己的变量,如 `i`,并且可以对这个变量进行修改而不影响其它函数内的名为 `i` 的变量。

声明一个局部变量,只需要将变量的声明放在函数的定义内。例如,变量 `c` 在下

面的函数定义 `Pythagoras` 中是局部变量。对该变量的修改不影响任何其它函数中 `c` 的值。

```
#include <math.h>

double Pythagoras(double a, double b) {
    double c;

    c = sqrt(a * a + b * b);
    return c;
}
```

C++ 与其它语言的不同之一在于在 C++ 中, `main` 函数与其它函数一样是一个普通的函数。`main` 函数有两个特点:一是它是程序的入口;二是它不需要函数原型。`main` 函数有自己的局部变量。例如,在下面的程序中,变量 `a`、`b`、`c` 和 `h` 都是 `main` 的局部变量。

```
void main() {
    double a, b, c, h;

    a = b = c = 1;
    print_vars(a, b, c);
    h = Pythagoras(a, b);
    print_vars(a, b, c);
}
```

全局变量

通常都将变量声明为局部变量(因为这样就可以控制代码对变量的修改),但是有时又需要将变量的作用范围扩展到整个函数。全局变量的作用域及生存期可以扩展到整个源文件。全局变量使得函数之间可以通过共享信息来传递消息。

将变量声明为全局变量,只需要在所有函数定义之外对变量进行定义。

图 2-7 中的样例程序使用了三个全局变量。程序中的所有函数都可以访问这三个变量:`a`、`b` 和 `c`。

● 注意

从定义全局变量的地方开始一直到源文件结尾,全局变量都是可用的。通常在文件的开始处定义全局变量。

```

#include <stdio.h>
#include <math.h>

void Pythagoras(void);
void Setvars(void);

double a, b, c;

void main() {
    Setvars();
    Pythagoras();
    printf("\nThe hypotenuse is %f.", c);
}

void Pythagoras(void) {
    c = sqrt(a * a + b * b);
}

void Setvars(void) {
    printf("Enter value of Angle 1: ");
    scanf("%lf", &a);
    printf("Enter value of Angle 2: ");
    scanf("%lf", &b);
}

```

包含指令

全局变量的声明
(注意是在 main
函数之外对它们
进行声明)

图 2-7

全局变量的
例子

静态变量

静态变量综合了局部变量的作用域与全局变量扩展的生存期。如果想要使用函数的局部变量,而且在两次函数调用之间保持变量的值不变,这种情况下静态变量就十分有用。例如:

```

void print_vars(int i1, int i2, int i3) {
    static int count = 0;

    printf("The value of param1 is %d \n", i1);
    printf("The value of param2 is %d \n", i2);
    printf("The value of param3 is %d \n", i3);

    count = count + 1;
    printf("I've been called %d time(s). \n \n", count);
}

```

变量 `count` 仅在第一次调用此函数时被初始化为零。(相比较来说,一般的局部变量在每次函数调用时都进行初始化。)每次调用 `print_vars` 之后, `count` 的值都增加 1。同时变量 `count` 是此函数的私有成员,不受其它函数中 `count` 变量的影响。

外部变量

随着程序大小的增加,一般要将源代码分成多个模块,分别编译连接。一个模块相当于一个 C++ 源文件。

除非将函数声明为静态的(static)(在函数原型以及函数定义的标题头前加入关键字 static),否则默认情况下,每个函数对整个项目中的所有其它模块都是可见的。在将函数声明为静态之后,此函数就仅仅在此模块中是可见的。

除非将变量声明为外部的,变量仅仅在对它们进行声明的模块中是可见的。将变量声明为外部的,首先需要在某个模块中将其声明为全局变量:

```
int global_count;
```

所有使用此变量的模块需要包含外部(extern)声明:

```
extern int global_count;
```

上面的声明告诉编译器,“将变量 global_count 看作外部变量。它的定义可能在本模块中,也可能在其它模块中。”

使用外部变量的另外一种方法是将所有的 extern 声明放在一个头文件中,然后在每个源文件中包含这个头文件。例如:

```
//-----  
//MYPROG.H- Extern declarations and function  
//prototype for myprog.  
  
extern int global_count;  
extern int current_checkno;  
extern double accumulator;  
//...
```

上面这些变量的每一个都必须在某个(只能在一个)源文件模块中定义。例如,模块 A 定义了头两个变量:

```
//-----  
//A.CPP  
  
#include "myprog.h"  
  
int global_count;
```

```
int current_checkno;  
//...
```

模块 B 定义了第三个外部变量:

```
//-----  
//B. CPP  
  
#include "myprog. h"  
  
double current_accumulator;  
//...
```

不管在哪里进行的定义,由于在头文件中进行了 extern 声明,程序中所有的函数都可以使用这三个变量。extern 声明并不创建数据;所有必须使用标准的变量声明来创建数据的方式也称之为定义。

有趣的控制结构

控制结构可以用清晰的方式描述判断以及循环。控制结构的这种能力将程序员从早期的 BASIC 以及汇编语言中使用的繁琐的编程方式中解脱出来。

这一部分介绍最常用的两种控制结构:if 和 while。C++ 也支持 do、for 及 switch 等结构。在第十三章中将对它们进行说明。

if 语句

C++ 中的 if 语句的语法如下:

```
if(expression)  
    statement  
[else  
    statement]
```

其中,中括号表明 else 语句是可有可无的。if 语句可以没有 else 语句,如下面的例子所示:

```
if(age<21)  
    printf("what do you think you're doing? \n");
```

if 语句也可以包含 else 语句,如下面的例子所示:

```
if(age<21)
    printf("what do you think you're doing? \n");
else
    printf("Eat, drink, and be computer literate. \n");
```

理论上讲,在 C++ 中没有 Visual Basic 中的关键字“elseif”。但是,else 语句后面的语句可以又是一个 if 语句。这种嵌套的例子如下:

```
if(age<21)
    printf("what do you think you're doing? \n");
else
    if(age == 21)
        printf("Okay, just one drink. \n");
    else
        printf("what do you think you're doing? \n");
```

因为 C++ 忽略空格,所有上面的例子也可以这样编写:

```
if(age<21)
    printf("what do you think you're doing? \n");
else if(age == 21)
    printf("Okay, just one drink. \n");
else
    printf("Eat, drink, and be computer literate. \n");
```

象 if 语句这样的控制结构通常与复合语句一起使用。一个复合语句可以由位于大括号({})之间的多个语句组成。在单个语句可以使用的地方都可以使用复合语句。

```
if(age<21){
    printf("Hey! What do you think you're doing? \n");
    printf("Serve minors? Just what kind of place ");
    printf("do you \n think we run? \n");
}
else if(age == 21)
    printf("Okay, just one drink. \n");
else {
    printf("Eat, drink, and be computer literate. \n");
    printf("But we suggest that you be careful \n");
    printf("driving home on the information super- \n");
    printf("highway tonight. ...");
}
```

● 注意

进行相等判断时,应该使用双等号(==);

while 语句

while 语句是经常与复合语句混合使用的另外一个控制语句。while 语句的使用语法如下:

```
while(expression)  
    statement
```

与 if 语句一样,while 语句中的条件表达式(*expression*)可以是任何有效的整型表达式。所有非零值都为真。比较运算符(<、>、==、<=、>=以及!=)返回真(1)或假(0);

例如,下面的程序进行从 5 到 1 的递减运算。while 语句重复运行这些语句直到条件 $n > 0$ 为假。

```
int n = 5;  
  
while(n > 0) {  
    printf("%d \n", n);  
    n = n - 1;  
}
```

下面的例子实现同样的功能,但是略微简洁。

```
int n = 5;  
  
while(n) {  
    printf("%d \n", n);  
    n = n - 1;  
}
```

下一部分将介绍如何使这个例子更简洁。

加强对 C++ 运算符的印象

C++ 有许多其它语言(除了 C 以外)所没有的有趣的运算符。这些运算符包括递增运算符和递减运算符、赋值运算符以及位运算符。

递增运算符和递减运算符

在 C++ 众多有趣的运算符当中,最常用的应该是递增运算符和递减运算符。这两个运算符可以从变量中简单地加 1 或减 1。例如:

```
n++;    //将 n 加 1。
n--;    //将 n 减 1。
```

可以在大的表达式中使用这些运算符。例如,递减运算符可以使上一部分中的从 5 倒数到零的例子更为简洁:

```
#include <stdio.h>
//...
int n=5;
while(n)
    printf("%d\n",n--);
```

这段程序使用递减运算符的后缀用法(`n--`),在获得变量的值之后才进行减法运算。在上面的例子中,递减运算符在程序输出 `n` 的值之后将 `n` 的值减 1。这样例子程序就会象预期的那样输出 5、4、3、2、1。

如果某个语句只使用递减运算符,那么前缀用法与后缀用法之间就没有什么不同。例如,下面的两个语句实现同样的功能:

```
n--;
--n;
```

但是在很多情况下前缀用法与后缀用法之间的区别是十分重要的。下面是递增运算符两种用法的总结:

运算符	功能
<code>++n</code> (前缀用法)	<code>n</code> 加 1,随后如果有更大的表达式,则将此时 <code>n</code> 的值传给更大的表达式。
<code>n++</code> (后缀用法)	先传递 <code>n</code> 的值,之后将 <code>n</code> 加 1。

对使用者来说,如果没有习惯使用后缀用法的话,它可能是最古怪用法。注意下面语句中使用后缀前与使用后缀之后的变化:

```
int count=3;
```

```
printf("%d\n",count);           //输出 3。
printf("%d\n",count++);         //输出 3,然后递增。
printf("%d\n",count);           //输出 4。
```

在前缀用法中,变化更为明显:

```
int count=3;

printf("%d\n",count);           //输出 3。
printf("%d\n",++count);         //输出 4。
printf("%d\n",count);           //输出 4。
```

递减运算符的用法与之类似。

运算符	功能
--n(前缀用法)	将 n 减 1,随后如果有更大的表达式,则将此时 n 的值传给更大的表达式。
n--(后缀用法)	先传递 n 的值,之后将 n 减 1。

赋值运算符

C++ 的赋值运算符不仅包括标准的赋值运算符(=),而且有许多与其它运算符混合使用的赋值运算符。因为这些运算符通过进行某个操作来改变变量的值,所以它们与递增运算符和递减运算符类似。

例如,加法赋值进行加法和赋值两个操作。如:

```
n+=10;
```

等价于:

```
n=n+10;
```

可以将递增运算符和递减运算符看作是加法赋值运算符和减法赋值运算符的特例。例如,下面的两个表达式功能相同:

```
(--n);    //从 n 中减一,随后返回结果。
(n-=1);   //从 n 中减一,随后返回结果。
```

C++ 支持许多按相同方式工作的赋值运算符:先执行某个操作随后将结果赋值给左边的变量。这些操作符有 *=(乘法赋值)、/=(除法赋值)等等。关于这方面的内容,请参看第十一章。

位运算符、逻辑运算符以及移位运算符

对于刚开始学习 C 和 C++ 的程序员来说,位运算符是 C 和 C++ 语言最有意思的部分。这些运算符允许对单个位进行操作。

表 2-3 总结了位检测运算符与逻辑运算符。除了不能对单个位进行检测外,逻辑运算符与位运算符没有什么不同。逻辑运算符对所有非零操作数一视同仁,因此忽略单个位的值。

表 2-3 逻辑运算符与位运算符

运算符	说明
&	位与。如果两个操作数相应的位为 1,将结果中相应的位为 1。
	位或。如果两个操作数相应的位有一个为 1,将结果中相应的位为 1。
~	位非(补码)。如果单个的操作数中某一位为 1,将结果中相应的位为 0,反之亦然。
&&	逻辑与。如果两个操作数都非零,结果为真(1)。否则结果为假(0)。获得正确的布尔值。
	逻辑或。两个操作数中,如果有一个非零,结果为真(1),否则结果为假(0)。获得正确的布尔值。
!	逻辑非。如果操作数为零,则结果为真(1);如果操作数非零,则结果为假(0)。获得相反的布尔值。

C++ 也支持左位移和右位移运算符。如表 2-4 所示。

表 2-4 右位移和左位移运算符

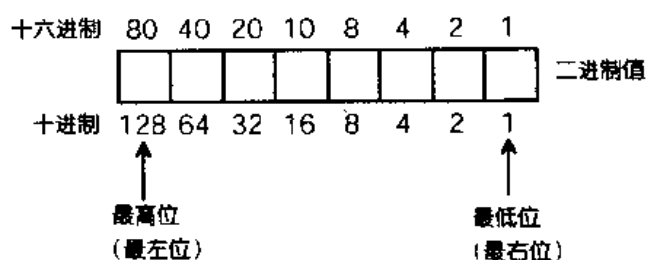
运算符	说明
val>>n	将 val 右移 n 位并返回结果。
val<<n	将 val 左移 n 位并返回结果。

无论操作数是如何表示的,所有的位运算符和移位运算符都使用操作数的二进制形式进行运算。二进制形式由一组 1 和 0 组成(图 2-8)。

例如,数 18 存储为 00010010。数字 6 存储为 00000110。使用二进制的与 (AND)可以对每一位进行检测,当两个操作数相应的位都为 1 时将运算结果相应的相应位为 1。18 与 6 的与的结果为 00000010,即 2。

下面的程序使用左移位操作符和位与输出整数的二进制形式。注意 0x8000 中的“0x”在 C++ 中是十六进制的记数法。0x8000 在最左端有一个 1,其它位均为零。

图 2-8
二进制表示
形式



```
#include <stdio.h>

void print_binary(short input_field);

void main() {
    short n;

    do {
        printf("\nEnter a short integer ");
        printf("(0 to quit):");
        scanf("%hd", &n);
        print_binary(n);
    } while(n);
}

void print_binary(short input_field) {
    int i = 1, bit_set;

    while(i <= 16) {
        bit_set = ((0x8000 & input_field) > 0);
        printf("%d", bit_set);
        input_field = input_field << 1;
        i++;
    }
}
```

上面的程序利用了 C++ 布尔条件语句来得到 1 或 0(分别为真或假)。在 while 循环中,将参数与 0x8000 进行与操作。结果或者为 0x8000 或者为 0。然后所得结果再与 0 进行比较,比较表达式的值或者为 1 或者为 0,然后输出:

```
(0x8000 & input_field) > 0
```

C++ 支持左移位赋值运算符(<<=)。左移位赋值运算符具有左移位和赋值功能。在循环底部对 input_field 使用此运算符,可以使上面的程序更为简洁。

位段:十分简洁的数据结构

从单个位中存储或者获得信息的另外一种方法是位段。位段是 C 和 C++ 所独有的特征。位段与位运算符和移位运算符的功能相同,但是有时位段更易于使用。

使用位段的数据结构,需要如下使用 `unsigned` 来声明(中括号的意思是可选项):

```
struct [struct_name] {  
    unsigned field1:width1;  
    unsigned field2:width2;  
    ...  
    unsigned fieldN:widthN;  
}[struct_vars];
```

例如,下面的程序声明了一个类型为 `card` 的结构。`card` 中有两个值,一个是 `rank`(1~13),另一个是 `suit`(1~4)。

```
struct card {  
  
    unsigned int rank:4;  
    unsigned int suit:2;  
};
```

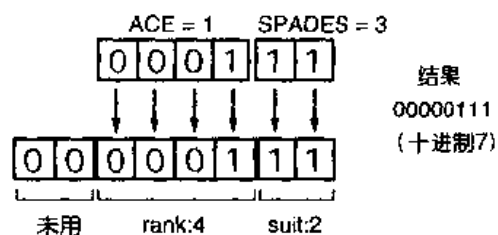
那么为什么 `rank` 与 `suit` 的宽度分别为 4 和 2 呢? 这个问题很重要。答案是 4 位存储的值小于等于 16,而 2 位存储的值小于等于 4。这样这两个字段就分别可以存储各自的最大数值 13 和 4。对于宽度为 `n` 的位段,它所能存储的最大值是 2 的 `n` 次方。

现在可以声明 `card` 结构并给这些字段赋值了。下面的例子假定 `club`、`diamond`、`heart` 和 `spade` 分别代表 0、1、2、3:

```
card cd1, cd2, cd3, deck[52];  
  
cd1.rank = 1;           //假定 ace = 1。  
cd1.suit = 3;           //赋值:spades = 3。
```

图 2-9 说明了如何在低 8 位存储这些值的。这个图表表明可以使用 1 个字节来存储 `rank` 与 `suit`,无须 2 个字节或更多字节。

图 2-9
位段赋值



好了,总结一下。本章介绍了 C /C++ 的语法和运算符。如果对你来说这些东西都很陌生,不要泄气。需要的时候可以重新看看这一章。如果你已经理解了本章的大部分内容,那么恭喜了。

第 三 章

指针、字符串及其它

将 C 和 C++ 从其它计算机语言中区分开的特征之一就是指针的使用。一开始指针可能很难或者有些古怪,但是其它语言实际一直都在使用指针。不同之处在于,在其它计算机语言中,指针的使用是不透明的——不让用户知道有指针这个东西。而 C 和 C++ 却允许对指针加以控制。

那么,什么是指针呢?随后将对这个问题给出详细的回答。不过简单说来,指针是用来存储地址的。地址是相对较小的数据单元(在 16 位系统中占据 2 字节),它存储另一个数据的位置。指针的好处在于,只要知道数据的位置,就可以访问任何大小的数据。

C++ 推荐使用 new 运算符动态创建对象,new 运算符返回一个指针。另外,某些运算符函数(在第七章中将会介绍)要求使用指针。在本章中,主要介绍参数、字符串以及数组指针的使用方法。

更为快捷的数据传递方法

简单地说,指针是存储地址的变量或参数。指针给出另一个变量的位置,函数可以使用这个位置控制任何大小的一段数据。

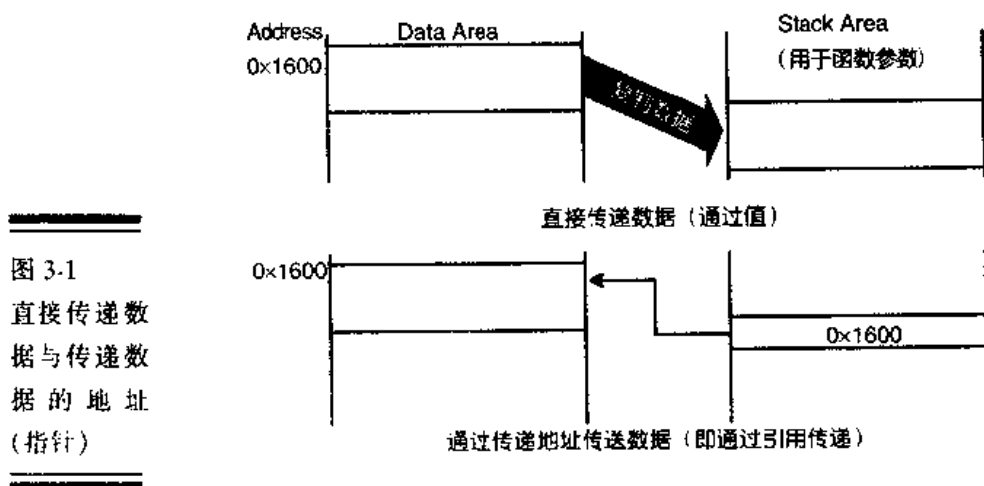
假定目前有两个不同的函数。其中一个函数有大段的数据需要与另外一个函数共享。例如,将一长串字符传递给另外一个函数用于在屏幕上显示。

除非需要传递的数据是全局类型的(要知道,具有许多全局变量的程序不是一个优秀的程序),否则就要进行数据的传递,包括两种方式:

- 通过复制整个数据结构来进行传递。在数据很少的时候,这种方法还可以。但是在处理数组、字符串或者其它大的对象时,这种方法的效率就太低了。
- 通过传递数据结构的地址进行数据的传递。地址看起来象一个整数,但是对

计算机来说,地址有特殊的含义。地址告诉处理器在内存的什么地方可以找到数据。

可以画出计算机的堆栈来进一步说明。堆栈是存储函数间传递的数据的地方。直接传递数据结构将会把数据完全复制到堆栈中。但是传递地址则是把相对小一些的数据放在堆栈中(一般来说,是 2 字节或 4 字节,这取决于计算机地址的大小)。在调用函数时,只需要知道数据结构的存储位置——地址,就可以访问整个数据。图 3-1 说明了这两种数据传递方法。



第二种方法只需要知道数据的地址。这样就可以访问或修改传递来的数据。通过地址对数据进行操作称为间接访问。现代处理器的设计支持大量的间接访问。

指针的好处不止在于节约了内存。在一些情况下,指针允许进行引用传递,这也是下一部分要介绍的内容。

指针与通过引用进行数据传递

如果你以前使用过 BASIC、FORTRAN 或 Pascal 等编程语言,可能对通过引用进行传递以及通过值进行传递十分熟悉:函数可以永久地改变传递过来的参数的值。

对 BASIC 及 FORTRAN 这些语言来说,通过引用进行传递十分方便并且可以很容易地使用关键字进行控制(如在 Pascal 中的 var)。但是在通过引用传递时,这些语言传递的实际是一个指针,你可能并没有意识到这一点。这些语言隐藏了这个事实,这样也就无须学习指针的使用。(Pascal 有指针的用法,但是仅限于内存的分配。)

● 注意

C++ 允许通过使用地址运算符(&)对引用的参数进行声明而,C 没有这个特点。在第六章将介绍这种技术。首先最好明白如何利用指针进行引用。在 C++ 中,迟早要使用指针,而学习指针的最好的方法是从通过引用进行传递开始学起。

如果再看一看图 3-1,指针与通过引用进行传递之间的关系就会十分清楚。通过值进行传递,得到的是原始数据的复制品。可以对这个副本进行修改,但是所作的修改不影响原始数据。在操作完成之后,所使用的堆栈完全释放。对副本作的修改对程序没有丝毫影响。

但是,如果获得的是数据的指针,那么对数据进行的修改将影响原始数据。指针并非是一个新的数据,它只是告知数据的位置,这样就可以对原始数据进行修改。这就像是传递了包含原始数据记录的文件位置及文件的组成结构,而不是传递了文件的一个副本。

因此,传递一个指针(地址)与通过引用进行传递实现的是同一个功能。从字面上来讲,通过引用进行传递意思是:指向原始数据副本的指针。

通过引用进行传递的步骤

使用指针引用的步骤如下:

1. 在函数原型以及函数定义的参数列表中,使用指针间接访问运算符(*)对参数进行声明。
2. 在传递参数时,要记住传递的是一个地址。一般要对参数使用地址运算符(&)。
3. 在函数定义内部需要访问指针指向的数据时,要使用指针间接访问运算符(*)。

下面对每一步都进行详细的说明。

首先,要使用指针类型的参数,必须使用间接访问运算符(*)对参数进行声明。除了在参数左边加入一个间接访问运算符(*)以外,声明方式与非指针参数的声明相同。例如,下面的函数原型声明了具有一个参数的函数。参数是一个 int 指针。

```
void double_it(int *n);
```

第二,必须传递地址参数。传递原始类型数据的地址,需要使用地址运算符(&)。例如:

```
int amount;  
//...  
double_it(&amount);    //传递 amount 的地址。
```

最后,在函数定义的内部,使用间接访问运算符(又一次使用此运算符)对指针类型的数据进行访问。例如:

```
*p = *p * 2;
```

使用乘法赋值运算符,上面的语句也可以写成:

```
*p *= 2;
```

这两个运算符(*和&)的作用是相反的。指针间接访问运算符(*)的意思是“指针指向的对象”而地址运算符的意思是“得到地址”。第一个运算符*从地址中获得数据的内容(例如,将int型的指针转换为真正的int型数据);第二个运算符(&)则获得该int型数据的地址。

● 注意

如果没有使用间接运算符(*)而直接使用p将影响指针本身,指针就不可能指向所要指向的地方。例如,(*p)++将指针所指向的数据增加一,而p++则是将指针指向了下一个地址。编译器不会找出这种错误,所以在使用指针时要十分小心;修改指针所指向的数据时要使用间接访问运算符(*),要修改指针所指向的地址时,只需要修改指针本身。

两个通过引用进行传递的完整的例子

正如上面所说的那样,指针间接访问运算符(*)与地址运算符(&)对于指针的使用都是必须的。要通过引用进行传递,必须首先在堆栈中放一个地址(这就用到&);随后,在函数定义中,要使用这个地址对地址所指向的数据进行访问(这就用到了*)。

图3-2显示了上面的例子完整实现。这个例子首先定义了一个变量amount并将它的值设置为5。随后调用了函数double_it并给函数传递了amount的地址。此时amount的值为0。如果函数按值的方式给函数double_it传递参数,函数double_it就不会作任何事情。

图 3-3 是通过引用传递两个参数的例子。(有时可能需要象例子中的参数列表一样,将指针和简单的数据类型混合在一起。)在这个例子中,因为传递的是 a 和 b 的地址,函数可以直接对 a 和 b 而不是对 a 和 b 的副本进行访问,所以函数可以修改这两个参数的值。

图 3-2

将 amount
的地址传递
给函数 dou-
ble_it

```
#include <stdio.h>

void double_int(int *p);

main () {
    int amount = 5;

    printf("The value of amount is %d.\n", amount);
    double_int(&amount);
    printf("The value of amount is %d.\n", amount);
}

void double_int(int *p) {
    *p = *p * 2;
}
```

传递 amount
的地址作为参
数值

图 3-3

传递两个变
量 a 和 b 的
地址

```
#include <stdio.h>

void swap(double *x, double *y);

main () {
    double a = 1.5, b = 3.9;

    printf("a = %lf, b = %lf\n", a, b);
    swap(&a, &b);
    printf("a = %lf, b = %lf\n", a, b);
}

void swap(double *x, double *y) {
    double temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```

将 a 和 b 的地址
传递给 *x 和 *y

指针与数组

在 C++ 中,指针与数组紧密相连。使用指针可以大大提高对数组进行处理的效率。

数组是几乎每个计算机语言都使用的重要的语言成分。下面的两个原因可以解释为什么在重要的项目中必须使用数组：

- 数组可以方便地为大块数据分配内存。
- 只需要简单的几行代码,就可以使用数组进行大量的操作。将数组与循环结合使用,可以执行一系列的语句并且可以按任意次数重复进行这些操作。

不能低估数组与循环处理的重要性。没有这种编程方法,计算机程序以及计算机本身的能力将大大降低。通过使用指针,C++增强了计算机的运算能力。

数组的基本知识

下面是 C++ 中声明数组的语法：

```
type name[length];
```

可以使用下面的语法定义多维数组。其中省略号(...)表明[length]可以重复任意次。

```
type name[length][length]...[length];
```

在上面这两种用法中,必须输入中括号。在多维数组的定义中,维数可以是 2、3 或 n,n 可以是任意大的数。但是不要忘了,没有任何原因而定义高维数的数组很可能浪费大量的内存。1000 乘 1000 大小的每个字符串长为 256 字节的字符串数组至少需要 250 兆字节。

第一个语句的其余部分说明了一维数组的用法,当然,多维数组的使用与一维数组的用法类似。数组定义中的 length 说明数组元素的个数。(例如,在多维数组中,数组元素的个数为 length1 * length2 * ... lengthN。)数组中最小的下标总是 0,最大的下标总是 length - 1。

下面的几个例子有助于理解这些概念。先对数组进行定义：

```
int a[5];
```

上面的代码定义了五个整数。这些变量分别为：

```
a[0];  
a[1];  
a[2];  
a[3];  
a[4];
```

此处最大的下标是 4,比数组的长度(5)小 1。可以如下定义一个 10 个元素的数组:

```
int b[10];
```

上面的代码定义了 10 个整数。最大的下标为 9,比数组的长度(10)小 1。

```
b[0];  
b[1];  
b[2];  
b[3];  
b[4];  
b[5];  
b[6];  
b[7];  
b[8];  
b[9];
```

● 技巧

强调数组的最大下标比数组的长度小 1 是很重要的。在使用数组时,一个常见的错误就是不正确地设置了循环的初始条件和终止条件。由于忘记了数组的最大下标比数组的长度小 1,甚至最有经验的程序员偶尔也会错误地设置了循环的终止条件。所以要认真检查循环的条件。

将数组定义 `int a[5]` 说成是定义了五个整数解释了数组定义的精确含义。图 3-4 说明了这五个整数在内存中是如何顺序地排列的。图中,假定每个整型数占用 2 个字节,尽管在许多系统中整型数占用的字节数可能更大。

图 3-4

具有五个整型数的数组



五个整型数数组的声明类似于单独声明五个整型数。可以如下定义五个整型数:

```
int a0;  
int a1;  
int a2;  
int a3;
```

```
int a4;
```

这种方法与声明一个具有五个整型数的数组 `a` 有什么不同呢？五个整型数的数组 `a` 的声明生成五个数组元素 `a[0]`、`a[1]`、`a[2]`、`a[3]`、`a[4]`。

最明显的地方在于数组的声明更为简洁。当只声明五个整数时，单独进行声明也许还不繁琐，如果要声明 1000 个整型数，使用数组声明就会省掉许多代码：

```
int a[1000];
```

还有一个原因是：可以使用变量或常量来引用数组中的元素（例如，下标可以是变量 `n`）。例如：

```
int a[5];
```

```
a[0] = 10;
```

```
a[1] = a[0] + 1;
```

```
int n = 2;
```

```
a[n] = a[1];    //将 a[1] 的值赋给 a[2]
```

对数组使用变量下标是十分有用的技巧。这种方法可以用于数组的循环处理，如下的例子所示：

```
//将具有 1000 个元素的数组的所有元素初始化为 100
```

```
int a[1000];
```

```
int i = 0;
```

```
while(i < 1000)
```

```
    a[i++] = 100;
```

其中递增运算符 (`++`) 在获得 `a[i]` 的值后将 `i` 的值加 1。此处使用的是后缀递增运算符。（后缀运算符先获得 `i` 的值并使用该值，之后将 `i` 的值增加 1）。为了说得更清楚些，上面的例子中的最后两行可以写做：

```
while(i < 1000){
```

```
    a[i] = 100;
```

```
    i++;
```

```
}
```

使用指针进行循环处理

指针是进行循环处理的另外一种方法。使用指针一般比使用数组的下标更为有

效。(尤其是在与多维数组进行比较时。)例如,可以将上面的例子重新编写如下:

```
//将具有 1000 个元素的数组的所有元素初始化为 100
```

```
int a[1000];
int i = 0;
int *p = a;
while(i < 1000)
    *p++ = 100;
```

上面的某些语句需要认真看一下。第三句定义了一个指针 *p* 并将 *p* 初始化为数组 *a* 的开始地址。对数组进行引用可以获得数组的开始地址。

```
int *p = a;
```

这个语句等价于略微长一些的,但是更清晰的一条语句:

```
int *p = &a[0];
```

因为数组的名称等价于数组第一个元素的地址,所以这两个语句的功能是相同的。

循环语句根据 C++ 的相关规则按照某一顺序完成某些事情。

```
while(i++ < 1000)
    *p++ = 100;
```

参看了表 11-1 或表 11-2 之后就会明白,指针运算符(或称为“间接访问”运算符)(*)与递增运算符(++)具有相同的执行顺序,所以它们的运算顺序为从右到左(大多数运算符的运算顺序是从右到左的),首先执行递增运算符。所以指针增加 1,而不是指针最初指向的值。使用括号可能更清楚一些:

```
while(i++ < 1000)
    * (p++) = 100;
```

因为使用的是后缀递增运算符,在进行表达式计算之后才进行增量运算。应该记住后缀递增运算符先获得操作数的值,之后才改变运算符的值。表达式 *(p++) 执行下面这些操作:

1. 获得指针 *p* 的值。
2. 将 *p* 的值增加 1。
3. 在 *p* 的值增加之前先访问 *p* 所指向的数据。

因此除了在执行后 p 的值增加 1 外,表达式 $*(p++)$ 与 $*p$ 具有相同的作用。

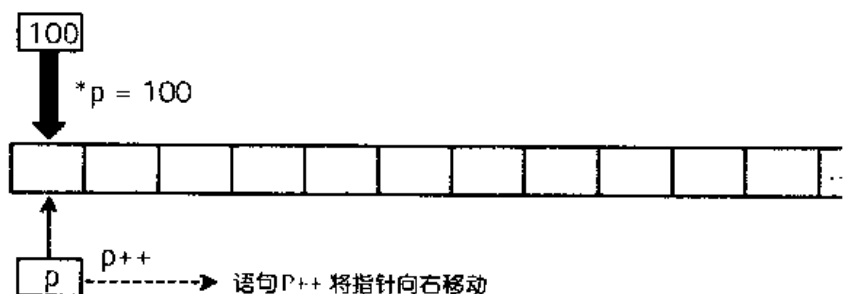
可以将上面的语句写的更清楚一些,不过略微冗长一些:

```
while(i++ < 1000){
    *p = 100;    //将 100 赋值给 p 所指向的元素
    p++;        //指向下一个元素
}
```

循环中的第一个语句($*p++$)是对 p 所指向的元素进行的操作。但是第二个语句($p++$)对指针 p 本身进行操作。无论数据的类型如何,将指针加 1 总是将指针指向下一个元素。C 和 C++ 使用指针算法时,按照数据的类型进行操作:将指针增加某个整数时,用数据类型的大小乘以这个整数。

图 3-5 说明了上面的例子中循环是如何将值赋给数组以及将指针指向下一个元素:

图 3-5
使用指针对
数组进行循
环处理



可以使用类似的语句将一个数组 b 的所有元素复制给另一个数组 a :

```
int a[1000], b[1000];
//...
int i = 0;
int *pa = a;
int *pb = b;
while(i++ < 1000)
    *pa++ = *pb++;
```

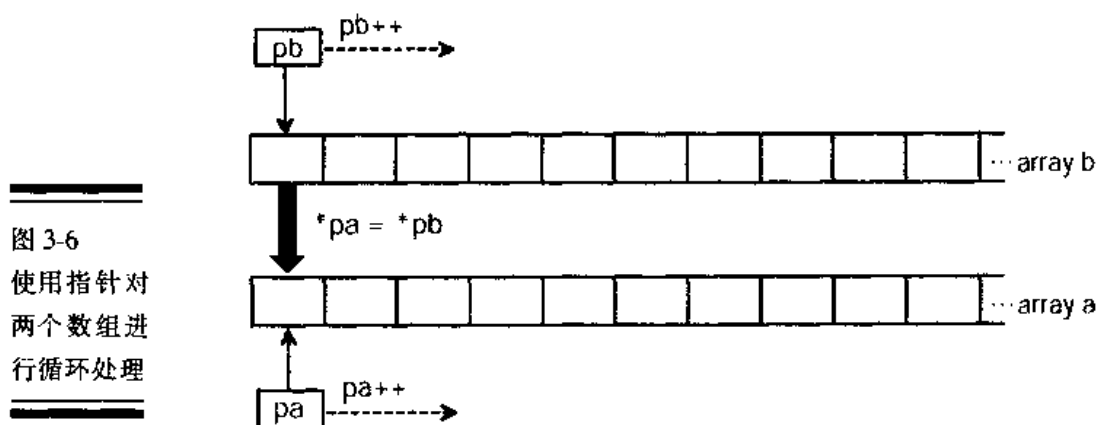
因为指针运算符($*$)与后缀递增运算符($++$)的执行顺序是自右至左的,所以这里又一次在指针运算符($*$)之前使用了后缀递增运算符($++$)。由于使用的是后缀递增运算符($++$),所以是在使用了 pa 和 pb 的值之后才分别将这两个变量的值加 1。因此上面的循环与下面的循环具有相同的功能:

```
while(i++ < 1000) {
```

```
*pa = *pb;           //从 b 中将元素复制给 a
pa++;                //指向 a 中的下一个元素
pb++;                //指向 b 中的下一个元素
```

在上面的代码中，“pa”代表“指向变量 a 的指针”，“pb”代表“指向变量 b 的指针”。应该给变量使用一个易于理解的名称。

图 3-6 说明了上面的例子中，循环是如何将值赋给数组以及将指针指向下一个元素：



C++ 字符串

数组的处理技巧，特别是涉及到指针的数组处理技巧在处理 C++ 字符串时十分有用。如果你对使用指针处理数组元素还不熟悉，最好再看看上面那一部分。

在 C++ 中，字符串是一个字符数组。理论上，一个字符就是一个整数，长度为一个字节。（但是字符串与一个字节之间没有必然的联系；宽字符的格式每个字符使用 2 个字节。）在屏幕上输出时，每个字符值都映射为一个可以输出的字符。最广泛使用的映射规则是 ASCII（美国标准信息交换码）代码转换规则。

简单地说，一个字符串是一个元素长度为 1 个字节的整数数组，每个整数代表一个 ASCII 字符代码。（附录 D 列出了前 127 个 ASCII 代码的表格）

● 注意

如果你习惯了 BASIC 中的字符串处理，你可能觉得 C++ 中的字符串处理有些繁琐。例如，将一个字符串的值赋给另一个字符串时必须使用一个库函数（str-

cpy)。在第 5~7 章中,使用了 C++ 面向对象的特征创建了一个字符串类,使用该对字符串处理与在 BASIC 对字符串的处理一样方便。这个类也说明了 C++ 面向对象的能力。但是要明白第 5~7 章中的程序,必须首先明白 C++ 字符串的处理方式。

无论在什么计算机语言中,字符串从来都不是简单的一些值。象 BASIC 这样的计算机语言只不过隐藏了数组的处理方式。C 和 C++ 没有隐藏数组的处理方式,从而给与用户更多的控制权。

除了是一个数组之外,C++ 字符串还有一个特别的特性。该特性是在进行字符串处理时必须牢记的:字符串以一个空操作字节(' \0')作为结束。这个空操作字节的值为 0(与可输出字符“0”区分开)。

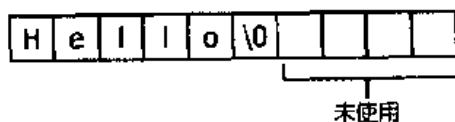
空操作终止符的位置决定了字符串的有效长度。对于刚刚开始学习 C/C++ 的人来说,这一点常常让人费解。可以为字符串分配任意多的字节,但是字符串的有效长度与第一个空操作符的位置有关。例如,如果向 printf 函数传递一个字符串,则只有第一个空操作符之前的字符被打印出来,空操作符之后的所有字符都被忽略。字符串处理的第二个规则是:字符串的数组维数决定了字符串的最大长度(因为有空操作符,所以字符串的最大长度比维数小 1)而不是字符串的有效长度。

下面的简单的例子可以清楚的说明这些规则。假定声明了如下的字符串:

```
char str[10] = "Hello";
```

str 声明为一个字符数组。因此字符串具有的最大长度为 9(10 减去 1 个空操作符)。声明时将字符串初始化为“Hello”,图 3-7 说明了字符串在内存中的表示形式。

图 3-7
具有未使用
字节的字符串



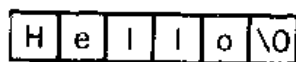
最后的四个字节没有使用。在将新的一串字符复制到这个字符串中时(使用 strcpy 函数,后面将会提到),这四个字节也是可以使用的。但是无论在何种情况,不能将长度大于 10(包括空操作符)的字符复制给此字符串。如果这样做,将会覆盖其它变量甚至其它程序所使用的数据区,这样做会导致无法预见的后果。这也是 C++ 与其它语言如 BASIC 的不同之处。必须留意字符串的最大长度。

创建字符串的另外一种方法是省略数组维数的说明。这种情况下,C++ 为初始化字符串分配相应大小的内存。如果以后不改变字符串的长度,这种创建方法也不错。

```
char str[] = "Hello";
```

图 3-8 说明了字符串在内存的形式。

图 3-8
没有未用字
节的字符串



在创建 C++ 字符串时,必须对字符串进行初始化,如最后的两个例子所示。等号(=)的使用是初始化但没有赋值;这仅限于使用同一个语句对字符串进行声明和定义的情况。

在声明了字符串之后,必须使用 strcpy 函数(或者自己编写一个 strcpy 函数,这可能方便一些)进行字符串的赋值。不能象数字或简单的对象的赋值那样将一个字符串赋给另外一个字符串。这可以称之为 C++ 字符串处理的第三个规则:因为字符串是数组,所以字符串之间不能直接进行赋值操作。必须调用函数进行赋值或者使用一次复制一个字符的循环。

这也是 C++ 中的字符串处理看起来比其它语言要繁琐的原因。不能象 BASIC 中那样使用 A\$ = B\$ 直接进行字符串的赋值。下面的代码将出现错误:

```
char string1[10] = "One";
char string2[10] = "Two";

string2 = string1;    //错! 不能给字符串直接赋值
```

编译器可能给出一句无用的、无法理解的信息,如“Cannot assign to constant.”只有想到 C++ 的数组名称,事实上是常量(字符串的名称等价于第一个字节的地址)时,才能理解这句信息的意思。

要将一个字符串的内容复制到另一个字符串中,需要使用 strcpy 函数:

```
#include <string.h>

char string1[10] = "One";
char string2[10] = "Two";
strcpy(string2, string1);    //正确地将 string1 复制给 string2
```

#include 命令支持 strcpy 函数。使用 strcpy 函数需要 string.h 中对此函数的声

明。在使用 C++ 标准库中的字符串处理函数时必须包含这个头文件。表 3-1 列出了一些其它的字符串处理函数。

表 3-1 其它的 C++ 字符串处理函数

函数	说明
<code>strlen(char * s)</code>	返回第一个空操作符之前的字符数, 不包括空操作符
<code>strcpy(char * dest, char * src, int n)</code>	复制 n 个字符
<code>strcat(char * dest, char * src)</code>	将 src 的内容加到 dest 的后面

下面的代码使用标准库中的 `strcpy` 以及 `strcat` 函数:

```
#include <stdio.h>
#include <string.h>
...
char * name[81];          //最大长度为 80 个字符, 因为有空操作符, 所以为 81

strcpy(name, "Archie");
strcat(name, " ");
strcat(name, "Leach");
printf("%s, he my friend. \n", name);
```

格式字符 `%s` 指定相应的参数是字符串的地址。记住, 数组名称(此时为一个字符数组)代表数组的开始地址。上面的代码输出:

```
Archie Leach, he my friend.
```

指针在编写字符串处理函数时指针提供了一些方便。例如, 用户可以编写自己的 `strcpy` 函数。在实际应用中, 因为标准库中提供了 `strcpy` 函数, 所以没有必要编写此函数。但是, 在 C++ 中编写这样的函数十分简单, 了解这一点是有好处的。

```
char * strcpy(char * dest, char * src){
    while( * src != '\0'){
        * dest = * src;
        dest++;
        src++;
    }
    * dest = '\0'; //添加终止空操作符
    return dest;
}
```

当然, 有多种方法可以使上面的程序更简洁, 但是上面的程序是最容易理解的。

(在实际应用中,可以使用前面提到的一些技巧将例子的大小减少几行。)图 3-9 说明了该函数的运行过程。

指针与动态内存分配

“动态”是编程世界中最广泛使用(甚至令人费解)的几个术语之一。一般来说,动态意味着“变化”。

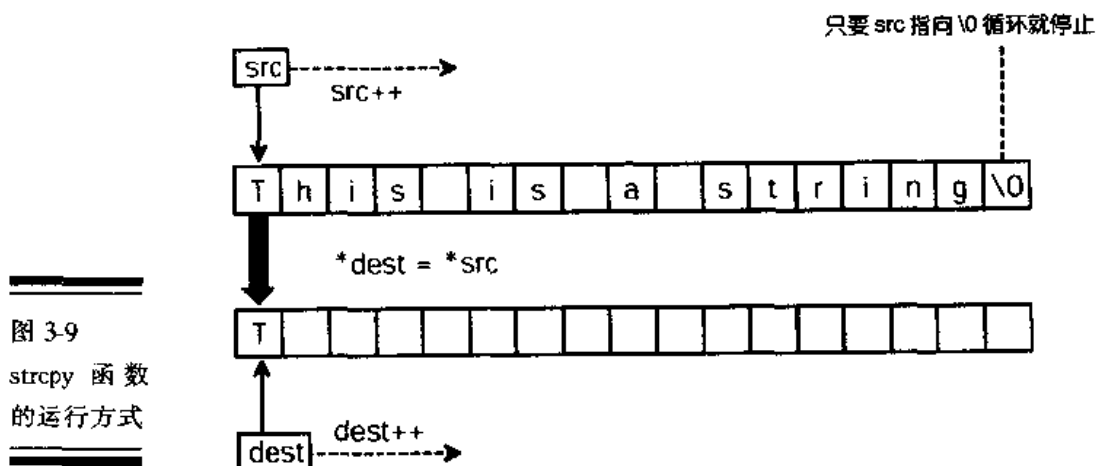


图 3-9
strcpy 函数
的运行方式

在程序运行时,如果需要修改内存就需要动态内存分配。例如,如果要将一个文件的内容读到内存中,所需要的内存的大小取决于文件的大小。如果要构造一个结构列表,每次添加一个新结构时就需要分配新的内存。

C++ 提供了两种方法进行内存的动态分配。可以使用从 C 标准库中继承来的一组 `malloc` 函数,也可以使用 `new` 和 `delete` 运算符。这两个运算符是 C++ 特有的。

使用任何一种方法都必须理解和使用指针。与普通的数据结构一样,编译器将地址分配给变量或对象。你无须知道这些地址;只需要使用变量的名称即可。

使用 `malloc` 和 `free`(C 与 C++ 均支持)

调用 `malloc` 函数是向系统请求指定大小的内存的一种方法。如果条件允许,操作系统就会进行内存的分配。返回值是一个指向内存块的指针。该内存块由内存中一些连续的字节构成。

`malloc` 及 `free` 库函数的使用需要以下几步:

1. 包含 malloc.h 头文件:

```
#include <malloc.h>
```

2. 在声明了相应类型的指针之后,调用 malloc 函数并指定所需大小的字节数作为参数。使用指针类型转换,将类型转换为合适的指针类型,然后将结果赋给指针:

```
ptr = (type *) malloc(elements * sizeof(type));
```

3. 在结束内存操作后,调用 free 函数并将该指针作为参数:

```
free(ptr);
```

按照编程术语,动态内存分配与普通变量的使用有一点明显的不同:使用动态内存不能直接访问变量。只能通过指针进行访问。幸好通过指针可以很方便地访问内存块的任何部分。在 C++ 中(以及在 C 中),指针与数组名称几乎是可以互换使用的。一旦系统分配了内存块,操作系统返回一个指针,在该指针包含了内存块的开始地址。失去指针(或者没有存储就改变了指针的值)将会丢失与之相连的内存。

例如,下面的代码分配可以存储 500 个整数的内存块。在 malloc 成功返回之后,通过对指针使用数组的下标,可以象引用数组的成员一样对这些整数进行引用。

```
#include <malloc.h>
#include <stdio.h>
//...
int *p;

p = (int *) malloc(500 * sizeof(int));

if(p) {
    p[0] = 5;           //将 5 赋给第一个元素
    p[50] = -33;        //将 -33 赋给第 51 个元素
    printf("The 51st element is %d.\n", p[50]);
    //...
    free(p);
}
```

malloc 函数使用一个参数——所需的字节大小。因为需要存储 500 个整型数的内存,相应的大小是每个整型数大小的 500 倍。sizeof 运算符(看起来象一个函数)是 C 和 C++ 中特有的内嵌运算符,返回类型的大小。

```
p = (int *) malloc(500 * sizeof(int));
```

这个语句也使用了一个数据转换符(int *)用来改变返回值的类型。malloc 函数

返回一个 void 类型的指针。void 是一个通用指针类型。在 C++ 中,在对 sizeof 返回的数据赋值之前必须改变此数据的类型。数据转换符将 void * 类型转换为 int * 类型,这样返回的数据才与指针 p 的类型相匹配。

```
(int *)expression
```

与其它的例子类似,下面的代码分配 100 个可以存储双精度浮点数(double)的内存:

```
double *p;  
p=(double *)malloc(100*sizeof(double));
```

● C /C++

数据转换的需要是 C++ 在 C 的基础上添加的限制。在 C 中,可以在不同类型的指针之间进行赋值。而在 C++ 中,在将 void 类型的指针赋给任何其它类型的指针之前必须进行类型转换。

在 malloc 返回之后,最好还是检测一下返回的结果。如果函数成功,指针包含一个有效的地址。否则,将一个空值赋给指针,也就是说指针的值为零。if 语句将空值解释为假(false)。如果内存不够,malloc 函数失败并将指针的值设置为零(false)。因此非零值表示函数成功。

```
if(p){  
    p[0] = 5;        // 将 5 赋给第 1 个元素  
    p[50] = -33;     // 将 -33 赋给第 51 个元素  
    printf("The 51st element is %d \n",p[50]);  
    //...  
    free(p);  
}
```

最后,通过调用 free 函数,代码将内存块完全释放,这是动态内存与常规变量之间另一个主要的不同点。如果你还不习惯释放动态内存块,那么代码将用完所有内存,从而导致内存泄露,程序及操作系统崩溃。

使用 new 和 delete(C++ 特有)

任何使用 malloc 和 free 函数的地方都可以使用 new 和 delete 运算符。使用 new 和 delete 具有以下优点:

- 使用 new 和 delete 不必另外包含头文件。
- 在对指针进行赋值之前无须转换类型,new 运算符将自动返回正确的类型值。

- 最重要的是, `new` 和 `delete` 不仅仅分配内存块, 当使用 `new` 来分配对象时, 它将自动调用该对象的构造函数; 与之相似, 使用 `delete` 释放内存时, 将自动调用析构函数。本书第五章介绍了构造函数与析构函数的概念。

使用 `new` 分配内存时要将对象类型置于 `new` 之后, 也可以通过使用下述第二个用法分配一组对象, 这时必须输入中括号。

```
pointer = new type;  
pointer = new type[ n ];
```

这样, 你就能分配 1 个整型数或者是一个包含 500 个整型数的数组。元素数量 `n` 可以是变量, 该变量包含一个运行时可以确定的值。

```
int * p1, * p2, * p3;  
int num = 75;  
//...  
p1 = new int;           // p1 指向一个整型数  
p2 = new int[500];      // p2 指向 500 个整型数中的第一个数  
p3 = new int[num];      // 分配 75 个整型数
```

要释放由 `new` 分配的内存就要使用 `delete` 运算符, 如果 `new` 是以数组形式使用的, 就要在指针名前加上空的中括号 `[]`, 如下所示:

```
delete pointer;         // 如果 new 生成的是一个对象  
delete[] pointer;      // 如果 new 生成的是一个数组
```

前文提到的例子可以用 `new` 和 `delete` 写成如下形式:

```
#include <stdio.h>  
//...  
int * p = new int[500];  
  
if(p) {  
    p[0] = 5;           // 将 5 赋给第 1 个元素  
    p[50] = -33;        // 将 -33 赋给第 51 个元素  
    printf("The 51st element is %d. \n", p[50]);  
    //...  
    delete[] p;  
}
```

注意, 与 `malloc` 函数相似, 在内存不足时 `new` 将返回空值, 因此为了得到最好的结果, 在使用或删除新对象之前最好测试一下该值。总结: `delete` 只能用于删除由 `new` 分配的内存, `free` 只能用于删除由 `malloc` (或相关函数, 见第十五章) 分配的内存。

第 四 章

输入、输出和 C++

为了帮助你编写简单的程序,我们在第二章已经介绍了输入/输出(I/O)技术。但在 C++ 中还需要对输入、输出作更多的说明。

C++ 为输入和输出提供了一个面向对象的方法。C++ 函数库把流对象(stream objects) cin 和 cout 作为输入输出的接受和发送终端。这两个对象代表数据流动的方向,用户可以通过它们来发送或接收数据。这看起来或许有点抽象,但是这些对象是比较容易使用的。而且你还会发现,使用 C++ 中的流式输入输出方法有一些好处。

并不是所有的输入、输出都是和键盘和显示屏有关的。在这一章的后面部分中将会看到面向对象的方法能够象应用在控制台那样应用于文件-流目的文件。

流的概念

本章将经常使用“流”这一术语。它通常用在系统程序设计中。那么流到底是什么呢?

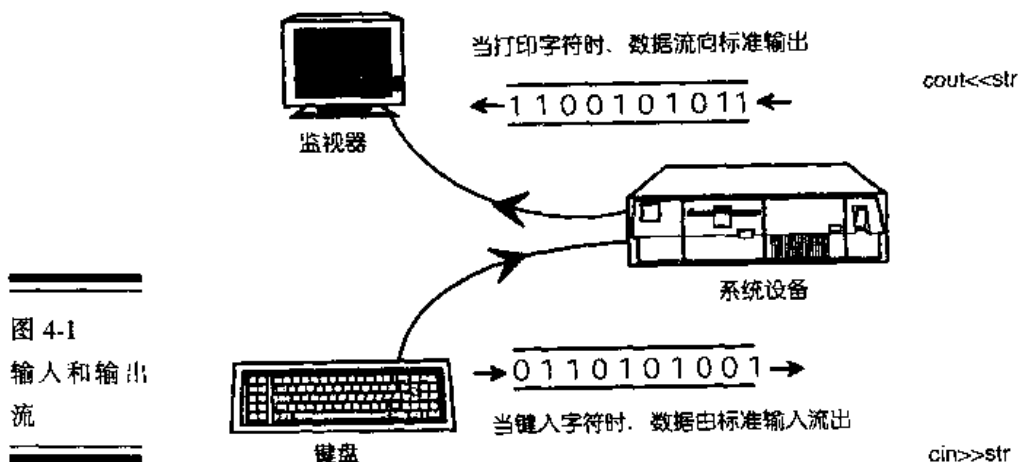
在日常生活中,流通常表示流动的水。在计算机程序设计中,流表示一系列流动的数据。流的一个主要的特征是它是单向的。尽管你可能认为这不完全正确(因为即使是一个小的流都可以有旋流和回流),但在这里用单向性来表述是数据流是正确的。一个输入/输出流是一系列持续不断地流向一个方向或另一个方向的字节,它不是输入就是输出。

流的另一个特征是它几乎是无穷无尽的。有时流会枯竭,但你是不会希望这种事情经常发生的。特别的是,除非在错误发生时,例如磁盘已满,一个输出流将一直接收另外的字节。

在程序设计中经常要用到的流是标准输入和标准输出,它们是不会枯竭的。你可以一直要求标准输入设备从键盘上提供另外的 ASCII 字符,你也可以一直向显示

屏上打印另外的字符(查看图 4-1)。

文件经常被看作流。当执行文件输入或输出时,你可以一直发送或得到下一个字节。除了出现错误,“执行至下一个字节”总是合法的。一个拥有数据但通常不被看作流的例子是数组或结构。数组经常具有指定的大小,不象输出文件那样能够增长。(尽管如此,C++ 库函数可以将数组视为流,但这些流可能会占据整个内存空间。关于这一点,请查看第十六章的 `stringstream`。)



从某些程度上讲,流的概念是由计算机科学家虚构出来的。例如,随机存取存储器(RAM)的一个区域可以被看作一个大的数组,它还可以被用作虚拟磁盘存储器,这两者都支持流。但在两种情况下,它都是同一个硬件。象程序设计中的许多概念一样,将某事看作流要视具体情况而定。我们可以这样定义流:它是一种使用环境,在这种环境下,“对下一个字节进行处理”是可行,同是也是有意义的。

一个流不是一个输出流就是一个输入流。除了在象已到文件尾或者磁盘已满这样的错误的条件下,下面的这些声明都是正确的:

- 对于一个输入流,你可以一直读下一个字节。
- 对于一个输出流,你可以一直写另外的字节。

具有读或写下一个字节的能力意味着具有读或写任何数量字节的能力。例如,同时写 2 个字节和先写一个字节然后再写一个字节的 effects 是相同的。当然,这些字节必须按一致的顺序进行读或写。

流操作符 << 和 >>

在 C++ 的输入输出操作中,既可以使用流操作符(<< 和 >>),也可以使用 printf、scanf 和其它在 stdio.h 头文件中定义的函数。流操作符提供两种方便:一是如果你打算使用默认的输入输出格式,就不必使用格式指定符;第二种是它能够扩展操作符,这样它们就能工作于你自己的类中。(类将在第五章中进行介绍。)要获得如何扩展操作符的信息,请查看第十六章。

● 注意

这一章介绍了几种输入/输出的技术:printf 和 scanf 函数、流操作符,还有基于行的输入。C++ 标准库为每种技术使用了不同的输入输出(I/O)缓冲器,因此混合使用它们将导致不可预料的后果。如果必须混合使用它们,就需要采取特殊的措施。C++ 中的输入输出(I/O)类提供 sync_with_stdio 函数来调整 printf 和 scanf 函数与输入输出(I/O)流数据的关系。请查看第十六章以获得更多的信息。

下面是一个得到两个浮点数并打印二者之和的简单程序:

```
#include <iostream, h>

void main() {
    double a,b;

    cout << "Enter the first number: ";
    cin >> a;
    cout << "Enter the second number: ";
    cin >> b;
    cout << "The total is ";
    cout << a + b << endl;
}
```

下面使用 printf 和 scanf 函数的程序与上面的程序完成同样的操作:

```
#include <stdio, h>

void main() {
    double a,b;

    printf("Enter the first number: ");
    scanf("%lf", &a);
```

```
printf("Enter the second number: ");
scanf("%lf", &b);
printf("The total is %lf\n", a + b);
```

注意上面使用流操作符 `cin` 和 `cout` 的程序和使用 `printf` 和 `scanf` 函数的程序的不同之处。

- 头文件是 `iostream.h` 而不是 `stdio.h`。
- 不需要格式指定符。在 C++ 中根据所使用对象的类型(例子中是 `a` 和 `b`)来决定如何传递相关数据。在这一方面,流操作符象 BASIC 语言中的 PRINT 语句,但是它比 `printf` 和 `scanf` 函数用起来简单一些。
- 在流操作符 `cin` 中没有对操作数使用地址操作符(`&`),但在 `scanf` 函数中却必须使用。(这里,输入流在使用引用变量上与 BASIC 语言很相似,这些将在第六章进行更多的介绍。)

- 数据流向标准输出设备(`cout`),它通常是指显示屏:

```
cout << "Enter the first number: ";
```

- 数据来自于标准输入(`cin`),它通常代表键盘:

```
cin >> a;
```

- C++ 的流对象使用 `endl` 打印一个回车,而 C 使用 `\n`(C++ 同样支持这种方式)。

箭头的方向起先看起来好象是任意的,但如果考虑数据的流向,你就会记住箭头应指的方向了。查看图 4-1 的示例。

● 注意

流操作符很象是左移和右移操作符。事实上,它们的确是这样的! 这种使用左移和右移操作符做为流操作符的方式是操作符重载的一个例子,这种技术将在第七章进行详细讨论。左移操作符的功能在 `istream` 和 `ostream` 类(这些类定义放在头文件 `iostream.h` 中)里被重新定义,所以在这些类里它就变成了“put to”操作符。

程序员在重载操作符时通常致力于保留操作符的普通意义。例如,你想让加号(`+`)在所有情形下都代表某种加法操作(尽管将两个字符串加起来与将两个数加起来的意义是完全不同的)。而将 `<<` 和 `>>` 用做流操作符是这种设计原

则的一个例外。从输入输出(I/O)流发送或接收数据与移位是没有任何关系的。为流操作重载 << 和 >> 操作符是因为它们在句法结构上和视觉上具有一定的便利(它们暗示数据流动)。

在这部分开始的例子中,最后的两行程序可以合并为一行。这两行程序打印两组数据:一个是字符串,一个是数字。

```
cout << "The total is ";  
cout << a + b;
```

可以使用下面更紧凑的语句代替上面的两条语句:

```
cout << "The total is " << a + b;
```

上面的语句是从左向右执行的,这意味着下面的表达式被首先执行:

```
cout << "The total is "
```

通常在 C 和 C++ 中,上面的表达式做两件事情:首先它将字符串送到 cout(这是它的侧面作用),然后它得到一个值。这种操作表明 cout << item 形式的表达式等价于 cout 本身。因此,cout << "The total is " 可以用 cout 代替。从而,C++ 按下面的方式执行这个表达式:

```
cout << "The total is " << a + b;  
(cout << "The total is ") << a + b;    //打印字符串。  
cout << a + b;                          //打印 a + b。
```

这是 C++ 中的一个技巧,它使你能在一个大的、复杂的表达式中重复使用 cout。在上面的例子中,首先打印字符串,然后再打印 a + b。

因为具有 cin >> item 形式的表达式等价于 cin,所以你可以使用同样的技巧在一个声明中同时输入几个数。例如:

```
cin >> a >> b >> c >> n;
```

图 4-2 表明 C++ 是如何分解这个声明的,作为表达式的侧面作用,每一时刻口述指令执行一个输入操作。

输入输出的格式

特别是对初学者来说,使用 cout 和 cin 的一个好处是你不必再使用 printf 和 scanf 函数中使用的那些古怪的符号。每种数据类型都由操作符来确定,而且每种数据类型都具有合理的默认格式。例如,在默认情况下,下面的语句按十进制格式打印

n, 十进制是数据的最普通格式。

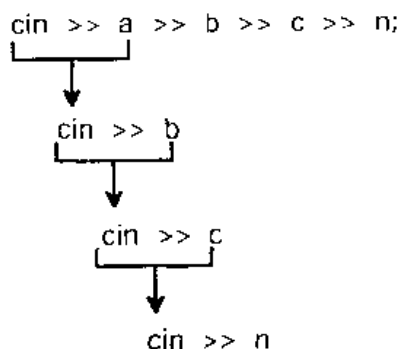


图 4-2

C++ 是如何处理 cin 的多个输入

```
cout << n;
```

但如果你打算用十六进制或八进制来打印该怎么做呢？例如，你可以使用 printf 声明完成下面的操作：

```
int n = 16;
printf("n is %x hex, %o octal, and %d decimal. \n", n, n, n);
```

此代码打印的结果是：

```
n is 10 hex, 20 octal, and 16 decimal.
```

下面是使用 C++ 流操作符的例子，它打印与上例相同的结果：

```
int n = 16;
cout << "n is " << hex << n << "hex,";
cout << oct << n << " octal, and ";
cout << dec << n << " decimal." << endl;
```

上面例子中 hex, oct 和 dec 代表的是什么呢？它们是特殊类型的对象（这将在第五章中进行更多的介绍）。当这些对象作为“output”送到 cout 或作为“input”送到 cin 时，它们改变流的整数格式直到有其它语句对它作出修改。

stdio.h 中基于行的输入

一些 C++ 程序员喜欢使用从 C 中继承来的在头文件 stdio.h 中声明的输入输出 (I/O) 库函数。如果你有一个 C 语言编程环境并且习惯于使用这些函数，或者你正在维护继承来的以 C 语言编写的代码，继续使用这些函数是有意义的。如果混用不同的输入或输出技术，你就可能会遇到问题。因此，对于现有的代码，在整个程序中

继续使用 `stdio.h` 函数还是上上之选。

除了第二章介绍的以外,输入和输出函数还有许多要解释的地方。首先, `printf` 和 `scanf` 函数支持多种不同类型的格式指定符,这些格式符对 `printf` 函数是特别有用的。你还可以利用它们控制浮点数的间隔和精度来调整你的输出格式。详细内容请查看第十五章有关 `printf` 和 `scanf` 函数的部分。

在 `stdio.h` 头文件中声明的一些十分有用的函数都支持面向行的输入和输出。例如,你可以使用 `puts` 函数作为打印字符串的另一种更高效方式。语句如下:

```
printf("%s\n", string);
```

下面的语句完成同样的功能:

```
puts(string);
```

后者效率更高,因为它避免了 `printf` 函数前面的格式描述部分。但同时, `puts` 函数的使用是不灵活的,因为它从新的一行打印一条语句,而这可能并不是你想要的。

在 `stdio.h` 中声明的一个非常有用的函数是 `gets` (“得到字符串”)。当你调用这个函数时,它将等待用户输入直到按回车键。然后这个函数把输入的整条语句(包括空格和所有的字符)放到字符串变量中。例如:

```
#include <stdio.h>
//...
char str[81];    //显示屏上允许一行显示的最多字符数
gets(str);       //得到输入的语句和字符串中的空格。
```

已经发现在除了最简单的程序之外的所有程序中,使用 `gets` 函数较使用 `scanf` 函数更具优越性。一旦已经将输入读取到一个字符串中,你就可以自由地选择任何方式来分析和解释它的内容。这个特点在编写一个编译器或其它复杂的应用程序时是有用的,因为此时你可以使用词法来分析输入。这就意味着你已经完全控制了是哪几个特征划分了不同的区域,以及间隔的顺序应被如何解释。当你使用 `scanf` 或 `cin` 时,你是不能控制一个数字或字符串区域定义方式。只有当 `scanf` 函数确定用户的输入是合法时,它才会允许用户输入。

做为行输入的一个简单的例子,你应该确定你的程序能够接收由 `at` 符号(`@`)描述的输入:

```
Here is some input@1234@More input@34.0005
```

下面的代码使用 `gets` 函数输入这个完整的行。然后程序将每个输入部分由 `@` 描述,再分配到不同的字符串中去。这个例子演示了如何输入嵌有空格的字符串,这

是 scanf 函数和 cin (>>) 输入操作符所不支持的。每一个单独的字符串都被读入二维数组 sarray 的不同行中。

```
#include <stdio.h>
char str[81];    //显示屏上一行允许显示得最大字符数。
char sarray[50][81];

void main() {

    int i = 0;
    char *p = str;
    char *s;

    gets(str);    //得到输入行并将它放入 str 中。

    while( *p != '\0' ) {

        //设置 s 等于数组中的下一个字符串。
        s = sarray[i++];

        //将下一个@符号以前的字符读入到字符串中。
        while( *p != '\0' && *p != '@' )
            *s++ = *p++;
        //停止读取字符——结束这个字符串并将指针前提越过@符号。
        *s = '\0';
        p++;
    }
}
```

当你已经隔离一个输入字符串的数字部分(例如,“1234”)并将其输入到目标字符串中的一个,你可以通过调用 atoi 或 atof 函数将其转换成数字值。查看第十五章以获得这两个函数的更多信息。在第十五章中还介绍了字符串提取函数 strtok,这个函数可以用来自动地完成这个工作中的一部分。

这个程序看起来没有做任何事。然而,可以通过在程序的末尾处添加如下语句来打印读入到字符串中的所有内容:

```
i = 0;
s = sarray[0];
while ( *s != '\0' ) {
    s = sarray[i++];
    puts(s);
}
```

gets 函数可以读取长度为 0 的字符串。例如,当用户键入一行输入"@This is a @@@string."时,如果读到长度为 0 的字符串,这段代码仍能很好地运行。

用 stdio.h 对文件进行输入输出

前面部分中谈到的面向行的输入输出(I/O)函数 gets 和 puts 在下面的例子中得到很好的使用,这个程序演示了对于文件的顺序输入和输出。例子中提示用户输入文件名,如果用户只是按回车键,gets 函数仍会以合理的方式响应这一输入,即输入一个空的字符串。

C++ 语言中的文件输入输出几乎和任何其它语言中一样地简单。如果你使用在 stdio.h 中声明的标准库函数,就应该依据下面的四个基本步骤(这些步骤在 C in Plain English 中也概括了):

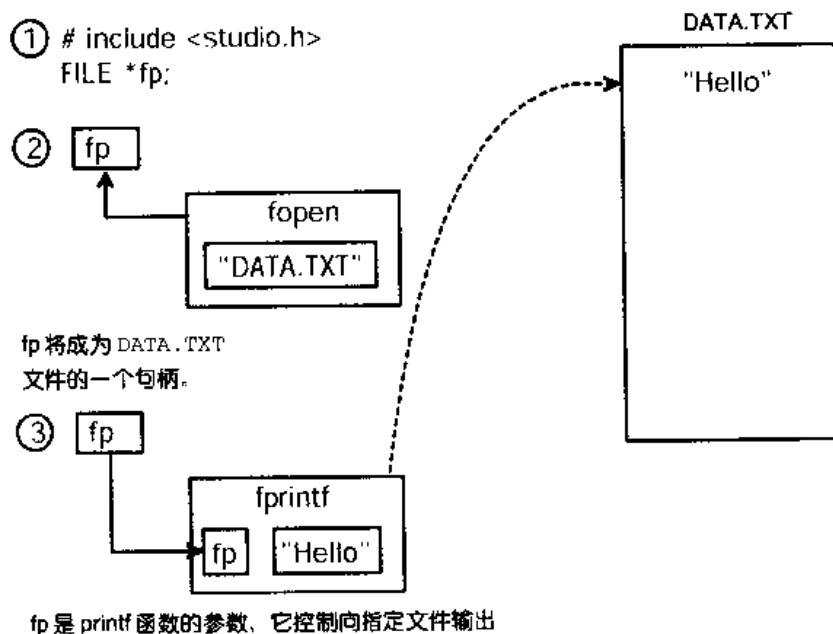


图 4-3
文件的 I/O
编程小结

1. 包含头文件 stdio.h,并声明一个 FILE * 类型的文件指针。(要为你想打开的每一个独立文件定义一个 FILE * 类型的指针。)
2. 通过调用 fopen 函数来打开文件。这个函数为文件指针返回一个值。
3. 恰当地使用诸如 fprintf ,fscanf,fputs,fgets 等文件输入输出(I/O)函数对文

件进行读或写。这些函数除了拥有一个文件指针作为额外的参数外,差不多和它们的副本 printf,scanf,puts,gets 做同样的工作。

4. 调用 fclose 函数来关闭文件。

图 4-3 举例说明了前三个步骤。这个图看起来有点过于简单了。在第二步中, fopen 函数主动地检查文件系统。(fopen 函数并不是仅仅将文件和一个名字联系起来。)如果被请求的文件没有被发现,或者是因为某些原因不能打开,那么程序必须停止并对错误状况作出响应。

下面的例子程序举例说明了这四个步骤,它调用的是 fgetc 函数而不是 fprintf 函数。请注意程序中的主函数有一个返回值。它使主函数能够向系统返回一个代码:0(表示成功),-1(表示失败)。

```
//Sample program to print out the contents of a text
//file, converting all lowercase to uppercase.
//The program prompts for the file name.

#include<stdio.h>
#include<ctype.h>

int main()
{
    int c;
    FILE * fp;
    char filename[81];

    printf("Enter file name, please:");
    gets(filename);
    if(( * filename) == '\0') { //Is string empty?
        puts("No file name entered. \n");
        return -1;
    }

    /* Open for reading; "r" specifies read mode.  * /

    fp = fopen(filename, "r");
    if(! fp){
        puts("Error: file name not found. \n");
        return -1;
    }
    while((c = fgetc(fp)) != EOF)
```

```
        putchar(toupper(c));  
  
    fclose(fp);  
    return 0;  
}
```

在这个例子中,文件被看作是输入文件,因此在第 23 行以读模式打开了这个文件(在下面重写了这行代码)。对于读模式,第二个变量是一个包含“r”的字符串。写模式是“w”,读写模式是“rw”。

```
fp = fopen(filename, "r");
```

程序中调用 `fgetc` 函数从文件中读取数据。此函数从输入流中读取一个单个字符,它是 `getchar` 的文件-输入版本,`getchar` 从标准输入上读取单一的字符。读取的字符与 EOF 进行比较,EOF 是 `stdio.h` 头文件中定义的一个常量,它代表文件结束标志。

在第 28 行(在下面重写了这行代码)中调用了 `fgetc` 函数。这个复杂的代码行调用 `fgetc` 函数得到一个字符,将结果分配到整型变量 `c` 中;并将其与 EOF 进行比较。只要返回的值不是 EOF(还没有到文件的末尾),这个循环就会继续。

```
while ((c = fgetc(fp)) != EOF)  
    //...
```

本章中所有的文件都假定使用基于文本的输入输出进行访问。这种技术将文件看作流,类似于显示屏和键盘。这种访问模式对于程序来说是最简单的,因为它是大多数初学者最初学习的标准输入输出技术(`printf`, `scanf` 等)。

然而,使用 `stdio.h` 函数读或写二进制文件也是很简单的。可以通过移动文件指针针对文件进行随机读写。要得到有关文件输入输出函数的更多信息,请查看第十五章。

文件操作符和流操作符

流的概念应用在磁盘文件上差不多和应用在显示屏和键盘上是一样的。C++ 支持 `ifstream`(输入文件流)和 `ofstream`(输出文件流)类型的对象。可以在初始化过程中通过指定名字的方法创建这些对象。但首先必须包括头文件 `fstream.h`。此文件自动包含 `iostream.h` 文件的所有内容。

```
#include <fstream.h>
```

为了创建文件-流对象,应当在定义这个对象时在圆括号内指定它的名字。例

如：

```
ifstream inf("C: \\ TXT \\ \\ DATA. TXT");  
ofstream outf("C: \\ \\ TXT \\ \\ OUTPUT. TXT");
```

当指定了完整的路径名后,要记住使用双引号将其括起来。一旦对象使用完整的或相关的路径名进行了定义和初始化,就可以象使用 cin 和 cout 操作符那样来使用它们。

```
outf << "Here is a string written to a file. ";
```

```
long n;  
inf >> n;
```

流对象在错误条件下得到 0 值,例如到达文件尾。因此可以象下面的语句那样,利用它编写循环,当文本结束时这个循环停止。

```
char string[81]  
while (inf)  
    inf.getline(string,80);
```

getline 函数是 inf 对象的一个成员函数;它提供与 gets 和 fgets 同样的功能。函数的第一个变量是一个字符串,读到的字符就被存储在这个字符串中;第二个变量是可读取的最大字符数。

下面的程序打印文件 DATA. TXT 的内容:

```
#include<fstream.h>  
  
void main(){  
    char buffer[81];  
  
    //打开输入文件 C: \\ STUFF. TXT.  
  
    ifstream inf("C: \\ \\ STUFF. TXT");  
  
    //如果没有到达文件尾就得到并打印一个文本行,每次都从新的一行开始打印。  
    //a line of text, appending a newline each time.  
    while (inf){  
        inf.getline(buffer,80);  
        cout << buffer << "\n";  
    }  
}
```

还可以使用输入输出流类来读和写二进制文件,也可以进行随机访问。查看第十六章以获得所有输入输出流类的更多信息,在这些类中包括文件流类 `ifstream`, `ofstream` 和 `fstream`。

争议:使用流还是不使用流

本章的主题是告诉读者可以有多种方法来完成输入和输出。大多数 C++ 的书偏爱使用这些方法中的一种。

这样做是有原因的(那些作者不但固执而且带有太多的偏见)。前面已经谈到,不应该将不同的输入输出技术混在一起使用,例如在一条语句中使用 `cin`,而在另一条语句中使用 `scanf`。除非你调用 `sync_with_stdio`,否则混用不同的输入输出技术将导致错误。通常,C++ 的作者很容易得到一种方法优于其他方法的观点,并坚持这种观点。他们没有遇到错误是因为没有混用不同的输入输出方式。

事实是你可以使用任何方式来编写一样有效、可读和雅致的代码。在简单的程序中使用流对象是较简单的。另一方面,当涉及到重要的格式时,使用 `printf` 和 `scanf` 函数可以获得可读性更高的代码。

流对象是令人感兴趣的,因为它们可以将类和对象的概念很快地引入到日常的编程中去。如果你没有大量的继承来的 C 代码需要维护,而且对面向对象的编程感兴趣,你就可以立即开始使用 `cin` 和 `cout` 了。

流操作符的最大优势是你可以为自己的类重载它们;这种能力说明面向对象的编程方法是可继承的,而且较传统的函数如 `printf` 有更多的延展性。例如,你不能为 `printf` 函数编写一个新的格式指定符,但使用操作符重载,你可以扩展 C++,这样它就知道如何打印你自己类中的对象了(把它们送到 `cout`):

```
MyClass myobject;  
cout << "Here is my object: " << myobject << endl;
```

在解释如何重载操作符之前,你需要面对类、对象、构造函数和成员函数等神秘的概念,它们将在下一章说明。这是一个过渡章节。幸运的是,现在已经激起你对对象的好奇心,而且也知道了它们在 C++ 中的角色。

第 五 章

类

*****x*****

C++ 中的一个重要概念是活跃类型:这种类型是由用户根据自己想要进行的操作和该类型的内部结构而定义的。这就是我们所说的类。

例如,如果你创建自己的字符串类型,那么把两个字符串加起来意味着什么呢?如果你爱好数学,你可以定义一个复数类型,那么对复数类型进行加或乘又意味着什么呢?在 C++ 中,你可以定义任何一种新的类型并为其定义任何的操作。

类描述了用户定义的所有种类的类型。一旦一个类被定义,它几乎成为 C++ 语言的一部分。简单的说,类扩展了 C++。

类的开发:一个更好的字符串类型

人们在学习 C 语言的过程中经常抱怨的一件事是字符串太麻烦了。例如,在 BASIC 语言中,你可以这样做:

```
str1 = "My name is "  
str2 = "Bill."  
str3 = str1 + str2
```

str3 中包含的消息是“My name is Bill.”。在 C 语言中完成这样的操作要花费更多的工作:

```
strcpy(str1,"My name is");  
strcpy(str2,"Bill");  
strcpy(str3,str1);  
strcat(str3,str2);
```

本章和接下来的两章将建立一种新的字符串类型,这种字符串类型和 BASIC 中的字符串类型一样的好。在许多方面,这种新的字符串类型还将优于 BASIC 中的字符串类型,因为 C++ 赋予你定制这种类型的能力。

从现在开始无论何时,只要谈到用户定义的类型,我们就使用 C++ 术语中的“类”来表示。这一术语将应用于结构、联合和任何由 class 关键字创建的数据类型。

按照下面的语法使用关键字 class 就可以创建一个类:

```
class 类名 |
    声明语句
};
```

在 C++ 中,声明包括函数声明和数据声明。(类中的函数和变量分别被称为成员函数和成员变量。)首先,让我们看一看最简单的一个字符串类:

```
class CStr |
    char sData[256];
};
```

在这个类定义中,CStr 是类名,sData 是数据成员,它包含了类的整个内容。这个简单的类用来存储字符串数据。现在向其中添加一些成员函数:

```
class CStr {
private:
    char sData[256];
public:
    char * get(void);
    int getlength(void);
    void cpy(char * s);
    void cat(char * s);
};
```

在这个类中,字符串数据 sData 被声明为 private 模式。此时只有本类内定义的成员函数才可以访问它。在本章的后半部分重写了这个类的内部成分,但并没有破坏它在旧版本中的代码。这是类的一个重要的优势。

当用 class 关键字声明类时,类内所有的成份缺省地置为 private 模式。使用 private 关键字来明确地声明私有成分是一个好的想法,但你也可以通过忽略这个关键字来节省工作。先前的声明与下面的声明是完全相同的:

```
class CStr |
    char sData[256];
public:
    char * get(void);
```

```

    int getlength(void);
    void cpy(char * s);
    void cat(char * s);
};

```

成员函数

你可以象为其它函数那样为类中的成员函数编写代码,唯一的区别是成员函数名必须带有特殊的前缀。

```
class_name::
```

两个冒号(:)构成一个单一的操作符,叫做作用域分辨符。这个操作符和类名一起阐明你正在谈论的是哪个类的函数。不同的类可以拥有相同名字的成员函数。例如,一个名为 CMyclass 的类也可以象类 CStr 一样拥有一个名为 get 的函数。在类声明之外的:

```
CMyclass::get
```

说明 get 函数是属于 CMyclass 类的成员函数,

```
CStr::get
```

说明 get 函数是属于 CStr 类的成员函数。类成员函数的语法格式为:

```

return_type class_name::function_name (arguments){
    statements
}

```

此时,类名是 CStr。在下面的函数定义中,要记住“CStr::”仅仅是函数名的一部分。除了这个 *class_name* 的前缀之外,这些函数的定义和普通函数的定义看起来是完全相同的。

```

#include< string.h>
char * CStr::get(void) {      //Return ptr to string data.
    return sData;
}

int CStr::getlength(void) {    //Return length.
    return strlen(sData);
}

void CStr::cpy(char * s) {      //Copy from string arg.
    strcpy(sData,s);
}

```

```

|
|
void CStr::cat(char *s) {      //Concatenate string arg
    strcat(sData,s);        //onto object.
|
|

```

图 5-1 总结了声明和函数定义是如何和类名联系在一起的,此时类名为 CStr。

```

class CStr {
    char sData[256];
public:
    char *get(void);
    int getlength(void);
    void cpy(char *s);
    void cat(char *s);
};

#include <string.h>
char *CStr::get(void) { // Return ptr to string data.
    return sData;
}

int CStr::getlength(void) { // Return length.
    return strlen(sData);
}

void CStr::cpy(char *s) { // Copy string arg to object.
    strcpy(sData, s);
}

void CStr::cat(char *s) { // Concatenate string arg
    strcat(sData, s);    // onto object.
}

```

图 5-1
CStr 类声明
和成员函数
的定义

将代码组织到文件中

类声明应该放在头文件中。函数代码应放在 C++ 的源文件中并编译、连接到当前的工程文件或一个库中。根据这种方法,CStr 的代码应按下面的方式进行组织:

```

//-----
//cstr.h
//This file declares the CStr class.
//Every module that refers to CStr must include
//this file.

class CStr {
    char sData[256];
public:

```

```

    char * get(void);
    int getlength(void);
    void cpy(char * s);
    void cat(char * s);
};

//-----
//cstr.cpp
//This file contains definitions of CStr functions.
//Compile this file and link to the project.

#include "cstr.h"
#include <string.h>

char * CStr::get(void) {           //Return ptr to string data.
    return sData;
}

int CStr::getlength(void) {        //Return length.
    return strlen(sData);
}

void CStr::cpy(char * s) {         //Copy from string arg.
    strcpy(sData,s);
}

void CStr::cat(char * s) {         //Concatenate string arg
    strcat(sData,s);              //onto object.
}

```

如果你想仅仅在某一工程文件的某一模块来声明一个类并使用它,你可以将包括类声明的所有代码放在该.cpp 文件中。此处描述的是将标题和类实现文件分开来的方法,这种方法是适合于类的,你将会看到在不同的工程文件中大量使用这种方法。

另一种方法是将一个工程文件中所有的类声明放在某一头文件中。然而在任何情况下,函数定义都不是类声明的一部分,应该把它们放在一个或多个.cpp 文件中进行编译。

分号符(;):一个备须注意的语法现象

除非你非常小心,否则使用了太多或太少分号(;)的语法错误你肯定遇到过。类

声明的结尾要用到分号,但函数定义的结尾是不用的。

作为一个普通的规则,C 和 C++ 声明的结尾“`!`”符号之后不跟随一个分号。所有函数的定义都遵循这个规则:

```
char * CStr::get(void) {  
    return sData;  
}
```

但是,类、结构和联合的声明的结尾“`!`”号之后需要分号来做为结束符。

```
class CStr {  
    char sData[256];  
public:  
    char * get(void);  
    int getlength(void);  
    void cpy(char * s);  
    void cat(char * s);  
};
```

这个规则在 C 和 C++ 中似乎太武断了,但它对于帮助编译器辨别是声明还是函数定义是有好处的。除了函数定义之外的所有声明都需要用分号作为结束符号。这也包括函数原型,无论它们是否在一个类内:

```
char * get(void);
```

对象

类声明和函数定义一起定义了字符串类型是如何工作的。下一步要做的是使用 CStr 声明来创建一些字符串:

```
#include "cstr.h"  
  
CStr str1, str2, str3;
```

在 C++ 中,一旦你已经成功地声明了一个类,你就可以使用这个类名来声明变量,就象使用 `int`, `long`, `float`, `double` 等声明变量一样。在语法中,类名与数据类型关键字具有相同的地位。

● C/C++

能够定义变量的能力被扩展到 `struct` 和 `union` 关键字。在 C++ 语言中,一旦你

声明了一个 Mystruct 结构,就可以使用它来声明变量。例如:

```
Mystruct a,b,c,d;
```

在 C 语言中,你必须使用 struct 关键字:

```
struct Mystruct a,b,c,d;
```

如果你使用 typedef 对结构名称进行了声明,就可以省去这一点麻烦。但对于上面的述的任何一种情况,C 语言都需要做一些额外的工作。C++ 里为了兼容 C 的这些语法特性,它也在支持在这种情况下使用 strut 和 typedef,但随着对 C++ 的使用,你会很快习惯直接定义变量的方式。

我们按下面方法声明的字符串应该叫什么名字呢? 这里的 a,b,c,d 究竟是什么呢?

```
CStr a,b,c,d;
```

这些变量都是对象。如果你曾经听说过面向对象的编程技术,这对你来说是太令人兴奋了,因为你终于得到对象了。

然而,在进行庆祝之前,你应该认识到其实你一直在使用对象。C++ 中的一个变量或一条数据就是一个对象。在 C++ 中,对象令人兴奋的地方是它们可以通过对应的类声明来支持函数调用和其它操作。

调用一个成员函数

在进一步讨论以前,让我们看看字符串对象内的行为。这些对象都知道怎样响应 get, getlength, cpy 和 cat 函数:

```
#include "cstr.h"
```

```
CStr string1, string2;
```

```
string1.cpy("My name is:");           //将字符串 My name is 复制到 string1  
string2.cpy("Bill, ");                 //将字符串 Bill 复制到 string2  
string1.cat(string2.get());            //将 string2 连接到 string1  
puts(string1.get());                  //得到 string1 并打印它
```

上面例子中的每个声明都与一个成员函数的调用相联系。注意在函数调用中使用的圆点(.):

```
object.member_function(arguments)
```

如果你在 C 语言中曾经使用过结构,可以看出这两种语法是相似的。它是对结

构成员语法的一个自然扩展：

object.member

进一步研究这些函数调用,你会发现第一个函数调用发给对象 `string1` 一个命令,相当于:“把字符串复制给自己”。

```
string1.cpy("My name is");    //将字符串复制到 string1
```

这个函数调用告诉 `string1` 把字符串“My name is”复制给自己。因为 `string1` 对象有内建的 `cpy` 函数,所以它知道如何按这种方式复制其它的字符串。可以将 `cpy` 比喻成对象行为的一部分。

实际上,函数调用过程中所发生的事是,编译器将 `string1.cpy()` 看成一个函数调用。它知道 `string1` 是 `CStr` 类的一个对象。因此这个函数调用就变成了对 `CStr::cpy` 的一个调用(参见图 5-2)。

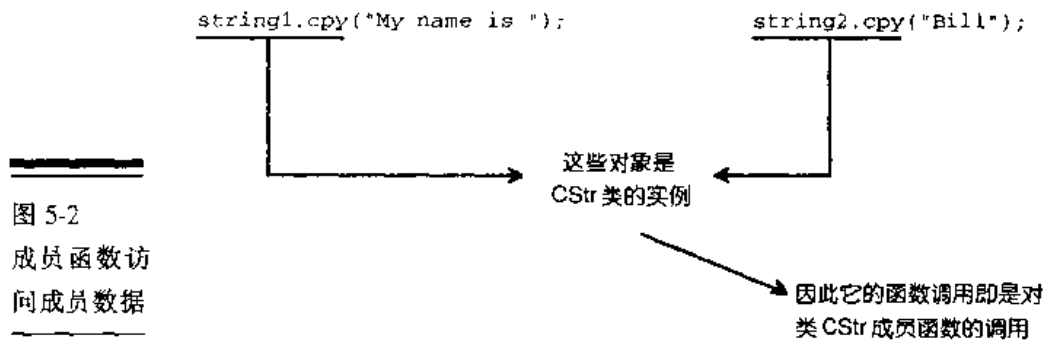


图 5-2
成员函数访问成员数据

在程序的开始处, `string1` 和 `string2` 被定义为类 `CStr` 的对象。因此对 `string2.cpy()` 的调用也将导致对 `CStr::cpy` 的调用。同一个类中的所有对象共享它们全部的函数代码,因此也就具有相同的行为。

`CStr::cpy` 函数调用过程中发生了什么呢? 向 `CStr::cpy` 函数传递控制与向常规函数传递控制是一样的。

但是此时发生了有趣的事,如图 5-3 所示,变量 `sData` 是 `CStr` 类的成员变量,这个成员变量是属于对象 `string1` 的。

每个 `CStr` 类的对象都有自己的对数据成员 `sData` 的拷贝。在这个例子中,声明通过字符串对象 `string1` 调用 `CStr::cpy` 得到 `sData` 的拷贝。因此,当函数涉及到 `sData` 时,被引用的是 `string1` 对 `sData` 的拷贝。图 5-3 说明了这种关系。

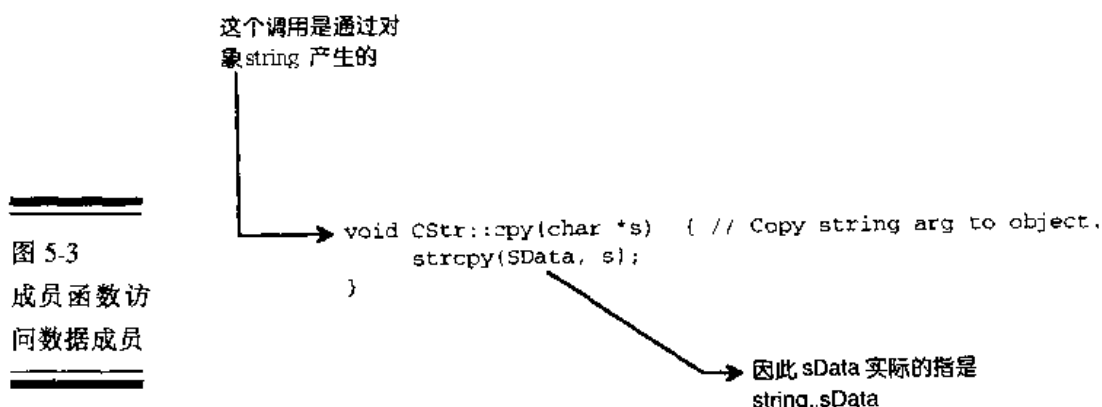


图 5-3
成员函数访问数据成员

代码和数据在对象中的作用是不同的。无论类内成分是公共的还是私有的,下面的规则都是普遍适用的:

- 同一个类中的对象共享函数代码。C++ 通过确定对象所对应的类,然后调用 `class::member-function` 来解决对一个成员函数的调用。
- 每个单独的对象有它自己对数据成员的拷贝。

理解这种区别的一个方法是记住对象是数据的一个封装。如果你在 C 语言中曾经使用过结构,你会知道实际上它与 C++ 中有数据成员的对象是没有什么不同之处的。因为这些封装是变量的集合,所以它们可以是互不相同的。每一个 C++ 的对象都有它自己的对数据成员的拷贝。

成员函数的概念是新的。但它只不过是限制工作于某一特定类的数据的函数。一个成员函数在它类中的每个对象里都一样工作。因此,可以认为成员函数存在于类的级别上,而数据存在于对象识别上。

图 5-4 说明了类和对象的关系和它是如何包括代码和数据。假定你定义了四个字符串对象:

```
str1, str2, str3, str4;
CStr str1, str2, str3, str4;
```

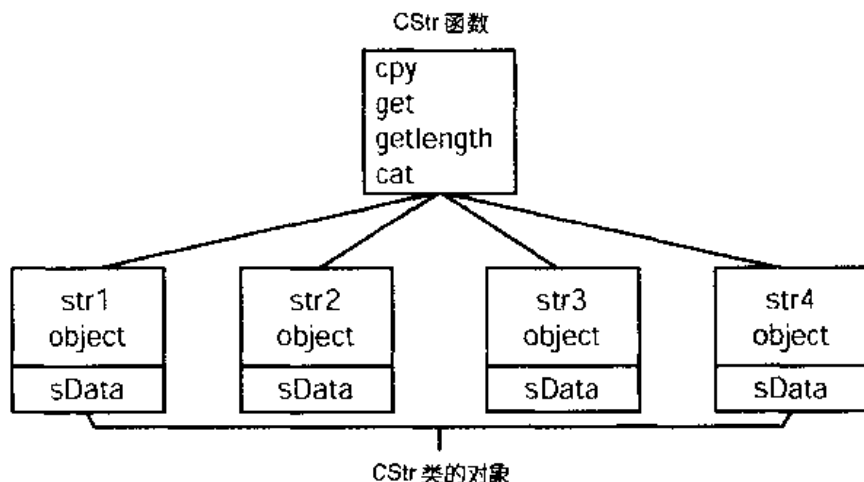
每个对象都共享为 CStr 类创建的代码,同时有自己的对 CStr 类中数据成员 sData 的拷贝。

在 CStr 类中只有一个数据成员 sData,但大多数情况下数据成员是多于一个的。

例如,我们将创建一个 CTemp_point 类,它定义了三维空间上的一个点和这个点上的一个属值:

```
class CTemp_point
{
    int x,y,z;
    double temp;
public:
    void set_point(int x,int y,int z);
    void get_point(int *x,int *y,int *z);
    void set_temp(double new_temp);
    double get_temp(void);
};
```

图 5-4
CStr 类中的
代码和数据



同样假定这个类的四个对象:

```
CTemp_point pntA, pntB, pntC, pntD;
```

在这里,这个类中的所有实例,也就是单独的一个点,共享 CTemp_point 的函数代码,就象字符串对象共享 CStr 的函数代码一样。但是每个点都有自己的对四个数据成员:x,y,z 和 temp 的拷贝(图 5-5 所示)。

成员函数

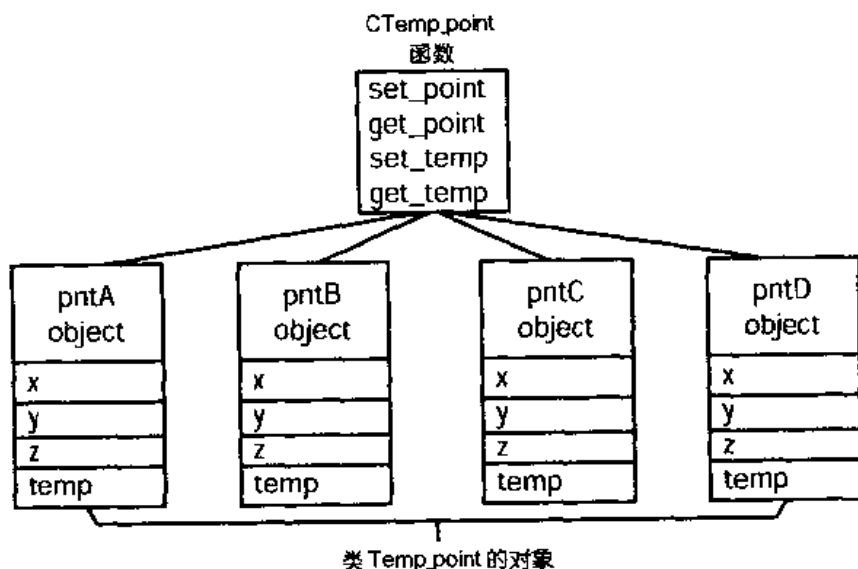
如果你明白成员函数是如何工作的,你可以跳过这一节。这一节通过完成一系列的函数调用来说明成员函数是怎样工作的。考虑下面的代码:

```
#include "cstr.h"
```

```
CStr string1, string2;
```

```
string1.cpy("My name is ");    //调用成员函数 CStr::cpy。
string2.cpy("Bill. ");         //调用成员函数 CStr::cpy。
string1.cat(string2.get());     //调用成员函数 CStr::cat 和 CStr::get。
puts(string1.get());           //调用成员函数 CStr::get。
```

图 5-5
在 CTemp-
point 类中
的代码与数
据



因为 string1 是 CStr 类的一个对象,所以第一个函数调用导致对 CStr::cpy 函数的调用。调用 CStr::cpy 函数的结果是设置了 string1 的数据成员的拷贝。

```
string1.cpy("My name is ");    //调用成员函数 CStr::cpy。
```

下面一行代码导致对同一函数 CStr::cpy 的调用。但是这个调用是通过对象 string2 完成的,因此对象 string2 的数据成员改变了。

```
string2.cpy("Bill. ");        //调用成员函数 CStr::cpy。
```

再下面的两行代码导致对函数 CStr::cat 和 CStr::get 的调用(函数 CStr::get 被调用了两次)。

```
string1.cat(string2.get());    //调用成员函数 CStr::cat 和
                               //CStr::get。
puts(string1.get());           //调用成员函数 CStr::get。
```

作为最后的一个例子,让我们考虑另外的一个类 CTemp_point 的声明。假定在程序的其它部分提供函数的定义。

```
class CTemp_point{
    int x,y,z;
    double temp;
public:
    void set_point(int x,y,z);
    void get_point(int *x,*y,*z);
    void set_temp(double newtemp);
    double get_temp(void);
};
```

我们可以声明这个类的两个对象：

```
CTemp_point mypoint, point_break;
```

下面的函数调用导致对 CTemp_point::set_point 的调用。如果任何数据成员受到影响,那么它们将成为对象 mypoint 对这些成员的拷贝。

```
mypoint.set_point(0,0,0);
```

对象指针

就象可以定义指向结构和变量的指针那样,同样可以在 C++ 中定义指向对象的指针。

为了动态地创建对象,就不得不使用指针。这意味着在创建对象时可以任意的分配和释放它们,而不必给出它们在程序中甚至一个函数调用中的生命周期。

C++ 支持 new 和 delete 关键字来动态创建对象。(它们在第三章中介绍过。) new 关键字返回一个指针。当使用它时,应在它的后面跟随一个类型或类名。

```
CStr * pString;
pString = new CStr;
//...
delete pString;    //释放分配的内存。
```

尽管也可以使用 malloc 和 free 函数为对象动态地分配内存,但因为 new 和 delete 可以自动地调用构造函数和析构函数,所以它们更方便。本章将在后面部分中介绍构造函数和析构函数。

在 C++ 中使用对象指针的语法是在 C 中使用结构指针的一个自然的扩展。当指针指向一个数据成员时,使用 → 操作符:

```
ptr→member
```

毫无疑问,也可以使用相同的语法来调用对象指向的成员函数:

```
ptr→member_function(args)
```

使用私有数据的好处

如果你善于思考,你可能会考虑类和成员函数的精彩之处究竟是什么?难道使用成员函数只是为了适合语法规则吗?让我们来考虑下面两个函数的区别。

```
strcpy(string1, "hello");  
string1.cpy("hello");
```

在这里成员函数至少有一方面主要的好处。在标准的方法中,重写一个模块通常具有冒险性并可能在使用这个模块的代码处引起问题。但使用类时,你可以重写函数的定义和私有数据成员而不会引起错误。

在软件开发上的一个十分紧迫的问题是:如何在修改、升级或改变程序的一部分时,不会破坏与此部分相联系的其它部分。面向对象的编程技术帮你解决了这个问题。除了公共成员外,你可以改变所有的私有成员。只要不改变变量和返回类型,你还可以重新定义公共函数。

本章最初对类 CStr 的声明与理想的情况是相差很远的。它可以被改进的更高效和更灵活。当前的 CStr 实现的一个问题是将其存储字符串最大范围固定为 256 个字节。这个范围在一些事例中太小,而在其它的事例中又太大。

```
class CStr{  
    char sData[256];  
public:  
    char * get(void);  
    int getlength(void);  
    void cpy(char * s);  
    void cat(char * s);  
};
```

这个类定义的效率是较低的。一个好的类实现使用可以指向任何地址的指针。为了准确表达字符串长度,它把长度做为整数来存储。

```
class CStr{  
    char * pData;
```

```

        int nLength;
public:
    char * get(void);
    int getlength(void);
    void cpy(char * s);
    void cat(char * s);
};

```

类 CStr 的内部成员被改变了,但是因为公共部分的声明没有改变,所以程序的其它部分是不需要改动的。

因为成员变量 sData 被去除了,所以所有的函数都必须重新定义。在这个实现中,创建字符串和在需要的时候重新创建字符串时,使用了动态地内存分配技术。

```

#include< string.h>
char * CStr::get(void) {           //将 ptr 返回给字符串数据。
    return pData;
}

int CStr::getlength(void) {        //返回字符串长度。
    return nLength;
}

void CStr::cpy(char * s) {        //复制字符串 arg。
    int n;

    n = strlen(s);
    if(nLength != n){
        if(pData)
            delete [] pData;
        pData = new char[n+1];
        nLength = n;
    }
    strcpy(pData,s);
}

void CStr::cat(char * s) {        //连接字符串 arg。
    int n;
    char * pTemp;

    n = strlen(s);
    if(n == 0)
        return;

```

```
pTemp = new char[n + nLength + 1];
if(pData){
    strcpy(pTemp,pData);
    delete [] pData;
}
strcat(pTemp,s);
pData = pTemp;
nLength += n;
}
```

尽管程序中做了许多改动,但所有使用 CStr 类的代码仍可以工作。例如,先前版本中的下述代码可以继续工作而不必作任何改动:

```
CStr string1,string2;

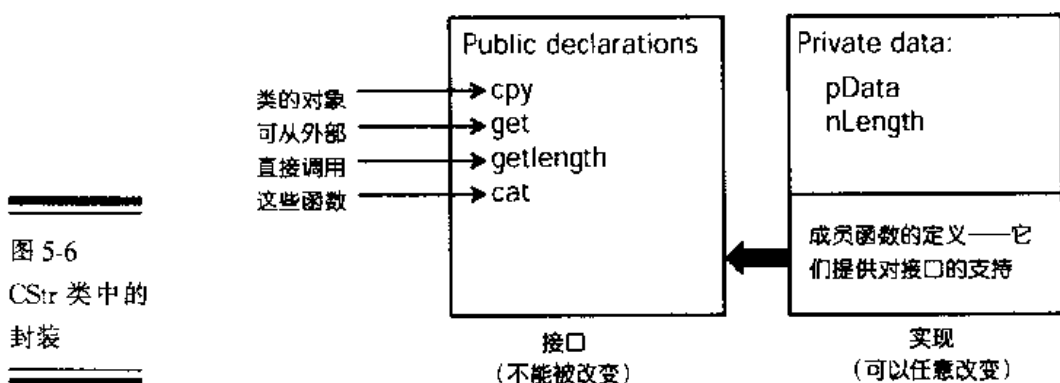
string1.copy("My name is"); //将字符串复制到 string1。
string2.copy("Bill.");      //将字符串复制到 string2。
string1.cat(string2.get()); //将 string2 连接到 string1 上。
puts(string1.get());        //得到 string1 的数据并打印它。
```

尽管类 CStr 的外部代码没有改变,但前面对这个类的改动的意义是深远的。此时的 CStr 类象改动之前一样的工作,对于外部世界来说它们是一样的,但是它完成工作的方式更加有效。现在类 CStr 的实现使用的是 new 操作符,“在任何特定的时间都能精确地分配数据所需要的字节数。下一部分介绍这些函数是如何工作的。

你将体会到面向对象的编程技术的魔力所在,那就是你可以改变你想改变的类定义代码的一大部分而不必担心会在程序的其余部分中引入不可预见的错误。

这种魔力被称为封装(encapsulation)。它的意思是对象的某一部分对于外部来说是被封装起来或被保护起来,如图 5-6 所示。当然它必须同外部世界有一定的交互作用,否则类就是毫无用处的了。外面的部分通常被称为接口(interface)。

在标准 C 语言中,任何声明都可以到达一个数据结构的内部并访问它的任何部分。如果你不重写这个数据结构的某些方面,这种做法是好的。但如果重写这个数据结构会使调试工作变成一件十分可怕的事情,因为任何地方涉及到这个结构的所有代码都必须重写,但是在一个大的程序中,你可能已经忘了涉及你的数据结构的所有地方。使用封装就会阻止这些可怕事情的发生。



动态内存分配的实现

在上面的部分中,我们说类 CStr 的新的实现是更加高效的。即使有更多的代码要执行,这一点也是正确的。

在字符串使用内存方面,这个新类是更加有效的。每一个字符串在任何时候只占据它所需要的存储空间。这种内存的管理方式不允许太大的自由度:为管理这个内存还需要其它的语句。但是从基本上说,字符串类是较好的。首选在于它没有把存储空间武断的限定为 256 个字节。

cpy 和 cat 函数使用了 new 和 delete 操作符。在第三章已经介绍过,new 操作符象 malloc 函数一样返回一块被请求的内存。但是,new 操作符使用起来较简单并且不需要头文件。new 的使用格式如下:

```
pointer = new type[num_elements];
```

当分配的内存不再需要时,可以调用 delete 操作符来释放它:

```
delete [] pointer;
```

●—注意—

分配给 CStr 的内存也可以使用 malloc 和 free 函数来完成而不必使用 new 和 delete 操作符。但要确保 free 与 malloc 对应使用,delete 与 new 对应使用。不要试图将它们交叉使用。

在新的实现中,成员函数 cpy 将一个新的字符串复制到当前的字符串区域内。但是它有一个重要的侧面影响。如果新的字符串比当前的字符串长或短,当前的字

符串就必须增长或改变它的大小。如果新的字符串比较长,这一点至关重要,因为在这个实现中已没有可用的字节了。字符串的增长会破坏其他的数据。

解决的方法是分配一块新的内存并将新的数据复制到那里。它的步骤是:

1. 得到新的字符串数据的长度。
2. 如果新的长度不同于当前的长度,那么释放当前的内存并按照新的字符串的大小分配一块新的内存。
3. 复制新的字符串到这个新的内存中。

首先,函数得到新字符串的长度并将它存储到局部变量 `n` 中:

```
n = strlen(s);
```

然后,新字符串的长度和当前字符串的长度进行比较,如果不相等,当前的字符串数据必须增长或缩短到新的大小,同时还要附加 1 个字节作为终止符。最简单的方法是释放当前的内存然后按正确的大小分配一块新的内存。最后,类的变量 `nLength` 更新为新字符串的长度。

记住要为终止符分配一个额外的字节。

```
if(nLength != n){  
    if(pData)  
        delete [] pData;  
    pData = new char[n+1];  
    nLength = n;  
}
```

函数的其余部分把新的字符串数据复制到对象的当前字符串中去(由类的变量 `pData` 指向)。

```
strcpy(pData,s);
```

成员函数 `cat(concatenate)` 是比较复杂的。在连接新的字符串以前,它不但要创建一块新的内存区而且还要从当前字符串数据中复制数据。没有这个预备工作,就没有空间来让对象的字符串数据在不破坏其他数据的条件下进行增长。步骤是:

1. 得到新字符串数据的长度。如果字符串的长度是 0, 就立即返回, 因为这没有什么可以做的。
2. 连同终止符的一个字节一起, 分配一块足够大的内存空间来容纳组合的字符串。
3. 如果是当前的字符串, 就把它复制到新的内存区中去然后删除旧的内存区。
4. 最后, 连接新的字符串并将 pData 指向新的内存区。同时更新 nLength。

前三条语句得到新字符串数据的长度, 如果长度是 0, 就立即返回。

```
n = strlen(s);  
if(n == 0)  
    return;
```

然后函数为组合的字符串和一个字节的终止符分配足够大的内存区。

```
pTemp = new char[n + nLength + 1];
```

现在我们有了一块由 pTemp 指向的新的内存, 它足以容纳组合的字符串。接下来要做的是将当前的字符串复制到这块内存中来, 然后释放原先存储数据的内存空间。

```
if(pData){  
    strcpy(pTemp, pData);  
    delete [] pData;  
}
```

可以看出, 这里有一块内存区用来容纳当前的字符串, 它还有额外的空间用来将新的字符串数据连接到末端。在执行完连接后, 函数以更新 pData 和 nLength 作为结束。

```
strcat(pTemp, s);  
pData = pTemp;  
nLength += n;
```

对象的生存期:构造函数及其它

CStr 类有两个问题。第一个是在字符串数据赋值以前,成员变量 pData 不会指向任何东西。这是很冒险的,因为用户可能希望得到一个有意义的地址,即使它是一个空字符串的地址。但是 get 函数简单的返回 pData,它可能是一个空指针。

```
char * CStr::get(void){    //将 ptr 返回给字符串数据。  
    return pData;  
}
```

最好的解决办法是在创建一个对象时对 pData 进行初始化。在默认情况下,成员变量被初始化为 0 或者(在指针的情况下)NULL。但这不是一个好的编程习惯。

还有一个更坏的问题是,当字符串被破坏时,不能释放当前的内存。它造成的结果是一个内存泄漏;每当一个字符串被创建、初始化和破坏时,它都留下一个内存的漏洞。如果你的内存不是无限大的话,这是一个非常严重的问题。

●—注意—

当一个对象作为一个函数的局部对象时,函数一结束,这个对象也就被破坏了。全程的对象只有在程序结束时才终结。

一般的问题是在创建一个对象时需要采取某些措施,在破坏这个对象时也需要采取某些措施。幸运的是,C++ 通过提供构造函数和析构函数使得对象的初始化和清除变得简单了。这些特殊的成员函数控制一个对象是如何被创建或破坏的。函数的命名语法也与众不同,对于一个特定的类,

- 构造函数的名字是本类的类名,
- 析构函数的名字是本类的类名,在前面加“~”符号。

前面提到的字符串类的名字是 CStr,因此构造函数和析构函数分别被命名为 CStr 和 ~CStr:

```
class CStr{  
    char * pData;  
    int nLength;
```

```
public:
    CStr();           //构造函数。
    ~CStr();          //析构函数。

    char * get(void);
    int getlength(void);
    void cpy(char * s);
    void cat(char * s);
};
```

构造函数和析构函数与普通函数有一些不同的地方,一个是它们没有任何返回类型,甚至连 void 类型也不可以有。这是 C++ 的函数必须有一个返回类型的规则的一个明显的例外。另一个不同之处是函数的参数列表可以什么都不写,而不使用 void。

构造函数的定义初始化了两个类成员变量,同时分配给类一个单字节的字符串。这个字符串包含一个 \0 终止符。

```
CStr::CStr() {
    pData = new char[1];
    *pData = '\0';
    nLength = 0;
}
```

在函数的标题中,CStr 出现了两次。CStr 第一次出现时应该带有作用域分辨符 (::),表明这个函数是 CStr 类的成员函数。因此“CStr::”是一个前缀。CStr 第二次出现是函数本身的名字。

析构函数是较简单的。它所做的只是使用 delete 操作符释放当前的内存区。要记住函数名是 ~CStr。

```
CStr::~CStr() {
    delete [] pData;
}
```

析构函数容易受限制。它所做的只是释放附近的系统资源。构造函数较析构函数更令人感兴趣。我们将在第六章介绍它们的更多信息。

内联函数

封装看起来很精彩,但它的效率太低。(封装是这样的一个概念,就是内部成分,例如 `pData` 和 `nLength` 不能从外部进行访问。一个对象的用户只能调用公共函数才能得到它们的值。)封装有许多的好处,但它也意味着你要经常以单行函数结束,如下所示:

```
char * CStr::get(void) |      //将 ptr 返回给字符串数据。
    return pData;
|

int CStr::getlength(void){    //返回字符串长度。
    return nLength;
|}
```

每一个函数调用都有一个类内的确定的函数原型与其相联系,因此调用一个函数在得到一个值的同时使得程序的性能下降了。幸运的是,C++ 提供了一个优化的解决办法:内联函数。可以将函数定义的代码块,直接写入到类声明中相应函数原型的结束分号之前,来代替通过处理器的 `CALL` 指令执行的一个调用。在这种情况下,如果 `getlength` 是一个内联函数,那么编译器将把下面的声明:

```
x = string1.getlength();
```

替换为:

```
x = string1.nLength;
```

这种管理方式消除了函数调用,保持了程序的运行速度。

通常情况下,第二个 C++ 的声明是不合法的(`nLength` 是一个私有成分不能进行访问),但因为 `nLength` 是通过公共成员函数 `getlength` 来访问的,此时这样表达是没有问题的。使用内联函数的方法既保留了封装又保持了效率。

在 C++ 中,如果把函数代码放在类声明的内部,它们是自动被内联的。此处,`get` 和 `getlength` 函数就是这样的:

```
class CStr{
    char * pData;
```

```
    int nLength;  
public:  
    CStr();           //构造函数。  
    ~CStr();          //析构函数。  
  
    char * get(void) {return pData;}  
    int getlength(void) {return nLength;}  
    void cpy(char * s);  
    void cat(char * s);  
};
```

尽管它们变得简短了,但 `get` 和 `getlength` 函数遵守的语法规则与常规的函数是一样的。尽管大多数的内联函数非常短小,但如果你选择的话,它们也可以有多条语句声明。象标准函数定义一样,在这些函数定义的结束符之后没有跟随一个分号。

C++ 也支持 `inline` 关键字,因此可以选定哪个函数做为内联函数。但是,在类声明的内部的成员函数是自动内联的。

类的一种特殊情况:结构

类和结构的区别是什么呢?如果你在 C 语言中曾经使用过结构,就会记得它们是不同的类型数据的集合。它与类的区别是什么呢?

在 C++ 中,你既可以向结构声明中添加成员函数,也可以向类中添加成员函数。类和结构的唯一区别是:在结构中所有的成员默认是公共模式,而在类中所有的成员默认是私有模式。顺便说一下,C++ 中的结构需要后向兼容于 C 中的结构。

这种情况容易引起混淆,因为词“class”不但涉及到用 `class` 关键字创建的类型,而且涉及到用 `struct` 和 `union` 创建的类型。可以将结构和联合看做类的特殊情况,在这种情况下,所有的成员默认是公共模式的。当使用 `class` 关键字时,所有的成员默认是私有模式,这是提倡的方式,因为 C++ 鼓励使用封装。

类的远景

这一章篇幅较长,但类是一个简单而雅致的概念。类是用户定义的一个类型,类中的部分代码可以包括它支持的操作(函数)。换句话说就是,你告诉我它的功能,我就知道它是什么。

封装

C++ 类的一个最重要的特征是封装。它意味着在类与界面之间存在一个清楚的分割,类对外界来说是公共的,但类内的成分是不能被类外的部分访问的。这样的分割是十分有用的,因为它意味着你可以重写类内的部分而不会将程序的其他部分搞糟。(这样对类进行修改一般是不会发生错误的。)在 C 语言的旧版本中,贯穿整个程序内部的令人难以理解的关系意味着在任何时候的任何修改,不管它们是如何正确,都可能破坏整个系统。

我经常说面向对象的编程技术消除了引起错误的原因,但你知道这种说法是不完全正确的。然而,许诺和使用封装技术,的确避免了一个主要的引起错误的原因。

类、对象和实例

这一章中另一个重要的概念是类和对象之间的区别。类是用户定义的一个类型;对象是相应类的实际实现。这并不象听起来那么抽象。类是一系列对象的概括或装配线。类决定了每个对象的大小、形状、特性和内嵌的行为。一旦类定义好了,属于这个类的许多对象就可以跟着被创建。

对于每个类,它可以有或没有多个对象。除了当使用多重继承时,每个对象只能属于一个单一的类。(多重继承是类的一个高级的特征)。在很大程度上,类与对象的关系是一对多的关系。创建某个类的一个对象的过程被称为实例化。没有对象的类是不能被实例化的。一些被称为抽象类的类,尽管没有被实例化,但它们仍能产生作用。在第九章中将进一步讨论这样的类。

类的重新使用及发布

在结束这一章前,应该强调的是,在处理大的程序,或者创建一个新的功能强大的能够被用在多个程序中的类型时,类是十分有用的。

我们正在把 CStr 创建成一个有用的类。一旦创建完成,它将成为一个有效的语言的扩展,而且它可以用在许多程序中。然而,要完成它还有许多工作要做。

当你把你编写的一个类交给某人并对他说“把它用在你自己的程序中吧”,这时类是最有用的了。为了使这个类的源代码可用,你应该按如下方式组织你的代码:

- 将你的类声明放在一个头文件中。头文件中通常包含程序中每个模块要用到的类型信息,因此程序员使用 `#include` 指令将此类型信息读到每个模块中去。
- 将函数的定义放在一个单独的源文件中(除了那些内联函数),源文件编译完成后送给类用户将其连接到他或她的工程中去。做为一个如何将代码分成头文件和源文件的例子,请查看本章的“将代码组织到文件中”部分。

将头文件和源代码分开的好处是,你不必再与使用你的类的用户共享你的大多数源代码。你只需要提供头文件和包含经过编译的目标文件就行了。类用户将头文件包含到他们的每个模块中,然后再把你的目标文件添加到连接器的命令行中就可以使用了。可以使用“make”文件、工程或批处理文件来完成后面的一步。

通过对类的介绍,你差不多能够编写一些可以重复使用在多个程序中的有用的类了。但是,即使象 CStr 这样中等复杂的类都需要仔细的编写复制构造函数、操作符和标题。这些将在下面的两章中介绍。

第 六 章

构造函数

构造函数的名字是非常切题的,构造函数就是制造一个对象的函数。你可以简单的把构造函数看成初始化函数,但实际上它们有一些重要的微妙差别。

为了正确编写你的类,常常需要特殊的构造函数。一个非常重要的函数是复制构造函数,它告诉类如何将自己的一个实例做为一个变量来传递或者是如何返回一个值。就象下一章讨论的那样,没有一个正确编写的复制构造函数,基于 CStr 类的代码可能会引起错误。

这一章的内容比第五章少。只有一些关于构造函数的关键结论需要你记住,但是这些结论对于你的 C++ 函数的正常运行是至关重要的。

构造函数的重载

当定义一个字符串类时,你可能想用不同的方式来初始化它。例如,将下面代码的一些或全部放入你的程序中好吗?

```
CStr a,b,c;           //定义但没有初始化值。
CStr name("Joe Bloo"); //使用 char * 类型进行初始化。
CStr name2(name);     //使用 CStr 的另一个对象进行初始化。
```

或者也可以将它们合并为一条语句:

```
CStr title("The Big Show");
CStr a,name("Joe"),b(title);
```

每一个声明都调用不同的构造函数。在 C++ 中,你可以为同一个类编写许多不同的构造函数,在每个构造函数中有不同的变量或变量类型。这是函数重载技术的一个例子,它意味着相同的函数名可以用在不同的环境中。

要记住构造函数的名总是与类名相同的。下面是我们需要的三个构造函数的原

型:

```
CStr();           //无初始化参数。  
CStr(char * s);   //使用 char * 类型进行初始化。  
CStr(char & str);  //使用类 CStr 的另一个对象进行初始化。
```

首先让我们先看一看前两个构造函数。第三个构造函数介绍了“&”符号的一个新的使用方法,将在后面的部分“复制构造函数和引用”中讨论。

●—注意—

术语重载(overloading)出现在大量的 C++ 文献中。通常不仅类函数,差不多所有的 C++ 函数都可以被重载。重载意味着你可以重复使用相同的函数名并依靠参数列表的不同来区别它们。参数的个数和类型必须是不同的。(参数的名是无关的。)例如,在 C++ 中,下面的两个函数认为是不同的,而且每个函数必须有自己的定义:

```
Display(int);  
Display(char * );
```

在 C++ 中,调用不同的函数 Display(int)或者是 Display(char *),依靠的是在编译时发现的源代码的参数类型来决定的。

构造函数的两个例子

第一个构造函数最先出现在第五章中。它的任务是初始化一个空的字符串:

```
CStr::CStr() {  
    pData = new char[1];  
    * pData = '\0';  
    nLength = 0;  
}
```

构造函数 CStr(char *) 由一个标准 C 的字符串来初始化一个字符串,它们的长度是相同的。对 CStr 构造函数的基本要求通常是相同的,它们都需要为容纳字符串数据分配内存,初始化数据并初始化成员变量 nLength。

```
CStr::CStr(char * s) {  
    pData = new char[strlen(s) + 1];  
    strcpy(pData, s);  
    nLength = strlen(s);  
}
```

在两种情况下,作用域分辨符(CStr::)都作为构造函数名的前缀,这是因为这些函数的定义是发生在类定义之外的。

这些构造函数在下面的例子中被调用:

```
CStr string1;                //调用 CStr()  
CStr string2("Hello,C++."); //调用 CStr(char*)
```

默认构造函数

在介绍第三个构造函数 CStr(CStr&)以前,先让我们看一类特殊的构造函数——默认构造函数。你会惊奇的发现,它就是我们第一个介绍的没有参数的那个构造函数。

```
CStr::CStr(){  
    pData = new char[1];  
    *pData = '\0';  
    nLength = 0;  
}
```

每个类都有或者应该有一个默认的构造函数。在每一个类中,默认的构造函数是没有参数的最简单的构造函数。C++ 十分关心对构造函数的处理:

- 如果一个类没有声明任何类型的构造函数,那么编译器就提供一个默认的构造函数。这是一个隐藏的构造函数。它并不复杂,只是简单地把所有的成员变量初始化为 0。在上面提到的 CStr 类中这种做法是不行的,但在较简单的类中却可以这样做。
- 如果你回过头去为类声明添加任何的构造函数,编译器提供的默认的构造函数就自动消失了。

从上面的介绍可以得到下面的结论:你应该总包含一个默认的构造函数,尽管它并不做任何事。这是防御编程技术的一种形式。它保证在你添加其他的构造函数时,默认的构造函数不会消失。(如果你使用编译器提供的默认的构造函数,它是会消失的,但无论你自己编写的构造函数多么简单,它都不会消失。)

● 注意

记住默认的构造函数是通过简化参数得到的构造函数。对于 CStr 类来说,默认的构造函数是 CStr()。

这样的构造函数不会做任何事情。例如：

```
class CHorse{
    BREED horse_breed;
    char * name;
public:
    CHorse();
};
CHorse::CHorse(){}
}
```

拥有一个默认的构造函数的重要性不能被过分的强调。在许多情况下,调用默认的构造函数只是用来定义那些没有初值的变量,就象是声明一个带有未被初始化的成员的对象矩阵一样。

```
CHorse posse[30];
```

当使用不带参数的 new 操作符时,默认的构造函数也会被调用。new 操作符(在前面一章已经介绍过)在 C++ 中是非常重要的而且经常被使用。如果缺少了默认的构造函数,所有的这些情形都会导致一个错误。

● 注意

从对 C++ (一旦加入属于自己的构造函数就要将隐藏的默认构造函数去掉)的行为描述中可以看出,好象编译器试图和你玩一个令人讨厌的恶作剧。但仔细考虑这种行为是有意义的。C++ 要设计的尽可能和 C 兼容,它包括来自于 C 中的一些结构并把它们看成类。

C 语言中没有成员函数和构造函数的概念,但是过分单纯化的类(C 中的结构类型)必须传给 C++ 并能正确的工作。编译器支持默认的构造函数的原因是:处理根本没有成员函数的后向兼容事件。但是,在定义新的类和编写成员函数时,应提供自己的默认构造函数。

复制构造函数和引用

复制构造函数是另外一种非常重要的构造函数。象前面提到的那样,当使用一个对象来初始化同一个类中的另一个对象时,要调用复制构造函数。

```
CStr name("Joe Bloe");
CStr name2(name);           //调用复制构造函数。
```

在第一行代码中,name 被定义并由“Joe Bloe”初始化。C++ 调用构造函数 CStr

(char*)来初始化这个字符串。在第二行代码中,name2 做为 name 的拷贝被初始化。对象 name2 和对象 name 的类型是一样的,而且两者应该分配相同的内容。

告诉如何从一个相同类型的对象处得到一个复制的构造函数就是类的复制构造函数。这种构造函数是非常重要的,因为除了定义变量之外,在下面的情形会自动调用它:

- 当类的对象由数值来传递时。C++ 调用类的复制构造函数来创建变量的一个拷贝,然后将其放入堆栈:

```
void Print...string(CStr str);
```

- 当函数返回类的一个对象,也就是返回值的类型是类时。C++ 调用复制构造函数来创建一个对象返回给函数的调用者:

```
CStr Func1(void);
```

因为这些情况太通常了,所以如果你没有提供一个明确的复制构造函数,编译器就会提供一个隐藏的复制构造函数。但是,编译器提供的复制构造函数是过分单纯化的。它所做的只是完成一个成员到成员的复制。在某些情况下,这样做也足够了。

但成员到成员的复制对于象 CStr 这样的类是不适当的,而且它的结果不尽如人意。图 6-1 说明了当使用这种方法将 CStr 类的对象复制到另一个对象时所发生的结果。

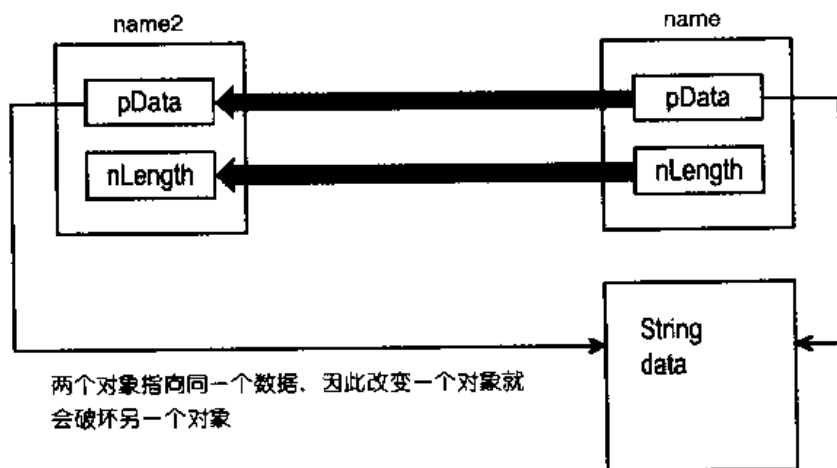


图 6-1

对 CStr 对象的一个简单成员到成员复制的结果

当成员到成员的复制执行后,第二个对象(name2)得到一个指针,这个指针指向第一个对象指向的内存区。在这种情况下,对任何一个对象的改变都会破坏另一个

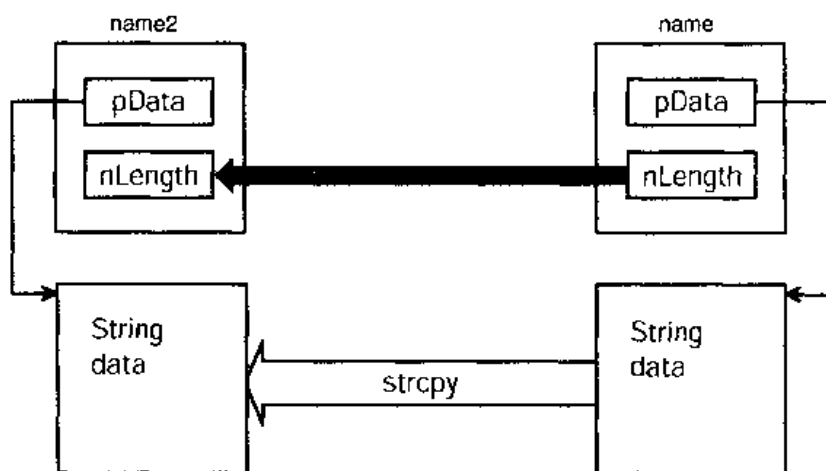
对象的数据。这将引起许多后果,但最明显的是一个对象被删除了。字符串数据被移动并且 `name2` 得到一个什么也不指向的指针。

```
pString = new CStr;  
pString->copy("John Q. Public");  
CStr name2( * pString);  
//...  
delete pString;           //去除字符串数据。  
puts(name2.get());        //错误! name2 有一个非法的 ptr 指针。
```

复制的基本想法是当原始的对象丢失或被破坏后,这个对象的拷贝仍然存在并且是可用的。但当使用成员到成员的复制时,是不会发生这种结果的。当复制构造函数被调用时(例子中的第三行),它初始化 `name2.pData` 的指针,将其指向 `pString->pData` 指向的地址。当 `* pString` 被删除时,这个内存块就被释放了。但 `name2.pData` 仍指向这个已经不合法的内存块。

很明显,复制构造函数应该做构造函数 `CStr(char*)` 所做的事:分配一个新的内存区然后产生一个字符串数据的物理拷贝。这样的复制构造函数是容易编写的,因为它与 `CStr(char*)` 差不多完全一样。但是,首先你要理解引用操作符(`&`)是什么和在复制构造函数中怎样使用它。图 6-2 举例说明了一个正确的复制构造函数。

图 6-2
复制构造函数
对于 CStr
的一个正确
的实现



引用:使用地址操作符(&)的一个新方式

为了编写复制构造函数,必须使用引用类型。这需要在声明中使用地址操作符(`&`)。

在一个简单的变量声明中,符号“&”表明一个变量将做为另一个变量的别名来使用。

```
int a;  
int &b = a;           //b 是 a 的别名  
a = 10;  
b++;                 //a 也加 1
```

在这个例子中,符号“&”用来把 b 当做 a 的别名。上面的代码在功能上等价于下面使用指针的例子:

```
int a, *b;  
b = &a;               //*b 是 a 的别名  
a = 10;  
(*b)++;              //a 也加 1
```

仔细比较这两个例子,你会发现一个重要的不同:当 b 在第一个例子中做为一个引用类型(&)来定义时,在源文件中它不被当做一个指针来对待。编译器可以将 b 当做一个指针来实现,但 C++ 的语法掩盖了这个事实,这使得 b 看起来象一个普通的整数。

● 注意

尽管声明一个引用和得到一个地址都要使用符号“&”,但它们是两个不同的行为。因为在 C++ 中有三或四种对“&”不同的使用方式,所以很容易混淆它。除了用来声明引用和得到地址外,在二进制操作中,符号“&”还可以用做位逻辑运算符 AND。

你可以使用“&”符号简单地为变量创建别名,但最重要的使用是发生在函数调用时。考虑 CStr 的复制构造函数的原型;

```
CStr(CStr & str);
```

这个构造函数只有一个参数 str,对于 CStr 它是一个引用类型。这将告诉编译器此参数可以做为指针来传递,这样做的效率较高,但是随后对 str 的使用不能再指针。

简单地说,为了产生一个对象的拷贝,复制构造函数得到原始对象的一个指针并使用这个指针来访问成员。源对象通过引用来传递,就象在 BASIC、Pascal 或者 FORTRAN 中所做的一样,并没有使用明显的指针语法。

考虑一下就会发现,在这里需要用引用来传递源对象,因为这样做,复制构造函

数得到的就是一个指针而不是对它自己参数(str)的一个简单的复制。如果复制构造函数不得不在它创建一个拷贝之前得到一个对象的拷贝,那么它就不得不在开始之前调用它自己。这样做将弄巧成拙,结果是一个无限的循环。

编写复制构造函数

所有你要记住的有关引用操作符的是:把参数当作是一个值而不是指针,尽管你知道它是由引用来传递的。CStr 的一个正确的复制构造函数是:

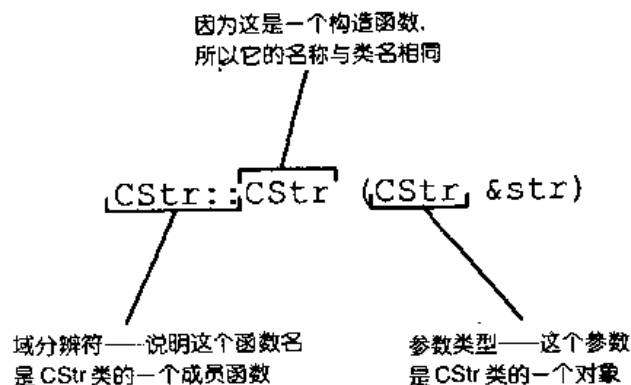
```
CStr::CStr(CStr &s){  
    char* sz = s.get();  
    int n = s.getLength();  
    pData = new char[n + 1];  
    strcpy(pData, sz);  
    nLength = n;  
}
```

这个函数与构造函数 CStr(char*)是相似的。主要的区别是复制构造函数必须首先得到字符串和字符串长度数据。

因为类名 CStr 被使用了三次,尽管每次都有轻微的差别,但函数定义的标题还是很容易混淆。图 6-3 对三次使用 CStr 分别作出了分析和解释。它们三者是紧密相关的,但在语法中它们履行不同的职责,分别是作用域分辨符、名字和参数类型。

图 6-3

复制构造函数
定义的语
法



const 关键字

使用引用传递源对象的一个原因是使函数能够改变参数的值。如果你曾经学过 BASIC、Pascal 或者 FORTRAN 的话,毫无疑问你知道这是使用引用传递参数的一个主要原因。

但是复制构造函数最后将要做的事情是改变它的参数。在这里,用引用来传递参数的目的是为了效率。人们愿意使用引用来传递参数,但要阻止复制构造函数改变参数的能力。

幸运的是,C++ 提供了这样的一个技术,在变量声明的前面放置一个 const 关键字:

```
class CStr{
    CStr(const CStr &s);
    //...
};

//使用 const 类型参数的复制构造函数
//除了参数的声明之外,其余的代码都没有改变

CStr::CStr(const CStr &s){
    char * sz = s.get();
    int n = s.getlength();
    pData = new char[n + 1];
    strcpy(pData,sz);
    nLength = n;
}
```

现在复制构造函数就不能改变对象 s 中成员的值了。任何试图向 s 或它的数据成员赋值的行为都会被编译器做为错误来标记。s 只能传递给具有一个 const 参数的函数。使用 const 参数是编写复制构造函数首选的方式,因为它阻止一个导致错误出现的原因。

但现在又出现了一个问题:函数的前两行通过参数 s 来产生调用。编译器如何知道这些函数调用不会破坏 s 呢?解决的办法是将 get 和 getlength 函数声明为 const 函数。这样做意味着这些函数有一个不改变任何数据成员的约定,用它们调用一个 const 对象就是安全的了。

```
int getlength(void) const {return nLength;}
char *get(void) const {return pData;}
```


相对于它的价值来说,这个额外的工作使 `const` 似乎更麻烦了。但仔细考虑一下,你会发现所有的规则都有它们的意义。要想了解 `const` 关键字的优点,请查看第十三章。

其他构造函数的例子

另一个例子让我们考虑以下第五章介绍的 `CTemp_point` 类。这个类产生的对象记录了三维空间网格上的一个点和在这个点上的属性值。下面在第五章曾经提到的代码中加入几个构造函数:

```
class CTemp_point{
    int x,y,z;
    double temp;
public:
    CTemp_point();
    CTemp_point(int xx,int yy,int zz);
    CTemp_point(const CTemp_point& pt);

    void set_point(int xx,int yy,int zz);
    void get_point(int *xx,int *yy,int *zz);
    void set_temp(double new_temp);
    double get_temp(void);
};
```

这个类定义中的第一个和第三个构造函数分别是默认构造函数和复制构造函数。`CTemp_point` 类比 `CStr` 类简单得多,它没有对内存的动态分配、再分配或整理,因此在这种情况下,默认构造函数是没有什么事可做的。但是,根据前面解释的原因,类定义中必须包含一个默认构造函数。否则,C++ 就认为这里没有默认构造函数,它会阻止你使用 `new` 操作符或者声明未被初始化的该类的对象。

默认的复制构造函数是容易编写的:

```
CTemp_point::CTemp_point() {
    ...
}
```

接下来的构造函数有趣一些。它有三个参数,构造函数是可以有许多个参数的。在这种情况下,成员变量 `temp` 仍然设置为 0。

```
CTemp_point::CTemp_point(int xx,int yy,int zz){
    x = xx;
    y = yy;
```

```
z = zz;  
}
```

最后一个构造函数通过复制它的参数 `pt` 的成员变量来初始化一个对象。这个参数是 `CTemp_point` 类的另一个对象。可以在程序中忽略这个构造函数而不会产生任何影响,这是因为它和编译器提供的复制构造函数的行为是一样的(完成一个成员到成员的复制)。

```
CTemp_point::CTemp_point(const CTemp_point& pt){  
    x = pt.x;  
    y = pt.y;  
    z = pt.z;  
    temp = pt.temp;  
}
```

下面的三个声明分别调用上面的三个构造函数:

```
CTemp_point pt1;  
CTemp_point pt2(100,101,200);  
CTemp_point pt3(pt2);
```

C++ 如何调用构造函数

C++ 可能使用你不希望的方式调用构造函数。考虑下面的声明:

```
CStr name1("John Doe");  
CStr name2 = name1;  
CStr name2 = "Jane Doe";
```

明显可以看出,第一个声明导致一个对 `CStr(char*)` 构造函数的调用。更令人惊奇的是,第二个声明也调用复制构造函数。这是不明显的。你可能认为第二个声明通过调用默认的构造函数来创建 `name2`,然后再执行一个从 `name1` 到 `name2` 的分配。然而,它并不是这样的。下面的两个声明不仅在最终的结果还是调用构造函数的方式上都是等价的。

```
CStr name2 = name1;  
CStr name2(name1);
```

在 C++ 中,尽管赋值和初始化看起来十分地相似,但它们是有着本质的区别的(二者没有任何交迭)。在某些情况下,等号(=)总是代表赋值。然而,等号(=)还代表初始化,它将导致对适当的构造函数的调用,而不是代表赋值操作符。

在第七章中你会发现,赋值(=)是一个你可以定义它的行为的操作符。它与复制构造函数是不同的。有人可能希望复制构造函数和赋值操作符做同样的工作,但这并不是所有的情况下都正确。(两者主要的区别是复制构造函数不能假设一个对象先前已被初始化了,因此它不得不做更多的工作。)

同样,最后一个声明调用构造函数 CStr(char *):

```
CStr name3 = "Jane Doe";
```

调用构造函数的另一种情况是:在将其它数据类型(此时为 char *)转换为当前的类的数据类型(此时为 CStr)时要使用它。做为一个转换函数,当满足一定的条件时构造函数是有两种用途的。需要满足的条件是:构造函数只有一个参数,且这个参数的类型与类的类型是不同的。换句话说就是,一个 class(type)形式的构造函数知道如何根据 type 的形式进行转换。

构造函数 CStr(char *)告诉编译器如何将一个 char * 形式的字符串转换为一个 CStr 类的对象。因此,无论何时需要一个 CStr 类的对象,都可以使用一个 char * 形式的字符串来代替它,并且编译器会为你转换它。例如:

```
CStr make_uppercase(CStr s);    //原型
```

```
CStr string1;  
string1 = make_uppercase("cia /fbi");
```

代码的最后一行连续的调用构造函数 CStr(char *) 来从字符串“cia /fbi”创建一个 CStr 类的对象。然后这个对象被传递到 make_uppercase 函数中去。

总结:构造函数的重点

在最简单的情况下,构造函数只不过是一个便利的初始化函数,但是构造函数有许多微妙之处。主要的要点总结如下。

重载构造函数

构造函数的名总是与类名相同。因此对于 CStr 类来说,每一个构造函数的名都是 CStr。构造函数没有返回类型,甚至都不能将其定义为 void 类型。

象大多数的 C++ 函数一样,构造函数也可以被重载。这意味着你可以重复使用同一个函数名,只要它们的参数列表是不同的(参数的个数和类型都应该是不同的)。这一章中重载的重要性是,你可以创建许多构造函数并且每个函数都可以由不同类型的参数来初始化。

默认的构造函数

```
class();
```

一个非常重要的构造函数是默认构造函数,对于任何给定的类,默认构造函数是没有参数的。当定义了一个没有初始化的对象、创建一个没有初始化的对象矩阵或者使用不带参数的 new 操作符时,默认构造函数就会被调用。编译器是有一点欺骗性的,因为它为你创了建一个隐藏的默认构造函数,但是如果你添加自己的构造函数,那么这个隐藏的默认构造函数就自动消失。这就是为什么要编写自己的构造函数的原因,虽然它并不做太多的工作。

复制构造函数

```
class(class&);
```

对于每个类来说,复制构造函数是另一个非常重要的构造函数。它最明显的用处是用相同类型的一个对象来初始化另一个对象。但是当象函数中的参数那样使用值来传递类型或者使用类型做为一个函数调用的返回值时,也会调用这个构造函数。当你没有提供复制构造函数时,编译器会提供一个默认的复制构造函数。但是要记住的是,编译器提供的复制构造函数完成的是成员到成员的复制,对于象 CStr 这样的类来说,这是不够的。

复制构造函数的参数使用引用操作符(&),就象 BASIC 或者是 FORTRAN 语言那样使用引用来传递参数,它没有使用指针语法。

初始化和转换

对象的定义和初始化经常要调用构造函数,甚至于在使用等号(=)时也要调用构造函数。在这种情况下,等号(=)与赋值时使用的等号的意义是不同的。

```
CStr string1 = "Hi world?" //调用 CStr(char*)
```

调用构造函数的另一种情况是类型转换。对于给定的构造函数 CStr(char*),你可以传递一个 char* 参数到一个函数中需要 CStr 的地方,编译器会为你执行转换。

在介绍完类和构造函数之后,我们将要考虑的是 C++ 中十分有趣又节省程序运行时间的部分:操作符重载。

第 七 章

类的运算(操作符重载)

C++ 的一个非常有趣的功能是可以进行操作符的重载,即用户可以针对自己的类来定义特定的操作符运算。除了非常少的几种特例外,用户几乎可以自定义所有的 C++ 操作符。

有了操作符重载,用户使用非常简洁的语句就可以扩展 C++ 的功能。例如,你可以定义一个字符串类型和针对此类型的“加”操作:

```
CStr first("Norma"), last("Jean");  
CStr str = first + " " + last;
```

操作符功能的定义过程直观明了。在每一个有操作符出现的表达式里,都有一个相应的函数调用来计算该表达式的值,而这一函数又是根据操作符和操作数的类型来决定的。

完全有理由相信,在你使用 C++ 的过程中,操作符重载会成为你最感兴趣的一部分。

基本表达式

当你刚开始使用操作符函数时,也许会遇到一些小麻烦,但操作符函数的基本表达式非常简单。对于某一操作符@,对应的操作符函数名称为:

```
operator@
```

因此 +、-、*、/ 的操作符函数名称分别为:

```
operator +  
operator -  
operator *  
operator /
```

上面的名称定义也许简单得让人难以置信,你一定会怀疑此操作符所做的工作远远不限于此。首先,该函数的参数是什么?这一问题的答案取决于以下几个因素:

- 函数所进行的操作是双操作数的(binary)还是单操作数的(unary)?(有些操作符,如减号/负号,在两种情况下都可以使用。)
- 如果所进行的操作是双操作数的,是否支持操作对象出现在操作符任何一侧的情况?例如,你是否既允许 `string + CStr_object`,也允许 `CStr_object + string`? 如果答案是肯定的,就需要使用友元函数(使用了 `friend` 关键字,类就可以访问这些全局函数)。

说到这里情况就变得复杂了,这一章的后面会陆续向你解释 `friend` 函数的一些较深奥的涵义。总之,在任何情况下,下面的语法结构在加法函数里都是合适的:

```
class CStr {
//...
    friend CStr operator+ (CStr str1, CStr str2);
    friend CStr operator+ (CStr str, char * s);
    friend CStr operator+ (char * s, CStr str);
};
//Note that the friend keyword isn't needed in the
//definition below

CStr operator+ (CStr str1, CStr str2) {
//...
}

CStr operator+ (CStr str, char * s) {
//...
}

CStr operator+ (char * s, CStr str) {
//...
}
```

从上面的例子里你可以看到函数重载和操作符重载的紧密联系。名称为 `operator+` 的函数可以有多个,而区分这些函数则是根据它们参数的不同。那些在参数表里有一个或多个 `CStr` 类型的 `operator+` 函数就定义了该操作符对 `CStr` 对象的操作。

编写加法(+)操作符函数

下面的定义可以实现两个 `CStr` 对象的相加,每一个参数在表达式 `str1 + str2` 里

都代表一个操作数。

```
CStr operator+(CStr str1, CStr str2) {  
    CStr new_string(str1);  
    new_string.cat(str2.get());  
    return new_string;  
}
```

CStr 出现在函数的起始位置,因此该函数的返回值为 CStr 类型。在 operator + 函数的第一行里首先声明了一个新的 CStr 对象,并且最终由它将结果返回。

```
CStr new_string(str1);
```

以上语句调用了复制构造函数 new_string 进行初始化,获得了字符串 str1(即第一个参数)的硬拷贝。下一条语句把第二个参数 str2 连接到 new_string 的末端,其结果即为我们所要的两个字符串的和。

```
new_string.cat(str2.get());
```

最后一步是把得到的新字符串作为运算结果返回。

```
return new_string;
```

该语句为 new_string 在其作用域内创建一个拷贝,而 new_string 则在函数调用结束后被销毁。正如第六章所介绍的那样,编译器自动激活复制构造函数 CStr(CStr&)来建立返回值的拷贝,并且把该对象存储在堆栈里。这样一来,主调函数就可以顺利地访问到它。

对于其它象 CStr 一样复杂的类,如果复制构造函数编写得有问题就会导致运算结果的返回失败,而编译器提供的隐含复制构造函数将执行一个直接的、成员到成员的拷贝,结果也是出错。正是因为这个原因,本书的编排上首先在第六章介绍复制构造函数,而在后面才介绍操作符的重载。

操作符函数的调用过程

当 C++ 遇到一个诸如 str1 + str2 的表达式时,它就将其转换为相应的函数调用,图 7-1 显示了它们之间的调用关系。如果此函数不存在,系统将在编译的时候给出错误报告,提醒编程者该操作尚未定义。

下面的代码是字符串加法操作的实例。其中每一条语句定义了一个对象,最后一条语句则进行了一次加法操作(调用 operator + 函数),将所得两字符串的和(CStr 类型)传递给复制构造函数,并最终由它来初始化 CStr 对象。


```
CStr first("Archie ");
CStr last("Leach");
CStr name(first + last);
```

图 7-1

将表达式转
换为函数调
用

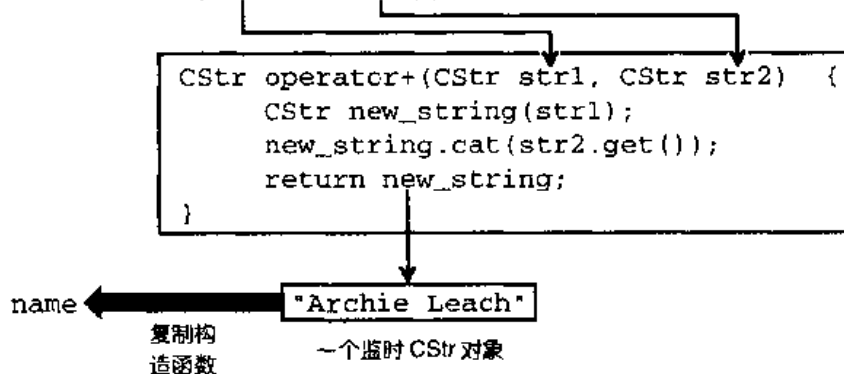
```
str1 + str2
    ↓
operator+(str1, str2)
```

图 7-2 显示了加法操作的运算情况。表达式 `first + last` 的结果被存储在一个独立的 `CStr` 对象中,而正是由 `operator+` 函数创建了此对象并把它以 `CStr` 类型返回给主调函数(其值为“Archie Leach”)。另外,该对象是一个临时对象,当程序不再需要它时,编译器就会把它销毁。

```
CStr first("Archie ");
CStr last("Leach");
CStr name(first + last);
```

图 7-2

`operator+` 函
数的调用情
况



还需注意的问题(其它加法函数)

其它针对 `CStr` 类操作的 `operator+` 函数酷似在前两节里出现的第一个 `operator+` 函数。

```
CStr operator+(CStr str, char * s) {
    CStr new_string(str);
    new_string.cat(s);
    return new_string;
}
```

```
!

CStr operator+ (char * s, CStr str) {
    CStr new_string(s);
    new_string.cat(str.get());
    return new_string;
}
```

所有这些 `operator+` 函数都很相似。在各种情况下,计算字符串和的操作基本上都一样:

1. 建立一个新的 `CStr` 对象,用第一个操作数的值对其进行初始化。
2. 将第二个操作数的内容添加在 `CStr` 的尾部。
3. 将得到 `CStr` 对象返回。

友元的使用

在编写双操作数操作符函数时,经常需要使用关键字 `friend`(在这种情况下,一个双操作数操作符只能用在两个操作数之间,如表达式 `X + Y` 里的 `+`)。

一般来说,全局函数不能访问类的私有成员,而 `friend` 关键字超出了这一界限,它允许对类里所有成员的访问。例如,函数 `setbk` 是类 `CBook` 的友元,则 `setbk` 就可以访问类 `CBook` 的私有成员(`a`、`b` 和 `c`)。

```
class CBook {
    int a, b, c;
    friend void setbk(CBook bk, int x, int y, int z);
};

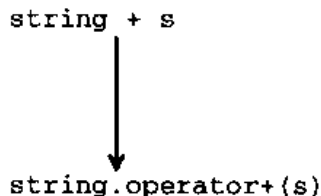
void setbk(CBook bk, int x, int y, int z) {
    bk.a = x;
    bk.b = y;
    bk.c = z;
}
```

`friend` 关键字可以用来编写如 `operator+ (char *, CStr)` 等函数,你也可以不使用 `friend` 关键字而以成员函数的方式来编写操作符函数,但这种函数缺省地认为类的对象出现在操作符的左侧。例如,函数 `operator+ (CStr, char *)` 可以写成如下的成员函数形式:

```
class CStr {  
    //...  
    CStr operator+(char * s);  
}  
  
CStr CStr::operator(char * s){  
    CStr new_string( * this);  
    new_string.cat(s);  
    return new_string;  
}
```

函数定义部分的第一句话用 * this 指向的对象初始化一个新字符串。* this 的值是左操作数。为了增进对这种代码工作方式的理解,图 7-3 给出了把加法操作转换为函数调用的过程。

图 7-3
加法操作到
成员函数调
用之间的转
换



使用这种方法,加法可以转换成一个针对左操作数(CStr 对象)的成员函数的调用。在 C++ 里, this 关键字是一个指向当前对象的指针,* this 则代表当前对象。因此在第一条语句里可以用 * this 的内容对 new_string 进行初始化。

```
CStr new_string( * this);
```

在本书的第十三章里,我们将进一步介绍 this 关键字。

在任何情况下,用这种方法编写的操作符函数将限制你只能在操作符左侧使用类对象,并且只有编写另一个非成员函数才能计算下式的值(操作数的顺序颠倒过来):

```
s + string
```

正是出于这种原因,双操作数操作符一般以友元函数实现而不以成员函数方式实现(因为它们不是成员函数,要想访问操作对象的私有数据,就必须要被定义为对象类的友元),也就是本章开始时使用的方法:

```
class CStr {  
    //...
```

```
friend CStr operator+ (CStr str1, CStr str2);  
friend CStr operator+ (CStr str, char * s);  
friend CStr operator+ (char * s, CStr str);  
};
```

请注意,赋值操作符是一个特例,它必须以成员函数而非友元函数的方式实现,在下一节将对此加以介绍。

赋值函数的编写

赋值号(=)是一个特殊的操作符,如果你没有对它编写程序,编译器将提供一个隐含函数来实现它的功能。C++里有这样一个普遍的规则,即同种类型对象之间的赋值总是被支持的,正如标准C里支持结构之间的直接赋值一样。因此,这样的语句都是合法的:

```
CStr string1, string2;  
//...  
string1 = string2;           // Assign value of string2 to  
                             // string1
```

●—注意

用户可以自定义一个赋值操作符函数并使之成为类的私有成员,这样就可以避免在赋值操作时对所有数据成员赋值。

然而,对于诸如CStr这样的类,由于它的复制构造函数使用了动态内存分配,编译器提供的赋值操作就显得不够了。正如第六章所指出的那样,它仅仅执行一个简单的成员到成员的拷贝,其结果是不正确的。

因此,在编写象CStr这样的类时,用户必须仔细设计赋值操作符的工作方式,它所对应的操作符函数必须是类的成员,并且在类的声明里加入如下的函数原形:

```
class& operator=(const class &arg);
```

参数名称(arg)在函数声明里是可选的,其相应函数定义的语法结构为:

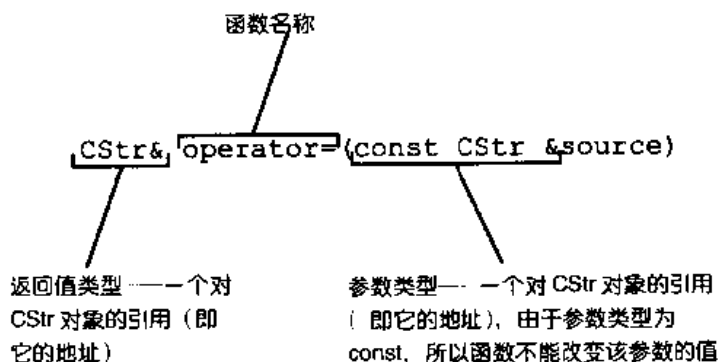
```
class& class::operator=(const class &arg) {  
    statements  
}
```

在CStr类里,赋值操作符函数可以有如下的函数原形。在此种声明方式下,用户唯一可以选择的只有参数名称。

```
class CStr {
//...
    CStr& operator=(const CStr &source);
};
```

由于编写赋值操作符函数使用了很多 C 里面没有的表达式,因此对于 C++ 的初学者来说显得比较困难。在本章和前面的章节里我们已经陆续介绍了这些表达式,翻看一下这些内容会对你理解后面的程序有所帮助。图 7-4 分析了从 CStr 到 CStr 的赋值函数的表达式。

图 7-4
对赋值操作
符函数声明
的分析



赋值函数的定义

如果你已经掌握了 C++ 的语法,那么赋值函数的编写将会非常容易。下面的例子是这一函数的最短定义:

```
CStr& CStr::operator=(const CStr &source) {
    cpy(source, get());
    return * this;
}
```

这个函数虽然很短,但它的功能已经足够了。我们利用它就可以很好地实现赋值操作。事实上,对于这么短的函数,它更适合作为内联函数,即它的定义可以出现在 CStr 的声明部分。

```
class CStr {
//...
    CStr& operator=(const CStr &source)
    {cpy(source, get()); return * this;}
};
```

```
};
```

在上面所示的两种情况里,函数体的定义都只有短短的几条语句,并且只执行了简单的工作。首先,第一条语句调用了 `cpy` 函数,它是一个业已存在的函数,执行拷贝操作(在第五章里我们已经介绍了如何编写 `cpy` 函数)。第二条语句把对象本身作为结果返回,即待赋值的 `CStr` 对象。`return *this` 语句的作用等同于:

```
Return myself!
```

图 7-5 显示了 C++ 如何处理一个赋值表达式。请注意,该函数通过 `str1` (左操作数)被主调函数调用(下一节里这一点显得尤为重要,我们将在那里详细探讨关键字 `this`)。

图 7-5
赋值表达式
的转换

```
str1 = str2  
      ↓  
str1.operator=(str2)
```

this 指针及其用法

在成员函数中,关键字 `this` 是一个指向当前对象的指针(这里所说的当前对象,是指主调函数通过该对象调用某一函数)。每次用户调用一个成员函数时,C++ 都把一个隐含的 `this` 指针传递给函数的参数,如图 7-6 所示。

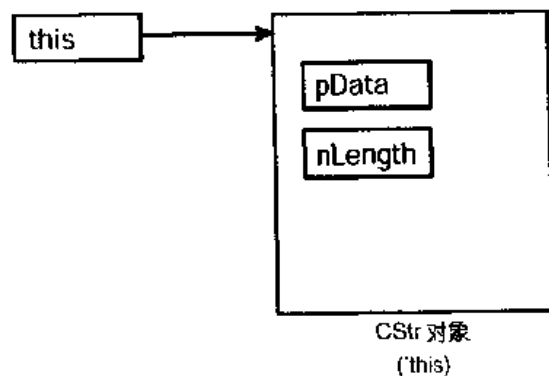


图 7-6
CStr 类里
“this”指针
的使用

在成员函数的定义式里,`this` 指针把对数据成员的引用自动转换为对对象成员的引用,如表 7-1 所示。许多成员函数从不使用 `this` 指针,这是因为在函数内部已经

隐式地使用了这一指针。

表 7-1 "this"指针的隐式使用

数据成员的引用	等同于
pData	this->pData
nLength	this->nLength

如果你要显式地指向某个对象,比如赋值函数,就需要使用 this 关键字了。在 C 和 C++ 里赋值表达式一般都要求返回运算结果,即左操作数。因此,赋值语句还支持如下的使用方式:

```
int      x, y, z;  
CString  stringA, stringB, stringC;  
x = y = z = 0;  
stringA = stringB = stringC = "hello!";
```

当字符串"hello!"赋给 stringC 之后,表达式 stringC = "hello!"又把 stringC 的值赋给 stringB。为支持这种语法规则,赋值函数操作的最后一步必须把左操作数的值作为结果返回。

用户还应该记住赋值函数是作为左操作数的成员函数被调用的,而左操作数就是当前对象(因为函数调用是通过该对象实现的)。因此,如果你一直按照本书的程序设计思路来编程的话,在实现对对象进行赋值操作的最后一步就要把对象本身作为结果返回。在 C++ 里实现这一目的的语句是:

```
return * this;
```

this 关键字是指向当前对象的指针,因此 * this 就代表对象自己。

赋值操作里的引用类型(&)

在赋值操作符的定义式里需要使用引用类型。引用操作允许用户在对象之间传递数据而不必创建多余的拷贝,它也不必使用指针表达式。

图 7-7 显示了两个 CStr 对象的赋值操作,其返回值是一个指向左侧对象的指针(源代码里是把一个引用返回,而编译器的实际处理是把一个指针返回),此对象将成为下一次赋值操作的参数。

编写类型转换函数

迄今为止,CStr 类已经变得非常灵活易用。例如,你可以把它们连接起来进行

赋值,也可以用表达式对它们赋值,就象 BASIC 里所使用的字符串一样。

```
CStr first = "Norma";
CStr middle = "Jean";
CStr last = "Baker";
CStr name;
name = first + "" + middle + "" + last;
```

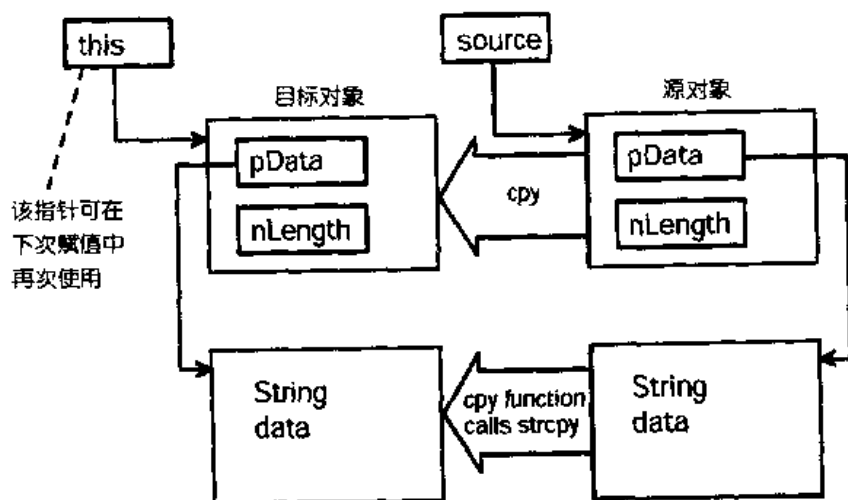


图 7-7
CStr 的赋值
操作

在我们停止对 CStr 例子的讨论之前,还应该在最后赋予给它另外一个非常有用的功能,使 CStr 类在所有使用 `char *` 的场合下都可以正常运转。例如,标准库函数里有许多参数为 `char *` 的函数,puts 便是其中之一。如果能够把一个 CStr 对象传递给它并打印其内容(如同使用 `char *` 参数一样),那将会给实际操作带来非常大的方便。

```
#include <stdio.h>
```

```
CStr warn = "This is your final warning";
puts(warn);    // puts takes a char * argument
```

在 C++ 里,你可以写一段类型转换函数来实现这一功能。这个函数告诉编译器如何把一个类的对象转换为另一种类型。如第六章所指出的,构造函数本身也执行一个类型转换操作。编译器究竟是调用类型转换函数还是构造函数取决于类型转换的方向(图 7-8)。对任何特定的类,类型转换函数处理向外的转换(转变为另外一个类型),构造函数执行向内的转换(从其它类型转变为此类型)。

在类的声明部分,类型转换函数作为类的成员按下面的方式被声明,它既没有返

返回值类型也没有参数。

```
operator type ()
```

把一个 CStr 对象转换为 char * 类型非常容易,函数只需要把数据指针提取出来即可。下面的代码出现在 CStr 类的声明部分,它给出了对类型转换函数的声明。因为函数定义部分非常短小,所以这里使用的是内联函数。

```
class CStr {  
    //...  
    operator const char * () {return get();}  
};
```

这里的转换类型是 const char * 而不是 char * 类型,它和普通的 char * 类型没有太大差别,只是 const 关键字保护了字符串在转换过程中不能被随便更改。当然,你可以去掉 const 关键字,但这时候就应该注意防止数据被破坏。

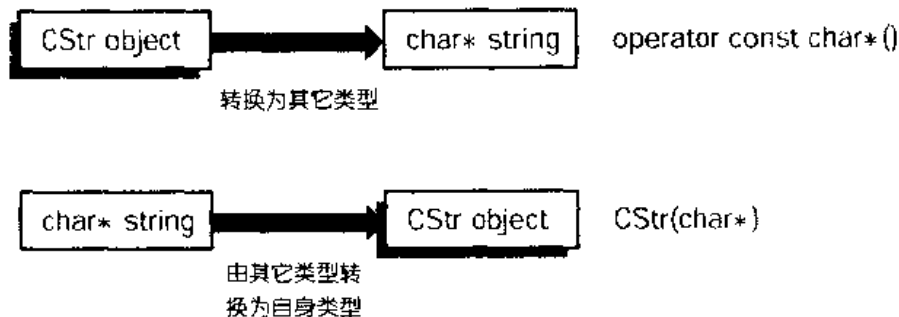


图 7-8
类型转换函数和构造函数的对比

标准 C++ 库里的大多数字符串处理函数(包括 puts)都可以接受 const char * 参数,而其中的一个例外是向字符串的写入操作,此时只能接受 char * 参数。由于 CStr 里使用了内存分配功能,如果允许通过指针直接修改字符串里的内容将导致危险的后果。

CStr 类的小结

迄今为止,我们在 CStr 类上已经花费了大量的篇幅,在上面的三章里我们一步一步地把它建立起来,同时,我们在编写程序时也接触到了很多 C++ 的核心理论。

读者也许会提出疑问:在哪种情况下值得我们去专门编写一个类呢?要给出这个问题的答案我们首先要权衡一下这几个条件。如果你仅仅用到了某一类的很少的几个实现,并且在其它程序里不再使用该类,那么就不必单独创建。但如果你已经成功建立并调试好某个类,那么在今后的许多程序里你都可以重复使用它。作为 C++

语言的扩展,CStr 的使用简单易行,只要你按照下面的步骤操作,就可以在任何程序里定义 CStr 对象:

- 把相应的头文件 cstr.h 包含在每一个使用 CStr 的程序模块里。
- 如果程序的建立和连接用到 cstr.obj 文件(它是 CStr 成员函数的实现经编译后所得的目标文件),你甚至可以把 cstr.obj 添加到标准库里。

下面是 cstr.h 文件的最终内容,它包含了到现在为止编写的所有 CStr 类成员的声明:

```
// CSTR, H - Declaration of the CStr class

class CStr {
    char * pData;
    int nLength;
public:
    CStr();          // Constructors
    CStr(char * s);
    CStr(const CStr & str);
    CStr();          // Destructor

    char * get(void) const {return pData;}
    int getlength(void) const {return nLength;}
    void cpy(char * s);
    void cat(char * s);

    friend CStr operator+ (CStr str1, CStr str2);
    friend CStr operator+ (CStr str, char * s);
    friend CStr operator+ (char * s, CStr str);

    CStr& operator= (const CStr & source)
        {cpy(source.get()); return * this;}
    operator const char * () {return get();}

};
```

在 CStr 的声明部分里有四个以内联函数形式定义的成员函数。由于它们已经在这里进行了定义,所以在 cstr.cpp(类的实现部分)里不必再有相应的代码。文件 cstr.cpp 里是其它函数的定义。在使用 CStr 类的工程文件里,必须要编译这段代码并且把生成的目标文件连接进来。

```

// CSTR.CPP - implementation of CStr class

#include "cstr.h"
#include <string.h>

CStr::CStr() {
    pData = new char[1];
    *pData = '\0';
    nLength = 0;
}

CStr::CStr(char *s) {
    pData = new char[ strlen(s) + 1];
    strcpy(pData, s);
    nLength = strlen(s);
}

CStr::CStr(const CStr &s) {
    char* sz = s.get();
    int n = s.getlength();
    pData = new char[n+1];
    strcpy(pData, sz);
    nLength = n;
}

CStr::~CStr() {
    delete [] pData;
}

void CStr::cpy(char *s) {    // Copy string arg.
    int n;

    n = strlen(s);
    if (nLength != n) {
        if (pData)
            delete [] pData;
        pData = new char[n+1];
        nLength = n;
    }
    strcpy(pData, s);
}

void CStr::cat(char *s) {    // Concatenate string arg.
    int n;

```

```

char * pTemp;

n = strlen(s);
if (n == 0)
    return;
pTemp = new char[n + nLength + 1];
if (pData) {
    strcpy(pTemp, pData);
    delete [] pData;
}
strcat(pTemp, s);
pData = pTemp;
nLength += n;
}

CStr operator+ (CStr str1, CStr str2) {
    CStr new_string(str1);
    new_string.cat(str2.get());
    return new_string;
}

CStr operator+ (CStr str, char * s) {
    CStr new_string(str);
    new_string.cat(s);
    return new_string;
}

CStr operator+ (char * s, CStr str) {
    CStr new_string(s);
    new_string.cat(str.get());
    return new_string;
}

```

另一个类操作符的实例

在介绍完操作符重载之前,我们再看一下另一个使用操作符函数的例子:在第五章里提到的 CTemp_point 类。这个类定义了一个三维网格里的点和一个属性参量。与 CStr 类相比,它的一些成员函数更容易实现,这是因为 CTemp_point 类不牵涉到内存分配问题。

```

class CTemp_point {
    int x, y, z;
    double temp;

```

```

public:
    CTemp_point();
    CTemp_point(int xx, int yy, int zz);
    CTemp_point(CTemp_point &pt);

    void set_point(int xx, int yy, int zz);
    void get_point(int *xx, int *yy, int *zz);
    void set_temp(double new_temp);
    double get_temp(void);

    CTemp_point operator+ (double temp_diff);
    CTemp_point operator-(double temp_diff);
    CTemp_point& operator= (const CTemp_point &pt);
};

```

在设计一个类时,决定要支持何种运算是一个非常重要的环节。对于 CStr 类,把加法操作定义为两个字符串结合起来生成另一个字符串是合情理的。而对于 CTemp_point 类,加法运算就不那么好设计了。在坐标系里两个点的相加表示什么呢?

CTemp_point 类只支持两种简单的操作,加和减都是针对于属性数据成员 temp 的,而网格的坐标不受其影响。因此这里没有定义两个点的加法操作。

```

pt1 = pt2 + 33.3;    // Ok; addition to flt pt defined.
pt2 = pt2 + 1;       // Ok; 1 can be converted to floating pt.
pt2 = pt1 + pt3;     // ERROR! Operation not defined.

```

如 CStr 的加法函数一样,这些操作符函数把一个对象作为返回值。编程过程遵循同样的步骤:创建一个新的对象,设定一些值,把对象作为结果返回(最后一步隐式地调用了类的复制构造函数)。

```

CTemp_point CTemp_point::operator+ (double temp_diff) {
    CTemp_point new_pt(*this);
    new_pt.temp += temp_diff;
    return new_pt;
}

CTemp_point CTemp_point::operator-(double temp_diff) {
    CTemp_point new_pt(*this);
    new_pt.temp -= temp_diff;
    return new_pt;
}

```

下面我们还要加上赋值操作符函数。在这种情况下,由于编译器所提供的赋值操作可以满足需要,所以该函数并不是必需的。这里专门编写一段代码只是作为例子帮助读者理解。下面的程序执行了一个简单的拷贝操作,把右操作数(转换为参数 *pt*)按从成员到成员的方式赋值给左操作数(函数调用的目标对象)。对于几乎所有的赋值函数,最后一行都以返回 `* this` 结束。

```
CTemp_point& CTemp_point::operator= (  
    const CTemp_point &pt) {  
    x = pt.x;  
    y = pt.y;  
    z = pt.z;  
    temp = pt.temp;  
    return * this;  
}
```

操作符重载进阶

对于操作符重载问题的讨论,到现在为止我们还仅仅触及到皮毛。在下面的章节里,我们将总结一下前面的知识,并且通过这一总结,使我们对 C++ 在操作符的处理上有一个整体的认识。

操作符函数的命名

操作符元素的命名方式非常简单,对于任何给定的操作符@,其函数名称为 `operator@`。唯一的例外是类型转换函数,名称为:

```
operator type ()
```

双操作数操作符

编译器在计算形如 `obj1@obj2` 的表达式的值时,若 `operator@` 函数是在 `obj1` 类之内定义的话,那么它将该表达式转换为如下形式的函数调用:

```
obj1.operator@(obj2)
```

如果 `operator@` 函数是在 `obj1` 类之外定义的,编译器就把表达式转换为如下形式的函数调用:

```
operator@(obj1, obj2)
```

要是把该函数声明为类的友元,那它就可以被所有类的成员调用。

使用第二种函数声明方式具有一个优点,无论 *obj1* 出现在操作符的左侧还是右侧都可以执行同样的操作。要支持逆序操作(两个操作数颠倒过来),还要再写一个函数:

```
operator@(obj2, obj1)
```

如果 *obj2* 类的声明出现在其它程序里(例如, *obj2* 属于标准类型 `int` 或 `char*`),就只能使用这种方式来定义函数的操作。

单操作数操作符

当编译器计算形如 *@obj1* 的表达式的值时,如果 `operator@` 函数是在 *obj1* 类里定义的,它就把该表达式转换为如下形式的函数调用:

```
obj1.operator@()
```

如果 `operator@` 函数是在 *obj1* 类之外定义的,编译器就把表达式转换为如下形式的函数调用:

```
operator@(obj1)
```

要是把该函数声明为类的友元,那它就可以被所有类的成员调用。

举一个例子,对于 `CTemp_point` 类定义的单操作数减(−)操作实现对数据成员 `temp` 符号的取反操作:

```
class CTemp_point {  
    //...  
    CTemp_point operator-();  
};  
  
CTemp_point CTemp_point::operator-() {  
    CTemp_point new_point( * this);  
    new_point.temp * = -1;  
    return new_point;  
}
```

对于除类型转换函数之外的所有操作符函数,返回值的类型必须显式地加以声明,编译器不对返回值设置任何缺省类型。例如,你可以对类 `A` 定义一个操作符函数,它可以返回你所需要的任何类型的对象。

赋值操作符

赋值操作符的目的是实现在同一个类里不同对象之间的赋值。尽管赋值符号大体上同其它双操作数操作符一样,它还是有一些特殊的限制。例如,赋值函数必须被定义为成员函数,对它的声明为:

```
class C operator=(const class C arg)
```

在赋值语句里,该函数作为一个成员函数被左操作数调用,右操作数为参数 *arg*。赋值函数在最后使用 *this* 指针把左操作数返回:

```
return * this;
```

不同类型对象之间的赋值

赋值函数决定了编译器如何实现同一个类里不同对象之间的赋值,但它不能实现不同类型对象之间的赋值,编译器必须通过调用适当的构造函数和类型转换函数才能实现一种数据类型到另一种数据类型的转换。

```
CHorse stacy, sugar;  
stacy = sugar;           // Calls CHorse::operator=  
stacy = "fast";          // Calls CHorse::CHorse(char *)  
char * name = sugar;     // Calls CHorse::operator char *()
```

其它赋值操作符(+=、-=等)

读者也许会有这样的想法,如果既定义了类的加法(+)也定义了类的赋值(=)操作,那么加等于操作符(+=)也就自然而然地被定义了,但实际结果不是这样。每一个操作符号(如+=、-=、*=等)都被视为一个独立的操作符,如果你要支持对这个操作符的操作,就需要编写另外一个函数。例如,下面的代码就实现了 CStr 类的 += 操作。(由于这一个函数内容简练,所以它非常适合作内联函数。)

```
class CStr {  
    //...  
    CStr& operator+=(CStr &str);  
};  
  
CStr& CStr::operator+=(CStr &str) {  
    cat(str.get());  
}
```



```
return *this;
```

加等于函数的最后一条语句仍然应该是 `return *this`(同赋值语句一样)。

自增和自减操作符

尽管自增(`++`)和自减(`--`)操作符属于单操作数类型并遵循单操作数操作符的大多数约定,但它们既可以作为操作数的前缀也可以作为后缀,因此它们和普通的单操作数操作符有如下的区别:

- 作为前缀时,操作符首先改变操作数的值,然后再把结果返回,函数实际返回的是一个当前对象的引用。
- 作为后缀时,函数返回的是对象原始值的拷贝(见示例)。其参数的缺省类型是 `int` 型,如果没有对它赋值,则它的值设为 0。这样做的唯一目的是为区别不同的函数定义。

下面的例子显示了自增操作符(`++`)分别作为前缀和后缀时的操作。

```
class CPoint {
private:
    double x, y;
public:
    CPoint& operator++();           // prefix
    CPoint operator++(int);        // postfix
};

// Prefix ++. For efficiency's sake, returns
// a reference (&).

CPoint& operator++() {             // Prefix version
    x++;
    y++;
    return *this;
}

// Postfix ++. Must include dummy arg.
//
CPoint operator++(int dummy) {
    CPoint point(*this);           // Make a copy;
    x++;                           // Change original.
```

```
y++;  
return point;           // Return the copy.  
}
```

下标操作符([])

在编写类组时我们可能需要实现这一操作符。operator[]函数的参数是一个从零开始的整形数。函数应该返回一个引用,从而使下标可以出现在赋值号的任意一侧。

new 和 delete 操作符

用户可以自己编写这些操作符函数来管理内存。编写这些操作符函数时,应该包含头文件 `stddef.h`,并使用下面的声明语句(把 `myclass` 换为你自定义的类名)。

```
#include <stddef.h>  
  
class myclass {  
    //...  
    void * operator new(size_t);  
    void operator delete(void *, size_t);  
};
```

operator new 函数总是返回一个 void * 类型的指针,operator delete 函数则需要一个 void * 类型的参数(指针将根据需要自动进行类型转换)。operator new 函数可以包括其它参数,这些参数在对象的初始化阶段对应一个或多个值。在需要的时候,该函数还可以被重载。

大多数情况下,我们不需要自己编写 operator new 和 operator delete 函数,编译器完全可以胜任这一工作。

函数调用操作符()

这个操作符的用途是使类的对象可以象函数名称一样使用。用户可以用多个定义对此操作符进行重载。下面的例子里,对象 `Print_it` 就可以象函数名一样使用。

```
class Print_it {  
public:  
    int operator()(long a);  
    int operator()(char * s, int n);  
    //...  
};
```

```
};  
  
Print_class print_it;  
// This next statement calls  
// Print_class::operator()(char *, int)  
print_it("cat", 3);
```

语法规则小结

下面的规则普遍适用于所有操作符函数。

- 只能对已经存在的 C++ 操作符进行重载。但下面几个操作符不能被重载: 成员符号(.)、范围标志(::)、指向成员的指针(.*)和条件判断(?:)。
- 不论操作符适用于哪些类, 它们都保持固定的结合性和优先级顺序。
- 除赋值号(=)以外的所有操作符函数都可以被派生类继承。
- 每一个操作符都要独立定义。例如, 即使已经编写了赋值(=)函数和加法函数(+), 加等于操作符(+=)也不可能被自动定义。
- 在操作符函数里不能使用缺省的参数值。
- 不能把任何操作符的实现定义为静态成员函数。

第 八 章

继承 C++ 的优越特性

C++ 的继承特性在软件的循环使用上居于核心地位。有了它,我们再也不必重复编写相同的程序代码,只需简单的编程就可以达到事半功倍的效果。当然,学会使用继承并不是那么容易的事情。但由于它为我们编程提供了巨大的便利,比如我们可以使用继承方便地把一个类的属性扩展到其它类里,从而不必在两个类里重复相同的程序,因此作为一个程序员,我们还是应该努力掌握这一内容。

除此以外,继承还提供了虚拟函数的基础结构,这一点在第九章里再详细讨论。

对于任何 C++ 的 buzzword,继承以多态的方式赋予对象新的功能。继承得到的属性既属于继承者,同时也属于被继承者。根据这样的关系,我们可以得到一幅有用的派生层次图,它如同达尔文的进化论一样由普通逐步发展成为特别。

由 CStr 类谈起:软件工程里的一个窘境

如果你是一章一章按顺序阅读本书的话,那么通过前面的讲解想必你已经了解了如何编写 CStr 类。现在让我们假设一种情况,如果其他人独立开发了这样一个 CStr 类并且把这个软件推销给我们,一般来说,我们将得到这些东西:

- 包含类的声明在内的头文件。
- 经过编译的、包含类的成员函数实现的目标文件(.obj 文件)。

大多数程序开发者不会向用户提供源代码。那么用户怎样才能扩展或修改 CStr 类呢?一种途径是自己从头编写 CStr 类;另外一个好一些的方案是把一个 CStr 对象作为新类的成员包含进来,这样的话,你相当于围绕 CStr 建立起一个外部类。

根据用户编程的工作量,更好的解决方案莫过于使用继承,也就是建立一个新类,并且让它继承 CStr 类的属性。由一种类到另一种类的派生,新类将继承基类(在这里即为 CStr)的所有成员,并且可以在新类里自由地添加成员和变量。

CStr 的派生类

上一章里生成的 CStr 类支持这样一些有用的函数和运算操作,如表 8-1 所示。

表 8-1 CStr 类的函数和运算操作

函数和操作	说明
get	返回一个字符串指针,该字符串以空字符(null)结束。
getlength	返回字符串的长度。
cpy	拷贝 char * 类型的字符串参数。
cat	把 char * 类型的字符串参数添加到当前字符串的尾部。
+	把两个字符串相加,这两个字符串中至少有一个是 CStr 类型,另一个既可以是 CStr 类型也可以是 char * 类型。
=	把另一个 CStr 对象赋值给当前对象。

此外,CStr 类也支持一些有用的构造函数。现在我们要做的是既保持所有这些特性,同时再增加下面两种功能:

- output:把字符串里的数据打印到标准输出设备里去。
- input:从标准输入设备里读入字符串。

即使你除了头文件可以利用外无其它方式访问原来 CStr 类的源代码,你仍然可以通过继承关系派生出一个新类来轻松地增加这些功能。下一节里就向你介绍相关的语法。

派生类的语法表达式

首先假设我们早已声明了 Base_class 类,下面的语句从 Base_class 里派生出 Derived_class 类,Derived_class 自动拥有了在 Base_class 里定义的所有成员,此外,它还可以声明自己的成员。

```
class Derived_class : public Base_class {  
    declarations  
};
```

在这个声明语句里,唯一新出现的东西是冒号(:)和它后面的语句:

```
public Base_class
```

在这样的上下文结构里,public 是对基类的访问标识符。你也可以使用 private 或 protected 关键字,但对于大多数情况,使用 public 是最好的选择。在第十三章里,我们再介绍使用 private 和 protected 的情况。

在 *Derived_class* 的声明语句里可以包含 *Base_class* 里没有的新成员。例如,下面的代码从 CStr 里派生得到 CIOStr,并且在新类里添加了两个函数。要记住,程序里必须把头文件 cstr.h 包含进来,只有这样才能读入 CStr 的声明语句,同时 CStr 必须在使用前具有完整的结构。

```
// CIOSTR.H - declaration of the CIOStr class
```

```
#include "cstr.h"
```

```
class CIOStr : public CStr {  
public:  
  
    void input(void);  
    void output(void);  
};
```

声明完 CIOStr 类之后,就可以用它来定义对象了。这些对象支持所有 CStr 的成员(因为 CStr 是它们的基类)和 input 和 output 函数。

```
CIOStr iostream1, iostream2, iostream3;
```

```
iostream1 = "This is a new string";  
iostream2 = iostream1 + ".";  
iostream2.output();  
iostream1.input();
```

```
iostring3.cpy(iostring1);
```

编写新类的函数

在成员函数的实现方面, CIOStr 遵循同其它类一样的语法规则: 每一个成员函数, 你既可以在类的声明里定义函数(此时它自动成为内联函数), 也可以在声明语句外定义(例如, 在文件 ciostr.cpp 里)。对于第二种格式, 需要在函数名称前面加上前缀 CIOStr::。

新的 CIOStr 函数定义式非常简单。

```
#include <stdio.h>
#include "ciostr.h"

void CIOStr::output(void) {
    printf("%s", get());
}

void CIOStr::input(void) {
    char buffer[256];

    gets(buffer);
    cpy(buffer);
}
```

在这里只定义了两个函数(input 和 output), 它们都冠以前缀 CIOStr::。许多函数是从基类 CStr 里继承得来的, 这些基类里的函数在文件 cstr.cpp 里定义并附以前缀 CStr::。例如:

```
void CStr::cpy(char *s) {    // Copy string arg.
    int n;

    n = strlen(s);
    if (nLength != n) {
        if (pData)
            delete [] pData;
        pData = new char[n+1];
    }
```

```
        nLength = n;  
        :  
        strcpy(pData, s);  
    }
```

由此可见, CIOStr 支持两种类型的函数: 由基类 CStr 提供的函数和 CIOStr 自带的函数。函数定义部分的前缀依据成员函数声明和定义的位置(要么出现在基类里, 要么出现在派生类里)而变。

函数重载和作用域的划分

上一节里读者已经接触到了继承是如何实现的, 在 CIOStr 函数的定义里使用了基类里的东西。现在 we 再看一下 input 函数:

```
void CIOStr::input(void) {  
    char buffer[256];  
    gets(buffer);  
    cpy(buffer);  
}
```

程序里的最后一条语句调用了 CStr::cpy 函数。同其它成员函数的定义一样, 在程序里使用了从 CStr 里继承得来的 cpy 函数。

如果我们试图在派生类 CIOStr 里重载一个或多个基类里的函数, 那么情况会变得复杂一些。在本书的例子中, 虽然没有多大必要重载 CStr 里的函数, 但是这样的操作是完全允许的, 如下例:

```
class CIOStr : public CStr {  
public:  
    void input(void);  
    void output(void);  
  
    // Overridden from base class CStr  
    void cpy(char * s);  
};  
  
void CIOStr::cpy(char * s) {
```



```
// Alternative implementation of cpy function  
{
```

这样一来,类 CIOStr 就有两个不同的 cpy 函数了:一个是在 CIOStr 类里定义的,另一个是在它的基类 CStr 里定义的。CIOStr 的成员可以调用这两个函数中的任何一个。如果要用基类里的 cpy,必须要加上作用域操作符(::)来指明使用的是 CStr 里定义的 cpy,否则的话,系统就认为你调用的是 CIOStr::cpy。

```
void CIOStr::input(void) {  
    char buffer[256];  
  
    gets(buffer);  
    CStr::cpy(buffer);    // Call base-class version of cpy.  
}
```

一般来讲,当编译器读到一个函数名时,它按照下面的顺序寻找函数的来源:

在定义成员函数时出现的函数名,编译器首先检查该名称是否在类里进行了声明,如果发现了此声明,就使用这个声明过的函数。

如果编译器没有在类里找到函数声明,下一步它将在基类里寻找此函数的声明(这种操作是逐级进行的,如果基类之上还有高一级的基类,编译器还将继续在上级的基类里寻找)。

最后,如果函数名称没有在类的派生层次里声明过,编译器将检查它是不是全局函数。

● 注意

如果一个成员函数在派生类里被重载,那么该函数通常应该被定义为虚函数,其原因我们将在本章的末尾加以解释。目前读者可以不考虑虚函数的使用。

继承的层次

在上一章的结尾我们提到,基类也可以有它们自己的基类,派生类也可以继续向下派生出新的子类。为了不致把这种关系弄得更复杂,我们首先考虑一种简单情况。假设你想要声明一个比 CIOStr 的成员函数更多的类,就可以直接从 CIOStr 里派生,

而不必考虑 CIOStr 是由 CStr 里派生出来的。

```
#include "ciostr.h"

class CFontIOStr : public CIOStr {
    int psize;
public:
    void clr(void) { copy(""); }
    void setchar(int c, int n);
    void set_font(int size);
};
```

这里的新类 CFontIOStr 继承了 CStr 和 CIOStr 的所有成员。由于每一次继承都相对于基类添加或重载了一些函数,所以最后的子类不会丢失上级类的任何特性(尽管访问权限会发生变化,这一点我们在本章的后面再讲)。CFontIOStr 具有一个新的数据成员(psize)和三个其它类不具有的新的成员函数。

三个类——CStr、CIOStr 和 CFontIOStr 组成了一个简单的类的层次。原来的字符串类 CStr 是 CFontIOStr 的间接基类。通过继承,CStr 把本身的内容传递给 CFontIOStr,就象祖父把遗传基因传递给孙子一样。图 8-1 显示了这个简单的类层次图。

在 C++ 里有很多非常复杂的类层次图。用户可以从一个基类里派生出多个子类,如此一步步进行将会形成一个非常庞杂的类族。

使用继承和不使用继承的比较

尽管继承不是保持代码重复使用的唯一工具(它也不是在所有情况下都表现出色的最佳选择),但通过下面的说明读者可以看到它是如何节约人力的。

假设你无法访问 CStr 的源程序,而只能通过类的定义了解它(即你看不到函数定义部分的源代码)。但你又要编写一个新类 CIOStr,使它包含 CStr 的所有成员和另外两种功能。如果不使用继承,且不重新编写 CStr 函数,怎样才能建立起这个新类呢?

我们可以不使用继承,但必须围绕一个 CStr 对象编写新类。事实上,这样做无

非是把对该类函数的调用转换为对 CStr 类函数的调用。图 8-2 显示这一内部关系。

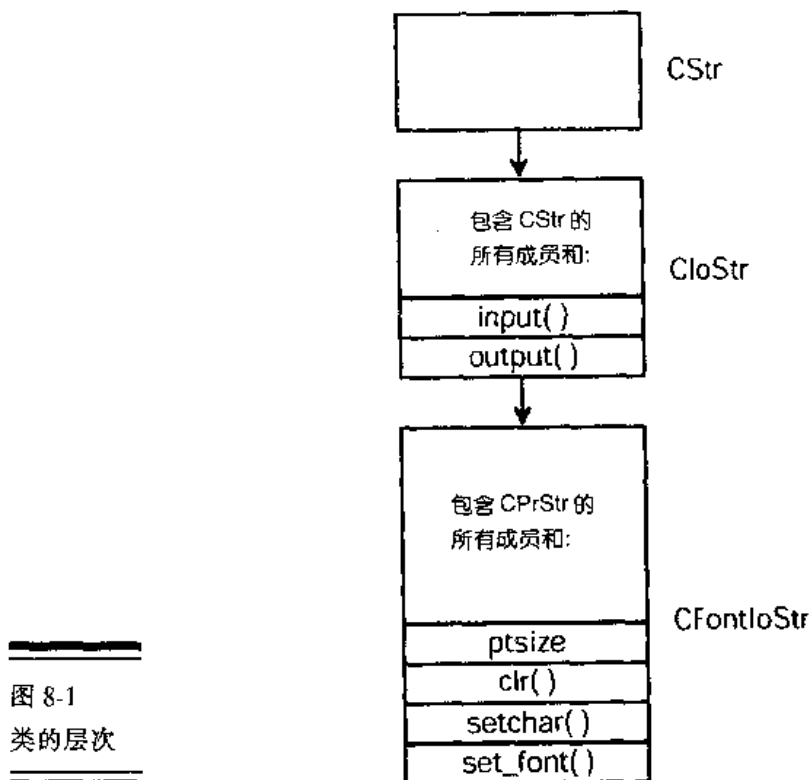


图 8-1
类的层次

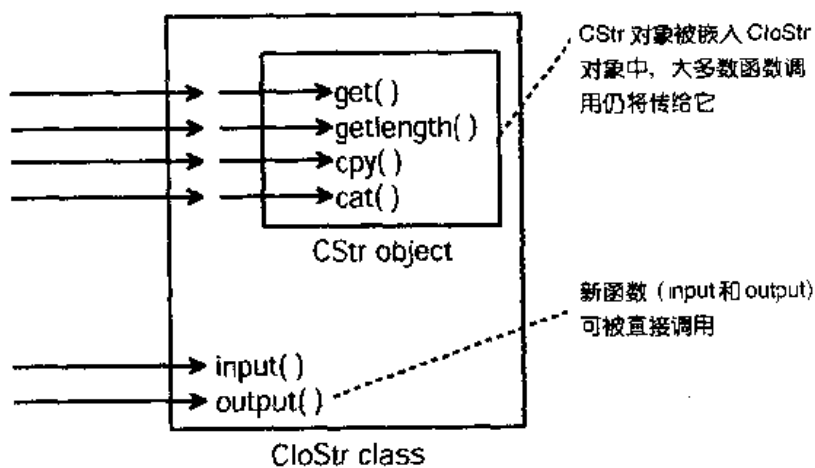


图 8-2
通过包含对象的方法来模仿继承

这种方法的一个弊端是对 CStr 函数的每一次调用都要被显式地转换为对 CStr 对象的调用。下面是程序代码：

```
#include "cstr.h"

class CIOStr {
    CStr str;
public:
    // New functions
    void input(void);
    void output(void);
    // Old functions, simulating CStr inheritance
    char * get(void) const {return str.get(); }
    int getlength(void) const
        {return str.getlength(); }
    void cpy(char * s) {str.cpy(s);}
    void cat(char * s) {str.cat(s);}
    //...
```

除这些函数以外,作为一个完整的实现,在 CIOStr 里还需要为 CStr 类的操作符函数、类型转换函数和构造函数建立一些过渡函数。

现在,我们比较一下这里的 CIOStr 和前面通过继承创建的 CIOStr:

```
#include "cstr.h"

class CIOStr : public CStr {
public:
    void input(void);
    void output(void);
};
```

这两种方法实现的功能是一样的。CIOStr 类既支持 CStr 的所有成员函数,同时又增添了两个新函数(input 和 output)。显然,使用继承的方法定义新类更加简单、易读并且语法清晰。

理论上我们认为继承和对象包含(在一个对象里面使用另一个对象)是不一样的。根据面向对象的理论,继承的使用表示“A 是 B 的一个种属”,派生类 A 仅仅是基类 B 的一种特殊情况。因此我们可以说 CIOStr 是一种比 CStr 更特别的字符串,它增加了一些基类不具有的函数,但不管它们怎么变化,两者都是字符串。

而对象的包含,即某个类使用其它类,就属于另外一种情况了。例如,你可以建立一个 CAddress 类,它里面包含了几个字符串(CStr 对象)和其它一些数据。

```
class CAddress {
```

```
CStr name;  
CStr line1;  
CStr line2;  
int months_at_residence;  
};
```

这里就不能把 CAddress 认为是一种更加特殊的字符串了,它同 CStr 有着根本上的不同。

上面列举的情况非常容易鉴别。但在复杂一些的情况下用户就很难判断一个类究竟应该是从其它类里派生出来好呢(继承)还是应该包含其它的类。出现这种情况时读者应该掌握这样一个最佳判据:使用继承,用户可以方便地把一个类的成员包含到另一个类里,而且这两个类又不会有太大的差别(另外,继承还在虚函数里充当一个重要角色,我们放到第九章里讨论)。

此外,我们还可以使用对象包含的方法而不必编写过渡函数,在使用对象的地方进行相应的函数调用。

```
#include "cstr.h"  
  
class CloStr {  
public:  
    CStr str;  
  
    void input(void);  
    void output(void);  
};
```

这样定义类好编写但不容易用。大多数的函数调用还要加上成员 str。例如,下面的代码段调用了 cpy 函数:

```
CloStr lostring;  
//...  
lostring.str.cpy("Let's go to the Oscars.");
```

如果使用继承,对 cpy 的调用将会变得非常直接。在这一节所介绍的例子里,显然使用继承比其它方法更为方便。

Public、Private 和 Protected 所决定的访问权限

迄今为止,我们还只向读者介绍了一种成员访问权限关键字——public。在某些情况下,这个关键字最有用处。除此之外,C++ 还提供了另外两个关键字,把它们放

在一起,依次是:public、private 和 protected。

在函数的声明部分,private 是缺省的成员访问权限。因为 private 是缺省定义,可以不写出这个关键字(当然,把它写出来可以增加声明语句的易读性)。例如,这里在 CFontIoStr 的定义里使用了 private 关键字指定 psize 的访问权限:

```
#include "CioStr.h"

class CFontIoStr : public CioStr {
private:
    int psize;
public:
    void clr(void) {copy("");}
    void setchar(int c, int n);
    void set_font(int size);
};
```

第三种访问权限 protected 是一种介于 private 和 public 之间的访问方式。一个具有 protected 访问权限的成员,可以在类的作用域和它的派生类里被读写,但在此范围之外受到保护。

现在再看一下对 CIoStr 的声明。该类从基类 CStr 里派生出来,并添加了两个新的函数。

```
#include "cstr.h"

class CioStr : public CStr {
public:
    void input(void);
    void output(void);
};
```

在 CStr 的声明部分,两个数据成员(pData 和 nLength)被声明为私有(private)变量。这种声明是根据缺省情况进行的,因为我们没有使用关键字 public。如果你不必在 CIoStr 类的新函数里使用这些数据成员,那么这样做完全可以,否则就会出现問題,如下例:

```
void CioStr::print(void) {
    printf("%s", pData);    // ERROR! No access to pData
}
```

在前面我们了解到派生类继承了基类的所有成员,在这里也不例外。数据成员

pData 出现在 CIOStr 的任何一个实例里,并且可以通过 CStr 的成员函数,如 get 或 cpy 间接地调用。然而,在派生类 CIOStr 里 pData 对于函数定义是不可见的。

如果 pData 和 nLength 在 CStr 的声明部分里被说明成 protected 类型的话,那么它们既可以在 CStr,也可以在 CIOStr 的函数定义里使用。

```
class CStr {  
protected:  
    char * pData;  
    int nLength;  
public:  
    CStr();           // Constructors  
    CStr(char * s);  
    CStr(const CStr & str);  
    ~CStr();         // Destructor  
    //...
```

图 8-3 总结了三种成员访问权限——public、protected 和 private。

不能断言把数据成员声明为 protected 或是 private 类型孰优孰劣,这要根据成员的具体用途而定。在 CStr 类里,因为一系列公有(public)函数——get、getlength、cpy 和 cat 提供了对数据比较完全的访问,因此没有多大必要把 pData 和 nLength 说明成 protected 型。

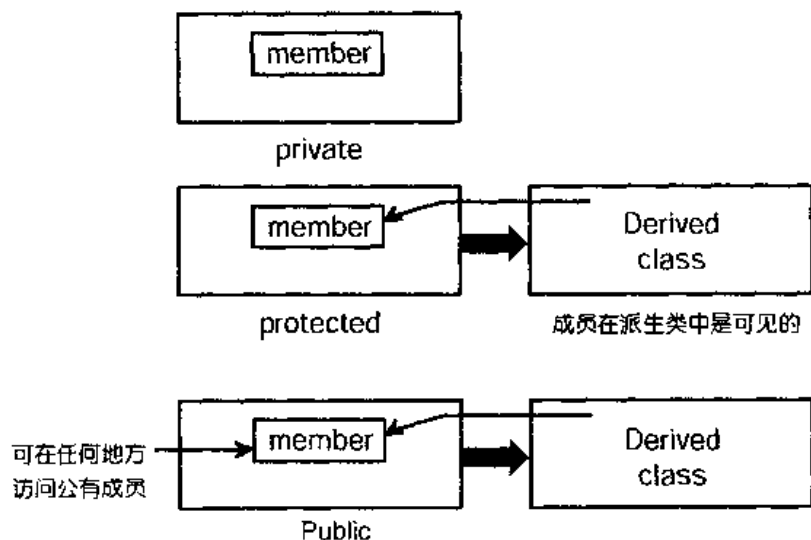


图 8-3
成员访问权
限的小结

在本例里,类的接口(一系列公有函数)把数据完整的封装起来。如果你要把这

个类转给其他程序员,不应该允许他们直接在派生类里更改 nLength,而应合法使用 copy 函数设定这一值。因此,应该把 pData 和 nLength 的访问权限声明成为 private 类型。

但是在其它情况下,使成员函数对于派生类可见是有必要的。声明成员为 protected 类型将为派生类提供最大的灵活性,同时也保留了 private 访问权限的大多数优点。

●—注意—

成员函数也可以象数据成员一样被声明为 public、private 和 protected 类型,用户也可以把数据成员声明为公有类型。但要记住,公有数据成员已经成为类接口的一部分,如果类已经在其它程序里使用了,对接口的更改有可能会致这些程序出错。

另一个实例:轿车类(Fast Cars)和继承关系树型图

继承的中心思想是先定义一个通用类型,然后再由它派生出其它特殊类型,这些派生类型的数量几乎不受限制,所以一个类族一步一步繁衍下来可以变得非常庞大。

例如,你可以定义一个通用类来存储不同轿车的信息:

```
class CAuto {
public:
    char make[20];
    char model[20];
    int year;
    int color_selection;

    // Constructors
    CAuto() {}
    CAuto(char mak[], char mod[], int y, int c);
};

#include <string.h>

CAuto::CAuto(char mak[], char mod[], int y, int c) {
    strcpy(make, mak);
    strcpy(model, mod);
    year = y;
    color_selection = c;
}
```


● 技巧

这里,程序使用了两个 char 类型的数组(make 和 model)来存储字符串。使用 char 类型只是为了使这个例子可以直接运行,如果你已经声明且编辑好了字符串类 CStr 的代码,这里就是一个使用它的好地方。程序开始对两个 CAuto 成员的声明可以改为:

```
CStr make;  
CStr model;
```

对 make 和 model 初值的设定可以通过简单的赋值操作完成,而不必调用 strcpy。使用 CStr 类时,别忘了把头文件 cstr.h 包含在程序的源代码里面,并且把程序连接到 cstr.obj 上。

对于轿车而言,上面定义的消息内容已经足够。但对于某些特定种类的轿车,你也许想要存储更多的消息。比如跑车,你既关注 CAuto 类里的内容(生产厂商 make、型号 model、年代 year 和颜色 color),也想要了解它的马力和速度从零加到 60mph 所用的时间。下面定义的 CSportsCar 就可以存储这些数据,它继承了 CAuto 的成员,同时增加了一些自己的成员。

```
class CSportsCar : public CAuto {  
public:  
    double horse_power;  
    double accel_0_60;  
  
    // Constructor  
    CSportsCar() {}  
};
```

可以认为跑车(CSportsCar)是普通轿车(CAuto)的一个子类。CAuto 的另一个子类可以是客货两用车(CWagon)。对于这个类,我们关注的又是另外一些信息,比如说载货量和最大乘客数等。

```
CWagon : public CAuto {  
public:  
    double storage;  
    int passengers;  
  
    // Constructor  
    CWagon() {}  
};
```

到现在为止,我们已经有了一个基类(CAuto)和两个派生类(CSportsCar 和 CWagon)。下面我们再增加一个新类;CRaceCar。

设计 CRaceCar 类时,我们首先要决定它出现在派生层次的哪一个位置。在本例里,我们把赛车(race car)假设为跑车的一个子类。对于这个类,我们感兴趣的内容除跑车的所有内容之外,还有此型号车在比赛中获胜的次数。

```
CRaceCar : public CSportsCar {  
public:  
    int races_won;  
};
```

因为跑车类是在已经存在派生层次里设计的,因此它的程序代码很简单。CRaceCar 里的大多数成员都是派生得来的。

你也许会注意到在 CRaceCar 的程序里没有构造函数,而在其它的类实例里都有构造函数。本书在第五章曾经指出,应该对类添加一个缺省的构造函数,尤其当以后还会编写一些其它的构造函数时这样做将更加有用。不过在这里由于 CRaceCar 非常简单,我们省去了对它编写构造函数。

图 8-4 给出最后得到的派生层次。

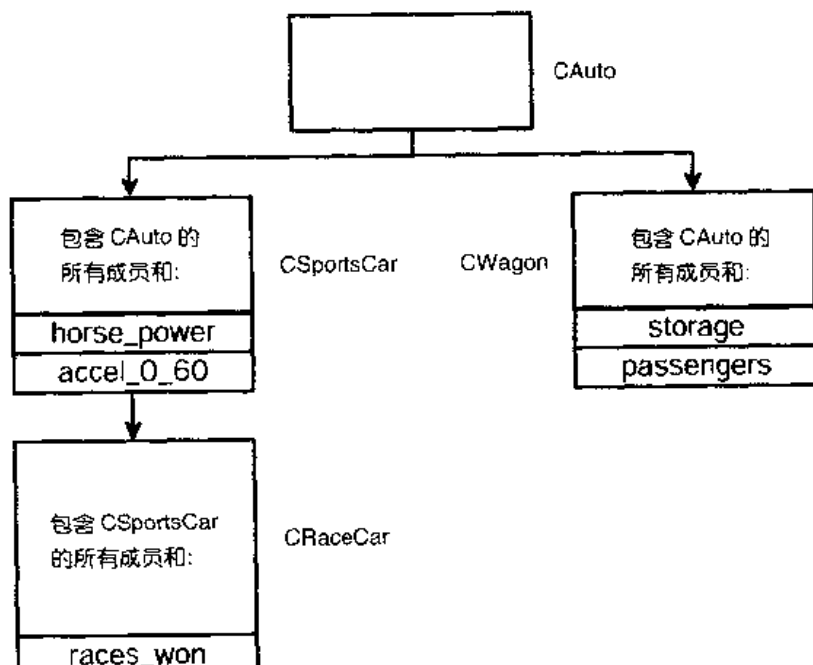


图 8-4
轿车类的派
生层次图

基类构造函数

继承是一种把以前声明过的类的成员(包括数据和函数)包含到新类里的方法。C++ 支持一种特殊操作——基类构造函数,可以有效地实现对派生成员的初始化。

上一节里介绍的跑车类 CSportsCar,它继承了 CAuto 类的成员并添加了新的功能。在它的声明语句里,最好能加上另一个构造函数来对它的所有数据成员初始化(还包括那些从 CAuto 里继承过来的数据)。

```
CSportsCar : public CAuto {
public:
    double horse_power;
    double accel_0_60;

    // Constructor
    CSportsCar() {}
    CSportsCar(char mak[], char mod[], int y, int c,
               double hp, double a);
};
```

新的构造函数依次对所有数据成员做初始化。

```
#include <string.h>

CSportsCar::CSportsCar(char mak[], char mod[], int y,
    int c, double hp, double a) {
    strcpy(make, mak);
    strcpy(model, mod);
    year = y;
    color_selection = c;
    horse_power = hp;
    accel_0_60 = a;
}
```

这个函数还有一些问题需要读者注意。首先,它的执行效率不高。C++ 总是试图调用基类的构造函数来创建新的对象,而这里没有指定基类构造函数,编译器就会调用 CAuto 的缺省构造函数。因此,在这个函数执行之前,原来属于 CAuto 类的成员已经进行了初始化操作(在这里都被赋值为 0),即对 CAuto 成员的初始化进行了两次。

该函数的第二个问题就更加严重了。这里 CAuto 的成员被声明成公有类型

(public), 如果它们被声明为私有(private)类型, 那么此处的 CSportsCar 构造函数根本无法对它们进行写操作。

解决这些问题的方法是在 CSportsCar 的构造函数定义里指定一个合适的基类构造函数, 其语法格式为:

```
class::class(arglist1) : base_class(arglist2) {  
    statements  
}
```

在这种语法结构里, *arglist2* 可以选用 *arglist1* 里的任意参数, 并把这些参数传递到基类构造函数里去, 而基类构造函数的参数必须在数量和类型上与 *base_class* 所定义的构造函数完全匹配。

在下面的函数定义里, 一个冒号(:)位于 CSportsCar 的参数列表和 CAuto 构造函数之间。前四个参数被传递给了基类 CAuto 里的构造函数(它需要四个参数)。如果你现在翻到前面看一看 CAuto 的声明, 就会发现它的确有这样的构造函数。

```
CSportsCar::CSportsCar(char mak[], char mod[], int y,  
    int c, double hp, double a) :  
    CAuto(mak, mod, y, c) {  
    horse_power = hp;  
    accel_0_60 = a;  
}
```

这样一来, 从 CAuto 派生而来的四个数据成员就只需进行一次初始化操作了。它还带来了另外一个好处: 使 CSportsCar 的构造函数更加简单, 更加容易编写。

基类和指针

通过对继承的讨论, 我们还将接触到虚函数的基本理论, 这一内容我们留到第九章里介绍。许多程序员把虚函数的功能视为面向对象编程的核心, 因此这一问题是值得我们花一些时间去了解的。

继承创建了 C++ 类型之间的一种单向联系。如果不使用数据类型转换, C++ 是不允许把一种类型的指针赋值给另外一种类型的指针的, 但允许程序员用基类类型的指针指向派生对象。

图 8-5 显示了这一规则的用法。

非常有趣的是,这种指针赋值只允许在一个方向上进行:基类的指针可以指向派生类,反之则不行。看一看下面的例子,你就能体会到这样规定的意义:

```
CStr * pBase;
CioStr DerivedObj;

pBase = &DerivedObj;    // Assgn. to base-class ptr OK
```

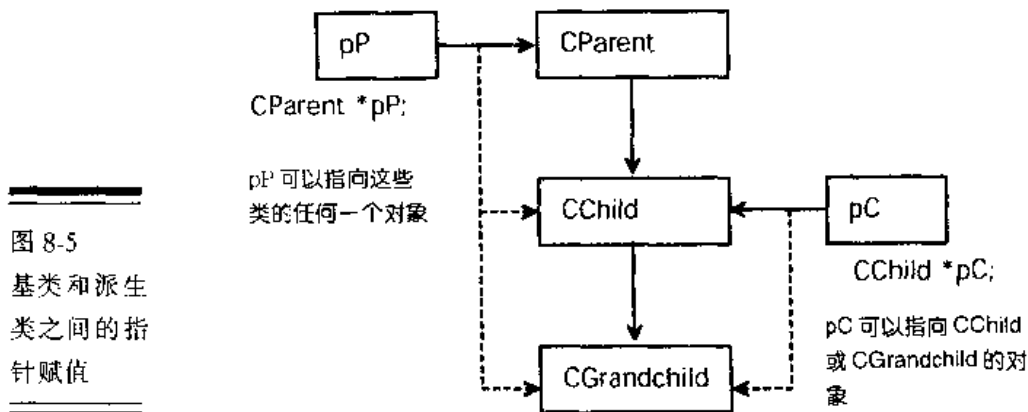


图 8-5
基类和派生
类之间的指
针赋值

声明了指针 `pBase` 之后,编译器假设该指针将指向一个 `CStr` 类型的对象。如果让它指向一个整形(`int`)变量将会出错,这是因为 `int` 类型不支持 `CStr` 的成员。但让 `pBase` 指向一个其派生类的对象(这里是 `CioStr`)则是完全可以的,因为在派生类里至少具有与 `CStr` 类里一样多的成员。不管 `pBase` 指向 `CStr` 对象还是 `CioStr` 对象,下面的语句都是有效的:

```
pBase->ca:(" blah bian blah");
```

但反过来就不成立了。如果 C++ 允许用户把基类(`CStr`)对象的地址自由地赋给派生类指针(`CioStr`),将会导致什么问题呢? 请看下面的例子:

```
CStr BaseObj;
CioStr * pDerived;

pDerived = &BaseObj;    // ERROR! Cannot assign to
                        // derived type without cast
```

问题出在基类 `CStr` 里的元素只构成了 `pDerived` 所指对象的一个子集,而不是超集。而 `pDerived` 所指对象应该至少包含 `CioStr` 的全部成员。这里的 `CStr` 就不能涵盖 `CioStr` 的全部内容。例如,如果出现了这样的函数调用,编译器就无法处理。

```
pDerived->input();
```

input 函数是在 CIOStr 里声明的, CStr 里没有这一个成员函数, 所以当 pDerived 指向一个 CStr 对象时这样的操作就是非法的。

总之, 类的指针可以用这一个类或其派生类的地址赋值, 指针的类型决定了它可以访问哪些成员。例如, 你可以定义一个 CStr 类的指针组, 它里面的指针分别指向不同类型的对象——CStr、CIOStr 或 CFontIOStr 型。除此以外, 指针还必须指向基类里的成员(即在 CStr 里声明的), 如 cpy 函数。

```
CStr * pStr[10];  
pStr[0] = new CStr;  
pStr[1] = new CIOStr;  
pStr[2] = new CFontIOStr;
```

如果某个成员函数被重载将会出现一个有趣的问题。假设在 CIOStr 里 cpy 函数被重载(如这一章的前面所述), 这样一来将有两个同名的函数: CStr::cpy 和 CIOStr::cpy。那么如下的语句将会调用哪一个 cpy 呢?

```
CIOStr iostr;  
CStr * pStr = &iostr;  
pStr->cpy("Initialize me.");
```

最后一条语句实际调用的是 CStr::cpy(基类的函数)而不是 CIOStr::cpy(派生类的函数)。当编译器浏览代码并将函数调用与地址相结合时, 它必须使用那些已经得到的类型信息。这里, 编译器仅仅知道 pStr 与类型 CStr 有一定的关系, 因此它将会调用 CStr::cpy。

```
pStr->cpy("Initialize me.");
```

可能有些读者会有这样的疑问, 对象本身当然知道自己是属于 CIOStr 类型而非 CStr 类型, 所以该对象会到 CIOStr 类里去找相应的函数。

事实上, 只有在运行的时候才会象上面所说的那样进行判断, 而在编译的时候编译器并不清楚对象所代表的类型。而这时编译器又必须明确指出函数地址, 因此它只能利用给出的类型信息来假设该函数来自 CStr 类。

例如, 在下面的代码里, 表达式 pStr[n] 可以指向任意一个 CStr、CIOStr 或 CFontIOStr 对象, 但编译器只会认为函数来自于 CStr 类。

```
CStr * pStr[10];
```

```
//...  
pStr[n]->cpy("hi!");
```

在程序运行当中, `pStr[n]` 所指向的准确类型可以最终判断出来。如果对函数调用的判断可以延迟到运行时再作出, 那么表达式 `pStr[n]->cpy("hi!")` 就能够准确调用函数 (`CStr::cpy` 或 `CloStr::cpy` 中的某一个)。

这种延迟只可能发生在虚函数里, 我们在下一章将详细介绍这一内容。虚函数使用了延迟绑定, 即在引用函数 (如 `pStr[n]->cpy`) 时, 并不将其绑定在内存里一个实函数的地址上, 而是在运行的时候才决定其准确地址。

第九章

虚函数及其性质

面向对象的程序可以说就是一系列对象实体通过互相发送消息的方法进行相互通讯。同时,这些消息本身应该具有一定的独立性,比如我们可以自由地发出“Hello”或“What is your name?”等消息而无须知道接收方是谁。

也许看似简单,但以上这些就是虚函数的全部内涵。虚函数的调用和上面的情况类似,我们不必在意是通过哪种类型的对象向这些虚函数发出调用请求的。事实上,该对象所属的类可能在编写完调用代码后还不存在。另外,新类型的对象、新的函数代码还可以在函数的调用程序编写完后陆续添加,而不必时时对主模块重新编译。这样一来,程序的功能就具有了更大的分散性,函数调用的主动权也就完全掌握在对象自己手中。

当你刚开始用 C++ 编程时,也许还不了解这样做的好处,其中的优点只有通过长期的使用才能显现出来。在这一章里,我们将通过菜单命令的例子向你逐步阐释虚函数的概念。

●—注意—

虚(virtual)这个词通常是描述某种东西不是实物,但却又具有实物的某些功能或性质。对于函数而言,这里的术语——虚是指一个函数调用可以对应不同的函数入口地址,而虚函数本身却是真实存在的,由此可见,虚这个名称用在这儿并不太确切。准确地说,虚函数仅仅表示函数是以间接的方式被调用而不是由一个固定的函数入口地址来调用。

关键字 virtual 的使用

把一个成员函数变为虚函数,只需在函数声明前加上关键字 virtual。这个关键字必须出现在声明的开始部分和返回值的类型说明前。


```
virtual return_type name(arguments);
```

上面语句执行的结果就是把某函数声明成虚函数,它将被延迟绑定,即直到程序运行时才决定该函数的入口地址。下面我们就具体谈一谈这样做的好处。

一旦在函数的声明里使用了关键字 `virtual`,你就不必在函数的定义部分和它的派生类的声明部分里重复使用该关键字了。它的所有派生类都保持 `virtual` 类型。

例如,假设你的某个类 `CTimePiece` 里有函数 `ShowTime`,并且该函数被声明为 `virtual` 型。

```
class CTimePiece {
public:
    virtual void ShowTime(void);
    //...
};
```

在 `ShowTime` 函数的定义部分不必再使用关键字 `virtual` 了,因为它已经被声明为虚函数了。

```
#include <iostream.h>

void CTimePiece::ShowTime(void) {
    cout<<"hours:<<"<<minutes;
};
```

对于所有从 `CTimePiece` 里直接或间接派生出来的类,其 `ShowTime` 函数都是 `virtual` 型的,甚至包括下面这个对 `ShowTime` 函数进行重载的类:

```
class CClock : public CTimePiece {
public:
    void ShowTime(void);    // Override function
    //...
};

void CClock::ShowTime(void) {
    paint_clock_face(hours, minutes);
}
```

用户无需在派生类里重载虚函数或通过指针来调用它,虚函数的调用同其它成员函数的调用没什么两样。即使是在声明和使用上,两者的语法上也没有什么差异(也就是说,虚函数的源代码和普通成员函数是一样的,尽管在外观上有一点不同)。

```
CTimePiece time;  
CClock clock;  
  
time.ShowTime();  
clock.ShowTime();
```

虚函数除了不能作为内插函数外,它遵循普通成员函数的所有语法规则。

把函数转变为虚函数的作用是为了配合通过基类指针访问某一对象的操作(请参阅第八章的最后一部分)。借助 ShowTime 函数,用户就可以使用同样的代码而调用不同的函数实现。在下例里,假设 CClock 和 CDigital 都是 CTimePiece 的派生类:

```
CTimePiece * pTime;  
  
pTime = new CClock;  
pTime->ShowTime();    // Calls CClock::ShowTime  
pTime = new CDigital;  
pTime->ShowTime();    // Calls CDigital::ShowTime
```

虚函数的使用场合

尽管 C++ 对虚函数和普通函数的处理大相径庭,但它们两者除了在声明部分里一个有关键字 virtual 而另一个没有外,语法上再也没有其它区别。因此,用户很容易把虚函数引入已经设计好的 C++ 工程文件里。

但我们是不是有必要把所有的函数都声明成虚函数呢?除了内插函数外,我们只要在函数的声明部分加上关键字 virtual 即可把普通函数转变为虚函数,但实际上没有必要对所有函数都这样做。

如果在派生类里没有对某一函数进行重载,也就不必把此函数声明成虚函数。在进行函数调用时,虚函数的表现要略逊与普通函数,并且它也比代码相同的实函数占据较多的内存空间。因此如果普通成员函数可以满足要求,就不要使用虚函数。

如果你在基类里编写了一个函数,并且知道在派生类里该函数必然会被重载,那么就应该把这个函数声明成 virtual 类型。如果你没有这样做,尽管定义时还是可以对派生类里的普通函数进行重载,但在调用时可能会误调其它函数。

在上一章的最后一个例子里,下面的程序将调用函数 CStr::cpy(基类函数),而不是 CloStr::cpy(派生类里的函数)。

```
CloStr iostr;  
CStr * pStr = &iostr;  
pStr->cpy("Initialize me."); // Call CStr::cpy or  
                             // CloStr::cpy?
```

但如果 cpy 函数被声明成 virtual 类型(如下例),上面的程序就将调用 CloStr::cpy:

```
class CStr {  
    //...  
    virtual void cpy(char * s);  
    //...
```

上面这个例子并不实用,因为没有多大必要重载 cpy 函数(除非专门用于举例说明)。因此也就不需要把该函数定义成虚函数。

下一节里给出的成员函数都非常适合作虚函数,因为它们在设计时就已经考虑到会在派生类里被重载。

菜单命令的实例

窗口系统是一个简单的面向对象的程序范例。该程序显示了一系列菜单并由用户作出选择,程序再根据用户所选择的操作作出反应。

使用传统的面向过程的程序设计方案也可以简单地实现这一系统,但使用面向对象的方法还是会带来一定好处。如果把每一个菜单命令都作为独立的对象进行编程,那么在一定范围内它们自己就可以独立实现一些功能而不必受外界控制。这种编程方式无须主程序或 switch 语句来管理(当添加新的命令时这些语句都要作出相应的修改)。

总之,面向对象的编程使程序易于编写并减少了出错的可能,从而赋予了程序开发者更大的灵活性。

基类的声明和定义

基类 CMenuItem 是所有菜单命令函数的原形。但程序中没有这一个类的实例,即不使用 CMenuItem 类来定义任何对象,而只用 CMenuItem 的派生类来定义对象。

CMenuItem 的形式非常简单,它只包含两个成员。下面的 cmenu.h 文件给出该类的完整声明。

```
// CMENU.H - declaration of the CMenuItem base class
class CMenuItem {
public:
    char title[81];
    virtual void Do_Command(void) = 0;
};
```

这里出现了一个新的语法句型。在 Do_Command 声明语句之后的字符“=0”表明该函数是纯虚函数(*pure virtual function*)。这种函数同其它虚函数一样,只是在基类里没有实现语句,而只能在 CMenuItem 的派生类里实现它。在这个类里,Do_Command 函数就没有被定义,它只是作为一个空函数等待以后在派生类里进行定义。

我们将在这一章的后面一节“无实现函数(纯虚函数)”里进一步介绍这种特殊函数。

每一个菜单命令对象都实现自己的 Do_Command 函数。主程序唯一的功能是调用选定对象的 Do_Command 函数来执行某一指令。事实上,对 Do_Command 函数的调用就是向对象发出“执行你的指令”这一消息。

菜单(Menu)对象的声明和定义

每一个菜单命令都代表一个 CMenuItem 对象并执行 CMenuItem 的函数 Do_Command。示例程序里使用了三个菜单命令,命令的结果分别是:

1. 使扬声器发出鸣叫。

2. 显示一句名言。

3. 将两个数字相加。

这三个菜单命令类的声明除了名称不同外其它完全相同。它们都需要重新声明一下 Do_Command 函数以表示该函数重载了基类里的函数。

```
// CMDS.CPP - Defines and initializes menu commands
```

```
#include <stdio.h>
#include "cmenu.h"
class CMenuBell : public CMenuItem {
    void Do_Command(void);
};

class CMenuSaying : public CMenuItem {
    void Do_Command(void);
};

class CMenuAdd : public CMenuItem {
    void Do_Command(void);
};
```

每一个 Do_Command 函数都执行了一个实际的操作,这些函数可长可短,彼此之间可以千差万别。

```
// CMDS.CPP (continued)
void CMenuBell::Do_Command(void) {
    puts("\ 007", stdout);    // "\ 007" rings a bell
}

void CMenuSaying::Do_Command(void) {
    puts("If you know the meaning of the universe,");
    puts("Make the sound of one hand clapping.");
}

void CMenuAdd::Do_Command(void) {
    double x, y;
```

```
printf("Enter a number: ");
scanf("%lf", &x);
printf("Enter a number: ");
scanf("%lf", &y);
printf("The total of the numbers is %f.", x + y);
}
```

在 `cmds.cpp` 文件的最后声明了一个指向 `CMenuItem` 对象的类组、计算指令数目的整型数和一个初始化函数。这些变量都将在主程序里使用。

```
int num_commands;
CMenuItem * commands[20];

void Init_Commands(void) {
    commands[0] = new CMenuBell;
    strcpy(commands[0]->title, "Sound a bell.");
    commands[1] = new CMenuSaying;
    strcpy(commands[1]->title, "Print a message.");
    commands[2] = new CMenuAdd;
    strcpy(commands[2]->title, "Add two numbers.");
    num_commands = 3;
}
```

对象的使用

因为菜单对象已经完成了几乎所有的工作,所以主程序就变得非常简单。这正是面向对象编程的风格,它根据对象而不是主程序流程来进行控制和分配资源,从而实现了功能的分散化。

主程序首先必须获得对两个全局变量: `num_commands` 和 `commands` 数组的访问权,因此在变量声明里使用了 `extern` 关键字。如果你还要在其它程序模块里使用这两个变量, `extern` 关键字就不能省。

```
// MAIN.CPP - Uses the menu objects to manage a menu

#include <stdio.h>
#include "menu.h"
```

```
extern int num_commands;  
extern CmenuItem * commands[];
```

主函数首先调用 Init_Commands 对菜单对象进行初始化,然后再建立一个循环分别显示菜单和相应的指令。如果用户输入的数值同菜单的允许值不符,那么循环终止。

```
void main(void) {  
    int i, sel;  
  
    Init_Commands();  
    do {  
        puts("\nMENU: \n");  
        for (i = 0; i < num_commands; i++)  
            printf("%d. %s", i+1, commands[i]->title);  
        printf("\nEnter a selection: ");  
        scanf("%d", &sel);  
        if (sel > 0 && sel <= num_commands)  
            commands[sel]->Do_Commands();  
    } while (sel <= num_commands);  
}
```

以上就是全部程序代码。在这里 virtual 关键字的作用表现得非常突出。假设程序的其它部分都不变,仅仅是 Do_Command 没有被声明为虚函数,那么不论用户作出何种选择,下面这一条语句都将执行相同的指令(基类里的 Do_Command):

```
commands[sel]->Do_Command();
```

如果 Do_Command 没有被声明成虚函数,那么在编译时 C++ 将根据 commands[sel]指向对象的类型进行相应的函数调用。变量 commands 是一个指向 CmenuItem 对象的数组。由于 Do_Command 不是虚函数,编译器将把这一函数调用解释成对 CmenuItem::Do_Command 的调用。

在本程序里,由于 CmenuItem 里的 Do_Command 函数是一个纯虚函数,所以 CmenuItem 不能提供任何 Do_Command 的实现。但如果 Do_Command 不是纯虚函数的话,那么基类必须提供一些实现语句作为缺省操作。总之,要是没有虚函数,主程序将总是调用缺省的菜单操作,程序员所指定的菜单指令也就永远不会被执行。

虚函数在应用上的优点

如果按照上面的例子编写程序,显然虚函数是不可缺少的一部分。但换一种方式,不使用虚函数编写以上程序显然也是可以的。因此就产生了一个问题,采用虚函数编写程序菜单究竟有哪些好处呢?

你完全可以使用传统的编程方式编写菜单程序,但必须在主函数的循环语句里添加相当多的控制结构。例如,在主函数的循环语句里需要有对选定值的测试语句和相应的操作,而不是象面向对象的程序设计里直接通过当前对象调用 `Do_Command` 函数:

```
switch (sel) {  
    case 1:  
        Bell_Do_Command();  
        break;  
    case 2:  
        Saying_Do_Command();  
        break;  
    case 3:  
        Add_Do_Command();  
        break;  
};
```

每次对菜单结构的更改都要对主程序作出相应的调整,并且还要修改任何激活菜单指令的程序行里的代码。如果使用虚函数,程序就变得非常容易维护了,你只需使用下面一条语句就可以激活所选定的菜单命令:

```
commands[sel]->Do_Command();
```

传统做法的最大缺点并不是它需要编写更多的程序代码和形式上的不规范(尽管它的确如此),而是它把所有的控制都集中在主函数的循环语句里面。如果使用虚函数就可以避免这个问题。我们可以看到,今后不管有几个菜单命令加进来,main.cpp 里的代码都不必重新编译,当然也就不用修改和重新创建主程序了。

虚函数还有其它几个优点。首先,尽管对象必须在某些位置被初始化,但进行初始化的位置可以非常灵活,在本例这一操作出现在 `Init_Commands` 函数里。当虚函数的框架建立起来时,你可以使用任意方式对类实例进行初始化。例如,你可以用数据文件对其初始化(在 Windows 编程里,建立可执行文件的过程类似于使用资源编

辑器创建菜单)。

其次,使用虚函数甚至可以在程序运行之时改变菜单的属性。例如,因为运行时的条件不适合而导致菜单项无法工作或需要新的菜单项(可以参阅一下 Microsoft Word 和其它常用软件的 File 菜单在运行时是如何变化的)。虚函数对这种情况的处理具有更大的灵活性,它可以在任何时候改变菜单结构。而使用传统的 switch 语句则很难对这种动态的菜单结构编写代码,换句话说,switch 语句结构同程序的流程控制紧紧结合在一起,因此在程序编译后无法视情况作出相应变化。

● C/C++

C++ 里大多数面向对象的编程都可以由 C 仿真实现,尽管用 C 编程时代码比较复杂并需要做一些多余的工作,但这种仿真一般还是能够令人满意的。对于虚函数来说也是如此。你可以用回调函数(callback function)来仿真虚函数(这里的回调函数是指那些把函数的入口地址传递给主程序或特定的例程,并根据此地址进行调用的函数)。

虽然回调函数可以取代虚函数,但你得要对每一个菜单命令都编写一个回调函数,并且建立一个结构数组来存储每一个菜单命令和相应的回调函数入口地址。由此看来,在这个程序里使用虚函数编程工作量要少得多,而且代码紧凑易读。

后面的一节“如何实现虚函数”里会进一步阐述虚函数和回调函数之间的相同之处。

无实现函数(纯虚函数)

上面的 Do_Command 函数就是一个纯虚函数的例子。在 cmenu.h 头文件里,对 Do_Command 成员函数的声明使用了字符“=0”表明了在此类里没有实现语句。即没有对 CmenuItem::Do_Command 的函数定义。因此在 CmenuItem 的派生类里必须先定义 Do_Command 的实现之后才能使用它。

```
class CmenuItem {
public:
    char title[81];
    virtual void Do_Command(void) = 0;
};
```

我们把具有虚函数的类定义为抽象类。CmenuItem 声明了纯虚函数 Do_Command,所以它就是一个抽象类。抽象类在使用时有一个非常致命的限制:它不能被

直接用来创建对象,但可以声明一个指向它的指针。

```
CmenuItem * pMenu;           // OK
CmenuItem menu_thing;        // Error! Cannot instantiate
```

如果不能创建一个抽象类的实例,那使用它们又有何好处呢?正如前面所说,抽象类奠定了一个类族的基础。本节例子里的 CmenuItem 类虽然简单,但仍然很好地实现了这一功能。

CmenuItem 类把 Do_Command 的实现定义成什么也不做:

```
// CMENU.H - declaration of the CmenuItem base class

class CmenuItem {
public:
    char title[81];
    virtual void Do_Command(void);
};

// MENU.CPP - implementation of CmenuItem

void CmenuItem::Do_Command(void) {
}
```

然而,这样一来还是增加了一些不必要的工作(因为根本不会直接或间接地建立 CmenuItem 的实例或调用 CmenuItem::Do_Command)。另外,把该函数定义为纯虚函数已经保证了在派生类里它必然会被重载。如果你直接创建了一个 CmenuItem 类的对象,编译器必然会给出相应的出错信息。

如何实现纯虚函数

本书没有用过多的篇幅介绍 C++ 在底层是如何执行的。编译器在执行 C++ 的某项功能时可以使用多种途径,作为用户,我们应该把侧重点放在 C++ 面向对象的编程上而无需了解这些底层编译器的执行细节。

然而 C 的程序员经常会考虑程序效率以及执行速度和程序所占内存空间的比例关系(有些 C++ 的程序员也会有类似的顾虑)。这里我们就介绍一下使用虚函数时速度和空间两者之间的关系。对于简单的情况(无多重继承),虚函数的实现比较直接,在不同的 C++ 编译器里也没有什么区别。

正如前面所指出的,虚函数和回调函数密切相关。两者都利用了处理器的功能进行间接调用(通过指针实现函数调用)。例如,下面的代码使用了 C 里的语法来间接地调用 Hello 函数:

```
void Hello(int n) {  
    printf("Hello, your lucky number is %d. \n", n);  
}  
  
void (* pFunction)(int);  
  
pFunction = &Hello;  
(* pFunction)(5);    // Call Hello through pFunction
```

虚函数给这种间接调用增加了一些新的内容:它利用成员函数的方式来表达间接函数调用;它对每一个类都维护一个函数指针表,并且它隐藏了间接调用的语句,使对虚函数的调用看起来象是对某个成员函数的调用。有时当你编写一个派生类时,甚至都可能忘了某一函数是虚函数。在 C++ 里,对虚函数的编写和调用同普通函数相比没有太大区别。

所有这些东西都是最终为了给编程提供方便,但并不是说它们本身就不重要。虚函数的间接调用越方便,用户就越倾向于使用它。此外,它还提供了一个框架(类层次)来显示这些虚函数的结构和使用范围。

编译器会为任何一个包含虚函数的类建立起一个函数指针表。表里的每个条目都对应一个虚函数的入口地址。例如,对于下面的类:

```
class CShape {  
public:  
    virtual int SetPoints(double ptArray[]);  
    virtual void DrawMe(void);  
    virtual void Move(double x, double y);  
};
```

其中三个函数 SetPoints、DrawMe 和 Move 都适合作虚函数,它们在 CShape 的派生类里很可能会被重载。

C++ 编译器创建了一个 CShape 类的函数指针表。需要指出的是,编译器并不是对任何一个 CShape 对象都创建这样的表。由于所有的对象都共用这些成员函数,所以某个类里有一个这样的函数指针表就已足够了。

根据表里所存储的变量是类的虚函数指针,我们可以把它称为 vtable。里面的指针指向各个虚函数的入口地址。如果类里还有其它声明的非虚函数,那么指向这些函数的指针不会出现在该表里,这是因为编译器对非虚函数的处理遵循普通函数的调用规则,当然也就没有必要把它们入口地址也存储在 vtable 里。

图 9-1 显示了 CShape 函数表的结构。

我们可以从 CShape 里派生出任意数目的子类,并且这些类分别执行自己的函数

(它们还可以添加新的虚函数,这些虚函数的指针被依次置于 vtable 的尾部)。对于某个对象而言,它又是如何知道自己是 CShape 类型而不是 CShape 的其它派生类型呢?

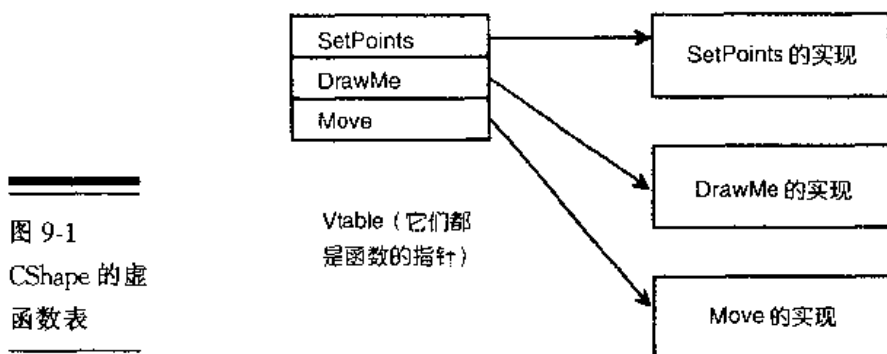


图 9-1
CShape 的虚
函数表

这一问题的答案在于每一个对象都具有一个隐藏数据成员,并且由它使类的对象和虚函数指针表 vtable 建立起一一对应的联系(如果对象的类里没有虚函数,那么编译器就不会给它添加这一成员)。此隐藏数据成员是指针类型,我们把它命名为 pVtable,它位于对象的开始位置。事实上,对象不知道也不必知道自己本身是何种类型,因为程序是通过 pVtable 指针对函数进行间接调用而不是根据对象类型进行函数调用的,同时,这样做也确保了函数调用的准确性。每一个对象的 pVtable 成员都指向相应的 vtable(例如,CCircle 和 CSquare 分别拥有自己的 vtable),所以每个类的对象都与各自的函数代码联系到了一起。

图 9-2 显示了 CShape 对象的构成。我们可以看到通过 pVtable 进行的间接调用是如何过渡到对函数实现的调用。

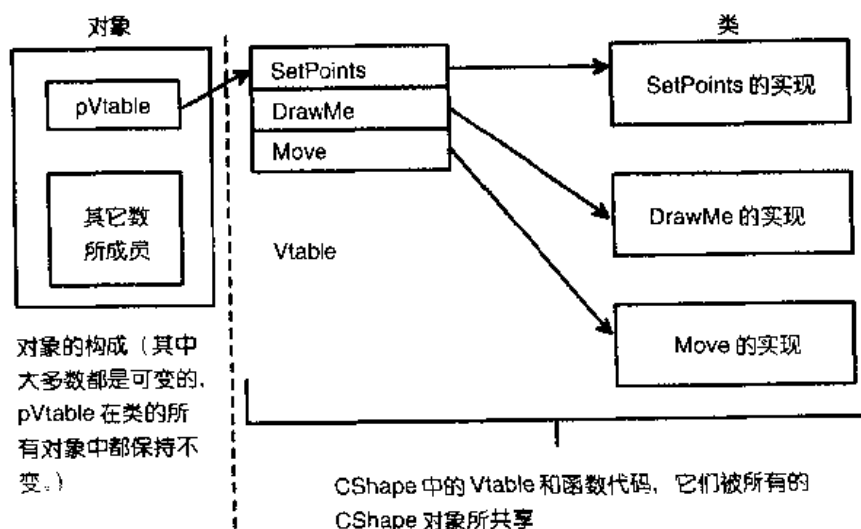


图 9-2
CShape 对象
和虚函数表
(vtable)

我们再看一下如何通过对象来调用虚函数，

myshape:

```
CShape myshape, * pShape;  
pShape = &myshape;  
pShape->DrawMe();
```

C++ 的编译器把对 DrawMe 的调用转换为对下列函数的间接调用：

```
(* (pShape->pVtable->DrawMe))();
```

上面的虚函数调用操作可以用标准 C 来实现,但必须执行下面的几步工作:

1. 对于每一个类都要声明一个结构,并且在该结构里必须包含指向虚函数的函数指针成员。
2. 对该结构进行初始化,使它里面的函数指针指向相应的函数。
3. 改变类的声明,使它包含一个新成员 pVtable。
4. 初始化 pVtable,使它指向类的虚函数表。
5. 按照前面的代码格式改写所有虚函数调用语句。

在标准 C 里上面的操作都是可以实现的,但使用 C++ 将更加方便,因为它自动完成了很多工作。C++ 的一个非常重要的特性是可以在封装程序内部执行虚函数指令而不必更改调用这些函数的源代码。

在这一章里我们对虚函数进行了比较细致的介绍,可以这么说,如果你掌握了虚函数,你就已经进入到了面向对象编程的核心。

第二部分

C++ 参考大全

轻松学习 C++

C++ 元素速查表

第十章 数据类型

第十一章 运算符

第十二章 类型转换操作符 (cast)

第十三章 C++ 的关键字

第十四章 预处理指令、宏和运算符

第十五章 库函数

第十六章 I/O 库类与对象

为方便读者使用C++，在第二部分里我们提供关于C++详细的参考内容，它里面几乎涵盖了C++的所有元素，包括运算符、关键字、预处理命令、库函数、库类和对象。这一部分以两个速查表作为开篇，它们分别是“轻松学习C++”和“C++元素速查表”。如果你清楚自己要做的工工作，但不知道应该使用C++的哪些工具来完成，就请翻阅“轻松学习C++”部分。如果你已经对C++有所了解，但还想进一步了解某些元素的信息，就可以使用速查表来寻找你感兴趣的内容。为方便读者快速检索某一元素所在的章节，“C++元素速查表”里的内容是按照字母顺序从A到Z依次编排的。

轻松学习 C + +

这里我们首先用浅易的文字描述用户在编程时想要进行的操作(左侧),然后在右侧对应位置给出实现这些操作的关键字、运算符和库名称。如果你对右侧给出的术语不甚明了,可以继续查阅“C++ 元素速查表”。请注意,这里使用了一个通用的约定,非英文字母符号都表示运算符(第十一章),而以井号(#)开头的术语为指令(第十四章),其它都是关键字和库函数(第十三章和第十五章)。

要实现的操作	所用元素
求绝对值	abs, labs, fabs
访问对象的成员	.
通过指针访问对象成员	->
允许访问类的私有成员	friend
相加后赋值	+ =
两个数相加	+
自加1 操作	++
获得变量的地址	取地址符(&)
直接进入下一次循环	continue
给某一类型定义一个别名	typedef
分配内存给新对象	new
给数组分配内存	calloc
给单个变量分配内存	malloc
分配寄存器变量	register
给对象集合分配内存	new []
分配静态数据变量内存	static
按位与(AND)操作	&

要实现的操作	所用元素
逻辑与(AND)操作	&&
反余弦	acos
反正弦	asin
反正切	atan, atan2
使用与变量等长的参数列表	va_arg, va_end, va_start
获得命令行参数	main
访问数组成员	[]
进行数组复制	memcpy
大数组查找	bsearch
比较两个数组	memcmp
获得数组的长度	sizeof
转换汇编语言程序代码	asm
对变量赋值	=
从二进制文件里读取数据	fread
以二进制方式向文件里写入数据	fwrite
执行二进制方式的搜索操作	bsearch
二进制数的移位操作	<< >>
进行位操作	AND(&), OR(), NOT(!)
执行内存块操作	memchar, memcmp, memcpy, memmove, memset
布尔数据类型(真或假)	bool
设定文件操作所需的缓冲区	setbuf, setvbuf
数据流输入 / 输出缓冲区	fflush
检测 C++ 的使用情况	_cplusplus
case 条件语句的标志	case
动态类型变量之间的类型转换	dynamic_cast
原始数据类型变量之间的类型转换	static_cast
相关类型变量之间的类型转换	static_cast
消除变量的常量属性	const_cast

要实现的操作	所用元素
将指针类型转换为其它类型	reinterpret-cast
改变文件名称	rename
保护变量不被更改	const
允许改变常量(const)类型的成员	mutable
访问正处于变换中的数据对象	volatile
使用字符型数据	char
从文件里读取一个字符	fgetc,getc
向文件写入一个字符	fputc,putc
向文件写入一个字符	putchar
取消读入的字符	ungetc
检测单个字符	is<cond>
接收从键盘输入的字符	getchar
连接外部 C 程序模块	extern
编写类模版	template
声明一个类	class, struct, union
清除错误信息	clearerr
清空文件缓冲区	fflush
关闭并重新打开某一文件	freopen
关闭某一文件	fclose
合并两个二进制数	bitwise OR ()
连接两个字符串	strcat
使用系统外部命令处理器	system
获取命令行输入	main
比较两个字符串	strcmp
比较两个内存块	memcmp
报告编译的时间	_TIME_, _DATE_
条件编译	# if
报告编译错误	# error
连接两个字符串	strcat

要实现的操作	所用元素
条件编译	#if
向控制台输出数据	putchar, puts, printf
从控制台读入数据并赋给变量	scanf
从控制台读入字符	getchar
从控制台读入字符串	gets
使用常数变量	const
定义符号常量	#define, enum
允许改变常数变量的值	mutable
直接进入下一次循环	continue
禁止进行类型转换	explicit
改变数据的类型	const_cast, dynamic_cast, reinterpret_cast, static_cast
把数字转换为数字字符串	sprintf
把字符串转换为浮点数	atof, strtod
把字符串转换为整型数	atoi, atol, strtol, strtoul
把字符串转换为全小写	tolower
把字符串转换为全大写	toupper
复制数组	memcpy
复制内存块	memcpy
复制字符串	strcpy
余弦	cos
通过计数器变量使用循环结构	for
将变量放在 CPU 的寄存器里	register
获得指向当前对象的指针	this
获取当前源文件的行号	__LINE__
获取日期	time
获取编译的日期	__DATE__
打印周日(星期几)	strftime
获取周日	mktime, time

要实现的操作	所用元素
声明一个指针	*
声明引用变量	&
减小内存块的大小	realloc
变量自减	--
对某一类型定义一个别名	typedef
定义一个宏	# define
从磁盘里删除一个文件	remove
删除一个对象	delete
访问指针所指的对象	*
将类成员的访问权限扩展到它的继承类里	protected
检测是否到达文件尾部	feof
要求使用直接内存访问操作	volatile
显示数据的值	printf
以字符串的形式显示数据的值	fprintf
变量相除后赋值	/=
变量相除并获取余数	%, fmod, div, ldiv
变量相除	/, div, ldiv
去掉某一个数的小数部分	modf
动态分配内存	malloc, new
动态分配对象	new
e 指数	exp
执行 else 语句	else
清空文件缓冲区	fflush
允许从外部访问类的成员	public
将类的成员封装起来	private
检测文件是否结束	feof
定义枚举类型的变量	enum
获取环境变量的设置	getenv

要实现的操作	所用元素
检验表达式是否相等	<code>==</code>
检测数组是否相等	<code>memcmp</code>
检测字符串是否相等	<code>strcmp</code>
打印编译时的错误	<code># error</code>
打印错误消息流	<code>cerr</code>
获取文件错误类型	<code>ferror</code>
清除错误	<code>clearerr</code>
异常处理	<code>throw</code>
设置异常处理模块	<code>try</code>
进行异或运算	<code>^</code>
重复执行某些指令	<code>do while</code>
执行操作系统的指令	<code>system</code>
异常退出	<code>abort</code>
从函数里退出	<code>return</code>
退出循环	<code>break</code>
从程序里正常退出	<code>exit</code>
将函数代码扩展为内联函数	<code>inline</code>
获取尾数和指数	<code>frexp</code>
执行乘方操作	<code>pow</code>
扩充数据流运算符(<<、>>)	<code>istream, ostream</code>
声明一个外部变量	<code>extern</code>
检测逻辑值是否为非(false)	<code>!</code>
获取文件错误	<code>ferror</code>
使用文件的输入/输出字符流	<code>fstream, ifstream, ofstream</code>
获取文件所在位置	<code>fgetpos, ftell</code>
从磁盘上删除某一文件	<code>remove</code>
获取当前源文件	<code>__FILE__</code>
获取文件里的标准输入数据	<code>fscanf</code>
从文件里读取字符	<code>fgetc, getc</code>

要实现的操作	所用元素
打开一个文件	fopen
将字符输出到文件里	fputc, putc
将格式化字符串输出到文件里	fprintf
把字符串输出到文件里	fputs
从文件里读取二进制数据	fread
从文件里读取字符串	fgets
确定字符串里某一子串的位置	strstr
浮点数类型	double, float
填满一个文件缓冲区	fflush
格式化字符串里的数据	fprintf
从文件里读取格式化输入	fscanf
打印格式化字符串	printf
四字节整型数类型	long
获得某一数字的小数部分	modf
释放已分配的内存空间	free, delete
退出某一函数	return
编写通用类或函数	template
产生随机数	rand
从字符串里读取数据	sscanf
从输入字符流里读取数据	ifstream
从控制台读取数据	scanf
指针移位到文件的某一位置	fseek, fsetpos
跳转到某一语句	goto
大于	>
大于等于	>=
转换为格林威治标准时间	gmtime
异常处理	catch
信号处理	signal
包含某一头文件	#include

要实现的操作	所用元素
计算双曲余弦	cosh
计算双曲正弦	sinh
计算双曲正切	tanh
确定某一变量类型	typeid
从名称域里读入符号	using
包含某一头文件	#include
增大内存块的大小	realloc
自增运算	++
在内存块里检索字符	memchr
检验是否不相等	!=
以内联函数的方式进行编译	inline
从文件里输入二进制数据	fread
以数据流方式从文件里输入数据	ifstream
从键盘输入读取一行数据	gets
以数据流方式从文件里读取对象	istream
从输入的数据流里读取数据	cin
通过文件输入数据	fgets
通过字符串输入数据	sscanf
通过键盘输入数据	scanf
使用文件输入数据流	ifstream
安装信号句柄	signal
整数类型	int
长整型数	long
获取一个数字的整数部分	modf
计算反余弦	acos
计算反正弦	asin
计算反正切	atan, atan2
连接两个字符串	strcat
根据测试结果跳转到相应语句上	switch

要实现的操作	所用元素
在函数之间跳转	longjmp
跳转到某条语句	goto
从键盘输入读取数据的值	scanf
从键盘输入读取一个字符	getchar
从键盘输入读取一个字符串	gets
switch 语句的跳转目标标志	case
声明一个延迟绑定函数	virtual
向左移位	<<
获取字符串的长度	strlen
小于	<
小于等于	<=
读取输入的一行内容	gets
获得当前一行源代码	_LINE_
定义常数列表	enum
设定长跳转的目标语句	setjmp
以 10 为底的对数	log10
自然对数	log
长整数类型	long
反复执行循环体	do, while
根据计数器的值执行循环体	for
直接跳入下一次循环	continue
从循环语句里退出	break
将字符转换为小写	tolower
检验某一字符是否是小写字符	islower
定义宏函数	# define
维护一个程序的多个版本	# if
计算尾数和指数	frexp
屏蔽数据的某些位	bitwise AND (&)
限制对成员的访问	private, protected

要实现的操作	所用元素
允许改变成员的值	mutable
复制内存块	memcpy
在内存块里检索某一字符	memchr
在内存块里设置一个值	memset
改变内存块里的数值	memmove
比较两个内存块里的内容	memcmp
要求刷新内存	volatile
给对象或数据分配内存空间	new
分配原始内存空间	malloc
释放已分配的内存空间	free, delete
执行有模数的除法运算	%, fmod
使不同的程序模块可以共用某些变量	extern
在显示器上显示数据	putchar, puts, printf
打印当前月份	strftime
相乘后赋值给左操作数	* =
两数相乘	*
防止名称混淆	namespace
改变文件名称	rename
允许使用名称域里的名称	using
声明一个名称域	namespace
自然整数数据类型	int
自然对数	log
进行按位取反	~
逻辑非操作	!
检测是否不相等	!=
从字符串里读数	atof, atoi, atol, strtod, strtol, strtoul
从内存里释放某个对象	delete
将对象输入或输出到数据流里	istream, ostream
获取指向当前对象的指针	this

要实现的操作	所用元素
分配内存给一个或多个对象	new
计算某数的反码	~
打开一个文件供数据输入输出操作	fopen
打开一个文件进行数据流的输入	ifstream
打开一个文件进行数据流的输出	ofstream
编写运算符函数	operator
执行按位或(OR)操作	
执行逻辑或操作	
向文件输出二进制数据	fwrite
将对象输出到数据流里	ostream
打印到输出数据流里	cout
输出到文件流对象	ofstream
强制输出到物理设备里	fflush
将变量输出到屏幕上	printf
允许对类成员的外部访问	public
声明重复的数据类型	union
重载运算符	operator
允许外部程序访问类的成员	public
得到圆周率 pi	atan
将字符放回到文件流里	ungetc
获取指针指向的数据	*
改变指针的类型	reinterpret_cast
得到指向当前对象的指针	this
通过指针访问某一成员	->
访问指针所指内容或声明一个指针变量	*
使用“指针到成员”运算符	. * -> *
使用多态类型转换	dynamic_cast
声明多态函数	virtual
获得文件的位置	fgetpos, ftell

要实现的操作	所用元素
设定文件的位置	fsetpos, fseek
计算数的乘方	pow
在函数的调用过程中保护数据不被更改	static
禁止转变为一个变量	const
禁止变量进行类型转换	explicit
把字符输出到文件里	fputc, putc
把字符输出到控制台里	putchar
把数据打印到文件输出流里	ofstream
把格式化的数据打印到字符串里	sprintf
把格式化的数据打印到文件里	fprintf
把格式化的数据打印到控制台里	printf
把字符串打印到文件里	fputs
把字符串打印到控制台里	puts
打印到输出数据流里	cout
打印变量的值	printf
指定私有成员的访问权限	private
允许对类的私有成员进行访问	friend
程序编写的起始点	main
获取程序的运行时间	clock
指定保护类型的访问权限	protected
指定公有模式的访问权限	public
不接受字符	ungetc
使用快速排序算法	qsort
计算数的乘方	pow
计算 e 的乘方	exp
调用异常处理	throw
调用信号处理例程	raise
得到一个随机数	rand
从文件里读入二进制数据	fread

要实现的操作	所用元素
从字符串里读入数据	sscanf
从键盘输入里读入数据	scanf
从文件输入流里读取数据	ifstream
从头文件里读入数据	#include
从文件里读一个字符	fgetc,getc
从键盘输入里读取一个字符	getchar
从文件里读入字符串	fgets
从键盘输入里读入字符串	gets
重新分配内存空间	realloc
再次改变数据的类型	static_cast
再次改变指针的类型	reinterpret_cast
声明引用类型的变量	&
使用一个寄存器变量	register
释放分配的内存	free, delete
释放一个对象	delete
获得除法运算的余数	%, fmod, div, ldiv
从磁盘上移走一个文件	remove
取消宏定义	#undef
更改文件名称	rename
再次打开某个文件	freopen
报告编译的时间	__TIME__, __DATE__
要求使用内存块	malloc, new
限制成员的访问权限	private, protected
返回当前对象	this
返回到主调函数	return
设置返回值	return
对变量某一位的值取反	~
将真/假(true/false)的值取反	!
按位右移	>>

要实现的操作	所用元素
计算平方根	<code>sqrt</code>
向下取整	<code>floor</code>
向上取整	<code>ceil</code>
计算程序运行时间	<code>clock</code>
获得运行时类型的信息	<code>dynamic_cast</code> , <code>typeid</code>
记录长跳转的位置	<code>setjmp</code>
指定类的作用域	<code>::</code>
把数据打印到屏幕上	<code>putchar</code> , <code>puts</code> , <code>printf</code>
在数组里搜索某一个值	<code>bsearch</code>
在字符串里搜索某子串	<code>strstr</code>
设定随机数种子	<code>srand</code>
使用条件跳转逻辑结构	<code>switch</code>
设定二进制位	bitwise OR (<code> </code>)
给文件操作设置缓冲区	<code>setbuf</code> , <code>setvbuf</code>
设置文件位置	<code>fseek</code> , <code>fsetpos</code>
设置长跳转的目标位置	<code>setjmp</code>
设置随机数种子	<code>srand</code>
设置返回值	<code>return</code>
在内存块里设置变量值	<code>memset</code>
声明可共享的变量	<code>extern</code>
移位操作	<code><<</code> , <code>>></code>
在内存块里改变变量值	<code>memmove</code>
短整型数类型	<code>short</code>
安装信号句柄	<code>signal</code>
调用信号处理例程	<code>raise</code>
计算数据类型的长度	<code>sizeof</code>
对数列排序	<code>qsort</code>
获得当前源文件的行号	<code>__LINE__</code>
获得当前源文件名称	<code>__FILE__</code>

要实现的操作	所用元素
指定某一类或名称域	::
计算平方根	sqrt
开始运行一个程序	main
分配静态数据	static
数据流操作(<<,>>)	istream, ostream
使用数据流运算符(<<,>>)	<<,>>
字符串复制	strcpy
从文件里读取字符串	fgets
从键盘输入里读取字符串	gets
把字符串打印到文件里	fputs
把字符串打印到控制台	puts
把格式化的数据打印到字符串里	sprintf
从字符串里读取数据	sscanf
以输入数据流对象的方式用字符串	istream
以输出数据流对象的方式用字符串	ostream
计算字符串长度	strlen
执行字符串运算	str<op>
检测字符串是否相等	strcmp
寻找某个子字符串	strstr
相减后赋值	-- =
数字相减	--
执行从某一变量里减去 1 的操作	--
使用 switch /case 逻辑结构	switch
定义符号常量	# define, enum
允许在名称域里使用符号常量	using
获取系统时间	time
执行系统的外部命令	system
求正切	tan
编写类或函数模版	template

要实现的操作	所用元素
获得临时文件	tmpfile
异常终止	abort
寄存器终止例程	atexit
检验是否相等	==
检验是否大于	>
检验是否大于等于	>=
检验是否不相等	!=
检验是否小于	<
检验是否小于等于	<=
检验单个字符	is<cond>
检验两字符串是否相等	strcmp
根据测试值跳转到相应语句	switch
掷出一个异常	throw
获取时间和日期	time
以格式化方式打印时间和日期	strftime
获取编译的时间	_TIME_
把一个字符串标志化	strtok
使用真/假(true/false)值	bool
关闭某些位	bitwise AND (&)
关闭文件缓冲	setbuf, setvbuf
双字节整型数	short
将数据类型作为参数	template
创建类型定义	typedef
获取类型信息	dynamic_cast, typeid
声明一个新的类型	class, struct, union
取消符号常量的定义	# undef
无符号数据类型	union
将字符转变为大写	toupper
检验某一字符是否是大写	isupper

要实现的操作	所用元素
声明用户自定义的类型	class, struct, union
使用变量长度参数列表	va_arg, va_end, va_start
声明可变的记录类型	union
声明虚函数或基类	virtual
当 while 条件成立时执行循环	while
获取数据类型的宽度	sizeof
向一个文件里写入二进制数据	fwrite
编写类或函数模版	template
向一个字符串写入数据	sprintf
向控制台写入格式化的字符串	printf
向文件输出数据流里写入数据	ofstream
异或 XOR 运算	^

C++ 元素速查表

在这里我们为读者提供了一个按照字母排序的对照表,读者利用它可以迅速地找到在第二部分里介绍的语法表达式和库元素。该表的内容包括了关键字、指令、函数、类和运算符。在左侧是读者待查的元素名称,右侧给出在哪些章节里有对它的介绍。

元素	所在章节
abort	第十五章
abs	第十五章
acos	第十五章
相加(+)	第十一章
取地址符(&)	第十一章
AND 运算符(& 和 &&)	第十一章
访问数组成员([])	第十一章
asctime	第十五章(Time)
asin	第十五章
asm	第十三章
assert	第十五章
赋值号(=)	第十一章
赋值运算符(+ =、- = 等等)	第十一章
atan	第十五章
atan	第十五章
atexit	第十五章
atof	第十五章

元素	所在章节
atoi	第十五章
atol	第十五章
auto	第十三章
位运算符	第十一章
bool	第十章
break	第十三章
bsearch	第十五章
calloc	第十五章
case	第十三章
catch	第十三章
ceil	第十五章
cerr	第十六章
char	第十章
cin	第十六章
class	第十三章
clearerr	第十五章 (Files)
clock	第十五章 (Time)
clog	第十六章
逗号运算符(,)	第十一章
条件运算符(? :)	第十一章
const	第十三章
const_cast	第十二章
continue	第十三章
cos	第十五章
cosh	第十五章
cout	第十六章
eplusplus	第十四章
ctime	第十五章 (Time)
DATE	第十四章

元素	所在章节
自减(--)	第十一章
default	第十三章
# define	第十四章
delete	第十一章
difftime	第十五章(Time)
div	第十五章
除号(/)	第十一章
do	第十三章
double	第十章
dynamic_cast	第十二章
# elif	第十四章
else	第十三章
# else	第十四章
# endif	第十四章
enum	第十三章
检验是否等于(==)	第十一章
# error	第十四章
异或运算符	第十一章
exit	第十五章
exp	第十五章
explicit	第十三章
extern	第十三章
fabs	第十五章
fclose	第十五章(Files)
feof	第十五章(Files)
ferror	第十五章(Files)
fflush	第十五章(Files)
fgetc	第十五章(Files)
fgetpos	第十五章(Files)

元素	所在章节
fgets	第十五章 (Files)
FILE	第十四章
float	第十章
floor	第十五章
fmod	第十五章
fopen	第十五章
for	第十三章
fprintf	第十五章 (Files)
fputc	第十五章 (Files)
fputs	第十五章 (Files)
fread	第十五章 (Files)
free	第十五章
freopen	第十五章 (Files)
frexp	第十五章
friend	第十三章
fscanf	第十五章 (Files)
fseek	第十五章 (Files)
fsetpos	第十五章 (Files)
fstream	第十六章
ftell	第十五章 (Files)
fwrite	第十五章 (Files)
getc	第十五章 (Files)
getchar	第十五章 (Console)
getenv	第十五章
gets	第十五章 (Console)
gmtime	第十五章 (Time)
goto	第十三章
大于(>)	第十一章
if	第十三章

元素	所在章节
#if	第十四章
#ifdef	第十四章
#ifndef	第十四章
ifstream	第十六章
#include	第十四章
自增(++)	第十一章
检测是否不相等(!=)	第十一章
inline	第十三章
int	第十章
is<条件>	第十五章
istream	第十六章
istrstream	第十六章
labs	第十五章
ldexp	第十五章
ldiv	第十五章
小于(<)	第十一章
LINE	第十四章
#line	第十四章
localeconv	第十五章
localtime	第十五章(Time)
log	第十五章
log	第十五章
逻辑运算符	第十一章
long	第十章
longimp	第十五章
main	第十三章
malloc	第十五章
成员访问符(.)	第十一章
通过指针访问成员(->)	第十一章

元素	所在章节
memchr	第十五章
memcmp	第十五章
memcpy	第十五章
memmove	第十五章
memset	第十五章
mktime	第十五章(Time)
modf	第十五章
取模运算符(%)	第十一章
乘号(*)	第十一章
mutable	第十三章
namespace	第十三章
按位取反和逻辑取反	第十一章
new	第十一章
ofstream	第十六章
operator	第十三章
或运算符(和)	第十一章
ostream	第十六章
ostream	第十六章
perror	第十五章
指针间接引用(*)	第十一章
成员指针(.*)	第十一章
pow	第十五章
#pragma	第十四章
printf	第十五章
private	第十三章
protected	第十三章
public	第十三章
putc	第十五章(Files)
putchar	第十五章(Console)

元素	所在章节
puts	第十五章(Console)
qsort	第十五章
raise	第十五章
rand	第十五章
realloc	第十五章
register	第十三章
reinterpret_cast	第十二章
remove	第十五章
rename	第十五章
return	第十三章
rewind	第十五章(Files)
scanf	第十五章
作用域标识符(::)	第十一章
setbuf	第十五章(Files)
setjmp	第十五章
setlocale	第十五章
setvbuf	第十五章(Files)
移位运算符(<<和>>)	第十一章
short	第十章
signal	第十五章
sin	第十五章
sinh	第十五章
sizeof	第十一章
sprintf	第十五章
sqrt	第十五章
srand	第十五章
sscanf	第十五章
static	第十三章
static_cast	第十二章

元素	所在章节
str<op>	第十五章(Strings)
流运算符(<<和>>)	第十三章
strftime	第十五章
strstream	第十六章
strtod	第十五章
strtol	第十五章
strtoul	第十五章
struct	第十三章
减号(-)	第十一章
switch	第十三章
system	第十五章
tan	第十五章
tanh	第十五章
template	第十三章
this	第十三章
throw	第十三章
TIME	第十四章
time	第十五章(Time)
tmpfile	第十五章
tolower	第十五章
toupper	第十五章
try	第十三章
typedef	第十三章
typeid	第十一章
# undef	第十四章
ungetc	第十五章(Files)
union	第十三章
unsigned	第十章
using	第十三章

元素	所在章节
va_arg	第十五章
va_end	第十五章
va_start	第十五章
virtual	第十三章
void	第十三章
volatile	第十三章
while	第十三章
XOR 异或运算符	第十一章

第 十 章

数据类型

C++ 支持一系列基本的数据类型,这些数据类型有时也被称为原始数据类型。它们具有一个共同特点,那就是它们都是由 C++ 语言本身定义的,任何其它类型都是建筑在这些数据类型的基础之上的。表 10-1 总结了 ANSI C++ 里指定的原始数据类型。为了方便读者查阅和使用,表里面还做了一些简单的假设,关于这些内容将在后面的章节里陆续介绍。

表 10-1 所给出的数值范围是根据 Microsoft Visual C++ 的标准设置的。尽管在理论上讲 C++ 的具体实现会根据不同的系统而呈现一定的差异,但总的来说,对于当今的个人电脑和微型计算机而言,这一数值范围不会因 C++ 编译器的不同而相差太远。在表中所列举的数据类型中,整型数的范围一般比较稳定,而浮点数数域的变化最大。对于不同的编译器,其准确的数域范围记录在头文件 limits.h 和 float.h 里面。

表 10-1 C++ 的原始数据类型

数据类型	说明	典型数值范围
bool	布尔值	真(true)或假(false)
char	用于存储单个字符的单字节整型数	-128~127,或 0~255
unsigned char	单字节无符号整型数	0~255
signed char	单字节整型数	-128~127
int	标准长度的整型数,字长为 2 字节或 4 字节	范围与 short 或 long 类型相同
unsigned int	无符号整型数	0~65,535
short	2 字节整型数	-32,768~32,767
unsigned short	2 字节无符号整型数	0~65,535
long	4 字节整型数	大约在正负 20 亿之间

unsigned long	4 字节无符号整型数	大约从 0~40 亿
float	单精度浮点数	在 $10e38$ 的正负 3.4 倍之间
double	双精度浮点数	在 $10e308$ 的正负 1.8 倍之间
long double	长双精度整型数	至少与 double 类型相同
wchar_t	长字符, 使用在国际字符集里 (如 Unicode)	同 unsigned 类型

下面解释一下 C++ 类型的一些技术细节。

- 从纯技术的角度来讲, short、long、signed 和 unsigned 都是修饰词而不是基本类型, 虽然它们已经被假设成 int(整数)型, 但仍然可以和 int 连用。因此, 在书写长整型数类型时, 完全可以把它写为 long int, 虽然这样做没有任何好处。
- 如果关键字 signed(有符号类型)与某个整数类型书写在同一行上, 那么它就是这个整数类型的修饰符。然而除了 signed char(有符号字符类型)之外, 在其它地方根本不必这样使用, 这是因为大多数整数类型的缺省状态即为有符号类型。因此, 长整型数类型可以写为 signed long, 甚至 signed long int。
- 在 ANSI 标准里, 整型数负值的范围实际上比表中所列举的小 1, 例如, -127~127 是 signed char 的实际数域范围, 而不是上面说的从 -128~127。但要是某些编译器使用的是 2 的补码格式(见词汇表), 那么负数域范围就扩展了 1, 表 10-1 里就假设使用的是这一种数据格式。
- 我们可以使用浮点数准确地表达 0, 但受精度限制, 我们无法用计算机处理非常非常接近于 0 的数。例如, 一个 float(浮点数)类型变量不能存储 $1e-50$ (1 除以 10 的 50 次方), 但 double(双精度)类型变量可以做到。在后面对每种数据类型的单独讨论里, 我们将陆续给出这些精度极限。

我们将在后续的几节里系统阐述表 10-1 所列举的数据类型。

整型数和浮点数

C++ 的所有原始数据类型的核心都是数值类型, 甚至包括看似非数值类型的 char(字符类型)和 bool(布尔类型)。C++ 的这一特点反映了数据在计算机里的存储方式: 每一个地址存储一些数据, 而这些数据又可以代表一段代码或其它东西(如字符类型 char)。

除了 float、double 和 long double(长双精度)类型之外,所有的原始数据类型都对应整型数,也就是说它们在内存里是以无小数部分的整数方式存储的。大多数类型,除非冠以 unsigned(无符号)修饰符,都可以既表示正数也表示负数。关于负数的说明和定义方式,请参阅词汇表里“二的补码”部分。

用户也可以对整型常量使用十六进制和八进制标记,该标记可以用于对任何类型的整数进行初始化。

- 如果整型常量前面冠以前缀 0x 则表示它是十六进制常数(逢 16 进位)。它的每一位数字既可以是 0~9 之间的某一个,也可以是 A~F 之间的某一个。在这种情况下,字母的大小写不做区分,一个十六进制数的通用格式为:

0Xdigits

- 如果整型常量前面冠以前缀 0 则表示它是八进制常数(逢 8 进位)。它的每一位数字只可以是 0~7 之间的某一个。八进制数的通用格式是:

0digits

例如,十进制数 255 可以写为 0xFF(十六进制形式)或 0377(八进制形式)。

对于所有整型常量,后缀 U 和 L 分别表示该常量是无符号类型(unsigned)或长整型(long)。用户可以用这两个后缀强制系统以某种格式存储一个数字。例如 63L 或 1024UL,它们分别按长整型和无符号长整型格式存储在内存里。

C++ 允许使用小数点和指数符号来设定某一浮点数常量。用户可以用它们来定义非常大的数或非常小的数(这里所说的小是指接近于 0)。这两个符号我们留到 float 和 double 两节里再详细讲解。需要指出的是,小数点和指数符号只适用于十进制数。后缀 F 可以用来强制指定一个数为浮点数类型(不是双精度类型)。

整型数和浮点数之间存在一定的平衡关系。使用浮点数显得更加灵活,然而在很多场合下根本没有必要存储或处理一个数的小数部分(例如在循环控制中)。在这些情况下,整型数的优点就显现出来,它们不仅效率更高,而且可以准确地存储整数,而不存在浮点数里常有的舍入误差。

bool(布尔类型)

真(true) / 假(false)值

bool 数据类型只有两个值:真(true)和假(false)。用户使用 bool 量一般出于两个

目的:第一,建立复杂的条件判断。第二,作为控制循环操作的标志量。下面的例子显示了如何使用 `bool` 变量建立条件:

```
bool b1, b2, b3;  
b1 = (32 * y) / 2 < x;  
b2 = x < 500;  
b3 = b1 && b2;      // b3 = b1 AND b2  
if (b3)  
    func1();
```

下一个例子使用 `bool` 类型的变量作为标志量,该变量被初始化为 `true`:

```
bool do_more = true;  
while (do_more) {  
    // Do some stuff...  
    if (x > 100)  
        do_more = false;  
    // Do some more stuff...  
}
```

在 C 和 C++ 里,布尔值的传统处理方式是把 `true` 或 `false` 的值赋给整型数。任何不为 0 的整型数值都被 `if`、`while` 和 `for` 语句视为真(`true`),但在关系运算符和逻辑运算符里返回的布尔值只有 1 或 0。

现在我们最好使用 `bool` 数据类型来存储一个表示真或假的布尔量。你也完全可以把一个整数赋值给 `bool` 类型的变量(如果你不使用类型转换的话,编译器将给出一条警告消息),这是任何一个非零的数值都将被自动转变为 `true(1)`。

● ANSI

`bool` 数据类型是 ANSI 的一个扩展类型,在早期的 C++ 版本里不支持这种类型。

char

单字节字符

从纯技术的角度来讲,`char` 类型所存储的数据完全类似于整数类型所存储的数据,但从使用上我们是用它来存储一个字符的。如前面所述,由于计算机里实际上是以数值代码存储字符,因此这种从字符到数字的转换是自动进行的。例如,如果你声明一个 `char` 类型的变量并且把一个字符赋给它,计算机实际接收和存储的是一个数字。如果你要知道这个数字到底是多少,可以查阅附录 D 里给出的 ASCII 字符集。

```
char ch;  
ch = 'A';    // Uppercase 'A' represented by  
             // integer value 65.
```

在实际使用中,我们很少用 `char` 数据类型来存储单个字符,一般都使用 `int` 或 `short` 类型来完成这一工作。这是因为 `int` 类型可以存储一些特别的值,如 `EOF` 等,这些特殊字符经常出现在针对字符操作的函数里。

`char` 类型主要用于构成字符串(即字符组成的数组)。由于字符串可能包含很多的字符,因此 `char` 类型长度短的优点在这里就显得非常突出了。对字符数组我们可以直接使用一串字母来对它初始化。在下面的例子里,我们给字符数组分配了 20 个字节的存储空间,并且它的前三个字符已经进行了初始化。

```
char message[20] = "Hit";
```

在下一个例子里,编译器为字符数组分配的空间恰恰够存储后面的字符串(包括最后一位结束标志符)。

```
char message2[] = "This is a warning.";
```

根据编译器的具体实现情况的不同,`char` 类型要么与 `signed char`(有符号字符)类型等同,要么与 `unsigned char`(无符号字符)类型等同。

● 注意

在一些字符集里,尤其是应用程序的国际发行版本里使用的字符集 `Unicode`,需要编译器提供更大的存储空间来存储一个字符。为支持这种特殊的字符集,`C++` 还提供了 `wchar_t` 类型。

在使用 `char` 值时,你可以把一个标准的、可打印输出的字符括在单引号或双引号里(见前面的例子所示)进行赋值等操作,也可以使用表 10-2 所列举的转义字符,其中最常用的是换行符“`\n`”。在 `C` 和 `C++` 的字符串里反斜杠“`\`”有着特殊的意义,因此要想把反斜杠作为字符使用,就必须写成“`\\`”格式,而不是“`\`”。

表 10-2 的最后两行给出了如何在字符串里插入 `ASCII` 字符的值。例如,你可以在字符串里使用值 255(十六进制表示为 `FF`)表示一个 `char` 类型字符:

```
char s[] = "This has a funny \xFF character inside.";
```


表 10-2 转义字符

字符	含义
\a	鸣叫(警告)
\b	退格
\f	进纸(换页)
\n	换行
\r	回车
\t	跳表
\v	纵向跳表
\'	单引号
\"	双引号
\\	反斜杠字符
\0	空值(0)
\ddd	八进制表示的 ASCII 字符
\xdd	十六进制表示的 ASCII 字符

unsigned char**无符号单字节整型数**

unsigned char 类型可以表示一个从 0~255 之间的无符号数值。同 char 类型一样, unsigned char 主要也是用于定义数组而很少表示单个字符。unsigned char 类型经常用于定义存储二进制数据的缓冲区, char 类型也可以这样使用, 但由于 unsigned char 类型的数域范围固定, 因此使用 unsigned char 更容易进行处理。

下面的例子是使用 unsigned char 定义一个存放二进制数据的缓冲区:

```
unsigned char buffer[1000];
```

与 char 类型不同的是, 在哪种编译器里 unsigned char 的数域范围都保持在 0~255。因此使用 unsigned char 存储数据将更加方便, 尤其是存储从 0x0~0xFF 之间十六进制数。

signed char**有符号单字节整型数**

signed char 类型可以表示一个从 -128~127 之间的无符号数值。在很多情况下,

它等同于 char 类型,但并不能保证绝对是这样。同 char 和 unsigned char 类型一样, signed char 也可以用来定义原始数据缓冲区。

```
signed char buffer[1000];
```

在大多数情况下, char、signed char 和 unsigned char 类型的数据具有相同的特性。然而,当你把一个单字节的数赋值给一个大整型数域时,便会看到它们在符号扩展上的差异(关于这一问题的论述,请参阅词汇表里的“符号扩展”部分)。在下面的例子里,我们把二进制数 11111111 赋值给一个 signed char 类型的数和一个 unsigned char 类型的数,两者的结果截然不同:

```
signed char sbuf[100];
unsigned char ubuf[100];
sbuf[0] = 255;           // All bits set to 1; represents -1.
ubuf[0] = 255;           // All bits set to 1; represents 255.
short i = sbuf[0];       // Extend to 11111111 11111111.
short j = ubuf[0];       // Extend to 00000000 11111111.
```

char、signed char 和 unsigned char 的另外一个差别表现在当把一个介于 128 和 255 之间的数赋值给 signed char 类型的变量时,编译器必须先进行数值转化,同样还会出现一条恼人的警告信息。如果你要使用十六进制数(如 0xFF,等于十进制数 255)对变量进行赋值的话,使用 unsigned char 类型要更方便一些。

int

有符号整型数

int 类型是 C 和 C++ 里的“自然”整数类型。对于 int 型的整数我们唯一能确定的是:它必然不小于 short 类型,不大于 long 类型。也就是说,它要么是两个字节长(在 16 位系统里),要么是四个字节长(在 32 位系统里)。

你可以使用 int 类型来声明一个有符号整型数。它的最小值为 -32,768,最大值为 32,767。在 32 位系统里面,int 类型的字长与 long 类型一样。

```
int i, j, kount, hexno, octno;
i = 25;
j = -1;
kount = 10500;
hexno = 0xFF;    // hex FF = 255;
octno = 010;     // octal 10 = 8;
```

对于 int 数据类型比较特殊的一点是:虽然它的数域范围可以同 long 类型一样

大,但如果你想要编写短小精悍的程序代码,还得要假定 `int` 类型的字长同 `short` 类型一样。因此,如果你真要力图编写代码简练的程序,`int` 类型一点也不比 `short` 类型强。另外,使用 `int` 类型也存在一定的风险性,因为在 32 位系统里运行正常的 `int` 类型数据,放到其它低位系统的编译器里有可能出现数值越界的错误。

对于处理器而言,根据寄存器的大小存取 `int` 类型的变量是非常自然的事情。然而,这种做法的优越性却不得不受到质疑,因此在编写某些非常重要的程序时,有经验的程序员会避免使用 `int` 类型的变量。本书的作者就是这样做的,在编写小程序时使用 `int` 类型的变量,而在开发复杂、重要的程序时不使用它。然而在 C 和 C++ 的标准库里,很多函数和类的编写都使用的是 `int` 类型,所以读者在使用时也许会遇到一些麻烦。较好的做法莫过于用 `short` 类型取代这些 `int` 类型。

unsigned int

无符号整型数

`unsigned int` 类型同 `int` 类型具有相同的长度,它们在 16 位系统里都是双字节的,在 32 位系统里都是 4 字节的。前面所说的适用于 `int` 类型的东西大多数都使用于 `unsigned int` 类型。但需要注意的是 `unsigned int` 不能存储负号。由于它的长度不定,因此也不大适合编写代码简练的程序。在这种情况下最好使用 `unsigned short` (无符号短整型)或 `unsigned long` (无符号长整型)。

`unsigned int` 的最小数域范围是从 0~65,535。在 32 位系统里,它的数域范围同 `unsigned long` 一样。

```
unsigned int i = 55;  
unsigned int j = 60100;  
unsigned int mask = 0xFFFA;
```

`unsigned int` 类型的一个有趣的特征是它适合进行位域操作。位域操作应该建立在一个无符号数据成员上,因为只有对于无符号数而言,位的模式才能得到一致的解釋。例如,由于位域里没有符号位,字长为三位的数值 111 才能被正确处理成 9 而不是 -1。

```
struct card {  
    unsigned int rank; 4;  
    unsigned int suit; 2;  
    unsigned int marked; 1;  
} hand[5];
```

前面提到的关于位域的声明,请参阅第二章的最后一节。如果读者想要了解后

缀 U(无符号标志)的使用方法,请参阅 unsigned short 部分。

short

有符号双字节整型数

short 类型是双字节的整型数,它可以存储负数,其数值范围为 $-32,768 \sim 32,767$ 。short 类型比 int 类型更加稳定。

你可以在它的数域范围内用 short 类型来声明有符号整型数。

```
short i = 2500, j = -5, k = -1000;
```

上面所说的 short 类型的数域范围适用于那些使用二的补码算法的操作平台,当今几乎所有个人电脑都采用这种数字构造方式。在其它系统里,按照 ANSI 标准,它们所支持的 short 类型数域范围应当是从 $-32,767 \sim 32,767$ 。

缺省情况下,只要一个整型常数在以上的范围之内,编译器就以 short 类型对它进行存储。

unsigned short

无符号双字节整型数

unsigned short 类型表示一个双字节的整型数,它不能用来存储负数。unsigned short 的数域范围是从 $0 \sim 65,535$,你可以使用这一类型来声明一个在此范围之内的无符号整型数。

```
unsigned short i = 251;  
unsigned short j = 25000;
```

尽管 unsigned short 类型不能存储负数,但它所能存储整数的数量比 short 类型可存储的要大一倍。一般来说,如果你使用的整型数接近 unsigned short 类型的上界,最好在它的前面加上 long 关键字。unsigned short 类型比较紧凑,因此在很多场合下它都非常实用。

如果一个整型常数在 short 类型的数域范围之内,编译器在缺省情况下就以 short 类型来存储它,如果整型常数超出了这一范围,编译器就将以 long 类型来存储它。然而用户完全可以使用后缀 U 来要求编译器以 unsigned short 类型来存储该常量,例如:

```
unsigned short j = 65000U;
```

如果你忘记了加上后缀 U,仍然不会有太大问题。不论该常量 65000 以何种方

式存储在计算机里,它都可以被赋值给 unsigned short 类型变量里,而不会损失任何信息:

```
unsigned short j = 65000;
```

long

有符号四字节整型数

long 类型是四字节的整型数,它可以存储负数,其数值范围为 $-2,147,483,648 \sim 2,147,483,647$ (大约在正负二十亿之间)。long 类型是 ANSI 标准下最大的整型数类型,虽然有些编译器里可能还支持更大的整型数。

你可以在 long 类型的数域范围内用 long 关键字声明有符号整型数。

```
long i = 350100, j = -5, k = -123000555;
```

上面所说的 long 类型的数域范围适用于那些使用二的补码算法的操作平台,当今几乎所有个人电脑都采用这种数字构造方式。在其它系统里,按照 ANSI 标准,它们所支持的 long 类型的数域范围的最小值为 $-2,147,483,647$, 比上面的最小值大 1。

你可以在整型常量的尾部加上后缀 L 来指定编译器以 long 类型存储该常量(即使这个数不大也可以这么做)。如果不使用这一后缀,只要常数在 short 类型的数域范围内,编译器就会以 short 类型存储它。

```
long i = 5L; // 5L is stored as a long.
```

在上面的例子里,使不使用后缀 L 差别不大,因为在表达式里已经把常数赋值给已经声明为 long 类型的变量 i 了。然而,在其他一些情况下使不使用后缀 L 就有差别了:比如说在函数调用里,你需要确保被指向的数据具有特定的字节长度(请参阅前面章节里关于后缀 U 使用情况的说明)。

unsigned long

无符号四字节整型数

unsigned long 类型是四字节的整型数,它不能用来存储负数,其数值范围为 $0 \sim 4,294,967,296$ (大约四十亿)。你可以在 unsigned long 类型的数域范围内用 unsigned long 关键字来声明有符号整型数。

```
unsigned long i = 500;  
unsigned long j = 4123000123;
```

尽管 unsigned long 类型不能被用来存储负数,但它所能存储的整数的数目比 long 类型要大一倍。因此 unsigned long 类型提供了 ANSI 标准里最大的整型数范围。

float

单精度浮点数

float 类型是四字节的浮点数。在 ANSI 标准里, float 类型的数域范围是浮点数范围里最小的,因此 float 类型不如 double 类型使用得普遍(请参阅本节最后的“注意”)。

float 类型数值的范围在正负 3.4×10^{38} 之间。每个 float 类型的数值具有不小于六位的数字精度。

一个 float 值可以精确表示 0,也可以表示某些非常接近于 0 的数字,但具体精度受到下面这一条件的约束:float 值所能存储的最小数字为正负 1.175×10^{38} (10^{-38} 就是 $1/10^{38}$)。

用户既可以用小数点形式也可以用指数形式来建立浮点数常量,指数形式的构造方式为“numExp”,表示 num 乘以 10 的 exp 次方(在这个表示式里,既可以使用大写“E”,也可以使用小写“e”)。用户还需要使用后缀 F 来通知编译器以 float 类型存储该常量。例如:

```
float amt = 0.0F;           // amt assigned zero precisely
float x = 25.72F;           // Standard decimal form.
float y = 3.5e2F;           // y = 350.0
float tiny = 3.1e-3F;       // tiny = 0.0031
float biggie = 2e5F;         // biggie = 200,000.0
```

如果用户不使用后缀 F 或不加上“(float)”进行强制类型转换,编译器将以 double 类型来存储常量,同时还会给出警告消息——提示用户在把常量(double 类型)赋值给变量(float 类型)时有可能出现数据丢失。

一些编译器可能了解用户这样做的目的而不给出警告,但我们不能对编译器过分依赖。因此,最好的策略还是在对 float 类型变量初始化的过程中使用后缀 F。

● 注意

在大多数浮点运算里, float 类型都不是最好的选择。当 C++ 计算浮点表达式的值时,为提高运算精度,编译器很可能把 float 类型变换为 double 类型。float

类型主要用于声明大的数组,由于它的字长小,因此占用的存储空间也小。另外,如果浮点数所存储的信息还要转存到磁盘上,那么使用 float 就更加合适了。

double

双精度浮点数

double 类型是八字节的浮点数,它是 C++ 里的标准浮点类型。在很多表达式里, float 类型的数据都会被转换为 double 类型进行存储和运算。

double 类型的数域范围在正负 1.797693×10^{308} 之间,它具有至少十位的数字精度。如此大的数值范围相信可以满足大多数人的要求。

double 类型可以准确地表示 0,它也可以表示非常趋近于 0 的小数。double 类型可存储的最小数字为正负 $2.225074 \times 10^{-308}$ (10^{-308} 就是 $1/10^{308}$)。这个数字的确非常非常小。

用户既可以用小数点形式也可以用指数形式来建立双精度浮点数常量,指数形式的构造方式为“numEexp”,表示 num 乘以 10 的 exp 次方(在这种表示式里,既可以使用大写“E”,也可以使用小写“e”)。例如:

```
double amt = 0.0;           // amt assigned zero precisely
double x = 25.72;           // Standard decimal form.
double y = 3.5e2;            // y = 350.0
double tiny = 3.1e-3;        // tiny = 0.0031
double biggie = 2e5;          // biggie = 200,000
```

因为 double 类型是浮点数的缺省类型,所以以上所有语句都是以 double 类型存储某一常量。即使那些处于 float 类型数域范围的浮点数(如上面的常数)也会以 double 类型存储在计算机里。

long double

超长浮点数

除 float 和 double 类型外,根据 ANSI 规范,C++ 还提供了 long double 浮点数类型。然而,对于 long double 类型而言,我们唯一能确定的是它的字长不会小于 double 类型,也就是说,它至少具有与 double 类型一样的数域范围和可表示的最小值。如果你希望自己的程序代码简练,就必须对 long double 类型的某些特性作出一定的假设,否则就很难充分利用它(你可以查看所使用编译器的帮助文档,从而确定 long double 类型是否提供优于 double 类型的精度和数域范围)。但有一点还是确定无疑的,那就是如果你需要使用编译器可以提供的、最大精度的浮点数的话, long double

类型便是最好的选择。

wchar_t

长字符

wchar_t 数据类型是整型数类型的一种,它用来存储那些双字节字符而不是单字节字符。较长的字节数可以支持在应用程序的国际发布版本里所使用的扩展字符集(如常用的 Unicode 字符集)。

因为字符串是 char 类型的数组,所以对 wchar_t 类型字符串的初始化不象对普通字符串初始化那样简单。下面的程序代码是对 wchar_t 字符串初始化的例子,一次只能对字符串里的一个字符赋值。如果你的程序里需要大量使用 wchar_t 类型的字符串,那么最好还是编写一个函数简化相应的初始化操作。

```
wchar_t wstr[10];  
wstr[0] = 'T';  
wstr[1] = 'h';  
wstr[2] = 'e';
```

尽管 wchar_t 类型是根据实现方式而定义的,但它经常以 unsigned short 类型实现。

● ANSI

wchar_t 数据类型是 ANSI 的扩展功能,在早期版本的 C++ 里不支持此种数据类型。

第十一章

运算符

C++ 里支持大量的运算符,它们用于把简单的表达式合并为大的表达式(每一个变量、常量都可以被看作是一个简单表达式)。这些使用运算符的表达式被称为该运算符的操作数。

运算符可以具有一个操作数、两个操作数和三个操作数。一些运算符只能针对一个操作数使用,它们被称为单操作数运算符,还有一些运算符只能使用在两个操作数里,它们被称为双操作数运算符。条件运算符(?)是唯一的一个三操作数运算符,它具有三个操作数。

表 11-1 按照优先级顺序列举出了 C++ 的运算符,具有同一优先级的运算符放在一行里,具有最高优先级的运算符列在表的顶端。除特别声明外,所有的运算符都具有双操作数,并且其结合性都是从左到右。

表 11-1 C++ 的运算符一览表

结合性(类型)	运算符
	() [] -> :: .
从右到左(单操作数)	! ~ ++ -- - * & sizeof new delete typeid casts . * -> * * / % + - << >> < <= > >= == != & . &&

从右到左(三操作数)	?:
从右到左	= + = - = * = / = % = >> = << =
	& = ~ = =
	,

表 11-2 列举了运算符、运算符的简短说明和相应的语法表示式。优先级顺序依次从最高级(第一级)到最低级(第十六级)。具有最高优先级的运算符在表达式里出现时首先被计算。对于结合性而言,除了在优先级数字后面标明“R”的结合顺序为从右到左之外,其它均为从左到右(例如,“2R”表示该运算符的优先级为二级,结合顺序为从右到左)。

表 11-2 C++ 的运算符及说明

优先级别	运算符	说明	语法表达式
1	()	函数调用	func(args)
1	[]	访问数组成员	array[int]
1	->	访问成员	ptr->member
1	.	访问成员	obj.member
1	::	作用域标识	class::symbol ::symbol
2R	!	逻辑取反	! int
2R	~	按位取反	~ int
2R	++	自增 1	++ lval lval ++
2R	--	自减 1	lval lval--
2R		算术负号	-num
2R	*	指针引用	* ptr
2R	&	取地址	&lval
2R	sizeof	计算数据长度	sizeof(expr) sizeof(type)

2R	new	分配数据	new type new type(args) new type[size]
2R	delete	删除数据	delete ptr delete [] ptr
2R	typeid	获得类型信息	typeid(expr)
2R	casts	类型转换	见第十二章
3	. *	指向成员的指针	obj. * ptr_mem
3	-> *	指向成员的指针	ptr-> * ptr_mem
4	*	乘号	num * num
4	/	除号	num / num
4	%	取模符号(求余数)	int % int
5	+	加号	expr + expr
5	-	减号	expr - expr
6	<<	向左移位	expr << int
6	>>	向右移位	expr >> int
7	<	小于	expr < expr
7	<=	小于等于	expr <= expr
7	>	大于	expr > expr
7	>=	大于等于	expr >= expr
8	==	等于	expr == expr
8	!=	不等于	expr != expr
9	&	按位与(AND)	int & int
10	^	按位异或(XOR)	int ^ int
11		按位或(OR)	int int
12	&&	逻辑与(AND)	expr && expr
13		逻辑或(OR)	expr expr
14R	?:	条件运算符	expr ? expr : expr
15R	=	赋值号	lval = expr
15R	+=	相加后赋值	lval += expr

15R	--	相减后赋值	lval -= expr
15R	*=	相乘后赋值	lval *= expr
15R	/=	相除后赋值	lval /= expr
15R	%=	模除后赋值	lval %= expr
15R	>>=	右移后赋值	lval >>= int
15R	<<=	左移后赋值	lval <<= int
15R	&=	按位与(AND)后赋值	lval &= int
15R	^=	按位异或(XOR)后赋值	lval ^= int
15R	=	按位或(OR)后赋值	lval = int
16	,	逗号运算符(返回 expr2 的值)	expr1, expr2

表 11-2 里我们使用了几十个标志符作为对语法表达式的解释,下面是这些符号的说明:

标志符	说明
expr	表达式:既包括数学表达式,也包括地址表达式。例如,加法(+)的语法表达式为 <code>expr + expr</code> ,这是因为在 C++ 里的加法运算的操作数是表达式,如 <code>array + 5</code> 。
num	数字;它不包括地址表达式。
int	整型数;它包括任何整数类型的表达式。
lval	左值:它可以是出现在赋值号左侧的任何一种类型的表达式(请参阅词汇表的“左值”部分)。最常用的左值是变量。
ptr	指针;它可以是任何一个地址表达式。

在这一章后面将陆续向读者介绍这些运算符:

- 运算符关键字(`sizeof`、`new`、`delete`、`typeid`)
- 赋值运算符
- 位运算符
- 逗号运算符(,)
- 条件运算符(?:)
- 自减运算符(--)
- 自增运算符(++)

- 逻辑运算符
- 模除运算符(%)
- 指针运算符
- 指向某一成员的运算符
- 关系运算符
- 作用域标识符(::)

sizeof**计算数据结构的字节长度**

sizeof 运算符返回一个表达式或某一类型的字节大小。它主要有两种语法格式:

```
sizeof(expression)  
sizeof(type)
```

sizeof 运算符的操作数既可以是原始数据类型,也可以是复合数据类型,对于复合类型的处理一般遵循下面几条原则:

如果操作数是静态分配的数组, sizeof 将把整个数组的字节长度作为结果返回。

如果操作数是一个指针而不是数组, sizeof 将把该指针的长度,即地址的长度作为结果返回。

如果操作数是一个数组,但该数组的长度在编译的时候不能确定下来,那么 sizeof 运算就不能对该数组进行。

如果操作数是一个类、结构、联合或是以上三者中某一个的实例, sizeof 将把这三种数据结构的一个实例的字节长度作为结果返回(不管该实例里面的内容是否进行过初始化)。

下面的例子显示以上规则的实际应用情况:

```
#include <iostream, h>  
//...  
cout << sizeof(short) << '\n';           // Prints 2  
cout << sizeof(short*) << '\n';          // Prints 2 or 4  
cout << sizeof(short[10]) << '\n';        // Prints 20  
  
long arr[100];  
long *p = arr;  
cout << sizeof(arr[1]) << '\n';          // Prints 4
```

```
cout << sizeof(p) << '\n';           // Prints 2 or 4
cout << sizeof(arr) << '\n';         // Prints 400
```

● 注意

sizeof 运算符在实际使用时并不需要计算某一个字符串的真实长度。sizeof 要么返回地址长度(如果操作数是一个指针),要么返回所分配内存空间的总长度(如果操作数是一个数组)。如果用户想要得到字符串的当前长度,可以使用标准库函数 strlen。

new

动态分配数据

new 运算符根据某种数据类型的实际长度在内存里分配相应的空间来创建一个或多个该数据类型的实例。new 运算符具有两种特殊功能:它可以自动返回一个指向某种数据类型的指针,如果某一数据类型为类,在执行过程中 new 运算符还将调用相应的构造函数。使用 new 运算符和直接声明一个对象都可以创建某种数据结构的实现,但前者优于后者的一点在于:使用 new 运算符,用户可以完全控制数据的生存周期,如果你想要销毁该数据实例,可以使用 delete 运算符(在下一节里我们会详细介绍)。

new 运算符具有如下的几种语法结构:

```
new type
new type(args)
new type[size]
```

不管在上面哪种语法结构里,new 运算符都将返回一个指向操作数据的指针,并且该指针的类型为 type *。例如,在下面语句里使用的 new 运算符都将返回一个类型为 int * 的指针:

```
int *p1, *p2;
p1 = new int;           // Allocate one int.
p2 = new int[50];       // Allocate 50 ints.
```

如果用户使用 new 运算符来分配某一对象,那么还可以在表达式里使用初始化参数。这些参数在运行期间被传递给相应的构造函数。例如,假设类 CAuto 定义了几个构造函数。

```
CAuto *pcar1, *pcar2, *pcar3;
pcar1 = new CAuto;
```

```
pcar2 = new CAuto("Jag", 97);  
pcar3 = new CAuto[10];
```

上面语句中的第二行调用了 CAuto 类的缺省构造函数,第三行调用了类型为 CAuto(char *, int)的构造函数,最后一行创建了多个对象,因此调用了十次类的缺省构造函数。

size 参数可以是任何数值表达式,如某一变量。换句话说,用户可以动态地定义数组的大小。

● 注意

我们在使用 new 运算符时,完全可以用 new type[size] 这样的语法表达式来创建类的“一个”对象,例如 new CAuto[1]。如果你使用了这种语句,在销毁该对象时必须使用 delete [] type,而不是 delete type,在下一节里我们会详细讨论 delete 的用法。

delete

释放已分配的数据

delete 运算符是用来删除对象和数据的,而这些对象和数据又是在前面由 new 运算符分配得到的。delete 运算符的操作数是一个指针,该指针所存储的地址必须是前面某一 new 运算符返回的地址,否则就会导致结果出错。delete 运算符具有两种语法结构:

```
delete pointer;  
delete [] pointer;
```

在上面的例子里,如果 pointer 在前面是 new type[] 语法表达式的返回值,就必须使用第二条语句。否则使用上面的哪一条语句都是可以的。例如:

```
int * p1, * p2;  
p1 = new int;  
p2 = new int[100];  
//...  
delete p1;  
delete [] p2;
```

用户在使用 delete 语句删除对象时,该运算符还调用了相应的对象销毁函数。C++ 在这里使用的术语——销毁函数 (destructor) 可能听起来带有一定的破坏性,但该函数同其它函数没太大不同,它只是在对象使用完毕时执行一些例行的操作(请参阅词汇表里面的“销毁函数”部分)。

typeid**获得类型信息**

`typeid` 运算符根据运行时类型信息 (RTTI) 来决定其操作数在程序运行时刻的类型。在编程中我们不常使用 `typeid` 运算符,这是因为在绝大多数情况下程序员都清楚自己所使用的数据类型。尽管如此,`typeid` 运算符在某些情况下还是有一定作用的。例如,如果某一个表达式非常复杂而你又想要确定它的返回值类型,就可以使用 `typeid` 语句。

`typeid` 运算符经常在指向某一对象的指针里使用。当某一类具有多态特性时,即它的基类包含至少一个虚函数,使用 `typeid` 运算符就可以得到对象所指的 actual 类型 (详情见本节的示例程序)。`typeid` 运算符的语法表达式为:

```
#include <typeinfo.h>
```

```
typeid(expression | type)
```

上面的语法表达式指出 `typeid` 的参数既可以是一个表达式,也可以是一个类型的名称。使用 `typeid` 运算符时必须要把头文件 `typeinfo.h` 包含到程序里面。下面的程序段显示了如何使用 `typeid` 来准确判断指针 `pb` (基类指针) 所指的 object 属于哪一个类:

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

```
#include <typeinfo.h>
```

```
class B {
```

```
public:
```

```
    virtual int func1(void) { return 0; }
```

```
};
```

```
class D : public B {
```

```
public:
```

```
    int func1(void) { return 1; }
```

```
    int func2(void) { return 2; }
```

```
};
```

```
void main() {
```

```
    B * pb;
```

```
    D objd;
```

```
    pb = &objd;
```

```
    cout << typeid(objd).name() << '\n';
```

```
    cout << typeid(pb).name() << '\n';
```

```
    cout << typeid(*pb).name() << '\n';  
}
```

上面程序运行结束后将打印出如下的计算结果:

```
class D  
class B *  
class D
```

在上面的程序里,最关键的一行是最后一条语句,它的输出结果是字符串“class D”。我们最感兴趣的是 typeid 运算符是如何确定出来 *pb 所指向的是类 D 的对象而不是其它类的对象呢?即使我们把 pb 声明为 B* 类型,结果也不会出错。

● 注意

如果用户要使用运行时类型信息,必须把编译器的某些开关设置到合适位置。例如,在 Microsoft Visual C++ 里面,用户需要使用 /GR 选项作为命令行参数。具体情况请参阅编译器的帮助文档。

typeid 运算符返回一个特定类型的对象——type_info,该对象是在 typeinfo.h 里定义的。type_info 类型可以支持两种运算符(“==”和“!=”)和两个成员函数(before 和 name)。其中,before 函数是在对象内部使用的,不能由用户调用,而其它三种运算和操作可以按照下面的例子使用:

用法	运行结果
<code>typeid(a) == typeid(b)</code>	如果 a 和 b 的类型一致就返回 true
<code>typeid(a) != typeid(b)</code>	如果 a 和 b 的类型不一致就返回 true
<code>typeid(a).name()</code>	返回包含类型名称的一个字符串

typeid 最大的用途是检验一个对象指针的类型(该指针作为参数传递给 typeid)。这里 typeid 的用法类似于 dynamic_cast 运算符,我们将在第十二章里讲解 dynamic_cast。例如,你可以使用 typeid 来检验参数 pb 是否指向一个类型 D 的对象,如果 pb 确实指向类型 D,那么就可以安全地调用那些只在 D 里面声明过的函数。(如果你要调用 D 里的函数 func2,在这里还必须使用 reinterpret_cast。如果前面已经使用了 dynamic_cast,那么在这里就不必再使用 typeid 或 reinterpret_cast 了。)

```
void test_func(B *pb) {  
    if (typeid(*pb) == typeid(D)) {  
        D *pd = reinterpret_cast<D*>(pb);  
        pd->func2();  
    }  
}
```

```
//...
```

对于 typeid 的返回值所进行的最常用的操作是检验类型是否相同(==)。调用 name 函数(如前面示例)可以让用户打印出类型名称,这一点对于进行调试非常有用。

● ANSI

typeid 运算符是 ANSI 的扩展功能,在所有早期版本的 C++ 里都不支持这一运算符。typeid 对名为 meta-data 的数据结构加以描述,借此获得对象本身的信息。这里新增添的 meta-data 数据结构为 C++ 带来了一些新的功能,在其它面向对象的语言里,如 SmallTalk,也都具有类似的功能。

赋值运算符

同其它计算机语言一样,C 和 C++ 都支持一个简单的赋值运算符。然而在 C / C++ 里,该运算符却颇有些古怪之处:首先,用户必须仔细区分赋值号(=)和检验是否相等符号(==)。其次,赋值号同其它运算符虽然没有太大不同,但是它要返回一个数值-即所要赋予变量的数值。

```
a = b;      // Assign b to a and return b.
```

该数值也可以用在复杂一些的表达式里。例如,下面的语句把数值 b 赋给 a,然后把同一值赋给 c:

```
c = a = b;
```

除了简单的赋值操作之外,C 和 C++ 还支持大量的其它赋值运算符。这些运算符对两个操作数执行某些运算之后再将其结果赋值给左操作数。一般来说,赋值运算符具有如下的使用格式:

```
a assignment-op b
```

上面的语句等同于:

```
a = a op b
```

例如,下面的两条语句具有相同的操作结果:

```
x += y;  
x = x + y;
```

同样,下面的两条语句也是等效的:

```
x *= y;
```

```
x = x * y;
```

对于赋值运算符有一些显而易见的约束条件。首先,所进行的运算必须使用于两个操作数。其次,左侧的操作数必须可以被合法地赋值(即它可以作为左操作数出现)。例如,你可以对一个变量赋值,但不能对某一常量赋值。

在所有情况下,赋值运算符都把所赋的值作为结果返回。

位运算符

位运算符所执行的操作正如它的名称所指明的那样:它们所处理的对象是两个数字的某些位。而这两个数字必须都是整数类型,如 `int`、`short`、`long` 或 `unsigned int`。只有当你想要对某些单独的位进行处理时才可以使用这些位运算符。如果你想要把布尔类型的数值合并起来,如关系运算符所得到的结果,那最好还是使用逻辑运算符(请参阅本章“逻辑运算符”部分)。

所有的位运算符都要首先比较两个操作数里出现的相同位置的某一位。例如,在进行按位与(`AND`)操作时,程序依次比较第一个操作数的某一位同第二个操作数的相应位,如果两者有一个是 0,与运算的结果都将等于 0。这样的比较将分别针对第一位、第二位……依次类推(见图 11-1)。

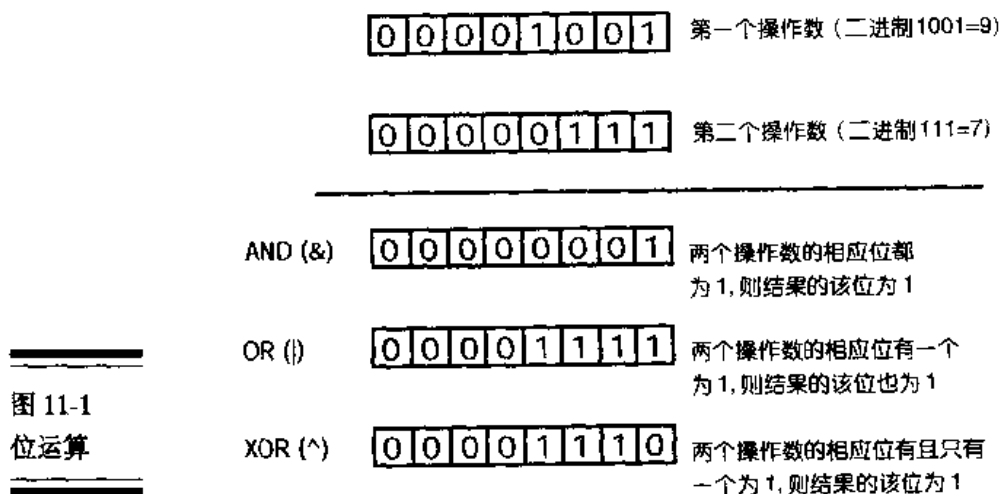


图 11-1

位运算

位运算主要用在位屏蔽的处理上。按位与(`&&`)运算设定了一个屏蔽参数,进行与运算之后,只有那些在屏蔽参数和待处理值的对应位置均为 1 的位保持 1,其它各位都设置为 0。而按位或(`||`)运算则可以无条件地把某些位打开(即把该位置的值置成 1)。关于这些内容,读者可以参阅一下前面第二章里的一个例子,那里给出了位屏蔽是如何在程序里发挥作用的。

图 11-1 显示了按位与、或和异或的计算。

逗号运算符(,)

按顺序计算表达式的值

逗号运算符的使用非常简单,它按顺序计算两个表达式的值,并且把第二个表达式的值作为结果返回。逗号运算符同其它运算符一样,可以把两个表达式连接成为一个大的表达式。

expr1, expr2

C 和 C++ 里提供的逗号运算符可以使我们在返回运算结果之前计算多个表达式的值。这一点看起来可能有点象是在行骗。我们可以用逗号运算符把很多表达式连成一串,而不管前面的表达式计算出来什么值,程序都将返回最后一个表达式的结果。不过在编程时这种操作还是有用的,比如说当你想要把一系列表达式运算放到一块,而只取其中的一个表达式的结果单独处理,就可以使用逗号运算符(在大多数其它算法语言里,这种操作一般需要使用多条语句)。例如:

```
while (a = func1(), b = a * 2, b > 0) {  
    // Do some stuff ...  
}
```

这里使用的逗号运算符把几个表达式合并成一个表达式。该程序运行时,所有的表达式的值都将按以下顺序计算出来。

*a = func1(), b = a * 2, b > 0*

这段代码的优点在于它允许你先进行一些运算,然后再计算循环条件(*b > 0*)。从程序来看,虽然循环的条件判断是在每次循环开始之前进行计算的,但实际上不然,在每次计算循环条件之前都有一些其它的运算先被执行。因此,如前面所说的,这种操作很有些欺骗的味道,因为它允许用户回避一些正规的语法限制。

条件运算符(?:)

紧缩型 if-then-else 条件句

条件运算符类似于 if 语句,但它的结构更为紧凑。它的语法结构为:

test_expr ? true_expr : false_expr

编译器在计算条件运算符表达式时,首先要计算 *test_expr* 的值。如果 *test_expr* 的值为真(任何非 0 值)的话,表达式将返回 *true_expr* 的值。否则返回 *false_expr* 的值。问号后面的表达式只有在需要的时候才计算,即如果 *test_expr* 的值为真的

话, `false_expr` 表达式的值永远不会被计算。

条件运算符用途比较广泛。首先,由于它代码简练,所以可以在某些地方取代 `if` 条件语句。不过它最大的用处还是构成 `if-then` 逻辑的宏函数,并且该宏函数经常被插入某些表达式里面。例如,我们经常使用条件运算符来定义一个计算最大值和最小值的函数:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

如果你把条件运算符堆叠起来使用,还可以构成 `if-then-else-if` 逻辑结构链。

自减(--)

减去 1

C 和 C++ 里的自减运算符可以使用户非常简便地执行一个普通运算:从某一个变量里减去 1。自减运算符的使用格式有两种,将 `--` 置于变量前面或置于变量的后面:

```
--lvalue  
lvalue--
```

上面的第一种格式是先从变量里减去 1,然后把新的值作为结果返回。它等效于表达式 `(lvalue -= 1)`。第二种格式先把变量 `(lvalue)` 的当前值返回,然后再从变量里减去 1。这两种格式对计算结果的差异有时会严重影响程序的流程,例如,对于如下的代码,当 `n` 自减至 0 时,循环立即终止:

```
long fact(int i) {  
    long amt = i;  
    int n = i;  
    while (--n)  
        amt = amt * n;  
    return amt;  
}
```

很容易证实,如果在上面的程序里使用后置的自减运算 `(n--)`,那么函数将总是返回 0。在循环的某些时候 `n` 将成为 1,下一步立刻就被设置为 0,因此在下一句的乘法运算里, `amt` 将与 0 相乘。

还要注意的是,如果把 0 或某一负数传递给该函数的参数,则循环永远不会终止。为避免这一情况的发生,你可以用语句 `--n > 0` 来代替 `--n`。虽然这样会使程序变得长一些,但容错性提高了不少。

如果你想进一步了解左值(lvalue)的信息,请参阅词汇表里“左值”部分。左值是指那些可以出现在赋值号左侧的任何东西,例如,变量就是一种左值。

自增(++)

增加 1

C 和 C++ 里的自增运算符可以使用户非常简便地执行一个普通运算:对某一个变量增加 1。自增运算符的使用格式有两种,将(++)置于变量前面或置于变量的后面:

```
++ lvalue  
lvalue ++
```

上面的第一种格式是先使变量增加 1,然后把新的值作为结果返回。它等效于表达式(lvalue += 1)。第二种格式先把变量(lvalue)的当前值返回,然后再使变量增加 1。

下面的例子是阶乘函数的另外一种实现方式,这里使用了自增运算符。请注意,这里使用的是后置的自增运算,因此 n 的值在参与了同 amt 的乘法运算之后才增大 1 的。

```
long fact(int i) {  
    long amt = 1;  
    int n = 2;  
    while (n <= i)  
        amt = amt * n++;  
    return amt;  
}
```

如果你想进一步了解左值(lvalue)的信息,请参阅词汇表里“左值”部分。左值是指那些可以出现在赋值号左侧的任何东西,例如,变量就是一种左值。

逻辑运算符

C /C++ 提供的逻辑运算符可以执行很多有用的操作,虽然这些操作并不都能在程序里用得上。同位运算不同的是,逻辑运算首先要把操作数转换为逻辑值(真或假):所有的非 0 数都被视为真(1),只有 0 被视为假。另外,双操作数逻辑运算符(与和或)都使用了“快捷”逻辑。例如,在下面的与(AND)表达式里,如果第一个关系运算的结果为假(0)的话,第二个操作数的值将不会被计算:

```
if (a > b && func1() >= c)
//...
```

每一个具有双操作数的逻辑运算符(&& 和||)都是由两个相同字符组成,这样做是为了把它们和位运算符区分开来提供方便。在我们脑子里应该有这样的印象,逻辑运算是针对整个表达式而使用的(因此逻辑运算符比较大),而位运算符是针对某些位而使用的(因此位运算符较小)。

运算符	说明
&&	逻辑与
	逻辑或
!	逻辑非

逻辑运算符还包括逻辑取非(!),它是一个单操作数运算符,作用是改变操作数的真/假值。如果操作数的值不为0,则返回0,如果操作数的值等于0的话,则返回1。

取模运算符(%)

取模运算符是用来计算两个整型数相除所得的余数。如果你把该运算符看作余数运算符可能有助于记忆,这是因为它的名称“取模”是源自数学里的一个术语,在其它场合很少用到。

尽管它的名称有些古怪,取模运算符的操作却非常简单:用被除数除以除数,返回所得的余数。其中,被除数和除数都必须是整型数。

```
dividend % divisor
```

取模运算符主要用在判断一个整型数是否能整除另外一个整型数,如果可以,运算符返回0。例如,当参数为偶数时下面的函数将返回“真”:

```
bool is_even(int n) {
    return (n % 2) == 0;
}
```

指针运算符

在C和C++里还有几个专门处理地址的运算符(如获得地址、使用地址等)。一般来说,地址值是存储在称为指针的变量里。在第三章里,我们已经初步介绍了指针的概念。

在下面的表格里,我们粗略地列出并总结一下这些专门处理标准指针运算的运算符。表中的 `ptr` 表示任何一个合法的地址表达式,如 `array + 5`。对于这种表达式,根据运算符优先级的关系,必须在适当的位置使用括弧。

运算符(及语法结构)	说明
<code>* ptr</code>	取指针 <code>ptr</code> 的内容:它表示获取 <code>ptr</code> 所指对象的内容,也就是说,获得地址 <code>ptr</code> 里的数据。
<code>&lvalue</code>	获得 <code>lvalue</code> 的地址(<code>lvalue</code> 必须可以出现在赋值号的左侧,比如一个变量)。
<code>ptr->member</code>	取指针 <code>ptr</code> 的内容(<code>ptr</code> 必须已经指向了某一对象),然后访问对象的成员 <code>member</code> 。该成员既可以是数据,也可以是成员函数。 <code>ptr->member</code> 的操作等同于 <code>(* ptr).member</code> 。
<code>ptr[n]</code>	以数组方式检索 <code>ptr</code> 。这里的 <code>ptr</code> 既可以是数组名称,也可以是某一数组成员的地址。该操作等效于 <code>*(ptr + n)</code> 。

这里的符号“*”和“&”都是针对与单个操作数的。如果用在两个操作数之间,它们的意义就完全不同了,作为双操作数运算符使用时,它们分别表示乘号(*)和按位与(&)。

需要提醒读者注意的是,在数组检索的语法表达式里,`ptr[n]`的中括弧具有特定的含义。在这里,`ptr`经常直接就是数组的名称,而在编译器里它则被转换成数组里第一个元素的地址。中括弧里的数字 `n` 需要乘以数组元素的字节长度(例如对于浮点数数组,`n` 必须乘以 4),然后再加上数组的首地址,这样一来,程序就可以根据这一地址来调用数组元素的内容了。

指针到成员(Pointer-to-Member)运算符

伴随着我们对 C++ 运算符认识的一步步深入,我们逐渐从一些普遍、常用的概念过渡到一些隐晦、罕见的用法上,这里就将向读者介绍 C++ 提供的两个特殊运算符,它们被称作“指针到成员”运算符。在你使用 C++ 进行编程的整个时期里,很可能永远不会、也永远不需要使用这两个运算符,因此你完全可以跳过这一节,它对你今后使用 C++ 没有丝毫影响。但是作为概念的解释,这里还是提一下这两个运算符。

指针到成员运算符是与一种特殊类型的变量结合使用的(这种变量被称为指针到成员变量),该变量在声明时需要加上类前缀,但它本身并不属于类声明的一部分:

```
type class::* ptr-to-mem;
```

通过这种方式声明的变量根本不是一个真正意义上的指针,而是在某一类里的一个偏移量。一个成员到指针变量可以在不同的时间代表不同的类成员。下面就把这些变量的操作语法小结一下:

运算符(含表达式)	说明
<i>object</i> . * <i>ptr-to-mem</i>	使用存储在 <i>ptr-to-mem</i> 里的偏移量来访问某一对象的成员。
<i>ptr</i> -> * <i>ptr-to-mem</i>	通过访问 <i>ptr</i> 的内容来获得某一对象,然后使用存储在 <i>ptr-to-mem</i> 里的偏移量来访问该对象的成员。

假设你已经声明了 CHorse 类,并且定义了一个名为 horse 的对象:

```
class CHorse {
public:
    int breed;
    int age;
    int race(void);
} horse;
```

经过以上工作之后,你就可以使用以下的语句来声明一个指针到成员变量。这里的声明表示 pData 可以指向 CHorse 的任何一个整型数据成员:

```
int CHorse::* pData;    // Declare pData as ptr-to-mem
                        // for an int data member.
```

下面一条语句建立了 pData 和数据成员 age 之间的关系。该语句得到了 age 的偏移量。

```
pData = &CHorse::age;    // Store offset of age within
                        // the CHorse class.
```

程序到这一步为止,pData 已经存储了从 CHorse 类的起点到数据成员 age 之间的偏移量。如果给定了一个 CHorse 对象(在这里是 horse),编译器就可以使用该偏移量来访问成员 age 了。请注意,pData 在这里的使用格式为“.*”。

```
int i = horse.*pData;    // Get data at offset
                        // pData (= age)
```

通过上面的示例程序,你可能会提出这样的问题,以指针到成员的方式访问一个数据成员同直接访问类成员(如下面的访问方式)到底有多大的差别呢?

```
int i = horse.age;
```

事实上,两者也的确没有多大不同。通过指针到成员变量来访问数据成员唯一的好处在于:此种类型的变量在不同时刻可以作为不同成员的替身。例如,你现在可以用它来代表变量 `age`,过一会儿还可以用它来代表变量 `breed`(另外一个 `int` 类型的变量)。

```
pData = &CHorse::age;
int i = horse.*pData;    // Get horse.age

pData = &CHorse::breed;
int j = horse.*pData;    // Get horse.breed
```

同样,你还可以通过指针来访问某一个对象(使用“`->*`”)。

```
CHorse * ptr = &horse;
int i = ptr->*pData;
```

你还可以使用指针到成员对象来存储某一函数的偏移量,只是在对指针到成员对象进行声明的时候,可能需要使用一些附加的括弧。不过总的来说,其操作过程和声明一个指向数据成员的对象差不多。

```
int (CHorse::* pFunc)();    // Declare pFunc as ptr to
                           // an int member function.

pFun = &CHorse::race;
int k = (horse.*pFunc)();  // Call horse.race()
```

在这一节的开头已经说过,这些运算符的使用非常晦涩,并且读者多半不会使用它们。指针到成员变量可以存储偏移量,这一点对于程序调试还是有用的,但不幸的是,C++ 不提供任何途径来打印指针到成员变量的数值,并且不允许将它转换为整数类型(不过,你还是可以比较两个这种类型的变量是否相等)。指针到成员的运算最有可能出现的地方是在那些具有大量同种类型成员类里,并且程序员想要在不同的时间使用同一变量来指向不同的数据成员。你可以把一个成员到指针类型的变量传递给某一个函数来指明该函数的操作对象究竟是哪一个成员,而不必使用其它方法(如枚举类型常量)完成同样的函数调用。

关系运算符

关系运算符比较两个表达式并把比较结果——真(`true`)或假(`false`)返回给主调

函数。其结果为 `bool` 类型(你可以使用 `typeid` 来进行一下测试,不过别忘了布尔类型里的真和假分别等同于 1 和 0)。

关系运算符比较的两个表达式既可以都是数值表达式,也可以都是地址表达式。你也可以直接比较任何两个数字,不论它们是整型也好,浮点型也好。在必要的时候,C++ 会对两个操作数中字长较短的那个进行类型转换,使两者类型匹配。

你也可以比较两个地址表达式。在我们编程当中,有时有必要检验两个地址是否相同。同样,对于地址表达式而言,大于和小于的比较也都有一定的用途。例如, `arr[0]` 的地址总是小于 `arr[1]` 的地址,意思是在程序的内存里变量 `arr[0]` 出现在 `arr[1]` 的前面。

```
int *p1, *p2, arr[10];

p1 = arr;          // Assign addr of arr[0].
p2 = arr + 1;      // Assign addr of arr[1].
if (p1 < p2)
    cout << "First address less than second.";
```

然而,把一个地址同一个数值进行比较是毫无意义的,因此在 C++ 里不允许在关系运算里出现混合比较(除非对其中的某种数据进行了合适的类型转换)。但这一规则有唯一的一个例外:即任何地址都可以同 0 进行比较,其作用主要是检验地址值是不是一个空值。

关系运算符包括检验是否相等的符号(==)、检验是否不等的符号(!=)、小于号(<)、大于号(>)、小于等于号(<=)和大于等于号(>=)。请注意检验是否相等的符号(==)和赋值号(=)之间书写上的区别。在 C/C++ 程序里,检验是否相等(==)和赋值(=)操作有着截然不同的结果。

关系运算符的优先级一般都比较低,但它还是比逻辑运算符和所有的赋值运算符的优先级要高。

作用域标识符(::)

作用域标识符具有两种语法格式:

```
class::symbol    // Reference to class member
::symbol         // Reference to global variable
```

第一种格式实际上是某种通用格式的特例:

namespace::symbol

毫无疑问,作用域标识符具有非常高的优先级,这是因为对名称确切含义的说明必须要在其它工作完成之前进行。

在 C++ 里允许在某一个类的内部声明一个符号,而在其它位置定义该符号的操作,尤其是对于成员函数,这种实现方式用得非常普遍,比如在下面出现的 `go_faster` 函数就是这样的例子。该函数的全称应该是 `CAuto::go_faster`。如果成员函数的定义出现在类的声明的外面(如此例),就必须使用作用域前缀 `class::`。

```
class CAuto {
private:
    double speed;
public:
    double go_faster(double inc);
};

CAuto::go_faster(double inc) {
    speed += inc;
    return speed;
}
```

如果你所引用的符号是一个全局符号而不是某个类的成员,那么就要使用作用域标识符的第二种格式是 `::symbol`(缺省情况下,如果类里声明的函数与外部函数重名,并且在程序里没有使用全局标识符,那么实际指代的对象将是类里的成员而不是全局变量)。例如,你可以改写一下 `go_faster` 函数,使它同时还可以设置一个名称也是 `speed` 的全局变量)。

```
double speed;    // Global var, declared outside class.

CAuto::go_faster(double inc) {
    speed += inc;
    ::speed = speed;    // Assign to global var.
    return speed;
}
```

第十二章

类型转换操作符(cast)

一个数据的类型转换能够固定一个表达式的类型:它首先得到一个值,然后改变它的类型,并传递类型改变后的结果。在某些情况下,这种数据转换有可能改变数据实际的比特位模式。如果你是第一次接触类型转换这个概念,你也许感觉它非常怪诞。类型转换实际上是一组相关的能力,它包括下面的四个操作符:

类型转换操作符	主要功能
<code>const_cast</code>	去除一个指针的 <code>const</code> 属性;这使你能够将一个 <code>const</code> 型指针传递到一个非 <code>const</code> 型的参数(查看第十三章以获得 <code>const</code> 型指针的更多信息。)
<code>dynamic_cast</code>	在程序运行时,核实一个对象所指向的是不是一个明确的类型。
<code>reinterpret_cast</code>	在不相关的两个指针类型之间进行类型转换,它使你能将 <code>void *</code> 类型的指针转换为某一特定的类型。
<code>static_cast</code>	在相关的指针类型或对象之间进行类型转换,当将一个长度较大的类型转换为一个较小的类型时,可以使用这个类型转换符来制止编译器发出警告。

老版本 C 语言中的类型转换除了不支持 `dynamic_cast` 操作外,支持其它三种类型转换操作。`dynamic_cast` 操作符是一个新的功能。为达到上面描述的四个类型转换的目的,ANSI C++ 提供了这四种不同的类型转换操作符。同时为了保持后向兼容性,C++ 也支持在 C 语言老版本中的类型转换操作符,但现在不鼓励使用这些旧版本的类型转换符。

(type)**老版本的数据类型转换**

C 语言和 C++ 的早期版本为几乎所有需要进行类型转换提供了一个类型转换操作符。为了保持后向兼容性,ANSI C++ 仍然支持这些类型转换操作符。

语法

下面的表达式与 `expr` 有相同的值,但它被转换为指定的类型:

```
(type)expr
```

尽管理论上 `(type)expr` 与 `expr` 的值是相同的,但对它持行的一个数据转换有可能改变数据实际的比特位模式。例如,对于二进制所表示的数字,整型数 10 和浮点数 10.0 的组成方式是不同的。

使用下面的语法也可以指定老版本的类型转换格式:

```
type(expr)
```

例子

下面是一个简单的例子,它的代码以浮点数格式打印整型数 `i` 的当前值:

```
int i = 5;  
cout << (double)i;
```

在这种情况下,输出结果和下面语句的结果是相同的:

```
cout << 5.0;
```

在许多程序中的类型转换,并不是象上面这个简单的例子中的类型转换那样不会对程序有任何影响。当程序中有赋值和函数调用发生时,编译器会根据需要自动的将整型数格式升级为浮点数格式。例如在下面的代码中, `(double)` 强制类型转换就不是必须的:

```
int i = 5;  
double x = (double)i;
```

除了 `dynamic_cast` 操作符之外,老版本 C 中的类型转换可以用在 ANSI 类型转换操作符允许使用的所有情形。要了解这些特殊用法的更多信息,请阅读本章其他部分。例如,下面是对一个 `malloc` 返回值的两种类型转换的比较。这两条语句的功能是相同的:

```
char *p = (char *)malloc(n);  
char *p = reinterpret_cast<char *>(malloc(n));
```

同样,下面制止编译器发出警告的两条语句也完成同样的操作:

```
bool flag1 = (bool)12;  
bool flag1 = static_cast<bool>(12);
```

const_cast

去除 const 属性

`const_cast` 操作符用来帮助调用那些应该使用而没有使用 `const` 关键字的函数。这样的函数有一个指针变量作为参数,并且函数从不改变指针所指向数据的值,同时程序员忽视了应该将这个函数的参数声明为 `const` 类型。最好能够向这样的函数传递一个 `const` 指针,但是 C++ 的规则禁止这样做。针对上述情况,`const_cast` 操作符提供了一个间接的解决方法。(请查看第十三章以获得 `const` 关键字和 `const` 指针的更多信息。)

语法

下面的表达式的值与 `expr` 的值相同。除了 `const` 和 `volatile` 属性可以去除外,指定的类型必须与 `expr` 的类型相同。典型的情况,这里的 `type` 指的就是一个指针类型。

```
const_cast<type>(expr)
```

例子

在下面的例子中,函数 `display_num` 有一个指针参数(`p`)但这个函数没有改变它指向的数据(`*p`)的值。

```
void display_num(double *p){  
    printf("The value is %2.3f\n", *p);  
}
```

因为 `p` 指向的数据没有改变,所以你在调用该函数时应该可以传递一个 `const` 型的指针给该函数,但实际 C++ 的规则禁止这样做,因为一个 `const` 指针通常不能传递给一个非 `const` 类型的参数。

```
const double x;  
display_num(&x);    //DISALLOWED; &X is const
```


绕过这种限制方式来转换指针类型就能够去除 `const` 属性。下面的代码将 `&x` 由 `const double *` 类型转换为 `double *` 类型,从而加强了清道地址的能力:

```
const double x = 17.5;
display_num(const_cast<double * >(&x));
```

当使用 `const_cast` 操作符时,必须保证不改变指针所指向的数据。如果你使用 `const_cast` 操作符但又设法改变指针所指数据,那么将使实际结果无法预料。如果这样做,编译器会禁止赋值生效,或者由于操作平台将 `const` 变量放入只读存储器,同样使赋值无效,由于这种操作使一次函数调用都带来了不可预见的后果。

```
const double x = 17.5;
double *p;
p = const_cast<double * >(&x);
*p = 33.0; //THIS OPERATION IS UNDEFINED
```

● ANSI

`const_cast` 操作符是一个扩展的 ANSI 特征,在早期的 C++ 版本中不支持它。

dynamic_cast

程序在运行时对 类型进行的检测

`dynamic_cast` 操作符让你检测一个基类的指针指向的是不是一个特定的子类型的对象。事实上 `dynamic_cast` 使用运行时类型信息 (run-time type information RTTI) 在程序运行时检测一个对象的类型。

语法

下面的表达式试图为指定的类型返回一个指针。在程序运行时,由 `expr` 指向的对象必须是这个指定的类型,或者是由 `expr` 派生出来的类,否则这个类型转换就会失败并返回一个空指针。

```
dynamic_cast<type * >(expr)
```

你可以把 `dynamic_cast` 操作符和引用类型结合起来使用。如果类型转换因为 `expr` 不具有指定的类型或者是不属于由这个指定类型派生出来的类而失败,那么将产生一个 `bad_cast` 异常。

对 `dynamic_cast` 操作符的一个限制是如果你使用它来转换一个派生类的指针——这是它主要的应用,那么 `expr` 类就必须至少有一个虚函数。因为这个原因,dy-

`dynamic_cast` 操作符被称为多态类型转换 (polymorphic cast)。你还可以无限制地使用 `dynamic_cast` 操作符转换一个基类的指针。如果类型之间是根本不相关的,那么编译器就不会允许这个类型转换。

例子

使 `dynamic_cast` 操作符有用的原因是一个基类的指针能够指向许多不同的子类型 (派生类)。在这些派生类中,一些支持其它子类里不支持的函数。

例如,下面的代码声明一个基类 B 和它的一个派生类 D,然后为这个基类的指针分配一个对象。

```
class B{
public:
    virtual void func1(int);
};

class D : public B{
public:
    void func2(void);
};

//...
D od;
B * pb;
pb = &od;
```

最后一条语句将类 D 的一个对象的地址分配给 pb,类 D 是一个子类型。这样做是完全合法的。表面上看 pb 指向的类型是 B*,但实际上它指向的类型是 D。

下面的函数使用 `dynamic_cast` 操作符检查类型为 *B 的参数是否真的指向类 D 的一个对象。如果真是这样的话,函数就可以利用这个事实来调用 func2,尽管 func2 仅定义在类 D 和它的派生类中。

```
void process_B(B* arg){
    d * pd;
    pd = dynamic_cast<D*>(arg);
    if(pd)
        pd->func2();
    //...
}
```

如果 `arg` 指向类 `D` 或类 `D` 的派生类的对象,那么这个类型转换是成功的,而且通过指针来调用 `func2` 也是安全的。否则的话,强制类型转换就是失败的,`pd` 将被赋值为 `NULL`。

```
pd = dynamic_cast<D*>(arg);
```

● ANSI

`dynamic_cast` 操作符是一个扩展的 ANSI 特征,在早期的 C++ 版本中并不支持它。要注意一些编译器只有在打开编译器标志时才提供这个特征(例如在 Microsoft Visual C++ 6.0 中的标志是 `/GR`)。

reinterpret_cast

转换指针的类型

`reinterpret_cast` 操作符有一个主要的目的:将一个指针转换成其它类型的指针。新类型的指针在任何一方面都不必与旧类型的指针相关。这个操作符也可以用在指针和整型数之间的类型转换上。

语法

下面的表达式返回与 `expr` 的值相同的值,但这个值被转换成指定的类型。这个指定的类型通常是一个指针类型,它只在使用方式上与 `expr` 的类型不同。例如,一个 `float *` 类型的指针的用法是不同于一个 `char *` 类型的指针的。这类强制类型转换不能直接的改变数据的值。

```
reinterpret_cast<type>(expr)
```

这个类型转换不能用来去除来自于 `expr` 的一个 `const` 或 `volatile` 的属性。它可以没有限制地在指针类型之间、指针和整型数之间进行类型转换。

通常,`reinterpret_cast` 操作符从不改变它的指针表达式 `expr` 的内在的比特位模式,也就是地址本身对应的数字值没有被改变。然而,当一个指针作为一个不同的类型(例如,做为 `int *` 类型而不是 `float *` 类型)被引用时,会得到一个戏剧性的结果。

例子

目前这个操作符最普通的用法是,将一个 `void *` 类型的返回值或参数转换为一个更明确的指针类型。在 C++ 中,没有用类型转换而将一个 `void *` 类型的指针赋给另外类型的指针是不合法的。例如:

```
char *p = reinterpret_cast<char*>(malloc(100));
```

尽管 `malloc` 的返回值不需要立即把返回的指针转换为另外的指针类型,但在引

用它之前必须对它进行类型转换。void * 类型的指针只能用来传递未加工的指针数据。new 操作符相对于 malloc 的一个优势是它不需要强制类型转换。

reinterpret_cast 操作符在定义函数代码时也是有用的,此时函数得到一个 void * 类型的指针做为参数。查看第十五章的 qsort 以获得它的例子。

reinterpret_cast 操作符存在潜在的危險,除非有使用它的充分理由,否则就不要使用它。例如,它能够将一个 int * 类型的指针转换为 float * 类型的指针,但是这样做很容易造成整数数据不能被正确地读取。这样做也就容易造成程序打印出来无用的东西,如下面的程序所示:

```
int i = 100;
int * pi = &i;
float * pf = reinterpret_cast<float*>(pi);
cout<< * pf;    //THIS OUTPUTS GARBAGE
```

转换指针与转换对象(使用 static_cast 操作符来做)的结果是完全不同的。当一个指针被重新转换然后被引用时,你好象是在说“以浮点数的格式来读取一个整数”。除非有这样做的充分理由,否则应该尽量避免使用它。相反,可以使用 static_cast<float>来得到一个值并用正确的浮点数格式表示它,如下所示:

```
int i = 100;
float f = static_cast<float>(i);
cout<<f;
```

● ANSI

reinterpret_cast 操作符是一个扩展的 ANSI 特征,在早期的 C++ 版本中是不支持它的。

static_cast

转换成为相关的对象或指针

static_cast 操作符能在相关的对象和指针类型之间进行类型转换。有关的类之间必须通过继承或者构造函数或者转换函数发生联系。static_cast 操作符还能够在数字的(原始的)类型之间进行类型转换。

语法

下面的表达式的值和 expr 的值是相同的,但这个值被转换成指定的类型:

```
static_cast< type >(expr)
```

尽管上述表达式的值和 `expr` 的值在理论上是相同的,但它要涉及到一个数据类型转换,它改变数据的实际比特位模式。例如,整型数 10 和浮点数 10.0 用二进制数来表示是不同的。

`static_cast` 操作符可以用在下面的情形中:

- 如果两个不同的指针指向的类通过继承相联系,那么就可以在这两个指针之间进行类型转换。与 `dynamic_cast` 操作符不同的是,你可以在没有虚函数存在的情况下转换一个派生类指针的类型,而且编译器不会执行运行时状态检查。
- 可以在原始类型间进行强制类型转换。
- 可以将类型为 A 的表达式转换为 B 类型,只要 B 提供适当的构造函数或者 A 提供给 B 一个转换函数。

通常,不同类型的数据在同一表达式中进行操作时,其中一些会自动改变自身数据类型,在与这种自动改变相反的方向上,都可以使用 `static_cast` 操作符进行类型转换。例如,整型数在象 `f = i` 这样的表达式中会自动的变为浮点格式的数字。在与此相反的表达式情况下(`i = f`)就应该使用 `static_cast` 操作符进行类型转换。

例子

通常,`static_cast` 操作符大多用在将数域宽度较大的类型转换为较小的类型的情况。当转换的对象是原始数据类型时,这种操作可以有效地禁止编译器发出警告。

```
long j = 17;  
short i = static_cast< short >(j);
```

当使用这样一个类型转换时,你好象是在说“是的,我的确想这样做”,编译器为这个类型转换做注释并避免警告的发生。(通常它发出一个潜在的数据丢失的警告)你的责任是要保证原类型数据不能太大以致于不能存储到新的类型中去。在上面的例子中,要保证数据可以存储到 `short` 型的变量中。

同样,可以没有任何限制地将一个基类的指针转换为一个派生类的指针。因为没有执行运行时状态检查,你要保证实际的数据支持这个类型转换。例如,考虑下面的代码,其中类 B 是类 D 的一个基类:

```
B * pb;  
//...
```

```
d * pd;  
pd = static_cast<D*>(pb);
```

除非 `pd` 指向的对象确实是属于类 `D` 或者是 `D` 的派生类, 否则这个类型转换可能会造成错误。确保 `pd` 指向的对象属于类 `D` 或它的派生类是你的责任。(查看 `dynamic_cast` 操作符以获得更多的有关它转换一个派生类指针的细节。)当然, 你也可以按相反的方向来做——转换一个基类指针, 但此时这种类型转换是不需要的。

```
pb = static_cast(B*>(pd);
```

还有一些其它的情况 `static_cast` 操作符能够被有效地使用, 但是这在大多数程序中是少见的。例如, 你可以使用 `static_cast` 对一串数据进行类型转换。在下面的代码中, 假定 `A` 和 `B` 都是类, 其中在类 `A` 中定义了一个到类 `B` 的类型转换, 在类 `B` 中定义了一个到 `int` 的类型转换。按照如下的操作类 `A` 的一个对象能够被转换为 `int` 型。

```
A oa;  
int i = static_cast<int>(static_cast<B>(oa));
```

如果此处没有使用类型转换, 编译器就不知道如何将 `oa` 转换成整型的(假定一个从 `A` 到 `int` 的转换是不存在的)。顺便说一下, 上面的类型转换也可以象下面那样在老版本的 C 中实现:

```
int i = (int) (B) oa;
```

对于程序员来说, 使用 `static_cast` 操作符进行类型转换要比老版本中的类型转换做更多的工作。但是, ANSI C++ 中的类型转换提供的一个重大的便利是: 它阐明了无次类型转换的用途而且在大量的程序代码中容易找到它们。

`static_cast` 操作符在接收多个数据格式的情况下也是有用的。如果你想指定的格式不同于表达式实际假定的格式, 就可以使用类型转换。例如, 因为 `cout<<` 为了接收多种类型的数据而进行了重载, 所以 `static_cast` 操作符在这里就发挥了作用。下面的例子按浮点数格式而不是整型格式打印 `i`。

```
int i = 25;  
cout<<static_cast<double>(i);
```

要记住当把一个字符串以地址方式而不是一组字符的方式来打印输出时, 应该使用 `reinterpret_cast` 操作符。这是因为字符串名是一个指针类型而不是一个原始类型。

```
char str[] = "Hello";
```

```
cout<<hex<<reinterpret_cast<int>(str);
```

● **ANSI**

`static_cast` 操作符是一个扩展的 ANSI 特征,在早期的 C++ 版本中并不支持它。

第十三章

C++ 的关键字

计算机语言中的关键字是一些在这种语言内部具有普遍意义的预先定义好的名字。在 C 和 C++ 中,关键字不包括库名和类名,也不包括来自于标准库中的那些名字(这将在第十五章和第十六章进行讨论)。

在这一章中将介绍除了数据类型和操作符关键字以外的所有 ANSI C++ 中的关键字。(数据类型的关键字已在第十章讨论,操作符关键字已在第十一和第十二章讨论。)

asm

解释汇编代码

尽管 ANSI 规范支持 `asm` 关键字来解释汇编代码,但是它的使用是依具体软件而定的。主要的编译器供应商提供他们自己对这个关键字的扩展。汇编代码是一种机器语言而且不容易移植。当使用汇编代码时,应尽可能把它限制在尽可能少的模块中。查看编译器文件以获得更多的信息。

auto

分配自动变量

C++ 中有一些令恼人的关键字,其中最讨厌的是 `auto` 关键字,因为它为没用的目的服务。`auto` 关键字用于局部变量的定义,它指定自动存储类。(这样就指定了函数的每个调用都要建立自己的对变量的拷贝,通常是放在程序的堆栈段。)因为局部变量本身就是存储类,所以 `auto` 关键字基本上这些拷贝什么都没有做。

```
auto data-declaration;
```


break**退出当前的循环或一个
switch 语句**

`break` 关键字用来退出它所在的最近的一层循环或者是一个 `switch` 语句。`break` 关键字后面跟随一个分号就能构成一条完整的语句；

```
break;
```

这条语句仅在循环(例如, `do`、`for` 或 `while`)或者是 `switch` 语句中才是合法的。当执行 `break` 语句后, 程序就跳到 `switch` 语句或循环语句外的下一条语句处执行。在 `switch` 语句内必须使用 `break` 关键字。例如:

```
case 1:
    strcpy(strNum, "One");
    break;
case 2:
    strcpy(strNum, "Two");
    break;
//...
```

查看有关 `switch` 语句的部分来获得更多的信息。

case**为 switch 语句定义目标**

`case` 关键字用来在 `switch` 语句内部标志一条语句。被标志的语句成为 `switch` 的一个可能跳转的目标。

```
case constant_value:
    statement
```

在 `switch` 语句内部, 如果 `case` 后面的常量表达式与 `switch` 括弧内的表达式匹配, 就执行该 `case` 后面的语句。(查看 `switch` 关键字来获得更多的信息。)

可以使用任何数字值来作为 `case` 语句的标签。如果这个数字值与 `switch` 括弧中的检测值是匹配的, 那么就执行它后面的语句。

catch**定义异常的句柄**

`catch` 关键字定义一个异常处理的句柄。这个关键字必须作为 `try-catch` 控制结

构的一部分出现。try-catch 控制结构可以有一系列的 catch 语句块,每个 catch 语句块捕获不同类型的异常。查看 try 关键字以获得更多的信息。

```
catch(exception_type [object]){  
    statement  
}
```

中括号中的 object 指的是被传递的异常对象;这个对象会在 statement 部分中用到。在上面的语法中,中括号表示 object 是可选择的。

还可以使用这个关键字编写一个默认的异常处理,这个默认的异常处理可以捕获任何类型的异常。在下面的语法中,省略号(...)应原封不动地使用:

```
catch(...){  
    statements
```

class

声明用户定义的类型

本书已不止一次地提到类这个词。此处我们要总结 class 关键字的一些通用的语法。查看第五章有关这方面的介绍。

class、struct 和 union 都能声明一个类。它们的区别是当使用 class 来声明一个类时,它的成员默认是私有模式的。class 关键字有下面的语法格式,其中中括号内的部分是可选择的。

```
class name[:base_class_declarations]{  
    declarations  
}[object_definitions];
```

declarations 部分可以包括成员变量和成员函数。就象第五章描述的那样,可以在类定义的内部声明成员函数,而在类外提供它的定义。

例如,下面的代码声明了 CStr 类的一个派生类 CIOStr。它同时声明了三个对象 str1, str2, str3。因为类的成员默认是私有模式的,所以虽然没有明确地声明 status 的访问模式,但它仍是私有的。

```
class CIOStr : public CStr{  
    int status;  
public:  
    void input(void);
```

```
void output(void);  
}str1, str2, str3;
```

不要忘记了用分号作为终止符。不管是否声明对象,类定义都要用分号作为结束。

const

阻止对变量的改变

`const` 关键字阻止对变量或参数的改变。`const` 和指针参数一起使用(这部分的用法 3)是它最重要的用法:用户可以得到指向数据的一个指针同时同意不会改变所指向的数据。这象是个约定。

当把 `const` 关键字用在变量或参数声明的前面时,它就会修改这个变量或参数的类型。这些变量或参数可以被初始化,但此后是不能被改变的。

```
const type item
```

还可以将一个指针声明为 `const` 类型。

```
const type * pointer
```

任何试图使用这样的指针向它指向的条目分配一个值的行为都将被标记为错误操作。

也可以在一个指针声明的内部使用 `const` 关键字:

```
type * const pointer
```

甚至可以声明一个 `const` 类型的 `const` 指针,它表明不论是指针还是指针指向的对象都是不可改变的:

```
const type * const pointer
```

用法 1: `const` 变量

可以将 `const` 应用于一个简单的变量。一旦变量被使用 `const` 定义后,编译器将拒绝所有的为这个变量赋值的语句。给这个变量赋值的唯一方法是通过初始化来完成。

```
const int id = 12345;    //This is valid; id may be  
                        //initialized  
...  
id = 10000;             //ERROR! attempt to assign new value.
```

也可以将象数组和类这样的复合类型声明为 `const` 的。这时,对象中的所有成员(元素)都将被保护而不被改变。

用法 2: `const` 指针

`const` 指针的规则是 `const` 变量在逻辑上的一个扩展。下面的声明把指针 `p` 指向一个 `const` 的整型数。也就是说,`p` 指向一个不能被改变的整型数。

```
const int * p;
```

尽管 `p` 本身可以改变,但 `*p` 是不能被改变的。C++ 中关于指针和 `const` 型数据相互作用的规则是:

- 一个 `const` 型的指针既可以指向 `const` 型的数据也可以指向非 `const` 型的数据。
- 一个普通的指针不能指向 `const` 型的数据。

第二个规则对于阻止象下面那样隐秘的骗局是必须的:

```
const int id = 12345;  
int * p = &id;           //ERROR! This is not allowed  
(*p)++;
```

你可以试着将这个规则和一个 `const` 指针一起使用,但它还是不能工作,这是因为如果 `p` 是一个指向 `const` 型的指针,那么 `*p` 是不能被改变的。

```
const int id = 12345;  
const int * p = &id;      //This is okay.  
(*p)++;                 //ERROR! *p is const!
```

用法 3: `const` 参数类型

通常由引用来传递数据时,要防止对数据的改变。可以声明一个 `const` 型的指针参数来完成这项工作。函数得到指向数据的一个指针,但它不允许使用这个指针来改变数据的值。例如,函数应该按下面所示那样进行声明:

```
void fnct(const char str[], const double * px,  
         const double * py);
```

在函数定义的内部, `*str`、`*px` 和 `*py` 的值是可以读的。但下面试图向它们所指向的数据分配值的语句将会作为错误被标记出来:

```
*str = 'a';  
str[2] = 'z';  
*px = 0.0;  
*py = 98.6;
```

使用一个 `const` 型的指针改变所指向的数据是一个错误,即使是此时涉及到的是该指针所指地址的偏移量(上面的第二个表达式)。

用法 4: `const` 型的成员函数和对象

如果你使用 `const` 关键字声明了一个对象,那你就只可以调用这个对象的 `const` 型的成员函数。这样的函数不会改变成员变量。同时,除了使用构造函数外,你不能改变一个 `const` 型对象的任何成员变量。

通过将 `const` 关键字放在函数声明之后和函数定义之前,你可以将一个成员函数声明为 `const` 型的。例如:

```
class CStr{  
    int getlength(void) const {return nLength;}  
    char * get(void) const {return pData;}  
    //...  
};
```

一个 `const` 型的成员函数不能改变任何成员变量的值。然而你可以重载这一个成员函数,这样它就有两个版本:一个是 `const` 型的,一个是非 `const` 型的(除了 `const` 关键字外,两个函数没有区别)。

continue

跳转到下一次循环

`continue` 语句可以直接跳转到它所在的最近的 `do`、`for` 或 `while` 循环的下一次循环。它不是完全退出整个循环(`break` 语句退出这个循环),而是在继续执行循环之前简单地跳到循环的底部。`continue` 关键字的语法很简单:

```
continue;
```

例如,下面的循环打印所有从 1 到 `n` 之间的素数。如果一个数不是素数,循环就立即使 `i` 加 1,跳转到下一次循环并判断下一个值。否则,它打印这个素数并在下一次循环之前将 `num_of_primes` 加 1。此处假定的 `is_a_prime` 函数在程序的其它地方已经被定义。

```
for(i = 1; i <= n; i++){  
    if(! is_a_prime(i))  
        continue;
```

```
printf("%d\n", i);  
num_of_primes++;  
}
```

此处需要注意的是 `continue` 使循环计数 `i` 增加, 而不是使 `num_of_primes` 增加。

default

为 switch 语句定义 “else”目标

`default` 关键字用来标记 `switch` 语句内的一条语句。若所有 `case` 中表达式的值都不与检测值相匹配, 就执行 `default` 后面的语句。查看 `switch` 关键字获得完整的语法和其它的信息。

`default: statement`

do

重复执行循环

`do` 语句与 `while` 语句做差不多一样的事; 只要一个指定的条件满足, 它就执行一个循环。唯一的区别是 `do` 循环保证语句至少被执行一次, `do` 的语法格式如下所示:

```
do  
    语句  
while(表达式);
```

只要表达式内的值是非零(真), 那么循环就重复执行。循环内部的语句通常由复合语句组成。例如, 下面的代码打印一个文件中的字符直到到达文件尾为止。此时至少有一个字符被读取。

```
#include<stdio.h>  
//...  
do{  
    c = getc(fp);  
    putchar(c);  
}while(c != EOF);
```

else

可供选择的执行语句标志

`else` 关键字是 `if` 语句中的一个可选择的部分。如果在 `if` 语句中有 `else` 子句, 那

么当 if 语句的条件为假(等于 0)时,就执行 else 后面的语句。尽管 C++ 中没有包含“elseif”关键字,但可以重复使用 if 和 else 关键字来检测一系列二选一的条件。查看 if 关键字来获得更多的信息。

enum

定义连续常量的一个列表

对于列表来说枚举是一个奇特的词。在 C++ 中,枚举是一系列的条目,其中每个条目都分配给一个识别数字:0,1,2,3 等等。

在编写程序时,有时需要你定义一系列常量。它们的实际值可能很重要也可能无关紧要,但必须保证它们是各不相同的。enum 关键字是创建这些常量的一种有效方式;它的语法格式如下所示(括号内是可选择的部分):

```
enum[ enum_type_name ] {  
    item_1, item_2, ..., item_n  
}[ variable_declarations ];
```

结果是将每个条目定义为一个符号常量。第一个条目的默认值是 0。其它条目的默认值是前面一个条目的数值加 1。每个条目都可按用户意图赋给一个明确的整数值:

```
item = integer_value
```

例如,下面的声明生成四个常量 CLUBS, DIAMONDS, HEARTS, SPADES, 并分别将 0,1,2,3 分配给它们作为它们的值。这与使用一系列 #define 命令的效果是相似的。

```
enum {  
    CLUBS,  
    DIAMONDS,  
    HEARTS,  
    SPADES  
};
```

为了节省空间可以将它们放在一行中:

```
enum { CLUBS, DIAMONDS, HEARTS, SPADES };
```

* 通常,从 0 开始的一系列值就能满足程序设计的要求。但是有时你可能想以一个确定的值开始。下面的代码为某些数词声明枚举常量,你可以赋给它们你希望的值:

```
enum{
    twelve = 12,
    thirteen,
    fourteen,
    fifteen,
    twenty = 20,
    twenty_one,
    twenty_two,
    hundred = 100
};
```

因为每个条目的默认值是它前面条目的值加 1, 所以这样声明枚举常量是可行的。例如 `thirteen` 没有明确地赋给一个值, 因此它的值就是 `twelve` 的值加 1。 `twenty` 被明确地赋给一个值, 否则的话它的值就是 `16(fifteen + 1)`。

可以为枚举赋予一个类型名并用这个类型名来初始化变量。例如:

```
enum card_suit{
    CLUBS,
    DIAMONDS,
    HEARTS,
    SPADES
} card1;

card_suit card2, pack[52];
```

此处 `card1` 和 `card2` 是 `card_suit` 类型的变量, `pack` 是一个有 52 个元素的数组, 其中每个元素都是 `card_suit` 类型的。 `card_suit` 类型的成员的值只能是 `CLUBS`, `DIAMONDS`, `HEARTS` 和 `SPADES` 四者之一。例如:

```
card1 = CLUBS;
pack[10] = card2 = card1;
```

● C / C++

在 C++ 中, 枚举直接创建一个类型名; 在 C 中, 它只创建了一个附属名, 这个附属名必须和 `enum` 关键字一起使用。因此, 在 C 中, `card2` 和 `pack` 必须按如下格式进行声明:

```
enum card_suit card2, pack[52];
```


explicit**禁止自动转换**

构造函数的好处之一是它们自动支持类型转换,就象第六章描述的那样。例如,一个构造函数 `C(type t)` 自动提供一个从指定类型到类 `C` 的一个实例的类型引用转换(例如 `cobj = t`)。

`explicit` 关键字禁止这种自动转换。因为 `explicit` 剥夺了构造函数的一种好的副作用,所以它很少使用。在使用这个关键字时,应将它放在类定义内构造函数声明的前面。如下所示:

```
class CStr|
//...
    explicit CStr(char *s);
```

● **ANSI**

`explicit` 关键字是 ANSI C++ 中一个非常新的特征,其它版本的 C++ 都不支持它。

extern**寻找外部定义**

`extern` 关键字告诉编译器,“这个变量可能定义在这个模块或其它模块中”。一个 `extern` 声明并没有生成数据,它仅表明这个数据是共享的。这个变量必须是在别处定义过的,而且它只能被定义一次(为加深理解请看例子)。

这个关键字的语法格式如下所示,其中括号表示“C”是可选择的:

```
extern [“C”] declaration;
```

如果语句中包括“C”,表明这个条目是根据 C 语言的约定来声明的。这个特征使得 C 语言的模块,不必象 C++ 代码需要进行改写和编译,而可以直接访问这些条目。

下面的简单例子中在 `module1.cpp` 中定义了变量 `amount`,并使它在所有的模块中都是可使用的。在 `module1.cpp` 中 `extern` 的声明不是真正需要的,但是这样做也不会引起错误。

```
//MODULE.CPP extern int amount; int amount = 0;
//Variable defined here.
```

```
//MODULE2. CPP  
extern int amount;  
//MODULE2. CPP  
extern int amount
```

此处可供选择的一个方法是将 `extern` 声明放在一个工程的头文件中。包含这个头文件的所有模块都会得到这个 `extern` 声明。

for

按计数值执行循环

C /C++ 中的 `for` 语句能够执行一个固定次数的循环,就象 BASIC 中的 `for` 语句一样。但是在 C /C++ 中 `for` 语句的使用是非常灵活的。

```
for( initializer ; condition ; increment )  
    loop_statement
```

上面的语句和下面使用 `while` 的语句的功能相同(作者为下面的例子添加了注解)。`initializer` 语句只被执行一次,每完成一次循环 `increment` 语句都要被执行一次。

```
initializer ;           //Set i = 1,  
while(condition) {      //While i  <= 100, do more,  
    loop_statement  
    increment ;         //i = i + 1  
}
```

两者的唯一不同是:对于 `for` 循环来说, `continue` 语句的作用是在执行下一次循环前先执行 `increment` 语句。

使用 `for` 循环的标准方式是给它设一个循环变量,例如下面例子中的 `i`。下面的例子打印从 1~100 的所有整数。在你自己的代码中, `initializer`, `condition` 和 `increment` 可以是任何合法的表达式。

```
#include<iostream, h>  
//...  
  
for(i = 1; i <= 100; i++) {  
    cout<<i;  
    cout<<" ";  
}
```

逗号在 `for` 循环中特别有用,因为它使你能够将多个操作放在一个表达式中。

例如:

```
for(i = 0,j = 0;i < MAX_SIZE;i++,j++)  
    //...
```

friend

允许对类的外部访问

一个类的友元可以访问这个类的所有成员。一个类对于它的友元来说是没有秘密可言的。友元的一个最常用的类型是函数。一个完整的类也可以成为一个友元。

将一个函数声明为一个类的友元函数,需要将这个函数的原型包含在类定义的内部,并在它的前面加上 `friend` 关键字:

```
class name {  
    //...  
    friend function_prototype;
```

为这个函数声明为访问级别(`public`, `private` 或者 `protected`)是无关紧要的。因为此时访问级别被忽略。除此之外就不需要任何别的调整了。在类定义之外是不使用关键字 `friend` 的。

为了将另外的一个类声明为友元,使用下面的语法。(如果友元类是由 `struct` 或 `union` 声明的,就用 `struct` 或 `union` 来代替例子中的 `class`。)

```
class name {  
    //...  
    friend class class_name;
```

在 C++ 中,除了需要的地方外应该避免友元关系,因为它趋向于削弱封装的作用。友元函数有一个主要的目的:帮助编写二进制操作符函数。例如,在下面的代码中,函数 `operator+` 是作为全局函数来编写的。但是它需要访问类 `CFix` 中的一个私有成员,此时就可以在 `CFix` 类中将 `operator+` 定义为一个友元函数,这样函数 `operator+` 就能访问 `CFix` 的私有成员了。查看第七章可获得 `operator+` 函数的定义。

```
class CFix {  
private:  
    long amount;  
    friend CFix operator+ (long lng, CFix cf);  
    //...  
};
```

```
//Operator function for long + CFix.  
//Because this function is a friend, it can access amount,a private member.  
//  
CFix operator+ (long lng,CFix cf){  
    CFix result;  
    result.amount = (lng * 1000) + cf.amount;  
    return result;  
}
```

友元关系有两个规则,只有在特别情形下才须对它们加以注意:

- 如果类 C1 是类 C2 的一个友元类,而类 C2 又是类 C3 的一个友元类,那么 C1 是不会自动的成为 C3 的友元类的。也就是说,我朋友的朋友不必是我的朋友。
- 如果类 C1 是类 C2 的一个友元类,那么 C1 的派生类 D 不会自动成为 C2 的友元类。也就是说,我朋友的孩子不必是我的朋友。

goto

无条件跳转语句

C++ 与其它程序语言一样包括一个 goto 语句。使用 goto 语句的危险是将使程序的流程无规律、可读性差,结构化程序设计中是主张限制 goto 语句的使用的。goto 语句使用下面的语法格式:

```
goto label;
```

这条语句无条件地转到语句标号所指定的语句上,所指定的语句必须与 goto 语句在同一个函数内。语句标号由一个标识符跟随一个冒号组成。

```
goto the_end;  
//...  
the_end:  
    return n;
```

当从一个多重循环中直接跳出时,goto 语句是很有用的。在大多数情况下,如果可以使用 break 和 continue 语句来完成同样的功能,就不必使用 goto 语句了。

if

条件执行

任何程序语言的一个核心部分都包括 if 关键字或者是某种类型的条件转移。

C++ 也不例外。if 语句的语法格式如下：

```
if(表达式)
    statement1
[ else
    statement2 ]
```

此处的中括号表示 else 子句是可选择的。要注意 statement1 和 statement2 通常由复合语句组成,它们要使用大括号{}来括起来。

如果表达式是非零值(为真),就执行 statement1。否则如果假定这里有 else 子句的话,就执行 statement2。表达式可以是任何 C 中合法的整型表达式。所有关系运算符和逻辑运算符的返回值不是 1(真)就是 0(假)。

下面是使用 if 语句的一个简单的例子

```
if(x == y) puts("x equals y!");
```

通过在 else 子句中嵌套 if 语句,你可以创建一个虚拟的“elseif”关键字:

```
if (a < b)
    return -1;
else if (a == b)
    return 0;
else
    return 1;
```

上面的语句和下面的语句是等同的:

```
if (a < b)
    return -1;
else {
    if (a == b)
        return 0;
    else
        return 1;
}
```

有时候需要使用大括号{}将 else 子句与一个 if 语句联系起来。默认情况下,else 子句与离它最近的相匹配的 if 语句相联系。

● 注意

在条件判定表达式中,不要将关系运算符(==)与赋值符(=)混淆。在第二章中已经深入的讨论了这个问题。

inline**将函数代码设置
为内联形式**

inline 关键字按下面的方式修改一个函数的定义:它建议编译器这个函数应该被设置为内联函数,也就是说,在编译时应该将函数相应的代码内联起来而不是产生调用和返回的代码(查看附录中的 inline 部分)。

在使用这个关键字时应该将其放在函数定义的第一行的前面。如下所示:

```
inline double cube_it(double x){  
    return x * x * x;  
}
```

inline 关键字基本上是对编译器的一个建议。一些编译器将一个函数内联起来,这将优化整个程序。查看编译器文件来获得更多的信息。

main**程序入口点**

main 函数是标准控制台应用程序的入口点。特殊的应用程序,象 Windows 应用程序和 DLL 文件是不使用 main 函数的。main 函数的定义可以是图 13-1 所示的两种格式中的任何一种。此时参数 *argv[] 中的中括号应原封不动地使用。

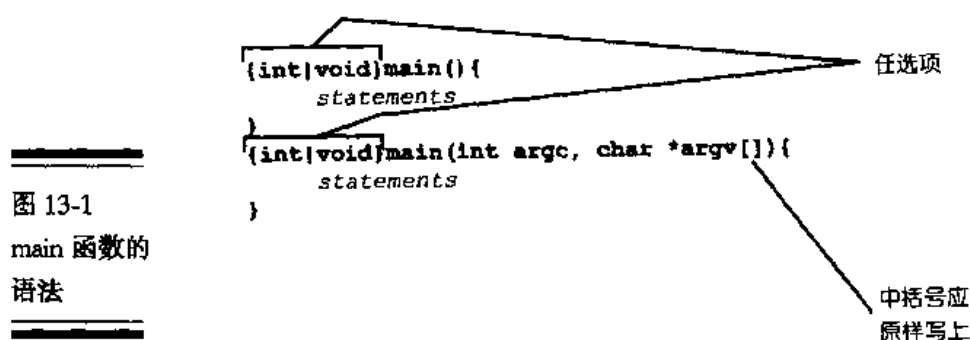


图 13-1
main 函数的
语法

在这里,表达式 {int|void} 表明 main 的返回值必须是 int 型或者是 void 型。

例如,下面的代码定义不返回参数类型的 `main` 函数。

```
void main () {  
    init_vars();  
    prompt_for_input();  
    print_results();  
}
```

你可以使用特殊的参数 `argc` 和 `argv` 来得到命令行参数。例如,假设你编写了一个叫作 `sort.exe` 的应用程序。它接收一个输入文件和一个输出文件。程序的命令行方式应该是:

```
sort datafile.txt results.txt
```

参数 `argc` 包含命令行中的全部参数的数目,也包括程序名本身。(在此时的情况下,`argc` 是 3)。参数 `argv` 是一个字符串数组。此时的 `argv[0]`,`argv[1]` 和 `argv[2]` 分别指向字符串 "sort", "datafile.txt" 和 "results.txt"。

下面的代码在输入参数不为两个时打印一个错误消息:

```
int main(int argc, char * argv[])  
{  
    if(argc != 3){  
        printf("Bad number of arguments. \n");  
        printf("Syntax: sort infile outfile. \n");  
        return -1;  
    }  
    //...
```

mutable

使 const 类型可改变

`mutable` 关键字使你能够制造一个有限的 `const` 所规定属性的例外。前面曾经说过,如果一个对象被声明为 `const` 型,那么它所有的成员就都是 `const` 型的,它们是不能被改变的。但是有时候你可能要为一个特殊的成员设定一个例外情况。这就是 `mutable` 关键字所能做的。

尽管这样的情况不常出现,但是如果一些成员被当做高速缓冲存储器(为计算临时存储数据)来使用而不是用来永久存储数据时,这种情况就会发生。此时将这样的成员声明为 `mutable` 型是有意义的。

例如,在下面的代码中,`a` 是一个 `mutable` 型的成员。即使将这个类的对象声明为 `const` 型,改变 `a` 也是允许的。

```
class CSilly{
public:
    mutable int a;
    int b,c;
    CSilly() {}
};
//...
const CSilly thingie;
thingie.a = 1;
```

ANSI

`mutable` 关键字是 ANSI C++ 中一个相当新的特征,在 C++ 的早期版本中是不支持它的。

namespace

创建分隔开的名称空间

术语名称空间(namespace)在 C++ 中被多次谈及。通常,名称空间是一组不能互相冲突的符号(例如名字)。但是同一个名字可以无冲突的出现在不同的名称空间中。例如,每个类都定义自己的名称空间,所以在不同的类中使用相同的名字是没有问题的。

在 ANSI C++ 中,你也可以明确地定义名称空间。因为一个成员在没有涉及到它时是不可见的,所以显式地定义名称空间就提供了一个额外的名称域。`namespace` 关键字的语法格式如下所示,其中 `declarations` 部分可以由任何合法的 C++ 代码组成:

```
namespace name {
    declarations
}
```

在名称空间定义的外部,当涉及到名称空间定义内的任何名称时需要格外注意。通过使用作用域分辨符(`::`)你可以直接引用一个符号。

`namespace...name::symbol`

`using` 语句可以使编译器正确识别对这个符号的所有使用而不需要进行核实:

`using namespace...name::symbol;`

`using namespace` 语句可以使编译器正确识别名称空间定义中所有的符号:


```
using namespace namespace_name;
```

例如,假定在一个名称空间中定义了全程变量 Hugh 和 Cary。

```
namespace grants|
    int Cary = 0;
    int Hugh = 1;
|
```

在名称空间定义之外,当涉及它内部的变量时,可以使用 `grants::Cary` 和 `grants::Hugh` 来表示。但是你可以选择去掉前缀。例如:

```
using grants::Cary;
```

现在名 `Cary` 就可以没有限制的用在后面的语句中了。例如:

```
Cary = 0;    //Set grants::Cary to 0.
```

另一方面,还可以使两个变量都可用:

```
using namespace grants;
```

● ANSI

`namespace` 关键字是 ANSI 规范中一个必须的部分,但在早期的一些 C++ 版本中是不支持它的。

operator

重载操作符行为

`operator` 关键字用来帮助定义一个操作符函数,当这个操作符函数应用于特殊的类型时,`operator` 将被用来定义一个特殊操作符(例如 `+`, `-`, `*` 等)的行为。这种用法被称做操作符重载,当将它应用于你自己类的对象时,它允许用户自定义操作的意义。如果某一已重载的操作符和相应的类型被结合在一起,程序就会调用相应的操作符重载函数:

```
operator@ (type1, type2)    //binary
operator@ (type1)          //unary
```

查看第七章以获得更多的有关操作符函数的信息,包括完整的语法规则。

private**限制对成员的访问**

`private` 关键字在两种情况下使用:成员访问和基类访问。私有成员只能在类内被访问。

任何时候只要可能,将一个成员设为私有模式都是一个好的作法。私有成员的好处是你可以添加和删除它们而不会破坏其它的代码。私有成员与其余的程序之间不会发生从属关系。

要将成员声明为私有模式,须将 `private` 关键字放在成员声明的前面。这样在下一个成员访问控制关键字前面的所有成员就都被声明为私有访问模式。此外,当 `class` 关键字用来声明一个类型时,私有访问模式是默认的。在下面的代码中,成员变量 `a`, `b`, 和 `c` 都是私有访问模式:

```
class AClass{
    int a,b;
public:
    AClass(int i,int j,int k);
    double x,y;
private:
    int c;
};
```

当 `private` 作为基类访问模式指定符时, `private` 关键字修改所有继承得到的成员,使它们在派生类中都变成私有模式。在下面的例子中,类 `CDerived` 继承类 `CBase` 中的所有成员,但这些成员都变为私有模式。要记住 `private` 是默认的基于类的访问模式指定符。

```
class CDerived : private CBase{
    //Some declarations
};
```

protected**将对成员的访问限制
在基类与派生类之间**

`protected` 关键字有两种相关的使用:成员访问和基类访问。保护模式的成员只可以在类内和派生类内被访问。

保护模式访问的实际作用是赋予其它程序员使用你所使用的同一个变量的能力。在派生类里,使用它就可以访问基类中保护模式的成员,但还是不能访问私有成员。然而,保护访问模式和私有访问模式除了对派生类的处理上有所差别外,在别的方面是相同的。

为了把成员声明为保护访问模式,须将 `protected` 关键字放在成员声明的前面。这样在下一个成员访问控制关键字前面的所有成员就都被声明为保护访问模式。在下面的代码中,成员变量 `x`, `y` 和 `c` 都是保护模式的。成员变量 `a` 和 `b` 被默认为私有模式。

```
class AClass{
    int a,b;
protected:
    double x,y;
    int c;
public:
    void set_vars(int i,int j,int k);
};
```

此时,派生类中的函数就可以访问除了 `a` 和 `b` 之外的所有成员变量。

如果 `Protected` 作为基类的访问模式指定符, `protected` 关键字将修改被继承的成员的访问属性,基类 (`CBase`) 中的一个公共模式的成员在这个基类的派生类 (`CDerived`) 中变为保护模式。要记住 `private` 是默认的基于类的访问模式。

```
class CDerived : protected CBase {
    //Some declarations
};
```

public

允许对成员进行访问

`public` 关键字在 C++ 有两种使用:成员访问和基类的访问。一个公共成员可以在程序的任何地方被访问。

一个公共成员是一个类的接口的一部分——类将它展示给外部世界。公有成员应该保持最小化,这有利于隐藏私有模式成员和保护模式成员。在使用 `class` 关键字声明的类中默认的是私有访问模式。公有访问模式是结构和联合的默认访问模式。

为声明公有类型,应该将 `public` 关键字放在声明的前面。这样在下一个成员访问控制关键字前面的所有成员就都被声明为公有访问模式。在下面的例子中,成员

变量 a, b 和 c 是私有模式。成员变量 x 和 y 是公有模式。

```
class AClass {  
    int a, b;  
public:  
    double x, y;  
private:  
    int c;  
};
```

Public 如果作为基类的访问控制指定符, public 关键字使派生类不修改从基类中继承得到的成员。这不是默认的形式, 因此不要忘记使用 public 关键字。

```
class CDerived : public CBase {  
    //Some declarations  
};
```

register

将变量放在 CPU 的寄存器中

register 关键字请求将一个变量放置在中央处理器的内部寄存器中。如果在程序中多次使用这个变量, 将其放在存储器中能够提高程序的运行速度。register 关键字按如下形式修改一个数据的声明:

```
register data_declaration;
```

register 声明的变量应该是一个整型或指针类型。例如, 下面的语句将整型数 i 声明为一个寄存器变量:

```
register int i;
```

register 关键字的使用与编译器没有必然联系。它只是寄存器应该如何被分配的一个建议或暗示。为了达到优化目的, 现代的编译器倾向于形成自己的关于寄存器分配的看法, 这使得这个关键字几乎失去了原来的意义。

return

退出函数(可带有返回值)

return 语句使一个函数立即将控制返回给它的调用者, 它通常带有一个返回值。它与其它的语言中 Exit Function 语句相似。return 语句有两种语法格式:

```
return expression ;  
return;
```

第一种语法格式可以为任何非 void 返回类型的函数返回一个值。如下面例子所示。第二种语法格式使用在返回类型是 void 型的函数中,它的目的只是及早退出被调用函数。下面展示 return 关键字使用的例子并返回一个值:

```
long factorial(int n){  
    long result = 1;  
    while(n)  
        result = result * n--;  
    return result;  
}
```

static

将变量分配在数据段内

在 C++ 中,static 关键字有四种独特但相关的使用。这些使用的共同之处是它们将数据放在全局数据段中而不是放在一个对象或函数中。静态数据的生命期是很长的。static 关键字的特殊使用有:

- 赋予一个局部变量更长的生命期
- 限制一个函数在模块中的作用域
- 使类的所有对象中共享一个数据成员
- 限制一个成员函数只能访问静态数据

所有的这些使用有相同的语法。它的格式为:

```
static declaration
```

用法 1: 静态局部变量

static 关键字可以应用于一个局部变量。结果是在保持该变量的局部作用域的同时,赋予这个变量与程序相同的生命周期。在实际使用中,这意味着这个变量在程序调用之间保持它的值。

```
void call_waiting(double x,y){  
    static int number_of_calls = 0;  
    number_of_calls++;  
}
```

```
//...
```

局部静态变量的一个特征是它们只能被初始化一次。在上面的例子中, `number_of_calls` 只被初始化一次,并不是在每次函数调用时都把它初始化为 0。

用法 2: 静态函数

`static` 关键字可以应用于一个全局函数,使这个函数只在它自己的模块内可见。默认情况下,一个函数有一个外部存储类,这使得程序中的其它模块可以访问它。要注意 `static` 关键字只需要应用于函数原型中。

```
static void myfunctiononly(int n);  
void myfunctiononly(int n) {  
    //statements  
};
```

用法 3: 静态数据成员

`static` 关键字可以应用于一个数据成员,这使得这个数据成员可以被类内的所有对象共享。这个数据成员只有一个拷贝被放置到存储器中。为了生成一个静态的数据成员,你不仅要在类内声明它,而且只能在一个模块中定义它,就如同将其当做一个全局变量一样。如下所示:

```
class CStudent {  
public:  
    static long number_of_students;  
    CStudent() { ++ number_of_students; }  
    //...  
};  
  
long CStudent::number_of_students = 0;
```

这个数据成员作为类成员 `class::member` 或者是对象成员 `Object member` 来使用,其中的 `Object` 是一个类的实例。

用法 4: 静态成员函数

`static` 关键字可以用来创建静态成员函数。这样的函数可以使用类中的一个成员,只要这个成员也是静态的。但是这个函数不能使用 `this` 关键字,它不象其它的成员函数那样能够传递一个隐藏的 `this` 指针。使用静态成员函数的目的通常是访问静态数据成员。

例如:

```

class CStudent{
    static long number_of_students;
public:
    CStudent() { ++number_of_students; }
    static long get_num_students(void);
    //...
};
long CStudent::number_of_students = 0;
long get_num_students(void){
    return number_of_students;
}

```

struct

声明结构类

struct 关键字象 class 关键字一样可以声明一个类。在 C++ 涉及到的范围内,用 struct 和 class 关键字声明类的唯一区别是,用 struct 关键字声明的类中的成员的默认类型是公有的。在其它方面 class 关键字的定义和使用都适用于 struct 关键字。

struct 关键字的真实目的是与 C 语言中 struct 关键字的使用方式相兼容。尽管它也支持成员函数,但并不意味着一定要使用它们。因此,你可以象 C 语言中那样来使用 struct 关键字,将其作为一个公有访问类型的数据成员的一个集合。它的语法格式如下所示:

```

struct name [base_class_declarations]{
    declarations
    [object_definitions];
}

```

此处中括号表示其中的项是可选择的,而且 declarations 部分可以声明任意数量的成员变量和成员函数。

下面的代码创建了一个名为 movie_ratings 的类,它有四个成员变量。因为这个类是使用 struct 关键字声明的,所以它的四个成员变量都是公有类型。

```

struct movie_ratings{
    char movie_name[20];
    char director_name[30];
    int Roger;          //1 = thumbs up, 0 = thumbs down
    int Jane;
};

```

switch**多分支选择语句**

switch 语句是一个多分支语句。它首先检测一个整型值,然后它就跳到与它匹配的分支语句去执行。你可以使用 if else 语句来完成同样的工作,但是 switch 语句更易理解且更高效。switch 语句有下面的语法格式,其中中括号仍然表示其中的内容是可选的。

```
switch (expression) {  
    statement           //Any of these may be labeled with  
                        //case constant-expression :  
    [default:  
        statements ]  
}
```

switch 语句的行为是首先计算表达式的值,如果哪个分支的值与表达式的值匹配,就执行此分支后面的语句。例如,假定你有一个打印数字"one","two","three",而不是"1","2","3"的程序。完成这项工作的一种方式是使用一系列的 if else 语句:

```
if (n == 1)  
    printf("one");  
else if (n == 2)  
    printf("two");  
else if (n == 3)  
    printf("three");
```

同样也可以使用 switch 语句来完成。注意每个分支末尾 break 语句的使用。

```
switch(n) {  
    case 1:  
        printf("one");break;  
    case 2:  
        printf("two");break;  
    case 3:  
        printf("three");break;  
}
```

如果不使用 break 语句,程序就会继续执行下一个分支。这是因为 case 只是一个简单的语句标识,而不是一个控制结构。有时你也可以利用这个特点来达到自己的目的。在下面的例子中,y 分支语句执行完后将继续执行 a,e,i,o,u 分支语句:

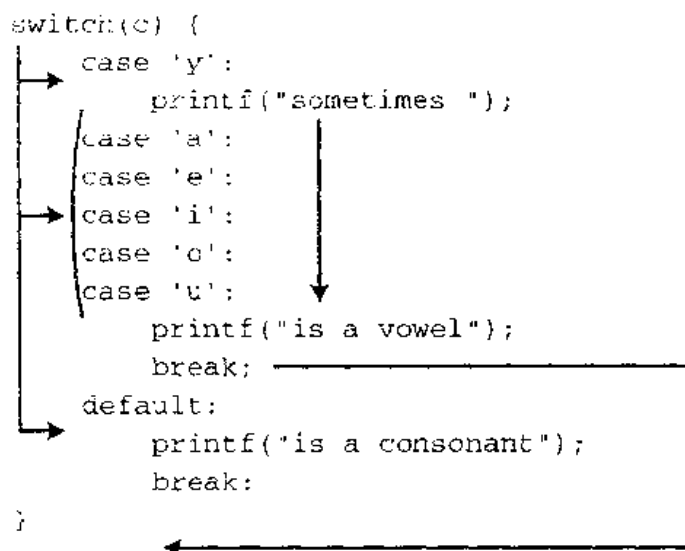

```
switch(c){
    case 'y':
        printf("sometimes ");
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        printf("is a vowel");
        break;
    default:
        printf("is a consonant");
        break;
}
```

如果 switch 表达式的值是 'y', 这个 switch 语句将打印下面的结果:

sometimes is a vowel

图 13-2 总结了 this switch 语句的控制流程。

图 13-2
switch 语句
中的控制流
程



注意 switch 语句不支持字符串或对象作为表达式的值,因为它们不是整数。如果你需要提供依靠判断一个字符串值而执行一系列可选择的操作,就只能使用一系列的 if else 语句并使用 strcmp 函数检测字符是否等同来作为判断表达式。

template

模板

template 关键字声明一个无显著特点的类或函数。一旦你编写了一个模板 (template), 你就可以将其应用于特定的类型中, 例如 int, float 或 double, 或者你也可以将其应用于一个类。模板引入一个普通类型 T 的概念, 其中的 T 稍后在后续操作中可以被任何你希望的类型替换。模板的语法格式如下所示:

```
template< class T > declaration
```

class 关键字在此处有特殊意义, 它可以是任何类型, 并不单指使用 class 关键字定义的类。declaration 可以包含 T 的引用, 它是后面指定的一个类型的替身。declaration 是一个类声明或者是一个函数定义。

一旦定义了一个模板, 你就可以使用它来产生一个基于指定类型的声明。通过填写参数 T 来应用一个模板的语法格式为:

```
template_name < type >
```

模板可以更复杂一些, 它可以带有几个参数, 但其中必须有一个是 class T 参数。

```
template< args > declaration
```

用户可以通过填写所有的参数来应用这样的一个模板:

```
template_name < arg_value >
```

例子 1: 一个简单的类模板

下面是将一个单一类型作为参数的类模板的语法:

```
template< class arg >
class template_name {
    declarations
};
```

参数化类型 arg 被扩充为声明。当模板被使用时, 只要 arg 出现在 declarations 内, 一个实际的类型 (例如 int 或 double 类型, 或者是一个类名) 将代替它。图 13-3 展示了对模板的扩充是如何进行的。

下面是一个简单模板的例子。对于任何给定的类型 T, 模板调用“pair”定义一个新的类型, 这个新的类型由两个类型为 T 的成员组成。

```
template< class T >
```

```
class pair{
    T a,b;
};
```

图 13-3
简单模板的
扩充

```
template <class arg>
class name {
    name <type> vars;
    declarations
};
```

为了使用这个模板,需要指定一个形如 `pair<T>` 一样的类型,其中 `T` 是一个实际的类型:

```
pair<int>    jeans;    //jeans contains two ints
pair<double> gloves;  //gloves contains two doubles
pair<CStr>   glasses; //glasses contains two strings
```

图 13-4 举例说明了 `int` 类型是如何扩充入 `pair<int>` 的。

图 13-4
`pair<int>`
中的模板如
何被扩充

```
template <class T>
class pair {
    T a,b;
};
```

因此类型规范 `pair<int>` 被扩充为包含两个成员的一个类:

```
class pair<int> {
    int a,b;
};
```

同样的, `pair<double>` 和 `pair<CStr>` 按如下所示被扩充。你虽然没有实际看到这个扩充是如何进行的,但这就是 C++ 解释 `pair<double>` 和 `pair<CStr>` 类型规范意义的方式。

```
class pair<double> {
    double a,b;
};

class pair<CStr> {
```

```
    CS: r a, b;  
};
```

例子 2: 一个简单的函数模板

模板的另一个使用是关于函数的。它的语法与上一部分中的语法相似。一个简单的函数模板具有下面的格式:

```
template <class arg>  
type name (args) {  
    statements  
}
```

换句话说, 一个函数模板就是 `template<class arg>` 语法的后面跟随一个函数定义。`arg` 可以出现在函数定义中任何需要它的地方, 它是随后被指定的一个类型的别名。下面的 `switch_values` 是一个简单函数模板的例子。这个模板是有实用价值的。

```
template <class T>  
void switch_values(T &a, T &b) {  
    T temp;  
    temp = a;  
    a = b;  
    b = temp;  
};
```

在这里别名类型 `T` 被用在参数列表中; 因此这个函数模板可以对任何两个类型为 `T` 的变量进行操作。代码同时使用这个类型生成另一个变量 `temp`。下面是使用这个模板的一个例子:

```
#include <iostream, h>  
//...  
int x = 2.0, y = 10.0;  
cout << "x = " << x << endl;  
cout << "y = " << y << endl;  
switch_values<int>(x, y);    //Switch x and y!  
cout << "x = " << x << endl;  
cout << "y = " << y << endl;
```

这个例子的关键语句是:

```
switch_values<int>(x, y);    //Switch x and y!
```

`switch_values<int>` 的使用造成一个隐函数被生成。这个函数的行为使程序看

起来好象已经进入合续语句。(在下面的函数定义中使用 int 代替了每次出现的 T, 编译器所要做的也就是实现这种替换。)

```
void switch_values<int>(int &a, int &b) {  
    int temp;  
    temp = a;  
    a = b;  
    b = temp;  
};
```

● 注意

这个例子使用 switch_value 作为函数名是因为 switch 是 C++ 中的一个关键字, 因此这样命名容易被接受。还要注意例子中使用地址操作符(&)声明了两个引用参数。详情请看术语表中的“reference”主题。最后还要注意的这个模板仅与已定义赋值(=)操作的类型结合在一起才能正确工作。

例子 3: 一个通用的堆栈类

尽管下面例子中的通用集合类仅仅是一个最简实现,但它还是有一定实用价值的。注意参数化类型 T 是如何在类定义中四次出现的。

```
template<class T>  
class stack {  
    T * stackp;  
    int size;  
    int index;  
public:  
    T pop(void) {return stackp[--index];}  
    void push(T item) {stackp[index++] = item;}  
    stack(int sz) {stackp = new T[size = sz];  
        index = 0;  
    ~stack() {delete [] stackp;}  
};
```

这个模板在错误检查方面较差,其中的一个缺陷将在下面指出。因为该模板只定义了一个构造函数,所以必须使用这个构造函数生成堆栈变量。如下所示:

```
stack<int> things(30);  
stack<CStr> strings(20);
```

通过上面的语句生成的堆栈 things 可容纳 30 个整数、strings 可容纳 20 个 Cstr

对象。一旦这两个堆栈被定义好了,就可以对它们进行压入和弹出操作。例如:

```
strings.push("A string");
strings.push("B string");
cout << strings.pop();    //Prints "B string"
cout << strings.pop();    //Prints "A string"
```

`stack<CStr>` 类型规范生成下面的类。在这个类定义中使用 `CStr` 代替了每次出现的 `T`,这也是编译器在解释 `stack<CStr>` 所代表的意思时所要做的。

```
class stack<CStr> {
    CStr * stackp;
    int size;
    int index;
public:
    CStr pop(void) {return stackp[--index];}
    void push(CStr item) {stackp[index++] = item;}
    stack(int sz) {stackp = new CStr[size = sz];
                    index = 0;
    }
    stack() {delete [] stackp;}
};
```

这个生成的类型能够工作是因为赋值(=)和拷贝操作已经为 `CStr` 定义了(查看第七章)。但这个类型是不能正确地处理数组。

例子 4: 带有函数模板的堆栈类

前面部分中的类声明包括函数定义。但是你可以通过提供函数自己的模板的方式在类声明的外部定义一个函数。

```
template <class T>
T stack<T>::pop(void) {
    //...
}
```

使用这种方法能够编写更好的压入(push)和弹出(pop)的实现。下面的版本改进了对错误的检查:

```
template<class T>
class stack{
    T * stackp;
    int size;
    int index;
```

```

public:
    T pop(void);
    int push(T item);
    stack(int sz) {stackp = new T[size = sz];
                    index = 0;
    ~stack() {delete [] stackp;}
};

//Template to define stack template's pop function.
//Check index: if stack empty ,return dummy object.
//Type T must have a default constructor!
//
template <class T>
T stack<T>::pop(void) {
    if (index > 0)
        return stackp[--index];
    else
        {T dummy;return dummy;}
}

//Template to define stack template's push function.
//Check index: if stack full,return 0 (failure).
//otherwise,return 1.
//
template <class T>
int stack<T>::push(T item) {
    if(index < size)
        stackp[index++] = item;
    else
        return 0;
    return 1;
}

```

下面的变量定义生成一个叫做 strings 的堆栈,它的最大容量是 50 个 CStr 对象:

```
stack<CStr> strings(50);
```

这条语句创建了 stack<CStr>类型的实例,它使编译器生成这个类型。当这个类型被生成时,编译器声明两个成员函数 pop 和 push,它们都有 stack<CStr>作用域。下面是这两个函数的原型:

```
CStr stack<CStr>::pop(void);
int stack<CStr>::push(CStr item);
```

这些声明对 `stack<CStr>::pop` 和 `stack<CStr>::push` 使用了函数定义模板, 使编译器生成适当的代码。

在最后的例子中, 这段代码显示了一个模板是可以有多个参数的。这种方法避免了使用构造函数, 相反它使用一个模板参数来决定堆栈大小。

```
template <class T, int sz>
class stack{
    T arr[sz];
    int index;
public:
    T pop(void);
    int push (T item);
    stack(){index = 0;}
};

//Template to define stack template's pop function.
//Check index: if stack empty, return dummy object.
//Type T must have a default constructor!
//
template <class T, int sz>
T stack<T, sz>::pop(void){
    if (index > 0)
        return arr[--index];
    else
        {T dummy; return dummy;}
}

//Template to define stack template's push function.
//Check index: if stack full, return 0 (failure).
//Otherwise, return 1.
//
template <class T, int sz>
int stack<T, sz>::push(T item){
    if (index < sz)
        arr[index++] = item;
    else
        return 0;
    return 1;
}
```

为了使用这些模板, 需要在声明中指定类型和大小。下面是一些例子:


```
stack<int,10>    ten_little_integers;  
stack<float,20> float_my_boat;  
stack<CStr,50>  tons_o_strings;
```

第一个声明生成一个名为 `ten_little_integers` 的堆栈,它的最大范围是 10。当编译器生成这个类型时,两个函数也被声明了:

```
stack<int,10>::push  
stack<int,10>::pop
```

这些声明使得编译器生成适当的函数定义。

● ANSI

ANSI C++ 规范需要对模板提供支持,但早期版本的 C++ 是缺乏这个支持的。

this

指向当前的对象

`this` 关键字是一个对象的指针。当你调用一个成员函数时,函数得到一个当前对象的指针,这样对象就可以通过这个指针调用成员函数。`this` 关键字指向当前的对象,但它的使用是隐含的。

理解这个概念的另一个方式是:隐含的指针(`this`)使一个函数知道“这个对象”是什么。它让代码知道它正在对哪个对象进行操作。

对这个关键字的一个使用是阐明一个变量正涉及到的的是一个类而不是别的什么东西。例如,下面的代码使用 `this` 关键字为成员变量 `x` 和 `y` 赋值。这种用法在此处是必要的,因为如果不这么用在 `x`、`y` 与类中成员变量 `x`、`y` 匹配之前,已经与函数中的局部变量 `x`、`y` 匹配了。

```
CPoint::set(double x,double y){  
    this->x = x;    //init class member x.  
    this->y = y;    //init class member y.  
}
```

这个关键字的另一个使用是返回当前对象,这里的当前对象在某些操作符函数中是必须的(请查看第七章)。因为 `this` 是一个对象的指针,所以 `*this` 就代表对象本身。

```
CStr& CStr::operator = (const CStr &source) {
```

```
    copy(source, get());  
    return * this;          //Return myself!  
}
```

throw

生成或传递异常

throw 关键字既可以用来生成一个异常也可以用来传递一个异常,就象在下面的部分(try)中描述的那样,它是一个运行时发生的事情并要求被立即处理。这个关键字有两种语法形式:

```
throw;  
throw exception-object;
```

第一个版本可以使用在一个异常句柄中。相当于说:“我不能处理这个异常”,并将这个异常传递到下一个 catch 块。

第二个版本生成一个异常。它的结果是把程序控制转到适当的异常句柄处。异常对象可以是任何类型的对象。程序控制转到第一个 catch 块接受这个类型的一个参数。查看 try 关键字以获得更多的信息。

try

定义异常处理块

try 关键字连同与它相关的关键字 throw 和 catch 一起处理 C++ 异常。这是处理运行时发生的事件的一个结构化的方法,它具有比传统的错误处理技术更多的好处。

try 和 catch 关键字使用下面的语法格式,它表明你可以有一个或多个 catch 块(仅有一个 catch 是必需的):

```
try {  
    statements  
}  
catch(arg1-declaration) {  
    statements  
}  
catch(arg2-declaration) {  
    statements  
}  
...
```

```
catch(argN_declaration)|  
    statements  
{
```

这段代码的结果是执行地无条件 try 语句后面的语句;这些语句是正常流程的一部分。如果在执行这些语句期间发生了一个异常(即使这个异常是发生在一个函数内部),程序控制就转移到最近的 try catch 结构去执行。

程序寻找与异常发出来的类型相匹配的第一个 *arg_declaration*。如果异常发出的类型与 catch 参数的类或者是由这个类直接或间接派生出的类的类型相同,那么它们的类型就是匹配的。因此,如果 *arg1_declaration* 的类型是 *ioerror*,它就能够捕获任何从 *ioerror* 派生出的类型的异常。如果一个异常产生了但没有被捕获,程序就会被终止。

每个 *arg_declaration* 包括一个异常类型和一个可选择的参数名。中括号表示其中的部分是可选择的。如果 *object* 被指定的话,它就代表异常对象,当异常被生成时,它将被传递(查看 throw 关键字)。

```
exception_type [object]
```

arg_declaration 的语法还可以使用省略号(...),它表明这个 catch 块可以捕获任何类型的异常。这个语法可以用来编写一个默认的异常处理。

...

如果一个异常已经被成功地处理而且程序没有终止,那么程序将在 try catch 结构的末尾后继续执行。同样,如果 try 语句块中所有的语句都执行了但没有任何异常发生,程序也会在 try catch 结构的末尾后继续执行。除非一个异常发生了,否则 catch 语句块将不被执行。

异常处理的普通概念

异常通常是一个要求立即被处理的程序事件。绝大多数异常描述的是某些类型的运行时错误,但它也可以用于其它的目的。

错误处理的传统方法是当调用一个函数时检查它,如果有错误的话就返回一个错误代码。对于简单的事件这种方法是好的,但是当错误在程序的深处发生时,它必须通过一系列的返回传递到程序的表层如图 13-5 所示。

C++ 异常处理使你可以将所有的错误处理集中在一个特定的区域。当一个异常生成时,程序控制立即转到适当的错误处理处执行。即使异常在函数的深层发生

也是这样的,如图 13-6 所示。

图 13-5
C++ 中错
误的传播

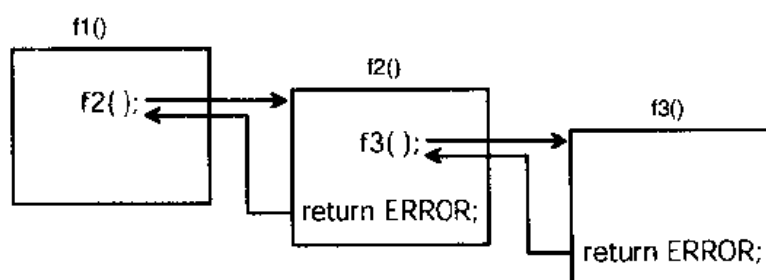
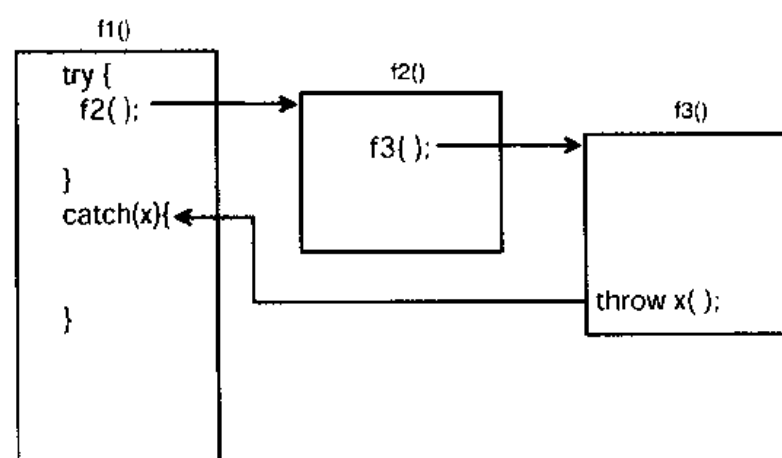


图 13-6
C++ 异常
处理



异常处理的例子

下面例子的开始处定义了一个异常处理类 file_err:

```

#include<stdio.h>
#include<string.h>
#include<stdlib.h>    //required to use exit()

class file_err{
public:
    char file[256];
    char mode[10];
    file_err(char * f, char * m)
        {strcpy(file,f);strcpy(mode,m);}
};
  
```

根据上面的声明,main 函数能够被编写成具有捕获 file_err 类型异常的能力。在正常的流程执行期间,main 函数调用三个函数:

```
void main() {
    try {
        open_file();
        init_vars();
        run_prog();
    }
    catch (file_err ferr) {
        close_existing_files();
        printf("File error opening %s in %s mode.",
            ferr.file, ferr.mode);
        exit(1); //exit is a library function
    }
}
```

一个 file_err 类型的异常可以在任何地方生成,并且可以被正确地处理。例如,如果一个文件未被打开,函数 open_file 就产生一个异常。

```
void open_file(void) {
    FILE * fp;
    char * frame = "MYFILE.DAT";
    fp = fopen(frame, "r");
    if (fp == NULL)
        throw file_err(frame, "r");
    //...
```

要注意 throw 语句调用 file_err 构造函数生成一个无实际意义的对象。该异常对象在这个函数的任何地方都是不需要的,因此代码生成一个未命名的对象作为 throw 语句的参数。

●—注意

查看附录 C 来得到 ANSI C++ 中预定义异常的一个列表。

typedef

为一个类型定义别名

typedef 关键字为复杂的类型生成一个别名。它在 C++ 中不象在 C 中那样重要,因为在 C++ 中你不需要用它将结构标记符变成类型。但是,typedef 还是有它的用处的。只须将 typedef 关键字放在变量声明的前面就可以使用它。

```
typedef declaration;
```

typedef 将指定的类型用一个别名来代替,而不是把目标标识符变成一个变量。如果没有使用 typedef,后面还是一个完整的类型和定义语句。例如,下面的声明将 LSTRING 作为一个大小为 256 的字符数组来定义:

```
typedef char LSTRING[256];
```

给定这个声明,你就可以使用它生成几个字符串,好象 LSTRING 是一个原始类型一样:

```
LSTRING a,b,c;
```

上面的声明与下面声明的效果是一样的:

```
char a[256],b[256],c[256];
```

一旦定义完成,一个 typedef 类型可以象其它类型那样使用。甚至可以在非常复杂的声明中使用一个 typedef 类型。例如,在下面的声明中,sarray 被定义成一个包含 100 个 char * 类型的指针的数组,get_name 是返回一个 char * 指针的函数:

```
typedef char * STYPE;    // STYPE = a char * pointer
```

```
STYPE sarray[100];      //sarray is array of 100 ptrs.
```

```
STYPE get_name(void);   // get_name returns char *.
```

● 注意

typedef 关键字还有另外一个作用。每次使用 typedef 创建一个独特的类型都要涉及到 typeid 操作符。例如,下面的语句专门将 SILLYTYPE 创建一个独特的类,尽管它和 int 一模一样:

```
typedef int SILLYTYPE;
```

typedef 语句通常被放置在一个头文件中,这样每个模块都能得到相同的类型定义。typedef 有比 #define 更强的作用。虽然下面的两条语句看起来是等价的,但它们产生不同的结果:

```
typedef int * PTI;      //Declare PTI as alias for int *.
```

```
#define PTI int *       //Replace "PTI" with "int *".
```

当象下面的语句那样在一条语句中定义几个变量时,这个区别是明显的。如果使用 #define 来生成 PTI,只有第一个变量 p1 变成指针;当使用 typedef 来生成 PTI 时,全部三个变量都被变成指针。

```
PT1 p1, p2, p3;
```

union

声明带有交迭数据的类

如果你从未使用过 `union` 关键字的话,它的性质可能显得非常古怪。`union` 关键字在同一地址声明一个或更多个数据成员。如果你打算在不同的时刻存储不同类型的数据,这样做是很有用的。

`union` 关键字的语法和 `class` 和 `struct` 关键字的语法相似。同使用 `struct` 关键字一样,利用 `union` 声明的类中可以包含成员函数和成员变量,而且访问模式默认是公有的。

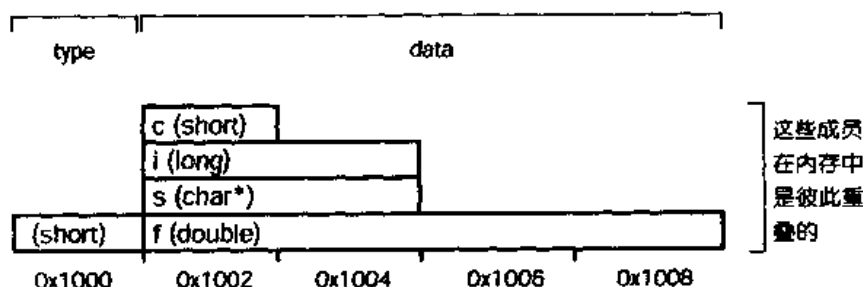
```
union name [base_class_declarations] {  
    declarations  
}[object_definitions];
```

当使用 `union` 关键字时,所有的成员变量共享相同的地址。例如,下面的代码将一个联合嵌入到一个结构中,变量 `c`, `i`, `s` 和 `f` 有相同的地址。

```
struct variant {  
    short type;  
    union {  
        short c;  
        long i;  
        char * s;  
        double f;  
    } data;  
};
```

图 13-7 展示了在存储器是如何存储这些变量的。

图 13-7
存储不同类型数据时存储器的布局



这个结构生成的变量能够存储字符串型、整型或浮点型数据的一个,但在任一时刻只能用来存储某一类型的数据。下面的代码生成两个对象。它将一个浮点值赋给第一个对象,将一个字符串赋给第二个对象。

```
enum { CHAR, INT, STR, FLOAT };
struct variant v1, v2;

v1.type = FLOAT;    //Use v1 to store a float.
v1.data.f = 27.5;
v2.type = STR;      //Use v2 to store a string.
strcpy(v2.data.s, "Hello");
```

为得到相关的信息,查看术语表中的“匿名联合”。

using

允许直接使用名称空间(namespace)

using 关键字使你在使用其它名称空间的一个符号时不必使用作用域分辨符(::)。使用这个关键字既可以直接访问一个特殊的符号,也可以访问一个完整的名称空间。例如,在下面的代码中的头两条语句使你能够访问 cats 名称空间中的 CB 和 BillTheCat。

最后一条语句使你能够访问整个 cats 名称空间。

```
using cats::CB;
using cats::BillTheCat;
using namespace cats;
```

要了解更多的信息,请查看 namespace 关键字。

virtual

将作用域的确定推迟到程序运行时进行

在 C++ 中,虚函数是一个非常重要的概念。它在别处已经进行过详细的讨论。这一部分只是总结它的语法。可查看第九章有关虚函数的介绍。virtual 关键字也可以用来生成虚基类。

用法 1: 虚函数

要将一个成员函数声明为虚函数, 只须将 `virtual` 关键字放在这个函数原型的前面就行。

```
class name {  
    //...  
    virtual function-prototype;  
    //...  
};
```

虚函数必须是一个类的成员函数(这个类也可以是由 `struct` 或者 `union` 关键字来声明的)。不能将虚函数的定义放在类定义的内部, 因为虚函数是不能被内联的。一旦一个成员函数被定义为虚拟的, 那么在所有的派生类中这个函数都是虚拟的, 而且这些函数不必再使用 `virtual` 关键字来重复声明。

用法 2: 虚基类

`virtual` 关键字也可以应用于基类。即使是多重继承造成基类被继承多次, 虚基类也只将它的成员的一个拷贝提供给它的派生类。这种情况只在非常复杂的继承中才会出现, 它要涉及到多重继承和至少三个派生类。例如, 在下面的例子中, 尽管 `CKid` 通过 `CMom` 和 `CDad` 基类进行多重继承, 但它只继承了 `CAncesor` 成员的一个拷贝。

```
class CMom : virtual public CAncesor {  
    //...  
};  
  
class CDad : virtual public CAncesor {  
    //...  
};  
  
class CKid : public CMom, public CDad {  
    //...  
};
```

void

声明空类型

`void` 关键字有一个普通的、失真的意思就是它声明一个空类型, 但对它的最好的理解是支持下面三种具体使用方式的一种。

- 声明一个无返回类型的函数:

```
void print_num(int n);
```

- 声明一个没有参数的函数原型。这比一个空的参数列表能提供更多的信息,空的参数列表是不明确的。一个 void 参数列表表明函数是没有参数的:

```
int get_num(void);
```

- 声明没有基类型的一个指针。这样的指针存储一个地址,但不能用它来访问数据:

```
void* p = malloc(n);
```

void * 类型的指针需要遵守许多规则。在没有使用 C 语言中的 cast 或者 C++ 的 reinterpret_cast(请查看第十二章)进行强制类型转换前,它们是不能被撤消的。另一方面,不必使用强制类型转换就可以为一个 void 型的指针赋值。void 型指针的度量因子是 1,与 char * 类型的指针一样。当函数返回数据的类型未知时,就返回一个 void * 型指针。

● C / C++

C 语言的严格性较 C++ 语言差得很多,它允许将一个 void 型的指针赋给其它类型的指针而不必使用强制类型转换。C++ 是不允许这样做的。

空类型的整个概念让人感到陌生,在 FORTRAN 或 BASIC 语言中没有这种类型。但是它在 C++ 中通常是有用的。

volatile

要求对数据进行刷新

volatile 关键字暗示某对象是不稳定的。在计算机术语中,一个外部用户可以改变一个可变的对象而不会被警告,这个可变对象可以是操作系统或计算机时钟。每当程序涉及到这个对象时,都需要执行直接内存访问(DMA)操作。编译器不能将这个值放在寄存器中并使它保持当前的状态。

当将 volatile 关键字应用于一个声明时,它就会通知编译器这个数据对象是可变的,在每次访问它之前程序必须更新它的值。

```
volatile data_declaration;
```

下面的例子中声明了两个可变的数据变量 x 和 y 还有一个可变的指针 * p_sys。指针本身不需要特殊的处理,但对 * p_sys 的每次引用都需要一个直接的存储器访问。

```
volatile int x, y, * p_sys;
```

volatile 关键字的语法规则将下面的信息反映给 const 关键字: 只有 volatile 型指针才能指向 volatile 型数据。让人费解的一种用法是一个变量既可以是 const 型的又可以是 volatile 型的:

```
extern const volatile int * p_to_sys_clock;
```

这个声明表明一个外部用户可以改变这个指针指向的数据而不会被警告, 但是程序是不能改变这个数据的。这就意味着系统可以改变这个数据值但程序却不能改变它。

● 注意

volatile 关键字主要出现在系统程序中。

While

重复执行循环

while 循环是 C 和 C++ 中的一个基本的控制结构。当指定的条件满足时, 它就执行某条语句。它的语法格式为:

```
while (expression)
    statement
```

只要表达式为真(非零)循环就继续执行。statement 部分通常是一个复合语句。例如, 下面的代码打印从 1 到 100 的所有整数。

```
#include <stdio.h>
//...
int i = 1;
while (i <= 100){
    printf("%d, ", i);
    i++;
}
```

有许多简洁的方式可以用来编写同样的一个循环。查看 for 语句(创建一个带有计数器的效率较高的循环)和 do while(在 while 子句中提供一个变数)语句。

第十四章

预处理器指令,宏和运算符

当编译器编译程序时,它做的第一件事是进入预处理阶段。在这一阶段,甚至可以人为地将编译器视为一个不同的实体——预处理器。该阶段中,编译器读入头文件、决定编译哪些行的源代码并执行文本替换。

预编译阶段的优越性在于它在编译及运行程序之前执行了某些特定的操作。这些操作并不添加额外的程序执行时间。打个比方来说,如果执行时间是金钱的话,那么预处理器命令就是免费的。同时,这些命令也不与程序运行时所发出的任何指令相对应。所以,你在使用预处理器指令时就需要兼顾实际的运行情况。

本章描述了预处理阶段的三个基本元素:

- 指令,在程序编译前所执行的特定命令。
- 预定义宏,在编译时被转换成实际的命令信息。
- 预处理器运算符,在 `#if` 和 `#define` 指令中使用。

预处理器语法与 C++ 的其他语法有所不同。指令在一行的末尾自然结束,而不需要使用分号。你可以用续行符(`\`)来摆脱物理行的限制。另外,指令必须从第一列开始。

指令

有一半预处理器指令提供了对条件编译的支持,条件编译的作用是用来维护同一程序的不同版本。这些指令包括:

<code>#if</code>	<code>#ifndef</code>	<code>#elif</code>
<code>#ifdef</code>	<code>#else</code>	<code>#end</code>

其余的预处理器指令提供了对其他功能的支持,例如包含头文件和宏定义:

# define	# include	# pragma
# error	# line	# undef

下面按字母顺序详细介绍每一条指令。

define

定义符号或宏

define 指令有三个主要目的:第一,提供了创建符号常量的有效途径;第二,使你能够写宏指令,这些宏指令看上去象函数调用,但实际上是通过文本替换来实现;第三,简单地定义某些符号作为 # ifdef 指令的开关。# define 语句有以下三种语法格式:

```
# define identifier replacement
# define identifier(arg [, arg]...) replacement
# define identifier
```

在格式中,中括号和省略号表示你可以使用任意个数的参数,只需每个参数之间以逗号分隔即可。第一个空格符(不计括弧和引号中的空格符)之后为 replacement 文字。

续行符(\)可用来建立较长的 replacement 字符段。

用法一:符号常量

有些程序用到了一些很重要但又非常难记或难以输入的数字。所以最好事先将它们转换成符号常量。在转换时,通常是以大写形式表示符号常量,以便将它们与变量名区分开。例如:

```
# define PI 3.14159265
```

有了这条 # define 指令,每当程序中出现 PI 时,预处理器就将 PI 替换为数字 3.14159265。

● 注意

若符号出现在注释、被引号括起的字符串或一个较长的单词的一部分中时,它不会被替换。例如,PIG 中的 PI 就不会被替换。

另外,你也可以通过将 pi 定义为一个 const 变量来达到同样的效果。例如:

```
const double pi = 3.14159265
```

然而,两者之间有一定的区别。如果 PI 是一个符号常量,它就是一个真正意义上的常量。在程序里使用 PI 如同直接输入数字。这样一来效率就有所提高。作为一个真正意义上的常量,PI 可在程序运行之前折算成常量表达式,从而加快了运行速度。例如,在下列代码中,PI / 2 在编译时而非运行时已经被计算出来了:

```
x = sin(y * (PI / 2));
```

符号常量的另一个特点是,在数组定义时可被用作设定数组大小。这是符号常量可被视为真正意义常量的另一结果。

```
#define ROWSIZE 30
#define COLSIZE 20
//...
int matrix1[ROWSIZE][COLSIZE]
int matrix2[ROWSIZE][COLSIZE]
```

从另一方面来说,const 变量利于编译和调试程序,这是因为编译器和调试器可通过名称来识别变量(而符号常量的名称已经被实际值所替换了,故难以识别)。同时,对于 C++ 的面向对象编程,const 变量可通过定义特定类的作用域来保持面向对象的特性。

用法二:宏指令

宏指令看上去类似于函数调用,但它们是通过文字替换而非真正的函数调用来实现的。例如,有以下定义:

```
#define max(A,B) ((A)>(B)? (A):(B))
```

预处理器在编译前用下列表达式来替换表达式 max(i,j) :

```
((i)>(j)? (i):(j))
```

宏指令使用在类似于上述的简单环境。然而,与真正的函数调用相比,宏指令有许多缺陷。例如,你不能限定参数的类型。另一个缺陷是你不能够使用复杂的语句,因为宏指令中的所有的一切都必须压缩于一条语句之内。通过使用 inline 函数(参见第十三章的 inline 关键字),可以解决这些缺陷。类似于宏指令,inline 指令被直接扩展于源代码体内,这就优化了运行速度,但很可能是以增加了可执行程序的长度为代价的。

在替换模式中对每一个参数使用括号是一个良好的编程习惯,因为这使得在解算周围的运算符之前强行地换算了参数。不然的话,如果参数本身含有运算符,就可能导致意外的(甚至是错误的)结果。

用法三:控制编译

一个符号可被定义为条件的开关。而这个开关能够控制 `# ifdef` 和 `# ifndef` 指令的操作。

`# ifdef` 和 `# ifndef` 指令并不在意一个符号是如何定义的,仅仅在于它是否被定义了。你可以定义一个符号而不用给出它的替换值;在这种情况下,该符号由一个空串替换。然而,只要 `# ifdef` 及其类似指令应用到它了,它就仍是一个合法符号。

在下例中,符号 `USE-8086` 被定义了;这就导致了预处理器编译 `# ifdef` 和 `# endif` 之间的代码。

```
#define USE-8086
//...

#ifdef USE-8086
//Optimize for 8086 processors.
#endif
```

`# elif`

在另一条件下编译

`# elif` 指令("else if")建立了另一个条件来作为一个 `# if /# endif` 模块的一部分。`# elif` 指令如同 `# if` 指令一样定义了编译条件。该指令定义了一个常量整数表达式:

```
# elif constant-expression
```

如同 `# if` 指令, `constant-expression` 可包含常量以及大多数 C++ 运算符,但是它不包含 `sizeof` 运算符、类型转换或 `enum` 常量。表达式可以包含特别的 `define` (symbol)运算符,如果 `symbol` 预先定义过了,就判为 1,否则就判为 0。

关于条件编译以及完整的语法可参见 `# if`。

`# else`

在条件为假的情况下编译

`# else` 指令标注了一个条件编译语句集合的最后一个语句块。该语句块仅在所有的前述 `# if` 和 `# elif` 条件为假的情况下才被编译。从语法上来说,该指令类似于 `else` 指令,但是,它是用于条件编译的,而非在运行时控制程序流。

一个相应的 `#endif` 指令终止 `#else` 语句块。

```
//...  
#else  
statement-block-n  
#endif
```

完整语法请参见 `#if`。

`#endif`

终止条件块

`#endif` 指令是条件编译语法中所必需的。只要你使用了 `#if` 指令或类似指令 (`#ifdef` 和 `#ifndef`), 你就必须用 `#endif` 来标注该条件编译语句块的结束。

`#endif` 指令单独出现在一个语句行中:

```
#endif
```

它终止了以 `#if` 指令开始的语法。关于条件编译及完整语法的讨论参见 `#if`。

`#error`

停止并汇报错误

`#error` 指令使编译器能够立即终止编译并且打印出一条信息。该指令具有以下语法格式:

```
#error message
```

例如, 如果符号 `ERR` 已被定义, 则下列代码使编译器终止并且输出一条错误信息:

```
#ifdef ERR  
#error Assumptions violated here.  
#endif
```

当编译器处理该 `#error` 指令时, 就终止编译并且打印出当前的行号和一条信息, 信息如下:

```
Assumptions violated here.
```

当违反了一些基本的程序假设环境时, `#error` 指令就提供了一条阻止继续编译程序的途径。特别是当继续编译的程序起不到任何作用时, 你就可以使用该指令来

避免此类情况的发生。最好使用 `message` 来提供有用的程序调试信息。

#if

在条件成立时编译

尽管 `#if` 指令看上去和 `if` 语句的使用差不多(参见第十三章),但是该指令作用效果却是完全不同的。`#if` 指令主要有两种使用:条件编译和多行代码的暂时注释。条件编译对于无须复制源代码而保有程序的多个版本是很有用的。

一条 `#if` 指令标注了一个条件编译块的开始。如果由 `#if` 指令所指定的表达式是非零的(真的),则 C++ 处理器编译随后的语句,直到另外一个相匹配的 `#elif`、`#else` 或 `#endif` 指令出现。

```
#if constant-expression
statement-block-1
[ #elif constant-expression
statement-block-2 ]
[ #elif constant-expression
statement-block-3 ]
.
.
.
[ #else
statement-block-n ]
#endif
```

此处的方括号表示可选项;该语法的唯一必要部分是 `#if` 和 `#endif` 指令。你可以使用任意数量的 `#elif` 指令。你也可以使用最多只能出现一次的 `#else` 指令,在使用过程中,它必须出现在所有的 `#if` 和 `#elif` 指令之后。

C++ 预处理器检查每一条常量表达式(`constant-expression`),直到其中的一条表达式为真(非零)。此时,相应的语句块(`statement-block`)就被编译。每一个 `statement-block` 都可以包含任意形式的 C++ 代码,包括说明、指令和可执行语句。

每一条 `constant-expression` 是由常量和运算符所构成的 C++ 表达式,但它不能包含 `sizeof` 运算符、强制类型转换或 `enum` 常量。表达式中的常量通常是由 `#define` 指令或命令行设置所预先定义的符号。

用法一:条件编译

条件编译是保有一个程序多个版本的有效方法。它在许多情况下都很有用。你可能正在为多个操作平台编写一个程序,而且发现不得不为每一个平台重写程序源

代码的某一特定部分。你也可能想给程序的正式发布版建立一个调试版本,以便获得调试诊断信息。

这就产生了一个进退两难的局面。当你在修改程序的各个不同版本时,就不得不做很多类似于添加和删除之类的额外工作。但是如果你想保留程序的各个完整版本时,又会占用大量的磁盘空间。更糟糕的是,每当你想给程序添加一个新特性时,就不得不给程序的每一个版本进行同样的操作。

解决这些问题的方法就是条件编译,这是一种将程序不同版本之间的特定代码部分分离出来的方法。例如,对不同的计算机想采用不同字长系统。首先,定义一系列常量:

```
#define SHORT    0
#define LONG     1
#define REAL     2
```

现在,对不同字长的系统进行重编译时,只需修改一个语句行,该行定义了系统的 COORDSYS 值。如下:

```
#define COORDSYS REAL
```

其实,有许多编译器都支持命令行或开发环境选项来将一个符号定义为编译命令的一部分,而不需要 #define 语句;这样的话,你就不需重写代码行。详情参阅你所使用的编译器的文档。

程序的余下部分检验 COORDSYS 的值来决定哪些部分需要编译:

```
#if COORDSYS == SHORT
short x,y;
#elif COORDSYS == LONG
long x,y;
#elif COORDSYS == REAL
double x,y;
#endif
```

请注意 #if 指令同 if 语句的区别。#if 指令与程序运行的条件无关。

用法二:注释语句行

有时候,你可能需要注释掉一些源代码行;即暂时性地从源程序中删除这些行,但又使这些代码很容易恢复。达到这一目的的一个明显方法是采用 C 类型的开始注释符(/*)和结束注释符(* /),将它们置于需要注释掉的代码段的两头。然而,

如果其中的某些行已经使用了注释符,该方法就会出错。

一个解决方法是使用 `#if 0` 和 `#endif` 来暂时中止一组语句的编译。编译器就能够跳过这些行,如同它们已被置入一个注释块。例如:

```
#if 0
int i,j;      /* Declare i and j as int.  * /
double x,y;   /* Declare x and y as double. * /
#endif
```

不同于多行注释符的是, `#if` 指令能够嵌套任意层数。

ifdef

如果符号被定义 则进行编译

`# ifdef` 指令或许是“if”指令集中最常用的一条指令。该指令有以下使用语法:

```
# ifdef symbol
```

其意如同:

```
# if defined(symbol)
```

如果已经定义了 `symbol`,紧接在 `# ifdef` 之后的代码行就被编译,直到下一个匹配的 `# elif`、`# else` 或 `# endif` 出现。`symbol` 的值并不重要;唯一重要的是它是否被定义。`symbol` 甚至可被定义为一个空串,如下符号 32-BIT-SUPPORT 所示:

```
# define 32-BIT-SUPPORT
```

许多编译器支持编译器选项来定义符号。通过使用这样的选项,你能够由命令行、批处理文件或开发环境来控制编译程序。详情请参阅你的编译器文档。

关于条件编译的具体讨论及示例,请参见 `# if`。

ifndef

在符号未定义的条件 下编译

`# ifndef` 指令也常用于条件编译。该指令使用语法如下:

```
# ifndef symbol
```

其意如同:

```
#if ! defined (symbol)
```

如果符号未被定义,接续在 `#ifndef` 之后的代码行就被编译,直到下一个匹配的 `#elif`、`#else` 或 `#endif` 出现。当你仅想通过简单的开关条件来控制条件编译时, `#ifdef` 和 `#ifndef` 指令是很有用的。

`#ifndef` 的一个实用之处是确保一个特定的文件仅被包含一次。例如,下列代码包含了文件 `myproj.h` 并确保了它不被再次包含。这四行程序可以添加在不同文件的任意处及任意次数,但是 `myproj.h` 仅被包含一次:

```
#ifndef HEADER-INCLUDED
#include "myproj.h"
#define HEADER-INCLUDED
#endif
```

关于条件编译的具体讨论及示例,请参见 `#if`。

#include

读入头文件

`#include` 指令使得 C /C++ 预处理器在编译期间将另一文件读入当前源文件。例如,如果你使用了指令 `#include <stdio.h>`,其效果就如同将文件 `stdio.h` 的全部内容读入你的当前程序。

虽然任意一个文件都可被指定为 `#include` 的参数,但最好还是使用头文件作为该指令的参数。这些文件中包含了较多模块所需要的声明和指令。

`#include` 指令支持下列两种语法格式:

```
#include "filename"    // Project header file
#include <filename>    // Standard lib header file
```

一旦遇到 `#include`,编译器就暂停读入当前文件,而读入由 `filename` 所指定的文件。通过这种方式所读入的所有文件都象是作为一个长的连续的源文件的一部分来进行编译。该过程可进行任意层次的嵌套,故一个被包含的文件可以包含其他文件。

两种 `#include` 的用法的区别在于使用引号时("filename"),C /C++ 预处理器先在当前目录下搜索该文件,然后再搜索其他目录。两种用法都在标准文件包含路径下查找文件。在 DOS 和 Windows 系统中,该路径由 `INCLUDE` 环境变量所指定。

一般说来,你应该使用第一种语法(`#include "filename"`)来定义自己的头文件,这些文件通常保存于工程文件的目录下。第二种语法最好是用于包含标准头文件,如 `stdio.h` 和 `stdlib.h`。例如:

```
#include <stdio.h>
```

● ANSI

在 ANSI C++ 中,你可以使用 `#include` 从虚拟头文件中直接获得标准库声明,如 `#include <cstdio>`。这些指令可能映射了一个物理头文件,也可能并未映射一个物理头文件。在以后,这是一个支持标准库的较好方式,因为它有可能提高效率。第十五章的每一个主题都提供了两种方法。

#line

设置行号与源文件

`#line` 指令设置了一个新的行号和一个新的源文件名,如宏 `_LINE_` 和 `_FILE_` 所反映的那样。一般很少使用该指令,它对物理源文件不产生任何作用。该指令有下列语法规则,其中,第二个参数为可选项:

```
#line num ["filename"]
```

在编译器处理了该行之后,程序下一行的当前行号就被设为 `num`;其后的行号都保持自然增加。若指定了 `filename`,它就成为了 `_FILE_` 的新值。`_FILE_` 的设置值含有引号。例如,见下列代码:

```
#line 100 "source1"
#include <stdio.h>

void main() {
    printf("%d: %s \n", _LINE_, _FILE_);
    //...
```

当执行了上述源代码之后,打印出下列结果:

```
103: source1
```

#pragma

设置编译器所指定的特性

`#pragma` 指令设置了编译器所指定的条件。该指令用法如下:

`#pragma arguments`

参数值以及它们的意思由你所使用的编译器提供。详情请参阅你的编译器的文档。

undef

删除符号定义

`#undef` 指令删除符号定义。(符号可能是由先前的 `#define` 指令或是在命令行所定义的。)虽然 `#undef` 并不常用,但当你想要暂时定义一个符号或是希望通过删除一些符号来关闭编译器的开关时,该指令还是很有作用的。其语法很简单,无须顾及符号名有没有参数值来对应:

`#undef symbolname`

预处理器读取该行之后就认为 `symbolname` 是未定义的。例如,下列源代码临时定义了一个符号 `LINESZ`,在代码末尾,`LINESZ` 被解除了定义。

```
#define LINESZ 81
//...
char inputstr[LINESZ];
//...
#undef LINESZ
```

注意, `symbolname` 并不需要事先被定义以保证 `#undef` 指令正常执行。

预定义宏

C++ 定义了一系列符号来报告程序编译时的事件状态。该方式的典型应用是报告正在运行的程序版本和程序出错之处。这些预定义的符号(也称作“宏”)有:

预定义宏	描述
<code>__cplusplus</code>	如果按 C++ 方式编译源文件,该宏就被定义。
<code>__DATE__</code>	被引号括起的包含编译日期的字符串。
<code>__FILE__</code>	被引号括起的包含源文件名的字符串。
<code>__LINE__</code>	当前行号。
<code>__TIME__</code>	被引号括起的包含编译时间的字符串。

下面几个小节详细描述了每一个宏。

__cplusplus**在 C++ 模块下被定义**

如果当前的文件是 C++ 文件, `__cplusplus` 宏就被定义。大多数 C++ 编译器可以读取扩展名为 .C 的文件, 将其作为 C 语言代码进行编译。`__cplusplus` 宏表示可以放心使用特定的 C++ 特性。例如:

```
#ifdef __cplusplus
// Use C++ specific features...
int i = get_random();
#endif
```

如果源文件以往是按 C 语言方式进行编译的, 对 `i` 的声明就会由于它含有函数调用而导致出错。使用 `#ifdef` 和 `__cplusplus` 可阻止这样的错误发生。这对于在 C 和 C++ 源文件之间来回复制大量的源代码很有好处, 当然, 此类情况并不普遍。

● C / C++

参阅附录 B 可获知 C++ 所支持的而 C 不支持的特性, 以及相反情况。

__DATE__**当前日期**

宏 `__DATE__` 被转换为由引号括起的包含当前编译日期的字符串, 以“月日年”的形式出现。例如, 考虑如下代码:

```
#include <stdio.h>
//...
printf("Date of compilation is %s \n", __DATE__);
```

`__DATE__` 的示例值可能为:

"Dec 22 1998"

由该日期, 上例可能输出如下字符串:

Date of compilation is Dec 22 1998.

__FILE__**当前源文件**

宏 `__FILE__` 被转换为由引号括起的包含当前源文件名的字符串。该名称可

由 #line 指令设置。例如,考虑下列代码:

```
#include <stdio.h>
//...
printf("Error in file %s. \n", __FILE__);
```

如果源文件为 c:\samples\tesdt.cpp,在预编译过程中, __FILE__ 宏就被转换成下述字符:

```
"c: \ \ samples \ \ test.cpp"
```

上例输出如下字符串:

```
Error in file c: \ samples \ test.cpp.
```

__LINE__

当前源代码行号

宏 __LINE__ 被转换为正在被编译的源文件的当前程序行的行号。此数字可由 #line 指令来重置。例如,考虑如下源代码:

```
#include <stdio.h>
//...
printf("Error occurred at line %d \n", __LINE__);
```

若当前行是 17,在编译时,宏 __LINE__ 就被转换成以下字符:

```
17
```

上例输出如下字符串:

```
Error occurred at line 17.
```

__TIME__

当前时间

宏 __TIME__ 被转换成由引号括起的包含当前编译时间的字符串,以“时:分:秒”的形式出现。例如,考虑如下的源代码:

```
#include <stdio.h>
//...
printf("TIME of compilation is %s \n", __TIME__);
```


__TIME__ 示例的可能值为:

"10:05:30"

由该时间,上例可能输出如下字符串:

Time of compilation is 10:05:30.

预处理器运算符

有一些特别的运算符仅被预处理阶段识别和处理:

运算符语法	描述
defined(symbol)	如果 symbol 被定义则返回 1, 否则返回 0。
# str	将一字符串 str 放入被引号括起的字符串。
text1 # # text2	将 text1 和 text2 连接起来。

后两个运算符, # 和 # #, 被用于 #define 指令的替换部分。# 运算符将它的参数用引号括起:

```
#define pr(s) puts( #s)
```

由该指令, 预处理器将以下语句:

```
pr(Hi there);
```

替换为:

```
puts("Hi there");
```

运算符将两段文字拼合在一起。例如, 假设有下列宏定义:

```
#define call(verb,adj,do) verb# #-# #adj(do)
```

由该指令, 预处理器将以下语句:

```
call(shrink,all,trees);
```

替换为:

```
shrink-all(trees);
```

大多数程序中用不到 # 和 # # 运算符。然而, 它们使你能够在预处理阶段增多可控制的文字。若你想控制文字的输入与输出, 这就很方便了。

下面有一个可能会更实用的示例。假设你的函数库中含有一个函数的多个版本,例如:

```
strcpy  
wstrcpy
```

问题是这样的:在写入一些宏时,你可能需要根据编译的情况决定来选择一个不同的函数。解决该问题的最有效的方法是通过控制性文字来建立函数名称。这就是##运算符的可用之处。下面是一个如何用宏来建立函数名的方法:

```
#define STRINGF PREFIX##str##OP
```

由以下定义:

```
#define PREFIX w  
#define OP cpy
```

宏 STRINGF 将被转化成如下函数名:

```
wstrcpy
```

通过该方法,使程序的另外一行来决定是否在调用宏 STRINGF 的代码行中使用“w”版本(相应于长型字符格式)的函数。由于 PREFIX 的值可由编译器的命令行设置,所以在切换格式时,你就不需改写代码行。

相对于其他的预处理器元素来说,##是很少使用的,故若你在程序中很少用到它也无须担心。

第十五章

库函数

C 和 C++ 的 ANSI 定义包含了大量的标准库函数。你可以用这些函数来完成文件的读写操作、控制台的读写操作、获取日期的时间及执行高级的数学计算。

虽然从技术上来说,函数库与语言本身并不一样,但编译器的供应商总会提供标准函数库。事实上,许多供应商提供了比这里介绍的函数库更大的库;附加的函数可能支持特定的数据类型或是提供了特定平台的函数。然而,如果你想编写简洁的源代码,就该尽可能地坚持使用本章所介绍的函数。

C++ 的标准函数库与 C 的相同。标准的 C++ 库中有许多 C 中所没有的特性,但是它们均是在类中实现的。第十六章将介绍这些类库。

库函数简介

虽然标准的函数库是相当大的,但它可以划分为几大功能组:

- **标准 I/O 函数。**通过包含 `stdio.h` 或虚拟头 `cstdio` 可获取这些函数的声明。此类函数包含了控制台 I/O 函数,如 `printf`、`scanf`、`gets`、`puts`、`getchar` 和 `putchar`。它还包含了所有的文件 I/O 函数:`fopen`,打开一个文件;`fprintf`、`fscanf`、`fputs`、`fgets` 等等,读写文本文件;`fread` 和 `fwrite`,读写二进制文件。另外还有许多其他函数来移动和获取文件的位置,如 `fseek`、`ftell` 和 `rewind`,它们对随机访问操作很有用。所有这些函数都包含于“Console”和“Files”主题下,其中,每一个函数都有自己的注释。
- **字符串控制函数。**通过包含 `string.h` 或虚拟头 `cstring` 可获取这些函数的声明。此类函数包含了对简单的 `char *` 串的控制:`strlen` 返回串长度,`strcpy` 将一个串复制到另一个串,`strcat` 连接两个字符串。你可以用 `strcmp` 来比较两个字符串是否相同。还有许多串操作函数,如 `strchr`,是用来查找串中的特定字符。`strtok` 函数是用来获取单个词或记号的(“标记”--一个串)。所有这些函数都包含于“str<op>”主题下。
- **字符测试函数。**通过包含 `cctype.h` 或虚拟头 `cctype` 可获取这些函数的声明。

这些函数可用来在特定条件下检测单个字符。例如,如果被测试的字符是一个字母, `isalpha` 就返回真。使用这些函数的好处是它们可构成简单的源代码;你不需要去参阅 ASCII 代码的特定数字值。此类中的其他函数包括: `isalnum`、`isctrl`、`isdigit`、`isgraph`、`islower`、`isprint`、`ispunct`、`isspace`、`isupper` 和 `isdigit`。所有这些函数都包含于“is<cond>”主题下。

- **数学函数。**通过包含 `math.h` 或虚拟头 `cmath` 可获取这些函数的声明。此类函数包含了标准的三角函数和双曲函数: `sin`、`cos`、`tan`、`asin`、`acos`、`atan`、`atan2`、`sinh`、`cosh` 和 `tanh`。其他数学函数有: `pow`, 幂次函数; `exp`, 指数函数; 自然对数函数 `log` 和常用对数函数 `log10`。混合数学函数包括: `ceil` (上舍入)、`floor` (下舍入)、`fabs`、`fmod`、`frexp`、`ldexp`、`modf` 和 `sqrt` (计算平方根)。每一个函数都有自己的主题。

注意,有一些数学函数,如 `abs` 和 `rand` 未在 `math.h` 内声明,而在 `stdlib.h` 内声明。这将在本清单的最后部分提及。

- **时间和日期函数。**通过包含 `time.h` 或虚拟头 `ctime` 可获取这些函数的声明。此类函数包括 `time` 函数,它以结构 `time_t` 的形式返回当前日期与时间。大多数此类中的其他函数都显示一个 `time_t` 值,或是将其转化为特定格式。它们包括 `asctime`、`ctime`、`difftime`、`gmtime`、`localtime`、`mktime` 和 `strftime`。(你可用 `mktime` 来创建一个新特定的日期/时间值。)此类函数还包括 `clock` 函数来决定程序的运行时间。所有这些函数都包含于“Time”主题下。
- **内存分配函数。**通过包含 `stdlib.h` 或虚拟头 `cstdlib` 可获取这些函数的声明。提供这些函数主要是为了与 C 语言后向兼容。它们包括 `malloc`、`free`、`calloc` 和 `realloc`。在 C++ 中,一般最好使用 `new` 和 `delete`。每一个函数都有自己的主题。
- **参量长度可变的函数。**通过包含 `stdarg.h` 可获取这些函数的声明。这些函数已经在 `stdarg.h` 中以宏的形式实现了,它们使你能够运用可变长度的参量。你可以利用它们来编写函数,如 `printf`,就有任意数量的参数。此类中的三个函数为 `va_start`、`va_arg` 和 `va_end`。所有这些函数都包含于“va<op>”主题下。
- **杂类函数。**有许多函数并不易于归结于某一类中。大多数此类函数的原型是在 `stdlib.h` 中声明的。(相应地,也可以包含虚拟头 `cstdlib`。)此类函数中含有一些面向数学的函数(`abs`、`labs` 和 `ldiv`),也含有生成随机数的函数 `srand` 和 `rand`;有与操作系统相交互的 `system` 和 `getenv`;有处理程序中止的 `abort`、`exit` 和 `atexit`。此类函数还包括类型转换函数 `atof`、`atoi`、`atol`、`strtod`、`strtoi`、`strol` 和 `strtoul`。`qsort` 和 `bsearch` 函数用于分类和查找任意类型的大数组。最后,还

有一些 C++ 中已不再采用而为了与 C 相兼容所提供的函数, 它们有 `setjmp`、`longjmp`、`raise` 和 `signal`。(C++ 由于有了异常处理就不再需要它们了。) 此类中的每一个函数都有自己的主题。

abort

异常终止一个进程

`abort` 函数异常终止一个程序, 跳过所有的收尾清理进程。文件缓冲区没有得到清理; 因此, 任何等待发送的输出都将丢失。在大多数情况下, 应该调用 `exit` 函数作为替代来终止程序。`abort` 函数声明如下:

```
#include <stdlib.h>    //或 #include <cstdlib>
void abort(void)
```

当该函数被调用时, 程序向操作系统返回一个出错信息。`abort` 意味着比执行其他任意的 `exit` 函数都更彻底的终止。

```
abort();
```

abs

整型数的绝对值

`abs` 函数返回一个整型数的绝对值。该函数有以下声明:

```
#include <stdlib.h>    //或 #include <cstdlib>
int abs(int num);
```

负数的绝对值即为舍去负号后的值, 正数和零的绝对值返回原来的值。例如, 下列代码输出数字 5:

```
#include <stdio.h>
#include <iostream.h>
//...
cout << abs(-5);
```

相关的函数有 `labs`, 返回长整数的绝对值; `fabs`, 返回浮点数的绝对值。

acos

反余弦值

`acos` 函数返回参数的反余弦值。该函数是余弦函数的反函数。`acos` 有以下声

明:

```
#include <math.h>    // 或 #include <cmath>
double acos(double x);
```

该函数的参数为 $-1 \sim 1$ 之间的数, 返回一个以弧度表示的角度值。例如, 下列代码输出 0 的反余弦值:

```
#include <math.h>
#include <iostream.h>
//...
cout << "acos(0) = " << acos(0) << " \n";
```

参见 `asin`、`atan`、`atan2`、`cos`、`sin` 和 `tan`。

asin

反正弦值

`asin` 函数返回参数的反正弦值。该函数是正弦函数的反函数。`asin` 有以下声明:

```
#include <math.h>    // 或 #include <cmath>
double asin(double x);
```

该函数的参数取 $-1 \sim 1$ 之间的数, 返回一个以弧度表示的角度值。例如, 下列代码输出 1 的反正弦值:

```
#include <math.h>
#include <iostream.h>
//...
cout << "asin(1) = " << asin(1) << " \n";
```

参见 `acos`、`atan`、`atan2`、`cos`、`sin` 和 `tan`。

assert

测试条件

如果指定的条件为假(0), `assert` 宏函数立即终止程序运行。该函数有以下声明:

```
#include <assert.h>    //或 #include <cassert>
void assert(int exp);
```

使用 `assert` 函数的目的是为了保证程序的基本执行条件为真。如果 `exp` 为假 (0), 该函数终止程序并以如下形式打印出一条错误信息:

Assertion failed: *expression*, file *fname*, line *num*

例如, 如果 `n` 超过了 100, 下列源代码就终止程序:

```
#include <assert.h>
//...
assert(n <= 100);
```

如果 `n` 大于 100, 程序终止并输出如下的信息:

Assertion failed: `n <= 100`, file `Test.cpp`, line 17

当定义了宏 `NDEBUG` 之后, 每一条 `assert` 都将被忽略。对于最终程序的非调试版本, 可定义 `NDEBUG`。

atan

反正切值

`atan` 函数返回参数的反正切值。该函数是正切函数的反函数。`atan` 有以下声明:

```
#include <math.h>    // 或 #include <cmath>
double atan(double x);
```

该函数的参数取两边之比并以弧度形式返回一个角度值。例如, 下列代码输出 5.2 的反正余值:

```
#include <math.h>
#include <iostream.h>
//...
cout << "atan(5.2) = " << atan(5.2) << " \n";
```

参见 `acos`、`asin`、`atan2`、`cos`、`sin` 和 `tan`。

● 注意

`atan` 函数的一个最佳应用是在双精度域下获得 `pi` 的最近似值:

```
double pi = atan(1) * 4;
```


atan2**y /x 的反正切值**

atan2 函数返回参数的反正切值。该函数是正切函数的反函数。该函数的参数和 atan 函数都是取两边之比作为参数。区别在于 atan2 函数更明确地以 y /x 形式表示了这一关系。atan2 有以下声明：

```
#include <math.h>      // 或 #include <cmath>
```

```
double atan2(double y, double x);
```

该函数以弧度形式返回一个角度值。例如，下列代码输出 108.3 /42 的反正切值：

```
#include <math.h>
#include <iostream.h>
//...
cout << atan2(108.3,42.0) << '\n';
```

参见 acos、asin、atan、cos、sin 和 tan。

atexit**注册终止函数**

atexit 函数注册一个函数作为终止函数。终止函数是很有用的，因为即使程序有多个出口，它都能使你可在一个模块内处理所有的收尾步骤。atexit 函数有以下的声明：

```
#include <stdlib.h>      // 或 #include <cstdlib>
void atexit(void (* func)(void));
```

atexit 函数的参数是一个指向终止函数的指针。该终止函数必须是 void 类型且无参数。例如，下列源代码注册了一个输出结束信息的函数为终止出口函数：

```
#include <stdlib.h>

void term1(void) {
    cout << "This is the term1 function. \n";
    cout << "You terminated successfully. \n";
}
//...
atexit( &term1);
```

如果程序是通过调用了 `abort` 来终止程序, `term1` 就不被执行了。如果程序是以其他方式终止的(甚至通过调用 `exit(EXIT_FAILURE)`), `term1` 就将被执行。最多可以注册 32 个函数。它们是按后进先出的顺序执行的(即最后注册的函数最先执行)。

atof

把字符串转换成浮点数

`atof` 函数以一个含整数或浮点数的字符串作为参数并返回一个数字。该函数有如下声明:

```
#include <stdlib.h>    // 或 #include <cstdlib>
double atof(const char *str);
```

该函数跳过起始的空格符并读入字符,只要这些字符构成合法的浮点表达式。随后的字符就可忽略。例如,“-24.56xy7”被转换成 -24.56。该函数读取“E”和“e”作为指数符号,故“5E2”被转换成 50.0。如果没有读入合法的数字,大多情况下就返回 0。

下例从键盘获取一个数字串并将数字存储于变量 `x`。例子假设了最大行长为 80。

```
#include <stdlib.h>
#include <stdio.h>
//...
char input_str[81];
double x = atof(gets(input_str));
```

相关函数有 `atoi` 和 `atol`,它们返回整数。

atoi

把字符串转换成整数

`atoi` 函数将一个数字串转换成数字值。该函数有以下的声明:

```
#include <stdlib.h>    // 或 #include <cstdlib>
int atoi(const char *str);
```

该函数返回一个整数。它跳过起始的空格符并在第一个非数字的字符处中止读入。例如,串“23.9x”以数字 23 读入。如果没有合法的数字可读入,大多情况下就返回 0。

下例从键盘获取一个数字串并将数字存储于变量 `x`。例子假设了最大行长为 80。

```
#include <stdlib.h>
#include <stdio.h>
//...
char input_str[81];
double x = atoi(gets(input_str));
```

相关函数有 `atol`, 返回一个长整数; `atof`, 返回一个浮点数。同样地, `strtol` 和 `strtoul` 以任意长度读取整数。

atoi

把一字符串转换成长整型

`atoi` 函数将一个数字串转换成数字值。它类似于 `atoi`, 但具有更长的位数。该函数有以下的声明:

```
#include <stdlib.h> // 或 #include <cstdlib>
```

```
int atoi(const char * str);
```

它跳过起始的空格符并在第一个非数字的字符处中止读入。例如,串“23.9x”以数字 23 读入。如果没有合法的数字可读入,大多情况下就返回 0。

下例从键盘获取一个数字串并将数字存储于变量 `x`。例子假设了最大行长为 80。

```
#include <stdlib.h>
#include <stdio.h>
//...
char input_str[81];
long n = atol(gets(input_str));
```

相关函数有 `atoi`, 返回一个整数; `atof`, 返回一个浮点数。同样地, `strol` 和 `stroul` 以任意基数读取整数。

bsearch**二分法搜索**

bsearch 函数采用二分法对一个任意大的数组进行搜索并返回一个指向目标项的指针。此数组必须按规则排列(你可通过调用 qsort 函数来实现)。当调用 bsearch 函数时,需指定一个回调函数的地址;该函数是你自己编写的,bsearch 用它来比较两个参数。bsearch 函数有以下声明:

```
#include <stdlib.h> // 或 #include <csdlib>

void *bsearch(const void *key, const void *buf,
              size_t num, size_t size,
              int (compare)(const void *, const void *));
```

三个参数 buf、num 和 size 描述了将被搜索的数组。buf 指向数组的起始,num 给出了元素的个数,而 size 指定了每一元素的字节数。key 参数指向了目标值的拷贝。

compare 参数是一个函数的地址,该函数根据所指的第一个参量值是小于、等于、或大于所指的第二个参量值来返回 -1、0、或 1。该函数的声明中有两个 * void 参量,实际上是指向被比较元素的指针。你必须将指针的类型强制转换成函数实际接收到的数据类型,然后根据指针获得数据。

例如,下例代码对一整型数组进行排序查找。cmp 函数获取 int * 型的指针,指定了函数操作所要求的正确类型并获得数据:

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

int cmp(const void *p1, const void *p2) {
    int i = *(int *)p1; // 获取整数指针。
    int j = *(int *)p2;
    return i < j ? -1 : (i == j ? 0 : 1);
}

void main() {
    int dat[] = {1, 4, 9, 34, 39, 40, 45, 50, 51, 99};
    int i = 45;
    void *p = bsearch(&i, dat, 10, sizeof(int), &cmp);
    if (p == NULL)
        cout << "i was not found.";
```

```
else  
    cout << "i was found.";
```

如果函数找到目标项的话,就返回一个指向它的指针。参阅本章后面介绍的 `qsort` 函数,可获得一些关于字符串数组的操作。

calloc

分配内存单元

`calloc` 函数可实现几乎与 `malloc` 函数一样功能:分配一块内存并返回该内存的地址(或是 `NULL`,当出现内存不够的情况时)。`calloc` 函数的主要优点在于它保证了每一内存单元初始化为 0。该函数有以下声明:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
void * calloc(size_t num, size_t size);
```

内存按 `size` 所指定的大小进行分配,空间大小为 `num * size` 个字节。

例如,下例源代码分配了足够存放 `n` 个整数的内存块;每一个整数都被初始化为 0:

```
#include <stdlib.h>
```

```
int n;
```

```
int * p;
```

```
//...
```

```
p = reinterpret_cast<int * >(calloc(n, sizeof(int)));
```

```
if (p == NULL)
```

```
    // 打印错误信息并结束。
```

具体详情,请参阅 `free`、`malloc` 和 `realloc`。

ceil

上舍入

`ceil` 函数取一浮点数为参数,并返回一个不小于该参数的最小整数量。该函数有以下声明:

```
#include <math.h>    // 或 #include <cmath>
```

```
double ceil (double x);
```

虽然返回的是一个整数量——它没有小数部分——但返回值仍然是一个双精度类型,所以你或许需要在返回该值之前将其类型进行一下强制转换。例如,以下函数返回一个范围在 1 到 n 之间的整型数:

```
#include <math.h>
#include <stdlib.h>

int random_ItoN(int n) {
    return (int) ceil(n * rand() / RAND_MAX);
}
```

参见 floor 和 modf 函数。

Console

控制台 I/O 函数

标准 C /C++ 函数库包含了许多控制台(显示器和键盘)读写函数。这些函数以简单的字符流方式来读写数据。为了使用这些函数,须首先包含 stdio.h 文件:

```
#include <stdio.h>    // 或
#include <cstdio>
```

关于 stdio.h 中定义的其他函数的总结,请参阅“文件 I/O 函数”。

总结

控制台 I/O 函数包括下列基本函数:

函数	操作
getchar()	从键盘中读字符,返回该字符对应的 int 型值。
gets(s)	从键盘输入流中读入一个以换行符终结的字符串,将其存在串 s 中。返回串参数 s。
perror(s)	以形式“s: error message”输出一个串,其中的错误信息(error message)由全局变量 error 来决定。如果 s 为 NULL,仅输出错误信息。perror 打印错误信息到 stderr 流(通常为显示器)。

<code>printf(fmt, ...)</code>	向标准输出流写格式化串,并返回所写的字符个数。详情请参阅主题 <code>printf</code> 。
<code>putchar(ch)</code>	向标准输出流写字符 <code>ch</code> 。若调用成功,返回所写的字符;否则返回 EOF。
<code>puts(s)</code>	向标准输出流写一个以换行符终结的字符串 <code>s</code> ,调用成功,返回非负值,反之返回 EOF。
<code>scanf(fmt, ...)</code>	用格式化串 <code>fmt</code> 从键盘中读取数据;返回输入的数据个数。详情请参阅主题 <code>scanf</code> 。

从键盘中读取字符

通过调用 `getchar` 函数可以从键盘中读入一个字符。操作系统通常一次缓存一个输入行。这意味着 `getchar` 在用户键入回车前并不返回任何输入值。

```
int c;  
c = getchar();    // Get next character when available.
```

为了达到响应每一次击键而又无须等待输入回车,你就需要捕获每次击键操作。标准的 C /C++ 并不支持击键捕获。请参阅你的编译器及操作系统文档来获得帮助。

`gets` 函数从键盘输入流中读入一个以换行符终结的字符串,将其存在串 `s` 中。但你需要首先创建一个足够大的字符串来容纳所有可能的输入行。

```
char s[81];  
gets(s);
```

应该注意的是,`gets` 并不将换行字符复制到串数据中。

`gets` 函数的一个替代是调用 `fgets` 函数并指定 `stdin`(标准输入流)作为输入流。不同于 `gets` 的是,`fgets` 对复制到串的字符格式有限制。在随后的“控制台 I/O 与文件 I/O 的对比”部分有一个示例。

从键盘中读取数字

`scanf` 函数提供了一个从键盘读入数字的通用方法。(详情请参阅主题 `scanf`。)

```
int i;
```

```
scanf("%d", &i);
```

虽然 `scanf` 函数很适用于简单的程序,但它并不灵活。一旦调用了 `scanf`,要直到用户输入了一个数字并按下回车后它才返回。如果用户在没有输入数字的情况下按下回车,`scanf` 就会一直等待输入。

为了获得对程序的更多控制,你可能想直接读入输入行,然后用类似于 `atoi`、`atol`、`atof` 和 `strtod` 函数来把所输入的字符转换成数字。例如:

```
#include <stdio.h>
#include <stdlib.h>
//...
char s[81];
gets(s);
int i = atoi(s);
```

由以上代码,如果用户在未输入任何文字的情况下按下回车键,则变量 `i` 获得缺省值 0。你可以用一个不同的缺省值来测试 `i` 是否为空。

向屏幕打印字符

`puts` 函数打印一个字符节并自动加上回车 /换行符。

```
puts(s);    // Print a string followed by newline.
```

如果你想向屏幕输出一个不含回车的串,可用 `fputs` 或 `printf` 函数。以下两行代码都有这种功能:

```
fputs(s, stdout);
printf(s);
```

`printf` 函数也可用来按指定格式输出数字。详情请参阅主题 `printf`。

在某些系统中,直到输入了回车符,所输入的内容才被回显到屏幕上。为了使数据立即回显,可调用 `fflush` 函数:

```
printf("Enter your number here = >");
fflush(stdout);
```

控制台 I/O 与文件 I/O 的对比

控制台函数与文件 I/O 函数并不相同。例如,`puts` 函数自动加上回车 /换行符,而 `fputs` 就无此操作。以下代码看上去像是在输出一个文本文件的内容,其实它

在文本的每一行之后添加了一个空白行：

```
#include <stdio.h>
#define LINESZ 256
int main() {
    FILE * inf;
    char s[LINESZ];

    inf = fopen("c: \ \ test.txt", "r");
    if (inf == NULL) {
        puts("Could not open input file.");
        return 1;
    }
    fgets(s, LINESZ, inf);
    while (! feof(inf)) {
        puts(s);
        fgets(s, LINESZ, inf);
    }
    return 0;
}
```

上例中的关键部分是：

```
while (! feof(inf)) {
    puts(s);
    fgets(s, LINESZ, inf);
}
```

问题在于 fgets 在每一串的末尾读入了一个新行，puts 又加上了另外一个新行。为了解决这个问题，可调用 fputs 并指定 stdout。这就可以达到输出到屏幕但并不添加额外的新行的效果：

```
while (! feof(inf)) {
    fputs(s, stdout);
    fgets(s, LINESZ, inf);
}
```

文件 stdio.h 定义了几个标准的 I/O 流：

函数	描述
stdin	标准输入。缺省情况下是键盘。
stdout	标准输出。缺省情况下是显示器。
stderr	标准错误。缺省情况下是显示器。

在上表中, stdout 和 stderr 看上去一样,但它们是相互独立的。即使标准输出被重定向到一个文件上(在 Unix 或 DOS 操作系统中不能这样做), stderr 仍为向显示器输出。

cos**余弦函数**

cos 函数返回它的参数的余弦值。cos 有以下声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double cos(double x);
```

该函数以弧度为单位读取一个角并返回一个介于 1 和-1 之间的值。例如,以下代码输出 $\pi/2$ 的余弦值。输出结果或许并不完全精确,但应该非常逼近于零:

```
#include <math.h>
#include <iostream.h>
//...
double pi = atan(1) * 4;
cout << "cos(pi /2) = " << cos(pi /2) << '\n';
```

参见 acos、asin、atan、atan2、sin 和 tan。

cosh**双曲余弦函数**

cosh 函数返回它的参数的双曲余弦值,参数以弧度为单位。cosh 有以下声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double cosh(double x);
```

如果计算结果超出了范围,函数返回 HUGEVAL 并将全局变量 `errno` 设为 ERANGE。参见 `sinh` 和 `tanh`。

div**整数相除**

`div` 函数将两个整数相除,返回一个包含商和余数的结构。该函数有以下声明:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
div_t div(int numerator, int denominator);
```

结果以结构 `div_t` 的形式返回,`div_t` 有以下格式,商(`quot`)表示一个乘以除数(`denominator`)后的结果不超过被除数(`numerator`)的最大整数:

```
struct div_t {  
    int quot;  
    int rem;  
};
```

相关的函数有 `ldiv`,它执行同样的操作但对象是长整型数。求余运算符(`%`)提供了获得余数的另一方法。

exit**终止程序**

`exit` 函数正常终止程序。缓冲输出内容被写完,所有终止进程得以执行(参见 `atexit`)。该函数有以下声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
void exit(int exit_code);
```

文件 `stdio.h` 至少给出口状态(`exit_code`)定义了两个值:`EXIT_SUCCESS(0)`和 `EXIT_FAILURE(非 0)`。C/C++ 的实现可以定义其他值。

下例终止了程序并表示程序运行成功:

```
exit(0);
```

exp**计算 e 的 x 次方**

exp 函数计算 e 的指定幂次方并返回结果。(对于计算任意数的指定幂次方,请参阅 pow 函数。)该函数有以下声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double exp(double x);
```

exp 函数的一个最通常应用是直接获得 e 的值:

```
#include <math.h>
//...
double e = exp(1);
```

fabs**浮点数的绝对值**

fabs 函数返回一个浮点数的绝对值。该函数有以下声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double fabs(double x);
```

负数的绝对值即为丢去负号后的值,正数和零的绝对值返回原来的值。例如,下列代码输出数字 5.07:

```
#include <stdio.h>
#include <iostream.h>
//...
cout << abs(-5.07);
```

相关的函数有 labs, 返回长整数的绝对值, abs, 返回整型数的绝对值。

Files**文件 I/O 函数**

标准 C /C++ 函数库提供了对文件进行读写操作的函数。如要使用这些函数,首先须包含文件 stdio.h。stdio.h 中声明的其他函数,请参阅“文件系统函数”和“控

制台 I/O 函数”:

```
#include <stdio.h>    // 或
#include <cstdio>
```

总结

一些函数处理打开和关闭文件;下表对它们进行了总结。参数 *f* 是类型为 `FILE *` 的指向文件流的指针,而 *fname* 是一个含文件名的字符串。

函数	操作
<code>fclose(f)</code>	关闭文件流 <i>f</i> 。
<code>fopen(fname, m)</code>	用模式 <i>m</i> 打开文件 <i>fname</i> 。操作成功,返回一个文件流指针;反之返回 <code>NULL</code> 。详情参阅主题 <code>fopen</code> 。
<code>freopen(fname, m, f)</code>	实现与 <code>fopen</code> 一样的功能,但它首先关闭文件流 <i>f</i> 。
<code>tmpfile()</code>	以二进制读写方式打开一个临时文件并返回一个文件流指针。详情参阅主题 <code>tmpfile</code> 。

一旦你获得了一个文件流指针,就可把它作为文件 I/O 函数的 *f* 参数。以下函数执行输出功能,它们都返回整型值:

函数	操作
<code>fflush(f)</code>	刷新输出流 <i>f</i> 并立即将输出发送到物理设备。调用成功时,返回 0;发生错误时,返回 <code>EOF</code> 。
<code>fprintf(f, fmt, ...)</code>	将格式化串输出到流 <i>f</i> 。返回所输出的字节数。有关格式的详情请参阅主题 <code>printf</code> 。
<code>fputc(ch, f)</code>	将字符 <i>ch</i> 输出到流 <i>f</i> 。调用成功时,返回所输出的字符;发生错误时,返回 <code>EOF</code> 。
<code>fputs(s, f)</code>	将字符串 <i>s</i> 输出到流 <i>f</i> ,但不包括串终止符 <code>null</code> 。调用成功时,返回非负值;发生错误时,返回 <code>EOF</code> 。
<code>fwrite(p, size, n, f)</code>	从由 <i>p</i> 所指向的 <code>char *</code> 缓存中,输出 <i>n</i> 个单元到流 <i>f</i> 。返回所输出的单元数。每一单元的长度定义为 <i>size</i> 。
<code>putc(ch, f)</code>	同 <code>fputc</code> 。

以下函数从一个输入文件中读取数据,它们都返回 `int` 或 `char *` 类型:

函数	操作
<code>fgetc(f)</code>	返回指定输入流 <code>f</code> 的下一个字符;如果字符不可读,返回 EOF。返回的字符被转换为整型值。
<code>fgets(s, n, f)</code>	从输入流 <code>f</code> 中读入 <code>n-1</code> 个字符到字符串中。当读到换行符或文件末尾时,函数停止读。在串 <code>s</code> 尾保留换行符。读入的最后一个字符后面附加一结束符。调用成功时,返回所指的字符串;出错时,返回 NULL。(本书中,文件未并不算是一个错误。)
<code>fread(p, size, n, f)</code>	从流 <code>f</code> 中输入 <code>n</code> 个单元到由 <code>p</code> 所指向的 <code>char *</code> 缓存。返回所输入的单元数。每一单元的长度定义为 <code>size</code> 。
<code>fscanf(f, fmt, ...)</code>	以格式化串 <code>fmt</code> 的格式从流 <code>f</code> 中输入。调用成功时,返回读取的单元数;出错时,返回 EOF。有关格式请参阅主题 <code>scanf</code> 。
<code>getc(f)</code>	同 <code>fgetc</code> 。
<code>ungetc(ch, f)</code>	把字符 <code>ch</code> 回退到指定的输入流 <code>f</code> 中,该流必须已经以读模式被打开。调用成功时,返回回退的字符;反之,返回 EOF。

有一些函数获取或设置文件的位置。在随机访问操作时,它们特别有用。`pos` 参数的类型为 `fpos_t`,它在文件 `stdio.h` 中定义;`fpos_t` 是一个用来记录每一个文件位置的足够大的整数或结构。

函数	操作
<code>feof(f)</code>	如果到达文件末返回非零值。
<code>fgetpos(f, * pos)</code>	返回当前文件位置指针并将指针的位置保存在 <code>pos</code> 所指向的位置。该值可用于 <code>fsetpos</code> 函数。返回类型为 <code>fpos_t</code> 。
<code>fseek(f, off, org)</code>	设置 <code>f</code> 所指文件的位置指针到一个新的位置,该位置与 <code>org</code> 给定的文件位置的距离为 <code>off</code> 字节。 <code>org</code> 的几种模式在下文中讨论。

<code>fsetpos(<i>f</i>, <i>pos</i>)</code>	将当前的文件位置指针设置到 <i>pos</i> 所指处。同 <code>fgetpos</code> 连用。
<code>ftell(<i>f</i>)</code>	以长整型返回文件 <i>f</i> 中的当前文件位置指针。

在 `fseek` 函数中,参数 `org` 的值取以下三者之一: `SEEK_SET`,定位于文件开始; `SEEK_CUR`,定位于当前文件指针位置; `SEEK_END`,定位于文件末尾。

其余的文件 I/O 函数处理错误和用户自定义的缓存区。参数 *p* 的类型为 `char *`。缺省情况下, C/C++ 文件流为自己提供缓存,大小为 `BUFSIZ`,以减少对磁盘的访问。

函数	操作
<code>clearerr(<i>f</i>)</code>	复位错误标志。
<code>ferror(<i>f</i>)</code>	返回一个包含当前错误号的整数;如果文件没有错误,此数为 0。
<code>rewind(<i>f</i>)</code>	复位错误标志并将文件的指针位置重置于文件的开始处。
<code>setbuf(<i>f</i>, <i>p</i>)</code>	使用由 <i>p</i> 指向的文件缓冲区,若 <i>p</i> 为 <code>NULL</code> ,在进行 I/O 时不进行缓冲。缓冲区的长度由 <code>BUFSIZ</code> 指定。
<code>setvbuf(<i>f</i>, <i>p</i>, <i>m</i>, <i>size</i>)</code>	除了由 <i>m</i> 指定了模式(见下文)和 <i>size</i> 指定了缓冲区的大小外,其余同 <code>setbuf</code> 。调用成功时,返回 0;反之,返回非零值。

在 `setvbuf` 函数中, *mode* 是下列之一的整型参量: `_IOFBF`,文件全部缓冲; `_IONBF`,文件不缓冲; `_IOLBF`,文件行缓冲,输出时,当把换行符写到文件中时,缓冲区被清除。

文本文件的读写

文件操作的起始点是调用函数 `fopen` 来打开一个文件流。如果调用该函数成功,就返回一个类型为 `FILE *` 的指针,你可把它作为其他函数的输入值。

```
#include <stdio.h>
//...
FILE * fp;
fp = fopen("C: \\ test.txt", "w");
```

此处,“w”是对文件进行操作的 mode(模式)参数;它指明了文件是以文本模式进行写操作。有关文件模式的完整列表,参见主题 fopen。从技术上来说,文本模式和二进制模式的区别很小。在文本模式下,文件中的每一回车 /换行都是作为单一的换行字符读入内存的,反过来,在输出时,它扩展为一个换行符。

文本和二进制的区别主要在于用法上。一个文本文件完全由文本字符组成,其中的数字也被表示成数字字符。为了保持这种格式,须使用 fprintf 函数,它是以文本串的形式写数字数据。详情参阅主题 printf。

```
#include <stdio.h>
//...
void test_files(void) {
    FILE * fp;
    int i = 12;
    fp = fopen("C: \ \ test.txt", "w");
    if (fp)
        fprintf(fp, "Here is a number: %d \n", i);
    else
        fputs("Error opening the file. \n", stderr);
}
```

当你对文本文件进行操作时,fgets 和 fputs 函数对同时读写一行文字很有作用。它俩互补地非常完美。fgets 函数以串的形式读入一行文字,在串尾保留换行符,并在最后一个字符后面附加一个空字符。当 fputs 输出该串时,就不再添加换行符或空字符了。

例如,以下程序从一个文件中读取数据发送到另一个文件,并在每一行前面添加大于符号(>)。程序中,有必要调用 feof 函数来检测文件结束标志,fgets 函数并不检测该标志。

```
#include <stdio.h>
#define LINESZ 256

void main(void) {
    FILE * inf, * outf;
    char s[LINESZ];
    inf = fopen("C: \ \ test1.txt", "r");
    outf = fopen("c: \ \ test1.txt", "w");
    if (inf == NULL || outf == NULL) {
        puts("Could not open files.");
        return 1;
    }
}
```



```

    }
    fgetc(s, LINESZ, inf);
    while (! feof(inf)) {
        fputc(">", outf);
        fputc(s, outf);
        fgetc(s, LINESZ, inf);
    }
}

```

二进制文件的读写

对二进制文件的操作通常包括了使用 `fread` 和 `fwrite` 函数。这些函数既可用于单字节操作,又可用于定长串的操作。

要对二进制文件进行操作,须把“b”添加在模式参数的后面,以此模式来打开一个文件。在二进制模式下,数据不需要进行转换。

```

#include <stdio.h>
//...
FILE * fb;
fb = fopen("c: \\ stuff.dat", "wb");

```

下例是一个挺有用的打印文件内容的十六进制值的输出程序,它可读入任意类型的文件的内容。(你也可用它来读取文本文件,但不会执行任何转换。)该程序根据输入的文件名将其内容向标准输出设备打印输出。

```

#include <stdio.h>
#include <ctype.h>
#define BYTES_PER_ROW 16

int main(void) {
    FILE * fb;
    char filespec[256], buf[BYTES_PER_ROW];
    int i, n;

    printf("Enter filespec: ");
    gets(filespec);
    fb = fopen(filespec, "rb");
    if (fb == NULL) {
        printf("Can't open file %s\n", filespec);
        return 1;
    }
    do {

```

```
n = fread(buf, 1, BYTES_PER_ROW, fb);
for (i = 0; i < n; i++)           // HEX DUMP
    printf("%2X ", buf[i]);
printf(" ");
for (i = 0; i < n; i++)           // ASCII DUMP
    if (isctrl(buf[i]))
        putchar('.');
    else
        putchar(buf[i]);
    putchar(' \n');
} while (n == BYTES_PER_ROW);
return 0;
}
```

上例中的关键行是：

```
n = fread(buf, 1, BYTES_PER_ROW, fb);
```

第二个参数指定了要读入的每一单元的大小。因为大小为 1，函数就按字节读入。第三个参数指定了要读入的单元的个数。此处被设为了常量 BYTES_PER_ROW，它在程序的开始处被定义为 16。在读取完字节之后，程序输出它们的十六进制和 ASCII 代码值。

●——注意

在指定特定数据类型的大小时，运算符所占的字节长度也起作用。

如果程序读入的内容少于 16 字节，它就打印输出所读取的那些字节并结束。记住，fread 函数返回实际读取的单元数。如果读取的比要求的单元数要少，就说明已经到达文件末了。

```
do {
    n = fread(buf, 1, BYTES_PER_ROW, fb);
    //...
} while (n == BYTES_PER_ROW);
```

fwrite 函数以类似的方式工作，也取同样的参数。fwrite 按指定元素个数进行写操作。如果 size 参数为 1，fwrite 就按字节输出数据。

```
n = fwrite(buf, 1, BYTES_PER_ROW, fb);
```

有关 gets、puts、printf 和 putchar 函数的内容，参阅“控制台 I/O 函数”。

● 注意

通过上面一个例程, 可以看到一些文本模式与二进制模式的区别。首先将该程序运行于文本文件时, 可注意到带有回车 / 换行符(十六进制 0D 0A)。然后把 `fopen` 调用中的“rb”改写为“r”(文本只读模式), 再运行程序。你将发现文件中的每一个回车 / 换行对变成了一个字符(0A)。

floor**下舍入**

`floor` 函数求得不大于一个浮点数的最大整数。该函数有以下声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double floor(double x);
```

即使被返回的是一个整型量(它不含小数部分), 但仍是双精度类型, 所以你也许要在结果返回之前将其强制转换。例如, 以下函数返回一个范围从 0 到 `n-1` 的整型量:

```
#include <math.h>
#include <stdlib.h>
int random_0toN_1(int n) {
    return (int) floor (n * rand() / RAND_MAX);
}
```

参见 `ceil` 和 `modf` 函数。

fmod**除法求余**

`fmod` 函数将一个浮点数去除另一个浮点数, 并返回余数。也就是说, 该函数用最大可能的整数 `i` 去乘 `y` 而不超过 `x`, 然后返回 `x - yi`。该函数的功能类似于求余运算符(`%`), 只不过该函数作用于浮点数并返回一个浮点数结果。该函数有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double fmod(double x, double y);
```

例如, 以下语句输出右侧所注释的结果:

```
#include <math.h>
#include <iostream.h>
//...
cout << fmod(31, 2) << '\n';           // 输出 1。
cout << fmod(31.05, 2) << '\n';        // 输出 1.05。
cout << fmod(7.6, 2.5) << '\n';        // 输出 0.1。
```

fopen

为执行读写操作而 打开文件

fopen 函数打开一个文件,并返回一个与文件流相关联的指针。该指针可被用作其他文件 I/O 函数的输入参数。(参阅“文件 I/O 函数”。)该函数有以下声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
FILE *fopen(const char *fname, const char *mode)
```

参数 fname 包含一个文件名,其间也可指定文件的路径。如果使用了文字串来描述文件的路径,就应在串中使用双反斜杠(\\)来替代单反斜杠(\\)。例如,假设你想打开以下文件:

```
c:\programs\drawing.dat
```

为了以二进制读方式打开此文件,就应输入以下语句:

```
FILE *fp;
fp = fopen("c:\\programs\\drawing.dat", "rb");
```

模式参数

参数 mode 是含下列值之一的字符串,见表 15-1:

表 15-1 fopen 所用到的字符串 mode

模式	操作
"r"	打开文本文件用于只读。
"w"	创建新的文本文件用于写。
"a"	打开文本文件用于添加。
"rb"	打开二进制文件用于读。

"wb"	创建新的二进制文件用于写。
"ab"	打开二进制文件用于添加。
"r+"	打开已存在的文本文件用于更新(读或写)。
"w+"	创建新的文本文件用于更新。
"a+"	打开文本文件用于更新。
"rb+"	打开二进制文件用于更新。
"wb+"	创建新的二进制文件用于更新。
"ab+"	打开二进制文件用于更新。

从技术上来说,文本文件和二进制文件的仅有区别在于对新起一行的处理上。在文本模式下,文件输出函数将每一新的换行(十六进制的 0A)转换为回车 /换行对(十六进制的 0D 0A);文件输入函数又将回车 /换行对转换成单个换行符读入。

因此,即使在文件中的文本行的末尾以双字符表示(0D 0A),程序中也用一个简洁的字符来表示一个文本行的末尾——在 C /C++ 中为串“\n”。

当然,在实际使用中,文本文件和二进制文件有其他一些差别。在文本文件中,所有数据通常是以可读格式——ASCII 代码来表示的,所以数字是用数字字符串来表达的,而非直接用二进制值存储。(所以常用 fprintf 函数来向一个文本文件写数据而不向二进制文件写数据。)

示例

以下示例为通过控制台获取一个文件名并以读文本模式打开该文件。NULL 返回值表明文件打开失败。

```
#include <stdio.h>
#include <stdlib.h>
//...
FILE * fp;
char fspec[81];
printf("Enter a file specification: \n");
gets(fspec);
fp = fopen(fspec, "r");

if (fp == NULL) {
    printf("Could not open file %s\n", fspec);
    exit(1);
}
```

```
}  
以 fp 来执行文件操作...
```

其他规则

在不同模式下使用 `fopen` 函数还有一些其他规则:

- 如果函数不能成功打开一个文件,则返回 `NULL`。
- 当你打开一个文件用于读或添加时,该文件必须已经存在,否则调用 `fopen` 就会失败。
- 当你打开一个文件用于写时,要求参数 `fname` 必须含有一个合法的文件名。如果已存在具有该文件名的文件,则文件被重写。
- 当你打开一个文件用于更新(读/写模式)时,如果已存在具有该文件名的文件,则文件被打开;反之,创建一个新文件。
- 如果打开一个文件用于读/写模式,若没有调用以下之一的函数: `fflush`、`fseek`、`fsetpos` 或 `rewind`,就不能同时执行输入操作和输出操作。(此类函数的信息,参阅“文件 I/O 函数”。)

free

释放已分配的内存

`free` 函数释放由 `malloc`、`calloc` 或 `realloc` 所分配的内存。该函数仅取用一个指针作为参数;如果指针并不指向以上函数所分配的内存块,则调用函数出错。`new` 和 `delete` 函数实现了与 `malloc` 和 `free` 函数同样的功能,不过也提供了其他方便之处。`malloc` 和 `free` 函数向后兼容。`free` 函数有以下的声明格式:

```
#include <stdlib.h>      // 或 #include <cstdlib>  
void free(void * pointer);
```

例如,以下代码分配了 `n` 个字符的内存块,然后又释放了它。在使用内存之前,有必要把 `malloc` 所返回的指针值进行类型转换:

```
#include <stdlib.h>  
int n;  
char * p;  
//...  
p = reinterpret_cast<char * >(malloc(n));
```

```
//...  
free(p);
```

虽然在调用 `malloc` 时,要把指针的类型转换为 `char *`,但在调用 `free` 时就不用转换了。这是因为指针被传递给 `void *` 变量。参见函数 `malloc`、`calloc` 和 `realloc`。

frexp

把一个数分解为尾数和指数

`frexp` (“free exponent”) 函数把一个浮点数分解为尾数和指数部分。`frexp` 对分析一个浮点数的构成有用; 否则的话, 就无多大用途。该函数有以下的声明格式:

```
#include <math.h>      // 或 #include <cmath>  
  
double frexp(double x, int *exp_ptr);
```

该函数返回尾数并将指数拷贝到参数 `exp_ptr` 所指向的地址。由于基底是二进制的, 所以尾数总是小于 1.0, 且指数表示了 2 的幂次。例如, 以下代码将数 48 分解为尾数 0.75 和指数 6 ($48 = 0.75 * 2$ 的 6 次方):

```
#include <math.h>  
#include <iostream.h>  
//...  
int exp;  
double man = frexp(48, &exp);  
cout << "man = " << man << endl;  
cout << "exp = " << exp << endl;
```

getc

从文件流中取字符

`getc` 函数从指定的流中取下一个字符。如果没有下一个字符, 函数返回 EOF。`getc` 函数有以下声明格式:

```
#include <stdio.h>      // 或 #include <cstdio>  
  
int getc(FILE * stream);
```

`getc` 函数与 `fgetc` 函数完全相同。之所以有此重复是因为 `getc` 通常是以宏来实现的, 提供了比 `fgetc` 较好的性能, 故 `getc` 比 `fgetc` 常用。有关文件流操作的更多内

容,参见本章中的主题“File”。

getchar

从控制台读字符

getchar 函数从指定的标准输入流——通常为键盘中读入下一个字符。系统往往缓冲键盘输入,故直到用户键入了回车之后,程序才取得输入。getchar 函数有以下声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
int getchar(void);
```

有关控制台 I/O 函数的更多内容,参见本章中的主题“Console”。

gets

从控制台读取一字符串

gets 函数返回从标准输入中读取到的串,一直读到换行符或 EOF 为止。该函数将输入值置于它的串参数中,并返回同样的串。前提是你必须定义一个足够大的串来容纳输入的内容;从这一点来考虑,fgets 函数较为安全,因为它使你能够指定被读入串的字符的最多个数。与 fgets 相比,gets 本身并不向串中拷贝换行符。gets 有以下声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
char * gets(char * str);
```

有关控制台 I/O 函数的更多内容,参见本章中的主题“Console”。

getenv

读取环境变量的当前值

getenv 函数返回一指定的环境变量值。可供选用的环境变量及其意义是由程序实现来定义的。该函数由以下声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
char * getenv(const char * str);
```

例如,以下代码打印输出了环境变量 PATH 的设置值:


```
#include <stdlib.h>
#include <iostream.h>
//...
cout << "PATH = " << getenv("PATH") << " \n";
```

is<cond>**字符检测函数**

标准的 C /C++ 函数库中包含了一系列检测不同字符的函数。使用这些函数前,应首先包含文件 `ctype.h`。此文件也声明了 `toupper` 和 `tolower` 函数。(见 `tolower` 和 `toupper`。)

```
#include <ctype.h> // 或 #include <octype>
```

语法

下表总结了这些函数。每一个函数以单个字符作为参数,它们以整型数出现。每一个函数的返回值为布尔型,故返回真(1)或假(0)。

函数	操作
<code>isalnum(<i>ch</i>)</code>	字符是字母还是数字。
<code>isalpha(<i>ch</i>)</code>	字符是字母吗?
<code>iscntrl(<i>ch</i>)</code>	字符是控制符(如 <code>backspace</code> , <code>Enter</code> , <code>DEL</code> 或 <code>tab</code>)吗? 这里所说的控制符是执行一些功能的非打印字符。大多数在范围 <code>0x</code> 到 <code>0x1F</code> 之间。
<code>isdigit(<i>ch</i>)</code>	字符是数字吗?
<code>isgraph(<i>ch</i>)</code>	字符是可见的吗? 即是不是一个可打印字符,但不包括空格符。
<code>islower(<i>ch</i>)</code>	字符是小写字母吗?
<code>isprint(<i>ch</i>)</code>	字符是打印字符吗? 含空格符。
<code>ispunct(<i>ch</i>)</code>	字符是标点符号吗? 即非字母,数字或空格的可打印符号。
<code>isspace(<i>ch</i>)</code>	字符是空格符(包括空格,制表符,回车,换行,和格式馈送符)吗?
<code>isupper(<i>ch</i>)</code>	字符是大写字母吗?
<code>isxdigit(<i>ch</i>)</code>	字符是十六进制数字吗? 包括数字和从 <code>A</code> 到 <code>E</code> 的字母,大小写均可。

注解

虽然所有这些函数以整型作为参数,但仅用到参数的低字节。因为它们是假设你向其传递一个字节长的值,即一个字符。C 和 C++ 都采用该方式,因为一个 char 值在表达式中被自动地扩展为 int。

在以下示例中,如果从一个样本串中获取的是空格符,程序就打印输出一条信息:

```
#include <ctype.h>
#include <iostream.h>
//...
char *s = "Here is a sample string.";
if (isspace(s[4]))
    cout << "Character is a space.";
```

labs

长整型数的绝对值

labs 函数返回一个整数的绝对值。该函数有以下的声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>

long labs(long num);
```

负数的绝对值为删除负号后的数字,正数的绝对值就是其本身。例如,以下代码打印输出数字 5:

```
#include <stdlib.h>
#include <iostream.h>
//...
cout << labs(-5) << endl;
```

相关函数有 abs,取整型数为参数;fabs,参数取双精度数。

ldexp

把尾数与指数结合起来

ldexp(load exponent)函数是 frexp 函数的反函数。ldexp 函数取一浮点数的尾数和指数部分,按公式: $\text{value} = x * 2^{\text{exp}}$ 来结合它们,返回结果 value。该函数有以下声明格式,其中参数 x 和 exp 分别表示尾数和指数:

```
#include <math.h> // 或 #include <cmath>
```

```
double ldexp(double x, int exp);
```

该函数并不很重要,因为 `pow` 函数可以执行同样的功能。如果你不想分析浮点数的构成,就可忽略它。

ldiv

两个长整数相除

`ldiv` 函数执行两个长整数相除,并返回一个含有商和余数的结构。该函数有以下的声明格式:

```
#include <stdlib.h> // 或 #include <cstdlib>
```

```
ldiv_t ldiv(long numerator, long denominator);
```

以结构 `ldiv_t` 返回执行结果,它有以下格式。商(`quot`)表示一个乘以除数(`denominator`)后不超过被除数(`numerator`)的最大整数。

```
struct ldiv_t {  
    long quot;  
    long rem;  
};
```

除数是零时未定义函数的操作。

相关函数有 `div`,执行两个整型数的相除。求余运算符(`%`)提供了获得余数的另一方法。

localeconv

获取环境设置

`localeconv` 函数返回一个包含当前环境设置的结构,该结构提供了特定国家的关于数字显示的信息。例如,此结构指定了是用逗号还是句点来显示一个小数点。`localeconv` 函数有以下声明格式:

```
#include <locale.h> // 或 #include <clocale>
```

```
struct lconv * localeconv(void);
```

下列代码演示了该函数的调用：

```
#include <stdio.h>
#include <locale.h>
//...
iconv * ic = localeconv();
```

localeconv 返回的结构不能更改。当然,随后调用 localeconv 和 setlocale 将重写此结构。关于具体细节,如此结构的排列,参见你的编译器文档。

log

自然对数

log 函数返回其参数的自然对数值;自然对数是以 e 为基底。该函数是 exp 函数的反函数,exp 函数也是以 e 为基底。(以 10 为基底的,参见函数 log10。)log 函数有以下声明格式:

```
#include <math.h> // 或 #include <cmath>

double log(double x);
```

所返回的数就是 x 的自然对数。如果 x 为负值时,返回 NaN(表示“Not a Number”);如果 x 为零,返回 INF(表示“无穷”)。

log10

常用对数

log10 函数返回其参数的以 10 为基底的对数值。该函数有以下的声明格式:

```
#include <math.h> // 或 #include <cmath>

double log10(double x);
```

所返回的数就是 x 的常用对数。如果 x 为负值时,返回 NaN(表示“Not a Number”);如果 x 为零,返回 INF(表示“无穷”)。

下例返回了几个数的对数值。代码打印输出数字 2、3 和 3.47712。

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

```
//...
cout << "log of 100 is" << log10(100) << '\n';
cout << "log of 1000 is" << log10(1000) << '\n';
cout << "log of 3000 is" << log10(3000) << '\n';
```

longjmp

跳转至地址存储的

longjmp 函数跳转至最后一次调用 setjmp 函数捕获的地址上。这两个函数使程序能够在函数之间直接跳转。longjmp 函数有以下声明格式:

```
#include <setjmp.h> // 或 #include <csetjmp>
```

```
void longjmp(jmp_buf envbuf, int status);
```

envbuf 参数是由调用 setjmp 函数所设置的环境缓冲。它记录了系统堆栈的状态,包括程序指针。

status 参数给 setjmp 指定了一个新的返回值。执行 longjmp 的结果是跳回含 setjmp 调用的语句;setjmp 被再次执行。然而,这一次 setjmp 返回为 status 指定的数字。longjmp 不能传递 0 值,如果把 0 传递给 status, longjmp 将用非零值代替。

下面是一个调用 longjmp 的例子:

```
#include <setjmp.h>
```

```
jmp_buf envbuf;
```

```
//...
```

```
longjmp(envbuf, 5);
```

// 该 longjmp 调用使 setjmp 被再次执行并返回 5。更完整的举例请见 Setjmp。

malloc

分配内存

malloc 函数分配一块指定大小的内存,并返回一个指向该内存块的指针。这种分配内存的方法是从 C 语言中继承过来的,之所以支持该方法,是考虑到向后的兼容性。C++ 的 new 运算符实现与 malloc 同样的功能,并提供了一些优点(参见第十一章)。使用 malloc 函数,须注意它有以下声明格式。类型 size_t 是一个足够大的整数以存储任意类型的字节数大小:

```
#include <stdlib.h>      // 或 #include <cstdlib>
```

```
void *malloc(size_t size);
```

如果系统没有你所需求的多余内存,函数 `malloc` 就返回 `NULL`。否则的话,就返回一个指向内存块的指针。该块内存根据数据结构的类型进行动态分配。例如,以下代码就分配了充分的空间来容纳 `n` 个整型数:

```
#include <stdlib.h>
int n;
int *p;
//...
p = reinterpret_cast<int*>(malloc(sizeof(int) * n));
if (p == NULL)
    // 输出错误信息并结束。
```

注意,需要对 `malloc` 的返回值进行类型转换;用 `new` 来分配内存时,就不需要进行这一步操作了。(可用老版本的 C 语言来达到相同的目的。参阅 `free`、`calloc` 和 `realloc` 函数。)

● 注意

一些 C 和 C++ 编译器提供了 `malloc` 函数的变形,以处理一些有关特定操作平台的问题,如分段内存。关于这些特例,请参阅你的编译器文档。相比之下, `new` 和 `delete` 总能相对当前的操作平台和内存模式实现更优的性能。且 `new` 和 `delete` 的用法更简单。

mem<op>

内存块函数

标准的 C /C++ 函数库包含了一些有效的内存操作函数。要在 C++ 中使用这些函数,就得首先包含文件 `string.h`,其中也声明了一些串函数(参见“String Function”)。在老版本的 C 中,文件 `memory.h` 声明了这些内存操作函数。

```
#include <string.h>      // 或 #include <cstring>
```

总结

下表总结了内存块函数。参数 `p`、`p1` 和 `p2` 是由类型为 `void*` 的指针所指向的内存块。参数 `n` 是指定内存块大小的整数;如果你只想查找或拷贝内存块的前一部分, `n` 就可以小于内存块的大小。参数 `ch` 的类型为 `int`,表示单个字符。

函数	操作
<code>memchr(p, ch, n)</code>	返回一个指针,该指针指向在内存块中与参数 <code>ch</code> 的低位相匹配的第一个字节。
<code>memcmp(p1, p2, n)</code>	比较两个块中的 <code>n</code> 个字符。根据 <code>p1</code> 小于、等于或大于 <code>p2</code> 来返回负数、零或正数。这些返回值类似于 <code>strcmp</code> 的结果。
<code>memcpy(p1, p2, n)</code>	将内存块 <code>p2</code> 拷贝到 <code>p1</code> 。
<code>memmove(p1, p2, n)</code>	将内存块 <code>p2</code> 拷贝到 <code>p1</code> ;重叠位置的字节也能够正确拷贝。
<code>memset(p, ch, n)</code>	将一个内存块的 <code>n</code> 个字符都设置为字符 <code>ch</code> 的低字节。

注意,除了函数 `memcpy` 和 `memmove` 中的第一个参数,这些函数中的所有指针参数被声明为 `const void *`。关键字 `const` 表明函数不更改所指向的任何数据类型。(函数 `memmove` 是一个例外;如果发生数据重叠,则由 `p2` 所指向的数据就会被更改,但不通过指针 `p2` 实现。)

在每一函数中,`n` 的类型为 `size_t`。该类型在文件 `stdlib.h` 中声明,它足以存储任意类型的内存块的长度。(其典型类型是 `unsigned long`。)

注释

这里的大多数函数返回指针。`memcpy`、`memmove` 和 `memset` 返回第一个参数:一个指向缓冲的指针。`memchr` 函数返回一个指向第一个匹配字节 `ch` 的指针。由于返回的是一个类型为 `void` 的指针,所以在使用该指针之前要将其转换成将定的类型。例如:

```
#include <string.h>
//...
char *p, *buf = "This is a character X array."
p = (char *) memchr(buf, 'X', strlen(buf));
int index = p - buf;    // 对指针进行数学加减来获得 index 的值。
```

函数 `memcpy` 和 `memmove` 彼此相似。`memcpy` 能够把一个数组(`arr1`)中的所有值复制到另一个数组(`arr2`),而且一般比 `memmove` 要快:

```
short arr1[100], arr2[100];
//...
memcpy(arr2, arr1, 100 * sizeof(short));
```

但是 `memmove` 函数更具有灵活性,它能在一个数组中移动数值。下面语句实现了用一个数组指针来移位 100 个数值,位置 1~100 被移位至 0~99:

```
short arr1[500];  
//...  
memmove(arr1, arr1 + 1, 100 * sizeof(short));
```

modf

把浮点数分解为 整数和小数

`modf` 函数将一个浮点数类型的参数分解,并分别返回其整数和小数部分。返回值与参数有相同的符号。该函数有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>  
  
double modf(double x, double * intptr);
```

`modf` 返回参数 `x` 的小数部分。整数部分保存在 `intptr`。例如,以下代码将 3.0 赋于变量 `f`,将 0.14159 赋于变量 `d`:

```
#include <math.h>  
//...  
double d;  
double f = modf(3.14159, &d);
```

参阅函数 `ceil` 和 `floor`。

perror

打印错误信息

`perror` 函数以形式“string:error-message”打印输出一条错误信息, `string` 由函数调用所定义, `error-message` 由当前错误情况来确定。(后者由全局变量 `errno` 的当前值来决定。编译器实现给每一 `errno` 定义了一个串。) `perror` 函数有以下的声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>  
  
void perror(const char * string);
```


错误信息以换行符终止,并向标准错误输出,通常是显示器。参数 `string` 应该是一个标准错误前缀。例如,以下代码在错误条件 1 下打印输出一条错误信息:

```
#include <stdio.h>
//...
errno = 1;
perror("ERROR MESSAGE");
```

该程序在 Microsoft Visual C++ 6.0 下输出如下:

```
ERROR MESSAGE:Operation not permitted
```

printf

打印输出格式化串

`printf` 函数向标准输出发送一个格式化串。`printf` 是 C 库中打印数据的标准函数。虽然 C++ 流对象 `cout` 能够实现同样的功能并具有一些优点,但是 `printf` 仍旧是输出任意类型值的好方法。`printf` 与 `cout` 的对比参照第四章。

该函数有以下的声明格式。其中至少必须有一个参数,即一个格式化字符串,其后可跟随另外的参数;每一个参数都是一数据域。未由格式指示符指定的字符按原样打印输出。

```
#include <stdio.h>    // 或 #include <cstdio>

int printf(const char * format,...);
```

格式化字符串包含了零个或更多的被称为“格式指示符”的字符。省略号(...)表示字符串 `format` 之后可跟随任意多的参数。对于每一个格式指示符,都必须有一个数据-数域参数。第一个格式指示符对应第一个数据的数域,第二个格式指示符对应第二个数域,等等。例如,以下的函数调用包含了一个带有两个指示符(%d 和 %f)的格式化串,及两个数域参数, `i` 和 `x`:

```
printf("i = %d and x = %f\n", i, x);
```

由 `printf` 返回的值是实际打印输出的字符个数。

格式指示符

表 15-2 总结了 `printf` 中的格式指示符。

表 15-2 printf 中的格式指示符

指示符	描述
%c	以简单的 ASCII 字符打印输出值。(取低字节)
%d, %i	以十进制整型打印输出值
%u	以无符号十进制整型打印输出值
%o	以八进制整型打印输出值
%x, %X	以十六进制整型打印输出值
%e	以指数格式打印输出浮点数;例如,1.273110e+01
%E	同 %e,但使用 E 作为指数标志
%f	以标志格式打印浮点数
%g	采用 %e 或 %f 的格式
%G	同 %g,但若用指数格式,就使用 E 作为指数标志
%s	打印输出由数据所指的以 null 结尾的字符串
%p	以十六进制格式打印输出指针的值
%%	打印一百分符(%)

格式修正符

当你调用 printf 函数时,也许需要控制所打印的字符个数。printf 的语法使你能够最多用两个数字修改每个格式的意思,在此处以 min 和 precision 表示。c 是表 15-2 中的格式指示符。

```
%[-]min  
%[-]min.precision
```

这里的方括号中的内容并非是必需的,而是表示负号为可选项。若写入了负号,则表示在打印输出区域内的数据是左对齐的,而非缺省的右对齐状态。

min 修正符是一个表示输出区域大小的十进制数。例如,以下的格式就指定了一个整数至少以五个字符宽度输出:

```
%5d
```

precision 修正符也是一个整数,但它的意思因格式而变:

- 如果输出区域是字符串格式(%s),precision 就指定了所要打印的最多字符个

数

- 如果输出区域是浮点数格式(%f 或 %e), precision 就指定了在小数点后的所输出的数字个数
- 如果输出区域是整型格式(%i 或 %d), precision 就指定了所输出的最少数字个数。若被输出的数的位数不够,就以前置的零填补。

示例

以下代码打印输出含有两个值——一个字符串和一个数的格式化串:

```
#include <stdio.h>
//...
char * state = "Alaskan";
double temp= 25.7;

printf("The %s temperature is %5.4f. \n", state, temp);
```

程序运行后,输出结果如下:

The Alaskan temperature is 25.7000.

pow

计算乘方

pow 函数计算一个指定基底的任意次幂。(不同于 exp 函数的是,它不限制一个固定的基底。)该函数有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>

double pow(double x, double y);
```

例如,以下语句计算数 x 的平方、立方和平方根的值:

```
#include <math.h>
#include <iostream.h>
double x;
//...
cout << "x to 2nd power is" << pow(x, 2) << " \n";
cout << "x to 3rd power is" << pow(x, 3) << " \n";
cout << "x to 0.5 power is" << pow(x, 0.5) << " \n";
```

putc**输出一个字符到文件流**

putc 函数用参数 *ch* 的低字节向一指定文件输出流发送一个字符。如果出错, putc 返回 EOF; 否则, 返回打印的字符。putc 的声明格式如下:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
int putc(int ch, FILE * stream);
```

putc 函数与 fputc 函数完全相同。之所以出现这种重复是因为 putc 通常是以宏实现, 提供了比 fputc 更优越的性能。故 putc 比 fputc 更常用。有关文件流操作的详情, 参见本章的主题“Files”。

putchar**打印输出字符**

putchar 函数用参数 *ch* 的低字节向标准输出打印输出一个字符。(标准输出通常是显示器。)若调用出错, 返回 EOF; 成功时, 返回所打印的字符。该函数有以下的声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
int putchar(int ch);
```

有关控制台 I/O 函数的详情, 参见本章的主题“Console”。

puts**打印输出一字符串**

puts 函数向标准输出打印输出一个字符串, 并在串的末尾自动加上换行符。(标准输出通常是显示器。)若调用出错, 返回 EOF; 成功时, 返回一个非负值。该函数有以下的声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>
```

```
int puts(char * str);
```

有关控制台 I/O 函数的详情, 参见本章的主题“Console”。

qsort

进行快速排序

qsort 函数将一个任意大小和类型的数组进行排序。在调用该函数时,你要指定一个回调函数的地址;回调函数是一个由你所定义的函数,qsort 函数用它来比较任意两个元素。qsort 函数有以下的声明格式:

```
#include <stdlib.h>          // 或 #include <cstdlib>

void qsort(void *buf, size_t num, size_t size, int (compare)(const void *, const void *))
```

参数 buf、num 和 size 描述了被排序的数组。buf 指向数组的起始处,num 是元素的个数,而 size 是每一个元素的大小。

参数 compare 是一个函数的地址,它根据所指向的第一个参量的值是小于、等于还是大于所指向的第二个参量来返回一个负数、零或正数。该函数的声明中有两个 void * 类型的参量。你必须将这两个参量强制转换成所返回的指针类型,并取用指针地址中的内容。

例如,以下代码对一个整型数组进行排序。cmp 函数获得类型 int * 的指针,这就成功地转换了数据类型,并获得了数值:

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

int cmp(const void *p1, const void *p2) {
    int i = *(int *)p1;    // 获取整型指针。
    int j = *(int *)p2;
    return i < j ? -1 : (i == j ? 0 : 1);
}

void main() {
    int dat[] = {1, 99, 17, 40, 39, 16, 45, 7, 51, 3};

    qsort(dat, 10, sizeof(int), &cmp);
    for (int i = 0; i < 10, i++)
        cout << dat[i] << "\n";
}
```

● 注意

如果你对一个字符串数组进行排序,则被排序的元素本身就是一个类型为 char

* 的指针。因此,函数 `cmp` 接收到的数据类型即为 `char **`,它必须被转换成正确的数据类型从而获取指针地址中的类型为 `char *` 的数据:

```
#include <stdio.h>

int cmp(const void * p1, const void * p2) {
    char * a = *(char **)p1; // 获取 char * 类型的指针。
    char * b = *(char **)p2;
    // strcmp returns values as required for qsort
    return strcmp(a, b);
}
```

诸如 `qsort` 的函数通过使用一个回调函数提供了强大的灵活性。事实上,用户自己定义了“大于”和“等于”的意思。但要牢记,对于指针数组来说,就有两层间接指针,如上例所示。

raise

发送一个软中断信号

`raise` 函数向正在执行的程序发送一个软中断信号,告知发生了一个运行时错误。如果没有 `absolutely passe`,该函数就显得过时了,因为 C++ 的出错处理可执行同样的功能,并且使用起来更为方便。(参见第十三章的 `try`。)然而,C++ 支持该函数和 `signal` 函数,是考虑到了向后的兼容性。该函数有以下的声明格式:

```
#include <signal.h>      // 或 #include <csignal>

int raise (int signal);
```

如果调用成功,该函数返回零;反之,则返回非零值。虽然具体实现时会给出一些其他设置值,C/C++ 标准库还是为参数 `signal` 定义了几个值:

信号	意义
SIGABRT	程序非正常终止
SIGFPE	浮点运算错误
SIGILL	非法的机器指令
SIGINT	Ctrl + C 中断
SIGSEGV	无效的内存存取权限
SIGTERM	程序正常终止

以下是一个调用函数 `raise` 的例子：

```
#include <signal.h>
//...
raise(SIGFPE); // 发出浮点运算错误消息。
```

`raise` 的效果是将执行权转交给适当的 `signal` 句柄。更多详情请参阅 `signal`。

rand

产生随机数

`rand` 函数产生一个介于 0 和 `RAND_MAX` 之间的随机数。该函数有以下的声明格式：

```
#include <stdlib.h>      // 或 #include <cstdlib>

int rand(void);
```

该函数返回随机数序列中的下一个数。(实际上是一个伪随机数序列,序列中的每一个数是由对其前面的数字进行复杂变换得到的。)为了模仿真正的随机性,首先要调用 `srand` 函数来给序列设置一个种子。例如,以下代码打印输出十个随机数：

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
//...
srand(time(NULL)); rand();
for (int i = 0; i < 10; i++)
    cout << rand();
```

你可以用 `RAND_MAX` 来除此结果,以获得一个介于 0 和 1.0 之间的浮点随机数。然后可通过倍乘和取整来把这些数转换成整数。例如,以下代码就以相等的概率返回整数 0,1,2,3 和 4。其中, `floor` 函数进行向下取整：

```
#include <stdlib.h>
//...
cout << floor(5 * rand() / RAND_MAX) << endl;
```

realloc

重新分配内存

`realloc` 函数类似于 Visual Basic 中的 `ReDim Preserve` 语句。它对已分配的内存

块(已由 malloc、calloc 或 realloc 分配的内存)进行处理,如果必要的话,重新调整大小,并对旧有的数据值尽可能的保存。大多数适用于 malloc 函数的语法也适用于 realloc。该函数有以下的声明格式:

```
#include <stdlib.h>      // 或 include <cstdlib>
```

```
void *realloc(void *memblock, size_t size);
```

realloc 返回重新分配块的地址,仅当 realloc 实现移动块的功能时,该地址才与旧指针值(memblock)有所不同。和 malloc 一样,如果系统没有你所要求的内存,realloc 返回 NULL,并且不改变原先的内存块。例如,以下代码分配了一块内存,然后重新分配了一个两倍大的内存:

```
#include <stdlib.h>
int n;
char *p;
//...
p = reinterpret_cast<char * >(malloc(n));
if (p == NULL)
    // 没有足够的内存,打印输出错误信息并返回。
//...
p = reinterpret_cast<char * >(realloc(p, n * 2));
if (p == NULL)
    // 没有足够的内存,打印输出错误信息并返回。
```

详情请参阅 malloc、calloc 和 free。

remove

删除文件

remove 函数从系统中删除一个文件,如果调用成功,返回零;不成功时,返回非零值。它有以下的声明格式:

```
#include <stdio.h>      // 或 #include <cstdio>
```

```
int remove (char const * fname);
```

例如,下列语句实现从根目录下删除文件 junk.txt 的功能:

```
#include <stdio.h>
//...
remove("c: \\ junk.txt");
```


rename

文件改名

rename 函数给一个文件或目录改名,如果调用成功,返回零;不成功时,返回非零值。它有以下声明格式:

```
#include <stdio.h>      // 或 #include <cstdlib>

int rename(char const * oldfname, char const * newfname);
```

oldfname 必须指向一个存在的文件或目录。如果 oldfname 指向一个文件,你可为 newfname 指定一个不同的位置,这样,该函数就达到了移动文件的目的。例如,下列语句改名 junk.txt 为 useful.txt 并移动 memo.txt 到 c:\mydocs 目录下:

```
#include <stdio.h>
//...
rename("c: \ \ junk.txt", "c: \ \ useful.txt");
rename("c: \ \ memo.txt", "c: \ \ mydocs \ \ memo.txt");
```

scanf

从键盘输入数据

scanf 函数从标准输入中读入数据,以多种格式来接收它(字符串、整数或浮点数)。scanf 并不打印输出提示信息,你需用 printf 来单独输出。C++ 为读取数据提供了库对象 cin,且它比 scanf 有更多的优越性。然而,scanf 仍旧是一个不错的获取输入数据的方式。

该函数有以下声明格式。该语法表明有一个参数是必需的,格式化字符串其后跟着零或是一些其他参数;每一个参数均为数据的地址。在本节的后面,还介绍了其他一些使 scanf 读入或是跳过的字符。

```
#include <stdio.h>      // 或 #include <cstdlib>

int scanf(const char * format, ...)
```

格式化字符串包含了零个或多个被称为“格式指示符”的字符。对于每一个格式指示符,都必须有一个数据-地址参数。第一个格式指示符对应第一个数据的地址,第二个格式指示符对应第二个地址,等等。例如,以下 scanf 调用有一个格式指示符(%d)和一个数据-地址参数(&i,一个整型数的地址):

```
scanf("%d", &i);
```

scanf 返回成功读入的字段个数。

scanf 的格式

表 15-3 描述了 scanf 的格式字符串中合法的格式指示符。

表 15-3 scanf 中的格式指示符

指示符	描述
%c	读入下一个字符,包括空白符。
%d, %i	读入一个整数。%i,而非%d,表示一个前导的 0 意味八进制;前导的 0x 意味着十六进制。
%u	同%i,但目的地址是一个无符号整型。
%o	以八进制读入一个整数。
%x, %X	以十六进制读入一个整数。
%e, %f, %g	读入一个浮点数,无论是标准格式还是指数格式。
%s	读入直到下一个空白符的所有字符,拷贝到一个字符串地址。
%p	以十六进制读入一个指针值。
%[chars]	匹配输入字段(chars)中的一个字符。
%%	匹配一个百分符(%)。

在格式化字符串后出现的所有 scanf 参数,必须是数据地址。这意味着你必须将一个变量与地址运算符(&)相结合。不过,由于一个字符串的名称已经是一个地址类型了,所以在字符串的情况下,就不需用地址运算符。例如:

```
#include <stdio.h>
//...
int id;
char name[256];
printf("Enter name and id number: ");
scanf("%s %i", name, &id);
```

格式修正符

在百分符(%)和格式指示符之间,若出现了字母“l”,就表示该格式是长型。要读入 double 类型的数,该参数是必须。若不使用“l”来指明类型,就无法得到正确的数据。例如:

```
double x,y,z;  
float f;  
  
scanf("%lf %lf %lf %f", &x, &y, &z, &f);
```

你可用“h”(短型)和“l”(长型)来修正整数的格式。如果一个变量被特别地声明为 short 或 long 而非 int 就应该使用这些修正符。

另一个修正符是星号(*)。当一个星号出现在%和格式指示符之间时,就表示下一输入字段被匹配但不赋给下一地址参数。例如,以下格式指示符使 scanf 忽略下一个字符:

```
%*c
```

格式化字符串中的其他字符

一个 scanf 格式化字符串有时候含有一些不仅仅是格式指示符的字符。scanf 通过匹配和忽略它们来对应这些字符。

当格式指示符中出现了空白符时,scanf 读入并忽略输入流中的所有空白符。前述的几个例子都是用空白符来划分数据域的。

当出现非空白字符时,scanf 就尝试去匹配它。如果不能匹配,scanf 就终止,不再读入任何更多的数据。

setjmp

为 longjmp 保存地址

setjmp 函数,以及 longjmp 函数,为函数间的直接跳转提供了一种方法。但该方法并不适用于结构化编程,所以应尽量避免,除非是低级系统编程所需。setjmp 有以下的声明格式:

```
#include <setjmp.h>      // 或 #include <csetjmp>  
  
int setjmp(jmp_buf envbuf);
```

setjmp 的效果是将系统堆栈的内容保存于变量 envbuf 中。实际上,它保存了调用 longjmp 时,能够跳转回来的程序位置。在初次调用该函数时,setjmp 返回 0。如果再次执行该函数(如来自对 longjmp 的调用),setjmp 返回一个非 0 值。下面的程序就利用了该返回值以避免一个无穷循环:

```
#include <setjmp.h>
```

```
#include <iostream.h>
jmp_buf envbuf;
int status;

void test_func(void) {
    cout << "I'm in test_func. " << endl;
    longjmp(envbuf, 5);    // 跳转到主程序的起始处。
}

void main()
{
    status = setjmp(envbuf);

    // 如果在从 longjmp 跳转回来之后执行了 setjmp, status 就被设为 5。

    cout << "This is just after setjmp. " << endl;
    if (status == 0)
        test_func();
    cout << "status = " << status << endl;
}
```

该程序首先调用 `setjmp` 来注册一个位置。在调用期间,它返回 0。当程序执行了 `test_func`,它就跳转回同样的位置。这就使 `setjmp` 被再次执行,但这次返回的是一个非零值(5)。接着,该值阻止 `test_func` 被再次执行。该程序打印输出如下的结果:

```
This is just after setjmp.
I'm in test_func.
This is just after setjmp.
status = 5
```

注意,为了成功执行 `longjmp`,调用 `setjmp` 的函数不能够被终止;也就是说,它必须仍旧在运行着。同时,如果一个函数调用了 `setjmp`,然后改变了自己的局部变量的值,则当 `longjmp` 被执行时,这些局部变量的值就变成没有被定义过的。(例外:局部声明的 `volatile` 并不受此约束。)

对以上的使用法则不必感到奇怪。当你在函数间跳转时,必定会碰到这样一些意外情况。

setlocale

设置新的环境格式

setlocale 函数指定了一个新的环境设置,该环境设置指定了一个特定国家的字符集和显示格式。setlocale 也可用来查询当前的设置值。该函数有以下的声明格式:

```
#include <locale.h>    // 或 #include <clocale>
```

```
char * setlocale(int type, const char * locale);
```

参数 *type* 指定了所使用的环境字符串:

类型值	描述
LC_ALL	所有类别。
LC_COLLATE	函数 strcoll 所用到的字符集。
LC_CTYPE	类似 isalnum 的字符函数。
LC_MONETARY	货币量的格式。
LC_NUMERIC	函数所显示的数字的格式,如 printf。
LC_TIME	如 asctime 等函数所使用的格式。

参数 *locale* 是一个包含新的环境设置值的字符串。ANSI 识别字符串“C”和“”。前者指定使用标准最小 C 设定值。后者指定使用由实现所决定的缺省设置。还有一些设置值由各自的实现来定义。(参见示例。)参数 *locale* 也可以被设置为 NULL,在此情况下,该函数只执行查询功能。

在任何情况下,函数 setlocale 返回一个包含当前环境设置值的字符串,如果环境参数是一个非 NULL 的值,setlocale 返回新的设置值。

下例演示了用 setlocale 函数来指定数字格式:

```
#include <stdio.h>
#include <locale.h>

void main()
{
    puts(setlocale(LC_NUMERIC, "French"));
    printf("%f\n", 8.007);
}
```

当此程序运行于 Microsoft Visual C++ 环境下时,打印输出以下结果。注意由 French 环境设置值,逗号(,)被用作小数点;由 printf 的缺省设置,显示小数点后六位数字:

```
French_France, 1252
8,007000
```

signal

注册信号句柄

signal 函数注册了一个信号的句柄。(即设置某一信号的对应动作。)虽然 C++ 异常处理(参见第十三章的 try)提供了一个更佳的出错处理,考虑到向后的兼容性,还是提供了对 signal 和 raised 的支持。函数 signal 有以下的声明格式:

```
#include <signal.h>      // 或 #include <csignal>

void (* signal)(int signal, void (* func)(int)) (int)
```

虽然 signal 的声明格式看上去很复杂,其实它仅有两个参数:

```
signal(signal, func)
```

参数 signal 的取值为 raise 一节中所介绍的信号数字,如 SIGINT、SIGILL 或 SIGFPE。

参数 func 可为以下的特殊值之一:SIG_DFL(缺省处理)或 SIG_IGN(忽略此信号)。例如,以下代码使程序忽略信号 SIGINT、SIGILL 和 SIGTERM:

```
signal(SIGINT, SIG_IGN);
signal(SIGILL, SIG_IGN);
signal(SIGFPE, SIG_IGN);
```

你也可以提供自己的信号句柄的地址,这些信号句柄必须是一个以整数作为参数的函数,并返回一个 void 类型(即不返回任何值)。例如,以下代码为两个不同的信号注册一个句柄,然后启用每一个信号作为测试。参数 sig 决定启用哪一个信号:

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

void handler(int sig) {
```

```
cout << "Signal received: " << sig << '\n';  
}  
//...  
signal(SIGINT, &handler);  
signal(SIGILL, &handler);  
raise(SIGINT);  
raise(SIGILL);
```

在信号句柄返回之后,程序自动跳转回到启用信号的语句的下一条语句,除非信号是 SIGFPE,因为它的返回行为是不确定的。

信号句柄的法则

信号句柄用到了一些法则。对于不同的操作系统平台,要参阅个人的编译器文档。通用的 ANSI 法则总结如下:

- SIGFPE 情况下,你要调用 `setjmp` 和 `longjmp` 函数来把程序的执行重置于启用信号之处。
- 除 SIGFPE 之外的其他信号,无须调用 `setjmp` 和 `longjmp` 函数。
- 避免调用以下函数: `stdio.h` 函数(如 `printf`),内存分配函数(如 `malloc`),和那些截取系统时间的函数。
- Windows NT 和 Windows 95(及更高版本)系统并不产生 SIGINT 信号来响应 Ctrl+C。然而,你可选择调用 `raise` 来产生任何信号。

返回值

`signal` 函数根据所给定信号返回先前句柄的地址。例如,如果用 `signal` 来为 SIGINT 注册一个句柄,该函数返回先前 SIGINT 句柄的值。为了以后的需要你可以保存此地址:

```
void (* saved_handler)(int);  
saved_handler = signal(SIGINT, &handler);  
//...  
signal(SIGINT, saved_handler);
```

`signal` 返回一个指向函数的指针也表明它具有不一般的声明格式。

sin**正弦函数**

sin 函数返回它的参量的正弦值。sin 有以下声明格式：

```
#include <math.h> // 或 #include <cmath>
```

```
double sin(double x);
```

该函数取一个角的弧度值作为参数,并返回一个介于 $1 \sim -1$ 之间的值。例如,以下代码打印输出 $\pi/2$ 的正弦值 1:

```
#include <math.h>
#include <iostream.h>
//...
double pi = atan(1) * 4;
cout << "sin(pi/2) = " << sin(pi/2) << "\n";
```

参见 acos、asin、atan、atan2、cos 和 tan。

sinh**双曲正弦函数**

sinh 函数返回它的参量的双曲正弦值。sinh 有以下声明格式：

```
#include <math.h> // 或 #include <cmath>
```

```
double sinh(double x);
```

如果计算结果超出了范围,该函数返回 HEGEVAL 并把全局变量 errno 设为 ERANGE。参见 cosh 和 tanh。

sprintf**写数据到字符串**

sprintf 函数是 printf 函数的一个变形,但它不是向控制台打印输出,而是向字符串写输出。当然,该字符串可在以后打印输出。可用 sprintf 函数以 ASCII 码的形式输出表示数字的一个字符串。

该函数有以下声明格式。第一个参数 buffer 是 printf 函数中所没有的。它必须是一个你所分配的足以容纳输出内容的字符串:


```
#include <stdio.h>    // 或 #include <cstdio>
```

```
int sprintf(char * buffer, const char * format, ...);
```

如同 printf 函数的声明格式,省略号(...)表示字符串 format 后面可以跟随任意个数的参数。这些参数包含要写的数据(参阅 printf)。例如,以下代码将一个浮点数 x 的 ASCII 表示法以及字符串“x = ”写到字符串 buf:

```
#include <stdio.h>
//...
char buf[156];
double x = 2.50e3;
sprintf(buf, "x = %f", x);
puts(buf);
```

此程序的输出如下:

```
x = 2500.000000
```

关于格式化字符的清单以及它们的含义,参见 printf。

sqrt

平方根函数

sqrt 函数返回它的参量的平方根值。该函数有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>
```

```
double sqrt(double x);
```

例如,以下语句返回 2 的平方根:

```
#include <math.h>
#include <iostream.h>
//...
cout << "square root of 2 is " << sqrt(2) << '\n';
```

srand

设定随机数种子

srand 函数设置随机数的种子(即初始化随机数发生器)。该函数有以下的声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
void srand(unsigned int seed);
```

该函数初始化一个伪随机数序列。通过调用 `rand` 函数来获取序列中的下一个数字。问题是如何给 `seed` 赋一个随机值。最简单的方法是调用系统时间。

```
#include <stdlib.h>
#include <time.h>
//...
srand(time(NULL));
rand();
```

不应该使用第一个随机数,因为它是基于系统时间的。但随后的对 `rand` 函数的调用就生成了一个很好的伪随机序列。详情请参阅 `rand`。

sscanf

从一字符串中读取数据

`sscanf` 函数提供了数据输入函数 `scanf` 的一个变形。它不是从键盘读取数据,而是从一个字符串中读取字符作为输入。例如,该函数能够将一个类似于“3.1415926”的数字字符串转换成一个实际的数字。`sscanf` 函数类似于 `atof` 函数族,但它可以在一次函数调用中输入一系列数字。

该函数有以下的声明格式。第一个参数 `buffer` 是 `scanf` 所没有的,它是一个含有要读入字符的字符串:

```
#include <stdio.h>    // 或 #include <cstdio>

int sscanf(char * buffer, const char * format, ...);
```

如同 `scanf` 的声明格式,省略号(...)表示字符串 `format` 后面可以跟随任意个数的参数。这些参数中的每一个都给出了一个目的地址,从串中读入的数据就置于该地址中。例如,以下代码将数字串“1.234”读入到浮点型变量 `x` 中:

```
#include <stdio.h>
//...
char data[] = "1.234";
double x;
sprintf(data, "%lf", &x);
```

有关格式字符的清单及它们的含义,参见 scanf。

str<op>

字符串函数

标准 C /C++ 库提供了大量对以 NULL 结尾的字符串进行操作和匹配的函数。要用这些函数,首先要包含头文件 string.h,其中也定义了一些内存块操作函数(参见“内存块函数”)。

```
#include <string.h>      // 或 #include <cstring>
```

语法

下表总结了最常用的字符串函数。参数 s、s1 和 s2 是由类型为 char * 的指针所表示的字符串,n 是一个类型为 size_t 的整数。除了特别注明的,每一个函数返回一个指向它的第一个串参数的指针。

函数	操作
strcat(s1, s2)	将串 s2 添加到串 s1 的后面。
strcmp(s1, s2)	比较两个字符串,根据 s1 是小于、等于或大于返回负数、零或正数。
strcpy(s1, s2)	将串 s2 拷贝到串 s1。
strlen(s)	返回串的长度,不含 NULL。
strncat(s1, s2, n)	将串 s2 添加到串 s1 的后面,但最多拷贝 n 个字符。
strncmp(s1, s2, n)	执行类似 strcmp 函数的功能,但最多比较 n 个字符。
strncpy(s1, s2, n)	将串 s2 拷贝到串 s1,但最多拷贝 n 个字符。
strstr(s1, s2)	返回一个指向串 s1 中最早出现 s2 处的指针。

函数库中还包含了以下的函数。它们中的一些是以上函数的变形;有些则提供了特定的匹配方式。

函数	操作
strchr(s, ch)	返回指向串 s 中最早出现 ch 处的指针。
strcoll(s1, s2)	根据环境设定实现与 strcmp 函数同样的功能,参见 setlocale。

<code>strcspn(s1, s2)</code>	返回在字符串 <code>s1</code> 中,串 <code>s2</code> 的任一字符相匹配的位置。
<code>strerror(n)</code>	返回指向错误信息字符串的指针。
<code>strpbrk(s1, s2)</code>	返回一指针,它指向 <code>s1</code> 中匹配 <code>s2</code> 中任一字符的首次出现的指针。
<code>strrchr(s, ch)</code>	返回指向串 <code>s</code> 中最晚出现 <code>ch</code> 处的指针。(该函数反向匹配。)
<code>strspn(s1, s2)</code>	返回在串 <code>s1</code> 中,串 <code>s2</code> 的任一字符不相匹配的位置。
<code>strtok(s1, s2)</code>	根据 <code>s2</code> 中的分隔符返回字符串 <code>s1</code> 中的相应标记。
<code>strxfrm(s1, s2, n)</code>	根据环境设定来变换串 <code>s2</code> ,并把结果保存于串 <code>s1</code> 。只转变前 <code>n</code> 个字符。参见 <code>setlocale</code> 。

注释

最常用的字符串函数是 `strcpy`、`strcat`、`strcmp` 和 `strlen`。当调用函数 `strcpy` 和 `strcat` 时要谨慎,因为它们极易增加一个串的长度,这就有出错的危险,使用它们前应确保用标字符串有足够的空间来容纳所有的数据。以下是一个避免此问题的示例:

```
#include <string.h>
//...
char name[30];
strcpy(name, "John");           //拷贝"John"
strcat(name, "Q. ");            //添加"Q. "
strcat(name, "Public");         //添加"public"
```

因为给第一个字符串 `name` 分配了 30 个字节,它就有足够的空间来容纳最终的字符串“John Q. Public”。在 C/C++ 中,确保字符串具有足够的空间是一个常见的问题。解决方法之一是采用其它函数:`strncpy` 和 `strncat`,它们限定了能够拷贝的字符个数。例如:

```
char s1[30], *s2;
//...
strncpy(s1, s2 29);
```

虽然上例的 `s1` 被分配了 30 个字节,但却要指明限定个数为 29,因为不这样的话,`strncpy` 有可能覆盖串终止符 `NULL`。同样的,对 `strncat` 的使用也要给 `NULL` 留下一个字节:

```
strncat(s1, s2, 29 - strlen(s1));
```

所有这些函数: `strcat`, `strcpy`, `strncat` 和 `strncpy` 将终止符 `NULL` 和字符串数据一同拷贝。

● 注意

考虑到使用 C/C++ 字符串时出现的缺陷,第五章至第八章所介绍的 `CStr` 类就比普通的字符串有更多的优越性。

在另外一些函数中, `strstr` 函数对串匹配和定位非常有用。它返回一个指向第一个字符串变量 `s1` 中的子串的指针。若你想要把此结果转换成一个索引值,只要加入对指针的数学运算即可。例如,下列代码查找单词“England”的索引位置:

```
#include <string.h>
//...
int index;
char aString[] = "This land, this realm, this England.";
index = strstr(aString, "England") - aString;
```

所有的字符串查找和定位函数都对字母的大小写敏感,这是因为它们是将字符的二进制数值进行对比。

函数 `strtok` 返回的是一些记号;它们是一些子串,通常为由分隔符划分的单词。例如,可以利用空格和标点符号作为分隔符,把一个句子解析成单个单词。初次调用 `strtok` 把要分析的字符串指定为参数 `s1`。为了继续分析该字符串,再次调用 `strtok`,并把 `NULL` 指定为参数 `s1`。(参见以下示例。)

如果没有查找到标记, `strtok` 返回一个空指针。这样,就可按如下语句编写一个循环调用来分析一个句子:

```
#include <string.h>
#include <iostream.h>
//...
char str[] = "This land, this realm, this England.";
char *p = strtok(str, " ,.");
while (p) {
    cout << p << '\n';
    p = strtok(NULL, " ,.");
}
```

运行该程序后,打印输出以下结果:

```
This  
land  
this  
realm  
this  
England
```

类型和返回值

有一些字符串函数取整型数作为一个参量。在这些函数中, `n` 的类型为 `size_t`。它是一个在文件 `stdlib.h` 内声明的类型, 有足够大的空间来存储任意类型的串。(它通常为 `unsigned long` 类型。) 返回一个整型数的字符串函数的类型也为 `size_t`。

除了有特别声明的, 每一个字符串函数通常返回一个指向第一个参量的指针。这就可使返回的值用于另一个函数调用。例如, 以下语句将两个串添加到字符串 `name` 的后面:

```
strcat(strcat(name, s1), s2);
```

strftime

输出以格式化表示的时间

`strftime` 函数(“string format time”)输出格式化的时间显示到一个串。它使你能够指定所要显示的日期和时间部分及以何种格式显示。你甚至能够指定以友好的界面方式来显示时间, 即显示月份和星期的名称。该函数有以下的声明格式:

```
#include <time.h> //或 #include <ctime>
```

```
size_t * strftime(char * str, size_t max_size, const char * fmt, const struct tm * time)
```

参数 `time` 是一个指向结构 `tm` 的指针, 该结构包含了时间和日期信息。有关此结构类型的描述请参阅“时间和日期函数”。

参数 `fmt` 是一个含有时间格式指示符的字符串, 串中还含有其他一些要写到目标串的字符, 如后面所描述的。

其他参数描述了 `strftime` 要写的目标串; `str` 指定了该串的地址, `max_size` 指定了要写的字符的最大个数。该函数返回所输出的字符数。

格式指示符

格式化字符串(fmt)可选用表 15-4 列出的指示符。

表 15-4 strftime 的格式化指示符

格式指示符	描述
%a	缩写星期名
%A	非缩写星期名
%b	缩写月份名
%B	非缩写月份名
%c	标准日期和时间
%d	每月的日子(1-31)
%H	小时(0-23)
%I	小时(0-12)
%j	每年的日子(1-366)
%m	月份(1-12)
%M	分钟(0-59)
%P	AM /PM
%S	2 位数字的秒数(0-61)
%U	星期日作为一周的第一天时的 2 位数字的星期号(00-53)
%w	星期号,星期日为 0(0-6)
%W	星期一作为一周的第一天时的 2 位数字的星期号(00-52)
%x	日期
%X	时间
%y	2 位数字的年号(00-99)
%Y	4 位数字的年号
%Z	时区名,如果没有时区则没有字符
%%	字符 %

请注意,由于百分号是转义字符,所有若想向目标字符串输出百分号,就须连续用两个百分符。

示例

以下示例以几种不同的格式打印输出当前的时间和日期:

```
#include <stdio.h>
#include <time.h>

void main()
{
    char s[100];
    time_t t = time(NULL);
    tm *tmp = localtime(&t);
    strftime(s, 99, "The month is %B.", tmp);
    puts(s);
    strftime(s, 99, "The day of the week is %A.", tmp);
    puts(s);
    strftime(s, 99, "The time is %H:%M %p.", tmp);
    puts(s);
}
```

以下是此程序的输出:

```
The month is December.
The day of the week is Sunday.
The time is 12:59 PM.
```

strtod

将串转换为双精度值

strtod 函数("string to double")将一数字串转换为数值。除了少数不同之处,该函数实现与 atof 同样的功能。strtod 函数有以下的声明格式:

```
#include <stdlib.h> // 或 #include <cstdlib>
```

```
double strtod(const char *str, char **end);
```

该函数跳过前导的空格符,读入能形成合法的浮点数表达式的字符。例如,“-24.56xy7”被看作-24.56。函数把“E”和“e”看作指数符号,所以“5E2”被看作50.0。

不同于 `atof` 函数的是, `strtod` 设置了一个指向使读入扫描停止的字符的指针, 该指针指向不能作为浮点数读入的第一个字符。溢出时, 返回 `HUGE_VAL` 或 `-HUGE_VAL`。下溢出时, 返回 0。在此两种情况下, 全局变量 `errno` 被设为 `ERANGE`。

● 注意

下溢出是在尝试读取一个非常接近于零的数时出现。例如, `5E-999` 就太小了, 而无法表示成双精度数。

下例从键盘读取一个字符串:

```
#include <stdlib.h>
#include <stdio.h>
//...
char *p;
char str[100];
double x = strtod(gets(str), &p);
```

strtol

将串转换为长整型值

`strtol` 函数(“`str`int to `long`”)根据给定的进制将一数字串转换为数值。该函数有以下的声明格式:

```
#include <stdlib.h> // 或 #include <cstdlib>

long strtol(const char *str, char **end, int radix);
```

该函数根据给定的 `radix` 读入字符串 `str`。例如, `radix` 的值为 16 时, 函数以十六进制读数。该函数跳过前导的空格符, 并在第一个非数字字符处停止读取。

参数 `end` 是一个 `char *` 指针的地址; 该指针被设为指向第一个不能读取的字符。当然你可忽略这个值, 但你必须指定指针的地址。

以下代码以十六进制方式读入字符串“FF”并返回结果 255:

```
#include <stdlib.h>
#include <stdio.h>
//...
char *p;
long n = strtol("FF", &p, 16);
```

相关函数有 `atoi`, `atol` 和 `atof`。函数 `strtoul` 实现与 `strtol` 同样的功能,只不过返回 `unsigned long`。

strtoul

将串转换为无符号长整型值

`strtoul` 函数(“`str`int to `unsigned long`”)根据给定的进制将一数字串转换为数值。该函数有以下的声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
unsigned long strtoul(const char * str, char ** end, int radix);
```

该函数实现与 `strtol` 同样的功能,只不过返回 `unsigned long`。详情请参阅 `strtol`。

以下代码以十六进制方式读入字符串“FF”并返回结果 255:

```
#include <stdlib.h>
#include <stdio.h>
//...
char * p;
unsigned long n = strtoul("FF", &p, 16);
```

system

执行系统命令

`system` 函数向操作系统的命令处理器发送一个字符串。它使你能够用系统的命令语法来运行一些其他程序或进程。该函数有以下的声明格式:

```
#include <stdlib.h>    // 或 #include <cstdlib>
```

```
int system(const char * str);
```

返回值的含义是由系统实现所定义的,但根据惯例,0 表示调用成功。例如,以下语句向系统发送 `DIR /W` 命令。(该命令在 DOS 和 Windows 操作系统下有意义。)

```
#include <stdlib.h>
//...
system("DIR /W");
```

tan**正切函数**

tan 函数返回它的参量的正切值。**tan** 有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>

double tan(double x);
```

该函数以弧度方式读入一个角并返回此角的两边之比。例如,以下代码打印输出 $\pi/4$ 的正切值 1:

```
#include <math.h>
#include <iostream.h>
//...
double pi = atan(1) * 4;
cout << "tan(pi / 4) = " << tan(pi / 4) << endl;
```

参见 **acos**、**asin**、**atan**、**atan2**、**cos** 和 **sin**。

tanh**双曲正切函数**

tanh 函数返回它的参量的双曲正切值。**tanh** 有以下的声明格式:

```
#include <math.h>    // 或 #include <cmath>

double tanh(double x);
```

参见 **sinh** 和 **cosh**。

tmpfile**打开临时文件**

tmpfile 函数以二进制读/写方式打开一个临时文件。当该文件被关闭时或程序终止时,文件被自动删除。由于文件将被删除,故临时文件只适合用来存储临时数据而不能用来存放永久性信息。它有以下的声明格式:

```
#include <stdio.h>    // 或 #include <cstdio>

FILE * tmpfile(void);
```

若调用成功,返回指向所创建的临时文件流的指针;反之,返回 NULL。有关文件 I/O 的详情,参阅“Files”。

Time

时间和日期函数

标准的 C/C++ 函数库中包含时间和日期函数,通过调用它们,可获取系统的当前时间,并按照不同的格式显示输出。当然,库中还含有一些相关函数,如获取程序运行时间,以提供对实时操作的支持。使用这些函数之前,须包含文件 `time.h`。

```
#include <time.h>      //或 #include <ctime>
```

总结

下表总结了时间和日期函数。参数 `t`、`t1` 和 `t2` 的类型均为 `time_t`,一个数字化的时间值;参数 `tm` 的类型是结构 `tm`。

函数	功能
<code>asctime(tm)</code>	以 <code>tm</code> 结构回显一个字符串
<code>clock()</code>	返回类型为 <code>clock_t</code> 的程序运行时间;被 <code>CLOCKS_PER_SEC</code> 除之后,该值被转换成秒
<code>ctime(t)</code>	以 <code>time_t</code> 结构回显一个字符串(使用本地时间)
<code>difftime(t2, t1)</code>	以秒数返回两个时间的差值(<code>t2-t1</code>)
<code>gmtime(*p_t)</code>	返回一个指向临时 <code>tm</code> 结构的指针,由 <code>p_t</code> 所指的 <code>time_t</code> 数据初始化。使用格林威治时间。
<code>localtime(*p_t)</code>	返回一个指向临时 <code>tm</code> 结构的指针,由 <code>p_t</code> 所指的 <code>time_t</code> 数据初始化。使用本地时间。
<code>mktime(*p_tm)</code>	为 <code>p_tm</code> 所指的 <code>tm</code> 结构设置星期日期和年日期。返回相应的 <code>time_t</code> 值
<code>strftime(s, size, fmt, *p_tm)</code>	由给定的 <code>tm</code> 结构返回一个格式化的字符串。参见 <code>strftime</code>
<code>time(*p_t)</code>	以 <code>time_t</code> 返回当前时间。如果 <code>p_t</code> 非 NULL,函数设置 <code>p_t</code> 指向的 <code>time_t</code> 值。

文件 `time.h` 中定义了三个特殊的类型 `tm`、`time_t` 和 `clock_t`。类型 `time_t` 是一个有足够字长来存放日期/时间值的整数。

注释

许多日期/时间操作都是通过调用函数 `time` 来启动的。它返回一个 `time_t` 值。你可以传递一个类型为 `*time_t` 的指针,它所指向的值是当前的时间,或者,你也可以传递一个空指针。函数在此两种情况下,都返回当前时间。

```
#include <time.h>
//...
time_t t;
t = time(NULL);
```

将 `time_t` 值转换成文本描述的最简单的方法是调用函数 `ctime`,它返回一个指向字符串的指针,该字符串有以下的格式:

```
weekday month dd hh:mm:ss yyyy \n
```

例如,以下的程序就显示了当前的时间:

```
time_t t = time(NULL);
cout << ctime(&t);    // 需要一个指向 time_t 的指针。
```

下面是在一个星期一的下午 1:04 执行了以上语句后的显示输出:

```
Mon Nov 02 13:04:55 1998
```

使用一个 `tm` 结构

另外一个用到 `time_t` 值的地方是通过调用 `gmtime` 函数或 `localtime` 函数将其转换成 `tm` 结构。例如,以下代码能够获取系统时间并用其来得到一个指向 `tm` 结构的指针。执行了此代码后, `tmp` 指向一个包含当前时间的结构里:

```
time_t t = time(NULL);
tm * tmp = localtime(&t);
```

● 注意

由 `localtime` 或 `gmtime` 函数返回的结构是临时性的,在每一次调用这些函数时,它都被覆盖重写。为了维持这个数据不变,须将它拷贝到你自己的 `tm` 结构:

```
tm tmNow = * tmp;    // Copy from struct pointed to.
```

一个 `tm` 结构把一日期/时间值分解为不同的数据域,包括星期值(0~6)和年代值(0~365)。文件 `time.h` 以如下形式声明 `tm` 结构:

```
struct tm {
```

```

int tm_sec;    // Seconds, 0-59
int tm_min;    // Minutes, 0-59
int tm_hour;   // Hours, 0-23
int tm_mday;   // Day of the month, 1-31
int tm_mon;    // Month, 0-11
int tm_year;   // Years - 1900
int tm_wday;   // Day of the week, 0-6
int tm_yday;   // Day of the year, 0-365
int tm_isdst;  // Daylight Savings Time indicator
               // > 0 if DST in effect, 0 if not.
}

```

在转换成 `tm` 结构后,你就可以访问各自的数据域,来得到日期或时间的特定部分。例如,以下代码打印输出某一月的当前日期:

```

time_t t = time(NULL);
tm *tmp = localtime(&t);

cout << "Today's day of month is : " << tmp->tm_mday;

```

函数 `mktime` 取用一个指向 `tm` 结构的指针并设置它的部分域值。特别是它可以基于其他域值来设置结构的 `tm_wday` 和 `tm_yday` 域的值。例如,你可用以下函数,以从 0 到 6 的数字形式返回星期值,而无须考虑年、月、日的值:

```

#include <time.h>
int get_weekday(int month, int day, int year) {
    tm t;

    t.tm_mon = month + 1;
    t.tm_mday = day;
    t.tm_year = year - 1900;
    if (mktime(&t) == -1)
        return -1;
    else
        return t.tm_wday;
}

```

如果 `mktime` 函数不能使用结构(上例为 `t`)中的数据来创建一个有意义的年历日期,它就返回 -1。上例就用该条件作为测试条件。注意,在调用 `mktime` 之前并不需要设置 `tm_wday` 和 `tm_yday` 域的值,因为这些值是由 `mktime` 来设置。

只要输入了 `tm` 结构,你就能调用 `asctime` 函数来较快地获得日期/时间值的字符串表达。`asctime` 函数使用与 `ctime` 函数同样的格式,前面已对此加以介绍。

打印输出格式化字符串

另一个用到 `tm` 结构之处是调用 `strftime` 函数并指定一个格式化字符串。该函

数类似于 `printf`, 但它对时间值的一部分进行格式化。例如, 以下代码以名称 (“Sunday”、“Monday”、“Tuesday”等) 打印输出星期日期:

```
time_t t = time(NULL);
tm * tmp = localtime(&t);
char s[100];
strftime(s, 99, "The day of the week is %A.", tmp);
puts(s);
```

详情参阅 `strftime`。

tolower

将字符转换为小写字母

`tolower` 函数将一个大写字母转换成小写字母, 并返回转换结果。如果被转换字母是小写字母或非字母字符, 则返回它的原值。该函数有以下的声明格式:

```
#include <ctype.h>    // 或 #include <cctype>

int tolower(int ch);
```

参数类型为 `int`, 但只有 `ch` 的低字节(存放单个字符的字节)起作用, 高字节被忽略。下例用 `tolower` 函数转换一个完整的字符串, 每次转换一个字符:

```
#include <ctype.h>
#include <string.h>

void convert_to_lower(char s[]) {
    int i;

    for (i = 0; i < strlen(s); i++) {
        s[i] = tolower(s[i]);
    }
}
```

下面是一个在转换前的样串:

```
I used to work for the FBI.
```

下面是该串在调用 `convert_to_lower` 函数后的返回结果。请注意, 小写字母和标点符号都被保留不变:

```
i used to work for the fbi.
```

toupper**将字符转换为大写字母**

`toupper` 函数将单个小写字母转换成大写字母,并返回转换结果。如果被转换字母是大写字母或非字母字符,则返回它的原值。该函数有以下的声明格式:

```
#define <ctype.h> // 或 #include <ctype.h>
```

```
int toupper(int ch);
```

参数类型为 `int`,但只有 `ch` 的低字节(存放单个字符的字节)起作用,高字节被忽略。下例用 `toupper` 函数转换一个完整的字符串,每次转换一个字符:

```
#include <ctype.h>
```

```
#include <string.h>
```

```
void convert_to_upper(char s[]) {
```

```
    int i;
```

```
    for (i = 0; i < strlen(s); i++) {
```

```
        s[i] = toupper(s[i]);
```

```
    }
```

```
}
```

下面是一个在转换前的样串:

```
I used to work for the FBI.
```

下面是该串在调用 `convert_to_upper` 函数后的返回结果。请注意,大写字母和标点符号都被保留不变:

```
I USED TO WORK FOR THE FBI.
```

va_<op>**变长度参数表**

标准的 C /C++ 函数库定义了一套宏函数,用以实现对变长度参数列表的访问。访问一个可变长度参数列表中的单元的方法由具体实现来定义。为了编写一个简洁的源代码,你就该使用此处所介绍的宏。用这些宏之前,须包含头文件 `stdarg.h`。

```
#include <stdarg.h>
```

总结

下表总结了这些宏。用到这些宏的函数至少必须含有一个参数,该参数在宏 `va_start` 里被引用。

函数	操作
<code>va_start(argptr, parm)</code>	使用 <code>argptr</code> (类型为 <code>va_list</code> , 见下文) 对参数列表和第二个参数 <code>parm</code> 的名称进行初始化。
<code>va_arg(argptr, type)</code>	返回参数表中的下一个参数, 它必须包含所指定的 <code>type</code> 。
<code>va_end(argptr)</code>	终止参数的读取。

为了读取一个可变长度参数的列表, 必须首先声明一个参数指针。指针是一个类型为 `va_list` 的变量, 此类型在文件 `stdarg.h` 中定义。一旦声明了参数指针, 就可将其用于宏中的 `argptr` 域。

```
va_list argptr;
```

以下语句定义了一个函数, 该函数可以取任意个数的整型参数, 然后打印输出每一个参数和它们的和。此代码要求所有的参数必须是整数类型:

```
#include <stdio.h>
#include <stdarg.h>

void print_ints(int num_of_args, ...) {
    int total = 0, i;
    va_list ap;

    va_start(ap, num_of_args);
    while(num_of_args--) {
        i = va_arg(ap, int);
        printf("%d \n", i);
        total += i;
    }
    printf("Total is %d \n", total);
    va_end(ap);
}
```

如果你用以下语句来调用该函数:

```
print_ints(3, 20, 35, 15);
```

就可得到如下输出:

```
20
35
15
Total is 70.
```

第十六章

I /O 库类与对象

在第四章中,我们介绍了标准 C++ 对象 `cin` 和 `cout`。但是,C++ 库中还有许多第四章中未描述过的对象或类。你可以象指定文件的读写那样来指定大量的 I /O 格式。

ANSI C++ 的最新规范为支持 `cin` 和 `cout` 及相关对象提供了新版本的库类。这些新版本的类与老版本的 C++ 库类的用法相同,只有一些小差别。考虑到简洁性,本章主要介绍老版本的 C++ 类,同时也注明了新老版本之间的区别。

本章的主要标题如下:

- I /O 库类的概述
- 与 I /O 类的通信
- I /O 操作符
- I /O 标志
- C++ 的新类
- C++ I /O 类与对象的总结

I /O 库类概述

C++ 的 I /O 库类性能优于 `printf` 和 `scanf`,但以此为代价的是系统变得较为复杂。你有可能会混淆各类之间的层次。为此,本章尽量写得简明扼要,主要介绍基本的函数与使用技巧。

在最初,你是由 `cin` 和 `cout` 的收发数据而开始用到 I /O 类的。例如,以下代码

打印输出一条提示信息,然后接收输入:

```
#include <iostream, h>
//...
cout << "Enter the value of x. ";
cin >> x;
```

为了实现前面要求的功能,不必使程序复杂化。然而,你却可以通过几种途径扩展 cin 和 cout 及相关类的功能与用途。

首先,你可以扩展自己定义的类,使它们能够发送和接收数据。cin 与 cout 分别是 istream 与 ostream 类的对象。通过定义正确的运算符函数,你可将自己的类与标准的 I/O 类一同使用。例如:

```
MyClass obj;
cin >> obj;    // 从键盘初始化对象。
```

下一节,“与 I/O 类的通信”,介绍了如何编写实现上述代码的运算符函数。

除了简单地收发数据,有时候你也想要定义一个特定的格式,如十六进制或八进制。可通过使用 I/O 操作符来达到这一要求,由移位运算符(<< 和 >>)把 I/O 操作符与 I/O 对象结合起来。例如,下列语句打印输出 n 的十六进制值,然后中止该行,由 endl 打印一新行:

```
#include <iostream, h>
//...
int n = 255;
cout << hex << n << endl;
```

程序输出 255 的十六进制表示——字符串“ff”。“I/O 操作符”一节介绍了所有的操作符,如 hex 和 endl。

为了得到更全面的控制,你可调用成员函数 setf 来设置格式标志符。基类 ios 给许多位标志符常量定义了特定的含意。例如,以下程序以大写字母显示十六进制数,并在数的前面加上表示十六进制的前缀——0X,程序的运行结果为打印字符串“0XFE”:

```
#include <iostream, h>
//...
cout.setf(ios::showbase | ios::uppercase);
int n = 255;
```

```
cout << hex << n << endl;
```

除了 `setf` 以外, I/O 类还支持许多成员函数。在“I/O 类与对象的总结”一节中将介绍它们。该章节总结了 `istream` 类和 `ostream` 类, 以及其他一些相关类:

- `fstream`, 文件输入/输出类
- `ifstream`, 从文件输入的类
- `istrstream`, 从字符串输入的类
- `ofstream`, 向文件输出的类
- `ostrstream`, 向字符串输出的类
- `strstream`, 字符串输入/输出类

与 I/O 类的通信

与文件 `stdio.h` 中的函数相比, I/O 类的最大优点是它们的可扩展性。简单地说, 就是你可以将 I/O 流操作作用于你自己所定义的类的对象。此优点很明显, 因为你不能自己编写一个新的类, 并为这个类发明一个格式符, 然后希望 `printf` 能够理解该格式。

然而, 却可以将你自己的对象读取或发送到 `cin` 和 `cout`。可使用那些用于输入输出原始数据的流运算符: `<<` 和 `>>` 来达到该目的。并且还可以完全控制类的数据格式。

扩展输出流移位符(<<)

为了用 `cout` 或 `cerr` 输出你自己定义的类, 就要编写一个 `operator<<` 函数, 去定义你的类如何与 `ostream` 类通信。该函数有以下的原型, 你须将它添加到你自己的类声明中去:

```
friend ostream& operator<<(ostream&, class &);
```

正如语法所显示的, 除了 `class` 以外, 所有的都须按以上所示逐个键入, 而 `class` 被替换为你自己的类的名称。由于左移运算符本身并非是一个类, 所以必须以友元 (`friend`) 函数的形式添加。(这就是为什么存在关键字 `friend` 来支持上面所示的运算符函数。)

该函数的定义有以下形式,你可用任意名称替代 `arg1` 和 `arg2`:

```
ostream& operator<<(ostream& arg1, class & arg2) {
    statements
    return arg1;
}
```

注意,此函数返回一个对 `ostream` 对象的引用。这就使以下表达式返回一个对 `cout` 本身的引用:

```
cout << obj
```

顺延之,就可以有以下的语句:

```
cout << obj << endl;
```

下面是一个简单的类的示例,该例与 `ostream` 通信。(它缺少 `istream` 类的右移运算符;这将在下一小节加上。)

```
#include <iostream.h>

class CPnt {
public:
    long x, y;
    CPnt() {x=y=0;} // default constructor
    CPnt(long newx, long newy) {x = newx; y = newy;}
    friend ostream& operator<<(ostream&, CPnt&);
};

ostream& operator<<(ostream& os, CPnt& pnt) {
    os << pnt.x << " " << pnt.y;
    return os;
}
```

有了这些声明,你就可写入如下的代码:

```
CPnt p1(2, 5);
cout << p1 << endl;
```

扩展输入流移位符(>>)

如同 `ostream` 类的<<左移位, `istream` 类的>>右移位。如果你想让自己的类与 `cin` 类通信,如同从文本文件中获得输入,就要在你的类声明中包含以下的声明:

```
friend istream& operator>>(istream&, class &);
```

如同 ostream 类的 operator<< 函数的原型一样,除了 class 以外,所有的都须按以上所示逐个键入,而 class 被替换为你自己的类的名称。

该函数的定义有以下形式,你可用任意名称替代 arg1 和 arg2:

```
istream& operator>>(istream& arg1, class & arg2) {
    statements
    return arg1;
}
```

此函数返回一个对 istream 对象的引用。这就使以下表达式返回一个对 cin 本身的引用:

```
cin >> obj1
```

顺延之,就可以有以下的语句:

```
cin >> obj1 >> obj2;
```

下面是一个完整的类的示例,该例用适当的移位运算符与 ostream 和 istream 通信。(<< 和 >>)

```
#include <iostream, h>

class CPnt {
public:
    long x, y;
    CPnt() {x=y=0;} // default constructor
    CPnt(long newx, long newy) {x = newx; y = newy; }
    friend ostream& operator<<(ostream&, CPnt&);
    friend istream& operator>>(istream&, CPnt&);
};

ostream& operator<<(ostream& os, CPnt& pnt) {
    os << pnt.x << " " << pnt.y;
    return os;
}

istream& operator>>(istream& is, CPnt& pnt) {
    is >> pnt.x >> pnt.y;
    return is;
}
```

其实,这些函数的一个重要之处是<<和>>运算符的兼容性。说它重要是因为 istream 和 ostream 运算符要被文件流类(ifstream 和 ofstream)所继承,故运算符函数需要适用于文本文件的读写。所以输出函数写数据的格式必须是输入函数可读的格式。

上述的代码提供了此种兼容性。在输出时, pnt. x 和 pnt. y 以空格符分隔; istream 类能够正确读取以这种格式分隔的数字。(不用空格符的分隔符号,如逗号,就会产生问题。)例如,该函数以如下格式输出一个点的坐标(25, 1025):

```
25 1025
```

● 注意

由于文本字符串通常有一些分隔限定,所以在对文本字符串进行读写操作时会产生问题。(如你用>>输入一个字符串变量时,只读入第一个空格符以前的字符。)一个简单的解决办法是以文本的独立行的形式读入字符串,参见 istream 类中的 getline 函数。

I/O 操作符

I/O 操作符是影响数据流的预定义对象——或是通过改变数据的格式,或是通过附加一个终止符。表 16-1 列出了 I/O 操作符。

表 16-1 I/O 操作符

I/O 操作符	描述
dec	将整数格式转为十进制
endl	添加一个换行
ends	添加一个 null
flush	刷新流,将缓存的数据送到物理设备
hex	将整数格式转为十六进制
oct	将整数格式转为八进制

例如,以下代码以十六进制打印输出 h,以八进制打印输出 o,然后回车换行:

```
#include <iostream, h>
int h, o;
//...
cout << hex << h << " " << oct << o << endl;
```

记住,左移位运算符(<<)把符号左面的与符号右面的关联起来,返回一个对流对象的引用——此处为 cout。以上代码的最后一行等效于以下语句:

```
cout << hex;    //切换到十六进制
cout << h;      //打印输出 h
cout << " ";    //打印" "
cout << oct;    //切换到八进制
cout << o;      //打印输出 o
cout << endl;   //打印新行
```

I/O 操作符的用法类似于输入对象。例如,以下语句以十六进制和八进制格式,通过键盘输入给 h 和 o 赋值:

```
cin >> hex >> n >> oct >> o;
```

I/O 标志符

每一个流对象都有自己内部的标志符设定。“标志”是一个有预定义意义的位。)这些标志中的任意一个都可由成员函数 setf 和 unsetf 来打开或关闭。表 16-2 列出了 I/O 标志:

表 16-2 I/O 标志

标志	描述
dec	为整数取十进制基底
fixed	以标准标注显示浮点数,如,123.5
hex	为整数取十六进制基底
internal	若域的宽度比输出的位数长,就在符号或基底前缀与数据之间填补字符
left	若域的宽度比输出的位数长,左对齐有放字符
oct	为整数取八进制基底
right	若域的宽度比输出的位数长,右对齐有放字符
scientific	以科学计数法显示浮点数,如,1.235e2
showbase	在输出十六进制和八进制数时显示前缀
showpoint	输出浮点数时显示小数点

showpos	正数时显示“+”
skipws	输入时跳过空格
stdio	每次输出之后清除 stdout, stderr
unitbuf	每次输出之后刷新所有的流
uppercase	大写的十六进制输出, 包括十六进制前缀 X 和指数标志 E

所有标志都是 `ios` 类的数据成员。设置标志时, 使用前缀 `ios::` 并调用任一流对象的 `setf` 函数。例如:

```
#include <iostream.h>
//...
cout.setf(ios::scientific);
cout.setf(ios::showpos);
cout << 1234.5 << endl;      // 输出 +1.234500e003
```

由于标志表示了各自的比特位, 故可用 OR 运算符(`|`)把标志结合起来。例如:

```
cout.setf(ios::scientific | ios::showpos);
```

你可调用成员函数 `unsetf` 来关闭任一条件。例如:

```
cout.unsetf(ios::showpos);    // 取消 showpos
```

所有的流类和流对象都支持三个有关标志设置的函数: `setf`、`unsetf` 和 `flags`。用后者可截取当前的标志设置, 也可同时设置所有的标志位。

记住, 在你未改变它们之前, 一个给定流的标志设置是固定不变的; 如果你用 `hex` 来以十六进制输出一个数到 `cout`, 随后的所有数都将以十六进制格式输出, 除非你又重置格式为 `oct` 或 `dec`。

C++ 的新类

如果你很乐于使用本章所介绍的基本 I/O 类, 那么请跳过这一小节。忽略本节是一件令人愉快的事, 然而, ANSI 委员会却重新为 I/O 类制定了标准并作了一些细微的修改。

这些修改在某种意义上导致了 I/O 类的两重性。一方面, 老版本的类和对象——正是本章所集中论述的是从向前的兼容性考虑, 继续受到编译器的支持; 另一方面, 从 ANSI 的兼容性考虑, 编译器又必须支持新版本的类和对象。或许在未来的某

一天,老版本的类最终将被抛弃,但这毕竟是很久以后的事了。

● 注意

如果你对老版本的 I/O 类很满意,完全可不受影响地跳过这一节。

通过以下两个主要步骤,你可将你的代码转换为新版本的 I/O 类:

1. 将包含指令 `#include <iostream.h>` 转换为 `#include <iostream>`。后一版本不含有文件扩展名,因为它并不指向一个实际存在的文件。(这是 `#include` 指令的新的使用方式。)
2. 将类和对象的名称从 `std` 处导入:

```
using namespace std;
```

第二步,向你的程序中加入 `using` 语句。它表明新类型的类的名称被放入一个独立的名字域,以 `std` 命名。也可跳过这一步,但如果这样做的话,就必须在 I/O 类的引用或预定义的对象前加前缀 `std::`,例如:

```
std::cout << "Enter a number please = >";  
std::in >> n;
```

一旦做了以上的修改,你就可以用与老版本的类相同的方式来使用新版本的类。换句话说,本章中的所有示例和语法都可用于新类。当然,也有一些区别,新类支持更多的格式标志符和 I/O 操作符。在函数运行方面也有一些细小区别。

本章的余下部分包含了一些有关调用成员函数差别的注释。在通常情况下,与老版本的类相比,新类有一些限制。然而,新类有许多此处并未介绍的新的兼容性。例如,除了 `cin`、`cout` 和相关的类,新版本的类支持以宽字符格式读写的 `wcin` 和 `wout`。有关细节参见个人的编译器文档。

C++ I/O 类和对象的总结

本节总结了 I/O 类和预定义的流对象,如 `cin` 和 `cout`。除了这里介绍的类以外, I/O 类继承还包含了一些抽象类,如 `ios` 和 `iostream`,它们并不被直接使用。

cerr

控制台出错对象

预定义的 `cerr` 对象("console error")提供了对标准错误输出的序贯文本访问。该流与显示器相关联,通常等同于标准输出(`cout`)。二者之间的区别是,尽管标准输

出可被重定向为一个文件,而向 `cerr` 传递的信息始终在显示器上输出。

`cerr` 是 `ostream` 类的一个实例。有关 `cerr` 可调用的函数清单,参见 `ostream` 类。

例如,以下代码向显示器发送一条出错信息:

```
#include <iostream.h>
//...
cerr << "You committed an error violation, code ";
cerr << error_number << endl;
```

不同于 `clog` 流,`cerr` 流直接向显示器发送非缓冲的输出。不过,`clog` 和 `cerr` 在许多方面是相同的。

cin

控制台输入对象

预定义的 `cin` 对象("console input")提供了对标准输入的序贯文本访问。该流通常是与键盘相关联。在某些系统中,如 UNIX, DOS, 或 Windows, 标准输入可能被重定向为一个文件。当被重定向后,`cin` 就被当作文本文件了。

`cin` 是 `istream` 类的一个实例。有关 `cin` 可调用的函数清单,参见 `istream` 类。

`cin` 的最常用方法是使用右移运算符(`>>`),即 `istream` 类从运算符中获取输入。例如,以下代码将随后的三个数字输入赋值给变量 `a`, `b`, 和 `c`:

```
#include <iostream.h>
int a, b, c;
//...
cin >> a >> b >> c;
```

假设用户从键盘输入了以下三个数:

3 4 5

在此情况下,程序将数 3, 4, 和 5 分别赋值给 `a`, `b`, 和 `c`。如同所有的 `istream` 实例,以这种方法输入的数字必须用一个或多个空白字符分隔,如空格符,制表符,和换行符。使用其他的分隔符,如逗号(,)也会出错。

一次输入多个数字的替代方法是调用 `getline` 函数,它以字符串的形式读入一整行的输入。然后你自己来解析该字符串,变成所需的数据(参见第十五章的字符串函

数。):

```
char input_string[81];  
cin.getline(input_string, 81);
```

clog

被缓冲的出错对象

预定义的 `clog` 对象 (“console log”) 提供了对显示器的序贯文本访问, 如同 `cout` 和 `cerr`。类似于 `cout` 和 `cerr`, `clog` 也是 `ostream` 类的一个实例。

`clog` 与 `cerr` 基本相同, 除了 `clog` 采用经缓冲的输出。`clog` 和 `cerr` 都是显示出错信息的好办法, 因为无论标准输出是否被重定向, 它们都向屏幕显示。

更多信息请参阅 `cerr` 和 `ostream` 类。

cout

控制台输出对象

预定义的 `cout` 对象 (“console output”) 提供了对标准输出的序贯文本访问。该流通常是与显示器相关联。在某些系统中, 如 UNIX, DOS, 或 Windows, 标准输出可能被重定向为一个文件。当被重定向后, `cout` 就被当作文本文件了。

`cout` 是 `ostream` 类的一个实例。有关 `cout` 可调用的函数清单, 参见 `ostream` 类。

`cout` 的最常用方法是使用左移运算符 (`<<`), 即 `ostream` 对象向运算符发送输出。例如, 以下代码输出两个数, `a` 和 `b`, 以及其它文字:

```
#include <iostream, h>  
int a, b;  
//...  
cout << "The value of a is " << a << endl;  
cout << "The value of b is " << b << endl;
```

`ostream` 类的成员函数对修改输出格式的细节很有用。例如, 你可指定最小输出字长, 并指定下一个数是左对齐还是右对齐。

fstream

输入 / 输出文件类

`fstream` 类支持流的输入和输出操作。你可以用输入或输出方式打开一个流, 也

可同时用两种方式打开一个流。(为了避免出错,对同一个文件从输入操作切换到输出操作,或从输出操作切换到输入操作之前,要刷新缓冲区。)

`fstream` 类的构造函数有以下几种声明格式。注意,在使用 `fstream` 类之前,必须包含头文件 `fstream.h`。当包含了该文件以后,在 `iostream.h` 中的所有声明也将被读入。(可同时包含 `iostream.h` 和 `fstream.h`,但这是多余的。)

```
#include <fstream.h>

fstream();
fstream(char * filename, int mode, int access = filebuf::openprot);
fstream(filedesc fd);
fstream(diedesc fd, char * pch, int nLength);
```

此处列出的第一个构造函数创建了一个空的对象,在调用了对象的 `open` 函数之后,它必须被打开。第二个构造函数是最常用的,它创建了一个 `fstream` 对象,并以指定的参数打开它。其中,最后一个参数 `access` 是可选项,它的缺省值是 `filebuf::openprot`,此为文件的普通访问模式(有关其他访问模式,参阅个人的编译器文档)。

在新版本的 I/O 类中,没有 `access` 参数,缺省是输入/输出文本模式(`ios::in | ios::out`)。

例如,以下代码以二进制模式初始化输入/输出流 `fio`:

```
#include <fstream.h>
//...
fstream fio(filename, ios::in | ios::out | ios::binary);
```

由于所有模式设置都是标志值(换言之,比特位设置),你就可用比特位的或(OR)运算符(`|`)。有关模式设置的完整列表参见下一节的表 16-3。

第三个和第四个构造函数用到了文件描述字,它是操作系统所用的低级句柄。第四个构造函数指定了一个缓冲;`pch` 的 `NULL` 值可禁止缓冲。

有关 `fstream` 的成员函数列表,参见 `ifstream`, `ofstream`, `istream`, 和 `ostream`。

ifstream

输入文件类

`ifstream` 类提供了以流的方式打开一个文件的功能。你可象从 `cin` 读取数据那样,从文本文件中读入数据。另外,可用 `ifstream` 读二进制文件。

ifstream 类的构造函数有以下的声明格式。注意,在使用 ifstream 类之前,要包含头文件 fstream.h。当包含了该文件以后,在 iostream.h 中的所有声明也将被读入。(可同时包含 iostream.h 和 fstream.h,但这是多余的。)

```
#include <fstream.h>

fstream();
fstream(char * filename, = ios::in, int mode, int access = filebuf::openprot);
fstream(filedesc fd);
fstream(tiledesc fd, char * pch, int nLength);
```

此处列出的第一个构造函数创建了一个空的对象,在调用了对象的 open 函数之后,它必须被打开。第二个构造函数是最常用的,它创建了一个 ifstream 对象,并以指定的参数打开它。其中,最后一个参数,access,是可选项,它的缺省值是 filebuf::openprot,此为文件的普通访问模式(有关其他访问模式,参阅个人的编译器文档)。

在新版本的 I/O 类中,没有 access 参数。

文本模式输入(ios::in)是缺省值。例如,以下语句创建了一个 ifstream 对象 fin,它被用来以文本模式从文件 c:\stuff.txt 读取数据:

```
#include <fstream.h>

ifstream fin("c:\\stuff.txt");
```

为了以二进制模式打开一个文件用于读,就得指定第二个参数为 ios::binary 标志。该标志应该用比特位或(OR)运算符(|)与 ios::in 合并起来。例如:

```
ifstream fin("c:\\stuff.dat", ios::in | ios::binary);
```

第三个和第四个构造函数用到了文件描述字,它是操作系统所用的低级句柄。第四个构造函数指定了一个缓冲;pch 的 NULL 值可去除缓冲区。

表 16-3 列出了参数 mode 常用的标志字。

表 16-3 模式设置值

模式标志字	描述
ios::app	打开一个文件流,追加数据——写在文件末尾

<code>ios::ate</code>	定位到文件的末尾
<code>ios::binary</code>	以二进制方式打开文件(不进行换行符的转换)
<code>ios::in</code>	为输入操作打开
<code>ios::nocreate</code>	如果文件不存在,则打开失败
<code>ios::noreplace</code>	如果文件已经存在,则打开失败
<code>ios::out</code>	为输出操作打开
<code>ios::trunc</code>	如果文件已经存在,则删除原先的内容并作为新文件打开

在新版本的 I/O 库类中,不支持 `ios::nocreate` 和 `ios::noreplace` 标志。

如果构造函数在打开一个文件流时失败, `ifstream` 对象的值就为零。即使文件不存在,构造函数也会打开一个流(打开一个零字节长的流),除非你指定了 `ios::nocreate`。在此情况下,文件必须已经存在,否则构造函数就会失败。你可在打开一个流的时候测试一下该对象,验证是否可用。例如:

```
ifstream("c:\stuff.dat", ios::in | ios::nocreate);
if (! ifstream)
    // File not opened; print error.
```

`ifstream` 类继承了所有的 `istream` 类的成员函数。另外,它还支持表 16-4 中的函数。`seekg`, `tellg`, 和 `read` 是在 `ostream` 类中声明的,但是它们实际上并不可用,除非有文件和字符串缓冲。

表 16-4 ifstream 类的成员函数

成员函数	描述
<code>close()</code>	关闭一个流
<code>gcount()</code>	返回最后一次输入操作所读入的字符个数
<code>open(fname, mode)</code>	以 <code>ifstream</code> 的构造函数的同样的参数打开流
<code>read(buf, num)</code>	读入 <code>num</code> 个字符或到 EOF 为止,返回一个指向当前输入流的引用。 <code>buf</code> 是一个 <code>char</code> 数组。(也可是 <code>signed char</code> 或 <code>unsigned char</code> 数组。)可用 <code>gcount</code> 来检查实际输入的字符个数。

`seekg(offset, org)`
`seekg(position)`

前者是相对于输入流中的三个位置之一 `org:ios::beg`, `ios::cur` 或 `ios::end` 移动 `offset` 个字节。后者将指针移到 `position`。参数 `offset` 和 `position` 的类型为 `streamoff` 和 `streampos`, 它们在文件 `iostream.h` 中定义。

`tellg()`

返回流的当前位置, 返回值为 `iostream.h` 中所定义的 `streampos` 类型。

通过打开一个 `ifstream` 类的实例, 并象从 `cin` 中读取数据那样, 就可实现对文本文件的操作。例如, 以下源程序打印输出一指定的文本文件的类型, 并在每一行前打印字符“= >”:

```
#include <iostream, h>
#include <fstream, h>

int main() {
    char s[81];
    cout << "Enter the name of the file: ";
    cin >> s;
    ifstream fin(s, ios::in | ios::nocreate);
    if (!fin) {
        cout << "File could not be opened.";
        return 1;
    }
    while (!fin.eof()) {
        fin.getline(s, 81);
        cout << "= >" << s << endl;
    }
    return 0;
}
```

你可用相似的程序来读取二进制文件。二进制模式与文本模式的区别在于, 当你读取二进制文件时, 换行符并不被转换; 除此之外, 两种模式完全一样。另外, 二进制文件通常用 `read` 函数来读取, 该函数直接读入字节而非将它们转换成文本。例如, 有一通过以指定的二进制输入模式打开一个文件的程序:

```
#include <fstream, h>
//...
ifstream fbin(fname, ios::in | ios::binary);
```

`read` 函数将字节直接读入到一个 `char`, `unsigned char`, 或 `signed char` 数组。可调

用 `gcount` 函数来检查实际读入的字节数。如果 `read` 函数读入的字节数少于所要求的,则表明已到达了文件的末尾。下面是程序中用 `gcount` 检测文件尾的语句:

```
char dat[16];  
//...  
do {  
    fbin.read(dat, 16);  
    n = fbin.gcount();  
    print_data(dat, n);  
} while (n == 16);
```

该代码假设 `print_data` 是程序中所定义的一个函数,它最多输出十六字节。

istream

输入类

`istream` 类定义了对象 `cin` 所支持的函数和操作。另外,由于 `istream` 是 `ifstream` 和 `istrstream` 的基类,所以,这些类都继承了 `istream` 的所有功能。

表 16-5 总结了通用的 I/O 成员函数。大多数函数处理从基类——`iostream` 所继承的状态和格式标志。注意,等号(=)表示此参数是缺省值;可在函数调用中省去该参数。

表 16-5 通用的 I/O 成员函数

成员函数	描述
<code>bad()</code>	如果发生致命错误,则返回非零值
<code>clear(flags = 0)</code>	若未指定参数,则清除格式标志;否则,将标志设定为给定值
<code>eof()</code>	如果到达文件末尾,返回非零值
<code>fail()</code>	如果发生错误,则返回非零值
<code>flags = flags(f)</code>	第一种表示式返回当前的格式标志设置。第二表示方式把 <code>long</code> 类型的变量 <code>f</code> 置为新标志设置并返回前一个标志。其类型为 <code>long</code>
<code>good()</code>	如果没有错误发生,返回非零值
<code>rdstateus()</code>	返回最后一次 I/O 操作的状态,状态值见下表

<code>setf(flags)</code>	第一种用法,打开那些在给定的 <i>flags</i> 中被标记的标志。
<code>setf(flags1 , flag2)</code>	第二种用法打开 <i>flags 1</i> 里的标志,但只影响那些同样被置位的 <i>flag2</i> 里的相应标志。这两种用法都以 <i>long</i> 类型返回旧的设置。
<code>sync_with_stdio()</code>	使 I/O 类能够和 <code>stdio.h</code> 中的函数同时使用

`rdstate` 函数返回最后一次 I/O 操作的状态,可能值如下:

Rdstate 值	描述
<code>ios::goodbit</code>	无错误
<code>ios::eofbit</code>	到达文件尾(EOF)
<code>ios::failbit</code>	发生非致命错误
<code>ios::badbit</code>	发生致命错误

例如,以下语句测试一个读操作的结果:

```
cin.read(buf, 10);
if (cin.rdstate() == ios::eofbit)
    goto stop_reading;
```

表 16-6 总结了输入数据的 `istream` 成员函数。(其中的一些仅与二进制文件有关的函数,如 `gcount`,`read`,和 `seekg`,是 `ifstream` 的成员函数。)注意,等号(=)表示此参数是缺省值;可在函数调用中省去该参数。

表 16-6 `istream` 类中的输入函数

成员函数	描述
<code>eatwhite()</code>	跳过输入中的连续空格
<code>get(ch)</code> <code>get(buf , num , delim = ' \n')</code> <code>get(buf , delim = ' \n')</code>	第一种用法读一个字符到 <i>ch</i> 中。第二种用法。读入 <i>num</i> 个字符,将结果拷贝到一个类型为 <code>char *</code> 的字符串 <i>buf</i> 。(也可是一个类型为 <code>signed char</code> 或 <code>unsigned char</code> 的数组。)第三种用法把字符读入到一个流对象 <i>buf</i> 中。以上三种用法均返回对当前输入字符流的引用。
<code>getline</code> <code>(buf , num , delim = ' \n')</code>	读入 <i>num-1</i> 个字符,将结果拷贝到一个类型为 <code>char *</code> 的字符串 <i>buf</i> 中,以 <code>null</code> 终止串。分隔符本身被去除。返回一个对当前输入流的引用

<code>ignore(num = 1, delim = EOF)</code>	跳过输入流中的 <code>num</code> 个字符, 如果遇到 <code>delim</code> 则停止。 返回一个对当前输入流的引用
<code>peek()</code>	返回流中的下一个字符, 但不将其从流中移去。如果到达文件尾, 则返回 <code>EOF</code>

可用 `istream` 类以及其派生类 `ifstream` 和 `istrstream` 类的成员函数来实现 `stdio.h` 中的函数的所有输入操作。最基本的键盘输入是将 `cin` 同右移运算符(`>>`)一起使用。通过`>>`, 你可自由地混合使用 `cin` 的各个成员函数。例如, 以下代码提示输入数据, 打印输出结果, 并在继续运行程序之前, 等候用户键入回车键:

```
int num = 0;
char inpstr[81];
cout << "Enter a hex number: ";
cin.setf(ios::hex);
cin >> num;
cout << "Decimal equiv. is: " << num << endl;
cin.getline(inpstr, 81); // Eat current newline.
cin.getline(inpstr, 81); // Wait for next Enter.
```

此代码的一个细节是, 为了达到预期的效果, 必须两次调用 `getline`。一次是在用户输入了 `num` 并键入回车后读取一个新行; 一次是等待第二次键入回车。一个替代方法如下:

```
cin.ignore(); // Discard next character(Enter).
cin.getline(inpstr, 81); // Wait for next Enter.
```

注意, 调用 `getline` 函数与使用右移运算符有一个重要的区别: 前者读入所有字符串数据, 直到换行符; 后者在下一个空白符前停止读入数据。

```
cin.getline(inpstr, 81); // Read to end of line.
cin >> inpstr; // Read to next whitespace.
```

istrstream

输入字符串类

`istrstream` 类使你能够从一个字符串读取数据到内存, 就如同从文件或控制台读取一样。其功能类似于 `stdio.h` 中的 `sscanf`。

`istrstream` 构造函数有以下的声明格式。注意, 使用 `istrstream` 或相关的类, 必须

包含头文件 `strstream.h`。

```
#include <strstream.h>

istream();
istream(char * pch, int length);
```

第一个构造函数创建了一个使用内部字符串缓冲的流,开始时,缓冲是空的,但会根据需要进行动态地扩展和移动。大多数操作中所使用的第二个构造函数采用固定长度的字符串区域。例如:

```
#include <strstream.h>
#include <string.h>
//...
char s[] = "10 25 76";
int x, y, z;
istream str_in(s, strlen(s));
str_in >> x >> y >> z;
cout << "x is: " << x << endl;
cout << "y is: " << y << endl;
cout << "z is: " << z << endl;
```

`istream` 类支持 `istream` 类的所有成员函数,包括 `read`, `seekg`, 和 `tellg` 函数(参见 `istream`)。该类也支持 `str` 函数(参见 `strstream`)。

ofstream

输出文件类

`ofstream` 类提供了以流的形式打开一个输出文件的功能。如同向 `cout` 写数据那样,你可向文本文件写数据。另外,可用 `ofstream` 写二进制文件。

`ofstream` 构造函数有以下的声明格式。注意,在使用 `ofstream` 类之前,必须包含头文件 `fstream.h`。当包含了该文件以后,在 `iostream.h` 中的所有声明也将被读入。(可同时包含 `iostream.h` 和 `fstream.h`,但这是多余的。)

```
#include <fstream.h>

ofstream();
ofstream(char * filename, int mode = ios::out, int access = filebuf::openprot);
ofstream(filedesc fd);
ofstream(filedesc fd, char * pch, int nLength);
```

在新版本的 I/O 类中,没有 access 参数。

文本模式输出 (ios::out) 是缺省值。例如,以下语句创建了一个 ifstream 对象 fin,它被用来以文本模式向文件 c:\stuff.txt 输出数据:

```
#include <fstream.h>

ofstream fout("c:\\stuff.txt");
```

为了以二进制模式打开一个文件用于写,就得指定第二个参数为 ios::binary 标志。该标志应该用比特位或(OR)运算符(|)与 ios::out 合并起来。例如:

```
ofstream fout("c:\\me.bin", ios::out | ios::binary);
```

有关 mode 参数可接受的值,参见 ifstream 目录下的表 16-3。

如果构造函数在打开一个可用文件流时失败,ofstream 对象的值就为零。你可通过检测一个对象来看看文件是否被打开。例如:

```
ofstream("c:\\stuff.dat");
if (!ofstream)
    // File not opened; print error.
```

第三个和第四个构造函数用到了文件描述字,它是操作系统所用的低级句柄。第四个构造函数指定了一个缓冲;pch 的 NULL 值可去除缓冲区。

ofstream 类继承了所有的 ostream 类的成员函数。另外,它支持表 16-7 列出的函数。(注意,seekp, tellp, 和 write 函数实际上是在 ostream 类中声明的,但除非是用于文件和字符串缓冲,一般它们并不能被调用。)

表 16-7 ofstream 类的成员函数

成员函数	描述
close()	关闭一个流
open(fname, mode)	用 ofstream 构造函数中的相同参数打开流
seekp(offset, org)	第一种用法把输出指针相对于输出流中的三个位置之一 org:ios::beg, ios::cur 或 ios::end 移动 offset 个字节。第二种用法把指针移动到 position 位置。参数 offset 和 position 的类型分别为在文件 iostream.h 中所定义的 streamoff 和 streampos
seekp(position)	

<code>tellp()</code>	返回文件输出指针的位置。返回值的类型为在文件 <code>iostream.h</code> 中所定义的 <code>streampos</code>
<code>write(buf, num)</code>	写 <code>num</code> 个字节到 <code>char</code> 数组 <code>buf</code> 中 (<code>buf</code> 也可能是 <code>unsigned char</code> 或 <code>signed char</code> 数组)

一旦一个 `ofstream` 对象被打开,你可把它当作文本文件来写,就如同向 `cout` 写数据。例如,以下代码打开两个文件,从输入文件中读入一个整数,然后把它写到输出文件:

```
#include <stdlib.h> // for exit function
#include <fstream.h>
//...
cout << "Enter input file name: ";
cin >> ifname;
cout << "Enter output file name: ";
cin >> ofname;

ifstream fin(ifname, ios::in, ios::nocreate);
ofstream fout(ofname);
if (! fin || ! fout) {
    cout << "Could not open one of the files. \n";
    exit(1);
}

int n;
fin >> n;           // Read integer from input file.
fout << n;           // Print integer to output file.
```

注意,移位运算符(`<<`和`>>`)以文本形式读写所有的数据;例如,以数字串的形式输出 `n`。因此把流当作文本文件来处理是比较妥当的。

当对二进制文件进行操作时,通常需要直接写数据。这就要用到 `write` 函数;它在写数据时不进行任何转换。例如,以下语句输出一个整数 `n` 到输出文件:

```
ofstream fout(ofname, ios::out | ios::binary);
int n = 5;
fout.write((char *) &n, sizeof(n)); // Write n.
```

函数 `write` 所输出的数据以后可由 `read` 函数从该流中读取。

ostream

输出类

ostream 类定义了 cout, cerr, 和 clog 对象所支持的函数与操作。另外, ostream 是 ofstream 和 ostrstream 的基类, 所以, 这些类继承了 ostream 类的所有功能。

ostream 类继承了处理状态字和格式标志的常用函数。有关这些函数, 参见 istream 一节的表 16-5 通用的 I/O 成员函数:

```
bad
clear
eof
fail
flags
good
restate
setf
sync_with_stdio
unsetf
```

表 16-8 对 ostream 类支持的输出及格式化函数进行了总结。

表 16-8 ostream 类的输出及格式化函数

成员函数	描述
fill() fill(<i>ch</i>)	第一种用法是返回当前的填充字符(缺省为空格符);第二种用法是设置填充字符为 <i>ch</i> 并返回原先值。为了添加前导零, 应指定 '0' 为填充符
flush()	刷新输出流
precision() precision(<i>p</i>)	第一种用法是返回当前的浮点数精度;第二种用法是设置浮点数的精度, 并返回原先值。该值决定了下一输出操作所显示的最多数字个数, 按要求对浮点数进行截短
put(<i>ch</i>)	输出的下一个字符为 <i>ch</i> , 返回一个对输出流的引用
putback(<i>ch</i>)	将字符 <i>ch</i> 放回到输出流中, 并返回一个指向流的引用。 <i>ch</i> 必须是所写的最后一个字符, 否则会出错
width() width(<i>w</i>)	第一种用法是返回当前的位域宽度, 小于该设置的输出被填充字符所填满。(参见 <i>fill</i> 。)如果宽度为零, 不进行填充;第二种用法是设置宽度并返回原先值。该值决定了下一次输出操作的输出宽度

precision 和 width 函数为下一次输出操作设置了打印输出域的格式特性。并不一定要设置这些值,实际上,缺省设置恰好能够满足被输出数的打印宽度。虽然填充字符的缺省值——空格符通常是所需的,你也可自己设置一个填充符。指定‘0’作为填充字符以获取前导的零是一个特例。

例如,以下语句在输出域内(宽为六个字符)右对齐三个浮点数:

```
cout.width(6);  
cout << 3.4 << endl;  
cout.width(6);  
cout << 1.07 << endl;  
cout.width(6);  
cout << 125.44444 << endl;
```

执行该语句后,有以下的输出结果:

```
      3.4  
     1.07  
    125.444
```

注意,最后一个数超出了输出域宽度,因为数的缺省精度为六位数字(为了给小数点留一个位置,就共有七个字符)。可通过指定精度为 5 来强制截短,使其只输出五个数字:

```
cout.precision(5);  
cout.width(6);  
cout << 125.44444 << endl;
```

现在的结果如下:

```
      3.4  
     1.07  
    125.44
```

ostream

输出字符串类

ostream 类使你能够向内存中的一个字符串写数据,就如同向文件或控制台写一样。其功能类似于 stdio.h 中的 sprintf。

ostream 构造函数有以下的声明格式。注意,使用 ostream 或相关的类,必须包含头文件 ostream.h。


```
#include <strstream.h>
```

```
ostrstream();
```

```
ostrstream(char* pch, int length, int mode = ios::out);
```

第一个构造函数创建了一个使用内部字符串缓冲的流,开始时,缓冲是空的,但会根据需要进行动态的扩展和移动。大多数操作中所使用的第二个构造函数采用固定长度的字符串区域。例如:

```
#include <strstream.h>
```

```
//...
```

```
char s[255];
```

```
int x = 10;
```

```
ostrstream str_out(s, 255, ios::out);
```

```
str_out << "The value of x is " << x << endl << ends;
```

```
cout << s;
```

注意,必须用 ends 操作符来以 null 明确终止字符串。

ostrstream 类支持 ostream 类中的所有成员函数,包括 write, seekp, 和 tellp 函数。(参见 ofstream。)该类也支持 pcount 和 str 函数(参见 strstream)。

strstream

输入/输出字符串类

strstream 类以及它的相关类 istrstream 和 ostrstream 使你能够对内存中的字符串进行读写,就如同对文件或控制台进行读写操作。功能类似于 stdio.h 中的函数 sprintf 和 sscanf。

strstream 的构造函数有以下的声明格式。在使用 strstream 和它的相关类之前,必须包含头文件 strstream.h:

```
#include <strstream.h>
```

```
strstream();
```

```
strstream(char* pch, int length, int mode);
```

第一个构造函数创建了一个使用内部字符串缓冲的流,开始时,缓冲是空的,但会根据需要进行动态的扩展和移动。大多数操作中所使用的第二个构造函数采用固定长度的字符串区域。例如:

```
#include <strstream.h>
//...
char s[256];
int x = 10;
strstream str_out(s, 255);
str_out << "The value of x is " << x << endl << ends;
cout << s;
```

注意,必须用 ends 操作符来以 null 明确终止字符串。

strstream 类支持 istream 和 ostream 类中的所有成员函数,包括 read, write, gcount, seekg, seekp, tellg 和 tellp 函数。(参见 ifstream 和 ofstream。)该类同时也支持以下的函数,等号(=)表示参数的缺省值。

成员函数	描述
freeze(<i>f = true</i>)	如果输入的是参数的缺省值 true,则函数冻结字符串的长度和存储位置;如果输入 false,函数解除对字符串的约束。该操作只影响动态字符串缓冲。
pcount()	返回字符串缓冲区中当前存储的字符个数
rdbuf()	返回一个指向 strstreambuf 对象的指针,该对象可被用于字符串缓冲的低级操作
str()	返回一个指向字符串的类型为 char * 的指针。对于动态字符串,它有冻结字符串数据的效果

如果流使用内部的动态缓冲区(参见第一个构造函数),则调用 str 函数的结果是冻结了该缓冲区;换句话说,就是在以后的操作中,字符串的大小和位置保持固定。可调用参数为 false 的 freeze 函数来解冻该字符串,如以下语句所示:

```
#include <strstream.h>
#include <string.h>

strstream str_io;           // Create str_io with strstream()
str_io << "Here is some text";
char *p = str_io.str();    // Freeze str data.

char strdata[256];         // Store the data.
strcpy(strdata, 255, p);
//...
```

```
str_io.freeze(false);      // < == = UNFREEZE
//...
p = str_io.str();          // Get new value of str data.
```

如果你获得了串数据,并随后将其解冻,则最好像上面介绍的那样,把数据储存在好。此例中,当解冻了字符串以后,指针 `p` 就无效,除非再次调用 `str` 函数。

附录 A

C 与 C++ 的区别

在将 C 程序向 C++ 程序转换时,应该看一看下面这些 C++ 程序与 C 程序不同的地方并做相应的修改:

- 在声明时,如果声明的函数具有 void 类型之外的返回类型,则在 C++ 函数的实现中必须具有返回值。如果在函数的实现中没有值返回,C 可能会给出警告,但是在 C++ 中,这种情况是无法通过编译的。
- 在 C++ 中,必须对函数定义标题中的变量进行声明,而不能使用旧式 C 函数的语法(C 的头几个版本支持这种旧用法)。例如,下面的函数定义标题在 C 中是有效的:

```
void swap(a,b)
double * a;
double * b;
{
//...
```

但是上面的语法在 C++ 中是无效的,必须作如下修改:

```
void swap(double * a, double * b) ; //...
```

- C++ 引入了严格的类型检查。特别是,不能将一个地址或指针赋值给另外一个不同类型的指针,除非使用类型强制转换或者给一个 void 类型的指针赋值。在 C 中,在不同类型的指针间赋值只不过会给出警告;但是在 C++ 中,这可是一个严重的错误。强制类型转换可以解决这个问题:

```
void * vp;
int * ip;

ip = (int *)vp;    //需要进行数据转换
vp = ip;           //现在才可以将 int * 类型的指针赋值给 void 类型的指针。
```

- 在 C++ 中,函数在使用之前必须先进行声明。而在 C 中,可以无须事先进行函数声明。这时 C 假定函数的返回类型是 int 型并且函数的参数列表是不确

定的。当然,这种编程习惯并不好,最好不要使用。但是如果对这种默认的情况十分满意的话,C也可以对这种情况进行编译。C++则不允许这种情况。

- 在C++程序中也可能会出现一些C程序中所没有的名称的冲突。例如,在C中,可以在将某个全局名称用于结构或联合的名称之后(这样的名称称为“标签名称”,与使用typedef定义的名称的用法正好相反),再用它命名某个函数或变量。因为C++将所有的标识符放在同一个命名空间,所以C++不允许出现这种名称重复使用的情况。幸好,这种名称的冲突很少出现。

在使用C和C++源模块混合编程时还要注意另外一个陷阱。对于同一个符号,在生成.obj文件时C++生成一个完全不同的名称。C++的名称转换规则在名称中加入类型信息,这样.obj文件中就会包含类型信息。于是,连接程序就不会将声明为int型的变量MyVar与声明为float型的MyVar看作同一个符号。(这称为“类型安全连接”。)而且,C++的转换规则在应用时是大不相同的,所以不要徒劳地对符号名称做什么假设。

结果,在将C和C++模块进行连接时,它们可能不能识别彼此的符号。不过,C++提供了一个简单的解决方法:在C++中使用extern“C”声明来引入在C中定义的符号。extern“C”声明禁止C++将类型信息加入到名称中。

```
extern "C" {  
    long way_home;    //在C模块中定义了 way_home  
};
```

因为C++支持许多新的特征,在将C++程序向C程序转换时则更加困难。不过很少有人这么做。如果需要进行这样的转换,可能需要重写大量的程序。尽管C++对数据类型的使用要求更为严格,但是它也提供了C所没有的许多自由:

- 可以在任何地方进行变量的声明以及编写语句(可执行语句)。在C中,必须在函数或语句块的开始处对所有的局部变量进行声明。
- 可以使用任何有效的表达式对变量或对象进行初始化。当然也可以使用常量进行初始化。
- 在进行类(包括结构或联合)的声明时,类的名称自动变成可以自由使用的类型名。在将这些名称作为类型使用时无须使用class、struct或union对名称进行说明。在C中,使用名称为Record的结构进行声明及类型转换时,必须使用struct Record或者使用typedef创建类型名。由于C++向下兼容,所以它也支持诸如struct Record和class Record这样的用法,尽管在编写C++程序时无须这样使用。

如果编写既可以在C中编译又可以在C++中编译的程序(例如,为各种用户编写的源代码库),就必须同时遵守C和C++二者的严格规定。

附录 B

ANSI C++ 特征总结

经过多年的修改,ANSI 为 C++ 所制定的规则现在已经相当稳定。也就是说,现在已经有有了一个被广为接受的 C++ 版本,而且在近几年内不会对该版本进行什么大的修改。编译器供应商应该为该版本的 C++ 提供支持,程序员应该逐渐遵守这些规则。

ANSI 规则中的许多新特征都是对最初的 C++ 进行的扩充。也可以不使用这些新特征。这些新特征包括关键字 mutable 以及 explicit。

有一些修改的潜在作用很大。现在 ANSI 规则鼓励某些编程方法而抛弃其他一些编程方法。尽管在一段时间内不会放弃对老式编程方法的支持,但是有一些编程方法最终是要被淘汰掉的——这意味着编译器将给出警告并建议使用新的编程方法。迟早有一天会抛弃掉对这些旧方法的支持。不管愿不愿意,改变是必须的。但是最好还是现在就开始使用这些新特征,以免将来在别人后面。

ANSI 规则的主要的改变包括:

- 新式头文件
- ANSI 类型转换运算符
- 模板及异常处理
- 其它新的关键字
- if 语句中的变量的作用域
- 具有枚举类型的函数的重载
- 嵌套类的前向引用

头两个部分的头文件以及类型转换运算符表示对曾经常用的规则进行了修改。

新式头文件

一般说来, `#include` 指令是一个简单的函数:从某个头文件中进行读取。这个文件总是一个普通的磁盘文件,可以象其它文件一样进行读取。

新的方法可以包括标准库的虚头文件。使用这种方法,通过指定标准库函数所在库的名称,例如 `cstdlib`、`cstdio` 或 `cmath`, `#include` 指令可以加入标准库函数的声明。通过读取相应的磁盘文件或者通过其它方法对声明进行加载,编译器可以作出正确的响应。这个修改是为了提高以后程序的效率。

实际应用中,可以将下面的语句:

```
#include <math.h>
```

替换为:

```
#include <cmath>
```

本书在样例程序中坚持使用老方法,因为对 C++ 程序员来说,老方法更为熟悉。但是在第十五章中为每个库函数同时列出了这两种声明方法。

ANSI 类型转换运算符

C 语言对所有情况都使用一种类型转换运算符,在 C++ 的早期版本中也支持同样的运算符,没有进行什么大的修改。(主要变化是对指针转换的要求更为严格。)下面是使用老式 C 运算符对 `char *` 进行类型转换的例子:

```
char * p = (char *) malloc(n);
```

尽管现在依旧支持这个运算符,甚至本书中也多次使用,但是 ANSI 已经明确表示不再支持这种运算符。ANSI C++ 提供了四个新的类型转换运算符:

```
const_cast  
dynamic_cast  
reinterpret_cast  
static_cast
```

这些运算符中,只有一个运算符——`dynamic_cast` 提供了新的功能。如果启动了支持运行时类型信息(RTTI)(详细内容请参看编译器手册),`dynamic_cast` 可以有

助于判断在运行时所指向的对象的确切类型。`dynamic_cast` 的用法与 `typeid` 运算符有关, `typeid` 运算符也是一个相对来说比较新的特征并且它也需要启动 RTTI 支持。

其它三个类型运算符与老式类型转换运算符具有相同的功能,但是每个运算符的目的都不同。这些运算符的好处在于它们在源程序中更为显眼,因为每个运算符都不相同,所以使得每行程序更为清晰。在实际应用时, `const_cast` 与 `reinterpret_cast` 在更容易出错的类型之间进行转换。通过使用这些关键字,可以在容易出错的程序中作出标记。

例如,在进行指针类型转换时需要十分小心。本部分开始的例子进行了指针类型的转换,这个例子可以重新编写如下:

```
char * p = reinterpret_cast<char * >(malloc(n));
```

关于每个运算符的详细内容,请参看第十二章。

模板与异常处理

模板与异常处理是 C++ 的两个主要特征。现在许多编译器都支持这两个特征,但是早期版本的 C++ 并没有这个特征。在 C++ 刚出现时,许多人要求将模板加入到 C++ 中。因为可以编写一个通用类或函数并可以在其中加入特殊的类型,所以模板技术可以使代码重复使用。但是令人惊奇的是,Microsoft 尽管是慢慢加入了这个功能但是最终还是满足了公众的要求。Microsoft 又加入了其它特征,例如异常处理以及更多地支持 DOS 和 Windows 的特征。但是 ANSI 规则中不包括大部分与平台有关的特征。

异常处理是用于响应运行时错误和其它事件的高级技术。它的用法大大优于老式 C 库函数 `raise` 和 `signal`。这两个函数只提供了处理有限的一些事件的方法。

公开发布的 ANSI 规则要求完全支持模板及异常处理。关于这些特征的详细内容,请参看关键字 `template` 以及 `try`、`throw` 和 `catch`。后三个关键字支持异常处理。可以在第十三章中看到所有这些关键字。

除了支持使用关键字 `template` 定义新的模板之外,ANSI 规则也列出了标准模板库,在这个标准模板库中包括许多通用的类。(可以使用这些类来生成所需要的任何类型的集合。)不幸的是,由于篇幅的限制,本书不能介绍标准模板库。标准模板库本身就是一个大的课题。

其它关键字

除了前面提到的关键字,ANSI 规则支持 `mutable`、`explicit` 和 `bool`:

- 关键字 `mutable` 改变成员的声明,甚至当成员是 `const` 对象的一部分时,也可以对成员进行修改。
- 关键字 `explicit` 阻止构造函数进行转换。
- 关键字 `bool` 定义一个数据类型,此数据类型只有两个值:真(`true`)和假(`false`)。

完全可以不使用 `mutable` 和 `explicit`。详细内容请参看第十三章。

`bool` 的使用与其它两个略有不同:它是一种处理真/假值十分有效的方法。许多年来,在 C 或 C++ 中存储真/假值必须将它存储在整型变量中。但是这种存储方式有许多问题:任何非零值都被视为“真”,尽管它可能是两个非零值进行位操作与(`&`)并得到一个零(即“假”)。而使用 `bool` 数据类型,所有的非零值都转换为 1(真);结果,两个真值的与(`&`)或或(`|`)运算——无论是位运算还是逻辑运算——都会得到一个正确的结果。

除了支持 `bool` 数据类型之外,ANSI C++ 添加了两个预定义的常量 `true` 与 `false`,分别等于 1 和 0。详细内容请参看第十章。

另一个早期 C++ 没有的 ANSI 数据类型是 `wchar_t`,它是宽字符数据格式。(请参看第十章。)因为这种格式为每个字符多分配一个字节,所有可以使用非欧洲字符。C++ 中引入这种类型说明国际市场的重要性越来越大。

if 语句中变量的作用范围

在 ANSI C++ 中,可以在 `if` 语句中声明一个变量。该变量的作用范围是整个 `if` 语句块。例如:

```
if(int j=1) |
    cout<<j<<endl;    //正确:定义了j
|

j=2;                    //错误!j超过了作用范围
```

上面的程序在 `if` 语句中定义了一个变量 `j`,而不是在 `if` 语句块中进行的定义。注意一个等号(=)表示将 1 赋给 `j`,而不是检测 `j` 是否等于 1。

这个特征最明显的用处是控制由 `dynamic_cast` 返回的指针的作用范围。在下面的程序中,在 `if` 语句中定义了 `pd`——所以,只有在 `dynamic_cast` 成功时它才可用。

```
if(D *pd=dynamic_cast<arg>){
    //仅当 arg 指向了 D 对象时 dynamic_cast 才成功返回
    //pd 可以在语句中安全使用
    :

    pd->func2();    //错误! pd 超出了作用范围
```

因为转换成功(因此返回一个非空值给 `pd`),所以在 `if` 语句块中使用 `pd` 是安全的。如果在 `if` 语句块之外使用 `pd`,在编译时就会产生错误。这样就可以在程序发布之前改正这个错误。这样的后果是,如果只在一处定义了 `pd`,编译器将阻止在 `pd` 作用范围之外使用它——因此也防止了在运行时出现错误。

关于 `dynamic_cast` 运算符的详细内容,请参看第十二章。

具有枚举类型的函数的重载

ANSI C++ 增强了声明为枚举类型的状态;特别是重载函数时,可以将 `enum` 类型与其它整数类型区分开。而在 C 和 C++ 的旧版本中,`enum` 类型与 `int` 类型在函数的参数声明时是可以互换的。下面例子说明了新特征:

```
enum suit {
    clubs, diamonds, hearts, spades
};

void func1(int n) {
    cout<<"inside func1(int)"<<endl;
}

void func1(enum card) {
    cout<<"inside func1(suit)"<<endl;
}
```

函数调用为:

```
func1(spades);
```

输出:

```
inside func1(suit)
```

嵌入类的前向引用

C 与 C++ 一直具有的一个特征是,可以在没有完全声明一个类(或者在 C 中,声明一个结构类型)之前对它进行引用。(详细内容请参看词汇表中的“前向引用”。)ANSI C++ 将这个功能扩展到嵌套类。例如:

```
class Chicken {  
  
    class Egg; //允许对 Egg 进行前向引用  
  
    int a, b;  
    Egg * p; //前向引用  
    class Egg { //嵌入类的声明  
        int n;  
    };  
  
};
```

也可以在类的外面对嵌套类进行声明,但是必须使用作用域辨别符(::)来说明。

```
class Chicken {  
  
    class Egg; //允许对 Egg 进行前向引用  
  
    int a, b;  
    Egg * p; //前向引用  
};  
  
class Chicken::Egg { //嵌入类的声明  
    int n;  
};
```

附录 C

标准异常

正如在第十三章中说明的那样,可以抛出并捕捉任何类型的异常。也可以捕捉 C++ 中定义的标准异常。在某些情况下会自动抛出这些异常,这样就不需要程序员自己使用 `throw` 语句来抛出异常。

C++ 标准异常基类的名称恰如其分: `exception`。该类声明了构造函数和析构函数,但是不包括数据成员。(详细内容请参看编译器手册。)这个基类有两个子类,每个子类都对应一组通用异常。

异常类	说明
<code>logic_error</code>	报告不能执行的操作。运行时不允许出现这种情况。
<code>runtime_error</code>	报告无效的操作或不正确的结果。这种异常的原因是操作的参数正确而结果不正确。

一般意义上讲,所有这些标准异常都是运行时出现的错误。(这些错误不是语法造成的,编译器无法找出它们。)但是, `logic_error` 异常的出现是因为试图——例如在运行时——进行某些不合逻辑的操作,甚至在进行这些操作之前就可能出现这种异常。例如,使用无效的参数进行函数调用。 `runtime_error` 异常包括那些失败的操作。

下面的表格列出了通用类型派生的特殊的异常类。

异常类	基类	说明
<code>domain_error</code>	<code>logic_error</code>	违反了域条件优先于操作的规则
<code>invalid_argument</code>	<code>logic_error</code>	试图使用错误的参数调用函数
<code>length_error</code>	<code>logic_error</code>	试图创建比最大长度 <code>size_t</code> 大的对象

<code>out_of_range</code>	<code>logic_error</code>	试图使用一个超过范围的参数调用函数
<code>bad_cast</code>	<code>logic_error</code>	错误地使用了 <code>dynamic_cast</code> ; (见第十二章)
<code>bad_typeid</code>	<code>logic_error</code>	在表达式标识类型中使用了错误的值
<code>range_error</code>	<code>runtime_error</code>	运算的结果不正确
<code>overflow_error</code>	<code>runtime_error</code>	运算的结果溢出(对应相应的数据类型来说结果太大)
<code>bad_alloc</code>	<code>runtime_error</code>	动态内存分配错误

附录 D

ASCII 字符代码

下面的表格列出了前 128 个 ASCII 字符代码以及它们的十进制值。127 以及从 0 到 25 的值为不可输出字符,它们在大多数个人计算机和网络,包括 IBM 兼容机中具有特殊的含义。表格列出了最常用的特殊代码,包括 ACK(接收)、BS(退格)、LF(换行)、FF(进纸)以及 CR(回车)。

00 NULL	26	51 3	77 M	103 g
01	27	52 4	78 N	104 h
02	28 FS	53 5	79 O	105 i
03	29 GS	54 6	80 P	106 j
04	30 RS	55 7	81 Q	107 k
05	31 US	56 8	82 R	108 l
06 ACK	32	57 9	83 S	109 m
07 BEL	(space)	58 :	84 T	110 n
08 BS	33 !	59 ;	85 U	111 o
09	34 "	60 <	86 V	112 p
10 LF	35 #	61 =	87 W	113 q
11	36 \$	62 >	88 X	114 r
12 FF	37 %	63 ?	89 Y	115 s
13 CR	38 &	64 @	90 Z	116 t
14	39 ,	65 A	91 [	117 u
15	40 (	66 B	92 \	118 v
16	41)	67 C	93]	119 w
17	42 *	68 D	94 ^	120 x
18	43 +	69 E	95 _	121 y
19	44 -	70 F	96 `	122 z
20	45	71 G	97 a	123 {
21 NAK	46 '	72 H	98 b	124
22 SYN	47 /	73 I	99 c	125 }
23	48 0	74 J	100 d	126 ~
24	49 1	75 K	101 e	127 DEL
25	50 2	76 L	102 f	

C++ 术语及概念词汇表

抽象类(abstract class)

抽象类是不能用于创建对象的类。但可以使用它创建一套相关的类族。例如,已经有了一个名为 CShape 的抽象类,在该类中定义了用于形状描述的常用函数。尽管类 CShape 不能直接用于创建对象,它却可以作为诸如 CSquare、CCircle、CTriangle 等类的基类。

用技术角度来说,任何包含一个或多个纯虚函数的类都可以称之为抽象类。纯虚函数在对其进行声明的类中没有函数定义,但是它可以由派生类来实现。详细内容请参看纯虚函数。

抽象数据类型(abstract data types)

面向对象的理论中容易混淆的术语之一就是抽象数据类型。规则制定者对它喜欢倍至;也有人不喜欢。

一般来讲,抽象数据类型是没有详细定义的数据类型;可以在以后对它们进行定义或者把它们隐藏起来。按照理论,这个概念与面向对象的核心有关;可以首先定义对象和类之间的关系而在以后对细节进行说明。在实际应用中,这个概念并不是很重要,除了下面这种情况:诸如 C++ 这样的语言提供隐藏实现细节的方法,使得数据类型的定义是根据功能而不是根据数据类型的创建方式得来。

集合(aggregates)

集合是用于初始化复杂数据类型的一组常量。它有两种形式。第一种是组。组中包含一个或多个单元,每个单元都可以是一个简单的值或者是其它集合(例如字符串)。下面的例子中,需要大中括号并且可以添加任何数目的单元。

```
{item [,item]...}
```

集合的第二种形式是字符串,用于初始化字符数组。字符串的例子如下:

"text"

例如,下面的语句对 `char *` 指针型数组进行初始化:

```
char * cats[3] = {"Bill", "OB", "Purly"};
```

在上面的例子中,每个指针都指向不同的字符串。但是在下面的例子中,没有对所有的指针都进行初始化:

```
char * cats[5] = {"Bill", "OB", "Purly"};
```

结果是创建了五个 `char *` 类型的指针数组 `cats`,并初始化了前三个指针。后两个元素的值为空(NULL)。

在使用嵌入的组进行复杂结构的初始化时,每个嵌入的组初始化一个逻辑子单元。例如,下面的声明对矩阵 `matrix1` 的三行进行了初始化,但是只对每一行的前三个元素进行了初始化。没有进行初始化的元素的值默认为 0。

```
int matrix1[3][6] = {{1,2,3},{4,5,6},  
                    {7,8,9}};
```

集合在没有下标的数组的初始化中具有重要作用。在这种情况下,集合判断需要为数组分配的内存空间。例如,下面的声明正好分配六个元素的内存空间:

```
int lotto_numbers[] = {1,5,21,27,33,40};
```

匿名联合 (anonymous union)

匿名联合是没有名称的联合。ANSI C++ 支持这种联合。在使用这种联合的某个成员时,成员的名称可以清楚地说明它与联合的关系。例如,假定进行了如下的声明:

```
struct databag {  
    char fancy_name[20];  
    union {  
        char strdata[20];  
        double floatdata;  
    };  
} bag1;
```

可以直接引用 `bag1.strdata` 和 `bag1.floatdata` 而无须指定联合的名称。

这种用法有一个限制:匿名联合的使用不能产生名称冲突。在上面的例子中没有名称冲突,因为 `fancy_name` 与 `strdata` 或 `floatdata` 都不相同。详细内容,请参看第

十三章“联合”。

参数(argument)

参数是传递给函数或模板的值。假定有一个函数 `fact`, `fact(4)` 返回 4 而 `fact(5)` 返回 5。这里的数字 4 和 5 就是参数。在 `fact` 函数的定义中, 可以将参数声明为 `n`:

```
long fact(int n) {  
    //...  
}
```

有些书中使用参数和变量来区分参数声明中的参数(例如上面例子中的 `n`)和传递给函数的值(例子中的 4 或 5)。有的书分别使用术语形参和实参。简单起见, 本书只使用参数这个词。尽管有时有必要分清参数声明和参数值, 但这种区分有时也会带来不必要的麻烦。

数组(array)

大多数现代编程语言都支持数组。在数组声明时, 可以重复使用某一种数据类型。可以使用下标或指向成员的位置来访问数组中的任何成员。数组的下标特别适合循环运算。

数组的声明形式 `type name[n]` 创建了一个从 `name[0]` 到 `name[n-1]` 的数组。例如, 下面的声明:

```
int nums[3];
```

在内存中创建了以下单元, 每个单元都分别等价于单独的 `int` 变量:

```
nums[0];  
nums[1];  
nums[2];
```

与 C 相同, 在 C++ 中, 数组与指针之间有紧密的联系。详细内容, 请参看第三章。

注意, 数组的名称是一个常量。在上面的例子中, `nums` 是等于 `nums[0]` 的地址的常量。(因为尽管数组中的数据可以改变, 但是数组的地址却不能改变, 而数组的名称等于这个地址, 所以数组的名称是一个常量。当然, 可以改变指向数据的指针。)

赋值运算符(assignment operators)

赋值是在变量中放置一个新值(例如表达式 `x=1`)。C 与 C++ 支持大量的赋值运算符。大多数运算符, 例如 `+=` 及 `-=` 在赋值之前进行某些运算。只有完全赋值运

算符,或称为简单赋值运算(=)除外。

C 和 C++ 的另一个独有的特征是语法上它们没有赋值语句。赋值语句只不过是另外一种表达式,它可以出现在更大的表达式中。(这样做有时可能出现错误,但是允许这样使用。)与所有其它表达式一样,通过在赋值运算后加一个分号可以将赋值运算变成一个语句。

```
x = y;
```

另外,所有的赋值运算符都可以改变左操作数的值。所有左操作数必须是一个合适的 lvalue:即必须是一个单变量的表达式并且在内存中有明确的位置。(参看 lvalue。)lvalue 通常是一个变量。

关于赋值运算的一些特殊要求,请参看第二章。第十一章列出了所有的赋值运算符。

关联(associativity)

运算符的关联的作用是当运算符具有相同优先级时,决定如何进行复杂的表达式运算。(参看“优先级”。)例如,加号(+)和减号(-)具有相同的优先级。那么,如何计算下面的表达式呢?

```
amount = 10 - 5 + n;
```

C++ 语法规规定加号和减号的关联顺序是自左至右。于是,上面的表达式等价于:

```
amount = (10 - 5) + n;    //amount = 5 + n;
```

而不是:

```
amount = 10 - (5 + n)    //amount = 5 - n;
```

关联并不难判断。C++ 中的所有运算符的关联都是自左至右的,除了赋值运算符,unary 运算符以及条件运算符(?:)。如果弄不清关联的关系,请参看表 10-1 或 10-2,这两个表格给出了 C++ 中所有运算符的优先级和关联顺序。

计算机语言中的关联和优先级与小学时学的算术运算规则是一致的。

基类(base class)

基类为其它类(称为派生类)提供成员。例如,如果类 B 是类 C 的基类,那么 B 中所有的成员都自动成为 C 的成员。这种关系是 C++ 中继承的一个例子。有时基类也称为超类或父类。详细内容请参看第五章。

基类构造函数 (base-class constructor)

在编写一个构造函数时,可以调用基类的构造函数(如果基类有构造函数的话)。如果基类有私有成员,这种方法十分有用,因为不这样做的话,这些私有成员就无法进行初始化。下面是通用的语法:

```
class::class(args) : base_class(args) {  
    statements  
}
```

举一个例子来说明一下。假定有一个从类 CAuto 派生出来的类 CSportsCar。下面的程序调用了基类的构造函数。在这个例子中,参数 h 和 m 传递给 CAuto 的构造函数。

```
CSportCar::CSportsCar( double h, CStr m, double a) :  
    CAuto( h, 2, m) {  
  
    accel_0_60 = a;  
    stripes = 0;  
}
```

这段程序将初始化值 h 和 m 以及常量 2 传递给基类 CAuto 的构造函数。在第十六章的最后有此例子的详细说明。

位字段 (bit field)

位字段是特殊的 C/C++ 数据类型,它代表整数的某些位。字段可以小到只有一个比特,也可以包括多个比特。在编写与位字段有关的程序时,编译器产生可以获得、控制以及存储字段值的指令。尽管结果与直接使用位运算获得的结果相同,但是位字段具有易于读取和理解的好处,并且(在许多情况下)可以使用更少的代码。详细内容请参看第二章。

布尔值 (Boolean values)

布尔型变量可以有两个值:真和假。比较起来,大多数数据类型值的范围大于二。如果存储空间十分宝贵,只使用一个比特来存储布尔值是最有效的方法。(参看“位字段”。)但是对单独位的访问需要其它的命令;因此,通常使用整型数来存储布尔值。

在 C++ 的早期版本中,没有独立的布尔数据类型;将 0 看作假,而其它任何非零值都为真。如果不小心,这样做会产生奇怪的结果。例如,将 6 和 9 进行位与(&),其结果为 0,即为假,尽管 6 和 9 都为“真”。除了使用位运算之外,另外一种解决方法是使用逻辑运算符。C/C++ 的逻辑运算符将所有的非零值的处理都相同。

ANSI C++ 提供了另外一种解决方法,使用新增加的 `bool` 数据类型。赋给 `bool` 型变量的任何非零值都自动转换为 1。关于这种数据类型的详细内容,请参看第十章。

回调函数(callback function)

在调用某些函数时有些需要另外一个函数的地址;后一个函数就叫做回调函数。使用回调函数的结果是允许别人的程序调用你的程序,至少是暂时允许别人进行调用。尽管这听起来有些荒谬,但这是实现 `qsort` 和 `bsearch` 函数的灵活性的唯一方法(请参看第十五章)。在调用这样的函数时,必须将回调函数的地址传递过去。

回调函数的另外一个特点是它的参数是函数的地址。这牵涉到函数指针的用法。详细内容请参看第十五章的 `qsort` 和 `bsearch`。

类型转换(cast)

类型转换将一个表达式的值进行转换并传递转换后的结果。(如果转换某个变量,对原始变量的类型没有影响;改变的只是表达式的类型。)在某些情况下,类型转换的结果是改变了原值的某些位。例如,将一个整型值转换为 `double` 格式就对原数据的实际数据模式进行了修改。

```
double d = static_cast<double>(i);
```

另外,类型转换对数据本身没有影响,只是改变了表达式被处理的形式。例如,将一个 `int*` 指针转换为 `float*` 指针时,对指针的值本身并没有什么影响,但是如果以后对指针进行引用时区别就十分明显。

ANSI C++ 支持一系列的新的转换运算符,例如 `static_cast`,但是它也支持老式的 C 类型转换运算符。关于这些类型转换运算符的用法和使用规则的详细内容,请参看第十二章。也可以参看“提升”

类(class)

类的概念可能是 C++ 中最重要的概念。一般说来,类是任何用户定义的,除了数组或 `typedef` 类型之外的数据类型;类包括使用关键字 `class`、`struct` 或 `union` 定义的数据结构。在声明类的时候,由于创建了新的数据类型,所以也对 C++ 进行了扩充。

在声明了类之后,可以使用声明的类创建任意个类的实例,或对象。每个对象或实例都有定义的类的所有属性。第十五章介绍了类和对象。

类与 C 中的数据结构相似,都包含数据段,但是类又进行了扩充,它加入了支持函数。它们就是成员函数;成员函数定义了类的对象的功能。也可以定义在这些对象之间进行运算的运算符(如 `+`、`*`、`/` 等等)。(请参看第七章。)另外一个重要特征

是私有成员的访问。私有成员对外界隐蔽了类的成员。(请参看“封装”。)

类实例(class instance)

实例与对象一样。请参看“实例与对象”。

注释语句(comments)

注释语句是编译器忽略的文本。尽管没有进行编译,注释语句仍然是程序的一部分。理论上讲,注释语句可以包含任何内容。但是程序员一般都添加那些有助于理解程序的内容的注释。

C++ 支持两种注释方法。在 C++ 程序中最常见的用法是行注释。行注释告诉编译器忽略一行中双反斜杠之后的所有内容。(在 C++ 中很少将一个语句分两行写。)

```
x = y;    //此处为注释语句;将 y 赋值给 x。
```

另外一种用法——多行注释——是从 C 中继承过来的。编译器忽略符号 `/*` 和 `*/` 之间的所有内容。例如:

```
/* 此处为 C 形式的  
   多行注释 */
```

在使用多行注释时,要明白开始和结束的注释符号的作用与括号的作用不同,不能进行嵌套使用。编译器只是忽略从第一个注释开始符号(`/*`)到第一个注释结束符号(`*/`)之间的内容。所以,在下面的例子是不正确的:

```
/* This is the outer comment, and  
   /* This is an embedded comment.  */  
   Here is more of the outer comment.  */
```

在上面的例子中,编译器将第二行末尾的注释结束符号(`*/`)作为注释的结束标志,而对第三行进行编译。如果是临时删去某些代码,应该使用 `#if` 和 `#endif` 命令。这两个命令可以进行嵌套使用。关于这些命令的详细内容,请参看第十四章。

复杂数据类型(complex data types)

复杂数据类型是在其它类型的基础上创建的数据类型。复杂数据类型包括所有的数组、类、联合以及结构。非复杂数据类型是那些原始数据类型,如 `int`、`bool` 和 `float`。

复合语句(compound statements)

复合语句由一个或多个由大括号(`{|}`)包围的语句构成。这些语句成块运行。或

者执行所有的语句,或者一个都不执行。使用方法为:

```
{  
    statements  
}
```

复合语句有许多用处。它们可以用于定义语句块。在语句块中定义的本地变量的作用域一直到后大括号(`}`)。复合语句可以控制变量的作用域。但是最常使用复合语句的地方是控制结构。任何可以加入语句的地方都可以加入复合语句。例如下面的 `if` 语句块,或者执行 `if` 语句块中的所有语句或者一句都不执行:

```
if (switch_now) {  
    temp = a;  
    a = b;  
    b = temp;  
}
```

注意在 C 和 C++ 中,分号(`;`)是语句的终止符,而不是象在 Pascal 中那样是语句的分割符。按照我的经验,这使得 C++ 更容易使用。在每个独立的语句之后必须加上一个分号。唯一不是由分号终止的语句是复合语句块。因此,不要在后大括号(`}`)之后加上一个分号。类的声明除外。

条件编译(conditional compilation)

条件编译是维护程序多个版本的一门技术。在为不同的计算机平台编写程序时,可以将与平台有关的代码放在 `#if`……`#endif` 之间。使用这种方法可以方便地对程序进行编译。关于 `#if` 和其它命令的详细内容,请参看第十四章。

控制结构(control structure)

控制结构是能够改变控制流或作出判断的语句。(但是尽管“深蓝”战胜了 Kasparov,对于计算机真实地作出判断的争论仍然存在。事实上,所有的控制结构都执行简单的数学测试并执行指令。)

C++ 中的控制结构包括 `if-else`、`while`、`do`、`for` 和 `switch`。在某种程度上也可以将函数和 `goto` 语句看作是控制结构。关于这些控制关键字的详细内容,请参看第十三章。

常量(constant)

C++ 支持称为常量的数字。总地来说,常量是不能改变的值。C++ 中有好几种不同类型的常量:

- 数字常量, 如 100, 4.5, 0xff 以及 018。(后两个分别是十六进制和八进制格式。)
- 使用 #define 命令定义的符号常量。(也称为宏。)
- 使用关键字 enum 定义的常量。
- 字符串, 如 "This is a string"。
- 数组名, 等于第一个元素的地址的常量。(数组名是一个常量, 但是指针和数组的内容不是常量。)
- 任何使用关键字 const 声明的变量。

最后一类常量与其余的常量有很大不同。关键字 const 声明的变量仍然是变量, 尽管它的值不能改变。在运行时 const 类型的变量与其它类型的变量一样仍然占据内存空间。另外, 这样的变量可以由基本类型构造, 同样有作用域和存储类。

头三类常量与 const 型变量的不同在于它们包括的变量只在编译时存在。在编译时编译器对这些常量进行合并。例如象 2 + 2 这样的表达式将由 4 来代替。

关于数组常量的详细内容, 请参看第二章; 关于字符串常量, 请参看第十三章; 关于 #define 命令, 请参看第十四章; 关于关键字 enum 和 const, 请参看第十三章。

构造函数 (constructor)

构造函数是类的初始化函数。在创建对象时, 都要调用对象的类的构造函数。构造函数是进行数据成员初始化以及进行其它初始化任务的理想地方。构造函数的名称总是与类的名称相同, 它没有返回值, 甚至返回类型都不是 void 型的。

可以对同一个类编写多个构造函数, 只要每个构造函数的参数表不相同就可以。构造函数列表与用于初始化类的对象的参数相对应。可以在对象声明时或者使用 new 运算符创建类的对象。如果对象初始化时不需要初始化任何数据, 就执行类默认的构造函数。(请参看“默认构造函数”。)关于构造函数的详细内容, 请参看第六章。也可以参看“复制构造函数”。

类型转换函数 (conversion functions)

C++ 使用两种方法进行类型转换。可以使用构造函数来实现从类型 T 到类 C 的转换格式为 C(T)。这种转换定义了如何把 T 的一个实例赋值给 C 的一个实例 (就象 c-obj = t-obj 一样)。

转换操作符函数可以定义向外的转换。例如,如果类 C 里有一个转向 T 的转换函数,该函数就负责把 C 的实例赋值给 T 的实例(如同 `t-obj = c-obj`)。关于转换操作符函数的详情,请参阅第七章。

构造函数和转换函数的作用远不止定义赋值操作,它们还进行隐式的类型转换。例如,若 CStr 类里定义一个到 `char *` 的类型转换,那么在任意函数调用中使用的 `char *` 类型的参数都可由 CStr 的实例来替换。

复制构造函数(copy constructor)

在使用一个类的对象对新创建的同一个类的对象进行初始化时就需要复制构造函数——换句话说,对前一个对象进行复制。例如,如果将对象作为值进行传递,程序就必须复制对象然后进行传递。需要进行对象复制的另外一种情况是函数将对象作为值返回(假定没有使用指针或引用将对象返回)。

每个类都有一个复制构造函数。如果没有为某个类编写复制构造函数,编译器将自动定义一个功能不大的复制构造函数。这个复制构造函数完成简单的数据成员的复制功能。尽管在许多情况下这些功能已经足够,但是如果类中包括指针时就不能使用这个复制构造函数。对于类 C 来说,它需要一个对类 C 的对象的一个引用,这时复制构造函数的用法如下:

```
C(const C&);
```

详细内容请参看第六章。

数据抽象(data abstraction)

数据抽象是按照数据的功能而不是数据的内部结构进行类型定义的一种方法。在学习 C++ 时,最好不要忽略这个术语。请参看抽象数据类型。

数据成员(data members)

数据成员是类中的变量。一般来说,数据成员定义了类的每个对象的数据段。有一个例外就是:使用关键字 `static` 声明的数据成员在每个对象中没有相应的数据段。静态数据成员由同一个类的所有对象所共享——这意味着对单独的一个对象来说它们并没有什么作用。

声明(declaration)

声明是描述变量、类或结构的语句。这样的语句给出类型信息——告诉程序正要引用的项目的种类。有些声明创建变量或函数:这些声明也称为定义。(请参看“定义”。)例如:

```
int x, y, z;    //将 x,y,z 声明为整型
```

其它的声明没有定义变量或函数,但是为前向引用和外部引用提供正确的信息。这种声明的例子包括函数原型和 `extern` 声明。在使用声明时,C++ 比其它任何语言都更为严格。在调用函数之前必须或者进行函数定义或者给出完整描述函数的返回类型以及参数类型的原型。

要理解复杂的 C/C++ 的声明,必须反过头来问一问自己:当表达式出现时,它表示什么意思?例如,下面的声明:

```
char * sarray[100];
```

要明白这个声明,首先必须知道 C++ 在声明时使用同样的关联优先级规则。因为单操作数运算符的关联是自右至左,上面的声明等价于:

```
char * (sarray[100]);
```

假定表达式 `*(sarray[n])` 出现在程序中,n 可以是 0~99 的任何值。这说明数组的位置是从 0~99 并且通过引用可以得到一个字符。因此 `sarray` 是 100 个指向字符的指针构成的数组。反之:

```
char (* ptr_to_array)[100];
```

是一个指向 100 个字符组成的数组的指针,而不是一个 100 个指针构成的数组。

类似地,可以明白下面的两个表达式为什么不同。第一个表达式是函数原型;第二个表达式是一个函数指针。

```
int * ptr1(void);    //函数 ptr 返回整型值  
int (* ptr2)(void);  //ptr 指向函数返回的整型值
```

明白这些声明的关键在于记住在程序中,在函数调用之前 `ptr2` 必须获得相应的值。

修饰(decorating)

修饰是 C++ 编译器将类型信息加入到名称中的一种方法。例如,如果声明了一个名为 `Charley` 的整型数,在生成 `.obj` 或 `.o` 文件时,C++ 并非简单地输出 `Charley`。在输出 `Charley` 的同时,C++ 在名称中加入其它的字符,结果生成一个更长的名称,使用该名称可以知道 `Charley` 是整型的。

名称修饰的原因有两个。其一是支持类型安全连接。因为 `.obj`(或 `.o`)文件中的函数和变量的名称是经过修饰的,连接程序就不会将整型的 `Charley` 与浮点型的

Charley 看作同一个符号。这就阻止进行某些错误代码的连接。修饰的另外一个目的是支持函数的重载。名称修饰结果与特定的实现有关。

默认的参数值(default argument values)

函数定义中的每个参数都可以有一个默认值。如果在函数调用时没有为有默认值的参数指定数据,就使用参数的默认值。假定定义了如下的函数,该函数的两个参数一个需要赋值,另外一个有一个默认值:

```
void set_vars(char * name, int amt = 0);
```

如果只使用一个参数调用 set_vars, amt 的值就默认为 0。

```
set_vars("Joe Schmoe");
```

但是,如果使用两个参数调用 set_vars, amt 就使用指定的值。

```
set_vars("Joe Schmoe", 5);
```

默认参数的特性有许多限制。首先,所有的具有默认值的参数必须在没有默认值的参数之后。换句话说,具有默认值的参数必须位于参数列表的末尾。这一条规则是必须的,因为 C++ 按从左到右的顺序进行赋值。如果声明的函数有五个参数,在调用函数时指定了三个参数,那么头三个参数获得指定的值而后两个参数使用默认值。

另外一个限制与函数的重载有关。(请参看“重载”,“函数”。)在对函数名称进行重载时,函数的参数列表中必须至少有一个没有默认值的参数不同。

缺省构造函数(default constructor)

缺省构造函数是没有参数的初始化函数。当创建对象而又不需要对对象中的值进行初始化时,就调用对象类的缺省构造函数。对类 C 来说,它的缺省构造函数表示为:

```
C();
```

如果没有为类编写构造函数,编译程序将提供一个隐藏的缺省构造函数,该函数将所有的数据成员的值设置为 0。但是,如果定义了构造函数,编译程序就不再提供隐藏的缺省构造函数。所以最好还是为类编写一个缺省的构造函数,尽管这个构造函数没有实现任何功能。编译程序提供一个缺省的构造函数,而当编写了一个构造函数之后编译程序又不再提供缺省构造函数,这一点可能看起来很奇怪。但是这一点无碍大局。C++ 自动提供缺省构造函数是为了与 C 中使用的 struct 类型相兼容。不过在编写自己的类的时候,不应该过多地依赖编译器。

关于缺省构造函数以及其它析构函数的详细内容,请参看第六章。

定义 (definition)

定义用于创建变量或函数。在大的复杂的项目中,可以多次对函数或变量进行声明,但是却只能定义一次。(例外:虚函数可以由不同的类按照不同的方式来定义。)变量的定义是程序中真正创建变量的地方,也可以进行变量的初始化。例如:

```
int i;
```

函数定义包括函数被调用时要执行的代码。下面是一个简单的例子。(这是一个计算阶乘的函数。)

```
long fac(int n) {  
    long amount;  
    for (amount = 1; n > 1; n--)  
        amount * = n;  
    return amount;  
}
```

定义是一种声明,但是不是所有的声明都是定义。例如,extern 声明就没有定义变量。

间接引用 (dereference)

在对指针使用间接引用时,得到的是指针指向的数据。例如,如果指针 ptr 存储的是 x 的地址(ptr = &x),那么间接引用 ptr 就得到 x 的值。指针间接引用运算符(单独的一个*)就实现这种功能,例如:

```
cout << * ptr;    //输出指针指向的值
```

在进行指针的间接引用时,程序查看指针中存储的地址。它使用该地址获得某个值。例如,ptr 包含 0x1022,那么表达式 * ptr 就得到在内存地址 0x1022 中存储的值。

派生类 (derived class)

派生类继承另一个称为基类的部分或全部成员。例如,如果类 C 是从基类 B 派生出来,那么 B 中所有的成员都自动成为 C 的成员。(派生类 C 也可以添加自己的成员。)派生类有时也叫从属类或子类。详细内容,请参看第五章。

析构函数 (destructor)

析构函数的作用与构造函数的作用相反。构造函数是在创建对象之后执行,同时也是在从内存中删除对象之前运行。很少有人直接调用析构函数。在进行某些操作时自动调用析构函数,例如使用 delete 运算符删除对象。(但是对象的代码仍然在类中。)

尽管不需要为每个类都编写一个析构函数,但是析构函数可以用于进行对象的清扫工作。例如,可以使用析构函数关闭与类的对象相关的文件。与构造函数不同的是,析构函数只有一个:每个类只能有一个析构函数。对于某个类 C,它的析构函数可以如下进行声明:

```
~C();
```

详细内容请参看第五、六章。尽管名称比较蹩嘴(析构函数)以及语法十分古怪(只能有一个),有时析构函数还是很有用的。

指令(directive)

指令是编译器编译和运行程序的其它部分之前执行的特殊命令。与语句不同,指令没有相应的运行时操作;因此使用指令编写的语句不占用运行时间。最常用的指令有 #define,它用于创建符号常量:

```
#define PI 3.14159265
```

关于 ANSI 支持的指令以及指令的说明的表格,请参看第十四章。

空语句(empty statement)

在 C 和 C++ 中,使用空语句是合法的。空语句是由分号结尾的空白语句。在下面的例子中,那些分号是允许出现的。下面的程序可以安全地运行:

```
int i = 1;;;
```

严格来讲,上句不是一条语句,而是一个声明后紧跟两个空语句。

空语句可以用在下面这种情况中:标识出 goto 语句的目的地。特别是在跳转到进程的末尾时,就需要在跳转标签后使用一个空语句。要不然跳转标签后面就没有可执行语句了,这是非法的。

```
i = m * 2;
if( i == j)
    goto end_of_func;
    //...
end_of_func;
;
```

封装(encapsulation)

封装的意思是保护事物的内部。对于完全可以自我包含的数据结构来说,它需要数据的完整性。也就是说,它可以通过已经建立的通道与其它对象进行通讯(类似于神经细胞通过树突进行通讯),但是对象的内部是私有的。通过将类的某些部分声

明为 private 或 protected 类型,C++ 支持这种通讯方式。

封装是面向对象编程的三大支柱之一。另外两个支柱是继承和多态。在这三个支柱当中,封装可能是最重要的,特别是在中小规模的程序中。封装的好处在于在每个类中(因此在每个类的对象中),可以对其内部进行修改或者优化而又不影响程序的其它部分。封装这个词与术语抽象很相近。从根本上讲,二者完全是同一件事;隐藏细节。第五章用一个例子说明了封装的好处。

枚举(enumeration)

枚举就是指列表。在 C++ 程序中,枚举可以用于快速创建一串整型常量。当然可以使用关键字 #define 创建一系列的常量,但是关键字 enum 更方便更简洁。下面是枚举的一个简单例子,枚举的第一项默认为 0:

```
enum number {  
    zero, one, two, three;  
};
```

下面是一个更为实际一些的例子。在该例子中,CLUBS, DIAMONDS, HEARTS 以及 SPADES 的值分别为数字从 0~3。这些常量比从 0~3 的数字具有更好的可读性。但是输出时并不输出这些常量的名称。

```
enum suit {  
    CLUBS, DIAMONDS, HEARTS, SPADES;  
};  
//...  
cout << HEARTS << endl; //输出“2”。
```

关于关键字 enum 的使用方法的详细内容,请参看第十三章。附录 B 中也提到了 ANSI 对 enum 的扩展。

转义字符(escape sequence)

转义字符是 C/C++ 用于在引号括起来的字符串或单个 char 数值中加入特殊字符的技术。所有的转义字符(例如 \n, 表示开始新的一行)都是以反斜杠(\)开始。因为反斜杠是转义字符的开始,所以必须使用两个反斜杠(\\)才能在字符串中输出反斜杠。转义字符的列表,请参看第十章的“字符”主题。

可执行语句(executable statement)

可执行语句是在运行时需要执行的操作,例如函数调用或者进行某个计算。在 C 语言中,声明与可执行语句间的区别十分明显,每个语句不是声明语句就是可执行语句,但是不能即是声明语句又是可执行语句。而 C++ 则没有这么明显的区别。创

建对象时的变量定义不只是定义数据,它还调用构造函数。

因此可执行语句这个概念在 C++ 中不如在 C 中那么明确。不过它仍然可以指某些可以执行的语句。大多数语句——如控制结构、函数调用以及由分号(;)终止的表达式——都可以看作是可执行语句。在实际应用中,数据定义可以是也可以不是可执行语句。但是为了清晰起见,将它们称为声明而不是可执行语句。

异常(exception)

异常表示程序出现必须立即处理的非正常情况。未经处理的异常会终止程序的运行。一般情况下,异常是运行时错误,当然也可能出现非错误异常。ANSI C++ 提供关键字 try、catch 以及 throw 对异常进行处理。作为处理异常和其它事件的一种方法,C++ 异常处理的好处在于可以对这些事件进行拦截并在程序的任何部分作出响应。因此可以集中编写错误处理程序或者按照需要对错误处理程序进行修改。详细内容,请参看第十三章。

表达式(expression)

表达式是 C 和 C++ 中最基本的运行单元。一般说来,表达式可以是任何可以获得一个值的语句。例如,诸如简单变量 x 就是一个表达式,而 $(x * n + 1) / y^{2k}$ 则是一个复杂的表达式。C / C++ 语法的一个特别的地方在于只要在末尾加入一个分号(;),表达式就变成语句。另外,赋值运算也是表达式,因为它们返回一个值——并对该值进行了赋值操作。下面都是在表达式的末尾加入分号变成语句的例子,它们都是合法的:

```
x;  
y = x;  
y = x = z = 0;  
y = (x * n + 1) / y2k;  
func1();  
y = func1();
```

前向引用(forward reference)

在 C++ 中,必须在使用变量(或类)之前进行声明。如果在函数定义之前进行调用,必须在程序的开头或至少在调用函数的语句之前加入一个函数原型。

C++ 允许在类的完全声明之前进行引用。在两个类的实例分别指向对方时,就需要进行这样的引用。当然这种情况很少出现。(“我指向你;你指向我。”)这种操作完全合法,但是需要对类进行前向引用。将 class name; (如,对结构来说,可以是 struct name)形式的语句放在程序的开头就可以实现前向引用。例如:

```
class A;
```

```
class B;  
  
class A {  
public:  
    B * ptr_to_B;  
};  
  
class B {  
public:  
    A * ptr_to_A;  
};
```

ANSI C++ 将前向引用的能力扩展到嵌套类的内部。(参看嵌套控制结构和类。)

函数 (function)

函数是作为整体运行的语句集合,可以具有返回值也可以没有返回值。函数的概念是所有编程语言的重点,尽管在其它语言中函数又被称为进程或子例程。C 和 C++ 的独有特性是无论子例程是否有返回值,都将其看作是函数。没有返回值的函数应该声明为 void 类型,不然的话,有返回值的函数与没有返回值的函数之间就没有什么区别了。C /C++ 语法的另外一个特点是调用函数时可以不考虑函数的返回值。关于函数基本语法的介绍,请参看第二章。

函数成员 (function member)

请参看成员函数。

函数指针 (function pointer)

请参看指向函数的指针。

函数原型 (function prototype)

请参看原型。

头文件 (header files)

头文件中包含一般的定义以及函数原型。C++ 要求在使用函数之前必须详细地说明函数原型。另外,标准库中的许多函数都是以宏函数的形式实现。在调用库函数之前必须将所有的声明读到源文件中去。头文件可以用于包含这些声明。

使用 #include 指令来包含头文件。关于这个指令的详细内容,请参看第十四章。

ANSI C++ 最新推荐使用的编程技巧是用虚头文件代替了库头文件。例如,对于语句 `#include <stdio.h>`, 推荐方法使用的是 `#include <cstdio>`。这样 C++ 的未来版本可以用一系列的内部声明来替换头文件,这可以加快编译速度。除了支持标准库函数之外,对某个工程文件来说,自己编写的头文件也是十分有用的。这对于有多个源文件的工程文件来说更为有用。下面这些声明应该全都加入到头文件中:

- 所有函数原型,除了模块私有的函数原型。
- 工程文件中广泛使用的所有类的声明。
- 模块中使用的所有声明为 `extern` 型的变量。
- 宏函数的定义。
- 工程文件中广泛使用的由 `typedef` 定义的复杂类型。

因为可能需要将头文件包含在多个源文件中,所以有一些代码不能放在头文件中。例如,不能将函数的定义放在头文件中,除非它是一个宏函数。

标识符(identifiers)

标识符是在程序中定义的名称。程序中的任何字,除了用引号括起来的字符以及注释中的字符之外或者是关键字,或者是标识符。还有一种说法是标识符是创建的任何名称。另外,在库文件中定义的名称也是标识符。(库名称不是关键字。)变量、类、宏以及函数的名称都是标识符。

C++ 中的标识符与 C 中的标识符一样,都必须遵守下面的规则:

- 标识符不能与 C++ 的关键字相同。
- 标识符中的每个字符必须是大写或小写字母、从 0~9 的数字或者下划线(`_`)。
- 第一个字符不能是数字。

另外,标识符的使用还有其它规定。首先,尽管可以将下划线(`_`)作为标识符的开始,最好还是不要这样作。因为 C++ 标准库使用的一些特别低级的标识符(一般对程序员是不可见的)就是以划线(`_`)作为开始的。

第二,应该避免创建除了大小写之外完全相同的名称:例如,MrBig 与 mrBIG。因为尽管 C++ 区分大小写,连接程序也许不能区分大小写,所以这样做可能会出现错误。语言与连接程序之间的不一致可能产生莫名其妙的错误。最保险的方法就是保证所有的标识符都唯一并对每个标识符或者使用大写字符或者使用小写字符。当然可以在某个名称中混合使用大小写,这样有助于保证程序的清晰:例如,avgDollarsPerEmp。但是不要输入 AVGDollarsPERemp。

除了这些规则之外,名称的使用也与个人的编程风格有关。著名的“匈牙利命名法”(由一个匈牙利人,Microsoft 的 Charles Simonyi 提出)也有自己的规定,所以最好不要那么古板。

实现(implementation)

术语实现用于许多方面。首先,实现指的是函数定义。在派生一个类并重载函数定义时,可以说是在实现这个函数。其次,实现可以指类的所有内部成员,正好与类的外部(或称为接口)相对。第五章也使用了这个术语。最后,特定的实现也可以指某些 C++ 编译器的行为。

间接访问(indirection)

术语间接访问是指通过指针而不是直接引用数据。例如,请看下面的例子:

```
int i = 5;  
int *pi = &i;
```

指针 pi 包含 i 的地址。在使用 *pi 而不是使用 i 来对 i 进行引用时,就进行了间接访问。指针的间接引用也是进行间接访问。例如,表达式 * *ppi 进行了两次间接引用。请参看间接引用。

继承(inheritance)

继承可以使你在已有类的基础上声明一个新的类,因此也获得了旧类中的所有声明。继承也说明了在 OOPS(面向对象编程系统)中对类型重要性的强调。假定已经声明了一个类 A,可以将类 B 作为一种特殊的类 A。那么二者的关系就是:“B 是 A 的一种”。例如,假定已经声明了类 CAuto,只需要在新类中加入一些新类所需要的成员就可以创建一个新类 CSportsCar。CSportsCar 继承了 CAuto 中所有的成员。关于继承的详细内容,请参看第八章。

继承是面向对象的三大支柱之一。不过继承不如其它两个支柱那样重要。实际上,Microsoft 的部件对象模型(COM)更消减了继承的重要性。作为一种语言特征,继承与源代码的依赖性比较大;而 COM 作为一种面向对象模型,并不依赖机器或语

言。Microsoft 的 Visual Basic 也不支持继承,尽管它支持多态和封装。当然,继承的潜在作用很大:它鼓励有效地重复使用代码。不过,将一个对象包含在另外一个对象中也可以获得这样的好处。在 C++ 中,继承的作用是为虚函数以及多态提供了语法基础。

内联函数 (inline function)

编译器发现对内联函数进行调用时,它会将内联函数的定义加入到调用它的函数内。(参看图 G-1。)内联函数与宏函数类似,二者的执行方式与标准方式不一样,它们是将控制转换到一个不同的地址。不过内联函数也与宏不一样,内联函数遵循与普通函数一样的语法规则。

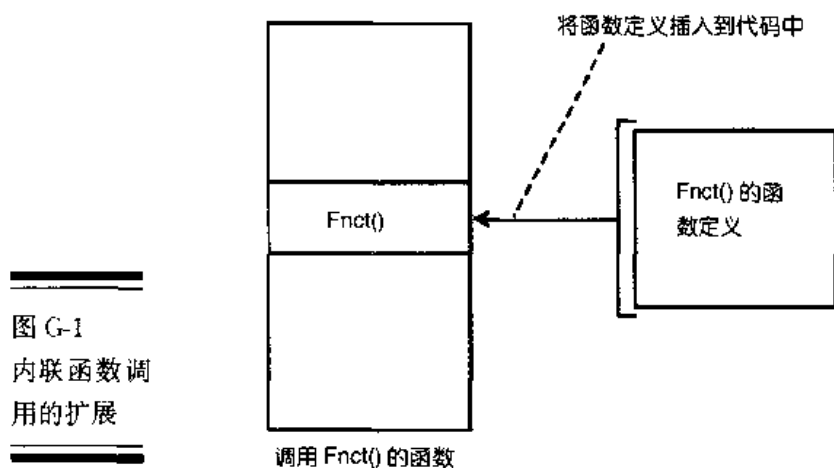


图 G-1
内联函数调用的扩展

在 ANSI C++ 中有两种方法创建内联函数:

如果内联函数是类的成员函数,可以在类的声明中对内联函数进行定义。例如:

```
class CHorse {  
    CStr name;  
    CHorse (CStr nm) {name = nm;}  
};
```

也可以使用关键字 `inline` 来定义内联函数。

```
inline double cube_it(double x) {  
    return x * x * x;  
}
```

另外,作为一种优化技术,有一些编译器也可以添加自己的内联函数。关于成员函数的详细内容,请参看第五章;关于关键字 `inline` 的详细内容,请参看第十三章。

实例(instance)

实例是对象的另外一种说法。在声明了一个类之后,可以用这个类来创建任意数目的实例。类与实例的关系大致类似于 `int` 类型与某个整型值之间的关系。类定义了通用类型以及函数代码,而实例包含特殊的数据值。

实现(instantiation)

实现的意思是“创建一个对象。”,或者说是“创建一个实例。”(对象与实例是同一回事。)尽管可以将类看作是创建对象的蓝图,但是也有一些类如常见的有抽象类,它不能用于创建对象,至少不能直接用于创建对象。一般称这样的类是“不能进行实例化。”

关键字(keyword)

关键字是 C++ 语言自己定义的名字。它不包括库的名称,库的名称是标识符。常用的标识符有 `if`、`while`、`do` 以及 `for`。关于关键字的说明,请参看第十三章。(第十一、十二章中的关键字也是运算符,例如 `new` 和 `delete`。)

左值(lvalue)

左值是可以出现在等号左边的表达式。变量可以是左值,但是常量(例如 105)不能是左值。通常在内存中有特定的地址的表达式都是左值。例如,如果将 `numbers` 声明为一个数组,那么 `numbers` 本身是一个常量,所以它不是左值,但是 `numbers[i]` 是左值。指针是左值。如果 `p` 是一个指针,`p` 与 `*p` 都是左值,因为即可以改变 `p` 的值又可以改变 `*p` 的值。

除了常量之外,最常见的不是左值的表达式是复杂的表达式,例如 `x + y`。尽管 `x` 和 `y` 在内存中都有特定的地址,但是 `x + y` 却没有。另外,被引用的指针表达式也是左值,因为(按照定义)它也有一个地址。例如表达式 `p + i + 2` 不是左值而表达式 `*(p + i + 2)` 却是左值。

标签(label)

标签又叫做语句标签,为 `goto` 语句提供目标。因为 `goto` 在大多数 C++ 程序中并不常用,所以也没有多大需要使用标签。标签的使用遵循下面的语法规则:

```
label. name: statement
```

关键字 `case` 和 `default` 也可以定义标签,不过这两个关键字常用在 `switch` 语句中。标签语句可以作为 `switch` 语句的可能目标。(请参看第十三章。)

C 和 C++ 语法中标签语句也是一个语句。这样就可以给一个语句添加任意多个标签。有些 case 语句就利用了这个特点。

```
case 1:  
case 2:  
case 3:  
    cout<< "Too small \n"; //执行 1,2 或 3
```

文字(literal)

文字是与符号名称无关的常量。它只是一些字符。文字有两种形式:数字文字(例如 0、-3、12、5.007、6E20 以及 0xFF)和字符串文字。字符串是由引号括起来的文本。请参看字符串文字。

逻辑运算符与位运算符(logic versus bitwise operators)

与 Visual Basic 不同的是,C++ 没有执行双重功能的逻辑运算符,它使用两套运算符进行条件的判断。一套(逻辑运算符如 && 和 ||)用于进行布尔量的计算。另一套(位运算符如 & 和 |)用于对整数的位进行操作。也可以使用位运算符进行布尔量的计算,不过使用时要十分小心。逻辑运算符将所有的非零值都看作“真”。位运算符更为仔细一些——它们对每一位都进行运算。在位表达式中可能得到莫名其妙的(甚至是错误的)值,如 4&(2>1)。比较起来,逻辑表达式 4&&(2>1)可以正确地计算出“真”。

逻辑运算符与位运算符的另外一个不同之处在于逻辑运算符按照“短路”逻辑进行运算。例如,如果与表达式(&&)的前半部分为假,程序就不会计算后面的值。位运算符总是对操作数进行完全计算。

宏(macro)

宏是使用 # define 指令定义的符号名。在预处理期间,编译器进行符号名的替换。例如,如果将 PI 定义为 3.14159265,编译器用这个数字串来代替所有出现 PI 的地方。(请参看预处理。)在预处理阶段,通常都要将符号名称进行扩展,因为替代文本一般都很长。有些汇编程序也有同样的功能。宏的名称也是从 Microsoft Macro Assembler(MASM)中得到的。

宏有两种用途:定义简单的宏,如上面定义的 PI;另外是定义宏函数。详细内容请参看第十四章的 # define。

main 函数(main function)

关键字 main 定义了一个作为程序入口的函数。也就是说,main 首先被执行,在 main 函数返回或停止运行时,程序也就终止了。main 函数主要用于标准控制台程序。动态连接库(DLL)以及 Windows 应用程序都没有 main 函数。详细内容,请参

看第十三章的“main”。

成员 (member)

成员是类中声明的变量或函数。请参看数据成员以及成员函数。

成员函数 (member function)

成员函数是类中声明的函数,与成员变量类似。成员函数为所有的类的对象提供功能。可以通过调用成员函数来使用对象;函数的定义会正确响应对函数的调用。因此成员函数的编写就是对象行为的实现。(例外:静态成员函数的范围是整个类,而不是某个单独的对象。)

可以将成员函数看作是作用域为类的函数。只能通过对象来调用成员函数,不过在调用时,可以访问对象的所有数据成员——甚至可以调用私有数据成员。下面的例子调用了类 CPnt 中的成员函数 move:

```
CPnt aPoint;  
aPoint.move(10,20);
```

很明显,通过函数调用可以对数据成员进行控制。关于成员函数以及类的介绍,请参看第五章。

多重继承 (multiple inheritance)

在 ANSI C++ 中,一个类可以从多个基类继承得到。(参看基类和继承。)例如,假定类 D 是从三个类 A、B 和 C 中继承来的。

```
class D:public A, public B, public C {  
    //添加 D 的特定声明...  
};
```

上面的声明结果是类 D 从类 A、B、C 中自动获得所有的声明。多重继承也产生了一些语言问题。如果 A 与 B 都有一个名为 x 的成员,那么在 D 中将如何引用 x 呢?解决方法是使用作用域运算符。可以使用 A::x 以及 B::x 来分别引用 A 或 B 中的 x。多重继承也产生了编译器的问题,不过这应该是编写编译器的人的事,与我们无关。

不过,除非十分需要,最好避免使用多重继承。多重继承违反了面向对象的理论,或者至少使面向对象更加复杂。如果 D 是从 B 中继承来的,那么 D 是 B 的一种。说 D 是一种 A 也是一种 B 有些牵强——正如说鸟是一种哺乳动物又是一种蜥蜴一样。

名称空间(namespace)

名称空间是 C++ 的基本概念。一般来说,名称空间是没有冲突的一组名称。例外是重载的函数名称以及具有不同作用域的变量。(请参看重载函数以及作用域。)不同名称空间的名称不会发生冲突。在使用一个新的名称空间时,不用担心某个名称是否已经在另外一个名称空间中使用。

类声明是命名空间的最常用例子。例如,可以在两个类中使用名称 `my_address` 而不会发生冲突。(这里的类是指使用关键字 `struct`、`union` 以及 `class` 声明的结构。)

C++ 也允许使用关键字 `namespace` 定义名称空间。这样定义名称空间减少了名称冲突的可能性并且在综合使用几个不同库的名称而又无法保证没有名称冲突时十分有用。关于关键字 `namespace` 的详细内容,请参看第十三章。

●—注意—

名称也叫标识符。请参看词汇表中的标识符。

嵌套控制结构以及嵌套类(nested control structures and classes)

一般说来,嵌套是在一个对象中有加入了一个对象。想象一下鸟巢中的鸟蛋被完全放置在鸟巢中,鸟巢保护鸟蛋不受外界伤害。(至少不受人的伤害。)

嵌套控制结构是在一种语句中包含另一个语句,另一个语句完全象鸟巢中的鸟蛋。嵌套必须遵循基本的语法规则。控制结构可以包含多个语句。不过,控制语句本身就是一个语句。因此,if 语句中可以包含一个 while 语句,while 又可以再包含 if 语句,可以这样无穷无尽地包含下去。也可以将循环无穷无尽地互相包含下去。

C++ 也支持嵌套类。类可以引用嵌套在其中的类,但是也不能任意进行引用。ANSI C++ 最新扩展的一个特性就是可以对嵌套类进行前向引用。详细内容,请参看附录 B。

换行(newline)

换行是用于中断一行的特殊字符(由引号括起来的字符串中的 `\n` 表示)。换行输出让显示器另起一行并从第一列开始。在使用 I/O 类及对象时,如 `cin` 和 `cout`,可以使用 `endl` 来换行。(例子程序请参看第十六章。)

对象(object)

对象是包含有与数据相关的函数的数据集合。可以将对象看作是一个智能数据结构,它可以对函数调用作出响应。每个对象都是特定类的一个实例。特定的类定义了数据字段以及类的行为。关于类及对象的介绍,请参看第五章。

面向对象(object orientation)

面向对象是编程语言设计、系统构建以及编写将数据与代码集成而不是按传统方式将数据与代码分开使用的程序的一种方法。简单说来,面向对象就是围绕对象编写程序。对象中包含状态(数据)以及行为(代码)。可以使用面向对象的方法来与传统的编程语言如 C 一起进行程序设计。不过,使用诸如 C++ 这样的面向对象的编程语言会使任务更为轻松,因为这些语言支持类。一个 C++ 的类定义了数据字段以及对象的成员函数。(请参看类。)

许多情况下,理解面向对象的最容易的方法是将程序看作是相互之间可以发送信息的自我包含的对象的集合。这种说法强调了每个对象的自治性,或智能性。为了更贴近计算机系统之外的生活,面向对象没有强调计算机系统的内部结构,在计算机中,数据与代码是截然分开的。例如,在生物系统中,每个单独的细胞都有自己的内部状态(数据),不过每个细胞也可以对刺激作出响应(内部代码规定的行为)。

许多书都谈到了对面向对象的需要。面向对象有三个支柱:封装、继承以及多态,这些本词汇表中都进行了定义。封装可能是最重要的。它在保持对象的私有性的基础上,澄清了程序单元(例如对象)之间的关系。多态在某些系统中也是很重要的。

面向对象是编程语言发展中十分自然的一个革新。在建立在结构化编程概念的基础之上而不是完全抛开结构化编程方式(尽管有些人建议这样作)。另一方面,面向对象也可能是初学者不知所措,因为简单的例子程序无法说明它的优点。面向对象的好处体现在大系统的设计中,大系统中需要定义各个复杂部分的相互操作。作为一名图形用户界面(GUI)以及大网络环境的程序员,面向对象的好处是十分明显的。

面向对象编程系统(OOPS)

OOPS 是面向对象编程系统(object-oriented programming systems)的缩写。面向对象是通用的概念;而 OOPS 试图将面向对象应用在特定的语言或系统中。实际应用时,术语面向对象与 OOPS 含义差不多。

操作数(operand)

操作数是一个子表达式,操作符用它来构成更大的表达式。例如,在表达式 $x + y$ 中,加法运算符(+)有两个操作数: x 和 y 。使用单操作数表达式时,例如 $*p$,就只有一个操作数(此时为 p)。

运算符(operator)

运算符是一个 C++ 语言定义的符号,用于将子表达式(又叫操作数)组合成更大的表达式。大多数运算符,但不是全部,有特殊的符号,如 $*$ 、 $+$ 、 $\&$ 以及 $/$ 。在某些

情况下,运算符可以是一个字母(如 `sizeof`、`new` 及 `delete`)。也有些运算符是由两个字符组合而成,如 `++`、`--`、`->` 以及 `==`。运算符使用的简单例子是使用加号(`+`)进行两个数的相加:

```
x+y
```

运算符与函数调用有许多共同之处。(第七章谈到了如何将运算符转换为函数调用。)运算符有一个或多个输入并有一个返回值。不过从语法上讲,二者之间有许多不同之处。首先,运算符仅限于 C++ 中定义的那些。可以对运算符进行重载,赋予它新的功能,但是不能自己创建一个新的运算符。第二,运算符有自己特有的用法,而且不需要括号。在没有括号的地方,可以按照优先级以及关联进行运算。

关于运算符的优先级、关联以及运算符的说明,请参看第十一章。关于如何用自己的类来定义运算符的详细内容,请参看第七章。

重载,函数(overloading, function)

重载意味着“重新使用某个符号或名称。”函数重载是使用同一个名称编写几个不同功能的函数的能力。也可以说,函数重载是编写了同一个函数的几个不同版本,每个版本都有不同的参数。

例如,下面的程序包含函数 `Init_var` 的三个不同版本的原型。这三个版本分别用于初始化不同的数据。

```
void Init_var(int);  
void Init_var(double);  
void Init_var(char * );
```

尽管这些函数的名称都是 `Init_var`,但每个函数都不相同,都必须有自己的定义。在编译时,编译器查看传递给 `Init_var` 的参数类型并以此来决定调用哪一个函数。如果没有发现匹配的声明,就会出现语法错误。

函数重载遵循某些限制。每个重载的函数必须有一个唯一的标题头——也就是说参数必须在类型或数目上各不相同。重载的函数不允许只在返回类型上有区别。只有参数列表不同,每个重载函数可以有不同的返回类型。(请参看标题头。)

尽管在概念上,重载函数与虚函数有些类似,但是二者是不能混淆的。虚函数的地址是在运行时决定的而不是在编译时决定。

重载,运算符(overloading, operator)

C++ 允许自己定义运算符的作用方式。例如,可以定义用加号(`+`)连接起来的两个类的实例时加号的作用。可以定义特定类型的两个对象的“和”是什么意思。运

算符重载是 C++ 最灵活、最有意思的特征之一。尽管有人认为没有运算符重载 OOPS 就不完全,但是有些面向对象的语言并不支持运算符重载。关于 C++ 中运算符重载的详细内容,请参看第七章。

形参(parameter)

形参指的就是函数中的参数(argument)。有些书将形参看作是函数声明中的参数,而用实参(argument)来表示传递给函数的值。不过,为简单起见,本书将形参和实参都看作是参数。

pragma

pragma 是由特定的编译器供应商提供的,与程序实现有关的命令。通常来说,pragma 产生的是不可移植性的代码。不过,有时也可以打开编译器的某些特性。ANSI C++ 支持用于此种目的的 #pragma 指令。详细内容请参看第十四章。

优先级(precedence)

优先级用于在复杂的表达式中判定运算符的执行顺序。运算符的优先级越高,就越先执行。例如,在表达式 $p = x + 2 * n$ 中,需要三步才能完成计算。优先级的执行顺序,从高到低是 $*$, $+$, $=$ ($=$ 也是一个运算符)。因此上面的表达式按照如下方式进行计算:

$$p = (x + (2 * n))$$

数学运算符($+$, $-$, $*$ 和 $/$)遵循标准优先级规则,乘除法优先于加法。关系运算符以及逻辑运算符的优先级更低,复杂运算符的优先级低于上面两种运算符。单操作数的运算符具有相对较高的优先级。当两种运算符具有相同的优先级时,C++ 利用关联性解决计算顺序的问题。

表 10-1 以及表 10-2 按优先级列出了所有的运算符。

预处理/预处理器(preprocessing/preprocessor)

在 C++ 编译器进行程序的编译时,首先要对源代码进行预处理。在预处理阶段,编译器通读整个源代码并对由特殊指令编写的程序进行修改。例如, `#define` 指令指定替代符, `#include` 指令告诉编译器到头文件中读取。这些变化并不影响源代码。源代码的修改是在一个中间媒介文件中进行的,尽管编译器可能允许打印输出这个中间媒介文件,我们还是看不见这个文件。

尽管这个阶段看似多余,但是对 C 和 C++ 程序员来说,预处理确实有一定的好处。预处理使得 C 和 C++ 支持许多有力的指令。在这些指令中,有条件编译,它有助于维护同一个程序的多个版本。预处理的另外一个好处是它只在编译时花费时间。这就允许程序员在程序运行之前定义替换字符以及某些操作。尽管这样做程序

可能会变大,但是在程序运行时,这些预处理指令并不占用时间。

程序员以及软件手册常常提到预处理器,好象预处理器是与编译器完全分开的。在第一个 C 编译器中,预处理器可能是一个单独的程序。但是现在构建程序的过程已经集成化,预处理与编译之间的已经没有什么时间间隔。尽管有时会认为“预处理器”是独立于编译器的,我认为最好是将预处理看作是编译器的第一个阶段。

预处理指令(preprocessor directive)

请参看指令。

指针(pointer)

在 C 和 C++ 中,一个最重要——有时也是最模糊——的概念是指针。指针是一个简单的变量,它存储的是一个地址。不过,如果以前从来没使用过地址,指针又有什么用呢?

在使用 BASIC, FORTRAN 或 Pascal 这些语言时,程序确实使用了地址。不过在这些语言中,地址的使用是不透明的。(Pascal 中也有指针变量,但是它们的用处很有限。)由于允许在源程序的层次上对地址进行操作,C 和 C++ 要求编程人员更好地了解计算机的工作方式;不过对地址操作的开放也为提高程序的效率提供了可能性。在通过引用进行变量的传递时需要用到地址。在通过引用进行传递时,函数将变量的地址传递给另外一个函数,而不是传递变量的一个复本。被调用的函数就可以对原始变量进行控制并可以永久地改变变量的值。另外,传递一个 2 字节或 4 字节的地址(无论地址的大小是多少)总比传递大的数据结构的复本要简洁的多。关于指针的各种用法,请参看第三章。

指向函数的指针(pointer to a function)

C 和 C++ 的另外一个好处在于可以声明一个存储函数地址的指针。(尽管常用指针来存储数据的地址。)函数指针的声明看起来有些奇怪,例如下面对 pf 的声明。这时,指针指向的函数需要两个整数作为参数并且必须返回一个 double 类型的值。

```
double (* pf)(int, int);
```

函数指针的语法有些特殊用法。其一是可以指定一个回调函数。例如,在调用 qsort 库函数时,可以将自己编写的比较函数的地址传递给它。(请参看第十五章的“qsort”。)另一个用处是构建函数的地址表,这些地址表可以用于间接调用。事实上,在 C 语言中,可以使用这些函数列表来模拟虚函数的功能。

函数指针和地址赋予程序特殊的能力,C++ 的虚函数不必使用函数指针就可以充分利用这些功能。函数指针存储的地址可以后来再进行赋值。因为可以通过指针进行函数的间接调用,所以尽管在编译时并没有编写被调用的程序,仍可以编写函数

调用的语句。被调用的程序可以在将来补上。

多态 (polymorphism)

多态是不清楚对象类型而调用对象函数的能力——效果上是向对象发送一个消息。如果构建了一个网络对象,理想情况下一个对象在不知道接收方的情况时,仍然可以向另外一个对象发送消息。只要电话旁边的两个人都说英语,尽管不知道另一方是谁,也可以明白“喂”的意思是提起注意。多态使得消息独立于对象,并且再进一步讲是独立于任何响应。

多态是面向对象的三大支柱之一,另外两个是封装和继承。在这三个支柱当中,多态是第二重要的,不过在某些系统,特别是图形用户界面(GUI)系统中,多态是必须的。例如,Microsoft Windows 就是多态的,因为它通过发送消息来运行程序。实际上,应用程序可以在创建 Windows 之后很长时间才运行。

在希腊语中,多态意思是多种形式。对消息(或者按照 C++ 的术语,对函数的调用)的响应可以有多种形式,而事实上可以有无穷多种响应。比较起来,传统的方法只允许程序与事先知道的特定数目的对象类型进行作用。

C++ 通过使用虚函数来支持多态。在 C 中,可以使用函数指针来模拟虚函数。关于多态以及虚函数的介绍,请参看第九章。

原始数据类型 (primitive data type)

原始数据类型由 C++ 语言中定义的类型集合构成,有时又称为原始类型。这些数据类型包括整数类型(例如 char、int、unsigned int 以及 long),浮点类型(float 和 double)以及 ANSI 支持的 bool 类型。bool 型不是作为通用目的的整数来使用,尽管它存储 0 和 1 两个值。所有 C++ 的原始类型都是数字类型。在 C++ 中字符串不是原始类型,它是字符数组。尽管很少对字符值进行数组运算,字符数组也是一种数字类型。

无论在 C++ 中还是在其它语言中,因为所有的其它类型最终都是由原始类型组成,所以原始类型都具有重要作用。原始类型也是构建数据结构块的基础。象类和数组这样的结构有时也称为复杂类型。

提升 (promotion)

提升是自动将数据向更大类型转换的过程。在混合类型的表达式中总要出现这种过程。例如,在将整数值赋给一个浮点型变量的时候(例如 float f = 1),C++ 将整数自动转换为浮点格式,这就是提升。此时不需要类型转换符。C 和 C++ 根据层次表按照需要进行数据的提升。例如,short 按照需要可以提升为 long, long 可以提升为 double。请参看类型转换。

原型 (prototype)

原型是包含返回值以及参数列表的完整的类型信息的函数声明。在 C++ 中(不像在 C 中), 每个函数都必须有一个有效的原型。

原型可以用于声明函数, 该函数可以在以后或在另外一个模块中进行定义。下面的原型的例子声明了一个函数, 该函数需要一个字符串作为参数并返回一个整数值。

```
int print_items(char *);
```

对于系统分析员以及用户界面专家来说, 原型有另外一种含义。对他们来说, 原型系统是指用于演示目的的具有最小功能代码可以响应用户操作的系统。这个意思与 C++ 中原型的含义截然不同。

纯虚函数 (pure virtual function)

纯虚函数是一个不包含任何函数定义代码的函数。纯虚函数为其它类提供原型。任何具有纯虚函数的类都是抽象类, 不能使用这些类来创建对象。这些类的用处就是可以从它们那儿派生类并实现所有或部分纯虚函数。

纯虚函数使用了表达式 = 0。例如, 下面的语句声明了一个样例函数, report_stats:

```
virtual void report_stats(int, char *) = 0;
```

尽管类中不包含纯虚函数的定义, 纯虚函数的声明却具有完整的类型信息。

递归 (recursion)

递归函数是调用自己的函数。为了避免无穷尽的递归调用, 这种函数需要某些方式来进行终止条件的检测。

C++ 完全支持递归调用, 不过在这样的函数内, 应该注意局部静态变量的使用。静态变量的使用可以打断递归函数的运行。

引用 (reference)

一个引用变量是另一个变量的别名。例如:

```
int b = 1;  
int &a = b;    //a 是 b 的别名。  
a = 5;        //将 b 的值变为 5。
```

参数引用是传递给函数的变量的别名。对该参数的改变将永久地改变传递过来的值。例如:

```
void switch_values(int &a, int &b) {  
    int temp;  
    temp = a;  
    a = b;  
    b = temp;  
};
```

可以使用指针来编写这个函数。不过上面的例子的好处在于不用使用指针运算符(& 和 *)就可以调用该函数。例如:

```
int x = 1, y = 2;  
switch_values(x,y);    //在执行完调用后,x=2,y=1。
```

在使用引用时,要记住参数引用传递一个指针,不过隐藏了指针的使用。(这有点象 BASIC 和 FORTRAN 中的通过引用进行传递。)类似地,返回值引用是函数返回一个指针。在使用诸如复制构造函数以及赋值运算符函数时引用是十分有用,甚至是十分必要的。这也是 C++ 中添加了引用的原因。详细内容,请参看第六、七章。

运行时类型信息(run-time type information(RTTI))

ANSI C++ 使用两种方式支持运行时类型信息(RTTI)。一种方式是通过使用 typeid 运算符。关于这个运算符的详细内容,请参看第十一章。另外一种方式是通过使用 dynamic_cast 运算符检查参数指针指向的对象的类,该运算符对指针参数进行运行时类型检查。关于这个类型转换运算符的详细内容,请参看第十二章。在这两种方法中,可以使用 RTTI 在运行时获得对象类型的信息。有些编译器,如 Microsoft Visual C++ 需要打开某个编译器选项才能支持 RTTI。

作用域(scope)

变量或函数的作用域是它们在程序中的可见范围。在 C 中,最重要的作用域类别是局部(local)和全局(global)。(在多模块的程序中,作用域也可以是外部的(external)。)局部变量只是在调用它的函数中是可见的。另外,每个复合语句也是一种作用域;在语句块内部定义的变量对这个语句块来说是局部的。C++ 添加了另外一种重要的作用域类别——类——这种作用域适用于类的数据成员以及成员函数。这些成员之间是可见的——成员函数可以访问任何数据成员——如果这些成员是公共的(public),也可以通过对象引用来对它们进行访问。

在 ANSI 中,在 if 条件语句中定义的变量的作用范围是 if 语句。详细内容,请参看附录 B。

存储类(storage class)

变量或对象的存储类决定了编译器如何在内存中分配变量:在数据段中、在堆栈中或是在寄存器中。具有自动存储类的变量在堆栈中进行分配,所以不是永久的。一旦函数返回,变量的值就会丢失。需要注意的是很可能存在具有局部作用域的静态存储类,就是将局部变量声明为 `static`。C/C++ 的存储类有 `auto`、`extern`、`register` 和 `static`。关于这些关键字的详细内容,请参看第十三章。

符号扩展(sign extension)

将有符号整数类型赋值给更大类型的时候就会出现符号扩展。详细内容请参看二进制补码。这种过程是在计算机内部进行的,除非需要进行不同大小的无符号和有符号数的混合编程,否则无须考虑符号扩展。进行不同大小的无符号和有符号数的混合编程时,需要特别注意零扩展以及符号扩展的影响。例如,考虑有符号数-1和无符号数 255。这两个数的位都是全为 1(二进制的 11111111),在分别将它们赋值给整型数时,将得到不同的结果。如果想对符号扩展的影响进行控制,就要在赋给更大的值之前,需要将值的类型转换为 `signed` 或 `unsigned`。

标识(signature)

标识可以按照名称、类型以及在有函数时按照函数的参数列表来唯一地识别出某个符号。不过参数的名称与标识无关。函数 `init(int i)` 与函数 `init(int j)` 具有相同的编译器无法识别的标识。不过函数 `init(int)` 与函数 `init(double)` 却是不同的。在生成 `.obj` 文件时(或 `.o` 文件,这与平台有关),编译器使用修饰(`decorating`)来输出符号名称及其标识。不要对符号名称进行什么假设,因为它们与程序的实现有关。

语句(statement)

语句与表达式是 C/C++ 程序中可执行的基本单元。这二者之间的关系十分紧密;在表达式末尾添加一个分号(`;`),表达式就变成了语句。另外,语句包括诸如 `if`、`while` 以及 `for` 语句这样的控制结构。C/C++ 语句可以是递归的。例如,单独的一个 `if` 语句可以由许多更小的语句构成。(请参看复合语句。)

赋值语句没有什么特殊的用法。在 C 和 C++ 中,赋值语句就是一个由分号结尾的表达式,例如 `x=y`。赋值语句也可以有如下的用法:

expression ;

一个 C++ 程序基本是由一系列的指令、声明以及函数定义构成的。函数定义又由一系列的语句构成。请参看可执行语句。

字符串(string)

字符串又称文本字符串或包含字符数据的数组。C++ 对字符串的支持允许对字、句子以及其它文本进行处理。请参看第三章。关于字符串库函数的详细内容,请参看第十五章的“字符串”主题。关于特殊字符的编写,请参看第十章的“字符”主题。

完全字符串(string literal)

完全字符串是由双引号(")括起来的文本构成。它是 C++ 中最主要的常量类型。(请参看常量。)完全字符串的例子如下:

```
"Hello there, C++!"
```

关于特殊字符的编写,请参看第十章的“字符”主题。

C++ 的编译器按照特殊方式处理完全字符串。首先,编译器在常量数据区中分配空间并在其中放置字符串文本。然后,编译器用数据的地址来代替程序代码中的字符串(但是并不改变源程序本身)。这种处理方法与 C/C++ 将所有的字符串作为字符数组来处理一样,等于用第一个元素的地址对数组进行引用。

子类(subclass)

请参看派生类。

超类(superclass)

请参看基类。

文本字符串(text string)

请参看字符串。

模板(template)

模板是包含通用类型的一个声明。这种通用类型,可以用一个标识符如 T 来表示,代表后来指定的某个类型。在声明之后,模板就可以用于产生特殊的声明,可以用实际类型来代替 T。例如,下面的模板声明创建了一个名为 pair 的通用数据结构,这个结构包含类型 T 的两个实例:

```
template <class T>
class pair {
    T a, b;
}
```

一旦编译器读到这个声明,模板 pair 就可以用来产生特殊的类型, pair<int>, 如下所示:

```
pair<int> jeans;
```

数据结构 jeans 现在是一个包含两个整数 a 和 b 的类。关于模板的详细内容,请参看第十三章的模板。

二进制补码(two's complement format)

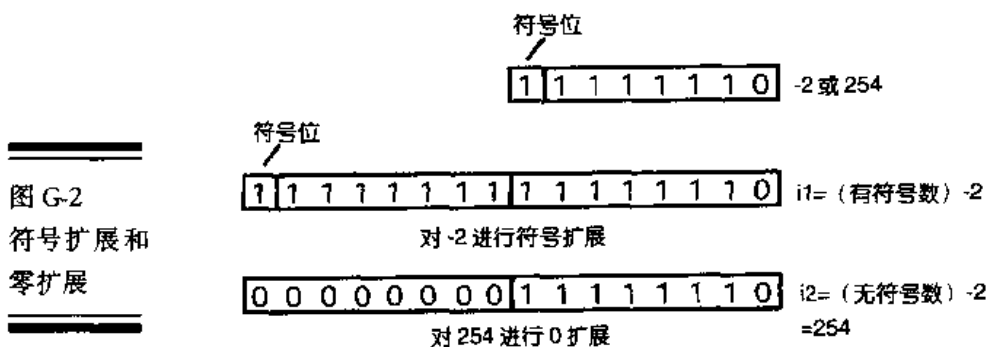
现代小型计算机以及几乎所有的个人计算机都使用补码来表示有符号整数的负值。这种格式的负数是按照下面的规则产生的：

1. 对相应正数按位取反；例如，要获得-1，首先将 00000001 变为 11111110
2. 加 1；例如，将 11111110 变为 11111111

现代计算机系统或者用全 1 来表示任何大小的整数类型的-1，或者用最左位的 1 来表示该值是负值。在有符号类型中，前一半数据是负值。在 2 字节长的整数中，最左位如果为 0，则表示的值可以从 0 到 32,767。如果最左位为 1，表示的值从-32,768 到-1。

可以无须考虑补码运算的细节。编译器产生相应的机器码指令对每种类型进行操作。大多数机器指令对有符号以及无符号类型进行同样的处理。在某些情况中，编译器要对这两种类型生成不同的指令。

例如在图 G-2 中，按照数值是负值还是正值对同样的位进行不同的处理。在将无符号数赋值给更大的整数类型时，总是进行零扩展，也就是在左边添加 0。有符号数在左边是添 1 还是添 0 取决于最左边的位的值。有符号数的这种处理方法称为符号扩展。

**类型安全连接(type-safe linkage)**

与 C 语言不同，C++ 支持类型安全连接——防止在不同模块中对同一个变量或函数名进行不一致的声明的一种措施。在 C 中，很容易将同一个变量在一个模块中声明为 short 而在另外一个模块中声明为 long。C++ 为了进行成功的连接，必须防止这种事情发生。关于类型安全连接是如何实现的详细内容，请参看“修饰”(decorating)。

无符号数据类型 (unsigned data types)

C 和 C++ 都支持有符号和无符号数据类型。默认情况下整数是有符号的;可以使用关键字将整数声明为无符号的。无符号整数最容易理解。它或者是 0 或者是一个正数。无符号类型的一个缺点是它不能表示负数。不过正是因为如此,无符号类型才可以比有符号的同类型表示的数多两倍。例如,无符号 short 的范围是从 0 到 65,536 而不是从 -32,768 到 32,767。

关于有符号类型在机器码中是如何处理的内容,请参看“二的补码”。

变量 (variable)

变量是一个指定名称的位置,该位置用于存储数据。除非将变量声明为 const,否则变量可以多次改变。在某些规则范围内,可以使用任何名称作为变量的名字。(请参看标识符。)在 C++ 中,变量可以包括对象——对象是类的实例——以及指针或者原始数据。注意在 C 和 C++ 中,变量在使用之前必须进行声明。

虚基类函数 (virtual base class)

如果基类是虚的,无论从基类继承的派生类是如何实现的,对象继承的只是基类成员的一个复本。如果这样说有些模糊,这并不奇怪。虚基类只是出现在复杂的继承层次中,至少要有经过三代派生以及多重继承。除非必须使用虚基类的特点,否则无须掌握虚基类。

关于对基类使用 virtual 关键字的例子,请参看第十三章。

虚函数 (virtual function)

虚函数是可以由不同类按不同方式定义的函数。无论对象的访问方式如何,虚函数都可以正确地执行。甚至在通过基类指针或通过不同类型的对象数组进行对象访问时,也可以执行正确的函数。或者说每个对象都知道如何正确执行虚函数。从而无须考虑访问方式。在对象概念中可以独立运行的自治单元来说,这种特性十分重要。这些自治单元类似于细胞。另外,可以将新的类型的对象加入到已经存在的框架中。这个对象带有一个指向自己的虚函数的指针。尽管可以对没有声明为 virtual 的函数进行覆盖,但是这样做有些冒险(通常会出现错误)。一般来说,任何在派生类中需要覆盖的成员函数必须在基类中声明为虚的。

无忧书库书籍版权申明

无忧书库提供的所有书籍、资料版权归原作者和出版商所有。

本站提供下载的书籍仅供个人学习、参考之用，任何集体、个人不得用于商业用途。

请您在下载书籍 24 小时之后删除书籍，购买正版书籍！

本站书籍如有侵犯您的版权，请与我们联系，我们将尽快删除。联系信箱：
pcbook@51soft.com。

无忧书库
<http://pcbook.51soft.com>
2000 年 4 月 13 日