

目 录

简 介.....	1
第一周浏览.....	3
第一章 入门(第一天).....	5
1.1 C++程序基础	5
1.2 装载 Turbo C++ IDE	6
1.3 Turbo C++ IDE 概述	6
1.4 File 菜单	7
1.5 Edit 菜单	10
1.6 Search 菜单.....	11
1.7 Run 菜单	14
1.8 Compile 菜单	15
1.9 Project 菜单	16
1.10 Browse 菜单	17
1.11 Options 菜单	18
1.12 Windows 菜单	20
1.13 Help 菜单	20
1.14 EasyWin 应用程序	20
1.15 退出 IDE	23
1.16 小结	23
1.17 问与答	24
1.18 专题讨论	24
第二章 C++程序的组成(第二天)	26
2.1 Turbo C++的预定义数据类型	26
2.2 Turbo C++的命名项	27
2.3 #include 伪指令.....	28
2.4 说明变量.....	28
2.5 说明常量.....	30
2.6 函数说明和函数原型定义.....	33
2.7 函数的局部变量.....	35
2.8 函数的静态变量.....	37
2.9 内联函数.....	38
2.10 函数退出	39
2.11 缺省变量	39
2.12 函数重载	41

2.13	小结	43
2.14	问与答	44
2.15	专题讨论	45
第三章	运算符和表达式(第三天)	47
3.1	算术运算符	47
3.2	算术表达式	49
3.3	增量运算符	50
3.4	赋值运算符	52
3.5	sizeof 运算符	54
3.6	强制类型转换	56
3.7	关系运算符和逻辑运算符	59
3.8	布尔表达式	61
3.9	位操作运算符	63
3.10	逗号运算符	65
3.11	运算过程和计算顺序	66
3.12	小结	67
3.13	问与答	68
3.14	专题讨论	68
第四章	输入/输出(I/O)管理(第四天)	71
4.1	格式化的输出流	71
4.2	输入流	72
4.3	printf 函数	74
4.4	小结	78
4.5	问与答	79
4.6	专题讨论	79
第五章	判断结构(第五天)	80
5.1	单选择的 if 语句	80
5.2	双选择的 if-else 语句	82
5.3	if 语句潜在的问题	84
5.4	多选择的 if-else 语句	84
5.5	switch 语句	87
5.6	嵌套的判断结构	91
5.7	小结	92
5.8	问与答	93
5.9	专题讨论	94
第六章	循环(第六天)	96
6.1	for 循环	96
6.2	使用 for 循环的开循环	99
6.3	do-while 循环	100

6.4	while 循环	103
6.5	跳过循环重复	104
6.6	退出循环	106
6.7	嵌套循环	107
6.8	小结	109
6.9	问与答	109
6.10	专题讨论	110
第七章	数组(第七天)	113
7.1	一维数组的说明	113
7.2	一维数组的使用	114
7.3	一维数组的初始化	116
7.4	函数中的数组参数	119
7.5	数组的排序	122
7.6	数组的查找	125
7.7	多维数组	130
7.8	多维数组的初始化	134
7.9	多维数组参数	135
7.10	小结	138
7.11	问与答	139
7.12	专题讨论	139
第一周回顾		141
第二周浏览		145
第八章	用户定义的类型和指针(第八天)	147
8.1	C++语言中的类型定义	147
8.2	枚举类型	148
8.3	结构	151
8.4	联合	154
8.5	引用变量	155
8.6	指针概述	156
8.7	指向变量的指针	157
8.8	指向数组的指针	159
8.9	指针递增/递减方法	161
8.10	指向结构的指针	162
8.11	指针和动态存储器	165
8.12	远指针	168
8.13	小结	168
8.14	问与答	169
8.15	专题讨论	170
第九章	字符串(第九天)	172

9.1 C++语言的字符串	172
9.2 字符串的输入	173
9.3 使用 STRING.H 库	173
9.4 赋值串	174
9.5 字符串的长度	175
9.6 字符串的连接	176
9.7 字符串的比较	179
9.8 变换字符串	183
9.9 倒序字符串	183
9.10 查找字符	186
9.11 查找子串	188
9.12 小结	192
9.13 问与答	192
9.14 专题讨论	193
第十章 高级函数参数(第十天)	195
10.1 使用数组作为函数参数	195
10.2 使用字符串作为函数参数	197
10.3 使用结构作为函数参数	199
10.4 由引用传递参数	201
10.5 用引用传递结构参数	201
10.6 用指针传递结构	202
10.7 递归函数	203
10.8 传递指向动态存储结构的指针	206
10.9 指向函数的指针	209
10.10 小结	216
10.11 问与答	216
10.12 专题讨论	217
第十一章 面向对象编程及 C++ 类(第十一天)	219
11.1 面向对象编程的基础	219
11.2 类与对象	219
11.3 信息和方法	220
11.4 继承	220
11.5 多态性	220
11.6 建立类	221
11.7 类的组成部分	221
11.8 构造函数	224
11.9 析构函数	227
11.10 构造函数和析构函数的应用举例	227
11.11 类层的定义	229

11.12	虚拟函数	233
11.13	虚拟函数的规则	237
11.14	友元函数	238
11.15	操作符和友元操作符	241
11.16	小结	244
11.17	问与答	245
11.8	专题讨论	246
第十二章	基本的流文件 I/O(第十二天)	248
12.1	C++ 语言流库	248
12.2	通用流 I/O 函数	248
12.3	文本文件的顺序流 I/O	250
12.4	二进制文件顺序流 I/O	253
12.5	随机存取文件流 I/O	258
12.6	小结	262
12.7	问与答	263
12.8	专题讨论	263
第十三章	ObjectWindows 基础(第十三天)	265
13.1	通用的 Windows 数据类型	265
13.2	ObjectWindows 数据类型约定	266
13.3	ObjectWindows 层次	267
13.4	Windows API 函数	294
13.5	激活 Windows API 函数	297
13.6	Windows 信息	299
13.7	响应信息	302
13.8	信息传递	304
13.9	用户定义的信息	304
13.10	小结	305
13.11	问与答	306
13.12	专题讨论	306
第十四章	建立基本的 ObjectWindows 应用程序(第十四天)	307
14.1	设计一个最小的 ObjectWindows 应用程序	307
14.2	扩展窗口操作	312
14.3	增加一个菜单	315
14.4	对菜单选择作出响应	321
14.5	建立多个事例	327
14.6	小结	334
14.7	问与答	334
14.8	专题讨论	335
第二周回顾	337

第三周浏览.....	341
第十五章 Windows 的对象(第十五天).....	343
15.1 创建一个只读的文本窗口.....	343
15.2 滚动文本.....	347
15.3 改变滚动条的尺寸设置.....	348
15.4 小结.....	355
15.5 问与答.....	355
15.6 专题讨论.....	355
第十六章 ObjectWindows 控制(第十六天).....	357
16.1 静态文本控制.....	357
16.2 编辑框.....	359
16.3 按钮控制.....	364
16.4 命令行计算器.....	367
16.5 小结.....	378
16.6 问与答.....	379
16.7 专题讨论.....	379
第十七章 组合控制(第十七天).....	380
17.1 复选框控制.....	380
17.2 单选按钮控制.....	382
17.3 组合钮控制.....	383
17.4 更新的计算器应用程序.....	385
17.5 小结.....	399
17.6 问与答.....	440
17.7 专题讨论.....	400
第十八章 列表框(第十八天).....	401
18.1 列表框控制.....	401
18.2 简单列表操作测试程序.....	406
18.3 处理多选项的列表.....	414
18.4 小结.....	422
18.5 问与答.....	422
18.6 专题讨论.....	422
第十九章 滚动条和组合框(第十九天).....	424
19.1 滚动条控制.....	424
19.2 递减计时器.....	426
19.3 组合框控制.....	433
19.4 组合框作为历史列表框.....	435
19.5 COCA 版本 3 应用程序.....	435
19.6 小结.....	448
19.7 问与答.....	448

19.8 专题讨论.....	448
第二十章 对话框(第二十天).....	450
20.1 构造对话框.....	450
20.2 运行模态对话框.....	450
20.3 控制数据的基本转换.....	455
20.4 数据转换的例子.....	458
20.5 小结.....	471
20.6 问与答.....	471
20.7 专题讨论.....	471
第二十一章 MDI Windows(第二十天)	473
21.1 MDI 应用程序特征和部件	473
21.2 建造 MDI 应用程序的基础	473
21.3 TMDIFrame 类	474
21.4 建造 MDI 框架窗口	476
21.5 TMDIClient 类.....	476
21.6 建造 MDI 子窗口	477
21.7 管理 MDI 消息	477
21.8 简单的文本浏览器.....	478
21.9 小结.....	483
21.10 专题讨论	483
第三周回顾.....	485
附录 A 控制资源脚本.....	489
A.1 对话框资源	489
A.2 DIALOG 选项语句	490
A.3 对话框控制资源	491
A.4 一般的控制资源	492
A.5 LTEXT 语句.....	493
A.6 RTEXT 语句	493
A.7 CTEXT 语句.....	494
A.8 CHECKBOX 语句.....	494
A.9 PUSHBUTTON 语句	494
A.10 DEFPUSHBUTTON 语句	495
A.11 LISTBOX 语句	495
A.12 GROUPBOX 语句	495
A.13 RADIOBUTTON 语句	496
A.14 EDITTEXE 语句	496
A.15 COMBOBOX 语句	496
A.16 SCROLLBAR 语句.....	497
附录 B 答案.....	498

B. 1	第一章,“入门”问题答案	498
B. 2	第二章,“C++程序的组成”问题答案	498
B. 3	第三章,“运算符和表达式”问题答案	500
B. 4	第四章,“输入/输出(I/O)管理”问题答案	501
B. 5	第五章,“判断结构”问题答案	503
B. 6	第六章,“循环”问题答案	506
B. 7	第七章,“数组”问题答案	508
B. 8	第八章,“用户定义的类型和指针”问题答案	510
B. 9	第九章,“字符串”问题答案	512
B. 10	第十章,“高级函数参数”问题答案	515
B. 11	第十一章,“面向对象编程及 C++类”问题答案	519
B. 12	第十二章,“基本流文件 I/O”问题答案	520
B. 13	第十三章,“ObjectWindows 基础”问题答案	522
B. 14	第十四章,“建立基本的 ObjectWindows 应用程序”问题答案	522
B. 15	第十五章,“Windows 对象”问题答案	522
B. 16	第十六章,“ObjectWindows 控制”问题答案	522
B. 17	第十七章,“组合控制 ”问题答案	523
B. 18	第十八章,“列表框”问题答案	523
B. 19	第十九章,“滚动条和组合框”问题答案	523
B. 20	第二十章,“对话框”问题答案	523
B. 21	第二十一章,“MDI Windows”问题答案	524

简介

本书有两个主要目的：第一是教你用 C++ 语言编程，第二是教你使用 Turbo C++ for Windows 建立 Windows 应用程序。它不要求你以前有任何编程经验。但是，如果你知道用其他语言编程，比如 BASIC 或 Pascal，那么对你的学习肯定会有所帮助的。本书不是为懦弱的读者而编写的，因为学会用 C++ 语言编程和用 C++ 语言编写 Windows 应用程序并不是两件很简单的事情！

这本书共包含二十一章，你每天可以学习一章。因为该书的目的就是从某种程度上加快你学习的速度。书中的每一章都包含一个问与答章节、一个测验章节和一个练习章节。附录 B 包含了所有测验的答案和一些练习的答案。

第一天简要地介绍了开发 C++ 程序所需要的 Windows 环境——Turbo C++ IDE，同时还向你展示了第一个 C++ 程序，用来说明一个非 Windows C++ 程序的基本组成部分。

第二天更详细地介绍 C++ 程序的各部分。这章讨论了变量、常量和函数的命名与说明，并且文中首先集中说明了 C++ 函数，因为它们是重要的程序构件。

第三天介绍各种 C++ 运算符和表达式。运算符使你能够操作数据，形成支持更复杂的数据操作的表达式。

第四天讨论格式化的流输入与输出，以及著名的 printf 函数。后面的这个函数支持各种用途的格式化输出。

第五天描述了 C++ 的判断结构。这些结构包括了各种 if 语句，以及 switch 语句。

第六天讨论 C++ 循环语句，它们包括 for, do-while, while 循环语句，并且还说明了如何使用 for 循环语句作为一个开循环。另外，本章讨论了如何跳过循环，退出循环和嵌入循环。

第七天介绍 C++ 中的数组。这章包含了一维数组和多维数组，并且讨论了如何说明和初始化它们。另外，该章还讨论了对一维数组的排序与搜索。

第八天讨论了用户自定义类型和指针。这章包含了枚举数据类型、结构、联合、参量和指针。本章说明了如何声明和使用简单变量、数组、结构及动态内存的指针。

第九天集中在字符串和从 C 语言中继承的 STRING.H 库。这章包括了像分配、连接、比较、转换及颠倒字符串这样的主题。另外，该章还讨论了如何在字符串中寻找字符和子串。

第十天讨论高级函数参数，主要包括数组、字符串、结构及指向函数的指针的参数。这章还讨论了把结构作为参数传递的各种方法，并且介绍了递归函数。

第十一天向你介绍了面向对象编程(OOP)的世界。这章介绍了 OOP 的基础和 C++ 的类。文中讨论了 C++ 类的基本组成部分，以及与使用这些部分相关的规则。

第十二天讨论基本 I/O 流文件，它是由 C++ 流库所支持的。这章包含了普通的流函数、顺序文本 I/O 流、顺序二进制 I/O 流及随机访问 I/O 流。

第十三天向你介绍使用 ObjectWindows 库(OWL)编写 Windows 应用程序。这章包括了广泛应用的 Windows 数据类型、以及 OWL 的简要概述、Windows API 函数和 Windows 的消息。

第十四天介绍非常简单的、基于 OWL 的 Windows 应用程序。这章从提供一个最小型的 OWL 程序开始,然后由它逐渐引伸到菜单,以及对鼠标点取的响应。

第十五天讨论在一个窗口中描绘文本。这章介绍了不滚动窗口和滚动窗口,并说明了如何在窗口中绘制文本(像图形那样)。

第十六天介绍 OWL 库的类,它们是静态文本控制、编辑控制和按钮控制的模型。这章还介绍了一个特殊的、面向命令行的计算器,作为使用这些控制的一个例子。

第十七天介绍作为复选框控制、单选按钮控制和成组控制模型的 OWL 库类。这章显示了如何组织这些控制,并且介绍了计算器应用程序的更新版。

第十八天讨论作为列表框模型的 OWL 库类。这章讨论了单选项列表框和多选项列表框。同时,该章的程序也说明了这两种列表框。

第十九天介绍作为滚动条控制和组合框控制模型的 OWL 库类。这章还讨论了如何使用组合框建立过去事情的记载框。另外,该章还介绍了使用组合框的计算器应用程序。

第二十天介绍建立和使用对话框。这章向你显示了如何使用资源文件来定义原型对话框和非原型对话框。另外,该章还讨论了对话框和它的父窗口之间的数据传输。

第二十天介绍多文本界面(MDI)窗口。这章介绍了支持 MDI 应用程序的类,并说明了如何管理 MDI 子窗口。

附录 A 介绍了用来创建菜单和对话框的资源描述语句。附录 B 包含了所有测验的答案和一些练习的答案。

注意:本书包含了讲解各方面编程的 Windows 程序,它超越了平常使用的各种可视化控制的方面。一定要认真研究这些程序;它们包含了可以丰富你的 Windows 编程的技术和诀窍。通过察看这些例子(包括特殊的例子)并向朋友请教其中不懂的问题

第一周浏览

学习编写 Windows 应用程序旅程的第一周是以介绍 Turbo C++ 环境—IDE(集成开发环境)开始。在这一周的剩余的几天里向你介绍 C++ 语言的基础。你可以学习到预定义的数据类型、常量、变量和函数的命名、C++ 运算符和表达式、基本输入和输出管理、判断、编写循环、以及说明和使用数组。因此,这周覆盖了 C++ 语言的所有基本组成部分。

第一章 入门(第一天)

欢迎你进入 C++ 和 Windows 编程世界。今天开始进入了那激动人心的冒险旅程。今天的课程的大部分信息是使你熟悉 Turbo C++ 集成开发环境 (IDE)。你将会学到下面的内容:

- ☐ C++ 程序的基础和历史。
- ☐ 如何装载和使用 Turbo C++ IDE。
- ☐ IDE 中的各种菜单。
- ☐ EasyWin 应用程序。
- ☐ 输入和运行你的第一个 C++ 程序。

1.1 C++ 程序基础

用本书学习 Turbo C++ 编程,不需要任何预备知识;但是,如果你以前用 Turbo C++ 编过程序,那么,一切事情就很简单了。像其他语言一样,C++ 也是由说明和语句组成的,这些说明和语句在程序运行时,指定了正确的执行指令。

C++ 语言是由贝尔实验室的 Bjarne Stroustrup 开发的。该语言试图取代目前流行的 C 语言,并在 C 语言基础上通过增加面向对象的语言向最高峰发展。

新术语:面向对象(object-oriented)语言表示对象的属性和操作。

另外,C++ 语言对不是面向对象的 C 语言作了大量的增强工作。然而,学会 C++ 语言也有助于你对 C 语言的进一步了解。但是,并不是像标准 C 语言一样,C++ 语言还要经受标准化的过程。

用 C++ 编程要求你必须了解那些支持库,它们用来完成各种不同的任务,比如输入/输出、文本操作、数学运算、文件 I/O(输入/输出)等等。而在像 BASIC 这样的语言中,对这样一些操作的支持显而易见是由程序实现的,也就是这些程序本身能够自动完成这些操作。所以,许多程序相互并列作为组成部分,它们独立于任何其他的编程部分。相反,用 C++ 编程使你更加了解一个程序对各种库的依赖性。这种语言特征的优点是,你可以在类似的库—包括你自己开发的库—之间进行选择。所以,C++ 程序是模块化的。C++ 编辑器—包括 Turbo C++—使用工程文件和程序文件。而 Turbo C++ IDE 使用工程文件来处理程序的创建和更新。

新术语:工程文件(project files)指定库。程序文件(program files)创建应用程序。

1.2 装载 Turbo C++ IDE

Turbo C++ IDE 是作为 C++ 编译器、连接器、调试器,以及其他用来创建、管理和维护 C++ 程序工具的可视化界面。你可以通过简单地单击 Turbo C++ 图标或者从 File Manager 中双击 TCW.EXE 程序来装载 IDE(文件 TCW.EXE 放在 \TCWIN\BIN 目录中)。

1.3 Turbo C++ IDE 概述

Turbo C++ IDE 是从属于 MDI 的应用程序,它有以下几个主要组成部分:

- ☐ 带有菜单系统、最小化图标和最大化图标的边框窗口。你可以缩放、移动、最大化和最小化 Turbo C++ IDE 窗口。该窗口有一个反映活动窗口名称的标题栏。
- ☐ 系统菜单提供了大量的选项。
- ☐ 速度条,包含了指定的位图按钮,用来提供特定命令的快速键。IDE 能够使你在速度条中拥有专用的位图按钮。另外,这些按钮对上下文都起作用。它们的数字和类型可以改变,这依赖于当前的任务或活动窗口。IDE 支持良好的特性,当你把鼠标移动到一个位图按钮上时,能够显示该按钮做什么用(正文出现在状态行中)。
- ☐ 用户区域,包含了各种窗口,例如源代码编辑窗口、消息窗口、变量观察窗口等等。
- ☐ 状态行位于 IDE 窗口的底部。该行在你把鼠标移动到速度条中的按钮上时,显示出简要的在线帮助,为不同的菜单项提供简要说明,显示光标位置,以及显示插入/重写模式的状况。

图 1.1 显示 Turbo C++ IDE 的界面样本。

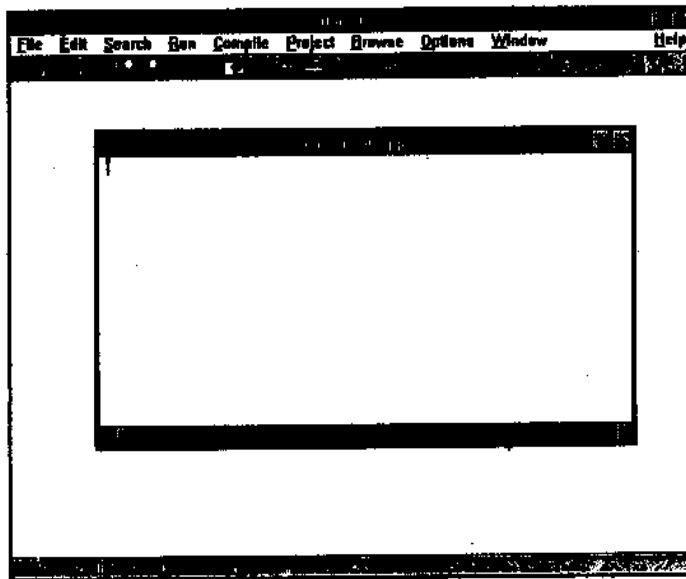


图 1.1 Turbo C++ IDE

注意:因为 IDE 目的是提供给软件开发者,所以如果你是一个初学者,那么许多选项可能对你来说是很高级的。但是,当你变的更有经验,这些选项和术语将成为你作为一个 Turbo C++ 程序员的部分知识了。

1.4 File 菜单

File 菜单提供了管理文件、打印文本和退出 IDE 的命令。表 1.1 摘要介绍了 File 菜单中的这些命令。同时,File 菜单还包含了最近打开的源文件的动态列表。

表 1.1 File 菜单中的命令摘要

命 令	快 键	功 能
New		打开一个新的编辑窗口。
Open...		把一个存在的源文件装入新的编辑窗口。
Save		存入活动编辑窗口的内容。
Save as ...		使用一个新的文件名存入活动编辑窗口的内容。
Save all		把所有打开的源代码窗口存入到它们各自的文件中。
Print...		打印源代码窗口的内容。
Print setup		设置打印机。
Exit	Alt+F4	退出 IDE。

1.4.1 New 命令

New 命令用来打开一个新的编辑窗口(也称作源代码窗口),并给它指定一个缺省的相关文件名。你打开第一个新窗口的缺省文件名是 NONAME00.CPP 你打开的第二个新窗口的缺省文件名是 NONAME001.CPP。新打开的窗口开始是空的,它具有与上次活动窗口同样的尺寸和位置。换句话说,如果上次的活动窗口是最大型的,那么新窗口也将是最大型的。

1.4.2 Open... 命令

Open...命令使你能够把一个存在的源文件的内容装入到一个新的编辑窗口。事实上,IDE 能装载多个文件。该选项首先弹出 Open a File 对话框,见图 1.2 所示。对话框含有几个列表框的组合框控制,它们能使你定位于某个源文件,然后选择它。这些控制允许你选择驱动器,目录和文件名列表,帮助你确定所寻找的源文件的位置。

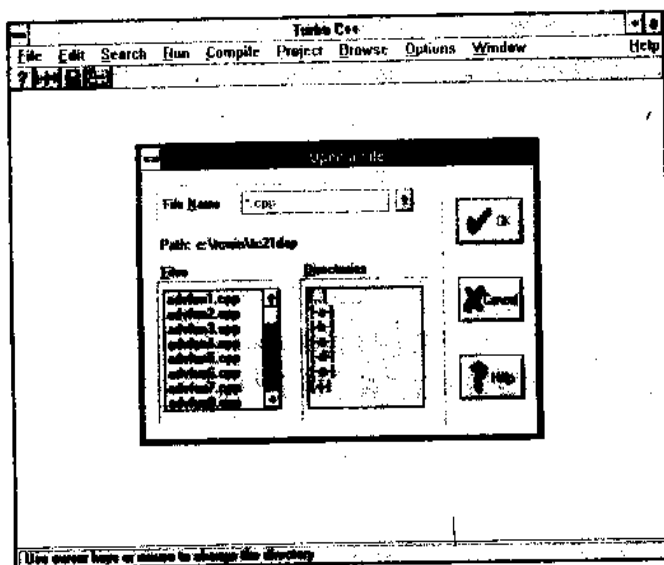


图 1.2 Open a File 对话框

1.4.2 Save 命令

Save 命令帮助你把活动编辑窗口的内容存入到它的一相关的文件里。如果你在新编辑窗口中激活这个选项,那么 Save 选项就会弹出 Save File As 对话框,见图 1.3 所示。这个对话框能够使你有选择地指定某个非缺省的文件名,同时还包括目的驱动器和目录。

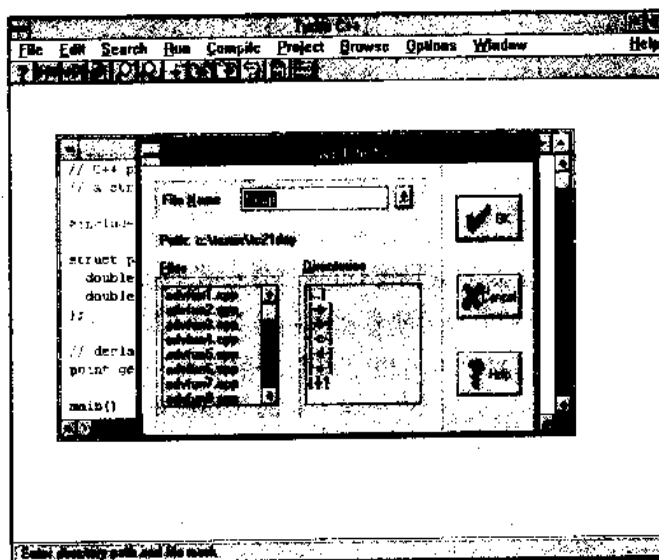


图 1.3 Save File As 对话框

1.4.3 Save as ... 命令

Save as ... 命令能够使你把活动编辑窗口的内容存入到与当前文件名不同的文件

中。事实上,新的文件名也就变成了活动窗口的新的文件。Save as ...命令弹出 Save File As 对话框,见图 1.3 所示。如果你选择一个存在的文件,那么该选项就会弹出一个消息对话框,询问你是否希望用活动编辑窗口的内容重写那个存在的文件。

1.4.4 Save all 命令

Save all 命令把所有修改过的编辑窗口的内容写入到它们各自相关的文件中。如果 IDE 含有新的编辑窗口,那么这个选项就会弹出 Save File As 对话框,存储这些新的窗口内容。

1.4.5 Print ...命令

Print ...命令能够使你打印活动编辑窗口的内容。

1.4.6 Print Setup ...命令

Print Setup ...命令能够使你在使用 Print...选项进行打印之前,首先设置好你的打印机。

Print Setup ...命令弹出 Setup 对话框,见图 1.4 所示(在这个图中的对话框是基于拥有 HP LaserJet II 系统)。这个对话框含有使你能够指定下列项目的控制。

- ☐ 纸张大小。
- ☐ 纸源。
- ☐ 打印的页数。
- ☐ 打印机内存的大小。
- ☐ 打印输出定位。
- ☐ 选择的字体库和字体。
- ☐ 保留额外内存用于打印一页的页保护。这一选项只有当你拥有 1M 以上的打印机内存时才能有效。

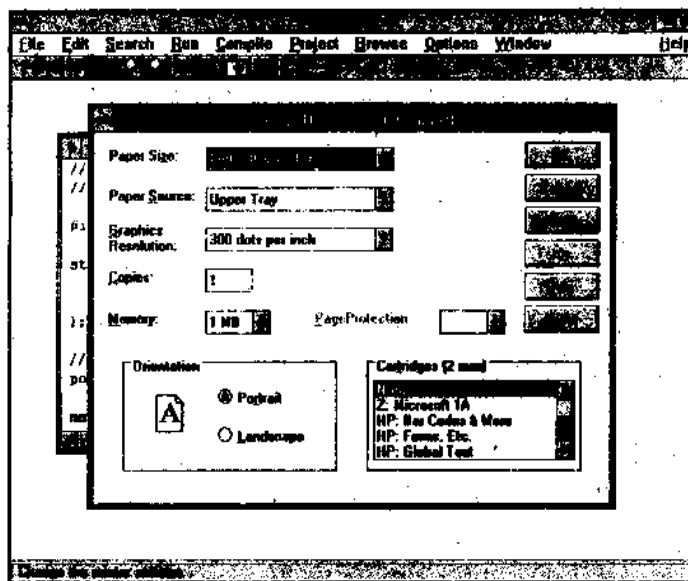


图 1.4 Setup 对话框

1.4.7 Exit 命令

Exit 命令能够使用户完全退出 Turbo C++ IDE。IDE 提示用户不能保存任何已修改编辑的窗口。

1.5 Edit 菜单

Edit 菜单含有使你能够在编辑窗口进行文本编辑的命令。表 1.2 摘要介绍了在 Edit 菜单中的命令。

表 1.2 Edit 菜单中的命令摘要

命 令	快 键	功 能
Undo	Alt + BkSp	撤消上次的编辑。
Redo	Alt+Shift+BkSp	保留上次 Undo 选项的编辑。
Cut	Shift+Del	删除选中的文本,并把它拷贝到剪裁板上。剪裁板以前的内容丢失。
Copy	Ctrl+Ins	拷贝选中的文本到剪裁板。剪裁板以前的内容丢失。
Paste	Shift+Ins	把剪裁板的内容插入到当前光标位置。
Clear	Ctrl+Del	删除选中的文本,但是不把它写到剪切板上。

1.5.1 Undo 命令

Undo 命令能够使你保存上次编辑任务的效果,并恢复活动窗口的内容。这个选项的快捷键是 Alt + BkSp。该选项使你能够快速而有效地处理编辑错误—尤其是在工作很长时间以后。

1.5.2 Redo 命令

Redo 命令能够使你保留 Undo 选项的活动。它的快捷键是 Alt + Shift + BkSp。Redo 命令能够使你在两种形式的编辑源代码之间进行转换。该命令对那些真正已经精疲力尽,不能决定源代码应该是怎样的程序员是很有益处的。

1.5.3 Cut 命令

Cut 命令用来删除选中的文本,并把放置到剪裁板上。剪裁板以前的内容被丢失掉。Cut 命令的快捷键是 Shift+Del。

1.5.4 Copy 命令

Copy 命令把选中的文本复制到剪裁板上。而剪裁板以前的内容被丢失。Copy 命令的

快捷键是 Shift+Ins。

1.5.5 Paste 命令

Paste 命令把剪裁板的内容插入到当前光标位置。而剪裁板的内容仍保持不受影响。所以你可以使用 Cut 和 Paste 命令,在同样的编辑窗口或交叉不同的编辑窗口中移动文本,你也可以使用 Copy 和 Paste 命令,在同样的编辑窗口或交叉不同的编辑窗口中复制文本块。Paste 命令的快捷键是 Shift+ Ins 。

1.5.6 Clear 命令

Clear 命令清除选中的文本,并且不把它复制到剪裁板上。这并不意味着被删除的文本将完全丢失,因为你可以使用 Undo 命令来恢复被删的文本。Clear 命令的快捷键是 Ctrl+Del 。

1.6 Search 菜单

Seach 菜单含有使你能够定位各种信息的命令,比如正文、符号定义、函数说明和程序建立出错的信息。表 1.3 摘要介绍了 Search 菜单中的命令。

表 1.3 Search 菜单中的选项摘要

命 令	快 键	功 能
Find...		寻找活动编辑窗口中的文本
Replace...		代替活动编辑窗口中的文本。
Search again	F3	重复上次的 Find 或 Replace 操作。
Go to line number		立到某上指定行。
Previous error	Alt+F7	选择先前程序建立的出错信息,并把光标定位在编辑窗口中的出错行上。
Next error	Alt+F8	选择下一个程序建立的出错信息,把光标定位在编辑窗口的出错行上。

1.6.1 Find... 命令

Find ...命令支持在活动编辑窗口中寻找文本。这个选项的快捷键是 Ctrl+QF,它能弹出 Find Text 对话框,见图 1.5 所示。这个对话框具有下面的控制:

- ☐ Text to find 组合框控制,它能使你或者在寻找文本中敲入,或者调用最近寻找过的文本。
- ☐ Options 复选框,包括下面几项:
 - ☐ Case Sensitire 复选框,能够使你选择相关型的或者相关型的文本搜寻。
 - ☐ Whole Words only 复选框,能够使你在匹配全部或者匹配字任何文本之间进行

选择。

- ☐ Regular expression 复选框, 打开或关闭 BRIEF 编辑器的正规表达特征的使用。这样的表达式将造成使用 Text to find 控制中的文本作为正式文本模型。

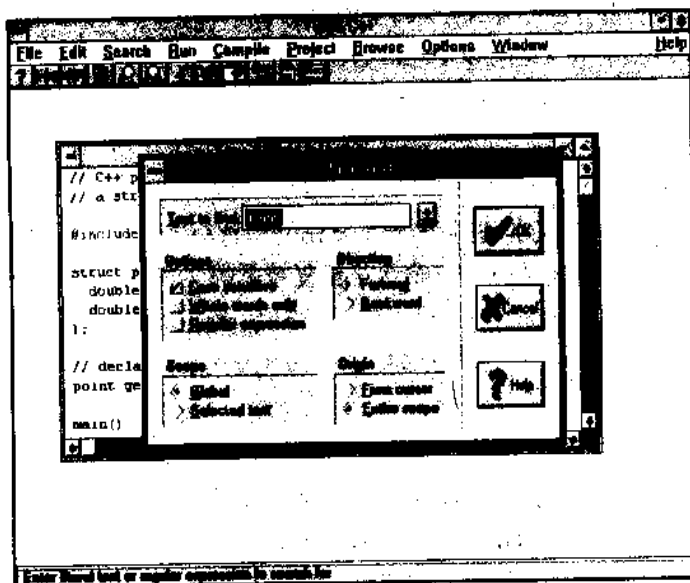


图 1.5 Find Text 对话框

- ☐ Direction diamond-shaped 单选钮控制。这些控制能够使你选择向前搜寻或者向后搜寻。
- ☐ Scope diamond-shaped 单选钮控制。这些控制能够使你在搜寻全部文本或者限制搜寻选中文本之间进行选择。
- ☐ Origin diamond-shaped 单选钮控制。这些控制能够使你在搜寻整个编辑窗口或者从光标的位置开始搜寻之间进行选择。
- ☐ OK、Cancel 和 Help 按钮。

1.6.2 Replace...命令

Replace...命令支持在活动编辑窗口中进行文本替换。这个选项拥有快捷键 Ctrl+QA, 它能弹出 Replace Text 对话框, 见图 1.6 所示。这个对话框含有下面的控制:

- ☐ Text to find 组合框控制, 它能够使你直接敲入要寻找的文本或者调用最近寻找的文本。
- ☐ New text 组合框, 它能够使你直接输入用来代替的文本或选择最近使用过的替换文本。
- ☐ Options 复选框包括下面几项:
 - ☐ Case sensitive 复选框, 它能使你选择相关型的或非相关型的文本搜寻。
 - ☐ Whole words only 复选框, 它能使你在匹配全部字或者匹配任何文本之间进行选择。
 - ☐ Regular expression 复选框, 打开或关闭 BRIEF 编辑器的正规表达形式的运用。

这样的表达形式将造成使用 Text to find 控制中的文本作为文本模型。

- ☐ Prompt on replace 复选框,使你能够选择是否不经你确定就进行文本替换。
- ☐ Direction diamond-shaped 单选钮控制。这些控制允许你选择是向前搜寻还是向后搜寻。
- ☐ Scope diamond-shaped 单选钮控制。这些控制使你能够在搜寻全部文本或者限制搜寻选中的文本之间进行选择。
- ☐ Origin diamond-shaped 单选钮控制。这些控制使你能够选择,是搜寻整个的编辑窗口还是从光标位置开始搜寻。
- ☐ Change All 按钮,它能够使你替换所有的匹配文本。相反,如果你单击 OK 按钮,那么你将只能替换下一个匹配的文本。
- ☐ OK、Cancel 和 Help 按钮。

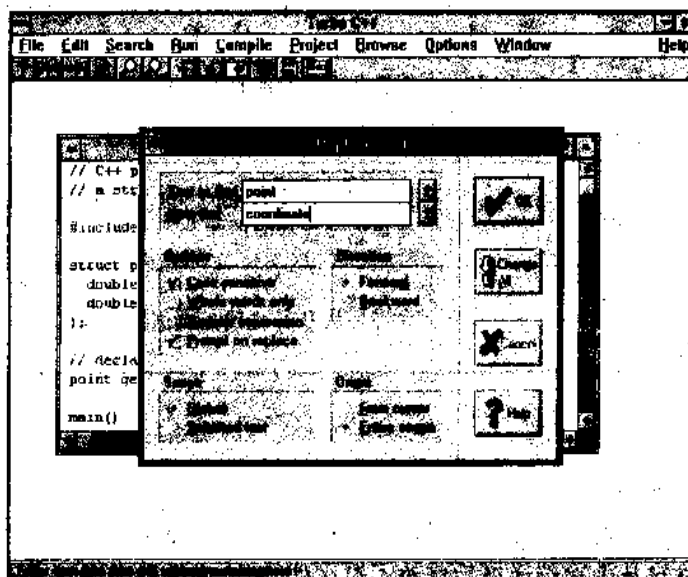


图 1.6 Replace Text 对话框

1.6.3 Search Again 命令

Search Again 命令使你能够重复上次的 Find...或者 Replace...选项。该选项的快捷键是 F3 功能键。

1.6.4 Go to Line Number 命令

Go to line number 命令能弹出简单的 Go to Line Number 对话框,它允许输入行数或者从输入组合框中选择一个以前输入的行数。

1.6.5 Previous Error 命令

Previous error 命令使你能够缩放出错的源文件行,该行是与先前在 Message 窗口中的出错信息相联系的。IDE 通过显示含有该出错行的编辑窗口来反映这个选项。这个命令

的快捷键是 Alt+F7。

1.6.6 Next Error 命令

Next error 命令使你能够缩放与 Message 窗口中的下一个出错信息相联系的出错源文件行。IDE 通过显示含有该出错行的编辑窗口来反映这个选项。这个命令的快捷键是 Alt+F8。

1.7 Run 菜单

Run 菜单提供了运行你的程序和启动 Turbo Debugger for Windows 的命令。表 1.4 摘要介绍了 Run 菜单中的命令。

表 1.4 Run 菜单的命令摘要

命 令	快 键	功 能
Run	Ctrl+F9	编辑和运行当前的工程文件。
Arguments		允许你从 IDE 内输入命令行数。
Debugger		启动 Turbo Debugger for Windows, 以便你能调试你的程序。
Debugger arguments		允许你从 IDE 内输入调试器参数。

1.7.1 Run 命令

Run 命令用来运行你的程序,它可以使用由 Run | Arguments 命令传递给它的任何参数。并且如果你自从上次编译以后修改了源文件,Run 命令就会激活 Project Manager,重新编译和连接的程序。该命令的快捷键是 Ctrl+F9。

1.7.2 Arguments 命令

Arguments 命令能够弹出 Program Arguments 对话框,你可以在运行你的程序之前输入命令行参数。IDE 能够准确地处理这些参数,就好像你已经在 DOS 命令行上输入了它们。

1.7.3 Debugger 命令

Debugger 命令能够激活 Turbo Debugger for Windows,以便你能调试你的程序,并且确定任何逻辑或运行错误的位置。Turbo C++ IDE 可以通知 Turbo Debugger 要调试哪个程序。

1.7.4 Debugger Arguments 命令

Debugger arguments 命令能够弹出 Debugger Arguments 对话框,它允许你把参数输

入到调试器。

1.8 Compile 菜单

Compile 菜单提供了编译活动窗口中的程序,以及建立你的工程文件的命令。表 1.5 摘要介绍了 Compile 菜单的命令。

表 1.5 Compile 菜单的命令摘要

命 令	快 键	功 能
Compile	Alt+F9	把活动文件编译成 .OBJ 文件。
Make	F9	更新工程文件。
Link		连接工程文件。
Build All		重新建立工程文件,不管它们是否更新。
Information		显示当前文件的信息。
Remor Messages		撤除 Message 窗口。

1.8.1 Compile 命令

Compile 命令用来把活动的源文件(C 或者 .CPP 文件)编译成 .OBJ 文件。这时,IDE 会弹出一个 Compile Status 信息框,显示出编译进展情况和结果。当编译过程结束时,单击 OK 按钮。如果发生错误,IDE 就会显示出 Message 窗口,并把第一个错误高亮度显示。

1.8.2 Make 命令

Make 命令能够激活 Project Manager 来制立目标工程文件。列出的 .EXE 文件名是由两个名称衍生出来的,这两个名称的顺序如下:

1. 由 Project | Open Project 命令指定的工程文件(.PRJ)。
2. 活动编辑窗口中的文件名。

如果没有定义工程文件,那么你就会获得由文件 TCDEFW.DPR 定义的缺省工程文件。Make 命令可以只重新建立那些当前的文件。

1.8.3 Build All 命令

Build All 命令重新建立用户工程文件中的所有文件。这个命令与 Compile | Make 命令相似,只是除为它能重新建立当前工程文件中的所有文件或非当前的文件以外。如果你通过按 Esc 键或单击 Cancel 按钮取消 Build 命令,或者如果你出错而停止了建立,那么你可以通过激活 Compile | Make 命令来恢复继续建立。

1.8.4 Link 命令

Link 命令用来连接当前工程文件中定义的文件,或者缺省工程文件中定义的文件。

1.8.5 Information 命令

Information 命令弹出 Information 对话框,显示出有关当前文件的统计信息。

1.8.6 Remove Message 命令

Remove message 命令从 Message 窗口中清除所有的信息。

1.9 Project 菜单

Project 菜单提供了创建、打开或关闭一个工程文件的命令,增加或删除一个工程文件中的文件的命令,以及察看工程中一个指定文件所包含的文件的命令。表 1.6 简要介绍了 Project 菜单的命令。

表 1.6 Project 菜单的命令摘要

命 令	快 键	功 能
Open project		打开一个新的或已存在的工程文件,并自动关闭当前的工程文件。
Close project		关闭当前的工程文件。
Add item		添加一个文件到工程文件列表中。
Delete item		从工程文件列表中删除一个文件。
Include files		显示出一个列有所包含文件的对话框。

1.9.1 Open Project 命令

Open project 命令显示出 Open Project File 对话框,你可以在其中选择和装载一个存在的工程文件,或者创建一个新的工程文件。Open Project File 对话框与 Open File 对话框相似,只是除了前者的对话框专用于打开工程文件以外。

1.9.2 Close Project 命令

Close Project 命令用来撤消你当前的工程文件,而返回到缺省的工程文件 TCDEF.DPR。

1.9.3 Add Item 命令

Add Item 命令弹出 Add to Project List 对话框,你可以在其中把一个文件增加到工程文件列表中。Project List 对话框与 Open File 对话框相似。

1.9.4 Delete Item 命令

Delete item 命令用来删除 Project 窗口中选中的文件。当 Project 窗口是活动的时，一个文件，然后按下 Del 键，删除那个文件。

1.9.5 Include Files 命令

Include files 命令弹出 Include File 对话框，见图 1.7 所示。这个对话框显示出你从 Project 窗口选择的那个文件所包含的文件列表。

如果你没有建立一个工程文件，那么 IDE 就使 Include files 命令无效。

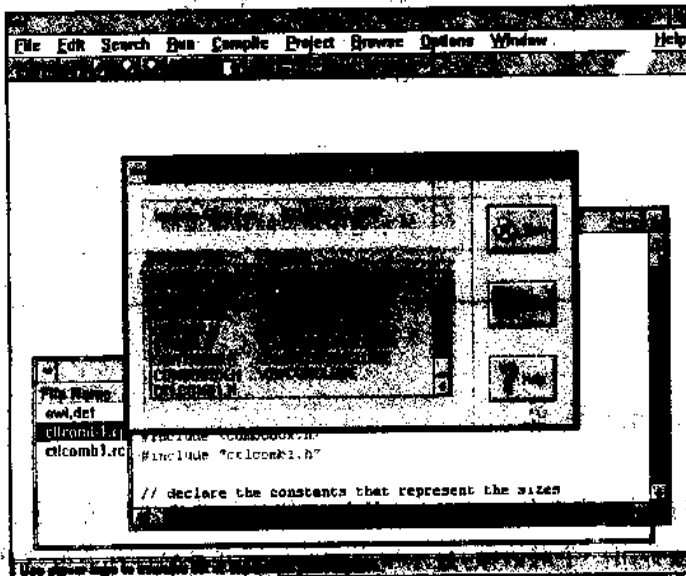


图 1.7 Include File 对话框

1.10 Browse 菜单

Browse 菜单提供了激活 ObjectBrowser 的命令，ObjectBrowser 允许你浏览各级类、函数和变量。一定要记住在你使用 ObjectBrowser 以前，需要首先编译你的程序。如果你编译的程序是由多种源文件组成，那么你在使用 ObjectBrowser 以前，必须打开 IDE 中相关的工程文件。表 1.7 摘要介绍了 Browse 菜单的命令。因为这些命令相对都是比较高级的，所以我显示的这个表将作有限的讨论。

表 1.7 Browser 菜单的命令摘要

命 令	快 键	功 能
Classes		显示出 ObjectBrowser 窗口, 该窗口显示了各级类。
Functions		显示出一个列有程序中所有函数的窗口。
Variables		显示出一个列有程序中所有变量的窗口。
Symbol at Cursor		检测符号(类, 函数或变量)。
Rewind		把 Object Browser 返回到先前的观察点。
Overview		显示类或类等级的概述。
Inspect		显示选中项目的详细说明。
Goto		把你带到选中项目的源代码中。

1.11 Options 菜单

Options 菜单提供了察看和改变 C++IDE 中的各种缺省设置的命令。在这个菜单中的大多数命令都能弹出一个对话框, 显示出当前的设置并且允许你改变这些设置。表 1.8 摘要介绍了 Options 菜单的命令。因为这些命令都是相对比较高级的, 所以显示这个表只是作了有限的讨论。

表 1.8 Options 菜单的命令摘要

命令	快捷键	功能
Application...		显示出 Application Options 对话框。使用这个对话框来设置编译和连接一个标准 Windows 可执行文件还是一个 Windows DLL(动态连接库)。

续表

命 令	快 键	功 能
Compiler		<p>显示一个列有附加命令的子菜单。其中的大多数命令都调用选择影响代码编译的选项。子菜单的命令是：</p> <ul style="list-style-type: none"> —Code Generation —Advanced Code Generation —Entry /Exit Code —C++ Options —Advanced C++ Options —Optimizations —Source —Messages —Names
Make ...		显示出 Make 对话框,你可以从中指定工程文件管理条件。
Linker		<p>显示出一个含有附加命令的菜单。这些命令能够弹出对话框,你可以从其中选择较好的连接形式。这些命令是：</p> <ul style="list-style-type: none"> —Settings —Libraries
Librarian ...		显示出 Librarian Options 对话框,你可以从其中选择协调使用内部库管理程序较好的设置。这个库管理程序能够把你工程文件中的 .OBJ 文件加入到一个 .LIB 文件中。
Directories...		弹出 Directories 对话框。它允许你选择 Turbo C++ IDE 运行和存储你的程序时可使用的目录。
Resources...		显示出 Resources 对话框。使用这个对话框来指定如何编译资源和加入到你的程序中,同时包括设置执行程序的最新版的 Windows。
Environment		<p>显示一个含有附加命令的弹出式菜单。这些命令都能弹出对话框,你在其中对环境进行设置。这些命令是：</p> <ul style="list-style-type: none"> —Preferences —Editon —Highlight —Mouse —Desktop
Save...		弹出 Save Options 对话框,你可在其中存储 C++ 环境,桌面平台和工程文件的设置。

1.12 Windows 菜单

Windows 菜单提供了在 IDE 用户区域中管理窗口的选项。这些选项在表 1.9 中作了摘要介绍,它们允许你排列、关闭、最小化和恢复一些或全部窗口。创造了标准选项以外,Windows 菜单还列有当前的窗口。

表 1.9 Windows 菜单的命令摘要

命 令	快 键	功 能
Cascade	Shift+F5	在 IDE 用户区域中级联式地排列窗口。
Tile	Shift+F4	把窗口水平放在 IDE 用户区域上。
Arrange uons		把图标排列在 IDE 用户区域中。
Close all		关闭所有的窗口,调试器窗口、浏览器窗口、 以及编辑器窗口。
Message		显示 Message 窗口。
Project		显示 Project 窗口。

1.13 Help 菜单

Help 菜单向你提供,各种可能在别的软件中也用到过的在线帮助。表 1.10 摘要介绍了 Help 菜单的这些选项。

表 1.10 Help 菜单的选项摘要

命 令	快 键	功 能
Index	Shift+F1	显示在线帮助系统的内容表
Topic Search	Ctrl+F1	显示有关光标所处在的那个键的帮助信息。
Using help		显示使用在线帮助系统中帮助你的信息。
About...		显示有关软件版本和版权的信息。

1.14 EasyWin 应用程序

Turbo C++ IDE 使你能够建立一个称作 EasyWin 应用程序的特殊类型的程序。这个应用程序介于 MS-DOS 程序和 Windows 程序之间,从今天到本书的第十二天,所有的程序都是 EasyWin 应用程序,它使你能够使用一个像 DOS 一样的接口和输入/输出过程,

来专心学习 C++ 语言。EasyWin 窗口是 C++ 程序(作为 EasyWin 应用程序编译)的标准输入和输出。

由单个 .CPP 源文件来创建 EasyWin 应用程序,需要进行下列步骤:

1. 装载 Turbo C++ IDE.
2. 输入源代码,并把它存储在一个非缺省的 .CPP 文件名下,或者装入一个 .CPP 文件。
3. 激活 Run|Run 命令(你可以直接按 Ctrl+F9 键)Turbo C++ IDE 就编译、连接并运行你的程序。紧接着这个程序就会出现在一个 EasyWin 窗口中。
4. 当程序停止时,EasyWin 窗口就显示出带有 inactive 前缀的可执行文件名。使用窗口的系统菜单中的 Close 命令(或按 Alt+F4 键)可以关闭 EasyWin 窗口。

本书中出现的第一个 C++ 程序显示了一行问候信息。这个简单的程序使你能够看到 C++ 程序的非常基本的组成部分。

程序列表 1.1 包含了程序 HELLO.CPP 的源代码,它标有行号。当你真正敲入这个程序时,不要输入行号进去。这些行号只是作为参考。这个简单程序显示 'Hello Programmer!' 字符串,你可以按照下列步骤来创建和运行这第一个 C++ 程序。

1. 装入 Turbo C++ IDE,如果它没有装载的话。
2. 输入源代码(不要输入行号和跟在它后面的号)。
3. 激活 File|Save as 命令,把这个程序存储在 HELLO.CPP 名称下。
4. 按下 Ctrl+F9 键,编译、连接和运行程序。

当 EasyWin 程序结束时,执行系统修改这个程序窗口的标题,把 inactive 包括进去。要关闭这个程序的窗口,可以选择系统菜单中的 Close 命令,或者简单地按下 Alt+F4 键。

程序列表 1.1 程序 HELLO.CPP 的源文件

```
1: // a trivial C++ program that says hello
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     cout << "Hello Programmer!";
8:     return 0;
9: }
```

图 1.8 中显示了该程序的输出。注意,输出窗口的标题是以 inactive 开始,用来表示该程序已经终止执行。

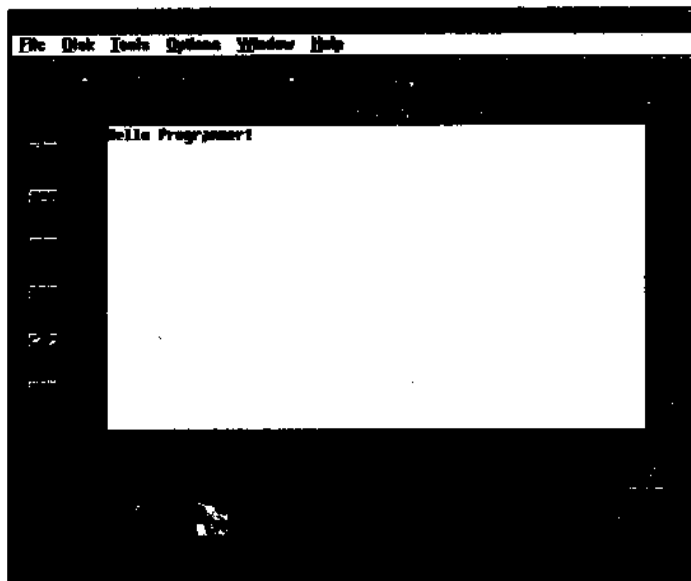


图 1.8 HELLO. EXE 程序的输出

分析:

检查 C++ 程序简短的码,可以注意到下列特点:

- ☐ C++ 使用 `//` 字符开头,一直到这行的结束都是作为注释。C++ 也支持 C 类型的注释,是以 `/*` 字符开始,并以 `*/` 字符结束。第一行包含了一条注释,简要地说明了这个程序。

新术语:Comments(注释)是你放在程序中用来解释或说明程序确定部分的陈述。编译器忽略注释。

- ☐ C++ 程序不保留任何说明程序结束的关键字。事实上,C++ 使用了一种简单的方案来组织程序。这个方案支持两种级别的代码,全局函数和低级函数。另外,在第 5 行开始的函数 `main` 起着一个非常关键的作用,因为程序执行是以这个函数开始的。所以,在一个 C++ 程序里,只能有一个 `main` 函数。你可以把函数 `main` 放在源文件中的任何地方。
- ☐ C++ 的字符串和字符分别要用双引号和单引号括起来。所以 `'A'` 表示一个字符,而 `"A"` 表示一个字符串,把 C++ 的单字符串和字符混淆在一起是错误的。

新术语:Strings(串)可以有任意多个字符,包括汉字字符。一个汉字字符的串称作 `empty string`(空串)。

- ☐ C++ 使用 `{` 和 `}` 字符定义块。分别见第 6 行和第 9 行的例子。
- ☐ 在 C++ 程序中,每条语句必须以分号 `;` 结束。
- ☐ C++ 包含有 `#include` 编译器指令。这个例子在第 3 行,指示 Turbo C++ 编译器

要包含 **IOSTREAM.H** 头文件。**C++** 扩充了原来存在于 **C** 中的流定义。**IOSTREAM.H** 提供了支持基本输入与输出流的操作。**C++** 语言没有包含固有的 I/O 路径。而是依靠各种类型的 I/O 专用库。

新术语: **compiler directive** (编译器指令) 是编译器的专用指令。**header file** (头文件) 包含有常量、数据类型和变量的说明以及函数的预先说明。**stream** (流) 是从计算机的某部分流向另一部分的一个数据队。

- ☐ **C++** 程序把字符串 **Hello Programmer!** 输出到标准输出流 **cout**, 也就是 **EasyWin** 窗口。另外, 程序使用提取器操作符 **<<**, 把输出的串输送到输出流。
- ☐ 函数 **main** 必须返回一个值, 反映 **C++** 程序的出错状况。如果返回 0 值到操作系统则程序正常终止。

1.15 退出 IDE

要退出 **IDE**, 选择 **File** 菜单中的 **Exit** 命令。

1.16 小结

今天的课程向你介绍了 **Turbo C++ IDE**, 并向你展示了第一个 **C++** 程序。你学习了这些基础知识:

- ☐ **C++** 程序是模块化的, 并依靠标准和专用库。
- ☐ 装载 **Turbo C++ IDE** 的两种方法是: 单击 **Turbo C++** 图标或者当你用 **File Manager** (或类似的工具) 时, 双击 **BCM.EXE** 文件。
- ☐ **Turbo C++ IDE** 是用于开发、维护和调试 **C** 与 **C++** 程序, **MS DOS** 库以及 **Windows** 应用程序的多功能环境。
- ☐ **File** 菜单从事创建新文件、打开文件、存储文件、打印文件和退出 **IDE**。
- ☐ **Edit** 菜单提供执行编辑操作的选项(比如: **undo**, **cut**, **copy**, **paste** 和 **delete**)。
- ☐ **Search** 菜单使你能够寻找和替换文件, 以及浏览各类符号, 定位函数和访问出错的源文件行。
- ☐ **Run** 菜单提供了运行和调试的应用程序的命令。这些命令包括提供命令行参数给你的程序和调试器的命令。
- ☐ **Compile** 菜单提供了编译、连接、文件和建立你的应用程序的命令。
- ☐ **Project** 菜单支持允许你创建一个新工程文件、打开一个存在的工程文件和关闭一个工程文件的命令, 以及增加和删除当前工程文件中的源文件的命令。
- ☐ **Browse** 菜单提供了让你浏览类、函数和变量的命令。
- ☐ **Options** 菜单使你能够协调好你工程文件的各方面: 环境、工具和工程文件风格。
- ☐ **Windows** 菜单用于管理、排列、关闭和恢复 **IDE** 工作平台中的窗口。
- ☐ **Help** 菜单提供了在线帮助。

- ☐ **EasyWin** 应用程序是提供专用窗口的 **Windows** 应用程序,它用作标准输入和输出设备。**EasyWin** 应用程序允许你编写像 **DOS** 一样的程序。
- ☐ 本书中的第一个 **C++** 程序是一个简单的问候程序,它用来说明一个 **C++** 程序的基本组成部分。这些部分包括注释, **#include** 指令和 **main** 函数。
- ☐ 你可以通过 **File** 菜单中的 **Exit** 选项退出 **IDE**。

1.17 问与答

问:**C++**使用行号吗?

答:不用。我们在本书程序列表中使用行号是为了参考。

问:**IDE** 编辑器检查输入的东西吗?

答:是的。事实上,当你输入一个 **C++** 关键字时,**IDE** 会迅速地改变它的颜色。

问:如果忘记在第一个程序中敲入第二个双引号,会发生什么事呢?

答:编译器会告诉你,程序中有一个错误。你需要加上第二个双引号,并建立这个工程文件。

问:我如何删除当前编辑窗口中的文本?

答:使用 **Edit** 菜单中 **Replace** 选项,并且指定空作为替换的串,或者使用 **Edit** 菜单的 **Cut** 和 **Clear** 命令。

1.18 专题讨论

专题讨论提供了测验题目,帮助你强化对所学资料的理解,同时,还提供了练习,帮助你增加使用你所学的东西的经验。在继续下一课学习以前,努力去理解测验和练习答案附录 **B** 提供了所有答案。

1.18.1 测验

1. 下面程序的输出什么?

```
1: //quiz program #1
2:
3: #include <iostream. h>
4:
5: main()
6: {
7:   cout <<<"C++ in 21 Days?";
8:   retrun 0;
9: }
```

2. 下面程序的输出什么?

```
1: // quiz program #2
2:
```



```

3: #include <iostream.h>
4:
5: main()
6: {
7:     //cout <<"C++ in 21 Days?";
8:     return 0;
9: }

```

3. 下面程序的错误在哪?

```

1: //quiz program #3
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     cout <<"C++ in 21 Days?"
8:     return 0;
9: }

```

1.18.2 练习

编写一个显示信息 **I am a C++ Programmer.**

第二章 C++程序的组成（第二天）

第一天介绍了 Borland IDE 和一个简单的 C++ 程序。今天,我们集中讨论 C++ 程序的基本组成部分,包括数据类型、变量、常量和函数。

你将学到以下内容:

- ☐ Turbo C++ 中的预定义数据类型
- ☐ Turbo C++ 中的项目命名
- ☐ #include 指令
- ☐ 变量说明
- ☐ 常量说明
- ☐ 函数说明和原型说明
- ☐ 函数中的局部变量
- ☐ 函数中的静态变量
- ☐ 嵌入函数
- ☐ 退出函数
- ☐ 缺省参变量
- ☐ 函数过重

2.1 Turbo C++ 的预定义数据类型

Turbo C++ 提供了 int、char、float、double 和 void 数据类型以分别表示整数、字符、单精度浮点数、双精度浮点数和无值数据。C++ 对一个函数的返回值使用 void 类型,表示该函数是作为一个过程。

C++ 支持数据类型的修饰符,从而对数据类型增加了更大的柔性,这些修饰符如下:signed、unsigned、short 和 long。表 2.1 显示了 C++ 的预定义数据类型(包括类型修饰符),以及它们以字节大小和范围。注意,int 和 unsigned int 是依赖于系统的。该表是显示了这些数据类型的 16 位值。

新术语: Data type modifiers (数据类型修饰符)改变数值的精度和范围。

表 2.1 C++ 的预定义数据类型

Data Type	Byte Size	Range	Examples
char	1	-128 to 127	'A', '1'
signed char	1	-128 to 127	23
unsigned char	1	0 to 255	200, 0x1a
int	2	-32768 to 32767	3000
unsigned int	2	0 to 65535	0xffff, 65535
short int	2	-32768 to 32767	100
unsigned short int	2	0 to 65535	0xff, 40000
long int	4	-2147483648 to 2147483647	0xffffffff, -123456,
unsigned long int	4	0 to 4294967295	123456
float	4	3.4E-38 to 3.4E+38 and -3.4E-38 to -3.4E+38	2.35, -52.354, 1.3e+10
double	8	1.7E-308 to 1.7E+308 and -1.7E-308 to -1.7E+308	12.354, -2.5e+100, -78.32544
long double	10	3.4E-4932 to 1.1E+4932 and -1.1E-4932 to -3.4E+4932	8.5e-3000

新术语: C++ 支持 hexadecimal numbers (十六进制数)。这些数字以字符 0x 开始, 后面跟有十六进制数值。例如, 数字 0xff 是等于十进制数 255 的十六进制数。

2.2 Turbo C++ 的命名项

Turbo C++ 要求你遵守下面的标识符规则:

1. 第一字符必须是一个字母或下划线(_).
2. 接下去的字符可以是字母, 数字或下划线。
3. 一个标识符的最大长度为 32 个字符。
4. 标识符在 C++ 中是区分大小写的。因此, 名称 rate、RATE 和 Rate 指的是三个不同的标识符。
5. 标识符不能是保留字, 比如 int, double, 或 static。

下面是有效标识符的例子：

```
X
x
aString
DAYS_IN_WEEK
BinNumber0
bin_number_0
bin0Number2
_length
```

DO	DON'T
DO 要使用具有适当长度的描述名。	
DON'T 不要使用太短或太长的标识符名。短名称会造成很差的可读性,长的名称容易输入错。	

2.3 #include 伪指令

伪指令是C和C++编译器的特殊指令。一个伪指令以#字符开始,后面跟有伪指令名。伪指令通常放在一行的第一列中。它们前面可以放有空格符和制表符。在第一天的C++程序含有#include伪指令。这个伪指令告诉编译器,要把一个文件的正文包含进去,就好像你自己已经敲进去了那个正文。所以,#include伪指令的替换方式要比从一个文件中剪裁正文,再把它粘贴到另一个文件中的方式好。回想第一天的那个程序,它使用#include伪指令把头文件包含进去。

语法

#include 伪指令

#include 伪指令的一般语法是

```
#include <filename>
```

```
#include "filename"
```

例子:

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

```
#include "string.h"
```

文件名表示包含文件的名称。#include伪指令搜寻包含的文件的方法有两种不同的形式。第一种形式是在指定目录里搜寻那个文件。第二种形式把搜寻扩展到当前目录。

2.4 说明变量

说明变量要求你标出变量的数据类型和变量名。单词variable表示你可以改变这些数据存放器的数据。

新术语: Variable(变量)是用来存储和调用信息的标识符。你可以把一个变量看作是一个有标号的数据存放器。

语法

说明变量

说明变量的一般语法是

```
type variableName;  
type variableName = initialValue;  
type var1 [= initVal1], var2 [= initVal2], ...;
```

例子:

```
int j;  
double z = 32.314;  
long fileSize, diskSize, totaleSize = 0;
```

C++允许你在一条说明语句中说明一系列变量(有相同类型)。例如:

```
int j, i = 2, k = 3;  
double x = 3.12;  
double y = 2 * x, z = 4.5, a = 45.7;
```

初始化值可以包含以前定义的其他变量。

DO	DON'T
DO 尽量不要使用全局变量。	
DON'T 不要在同一个程序里使用大小写不同的字符名称说明变量(比如 rate 和 Rate)。	

让我们来看看一个简单的例子。程序列表 2.1 显示了程序 VAR1.CPP 的源代码。这个程序说明了四个变量,其中两个在它们的说明中就进行了初始化。然后,该程序给未初始化的变量分配值,并显示所有四个变量的内容。最后,编译和运行 VAR1.EXE 程序。

程序列表 2.1 程序 VAR1.CPP 的源代码

```
1: // C++ program that illustrates simple variables  
2:  
3: #include <iostream.h>  
4:  
5: main()  
6: {  
7:     int i, j = 2;  
8:     double x, y = 355.0 / 113;  
9:  
10:    i = 3 * j;  
11:    cout << "i = " << i << "\n"  
12:         << "j = " << j << "\n";  
13:
```

```

14:  x = 2 * y;
15:  x = x * x;
16:  cout << "y = " << y << "\n"
17:      << "x = " << x << "\n";
18:  return 0;
19:
20: }

```

下面是程序列表 2.1 中的程序输出部分：

输出：

```

i = 6
j = 2
y = 3.141593
x = 39.47824

```

分析：

这个程序在第 3 行使用了 `#include` 伪指令，用来包含流 I/O 头文件 `IOSTREAM.H`。函数 `main` 出现在第 5 行。这个函数包含了第 7 行的 `int` 类型变量 `i` 和 `j` 的说明，以及第 8 行的 `double` 类型变量 `x` 和 `y` 的说明。说明初始化变量 `j` 和 `y`。第 10 行语句把变量 `j` (值为 2) 的值乘以 3，并把结果存放在变量 `i` 中。第 11 行和第 12 行的输出流语句用来显示变量 `i` 和 `j` 的值。该语句还包括显示标识输出对象的字符中。

第 14 行语句把变量 `y` 的值加倍，并存放在变量 `x` 中。第 15 行语句把变量 `x` 的值进行乘方，并把结果重新分配给变量 `x`。这个语句在等号的两边都使用了变量 `x`。第 16 行和第 17 行的输出流语句用来显示变量 `x` 和 `y` 的值。第 18 行语句返回 0，作为函数 `main` 的结果。

2.5 说明常量

许多语言，比如 BASIC (比较新的实用语言)，Modula-2，C，Pascal，And 和 C++，都支持常量。没有人可否认，通过用更具有描述性的常量标识符取代常数，能够增强程序的可读性。而且，使用常量使你通过只改变某一处参数的值，就能够改变该参数在程序中各处的值。这个性能肯定是比较简单的，而且不容易出错。然而，当你利用文本编辑器来把某些数字用其他数字代替时，这些错误可能发生。

新术语：常量 (constants) 是具有固定值的标识符。C++ 以两种风格提供常量：基于宏的和形式的。基于宏的常量是从 C 语言中继承下来的，使用 `#define` 编译器指令。

语法

`#define` 伪指令

`#define` 伪指令的语法是

```
#define constantName constantvalue
```

`#define` 伪指令引起编译器激活前置处理器和执行文本替换,把基于宏的常量用它们的值来代替。这个文本替换步骤发生在编译器处理源文件中的语句之前。因此,编译器从没有见过基于宏的常量本身,而仅仅是它们扩充到的地方。

例子:

```
#define ASCII_A 65
```

```
#define DAYS_IN_WEEK 7
```

C++的第二种常量类型是正规常量。

语法

形式常量

形式常量的一般语法是

```
const dataType constantName = constantValue;
```

`dataType` 项是一个选项,它用来指定常量值的数据类型。如果你省略数据类型,那么C++编译器就假定为 `int` 型。

例子:

```
Const unsigned char ASCII_A = 65;
```

```
Const DAY_IN_WEEK = 7;
```

```
Const char FIRST_DISK_DRIVE = 'A';
```

DO	DON'T
DO 常量要使用大写字母名。这种命名风格使你能快速判断一个标识符是否是常量。	DON'T 不要假定读你的代码的其它人知道嵌入的数字代码的含义是什么。一定要用常量说明以增强你的程序的可读性。

2.5.1 使用基于宏的常量

让我们看看一个使用基于宏的常量的例子。程序列表 2.2 显示了程序 `CONST1.CPP` 的源代码。这个程序首先提示你输入从午夜 0 点钟开始的小时、分钟和秒数。然后,该程序进行计算,并显示从午夜 0 点钟开始的总的秒数。按下 `Ctrl+F9` 键,编译并运行 `CONST1.EXE` 程序。

程序列表 2.2 程序 `CONST1.CPP` 的源代码

```
1: // C++ program that illustrates constants
2:
3: #include <iostream.h>
4:
5: #define SEC_IN_MIN 60
6: #define MIN_IN_HOUR 60
7:
8: main()
9: {
10:     long hours, minutes, seconds;
11:     long totalSec;
12:
13:     cout << "Enter hours: ";
14:     cin >> hours;
```

```

15:  cout << "Enter minutes: ";
16:  cin >> minutes;
17:  cout << "Enter seconds: ";
18:  cin >> seconds;
19:
20:  totalSec = ((hours * MIN_IN_HOUR + minutes) *
21:             SEC_IN_MIN) + seconds;
22:
23:  cout << "\n\n" << totalSec << " seconds since midnight";
24:  return 0;
25: }

```

下面是程序列表 2.2 中的程序输出部分:

输出:

```

Enter hours: 10
Enter minutes: 0
Enter seconds: 0
36000 seconds since midnight

```

分析:

这个程序在第 3 行第 5 行和第 6 行含有 `#include` 伪指令,用来说明基于宏的常量 `SEC_IN_MIN` 和 `MIN_IN_HOUR`。这两个常量的值为 60,但是每个值有不同的意义。函数 `main` 从第 8 行开始,说明了四个 `long` 型变量: `hours`, `minutes`, `seconds` 和 `totalSec` (分别第 10、11 行)。

该函数使用了成对的语句来输出提示信息 and 接收输入。第 13 行含有的输出流语句提示你输入小时数。第 14 行含有该输入流语句。标识符 `cin` 是标准输入流的名称;使用插入操作符 `>>` 从键盘读取数据,并把它存放在变量 `hours` 中。第 15 行到第 18 行的输入和输出语句执行同样的任务:提示输入和获取键盘的输入。

第 20 行含有计算从午夜 0 点钟开始的总秒数,并把结果存放在变量 `totalSec` 中的语句。该语句使用了宏常量 `MIN_IN_HOUR` 和 `SEC_IN_MIN`。像你所看见的那样,对比使用数字 60 代替这两个常量,使用宏常量增强了该语句的可读性。第 23 行含有一条输出流语句,显示从午夜开始的总秒数(存放在变量 `totalSec` 中),后面还跟有文本描述,用来阐明输出。

2.5.2 形式常量的使用

现在,让我们看看新版的程序,它使用了 C++ 的形式常量。程序列表 2.3 显示了程序 `CONST2.CPP` 的源代码。这个程序的作用像程序 `CONST1.CPP` 一样。按下 `Ctrl+F9` 键,编译并运行 `CONST2.EXE` 程序。

注意:这时,我假定你熟悉了创建 `.CPP` 源文件,创建 `.PRJ` 工程文件和在工程文件中增加 `CPP` 文件的过程。从现在起,我将提及如何创建这些文件,除非在一个工程文件中有一组特殊的源文件。

程序列表 2.3 程序 CONST2.CPP 的源代码

```
1: // C++ program that illustrates constants
2:
3: #include <iostream.h>
4:
5: const SEC_IN_MIN = 60; // global constant
6:
7: main()
8: {
9:     const MIN_IN_HOUR = 60; // local constant
10:
11:     long hours, minutes, seconds;
12:     long totalSec;
13:
14:     cout << "Enter hours: ";
15:     cin >> hours;
16:     cout << "Enter minutes: ";
17:     cin >> minutes;
18:     cout << "Enter seconds: ";
19:     cin >> seconds;
20:
21:     totalSec = ((hours * MIN_IN_HOUR + minutes) *
22:                 SEC_IN_MIN) + seconds;
23:
24:     cout << "\n\n" << totalSec << " seconds since midnight";
25:     return 0;
26: }
```

下面是程序列表 2.3 中的程序输出部分:

输出:

```
Enter hours: 1
Enter minutes: 10
Enter seconds: 20
4220 seconds since midnight
```

分析:

程序列表 2.2 和程序列表 2.3 中的程序是类似的。它们之间的不同之处在于如何说明它们的常量。在程序列表 2.3 中,我使用了形式 C++ 常量语法来说明常量。另外,我在第 5 行中说明的常量 SEC_IN_MIN 是在函数 main 的外面。这种说明使得该常量变成了全局常量。也就是说,如果程序中有另一个函数,它也能够使用这个常量 MIN_IN_HOUR。所以,常量 MIN_IN_HOUR 是函数 main 的局部常量。

2.6 函数说明和函数原型定义

大多数编程语言都使用函数和过程。而 C++ 不支持形式过程。不过,所有的 C++ 程序都是函数!

新术语:函数(Functions)是主要的建造块,这些块从概念上扩充了 C++ 语言,使它适合你的专用程序。

语法

函数说明

ANSI C 说明函数的风格(它被 C++ 沿用)的一般形式是

```
returntype functionName(typedefparameterlist)
```

例子:

```
double sqr(double y)
{return y * y;}
char prevChar(char c)
{return c - 1;}
```

在说明 C++ 函数时,请记住下面规则:

1. C++ 函数的返回类型出现在函数名之前。
2. 如果参数表是空的,那么你就要使用空的圆括号。C++ 也允许你使用关键字 void 选项,用来清楚地说明参数表是空的。
3. 输入的参数表是由使用了下列一般格式的参数组成:

```
[const] type1, parameter1, [const] type2 parameter2, ...
```

这种格式表明,单个参数是像变量一样说明——你首先说明类型,然后说明参数的标识符。在 C++ 中,参数表是用逗号分隔的。另外,你不可以把一队有完全相同的数据类型的参数组成一组。你必须清楚地说明每个参数。如果一个参数有 const 从句,那么编译器就会确保函数不改变那个参数的自变量。

4. C++ 函数的主体包含在一对大括号({})中。并且最后关闭的那个括号后面没有分号。
5. C++ 支持通过值或引用传递自变量。缺省情况下,参数通过数值传递它们的自变量。所以,函数具有复制数据,并保护原始数据的作用。如果要说明一个引用参数,则要在该参数的数据类型后面插入一个 & 字符。一个引用参数以别名作为它的自变量。并且引用参数所作的任何改变也都会影响到该自变量。引用参数的一般形式是:

```
[const] type1& parameter1, [const] type2& parameter2, ...
```

如果是一个参数有 const 从句,那么编译器就确保函数不改变该参数的自变量。

6. C++ 支持局部常量、数据类型和变量。这些数据项可以出现在嵌套的块语句中,但是 C++ 不支持嵌套的函数。
7. return 关键字返回函数的值。
8. 如果函数的返回类型是 void,那么你不必使用 return 关键字,除非你需要在函数中间退出来。

新术语: C++ 指定了你可以使用它之前或者说明或者定义函数。说明函数一般叫作 prototyping(原型说明),它列出了函数名、返回类型以及参数的数目和类型。并且参数名是可选择的。最后你还需要在那个关闭的括号后面放置一个分号。C++ 要求,如果你在定义函数之前想调用它,那么你就要首先说明这个函数。

下面是一个简单的函数原型说明的例子:

```
// prototype the function square
```

```
double sqr(double);
main()
{
    cout << "5 2 =" << sqr(5) << "\n";
    return 0;
}
double sqr(double z)
{return z * z;}
```

注意,函数 `sqr` 的说明仅仅包含了它单个参数的类型。

典型地,函数说明是全局性的。但是你可以在它的用户函数中对它进行原型说明。这样,对其他函数就隐藏了它的原型。

调用一个函数要求你为它的参数提供变量。这些变量必须按照参数说明的顺序对应于相应的参数。并且它们的数据类型也必须与那些参数匹配或兼容。例如,你有一个函数 `volume`,它定义如下:

```
double volume(double length, double width, double height)
{
    return length * Width * height;
}
```

如果要调用函数 `volume`,那么你需要提供 `double` 型的变量或兼容类型的变量(在这种情况下,它们是所有的数字数据类型)。下面是许多调用函数 `volume` 的例子:

```
double len = 34, width = 55, ht = 100;
int i = 3;
long j = 44;
unsigned k = 33;
cout << volume(len, width, ht) << "\n";
cout << volume(1, 2, 3) << "\n";
cout << volume(i, j, k) << "\n";
cout << volume(len, j, 22.3) << "\n";
```

注意: C++ 允许你丢弃函数的返回值。这种函数调用运用于当你关心的是函数干什么,而不是它的返回值的时候。

2.7 函数的局部变量

完好的结构化编程技术培养这种思想:函数应该是尽可能独立的和可重复使用的。因此,函数可以有它们自己的数据类型、常量。以及给以它们这种独立性的变量。

新术语: 函数的 `local variable` (局部变量)只是当函数被调用时才存在。一旦函数终止了,执行系统就会撤消局部变量。因此,局部变量在两个函数调用之间将丢失它们的数据。另外,执行系统在函数每次被调用时都会对局部变量进行初始化。

DO	DON'T
<p>DO 可以使用局部变量存放和改变用 const 从句说明的参数的值。</p> <p>DON'T 不要说明这样的局部变量,它具有与你要在该函数中访问的全局变量同样的名称。</p>	

让我们来看看一个例子。程序列表 2.4 显示了下面这个数学函数的值:

$$f(x) = x^2 - 5x + 10$$

它的斜率为自变量等于 3.5 时的值。这个程序使用下面这个近似公式计算:

$$f'(x) = (f(x+h) - (f(x)-h))/2h$$

其中 h 是一个很小的增量。

程序列表 2.4 程序 VAR2.CPP 的源代码

```

1: // C++ program that illustrates local variables in a function
2:
3: #include <iostream.h>
4:
5: double f(double x)
6: {
7:     return x * x - 5 * x + 10;
8: }
9:
10: double slope(double x)
11: {
12:     double f1, f2, incrim = 0.01 * x;
13:     f1 = f(x + incrim);
14:     f2 = f(x - incrim);
15:     return (f1 - f2) / 2 / incrim;
16: }
17:
18: main()
19: {
20:     double x = 3.5;
21:
22:     cout << "f(" << x << ") = " << f(x) << "\n"
23:          << "f'(" << x << ") = " << slope(x) << "\n";
24:
25:     return 0;
26: }

```

下面是程序列表 2.4 的输出部分:

输出:

f(3.5)=4.75

f'(3.5)=2

分析:

程序列表 2.4 中的这个程序说明了三个函数,取名为 f(在第 5 行),slope(在第 10 行)和 main(在第 18 行)。函数 f 是非常简单的,它返回数学函数的值。并且,该函数没有局部变量。相反,函数 slope 说明了局部变量 f1, f2 和 incrim。并且该函数还初始化了最后的那个局部变量。第 13 行给局部变量 f1 分配了 f(x+incrim)的值。第 14 行把 f(x-incrim)的值分配给了局部变量 f2。第 15 行返回了函数 slope 的值。函数 main 简

单地显示了当 $x=3.5$ 时,数学函数和它的斜率的值。

2.8 函数的静态变量

在程序列表 2.4 中,函数 `slope` 的局部变量在该函数一理终止时,就失去了它们的值。C++ 允许你通过在局部变量的数据类型的左边放上 `static` 关键字,把它说明成静态的。静态变量常常是初始化过的。这种初始化是当函数第一次被调用时进行的。

新术语:有些编程技术要求在两次函数调用之间保持局部变量的值。这些特殊的局部变量被称为静态变量(static variables)。

当函数终止调用时,静态变量会保持它们的值。编译器支持这种语言特征,它在程序执行当中把静态变量存储在一个单独的内存区域。你可以在不同的函数中使用同样的静态变量名。这种重名情况不会使编译器搞混乱,因为它保存了哪个数拥有哪个静态变量的路径。

让我们来看看一个简单的程序。程序列表 2.5 使用了一个带有静态变量的函数,这些静态变量保持了一个动态均值。该程序提供了它自己的数据,并且几次调用那个函数,以便获取和显示那个动态均值的当前值。

程序列表 2.5 程序 `STATIC1.CPP` 的源代码

```
1: // C++ program that illustrates static local variables
2:
3: #include <iostream.h>
4:
5: double mean(double x)
6: {
7:     static double sum = 0;
8:     static double sumx = 0;
9:
10:    sum = sum + 1;
11:    sumx = sumx + x;
12:    return sumx / sum;
13: }
14:
15: main()
16: {
17:     cout << "mean = " << mean(1) << "\n";
18:     cout << "mean = " << mean(2) << "\n";
19:     cout << "mean = " << mean(4) << "\n";
20:     cout << "mean = " << mean(10) << "\n";
21:     cout << "mean = " << mean(11) << "\n";
22:     return 0;
23: }
```

下面是程序列表 2.5 的程序输出部分:

输出:

```
mean=1
mean=1.5
mean=2.333333
```

mean=4.25

mean=5.6

分析:

在程序列表 2.5 中的程序说明了含有静态变量的函数 mean。第 7 行和第 8 行分别说明了静态变量 sum 和 sumx。该函数把这两个静态变量都初始为 0。第 10 行的语句使变量 sum 增值 1。第 11 行语句使变量 sumx 按参数 x 的值增加。第 12 行返回更新的动态均值,它是由 sumx 除以 sum 而获得。

函数 main 发出对函数 mean 的一系列调用。第 17 行到 21 行的输出流语句显示了更新的动态均值。这些结果是由于函数 mean 中的静态变量 sum 和 sumx 而获得的。如果 C++ 不支持静态变量,你就必须采用全局变量——一种非常有问题的编程选择!

2.9 内联函数

使用函数要求首先调用它们,传递它们的变量和返回它们的结果。C++ 允许你使用内联函数扩充它们的语句。所以,内联函数以扩充代码为代价为你提供了较快的执行速度——尤其是速度比较关键的地方。

语法

内联函数

内联函数的一般语法是

```
inline returnType functionName (typedParameterList)
```

例子:

```
inline double cube(double x)
```

```
{return x * x * x;}
```

```
inline char nextChar(char c)
```

```
{return c + 1;}
```

使用内联函数的可供选择方案是使用 #define 伪指令来创建基于宏的伪函数。许多 C++ 程序员(包括我)强烈推荐放弃这种方法。这种倾向的证据是内联函数提供了类型检查,而用 #define 伪指令创建的宏都没有这项功能。

DO	DON'T
DO 当你开发你的程序时,首先必须像普通函数那样说明内联函数。非内联函数比较容易调试。一旦你的程序开始执行了,再把 inline 关键字插入到需要的地方。	DON'T 不要用太多的语句说明内联函数。因为 EXE 程序的增大是不可接受的。

下面是一个使用内联函数的程序例子。程序列表 2.6 包含了程序 INLINE1.CPP 的源代码。该程序提示你输入一个数字。然后程序进行计算,并显示出你输入的那个数字的平方的立方值。

程序列表 2.6 程序 INLINE1.CPP 的源代码

```
1: // C++ program that illustrates inline functions
2:
3: #include <iostream.h>
4:
5: inline double sqr(double x)
6: {
7:     return x * x;
8: }
9:
10: inline double cube(double x)
11: {
12:     return x * x * x;
13: }
14:
15: main()
16: {
17:     double x;
18:
19:     cout << "Enter a number: ";
20:     cin >> x;
21:
22:     cout << "square of " << x << " = " << sqr(x) << "\n"
23:         << "cube of " << x << " = " << cube(x) << "\n";
24:
25:     return 0;
26: }
```

下面是程序列表 2.6 中的程序输出部分:

输出:

```
Enter a number: 2.5
square of 2.5=6.25
cube of 2.5=15.625
```

分析:

在程序列表 2.6 中,该程序说明了内联函数 `sqr` 和 `cube`。每个函数的开头都是以关键字 `inline` 开始,内联函数的其他方面与简短的正常函数相似。函数 `main` 调用函数 `sqr` 和 `cube`,分别显示出平方值和立方值。

2.10 函数退出

你常常需要从一个函数中提前退出来,因为有些特殊的条件不允许你继续执行那行函数中的语句。C++ 提供了 `return` 语句来退出函数。如果函数是 `void` 类型,那么你就可以利用 `return` 语句,并且后面不要包含表达式。相反,如果你退出一个非 `void` 类型的函数,那么返回语句就会产生一个值,表明退出函数的意图。

2.11 缺省变量

缺省变量是一种非常简单,但又非常有用的语言特性。当你忽略含有缺省变量的参数的变量时,那个变量会自动被运用。

新术语: C++ 允许你给函数的参数指定缺省变量。

使用缺省变量要求你遵守这些规则：

1. 一旦你给一个参数指定了缺省变量，你就必须为同一个参数表中所有后续参数也这样做。你不可以随机地给参数指定缺省变量。该规则意味着，参数表可以分成两个子表：没有缺省变量的引导参数和有缺省参数的跟随参数。
2. 你必须为每个没有缺省变量的参数提供变量。
3. 对于含有缺省变量的参数，你可以忽略变量。
4. 一旦你忽略了带有缺省变量的参数的变量，所有后续参数的变量也必须忽略掉。

注意：对带有缺省变量的参数进行列表的最好方法是，根据使用它们的缺省变量的可能性来排列次序。最不可能使用的变量放在第一位，而最可能使用的变量放在最后一位。

让我们来看看一个简单的，使用了带有缺省变量函数的例子。程序列表 2.7 显示了程序 DEFARGS1.CPP 的源代码。该程序提示你首先输入两点的 X、Y 坐标。然后程序进行计算，并显示两点之间以及每点与原点(0,0)之间的距离。

程序列表 2.7 程序 DEFARGS1.CPP 的源文件

```
1:  C++ program that illustrates default arguments
2:
3:  #include <iostream.h>
4:  #include <math.h>
5:
6:  inline double sqr(double x)
7:  { return x * x; }
8:
9:  double distance(double x2, double y2,
10:                 double x1 = 0, double y1 = 0)
11:  {
12:      return sqrt(sqr(x2 - x1) + sqr(y2 - y1));
13:  }
14:
15:  main()
16:  {
17:      double x1, y1, x2, y2;
18:
19:      cout << "Enter x coordinate for point 1: ";
20:      cin >> x1;
21:      cout << "Enter y coordinate for point 1: ";
22:      cin >> y1;
23:
24:      cout << "Enter x coordinate for point 2: ";
25:      cin >> x2;
26:      cout << "Enter y coordinate for point 2: ";
27:      cin >> y2;
28:
29:      cout << "distance between points = "
30:           << distance(x1, y1, x2, y2) << "\n";
31:      cout << "distance between point 1 and (0,0) = "
32:           << distance(x1, y1, 0) << "\n";
33:      cout << "distance between point 2 and (0,0) = "
34:           << distance(x2, y2) << "\n";
35:      return 0;
36:  }
```

下面是程序列表 2.7 的程序输出部分：

输出:

```
enter x coordinate for point 1:1
enter y coordinate for point 1:1
enter x coordinate for point 2:-1
enter y coordinate for point 2:1
distance between points=2
distance between point 1 and (0,0)=1.414214
distance between point 2 and (0,0)=1.414214
```

分析:

程序列表 2.7 中的程序包含两个头文件。第 4 行使用 `#include` 伪指令,包含了 `MATH.H` 头文件,它说明了平方根函数 `sqrt`。该函数在第 6 行还说明了内联函数 `sqr`。这个函数返回参数 `X` 变量的平方值。同时,这个程序还说明了带有 4 个 `double` 型参数的函数 `distance`。参数 `x1` 和 `y1` 分别表示第二点 `X` 和 `Y` 的坐标,而参数 `x1` 和 `y1` 有缺省变量 0。该函数返回这两点之间的距离。如果你忽略了 `x1y1` 的变量,那么该函数就返回点 `(x2,y2)` 与原点 `(0,0)` 之间的距离。如果你只忽略了最后那个参数的变量,那么该函数就产生点 `(x2,y2)` 与点 `(x1,0)` 之间的距离。

函数 `main` 在第 19 行和第 26 行使用语句的提示你输入两点的 `X` 和 `Y` 坐标。第 28 行和第 29 行的输出语句调用函数 `distance`,并给它提供了四个变量 `x1`、`y1`、`x2` 和 `y2`。所以,函数 `distance` 的调用没有使用缺省变量。相反,第 30 行和第 31 行的语句调用函数 `distance`,为它仅仅提供了三个变量。这次调用函数 `distance` 使用了最后那个参数的缺省变量。第 32 行和第 33 行的语句调用函数使用了第三、四参数的两个缺省变量。并且在第二次调用函数 `distance` 中,我可以忽略第三个变量,还能够编译和运行这个程序。

2.12 函数重载

函数重载是 C++ 的一种语言特征,而在 C、Pascal 或 BASIC 中都没有类似的特征。这种新特征使你能够说明多种同名,但不同参数表的函数。函数的返回类型不属于函数署名部分,因为 C++ 允许你丢弃返回值的类型。所以,当这些返回类型被忽略时,编译器就不能区分带有相同参数和不同返回类型的两个函数。

新术语:参数表也称作函数署名(function signature)。

警告:对于重载函数,使用缺省变量可能会造成几个函数的署名相重复(当这些函数使用缺省变量时)。C++ 编译器能够检测出这种意义不明确性,并产生一个编译错误。

DO	DON'T
DO 使用缺省变量,可以减少重载函数的数目。	
DON'T 不要使用重载函数来进行不同的操作。	

让我们来看看一个简单的,使用重载函数的例子。程序列表 2.8 包含了程序 OVERLOAD.CPP 的源代码。该程序完成下面任务:

- ☐ 说明 char、int 和 double 类型的变量,并用数值对它们进行初始化。
- ☐ 显示初始值。
- ☐ 激活对变量进行增值的重载函数。
- ☐ 显示存放在变量中的更新了的值。

程序列表 2.8 程序 OVERLOAD.CPP 的源文件

```

1: // C++ program that illustrates function overloading
2:
3: #include <iostream.h>
4:
5: // inc version for int types
6: void inc(int& i)
7: {
8:     i = i + 1;
9: }
10:
11: // inc version for double types
12: void inc(double& x)
13: {
14:     x = x + 1;
15: }
16:
17: // inc version for char types
18: void inc(char& c)
19: {
20:     c = c + 1;
21: }
22:
23: main()
24: {
25:     char c = 'A';
26:     int i = 10;
27:     double x = 10.2;
28:
29:     // display initial values
30:     cout << "c = " << c << "\n"
31:         << "i = " << i << "\n"
32:         << "x = " << x << "\n";
33:     // invoke the inc functions
34:     inc(c);
35:     inc(i);
36:     inc(x);
37:     // display updated values
38:     cout << "After using the overloaded inc function\n";

```

```

39:  cout << "c = " << c << "\n"
40:      << "i = " << i << "\n"
41:      << "x = " << x << "\n";
42:
43:  return 0;
44: }

```

下面是程序列表 2.8 中的程序输出部分:

输出:

```

c=a
i=10
x=10.2
After usilg the overloaded inc function
c=B
i=11
x=11.2

```

分析:

程序列表 2.8 中的程序说明了三种形式的重载 void 型函数 inc。第一种形式的函数 inc 有一个 int 型的引用参数 i。该函数使参数 i 以 1 为步长增值。因为参数 i 是函数变量的一个引用,函数 inc(int&) 活动要影响到这个函数范围外的那个变量。第二种形式的函数 inc 有一个 double 型的引用参数 x。该函数使参数 x 增值 1。因为参数 x 是函数变量的一个引用,所以函数 inc(double&) 的活动要影响到这个函数外的那个变量。第三种形式的函数 inc 有一个 char 型的引用参数 c。该函数使参数 c 增值 1。这个引用参数要影响到该函数外的变量。

函数 main 把变量 c, i 和 x 分别说明 char, int 和 double 型。该函数还使用数值 'A'、10 和 10.2 分别初始化了变量 c, i 和 x。第 30 行到第 32 行的语句显示了变量 c, i 和 x 的初始值。函数 main 在第 34 行到第 36 行调用了重载函数 inc。第 34 行的函数 inc 调用就是以调用了重载函数 inc(char&) 结束,因为使用的变量是 char 型的变量。第 35 行是调用函数 inc(int&),因为变量使用的是 int 型变量。第 36 行是调用函数 inc(double&),因为使用的变量是 dowble 型变量。第 39 行到第 41 行的输出语句显示了变量 c, i 和 x 更新后的值。

2.13 小结

今天的课介绍了 C++ 程序的基本要素。这些基本要素包括数据类型、变量、常量和函数。你学习了这些基础知识:

- ☐ Turbo C++ 中的预定义数据类型包括 int, char, float, double, 和 void 数据类型。C++ 由于支持数据类型的修饰符,从而增加了数据类型的柔韧性。这些修饰符改变了数值的精度和范围。它们是 signed, short 和 long。
- ☐ Turbo C++ 标识符可以达到 32 个字符,并且必须是以字母或下划线开头。标识符后续的字符可以是字母、数字或下划线。C++ 标识符是区分大小写的。
- ☐ #include 伪指令是编译器的一个特殊指令。该指令通知编译器,把指定文件的内

容包含进去,就好像你把它输入了当前的源文件。

- ☐ 说明变量要求你指明变量的数据类型和名称。当你说明变量时,C++允许你对它进行初始化。你可以在单个说明语句中说明多个变量。
- ☐ 说明常量包含了使用 `#define` 伪指令来说明宏常量,或者使用 `const` 关键字来说明形式常量。形式常量要求你指定常量的类型(缺省类型是 `int` 型)、常量的名称和有关的值。

- ☐ 定义函数的一般形式是:

```
returnType functionName(parameterList)
```

```
{
```

```
    <declarations of data items>
```

```
    <function body>
```

```
    return returnValue;
```

如果一个函数在定义它的原型以前就要被一个用户函数使用,那么你就需要对这个函数进行原型说明。函数原型说明的一般形式是:

```
returnType functionName(parameterList);
```

你可以忽略参数表中的参数名。

- ☐ 函数的局部变量支持非常独立的函数运用。局部变量的说明是与全局变量的说明类似的。
- ☐ 函数静态变量的说明需要在变量的数据类型前加上关键字 `static`。静态变量在函数调用之间保持它们的值。在大多数情况下,你需初始化静态变量。而且,这些初始值在程序第一次调用函数时就分配给了这些静态变量。
- ☐ 内联函数允许你像基于宏的伪函数一样,扩充它们的语句。然而,内联函数却不像这些伪函数一样,它们要进行类型检查。
- ☐ 你可以用 `return` 语句退出函数。同时 `void` 型函数不需要在 `return` 关键字后面包含表达式。
- ☐ 缺省变量允许你为一个函数的参数分配缺省值。当你忽略一个含有缺省变量的参数的变量时,那个变量会自动地运用。
- ☐ 函数重载使你能够说明具有同名但不同参数表(也称作函数署名)的多个函数。函数的返回类型不属于函数署名部分,因为 C++ 允许你忽略结果的类型。

2.14 问与答

问:命名标识符有特定的风格吗?

答:在近几年,有几种非常流行的风格。我使用的第一种是以小写字符开头的标识符。

如果标识符含有多个字,比如 `numberOfElements`,那么要使每个后续字的第一个字符为大写字母。

问:C++函数可以说明嵌套的函数吗?

答:不可以。嵌套的函数实际上在执行时增加许多顶层的执行时间。

问:都什么时候可以使用静态全局变量?

答:任何时候都不可以。全局变量不需要说明为静态的,因为它们在整个程序的生命

期都存在。

2.15 专题讨论

专题讨论提供了测验问答,帮助你加深对所学的资料的理解;同时,还提供了练习,帮助你提高使用你所学的知识经验。一定要在继续明天的课之前尽力去理解测验和练习的答案。所有答案都在附录 B 中提供了。

2.15.1 测验

1. 下面的变量哪个是有效的,哪个是无效的(为什么)?

```
numFiles
n0Distance _02 _Line
0Weight
Bin Number
static
static
```

2. 下面程序的输出结果是什么? 你对函数 swap 的理解是怎样的?

```
#include<iostream.h>
void swap(int i, int j)
{
    int temp=i;
    i=j;
    j=temp;
}
main()
{
    int a=10,b=3;
    swap(a,b);
    cout<<"a="<<a<<" and b="<<b;
    return 0;
}
```

3. 下面的程序的输出结果是什么? 你对函数 swap 是怎样理解的?

```
#include<iostream.h>
void swap(int& i, int& j)
{
    int temp=i;
    i=j;
    j=temp;
}
main()
{
    int a=10,b=3;
```

```

    swap(a,b);
    cout<<"a="<<a<<" and b="<<b;
    return 0;
}

```

4. 下面的重载函数有什么问题?

```

void inc(int& i)
{
    i=i+1;
}
void inc(int& i, int diff=1)
{
    i=i+diff;
}

```

5. 下面的函数错在哪里?

```

double volume(souble length, double width =1,double height)
{
    return length * width * height
}

```

6. 下面的函数错在哪里?

```

void inc(int& i, int diff=1)
{
    i=I + diff;
}

```

7. 下面的程序有什么错误? 你怎样改正它?

```

#include<iostream.h>
main()
{
    double x=5.2;
    cout<<x<<" 2="<<SQR(X);
    return 0;
}

double sqr(double x)
{return x * x;}

```

2.15.2 练习

通过在程序 OVERLOAD.CPP 中,给重载函数 inc 增加第二个带有缺省变量的参数来创建程序 OVERLOD2.CPP。新参数应该用缺省变量 1 来表示增量值。

第三章 运算符和表达式（第三天）

数据操作涉及由操作数和运算符组成的表达式。C++支持多种运算符和表达式。

新术语:运算符是特殊的符号,它们能够利用操作的值来产生一个新值。

每类运算符都是用一种特定的方式操作数据。今天,我们学习有关下面的主题:

- ☐ 算术运算符和表达式
- ☐ 增量运算符
- ☐ 算术赋值运算符
- ☐ 类型转换和数据变换
- ☐ 关系运算符和条件表达式
- ☐ 位操作运算符
- ☐ 逗号运算符

3.1 算术运算符

表 3.1 显示了 C++ 的算术运算符。编译器可以根据操作数,执行浮点或整数除法。如果两个操作数都是整数表达式,那么编译器就产生一个整数除法的代码。如果任一个或者两个操作数是浮点表达式,那么编译器就产生浮点除法的代码。

表 3.1 C++ 算术运算符

C++ 运算符	目的	数据类型	举例
+	一元加	数字	$X = +Y + 3$
-	一元减	数字	$X = -Y$
+	加	数字	$Z = Y + X$
-	减	数字	$Z = Y - X$
*	乘	数字	$Z = Y * X$
/	除	数字	$Z = Y / X$
%	求余	整数	$Z = Y \% X$

让我们来看看一个例子,这个例子使用了带有整数和浮点数的数学运算符。程序列表 3.1 显示了程序 OPER1.CPP 的源代码(我建议你使用工程文件 OPER1.PRJ)。该程序完成下面的任务:

- ☐ 提示你输入两个整数。
- ☐ 对两个整数应用+、-、*、/和%运算符,结果存放在单独的变量中。

- ☐ 显示整数操作的结果。
- ☐ 提示你输入两个浮点数(每次提示输入一个数)。
- ☐ 对两个数应用+、-、* 和/运算符,并把结果存放在单独的变量中。
- ☐ 显示浮点操作的结果。

程序列表 3.1 程序 OPER1.CPP 的源文件

```

1: // simple C++ program to illustrate simple math operations
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:
8:     int int1, int2;
9:     long long1, long2, long3, long4, long5;
10:    float x, y, real1, real2, real3, real4;
11:
12:    cout << "\nType first integer : ";
13:    cin >> int1;
14:    cout << "Type second integer : ";
15:    cin >> int2;
16:    cout << "\n";
17:    long1 = int1 + int2;
18:    long2 = int1 - int2;
19:    long3 = int1 * int2;
20:    long4 = int1 / int2;
21:    long5 = int1 % int2;
22:    cout << int1 << " + " << int2 << " = " << long1 << "\n";
23:    cout << int1 << " - " << int2 << " = " << long2 << "\n";
24:    cout << int1 << " * " << int2 << " = " << long3 << "\n";
25:    cout << int1 << " / " << int2 << " = " << long4 << "\n";
26:    cout << int1 << " mod " << int2 << " = " << long5 << "\n";
27:    cout << "\n\n";
28:    cout << "Type first real number : ";
29:    cin >> x;
30:    cout << "Type second real number : ";
31:    cin >> y;
32:    cout << "\n";
33:    real1 = x + y;
34:    real2 = x - y;
35:    real3 = x * y;
36:    real4 = x / y;
37:    cout << x << " + " << y << " = " << real1 << "\n";
38:    cout << x << " - " << y << " = " << real2 << "\n";
39:    cout << x << " * " << y << " = " << real3 << "\n";
40:    cout << x << " / " << y << " = " << real4 << "\n";
41:    cout << "\n\n";
42:    return 0;
43: }

```

下面是程序列表 3.1 中的程序输出部分:

输出:

10+5=15

10-5=5

10*5=20

10/5=2

10 mod 5=0

Type first real number:1.25

Type second real number:2.58

$1.25 + 2.58 = 3.83$

$1.25 - 2.58 = -1.33$

$1.25 * 2.58 = 3.225$

$1.25 / 2.58 = 0.484496$

分析:

程序列表 3.1 中的程序在函数 main 中说明了一组 int 型、long 型和 float 型的变量。这些变量中的一些存放你的输入,另外一些存放数学运算的结果。第 12 行的输出语句提示你输入第一个整数。第 13 行的输入语句获取你的输入,并把它存放在变量 int1 中。第 14 行和第 15 行执行同样的操作,提示你输入第二个整数,并把它存放在变量 int2 中。该程序在第 17 行到第 21 行执行整数数学运算,并把这些运算的结果存放在变量 long1 到 long5 中。把这些说明变量为 long 型,为了防止产生数字溢出。第 22 行到第 26 行的输出语句显示了整数操作数、使用的运算符和结果。

第 28 行的输出语句揭示你输入第一个浮点数。第 29 行的输入语句获取你的输入,并把它存放在变量 x 中,第 30 行和第 31 行执行同样的操作,揭示你输入第二个浮点数,并把它存放在变量 y 中。

新术语:浮点数(floating-point number)也称作实数。

这个程序在第 33 行到 36 行执行浮点数学运算,并把运算结果存放在变量 real4 中。第 37 行到 40 行的输出语句显示了操作数、使用的运算符和结果。

3.2 算术表达式

最简单的表达式是那些含文字的表达式,比如:

-12

34.45

'A'

"Hello"

新术语:在一般的术语中,算术表达式(arithmetic expression)属于包含数值的程序语句。

文字常量 -12 和 35.45 是最简单的算术表达式。而比较简单的算术表达式包含了简单的变量或常量,比如:

DAYS_IN_WEEK // a constant

i

x

稍微简单的算术表达式则包含了一个简单的运算符,它带有数字、常量和变量作为操作数。下面是几个例子:

355/113

4 * i

45. 67 + x

较高级的算术表达式包含了多种运算符、圆括号,以及函数,比如:

(355/113) * square(radius)

PIE * square(radius)

((2 * x - 3) * x + 2) * x - 5

(1 + x) / (3 - x)

在介绍了其他类型的运算符以后,将在今天课程的最后讨论执行运算符的顺序。

3.3 增量运算符

C++支持特殊的增量和减量运算符。

新术语:增量(++)和减量(--)运算符允许你对存放在变量中的值分别进行增1和减1。

语法

增量运算符

增量运算符的一般语法是

variable++ // post-increment

++variable // pre-increment

例子:

lineNumber++;

++index;

语法:

减量运算符

减量运算符的一般语法是

variable-- // post-decrement

--variable // pre-decrement

例子:

lineNumber--;

--index;

这种一般的语法说明。应用++和--运算符有两种方式。把这些运算符放在它们的操作数的左边将使得操作数在表达式中进行赋值之前,就改变该操作数的值。同样,把这些运算符放在它们的操作数的右边,将使得操作数在表达式中进行赋值以后,才改变该操作数的值。如果++或者--运算符仅仅是作为语句中的运算符,那么在使用它们的预先或者置后形式之间没有什么实际的差异。

下面是几个简单的例子:

int n, t = 5;

t++; // t is now 6, same effect as ++t

```
--t; // t is now 5, same effect as t--
t = 5;
n = 4 * t++; // t is now 6 and n is 24
t = 5;
```

第一个语句使用了置后增量++运算符。来对变量t的值进行增值。如果你写++t代替它,那么你在这条语句执行完后,将得到同样的值。第二个语句使用了预先减量--运算符。如果你写t--代替它,那么你也将得到同样的值。接下去的两个语句是首先给t赋值5,然后在一个简单的数学表达式中使用置后增量++运算符。该语句把t的当前值(也就是5)乘以4,并把结果20赋给变量n,然后变量t中的值增加到6。最后的两个语句用来显示一个不同的输出。其中第一个语句使变量t中的值增加(变量t中的值变成了6),然后执行乘法运算,最后把结果24赋给变量n。

让我们来看看一个简单的程序,该程序说明了增量运算符的特性。程序列表3.2显示了程序OPER2.CPP的源代码。这个程序不要求你输入。它简单地显示了两个使用增量运算符获取值的整数。

程序列表 3.2 程序 OPER2.CPP 的源文件

```
1: /*
2:  C++ program to illustrate the feature of the increment operator.
3:  The ++ or -- may be included in an expression. The value
4:  of the associated variable is altered after the expression
5:  is evaluated if the var++ (or var--) is used, or before
6:  when ++var (or --var) is used.
7: */
8:
9: #include <iostream.h>
10:
11: main()
12: {
13:     int i, k = 5;
14:
15:     // use post-incrementing
16:     i = 10 * (k++); // k contributes 5 to the expression
17:     cout << "i = " << i << "\n\n"; // displays 50 (= 10 * 5)
18:
19:     k--; // restores the value of k to 5
20:
21:     // use pre-incrementing
22:     i = 10 * (++k); // k contributes 6 to the expression
23:     cout << "i = " << i << "\n\n"; // displays 60 (= 10 * 6)
24:     return 0;
25: }
```

下面是程序列表 3.2 中的程序输出部分:

输出:

```
i=50
i=60
```

分析:

在程序列表 3.2 中,程序包含有函数 main,该函数说明了两个 int 型的变量 i 和 k。同时,该函数通过给变量 k 赋值 5,对它进行了初始化。第 16 行包含了对变量 k 应用置后增量运算符的语句,因此,该语句把 k 的初始值 5 乘以 10,然后把结果 50 赋给变

量 i。在赋值完以后,程序把变量 k 的值增 1。第 17 行的输出语句显示变量 i 中的值。第 19 行语句把变量 k 的值又减小到 5。第 22 行语句对变量 k 应用预先增量运算符。所以,程序首先把变量 k 的值增 1(从 5 增到 6),然后再把 k 的新值乘以 10。最后程序把乘法运算的结果 60 赋给变量 i。第 23 行的输出语句显示变量 i 的当前值。

3.4 赋值运算符

作为一个程序员,你常常会碰到像这样的语句:

```
IndexOfFirstElement = IndexOfFirstElement + 4;
```

```
GraphicsScaleRatio = GraphicsScaleRatio * 3;
```

```
CurrentRateOfReturn = CurrentRateOfReturn/4;
```

```
DosfileListSize = DosfileListSize-10;
```

接收表达式的结果的变量还是一个操作数(当然,加法和乘法是互相关联的运算。因此,赋值的变量也可以作为操作数进行这些运算)。注意,我选择了相对较长的名称是为了提醒你,如果你没有使变量名变短一点,那么你就需要缩短表达式。

新术语: C++ 提供赋值运算符,使得简单的数学运算符可以合并到一起。

你可以编写下面这样的语句:

```
IndexofFirstElement += 4;
```

```
GraphicsScaleRatio *= 3;
```

```
CurrentRateofReturn /= 4;
```

```
DOSfileListSize -= 10;
```

注意,变量名只出现一次。另外,还要注意,语句使用这些运算符: +=, *=, /= 和 -=。表 3.2 显示了算术赋值运算符。C++ 还支持其他类型的赋值运算符。

表 3.2 算术赋值运算符

赋值运算符	长型	举例
$x += y$	$x = x + y$	$x += 12;$
$x -= y$	$x = x - y$	$x -= 34 + y;$
$x *= y$	$x = x * y$	$scale *= 10;$
$x /= y$	$x = x / y$	$z /= 34 * y;$
$x \% = y$	$x = x \% y$	$z \% = 2;$

下面让看看把赋值运算符应用到整数和浮点数中的程序。程序 OPER3.CPP 的源代码。这个程序执行下面的任务：

- ☐ 提示你输入两个整数(每次提示输入一个)。
- ☐ 对这个两个整数应用赋值和增值运算符。
- ☐ 显示这个两个整数的新值。
- ☐ 提示你输入两个浮点数(每次提示输入一个)。
- ☐ 显示浮点数的新值。

程序列表 3.3, 程序 OPER3.CPP 的源代码。

```
6: {
7:   int i, j;
8:   double x, y;
9:
10:  cout << "Type first integer : ";
11:  cin >> i;
12:  cout << "Type second integer : ";
13:  cin >> j;
14:  i += j;
15:  j -= 6;
16:  i *= 4;
17:  j /= 3;
18:  i++;
19:  j--;
20:  cout << "i = " << i << " n";
21:  cout << "j = " << j << " n";
22:
23:  cout << "Type first real number : ";
24:  cin >> x;
25:  cout << "Type second real number : ";
26:  cin >> y;
27:  // abbreviated assignments also work with doubles in C++
28:  x += y;
29:  y -= 4.0;
30:  x *= 4.0;
31:  y /= 3.0;
32:  x++;
33:  y--;
34:  cout << "x = " << x << " \n";
35:  cout << "y = " << y << " \n";
36:  return 0;
37: }
```

下面是程序列表 3.3 中的程序输出部分：

输出：

Type first integer : 55

Type second integer : 66

i = 485

j = 19

Type first real number : 2.5

Type second real number : 4.58

`x = 29.32`

`y = 0.8.6667`

分析:

程序列表 3.3 中的程序包含函数 `main`, 它在第 7 行和第 8 行分别说明了两个整型变量 (`i` 和 `j`) 和两个双精度型变量 (`x` 和 `y`)。第 10 行的输出语句提示你输入第一个整数。第 11 行的输入语句接收你的输入, 并把它存放在变量 `i` 中。第 12 行和第 13 行与第 10 行和第 11 行相似——提示你输入第二整数并存入在变量 `j` 中。

这个程序使用第 14—19 行的语句来操作变量 `i` 和 `j` 中的值。在第 14 行中, 该程序使用 `+=` 运算符, 把变量 `i` 的值加上变量 `j` 的值, 再赋值给变量 `i`。第 15 行使用 `-=` 运算符, 把变量 `j` 的值减去 6, 再赋值给变量 `j`。第 16 行运用 `*=` 运算符, 把变量 `i` 的值乘以 4, 再赋值给变量 `i`。第 17 行使用 `/=` 运算符, 把变量 `j` 的值除以 3, 再赋值给变量 `j`。第 18、19 行分别对变量 `i` 和 `j` 使用增量和减量运算符。第 20、21 行的输出语句分别显示变量 `i` 和 `j` 的值。

第 23 行的输出语句提示你输入第一个浮点数。第 24 行的输入语句接收你的输入, 并把它放在变量 `x` 中。第 25、26 行与第 23、24 行相似; 它们提示你输入第二个浮点数, 并把它存放在变量 `y` 中。

该程序使用第 28—33 行的语句, 对变量 `x` 和 `y` 进行操作。在第 28 行中, 程序使用 `+=` 运算符, 把变量 `x` 的值加上变量 `y` 的值, 再把结果赋给变量 `x`。第 29 行使用了 `-=` 运算符, 把变量 `y` 的值减去 4, 再把结果赋给变量 `y`。第 30 行使用 `*=` 运算符, 把变量 `x` 的值乘以 4, 再赋值给变量 `x`。第 31 行使用 `/=` 运算符, 把变量 `y` 的值除以 3, 再赋值给变量 `y`。第 32、33 行分别对变量 `x` 和 `y` 使用了增量和减量运算符。第 34、35 行分别显示变量 `x` 和 `y` 的值。

3.5 sizeof 运算符

你的程序常常需要知道数据类型或变量的字节大小 C++ 为你提供了 `sizeof` 运算符, 它带有数据类型或变量名参数 (数量、数组、记录, 等等)。

语法

size of 运算符

`size of` 运算符的一般语法是

`size of ({variable-name | data-type})`

例子:

```
int sizeDifference = sizeof (double) - sizeof (float);
```

```
int intSize = sizeof (int);
```

DO	DON'T
DO 使用带有变量名而不是它的数据类型的 size of, 这种方法是最安全的。因为如果你改变了变量的数据类型, size of 运算符还会返回正确的答案。相反, 如果你使用带有变量的数据类型的大小运算符, 并且后来又改变了那个变量的数据类型, 那么, 如果你没有更新 size of 运算符的参数, 就会产生错误。	DON'T 不要使用数字来表示变量的字节大小, 这种方法经常引起错误。

让我们来看看一个例子, 这个例子使用了带有变量和数据类型参数的 size of 运算符。程序列表 3.4 包含了程序 SIZEOF1.CPP 的源代码。这个程序显示了两个相似的表格, 指出了 short int、int、long、int、char 和 float 数据类型的大小。该程序显示的第一个表格是对这些类型的变量运用 size of 运算符; 第二个表格是直接对这些数据类型运用 size of 运算符。

程序列表 3.4 程序 SIZEOF1.CPP 的源代码

```

1: /*
2:  simple program that returns the data sizes using the sizeof()
3:  operator with variables and data types.
4:  */
5:
6: #include <iostream.h>
7:
8: main()
9: {
10:
11:     short int aShort;
12:     int anInt;
13:     long aLong;
14:     char aChar;
15:     float aReal;
16:
17:     cout << "Table 1. Data sizes using sizeof(variable)\n\n";
18:     cout << "    Data type          Memory used\n";
19:     cout << "                (bytes)\n";
20:     cout << "-----";
21:     cout << "\n    short int          " << sizeof(aShort);
22:     cout << "\n    integer           " << sizeof(anInt);
23:     cout << "\n    long integer      " << sizeof(aLong);
24:     cout << "\n    character        " << sizeof(aChar);
25:     cout << "\n    float            " << sizeof(aReal);
26:     cout << "\n\n\n";
27:
28:     cout << "Table 2. Data sizes using sizeof(dataType)\n\n";
29:     cout << "    Data type          Memory used\n";
30:     cout << "                (bytes)\n";
31:     cout << "-----";
32:     cout << "\n    short int          " << sizeof(short int);
33:     cout << "\n    integer           " << sizeof(int);
34:     cout << "\n    long integer      " << sizeof(long);
35:     cout << "\n    character        " << sizeof(char);
36:     cout << "\n    float            " << sizeof(float);
37:     cout << "\n\n\n";

```

```

38:
39:     return 0;
40: }

```

下面是程序列表 3.4 中的程序输出部分:

输出:

Table 1. Data sizes using sizeof(variable)

Data type	Memory used (bytes)
.....
short int	2
integer	2
long integer	4
character	1
float	4

Table 2. Data sizes using sizeof(datatype)

Data type	Memory used (bytes)
.....
short int	2
integer	2
long integer	4
character	1
float	4

分析:

在程序列表 3.4 中,程序在函数 main 中说明了 5 个变量。每个变量都有不同的数据类型,并且它的名称根据它的数据类型相区别。例如,变量 anInt 是一个 int 型变量,变量 aLong 是一个 long 型变量等等。

第 17—25 行的语句显示数据字节大小的表格。第 21—25 行的输出语句使用了带有变量的 sizeof 运算符。

第 28—36 行的语句也显示数据字节大小的表格。第 32—36 行的输出语句使用了带有数据类型标识符的 sizeof 运算符。

3.6 强制类型转换

一个数值从一种数据类型自动转换到别一种相兼容的数据类型是编译器的责任之一。这种数据转换简化了表达式,并且减轻初学者和经验丰富的程序员困难。由于这种隐含的数据转换,你无需检查每个在你的程序中混合了相容数据类型的表达式。例如,编译器处理大多数混合了各种类型整数或者混合了整数和浮点型数的表达式。但是,如果你试图做一些不合语法的事情,那么你就会得到一个编译错误。

新术语:强制类型转换(TyPecasting)是一种语言特点,它使你能够清楚地指定如何把一个数值从它的初始数据类型转换成一种相容的数据类型

语法

强制类型转换

C++支持下面形式的

`type_cast(expression)`

`(type_Cast)expression。`

例子:

```
int i = 2;
```

```
float a, b;
```

```
a = float(i)
```

```
b = (float) i;
```

让我们来看看一个例子,该例子说明了隐含的数据转换和强制类型转换。程序列表 3.5 显示了程序 TIPECAST1.CPP 的源代码。这个程序说明的变量有字符型、整型和浮点型。然后,该程序执行两套相似的数学运算。第一套依靠由编译器执行的自动类型转换;第二套运算使用强制类型转换,清楚地指示编译器如何转换数据类型。这个程序不要求任何输入——它本身提供数据——并且能够显示两套运算的输出值。最后,该程序说明了编译器能够产生的两套运算的相同输出结果。

程序列表 3.5 程序 TPPECAST1.CPP 的源代码

```
1: // simple C++ program that demonstrates typecasting
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     short shortInt1, shortInt2;
8:     unsigned short aByte;
9:     int anInt;
10:    long aLong;
11:    char aChar;
12:    float aReal;
13:
14:    // assign values
15:    shortInt1 = 10;
16:    shortInt2 = 6;
17:    // perform operations without typecasting
18:    aByte = shortInt1 + shortInt2;
19:    anInt = shortInt1 - shortInt2;
20:    aLong = shortInt1 * shortInt2;
21:    aChar = aLong + 5; // conversion is automatic to character
22:    aReal = shortInt1 * shortInt2 + 0.5;
23:
24:    cout << "shortInt1 = " << shortInt1 << '\n'
25:         << "shortInt2 = " << shortInt2 << '\n'
26:         << "aByte = " << aByte << '\n'
27:         << "anInt = " << anInt << '\n'
28:         << "aLong = " << aLong << '\n'
```

```

29:         << "aChar is " << aChar << '\n'
30:         << "aReal = " << aReal << "\n\n";
31:
32:     // perform operations with typecasting
33:     aByte = (unsigned short) (shortInt1 + shortInt2);
34:     anInt = (int) (shortInt1 - shortInt2);
35:     aLong = (long) (shortInt1 * shortInt2);
36:     aChar = (unsigned char) (aLong + 5);
37:     aReal = (float) (shortInt1 * shortInt2 + 0.5);
38:
39:     cout << "shortInt1 = " << shortInt1 << '\n'
40:         << "shortInt2 = " << shortInt2 << '\n'
41:         << "aByte = " << aByte << '\n'
42:         << "anInt = " << anInt << '\n'
43:         << "aLong = " << aLong << '\n'
44:         << "aChar is " << aChar << '\n'
45:         << "aReal = " << aReal << "\n\n";
46:     return 0;
47: }

```

下面是程序列表 3.5 中的程序输出部分：

输出：

```

shortInt1 = 10
shortInt2 = 6
aByte = 16
anInt = 4
aLong = 60
aChar is A
aReal = 60.5

```

```

shortInt1 = 10
shortInt2 = 6
aByte = 16
anInt = 4
aLong = 60
aChar is A
aReal = 60.5

```

分析：

在程序列表 3.5 中，程序在函数 main 中说明了下列变量：

- ☐ short 型变量 shortInt1 和 shortInt2.
- ☐ unsigned short 型变量 aByte
- ☐ int 型变量 anInt
- ☐ long 型变量 aLong
- ☐ char 型变量 aChar
- ☐ float 型变量 aReal

第 15、16 行分别给变量 shortInt1 和 shortInt2 赋值了 10 和 6。第 18—22 行进行了不同的数学运算，并把结果赋给变量 aByte、anInt、aLong、aChar 和 aReal。

注意: C 和 C++ 把字符类型作为一种特殊的整数处理。每个字符型的文字(比如“A”), 常量或变量都有一个整数值, 该值就等于它的 ASCII 值。这种语言特点使你能够把一个整数存放在一个字符型的变量中, 以及把一个字符型数据项作为一个整数处理。第 21 行语句就是把整数 5 加上变量 aLong 的值, 并把结果(一个整数)赋给变量 aChar。这个赋值的整数值 65 就表示字母 A 的 ASCII 码。

第 24—30 行的输出语句显示存放在变量中的注意, 变量 aChar 的输出是字母 A。如果我把变量 aChar 的输出项写成这样: <<(int)aChar, 那么, 我就会得到存在 aChar 中的字符的 ASCII 码 65。第 32—37 行的语句执行与第 18—22 行的语句相似的运算。主要的不同是, 第 32—37 行的语句使用了强制类型转换, 明确地指示编译器如何转换结果类型。第 39—45 行的输出语句显示变量的内容。

3.7 关系运算符和逻辑运算符

表 3.3 显示了 C++ 的关系运算符和逻辑运算符。注意, C++ 不拼写出运算符 AND, OR 和 NOT, 而是使用简单的二元符号。还要注意, C++ 不支持关系 XOR 运算符。你可以使用下面的 #define 宏指令, 把 AND, OR 和 NOT 标识符定义为宏:

```
#define AND &&
#define OR  ||
#define NOT  !
```

新术语: 关系运算符(少于, 大于和等于)和逻辑运算符(AND, OR 和 NOT)在任何编程语言中都是判断结构的基本构造块。

表 3.3 C++ 的关系运算符和逻辑运算符

C++运算符	意 义	举 例
&&	逻辑 AND	if(i>1 && i<10)
	逻辑 OR	if(c==0 c==9)
!	逻辑 NOT	if(! (c>1 && c<9))
<	小于	if(i<0)
<=	小或等于	if(i<=0)
>	大于	if(j>10)
>=	大于或等于	if(x>=8.2)
==	等于	if(c=='\0')

续表

C++运算符	意义	举 例
!=	不等于	if(c! = '\n')
?:	条件赋值	k=(i<1)? 1;i;

虽然这些宏在 C++ 中都是允许的,但是,你可能会从读你的代码的经验丰富的程序员那里得到负的反应。谁说编程总是有目标的?

警告:不要使用 = 运算符作为相等关系运算符。这个普遍的错误来源于 C++ 程序中的逻辑错误。你可能在其他语言中习惯于使用 = 运算符来测试两个数据项相等。但是在 C++ 中,你必须使用 == 运算符。如果你在 C++ 使用 == 运算符,会发生什么事情呢?你会得到一个编译错误吗?答案是你可能会得到一个编译警告。除了那样以外,你的 C++ 程序也会运行。当程序获得的那个表达式是为了测试相等时,而它实际上却是试图把 = 号右边的操作数赋值给 = 号左边的操作数。当然,这样的最可能导致不可思议的运行结果,或者甚至系统终止执行。

注意:C++ 不支持预定义的布尔标识符。而是,该语言把 0 看作为假,非零值看作为真。要增加你的程序的清晰度,我建议你说明全局常量 TRUE 和 FALSE,并别给它们赋值 1 和 0。

注意:在表格 3.3 中,最后的那个运算符是?:。这个特殊的运算符支持可谓的条件表达式。

新术语:条件表达式(Conditiond expression)是简单的 if--else 语句的二元替换式。(关于 if 语句的更详细资料见第 5 章(第 5 天)。)

例如,下面是一个 if--else 语句:

```
if (condition)
variable = expression1;
else
variable = expression2;
```

同等的条件表达式如下:

```
variables=(condition)? expression1;expression2;
```

条件表达式用来测试条件。如果条件是真,那么,它就把 expression1 赋给目标变量。否则,它就把 exprssion2 赋给目标变量。

3.8 布尔表达式

你常常需要使用一组关系和逻辑运算符来组成一个重要的条件表达式。下面就是些这样的例子：

```
x < 0 || x > 11
(i != 0 || i > 100) && (j != i || j > 0)
x != 0 && x != 10 && x != 100
```

新术语：布尔(也称为逻辑)表达式是包含有逻辑运算符或关系运算符的表达式。

DO	DON'T
DO 要进行复检,避免布尔表达式总是真的或总是假的。例如,表达式(x<0 && x>10)总是假的,因为没有这样的 x 值可以是负的,同时又是大于 10。	
DON'T 不要使用 = 运算符来测试相等。	

让我们来看看一个例子,它使用了关系和逻辑运算符与表达式。程序列表 3.6 显示了程序 **RELOP1.CPP** 的源代码。这个程序提示你首先输入三个整数,然后接下去执行一组测试。该程序显示了关系和逻辑运算,它们的操作数以及它们的结果。

程序列表 3.6 程序 **RELOP1.CPP** 的源代码。

```
1: /*
2:  simple C++ program that uses logical expressions
3:  this program uses the conditional expression to display
4:  TRUE or FALSE messages, since C++ does not support the
5:  BOOLEAN data type.
6: */
7:
8: #include <iostream.h>
9:
10: const MIN_NUM = 30;
11: const MAX_NUM = 199;
12: const int TRUE = 1;
13: const int FALSE = 0;
14:
15: main()
16: {
17:     int i, j, k;
18:     int flag1, flag2, in_range,
19:         same_int, xor_flag;
20:
21:     cout << "Type first integer : "; cin >> i;
22:     cout << "Type second integer : "; cin >> j;
23:     cout << "Type third integer : "; cin >> k;
24:
25:     // test for range [MIN_NUM..MAX_NUM]
26:     flag1 = i >= MIN_NUM;
27:     flag2 = i <= MAX_NUM;
28:     in_range = flag1 && flag2;
```

```

29:     cout << "\n" << i << " is in the range "
30:         << MIN_NUM << " to " << MAX_NUM << " : "
31:         << ((in_range) ? "TRUE" : "FALSE");
32:
33:     // test if two or more entered numbers are equal
34:     same_int = i == j || i == k || j == k;
35:     cout << "\nat least two integers you typed are equal : "
36:         << ((same_int) ? "TRUE" : "FALSE");
37:
38:     // miscellaneous tests
39:     cout << "\n" << i << " != " << j << " : "
40:         << ((i != j) ? "TRUE" : "FALSE");
41:     cout << "\nNOT (" << i << " < " << j << ") : "
42:         << ((!(i < j)) ? "TRUE" : "FALSE");
43:     cout << "\n" << i << " <= " << j << " : "
44:         << ((i <= j) ? "TRUE" : "FALSE");
45:     cout << "\n" << k << " > " << j << " : "
46:         << ((k > j) ? "TRUE" : "FALSE");
47:     cout << "\n(" << k << " = " << i << ") AND ("
48:         << j << " != " << k << ") : "
49:         << ((k == i && j != k) ? "TRUE" : "FALSE");
50:
51:     // NOTE: C++ does NOT support the logical XOR operator for
52:     // boolean expressions.
53:     // add numeric results of logical tests. Value is in 0..2
54:     xor_flag = (k <= i) + (j >= k);
55:     // if xor_flag is either 0 or 2 (i.e. not = 1), it is
56:     // FALSE therefore interpret 0 or 2 as false.
57:     xor_flag = (xor_flag == 1) ? TRUE : FALSE;
58:     cout << "\n(" << k << " <= " << i << ") XOR ("
59:         << j << " >= " << k << ") : "
60:         << ((xor_flag) ? "TRUE" : "FALSE");
61:     cout << "\n(" << k << " > " << i << ") AND ("
62:         << j << " <= " << k << ") : "
63:         << ((k > i && j <= k) ? "TRUE" : "FALSE");
64:     cout << "\n\n";
65:     return 0;
66: }

```

下面是程序列表 3.6 中的程序输出部分：

输出：

Type first integer : 55

Type second integer : 64

Type third integer : 87

55 is in the range 30 to 199 : TRUE

at least two integers you typed are equal : FALSE

55 != 64 : TRUE

87 > 64 : TRUE

(87 == 55) AND (64 != 87) : FALSE

(87 < 55) XOR (64 > 87) : FALSE

(87 > 55) AND (64 <= 87) : TRUE

分析：

在程序列表 3.6 中，程序说明了四个全局常量。常量 MIN_NUM 和 MAX_NUM 定义了逻辑测试中使用的数据范围。常量 TRUE 和 FALSE 表示布尔值。函数 main

说明了许多整型变量,它们用于输入和各种测试中。第 21—23 行的语句提示你输入三个整数,并把它们分别存放在变量 `i`、`j` 和 `k` 中。

第 26—31 行的语句用来测试变量 `i` 的值是否在 `MIN_NUM` 和 `MAX_NUM` 的范围内。第 26 行的语句测试变量 `i` 的值是大于还是等于常量 `MIN_NUM`。并且程序把布尔结果赋给变量 `flag1`。第 27 行的语句测试变量 `i` 的值是小于还是等于常量 `MAX_NUM`。并且程序把布尔结果赋给变量 `flag2`。第 28 行语句对变量 `flag1` 和 `flag2` 运用 `&&` 运算符,并把布尔结果赋给变量 `in_range`。第 29—31 行的输出语句表明测试的是什么,并根据变量 `in_range` 中的值显示 `TRUE` 或 `FALSE`。这个语句使用了条件运算符`?:`,如果 `in_range` 是一个非零值,则显示字符串 `TRUE`;如果 `in_range` 是零值,则显示字符串 `FALSE`。

第 34—36 行的语句判断你输入的三个整数中是否至少有两个相等。第 34 行的语句使用了一个布尔表达式,该表达式运用了 `==` 关系运算符和 `||` 逻辑运算符。这个语句把布尔结果赋给变量 `Same_int`。第 35、36 行的输出语句表明测试对象,并显示输出结果 `TUER/FALSE`。该输出语句使用条件运算符,根据变量 `same-int` 的值来显示字符串 `TUER/FALSE`。第 39—49 行的语句进行各种包含了输入值的混合的测试,并且显示测试对象和结果。你可以随意改变这些语句来进行不同的测试。

注意:第 54—60 行的语句执行一个异或(`XOR`)测试,并显示输出结果。该程序使用了一种简单的编程方法来实现异或(`XOR`)运算。第 54 行的语句增加了子表达式 `(k<=i)` 和 `(j<=k)` 的布尔值。如果两个子表达式为假,则结果为 0;如果两个子表达式为真,则结果为 1。因为只要任何一个子表达式为真,则异或(`XOR`)运算就是真,所以如果前面的值是 1,则第 57 行语句就把 `TUER` 赋给变量 `xor_flag`;否则,该语句就把 `FALSE` 赋给 `xor_flag`。第 61—63 行的语句执行另一个混合测试。

3.9 位操作运算符

C++是一种适合于系统开发的编程语言。系统开发要求位操作运算符。

新术语:位操作运算符(Bit-manipulation operators)可以保留、设置、询问和移动字节或字的位。

表 3.4 显示 3 位操作运算符。注意,C++使用符号 `&` 和 `|` 分别表示按位与(`AND`)和按位或(`OR`)。回想一下,`&&` 和 `||` 字符分别表示逻辑与或逻辑或运算符。除了位操作运算符以外,C++还支持位操作赋值运算符,见表 3.5 所示(使用位操作运算符属于高级编程技术,它包含了对位进行操作。作为一个初级 C++程序员,你最好不要在近期内使用这些运算符)。

表 3.4 C++位操作运算符

C++运算符	意 义	举 例
&	按位与	i & 128
	按位或	j 64
^	按位异或	j ^ 12
~	取反	~j
<<	左移	j >> 3
>>	右移	

表 3.5 C++位操作赋值运算符

C++运算符	原型	举例
x &= y	x = x & y	i &= 128
x = y	x = x y	j = 64
x ^= y	x = x ^ y	k ^= 15
x <<= y	x = x << y	j <<= 2
x >>= y	x = x >> y	k >>= 3

现在介绍一个 C++ 程序, 该程序进行了简单的位操作。程序列表 3.7 包含了程序 **BITS1.CPP** 的源代码。这个程序不要求任何输入, 因为它使用了内部的数据。该程序运用了 |、&、^、>> 和 << 位运算符, 并显示这些位操作的结果。

程序列表 3.7 程序 BITS1.CPP 的源代码

```

1: // C++ program to perform bit manipulations
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:
8:     int i, j, k;
9:
10:    // assign values to i and j
11:    i = 0xF0;
12:    j = 0x1A;
13:
14:    k = j & i;
15:    cout << j << " AND " << i << " = " << k << "\n";
16:
17:    k = j | i;
18:    cout << j << " OR " << i << " = " << k << "\n";
19:
20:    k = j ^ 0x10;
21:    cout << j << " XOR " << 0x10 << " = " << k << "\n";
22:
23:    k = i << 2;
24:    cout << i << " shifted left by 2 bits = " << k << "\n";

```



```

25:
26:     k = i >> 2;
27:     cout << i << " shifted right by 2 bits = " << k << "\n";
28:     return 0;
29: }

```

下面是程序列表 3.7 中的程序输出部分:

输出:

```

26 AND 240 = 16
26 OR 240 = 250
26 XOR 28 = 6
240 shifted left by 2 bits = 960
240 shifted right by 2 bits = 60

```

分析:

在程序列表 3.7 中,程序说明了三个整型变量 *i*、*j* 和 *k*。第 11、12 行的语句给变量 *i* 和 *j* 分别赋值了一个十六进制数。第 14 行的语句对变量 *i* 和 *j* 运用了按位与运算符,并把结果存放在变量 *k* 中。第 15 行的输出语句显示了操作数、位运算符和结果。第 17 行的语句对变量 *i* 和 *j* 运用了按位或运算符,并把结果存放在变量 *k* 中。第 18 行的输出语句显示了操作数、位运算符和结果。第 20 行的语句对变量 *j* 和十六进制整数 0x1C 运用了按位异或运算符。第 21 行的输出语句显示了操作数、位运算符和结果。第 23--27 行语句对变量运用了左移和右移运算符。这些运算符把变量 *i* 的位移动了 2 位,并把结果赋给了变量 *k*。其中,左移运算符的效果就相当于把变量 *i* 的值乘以 4。同样,右移运算符的效果就相当于把变量 *i* 的值除以 4。

3.10 逗号运算符

逗号运算符要求在计算出第二个表达式之前,程序首先要完全计算出第一个表达式。两个表达式都要位于同一个 C++ 语句中!那么,位于同一个 C++ 语句中的确切意思是什么呢?首先,为什么要使用这种相当特殊的运算符呢?因为这种具有特别作用的逗号运算符在 for 循环中起着—个特殊而又非常重要的作用。

新术语:循环(Loops)是功能很强的语言结构,它使计算机能够完成重复性的任务。逗号运算符使你能够创造多个表达式来初始化多个有关循环的变量。

语法

逗号运算符

逗号运算符的一般语法是

expression1, expression2

例子:

```
for(i=0,j=0;i>10;i++,j++)
```

在第六章(第六天)中,你将会学到更多的有关 for 循环的知识。现在,这个例子为你显示了如何运用逗号运算符。

3.11 运算过程和计算顺序

既然你现在熟悉了大多数的 C++ 运算符(还有一些处理指针和地址的运算符),你就需要知道两个相关的方面:第一,C++ 运算的过程;第二,计算的方向(或顺序)。表 3.6 显示了至今已经讲过的 C++ 运算符的运算过程,并且指出了计算的方向。

表 3.6 C++ 运算符和它们的运算过程

类别	名称	符号	计算	过程
一元运算	置后增量	++	左到右	2
	置后减量	--	左到右	2
	取地址	&	右到左	2
	取反	~	右到左	2
	强制类型转换	(type)	右到左	2
	逻辑非(NOT)	!	右到左	2
	负号	-	右到左	2
	加号	+	右到左	2
	前置增量	++	右到左	2
	前置减量	--	右到左	2
	数据的字节大小	sizeof	右到左	2
多元运算	求余	%	左到右	3
	相乘	*	左到右	3
	相除	/	左到右	3
加法运算	相加	+	左到右	4
	相减	-	左到右	4
移位运算	左移	<<	左到右	5
	右移	>>	左到右	5
关系运算	小于	<	左到右	6
	小于或等于	<=	左到右	6
	大于	>	左到右	6
	大于或等于	>=	左到右	6
	等于	==	左到右	7
	不等于	!=	左到右	7

续表

类别	名称	符号	计算	过程
位运算	AND(与)	&	左到右	8
	XOR(异或)	^	左到右	9
	OR(或)		左到右	10
逻辑运算	AND(与)	&&	左到右	11
	OR(或)		左到右	12
三元运算	条件表达式	?:	右到左	13
赋值运算	算术赋值	=	右到左	14
		+=	右到左	14
		-=	右到左	14
		*=	右到左	14
		/=	右到左	14
		%=	右到左	14
	移位赋值	>>=	右到左	14
		<<=	右到左	14
	按位赋值	&=	右到左	14
		=	右到左	14
		^=	右到左	14
	逗号	,	左到右	15

3.12 小结

今天的课介绍了各种C++运算符,并且讨论了如何使用这些运算符来操作数据。在此期间,你学习了下列内容:

- ☐ 算术运算符包括+, -, *, /和%(求余)。
- ☐ 算术表达式在复杂程度上各不相同。简单的表达式只包含单个数据项(文字、常量或变量)。复杂的表达式则包括多个运算符、函数、文字、常量和变量。
- ☐ 增量和减量运算符包含了前置和后置形式。C++允许你对存放了字符的变量、整数、甚至浮点数使用这些运算符。
- ☐ 算术赋值运算符能够使你编写简短的算术表达式,并且,在这些表达式中,原始的操作数也可以是接收表达式的结果的变量。
- ☐ sizeof 运算符可以返回数据类型或变量的字节大小。
- ☐ 强制类型转换允许你强制转换一个表达式的类型。
- ☐ 关系和逻辑运算符允许你建立逻辑表达式。但是,C++不支持一个预定义的布尔

类型,而是把“0”看作为假,把任何非零值看作为真。

- ☐ 布尔表达式把关系和逻辑运算符组合在一起,构成一个特殊的条件表达式。这些表达式使程序能够作出复杂决策。
- ☐ 条件表达式为你提供了简单的二元替换式 if—else 语句的简短形式。
- ☐ 位操作运算符用来执行按位与(AND)、或(OR)、异或(XOR)和取反(NOT)运算。另外,C++还支持<<和>>移位运算符。
- ☐ 位操作赋值运算符提供了简单的位操作语句的简短形式。

3.13 问与答

问:当你说明了一个变量,但从来又不给它赋值时,编译器将如何反应?

答:编译器发出一个警告,说明该变量没有被提及(或使用)。

问:检查一个变量(不妨把它叫做 i)的值是否在某两个值的范围中(例如,定义的变量 lowVal 和 hiVal),该布尔表达式是什么?

答:判断变量 i 的值是否在某个范围中的表达式是:

`(i) >= lowVal) && (i <= hiVal)`

问:检查一个变量(不妨把它叫做 i)的值是否在某两个值的范围内(例如,定义的变量 loVal 和 hiVal),该布尔表达式是什么?

答:判断变量 i 的值是否位于某个范围内的表达式是:

`(i > loval && i < hival)`

3.14 专题讨论

专题讨论提供了测验题来帮助你加深对所学的资料的理解,并且还提供了练习来锻炼你使用所学知识的能力。一定要在你继续下一章的课程之前,尽力去理解这些测验和练习的答案。答案在附录 B,中提供了。

3.14.1 测验

1. 下面程序的输出是什么?

```
#include <iostream.h>

main()
{
    int i = 3;
    int j = 5;
    double x = 33.5;
    double y = 10.0;
    cout << 10 + j % i << "\n";
    cout << i * i * 2 * i + 5 << "\n";
    cout << (19 + i + j) / (2 * j + 2) << "\n";
    cout << x / y + y / x << "\n";
}
```

```

    cout << i * x + j * y << "\n";
    return 0;
}

```

2. 下面程序的输出是什么?

```

main()
{
    int i = 3;
    int j = 5;

    cout << 10 + j % i ++ << "\n";
    cout << --i * i - 2 * i + 5 << "\n";
    cout << (19 + i + j) / (2 * j + 2) << "\n";
    cout << x / y + y + y / x << "\n";
    cout << i * x + j * y << "  \n";
    return 0;
}

```

3. 下面程序的输出是什么?

```

main()
{
    int i = 3;
    int j = 5;

    i += j;
    j *= 2;
    cout << 10 + j % i << "\n";
    i = 2;
    j /= 3;
    cout << i * i - 2 * i + j << "\n";
    return 0;
}

```

4. 下面程序的输出是什么?

```

#include <iostream.h>
main()
{
    int i = 5;
    int j = 10;
    cout << ((i <= ) ? "TRUE" : "FALSE") << "\n";
    cout << ((i > 0 && j < 100) ? "TRUE" : "FALSE") << "\n";
    cout << ((i > 0 && i < 10) ? "TRUE" : "FALSE") << "\n";
}

```

```
cout << ((i == 5 && i == j) ? "TRUE" : "FALSE") << "\n";  
return 0;  
}
```

3.14.2 练习

1. 运用条件运算符编写函数 `max`, 要求它返回值是两个整数中较大者。
2. 运用条件运算符编写函数 `min`, 要求它返回的值是两个整数中的较小者。
3. 运用条件运算符编写函数 `abs`, 要求它返回的一个整数的绝对值。
4. 运用条件运算符编写函数 `isodd`, 要求如果它的整型参量是一个奇数, 则返回 0; 否则就返回 1。

第四章 输入/输出(I/O)管理(第四天)

像 C 语言一样, C++ 没有定义作为它的核心部分的 I/O 操作。而是, C++ 和 C 依靠 I/O 库来提供必要的 I/O 支持。这些库主要针对非 GUI(Graphics User Interface 图形用户界面)环境, 比如 MS-DOS。并且通常用来处理 EasyWin 应用程序, 这就是它们在此书中的趣味所在。但是, 因为我的主要目的是教你如何编写 Windows 程序, 因此我将尽力减少对 I/O 库的讨论。今天的这节短课, 我们来看看其中一小部分的输入和输出操作, 以及 STDIO.H 和 IOSTREAM.H 头文件支持的函数。你将要学习下列主题:

- ☐ 格式化的输出流
- ☐ 输入流
- ☐ printf 函数

4.1 格式化的输出流

C++ 带来了一族可扩展的 I/O 库, 因为该语言的设计者们认识到, 从 C 语言继承进来的 STDIO.H 中的函数在处理类时, 具有它们的局限性(你将会在第十一章(第十一天)学到更多的有关类的知识)。所以, C++ 扩展了这些流的概念。我们可以回想一下, 在 C 语言中存在的流是一系列的、从计算机的某一部分流到另一部分的数据。在我至今已经介绍的程序中, 你已经见到提取运算符 << 与标准输出流 cout 的操作。你也见到了插入运算符 >> 和标准输入流 cin。在这一节中, 我将向你介绍流函数 width 和 precision, 它们用来格式化输出。C++ 流含有大量、能够更好地协调输出的函数。但是, 像我前面说明的那样, 因为这些函数用于非 GUI 接口, 我不想给你过多地介绍这些与 windows 编程无关的信息。width 函数指定了输出的宽度, 使用这个 cout 流函数的一般形式是:

```
cout width( widthofOutput);
```

precision 函数指定了浮点数的数字个数。使用这个 cout 流函数的一般形式是:

```
cout precision(numberOfDigits);
```

让我们来看看一个例子。程序列表 4.1 包含了程序 OUT1.CPP 的源代码。这个程序不要求输入, 并且可以显示格式化的整数、浮点数和字符, 这是使用 width 和 precision 流函数实现的。

程序列表 4.1 程序 OUT1.CPP 的源代码

```
1: // Program that illustrates C++ formatted stream output
2: // using the width and precision functions
3:
4: #include <iostream.h>
5:
6: main()
7: {
8:     short    aShort    = 4;
```

```

9:  int    anInt    = 67;
10: unsigned char aByte = 128;
11: char    aChar    = '@';
12: float    aSingle  = 355.0;
13: double   aDouble  = 1.130e+002;
14: // display sample expressions
15: cout.width(3); cout << int(aByte) << " + ";
16: cout.width(2); cout << anInt << " = ";
17: cout.width(3); cout << (aByte + anInt) << '\n';
18:
19: cout.precision(4); cout << aSingle << " / ";
20: cout.precision(4); cout << aDouble << " = ";
21: cout.precision(5); cout << (aSingle / aDouble) << '\n';
22:
23: cout << "The character in variable aChar is "
24:      << aChar << '\n';
25: return 0;
26: }

```

下面是程序列表 4.1 中的程序输出部分：

输出：

128+67=195

355/113=3.14159

The character in variable aChar is @

分析：

在程序列表 4.1 中，程序说明了一组不同类型的变量。第 15—17 行的语句使用流函数 `Width` 来指定，由一个 `cout` 语句显示的下一项的输出宽度。注意，它使用了六个语句来显示三个整数。另外，还要注意，在第 15 行中，程序使用了表达式 `int(aByte)`，强制把无符号字符类型转换成整型。如果没有这种类型转换，那么变量 `aByte` 的内容就是作为字符出现，而不是作为数字。如果使用输出流来显示具有缺少宽度的整数，那么事实上，我就可以用一个语句来代替那六个输出语句。

第 19—21 行包含了第二组用于浮点数的输出流语句。这些行中的语句含有流函数 `Precision`，用来指定要显示的数字的总数。并且，它又使用了六个 C++ 语句来输出三个浮点数。另外，如果又使用输出流来显示具有缺省宽度的这些数字，那么，我也可以用一个语句来代替那六个输出流语句。

4.2 输入流

像标准输出流一样，C++ 提供了标准输入流 `cin`。这个输入流能够读取预定义的数据类型，比如 `int`、`unsigned long` 和 `char`。典型地，你可以使用插入运算符 `>>` 来获取对预定的数据类型的输入。我以前介绍的程序都使用了 `>>` 运算符来输入一个数据项。C++ 流允许你把 `>>` 运算符排成长链来输入多个数据项。不过，就多个数据项来说，你必须遵循下列规则：

1. 在两个相连的数字之间要输入一个空格，把它们分隔开。

2. 在两个相连的字符之间是否输入一个空格,可以任意选择。
3. 在一个字符和一个数字之间,只要该字符是一个数字,则必需输入一个空格。
4. 输入流忽略空格。
5. 你可以在不同行上输入多个数据项。输入流语句在它们获得所有指定输入以前,不含全部执行。

注意:我现在推迟讨论字符串输入。第九章(第九天)包含了字符串和字符串的输入。

让我们来看看一个程序,该程序说明了多个数据项的输入和数据类型的不同组合。程序列表 4.2 显示了程序 IN1.CPP 的源代码。这个程序执行下列任务:

- ☐ 提示你输入三个数。
- ☐ 计算三个数的总和。
- ☐ 显示你输入的三个数的总和与平均值。
- ☐ 提示你输入三个字符。
- ☐ 显示你的输入。
- ☐ 提示你输入一个数、一个字符和一个数。
- ☐ 显示你的输入。
- ☐ 提示你输入一个字符、一个数和一个字符。
- ☐ 显示你的输入。

程序列表 4.2 程序 IN1.CPP 的源代码

```
1: // Program that illustrates standard stream input
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     double x, y, z, sum;
8:     char c1, c2, c3;
9:
10:    cout << "Enter three numbers separated by a space : ";
11:    cin >> x >> y >> z;
12:    sum = x + y + z;
13:    cout << "Sum of numbers = " << sum
14:        << "\nAverage of numbers = " << sum / 2 << "\n";
15:    cout << "Enter three characters : ";
16:    cin >> c1 >> c2 >> c3;
17:    cout << "You entered characters '" << c1
18:        << "', '" << c2 << "', and '"
19:        << c3 << "'\n";
20:    cout << "Enter a number, a character, and a number : ";
21:    cin >> x >> c1 >> y;
22:    cout << "You entered " << x << " " << c1 << " " << y << "\n";
23:    cout << "Enter a character, a number, and a character : ";
24:    cin >> c1 >> x >> c2;
25:    cout << "You entered " << c1 << " " << x << " " << c2 << "\n";
26:
27:    return 0;
28: }
```

下面是程序列表 4.2 中的程序输出部分:

输出:

```
Enter three numbers separated by a space : 1 2 3
Sum of numbers = 6
Average of numbers = 3
Enter three characters : ABC
You entered characters 'A', 'B', and 'C'
Enter a number, a character, and a number : 12A34.4
You entered 12 A 34.4
Enter a character, a number, and a character : A3. 14Z
You entered A 3.14 Z
```

分析:

在程序列表 4.2 中,程序说明了四个双精度型变量和三个字符型变量。第 10 行的输出语句提示你输入三个数。第 11 行的输入语句获取你的输入,并把它们存放在变量 `x`、`y` 和 `z` 中。而且,你必须在每两个数之间输入一个空格,或者你输入一个数的同时,再加入一个分隔线。该语句把你输入的第一个数存放在变量 `x` 中,第二个存放在变量 `y` 中,第三个存放在变量 `z` 中。这个顺序是由这些变量在第 11 行中出现的顺序决定的。第 12 行的语句计算变量 `x`、`y` 和 `z` 中的值的总和。第 13、14 行的输出语句显示你输入的那些数的总和与均值。

第 15 行的输出语句提示你输入三个字符。第 16 行的输入语句获取你的输入,并把它们存放在变量 `C1`、`C2` 和 `C3` 中。你输入时不需要用空格把那些字符分隔开。因此,你可以这样输入字符,比如 `1A2`、`Bob` 和 `1 D d`。第 17—19 行的输出语句显示你输入的字符——它们由空格分隔开。

第 20 行的输出语句提示你输入一个数,一个字符和一个数。第 21 行的输入语句接着把你的输入存放在变量 `x`、`C1` 和 `y` 中。如果那个字符能被解释成某个数的一部分,那么你就需要在字符与每个数之间输入一个空格。例如,如果你想输入 12,圆点字符和 55,那么你就这样输入: `12 . 55`。圆点周围的空格可以确保输入流不会把它看作为浮点数的小数点。第 22 行的输出语句显示你输入的值,它们分别由空格分隔开。第 23 行的输出语句提示你依次输入一个字符、一个数和一个字符。第 24 行的输入语句接着把你的输入存放在变量 `C1`、`x` 和 `C2` 中。如果那两个字符能被解释成那个数的一部分,那么你就需要在字符与数之间输入一个空格。例如,如果你想输入字符 `—`、12 和数字 0,那么你就应该这样输入: `— 12 0`。第 25 行的输出语句显示你输入的值,它们分别由空格分隔开。

4.3 printf 函数

作为一个初级 C++ 程序员,你有许多 I/O 函数可供选择。在这节中,我将讨论函数 `printf` 的格式化特点,该函数属于 C 语言的标准 I/O 函数。这个函数在头文件 `STDIO.H` 中进行了原型说明。

`printf` 函数提供了许多功能和格式控制。而单独的格式化指令的一般语法是: %

[flags][width][.precision][F|N|h|l]<type character>

其中 flags 选项表示输出对齐、数字符号,小数点和尾随的零。另外,这些标志还指定了实际的十六进制前缀。表 4.1 显示了在 printf 函数的格式串中的标志选项。

Width 选项表示显示的字符的最小数目。如果需要的话,printf 函数使用零和空格来填充输出。当宽度数是以 0 开始时,printf 函数优先使用 0 来填充,而不是用空格。当出现 * 字符来代替宽度数时,printf 函数则从该函数的参数表中获取实际的宽度数。指定所要求的宽度的参数必须出现在实际被格式化了的参数之前。下面是一个使用两个字符来显示整数 3 的例子,其中 2 是由 printf 的第三个参数指定的:

```
printf("% * d,3,2);
```

precision 选项指定可显示的字符的最大数目。如果你包括的是一个整数,那么 precision 选项就定义了可显示的数字的最小数目。当使用 * 字符代替 precision 选项时,printf 函数就从参数表中获取实际的精度 (precision)。并且,指定所要求的精度的参数必须出现在实际被格式化的参数之前。下面是一个使用 10 个字符来显示浮点数 3.3244 的例子,其中 10 是由 printf 函数的第 3 个参数指定的:

```
printf("% 7. * f",3.3244,10)
```

F、N、h 和 l 选项用来测定那些用于排除参数的缺省大小的选项。其中 F 和 N 选项分别与远程、近程指针连同在一起使用;h 和 l 选项分别用来表示 short int 或 long。

表 4.1 转义字符序列

序列	十进制数值	十六进制数值	任务
\a	7	0x07	响铃
\b	8	0x08	退格
\f	12	0x0C	走纸换页
\n	10	0x0A	换行
\r	13	0x0D	回车
\t	9	0x09	横向跳格
\v	11	0x0B	竖向跳格
//	92	0x5C	反斜杠字符\
\'	44	0x2C	单引号字符
\"	34	0x22	双引号字符
\?	63	0x3F	问号字符
\000			1 到 3 位的八进制数
\Zhhh 和 xhhh		0xbbbb	十六进制数

printf 函数要求你在每个 % 格式代码后面必须指定一种数据类型。表 4.2 显示了在 printf 函数的格式串中的标志选项。表格 4.3 显示了在函数 printf 的格式中所使用的数据类型字符。

表 4.2 printf 函数的格式串中的标志选项

格式选项	输出结果
-	在指定的区域内,进行左验证
+	显示一个值的加号或减号
空白	如果值为正,则开头显示空白; 如果值为负,则显示一个减号。
#	对十进制整数无效;对于十六进制整数显示以 0X 或 0x 开头;对于八进制整数显示以零开头;对于实数显示小数点。

表 4.3 printf 函数的格式串中所使用的数据类型字符

类别	类型字符	输出类型
字符	c	单个字符
	d	有符号的十进制整型
	i	有符号的十进制整型
	o	无符号的八进制整型
	u	无符号的十进制整型
	x	无符号的十六进制整型;使用的数字字符序列是: 01234567890abcdef
	X	无符号的十六进制整型;使用的数字字符序列是: 01234567890ABCDEF
指针	P	只显示近指针的偏移量为:0000;显示远指针为: SSSS: 0000
指向整型的指针	n	
实数	f	以格式[-]dddd.dddd 显示有符号数值。
	e	以格式[-]d.dddde[+ -]ddd 显示有符号的科学记数值。
	E	以格式[-]d.ddddE[+ -]ddd 显示有符号的科学记数值。
	g	根据数值和指定的精度,既可以使用 f 格式,也可以使用 e 格式显示有符号数值。
	G	根据数值和指定的精度,既可以使用 f 格式,也可以使用 E 格式显示有符号的数值。
字符串指针	s	显示一串字符,直到遇上字符串的空终止符为止。

注意: 尽管函数 `printf` 在 Windows 应用程序的输出中没有起什么作用, 但是它的相关函数 `sprintf` 却起了作用。后面的这个函数创建了一串字符, 其中包含了输出的格式化映像。我将在第九章(第九天)讨论这个函数, 并且在本书的后面课程中使用这个函数来创建一个对话框, 该对话框含有所包括的数的信息。

让我们来看看一个简单的例子。程序列表 4.3 显示了程序 OUT2.CPP 的源代码。我通过编辑程序列表 4.1 中的 OUT1.CPP 创建了这个程序。新程序使用 `printf` 函数显示了格式化的输出。该程序还使用三种不同的格式代码, 显示出了相同的浮点数。

程序列表 4.3 程序 OUT2.CPP 的源代码

```
1: // C++ program that uses the printf function for formatted output
2:
3: #include <stdio.h>
4:
5: main()
6: {
7:     short    aShort    = 4;
8:     int       aInt      = 67;
9:     unsigned char aByte = 128;
10:    char       aChar     = '0';
11:    float      aSingle   = 355.0;
12:    double     aDouble   = 1.1300e+002;
13:    // display sample expressions
14:    printf("%3d %c %2d = %3d\n",
15:           aByte, '+', aInt, aByte + aInt);
16:
17:    printf("Output uses the %lf format\n");
18:    printf("%6.4f / %6.4f = %7.5lf\n", aSingle, aDouble,
19:           aSingle / aDouble);
20:    printf("Output uses the %le format\n");
21:    printf("%6.4e / %6.4e = %7.5le\n", aSingle, aDouble,
22:           aSingle / aDouble);
23:    printf("Output uses the %lg format\n");
24:    printf("%6.4g / %6.4g = %7.5lg\n", aSingle, aDouble,
25:           aSingle / aDouble);
26:
27:    printf("The character in variable aChar is %c\n", aChar);
28:    printf("The ASCII code of %c is %d\n", aChar, aChar);
29:    return 0;
30: }
```

下面是程序列表 4.3 中的程序输出部分:

输出:

128 + 67 = 195

Output uses the %lf format

355.0000 / 113.0000 = 3.14159

Output uses the %le format

3.5500e+002 / 1.1300e+002 = 3.14159e+000

output uses the %lg format

355 / 113 = #3.1416

The character in variable aChar is @

The ASCII code of @ is 64

分析:

在程序列表 4.3 中,程序首先说明了一组不同类型的变量。第 14、15 行的输出语句使用 %d 和 %c 格式控制来显示整数和字符。表 4.4 显示了在第 14 行的 printf 语句中,不同的格式控制的效果。注意,printf 函数把输出中的第一个数据项从无符号的字符型转换成了一个整型。

表 4.4 第 14 行的 printf 语句中的不同格式控制的效果

格式的控制	数据项	数据类型	输出
%3d	aByte	无符号字型	整数
%c	+	字符型	字符
%2d	anInt	整型	整数
%3d	aByte + anInt	整型	整数

第 18 行的输出语句使用了格式控制 %6.4f, %6.4lf 和 %7.5lf 来显示变量 aSingle、变量 aDouble 和表达式 aSingle/aDouble。这些控制指定了精度值分别为 4、4 和 5 位数字,最小宽度分别为 6、6 和 7 个字符。最后两个格式控制表示它们显示一个双精度类型的值。

第 21 行的输出语句与第 18 行相似。主要的不同是,第 21 行的 printf 使用了 e 格式,而不是 f 格式。因此,在 printf 语句中的三个数据项是以科学记数法出现。

第 24 行的语句也是与第 18 行的语句相似。主要的不同是,第 24 行的 printf 使用了 g 格式,不是 f 格式。因此,在 printf 语句中的开始两项没有出现小数部分,因为它们都是整数。

第 27 行的输出语句使用 %c 格式控制来显示变量 aChar 的内容。第 28 行的输出语句显示了两次变量 aChar 的内容:一次是作为一个字符,一次是作为一个整数(更准确地说,是一个字符的 ASCII)。第 28 行的 printf 函数通过分别使用 %c 和 %d 格式控制来执行这项任务。

4.4 小结:

今天的课介绍了 IOSTREAM·H 和 SFDIO·H 头文件支持的基本输入和输出操作,以及函数。你从中学习了下列内容:

- ☐ 格式化的输出流使用 precision 和 width 函数,提供了几种基本的格式输出。
- ☐ 标准输入流支持插入运算符 >> 来获取 C++ 中的预定义数据类型的输入。
- ☐ printf 函数的格式串中所包含的格式代码增强了 printf 函数对输出形式的控制,及执行类型转换的能力。

4.5 问与答

问:怎样把>>或<<运算符形成链?

答:这些运算符中的每一个都能返回一个特定流的数据类型,而该数据类型又可以是另一个相似流运算符的输入。

问:为什么不能在 Windows 应用程序中使用 I/O 流运算符?

答:windows 应用程序具有一种根本不同的,与你相互作用的方式。当一个 EasyWin 程序(它仿效一个非 GUI 的 MS-DOS 应用程序)执行一个输入语句时,它就进入一种特殊模式,在该模式中它对键盘输入实行监控。相反,Windows 应用程序(它是 GUI 应用程序)则一直监控鼠标(它的运动和它的按钮)与键盘,并时刻报导监控这些事件的窗口的当前状况。GUI 和非 GUI 应用程序之间的最大不同点是,非 GUI 的输入函数对 GUI 应用程序无效。

4.6 专题讨论

专题讨论提供了测验题,帮助你加深对所学的知识理解;同时,它还提供了练习,锻炼你使用你所学的知识的能力。在你继续下一章的课程之前,一定要尽力去理解测验和练习答案,所有的答案在附录 B,“答案”中都提供了。

4.6.1 测验

1. 下面的语句有什么错误?

```
cout<<"Enter a number">>x;
```

2. 下面的语句会发生什么情况?

```
cout<<"Enter three munber:";
```

```
cin>>x>>y>>x;
```

4.2.6 练习

1. 编写程序 OUT3.CPP,显示一张 2 到 10 范围内的整数的平方根表。要求使用 MATH.H 头文件来输入,用于并计算一个双精度型参数的平方根的 sqrt 函数。因为我还没有讨论 C++ 循环,所以,可使用重复语句来显示不同的值。但是,必须运用格式控制 %3.0f 和 %3.4f 来分别显示那个数和它的平方根。
2. 编写程序 OUT4.CPP,该程序提示你输入一个整数,然后显示它的十六进制和八进制形式。要求使用 printf 格式控制来实现十进制、十六进制和八进制之间的转换。

第五章 判断结构(第五天)

不同的编程语言支持的判断结构也各不相同。

新术语:判断结构(Decision-making constructs)使你的应用程序能够检查条件,并以此来决定它的执行过程。

今天的课介绍 C++ 中的判断结构,它包括下列内容:

- ☐ 单选择的 if 语句
- ☐ 双选择的 if-else 语句
- ☐ 多选择的 if-else 语句
- ☐ 多选择的 switch 语句
- ☐ 嵌套的判断结构

5.1 单选择的 if 语句

不像许多编程语言那样,C++ 在任何形式的 if 语句中都没有关键字 then。这种语言特点可能使你要问,if 语句如何把测试条件从执行语句中分隔开来。答案是,C++ 指示你用圆括号把测试条件括起来。

新术语:if 语句是单元选择(single _ alternative)语句。

语法

单元选择的 if 语句

单元选择的 if 语句的一般语法是:

```
if(condition)
```

```
    statement;
```

这是对于单个可执行语句而言的。对于一序列可执行语句,则为:

```
if(tested _ condition){  
    <sequence of statement >  
}
```

例子:

```
if (numberOfLines < 0)  
    numberOfLines = 0;  
if ((height _ 54) < 3){  
    area = length * width;
```



```

volume = area * height;
}

```

C++使用开和关大括号({})来定义一个语句块。图 5.1 显示了在单选择的 if 语句中的流程。

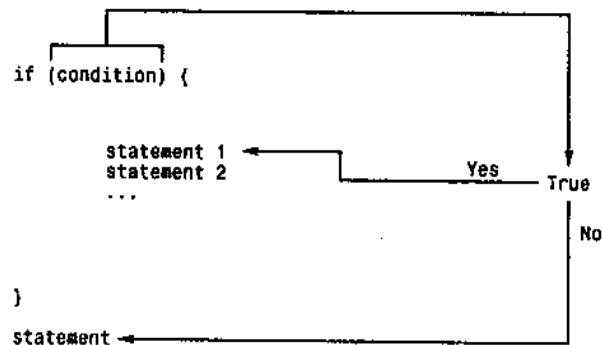


图 5.1 单选择 if 语句的程序流程图

让我们来看看一个例子。程序列表 5.1 显示了一个单选择的 if 语句的程序。该程序提示你输入一个非零数,并把它存放在变量 **x** 中。如果 **x** 的值是非零的,那么程序显示 **x** 的倒数。

程序列表 5.1 程序 IF1.CPP 的源代码

```

1: // Program that demonstrates the single-alternative if statement
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     double x;
8:     cout << "Enter a non-zero number : ";
9:     cin >> x;
10:    if (x != 0)
11:        cout << "The reciprocal of " << x
12:        << " is " << (1/x) << "\n";
13:    return 0;
14: }

```

下面是程序列表 5.1 中的程序输出部分:

输出:

Enter a non _ zero number: 25

The reciprocal of 25 is 0.04

分析:

在程序列表 5.1 中,程序在函数 main 中说明了一个双精度型变量 **x**。第 8 行的输出语句提示你输入一个非零数。第 9 行的输入语句把你的输入存入变量 **x** 中。第 10 行的 if 语句判断 **x** 是否不等于零。如果这个条件是真,那么程序就执行第 11 行和第 12 行的输出语句。该语句显示 **x** 的值和它的倒数 $\frac{1}{x}$ 。如果测试的条件是假,那么程序就放弃第 11、12 行的语句,重新从第 13 行的语句开始执行。

5.2 双选择的 if—else 语句

在 if 语句的双选择形式中,关键字 else 用于分隔每个选择执行的语句。

新术语:双选择的 if—else(dual—alternatile if—else)语句为你提供了两种选择执行路线,它们是基于测试条件的布尔值的。

语法

二元选择的 if—else 语句

二元选择的 if—else 语句的一般语法是

```
if (condition)
    statement1;
else
    statement2;
```

这是对于每个从句中的单个可执行语句,而对于两个句中一序列可执行语句,其语法为:

```
if (tested—condition{
    <sequence #1 of statements>
}
else {
    <sequence #2 of statements>
}
```

例子:

```
if (moneyInAccount > withdraw){
    moneyInAccount = withdraw;
    cout << "You withdrew $" << withdraw << "\n";
    cout << "Balance is $" << money InAccount << "\n";
}
else {
    cout << "Cannot withdraw $" << withdraw << "\n";
    cout << "Account has $" << moneyInAccount << "\n";
}
```

图 5.2 显示了在双选择的 if—else 语句中的程序流程。

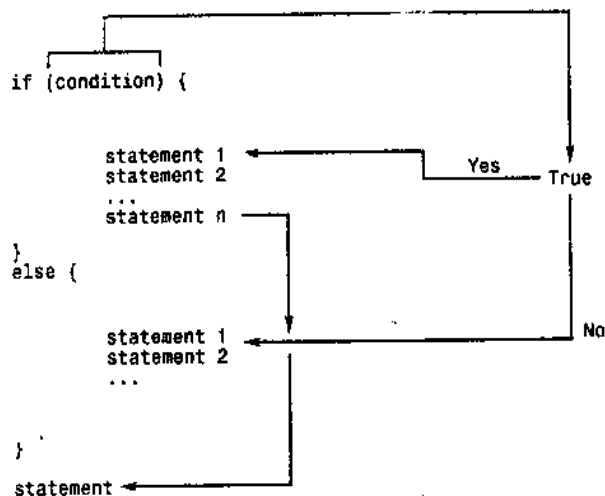


图 5.2 双选择的 if-else 语句中的程序流程图

让我们来看看一个例子,该例子使用了双 if-else 语句。程序列表 5.2 包含了程序 IF2.CPP 的源代码。该程序提示你输入一个字符,然后判断你输入的是不是一个字母。程序的输出把你的输入按照是一个字母还是一个非字母字符进行分类。

程序列表 5.2,程序 IF2.CPP 的源代码

```

1: // Program that demonstrates the dual-alternative if statement
2:
3: #include <iostream.h>
4: #include <ctype.h>
5:
6: main()
7: {
8:     char c;
9:     cout << "Enter a letter : ";
10:    cin >> c;
11:    // convert to uppercase
12:    c = toupper(c);
13:    if (c >= 'A' && c <= 'Z')
14:        cout << "You entered a letter\n";
15:    else
16:        cout << "Your input was not a letter\n";
17:    return 0;
18: }
  
```

下面是程序列表 5.2 中的程序输出部分:

输出:

```

Enter a letter:g
YOU entered a letter
  
```

分析:

在程序列表 5.2 中,程序在第 8 行中说明了字符型变量 c。第 9 行的输出语句提示你

输入一个字母。第 10 行的输入语句获取你的输入,并把它存放在变量 `c` 中。第 12 行的语句通过调用函数 `topper9`(在头文件 `CTYPE.H` 中进行了原型说明),把该中的值转换成大写。这种字符形式转换简化了第 13 行的 `if-else` 语句中的测试条件。`if-else` 语句判断变量 `C` 包含的字符是否在在 `A` 到 `Z` 范围中。如果此条件为真,那么程序执行第 14 行的输出语句。这个语句显示了一条信息说明你输出语句。相反,如果测试的条件为假,那么程序执行第 16 行的 `else` 从句的语句。该语句也显示一第信息,说明你输入的不是一个字母。

5.3 if 语句潜在的问题

双选择的 `if` 语句具有一个潜在的问题。当 `if` 从句包括了另一个单选择的 `if` 语句时,该问题就会发生。这时,编译器认为 `else` 从句附属于嵌套的 `if` 语句。(一个嵌套的 `if` 语句是在 `if` 或 `else` 从句中包含了另一个 `if` 语句。你将会在下一节中学到更多的有关嵌套的知识)。下面是一个例子:

```
if (i > 0)
    if (i == 10)
        cout << "You guessed the magic number";
    else
        cout << "Number is out of range";
```

在这个代码段中,当变量 `i` 是一个正数而不是 10 时,代码就显示信息: `Number is out of range`。而编译器处理这些语句,就好像该代码段意谓:

```
if (i > 0)
    if (i == 10)
        cout << "You guessed the magic number";
    else
        cout << "Number is out of range";
```

要修改这个问题,必须把嵌套的 `if` 语句括在一个语句块中:

```
if (i > 0)
{
    if (i == 10)
        cout << "You guessed the magic number";
}
else
    cout << "Number is out of range";
```

5.4 多选择的 if-else 语句

C++ 允许你嵌套 `if-else` 语句来创建一个多选择的形式。这种选择为你的应用程序提供了许多功能和柔性。

新术语:多选择的 if—else (multiple—alternative if—else) 语句包含了嵌套的 if—else 语句。

语法

多选择的 if—else 语句

多选择的 if—else 语句的一般语法是

```
if (tested—condition1)
    statement1; | { <sequence #1 of statement> }
else if (tested—condition2)
    statement2; | { <sequence #2 of statement> }
else if ( tested—conditionN)
    statementN; | { <sequence #N of statement> }
[else
    statementN+1; | {<sequence #N+1 of statement> }]
```

例子:

Example:char op;

```
int opOk = 1;
double x, y, z;
cout << "Enter operand1 operator operand2: ";
cin >> x >> op >> y;
if (op == '+')
    z = x + y;
else if (op == '-')
    z = x - y;
else if (op == '*')
    z = x * y;
else if (op == '/' && y != 0)
    z = x / y;
else
    opOk = 0;
```

多选择的 if—else 语句执行一系列级串联式的测试,直到下面之一发生为止:

1. 在 if 从句或 else if 从句中的条件之一为真。这时,执行伴随的语句(如果有一个 else 从句)。
2. 测试的条件没有一个是真的。程序执行所有相应的 else 从句中的语句。

图 5.3 显示了在多选择的 if—else 语句中的流程。

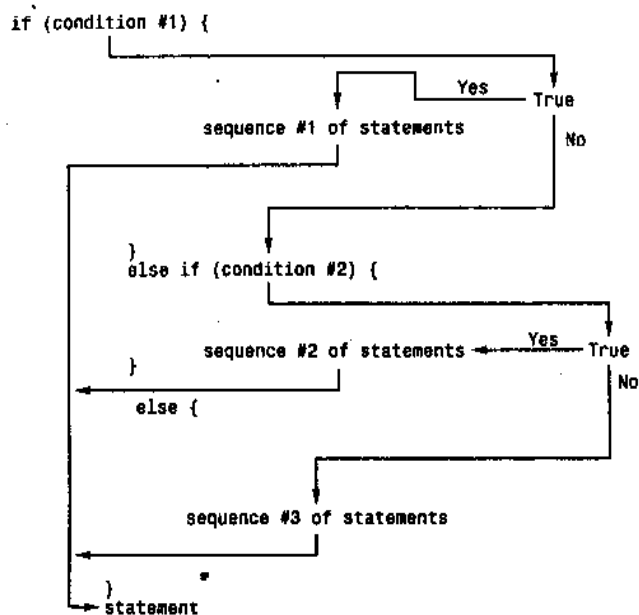


图 5.3 多选择的 if-else 语句中的程序流程

让我们来看看一个例子。程序列表 5.3 显示了程序 IF3.CPP 的源代码。这个程序提示你输入一个字符，然后使用多选择的 if-else 语句来判断你的输入是否属于下列情况之一：

- ☐ 一个大写字母
- ☐ 一个小写字母
- ☐ 一个数字
- ☐ 一个非字母与数字的字符

程序列表 5.3 程序 IF3.CPP 的源代码

```

1: // Program that demonstrates the multiple-alternative if statement
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     char c;
8:     cout << "Enter a character : ";
9:     cin >> c;
10:    if (c >= 'A' && c <= 'Z')
11:        cout << "You entered an uppercase letter\n";
12:    else if (c >= 'a' && c <= 'z')
13:        cout << "You entered a lowercase letter\n";
14:    else if (c >= '0' && c <= '9')
15:        cout << "You entered a digit\n";
16:    else
17:        cout << "You entered a non-alphanumeric character\n";
18:    return 0;
19: }
  
```

下面是程序列表 5.3 的程序输出部分：

输出：

Enter a character : l

You entered a non-alphanumeric character.

分析：

在程序列表 5.3 中，程序在第 7 行说明了字符型变量 *c*。第 8 行的输出语句提示你输入一个字母。第 9 行的输入语句获取你的输入，并把它存放在变量 *c* 中。多选择的 *if-else* 语句测试下列条件：

1. 在第 10 行中，*if* 语句判断变量 *c* 是否含有一个 A 到 Z 范围内的字母。如果这个条件为真，程序就执行第 11 行的输出语句。这个语句确认你输入了一个大写的字母。然后程序从第 18 行继续执行下去。
2. 如果第 10 行的条件为假，程序就跳到第 12 行的第一个 *else if* 从句。在那里程序再判断变量 *c* 是否含有一个 a 到 z 范围内的字母。如果条件为真，程序就执行第 13 行的输出语句。这个语句确认你输入了一个小写的字母。然后，程序从第 18 行继续执行下去。
3. 如果第 12 行的条件为假，程序就跳到第 14 行的第二个 *else if* 从句。在那里，程序判断变量 *c* 是否有一个数字。如果这个条件为真，程序就执行第 15 行的输出语句。该语句确认你输入了一个数字。然后程序从第 18 行继续执行下去。
4. 如果第 14 行的条件为假，程序就跳到第 16 行最终的那个 *else* 从句，并执行第 17 行的输出语句。这个语句显示一条信息，告诉你输入的既不是一个字母，也不是一个数字。

5.5 switch 语句

switch 语句提供了一种特殊形式的多选择判断。它使你能够通过检查一个与整数相容的表达式不同值，选择合适的行动路线。

语法

switch 语句

switch 语句的一般语法是

```
switch (expression) {  
    case constant1_1:  
[ case constant1_2 ... ]  
    <one or more statements>  
    break;  
    case constant2_1:  
[ case constant2_2: ... ]  
    <one or more statements>  
    break;  
    ...  
}
```

```

    case constantN_1:
[   case constantN_2: ... ]
        <one or more statements>
        break;
default:
        <one or more statements>
}

```

例子:

```

OK = 1;
switch (op) {
    case '+':
        z = x + y;
        break;
    case '-':
        z = x - y;
        break;
    case '*':
        z = x * y;
        break;
    case '/':
        if (y != 0)
            z = x / y;
        else
            OK = 0;
        break;
    default:
        OK = 0;
}

```

使用 switch 语句的规则是:

1. switch 语句要求一个与整数相关的值。这个值可以是一个常量、变量、函数调用或表达式。switch 语句不处理浮点数类型。
2. 每个 case 标号的值必须是一个常量。
3. C++ 不支持 case 标号带有数值的范围。而是,每个值必须作为一个单独的 case 标号。
4. 你必须在每组可执行语句后面使用一个 break 语句。break 语句使程序在当前 switch 语句结束后重新继续向下执行。如果你不使用 break 语句,程序就从接下去的 case 标号开始执行。
5. default 从句是一个最终可执行的从句。
6. 在每个 case 标号或每组 case 标号中的语句不需要用开和关大括号括起来。

注意:如果你有大范围的相邻数值,那么,带有数值范围的单个 case 标号的缺乏将会

使得运用多选择的 if-else 语句更具吸引力。

图 5.4 显示了在多选择的 switch 语句中的程序流程。

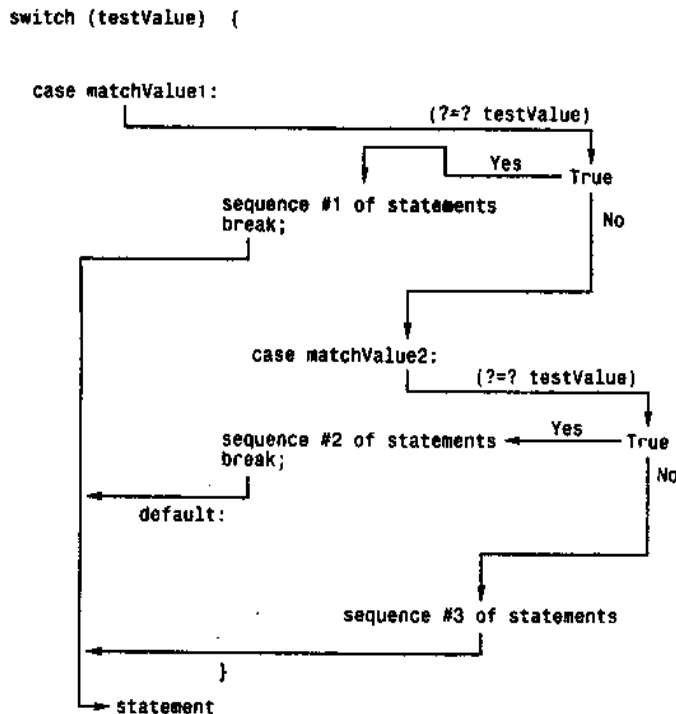


图 5.4 多选择的 switch 语句中的程序流程

现在让我们来看看一个例子,该例子使用了 switch 语句。程序列表 5.4 包含了程序 SWITCH1.CPP 的源代码,这是通过编辑程序列表 5.3 而获得的。新程序执行同样的任务,即对你输入的字符分类,但这次使用了一个 switch 语句。

程序列表 5.4 程序 SEITCH1.CPP 的源代码

```
1: // Program that demonstrates the multiple-alternative switch statement
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     char c;
8:     cout << "Enter a character : ";
9:     cin >> c;
10:    switch (c) {
11:        case 'A':
12:        case 'B':
13:        case 'C':
14:        case 'D':
15:            // other case labels
16:            cout << "You entered an uppercase letter\n";
17:            break;
18:        case 'a':
19:        case 'b':
20:        case 'c':
```

```

21:     case 'd':
22:         // other case labels
23:         cout << "You entered a lowercase letter\n";
24:         break;
25:     case '0':
26:     case '1':
27:     case '2':
28:     case '3':
29:         // other case labels
30:         cout << "You entered a digit\n";
31:         break;
32:     default:
33:         cout << "You entered a non-alphanumeric character\n";
34: }
35: return 0;
36: }

```

下面是程序列表 5.4 中的程序输出部分：

输出：

Enter a character : 2

You entered a digit.

分析：

在程序列表 5.4 中，程序说明了一个字符型变量 `c`。第 8 行的输出语句提示你输入一个字符。第 9 行的语句把你的输入存放在变量 `c` 中。`switch` 语句从第 10 行开始，第 11—14 行包含了字母 A 到 D 的 `case` 标号。为了使程序简短一点，放弃了用其余大写字母作为 `case` 标号。如果变量 `c` 中的字符与第 11—14 行的任一值匹配，程序就执行第 16 行的输出语句。该语句确认你输入了一个大写字母（因为我缩减了 `case` 标号的数目，所以只要你输入了一个 A 到 D 的字母，程序就会执行第 16 行的语句）。第 17 行的 `break` 语句使程序跳到第 35 行执行，结束了 `switch` 语句。

如果变量 `c` 中的字符不与第 11—14 行的任何 `case` 标号匹配，程序就从第 18 行开始执行，在那里，程序遇到了另一组 `case` 标号。这些标号应该表示小写的字符。正像你所见到的那样，我减少了标号的数目，目的是为了缩短程序。如果变量 `c` 中的字符与第 18—21 行的值匹配，那么程序就执行第 23 行的输出语句。这个语句确认你输入了一个小写的字母（因为减少了 `case` 标号的数目，所以，只要你输入了一个 a 到 d 的字母，程序就执行第 23 行的语句）。第 24 行的 `break` 语句使程序跳到第 35 行执行，结束了 `switch` 语句。

如果变量 `c` 的字符又不与第 18 行至第 21 行的任何 `case` 标号匹配，那么程序就从第 25 行开始执行，在那里，它又遇到了一组 `case` 标号。这些标号应该表示数字。这时，你再会看到，我减少了标号的数目，目的是为了缩短程序。如果变量 `c` 中的字符与第 25 行至第 28 行的任何值匹配，那么程序就执行第 30 行的输出语句。该语句确认你输入了一个数字（因为我减少了 `case` 标号的数目，所以，只要你输入 0 到 3 之间的数字，程序就会执行第 30 行的语句）。第 31 行的 `break` 语句使程序跳到第 35 行执行，结束了 `switch` 语句。

如果变量 `c` 中的字符又不与第 25 行至第 28 行的任何 `case` 标号匹配，程序就跳到第 32 行的最后相配的从句执行。然后程序执行第 33 行的输出语句。该语句告诉你，你输入了一个非字符与数字的字符。

5.6 嵌套的判断结构

你经常需要使用嵌套的判断结构来处理重要的条件。嵌套判断结构使你能够使用一种分散解决的方法来处理复杂的条件。外层结构帮助你测试初步的或者更一般的条件。内层结构帮助你处理更特殊的条件。

让我们来看看一个例子。程序列表 5.5 显示了程序 IF4.CPP 的源代码。这个程序提示你输入一个字符。然后程序判断你输入的是否是一个大写的字母，一个小写的字母，还是一个非字母的字符。该程序显示一条信息，对你的输入进行分类。

程序列表 5.5 程序 IF4.CPP 的源代码

```
1: // Program that demonstrates the nested if statements
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     char c;
8:     cout << "Enter a character : ";
9:     cin >> c;
10:    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))
11:        if (c >= 'A' && c <= 'Z')
12:            cout << "You entered an uppercase letter\n";
13:        else
14:            cout << "You entered a lowercase letter\n";
15:    else
16:        cout << "You entered a non-letter character\n";
17:    return 0;
18: }
```

下面是程序列表 5.5 中的程序输出部分：

输出：

Enter a character ; a

You entered a lowercase letter

分析：

在程序列表 5.5 中，程序说明了一个字符型变量 c。第 8 行的输出语句提示你输入一个字符。第 9 行的语句把你的输入存放在变量 c 中。该程序使用了嵌套的 if-else 语句，分别是在第 10 行和第 11 行开始。外层的 if-else 语句判断变量 c 是否含有一个字母。如果测试的条件为真，程序就执行第 11 行的内层 if-else 语句。否则，程序从外层 if-else 语句的 else 从句开始执行，并接着执行第 16 行的输出语句。该语句告诉你，你输入的不是一个字母。

同时，程序使用了内层 if-else 语句，进一步检查外层 if-else 语句的条件。第 11 行的 if-else 语句判断变量 c 是否含有一个大写的字母。如果这个条件为真，程序就执行第 12 行的输出语句。否则，程序执行第 14 行的 else 从句。这些输出语句告诉你，你输入了一个大写字母而不是一个小写字母。在执行了内层 if-else 语句以后，程序跳到第 17

行,结束外层 if-else 语句的执行。

5.7 小结

今天的课介绍了 C++ 中的各种判断结构,它们包括如下:

□ 单选择的 if 语句,比如:

```
if (tested_condition)
    statement; | { <sequence of statements> }
```

□ 双选择的 if-else 语句,比如:

```
if (tested_condition)
    statement1; | { <sequence #1 of statements> }
else
    statement1; | { <sequence #1 of statements> }
```

□ 多选择的 if-else 语句,比如:

```
if (tested_condition1)
    statement1; | { <sequence #1 of statements> }
else if (tested_condition2)
    statement2; | { <sequence #2 of statements> }
...
else if (tested_conditionN)
    statementN; | { <sequence #N of statements> }
[else
    statementN+1; | { <sequence #N+1 of statements> }]
```

□ 多选择的 switch 语句,比如:

```
switch (caseVar) {
    case constant1_1:
    case constant1_2:
    <other case labels>
        <one or more statements>
        break;
    case constant2_1:
    case constant2_2:
    <other case labels>
        <one or more statements>
        break;
    ...
    case constantN_1:
    case constantN_2:
    <other case labels>
        <one or more statements>
        break;
    default;
```

```
<one or more statements>
```

```
break;
```

```
}
```

你还学习了下列内容:

☐ if 语句要求你遵守两条规则:

1. 测试条件必须用圆括号括起来。
2. 语句块必须用一对(开和关)大括号括起来。

☐ 嵌套的判断结构能够使你运用一种分散解决的方法来处理复杂的条件。外层结构帮助你测试初步的或者更一般的条件。内层结构帮助你处理更特殊的条件。

5.8 问与答

问:在一个 if 语句的从句中,对于缩进书写语句,C++强加了任何规则吗?

答:没有。缩进纯粹是由你自己决定。典型的缩进范围是 2~4 个空格。使用缩进将使你的程序具有更好的可读性。下面是 if 语句带有无缩进的从句(语句)的情况:

```
if (i > 0)
```

```
    j = i * *;
```

```
else
```

```
    j = 10 - i;
```

比较一下带有缩进形式的程序的可读性:

```
if (i > 0)
```

```
    j = i * *;
```

```
else
```

```
    j = 10 - i;
```

显然,缩进形式更容易阅读。

问:编写一个 if-else 语句的条件的规则是什么?

答:有两种方法。第一种推荐你编写条件时,最好使它为真,而不要为假。第二种推荐你避免使用否定表达式(那些使用关系运算符!=和逻辑运算符!的表达式)。编程人员把这种 if 语句

```
if (i != 0)
```

```
    j = 100 / i;
```

```
else
```

```
    j = 1;
```

翻译成下列等价形式,

```
if (i == 0)
```

```
    j = 1;
```

```
else
```

```
    j = 100 / i;
```

即使变量 i 存有 0 的可能性很小,也是这样。

问:怎样处理下面的条件,该条件除以了一个可能为零的变量?

```
if (i != 0 && 1/i > 1)
```

```
j = i * i;
```

答: C++并不总是计算整个测试条件。当逻辑表达式中某一项可以使得整个表达式为真或假, 而不管其他项的值如何时, 则只进行部分计算。在这种情况下, 如果变量 i 是 0, 操作系统就不会计算那项 $1/i > 1$ 。这是因为 $i \neq 0$ 这项为假, 将使得整个表达式也为假, 而不管第二项产生什么。

问: 在多选择的 if-else 和 switch 语句中, 必须包括一个 else 或 default 从句吗?

答: 程序员们都极力推荐要包括这些最后相配的从句, 以便确保多选择语句能处理所有条件。

5.9 专题讨论

专题讨论提供了测验题, 帮助你强化对所学资料的理解, 同时, 还提供了练习来锻炼你使用所学知识的能力。答案在附录 B 中提供。

5.9.1 测验

1. 用单个 if 语句简化下面嵌套的 if 语句:

```
if (i > 0)
    if (i < 10)
        cout << "i = " << i << "\n";
```

2. 用单个 if 语句简化下面的 if 语句:

```
if (i > 0) {
    j = i * i;
    cout << "j = " << j << "\n";
}
if (i < 0) {
    j = 4 * i;
    cout << "j = " << j << "\n";
}
if (i == 0) {
    j = 10 * i;
    cout << "j = " << j << "\n";
}
```

3. 下面的 if 语句执行 if-else 语句同样的任务, 是对还是错?

```
if (i < 0) {
    i = 10 * i;
    j = i * i;
    cout << "i = " << i << "\n";
    cout << "j = " << j << "\n";
}
if (i >= 0) {
    k = 4 * i + 1;
```

```

        cout << "k = " << k << "\n";
    }
    if (i < 0) {
        i = 10 - i;
        j = i * i;
        cout << "i = " << i << "\n";
        cout << "j = " << j << "\n";
    }
    else {
        k = 4 * i + 1;
        cout << "k = " << k << "\n";
    }
}

```

4. 简化下面的 if-else 语句

```

if (i > 0 && i < 100)
    j = i * i;
else if (i > 10 && i < 50)
    j = 10 + i;
else if (i >= 100)
    j = i;
else
    j = 1;

```

5. 下面的 if 语句有什么错误?

```

if (i > (1 + i * i)) {
    j = i * i;
    cout << "i = " << i << " and j = " << j << "\n";
}

```

5.9.2 练习

1. 编写一个求解一元二次方程根的程序 IF5.CPP。一元二次方程是:

$$AX^2 + BX + C = 0$$

该方程的根是:

$$root1 = (-B + \sqrt{B^2 - 4AC}) / (2A)$$

$$root1 = (-B - \sqrt{B^2 - 4AC}) / (2A)$$

如果平方根中的某项为负,那么两根都是复数。如果平方根项是零,那么两根相同,并且都等于 $-B/(2A)$ 。

2. 编写程序 SWITCH2.CPP,它能完成一个简单的四功能计算器的功能。该程序应该提示你输入操作数和运算符,并显示输入和结果。其中,还要求包括对于错误运算和除数为零的错误检查。

第六章 循环(第六天)

回想一下第三章(第三天)的内容。就会发现,循环是功能很强的语言结构,它能够使计算机成功地完成重复性的任务。计算机能快速、准确而且不厌其烦地重复执行任务——在某些领域,计算机似乎比人做的还要好。今天的课介绍 C++ 中的下列循环:

- ☐ for 循环语句
- ☐ do-while 循环语句
- ☐ while 循环语句
- ☐ 跳过循环
- ☐ 退出循环
- ☐ 嵌套循环

6.1 for 循环

C++ 中的 for 循环是一种多功能的循环,因为它提供了固定的和有条件的重复。而且 for 循环的后面这个特点也与其他编程语言中的 for 循环的典型用途不完全相同,比如 Pascal 和 Basic。

语法

for 循环

for 循环语句的一般语法是

```
for (<initialization of loop control variables>;  
    <loop continuation test>;  
    <increment/decrement of loop control variables>)
```

例子:

```
for (i = 0; i < 10; i++)  
    scout << "The cube of " << i << " = " << i * i * i << "\n";
```

for 循环语句有三个组成部分,并且它们都是可选择的。第一部分初始化循环控制变量(C++ 允许你使用多个循环控制变量)。循环的第二部分是判断循环是否要进行另一次重复的条件。for 循环的最后一部分是循环控制变量增值或减值的从句。

注意: C++ 的 for 循环允许你说明循环控制变量。这样的变量可存在于循环的活动范围中。

让我们来看一个例子。程序列表 6.1 包含了程序 FOR1.CPP 的源代码。该程序提示你定义整数的范围——通过指定它的上边界和下边界。然后程序计算你指定范围中的整

数总和,以及均值。

程序列表 6.1 程序 FOR1.CPP 的源代码

```
1: // Program that calculates a sum and average of a range of
2: // integers using a for loop
3:
4: #include <iostream.h>
5:
6: main()
7: {
8:     double sum = 0;
9:     double sumx = 0.0;
10:    int first, last, temp;
11:
12:    cout << "Enter the first integer : ";
13:    cin >> first;
14:    cout << "Enter the last integer : ";
15:    cin >> last;
16:    if (first > last) {
17:        temp = first;
18:        first = last;
19:        last = temp;
20:    }
21:    for (int i = first; i <= last; i++) {
22:        sum++;
23:        sumx += (double)i;
24:    }
25:    cout << "Sum of integers from "
26:        << first << " to " << last << " = "
27:        << sumx << "\n";
28:    cout << "Average value = " << sumx / sum;
29:    return 0;
30: }
```

下面是程序列表 6.1 中的程序输出部分:

输出:

Enter the first integer : 1

Enter the last integer : 100

Sum of integers from 1 to 100 = 5050

Average value = 50.5

分析:

在程序列表 6.1 中,程序在函数 main 中说明了一组整型和双精度型的变量。同时,该函数还把求和变量 sum 和 sumx 初始化为零。第 12—15 行的输入与输出语句提示你输入定义数值的范围的整数。然后,程序把这些整数存放在变量 first 和 last 中。第 16 行的 if 语句判断变量 first 的值是否比变量 last 的值大。如果这个条件为真,程序就执行第 17—19 行的语句块。这些语句把变量 first 和 last 的值进行调换。因此,if 语句确保了变量 first 中的整数小于或等于变量 last 中的整数。

程序在第 21 行使用了 for 循环执行求和运算。该循环说明了它自身的控制变量 i,并且用变量 first 的值初始化了该变量。循环连续条件是 $i \leq \text{last}$ 。该条件表示,只要 i 小于或等于变量 last 的值,循环就重复。循环增值部分是 $i++$,对于每次重复,它都

使循环控制变量 i 自增 1。这个循环含有两条语句。第一条语句使变量 sum 的值增加。第二条语句把变量 i 的值(把它强制转换成双精度型以后)加到变量 $sumx$ 。

注意:我可以重新编写 `for` 循环,把第一个循环语句移到循环增值部分:

```
for (int i = first; i <= last; i++, sum++)
    sumx += (double) i;
```

第 25—27 行的输出语句显示你指定范围的整数的总和与均值。

为了证明 `for` 循环的柔性,我创建了程序 `FOR2.CPP`,见程序列表 6.2 中所示,它是由编辑程序 `FOR1.CPP` 而得来的。这两个程序执行同样的任务,并且与交互过程也是一样的。所作的改变是在第 10 行和第 21—25 行。第 10 行说明了循环控制变量。在第 21 行,使用变量 `first` 的值初始化了变量 i 。`for` 循环位于第 22 行。该循环没有初始化部分,因为在第 21 行已经进行了初始化。另外,删除了循环增值部分,而在第 24 行对变量 i 运用了后置增值运算符来弥补它。

程序列表 6.2 程序 `FOR2.CPP` 的源代码

```
1: // Program that calculates a sum and average of a range of
2: // integers using a for loop
3:
4: #include <iostream.h>
5:
6: main()
7: {
8:     double sum = 0;
9:     double sumx = 0.0;
10:    int first, last, temp, i;
11:
12:    cout << "Enter the first integer : ";
13:    cin >> first;
14:    cout << "Enter the last integer : ";
15:    cin >> last;
16:    if (first > last) {
17:        temp = first;
18:        first = last;
19:        last = temp;
20:    }
21:    i = first;
22:    for (; i <= last; ) {
23:        sum++;
24:        sumx += (double)i++;
25:    }
26:    cout << "Sum of integers from "
27:        << first << " to " << last << " = "
28:        << sumx << "\n";
29:    cout << "Average value = " << sumx / sum;
30:    return 0;
31: }
```

下面是程序列表 6.2 中的程序输出部分:

输出:

Enter the first integer : 10

Enter the last integer : 100

Sum of integers from 10 to 100 = 5005

Average value = 55

6.2 使用 for 循环的开循环

当向你介绍 C++ 的 for 循环时,就说明了 for 循环的三个组成部分是可选择的。事实上,C++ 允许你让这三个部分都空着!

新术语:当你让一个循环的三个组成部分都空着时,结果就是一个开循环。

值得指出的是,其他语言,比如 Ada 和 Modula-2,都支持正规的开循环,并且提供了途径来退出这些循环。C++ 允许你以下面两种方式之一退出循环:

1. break 语句使程序跳到当前循环的末端以后重新开始执行。当你希望退出一个 for 循环,重新执行程序的剩余部分时,可以使用 break 语句。
2. exit 函数(在头文件 STDLIB.H 中说明)允许你退出程序。如果你想停止重复操作,并且退出程序,那么你可以使用 exit 函数。

让我们来看看一个例子。程序列表 6.3 包含了程序 FOR3.CPP 的源代码。该程序使用了一个开循环,反复提示你输入一个数。程序获取你的输入,并显示它和它的倒数。然后程序又问你是否希望计算另一个数的倒数。如果你输入了字母 Y 或 y,那么程序就执行另一次重复。否则,程序结束。如果你对后面提示一直保持输入 Y 或 y,那么程序就一直保持运行——直到计算机崩溃为止!

程序列表 6.3 程序 FOR3.CPP 的源代码

```
1: // Program that demonstrates using the
2: // for loop to emulate an infinite loop.
3:
4: #include <iostream.h>
5: #include <ctype.h>
6:
7: main()
8: {
9:     char ch;
10:    double x, y;
11:
12:    // for loop with empty parts
13:    for (;;) {
14:        cout << "\nEnter a number : ";
15:        cin >> x;
16:        // process number if non-zero
17:        if (x != 0) {
18:            y = 1/ x;
19:            cout << "1/(" << x << ") = " << y << "\n";
20:            cout << "More calculations? (Y/N) ";
21:            cin >> ch;
22:            ch = toupper(ch);
23:            if (ch != 'Y')
24:                break;
25:        }
```

```
26:     else
27:         // display error message
28:         cout << "Error: cannot accept 0\n";
29:     }
30:     return 0;
31: }
```

下面是程序列表 6.3 中的程序输出部分:

输出:

```
Enter a number : 5
1/(5) = 0.2
More calculations? (Y/N) y
Enter a number : 12
1/(12) = 0.083333
More calculations? (Y/N) y
Enter a number : 16
1/(16) = 0.0625
More calculations? (Y/N) n
```

分析:

在程序列表 6.3 中,程序说明了字符型变量 `ch` 和两个双精度型变量 `x` 与 `y`。函数 `main` 在第 13 行使用了 `for` 循环作为开循环,该 `for` 循环消除了所有三个循环组成部分。第 14 行的输出语句提示你输入一个数。第 15 行的输入语句获取你的输入,并把它存放在变量 `x` 中。第 17 行的 `if-else` 语句判断变量 `x` 的值是否为非零。如果这个条件为真,程序就执行第 18—24 行的语句块;否则,程序执行第 28 行的 `else` 从句。这个语句显示出一条出错信息。

第 18 行的语句把变量 `x` 的值的倒数赋给变量 `y`。第 19 行的输出语句显示变量 `x` 和 `y` 的值。第 20 行的输出语句提示你是否还要进行计算,它要求输入一个 `Y/N`(用大写或小写都行)作为回答。第 21 行的语句把你输入的那个字符存入到变量 `c` 中。第 22 行的语句把你的输入转换成大写形式,它使用了函数 `toupper`(这个函数在头文件 `CTYPE.H` 中进行了原型说明)。第 23 行的 `if` 语句判断变量 `c` 中的字符是否是字母 `Y`。如果这个条件成立,那么程序执行第 24 行的 `break` 语句。该语句使程序退出开循环,而从第 30 行开始执行。

6.3 do-while 循环

在 C++ 中,do-while 循环是一种条件循环。所以,do-while 循环至少要重复执行一次。

新术语:条件循环(conditional loop)只要条件成立,就重复执行。这个条件是在循环的结尾进行测试。

语法

do-while 循环

do-while 循环的一般语法是

```
do {  
    sequence of statements  
} while (condition);
```

例子:

下面的循环显示了 2 到 10 的平方值:

```
int i = 2;  
do {  
    cout << i << " 2 = " << i * i << "\n";  
} while (++i < 11);
```

让我们来看看一个例子。程序列表 6.4 显示了程序 SOWHILE1.CPP 的源代码,它主要计算平方根值。该程序执行下列任务:

- ☐ 提示你输入一个数(如果你输入了一个负数,程序就再提示你输入一个数)。
- ☐ 计算并显示你输入的那个数的平方根值。
- ☐ 询问你是否希望输入另一个数(如果你输入了字母 Y 或 y,程序就重新从第一步开始执行;否则,程序结束)。

程序列表 6.4 程序 DOWHILE1.CPP 的源代码

```
1: // Program that demonstrates the do-while loop  
2:  
3: #include <iostream.h>  
4:  
5: const double TOLERANCE = 1.0e-7;  
6:  
7: double abs(double x)  
8: {  
9:     return (x >= 0) ? x : -x;  
10: }  
11:  
12: double sqrtroot(double x)  
13: {  
14:     double guess = x / 2;  
15:     do {  
16:         guess = (guess + x / guess) / 2;  
17:     } while (abs(guess * guess - x) > TOLERANCE);  
18:     return guess;  
19: }  
20:  
21: double getNumber()  
22: {  
23:     double x;  
24:     do {  
25:         cout << "Enter a number: ";  
26:         cin >> x;  
27:     } while (x < 0);  
28:     return x;  
29: }  
30:
```

```

31: main()
32: {
33:     char c;
34:     double x, y;
35:
36:     do {
37:         x = getNumber();
38:         y = sqrt(x);
39:         cout << "Sqrt(" << x << ") = " << y << "\n"
40:             << "Enter another number? (Y/N) ";
41:         cin >> c;
42:         cout << "\n";
43:     } while (c == 'Y' || c == 'y');
44:     return 0;
45: }

```

下面是程序列表 6.4 中的程序输出部分：

输出：

```

Enter a number: 25
Sqrt(25) = 5
Enter another number? (Y/N) y
Enter a number: 144
Sqrt(144) = 12
Enter another number? (Y/N) n

```

分析：

在程序列表 6.4 中，程序说明了全局常量 TOLERANCE 和函数 abs, sqrt 与 main。位于第 7 行的函数 abs 返回双精度型参数的绝对值。

第 12 行的函数 sqrt 返回参数 x 的平方根。该函数在第 14 行设置平方根的初始猜测值为 x/2。然后函数使用一个 do-while 循环，反复改进那个平方根的猜测值。在 while 从句中的条件判断当前猜测值与参数 x 之间的绝对差值是否大于可允许的误差（由常量 TOLERANCE 给定）。该循环只要条件为真，就重复执行。而函数在第 18 行返回平方根的猜测值。函数 sqrt 利用了 Newton 方法反复获取一个数的平方根。第 21 行的函数 getNumber 提示你输入一个数，并把你的输入存于局部变量 x 中。该函数也使用了一个 do-while 循环，确保你输入了一个非负数。第 27 行的 while 从句判断变量 x 的值是否是负数。只要这个条件为真，do-while 循环就重复执行。在第 28 行的返回语句产生出了 x 的值。

函数 main 位于第 31 行，它使用了一个 do-while 循环执行下面的任务：

- ☐ 提示你输入一个数，这通过调用函数 getNumber 实现（第 37 行的语句包含了该函数调用，并把结果赋给局部变量 x）。
- ☐ 通过调用函数 sqrt，计算 x 的平方根，并把结果赋给局部变量 y（包含这个函数调用的语句在第 38 行）。
- ☐ 显示变量 x 和 y 中的值。
- ☐ 询问你是否想要输入另一个数（第 41 行的输入语句获取你的单个字符 Y/N 输入，并把它存放在变量 c 中）。

位于第 43 行的 while 语句判断变量 c 是否含有 Y 或 y。只要这个条件成立，do-while 循环就重复执行。

在程序列表 6.4 中,程序说明了 do-while 循环的下列用途:

1. 重复计算。这在函数 sqrt 中的循环显示了这一方面。
2. 数据合法性。这在函数 getNumber 中的循环说明了这一方面。
3. 程序继续。函数 main 中的循环显示了这一方面。

6.4 while 循环

C++ 中的 while 循环是另一种条件循环,它只要条件成立,就重复执行。所以,如果测试条件初始时就为假,那么 while 循环就不重复执行。

语法

while 循环

while 循环的一般语法是

```
while (condition)
    statement; | { sequence of statements }
```

例子:

```
function power (double x, int n)
{
    double pwr = 1;
    while (n-- > 0)
        pwr *= x;
    return pwr;
}
```

现在,让我们来看看一个例子。程序列表 6.5 显示了程序 WHILE1.CPP 的源代码。这个程序执行与程序列表 6.1 中的程序 FOR1.CPP 同样的操作。两个程序作用的方式也相同,并且产生同样的结果。

程序列表 6.5 程序 WHILE1.CPP 的源代码

```
1: // Program that demonstrates the while loop
2:
3: #include <iostream.h>
4:
5: main()
6: {
7:     double sum = 0;
8:     double sumx = 0.0;
9:     int first, last, temp, i;
10:
11:     cout << "Enter the first integer : ";
12:     cin >> first;
13:     cout << "Enter the last integer : ";
14:     cin >> last;
15:     if (first > last) {
16:         temp = first;
17:         first = last;
18:         last = temp;
```

```

19:     }
20:     i = first;
21:     while (i <= last) {
22:         sum++;
23:         sumx += (double)i++;
24:     }
25:     cout << "Sum of integers from "
26:          << first << " to " << last << " = "
27:          << sumx << "\n";
28:     cout << "Average value = " << sumx / sum;
29:     return 0;
30: }

```

下面是程序列表 6.5 中的程序输出部分：

输出：

Enter the first integer : 1

Enter the last integer : 100

Sum of integers from 10 to 100 = 5050

Average value = 50.5

因为程序列表 6.5 和 6.1 中的程序相似，所以我集中说明第 20–24 行，两个程序的主要不同就存在于此。第 20 行的语句把变量 first 的值赋给变量 i。while 循环从第 21 行开始。该循环只要变量 i 的值小于或等于变量 last 的值，就重复执行。变量 i 起着循环控制变量的作用。第 22 行的语句对变量 sum 中的值进行增值。第 23 行的语句把变量 i 的值加到变量 sumx，并且让变量 i 自增 1。该语句通过对变量 i 运用后置增值运算符来执行后面的那个任务。

6.5 跳过循环重复

C++ 允许你跳到一个循环的末端，并通过使用 continue 语句重新执行下一次重复。这种编程特点使你的循环能够跳过，对于那些可能引起系统运行出错的特殊值的重复。

语法

continue 语句

使用 continue 语句的一般语法是

```

<loop-start clause> {
    // sequence #1 of statements
    if (skipCondition)
        continue;
    //sequence #2 of statements
} <loop-end clause>

```

例子(在一个 for 循环中)：

```

double x, y;
for (int i = -10; i < 11; i++) {
    x = i;

```



```

    if (i == 1)
        continue;
    y = 1/sqrt(x * x - 1);
    cout << "1/sqrt(" << (x * x - 1) << ") = " << y << "\n";
}

```

这种形式表明了，在 for 循环中，第一序列语句的计算就产生了一个条件，该条件用一个 if 语句来测试。如果条件成立，if 语句就引起 continue 语句的执行，跳过 for 循环中的第二序列语句。

现在，让我们来看看一个例子。程序列表 6.6 显示了 FOR4.CPP 的源代码。这个程序显示函数 $f(X) = \sqrt{X^2 - 9}$ 在 $-10 \sim 10$ 的整数值范围的函数值表。因为 $-2 \sim 2$ 之间的整数会产生复数结果，程序省略了这些结果，因此，这个表没有显示 $-2 \sim 2$ 之间的 $f(X)$ 的复数值。

程序列表 6.6 程序 FOR4.CPP 的源代码

```

1: // Program that demonstrates using the continue statement
2: // to skip iterations.
3:
4: #include <iostream.h>
5: #include <math.h>
6:
7:
8: double f(double x)
9: {
10:     return sqrt(x * x - 9);
11: }
12:
13: main()
14: {
15:     double x, y;
16:
17:     cout << "      X";
18:     cout << "      f(X)\n";
19:     cout << "                      \n\n";
20:     // for loop with empty parts
21:     for (int i = -10; i <= 10; i++) {
22:         if (i > -3 && i < 3)
23:             continue;
24:         x = (double)i;
25:         y = f(x);
26:         cout << "      ";
27:         cout.width(3);
28:         cout << x << "      ";
29:         cout.width(7);
30:         cout << y << "\n";
31:     }
32:     return 0;
33: }

```

下面是程序列表 6.6 中的程序输出部分：

输出:

X	f(X)
-10	9.539392
-9	8.485281
-8	7.41698
-7	6.324555
-6	5.196152
-5	4
-4	2.645751
-3	0
3	0
4	2.645751
5	4
6	5.196152
7	6.324555
8	7.41698
9	8.485281
10	9.539392

分析:

在程序列表 6.6 中,程序说明了函数 f 以代表数学函数 $f(X)$ 。函数 `main` 在第 15 行说明了双精度型变量 x 和 y 。第 17—19 行的输出语句显示了表头。第 21 行的 `for` 循环说明了它自身的控制变量,并且从 -10 到 10 进行重复,每次增值 1。循环内的第一个语句是位于第 22 行的 `if` 语句。这个语句判断变量 i 的值是否大于 -3,而小于 3。如果这个条件为真,程序就执行第 23 行的 `continue` 语句。所以,`if` 语句使 `for` 循环能够跳过出错的重复,而继续执行下一次重复。第 24 行的语句把变量 i 的值赋给变量 x 。第 25 行的语句调用函数 f ,并把参数 x 提供给它。然后,该语句把结果赋给变量 y 。第 25—30 行的输出语句显示变量 x 和 y 的值。这些语句使用函数 `width` 定义了简单的格式。

6.6 退出循环

C++ 支持 `break` 语句以退出循环。`break` 语句使程序结束当前的循环,而从循环后面开始执行。

语法

`break` 语句

在一个循环中,使用 `break` 语句的一般语法是

```

<start-loop clause> {
    // sequence #1 of statements
    if (exitLoopCondition)
        break;
    //sequence #2 of statements
} <end-loop clause>
//sequence #3 of statements

```

例子:

```

//calculate the factorial of n
factorial = 1;
for (int i = 1; ; i++) {
    if (i > n)
        break;
    factorial *= (double)i;
}

```

这种形式表明了，在 for 循环中，第一序列语句的计算产生了一个条件，该条件用 if 语句测试。如果条件成立，那么 if 语句就引起 break 语句的执行，退出循环。程序则重新从第三序列语句开始执行。

作为使用 break 语句的一个好例子，我推荐你重新查阅一遍程序列表 6.5 中的程序 FOR3.CPP。

6.7 嵌套循环

嵌套循环能够使你把所包含的重复性任务作为其他重复性任务的一部分。C++ 允许你嵌套任何种类的循环，并可调整到所需要的任何一层上。嵌套循环经常被用来处理数组(在第七章讨论)。

下面是一个使用嵌套循环的例子。程序列表 6.7 显示了程序 NESTFOR1.CPP 的源代码。该程序显示了 1 到 10 范围中的整数的平方根表。这个程序使用一个外层循环来重复这个范围的数字，而利用一个内层循环来重复计算平方根。

程序列表 6.7 程序 NESTFOR1.CPP 的源代码

```

1: // Program that demonstrates nested loops
2:
3: #include <stdio.h>
4:
5: const double TOLERANCE = 1.0e-7;
6: const int MIN_NUM = 1;
7: const int MAX_NUM = 10;
8:
9: double abs(double x)
10: {
11:     return (x >= 0) ? x : -x;
12: }
13:
14: main()
15: {

```

```

16: double x, sqrt;
17:
18: printf(" X      Sqrt(X)\n");
19: printf("      \n\n");
20: // outer loop
21: for (int i = MIN_NUM; i <= MAX_NUM; i++) {
22:     x = (double)i;
23:     sqrt = x / 2;
24:     // inner loop
25:     do {
26:         sqrt = (sqrt + x / sqrt) / 2;
27:     } while (abs(sqrt * sqrt - x) > TOLERANCE);
28:     printf("%4.1f      %8.6lf\n", x, sqrt);
29: }
30: return 0;
31: }

```

下面是程序列表 6.7 是的程序输出部分：

输出：

X	Sqrt(X)
1.0	1.000000
2.0	1.414214
3.0	1.732051
4.0	2.000000
5.0	2.236068
6.0	2.449490
7.0	2.645751
8.0	2.828427
9.0	3.000000
10.0	3.162278

分析：

在程序列表 6.7 中，程序包括了头文件 `STDIO.H`，为了使用具有强大的格式化能力的 `printf` 输出函数。第 5—7 行定义了常量 `TOLERANCE`，`MIN_NUM` 和 `MAX_NUM`，分别用来表示平方根值的误差，输出表格中的第一个数，及输出表格中的最后数字。该程序还定义了函数 `abs`，它返回一个双精度型数字绝对值。

函数 `main` 说明了双精度型变量 `x` 和 `sqrt`。第 18, 19 行的输出语句显示了表头。第 21 行包含了外层循环——一个 `for` 循环。这个循环说明了它的控制变量 `i`，并从 `MIN_NUM` 到 `MAX_NUM` 以每次变量 `i` 自增 1 的步长重复。第 22 行把 `i` 的强制类型转换值存入变量 `x` 中。第 23 行语句获取平方根的初始猜测值，并把它存入变量 `sqrt` 中。第 25 行包含了内层循环——一个 `do-while` 循环，它反复改进平方根的猜测值。第 26 行语句改进平方根的猜测值。第 27 行的 `while` 从句判断那个改进的猜测值是否足够了。第 28 行的输出语句显示变量 `x` 和 `sqrt` 的格式化的数值。

6.8 小结

今天的课介绍了 C++ 循环和与循环相关的主题。你已经学习了下列内容：

- C++ 中的 for 循环具有下面的一般语法：

```
for (<initialization of loop control variables>;  
    <loop continuation test>;  
    <increment/decrement of loop control variables>)
```

for 循环包含了三个组成部分：循环初始化，循环继续条件，以及循环变量的增值/减值。

- 条件循环 do-while 循环具有下面的一般语法：

```
do {  
    sequence of statements  
} while (condition);
```

do-while 循环至少重复一次。

- 条件循环 while 具有下面的一般语法：

```
while (condition)  
    statement; | {sequence of statements}
```

如果 while 循环的测试条件初始时为假，那么它不可以重复。

- continue 语句能够使你直接跳到循环的结尾，并重新开始执行下一次重复。continue 语句的优点是，它不需使用标号来指示跳动的位置。

- 开循环是不带有循环控制部分的 for 循环。break 语句能够使你退出当前的循环，而重新从循环后面的第一个语句开始执行。exit 函数(在 STDLEB.H 中说明)使你能够通过终止 C++ 程序来紧急退出循环。

- 嵌套循环能够使你把所包含的重复性任务作为其他重复性任务的一部分。C++ 允许你嵌套任何种类的循环，并可调整到你所需要的任何一层。

6.9 问与答

问：while 循环如何模拟 for 循环？

答：下面是一个简单的例子：

```
int i = 1;  
for (int i = 1; i <= 10; i += 2) {  
    cout << i << "\n";  
}  
while (i <= 10) {  
    cout << i << "\n";  
    i += 2;  
}
```

while 循环需要一个先导语句，来初始化循环控制变量。同时，还要注意，while 循环在它内部使用了一条语句来改变循环控制变量的值。

问：while 循环如何模拟 do-while 循环呢？

答：下面是一个简单的例子：

```

i = 1;
do {
    cout << i << "\n";
    i += 2;
} while (i <= 10);

i = 1;
while (i <= 10) {
    cout << i << "\n";
    i += 2;
}

```

这两个循环在它们的 while 从句中都有同样的条件。

问: 开 for 循环如何才能仿效 while 和 do-while 循环呢?

答: 开 for 循环能够通过把跳出循环的 if 语句放置在循环开始或结尾的附近, 来仿效其他的 C++ 循环。下面是开 for 循环如何仿效一个 while 循环的例子

```

i = 1;
while (i <= 10); {
    cout << i << "\n";
    i += 2;
}

i = 1;
for (;) {
    if (i > 10) break;
    cout << i << "\n";
    i += 2;
}

```

注意, 开 for 循环使用了一个跳出循环的 if 语句作为循环内的第一条语句。由 if 语句测试的条件是 while 循环条件的逻辑反。下面是一个显示仿效 do-while 循环的简单例子:

```

i = 1;
do {
    cout << i << "\n";
    i += 2;
} while (i <= 10);

i = 1;
for (;) {
    cout << i << "\n";
    i += 2;
    if (i > 10) break;
}

```

在该例子中, 开 for 循环使用了一个跳出循环的 if 语句, 它正好放在循环结束之前。if 语句测试的条件也与 do-while 循环条件相反。

问: 在嵌套的 for 循环中, 我可以使用外层循环的控制变量作为内层循环的取值部分吗?

答: 可以。C++ 不反对这样使用。下面是一个简单的例子

```

for (int i = 1; i <= 100; i += 5)
    for (int j = i; j <= 100; j++)
        cout << i * j << "\n";

```

问: C++ 严格限制不同类型的循环嵌套吗?

答: 不。你可以在 C++ 程序中嵌套任意组合的循环。

6.10 专题讨论

专题讨论提供了测验题, 帮助你加深对所学的资料的理解, 同时, 它还提供了练习, 锻炼你使用所学过的知识的能力。因此, 在你继续下一课的学习之前, 一定要尽力去理解测验和练习的答案。这些答案在附录 B 中提供。

6.10.1 测验

1. 下列循环有什么错误?

```
i = 1;
while (i < 10); {
    j = i * i - 1;
    k = 2 * j - i;
    cout << "i = " << i << "\n";
    cout << "j = " << j << "\n";
    cout << "k = " << k << "\n";
}
```

2. 下面的 for 循环的输出是什么?

```
for (int i = 5; i < 10; i -- 2)
    cout << i - 2 << "\n";
```

3. 下面的 for 循环输出什么?

```
for (int i = 5; i < 10;)
    cout << i - 2 << "\n";
```

4. 下面的代码有什么错误?

```
for (int i = 1; i <= 10; i++)
    for (i = 8; i <= 12; i++)
        cout << i << "\n";
```

5. 下面的嵌套循环错在哪里?

```
for (int i = 1; i <= 10; i++)
    cout << i * i << "\n";
    for (int i = 1; i <= 10; i++)
        cout << i * i * i << "\n";
}
```

6. 下面的循环错在哪里?

```
i = 1;
while (1 > 10); {
    cout << i << "\n";
    i++;
}
```

7. 一个数的阶乘等于从 1 到那个数的整数连乘。下面的一般性等式定义了阶乘形式 (它使用了符号!):

$$n! = 1 * 2 * 3 * \dots * n$$

下面是一个计算数的阶乘的 C++ 程序。问题是对于你输入的任何正值, 程序都显示阶乘为 0 值。这个程序错在哪里?

```
int n;
double factorial;
cout << "Enter positive intese n: ";
```

```
for (int i = 1; i <= n; i++)  
    factorial *= i;  
cout << n << " ! = " << factorial;
```

6.10.2 练习

1. 编写程序 FOR5.CPP, 要求使用一个 for 循环来获取并显示 11 到 121 范围内的奇数总和。
2. 编写程序 WHILE2.CPP, 要求使用一个 while 循环来获取并显示 11 到 121 范围内的奇数平方和。
3. 编写程序 DOWHILE2.CPP, 要求使用一个 do-while 循环来获取并显示 11 到 121 范围内的奇数平方和。

第七章 数组(第七天)

数组属于最普通的数据结构。它们能够为程序存储数据,用于以后的继续处理。大多数编程语言都支持静态数组,而且许多语言还支持动态数组。

新术语:数组就是一组变量。

今天,你将学习下面的有关静态数组的主题:

- ☐ 一维数组的说明
- ☐ 一维数组的使用
- ☐ 一维数组的初始化
- ☐ 作为函数参数的一维数组的说明
- ☐ 数组的排序
- ☐ 搜寻数组
- ☐ 多维数组的说明
- ☐ 多维数组的使用
- ☐ 多维数组的初始化
- ☐ 作为函数参数的多维数组的说明

7.1 一维数组的说明

一维数组是一种最简单的数组。在一维数组中,每个变量可以使用单个下标来访问。

新术语:一维数组(single-dimensional array)是一组享有相同名称(也就是数组名)的变量。

语法

一维数组

说明一维数组的一般语法是:

```
type arrayName[numberOfElements];
```

C++要求你在说明一维数组中遵守下列规则:

1. 一个C++数组的最小边界设置为0。C++不允许你超越或改变这个最小边界。
 2. 说明一个C++数组必须指定成员数。要记住,成员的数目等于最大边界加1。
- 对于这种一般的形式,下标的有效范围可以从0扩展到 `numberOfElements-1`。

例子:

```
int intArray[10];
char name[31];
double x[100];
```

7.2 一维数组的使用

使用一维数组包含了说明它的名称和访问它的成员的有效下标。根据一个数组元素的引用所发生的位置,它可以存入或调用一个值。这些需要记住的规则是:

1. 在访问一个调用数据的数组元素之间,你可以给那个数组元素赋值。否则,你将得到无用的数据。
2. 一定要使用一个有效的下标。

DO	DON'T
DO 对访问数组的下标,一定要进行适当的检查。	
DON'T 不要假定下标总是有效的。	

让我们来看看一个例子。程序列表 7.1 显示了程序 ARRAY1.CPP 的源代码。这个程序使用了一个 30 个元素的数组来计算一个数组中的数据的均值。该程序执行下列任务:

- ☐ 提示你输入实际数据点的数目(这个值必须存在于由提示信息指定的有效数字范围内)。
- ☐ 提示你为数组元素输入数据。
- ☐ 计算数组中的数据的均值。
- ☐ 显示均值。

程序列表 7.1 程序 ARRAY1.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of one-dimension
3:  arrays. The average value of the array is calculated.
4:  */
5:
6: #include <iostream.h>
7:
8: const int MAX = 30;
9:
10: main()
11: {
12:
13:     double x[MAX];
14:     double sum, sumx = 0.0, mean;
15:     int i, n;
16:
17:     do { // obtain number of data points
18:         cout << "Enter number of data points [2 to "
19:             << MAX << " ] : ";
20:         cin >> n;
21:         cout << "\n";
22:     } while (n < 2 || n > MAX);
```

```

23:
24:     // prompt user for data
25:     for (i = 0; i < n; i++) {
26:         cout << "X[" << i << "] : ";
27:         cin >> x[i];
28:     }
29:
30:     // initialize summations
31:     sum = n;
32:
33:     // calculate sum of observations
34:     for (i = 0; i < n; i++)
35:         sumx += x[i];
36:
37:     mean = sumx / sum; // calculate the mean value
38:     cout << "\nMean = " << mean << "\n\n";
39:     return 0;
40: }

```

下面是程序列表 7.1 中的程序输出部分：

输出：

Enter number of data points [2 to 30] : 5

X[0] : 12.5

X[1] : 45.7

X[2] : 25.6

X[3] : 14.1

X[4] : 68.4

Mean = 33.26

分析：

在程序列表 7.1 中，程序说明了全局常量 MAX，作为中在程序所使用的数组的大小。函数 main 在第 13 行说明了双精度型数组 x，它含有 MAX 个元素。该函数还在第 14、15 行说明了其他非数组的变量。

位于第 17—22 行的 do-while 循环获取你想存入数组 x 中的数据的数据数目。第 18、19 行的输出语句提示你输入数据点的数目。该输出语句指出了有效数字的范围是 2 到 MAX。第 20 行语句获取你的输入，并把它存入变量 n 中。while 从句用来证实你的输入。该从句判断变量 n 的值是否小于 2 或大于 MAX。如果这个条件为真，do-while 循环就再重复一次来获取一个正确的输入。

第 25—28 行的 for 循环语句提示你输入数据。该循环使用控制变量 i，从 0 到 n-1，以每次增值 1 的步长进行重复。第 26 行的输出语句提示你输入所指示的数组元素的值。第 27 行的输入语句获取你的输入，并把它存入元素 x[i] 中。

第 31 行语句把变量 n 中的整数赋给双精度型变量 sum。第 34、35 行的 for 循环把数组 x 中的值加到变量 sumx。该循环使用控制变量 i，从 0 到 n-1，以每次自增 1 的步长进行重复。第 35 行语句使用自增赋值运算符，把元素 x[i] 中的值加到变量 sumx。第 37 行语句计算平均值，并把它存入变量 mean 中。第 38 行的输出语句显示平均值。

注意：在程序列表 7.1 中的程序显示了如何使用 for 循环来处理数组元素。该循环继

续条件的测试使用了<运算符和一个大于最后那个有效下标的数值。你也可以使用<=运算符,后面跟有最后下标。例如,我可以把数据输入循环写成:

```
24: //prompt user for data
25: for (i = 0; i <= (n - 1); i++) {
26:     cout << "X[" << i << "] : ";
27:     cin >> x[i];
28: }
```

但是,这种形式并不流行,因为它要求一个辅助运算符,而条件*i*<*n*不需要。

DO	DON'T
DO 要写出循环继续表达式,以便循环可以使用最小数量的运算符。这种方法可减少代码的大小,并加快循环的执行。	DON'T 在循环继续条件中不要使用<=运算符,除非该运算符能够帮助你编写最小运算的表达式。

7.3 一维数组的初始化

C++允许你初始化数组,并且初始化具有灵活性。你必须把初始化值表用一对大括号({})括起来。该表用逗号分隔,并可以在多行上继续。如果初始化表中的数据项要少于数组元素,那么编译器通过赋值0来平衡数组元素。相反,如果初始化值表所含有的数据项多于数组元素,编译器就标出一个编译错误。

程序列表 7.2 中的程序修改了上一个程序以提供内部数据。因此,删除了那些提示你输入数据的数目和数据本身的步骤。这个程序简单地显示各数组元素(从初始化表中获得)和数据的均值。尽管该程序不与用户进行交互作用,但是它提供了一种在源代码中存储数据的形式。你可以在重新计算一个新的均值以前,定期编辑程序以增加、编辑和删除数据。

程序列表 7.2 程序 ARRAY2.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of single-dimensional
3:  arrays. The average value of the array is calculated.
4:  The array has its values preassigned internally.
5:  */
6:
7: #include <iostream.h>
8:
9: const int MAX = 10;
10:
11: main()
12: {
13:
```

```

14:    double x[MAX] = { 12.2, 45.4, 67.2, 12.2, 34.6, 87.4,
15:                      83.6, 12.3, 14.8, 55.5 };
16:    double sum = MAX, sumx = 0.0, mean;
17:    int n = MAX;
18:
19:    // calculate sum of observations
20:    cout << "Array is:\n";
21:    for (int i = 0; i < n; i++) {
22:        sumx += x[i];
23:        cout << "x[" << i << "] = " << x[i] << "\n";
24:    }
25:
26:    mean = sumx / sum; // calculate the mean value
27:    cout << "\nMean = " << mean << "\n\n";
28:    return 0;
29: }

```

下面是程序列表 7.2 中的程序输出部分：

输出：

```

Array is:
X[0] : 12.2
X[1] : 45.4
X[2] : 67.2
X[3] : 12.2
X[4] : 34.6
X[5] : 87.4
X[6] : 83.6
X[7] : 12.3
X[8] : 14.8
X[9] : 55.5
Mean = 42.52

```

分析：

让我们来集中讨论一下程序列表 7.2 中的数组 `x` 的初始化问题。第 14 行包含了数组 `x` 的说明和初始化。初始化表一直延续到了第 15 行，它用一对大括号括起来，并且由逗号把各个值分隔开。第 16 行语句说明变量 `sum` 和 `sumx`，并且分别把这两个变量初始化为 `MAX` 和 0。第 17 行语句说明了整型变量 `n`，并且用数值 `MAX` 初始化它。程序的其余部分则与程序列表 7.1 中的程序部分相似。

如果你在某种程度上对你必须数出初始化值的准确数目而感到沮丧，那么我有一些好消息要告诉你：C++ 允许你通过使用相应的初始化表中的数据项数来自动确定一个数组的大小。因此，你不需要在数组的方括号中放置一个数字，而让编译器为你做这项工作。

DO	DON'T
DO 如果初始化的数组以后需要扩展,那么在初始化表中可以包括空值。	
DON'T 不要依靠对初始化表中的数据项进行计数来提供数组元素的数目。	

程序列表 7.3 显示了程序 ARRAY3.CPP 的源代码。这个新的程序使用了自动确定数组大小的特性。

程序列表 7.3 程序 ARRAY3.CPP 的源代码

```

1: /*
2:  C++ program that demonstrates the use of single-dimensional
3:  arrays. The average value of the array is calculated.
4:  The array has its values preassigned internally.
5:  */
6:
7: #include <iostream.h>
8:
9: main()
10: {
11:
12:     double x[] = { 12.2, 45.4, 67.2, 12.2, 34.6, 87.4,
13:                   83.6, 12.3, 14.8, 55.5 };
14:     double sum, sumx = 0.0, mean;
15:     int n;
16:
17:     n = sizeof(x) / sizeof(x[0]);
18:     sum = n;
19:
20:     // calculate sum of observations
21:     cout << "Array is:\n";
22:     for (int i = 0; i < n; i++) {
23:         sumx += x[i];
24:         cout << "x[" << i << "] = " << x[i] << "\n";
25:     }
26:
27:     mean = sumx / sum; // calculate the mean value
28:     cout << "\nNumber of data points = " << n << "\n"
29:          << "Mean = " << mean << "\n";
30:     return 0;
31: }

```

下面是程序列表 7.3 中的程序输出部分:

输出:

Array is:

X[0]: 12.2

X[1]: 45.4

X[2]: 67.2

X[3]: 12.2

X[4]: 34.6

X[5]: 87.4

```
X[6] : 83.6
X[7] : 12.3
X[8] : 14.8
X[9] : 55.5
Number of data points = 10
Mean = 42.52
```

分析:

注意,在程序列表 7.3 中,程序没有说明常量 MAX,而它在前面的程序中出现了(见程序列表 7.2)。该程序如何决定数组元素的数目呢?第 17 行显示了程序通过把数组 `x` 的大小(通过使用 `sizeof(x)` 获得)除以第一个元素的大小(通过使用 `sizeof(x[0])` 获取)来计算数组 `x` 的元素数目。你也可以使用这种方法来获取任何类型的任一数组的大小。

7.4 函数中的数组参数

C++ 允许你用数组作为函数的参数。事实上,C++ 支持你对数组参数的大小进行特殊或一般的说明。如果你想要一个数组参数接受固定大小的数组,那么你可以在参数说明中指定数组的大小。相反,如果你想要数组参数接受同样类型但不同大小的数组,那么你可以使用空括号的数组参数。

语法

固定大小的数组参数

说明固定大小的数组参数的一般语法是

```
type parameterName[arraySize]
```

例子:

```
int minArray(int arr[100], int n);
void sort(unsigned dayNum[7]);
```

语法

未固定大小的数组参数

说明未固定大小的数组参数的一般语法是

```
type parameterName[ ]
```

例子:

```
int minArray(int arr[ ], int n);
void sort(unsigned dayNum[ ]);
```

DO	DON'T
<p>DO 在函数中,最好使用未定大小的数组参数。</p> <p>DON'T 在一般目的的函数中,不要忘记检查一个未定大小的数组参数的上边界。</p>	

让我们来看看一个简单的例子。程序列表 7.4 显示了程序 ARRAY4.CPP 的源代码。
该程序执行下列任务：

- ☐ 提示你输入数据点的数目,这些数据点的范围在 2 到 10 之间。
- ☐ 提示你为数组输入整数值。
- ☐ 显示数组中的最小值。
- ☐ 显示数组中的最大值。

程序列表 7.4 程序 ARRAY3.CPP 的源代码

```

1: // C++ program that passes arrays as arguments of functions
2:
3: #include <iostream.h>
4:
5: const int MAX = 10;
6:
7: main()
8: {
9:     int arr[MAX];
10:    int n;
11:
12:    // declare prototypes of functions
13:    int getMin(int a[MAX], int size);
14:    int getMax(int a[], int size);
15:
16:    do { // obtain number of data points
17:        cout << "Enter number of data points {2 to "
18:            << MAX << "} : ";
19:        cin >> n;
20:        cout << "\n";
21:    } while (n < 2 || n > MAX);
22:
23:    // prompt user for data
24:    for (int i = 0; i < n; i++) {
25:        cout << "arr[" << i << "] : ";
26:        cin >> arr[i];
27:    }
28:
29:    cout << "Smallest value in array is "
30:        << getMin(arr, n) << "\n"
31:        << "Biggest value in array is "
32:        << getMax(arr, n) << "\n";
33:    return 0;
34: }
35:
36:
37: int getMin(int a[MAX], int size)
38: {
39:     int small = a[0];
40:     // search for the smallest value in the
41:     // remaining array elements

```



```

42:   for (int i = 1; i < size; i++)
43:       if (small > a[i])
44:           small = a[i];
45:   return small;
46: }
47:
48: int getMax(int a[], int size)
49: {
50:     int big = a[0];
51:     // search for the biggest value in the
52:     // remaining array elements
53:     for (int i = 1; i < size; i++)
54:         if (big < a[i])
55:             big = a[i];
56:     return big;
57: }

```

程序列表 7.4 中程序执行结果如下:

输出结果:

arr[0]:

arr[1]:69

arr[2]:47

arr[3]:85

arr[4]:14

Smallest value in array is 14

Diges valne is array is 85

分析:

程序列表 7.4 中程序在第 5 行定义了全程常量 MAX 以规定数组的大小。主函数 main 在第 9 行。定义了 int 型数组 arr。第 10 行定义了 int 型变量 n。第 13 和 14 行定义了函数 getMin 和 getMax 的函数原型。函数 getMin 以一个固定大小的数组作为函数参数。而函 getMax 是以一个开式数组作为函数参数。为了更好地说明,我们采用了这两种数组形式。

程序第 16 行至 21 行,用到 do-while 循环。便于你把数据存储在数组 arr 中。第 17 和 18 行的输出语句提示信息,请你输入数据的数目。这个输出说明了有效的输入值范围,介于 2 到 MAX 之间。第 19 行语句得到你的输入,并有存储在变量 n 中。while 从句判断你的输入是否有效。这个从句决定变量 n 的值是否小于 2 或大于 MAX。如果以上情况发生,则 do-while 重新执行。

第 24 至 27 行的 for 循环请你输入数据。该循环用控制变量 i 控制,i 从 0 变化到 n-1,每次增量为 1。第 25 行的输语句请你输入指定数组的元素值。第 26 行把你的输入存储在元素 arr[i]中

第 29 至 32 行的输出语句显示数组 arr 中最小和最大的整数值。该语句调用函数 getMin 和 getMax 变量为 arr 和 n。

程序第 37 行至 46 行定义了 getMin 函数,该函数有两个参数:int 型的变量 size 和 int 型固定大小数组 a。函数还定义了局部变量 small,并把它初始化为 a[0]。函数查找数组 a 中的最小值。在第 42 行用到 for 循环。循环控制变量为 i,从 1 变化到 size-1,每次增量为 1。循环体中有一个 if 语句,如元素 a[i]小于变量 small,则把 a[i]赋予变量

small。函数返回变量 small 中值。函数 getMin 只能接收具有 MAX 个元素的 int 型数组。

程序第 48 至 57 行定义了函数 getMax。这个函数与函数 getMin 相似,有两个参数:一个是 int 型开式数组参数 a,另一个是 int 型变量 size。此外,函数中还定义了局部变量 big,并把它初始化为 a[0]。该函数查找数组参数 a 中的最大值。在第 53 行用到一个 for 循环。循环控制变量为 i,由 0 变化到 size-1,每次增量为 1。在循环体中,有一个 if 语句,在变量 big 的值小于元素 a[i]时,把元素 a[i]赋值给变量 big。该函数返回变量 big 中的值。函数 getMax 可接受任何大小的 int 型数组。

7.5 数组的排序

数组的排序和查找是最常用的数组的非数值操作。典型的数组排序如:按由小到大的顺序排列数组元素。这个过程用部分或全部的数组元素值来决定数组中各元素的先后顺序。在排序后的数组中查找数据比在杂乱无序的数组中查找数据要更为容易。

计算机科学家们花费了大量的时间研究、探讨数组排序的方法,并对它们进行讨论和比较。这些内容已超出本书研究的范围,在此,我只谈几种常用的数组排序法。如:Quick-Sort,Shell-Metzner 排序,堆排序和最新的 Comb 排序。通常认为 QuickSort 是一种最快速的排序方法,但它需要一些操作的额外开销。Shell-Metzner 和 Comb 排法不需要类似的操作额外开销。这节中的示例是最新的 Comb 排序法,这是一种比 Shell-Metzner 排序法更加有效的方法。

Comb 排序法的执行步骤如下,首先给定一个具有 N 个元素的数组 A:

1. offset 初始化为 N,用于比较元素值大小。
2. 把 offset 值置为 $8 * offset / 11$ 和 1 中较大的一值。
3. InOrder 标志设置为 true。
4. 使用循环控制变量 i, i 从 0 到 N-Offset 进行下列循环:
 - ☐ 把 I+offset 赋值给 j;
 - ☐ 如果 A[i]大于 A[j],则交换 A[i]与 A[j]中的元素,并且把 InOrder 标志为 false。
5. 如果 offset 不等于 1 且 InOrder 标志为 false 时,重新执行第 2 步。

下面让我们看看整型数组的排序程序。程序列表 7.5 显示了 ARRAY.CPP 的源程序。这个程序执行以下几个功能:

- ☐ 输入数组大小值
- ☐ 请你输入整形的数组元素值
- ☐ 显示未经排序的数组元素
- ☐ 显示已经排序的数组元素

程序列表 7.5: 程序 ARRAY5.CPP 的源代码

```

1: // C++ program that sorts arrays using the Comb sort method
2:
3: #include <iostream.h>
4:
5: const int MAX = 10;
6: const int TRUE = 1;
7: const int FALSE = 0;
8:
9: int obtainNumData()
10: {
11:     int m;
12:     do { // obtain number of data points
13:         cout << "Enter number of data points [2 to "
14:             << MAX << "] : ";
15:         cin >> m;
16:         cout << "\n";
17:     } while (m < 2 || m > MAX);
18:     return m;
19: }
20:
21: void inputArray(int intArr[], int n)
22: {
23:     // prompt user for data
24:     for (int i = 0; i < n; i++) {
25:         cout << "arr[" << i << "] : ";
26:         cin >> intArr[i];
27:     }
28: }
29:
30: void showArray(int intArr[], int n)
31: {
32:     for (int i = 0; i < n; i++) {
33:         cout.width(5);
34:         cout << intArr[i] << " ";
35:     }
36:     cout << "\n";
37: }
38:
39: void sortArray(int intArr[], int n)
40: {
41:     int offset, temp, inOrder;
42:
43:     offset = n;
44:     do {
45:         offset = (8 * offset) / 11;
46:         offset = (offset == 0) ? 1 : offset;
47:         inOrder = TRUE;
48:         for (int i = 0, j = offset; i < (n - offset); i++, j++) {
49:             if (intArr[i] > intArr[j]) {
50:                 inOrder = FALSE;
51:                 temp = intArr[i];
52:                 intArr[i] = intArr[j];
53:                 intArr[j] = temp;
54:             }
55:         }
56:     } while (!(offset = 1 && inOrder == TRUE));
57: }
58:
59: main()
60: {
61:     int arr[MAX];
62:     int n;
63:

```

```

64:  n = obtainNumData();
65:  inputArray(arr, n);
66:  cout << "Unordered array is:\n";
67:  showArray(arr, n);
68:  sortArray(arr, n);
69:  cout << "Sorted array is:\n";
70:  showArray(arr, n);
71:  return 0;
72: }

```

下面是程序列表 7.5 中程序的执行示例:

输出:

```

Enter number of data points [2 to 10]:10
arr[0]:55
arr[1]:68
arr[2]:74
arr[3]:15
arr[4]:28
arr[5]:23
arr[6]:69
arr[7]:95
arr[8]:22
arr[9]:33
unordored array is:
55 68 74 15 28 23 69 095 22 23
sorted array is:
15 22 23 28 33 35 68 69 74 95

```

分析:

程序列表 7.5 中的程序在第 5 行至第 7 行定义了变量 MAX, TRUE 和 FALSE。变量 MAX 表示程序中数组的大小。变量 TRVE 和 FALSE 定义为布尔(Boolean)值。在这个程序中定义了 ObtainNumData, iwputArray, ShiwArray, SortArray 等函数,以及主函数 main。

无参函数 obtainNunData 在程序第 9 行至第 19 行给出定义,其函数功能为输入数组的大小值。在程序的第 13 和 14 行的输出提示语句中限定了你应输入值的有效范围。第 15 行把你输入的数值赋与局部变量 m。这个函数使用 do-while 循环确保它能返回一个有效的数值。只要变量 m 的值小于 2 或大于 MAX,这个循环就会执行。这个函数返回变量 m 的值。

函数 inputArray 在程序第 21 行至第 28 行给出定义,其函数功能对该测试数组赋值。这个函数具有两个参数,开型的数组参数 intArr,把输入的元素值传送给调用该函数的数组。参数 n 规定了可从赋值给参数 intArr 的元素个数。这个函数使用了一个 for 循环,i 由 0 到 n-1,增量为 1 进行循环。每次循环都需要你输入一个数值,并把这个

数值存入 intArr 数组的一个元素中。

注意:函数 inputArray 说明了 C++ 函数是如何处理形参数组的。因为这些形参数组好像对函数作用范围之外的实参数组中的元素仍有影响,所以认为它们与实参数组的元素有关。但事实上,C++ 编译程序在处理形参数组时,是把实参数组的起始地址传递给函数的形参数组。这样两个数组就共占同一段内存单元,C++ 函数就能在函数范围之外改变数组中的元素值。

函数 showArray 在程序第 30 行至第 37 行给出定义,其函数功能为显示一数组中的有义的数据。这个函数有两个参数。开型数组形参 intArr 通过调用该函数传递数组元素以进行显示。形参 n 规定 intArr 中的数组元素个数(记住:并不是所有的数组元素都是用来存储你的据)。该函数用到一个 for 循环,i 由 0 到 n-1,每次增量为 1 进行循环。每次循环显示一个数组元素值。所有的数组元素值显示在同一行上。

函数 sortArray 在程序第 39 行至第 57 行给出定义。其函数功能为利用 Comb 排序法对数组元素进行排序。该函数有两个参数。开数组形参 intArr 通过该函数传递数组元素值进行排序。形参 n 规定进行排序的数组元素个数。函数 sortArray 中的语句实现了先前提到的 Comb 排序法。

注意:函数 sortArray 说明了形参数组是如何传递数据的。函数 sortArray 接收一个无序数组,对它排序,然后把排序后的数组传递给实参数组。编译程序通过把实参数组的地址传递给形参数组,就使函数能达到上述功能。这样,函数就不需要明确地返回数组,因为它本身就是在对同一组数组元素进行处理。

主函数 main 通过调用前面提到的各种函数,可以完成不同的程序功能。该函数在程序第 61 和 62 行定义了数组 arr 和简单变量 n。第 64 行的语句调用了函数 obtainNumData,获得你想存在数组中的数据个数。该语句把函数返回的结果赋于变量 n。第 65 行的语句调用函数 inputArray,请你输入数据。该函数通过实参 arr 和 n 进行调用。第 66 行的输出语句显示信息,说明程序将要显示无序数组元素值。第 67 行的语句调用函数 showArray,调用实参为 arr 和 n。这个函数把数组 Arr 中的元素值显示在同一行。第 68 行的语句调用函数 sortArray 对数组 arr 中的前 n 个元素进行排序。第 69 行的输出语句显示信息:程序将显示排序的数组元素。第 70 行的语句调用函数 showArray,调用实参为 arr 和 n,该函数把排序后的数组元素显示在同一行中。

7.6 数组的查找

数组的查找是另一种重要的非数值操作。因为数组可能是有序的,也可能无序的,所以对于每一种都有一种通用的查找方法。用于无序数组的最简单的查找方法是线性查找法。用于有序数组的最简单的查找方法是通用的对分查找法。用于无序数组的查找方法同样也可用于有序数组的查找。然而,这样就没有利用数组的有序特点。

新术语:线性查找法是顺序地检查数组元素,寻找一个与查找值相符的元素。如果查找的数值不在该数组中,线性查找法就要检查整个数组所有元素。

新术语:对分查找法利用了数组中的顺序,这种方法通过使用逐渐收缩的区间来查找一个匹配值。初始的查找区间包括所有有意义的数据的数组元素。对分查找法把区间的中项元素与查找值相比较。如果二者匹配,则停止查找。否则对分查找法决定哪一个子区间作为下一个查找区间。所以,每个查找区间是前一个查找区间大小的一半。如果查找值在所检验的数组中没有与它相匹配的元素,则对分查找法所进行的检验比线性查找法少得多。对于已分类数组而言,对分查找法是最有效的通用查找法。

DO	DON'T
DO 当不清楚数组是否已分类时,用无序数组查找法。	
DON'T 不要对无序数组使用已分类数组查找法。这样做的结果不可靠。	

现在让我们来看一个使一个整数数组分类的程序。程列表 7.6 显示的是程序 ARRAY6.CPP 的源代码。这个程序是通过把函数和操作添加到程序 ARRAY5.CPP 而建立的。该程序执行下列任务:

- ☐ 提示用户一个查找值。
- ☐ 显示查找结果(如果程序查找到了一个匹配元素,则显示该元素的下标;否则程序告诉用户未找到与查找值匹配的元素)。
- ☐ 重新回到步骤 4。
- ☐ 显示已分类数组的元素。
- ☐ 询问用户是否要在无序数组中查找数据(如果键入除 Y 或 y 以外的任意字符,程序结束)。
- ☐ 提示用户一个查找值。
- ☐ 显示查找结果(如果程序找到一个相匹配的元素,就显示该元素的下标。否则,程序告诉用户未找到与查找值相匹配的元素)。
- ☐ 重新回到步骤 9。

程序列表 7.6 程序 ARRAY6.CPP 的源代码

```

1: // C++ program that searches arrays using the linear
2: // and binary searches methods
3:
4: #include <iostream.h>
5:
6: const int MAX = 10;
7: const int TRUE = 1;
8: const int FALSE = 0;
9: const int NOT_FOUND = -1;
10:
11: int obtainNumData()
12: {
13:     int m;
14:     do { // obtain number of data points
15:         cout << "Enter number of data points ]2 to "
16:             << MAX << "] : ";
17:         cin >> m;
18:         cout << "\n";
19:     } while (m < 2 || m > MAX);
20:     return m;
21: }
22:
23: void inputArray(int intArr[], int n)
24: {
25:     // prompt user for data
26:     for (int i = 0; i < n; i++) {
27:         cout << "arr[" << i << "] : ";
28:         cin >> intArr[i];
29:     }
30: }
31:
32: void showArray(int intArr[], int n)
33: {
34:     for (int i = 0; i < n; i++) {
35:         cout.width(5);
36:         cout << intArr[i] << " ";
37:     }
38:     cout << "\n";
39: }
40:
41: void sortArray(int intArr[], int n)
42: // sort the first n elements of array intArr
43: // using the Comb sort method
44: {
45:     int offset, temp, inOrder;
46:
47:     offset = n;
48:     do {
49:         offset = (8 * offset) / 11;
50:         offset = (offset == 0) ? 1 : offset;
51:         inOrder = TRUE;
52:         for (int i = 0, j = offset; i < (n - offset); i++, j++) {
53:             if (intArr[i] > intArr[j]) {
54:                 inOrder = FALSE;
55:                 temp = intArr[i];
56:                 intArr[i] = intArr[j];
57:                 intArr[j] = temp;
58:             }
59:         }
60:     } while (! (offset = 1 && inOrder == TRUE));
61: }
62:

```

```

63: int linearSearch(int searchVal, int intArr[], int n)
64: // perform linear search to locate the first
65: // element in array intArr that matches the value
66: // of searchVal
67: {
68:     int notFound = TRUE;
69:     int i = 0;
70:     // search through the array elements
71:     while (i < n && notFound)
72:         // no match?
73:         if (searchVal != intArr[i])
74:             i++; // increment index to compare the next element
75:         else
76:             notFound = FALSE; // found a match
77:     // return search outcome
78:     return (notFound == FALSE) ? i : NOT_FOUND;
79: }
80:
81: int binarySearch(int searchVal, int intArr[], int n)
82: // perform binary search to locate the first
83: // element in array intArr that matches the value
84: // of searchVal
85: {
86:     int median, low, high;
87:
88:     // initialize the search range
89:     low = 0;
90:     high = n - 1;
91:     // search in array
92:     do {
93:         // obtain the median index of the current search range
94:         median = (low + high) / 2;
95:         // update search range
96:         if (searchVal > intArr[median])
97:             low = median + 1;
98:         else
99:             high = median - 1;
100:     } while (!(searchVal == intArr[median] || low > high));
101:     // return search outcome
102:     return (searchVal == intArr[median]) ? median : NOT_FOUND;
103: }
104:
105: void searchInUnorderedArray(int intArr[], int n)
106: // manage the linear search test
107: {
108:     int x, i;
109:     char c;
110:     // perform linear search
111:     cout << "Search in unordered array? (Y/N) ";
112:     cin >> c;
113:     while (c == 'Y' || c == 'y') {
114:         cout << "Enter search value : ";
115:         cin >> x;
116:         i = linearSearch(x, intArr, n);
117:         if (i != NOT_FOUND)
118:             cout << "Found matching element at index " << i << "\n";
119:         else
120:             cout << "No match found\n";
121:         cout << "Search in unordered array? (Y/N) ";
122:         cin >> c;
123:     }
124: }
125:
126: void searchInSortedArray(int intArr[], int n)
127: // manage the binary search test

```



```

128: {
129:     int x, i;
130:     char c;
131:     // perform binary search
132:     cout << "Search in sorted array? (Y/N) ";
133:     cin >> c;
134:     while (c == 'Y' || c == 'y') {
135:         cout << "Enter search value : ";
136:         cin >> x;
137:         i = binarySearch(x, intArr, n);
138:         if (i != NOT_FOUND)
139:             cout << "Found matching element at index " << i << "\n";
140:         else
141:             cout << "No match found\n";
142:         cout << "Search in sorted array? (Y/N) ";
143:         cin >> c;
144:     }
145: }
146:
147: main()
148: {
149:     int arr[MAX];
150:     int n;
151:
152:     n = obtainNumData();
153:     inputArray(arr, n);
154:     cout << "Unordered array is:\n";
155:     showArray(arr, n);
156:     searchInUnorderedArray(arr, n);
157:     sortArray(arr, n);
158:     cout << "\nSorted array is:\n";
159:     showArray(arr, n);
160:     searchInSortedArray(arr, n);
161:     return 0;
162: }

```

程序列表 7.6 中程序执行示例如下：

输出：

Enter number of data points [2 to 10]: 5

arr[0] : 85

arr[1] : 41

arr[2] : 55

arr[3] : 67

arr[4] : 48

Unordered array is:

85 41 55 67 48

Search in unordered array? (Y/N) y

Enter search value : 55

Found matching element at index 2

Search in unordered array? (Y/N) y

Enter search value : 41

Found matching element at index 1

Search in unordered array? (y/N) n

Sorted array is:

41 48 55 67 85

```
Search in sorted array? (Y/N) y
Enter search value : 55
Found matching element at index 2
Search in sorted array? (Y/N) y
Enter search value : 67
Found matching element at index 3
Search in sorted array? (Y/N) n
```

分析:

程序列表 7.6 中程序定义了函数 obtainNumData, inputArray, ShowArray, SortArray, linearSearch, binarySearch, SearchInUnorderArray, SearchInSortedArray 和 main。

前三个函数在分析程序列表 7.5 的程序时已讨论了,在此,我们仅讨论其余几个函数。

函数 linearSearch 执行线性查找功能,在数组 intArr 中,查找第一个与参数 searchVal 匹配的值。函数在数组 intArr 中查找第一个 n 元素。函数 linearSearch 返回 intArr 中与查找字符相符的元素序号,或者如果没有查找到,则函数返回全局常量 NOT-FOUND。该函数使用 while 循环,逐检查数组 intArr 中的元素,当变量 i 的值小于变量 n 的值时,继续循环。此时,局部变量 notFound 的值为 TRUE。第 78 行的语句用条件表示的查找结果。

函数 binarySearch 与函数 linearSearch 的参数相同,也返回同样的值。该函数用局部变量 low 和 high 存储当前查找间隔。函数分别把变量 low 和 high 初始化为 0 和 n-1。第 92 至 100 行的 do-while 循环计算中间元素的标号并且将中间的元素与查找的数值比较。第 96 行的 if 语句进行比较功能,并根据比较结果对 low 变量或 high 变量中的值重新赋值。任意一个变量的重新赋值都能缩短查找间隔。第 102 行的返回语句基于查找数值与当前查找间隔中间数值比较返回函数结果。

函数 searchInUnorderedArray 完成无序数组的查找功能,该函数用开数组参数 intArr 访问数组。函数定义了局部变量用于输出数值,并将该数值存储在此局部变量中,程序第 116 的语句调用函数 linearSearch,传递参数 x(有存储、查找值的局部变量),intArr 和。该语句把函数的结果赋给局部变量 i。第 117 行的 if 语句确定变量 i 中的值是否为 NOT-FOUND。如果条件成立,第 118 行的输出语句显示找到的元素标号,否则,第 120 行的输出语句显示 no-match-found 信息。

函数 searchInSortedArray 类似于函数 searchInUnorderArray。主要不同之处在于:函数 searchInSortedArray 处理有序数组。所以称 binarysearch 函数为用二叉树查找有序数组的方法。

主函数调用这些函数以支持程序执行上面我们要求的功能。

7.7 多维数组

多维数组,每追加一维,就要提供相应的访问方法。二维数组是最常用的多维数组,三

维数组就不太常用了。

新术语:多维数组是一维数组的超集。

语法

二维数组和三维数组

定义二维数组 T 和三维数组的语法是

```
type ovrarray [size1][size2];
```

```
type array [size1][size2][size3];
```

如一维数组一样,每一维下标从 0 开始,且上述定义了每维元素的个数。

例子:

```
double matrixA [100][10];
```

```
char table [4][22][3];
```

```
int index [7][12];
```

C++ 语言是如何存储多维数组的元素呢?许多编译程序是把多维数组的元素依次存储在相邻的存储单元中。当访问某个元素时,需计算该元素的地址。为了更好地说明多维数组的存储方案,我们采用根据不同维的序号的原则。这个方案对不同的维进行编号,并给出高位维和低位维的概念。以下是一个六维数组的例子,它能详细地说明这个存储方案。

```
      1      2      3      4      5      6 <-- dimension number
M [20] [7] [5] [3] [2] [2]
      higher dimension order -->
```

数组 M 的第一个元素是 $M[0][0][0][0][0][0]$,它被存储在数组的第一个存储单元中。数组 M 是被存储在 8400 连续的存储单元中。根据存储方案,存在 M 的第二个存储单元中的元素是第六维为 1 的元素(即 $M[0][0][0][0][0][1]$),以此类推,当第六维的序号由 0 变到最大序号时,存储第五维序号为 1 的元素,此时第六维的序号重新设置为 0。这个过程不断循环进行。直到多维数组中的每个元素都被访问过时为止。这就象在你给汽车加油时一样,油表右边的数字变化的快,左边的数字变化的慢。

下面是另一个三维数组 $M[3][2][2]$ 例子:

$M[0][0][0]$ ← 数组开始有储的单元地址

$M[0][0][1]$ ← 第三维的元素存储在第一项中

$M[0][1][0]$

$M[0][1][1]$ ← 第二维和第三维的元素存储在第二项中

$M[1][0][0]$

$M[1][0][1]$ ← 第三维的元素存储在第二项中

$M[1][1][0]$

$M[1][1][1]$ ← 第二维和第三维的元素存储在第二项中

M[2][0][0]

M[2][0][1]←第三维的元素存储在第三项中

M[2][1][0]

M[2][1][1]←所有的数组元素存储完毕

程序列表 7.7 为 MAT1.CPP 的源程序,它是一个基本的二维数组操作的程序示例。

程序规定这个二维数组为 10 行,30 列,此程序完成下列任务:

- ☐ 请你输入列数,并判断输入值是否有效。
- ☐ 请你输入行数,并判断输入值是否有效。
- ☐ 请你输入数组元素。
- ☐ 计算并显示二维数组每列元素的平均值。

程序列表 7.7 程序 MAT1.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of two-dimension arrays.
3:  The average value of each matrix column is calculated.
4:  */
5:
6: #include <iostream.h>
7:
8: const int MAX_COL = 10;
9: const int MAX_ROW = 30;
10:
11: main()
12: {
13:     double x[MAX_ROW][MAX_COL];
14:     double sum, sumx, mean;
15:     int rows, columns;
16:
17:     // get the number of rows
18:     do {
19:         cout << "Enter number of rows [2 to "
20:             << MAX_ROW << "] : ";
21:         cin >> rows;
22:     } while (rows < 2 || rows > MAX_ROW);
23:
24:     // get the number of columns
25:     do {
26:         cout << "Enter number of columns [1 to "
27:             << MAX_COL << "] : ";
28:         cin >> columns;
29:     } while (columns < 1 || columns > MAX_COL);
30:
31:     // get the matrix elements
32:     for (int i = 0; i < rows; i++) {
33:         for (int j = 0; j < columns; j++) {
34:             cout << "X[" << i << "][" << j << "] : ";
35:             cin >> x[i][j];
36:         }
37:         cout << "\n";
38:     }
39:
40:     sum = rows;
41:     // obtain the sum of each column
42:     for (int j = 0; j < columns; j++) {
43:         // initialize summations
44:         sumx = 0.0;
```

```

45:     for (i = 0; i < rows; i++)
46:         sumx += x[i][j];
47:     mean = sumx / sum;
48:     cout << "Mean for column " << j
49:         << " = " << mean << "\n";
50: }
51: return 0;
52: }

```

程序列表 7.7 的程序执行结果如下:

输出:

```

Enter number of row [2 to 30]:3
Enter number of columns [1to 10]:3
x[0][0]:1
x[0][1]:2
x[0][2]:3
x[1][0]:4
x[1][1]:5
x[1][2]:6
x[2][0]:7
x[2][1]:8
x[2][2]:9
mean for column 0=4
mean for column 1=5
mean for column 2=6

```

分析:

程序列表 7.7 中程序在第 8、9 行定义了全局常量 MAX_COL 和 MAX_ROW。这些常量规定了程序中二维数组的大小。函数 main 定义了数组 x, 它为 MAX_ROW 行, MAX_COL 列, 此外还定义了其它非数组变量。

程序第 18 至 22 行的 do-while 循环提示你输入含有你的数据的数组 x 的行数。第 19 和 20 行的输出语句指出有效的行数范围。第 21 行的语句将你输入的行数存储在变量 rows 中。

程序第 25 至 29 行的第二个 do-while 循环提示你输入含有你的数据的数组 x 的列数。第 26 和 27 行的输出语句显示有效的列数范围。第 28 行的语句将你输入的列数存储在变量 columns 中。

程序第 32 至 38 行的嵌套循环请你输入数组元素。外层循环由变量 j 控制, 从 0 变化到 row3-1。每次循环增量为 1。内层循环由变量 i 控制, 由 0 变化到 columns-1, 每次循环增量为 1。程序第 34 行的输出语句显示你将输入的元素号。第 35 行把你输入的数值存储在数组元素 x[i][j] 中。

程序第 40 行开始计算各行元素的平均值。该语句把变量 rows 中的整数赋值给 double 型变量 sum。程序第 42 至 50 行为嵌套 for 循环。外层 for 循环由变量 j 控制, 由 0 变化到 columns-1, 每次循环增量为 1。外层循环的第一条语句把 0 赋值给变量 sumx。内层 for 循环从第 45 行开始。由变量 i 控制, 由 0 变化到 rows-1, 每次循环增量为 1。第 46 行的语句计算各列元素的平均值, 并把该值赋值给变量 mean。第 48 和

49 行的输出语句显示列值和各列的平均值。

注意:第 42 行的 for 循环重新定义了控制变量 j(第 45 行也同样重新定义了控制变量 i)。为什么? 第 33 行也定义了控制变量 j, 然而第 33 行的循环范围不超过外层 for 循环。一旦第一对嵌套循环执行完毕, 循环控制变量 j 就被实时系统删除。

7.8 多维数组的初始化

C++ 语言允许你用类似一维数组初始化的方法来对多维数组初始化。你需要使用一个数据表, 它与被存储的初始化的数组元素顺序相同。现在, 你就认识到理解 C++ 语言是如何存储多维数组元素的重要性了。我们改动前面 C++ 语言程序, 利用程序提供一个初始化数据表, 这样就无需你输入任何数据。此外, 程序显示数组元素, 并计算各列元素的平均值。

程序列表 7.8 程序 MAT2.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of two-dimension arrays.
3:  The average value of each matrix column is calculated.
4:  */
5:
6: #include <iostream.h>
7:
8: const int MAX_COL = 3;
9: const int MAX_ROW = 3;
10:
11: main()
12: {
13:     double x[MAX_ROW][MAX_COL] = {
14:         1, 2, 3, // row # 1
15:         4, 5, 6, // row # 2
16:         7, 8, 9 // row # 3
17:     };
18:     double sum, sumx, mean;
19:     int rows = MAX_ROW, columns = MAX_COL;
20:
21:     cout << "Matrix is:\n";
22:     // display the matrix elements
23:     for (int i = 0; i < rows; i++) {
24:         for (int j = 0; j < columns; j++) {
25:             cout.width(4);
26:             cout.precision(1);
27:             cout << x[i][j] << " ";
28:         }
29:         cout << "\n";
30:     }
31:     cout << "\n";
32:
33:     sum = rows;
34:     // obtain the sum of each column
35:     for (int j = 0; j < columns; j++) {
36:         // initialize summations
37:         sumx = 0.0;
38:         for (i = 0; i < rows; i++)
```

```

39:         sumx += x[i][j];
40:         mean = sumx / sum;
41:         cout << "Mean for column " << j
42:              << " = " << mean << "\n";
43:     }
44:     return 0;
45: }

```

程序列表 7.8 中程序的执行结果如下：

输出：

Matrix is

1 2 3

4 5 6

7 8 9

mean for column 0=4

mean for column 1=5

mean for column 2=6

分析：

程序列表 7.8 中程序定义了二维数组 `x`，并用一个数据表对它初始化。注意：程序定义了常量 `MAX_COL` 和 `MAX_ROW`，用以表示二维数组的大小。第 13 至 17 行的定义语句即为对数组进行初始化。主函数 `main` 也用常量 `MAX_ROW` 和 `MAX_COL` 对变量 `rows` 和 `columns` 进行初始化。函数进行初始化有两个原因：第一程序无需请你输入任何数值；第二程序处理固定大小的数组 `x`。

程序在第 21 至 30 行使用嵌套 `for` 循环以显示二维数组 `x` 的元素。第二个嵌套 `for` 循环计算各列元素的平均值。

7.9 多维数组参数

C++ 语言允许用多维数组作为函数参数。与一维数组类似，C++ 语言允许你规定或省略数组参数大小的说明，然而，第二种情况，你反能省略数组的第一维大小说明。如果你希望数组参数能接收固定维数的数组，你就在参数定义时定义每维的大小。反之，如果你希望数组参数接受同一类型，但第一维大小不同的数组时，则用 `[]` 代表数组的第一维

语法

固定数组参数

定义固定数组参数的语法是

`type paraneter Name [dim1size][dim2size]..`

例子：

```

int minMatrix (int intMat[100][20],int rows, int cols);
void sort (unsigned mat[23][55],int rows, int cols,int colIndex);

```

语法

开式数组参数

定义开式数组参数的语法是

```
type parameterName [ ][dim2size]...
```

例子:

```
int minMat (intintMat[ ][100]) int rows, int cols);  
void sort (unsigned mat[ ][55],int rows, int cols,int colIndex);
```

让我们看看下面这个例子。程序列表 7.9 为 MAT3.CPP 的源程序。该程序与程序列表 7.7 中的 MAT1.CPP 执行同样的功能。通过对程序 MAT1.CPP 改动,生成程序 MAT3.CPP。并且把程序执行的各项功能写成单独的函数。这样 MAT3.CPP 就是一个程序 MAT1.CPP 的高结构化版本。

程序列表 7.9 程序 MAT3.CPP 的源代码

```
1: /*  
2:  C++ program that demonstrates the use of two-dimension arrays.  
3:  The average value of each matrix column is calculated.  
4:  */  
5:  
6: #include <iostream.h>  
7:  
8: const int MAX_COL = 10;  
9: const int MAX_ROW = 30;  
10:  
11: int getRows()  
12: {  
13:     int n;  
14:     // get the number of rows  
15:     do {  
16:         cout << "Enter number of rows [2 to "  
17:             << MAX_ROW << "] : ";  
18:         cin >> n;  
19:     } while (n < 2 || n > MAX_ROW);  
20:     return n;  
21: }  
22:  
23: int getColumns()  
24: {  
25:     int n;  
26:     // get the number of columns  
27:     do {  
28:         cout << "Enter number of columns [1 to "  
29:             << MAX_COL << "] : ";  
30:         cin >> n;  
31:     } while (n < 1 || n > MAX_COL);  
32:     return n;  
33: }  
34:  
35: void inputMatrix(double mat[][MAX_COL],  
36:                 int rows, int columns)  
37: {  
38:     // get the matrix elements  
39:     for (int i = 0; i < rows; i++) {  
40:         for (int j = 0; j < columns; j++) {  
41:             cout << "X{" << i << "][" << j << "]: ";  
42:             cin >> mat[i][j];  
43:         }  
44:         cout << "\n";  
45:     }  
46: }
```



```

47:
48: void showColumnAverage(double mat[][MAX_COL],
49:                        int rows, int columns)
50: {
51:     double sum, sumx, mean;
52:     sum = rows;
53:     // obtain the sum of each column
54:     for (int j = 0; j < columns; j++) {
55:         // initialize summations
56:         sumx = 0.0;
57:         for (int i = 0; i < rows; i++)
58:             sumx += mat[i][j];
59:         mean = sumx / sum;
60:         cout << "Mean for column " << j
61:              << " = " << mean << "\n";
62:     }
63: }
64:
65: main()
66: {
67:     double x[MAX_ROW][MAX_COL];
68:     int rows, columns;
69:     // get matrix dimensions
70:     rows = getRows();
71:     columns = getColumns();
72:     // get matrix data
73:     inputMatrix(x, rows, columns);
74:     // show results
75:     showColumnAverage(x, rows, columns);
76:     return 0;
77: }

```

程序列表 7.9 中的程序执行结果如下:

输出:

```

Enter number of rows [2 to 30]:3
Enter number of columns [1 to 10]:3
X[0][0]:10
X[0][1]:20
X[0][2]:30
X[1][0]:40
X[1][1]:50
X[1][2]:60
X[2][0]:70
X[2][1]:80
X[2][2]:90
mean for column 0=40
mean for column 1=50
mean for column 2=60

```

分析:

程序列表 7.9 中程序定义了函数 `getRows`, `getColumns`, `inputMatrix`, `showColumnAverage`, 和 `main`。函数 `getRows` 请你输入所需的数组行数。该函数返回你的有效输入。同样, 函数 `getColumns` 返回数组列的有效输入。

函数 `inputMatrix` 请你输入数组元素, 该函数有三个参数, 参数 `mat` 为数组参数 (一个

开式数组参数)。参数 rows 和 column 确定数组 mat 的大小。

函数 showColumnAverage 计算并显示数组参数 mat 各列元素的平均值。参数 rows 和 columns 规定数组 mat 的行列大小。

这个函数中的语句与程序 MWAT1.CPP 相同,程序 MAT3.CPP 作为调用这些函数的框架来执行不同的功能。从结构化程序设计观点来看,程序 MAT3.CPP 优于程序 MAT1.CPP。

函数 main 定义了 MAX_ROW 行,MAX_COL 列的数组 x。主函数调用函数 getRows 和函数 getColumnbs 分别得到确定的行数和列数。第 73 行的语句调用函数 inputMatrix,实参为 x,rows 和 colwmns。第 75 行的语句调用函数 showColumnAverage,实参为 x,rows 和 columns。

7.10 小结

今天的课程我们讨论了几个处理数组的主题,包括一维数组和多维数组。你学习了下列主题:

- ☐ 定义一维数组需要你规定数组元素的数据类型、数组名、数组大小(或用“[]”符)。所有 C++ 语言的数组下标都从 0 开始。上标等于数组大小减 1。
- ☐ 使用一维数组需要你规定数组名加有效的数组大小(可以用“[]”符)
- ☐ 在定义一维数组的同时,可对它初始化。初始化的数据表包括在括号内,各数据项由逗号隔开。C++ 语言允许你的数据数目小于数组的大小。这种情况下,编译程序能自动地把你未初始化的元素赋值为 0。此外,C++ 语言允许你省略规定数组的大小。根据你初始化的数据项数目确定数组的大小。
- ☐ 数组的排序是很重要的一种非数值数组操作。把数组元素或按从小到大的顺序排列,或按从大到小的顺序排列。数组元素的排序便于查找,对于数组排序而言,新的 Comb 排序法是最有效的方法。
- ☐ 数组的查找即为查找与欲查找值相同的数组元素。查找的方法因数组的无序或有序的不同而不同。线性查找法用于无序数组;二叉数查找法用于有序数组。
- ☐ 定义多维数组,需要你规定数组元素的类型、数组名和数组元素的数目(允许使用“[]”符)。所有 C++ 数组都有低边界 0。数组上界等于元素数目减 1。
- ☐ 调用多维数组需要你规定数组名和数组的大小(允许使用“[]”符)。
- ☐ 多维数组的初始化与多维数组的定义可同时进行。初始化的数据表写在括号中,各项由逗号分隔开。C++ 语言允许你初始化的数据项少于数组的大小,在这种情况下,编译程序能自动把你未初始化的数组元素赋值为 0。
- ☐ 多维数组作为函数参数有两种形式:第一种是固定数组参数;第二种为开式数组参数。固定数组参数需给定数组参数各维的大小。开式数组参数仅允许第一维省略规定其大小。这种情况下,其第一维的大小可以变化。

7.11 问与答

问:C++语言允许改变数组的大小吗?

答:不允许。C++语言不允许你重新定义数组的大小。

问:是否可以定义 void 型数组(例如:void array [81];)以建立存储区?

答:不可以,C++语言不允许你调用 void 型数组。因为 void 类型无法定义数组的大小。用 char 或 Unsigned char 型数组可开辟一段存储区。

问:C++语言允许重新定义一个数组吗?

答:C++语言允许重新定义一个数组来改变它的基本类型、维数及数组大小。如下示例:

```
#include <iostream.h>
const MAX = 100;
const MAX_ROWS = 100;
const MAX_COLS = 20;
main()
{
    //declare avariables here?
    { double x[MAX];
        //declare other variables?
        //statements to manipulate the single—dimensional
        //array x
        {
            double x[MAX_ROWS][MAX_COLS];
            //declare other variables?
            //statements to manipulate the matrix x
        }
    }
    return 0;
}
```

主函数 main 在第一个嵌套语句块中定义了数组 x。当程序执行到该块的最后的语句时,实时系统删除数组 x 和此块中定义的其它变量。然后,函数在第二个块中重新定义数组 x,当程序执行到该块的最后一条语句时,实时系统删除数组 x 和此块中定义的其它变量。

问:数组受预先定义的类型限制吗?

答:不受限制。C++语言允许你根据用户需要定义数组的类型。

7.12 专题讨论

本章的专题讨论包括一些测验问题,帮助你巩固本章所学的内容。同时也提供了一些练习,便于你在程序编制过程中复习学到的方法。在进行下章的内容之前,务必搞懂测验和练习的答案。答案在附录 B 中给出。

7.12.1 测验

1. 下列程序的输出结果是什么?

```
#include<iostream.h>
const int MAX=5;
main()
{
    double x[MAX];
    x[0]=1
    for(int i=1;i<MAX;i++)
        x[i]=i * x[i-1];
    for(i=0;i<MAX;i++)
        cout<<"x["<<i<<"]<<"\n";
    return 0;
}
```

2. 下列程序的输出结果是什么?

```
#include<iostream.h>
#include<math.h>
const int MAX=5;
main()
{
    double x[MAX];
    for(int i=0;i<MAX;i++)
        x[i]=sqrt(double(i));
    for(i=0;i<MAX;i++)
        cout<<"x["<<i<<"]="<<x[i]<<"\n";
    return 0;
}
```

3. 下面程序有何错误?

```
#include<iostream.h>
const int MAX=5;
main()
{
    double x[MAX];
    x[0]=1;
    for(int i=0;i<MAX;i++)
        x[i]=i * x[i-1];
    for(i=0;i<MAX;i++)
        cout<<"x["<<i<<"]="<<x[i]<<"\n";
    return 0;
}
```

7.12.2 练习

通过编辑程序 ARRAY6.CPP 生成程序 ARRAY7.CPP。用 ShellMetzner 查序法代替函数 sortArray 中的 Comb 查序法。

第一周回顾

在进行下周的 Turbo C++ 语言编程的内容之前,让我看一个典型的例子。这个例子在下两周结束时也被引用。这个例子是一个简单的猜数游戏。如程序列表 R1.1 所示。该程序在 0 到 1000 的数中,随意挑选一个。而后请你输入一个 0 到 1000 的数。如果你输入的数大于被选的数程序将告诉你,你所猜的数过大。反之,如果你输入的数小于被选的数,则程序将以你的胜利而告终。该程序允许你最多猜 11 次,你可以输入一个负数退出此程序。在这种情况下,程序将停止此游戏,并告诉你答案。

程序列表 R1.1 程序 GAME1.CPP 的源代码

```
1: #include <stdlib.h>
2: #include <iostream.h>
3: #include <time.h>
4:
5: // declare a global random number generating function
6: int random(int maxVal)
7: { return rand() % maxVal;
8:
9:
10: main()
11: {
12:     int n, m;
13:     int MaxIter = 11;
14:     int iter = 0;
15:     int ok = 1;
16:
17:
18:     // reseed random-number generator
19:     srand((unsigned)time(NULL));
20:     n = random(1001);
21:     m = -1;
22:
23:     // loop to obtain the other guesses
24:     while (m != n && iter < MaxIter && ok == 1) {
25:         cout << "Enter a number between 0 and 1000 : ";
26:         cin >> m;
27:         ok = (m < 0) ? 0 : 1;
28:         iter++;
29:         // is the user's guess higher?
30:         if (m > n)
31:             cout << "Enter a lower guess\n\n";
32:         else if (m < n)
33:             cout << "Enter a higher guess\n\n";
34:         else
35:             cout << "You guessed it! Congratulations.";
36:     }
37:     // did the user guess the secret number
38:     if (iter >= MaxIter || ok == 0)
39:         cout << "The secret number is " << n << "\n";
40:
41:     return 0;
42: }
```

程序列表 R1.1 中的程序执行结果如下:

输出:

Enter a number between 0 and 1000;500

Enter a lower guess

Enter a number between 0 and 1000;250

Enter a higher guess

Enter a number between 0 and 1000;-1

Enter a higher guess

The secret number is 399

分析:

程序列表 R1.1 中程序定义了函数 random1 返回一个 0 到 1000 范围内的随机值。该程序也定义了主函数 main,该函数实施猜数游戏。主函数在第 12 至 15 行定义了一

些局部变量。第 19 行中的语句重新查找生成的随机数。第 20 行的语句把这所选的随机数赋值于变量 `n` 中。第 21 行的语句把 -1 赋值给变量 `m`, 它用于存储你所猜的数。

程序第 24 至 36 行的 `while` 循环进行猜数游戏是否执行这个 `while` 循环取决于下面条件是否为真。

- ☐ 你猜的数(即变量 `m` 的值)与变量 `n` 中的随机值不同。
- ☐ 循环的次数(变量 `iter` 的值)小于规定的最大次数(即变量 `MaxIter` 的值)。
- ☐ 变量 `ok` 中的值为 1。

该循环体中的第一条语句请你输入 0 到 1000 中的一数。第 26 行的语句把你输入的数值存储在变量 `m` 中。如果你输入一负值, 则第 27 行的语句把变量 `ok` 赋值为 0, 否则赋值为 1。第 28 行的语句使变量 `iter` 加 1。

程序第 30 至 35 行的多重 `if` 语句判断你输入与随机数的大小关系, 并反馈给你准确的比较信息。

第 38 行的 `if` 语句是如果你进行了 `MaxIter` 次猜数失败时, 或当你输入一负数时, 向你显示答案。

```
1: /*
2:  C++ program that demonstrates enumerated types
3:  */
4:
5: #include <iostream.h>
6:
7: enum mathError { noError, badOperator, divideByZero };
8:
9: void sayError(mathError err)
10: {
11:     switch (err) {
12:         case noError:
13:             cout << "No error";
14:             break;
15:         case badOperator:
16:             cout << "Error: invalid operator";
17:             break;
18:         case divideByZero:
19:             cout << "Error: attempt to divide by zero";
20:     }
21: }
22:
```


第二周浏览

第二周,我们将继续向你介绍 C++ 语言。内容覆盖了 C++ 语言许多高级部分。你将学到用户定义的数据类型—特别是结构和指针。这周你还将学到有关函数的内容,同时还向你介绍 C++ 语言的面向对象的编程方法(OOP)。你将学习类、部件和用这些部件的规则。此外,你还将学习使用 C++ 流库的基本文件 I/O。第十三天将向你介绍在 Windows 中使用 OWL 库编程的基础知识。第十四天介绍了简单的基于 OWL 的 Windows 程序。

第八章 用户定义的类型和指针(第八天)

生成用户定义的数据类型是现代编程语言所期望具有的功能之一。本章我们将着重讨论枚举数据类型和结构。它们会使你更好地组织你的数据。此外,我们还将讨论如何使用指向简单变量,数组、结构以及动态存储数据的指针。归纳起来,我们将讨论以下几个问题:

- ☐ 用 typedef 定义类型
- ☐ 枚举类型
- ☐ 结构
- ☐ 联合
- ☐ 引用变量
- ☐ 指向变量的指针
- ☐ 指向数组的指针
- ☐ 指向结构的指针
- ☐ 使用指针指向和组织动态存储的数据
- ☐ 远程指针

8.1 C++语言中的类型定义

C++语言提供 typedef 关键字,用 typedef 可以定义新的数据类型名以代替已有的类型名。

语法

关键字

typedef 使用语法为:

typedef newType;

例子:

typedef unsigned word;

typedef unsigned char byte;

typedef unsigned char boolean;

typedef 定义新的类型名代替已有的类型名。你可以用 typedef 对已有的数据类型名缩写,或把已有的数据类型名定义成一个对你更为熟悉的类型名(如上述第二个例子,用 typedef 定义了一个字节类型)。此外,typedef 语法也能定义一个能更好说明已有的数据类型名字。上述第三个例子就说明了这一点。进一步,你也同样可用 typedef 定义一个数组类型名。

语法

数组类型名

定义一个数组类型名的语法为:

```
typedef baseType arrayTypeName [arraySize];
```

用 typedef 语句定义 arrayTypeName, 其基本类型和数组的大小分别为 baseType 和 arraySize。

例子:

```
typedef double vector[10];
```

```
typedef double matrix[10][30];
```

这样, 标识符 vector 和 matrix 就是数组类型名。

8.2 枚举类型

枚举类型遵循的规则是: 定义的枚举类型的标识符是唯一的, 但是与此标识符有关的值却不是唯一的。

新术语: 枚举类型就是定义了一个具有唯一标识符的表和与这些标识符有关的值。

语法

枚举类型

定义枚举类型的语法为:

```
enum enumType {<list of enumerated identifiers>;};
```

例子:

```
enum Boolean {false, true};
```

```
enum YesNo {no, yes, dontCare, maybe};
```

```
enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

下面是一个定义枚举类型的例子:

```
enum CPUtype {i8088, i80286, i80386DX, i80386SX, i80486DX, i80486SX};
```

C++ 语言用整数值与之相对应。例如, 在这个类型中, 编译程序用 0 对应 i8088, 1 对应 i80286 等等。

C++ 语言在枚举类型的定义中是很灵活的。首先, C++ 语言允许你明确地规定一个值对应一个枚举标识符, 如下面示例:

```
enum weekday {sunday=1, Monday, Tuesday,  
              Wednesday, Thursday, Friday, Saturday};
```

这个定义明确地规定 1 对应枚举标识符 Sunday。编译程序而后就将下一个整数 2 对应下一个标识符 Monday, 以此类推。C++ 也允许你规定一个值对应枚举列表中的每一个元素。这些值不需要是唯一的。下列的一些示例就充分说明了 C++ 语言在枚举类型定义中的灵活性。

```
//explicit value assignment for every list number
enum colors{black=1,red=2,blue=3,green=5,yellow=7,white=11};
//intermittent value assignment
enum colors{black=1,red,blue,green=5,gelow=7,white=11};
//duplicate values
enum CPUtype{i8088=1,i80286=2,i80386DX=3,i80386bx=3,i80486DX=4,i80486sx=4};
enum choiceType{false,true,dontcare=0}
```

在最后一个例子中,编译程序在出错时将 0 与 false 对应,然而,编译程序也允许 0 与 DontCare 对应,因为已给予明确规定。

C++ 语言允许你用以下方法用枚举类型来定义变量:

1. 枚举类型的定义可以包括此类型变量的定义。语法为:

```
enum enumType{<list of enumerated identifiers>}<list of variables>;
```

例子:

```
enum weekDay{Sun=1,Mon,Tue,Wed,Thu,Fri,Sat}
recyceDay,payDay,MovieDay;
```

2. 枚举类型分离和枚举类型变量的分离定义包括多行语句以便分别定义类型和有关变量,一般语法为:

```
enum enumType{<eist of enumerated identifiers>;};
enumType var1,var2,...,varN;
```

下面让我们看一个例子。程序列表 8.1 为 ENVMI.CPP 的源程序。这个程序执行一个简单的四则运算,主要完成以下功能:

- ☐ 请你输入一个数,一个运算符(+, -, *, /),再输入一个数。
- ☐ 执行所需运算。
- ☐ 显示运算值,运算符以及运算结果。如运算无效,就显示一个错误信息,说明错误类型(你可能输入一个错误的运算符或除数为 0)。

程序列表 8.1 程序 ENUM1. CPP 的源代码

```

23: main()
24: {
25:     double x, y, z;
26:     char op;
27:     mathError error = noError;
28:
29:     cout << "Enter a number, an operator, and a number : ";
30:     cin >> x >> op >> y;
31:
32:     switch (op) {
33:         case '+':
34:             z = x + y;
35:             break;
36:         case '-':
37:             z = x - y;
38:             break;
39:         case '*':
40:             z = x * y;
41:             break;
42:         case '/':
43:             if (y != 0)
44:                 z = x / y;
45:             else
46:                 error = divideByZero;
47:             break;
48:         default:
49:             error = badOperator;
50:     }
51:
52:     if (error == noError)
53:         cout << x << " " << op << " " << y << " = " << z;
54:     else
55:         sayError(error);
56:     return 0;
57: }

```

程序列表 8.1 中程序执行结果如下:

输出:

Enter a number,an operator,and a number,355/113

355/113=3.141593

分析:

程序列表 8.1 中程序的第 7 行定义了一个枚举类型 `mathError`。这个类型有 3 个枚举值: `noError`, `badOperator` 和 `divideByZero`。

程序的第 9 行至第 21 行定义了函数 `sayError`, 该函数的功能为基于枚举类型参数 `err` 的值显示一条信息。这个函数在程序第 11 行用到 `Switch` 语句以显示对应于不同枚举值的信息。

主函数 `main` 定义了 `double` 型变量 `x, y, z`, 以分别表示运算数和运算结果。此外, 主函数定义了一个 `char` 类型的变量 `op` 以存储需要的运算符, 并且还定义了枚举类型变量 `error` 以存储错误状况。函数用枚举值 `noError` 初始化变量 `error`。

在程序第 29 行的输出语句请你输入运算数和运算符, 第 30 行的语句, 把你输入的值依次存入变量 `x`, `op` 和 `y`。函数在第 32 行使用 `switch` 语句, 用于检验变量 `op` 的值是否为所需的运算符。在第 33、36、39 以及 42 行的 `case` 语句的某个条件满足时, 提供运算数, 完成所需运算操作。最后的一个 `case` 标号包含一个 `if` 语句, 用于测试除数是否为

0。如果真,则 else 从句语句把枚举值 divideByZero 值赋于变量 error。

在程序第 48 行的 default 语句处理无效的运算符。第 49 行的语句把枚举值 badOperator 赋于变量 error。

程序第 52 行的语句判断是否变量 error 中值为 noError,如为真,则程序执行第 53 行的输出语句、显示运算数、运算符及运算结果。否则程序调用函数 sayError 以显示错误信息。

8.3 结构

C++ 语言支持结构定义,其成员可能是预先定义的类型或具它结构。

新术语:结构使你定义一个新的类型,它能把一些字段或成员逻辑地组合在一起。

语法

结构

定义结构的语法为:

```
struct structTag{  
    <list of members>  
}
```

例子:

```
struct point {  
    double x;  
    double y;  
};  
  
struct reetangle {  
    point upperLeftCorner;  
    point loweRightcorner;  
    double area;  
};  
  
struct eirde{  
    point circle;  
    double area;  
};
```

一旦你定义了一个结构类型,就可以用这种类型定义变量,以下示例说明了这点:

```
point p1,p2,p3;
```

当你定义结构自身时,也可定义结构变量:

```
struct point{  
    double x;  
    double y;
```

```
}p1,p2,p3;
```

新术语:非标记结构 (Untagged structures) 允许定义结构变量, 无需定义结构名。

注意, C++ 语言允许定义非标记结构。例如, 下列结构变量 p1, p2, p3 的定义就省略了结构名:

```
struct {  
    double x;  
    double y;  
} p1,p2,p3;
```

C++ 语言允许你初始化结构变量, 如下示例:

```
point pt={1.0, -8.3};
```

用. 运算符可以访问结构成员。如下示例:

```
p1.x=12.45  
p1.y=34.56  
p2.x=23.4/p1.x  
p2.y=0.98*p1.y
```

让我们看下面的这个例子。程序列表 8.2 显示了 STRUCT1.CPP 的源程序。此程序请你确定四组坐标系, 从而确定四个矩形。每个矩形由左上角点和右下角点的 c、y 坐标值确定, 程序计算每个矩形的面积, 并对矩形的面积排序。而后把矩形按其面积大小的顺序依次显示。

程序列表 8.2 程序 TRUCT1.CPP 的源代码

```
1: /*  
2:  C++ program that demonstrates structured types  
3:  */  
4:  
5: #include <iostream.h>  
6: #include <stdio.h>  
7: #include <math.h>  
8:  
9: const MAX_RECT = 4;  
10:  
11: struct point {  
12:     double x;  
13:     double y;  
14: };  
15:  
16: struct rect {  
17:     point ulc; // upper left corner  
18:     point lrc; // lower right corner  
19:     double area;  
20:     int id;  
21: };  
22:  
23: typedef rect rectArr[MAX_RECT];  
24:  
25: main()  
26: {
```



```

27: rectArr r;
28: rect temp;
29: double length, width;
30:
31: for (int i = 0; i < MAX_RECT; i++) {
32:     cout << "Enter (X,Y) coord. for ULC of rect. # "
33:         << i << " : ";
34:     cin >> r[i].ulc.x >> r[i].ulc.y;
35:     cout << "Enter (X,Y) coord. for LRC of rect. # "
36:         << i << " : ";
37:     cin >> r[i].lrc.x >> r[i].lrc.y;
38:     r[i].id = i;
39:     length = fabs(r[i].ulc.x - r[i].lrc.x);
40:     width = fabs(r[i].ulc.y - r[i].lrc.y);
41:     r[i].area = length * width;
42: }
43:
44: // sort the rectangles by areas
45: for (i = 0; i < (MAX_RECT - 1); i++)
46:     for (int j = i + 1; j < MAX_RECT; j++)
47:         if (r[i].area > r[j].area) {
48:             temp = r[i];
49:             r[i] = r[j];
50:             r[j] = temp;
51:         }
52:
53: // display rectangles sorted by area
54: for (i = 0; i < MAX_RECT; i++)
55:     printf("Rect # %d has area %5.4lf\n", r[i].id, r[i].area);
56: return 0;
57: }

```

程序列表 8.2 中程序执行结果如下：

输出：

```

Enter(x,y) coord. for VLC of rect. #0:1 1
Enter(x,y) coord. for VLC of rect. #0:2 2
Enter(x,y) coord. for VLC of rect. #1:1.5 1.5
Enter(x,y) coord. for VLC of rect. #1:3 4
Enter(x,y) coord. for VLC of rect. #2:1 2
Enter(x,y) coord. for VLC of rect. #2:5 8
Enter(x,y) coord. for VLC of rect. #3:4 6
Enter(x,y) coord. for VLC of rect. #3:8 4
Rect #0 has area 1.0000
Rect #1 has area 3.7500
Rect #2 has area 8.0000
Rect #3 has area 24.0000

```

分析：

程序列表 8.2 中程序包含头文件 IOSTREAM.H, MATH.H 和 STDWO.H。定义了全局变量 MAX_RECT 来规定矩形的最多数目。程序第 11 行定义了结构 point, 它是由两个 double 类型的成员 x,y 组成的。这个结构的模型为一个二维的点。程序第 16 行定义了结构 rect, 它的模型为一个矩形。该结构只有两个 point 类型的成员 ulc 和 lrc。一个 double 类型的成员 area 和一个 int 类型的成员 id。成员 ulc 与 lrc 分别代表矩形的左上角点和右下角点的坐标值。矩形的面积赋于成员 area。成员 id 存储一个数值标识号。

程序第 23 行的 typedef 语句定义了 rectArr 类型为结构 rect, 数组大小为 MAX_RECT。

主函数定义了 rectArr 类型的数组 r、rect 类型的结构 temp 和 double 类型的变量 tengty 和 width。

主函数 main 在程序第 31 至 42 行用到一个 for 循环。请你输入矩形两角点坐标, 计算其矩形面积, 同时设定 id 数值。在第 32、33 行和第 35、36 行的输出语句就是请你分别输入左上角点和右下角点的坐标值。第 34 和 37 行的输入语句就分别把你输入的坐标值存入 r[i].ulc.x, r[i].ulc.y, r[i].lrc.x 和 r[i].lrc.y 中。第 38 行语句把循环控制变量 i 的值赋给 r[i].id。第 39 行的语句用元素 r[i] 中的成员 lc 和 lrc 的 x 值计算矩形长度。第 40 行的语句是用元素 r[i] 中的成员 ulc 和 lrc 的 y 成员计算矩形宽度。第 41 行语句计算矩形的面积, 并把面积值赋于成员 r[i].area。

程序第 44 到 51 行的嵌套循环的作用是根据成员 area 的值对数组 r 的元素进行排序。此循环采用的是简单的冒泡排序法(这种方法对小数组排序很有效)。第 47 行的 if 语句比较元素 r[i] 和 r[j] 的 area 值大小, 如果矩形 r[i] 的面积大于矩形 r[j] 的面积, 则经过第 48 至第 50 行的语句, 交换 r[i] 和 r[j] 中的所有成员。交换过程用到结构 temp。这一过程说明了可用一条语句把一个结构的所有成员赋值给另一个结构。程序第 54 和 55 行的 for 循环根据矩形面积大小显示排序后的矩形。第 55 行的输出语句使用函数 printf 以显示矩形的 id 和 area 值。

8.4 联合

联合的大小为其成员的最大数目。

新术语: 联合是一种特殊的结构, 其成员相互排斥。

语法

联合

联合的语法为:

```
union unionTag{
    type1 member1;
    type2 member2;
    ....
    typeN memberN;
}
```

例子:

```
union Long{
    unsigned mword[2];
    Long mlong;
};
```

联合提供了一个简单的用于快速数据转换的介值。在过去几十年中,Union 联合具有特别重要的作用。当时存储器的价格很高,使用联合来统一使用存储器就十分可行。使用操作符也可以访问到联合的成员。

8.5 引用变量

在这节中,你将学会用一个参数类型名加上 & 符号来定义引用参数的方法。引用参数可以是它变元的别名。作用在引用参数上的任何变换,也同样会作用在它的变元上。除了引用参数之外,C++ 语言还支持引用参变量。你可以用它的别名来操作引用变量。做为一个初学 C++ 语言的程序员,则开始运用引用变量,可能会感到较困难。另一方面,你可能会频繁地使用引用变量。但当你熟练地运用 C++ 语言时,你将会发现引用变量是实现处理高级分类设计编程技巧的一种有效工具。此书只讨论引用变量的基础知识。

新术语:引用变量(reference variable)正如引用参数,是某个变量的别名。

语法

引用变量:

定义引用变量的语法:

```
type&refvar;
```

```
type&refvar=avar;
```

refvar 是一个引用变量,当它被定义的同时,也可对它进行初始化。在使用一个引用变量之前,你必须确保该引用变量已被初始化。

例子:

```
int x=10,y=3;
```

```
int& rx=x;
```

```
int& ry;
```

```
ry=y;//take the reference
```

这个示例就说明了两个程序中的引用变量。程序列表 8.3 是 DEFVAR1.CPP 的源程序,这个程序的功能是通过使用变量本身或它的引用变量,来显示和改变变量中的值。该程序没有输入。

程序列表 8.3 程序 REFVAR1.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates reference variables
3:  */
4:
5: #include <iostream.h>
6:
7: main()
8: {
9:     int x = 10;
10:    int& rx = x;
```

```

11: // display x using x and rx
12: cout << "x contains " << x << "\n";
13: cout << "x contains (using the reference rx) "
14:     << rx << "\n";
15: // alter x and display its value using rx
16: x *= 2;
17: cout << "x contains (using the reference rx) "
18:     << rx << "\n";
19: // alter rx and display value using x
20: rx *= 2;
21: cout << "x contains " << x << "\n";
22: return 0;
23: }

```

程序列表 8.3 中的程序执行结果如下:

输出:

```

x contains 10
x contains (using the reference rx) 10
x contains (using the reference rx) 20
x contains 40

```

分析:

程序列表 8.3 的程序定义了一个整型变量 `x` 和一个整型引用变量 `rx`, 程序初始化变量 `x` 为 10, 初始化引用变量 `rx` 为变量 `x`。

程序第 12 行的输出语句用变量 `x` 来显示变量 `x` 的值, 相反, 程序的第 13、14 行则是用引用变量 `rx` 来显示变量 `x` 的值。

程序第 16 行的语句使变量 `x` 中的整数乘 2。程序第 17、18 行的输出语句是用引用变量 `rx` 来显示变量 `x` 中的最新值。如输出结果所示, 引用变量准确地显示变量 `x` 的当前值。

程序第 20 行的语句, 通过使用引用变量 `rx` 使变量 `x` 中的值翻倍。第 21 行的输出语句用变量 `x` 来显示变量 `x` 的当前值。输出结果再次说明变量 `x` 和引用变量 `rx` 是同步的。

8.6 指针概述

每条信息如程序和数据在计算机的内存中都有确定的地址, 且占据一定数目的字节。当你运行一个程序时, 你的变量就具有确定的地址。对于高级语言如 C++ 语言, 你无需清楚地知道每个变量的确切地址。有关地址的处理操作由编译程序和执行系统完成。概念上, 程序中的每个变量名都是一个内存地址的标识符。使用这个标识符来处理数据比处理实际的数据地址要容易的多, 例如: 0F64:01AF4。

新术语: 地址是一个内存中的位置。标识符是变量名。

C++ 语言和 C 语言都是能够进行低级系统编程的语言。事实上, 许多人称 C 语言为

高级汇编语言。低级系统编程需要你经常处理数据的地址。这样就引入指针的概念。知道每个数据的地址,对它的值进行设置和查询。

新术语:指针是一种特殊的变量,是用来存储另一个变量或信息的地址的变量。

警告:指针是一种非常有用的编程工具;同时,它们也是很危险的。编程时,必须谨慎使用指针,因为它们可以中止你的系统。当指针偶尔被赋予一些临界数据或函数的低位存储地址时,这种错误就会出现。

8.7 指向变量的指针

这节中,你将学会如何使用指针来访问变量中的数值。C++语言需要你对指针进行类型定义(包括 void 型)。类型可以是预先定义的类型或用户定义的结构。

语法

指针

定义指针的语法为:

```
type * pointerName;
```

```
type * pointerName=&variable;
```

& 操作符是取地址的操作符,用于得到某个变量的地址。

例子:

```
intPtr;//pointer to an int
```

```
dowble * realPtr;//pointer to a dowble
```

```
char * aString;//pointer to a character
```

```
long lv;
```

```
long * lp=&lv;
```

你也可在指针定义的同一行定义非指针变量。

```
int * intPtr,an Int;
```

```
doUble * realptr,x;
```

```
char * aString,akey;
```

注意:C++语言允许你把 * 号放置于数据类型之后,表示在同一定义行中定义的标识符都被自动地定义成指针:

```
int * intPtr; //pointer to an int
```

```
double * realPtr;//pointer to a dowble
```

```
char * aString;//poointer to a character
```

```
int * realper, * doublePtr; //bloth identifiers are
```

```
//pointers to double
```

在你使用一个指针前,应对它进行初始化。正如你处理一个变量一样。事实上,指针的初始化是必须进行的。使用未经初始化的指针会使程序运行产生意想不到的错误,甚至会中断整个系统。

一旦指针获得某个变量的地址,你就可用 * 加上指针名来访问该变量中的数值。例如,如果 px 是一个指向变量 x 的指针,你就可以用 *px 来访问变量 x 的数值。

用 * 加上指针名可以用来访问地址存于指针中原变量。

不要忘记使用 * 操作符,没有 *, 则语句处理的是指针中的地址,而不是地址中的数据。

下面是一个说明指针如何使用的简单示例。程序列表 8.4 为 POTR1.CPP 的源程序。这个程序的功能是使用变量本身或指针,改变和显示变量中的数值。这个程序没有输入。

程序列表 8.4 程序 PTR1.CPP 的源代码

```
1: /*
2:  C++ program* that demonstrates pointers to existing variables
3:  */
4:
5: #include <iostream.h>
6:
7: main()
8: {
9:     int x = 10;
10:    int* px = &x;
11:    // display x using x and rx
12:    cout << "x contains " << x << "\n";
13:    cout << "x contains (using the pointer px) "
14:         << *px << "\n";
15:    // alter x and display its value using *px
16:    x *= 2;
17:    cout << "x contains (using the pointer px) "
18:         << *px << "\n";
19:    // alter *px and display value using x
20:    *px *= 2;
21:    cout << "x contains " << x << "\n";
22:    return 0;
23: }
```

程序列表 8.4 中程序执行结果如下:

输出:

```
x contains 10
x contains (using the pointer px) 10
x contains (using the pointer px) 20
x contains 40
```

分析

程序列表 8.4 的程序定义了一个 int 类型的变量 x 和一个 int 类型的指针 px。变量 x 初始化为 10,指针 px 初始化为变量 x 的地址。

程序第 12 行的输出语句用变量 x 显示变量 x 中的值。程序的第 13、14 行的输出语句采用指针 px 来显示变量 x 中的数值。注意:用 $*px$ 来访问变量 x 中的数值。

程序第 16 行的语句使变量 x 中的数值翻倍。第 17、18 行中的输出语句采用指针 px 来显示变量 x 中的新值。如输出结果所示,指针能准确地访问变量 x 的当前值。

程序第 20 行语句采用指针 px 使变量 x 中的数值翻倍。注意:在 $=$ 号的左边用 $*px$,用于访问变量 x 的当前值。程序第 21 行的输出语句用变量 x 显示变量 x 的当前值。输出结果再次说明变量 x 和指针 px 是同步的。

8.8 指向数组的指针

C++ 语言和 C 语言都支持数组名的一种特殊用途。编译程序把数组名解释为其第一个元素的地址。这样,如果 x 是一个数组,那么 $\&x[0]$ 与 x 是等价的。又如果 mat 为一个二维数组,那么 $mat[0][0]$ 与 mat 是等价的。这就是 C++ 语言和 C 语言作为高级汇编语言的一面。一旦你得到一个数据项的地址,你就可以访问它的数值。有关变量或数组内存地址的知识,使你可以用指针来处理它的内容。

新术语: 程序变量是一个标识内存地址的标识符,在程序中使用一个变量意味着通过它的变量名来访问相应的内存地址。在这种意义上,变量就是一个指向内存特定位置的名字—指针。

C++ 语言允许你使用指针来访问数组中的不同元素,当你想访问数组 x 的元素 $x[i]$ 时,编译程序完成两种操作。首先,它得到数组的基地址(即数组第一个元素的地址)。第二,它使用设址 i 来计算偏离数组 x 的基地址的偏移量。偏移量等于 i 与数组基本类型的大小:

address of element $x[i] = \text{address of } x + i * \text{sizeof}(\text{basicType})$

看看下面的等式。设想有一个指针 ptr ,取得数组 x 的基地址:

$ptr = x;$ // pointer ptr points to address of $x[0]$

现在,我们可以用 ptr 代替等式中的 x :

address of element $x[i] = ptr + i * \text{sizeof}(\text{basicType})$

C++ 语言和 C 语言为了成为高级汇编语言,它们通过把基本的数组类型的大小看成为 1,从而简化上述等式:

address of element $x[i] = ptr + i$

这个等式说明数组元素 $x[i]$ 的地址为表达式 $(ptr + i)$ 。

让我们通过下面的程序 PTR2.CPP,更好明地说明用指针是如何访问一维数组。这个程序是对程序 ARRAY1.CPP 进行修改而获得的,它完成计算一个数组中元素的平均值的功能,同时再显示该平均值。

程序列表 8.5 程序 PTR2.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of pointer with
3:  one-dimension arrays. Program calculates the average
4:  value of the data found in the array.
5: */
6:
7: #include <iostream.h>
8:
9: const int MAX = 30;
10:
11: main()
12: {
13:
14:     double x[MAX];
15:     // declare pointer and initialize with base
16:     // address of array x
17:     double *realPtr = x; // same as = &x[0]
18:     double sum, sumx = 0.0, mean;
19:     int n;
20:     // obtain the number of data points
21:     do {
22:         cout << "Enter number of data points [2 to
23:             << MAX << "] : ";
24:         cin >> n;
25:         cout << "\n";
26:     } while (n < 2 || n > MAX);
27:
28:     // prompt for the data
29:     for (int i = 0; i < n; i++) {
30:         cout << "X[" << i << "] : ";
31:         // use the form *(x+i) to store data in x[i]
32:         cin >> *(x + i);
33:     }
34:
35:     sum = n;
36:     for (i = 0; i < n; i++)
37:         // use the form *(realPtr + i) to access x[i]
38:         sumx += *(realPtr + i);
39:     mean = sumx / sum;
40:     cout << "\nMean = " << mean << "\n\n";
41:     return 0;
42: }
```

程序列表 8.5 中程序的执行结果如下:

输出:

Enter number of data points [2 to 30]:5

x[0]:1

x[1]:2

x[2]:3

x[3]:4

x[4]:5

Mean=3

分析:

程序列表 8.5 中程序定义了一个 double 型的数组 x, 具有 MAX 个元素。此外, 程序

还定义了指针 `realPtr`, 并对它初始化为数组名 `x`。这样, 指针 `realPtr` 就存储了 `x[0]` 的地址, 即存储的是数组 `x` 的第一个元素的地址。

程序在第 342 行使用指针 `*(x+i)`。这样标识符 `x` 就作为一个指向数组 `x` 的指针。用表达式 `*(x+i)` 来访问数组 `x` 的第 `i` 个元素, 正像使用表达式 `x[i]` 一样。

程序在第 36 至 38 行的 `for` 循环中使用指针 `realptr`。

表达式 `*(realPtr + i)` 等价于 `*(x+i)`, 也等价于 `x[i]`。这样, 使用 `for` 循环使指针 `realPtr` 与偏移量 `i` 相加, 用来访问数组 `x` 的各元素。

8.9 指针递增/递减方法

以前的 C++ 程序中的指针 `realPtr` 中的地址保持不变, 如想访问数组 `x` 的某元素, 就用指针加上 `for` 循环的变址 `i`。我们还可重新写一个程序采用指针递增的方法, 访问数组 `x` 中的元素。C++ 语言为你提供了这种功能, 无需使用确切的偏离值, 允许你顺序地访问数组元素。这种方法就是使指针递增或递减。你仍然需要把指针初始化为数组的基址, 然后用 `++` 运算符访问下一个数组元素。下一个程序中就采用了这种方法。

程序列表 8.6 程序 PTP3.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the use of pointers with
3:  one-dimension arrays. The average value of the array
4:  is calculated. This program modifies the previous version
5:  in the following way: the realPtr is used to access the
6:  array without any help from any loop control variable.
7:  This is accomplished by 'incrementing' the pointer, and
8:  consequently incrementing its address. This program
9:  illustrates pointer arithmetic that alters the pointer's
10: address.
11:
12: */
13:
14: #include <iostream.h>
15:
16: const int MAX = 30;
17:
18: main()
19: {
20:
21:     double x[MAX];
22:     double *realPtr = x;
23:     double sum, sumx = 0.0, mean;
24:     int i, n;
25:
26:     do {
27:         cout << "Enter number of data points [2 to "
28:             << MAX << "] : ";
29:         cin >> n;
30:         cout << "\n";
31:     } while (n < 2 || n > MAX);
32:
33:     // loop variable i is not directly involved in accessing
34:     // the elements of array x
35:     for (i = 0; i < n; i++) {
36:         cout << "X[" << i << "] : ";
37:         // increment pointer realPtr after taking its reference
```

```

38:     cin >> *realPtr++;
39: }
40:
41: // restore original address by using pointer arithmetic
42: realPtr -= n;
43: sum = n;
44: // loop variable i serves as a simple counter
45: for (i = 0; i < n; i++)
46:     // increment pointer realPtr after taking a reference
47:     sumx += *(realPtr++);
48: mean = sumx / sum;
49: cout << "\nMean = " << mean << "\n\n";
50: return 0;
51:
52: }

```

程序列表 8.6 中程序的执行结果如下

输出:

Enter number of data points [2 to 30]:5

x[0]:10

x[1]:20

x[2]:30

x[3]:40

x[4]:50

Mean=30

分析:

程序列表 8.6 中的程序初始化指针 `realPtr` 为数组 `x` 的基地址。程序第 38 行,在键盘输入语句中使用了 `realPtr` 指针。这个语句用 `*realPtr++` 来存储你输入到当前数组元素中的数值,然后指针加 1,指向下一个数组元素。当输入循环终止时,指针 `realPtr` 指向数组 `x` 的尾部。为了把指针重新设置为数组 `x` 的基地址,程序使用了如第 42 行所示语句。这个语句使用指针运算,让指针 `realPtr` 中当前的地址与 `n` 倍的 `sizeof (real)` 相减。这个语句重新设置 `realPtr` 指针中的地址,使它能够访问元素 `x[0]`。在程序的第 47 行,使用同样递增的方法计算数组元素的总和。

8.10 指向结构的指针

C++ 语言支持定义和使用指向结构的指针。使用与简单变量相同的方法,你也可把一个结构变量所占内存段的起始地址赋予一个同类型的指针变量。一旦指针中存有结构变量的地址,它就需要用 `-->` 操作符对结构变量的成员进行访问。

语法

访问结构成员

用一个指针对结构变量的成员进行访问的语法为:

`structPtr->aMember`

例子:

```
struct point{
```

```

    double x;
    double y;
}

point p;
point * ptr = &p;
ptr->x = 23.3;
ptr->y = ptr->x + 12.3;

```

下面就是一个用指向结构的指针的示例程序。程序列表 8.7 是 PTR4.CPP 的源程序。这个程序对 STRUCT1.CPP 程序做了一些改动。程序功能为：请你输入四组坐标，从而定义四个矩形。每个矩形由其左上角点和右下角点的 X,Y 坐标值唯一确定。程序计算每个矩形的面积，并根据面积值对矩形进行排序，同时按顺序在屏幕上显示矩形。

程序列表 8.7 程序 PTR4.CPP 的源程序

```

1: /*
2:  C++ program that demonstrates pointers to structured types
3:  */
4:
5: #include <iostream.h>
6: #include <stdio.h>
7: #include <math.h>
8:
9: const MAX_RECT = 4;
10:
11: struct point {
12:     double x;
13:     double y;
14: };
15:
16: struct rect {
17:     point ulc; // upper left corner
18:     point lrc; // lower right corner
19:     double area;
20:     int id;
21: };
22:
23: typedef rect rectArr[MAX_RECT];
24:
25: main()
26: {
27:     rectArr r;
28:     rect temp;
29:     rect* pr = r;
30:     rect* pr2;
31:     double length, width;
32:
33:     for (int i = 0; i < MAX_RECT; i++, pr++) {
34:         cout << "Enter (X,Y) coord. for ULC of rect. # "
35:              << i << " : ";
36:         cin >> pr->ulc.x >> pr->ulc.y;
37:         cout << "Enter (X,Y) coord. for LRC of rect. # "
38:              << i << " : ";
39:         cin >> pr->lrc.x >> pr->lrc.y;
40:         pr->id = i;
41:         length = fabs(pr->ulc.x - pr->lrc.x);
42:         width = fabs(pr->ulc.y - pr->lrc.y);
43:         pr->area = length * width;
44:     }
45:

```

```

46: pr = MAX_RECT; // reset pointer
47: // sort the rectangles by areas
48: for (i = 0; i < (MAX_RECT - 1); i++, pr++) {
49:     pr2 = pr + 1; // reset pointer pr2
50:     for (int j = i + 1; j < MAX_RECT; j++, pr2++)
51:         if (pr->area > pr2->area) {
52:             temp = *pr;
53:             *pr = *pr2;
54:             *pr2 = temp;
55:         }
56:     }
57:
58: pr = MAX_RECT - 1; // reset pointer
59: // display rectangles sorted by area
60: for (i = 0; i < MAX_RECT; i++, pr++)
61:     printf("Rect # %d has area %5.4lf\n", pr->id, pr->area);
62: return 0;
63: }

```

程序列表 8.7 中程序的执行结果如下:

结果:

```

Enter(x,y)coord. for ULC of rect # 0:1 1
Enter(x,y)coord. for LRC of rect # 0:2 2
Enter(x,y)coord. for ULC of rect # 1:1.5 1.5
Enter(x,y)coord. for LRC of rect # 1:3 4
Enter(x,y)coord. for ULC of rect # 2:1 2
Enter(x,y)coord. for LRC of rect # 2:5 8
Enter(x,y)coord. for ULC of rect # 3:4 6
Enter(x,y)coord. for LRC of rect # 3:8 9
Rect # 0 has area 1.0000
Rect # 1 has area 3.7500
Rect # 2 has area 8.0000
Rect # 3 has area 24.0000

```

分析:

程序列表 8.7 中程序在第 29 和 30 行定义了两个指针变量 `pr` 和 `pr2`。这些指针用来访问 `rect` 类型的结构。程序初始化指针 `pr` 为数组 `r` 的基地址。

在第 33 行的第一个 `for` 循环,用指针 `r` 使输入值存入数组 `r` 的元素中,循环增量表达式为 `pr++`。通过指针运算,使指针指向数组 `r` 的下一个元素。在程序第 36 至第 39 行的输入语句使用指针 `pr` 来访问成员 `ulc` 和 `lrc`。注意:该语句通过指针加上一>>操作符来访问成员 `ulc` 和 `lrc`。程序第 40 至第 42 行的语句也用指针 `pr` 加上一>>操作符来访问成员 `id`,`ulc`,`lrc` 和 `area`。

程序第 46 行的语句用 `MAX_RECT` 对指针 `pr` 重新设置(`MAX_RECT` 为 `* sizeof (double)` 个字节)。程序第 48 行至第 56 行的嵌套循环用到指针 `pr` 和 `pr2`。最外层的 `for` 循环在下次循环之前,使指针中的地址加 1。第 49 行的语句把 `pr+1` 赋予指针 `pr2`,这条语句对指针 `pr2` 进行初始化,可对数组 `r` 的第 `i+1` 个元素访问。内层的 `for` 循环在下次循环前使 `pr` 加 1。这样内层的 `for` 循环通过用 `pr` 和 `pr2` 指针访问数组 `r` 的元素。程序第 51 行的 `if` 语句用指针 `pr` 和 `pr2` 访问成员 `area`,用于比较不同矩形

的面积。第 52 至 54 行中的语句也是用指针 `pr` 和 `pr2` 交换数组 `r` 中的元素。注意, 语句用 `*pr` 和 `*pr2` 来访问数组 `r` 的全部元素。

程序第 38 行的语句通过与 `MAX_RECT-1` 的差运算对指针 `pr` 中的地址重新设置。最后一个 `for` 循环也用到指针 `pr`, 用它访问并显示数组 `r` 中不同元素的成员 `id` 和 `area`。

这个程序充分说明仅用指针变量就可对一个数组进行各种操作。指针是非常有力, 灵活的编程工具。

8.11 指针和动态存储器

程序在编译的时候为它的变量生成一定存储单元。当程序开始运行时, 这些变量存于它们预先分配的存储空间, 在程序运行过程中, 当你需要生成新的变量时, 就会用到上述技术。在程序运行时, 你需要为这些新的变量动态分配存储单元。C++ 语言的设计者于是就引入一些 C 语言所没有的新的操作符, 用来处理动态分配及重新分配存储单元。这些新的 C++ 语言操作符是 `new` 和 `delete`。而且 C 语言动态分配存储单元的函数 `malloc`, `calloc` 和 `free` 仍然可以使用。但当操作符 `new` 和 `delete` 用于处理生成的动态数据类型时, 它们比函数 `malloc`, `calloc` 和 `free` 更加方便。

语法

new 和 delete 操作符

在生成动态标量变量的过程中, 使用 `new` 和 `delete` 操作符的语法是

```
pointer=new type;
```

```
delete pointer;
```

操作符 `new` 返回被动态分配给内存单元的变量地址, 操作符 `delete` 通过指针释放被动态分配的存储单元。如果动态分配操作符 `new` 执行不成功, 则它返回一个 `NULL` (等于 0) 指针。因而你需要在使用 `new` 操作符之后, 测试返回值是否为 `NULL` 指针。

例子:

```
int *pint;
pint=new int;
*pint=33
cout<<"Pointer pint stores"<<*pint;
delete pint;
```

语法

动态数组

对一个动态数组分配和重新分配存储单元的语法是

```
arrayPointer=new type[arraySize];
```

```
delete [ ] arrayPointer;
```

操作符 `new` 返回被动态分配给存储单元的数组的地址。如果失败, 则返回 `NULL` 指

针。操作符通过使用指针释放动态分配给数组的存储单元。

例子:

```
const int MAX=10;
int * pint;
pint=new int[MAX]
for(int i=0; i<MAX;i++)
    * pint[i]=i*i;
for (i=0;i<MAX;i++)
    cout<< * u(pint+i)<<"\n";
delete [] pint;
```

DO	DON'T
DO 要做到能随时访问动态变量和动态数组。这样的访问不需要对指针初始化。如下示例:	
<pre>int * p=new int; int * q; * p=123; q=p; //q now also point to 123 p=new int; //creat another dynamic variable * p=345; // p point to 345 uhcreas q points to 123 cout<< * p<<" "<< * q<<" "<<(* p+ * q)<<"\n"; delete p; delete q;</pre>	
DON'T 不要忘记在动态变量和动态数组完成其作用时,释放动态分配的存储单元。	

使用指针生成并访问动态数据的方法在下面程序中得以充分说明。程序列表 8.8 为 PTRS. CPP 的源程序。这个程序是对程序 ARRAYL. CPP 做了一些改动。程序功能是:计算数组元素的平均值。程序开始时请你输入数组元素值,并检查你输入的值是否有效。其次,程序计算数组元素的平均值,并显示该平均值。

程序列表 8.8 程序 PTR5. CPP 的源代码

```
1: /*
2:  C++ program that demonstrates the pointers to manage
3:  dynamic data
4:  */
5:
6: #include <iostream.h>
7:
8: const int MAX = 30;
9:
10: main()
11: {
12:
```

```

13:     double* x;
14:     double sum, sumx = 0, mean;
15:     int *n;
16:
17:     n = new int;
18:     if (n == NULL)
19:         return 1;
20:
21:     do { // obtain number of data points
22:         cout << "Enter number of data points [2 to "
23:             << MAX << "] : ";
24:         cin >> *n;
25:         cout << "\n";
26:     } while (*n < 2 || *n > MAX);
27:     // create tailor-fit dynamic array
28:     x = new double[*n];
29:     if (!x) {
30:         delete n;
31:         return 1;
32:     }
33:     // prompt user for data
34:     for (int i = 0; i < *n; i++) {
35:         cout << "X[" << i << "] : ";
36:         cin >> x[i];
37:     }
38:
39:     // initialize summations
40:     sum = *n;
41:     // calculate sum of observations
42:     for (i = 0; i < *n; i++)
43:         sumx += *(x + i);
44:
45:     mean = sumx / sum; // calculate the mean value
46:     cout << "\nMean = " << mean << "\n\n";
47:     // deallocate dynamic memory
48:     delete n;
49:     delete [] x;
50:     return 0;
51: }

```

程序列表 8.8 中程序执行结果如下：

输出：

Enter number of data points [2 to 30]:5

x[0]:1

x[1]:2

x[2]:3

x[3]:4

x[4]:5

mean=3

分析：

程序列表 8.8 中和程序定义两个指针用于动态分配。程序第 13 行定义了第一个指针，它用于对动态数组存储单元的分配和访问。程序第 15 行定义了第二个指针用于生成动态变量。

程序第 17 行中的语句，使用操作符 new 以动态 int 型变量的存储单元。该语句返回

动态数据的地址到指针中,第18行中的if语句判断分配过程是否失败。如果失败,则主函数main中止,并返回终止代码1(错误标志)。

程序第21行中的do-while循环请你输入数组元素的数目。第24行的语句用指针,把输入的数据存入动态变量中。该语句用*n对动态变量访问。while从句也是用*n访问动态变量中的数据。事实上,程序的所有语句都是用*n对数据进行访问。

程序第28行中的语句用new操作符生成一个动态数组。该语句生成一个动态double型的数组,数组大小你已经规定了。这一特点说明了使用动态分配存储单元来生成特定大小的数组的优越性。程序第26行中的if语句判断动态分配存储单元的过程是否成功。如果失败,则第30与31行的语句将使用指针n重新分配动态变量的存储单元,且函数终止,返回值1。

程序第34至37行使用for循环,请你输入动态数组的元素值,第36行的语句把你输入的数值存于数组的第i个元素中。注意,该语句用表达式x[i]访问目标元素,这种形式类似于静态数组元素的访问。C++语言认为表达式x[i]等价于*(x+i)。事实上,程序在第42和43行的第二个for循环中就用第二种表达形式。第43行的语句用*(x+i)访问动态数组的元素。

main的最后一条语句就是释放动态变量和动态数组的存储单元。第48行的语句用指针n释放动态变量的存储单元,第49行的语句释放了动态数组的存储单元。

8.12 远指针

处理器的结构(例如Intel80x86)是使用分段存储器。每段为64K,使用段既有许多优点,也有许多缺点。这种设计支持两种指针:近指针和远指针。

新术语:近指针是访问同一段内数据的指针。这种指针仅存储段内的偏移地址,因此只需较少的字节存储地址。

远指针存储段地址和偏移地址。这样就需要较多的字节存储地址。Windows的许多操作都使用远指针。

定义远指针只需在指针类型和指针名之间插入关键字far或_far即可。

8.13 小结

这章主要介绍了用户定义的数据类型。主要内容如下:

☐ 你可用typedef语句定义一个已存在类型的等价类型,并能定义数组类型。使用typedef的语法为:

```
typedef knowType newType;
```

☐ 枚举类型允许你定义唯一的标识符来代表一组有关变量的集合。定义枚举类型的语法为:

```
enum enumType{(list of enumerated identifiers)}
```


- ☐ 结构允许你将不同类型的数据有机地组合在一起,它的成员可以是事先定义的类型或其它结构。定义结构的语法为:

```
struct structTay{(list of members)};
```

- ☐ 联合是一种特殊形式的结构。定义联合的语法为:

```
union unionTay{  
    type1 member1;  
    type2 member2;  
    ...  
    typeN memberN;  
}
```

- ☐ 引用变量是引用它们的变量的等价物。定义一个引用变量只需在引用变量的类型后或变量名前键入 &。
- ☐ 指针是存储变量或数据地址的变量。C++ 语言在处理数据和系统中使用指针,使 C++ 语言编程更加灵活、方便。
- ☐ 指向变量的指针用 & 操作符获得变量的地址。通过这些地址,指针可以对有关变量中的数据进行访问。用指针访问变量,只需在指针名前加上 * 操作符。
- ☐ 指向数组的指针可通过数组的基地址对数组元素访问。C++ 语言认为数组名等价于指针中的数组基址。例如,数组 x 名等价于 &x[0]。指针也可用递增或递减的方法顺序访问数组元素。
- ☐ 指向结构的指针用于处理结构及它们的成员。C++ 语言提供 -> 操作符,允许一个指针使用它对结构成员访问。
- ☐ 指针可以用操作符 new 和 delete 对动态数据分配存储单元,并访问这些动态数据。这些操作符也允许用于生成动态变量和动态数组。操作符 new 把动态分配给数据的存储单元的地址赋予一个指针。操作符 delete 当上述数据信息不再需要时,释放分配给它们的存储单元。
- ☐ 远指针是一种存储段地址和偏移地址的指针,近指针仅存储偏移地址。远指针需要的字节较多。

8.14 问与答

问:C++ 语言支持指向 void 类型的指针吗?

答:支持。void 指针被认为是无类型指针,用于拷贝数据。

问:因为 C++ 指针属于某种类型,是否能用 void 型指针将数据传递给非 void 型指针?

答:可以。C++ 语言支持这种数据传递。例如:

```
void * p=data;  
long * lp=(long *)p;
```

指针 lp 通过将指针 p 强制转换为类型 long 进行数据传递。

问:当用 delete 操作符释放一个动态数组的存储单元时,后面不加 [] 时,会发生什么事情?

答:运行时,系统仅释放动态数组的第一个元素的存储单元。其它数组元素仍在内存中。

问:结构成员可以是指向该结构的指针吗?

答:可以。许多结构都采用这种形式。例如,下面的结构:

```
struct listNode{
    dataType data;
    listNode * next;
}
```

问:C++语言允许在定义结构之前,定义指向该结构的指针吗?

答:可以。这一特点使动态数据结构的定义成为可能。

问:C++语言允许指针访问其它指针的地址吗?

答:允许。C++语言支持指向指针的指针(也称为 double pointers)。定义这样的指针只需用两个 * 即可。如下面示例,定义了双重指针 p:

```
int x;
int * px=&x;
int * * p=&px;
```

表达式 &p 可访问指针 px,表达式 * * p 可访问变量 x。

8.15 专题讨论

专题讨论列出几个测验的问题帮助你巩固你在这章中学到的内容。此外还附加一些练习,便于你在编程过程中更好地理解所学内容。在继续下章的学习之前,尽量理解测验和练习的答案。答案在附录 B 中给出。

8.15.1 测验

1. 下列语句有何错误?

```
enum Boolean {false,true};
enum State {on,off};
enum YesNo {yes,no};
enum DiskDriveStatus {on,off};
```

2. 下面枚举类型的定义正确吗?

```
enum YesNo(no=0, No=0, yes=1; Yes=1);
```

3. 下面程序有何问题?

```
#include(iostream. h)
main( )
{
    int * p=new int;
    cout<<"Enter a number:"
    cin>> * p;
    cout<<"The square of"<< * p<<"="<<(* p * * p);
    return 0;
```

}

8.15.2 练习

1. 修改程序 PTR4.CPP 以建立程序 PTR6.CPP, 使该程序用 COMB 排序法对矩形数组排序。
2. 定义一个结构用于动态整型数组模型。这个结构的成员可以访问动态数据并能存储动态数组的大小, 结构名为 `intArrStruct`。
3. 定义一个结构用于动态二维数组模型。该结构的一个成员用于访问动态数据, 另两个成员用于存储数组行、列的大小。结构名为 `matStruct`。

第九章 字符串(第九天)

第一章至第八章中的所有程序示例主要是关于数值及少量字符的操作。你可能会奇怪,为什么所有的程序示例中都没有字符串呢?在这章中,我们着重讨论 C++ 语言的字符串。本章的主要内容如下:

- ☐ C++ 语言的字符串
- ☐ 字符串的输入
- ☐ 标准串库的使用
- ☐ 字符串的赋值
- ☐ 字符串长度的获得
- ☐ 字符串的连接
- ☐ 字符串的比较
- ☐ 字符串的转换
- ☐ 字符串的倒序
- ☐ 查找字符
- ☐ 查找子字符串

9.1 C++ 语言的字符串

C++ 语言(和 C 语言)都没有预先定义的字符串类型。但是,C++ 语言和 C 语言一样认为字符串是字符型数组。在每个字符串的结尾是一个 ASCII 为 0 的空操作字符(\0)

新术语: '\0' 又称为空结束符。结尾带有空结束符的串,有时称为 ASCIIZ 串。字符 Z 代表 ASCII 码为 0 的字符,就是空结束符。

空结束符在所有串的尾都应该有,并可以根据空结束符度量串的长度。当你定义一个串变量为字符型数组时,必须保留一个额外的字节存结束符\0。使用空结束符的优点在于当你生成串时,不必考虑 C++ 语言的一些限制。此外,ASCIIZ 串具有很简单的结构。

注意:在第八章中,我们讨论了如何使用指针访问和处理数组元素。C++ 和 C 语言把这上特点也扩展到字符串的处理中。

DO	DON'T
DO 确定一个串的大小时,应包括用于空结束符的额外字节。	
DON'T 不要把串变量定义为一个单字符的数组,这样的串变量无效。	

9.2 字符串的输入

从先前介绍的程序中可知,用输出流语句可以显示字符串,C++语言支持这种情况:用输出流语句显示未事先定义数据类型的串。使用串变量进行串的输出也用到同样的操作符,并遵循相同的语法。当串输入时,插入操作符>>不能正常操作。因为串中通常包含由插入操作符忽略的空格,不用插入操作符,你可以用 `getline` 函数代替插入操作符。这个函数功能是读特定数目的字符。

语法

getline 函数

getline 函数使用语法是

```
istream&. getline(signed char * buffer,
                  int size,char delimiter='\n');
istream &. getline (unsigned char * buffer,
                   int size,char delimiter='\n');
```

参数 `buffer` 是一个指向串的指针。用于从流中接收字符。参数 `size` 规定读入字符的最大数目。参数 `delimiter` 指定定界字符。该字符可使串的输入在未达到参数 `size` 规定的字符数目时,停止。参数 `delimiter` 具有缺省变元“\n”

例子:

```
#include<iostream. h>
main ()
{
char name[80];
conll<<"Enter your name:"
cin. getline(name,sizeof (name)-1);
cout<<"Hello"<<name<<" ,how are you" ;
return 0;
}
```

9.3 使用 STRING. H 库

C 程序员已开发了标准串库 `STRING. H`。它包含了许多最常用的字符串处理函数。头文件 `STDIO. H` 和 `Iostream. H` 也支持字符串的输入/输出(I/O)。不同的 C++ 编译器的销售商也开发了 C++ 形式的字符串库,这些库对模型字符串使用类(class),(在

第十一章中,你将学到有关类的内容)。然而,这些串库不是标准的,虽然 `STRING.H` 中的 C 语言形式的字符串程序为 ANSI C 标准的一部分。在下节中,我们将介绍一些在 `STRING.H` 头文件中的原型化字符串的函数。

9.4 赋值串

C++ 语言支持两种对串赋值的方法。当你对串初始化时,你可把一个串赋值 给一个串变量。这种方法很简单,需要使用“=”操作符。

语法

字符串的初始化

字符串初始化的语法是

```
Char stringVar[stringsize]=stringLiteral;
```

例子:

```
char aString[81]="Turbo C++ in 21 days";
```

```
char name []="Namir Shammas";
```

第二种方法是用函数 `strcpy` 赋值一个 ASCII 串。该函数在串的结尾自动加上 `\0` 空字符。

语法

函数 `strcpy`

函数 `strcpy` 的原型是

```
char * (char * target,const char * source)
```

这个函数把原串 `source` 中的字符拷贝到目的串 `target` 中。函数假设目的串有足够的存储单元接受源串。

例子:

```
char name[41];
```

```
strcpy (name,"Turbo C++");
```

变量 `name` 中包含串 "Turbo C++"。

函数 `strdup` 允许你拷贝字符到另一个串中,并放置在目的串中所规定的位置。

语法

函数 `strdup`

函数 `strdup` 原型是

```
char * strdup(const char * source)
```

该函数拷贝源串中的字符,并返回一个指向复制串的指针。

例子:

```
char * string1="The reign in Spain";
```

```
char * string2;
```

```
string2=strdup(string1)
```

这个例子是在给 string2 分配存储单元后,把 string1 的内容复制到 string2 中。
串库也提供函数 strncpy,完成从一个串中拷贝特定数目的字符到另一个串中。

语法

函数 strncpy

函数 strncpy 的原型是

```
char * strncpy(char * target,const char * source,size - tnum);
```

该函数拷贝 num 个字符从源串到目的串中。如果必要,该函数可截断多余的字符或填充空白符。

例子:

```
char str[1]="Pascal";
```

```
char str[2]="Hello there";
```

```
strncpy(str1,str2,6);
```

变量 str1 现在包含串"Hello"。

注意:用指针处理串对于许多 C++ 编程的初学者而言,是很新的内容。事实上,你可通过把串的第一个字符的地址赋值给指针,用指针处理串的尾部。例如:

```
char str1[41]="Hello World";
```

```
char str2[41];
```

```
char * p=sStr1;
```

```
pt=6; //p now points to substring "World" in
```

```
str strcpy (str2,p);
```

```
cout<<str2<<"\n";
```

输出语句显示串"World"。这个示例说明如何使用指针将数据区中的字符合并在一起。

9.5 字符串的长度

许多字符串的操作需要串中字符的数目。STRING.H 库提供了函数 strlen 该函数,返回串中字符的数目,包括结束作符。

语法

函数 strlen

函数 strlen 的原型是

结果 size_t 的类型为整型。

例子:

```
char str[ ]="1234567890";
```

```
unsigned i;
```

```
i=strlen(str);
```

这些语句把 10 赋值给变量 i。

9.6 字符串的连接

通常,你可通过连接两个或多个串而生成一个新串。函数 `strcat` 允许你把两个串连接在一起。

新术语:连接就是把串连接在一起。

语法

函数 `strcat`

函数 `strcat` 的原型是

`char * strcat (char * target, const char * source)`

该函数在目的串的后面追加源串,并返回指向目的串的指针。函数假定目的串能容纳源串的字符。

例子:

```
char string[81];
strcpy(string, "Turbo");
strcat(string, "C++");
```

变量 `string` 中的内容此时为 "Turbo C++"。

函数 `strncat` 可把源串中特定数目的字符连接在目的串之后。

语法

函数 `strncat`

函数 `strncat` 的原型是

`char * strncat (char * target, const char * source, size_t num)`

该函数把源串的 `num` 个字符追加到目的串后,并返回指向目的串的指针。

例子:

```
char str1[81]="Hello I am";
char str2[41]="Thomas Jones";
strncat (str1,str2,6);
```

变量 `Str1` 中的内容为 "Hello I am Thomas"。

DO	DON'T
DO 当你不能确定目的串的容量时,使用函数 <code>strncat</code> 控制连接在目的串之后的字符数目。	
DON'T 不要认为目的串必定有足够的存储单元,存储源串。	

下面这个程序用到函数 `getline`, `strlen`, `strcat`。程序列表 9.1 为 `STRING1.CPP` 的源程序,程序执行以下功能:

- ☐ 请你输入一个串,你输入的字符数目不应超过 40。
- ☐ 请你输入第二个串,你输入的字符数目不应超过 40。
- ☐ 显示你输入的两个串的大小长度。
- ☐ 把第二个串与第一个串连接在一起。
- ☐ 显示连接后的串。
- ☐ 显示连接后生成的新串长度。
- ☐ 请你输入一个查找的字符。
- ☐ 请你输入一个准备更换的字符。
- ☐ 显示字符更换后连接生成的串。

程序列表 9.1 程序 `STRING1.CPP` 的源代码

```
1:  /*
2:   C++ program that demonstrates C-style strings
3:   */
4:
5:  #include <iostream.h>
6:  #include <string.h>
7:
8:  const unsigned MAX1 = 40;
9:  const unsigned MAX2 = 80;
10:
11:  main()
12:  {
13:
14:      char smallStr[MAX1+1];
15:      char bigStr[MAX2+1];
16:      char findChar, replChar;
17:
18:      cout << "Enter first string:\n";
19:      cin.getline(bigStr, MAX2);
20:      cout << "Enter second string:\n";
21:      cin.getline(smallStr, MAX1);
22:      cout << "String 1 has " << strlen(bigStr)
23:           << " characters\n";
24:      cout << "String 2 has " << strlen(smallStr)
25:           << " characters\n";
26:      // concatenate bigStr to smallStr
27:      strcat(bigStr, smallStr);
28:      cout << "Concatenated strings are:\n"
29:           << bigStr << "\n";
30:      cout << "New string has " << strlen(bigStr)
31:           << " characters\n";
32:      // get the search and replacement characters
33:      cout << "Enter search character : ";
34:      cin >> findChar;
35:      cout << "Enter replacement character : ";
36:      cin >> replChar;
37:      // replace characters in string bigStr
38:      for (unsigned i = 0; i < strlen(bigStr); i++)
39:          if (bigStr[i] == findChar)
40:              bigStr[i] = replChar;
41:      // display the updated string bigStr
42:      cout << "New string is:\n"
```

```

43:         << bigStr;
44:     return 0;
45: }

```

程序列表 9.1 中程序执行结果如下:

输出:

```

Enter first string:
he rain in Spain stays.
Enter second string:
ainly in the plain.
string1 has 23 characters
string2 has 20 characters
Concatenated strings are:
The rain in spain stays mainly in the plain
New string has 43 character Enter search character;a
Enter replacenet characeer ;A
New string is :
The rAin in SpAin stAys mAinly in the plain.

```

分析:

程序列表 9.1 中程序包含头文件 `STRING.H` 允许使用串操作函数。程序第 8 行至第 9 行定义全局变量 `MAX1` 和 `MAX2`, 分别规定小串和大串的长度。主函数 `main` 中定义了两个串, `smallStr` 和 `bigStr`, 程序第 14 行定义变量 `smallStr` 存储 `MAX1+1` 个字符(附加的存储单元用于存储空字符)。程序第 15 行定义变量 `bigStr`, 存储 `MAX2+1` 个字符。第 16 行定义了 `char` 型的变量 `findChar` 和 `replChar`。

程序第 18 行中的输出语句请你输入第一个串。第 19 行的语句用流输入函数 `getline` 获得你输入的串, 并将它存在变量 `bigStr` 中。函数调用规定你最多能输入 `MAX2` 个字符。第 20 行的输出语句提示请你输入第二个串, 第 21 行的语句用流输入函数 `getline` 获得你输入的串, 并将它存入变量 `smallStr` 串中。函数调用规定你最多能输入 `MAX1` 个字符。

程序第 22 行至第 25 行的输出语句显示变量 `bigStr` 和 `smallStr` 中字符的数目。两个输出语句都调用了函数 `strlen`。

程序第 27 行中的语句把变量 `smallStr` 中的串连接到变量 `bigStr` 的串后。第 28 至 29 行的输出语句显示 `bigStr` 中的当前串。第 30 和 31 行的输出语句显示当前 `bigStr` 串中字符的数目。这个语句同样也是调用函数 `strlen` 获得字符的数目。

程序第 33 行中的语句提示请你输入需查找的字符。第 34 行的语句获得你输入的字符。并将它存入变量 `findChar` 中。第 35 行的语句提示请你输入准备替换的字符。第 36 行的语句把你输入的字符存于 `replChar` 中。

程序第 38 至 40 行的 `for` 循环用于改变串中需要修改的字符。该循环由变量 `i` 控制, 从 0 到 `strlen(bigStr)-1`, 每次循环增量为 1。第 39 行的 `if` 语句判断是否 `bigStr` 串中的第 `i` 个字符与变量 `findChar` 中字符相同。如果相同, 程序执行第 40 行中的语句。该语

句将变 bigStr 串中第 i 个字符换成变量 replChor 中的字符。这个循环说明如何通过访问串中每个字符来处理串变量中的内容。

第 42 行和 43 行的输出语句显示串 bigStr 中的当前内容。

9.7 字符串的比较

因为字符串是字符数组,所以 STRING.H 库提供一组函数用于串的比较。这些函数根据字符的 ASCII 码值比较两个串中的字符。这组函数为 strcmp, strcmpi, strncmp 和 strnicmp。

函数 strcmp 是对两个字符串从左至右逐个字符相比较,直到出现不同的字符或遇到 \0 为止。

语法

函数 strcmp

函数 strcmp 的原型是

```
int strcmp (const char * str1, const char * str2);
```

函数比较串 1 和串 2,结果如下:

<0 when str1 is less than str2

=0 when str1 is equal to str2

>0 when str1 is grenter than str2

例子:

```
char string1[1]="Turbo C++";
```

```
char string2[ ]="TURBO C++";
```

```
int i
```

```
i=strcmp (string1,string2);
```

最后的语句把一正整数赋值给变量 i,因为变量 string1 中的串小于变量 string2 中的串。

函数 Stricmp 也对两个字符串自左向右逐个字符相比(按 ASCII 值大小比较),直到出现不同的字符或遇到 \0 为止。

语法

函数 strcmpi

函数 strcmpi 的原型是

```
int strcmpi(const char * str1,const char * str2);
```

函数比较 str1 和 str2,并且认为大、小写字符相同,结果的函数值如下:

<0 when str1 is less then str2

=0 when str1 equal to str2

>0 when str1 is greater than str2

例子:

```
char string1[ ]="Turbo C++";
```

```
char string2[ ]="TURBO C++";
```

```
int i;
```

```
i=stricmp (string1,String2);
```

最后的语句把一正整数赋值给变量 i, 因为变量 string1 中的串小于变量 string2 中的串只在形式上不同。

函数 strcmp 根据所规定需要比较的字符个数, 对两个串中的字符, 自左向右进行逐个比较。

语法

函数 strcmp:

函数 strcmp 的原型是

```
int strcmp (const char * str1,const char * str2,size _t num);
```

函数对 str1 与 str2 的前 num 个字符比较。表示比较结果的函数值如下:

<0 when str1 is less than str2

=0 when str1 is equal to str2

>0 when str1 is greater than str2

例子:

```
char string1[ ]="Turbo C++";
```

```
char string2[ ]="Turbo Pascal";
```

```
int i;
```

```
i=strncmp (string1,string2,7);
```

函数把一负整数赋值给变量 i, 因为 "Turbo C" 小于 "Turbo P"。

函数 strncmp, 根据规定需要比较的字符个数, 对两个串中字符自左向右进行比较。

该函数认为大小写字符相同。

语法

函数 strnicmp

函数 strnicmp 的原型是

```
int strnicmp (const char * Str1,const char * Str2,size _t num)
```

函数对 str1 和 str2 的前 num 个字符比较, 且忽略字符的大小写的区别。表示比较结果的函数值如下:

<0 when str1 is less than str2

=0 when str1 is equal to str2

>0 when str1 is greater than str2

例子:

```
char string1[ ]="Turbo C++";
```

```
char string2[ ]="TURBO Pascal";
```

```
int i;
```

```
i=strnicmp(string1, string2, 5);
```

该函数把值 0 赋值给变量 i, 因为 "Turob" 和 "TURBO" 仅在书写形式上不同。

让我们看看下面一个串比较的例子。程序列表 9.2 中程序的功能为: 生成一个串数

组,并对它初始化。然后程序显示这个无序的串数组。对数组中串排序后,再显示排序后的串组。

程序列表 9.2 程序 STRING2.CPP 的源代码

```
1: /*
2:  C++ program that demonstrates comparing strings
3:  */
4:
5: #include <iostream.h>
6: #include <string.h>
7:
8: const unsigned STR_SIZE = 40;
9: const unsigned ARRAY_SIZE = 11;
10: const int TRUE = 1;
11: const int FALSE = 0;
12:
13: main()
14: {
15:
16:     char strArr[STR_SIZE][ARRAY_SIZE] =
17:         { "California", "Virginia", "Alaska", "New York",
18:           "Michigan", "Nevada", "Ohio", "Florida",
19:           "Washington", "Oregon", "Arizona" };
20:     char temp[STR_SIZE];
21:     unsigned n = ARRAY_SIZE;
22:     unsigned offset;
23:     int inOrder;
24:
25:     cout << "Unordered array of strings is:\n";
26:     for (unsigned i = 0; i < ARRAY_SIZE; i++)
27:         cout << strArr[i] << "\n";
28:
29:     cout << "\nEnter a non-space character and press Enter";
30:     cin >> temp[0];
31:     cout << "\n";
32:
33:     offset = n;
34:     do {
35:         offset = (8 * offset) / 11;
36:         offset = (offset == 0) ? 1 : offset;
37:         inOrder = TRUE;
38:         for (unsigned i = 0, j = offset;
39:              i < n - offset; i++, j++)
40:             if (strcmp(strArr[i], strArr[j]) > 0) {
41:                 strcpy(temp, strArr[i]);
42:                 strcpy(strArr[i], strArr[j]);
43:                 strcpy(strArr[j], temp);
44:                 inOrder = FALSE;
45:             }
46:     } while (! (offset == 1 && inOrder));
47:
48:     cout << "Sorted array of strings is:\n";
49:     for (i = 0; i < ARRAY_SIZE; i++)
50:         cout << strArr[i] << "\n";
51:     return 0;
52: }
```

程序列表 9.2 中程序的执行结果如下:

输出:

Unordered array of strings is ;

California
Virginia
Alaska
New York
Michigan
Nevada
Ohio
Florida
Washington
Oregon
Arizona

Enter a non-space character and press Enter

Sorted array of strings is:

Alaska
Arizona
California
Florida
Michigan
Nevada
New York
Ohio
Oregon
Virginia
Washington

分析:

程序列表 9.2 中程序定义了全局变量 STR_SIZE, ARRAY_SIZE, TRUE 和 FALSE。变量 STR_SIZE 规定每个串的长度。变量 ARRAY_SIZE 规定数组中串的数目。变量 TRUE 和 FALSE 代表 Boolean 值,在数组的排序中用到。主函数 main 定义一个数组 strArr1 实际 strArr 是一个二维字符数组。该数组 ARRAY_SIZE 个元素,且每个元素有 STR_SIZE 个字符。注意,这种定义说明了串的长度和数组的大小,主函数也对数组 strArr 进行初始化。此外,函数还定义了变量 temp,用于交换存储单元的内容。第 21 至 23 行也定义了许多不同类型的变量。

程序第 25 行的输出语句显示提示行,第 26 和 27 行的 for 循环显示无序数组 strArr 中的元素。这个 for 循环用变量 i 进行控制,从 0 变换到 ARRAY_SIZE-1,每次增量为 1。第 27 行的输出语句,用表达式 strArr[i] 显示第 i 个数组元素中的串。

程序第 33 至 46 行采用了 Comb 排序法。注意,如果第 40 行的语句用函数 strcmp 来比较数组的第 i 个和第 j 个元素,应分别用表达式 strArr[i] 和 strArr[j]。第 41 至 43 行的语句用函数 strcpy 和变量 temp 交换数组的第 i 和第 j 个元素。

程序第 48 行的输出语句显示提示行,说明下面显示的是排序后的数组。第 49 和 50 行的 for 循环显示数组 strArr 中的元素。该循环用变量 i 进行控制,从 0 变换到 AR-

RAY_SIZE-1,每次循环增量为1。第50行的输出语句用表达式 `strArr[i]` 显示数组第*i*个元素中的串。

9.8 变换字符串

STRING.H 库提供函数 `_strlwr` 和 `_strupr`, 分别把串中的字符转换成大写或小写的形式。注意,C 语言书中, 这些函数通常称为 `strlwr` 和 `strupr`。

语法

函数 `_strlwr`

函数 `_strlwr` 的原型是

```
char * _strlwr(char * source);
```

函数 `_strlwr` 把串 `source` 中的大写字母转换成小写字母, 其它字符没有影响。该函数返回一个指向串 `source` 的指针。

例子:

```
char str[ ]="HELLO THERE";
```

```
_strlwr(str);
```

变量 `str` 中的内容为串 "hello there"。

语法

函数 `_strupr`

函数 `_strupr` 的语法:

```
char * _strupr(char * source);
```

函数 `_strupr` 把串 `source` 中的小写字母转换为大写字母, 其它字符没有影响, 该函数返回一个指向串 `source` 的指针。

例子:

```
char str[ ]="Turbo C++";
```

```
_strupr(str);
```

变量 `str` 中的内容为当前串 "TURBO C++"。

DO	DON'T
DO 如果在程序的后面需要用到原串, 则在调用函数 <code>_Strlwr</code> 和 <code>_strupr</code> 前对原串拷贝。	
DON'T 不要认为在调用函数 <code>_strlwr</code> 之后再调用函数 <code>_strupr</code> (或顺序相反), 能得到存储在变量中的源串。	

9.9 倒序字符串

STRING.H 库提供函数 `STRREN`, 把串中的字符顺序颠倒。

语法

函数 strrev

函数 strrev 的原型是

```
char * strrev(char * str);
```

该函数颠倒串 str 中的字符顺序,并返回一指针指向串 str。

例子:

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

```
strrev(string);
```

```
cout<<string;
```

This displays "olleH".

下面让我们看一个处理串中字符的程序。程序列表 9.3 为 STRING3.CPP 的源程序,该程序完成下面几个功能:

- ☐ 请你输入一个串。
- ☐ 显示你的输入。
- ☐ 显示你的输入串的小写形式。
- ☐ 按与你所输入相反顺序显示串。
- ☐ 如果你输入的串中无大写字母,显示提示信息。
- ☐ 如果你输入的串中无小写字母,显示提示信息。
- ☐ 如果你输入的串中的字符对称,显示提示信息。

程序列表 9.3 程序 STRING3.CPP 的源程序

```
1:  /*
2:   C++ program that demonstrates manipulating the
3:   characters in a string
4:   */
5:
6:   #include <iostream.h>
7:   #include <string.h>
8:
9:   const unsigned STR_SIZE = 40;
10:  const int TRUE = 1;
11:  const int FALSE = 0;
12:
13:  main()
14:  {
15:      char str1{STR_SIZE+1};
16:      char str2{STR_SIZE+1};
17:      int isLowerCase;
18:      int isUpperCase;
19:      int isSymmetrical;
20:
21:
22:      cout << "Enter a string : ";
23:      cin.getline(str1, STR_SIZE);
24:      cout << "Input: " << str1 << "\n";
25:      // copy str1 to str2
26:      strcpy(str2, str1);
27:      // convert to lowercase
28:      strlwr(str2);
29:      isLowerCase = (strcmp(str1, str2) == 0) ? TRUE : FALSE;
30:      cout << "Lowercase: " << str2 << "\n";
```



```

31:    // convert to uppercase
32:    strupr(str2);
33:    isUpperCase = (strcmp(str1, str2) == 0) ? TRUE : FALSE;
34:    cout << "Uppercase: " << str2 << "\n";
35:    // copy str1 to str2
36:    strcpy(str2, str1);
37:    // reverse characters
38:    strrev(str2);
39:    isSymmetrical = (strcmp(str1, str2) == 0) ? TRUE : FALSE;
40:    cout << "Reversed: " << str2 << "\n";
41:    if (isLowerCase)
42:        cout << "Your input has no uppercase letters\n";
43:    if (isUpperCase)
44:        cout << "Your input has no lowercase letters\n";
45:    if (isSymmetrical)
46:        cout << "Your input has symmetrical characters\n";
47:    return 0;
48: }

```

程序列表 9.3 中程序的执行结果如下:

输出

Enteer a string:level

Input:level

Lowercase:level

Uppercase:LEVEL

Reversed:level.

your input has symmetrical characters.

分析:

程序列表 9.3 中的程序定义了串变量 str1 和 str2。每个串都能存储 STR_SIZE+1 个字符(包括空结束符)。主函数 main 还定义了标志变量 isLowerCase, isUpperCase 和 is Symmetrical。程序第 22 行的输出语句提示信息请你输入一个串,第 23 行的语句用串输入函数 getline,把你输入的串存入变量 str1 中。第 24 行的输出语句回显你的输入。

程序第 26 行的语句把变量 str1 中的字符拷贝到变量 str2 中。第 28 行的语句调用函数 strlwr,转换变量 str2 中的所有字符为小定形式。程序仅处理变量 str2 中的串,变量 str1 中的原串保持不变。第 29 行的语句调用了函数 strcmp,比较串 Str1 和 Str2 中的字符,如果你输入的串中无大写字母,则两相等。这个语句用 条件表达式,当上述条件成立时,把 TRUE 赋值 给标志变量 islowerCase,否则将 FALSE 赋值给标志变量 islowerCase,第 30 行的输出语句显示亦 str2 中的串。

程序第 32 行的语句调用函数 strupr,把变量 str2 中的所有小写字符转换成大写字符,第 33 行的调用函数 strcmp,比较串 str1 和 str2 中的字符。如果你输入的串中无小写字符,则两串相等,此时,条件表达式将 TRUE 赋值给标志变量 isUpperCase,否则将 FALSE 赋值给标志变量 isUpperCase,第 34 行的输出语句显示变量 str2 中的串。

为了显示倒序的串,程序调用函数 strlpy 再次将变量 str1 中的字符拷贝到 str2 中。程序第 38 行语句调用函数 strrev,对变量 str2 中的串倒序。第 39 行语句调用函数 strcmp 比较串 str1 和 str2 中的字符。如你输入的串是对称的,则两串相等。此时,条

件表达式将 TRUE 赋值给标志变量 isSymmetrical, 否则条件表达式将 FALSE 赋值给标志变量 isSymmetrical。第 40 行的输出语句显示变量 str2 的串。

程序在 41, 43 和 45 行用 if 语句显示你输入的串具有的特点。第 41 行中的 if 语句在变量 isLowercase 中的值为 TRUE 时, 显示串中无大小写字符的信息。第 43 行的 if 语句在变量 isUpperCase 中的值为 TRUE 时, 显示串中无小写字符的信息, 第 45 行的 if 语句当变量 isSymmetrical 中的值为 TRUE 时, 显示串中字符对称的信息。

9.10 查找字符

STRING.H 库提供了一些查找串中字符的函数。这些函数包括 strchr, strrchr, strspn, strstr, 和 strpbrk, 这些函数允许你查找串中字符和简单的字符图案。

函数 strchr 查找串中第一次出现的所需查找的字符。

语法

函数 strchr

函数 strchr 的原型是

char * strchr (const char * target, int c)

函数查找串 target 中第一次出现的字符在 c。该函数返回一指针, 指向串 target 中与特定形式 c 相同的字符, 如果串 target 中无字符 c, 则函数返回 NULL。

例子:

```
char str[8]="Turbo C++";
```

```
char * strptr;
```

```
strptr=strchr(str, '+');
```

指针 strPtr 指向串 str 中的子串 "++"。

函数 strrchr 查找串中最后一次出现的所需查找的字符。

语法

函数 strrchr

函数 strrchr 的原型是

char * strchr(const char * target, int c)

该函数查找串 target 中最后一次出现的字符 c。该函数返回一指针, 指向串 target 中与特定字符 c 相同的字符。如果串 target 中无字符 c, 则函数返回 NULL。

例子:

```
char str[81]="Turbo C++ is here";
```

```
char * strptr;
```

```
strptr=strrchr(str, 't');
```

指针 strPtr 指向串 str 的子串 "++ is here"。

函数 strstr 可以得到两个串中相同字符的个数。

语法

函数 strspn

函数 strspn 的原型是

```
size_t strspn(const char * target, const char * pattern)
```

该函数返回串 target 和串 pattern 中相同字符数目。

例子:

```
char str[] = "Turbo C++";
```

```
char substr[] = "obrut";
```

```
int index;
```

```
index = strspn(str, substr);
```

这条语句把 6 赋值给变量 index, 因为 substr 中的字符与 str 中的前六个字符相同。

函数 strcspn 检查一个串, 并得到原串中完全与子串中字符不符的最左边子串的长度。

语法

函数 strcspn

函数 strcspn 的原型是:

```
size_t strcspn(const char * str1, const char * str2)
```

函数检查串 str1, 并返回完全没有子串中字符的最左边的子串的长度。

例子:

```
char string[] = "The rain in Spain";
```

```
int i;
```

```
i = strcspn(string, "in");
```

这个示例把 8 赋值给变量 i (8 为 "The rain" 的长度)。

函数 strpbrk 查找第一次出现在原串中与子串相同的字符。

语法

函数 strpbrk

函数 strpbrk 的原型是

```
char * strpbrk (const char * target, const char * pattern)
```

函数查找第一次出现在 target 中的子串 pattern 中的字符。如果子串 pattern 的字符在串 target 中没有发现, 则函数返回 NULL。

例子:

```
char * str = "Hello there how are you";
```

```
char * substr = "h r";
```

```
char * ptr;
```

```
ptr = strpbrk(str, substr);
```

```
cout << ptr << "\n";
```

该示例显示 "here how are you" 因为在串中 'h' 是在 'r' 的前面。

9.11 查找子串

STRING.H 库提供函数 `strstr`。用于查找串中的子串。

语法

函数 `strstr`

函数 `strstr` 的原型是

```
char * strstr (const char * str, const * substr);
```

该函数查找第一次出现在串 `str` 中的子串 `substr`，函数返回一个指针，指向串 `str` 与子串 `substr` 相同的第一个字符。如果串 `str` 中没有子串 `substr`，则函数返回 `NULL`。

例子：

```
char dtr[ ]="Hello there! how are you";
char substr[ ]="how";
char * ptr;
ptr=strstr(str,substr);
cout<<ptr<<"\n";
```

此示例显示 "how are you"，因为该串中最前面就是所需查找的子串 "how"。指针 `ptr` 指向 "how" 开始的原串的其它字符。

DO	DON'T
DO 如果你想查找最后出现的子串，只需在调用函数 <code>strstr</code> 之前调用函数 <code>strrev</code> 。	
DON'T 在用 <code>strrev</code> 函数查找最后出现的子串时，不要忘记把原串和子串都倒序。	

STRING.h 库还提供了函数 `strtok`。允许你根据一组规定的定界字符，将一个串分为若干个子串。

新术语：子串又称为 token(标志)。

语法

函数 `strtok`

函数 `strtok` 的原型是

```
char * strtok (char * target, const char * delimiters)
```

该函数查找 `target` 串中的标志。一个串提供一组定界字符。下面例子说明这个函数的怎样在返回串中的标志中的工作情况。函数 `strtok` 通过插入 '\0' 字符于每个标志之后，来改变原串。确保在调用函数之前将原 `target` 串拷贝到另一串变量中。

例子:

```
#include <stdio.h>
#include <string.h>
main()
{
    char* str = "(Base - Cost + Profit) * Margin";
    char* tkn = "+ * ()";
    char* ptr = str;
    printf("%s\n", str);
    // the first call looks normal
    ptr = strtok(str, tkn);
    printf("\n\nThis is broken into: %s", ptr);
    while (ptr) {
        printf(" , %s", ptr);
        // must make first argument a NULL character
        ptr = strtok(NULL, tkn);
    }
    printf("\n\n");
}
```

这个示例当程序运行时显示下面信息:

(Base _ Cost + Profit) * Marqin

这被分成: Base _ cost, Profit, Margin

DO	DON'T
DO 记住,在函数 strtok 查找下一个标志之前提供 '\0' 字符(即 NULL)。	
DON'T 不要忘记在调用函数 Strtok 之前存储原串的拷贝。	

让我们看看下面这个查找字符和字符串的程序,程序列表 9.4 为 STRING4.CPP 的源程序。该程序执行下述功能:

- ☐ 输入主串。
- ☐ 输入欲查找的串。
- ☐ 输入欲查找的串。
- ☐ 显示字符标尺和主串。
- ☐ 显示欲查找的串在主串中的位置。
- ☐ 显示欲查找的字符在主串中的位置。

程序列表 9.4 程序 STRING4.CPP 的源代码

```
1: /*
2:    C++ program that demonstrates searching for the
```

```

3:  characters and strings
4:  */
5:
6:  #include <iostream.h>
7:  #include <string.h>
8:
9:  const unsigned STR_SIZE = 40;
10:
11:  main()
12:  {
13:      char mainStr[STR_SIZE+1];
14:      char subStr[STR_SIZE+1];
15:      char findChar;
16:      char *p;
17:      int index;
18:      int count;
19:
20:      cout << "Enter a string : ";
21:      cin.getline(mainStr, STR_SIZE);
22:      cout << "Enter a search string : ";
23:      cin.getline(subStr, STR_SIZE);
24:      cout << "Enter a search character : ";
25:      cin >> findChar;
26:
27:      cout << "      1      2      3      4\n";
28:      cout << "01234567890123456789012345678901234567890\n";
29:      cout << mainStr << "\n";
30:      cout << "Searching for string " << subStr << "\n";
31:      p = strstr(mainStr, subStr);
32:      count = 0;
33:      while (p) {
34:          count++;
35:          index = p - mainStr;
36:          cout << "Match at index " << index << "\n";
37:          p = strstr(++p, subStr);
38:      }
39:      if (count == 0)
40:          cout << "No match for substring in main string\n";
41:
42:      cout << "Searching for character " << findChar << "\n";
43:      p = strchr(mainStr, findChar);
44:      count = 0;
45:      while (p) {
46:          count++;
47:          index = p - mainStr;
48:          cout << "Match at index " << index << "\n";
49:          p = strchr(++p, findChar);
50:      }
51:      if (count == 0)
52:          cout << "No match for search character in main string\n";
53:      return 0;
54: }

```

程序列表 9.4 中程序执行结果如下:

输出:

Enter a string: here,there,and everywhere

Enter a search string:here

Enter a search character:e

1 2 3 4

0123456789012345678901234567890123456789

here,there,and everywhere

Searching for string here

Match at index 0

Match at index 7

Match at index 23

Searching for character e

Match at index 1

Match at index 3

Match at index 8

Match at index 10

Match at index 17

Match at index 19

Match at index 24

Match at index 26

分析:

程序列表 9.4 中的程序定义了两个串 `mainStr` 和 `subStr` 分别代表主串和欲查找的串。程序还定义了变量 `findChar` 来存储欲查找的字符,此外程序中还定义了一个字符型指针 `p` 和 `int` 型变量 `index` 和 `count`。

程序第 20 行的输出语句提示信息请你输入一个串。第 21 行的语句调用流输入函数 `getline`,把你输入的串存储在变量 `mainStr` 中,第 22 行的语句请你输入欲查找的串。第 23 行的语句调用流输入函数 `getline`,把你输入的串存入变量 `subStr` 中。第 24 行的输出语句请你输入欲查找的字符。第 25 行的语句将你输入的字符存入变量 `findChar` 中。

程序第 27 至第 29 行的输出语句显示一个标尺,并把变量 `mainStr` 中的串显示在标尺下方。第 30 行的输出语句提示程序正在查找你欲查找的串。第 31 行语句调用函数 `strstr`,查找在串 `mainStr` 中第一次出现的串 `substr`,第 32 行的语句把 0 赋值给变量 `count`,变量 `count` 存储着串 `mainStr` 中包含的 `substr` 串的个数。

程序第 33 至第 38 行使用 `while` 循环,确定 `subStr` 串在 `mainStr` 串中出现的所有情形。`while` 循环进行的条件是判断指针 `p` 中的地址。如果指针不为 `NULL`,进行循环,循环体中的第一个语句使变量 `count` 加 1。第 35 行的语句计算欲查找的串在主串中出现的位置。该语句通过指针 `p` 中地址与变量 `mainStr` 中第一个字符地址的差运算,来得到查找的地址(记住,表达式 `&mainStr[0]` 与 `mainStr` 等价)。同时,这条语句把结果赋值于变量 `index`,第 36 行的输出语句显示变量 `index` 中的数值。

第 37 行的语句查找在串 `mainStr` 中下一次出现的欲查找串的位置。注意,语句调用函数 `strstr` 从指针 `p` 所指的地址开始查询。这就保证了调用 `strstr` 函数查找下一种情形。`while` 循环后的 `if` 语句用来检验变量 `count` 的数值。如果为 0,程序则执行第 40 行的输入语句,提示没有发现欲查找串的信息。

程序第 42 行的输入语句提示你程序正在查找指定的字符。查找变量 `findChar` 中字符的过程类似于查找串 `subStr` 的过程。主要区别在于:这里的字符串查找程序是调用函数 `strchr` 来查找字符。

9.12 小结

本章引出了 C++ 语言字符串的概念,并讨论了头文件 `STRING.H` 中的字符串处理函数。主要包括下述内容:

- ☐ C++ 语言中的串是以 `'\0'` (空字符) 结尾的字符数组。
- ☐ 串的输出需要用到流输入函数 `getline`。这个函数要求你定义输入变量、输入字符的最大数目和定界符。
- ☐ `STRING.H` 头文件包含 C 语言标准的串库。这个库有许多各种各样支持串拷贝、连接、转换、倒序和查找功能的函数。
- ☐ C++ 语言支持两种对串赋值的方法。第一种方法是你把一个串赋值给一个串变量。第二种方法是用函数 `strcpy` 把一个串赋值给另一个串。`STRING.H` 库还提供函数 `strdup`, 把一个串拷贝到指定位置。
- ☐ 函数 `strlen` 返回串的长度。
- ☐ 函数 `strcat` 和 `strncat` 允许你连接两个串, 函数 `strncat` 允许你连接特定数目的字符。
- ☐ 函数 `strcmp`, `stricmp`, `strncmp`, `strncmp` 允许你执行不同类型的串比较。函数 `strcmp` 对两个串中字符逐个比较。函数 `stricmp` 是函数 `strcmp` 的另一个版本, 它忽略字符的大、小书写形式。函数 `strncmp` 是比较两个串的前几个所规定数目的字符 `strncmp` 函数又是 `strncmp` 的另一个版本。与之不同的是忽略了字符的书写形式不同。
- ☐ 函数 `strlwr` 和 `strupr` 分别把串中字符转换为小写或大写的形式。
- ☐ 函数 `strrev` 使串中字符的顺序颠倒。
- ☐ 函数 `strchr`, `strrchr`, `strspn`, `strcspn` 和 `strpbrk` 允许你在串中查找字符或简单的图案。
- ☐ 函数 `strstr` 完成在一个串中查找另一个串的功能。函数 `strtok` 可根据你规定的一组定界字符, 把一个串分解成若干个子串。

9.13 问与答

问: 一个语句能用串初始化指针吗?

答: 可以。编译程序把串中的字符存储在内存中, 并且把它的地址赋予指针变量, 如下示例

```
char * p = "I am small string";
```

此外, 你可用指向字符串的指针重写串中的字符, 然而, 记住: 指针 `p` 只能访问固定字符数目的串。

问: 能定义一个指向字符串的指针常量吗?

答: 可以。这种定义我们在前面已经提到过。然而, 因为语句是定义了一个指针常量,

所以你不能重写初始化串中的字符,如下示例:

```
const char *p="Version 1.0";
```

用 char 类型的指针常量存储特定的信息和标题。

问:能定义一个指向一组字符串的指针数组吗?

答:可以。用指针数组访问信息、标题及其它各种特定串的集合是最容易的一种方法。

如下示例。

```
char * mainMenu[ ]= {"File", "Edit", "Search", "View",  
                    "Debug", "options", "Windows", "Help"};
```

数组元素 p[0]用于访问第一个串,p[1]用于访问第二串以此类推。

问:我们能调用函数 strcmp,从规定的第 i 个字符开始比较两个串吗?

答:可以。只需函数 strcmp 的参数加上偏移量,如下示例:

```
char s1[41]="Turbo C++";  
char s2[41]="TURBO Pascal";  
int offset=5;  
int i;  
i=strcmp(str1+offset, str2+offset);
```

问:我们能调用函数 strncmp,从规定的第 i 个字符开始比较两个串中特定数目的字符吗?

答:可以。只需将函数 strcmp 的参数加上偏移量即可。如下示例:

```
char s1[41]="Turbo C++";  
char s2[41]="TURBO Pascal";  
int offset=5;  
int num=3;  
int i;  
i=strcmp(str1+offset, str2+offset, num);
```

9.14 专题讨论

本专题讨论列出一些测验问题,帮助你巩固在这章中学到的内容。此外还附加了一些练习,便于你在编程过程中更好地理解所学的内容,在继续下章的学习之前,尽量理解测验和练习的答案。答案在附录 B 中给出。

9.14.1 测验

1. 下面程序有何错误?

```
# include<iostream.h>  
# include<string.h>  
const int MAX=10;  
main()  
{
```

```

char s1[MAX+1];
char s2[ ]="123456789012345678901234567890";
strcpy(s1,s2);
cout<<"String 1 is"<<s1
    <<"\nString 2 is"<<s2;
    return 0;
}

```

2. 上述程序如用函数 `strncpy` 代替函数 `strcpy`, 程序的结果会是什么?
3. 下面程序执行后, 变量的值是什么?

```

char s1[ ]="Borland C++";
char s2[ ]="Borland Pascal";
int i;
i=strcmp(s1 s2);

```

4. 下面程序执行后, 变量 `i` 的值为什么?

```

char s1[ ]="Turbo C++";
char s2[ ]="Turbo Pascal";
int offset =Strlen ("Turbo");
int i;
i=strcmp(s1+offset,s2+offset);

```

5. 如果串中无小写字母, 下面的函数是否能准确地返回 1? 又如果串中有小写字母, 该函数是否能返回 0?

```

int hasNolowerCase (const char * s);
{
char s2[strlen(s)+1];
strcpy (s2,s);
struapyr(s2.);
return (strcmp(s2)=0)? 1:0;
}

```

9.14.2 练习:

1. 自己编写一个函数 `strlen`, 用 `while` 循环计算变量中串的长度, 并返回结果。
2. 自己再编写一个函数 `strlen`, 这次用 `while` 循环和局部指针计算串的长度, 并返回结果。
3. 编写一个程序 `STRINGS.CPP`, 调用函数 `strtok` 将串 `"2 * (X+Y)/(X+Z)-(X+10)/(Y-5)"` 用定界符 `" + - * / () , " ()` 和 `" + - * / "` 分解成为若干个子串。

第十章 高级函数参数(第十天)

函数同样是C++程序的基本组成部分。每一个函数都能实现一个特定的功能。但C语言较C++语言而言,被称为“面向函数”的编程语言则更为恰当。他们的不同之处在于:C++支持类、继承及其它面向对象的编程功能(其它内容将在后面几章给予讨论)。尽管如此,在C++语言中,函数仍然起着重要的作用。在本章中,C++语言函数的高级内容将使你耳目一新:

数组作为函数参数。

☐ 字符串作为函数参数。

☐ 用值传递结构。

☐ 用引用传递结构。

☐ 用指针传递结构。

☐ 递归函数。

☐ 传递指向动态数据结构的指针。

☐ 指向函数的指针。

10.1 使用数组作为函数参数

当你编写一个数组作为函数值参数的C++语言函数时,你可以定义一个指向数组基本类型的指针作为函数参数。

语法

数组指针参数

具有数组指针参数的函数原型为

```
returnType fuunction (basicType * ,<other parameter types >);
```

定义此种函数的语法为:

```
returnType fuunction (basicType * arrParam,<other parameters>);
```

例子:

```
//prototypes void shellsort (unsigned * doubleArray ,unsigned arraySize);  
void quicksort (unsigned * intArray ,unsigned arraySize);
```

在第七章中,我们已经说明了C++语言允许你用[]符号来定义开式数组参数。这种定义就等价于数组指针参数。尽管用开式数组参数的形式更能清楚地说明参数的形式,但C++编程更多是采用指针的形式。

DO	DON'T
DO 在主函数中采用常数参数,避免其参数被改变。	
DON'T 不要忘记定义另一个参数来标识数组的大小。这样,便于函数操作只有部分被赋予有效值的数组类型参数。	

让我们看看下面示例。程序列表 10.1 为 ADVFUN1.CPP 的源程序。它是通过对程序 ARRAYS.CPP 进行修改而生成的。该程序主要执行以下功能:

- ☐ 请你输入数据的个数。
- ☐ 请你输入数组的元素值(整型数)。
- ☐ 显示无序数组中的元素。
- ☐ 显示有序数组中的元素。

程序列表 10.1 程序 ADVFUN1.CPP 的源代码

```

1: // C++ program that sorts arrays using the Comb sort method
2:
3: #include <iostream.h>
4:
5: const int MAX = 10;
6: const int TRUE = 1;
7: const int FALSE = 0;
8:
9: int obtainNumData()
10: {
11:     int m;
12:     do { // obtain number of data points
13:         cout << "Enter number of data points [2 to "
14:             << MAX << "] : ";
15:         cin >> m;
16:         cout << "\n";
17:     } while (m < 2 || m > MAX);
18:     return m;
19: }
20:
21: void inputArray(int *intArr, int n)
22: {
23:     // prompt user for data
24:     for (int i = 0; i < n; i++) {
25:         cout << "arr[" << i << "] : ";
26:         cin >> *(intArr + i);
27:     }
28: }
29:
30: void showArray(const int *intArr, int n)
31: {
32:     for (int i = 0; i < n; i++) {
33:         cout.width(5);
34:         cout << *(intArr + i) << " ";
35:     }
36:     cout << "\n";
37: }
38:
39: void sortArray(int *intArr, int n)
40: {
41:     int offset, temp, inOrder;

```

```

42:
43:  offset = n;
44:  do {
45:      offset = (8 * offset) / 11;
46:      offset = (offset == 0) ? 1 : offset;
47:      inOrder = TRUE;
48:      for (int i = 0, j = offset; i < (n - offset); i++, j++) {
49:          if (intArr[i] > intArr[j]) {
50:              inOrder = FALSE;
51:              temp = intArr[i];
52:              intArr[i] = intArr[j];
53:              intArr[j] = temp;
54:          }
55:      }
56:  } while (!(offset = 1 && inOrder == TRUE));
57: }
58:
59: main()
60: {
61:     int arr[MAX];
62:     int n;
63:
64:     n = obtainNumData();
65:     inputArray(arr, n);
66:     cout << "Unordered array is:\n";
67:     showArray(arr, n);
68:     sortArray(arr, n);
69:     cout << "\nSorted array is:\n";
70:     showArray(arr, n);
71:     return 0;
72: }

```

程序列表 10.1 中程序的执行结果如下：

输出：

Enter number of data points [2 to 10]:5

arr [0]:55

arr [1]:22

arr [2]:78

arr [3]:35

arr [4]:45

unordered array is

55 22 78 35 45

Sorted array is :

22 35 45 55 78

分析：

程序列表 10.1 中程序与 7.5 节中的 ARRAYS.CPP 程序几乎相同。不同之处在于：此程序在函数 inputArray, showArray 和 sortArray 中的参数不同。这些函数的第一个参数是指向 int 型的指针。函数 showArray 把指针定义为常量。这种定义表明编译程序函数 showArray 不能改变参数 IntArray 中的元素。

10.2 使用字符串作为函数参数

因为 C++ 语言把字符串作为字符数组处理，因此，适用于数组参数的规则也适用于

字符串参数。下面这个程序就是一个字符串作为函数参数的例子。程序列表 10.2 是 ADVFUN2.CPP 的源程序。该程序请你输入一个字符串。然后显示你所键入的字符,并且是以大写的字符形式显示。

程序列表 10.2 程序 ADVFUN2.CPP 的源代码

```
1: /*
2:  C++ program that declares functions with string parameters
3:  */
4:
5: #include <iostream.h>
6:
7: const unsigned MAX = 40;
8:
9: char* upperCase(char* str)
10: {
11:     int ascii_shift = 'A' - 'a';
12:     char* p = str;
13:
14:     // loop to convert each character to uppercase
15:     while ( *p != '\0') {
16:         if ((*p >= 'a' && *p <= 'z'))
17:             *p += ascii_shift;
18:         p++;
19:     }
20:     return str;
21: }
22:
23: int strlen(char* str)
24: {
25:     char *p = str;
26:     while (*p++ != '\0');
27:     return --p - str;
28: }
29:
30: main()
31: {
32:     char aString[MAX+1];
33:
34:     cout << "Enter a string: ";
35:     cin.getline(aString, MAX);
36:     cout << "Your string has " << strlen(aString)
37:         << " characters\n";
38:     // concatenate bigStr to aString
39:     upperCase(aString);
40:     cout << "The uppercase version of your input is: "
41:         << aString;
42:     return 0;
43: }
```

程序列表 10.2 中程序的执行结果如下:

输出:

Enter a string :Turbo C++

Your string has 9 characters

The upper case version of your input is TURBO C++

分析:

程序列表 10.2 中程序自定义了串操作函数:uppercase 和 strlen。函数 uppCase 仅有

一个参数 `str`, 它是一个指向 `char` 类型的指针。这个参数传递字符数组的地址。该函数通过指针访问字符, 把串中的字符转换成大写形式, 并返回一个指向字符串的指针。该函数定义了局部变量 `ascii_shift` 和局部 `char` 型指针。同时该函数把 `ascii_shift` 初始化为字符 `A` 和 `a` 的 ASCII 值之差。这样变量 `ascii_shift` 的值就可用于把小写字母变为大写字母。此外, 函数还把局部指针初始化为参数的 `str` 的地址。

函数 `uppercase` 在程序第 15 行调用 `while` 循环, 用于变换字符串字符的形式。而循环条件从 `while` 语句判断指针是否指向终止符 `'\0'`。第 16 行的 `if` 语句判断指针 `p` 访问的字符是否为小写形式, 如果是小写形式, 则函数执行第 17 行中语句。该语句把变量 `ascii_shift` 中的数值与指针 `p` 所访问的当前字符的 ASCII 的值求和。这一操作便可做到把小写字母形式转换成大写字母形式。函数返回指针 `str`。

函数 `strlen` 返回由指针参数 `str` 访问的串中的字符数目。该函数定义了局部 `char` 型指针 `p`, 并把它初始化为参数 `str` 的地址。函数调用 `while` 空循环语句, 使指针 `p` 访问到终止符 `'\0'`。返回语句是返回指针 `p` 中的地址与 `str` 的地址的差值。该语句首先是指针 `p` 指向前一个元素, 调整指针中的地址。

主函数 `main` 定义了变量 `aString`。程序 34 行中的输出语句提示信息请你输入一个字符串。第 35 行的语句调用流输入函数 `getline` 来获得你的输入, 并存储在变量 `aString` 中。第 36 行的输出语句显示你输出的字符串。该语句通过把 `aString` 作为函数 `uppercase`。同样 `aString` 用为函数参数。第 40 行的输出语句显示你的输入的大写版本。这个字符串现在仍被存储在变量 `aString` 中。

10.3 使用结构作为函数参数

C++ 语言允许你利用数值或引用传递结构, 在这节中, 我们主要讨论按值传递结构。下节中, 我们将介绍利用引用 (reference) 传递结构的内容。结构的类型在函数原型中应给予说明。并且写法与那些预定义的类型相似。

程序列表 10.3 是程序 `ADVFUN3.CPP` 的源程序, 该程序执行以下功能:

- ☐ 请你输入第一点的 X 坐标和 Y 坐标。
- ☐ 请你输入第二点的 X 坐标和 Y 坐标。
- ☐ 计算界于你输入的两点间中点坐标。
- ☐ 显示中间点坐标。

程序列表 10.3 程序 `ADVFUN3.CPP` 的源代码

```
1: // C++ program which uses a function that passes
2: // a structure by value
3:
4: #include <iostream.h>
5:
6: struct point {
7:     double x;
8:     double y;
9: };
10:
```

```

11: // declare the prototype of function getMedian
12: point getMedian(point, point);
13:
14: main()
15: {
16:     point pt1;
17:     point pt2;
18:     point median;
19:
20:     cout << "Enter the X and Y coordinates for point # 1 : ";
21:     cin >> pt1.x >> pt1.y;
22:     cout << "Enter the X and Y coordinates for point # 2 : ";
23:     cin >> pt2.x >> pt2.y;
24:     // get the coordinates for the median point
25:     median = getMedian(pt1, pt2);
26:     // get the median point
27:     cout << "Mid point is (" << median.x
28:         << ", " << median.y << ")\n";
29:     return 0;
30: }
31:
32: point getMedian(point p1, point p2)
33: {
34:     point result;
35:     result.x = (p1.x + p2.x) / 2;
36:     result.y = (p1.y + p2.y) / 2;
37:     return result;
38: };

```

程序列表 10.3 中程序的执行示例如下:

输出:

Enter the x and y coordinates for point #1:1 1

Enter the x and y coordinates for point #2:5 5

Mid point is (3,3)

分析:

程序列表 10.3 中程序定义了结构 point, 它构造了一个二维点。这个结构体具有两个 double 型的成员。程序第 12 行定义了函数 getMedian 的函数原型。该函数包含两个 point 类型的参数, 参数按值传递。

主函数 main 分别在程序的第 16 行和第 18 行定义了 point 类型的变量 pt1, pt2 和 median。第 20 行的输出语句提示信息请你输入第一点的 X, Y 坐标值。第 21 行的语句获得你的输入并把输入的值存储在成员 pt1.x 和 pt1.y。第 22 行和第 23 行重复同一输入操作。输入第二个点, 并把输入的值存于成员 pt2.x 和 pt2.y 中。第 25 行的语句调用函数 getMedian, 调用实参为 pt1, pt2。函数接受参数 pt1 和 pt2 的拷贝, 并把 point 类型的函数结果返回存储在变量 median 中。第 27 和 28 行的输出语句用于显示中间点的 x, y 坐标值。即显示变量 median 中的 x, y 成员值)。

函数 getMedian 定义了两个 point 类型的参数 p1 和 p2。此外, 函数还定义了局部 point 类型的变量 result, 第 35 行的语句把成员 p1.x 和成员 p2.x 的平均值赋值给成员 result.x, 第 36 行的语句是把成员 p1.y 和 p2.y 赋值给成员 result.y。注意, 这些语句都使用“.”操作符来访问结构 p1, p2 和 result 的成员 x, y。当函数传递结构的拷贝或其引用变量时, 语法规则与用值传递结构参数的语法相同。return 语句返回局部变量 result 中的值。

10.4 由引用传递参数

C++语言允许你编写利用引用(reference)传递参数的函数。这种方法可使你在函数的作用范围之外改变参数中的值。C++语言提供了两种方法可达到这一目的:第一种是利用指针;第二种是利用引用(reference)形式参数。下面几节,我们就讨论利用引用(reference)传递各种类型的函数参数。

10.5 用引用传递结构参数

你可以利用指针或引用(reference)形式把结构传递给函数。许多C++语言编程者认为这两种方法都比用值传递结构的方法有效。

注意:利用引用传递一个结构时,允许引用参数使用“.”操作符。这一特点就使引用参数不必生成原始参数的拷贝。因此它们能快速地存取存储单元。因为引用参数是某一参数的别名,发生在引用参数上的任何操作都同时会作用到该参数上。为避免这种情况发生,我们定义了 const 型引用参数,这样编译程序就不能对这种能上能下用参数重新赋值。

DO	DON'T
DO 在下面两种情况中采用指针或引用来传递结构:第一种为主函数不能改变参数;第二种为函数用结构返回值。	
DON'T 一般情况下不要用值传递结构,除非你需要向主函数提供一个将被函数修改的数据拷贝。	

请看如下程序,程序列表 10.4 是 ADVFUN4.CPP 的源程序。该程序与程序 ADVFUN3.CPP 执行相同的功能,但实现的方法不同。

程序列表 10.4 程序 ADVFUN4.CPP 的源代码

```
1: // C++ program which uses a function that passes
2: // a structure by reference
3:
4: #include <iostream.h>
5:
6: struct point {
7:     double x;
8:     double y;
9: };
10:
11: // declare the prototype of function getMedian
12: point getMedian(const point&, const point&);
13:
```

```

14: main()
15: {
16:     point pt1;
17:     point pt2;
18:     point median;
19:
20:     cout << "Enter the X and Y coordinates for point # 1 : ";
21:     cin >> pt1.x >> pt1.y;
22:     cout << "Enter the X and Y coordinates for point # 2 : ";
23:     cin >> pt2.x >> pt2.y;
24:     // get the coordinates for the median point
25:     median = getMedian(pt1, pt2);
26:     // get the median point
27:     cout << "Mid point is (" << median.x
28:         << ", " << median.y << ")\n";
29:     return 0;
30: }
31:
32: point getMedian(const point& p1, const point& p2)
33: {
34:     point result;
35:     result.x = (p1.x + p2.x) / 2;
36:     result.y = (p1.y + p2.y) / 2;
37:     return result;
38: };

```

程序列表 10.4 中程序的执行结果如下:

输出:

Enter the X and Y coordinates for point #1 : 1 1

Enter the X and Y coordinates for point #2 : 9 9

Mid point is (5,5)

分析:

程序列表 10.4 中程序与程序列表 10.3 中程序相似。只是此程序在函数 `getMedian` 中采用引用参数。这样,函数的原型和函数的定义都采用在结构类型 `point` 后加上 `&` 符。利用引用参数,调用函数 `getMedian` 的方法与程序列表 10.3 中相同。而且函数 `getMedian` 的功能与程序列表 10.3 相同。两种版本都是用“.”操作符访问结构 `point` 中的成员 `x` 和 `y`。

10.6 用指针传递结构

另一种有效的传递结构的方法是利用指针。如使用引用参数一样,用 `Const` 类型定义指针,以避免用指针参数访问结构变量的成员时,改变结构的内容。

下一个示例程序也是程序 `ADVFUN3.CPP` 的另一版本,它使用指针参数传递结构。程序列表 10.5 为 `ADVFVN5.CPP` 的源程序。

程序列表 10.5 程序 `ADVFUN5.CPP` 的源代码

```

1: // C++ program which uses a function that passes
2: // a structure by pointer
3:
4: #include <iostream.h>
5:

```

```

6: struct point {
7:     double x;
8:     double y;
9: };
10:
11: // declare the prototype of function getMedian
12: point getMedian(const point*, const point*);
13:
14: main()
15: {
16:     point pt1;
17:     point pt2;
18:     point median;
19:
20:     cout << "Enter the X and Y coordinates for point # 1 : ";
21:     cin >> pt1.x >> pt1.y;
22:     cout << "Enter the X and Y coordinates for point # 2 : ";
23:     cin >> pt2.x >> pt2.y;
24:     // get the coordinates for the median point
25:     median = getMedian(&pt1, &pt2);
26:     // get the median point
27:     cout << "Mid point is (" << median.x
28:           << ", " << median.y << ")\n";
29:     return 0;
30: }
31:
32: point getMedian(const point* p1, const point* p2)
33: {
34:     point result;
35:     result.x = (p1->x + p2->x) / 2;
36:     result.y = (p1->y + p2->y) / 2;
37:     return result;
38: };

```

程序列表 10.5 中程序的执行示例如下:

输出:

Enter the X and Y coordinates for point # 1 : 2 2

Enter the X and Y coordinates for point # 2 : 8 8

Mid point is (5,5)

分析:

程序列表 10.5 在函数 `getMedian` 中采用指针参数其函数原型和函数的定义中都采用 `const point *` 型参数。程序第 25 行用取地址操作符 `&` 把变量 `pt1` 和 `pt2` 的地址传递给函数 `getMedian`。函数 `getMedian` 用 `->` 操作符访问结构成员 `x` 和 `y`。

10.7 递归函数

许多问题都可通过把原问题分为一系列简单相似的问题来解决。这样的问题可用递归的方法解决。

新术语:递归函数:是直接或间接调用函数本身的函数。递归调用的次数必须是限好次,避免程序陷入死循环。因此,每个递归函数都必须判断某个条件,决定是否递归结束。

新术语:递归函数的一个常用示例就是阶乘函数。N 的阶乘就是整数 1 到整数 N 的各整数乘积。记为 N!。

阶乘的数学表达式为:

$$N! = 1 * 2 * 3 * \dots * (N-2) * (N-1) * N$$

其递归的形式为:

$$N! = N * (N-1)!$$

$$(N-1)! = (N-1) * (N-2)!$$

.....

$$2! = 2 * 1!$$

$$1! = 1$$

不断循环递归,就可能到最后的結果。许多递归问题可化为非递归问题解决。在某些情况下,采用递归的形式,可使程序更简洁。递归函数是一种数学函数,通过用递归循环求解。

DO	DON'T
DO 递归函数中应具有判断语句,决定是否递归终止。	
DON'T 不要輕易地采用递归形式,除非用递归的方法远远优于其非递归的方法。	

让我们仔细研究一下阶乘的递归函数示例。程序列表 10.6 为 ADVFUN6.CPP 的源程序。该程序请你输入两个正整数,第一个数必须大于或等于第二个数。程序显示这两个数的组合数和排列数。组合数的运算公式如下:

$$C = m! / (m-n)! * n!$$

排列数运算公式如下:

$$P = m! / (m-n)!$$

程序列表 10.6 程序 ADVFUN.CPP 的源代码

```

1: // C++ program which uses a recursive function
2:
3: #include <iostream.h>
4:
5: const int MIN = 4;
6: const int MAX = 30;
7:
8: double factorial(int i)
9: {
10:     if (i > 1)
11:         return double(i) * factorial(i - 1);
12:     else
13:         return 1;

```

```

14: }
15:
16: double permutation(int m, int n)
17: {
18:     return factorial(m) / factorial(m - n);
19: }
20:
21: double combination(int m, int n)
22: {
23:     return permutation(m, n) / factorial(n);
24: }
25:
26: main()
27: {
28:     int m, n;
29:
30:     do {
31:         cout << "Enter an integer between "
32:              << MIN << " and " << MAX << " : ";
33:         cin >> m;
34:     } while (m < MIN || m > MAX);
35:
36:     do {
37:         cout << "Enter an integer between "
38:              << MIN << " and " << m << " : ";
39:         cin >> n;
40:     } while (n < MIN || n > m);
41:
42:     cout << "Permutations(" << m << ", " << n
43:          << ") = " << permutation(m, n) << "\n";
44:     cout << "Combinations(" << m << ", " << n
45:          << ") = " << combination(m, n) << "\n";
46:
47:     return 0;
48: }

```

程序列表 10.6 中程序执行结果如下:

输出:

Enter an integer between 4 and 30 : 10

Enter an integer between 4 and 10 : 5

Permutation (10,5)=30240

Combination (10,5)=252

分析:

程序列表 10.6 中程序定义了递归函数 factorial 和函数 permutation, combination 以 main。同时程序还定义了全程常数 MIN, MAX, 它们规定了第一个输入数据的范围。函数 factorial 只有一个 int 型的参数 i。该函数返回一个 double 型的值。第 10 行的 if 语句比较参数 i 值和 1 的大小。这个比较表达式判断是否进行递归调用。当 i 的值小于 1 时, 返回值 1。第 11 行的递归调用是以实参 i-1 调用函数 factorial。这样, 递归调用就提供给函数一较小的数值。

函数 permutation 具有两个 int 型参数 m 和 n。该函数两次调用递归函数 factorial。一次是用实参 m, 一次是用实参 m-n。函数 permutation 返回这两次调用函数 factorial 所得返回的值的比值。

函数 combination 也具有两个 int 型参数 m 和 n。该函数调用函数 permutation 和函数 factorial。并且返回其调用函数 permutation 和 factorial 所得返回值的比值。

主函数 main 定义了 int 型变量 m 和 n,该函数使用两次 do-while 循环语句,请你输入两个正整数。第一个循环体系中的输出语句请你输入一个介于 MIN 和 MAX 之间的整数。第 33 行的语句把你输入的值存储在变量 m 中。do-while 循环的 while 从句判断你的输入是否有效。当你输入的值大于 MAX 或小于 MIN 时,重新循环。

第二个 do-while 循环体中的输出语句请你输入一个介于 M 和 MAX 中间 r 整数。第 39 行的语句,把你输入的值存储在变量 n 中。其 while 从句判断你的输入是否有效。如果你输入的值大于 MAX,或小于 m,则重新循环。

第 42 和 43 行的输出语句显示变量 m 和 n 中两数值的排列数。第 49 和 45 行输出语句显示变量 m 和 n 中两数值的组合数。组合数和排列数均是用实参 m,n 调用函数 permutation 和 combination 而得。

10.8 传递指向动态存储结构的指针

二叉树的操作至少包括以下几个函数:二叉树的插入,查询,删除及二叉树的遍历。所有这些函数都是通过它的根指针对二叉树进行访问的。有趣的是,如树的插入或删除操作可能会影响树的根节点本身。在这种情况下,根节点的地址就会改变。因此,你需要用到根节点指针的引用。使用这个指针的引用可确保根节点的当前最新的地址。

新术语:二叉树是最常用的一种动态数据结构。这种数据结构可使你把各项数据有序地组织在一起,而不必考虑数据项的最初内容。二叉树的基础构造模块是节点(node)。二叉树中每个节点都可看为是它下面子树的根。终节点就是没有子树的节点。二叉树包含一个特殊的节点即为所有其余节点的根。每个节点都有一个 field,(用作为一个排序关键字)、可选择的附加数据(称为 non-key data)和两个指针以建立与其它树节点的链。动态存储器分配可使你为每个节点都分配一定的存储单元,同时,在不同的节点之间动态也建立链接。如果想进一步了解二叉树结构,可参考有关数据结构的书籍。

DO	DON'T
DO 把那些指向数据结构的指针的引用(reference)定义为函数参数,用来处理那些指针。这种定义方法可在函数的作用范围之外,改变参数的地址。	
DON'T 不要认为,当函数改变一个指针的非引用参数地址时,这种改变也同样会做用在指针参数的地址上。	

请让我们看看下面这个在二叉树中插入并显示动态数据的程序示例。程序列表 10.7 为 ADVFUN7.CPP 的源程序。该程序自己提供一组数据(地名表),把数据插入二叉树

中,同时按递增的顺序显示这组数据。

程序列表 10.7 程序 ADVFUN7.CPP 的源代码

```
1: // C++ program which passes parameter to dynamic data
2:
3: #include <iostream.h>
4: #include <string.h>
5:
6: const unsigned MAX = 30;
7:
8: typedef struct node* nodeptr;
9:
10: struct node {
11:     char value[MAX+1];
12:     nodeptr left;
13:     nodeptr right;
14: };
15:
16: void insert(nodeptr& root, const char* item)
17: // Recursively insert element in binary tree
18: {
19:     if (!root) {
20:         root = new node;
21:         strncpy(root->value, item, MAX);
22:         root->left = NULL;
23:         root->right = NULL;
24:     }
25:     else {
26:         if (strcmp(item, root->value) < 0)
27:             insert(root->left, item);
28:         else
29:             insert(root->right, item);
30:     }
31: }
32:
33: void showTree(nodeptr& root)
34: {
35:     if (!root)
36:         return;
37:
38:     showTree(root->left);
39:     cout << root->value << "\n";
40:     showTree(root->right);
41: }
42:
43: main()
44: {
45:     char *names[] = { "Virginia", "California", "Maine", "Michigan",
46:                       "New York", "Florida", "Ohio", "Illinois",
47:                       "Alaska", "Arizona", "Oregon", "Vermont",
48:                       "Maryland", "Delaware", "NULL" };
49:     nodeptr treeRoot = NULL;
50:     int i = 0;
51:
52:     // insert the names in the binary tree
53:     while (strcmp(names[i], "NULL") != 0)
54:         insert(treeRoot, names[i++]);
55:
56:     showTree(treeRoot);
57:     return 0;
58: }
```

程序列表 10.7 中程序的执行结果如下:

输出:

Alaska
Arizona
California
Delaware
Florida
Illinois
Maine
Maryland
Michigan
New York
Ohio
Oregon
Vermont
Virginia

分析:

程序列表 10.7 程序定义了一个全程常量 MAX,用于指定二叉树中各节点所能存储的最大字符数。程序第 8 行定义了指针类型的 nodeptrd 第 10 行至 14 行程序给出了结构 node 的定义。这个结构有三个成员: value(用于存储一个字符串),指向左下节点的指针 left 和指向右下节点的指针 right。两个指针都是 nodeper 类型。

程序定义了递归函数 insert,功能为把一个字符串插入到二叉树中。该函数具有两个参数: root 和 item。参数 root 是一个 nodeptr 类型指针的引用。这个参数保留二叉树中不同节点的地址,并且在需要的时候,可改变这些地址。当二叉树中插入新的数据项时,这些地址就会改变。

程序第 19 行的 if 语句判断参数 root 是否为 NULL。如果条件为真,则函数执行第 20 至 23 行语句。第 20 行的语句通过使用操作符 new 分配给新节点一定的存储单元。第 21 行语句从参数 item 中拷贝 MAX 个字符到成员 value 中。第 22 和 23 行的语句把 NULL 分别赋值给新生成节点的指向左,右两个子节点的指针,第 20 至 24 行的语句不会影响子树实际的根,而且也改变了指向不同节点的指针。if 语句判断是否结束递归调用。

第 25 行的 else 语句处理参数 root 不为 NULL 的情形。第 26 行的 if 语句判断指针 item 所访问的字符串是否小于当前节点的成员 value 中的字符串。如果小于,则函数以 root->left 和 item 实参调用递归函数 insert。这一调用操作,就把一个新的字符串插入左子树中,且其根即为当前节点。否则,函数以 root _right 和 item 为实参,调用递归函数 insert,即把新的字符串插入当前的节点的右子树位置。

递归函数 showTree 将遍历二叉树的各个节点以及参数 root 的各子树。当参数 root 的当前值为 NULL 时,函数退出执行,这种情形说明当前参数 root 即为二叉树的树根。所以,终止递归调用。如果参数 root 的当前值不为 NULL 时,函数以 root->left 为实参递归调用函数 showTree。这一调用操作允许函数访问当前节点的左子树。一

一旦左子树被访问,函数就显示存储在当前节点成员 value 中的值。然后函数再进行另一次递归调用。这次是以 root->right 为实参调用。这次调用允许函数访问当前节点的右子树,一旦右子树被访问,函数就退出。

主函数 main 在程序第 45 至 48 行定义了一个初始化了的指针数组。此外,还定义了 nodeptr 类型的 treeRoot 作为二叉树的根,定义的同时;把它初始化为 NULL。函数又给出了 int 类型变量 i 的定义,并且把它初始化为 0。

主函数 main 在第 53 行用 while 循环,把地名表中的字符串插入二叉树中。循环进行到当前地名为“NULL”时终止。“NULL”串为一个特殊的名字,用来标识地名表的结束。如果你改变程序,增加新的地名,就必须保证仍把“NULL”串放置在地名表的结尾处。第 54 行语句调用函数 insert,把元素 name[i]插入二叉树中当前根 tree Root 的子树位置。

第 56 行的语句以 treeRoot 为实参调用函数 showTree。通过递归函数的调用按字符递增的顺序显示地名表中内容。

10.9 指向函数的指针

程序的编译过程就是把变量名翻译为存储变量中数据的存储地址。指向这些地址的指针也就可以访问这些地址中的内容。函数也经过同样的编译处理。编译程序把函数名也翻译成一段可执行代码的入口地址。C++ 语言允许把包括指向函数在内的指针作为变量来处理。

语法

指向函数的指针

指向函数的指针的定义语法为

return Type (* functionpointer [array size])(<list of paramceers>)

例子:

```
double ( * fx[3] )( int ( n ) );
```

```
double ( * lx ) ( int n ) void ( * sort ) ( int * intArray , unsigned n );
```

```
unsigned ( * search ) ( int searchdey , int * intArray , unsigned n );
```

```
Unsigned ( * search [ MAX - SEARCH ] ) ( int searchKey , int * intArray , Unsigned n );
```

第一个标识符 fx,是指向函数组的指针数组,其中每个函数都只有一个 int 类型的参数,返回值为 double 类型。第二个标识符 sort,也是指向函数组的指针数组,其中每个函数具有两个参数;一个为 int 类型的指针,另一个为 unsigned 型。且函数返回值为 void 类型。第三个标识符 search,同样也是指向函数组的指针数组,其中每个函数具有三个参数;一个为 int 类型,一个为 int 类型指针,还有一个为 unsigned 型。

语法

函数指针数组

```
returnType (*functionpointer[arraySize])(<List of parameters>);
```

这种形式告诉编译程序 function pointer 是一个指向函数的指针。函数回值的类型是 returnType。

例子:

```
double (*fx[3])(int n);
```

```
void (*sort[MAX_SORT])(int *intArray,unsigned n);
```

```
unsigned (*search[MAX_SEARCH])(int searchkey ,int *intArray,unsigned n);
```

第一个标识 fx 是一个指向函数的数组。该函数具有一个 int 类型的参数且函数返回值为 double 类型。第二个标识 sort 是一个指向函数数组的指针。每个元素返回一个 void 类型并具有两个参数:一个 int 类型的指针和一个 unsigned 型的变量。第三个标识符 search,同样也是一个指向函数的指针。每个元素返回一个 unsigned 类型并且具有三个参数:int 类型的变量,int 类型的指针和一个 unsigned 类型的变量。

对于任何指针,在使用它之前,你必须初始化函数指针。这上步很简单。你只需要把空的函数名分配给函数指针即可。

语法

函数指针的初始化

初始化指向函数的指针的语法为

```
function pointer = aFunction;
```

用于赋值的函数与函数指针必须具有相同的参数表及返回值类型。否则,编译程序显示错误信息。

例子:

```
void (*sort)(int * intArray ,unsigned n);
```

```
sort = qsort;
```

语法

分配函数到一个元素

分配函数到函数指针数组的元素的语法为

```
functionpointer [index] = aFunction;
```

一旦你把函数名赋值给一个函数指针,你就可用这个函数指针调用该函数。现在你就会明白,为什么用于赋值的函数与函数指针必须具有相同的参数表及返回值类型。

例子:

```
void (*Sort[2])(int * intArray ,unsigned n);
```

```
Sort[0] = shellsort;
```

```
Sort[1] = CombSort;
```

语法

函数指针表达式

调用函数指针表达式的语法为

```
(*function pointer)(<argnwent list>);
```

(* function pointer [index]) (<argument list>);

例子:

(* sort)(&intArray ,n);

(* sort[0])(&intArray ,n);

让我们看看下面这个示例。程序列表 10.8 为 ADVFN8.CPP 的源程序。该程序执行线性回归的功能。自变量为 X, 应变量为 Y。回归方程为:

$$f(Y) = \text{intercept} + \text{slop} * g(X)$$

函数 f(Y) 的自变量为 Y, 函数 g(X) 的自变量为 X。函数 f(Y) 和 g(X) 可是线性, 对数, 指数, 平方根, 平方或其它数学函数。当 f(Y)=Y, 且 g(x)=X 时, 回归方程变为:

$$Y = \text{intercept} + \text{slope} * x$$

一般线性回归是计算斜率和截距在 f(Y) 和 g(X) 一定时的最优值。回归过程还提供相关系数统计, 此表明 f(Y) 的百分之多少决定于 g(X)。相关系数为 1 时, 说明 g(X), f(Y) 成线性关系; 若相关系数为 0, 则说明 g(X) 与 f(Y) 不相关。

程序列表 10.8 中程序执行下述功能:

- ☐ 请你输入几个数据(你输入的数据必须在程序规定的范围内)。
- ☐ 请你输入观察的 X, Y 值。
- ☐ 请你选择 g(X) 的函数类型(程序显示了一个小菜单, 供你选择。菜单项包括: 线性, 对数, 平方, 平方根及倒数函数)。
- ☐ 请你选择 f(Y) 的函数类型(程序也显示了一个小菜单, 供你选择。菜单项包括: 线性, 对数, 平方, 平方根及倒数函数)。
- ☐ 执行回归运算。
- ☐ 显示线性回归的截距, 斜率及相关系数。
- ☐ 请你选择是否进行另一组回归运算(如果你选择“Y”, 则程序重新执行第 3 步)。

程序列表 10.8 ADVUN8.CPP 的源代码

```
1: /*
2:  C++ program which uses pointers to functions to implement a
3:  a linear regression program that supports temporary
4:  mathematical transformations.
5:  */
6:
7: #include <iostream.h>
8: #include <math.h>
9:
10: const unsigned MAX_SIZE = 100;
11:
12: typedef double vector[MAX_SIZE];
13:
14: struct regression {
15:     double Rsqr;
16:     double slope;
17:     double intercept;
18: };
19:
20: // declare function pointer
21: double (*fx)(double);
22: double (*fy)(double);
23:
```

```

24: // declare function prototypes
25: void initArray(double*, double*, unsigned);
26: double linear(double);
27: double sqr(double);
28: double reciprocal(double);
29: void calcRegression(double*, double*, unsigned, regression&,
30:                    double (*fx)(double), double (*fy)(double));
31: int select_transf(const char*);
32:
33: main()
34: {
35:     char ans;
36:     unsigned count;
37:     vector x, y;
38:     regression stat;
39:     int trnsfx, trnsfy;
40:
41:     do {
42:         cout << "Enter array size [2.."
43:              << MAX_SIZE << "] : ";
44:         cin >> count;
45:     } while (count <= 1 || count > MAX_SIZE);
46:
47:     // initialize array
48:     initArray(x, y, count);
49:     // transform data
50:     do {
51:         // set the transformation functions
52:         trnsfx = select_transf("X");
53:         trnsfy = select_transf("Y");
54:         // set function pointer fx
55:         switch (trnsfx) {
56:             case 0 :
57:                 fx = linear;
58:                 break;
59:             case 1 :
60:                 fx = log;
61:                 break;
62:             case 2 :
63:                 fx = sqrt;
64:                 break;
65:             case 3 :
66:                 fx = sqr;
67:                 break;
68:             case 4 :
69:                 fx = reciprocal;
70:                 break;
71:             default :
72:                 fx = linear;
73:                 break;
74:         }
75:         // set function pointer fy
76:         switch (trnsfy) {
77:             case 0 :
78:                 fy = linear;
79:                 break;
80:             case 1 :
81:                 fy = log;
82:                 break;
83:             case 2 :
84:                 fy = sqrt;
85:                 break;
86:             case 3 :
87:                 fy = sqr;
88:                 break;
89:             case 4 :
90:                 fy = reciprocal;

```

```

91:         break;
92:     default :
93:         fy = linear;
94:         break;
95:     }
96:
97:     /* call function with functional arguments
98:                                     |   |
99:                                     V   V */
100:    calcRegression(x, y, count, stat, fx, fy);
101:
102:    cout << "\n\n"
103:         << "R-square = " << stat.Rsqr << "\n"
104:         << "Slope = " << stat.slope << "\n"
105:         << "Intercept = " << stat.intercept << "\n\n\n";
106:    cout << "Want to use other transformations? (Y/N) ";
107:    cin >> ans;
108:    } while (ans == 'Y' || ans == 'y');
109:    return 0;
110: }
111:
112: void initArray(double* x, double* y, unsigned count)
113: // read data for array from the keyboard
114: {
115:     for (unsigned i = 0; i < count; i++, x++, y++) {
116:         cout << "X[" << i << "] : ";
117:         cin >> *x;
118:         cout << "Y[" << i << "] : ";
119:         cin >> *y;
120:     }
121: }
122:
123: int select_transf(const char* var_name)
124: // select choice of transformation
125: {
126:     int choice = -1;
127:     cout << "\n";
128:     cout << "select transformation for variable " << var_name
129:         << "\n"
130:         << "0) No transformation\n"
131:         << "1) Logarithmic transformation\n"
132:         << "2) Square root transformation\n"
133:         << "3) Square transformation\n"
134:         << "4) Reciprocal transformation\n";
135:     while (choice < 0 || choice > 4) {
136:         cout << "\nSelect choice by number : ";
137:         cin >> choice;
138:     }
139:     return choice;
140: }
141:
142:
143: double linear(double x)
144: { return x; }
145:
146: double sqr(double x)
147: { return x * x; }
148:
149: double reciprocal(double x)
150: { return 1.0 / x; }
151:
152: void calcRegression(double* x,
153:                    double* y,
154:                    unsigned count,
155:                    regression &stat,
156:                    double (*fx)(double),

```

```

157:         double (*fy)(double))
158:
159: {
160:     double meanx, meany, sdevx, sdevy;
161:     double sum = (double) count, sumx = 0, sumy = 0;
162:     double sumxx = 0, sumyy = 0, sumxy = 0;
163:     double xdata, ydata;
164:
165:     for (unsigned i = 0; i < count; i++) {
166:         xdata = (*fx)(*(x+i));
167:         ydata = (*fy)(*(y+i));
168:         sumx += xdata;
169:         sumy += ydata;
170:         sumxx += sqr(xdata);
171:         sumyy += sqr(ydata);
172:         sumxy += xdata * ydata;
173:     }
174:
175:     meanx = sumx / sum;
176:     meany = sumy / sum;
177:     sdevx = sqrt((sumxx - sqr(sumx) / sum)/(sum-1.0));
178:     sdevy = sqrt((sumyy - sqr(sumy) / sum)/(sum-1.0));
179:     stat.slope = (sumxy - meanx * meany * sum) /
180:                 sqr(sdevx)/(sum-1);
181:     stat.intercept = meany - stat.slope * meanx;
182:     stat.Rsqr = sqr(sdevx / sdevy * stat.slope);
183:
184: }

```

程序列表 10.8 中程序执行结果如下:

输出:

```

Enter array size [2..100] : 5
X[0] : 10
Y[0] : 50
X[1] : 25
Y[1] : 78
X[2] : 30
Y[2] : 85
X[3] : 35
Y[3] : 95
X[4] : 100
Y[4] : 212

select transformation for variable X
0) No transformation
1) Logarithmic transformation
2) Square root transformation
3) Square transformation
4) Reciprocal transformation

Select choice by number : 1

select transformation for variable Y
0) No transformation
1) Logarithmic transformation
2) Square root transformation
3) Square transformation
4) Reciprocal transformation

Select choice by number : 1

```

```

R-square = 0.977011
Slope = 0.63039
Intercept = 2.370556

```

```

Want to use other transformations? (Y/N) y

```

```

select transformation for variable X
0) No transformation
1) Logarithmic transformation
2) Square root transformation
3) Square transformation
4) Reciprocal transformation

```

```

Select choice by number : 0

```

```

select transformation for variable Y

```

```

0) No transformation
1) Logarithmic transformation
2) Square root transformation
3) Square transformation
4) Reciprocal transformation

```

```

Select choice by number : 0

```

```

R-square = 0.999873
Slope = 1.79897
Intercept = 32.041237

```

```

Want to use other transformations? (Y/N) n

```

分析:

程序列表 10.8 中程序定义了全程常量 MAX_SIZE。用来指定数组的大小。程序在第 12 行中定义 vector。此外,程序在第 14 行至第 18 行定义了结构 regression,这个结构存储有关线性回归的信息。在第 21 行和 22 行又定义了全程函数指针 fx 和 fy。每个指针处理一个函数。该函数只具有一个 double 型的参数,且返回值也为 double 类型。程序用这两个全程函数指针存储你所选择的数学函数。

程序还定义了函数 intArray, linear, sqr, reciprocal, calcRegression, Selecttransf 和 main。函数 intArray 请你输入数组 x 和 y 的元素值。函数 linear, sqr 和 reciprocal 是一些简单的函数,对数据进行一些数学操作。函数 sqrt 和 log 在头文件 MATH.H 中给出函数原型。这些数学函数与函数指针 fx, fy 具有相同的参数,并返回相同类型的函数值。

函数 calcRegression 根据数组参数 x, y 计算回归统计。该函数用函数指针 fx, fy 作为参数,传递数组 x, y 中的数据。第 166 和 167 行分别用函数指针 fx, fy 传递数组 x, y 的元素。

函数 select_transf 提供一个简单的菜单,用于选择函数的形式。该函数返回你所选的项号。

主函数 main 定义了数组 x, y 为 vector 类型。此外,主函数也定义了一个结构变量 Stat,用于存储回归信息。主函数请你输入你想处理的数据数目。然后,主函数调用函数 intArray,输入数组 x, y 的元素。接着,主函数调用二次 select_transf 函数,选择以 x, y 为自变量的函数类型。第 55 行的 switch 语句检查变量 trnsfx 中的数据,它包含

你选择的 x 函数的种类号。不同的种类号就把函数指针 fx 赋于为恰当的函数名,如一些函数 \log 和 $\sqrt{}$, 它们的函数原型在头文件 $MATH.H$ 中。第 76 行的 `Switch` 语句,对函数指针 fy 执行类似的操作。

主函数 `main` 以 $x, y, cont, stat$ 及函数指针为实参调用函数 `calcRegression`。第 102 至 105 行的输出语句显示了当前这组数学函数的回归信息。第 107 自动控制语句询问你是否还希望选择另一组数学函数。第 107 行的语句把你的输入存储在变量 `ans` 中。第 108 行的 `while` 语句判断是否重复选择函数类型及计算相应的回归统计的过程。

10.10 小结

这章我们主要讨论了高级的 C++ 语言函数。你已经学到如下内容:

- ☐ 你可用指向数组的基本类型的指针,把数组作为实参传递给函数。C++ 语言允许你用确定的指针类型或 `[]` 符的形式定义数组参数。这种参数便于你编写处理不同大小的数组的函数。此外,这些指针又可通过它们的地址来访问数组元素,而无需生成整个数组的拷贝。
- ☐ 使用传递数组的方法,把字符串作为实参传递给函数。因为 C++ 语言的字符实际上就是字符数组。
- ☐ 通过把不相关的信息封装在 C++ 的结构中,并且用结构作为函数的参数,这样就可大大缩短函数的参数表。C++ 语言支持通过值传递结构的方法。这种传递方法是把它的实参的拷贝传递给主函数。因此,在函数作用范围之外,发生在结构成员上的变化就不会影响到函数的实参。
- ☐ 用指针或引用形参传递引用参数。引用形参可看为函数实参的别名。发生在参数上的任何变化都同样会发生在函数作用范围之外的函数实参上。你可以定义一个常量引用参数,确保函数不会改变实参的内容。用 `操作符` 可访问结构引用参数的成员。
- ☐ 用指针结构传递给函数需要用 `→` 操作符来访问结构的不同成员。你可以使用 `const` 型指针。来避免函数改变结构的成员。
- ☐ 递归函数就是直接或间接调用自身的函数。递归调用的次数必须是有限次以避免程序陷入死循环。因此,每个递归函数都必须判断一定的条件来决定是否结束递归。
- ☐ 传递指向动态数据结构的指针需要使用树根或头指针的引用来管理这种数据结构,这章主要介绍了如何进行二叉树的插入和遍历。
- ☐ 指向函数的指针中存储着函数的入口地址。这样的指针和函数必须具有相同的参数表和相同类型的返回值。指向函数的指针便于你选择所希望的函数操作。

10.11 问与答

问:与使用普通变量作为参数相比,使用引用参数会对函数的设计有何影响?

答:引用参数也能改变函数的实参(除非它被定义为 `const` 类型参数)。这样函数就能

引用参数作为一个输入/输出数据的通路。

问:如何判断指针是用来传递一组数值还是用来返回参数?

答:你需要看看函数的定义。然而,你还可以用一个引用参数来定义一个返回调用者函数值的参数。

问:调用递归函数需要何种存储结构?

答:实时系统用堆栈来存储中间值,包括通过调用递归函数而生成的值。与其它存储区相似,堆栈也具有有限的存储单元。因此,用很长的、占据存储单元过多的参数进行递归调用会引起实时错误。

新术语:堆栈是以“后进先出”方式工作的一个内存分配。

10.12 专题讨论

这节中,我们给出一些测验问题以帮助你巩固这章学到的知识。通过编程,使你进一步理解本章的内容。在进行下一章的学习之前,务必理解测验和练习的答案。在附录 B 中给出了有关的答案。

10.12.1 测验

1. 如何用条件表达式来编写阶乘递归函数?
2. 下面的递归程序有何错误?

```
double factorial(int i)
{
    switch (i) {
        case 0:
        case 1:
            return 1;
            break;
        case 2:
            return 2;
            break;
        case 3:
            return 6;
            break;
        case 4:
            return 24;
            break;
        default:
            return double(i) * factorial(i-1)
    }
}
```

3. 把下面递归的 Fibonacci 函数($\text{Fib}(0)=0, \text{Fib}(1)=1, \text{Fib}(2)=1, \text{Fib}(3)=2, \text{Fib}(4)=3, \dots$), 转换为非递归函数。

```
double Fibonacci(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1 || n == 2)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

4. 下面两种版本的函数是否等价?

```
struct stringStruct {
    char source[MAX+1];
    char uprStr[MAX+1];
    char lwrStr[MAX+1];
    char revStr[MAX+1];
};

void convertStr2(const char* str, stringStruct& s)
{
    strncpy(s.source, str, MAX);
    strncpy(s.uprStr, str, MAX);
    strncpy(s.lwrStr, str, MAX);
    strncpy(s.revStr, str, MAX);
    _strlwr(s.lwrStr);
    _strupr(s.uprStr);
    strrev(s.revStr);
}

void convertStr2(const char* str, stringStruct* s)
{
    strncpy(s->source, str, MAX);
    strncpy(s->uprStr, str, MAX);
    strncpy(s->lwrStr, str, MAX);
    strncpy(s->revStr, str, MAX);
    _strlwr(s->lwrStr);
    _strupr(s->uprStr);
    strrev(s->revStr);
}
```

10.12.2 练习

由程序 ADVFUN8.CPP 生成程序 ADVFUN9.CPP 其中用函数指针数组 f 代替独立的函数指针 fx, fy。此外,用这个函数指针数组 f 做为函数 calcRegression 的参数。

第十一章 面向对象编程及 C++ 类(第十一天)

类是 C++ 语言针对面向对象编程(OOP)新增加的一项功能,它是 C++ 语言的重要组成部分。在这章中,将向您介绍如何建立独立的类及类层。主要内容如下:

- ☐ 面向对象的基础
- ☐ 建立类
- ☐ 构造函数
- ☐ 析构函数
- ☐ 类层的定义
- ☐ 虚拟函数
- ☐ 帮助函数
- ☐ 操作符与友元的操作符

11.1 面向对象编程的基础

我们生活在由“对象”(object)组成的世界中,两个对象都有自身的属性和操作方式。有的对象比其客观存在对象活跃。因此你可把对象分为类。例如:CASIO Data Bank 表就是一个对象,它属于 CASIO Data Bank 表类。

新术语:面向对象编程(OOP)是利用真实世界对象的概念来开发应用程序。

你可以涉及同一类层中各独立的类。CASIO Data Bank 表就是表类层的一部分。OOP 的基础是类、对象、信息、方法、继承及多态性。

新术语:类被认为是对象的类型。每个对象都是类中的一个例子。

11.2 类与对象

在同一类中的所有对象都具与其它类相同的属性和功能,典型地,一个对象只具有由它属性的当前值定义的唯一的状态。类的功能函数决定了类中的对象可能具有的功能。C++ 语言即调用类中数据成员的属性,又调用类中的成员函数的操作。类中封装了数据成员及成员函数。

回顾 CASIO 表的示例,表中的按键代表了 CASIO 表类中的成员函数。显示代表了类中的数据成员。你可以按下任意按钮来编辑数据的 and/or 操作。从 OOP 的观点来看。成

员函数可通过改变类的数据成员,而改变对象的状态。

11.3 信息和方法

面向对象编程,当信息传送给一个对象,或在不同对象间传送时,它就模拟了不同对象的关系。对象接受一个信息后,可进行适当的操作。C++如其它 OOP 语言一样,不明确地支持信息和方法的概念。然而,你会发现用术语“信息”讨论成员函数很方便。术语“方法”和“成员函数”是等价的。

新术语:信息就是说明对象将进行何种操作。方法:就是指接受到有关信息后,对象将被怎样操作。

11.4 继承

在面向对象设计的语言中,你可以从一个类衍生出其它类。

新术语:用继承,衍生类(也称后裔类 descendent class)继承了它的父辈或原始类的数据成员及成员数据。

通过增加新的属性和新的操作方式,改进原始类生成衍生类。衍生类定义了新的数据成员及成员函数。此外,衍生类也能取消成员函数,这样那些函数的操作就不再适合衍生类。

为了把继承的概念用于 CASIO Data Bank 表,我们认为下例情况是可能的。假设该表制造商决定生产一种 CASIO Data Comm 表,该表与 CASIO Data Bank 具有相同的特点而且还增加了一个发音装置。CASIO 设计根据现有的 CASIO Data Bank 的设计,在它的基础上重新设计新的产品(用 OOP 的术语来讲,就是一个新类)。这个设计过程对已有的设计增加了新属性和操作方式,并且改变了原来的一些操作方式,来适应新设计方案。这样 CASIO Data Com 就继承了 CASIO Data Bank 的属性及操作方式。用 OOP 设计的术语来说,CASIO Data Comm 类是 CASIO Data Bank 类的衍生类。

11.5 多态性

OOP 设计的多态性特点允许你用同样的方法处理多种不同类型的数据,便于同一符号名称可在同一类层共同。例如:在图形形状的类层中(点、线、正方形、矩形、圆、椭圆等)。每个图形都是用 Draw 函数进行恰当的操作而生成的。

新术语:多态性使不同的类,从多个同名的函数中选择相应的调用,从而处理不同的

类。

11.6 建立类

C++语言允许你建立一个类,封装数据成员和成员函数。这些函数检索数据成员的值,并且执行有关的操作。

语法

基类

建立基类的基本语法为

```
class className
{
private:
    <private data members>
    <private constructors>
    <private member function>
protected:
    <protected data members>
    <protected constructors>
    <protected member functions>
public:
    <public data members>
    <public constructors>
    <public destructors>
    <public member functions>
}
```

例子:

```
{
protected:double x;
           double y;
public:
    point (double xVal ,double yVal );
    double getX();
    double getY();
    void assign (double xVal,double yVal);
    point &. assign (point &. pt );
};
```

11.7 类的组成部分

前面的语法说明类的建立方法。C++语言的类对于不同的成员(数据成员和成员函

数)提供了三个可视的层次:

- ☐ private 段
 - ☐ protected 段
 - ☐ public 段
-

新术语:在 private 段,只有类中的成员函数可访问该段的成员。在 protected 段,只有类中的成员函数和它的衍生类可访问段中的成员。在 public 段,类中的成员函数,类中的对象,衍生类的成员函数,及衍生类中的对象都可访问该段的成员。

使用不同段的规则如下:

1. 类中的各段的顺序不定。
 2. 类中的各段出现的次数不定。
 3. 在类中,如果没有定义所有段,则 C++ 编译程序把其成员作为 protected 型处理。
 4. 在 public 段中,你可不设置数据成员,这样的方式可使你的设计十分简洁。数据成员通常在 protected 段中设置。允许该类及其衍生类的成员函数访问。
 5. 用成员函数设置 and/or 查询数据成员的值。这些成员函数根据成员,进行有效的操作,必要时,可更新旧的数据成员。
 6. 类中可以有多个构造函数,它主要存在于 public 段中。
 7. 类中仅可有一个析构函数,而且必须下义在 public 段中。
 8. 成员函数(和构造函数,析构函数)中有许多语句是在类的范围外定义的。这些定义与类的定义可在同一文件中。
-

新术语:构造函数是特殊的成员。构造函数的名称必须和类的名称一样。析构函数可自动地删除类的对象。

在软件库中,成员函数的定义可参考第 8 条规则。当你定义一个成员函数时,你必须用类的名称限定函数,这种限定的形式就是在类的名称后使用::符,其后才是函数名。例如:

```
class point
{
protected:
    double x;
    double y;
public:
    point (double xVal, double yVal);
    double get x();
    // other member function
}
```

构造函数和成员函数的定义为:

```

point::point (double xval, double yval)
{
    //statements
}

double point::getx()
{
    // statements
}

```

一旦你定义了一个类,你就可用类名作为类型标识符,定义类的对象。定义的语法就如同变量的定义。

我们看看下面程序示例,程序列表 11.1 为 CLASS1.CPP 的源程序。该程序功能为:请你输入一矩形的长和宽,然后显示你规定的矩形的长,宽和面积。

程序列表 11.1 程序 CLASS1.CPP 的源代码

```

1: // C++ program that illustrates a class
2:
3: #include <iostream.h>
4:
5: class rectangle
6: {
7:     protected:
8:         double length;
9:         double width;
10:    public:
11:        rectangle() { assign(0, 0); }
12:        rectangle(double len, double wide) { assign(len, wide); }
13:        double getLength() { return length; }
14:        double getWidth() { return width; }
15:        double getArea() { return length * width; }
16:        void assign(double len, double wide);
17: };
18:
19: void rectangle::assign(double len, double wide)
20: {
21:     length = len;
22:     width = wide;
23: }
24:
25: main()
26: {
27:     rectangle rect;
28:     double len, wide;
29:
30:     cout << "Enter length of rectangle : ";
31:     cin >> len;
32:     cout << "Enter width of rectangle : ";
33:     cin >> wide;
34:     rect.assign(len, wide);
35:     cout << "Rectangle length = " << rect.getLength() << "\n"
36:           << "          width = " << rect.getWidth() << "\n"
37:           << "          area  = " << rect.getArea() << "\n";
38:     return 0;
39: }

```

程序列表 11.1 中程序的执行结果为:

输出:

```
Enter Length of rectaule :10
Enter width of rectaule :20
Rectangle length = 10
        width= 12
        area= 120
```

分析:

程序列表 11.1 中程序定义了 rectangle 类来模拟矩形。这个类中有两个 double 类型的数据成员 length 和 winth。此外,类中还有两个构造函数缺省的构造函数和非缺省构造函数。类中还定义了几个成员函数:getLength,getWidth, getArea 及 assign。

新术语:缺省构造函数生成 0 维的对象。非缺省构造函数生成非 0 维的对象。

函数 getLength,定义在类的内部,仅返回成员 length 的值。函数 getWidth 也定义在类的内部,仅返回成员 windth 的值。函数 getArea 同样定义在类的内部,仅返回成员 length 和 width 的乘积。

函数 assign 在类的外部定义,把参数 len 和 winde 的实参赋值给数据成员 length 和 width。你可简化些函数,不必检查负值。

主函数 main 定义了 rectangle 类的对象 rect 和 double 类型的 len,wide。第 30 行的输出语句请你输入矩形的长度。第 31 行的语句获得你的输入,并存储在变量 len 中。第 32 行的语句获得你的输入矩形的宽度。第 33 行的语句获得你的输入,并将在存储在变量 wide 中。

主函数 main 调用成员函数 assign,把输入的值赋予对象 rect。从 OOP 设计观点来看,你可以认为主函数 main 把赋值信息传送给 rect 对象。信息的实参为 len 和 wide 对象 rect 调用成员函数 rectangle::assign(double double)。

程序第 35 行至 37 行的输出语句显示对象 rect 的长、宽及面积。这些语句把信息 getlength,getWidth 和 getArea 传送给对象 rect。对象 rect 依次调用相应的成员函数,进行各种操作。

11.8 构造函数

C++ 语言的构造函数和析构函数都是自动执行以确保类中对象的建立和删除。

语法

构造函数

构造函数的语法为

```
class className
{
    public:
        className(); //default constructor
```



```

        className (className & c); // copy
constructor
        className (<parameter list>); // another
constructor
};

```

例子:

```

class point
{
protected;
    double x;
    double y;
public:
    point();
    point (double xVal ,double yVal);
    point (point & pt );
    double getX1 ();
    double getY1 ();
    void assign (double xVal ,double yVal);
    point &. assign (point & pt);
};

```

新术语:拷贝构造函数允许你通过拷贝已有对象的数据,建立类中的新对象。

++语言认为构造函数应具有下述特点,并遵循下述规则:

1. 构造函数名与类的名称必须相同。
2. 构造函数没有返回值,因此也就没有函数类型的说明。
3. 类中即可包含一些构造函数,也可没有构造函数。在后者的情况下,编译程序全自动补上一个不做任何事的构造函数。
4. 缺省构造函数就是没有参数或参数表的构造函数。如下面两例:

```

//class use paramterless constructor
class point1
{
protected;
    double x;
    double y;
public;
    point1 ();
    //other member functions
};

//clsaa use constructor with default arguments
class point2

```

```

{
protected:
    double x;
    double y;
public:
    point(double xVal=0,double yVal=0);
    //other member functions
};

```

5. 拷贝构造函数可利用已有的对象来建立一个类中的新对象,如下示例。

```

class point
{
protected:
    double x;
    double y;
public:
    point ();
    point(double xVal,double yVal);
    point(point & pt);
    //other member functions
};

```

6. 类中对象的定义(包括函数参数和局部对象)涉及到构造函数。那个构造函数将被调用?这将决定于你在类中定义了多少构造函数,以及你定义类中对象的方法。例如 point 类中对象的定义

```

point p1; //involves the default constructor
point p2(1.1 1.3); //use the second constructor
point p3(p2); // use the copy constructor

```

因为对象 P1 没有参数,编译程序就调用缺省构造函数。对象 p2 规定了两个 float 型的实参,因此,编译程序调使用第二个构造函数。对象 p3 是以对象 p2 为实参,所以,编译程序用拷贝构造函数从 p2 对象行成 p3 对象。

DO	DON'T
<p>DO 特别是在用类模拟动态数据结构时,一定要定义拷贝构造函数。这些函数的操作方式称为 deep copy (深度拷贝)。缺省的情况下,编译程序生成 shallow copy 构造函数来拷贝类中成员为指针的对象。</p> <p>DON'T 不要用 shallow copy 构造函数来拷贝类中成员为指针的对象。</p>	

11.9 析构函数

C++语言的类包含析构函数,用于自动地删除类中的对象。

语法

析构函数

析构函数的语法为

```
class className
class String
{
public:
    className ;// default constructor
    //other constructors
    ~ className();
    //other member function
};
```

例子:

```
{
protected:
    char* *str
    int len ;
public :
    String ()
    String (String & s);
    ~String ();
    //other member functions
};
```

C++语言认为析构函数具有下列特点,并遵循下列规则:

1. 析构函数的名称必须是~符号加上类的名称。
2. 析构函数没有返回值,因此也没有函数类型说明。
3. 一个类仅能有析构函数。此外,如果类中未定义析构函数,编译程序也会自动地加上一个完全不做任何事的析构函数。
4. 析构函数没有参数。
5. 当类中的析构函数超出范围时,实时系统会自动地执行类的析构函数。

11.10 构造函数和析构函数的应用举例

让我们看看下面这个程序,从中你将可以清楚地看出析构函数以及构造函数的作用。

程序列表11.2 为 CLASS2.CPP 的源程序,主要执行以下功能:

- 生成一个动态数组(即建立一个对象)。

- ☐ 对动态数组中元素赋值
- ☐ 显示动态数组中值
- ☐ 删除动态数组

程序列表11.2 程序 CLASS2.CPP 的源代码

```

1: // Program demonstrates constructors and destructors
2:
3: #include <iostream.h>
4:
5: const unsigned MIN_SIZE = 4;
6:
7: class Array
8: {
9:     protected:
10:         double *dataPtr;
11:         unsigned size;
12:
13:     public:
14:         Array(unsigned Size = MIN_SIZE);
15:         ~Array()
16:         { delete [] dataPtr; }
17:         unsigned getSize() const
18:         { return size; }
19:         void store(double x, unsigned index)
20:         { dataPtr[index] = x; }
21:         double recall(unsigned index)
22:         { return dataPtr[index]; }
23: };
24:
25: Array::Array(unsigned Size)
26: {
27:     size = (Size < MIN_SIZE) ? MIN_SIZE : Size;
28:     dataPtr = new double[size];
29: }
30:
31: main()
32: {
33:     Array Arr(10);
34:     double x;
35:     // assign data to array elements
36:     for (unsigned i = 0; i < Arr.getSize(); i++) {
37:         x = double(i);
38:         x = x * x * 5 * x + 10;
39:         Arr.store(x, i);
40:     }
41:     // display data in the array element
42:     cout << "Array Arr has the following values:\n\n";
43:     for (i = 0; i < Arr.getSize(); i++)
44:         cout << "Arr[" << i << "] = " << Arr.recall(i) << "\n";
45:     return 0;
46: }

```

程序列表11.2中程序执行结果如下:

输出:

Array Arr has the following values;

Arr[0] = 10

Arr[1] = 6

Arr[2] = 4

```
Arr[3] = 4
Arr[4] = 6
Arr[5] = 10
Arr[6] = 16
Arr[7] = 24
Arr[8] = 34
Arr[9] = 46
```

分析:

程序列表11.2中程序定义了全程常量 MIN_SIZE, 用来规定动态数组最小的尺寸。程序在第7行还定义了 Array 类。该类中有两个数据成员: dataPtr 和 size。成员 dataPtr 是指向动态分配给数组的元素的指针。成员 size 存储着类 Array 中对象的元素数目。

类中定义了一个缺省构造函数。(该构造函数实际上用缺省值 MIN_SIZE 作为函数参数)。程序第25至29行定义了构造函数。代替参数 Size 的实参规定数组元素的数目。第27行的语句把参数 Size 和常量 MIN_SIZE 中较大的一值传递给数据成员 size。第28行的语句用操作符 new 分配动态存储空间给数组, 同时把动态数组的基地址赋值给成员 dataPtr。

析构函数 ~Array 通过使用操作符 delete 删除数组的动态存储空间。

成员函数 getSize 在类中定义返回值存入数据成员 size 中。

函数 store 也在类中定义, 用于把参数 x 的值存储在参数为 index 指定的元素中。你可不用判断标号是否超出范围, 从而简化函数的编写。

函数 recall 同样在类中定义。返回标号为 index 的元素值。你也要用上面的方法。简化函数。

主函数 main 定义了类 Array 中的对象 Arr。该对象具有10个元素。同时函数还定义了 double 类型的变量 x。第36行至40行的 for 循环把值存储在对象 Arr 中, 此循环用控制变量 i, 从0变化到 Arr.getSize()-1, 每次增量为1。该循环把信息 getSize 传送给对象 Arr 获得数组中的元素数目。第37行和38行的语句计算对象 Arr 中的元素值。第39行的语句把信息 store 传送给对象 Arr。并传递实参 x 和 i。对象 Arr 把变量 x 中的值保存在第 i 个元素中。

第42行的输出语句利用第43和44行的 for 循环进行输出。该循环用控制变量 i, 从0变化到 Arr.getSize()-1, 每次增量为1。第44行的输出语句显示对象 Arr 中的元素值。

11.11 类层的定义

C++ 语言作为 OOP 设计的语言, 最主要是因为你能根据已有的类, 而生成衍生类。衍生类继承了原始类的成员, 同时, 也能取消继承的某些功能。继承可使你重新使用衍生类中的程序代码。

语法

衍生类

定义衍生类的语法

```
class className: [public] parentClass
{
    <friend classes>
    <private data members>
    <private member functions>
protected:
    <protected data members>
    <protected constructors>
    <protected member functions>
public:
    <public data members>
    <public constructors>
    <public destructor>
    <public member functions>
    <friend functions and friend operators>
};
```

例子:

下面为例子示例了类 cRectangle 及它的衍生类 cBox 的定义:

```
class cRectangle
{
protected:
    double length;
    double width;
public:
    cRectangle(double len, double wide);
    double getLength() const;
    double getWidth() const;
    double assign(double len, double wide);
    double calcArea();
};

class cBox : public cRectangle
{
protected:
    double height;
public:
    cBox(double len, double wide, double height);
    double getHeight() const;
    assign(double len, double wide, double height);
    double calcVolume();
};
```

};

衍生类的定义形式为：“衍生类名:[public]原始类名”。当你用 Public 时,你可允许衍生类中的对象访问原始类中的 public(公共的)成员相反,如果你省略了 public,你就不能允许衍生类中的对象访问原始类中的成员。

衍生类继承了它原始类的数据成员。C++语言无法删除不希望要的继承来的数据成员。但是,C++语言允许取消继承的成员函数。大部分的有关内容将在这章的后面讨论。衍生类可定义新的数据成员、新的成员函数。同时取消继承的成员函数。并且你可把这些成员设置在 private、protected 或 public 段中。

DO	DON'T
DO 用缺省参数的方法,减少构造函数的数目。用成员函数访问数据成员中的数据,这些成员函数允许你控制并判断数据成员中数据的有效性。	
DON'T 不要把类中所有的构造函数都定义在 protected 段中,除非在通过使用 public 定义的衍生类来使用类中的构造函数。不要在 public 段中定义数据成员。	

让我们看看下面这个定义一个小类层的程序示例。程序列表11.3为 CLAASS3.CPP 的源程序。此程序定义的类包括一个类层,用来表示两个简单的几何图形:圆和圆柱。程序不需要输入。它用内部数据生成几何图形,并且显示它们的维数,面积和体积。

程序列表11.3 程序 CLASS3.CPP 的源代码

```
1: // Program that demonstrates a small hierarchy of classes
2:
3: #include <iostream.h>
4: #include <math.h>
5:
6: const double pi = 4 * atan(1);
7:
8: inline double sqr(double x)
9: { return x * x; }
10:
11: class cCircle
12: {
13:     protected:
14:         double radius;
15:
16:     public:
17:         cCircle(double radiusVal = 0) : radius(radiusVal) {}
18:         void setRadius(double radiusVal)
19:         { radius = radiusVal; }
20:         double getRadius() const
21:         { return radius; }
22:         double area() const
23:         { return pi * sqr(radius); }
24:         void showData();
25: };
26:
27: class cCylinder : public cCircle
28: {
29:     protected:
```

```

30:     double height;
31:
32: public:
33:     cCylinder(double heightVal = 0, double radiusVal = 0)
34:         : height(heightVal), cCircle(radiusVal) {}
35:     void setHeight(double heightVal)
36:     { height = heightVal; }
37:     double getHeight() const
38:     { return height; }
39:     double area() const
40:     { return 2 * cCircle::area() +
41:         2 * pi * radius * height; }
42:     void showData();
43: };
44:
45: void cCircle::showData()
46: {
47:     cout << "Circle radius      = " << getRadius() << "\n"
48:         << "Circle area        = " << area() << "\n\n";
49: }
50:
51: void cCylinder::showData()
52: {
53:     cout << "Cylinder radius    = " << getRadius() << "\n"
54:         << "Cylinder height    = " << getHeight() << "\n"
55:         << "Cylinder area      = " << area() << "\n\n";
56: }
57:
58: main()
59: {
60:     cCircle Circle(1);
61:     cCylinder Cylinder(10, 1);
62:
63:     Circle.showData();
64:     Cylinder.showData();
65:     return 0;
66: }

```

程序列表11.3中程序的执行示例如下:

输出结果:

```

Circle radius      =1
Circle area        =3.141593
Cylinder radius    =1
Cylinder height    =10
Cylinder area      =69.115038

```

分析:

程序列表11.3中程序定义了两个类:cCircle类及cCylinder类。cCircle类表示圆,cCylinder类表示圆柱。

Circle类中定义了一个数据成员radius,用于存储圆的半径,此类中还定义了一个构造函数和一些成员函数。构造函数把数据成员radius中赋予值。(当你定义类中对象时,赋值)。注意;构造函数使用新的语法初始化成员radius。函数setRadius和getRadius分别设置和查询成员radius。函数area返回圆的面积。函数showData显示类中对象的半径和面积。

cCylinder类是cCircle类的衍生类。其中也只定义了一个数据成员height,用来存储圆柱的高。此类继承了成员radius,存储圆柱的半径。cCylinder类中定义了一个构造函数和一些成员函数。当生成类中对象时,构造函数给成员radius和height赋值。注

意,成员 height 是用新的语法进行初始化,成员 radius 是用 cCircle 类中的构造函数,调用实参 radiusVal 而进行的初始化,函数 getHight 和 setHeight 分别用于设置和查询成员 height,此类用继承的 setRadius 和 getRadius 处理成员 radias。函数 area 取消了继承的函数 cCircle::area(),返回柱体的表面积。注意,此函数明确地用到继承函数 cCircle::area()。函数 showData 显示类中对象的半径、高及面积。

主函数 main 定义了 cCircle 类中的对象 Circle。并把1赋值给圆的半径。此外函数还定义了 cCylinder 类中的对象 gylinder,把10和1分别赋值给圆柱的高和半径。主函数把 showData 信息传送到对象 circle 和 Cylinder 中。每个对象都进行恰当操作。

11.2 虚拟函数

如前面所提到的,多态性是重要的面向对象设计的特点。考虑下面简单的类和主函数

main:

```
#include<iostream.h>
class cA
{
public:
    double A(double x){return x * x;}
    double B(double x){return A(x)/2;}
};
class cB: public cA
{
public:
    double A(double x){return x * x * x;}
};
main()
{
    cB aB;
    cout<<aB.B(3)<<"\n";
    return 0;
}
```

cA 类包含函数 A 和 B,且 B 中间用函数 A。cB 类是 cA 的衍生类,继承了函数 B,但取消了函数 A。目的在于:用继承函数 cA::B 调用函数 cB::A,从而支持多态性的操作。程序的输出是什么?答案是4.5而不是13.5。为什么?原因在于编译程序处理表达式 aB.B(3)时,是用继承函数 cA::B 调用 cA::A。所以函数 cB::A 没有参与程序的执行。该程序不能支持多态性操作。

C++语言是用虚拟函数支持多态性操作的。

新术语:虚拟函数与实时系统有关。定义形成为在函数返回类型前键入 virtual。

一旦你定义了虚拟函数,在衍生类中,就只能用虚拟函数来取消继承的虚拟函数。且

这些虚拟函数必须有相同的参数表。虚拟函数也可取消原始类中的非虚拟函数。

语法

虚拟函数

定义虚拟函数的语法是

```
class className1
{
    //member functions
    virtual returnType functionName(<parameter list>;
};
class className2:public className1
{
    //member functions
    virtual returnType functionName(<parameter list>;
};
```

例子:

这个示例说明了虚拟函数是如何成功地在 cA 类和 cB 类中进行多态性操作的。

```
#include <iostream.h>
class cA
{
public:
    virtual double A(double x){
        return x * x;
    }
    double B(double x){return A(x)/2;}
};
class cB : public cA
{
public:
    virtual double A(double x){return x * x * x;}
};
main()
{
    cB aB;
    cout<<aB.B(3)<<"\n";
    return 0;
}
```

这个示例显示结果13.5,因为此次继承函数 cB::B 是调用 cB::A。

DO	DON'T
DO 当你定义一个可调用的函数完成有关类的特定操作时,使用虚拟函数定义。定义虚拟函数可确保它选择适当的类进行正确的操作。	DON'T 不要用缺省的方法把成员函数定义为虚拟函数,虚拟函数有许多附加的额外操作。

让我们看看下面示例,程序列表11.4为 CLASS4.CPP 的源程序。该程序生成一个正方形和一个长方形,并显示他们的维数和面积。程序不需要输入数据。

程序列表11.4 程序 CLASS4.CPP 的源代码

```

1: // Program that demonstrates virtual functions
2:
3: #include <iostream.h>
4:
5: class cSquare
6: {
7:     protected:
8:         double length;
9:
10:    public:
11:        cSquare(double len) { length = len; }
12:        double getLength() { return length; }
13:        virtual double getWidth() { return length; }
14:        double getArea() { return getLength() * getLength(); }
15: };
16:
17: class cRectangle : public cSquare
18: {
19:     protected:
20:         double width;
21:
22:    public:
23:        cRectangle(double len, double wide) :
24:            cSquare(len), width(wide) {}
25:        virtual double getWidth() { return width; }
26: };
27:
28: main()
29: {
30:     cSquare square(10);
31:     cRectangle rectangle(10, 12);
32:
33:     cout << "Square has length = " << square.getLength() << "\n"
34:           << "          and area   = " << square.getArea() << "\n";
35:     cout << "Rectangle has length = "
36:           << rectangle.getLength() << "\n"
37:           << "          and width = "
38:           << rectangle.getWidth() << "\n"
39:           << "          and area   = "
40:           << rectangle.getArea() << "\n";
41:     return 0;
42: }

```

程序列表11.4中程序的执行示例如下:

输出结果:

```
square has length = 10
and area = 100
Rectangle has length = 10
and width = 12
and area = 120
```

分析:

程序列表11.4中的程序定义了两个类:csquare 类和 cRectangle 类,用以分别表示正方形和长方形。类 cSquare 仅定义了一个数据成员 length,用来存储正方形的边长。该类中还定义了参数为 len 的构造函数,把实参传递给成员 length。此外,该类还定义了函数 getLength,getWidth 和 getArea。函数 getLength 和 getWidth 返回一值存入成员 length 中。注意,类中把函数 getWidth 定义为虚拟函数。函数 getArea 返回正方形的面积值。通过调用函数 getLength 和 getWidth 可计算求得面积。我们调用这些函数而不利用数据成员,目的是为了说明虚拟函数 getWidth 是如何操作的。

程序定义类 cRectangle 类为类 cCircle 的衍生类。类 cRectangle 中定义了数据成员 width 和,并继承了成员 length,这些函数用来存储矩形的维数、长和宽。类中的构造函数具有参数 len 和 width,对成员 len 和 wide 赋值。注意:构造函数调用 cSquare 的构造函数,实参为 len。构造函数初始化数据成员 width 为参数 wide 的值。

类 cRectangle 中定义了虚拟函数 getWidth,函数的返回值存储在数据成员 width 中。此类继承了成员函数 getLength 和 getArea。因为这些函数也适用于类 cRectangle。

主函数 main 定义了 cSquare 类中的对象 square,这个对象边长为10。此外主函数还定义了 cRectangle 类中的对象 rectangle。此对象的长为10,宽为12。

程序第33和34行的输出语句显示对象 square 的边长和面积。该语句把信息 getLength 和 getArea 传送给对象 square,得到适当的操作。对象 square 在调用函数 getArea 时。分别调用了函数 cSquare::getLength 和 cSquare::getWidth。

第35至40行的输出语句显示对象 rectangle 的长、宽和面积。该句把信息 getLength, getWidth 和 getArea 传送给对象 rectangle,对象 rectangle 就调用继承函数 cSquare::getLength、虚拟函数 cRectangle::getWidth 和继承函数 cSquare::getArea。而函数 cSquare::getArea 又调用了继承函数 cSquare::getLength 和虚拟函数 cRectangle::getWidth,来正确计算矩形的面积。

DO	DON'T
DO 把析构造函数定义了虚拟函数。这就能够做到删掉类中对象的多态性操作。此外,它还能使你定义一个拷贝构造函数和一个访问操作符,用于任何类。	
DON'T 当正确生成衍生类时,不要忘记你能继承虚拟函数和析构造函数。你不需要另定义外壳函数和析构造函数调用原始类中的成员。	

11.13 虚拟函数的规则

定义虚拟函数的规则是“一旦为虚拟,永远为虚拟”。换句话说,一旦你在类中把一个函数定义为虚拟函数,那么任何子类也只有用另外与它具有相同参数表的虚拟函数来取消它。`virtual` 定义说明是衍生类中的强制性说明。首先,这个规则看起来是约束了你,这种约束用于面向对象设计将支持虚拟函数。在 C++ 语言中,这个规则很有趣,你可以定义非虚拟函数和重载函数与虚拟函数同名,但是参数表必须不同。而且,你不能继承一个与虚拟函数同名的非虚拟函数。下面示例就说明了这点:

```
#include <iostream.h>
class cA
{
public:
    cA() {}
    virtual void foo(char c)
        { cout<<"cA::foo() returns"<<c<<"\n"; }
};
class cB : public cA
{
public:
    cB() {}
    void foo(const char * s)
        { cout<<"cB::foo() returns"<<s<<"\n"; }
    virtual void foo(char c)
        { cout<<"virtual cB::foo() returns"<<c<<"\n"; }
};
class cC : public cB
{
public:
    cC() {}
    void foo(const char * s)
        { cout<<"cC::foo() returns"<<s<<"\n"; }
    void foo(int i)
        { cout<<"cC::foo() returns"<<i<<"\n"; }
    virtual void foo(char c)
        { cout<<"virtual cC::foo() returns"<<c<<"\n"; }
};
main()
{
    int n = 100;
    cA Aobj;
    cB Bobj;
```

```

cC Cobj;

Aobj.foo('A');
Bobj.foo('B');
Bobj.foo(10);
Bobj.foo("Bobj");
Cobj.foo('C');
// if you uncomment the next statement, program does not compile
// Cobj.foo(n);
Cobj.foo(144, 123);
Cobj.foo("Cobj");
return 0;
}

```

这段程序定义了三类: cA, cB 和 cC, 形成线性类层。类 cA 中定义了函数 foo(char) 为虚拟函数。类 cB 也定义了它自己的虚拟函数 foo(Char)。此外, 类 cB 定义了非虚拟重载函数 foo(const char * s) 和 foo(int)。类 cC 是类 cB 的衍生类。其中定义了虚拟函数 foo(char) 和非虚拟重载函数 foo(const char *) 和 foo(double)。注意, 类 cC 必须定义 foo(const char *) 函数, 因为它不能继承成员函数 cB::foo(const char *)。C++ 语言当一个重载虚拟函数被调用时, 支持不同的函数继承规则。主函数 main 生成了三个对象, 分属三类。并调用不同的成员函数。

11.14 友元函数

C++ 语言允许成员函数访问类中所有的数据成员。此外, C++ 语言也允许友元函数访问类中所有的数据成员。友元函数在类中定义, 定义形式为: 在函数名前键入 friend。友元函数看起来类似于成员函数, 但不能把一个引用返回友元的类。因为这需要返回自引用 * this。然而, 当你在友元类的外面定义它们的友元函数时, 你就不需要把函数名定义成与类名相同。

新术语: 友元函数是普通函数, 可访问一个或多个类中的数据成员。

语法

友元函数

友元函数的形式为:

```

class class Name
{
public:
    className();
    // other constructors
    friend returnType friendFunction(<parameter list>);

```

```
};
```

例子:

```
class String
{
    protected:
        char *str;
        int len;
    public:
        String();
        ~String();
        // other member functions
        friend String& append(String& str1, String& str2);
        friend String& append(const char * str1, String& str2);
        friend String& append(String& str1, const char * str2);
};
```

友元类能够完成枯燥、烦杂、困难甚至由成员函数不可能完成的一些功能。

让我们看看下面一个简单的友元函数的示例。程序列表11.5为 CLASS5.CPP 的源程序。程序内部生成两个复数,求和后,将结果存入另一复数。然后显示复数的操作及结果。

程序列表11.5 程序 CLASS5.CPP 的源代码

```
1: // Program that demonstrates friend functions
2:
3: #include <iostream.h>
4:
5: class Complex
6: {
7:     protected:
8:         double x;
9:         double y;
10:
11:     public:
12:         Complex(double real = 0, double imag = 0);
13:         Complex(Complex& c) { assign(c); }
14:         void assign(Complex& c);
15:         double getReal() const { return x; }
16:         double getImag() const { return y; }
17:         friend Complex add(Complex& c1, Complex& c2);
18: };
19:
20: Complex::Complex(double real, double imag)
21: {
22:     x = real;
23:     y = imag;
24: }
25:
26: void Complex::assign(Complex& c)
27: {
28:     x = c.x;
29:     y = c.y;
30: }
31:
32: Complex add(Complex& c1, Complex& c2)
```

```

33: {
34:     Complex result(c1);
35:
36:     result.x += c2.x;
37:     result.y += c2.y;
38:     return result;
39: }
40:
41: main()
42: {
43:     Complex c1(2, 3);
44:     Complex c2(5, 7);
45:     Complex c3;
46:
47:     c3.assign(add(c1, c2));
48:     cout << "(" << c1.getReal() << " + i" << c1.getImag() << ")"
49:          << " + "
50:          << "(" << c2.getReal() << " + i" << c2.getImag() << ")"
51:          << " = "
52:          << "(" << c3.getReal() << " + i" << c3.getImag() << ")"
53:          << "\n\n";
54:     return 0;
55: }

```

程序列表11.5中程序的执行结果如下:

输出:

$(2 + i3) + (5 i7) = (7 + i10)$

分析:

程序列表11.5中程序定义了 Complex 类,表示复数。此类定义了两个数据成员、两个构造函数、一个友元函数和一组成员函数。数据成员 x,y 分别存储复数的实部与虚部。

类中的两个构造函数,第一个具有两个参数(用缺省参数),允许你用复数的实部和虚部建立类的对象。因为两个参数有缺省值,构造函数也可看为缺省构造函数。第二个构造函数 complex(complex &)是拷贝构造函数。

类 complex 定义了三个成员函数。函数 assign 把类中对象拷贝生成另一个对象。函数 getReal 和 getImag 分别把返回值存储在成员 real 和 imag 中。

类 Complex 还定义了友元函数 add 来对两个复数求和。为了程序简短,我们没有用友元函数执行差、积、商的功能。友元函数 add 有什么特殊的地方呢?为什么不用普通的成员函数来对对象求和呢?下面的 add 成员函数的定义就可回答这些问题:

```
complex&. add(complex& c)
```

这个定义说明函数把参数 c 作为第二个操作数。下面显示了成员函数 add 是如何工作的:

```
complex c1(3,4),c2(1.2,4.5);
```

```
c1.add(c2); // adds c2 to c1
```

首先,成员函数 add 的功能是作为增量,而不是作为附加的函数。其次,目标类的对象总是第一个操作数。这一点对于加法和乘法操作不是问题,但对于减法和除法运算却很重要。这就是为什么友元函数 add 的优越之处。

友元函数 add 返回一个对象。函数生成一个 complex 类的局部对象,并返回此对象。

主函数 main 用成员函数 assign 和友元函数 add 执行简单的复数运算操作。此外,主函数 main 还用函数 getReal 和 getImag 处理类 Complex 的不同对象,显示各对象的组成。

11.15 操作符和友元操作符

上面的程序用一个成员函数和一个友元函数进行复数的运算操作。这种方法在 C 语言和 Pascal 语言中很典型,很常用。因为这两种语言不支持用户定义的操作符。但是,C++ 语言允许你定义操作符和友元操作符。这些操作符包括 +, -, *, /, %, ==, !=, <, <=, <, >=, >, +=, -=, *=, /=, %=, [], (), <<, 和 >>。参考有关 C++ 语言书以了解这些操作符的使用规则。C++ 语言把这些操作符和友元操作符定义为特殊的成员函数和友元函数函数来外理的。

语法

操作符与友元操作符

操作符与友元操作符的定义语法为

```
class className
{
    public:
        // constructors and destructor
        // member functions

        // unary operator
        returnType operator
operatorSymbol(operand);
        // binary operator
        returnType operator
operatorSymbol(firstOperand,
secondOperand);
        // unary friend operator
        friend returnType operator
operatorSymbol(operand);
        // binary operator
        friend returnType operator
operatorSymbol(firstOperand,
secondOperand);
};
```

例子:

```
class String
{
    protected:
        char *str;
        int len;
    public:
        String();
        ~String();
        // other member functions
        // assignment operator
```

```

        String& operator =(String& s);
        String& operator +=(String& s);
        // concatenation operators
        friend String& operator +(String& s1,
String& s2);
        friend String& operator +(const char* s1,
String& s2);
        friend String& operator +(String& s1, const
char* s2);
        // relational operators
        friend int operator >(String& s1, String&
s2);
        friend int operator >=(String& s1, String&
s2);
        friend int operator <(String& s1, String&
s2);
        friend int operator <=(String& s1, String&
s2);
        friend int operator ==(String& s1, String&
s2);
        friend int operator !=(String& s1, String&
s2);
    };

```

你编写函数时,可以就像使用先前定义的那些操作符一样,使用定义的操作符和友元操作符。所以,你用这些操作符可支持类的操作。例如:复数、串、数组或矩阵的操作。这些操作符使你编写的表达式的可读性优于使用函数编写的表达式。

让我们看看下面示例,程序列表10.6为 CLASS6.CPP 的源程序。该程序是对程序列表11.5修改而得的。新的程序可进行多组求和运算。

程序列表11.6 程序 CLASS6.CPP 的源代码

```

1: // Program that demonstrates operators and friend operators
2:
3: #include <iostream.h>
4:
5: class Complex
6: {
7:     protected:
8:         double x;
9:         double y;
10:
11:     public:
12:         Complex(double real = 0, double imag = 0)
13:         { assign(real, imag); }
14:         Complex(Complex& c);
15:         void assign(double real = 0, double imag = 0);
16:         double getReal() const { return x; }
17:         double getImag() const { return y; }
18:         Complex& operator =(Complex& c);
19:         Complex& operator +=(Complex& c);
20:         friend Complex operator +(Complex& c1, Complex& c2);
21:         friend ostream& operator <<(ostream& os, Complex& c);
22: };
23:
24: Complex::Complex(Complex& c)
25: {
26:     x = c.x;
27:     y = c.y;
28: }
29:
30: ...

```

```

30: void Complex::assign(double real, double imag)
31: {
32:     x = real;
33:     y = imag;
34: }
35:
36: Complex& Complex::operator =(Complex& c)
37: {
38:     x = c.x;
39:     y = c.y;
40:     return *this;
41: }
42:
43: Complex& Complex::operator +=(Complex& c)
44: {
45:     x += c.x;
46:     y += c.y;
47:     return *this;
48: }
49:
50: Complex operator +(Complex& c1, Complex& c2)
51: {
52:     Complex result(c1);
53:
54:     result.x += c2.x;
55:     result.y += c2.y;
56:     return result;
57: }
58:
59: ostream& operator <<(ostream& os, Complex& c)
60: {
61:     os << "(" << c.x << " + i" << c.y << ")";
62:     return os;
63: }
64:
65: main()
66: {
67:     Complex c1(3, 5);
68:     Complex c2(7, 5);
69:     Complex c3;
70:     Complex c4(2, 3);
71:
72:     c3 = c1 + c2;
73:     cout << c1 << " + " << c2 << " = " << c3 << "\n";
74:     cout << c3 << " + " << c4 << " = ";
75:     c3 += c4;
76:     cout << c3 << "\n";
77:     return 0;
78: }

```

程序列表11.6中程序的执行结果如下：

输出：

$(3 + i5) + (7 + i5) = (10 + i10)$

$(10 + i10) + (2 + i3) = (12 + i13)$

分析：

新类 complex 中用操作符=代替 assign(complex&)成员函数；用友元操作符+代替友元函数 add；

```
Complex& operator=(Complex& c);
```

```
friend Complex operator+(Complex& c1,Complex& c2);
```

操作符=具有一个参数,为类 Complex 中对象的引用(reference),并且把此引用返回同一类中。友元操作符+具有两个参数(都是类 Complex 中对象的引用),并且得到 complex 类型返回值。

在此,再增加两个操作符:

```
complex & operator+=(complex& c);
```

```
friend ostream& operator<<(ostream& os, complex& c);
```

操作符+=是类 Complex 的成员。它有一个参数,为类 Complex 中对象的引用。并且得到同类的引用。另一个新的操作符是友元操作符<<,为类定义了一个流抽取操作。友元操作符具有两个参数:一个是 ostream 类的引用,另一个是 complex 类的引用。操作符<<返回 ostream 类的引用。这种类型的值使你把流输出与其它事先定义的类型或类联系起来。友元操作符的定义有两个语句,第一句输出串和类 Complex 中的数据成员到输出流参数 os 中。操作符<<允许访问 Complex 类型参数 c 的 real 和 Imag 数据成员。第二句用来返回第一个参数 os。

主函数 main 定义了 Complex 类中对象:c1,c2,c3和 c4,其中 c1,c2和 c4是通过把非缺省值赋于数据成员 real 和 imag 而建立的。函数测试使用操作符=,+,<<,+=。此程序说明,你可利用这些新定义的操作符和友元操作符来编写具有更好的可读性的程序,同时支持高层次的抽象。

11.16 小结

这章我们介绍了 C++ 语言的类,主要讨论了从以下几个内容:

- ☐ 面向对象设计的基础是:类,对象,信息,方法,继承和多态性。
- ☐ 你可建立一个基类,并规定各个 private,protected 和 public 成员。C++ 语言的类包括数据成员及成员函数。数据成员存储类中对象的状态,成员函数查阅和操作各种状态。
- ☐ 构造函数和析构函数支持自动建立和删除类中对象。构造函数是特殊的成员,构造函数名必须与类名相同。你可以定义多个构造函数,或不定义构造函数。后者,编译程序自动生成一个构造函数。每个构造函数都可使你按不同方式建立类的对象。有两种特殊类型的构造函数:缺省构造函数和拷贝构造函数。与构造函数不同,C++ 语言仅允许你定义一个无参数的析构函数,析构函数可自动地删除类的对象。当类中对象超出范围时,实时系统能自动地调用构造函数和析构函数。
- ☐ 类层的定义使你能从已有的类得到它的衍生类。衍生类继承了原始类中的成员。C++ 语言的衍生类可通过定义自己的版本,而取消某些继承的成员函数。如果你取消一个非虚拟函数,你可以用不同的参数表定义新的版本,但是,你不能改变一个继承的虚拟函数的参数表。
- ☐ 虚拟成员函数可使你的类支持多态性操作。这些操作对类层中的各类进行正确的操作。一旦你把函数定义为虚拟函数,那么在衍生类中,你只能用虚拟函数来取消

它。在类层中,一个虚拟函数的所有版本必须具有相同的参数表。

- ☐ 友元函数是特殊的非成员函数,可用来访问 `private` 和 `private` 数据成员。这些函数可使你进行许多成员函数不可能做到的操作。
- ☐ 操作符和友元操作符使你支持不同的操作。如加法,赋值和检索等。这些操作使你的类进行抽象运算。此外,这些操作符的使用,也大大增加了程序的可读性。

11.17 问与答

问:如果把缺省、拷贝和其它构造函数定义成 `protected` 型,会怎么样?

答:用户程序不能建立这种类的对象。然而,用户程序通过 `public` 构造函数,定义该类的衍生类,则可使用该类。

问:能用构造函数定义对象吗?

答:能,你可用此法建立类中对象。例如,如果类 `complex` 具有构造函数 `Complex(double real, double imag)`,那么你可如下定义 `complex` 类中的对象:

```
Complex c = Complex(1.7, 2.4);
```

问:你能把信息与对象联系起来吗?

答:可以,只要信息调用成员函数,返回一个接收此信息同类的引用。例如,如果你有一个 `string` 类,具有下列成员函数:

```
String& upperCase();
```

```
String& reverse();
```

```
String& mapChars(char find, char replace);
```

你就可编写以下语句:

```
ss.upperCase().reverse().mapChar(' ', '+');
```

问:如果一个类由拷贝构造函数生成,即为用编译程序拷贝类中对象。分类中包含指针时会出现什么情况?

答:这些构造函数一个字节,一个字节地拷贝。因此,在两个对象中,相应的指针成员指向同一动态数据。这种复制过程容易出错。

问:能否建立对象数组?

答:可以。然而,这种类必须具有缺省构造函数。

问:能否用指针建立类中对象?

答:可以。你需要用操作符 `new` 和 `delete` 来分配或删除动态存储空间给该对象。如下示例说明此点:

```
Complex * pC;
```

```
pC = new Complex;
```

```
// manipulate the instance accessed by pointer pC delete pC;
```

Or

```
Complex * pC = new Complex;
```

```
// manipulate the instance accessed by pointer pC
```

```
delete pC;
```

11.8 专题讨论

本章讨论提供了一些测验问题, 友元你巩固本章所学内容, 此外, 还附加了一些, 通过实践编程, 使你进一步理解本章的内容。在进行下一章的学习之前, 必须理解测验及练习的答案。在附录 B 中给出了答案。

11.8.1 测验

1. 下面类的定义有何错误?

```
class String{
    char * str;
    unsigned len;
    String();
    String(String& s);
    String(unsigned size, char = '\0');
    String(unsigned size);
    String& assign(String& s);
    ~String();
    unsigned getLen() const;
    char * getString();
    // other member functions
};
```

2. 下面类的定义有何错误?

```
class String{
    protected:
        char * str;
        unsigned len;
    public:
        String();
        String(const char * s);
        String(String& s);
        String(unsigned size, char = '\0');
        String(unsigned size);
        ~String();
        // other member functions
};
```

3. 用下面语句定义类 String 中的对象, 是否正确?

```
s = String("Hello Turbo C++");
```

4. 参看程序 CLASS6.CPP, 如果你用如下方式改变主函数 main 中的对象定义, 程序会编译吗?

```
Complex c1 = Complex(3,5);
```

```
Complex c2 = Complex(7,5);  
Complex c3 = c1;  
Complex c4 = Complex(2,3);
```

11.18.2 练习

修改程序 CLASS6. CPP 以生成程序 CLASS7. CPP。把 c1到 c4四个独立的对象用一个对象数组 c 代替。

第十二章 基本的流文件 I/O(第十二天)

在本章中,将向你介绍使用 C++ 语言的流库进行文件的 I/O 操作。尽管 C 语言的 `STDIO.H` 库已被认为是 ANSI C 标准。但 C++ 语言的流库仍然没被认为是 ANSI C 标准,你可根据需要,在 `STDIO.H` 库和 C++ 的流库中选择文件 I/O 函数。这两个 I/O 库都提供很强的功能和很大程度上的灵活性。这章我们主要讨论一些基本的和实际的读,写文件的操作。主要内容如下:

- ☐ 普通流 I/O 函数
- ☐ 文本文件的顺序流 I/O
- ☐ 二进制文件的顺序流 I/O
- ☐ 二进制文件的随机存取流 I/O

为了了解 C++ 语言流库的更多内容,可参考 C++ 语言参考书。如 Tom Swan 的《C++ Primer》(Sams Publishing,1992)。

12.1 C++ 语言流库

C++ 语言的流 I/O 库是由许多在头文件中定义的类的层次结构组成的。`IOSTREAM.H` 头文件是我们目前文件用到的唯一的库文件。其它库文件还包括:`IO.H`, `ISTREAM.H`, `OSTREAM.H`, `IFSTREAM.H`, `OFSTREAM.H` 和 `FSTREAM.H`。`IO.H` 头文件定义了低层次的类和标识符。头文件 `ISTREAM.H` 和 `OSTREAM.H` 支持基本的输入/输出流类。`IOSTREAM.H` 已含前面两个头文件中类的操作符。类似地, `IFSTREAM.H` 和 `OFSTREAM.H` 头文件支持基本的文件输入/输出流类。`FSTREAM.H` 文件中包含前面两个头文件中类的操作符。其它的流库文件提供一些特定的流 I/O 操作。C++ ANSI 委员会将规定标准流 I/O 库。这将结束一些混乱的看法,认为类和头文件是标准流库的一部分。

12.2 通用流 I/O 函数

这节中,我们讨论一些通用的流 I/O 函数,包括顺序和随机存取 I/O。这些函数包括 `open`, `close`, `good`, `fail`, 此外还有一个操作符 `!`。

函数 `open` 使你打开一个文件流用于输入、输出、追加或输入和输出。函数也允许你规定有关的 I/O 是二进制文件,还是文本文件。

语法

`open` 函数

`open` 函数的函数原型为


```
void open (const char * filename,
int mode,
int m = filebuf::openprot);
```

参数 filename 规定将要打开的文件名。参数 mode 规定 I/O 模式。以下是参数 mode 可选择的由 IO.H 头文件提供的参数表:

in	打开流用于输入。
out	打开流用于输出。
ate	设置指向文件尾的流指针。
app	打开流用于追加。
trunc	如果文件已经存在,则截断文件使之长度为 0。
nocreate	如果文件不存在,则发出错误信息。
noreplace	如果文件存在,则发出错误信息。
binary	按二进制模式打开。

例子:

```
// open stream for input
fstream f;
f.open("\\AUTOEXEC. BAT",ios::in);
// open stream for output
fstream f;
f.open("\\AUTOEXEC. OLD",ios::out);
// open stream for binary input and output
fstream f;
f.open("INCOME. DAT",ios::in | ios::out | ios::binary);
```

注意:文件流类提供构造函数,其函数 open 的一些动作并包括相同的参数。

函数 close 关闭流并恢复所涉及的设备。这些设备包括用于流 I/O 操作的存储区。

语法

close 函数

函数 close 的函数原型为

```
void close();
```

例子:

```
fstream f;
// open stream
f.open("\\AUTOEXEC. BAT",ios::in);
// process file
// now close stream
f.close();
```

C++ 语言流库还包括一些用于检查流操作错误状态的基本函数。这些函数如下:

1. 函数 good():如果流操作中没有错误,则返回一个非零值。函数 good 定义如下:

```
int good();
```

2. 函数 fail(): 如果流操作中有一个错误, 则函数返回一个非零值。函数 fail 定义如下:

```
int fail();
```

3. 重载操作符!: 用于流操作中, 判断错误状态。

C++ 流库也提供了另外一些函数, 用来查询其它方面和流操作错误类型。

12.3 文本文件的顺序流 I/O

文本文件的顺序流 I/O 的函数和操作符很简单。在前面, 你已经涉及到许多。它所包括的函数和操作符如下:

1. 流提取符 <<, 把字符串和字符写到流中。
2. 流插入符 >>, 从流中读字符。
3. 函数 getline, 从流中读字符串。

语法

getline 函数

函数 getline 的函数原型为

```
istream& getline(signed char * buffer,  
                 int size,  
                 char delimiter = '\n');  
istream& getline(unsigned char * buffer,  
                  int size,  
                  char delimiter = '\n');
```

参数 buffer 是一个指向接受流中字符的字符串的指针。参数 size 规定可读的最多字符数目。参数 delimiter 规定定界字符。它将在参数 size 所规定的字符读取完之前, 终止串的输入。参数 delimiter 缺省值为 '\n'。

例子:

```
fstream f;  
char textLine[MAX];  
f.open("\\\\CONFIG.SYS", ios::in);  
while (! f.eof()) {  
    f.getline(textLine, MAX);  
    cout << textLine << "\n";  
}  
f.close();
```

让我们看看下面示例。程序列表 12.1 为 IO1.CPP 的源程序。该程序执行如下功能:

- ☐ 请你输入用于输出的文本文件名, 它必须是一个已经存在的文本文件。
- ☐ 请你输入用于输出的文本文件名(程序判断你所输入的文件名是否相同。如果相同, 请你重新输入不同的用于输出的文本文件名)。

- ☐ 从输入文件中逐行读出,且删除每行结尾的空白符。
- ☐ 逐行写入输出文件或标准输出窗口。

程序列表 12.1 程序 IO1.CPP 的源代码

```
1: // C++ program that demonstrates sequential file I/O
2:
3: #include <iostream.h>
4: #include <fstream.h>
5: #include <string.h>
6:
7: enum boolean { false, true };
8:
9: const unsigned LINE_SIZE = 128;
10: const unsigned NAME_SIZE = 64;
11:
12: void trimStr(char* s)
13: {
14:     int i = strlen(s) - 1;
15:     // locate the character where the trailing spaces begin
16:     while (i >= 0 && s[i] == ' ')
17:         i--;
18:     // truncate string
19:     s[i+1] = '\0';
20: }
21:
22: void getInputFilename(char* inFile, fstream& f)
23: {
24:     boolean ok;
25:
26:     do {
27:         ok = true;
28:         cout << "Enter input file : ";
29:         cin.getline(inFile, NAME_SIZE);
30:         f.open(inFile, ios::in);
31:         if (!f) {
32:             cout << "Cannot open file " << inFile << "\n\n";
33:             ok = false;
34:         }
35:     } while (!ok);
36:
37: }
38:
39: void getOutputFilename(char* outFile, const char* inFile,
40:                        fstream& f)
41: {
42:     boolean ok;
43:
44:     do {
45:         ok = true;
46:         cout << "Enter output file : ";
47:         cin.getline(outFile, NAME_SIZE);
48:         if (strcmp(inFile, outFile) != 0) {
49:             f.open(outFile, ios::out);
50:             if (!f) {
51:                 cout << "File " << outFile << " is invalid\n\n";
52:                 ok = false;
53:             }
54:         }
55:         else {
56:             cout << "Input and output files must be different!\n";
57:             ok = false;
58:         }
59:     } while (!ok);
60: }
61:
```

```

62: void processLines(fstream& fin, fstream& fout)
63: {
64:     char line[LINE_SIZE + 1];
65:
66:     // loop to trim trailing spaces
67:     while (fin.getline(line, LINE_SIZE)) {
68:         trimStr(line);
69:         // write line to the output file
70:         fout << line << "\n";
71:         // echo updated line to the output window
72:         cout << line << "\n";
73:     }
74:
75: }
76: main()
77: {
78:
79:     fstream fin, fout;
80:     char inFile[NAME_SIZE + 1], outFile[NAME_SIZE + 1];
81:
82:     getInputFilename(inFile, fin);
83:     getOutputFilename(outFile, inFile, fout);
84:     processLines(fin, fout);
85:     // close streams
86:     fin.close();
87:     fout.close();
88:     return 0;
89: }

```

程序列表 12.1 中程序执行结果如下:

输出:

Enter input file : sample.txt

Enter output file : sample.out

This is line 1

This is line 2

This is line 3

This is line 4

分析:

程序列表 12.1 中程序没有定义类,而是完全用文件流来输入、输出文本文件。程序定义了函数 trimStr, getInputFilename, getOutputFilename, processLines 和主函数 main。

函数 trimStr 通过参数 s 截断字符串尾部的空格。该函数定义了局部变量 i,并把空结束符之前的字符序号赋值给 i。在第 16 行函数用 while 循环从后向前查询字符串中字符,寻找第一个非空格字符。第 19 行中的语句把空结束符赋值给串 s 中最后一个非空格字符的下一个字符。

函数 getInputFilename 得到用于输入的文件名,并打开相应的输入文件流。参数 inFile 把用于输入的文件名传递给函数 caller。引用参数 f 把打开的输入文件流传递给函数 caller,函数 getInputFilename 定义了局部标志 ok。该函数在第 26 至 35 行用 do-while 循环来获得一个有效的文件名,打开文件用于输入。第 27 行是循环体的第一条语句。它把枚举值 true 赋给局部变量 ok。第 28 行的输出语句的提示信息请你输入用于输入的文件名。第 29 行的语句调用流输入函数 getline 来获得你的输入,并将它

存储在参数 `inFile` 中。第 30 行的语句用流参数 `f` 打开输入文件。`open` 语句用 `ios::in` 值说明流被打开,用于文本输入。第 31 行的 `if` 语句判断流 `f` 是否被成功地打开,如果失败,函数执行第 32 和 33 行的语句。这些语句显示一个错误信息,并把枚举值 `false` 赋予局部变量 `ok`。第 35 行的 `while` 从句判断 `! OK` 的状态。循环内容重复进行,直到你输入有效的文件名。这个文件名可成功地被打开用于输入。

函数 `getOutputFilename` 具有三个参数。参数 `outFile` 把用于输出的文件名传递给函数调用者。参数 `inFile` 把用于输入的文件名提供给函数。函数使用这个参数用于确保用于输入和输出的文件名不同。参数 `f` 把输出流传递给函数调用者。函数 `getOutputFilename` 的执行过程很简单,类似于函数 `getInputFilename`,其主要不同点在于函数 `getOutputFilename` 调用函数 `Stricmp` 以比较参数 `inFile` 和 `outFile` 中的值。函数用 `Stricmp` 的结果决定是否用于输入和输出的文件名相同。如果相同,函数执行第 57 和 58 行的语句。这些语句显示错误信息,并把 `false` 赋值给局部变量 `ok`。

函数 `processLine` 从输入文件流中逐行读出,截断,并把读出的内容写入输出文件流。参数 `fin` 和 `fout` 分别传递输入和输出文件流。函数定义了局部字符串变量 `line`,在程序第 67 至 73 行用 `while` 循环来处理每行文本中的内容。`while` 从句调用函数 `getline`,读出输入流 `fin` 的下一行内容,并把它赋值给变量 `line`,函数 `getline` 的结果在文本读完时会使 `while` 循环终止。循环体中的第一条语句调用函数 `trimStr`,实参为 `line`。第 70 行的语句把变量 `line` 中的串写入输出文件流。第 72 行的语句把变量 `line` 中的串回显到标准输出窗口。在此处,设置这条语句以便你能监视程序执行的步骤。

主函数 `main` 定义了文件流 `fin` 和 `fout`,以及字符串变量 `inFile` 和 `outFile`。第 82 行的语句调用函数 `getInputFilename`,实参为 `inFile` 和 `tin`。这个调用,分别通过实参 `inFile` 和 `lin` 输入文件和输入文件流的名字。第 83 行的语句调用函数 `getOutputFilename`,实参为 `outFile` 和 `lout`,这个调用,分别通过实参 `outFile` 和 `fout` 获得用于输出的文件名和输出文件流。第 84 行的语句调用函数 `ProcessLines`,实参为 `fin` 和 `fout`。这个调用读出输入文件流 `fin`,并把结果写入输出文件流 `fout`。第 86 和 87 分别关闭两个文件流。

12.4 二进制文件顺序流 I/O

C++ 流库提供重载流函数 `write` 和 `read`,用于顺序二进制流 I/O。函数 `write` 把多个字节传给一个输出文件流。这个函数能把任何变量和示例写入流中。

语法

write 函数

重载函数 `write` 的函数原型为:

```
ostream& write(const signed char * buff, int num);
```

```
ostream& write(const unsigned char * buff, int num);
```

参数 `buff` 是一个指向缓冲区的指针。该缓冲区内为将要送到输出流的字符数据。参数 `num` 为缓冲区中字节数。

例子:

```
const MAX = 80;
char buff[MAX+1] = "Hello World!";
int len = strlen(buffer) + 1;
fstream f;
f.open("CALC.DAT",ios::out | ios::binary);
f.write((const unsigned char *) * len, sizeof(len));
f.write((const unsigned char *)buff,len);
f.close();
```

函数 read 从输入流中接收多个字节。这个函数能从流中读出任何变量。

语法

函数 read

重载函数 read 的函数原型为

```
istream&. read (signed char * buff, int num);
istream&. read (unsigned char * buff, int num);
```

参数 buff 是一个指向缓冲区的指针,这个缓冲区用于接收从输入流中读出的数据,
参数 num 是缓冲区的字节数。

```
const MAX = 80;
char buff [MAX+1];
int len;
fstream f;
f.open ("CALC.DAT",ios::in | ios::binary);
f.read ((const unsigned char *) * len, sizeof(len));
f.read ((const unsigned char *)buff,Len);
f.close();
```

让我们看看下面这个示例,程序列表 12.2 为 IO2.CPP 的源程序。程序定义了一个类来模仿动态数字数组。流 I/O 操作使程序读、写单独的数组元素和二进制文件中的完整数组。程序生成数组 arr1, arr2 和 arr3, 然后执行下述功能:

- ☐ 赋值于数组 arr1 的元素(数组 arr1 有 10 个元素)
- ☐ 赋值于数组 arr3 的元素(数组 arr3 有 20 个元素)
- ☐ 显示数组 arr1 中的值
- ☐ 把数组 arr1 的元素写入文件 ARR1.DAT, 每次写一个元素。
- ☐ 从文件 ARR1.DAT 中读出数组 arr1 的元素, 且存入数组 arr2(数组 arr2 也具有 10 个元素, 与数组 arr1 大小相同)
- ☐ 显示数组 arr2 的值
- ☐ 显示数组 arr3 的值
- ☐ 一次就把数组 arr3 中的所有元素写入文件 ARR3.DAT。
- ☐ 把 ARR3.DAT 文件中的数据, 一次性就存储在数组 arr1 中。
- ☐ 显示数组 arr1 的值(输出的 arr1 的大小和数值与数组 arr3 相同)。

程序列表 12.2 程序 IO2.CPP 的源代码

```

1: /*
2:    C++ program that demonstrates sequential binary file I/O
3: */
4:
5: #include <iostream.h>
6: #include <fstream.h>
7:
8: const unsigned MIN_SIZE = 10;
9: const double BAD_VALUE = -1.0e+30;
10: enum boolean { false, true };
11:
12: class Array
13: {
14:     protected:
15:         double *dataPtr;
16:         unsigned size;
17:         double badIndex;
18:
19:     public:
20:         Array(unsigned Size = MIN_SIZE);
21:         ~Array()
22:         { delete [] dataPtr; }
23:         unsigned getSize() const { return size; }
24:         double& operator [] (unsigned index)
25:         { return (index < size) ? *(dataPtr + index) : badIndex; }
26:         boolean writeElem(fstream& os, unsigned index);
27:         boolean readElem(fstream& is, unsigned index);
28:         boolean writeArray(const char* filename);
29:         boolean readArray(const char* filename);
30: };
31:
32: Array::Array(unsigned Size)
33: {
34:     size = (Size < MIN_SIZE) ? MIN_SIZE : Size;
35:     badIndex = BAD_VALUE;
36:     dataPtr = new double[size];
37: }
38:
39: boolean Array::writeElem(fstream& os, unsigned index)
40: {
41:     if (index < size) {
42:         os.write((unsigned char*)(dataPtr + index), sizeof(double));
43:         return (os.good()) ? true : false;
44:     }
45:     else
46:         return false;
47: }
48:
49: boolean Array::readElem(fstream& is, unsigned index)
50: {
51:     if (index < size) {
52:         is.read((unsigned char*)(dataPtr + index), sizeof(double));
53:         return (is.good()) ? true : false;
54:     }
55:     else
56:         return false;
57: }
58:
59: boolean Array::writeArray(const char* filename)
60: {
61:     fstream f(filename, ios::out | ios::binary);
62:
63:     if (f.fail())
64:         return false;

```

```

65:     f.write((unsigned char*) &size, sizeof(size));
66:     f.write((unsigned char*)dataPtr, size * sizeof(double));
67:     f.close();
68:     return (f.good()) ? true : false;
69: }
70:
71: boolean Array::readArray(const char* filename)
72: {
73:     fstream f(filename, ios::in | ios::binary);
74:     unsigned sz;
75:
76:     if (f.fail())
77:         return false;
78:     f.read((unsigned char*) &sz, sizeof(sz));
79:     // need to expand the array
80:     if (sz != size) {
81:         delete [] dataPtr;
82:         dataPtr = new double[sz];
83:         size = sz;
84:     }
85:     f.read((unsigned char*)dataPtr, size * sizeof(double));
86:     f.close();
87:     return (f.good()) ? true : false;
88: }
89:
90: main()
91: {
92:     const unsigned SIZE1 = 10;
93:     const unsigned SIZE2 = 20;
94:     char* filename1 = "array1.dat";
95:     char* filename2 = "array3.dat";
96:     Array arr1(SIZE1), arr2(SIZE1), arr3(SIZE2);
97:     fstream f(filename1, ios::out | ios::binary);
98:
99:     // assign values to array arr1
100:    for (unsigned i = 0; i < arr1.getSize(); i++)
101:        arr1[i] = 10 * i;
102:
103:    // assign values to array arr3
104:    for (i = 0; i < SIZE2; i++)
105:        arr3[i] = i;
106:
107:    cout << "Array arr1 has the following values:\n";
108:    for (i = 0; i < arr1.getSize(); i++)
109:        cout << arr1[i] << " ";
110:    cout << "\n\n";
111:
112:    // write elements of array arr1 to the stream
113:    for (i = 0; i < arr1.getSize(); i++)
114:        arr1.writeElem(f, i);
115:    f.close();
116:
117:    // reopen the stream for input
118:    f.open(filename1, ios::in | ios::binary);
119:
120:    for (i = 0; i < arr1.getSize(); i++)
121:        arr2.readElem(f, i);
122:    f.close();
123:
124:    // display the elements of array arr2
125:    cout << "Array arr2 has the following values:\n";
126:    for (i = 0; i < arr2.getSize(); i++)
127:        cout << arr2[i] << " ";
128:    cout << "\n\n";
129:
130:    // display the elements of array arr3

```



```

131:  cout << "Array arr3 has the following values:\n";
132:  for (i = 0; i < arr3.getSize(); i++)
133:      cout << arr3[i] << " ";
134:  cout << "\n\n";
135:
136:  // write the array arr3 to file ARRAY3.DAT
137:  arr3.writeArray(filename2);
138:  // read the array arr1 from file ARRAY3.DAT
139:  arr1.readArray(filename2);
140:
141:  // display the elements of array arr1
142:  cout << "Array arr1 now has the following values:\n";
143:  for (i = 0; i < arr1.getSize(); i++)
144:      cout << arr1[i] << " ";
145:  cout << "\n\n";
146:  return 0;
147: }

```

程序列表 12.2 中程序的执行结果如下:

输出:

Array arr1 has the following values:

0 10 20 30 40 50 60 70 80 90

Array arr2 has the following values:

0 10 20 30 40 50 60 70 80 90

Array arr3 has the following values:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Array arr1 now has the following values:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

分析:

程序列表 12.2 中程序定义了 Array 类,类似于第十一章的程序列表 11.2。主要不同之处在于:此处用操作符[]代替成员函数 store 和 recall。这个操作符用于检查有效的标号如果参数,超出了范围,则返回一值存入成员函数 writeElem 中,除了操作符[]外,我们还增加了成员函数 writeElem, readElem, writeArray 和 read Array 以执行二进制文件的顺序流 I/O。

函数 write Elem 被定义在程序第 39 至 37 行。把一个数组元素写入输出流。参数 os 代表输出流。参数 index 规定哪个数组元素将写入输出流。函数 writeElem 如果参数的标号有效时,并且输出流没有任何错误时,返回 ture 值。在 writeElem 函数写入一个数组元素后,内部流指针指向下一个地址。

函数 writeElem 和 readElem 允许相同的类从多个流中分别读、写数据。

程序第 59 至 69 行给出函数 writeArray 的定义。把整个数组元素写入二进制文件。参数 filename 规定用于输出的文件名。函数打开一个输出流,并把数据的数目和动态数组的元素写入。函数 writeArray 当它成功地把数组写入输出流中的,返回 ture 值。否则返回 false 值。该函数用流函数 open 文件名和 I/O 模式参数来打开一个局部输出流。I/O 模式参数表示成:ios::out | ios::binary,用于规定此流被打开,仅用于二进制文件输出。函数两次调用流函数 write,第一次是写入数据的数目;第二次是写入动态数组元素。

程序 71 至 88 行的函数 `readArray` 从二进制文件中读出整个数组元素。参数 `filename` 规定用于输入的文件名。该函数打开一个输入流。把数据的数目和动态数组的元素读入。函数 `readArray` 当成功地把数组读入流中时,返回 `true` 值,否则返回 `false` 值。函数用流函数 `open`,并提供它与文件名及 I/O 模式参数以打开一个局部的输入流。I/O 模式参数表示为:`ios::in` | `ios::binary`。说明此流仅用于二进制输入。函数两次调用流函数 `read`,第一次读入数据的数目;第二次读入动态数组元素。`readArray` 函数的另一个特点是它可重新规定属 `Array` 类的对象的大小以接收二进制文件中的数据。这种特点就可使动态数组根据存储在二进制文件中的数组大小而缩小或扩大。

程序列表 12.2 中的成员函数规定程序进行两种类型的二进制流顺序 I/O。第一种 I/O 方式为:利用函数 `readElem` 和 `writeElem`;第二种 I/O 方式为:调用函数 `readArray` 和 `writeArray`。

主函数 `main` 执行下述功能:

- ☐ 第 96 行定义三个 `Array` 类的对象,命名为 `arr1`,`arr2` 和 `arr3`(前两个对象的动态数组大小相同,由常数 `Size1` 规定,而对象 `arr3` 的大小由常数 `Size2` 规定)。
- ☐ 第 97 行定义了文件流 `f`,并按二进制输出模式将它打开用于访问文件 `ARR1.DAT`。
- ☐ 第 100 至 104 行使用 `for` 循环分别对对象 `arr1` 和 `arr3` 输入任意值。
- ☐ 第 108 行使用 `for` 循环来显示 `arr1` 数组的元素值。
- ☐ 第 113 行使用 `for` 循环,把数组 `arr1` 的元素写入输出文件流 `f`。控制变量为 `i`。
- ☐ 通过发送关闭输出文件流 `f` 的信息来关闭输出文件流。
- ☐ 在第 118 行,打开文件流 `f`,用于访问文件 `ARR1.DAT` 的数据(这次是按二进制输入模式打开文件流)。
- ☐ 用第 120 行的 `for` 循环传送信息 `readElem` 给对象 `arr2`。从输入文件流 `f` 中,读出对象 `arr2` 的元素,循环控制变量为 `i`。
- ☐ 第 122 行关闭输入文件流。
- ☐ 用第 126 行的 `for` 循环显示对象 `arr2` 的元素(其中的元素与对象 `arr1` 相同)。
- ☐ 用第 132 行的 `for` 循环显示对象 `arr3` 中的元素。
- ☐ 通过把信息 `writeArray` 传给对象 `arr3`,把整个对象 `arr3` 的元素写入文件 `ARR3.DAT`。
- ☐ 把文件 `ARR3.DAT` 中的内容读入对象 `arr1` 中。
- ☐ 用第 143 行的 `for` 循环显示对象 `arr1` 中的新内容。

12.5 随机存取文件流 I/O

随机存取文件流操作也使用上节所介绍的流函数 `write` 和 `read`。流库提供一些流查询函数,使你可把流指针移到某个有效地址。函数 `seekg` 就是其中的一个流查询函数。

语法

`seekg` 函数

重载函数 seekg 的函数原型为:

```
istream&. seekg (long pos);
```

```
istream&. seekg (long pos, seek_dir dir);
```

参数 pos 在第一个版本中规定了流中某个字节的绝对位置。在第二个版本中,它规定相对于参数 dir 的偏移地址。参数 dir 可取如下值:

ios::beg 从文件开始

ios::cur 从文件当前位置

ios::end 从文件末尾

例子:

```
const BLOCK_Size = 80;
char buff[BLOCK_Size] = "Hello world!";
fstream f ("CALC.DAT", ios::in | ios::out | ios::binary);
f.seekg (3 * BLOCK_Size); // seek block #4
f.read ((const unsigned char *)buff, BLOCK_Size);
cout << buff << "\n";
f.close();
```

新术语:虚拟数组是一个基于磁盘数组,用于存储固定大小的字符串。

让我们看看下面使用随机存取文件流 I/O 的程序示例。程序列表 12.3 为 IO3.CPP 的源程序,它处理一个虚拟数组。该程序执行下述功能:

- ☐ 用内部地名表生成一个虚拟数组对象。
- ☐ 显示无序虚拟数组对象中的元素。
- ☐ 请你输入一个字符并按 Return 键。
- ☐ 对虚拟数组对象中的元素排序,这个过程需要随机存取 I/O。
- ☐ 显示排序后的虚拟数组对象中的元素。

程序列表 12.3 程序 IO3.CPP 的源代码

```
1: /*
2:    C++ program that demonstrates random-access binary file I/O
3: */
4:
5: #include <iostream.h>
6: #include <fstream.h>
7: #include <stdlib.h>
8: #include <string.h>
9:
10: const unsigned MIN_SIZE = 5;
11: const unsigned STR_SIZE = 31;
12: const double BAD_VALUE = -1.0e+30;
13: enum boolean { false, true };
14:
15: class VmArray
16: {
17:     protected:
18:         fstream f;
19:         unsigned size;
20:         double badIndex;
```