



中国科学院希望高级电脑技术公司高级程序设计丛书之一

面向对象的程序设计

Turbo C++

入 门

叶欣 张国锋

编译

王强 张莉

海 洋 出 版 社

内 容 提 要

Turbo C++1.0是在Turbo C的基础上推出的最新面向对象的程序设计软件包。该丛书共有五册,本书是其中的一册。本书共分七章,包括安装 Turbo C++, Turbo C++手册导引,学习新的IDE, C简介, C++要素,掌握 C++和新的IDE调试等内容,系统地介绍了 Turbo C++的基础知识和技术,是一本帮助和引导读者进行面向对象程序设计的教科书。

欲购本书及其软件的用户可直接与北京8721信箱联系,电话2562329,邮政编码100080。

Turbo C++ 入 门

叶 欣 张国锋
王 强 张 莉 编译
阎世尊 审校

海洋出版社出版(北京市复兴门外大街1号)
新华书店北京发行所发行 双青印刷厂印刷
开本: 787×1092 1/16 印张: 12.06 字数: 294千字
1991年3月第一版 1991年3月第一次印刷
印数 1—3000册
ISBN7—5027—1392—1/TP.21
定价 7.50元

前 言

随着计算机软件设计方法的不断提高和改进,面向对象的程序设计在我国也逐渐流行起来并大有迅猛发展的趋势,它提出了一种全新的程序设计思想,把数据和对数据进行的操作熔为一体。

美 BORLAND 公司在 Turbo C 的基础上,推出最新面向对象的程序设计软件包——Turbo C++ 1.0 版。它继承并发挥了原来 Turbo C 集成环境的优良特性,并包含了面向对象的基本思想和设计方法,是目前国际上最受欢迎的面向对象程序设计软件包。

我们在长期从事 Turbo C 程序设计的基础上,加上近几年对面向对象的研究,根据 BORLAND 公司的 Turbo C++ 软件和资料,特编译了 Turbo C++ 程序设计丛书。

本丛书共有五册,分别为《入门》、《用户手册》、《程序员指南》、《库函数参考》和《程序设计方法》,全面系统地介绍了 Turbo C++ 的基础知识和高级技术,是一套引导读者进行面向对象程序设计的教科书。

由于面向对象的概念在国内还处于初步阶段,加上我们水平有限,书中难免会有错误和缺点,敬请广大读者批评指正,以备再版时修订。

本丛书在编译过程中,得到了许多同志的帮助和支持,其中薛梅同志、吕良双同志、张阳同志为本书的成稿做了大量的工作,在此表示谢意。

本丛书出版过程中,得到海洋出版社的编辑同志以及中国科学院希望高级电脑技术公司资料部秦人华经理、杨淑新老师的大力帮助和支持,在此表示衷心的感谢。

编译者

1991 年 1 月于北京

目 录

简介	1
0.1 Turbo C++ 内容	1
0.2 硬件和软件环境	1
0.3 Turbo C++ 的实现	2
0.4 Turbo C++ 软件包	2
0.4.1 《入门》	2
0.4.2 《用户手册》	3
0.4.3 《程序员指南》	3
0.4.4 《库函数参考》	4
第一章 安装 Turbo C++	5
1.1 安装 Turbo C++	5
1.1.1 Lapton 系统	6
1.2 README 文件	6
1.3 HELPME!.DOC 文件	6
1.4 Turbo Calc	6
第二章 Turbo C++ 手册导引	7
2.1 特征	7
2.1.1 C++	7
2.1.2 VROOMM(覆盖)	7
2.1.3 Borland 新的集成环境	7
2.2 手册安排	7
2.2.1 新程序员或正在学习 C 的程序员	8
2.2.2 熟练的 C 语言程序员	8
第三章 学习新的 IDE	9
3.1 IDE	9
3.1.1 全菜单开/关(on/off)	10
3.1.2 鼠标器、热键和联机提示	10
3.1.2.1 鼠标	10
3.1.2.2 热键	11
3.1.2.3 联机求助	11
3.2 第一课: 启动、加载和编辑	12
3.2.1 产生一个新文件	13
3.2.1.1 选择块	13
3.2.1.2 拷贝和粘贴	13
3.2.2 修改新文件	14
3.2.2.1 查找与替换	14

3.2.2.2	在求助窗口粘贴	15
3.2.2.3	存储修改过的文件	16
3.3	第二课: 编译和运行	16
3.3.1	关闭编辑窗口	16
3.4	第三章: 退出 Turbo C++	16
3.5	进一步的资料	17
第四章	C 简介	18
4.1	回顾	18
4.2	基本编程操作	18
4.3	C 程序的基本结构	20
4.4	数值运算	22
4.4.1	数值数据类型	22
4.4.1.1	整数	23
4.4.1.2	长变形	24
4.4.1.3	有符号和无符号变量	24
4.4.2	浮点数	24
§ 4.4.2.1	双精度和长双精度	25
4.5	变量	26
4.5.1	初始化变量	26
4.5.2	赋值语句	26
4.5.3	复合赋值	27
4.5.4	命名	27
4.5.5	成组输入: sscanf	28
4.5.6	显示变量值	30
4.5.6.1	printf 中的类型转换	32
4.5.6.2	使用转义符(\)序列的格式	32
4.5.7	算术运算符	34
4.5.8	算术和类型转换	35
4.5.9	算术与赋值联合运算	36
4.5.10	增量和减量	36
4.5.11	位运算	37
4.6	表达式	38
4.6.1	表达式求值	38
4.6.2	表达式中赋值	39
4.7	字符和字符串	40
4.7.1	单个字符的输入与输出	40
4.7.2	显示一个字符	41
4.7.3	显示字符串	41
4.8	测试条件和作出选择	42
4.8.1	使用关系运算符	42

4.8.2	使用逻辑运算符	43
4.8.3	使用 if 和 if...else 的分支语句	44
4.8.4	使用 if...else 的多重选择	45
4.8.5	多重选择: switch	46
4.9	循环性重复运行	48
4.9.1	while 循环	48
4.9.2	do while 循环	49
4.9.3	for 循环	50
4.9.4	break 和 Continue	52
4.9.5	goto 语句	53
4.9.6	循环嵌套	53
4.9.7	选择合适的循环语句	54
4.10	使用函数和宏的程序设计	54
4.10.1	定义自己的函数	54
4.10.1.1	函数原型	54
4.10.1.2	函数定义	55
4.10.1.3	函数内的处理过程	55
4.10.1.4	函数返回值	56
4.10.1.5	应用返回值	56
4.10.2	多函数程序	56
4.10.2.1	函数原型和全局说明	59
4.10.2.2	设置图形显示	59
4.10.2.3	计算图形坐标	60
4.10.2.4	画出行星	60
4.10.3	头文件、函数和库	61
4.10.4	变量的范围和持续时间	62
4.10.4.1	范围(scope)	62
4.10.4.2	持续时间	63
4.10.5	使用常量值	64
4.10.6	使用宏来隐含细节	65
4.11	建立数据结构	66
4.12	说明和初始化数组	67
4.12.1	多维数组	68
4.12.2	数组和串	70
4.12.3	定义串变量	70
4.12.4	重新命名类型	71
4.12.5	枚举类型	72
4.12.6	把数据组合成结构	73
4.12.7	使用结构的数据项	73
4.13	建立合适的说明符	74

4.14	指针	75
4.14.1	说明和使用指针	76
4.14.2	指针和串	77
4.14.3	指针运算	78
4.14.4	指针、结构和列表	78
4.14.5	使用指针来从函数返回值	81
4.15	利用系统资源	82
4.15.1	使用文件和流	83
4.15.1.1	打开流	84
4.15.1.2	写文件	84
4.15.1.3	读文件	84
第五章	C++要素	85
5.1	封装(Encapsulation)	86
5.2	继承(inheritance)	88
5.3	多态性(Polymorphism)	89
5.4	重载(Overloading)	89
5.5	用类来模拟现实世界	89
5.5.1	建立类: 一个图形例子	90
5.5.2	说明对象	91
5.5.3	成员函数	91
5.5.4	调用一个成员函数	92
5.5.5	构造函数和析构函数	92
5.5.6	代码和数据相结合	94
5.5.7	成员访问控制: 私有的(private)、公有的(public)和保护的保护的(protected)	94
5.5.8	类: 缺省为私有的	95
5.5.9	运行一个C++程序	96
5.6	继承	98
5.6.1	重新思考 Point 类	98
5.6.2	把类装入模块	100
5.6.3	扩充类	103
5.6.4	多重继承	105
5.6.5	虚函数	110
5.6.6	虚函数的作用	111
5.6.7	定义虚函数	112
5.6.8	开发一个完整的图形模块	112
5.6.9	普通的还是虚拟的成员函数?	119
5.7	动态对象	120
5.7.1	析构函数和 delete	121
5.7.2	分配动态对象的一个例子	121
5.8	C++中更多的灵活性	126

5.8.1	在类定义之外的内部函数	126
5.8.2	带有缺省参数的函数	127
5.8.3	有关重载函数的更多说明	127
5.8.4	重载运算符以提供新的含义	129
5.8.5	友元函数(Friend function)	132
5.8.6	C++ 流库	133
5.9	用户定义数据类型的 I/O	138
5.10	现在干什么?	139
5.11	小结	139
第六章	掌握 C++	140
6.1	更好的 C: 从 C 过渡	140
6.1.1	程序 1	140
6.1.2	程序 2	141
6.1.3	程序 3	141
6.1.4	程序 4	142
6.2	对象支持	143
6.2.1	程序 5	143
6.2.2	程序 6	145
6.2.3	程序 7	149
6.2.4	程序 8	151
6.2.5	程序 9	153
6.3	小结	156
第七章	新的 IDE 调试	157
7.1	调试与程序开发	158
7.2	设计示例程序: PLOTEMP.C	159
7.3	编写程序原型	160
7.4	使用集成调试程序	162
7.5	跟踪程序的流程	163
7.6	继续程序的开发	164
7.7	设置断点	166
7.8	检查数据	168
7.9	计算和修改变量	169
7.10	通过设置 watches 监视程序	174
7.11	预防药	177
7.12	有系统的软件测试	177
7.13	完成 PLOTEMP.C	178
7.14	调试练习的答案	182

简介

Turbo C++适用于需要一个有快速和有效编译器的 C++和 C 程序员;适用于想使用所有“Turbo”优点而学习 C++或 C 语言的 Turbo Pascal 程序员;适用于正在学习 C++或 C 语言的任何人。Turbo C++也适用于 AT&T 公司的 C++ 2.0 版和 ANSI C 版的程序员。

Turbo C++与目前的 Turbo C 高度兼容。

C++是一个面向对象(OOP)的程序设计语言,它是 C 语言的进一步发展。C++语言是可移植的,所以你可以很容易地将一个由 C++编写的应用程序从一个系统移植到另一个系统上。在任何地方,你都可以用 C++从事任何程序工作。

0.1 Turbo C++内容

Turbo C++具有许多用户要求的最新特征:

- C++: Turbo C++提供 C++编程的全部功能(AT&T 实现 C++ 2.0 版)。为了帮助你开始工作,我们提供了一个 C++的类库。为了帮助移植 C++ 1.2 版本的程序,我们提供对 C++ 1.2 版流的支持。
 - ANSI C: Turbo C++提供给你至今为止最新的 ANSI C 标准的实现。
 - Borland 新的程序员平台(platform): 程序员平台是新一代的用户接口,它胜过了旧的集成环境,提供对计算机上全范围内程序和工具的访问。包括:
 - ▲鼠标器支持
 - ▲多重覆盖窗口
 - ▲多重文件编辑
 - ▲支持内部汇编码
 - 等等。
 - VROOMM(面向目标的实时虚拟存储管理): VROOMM 使你能很简单地覆盖代码。你只要为覆盖选择代码段, VROOMM 会处理其余的事情。
 - 新程序员平台的联机访问。
 - 联机的快速文本求助,用拷贝的模拟程序例子试验每一个函数。
 - 许多独立的库函数,包括堆检测函数和完整的复数与 BCD 数学函数集。
- 其它特征包括:
- 对-S 选择的扩展: C 源码作为注释可加到结果汇编代码上。
 - 远对象和大容量数组
 - 几个新的杂注(pragma)和警告: 一个不规范的杂注警告,一个 Argused 杂注和一个启动杂注。
 - 候补.CFG 文件: 你能创建几个这样的文件并且在需要的时刻使用合适的一个。
 - 用于命令行编译的响应文件。

0.2 硬件和软件环境

Turbo C++运行于 IBM PC 系列的计算机上,其中包括 XT、AT 和 PS/2,以及与 IBM

全兼容的计算机。Turbo C++ 要求 DOS 2.0 以上的版本和至少 640K 内存。监视器为 80 列即可。最低的要求是一个硬盘和一个软盘驱动器。

Turbo C++ 包含有使你的程序能利用 80×87 协处理器的浮点运算例程。如果没有协处理器，它可进行仿真运算。虽然运行 Turbo C++ 不必有 80×87 协处理器，但 80×87 协处理器可有效地增进你的程序运行。

Turbo C++ 也支持鼠标器。虽然鼠标器不是必须的，如果你有一个，它必须与下列要求之一全兼容。

- Microsoft Mouse 6.1 版本或更高版本，或者任何与此兼容的 mouse 版本。
- Logitech Mouse 3.4 版或更高版本。
- Mouse Systems'PC Mouse 6.22 版或更高版本。
- IMSI Mouse 6.11 版或更高版本。

0.3 Turbo C++的实现

Turbo C++ 全部实现了 AT&T C++ 的 2.0 版。同时也实现了美国国家标准局(ANSI)的 C 标准，并且支持 Kernighan 和 Ritchie 的定义。另外，Turbo C++ 包含有一定的扩展，使你能用混合语言和混合模型编程来扩充 PC 机的能力。在《程序员指南》的第一章“Turbo C++ 语言标准”中有详细的描述。

0.4 Turbo C++软件包

Turbo C++ 软件包由磁盘文件集和四本手册组成：

- Turbo C++ 入门(本手册)
- Turbo C++ 用户手册
- Turbo C++ 程序员指南
- Turbo C++ 库函数参考

除这四本手册外，你会方便地找到速成参考读物。磁盘文件中包含着所有的程序、文件和在建立、编译、连接和运行 Turbo C++ 程序时所需要的库，也包含有几个样本程序，几个卓有成效的应用程序，紧缩的提示文件，集成调试器以及在那些指南中未包括的附加 C 文档。

0.4.1 《入门》

本书介绍 Turbo C++，并且显示怎样生成和运行 C 和 C++ 程序。它由你要组织和快速运行的信息组成：安装、辅导、入门和 Turbo C++ 文件集指南。本手册的章次如下：

第一章“安装 Turbo C++”：告诉你怎样在系统上安装 Turbo C++。

第二章“Turbo C++ 手册导引”：介绍一些 Turbo C++ 最有趣的特征，在合适的地方，你会找到它们。你也可以尝试联机辅助 TCTOUR。

第三章“学习新的 IDE”：让你领略集成环境并且介绍新的编辑器、鼠标支持和其它新的或已变化的特征，这是 Turbo C++ 系统的简貌，更进一步的内容可在《用户手册》的第一章“IDE 指南”中找到。

第四章“C 简介”：C 语言的全貌。这章向你介绍 C 语言的组成、数据和数据类型、运算符、函数、数组、结构和 C 语言的其他方面。

第五章和第六章是个整体：前一章是理论，后一章是应用。

第五章“C++要素”：介绍利用 C++ 编程的面向对象的概念。

第六章“掌握 C++”：简洁地介绍 C++。

第七章“新的 IDE 调试”：介绍 Turbo C++ 的集成调试，并且让你利用已有错误的程序例子去演示调试器的不同特征。

0.4.2 《用户手册》

《用户手册》提供了与 Turbo C++ 为特征的参考章节：Borland 的新的集成环境，其包含有增强的编辑器和工程管理，也有关于利用 Turbo C++ 的实用程序细节、命令行编译程序和特殊程序。

第一章“IDE 指南”：提供了有关集成开发环境的全部参考。

第二章“多文件工程管理”：告诉你怎样利用工程管理程序管理多重文件。

第三章“编辑命令”：提供了全部的编辑器参考内容。

第四章“命令行编译程序”：告诉你怎样利用命令行，同时也解释了配置文件。

第五章“实用工具”：描述了 Turbo C++ 的实用程序。

第六章“设置 Turbo C++ 环境参数”：告诉你利用 TCINST 程序去生成特定的 Turbo C++。你可以设置屏幕颜色、缺省编辑器、缺省汇编和连接以及使用这个程序时所有的 Turbo C++ 特征。

附录 A “Turbo 编辑宏语言”：描述了 Turbo 宏编辑语言，它是个你可以利用扩展或改进的 Turbo C++ 编辑器实现的有效的实用语言。

0.4.3 《程序员指南》

对有经验的 C 语言用户，《程序员指南》提供了有用的材料：一个完整的 C 和 C++ 语言的参考、运行时库的引用、C++ 流、存储模型、混合模型编程、视频函数、浮点处理和覆盖以及错误信息。

第一章“Turbo C++ 语言标准”：描述了 Turbo C++ 语言。对 ANSI C 语言的扩展也在这里描述。这章是 Turbo C++ 下的 C 和 C++ 语言的基本参考和句法。

第二章“运行时间库交叉参考”：提供了有关运行时库的源代码信息，列出并且描述了头文件，通过主题提供了运行时库的参考。例如，如果你想要找与图形有关的函数，就可在主题“图形(Graphics)”下找到。

第三章“C++ 流”：告诉你怎样去使用 C++ 2.0 版的流库。早期版本的流库通过联机方式提供。

第四章“存储模型、浮点数和覆盖”：它包括存储模型、混合模型编程、浮点处理和覆盖。

第五章“视频函数”：它只对 Turbo C++ 中的文本处理和图形函数起作用。

第六章“和汇编语言接口”：告诉你怎样编写汇编语言程序以便 Turbo C++ 程序能够调用。

第七章“错误信息”：列出所有运行和编译时产生的错误和警告，并且提出可能的解决意见。

附录 A “ANSI 特殊实现标准”：描述了 ANSI 未严格定义或未定义的 ANSI C 标准特

性。这些特性随着每一个标准的实现将会改变。本附录告之 Turbo C++ 对于每一个这种特性的操作。

0.4.4 《库函数参考》

“库函数参考”包含有 Turbo C++ 库函数和全局变量的详细罗列和解释。

第一章“运行库”：依 Turbo C++ 库函数的字母顺序排列。每个条目都给出句法、包含的文件、运算描述、返回值和对函数的可移植信息，并且列出相关函数的引用表。

第二章“全局变量”：定义和讨论 Turbo C++ 的全局变量。对于公共需要的变量你可以使用全局变量来节约大量的编程时间(比如象数据、时间、错误信息、栈的大小等变量)。

第一章 安装 Turbo C++

如果你不会使用 DOS 命令, 在系统上安装 Turbo C++ 之前请先阅读 DOS 参考手册。

Turbo C++ 软件包包含有两种 Turbo C++ 编译器的不同版本: 集成环境版本和命令行版本。必须使用 INSTALL 程序去安装 Turbo C++ 到你的系统, 它会自动把配置盘上的文件拷贝到你的硬盘上。没有拷贝保护。为参考起见, README 文件放在含有一系列配置文件的安装盘上。

我们假设你已熟知 DOS 命令。例如, 你知道用 DISKCOPY 命令去备份配置盘, 当你得到配置盘时, 要做一次完全的备份, 然后把源盘放在安全处。

本章包含下列内容:

- 在系统上安装 Turbo C++
- 获得 README 文件
- 获得 HELPMENI 文件
- Turbo Calc 中更多的指针信息

一旦安装了 Turbo C++, 就可以开始钻研它。但是某些章节和手册的内容是为特定编程的需要而编写的。第二章 “Turbo C++ 手册导引” 告诉你在什么地方可找到有关本手册中 Turbo C++ 的特性。

1.1 安装 Turbo C++

Turbo C++ 有一个称之为 INSTALL 的自动安装程序, 你必须使用 INSTALL 来安装。这个程序检查所使用的硬件并且适当地配置 Turbo C++, 随需要它也能创建目录并且从配置盘(你购买的盘)上把文件拷贝到硬盘上。它的操作是自动进行的, 下列内容告诉你所需要知道的信息:

为了安装 Turbo C++:

1. 插入安装盘到驱动器 A, 敲如下命令, 然后按 Enter。
2. A:INSTALL。
3. 在安装屏幕上, 按回车键。
4. 顺序提示显示。

重要

当结束时, 在文件 README 中, INSTALL 程序使你读到有关 Turbo C++ 的最新信息, README 文件包含有重要的、最新的有关 Turbo C++ 的信息。文件 HELPMENI.DOC 也回答一些类似的技术服务问题。

同时, 一旦安装了 Turbo C++, 你就可以试用 TCTOUR。TCTOUR 是新的 Turbo C++ 集成环境中带有一些亮光标的指南导游。TCTOUR 是 TOUR 下的子目录(下属于你的 Turbo C++ 目录)。

为了退出 Turbo C++, 按 Alt-X。

一旦安装了 Turbo C++ 并且试用 TCTOUR, 如果你急于开始和运行, 请改变 Turbo C++ 的目录, 并键入 TC, 按 ENTER, 否则, 继续阅读此章后面的内容, 以便得到重要

的启动信息。

当试用了 Turbo C++ 的集成环境后, 你可能想要永久性地保留一个选择项。用 TCINST 可以容易地做到这一点。见《用户手册》第六章“设置 Turbo C++ 环境参数”的指令。

1.1.1 Laption 系统

如果你有一套 Laption 计算机(具有 LCD 或等离子显示), 并且能实现前几节所给出的处理操作, 那么在使用 Turbo C++ 之前你要设置屏幕参数。在运行 Turbo C++ 之前, 要使 Turbo C++ 的集成开发环境(TC.EXE)正常工作, 必须在 DOS 命令行键入:

MODE BW80

虽然可以产生一个批文件来做这些工作, 你也能很容易地用 Turbo C++ 定做程序 ICINST 在黑白显示屏上安装 Turbo C++。见《用户手册》第六章的指令。利用定做程序, 从屏幕模式菜单选择"Black and white"即可。

1.2 README 文件

README 文件包含有手册中没有的最新信息, 它同时罗列出配置盘中的每个文件, 并且带有简短的描述。

获得 README 文件:

1. 如果没有安装 Turbo C++, 把 Turbo C++ 盘插到 A 驱动器。如果安装了 Turbo C++, 转到第 3 步, 否则继续。
2. 键 A: 并且按 ENTER。
3. 键 README 及回车, 一旦进入 README, 利用键↑和↓上下翻看屏幕。
4. 按 Esc 退出。

看本书的第三章“学习新的 IDE”, 为了得到更多的使用新的集成环境的细节, 看《用户手册》第一章“IDE 指南”

如果已经安装了 Turbo C++, 可以打开 README 文件并把它调入编辑窗口, 操作步骤如下:

1. 在命令行键入 TC 启动 Turbo C++, 按 ENTER。
2. 按 F10, 选择 File|Open, 键入 README 并且按 ENTER, Turbo C++ 在编辑窗口中打开 README。
3. 当读完了 README 文件后, 选择 File|Quit(或者用新的环境继续操作)。

1.3 HELPMEI.DOC 文件

配置盘中包含有 HELPMEI.DOC 文件, 它是对用户普遍碰到的问题的解答。如果你遇到困难可使用它。你也可以利用 README 程序去观看 HELPMEI.DOC, 只要在命令行键入: README HELPMEI.DOC

1.4 Turbo Calc

你的 Turbo C++ 软件包包含有用于展开的称为 Turbo Calc 的程序源代码。在编译它之前, 请阅读联机文件 TCALC.DOC。

第二章 Turbo C++ 手册导引

本章完成两件事:

- 简明地告诉你有关 Turbo C++ 的最新特性: 它们的概念、含义和怎样使用它们。
- 告诉你找到更多的有关 Turbo C++ 的新特性和其它特征。

如果阅读了上一章安装 Turbo C++ 的指令, 你就知道了怎样启动 Turbo C++ 和怎样退出。如果没有, 请参阅上一章。

2.1 特征

Turbo C++ 有许多强有力的特性, 见本书简介。本节仅告诉你一小部分这些特性, 并且指引你阅读更进一步的信息。

2.1.1 C++

使用 Turbo C++, 可得到两个编译器, 它具有所有 ANSI C 的功能, 以及所有 C++ 的功能。第五章“C++ 要素”和第六章“掌握 C++”提供了 C++ 的编程理论和辅导。

除这些之外, 我们为你提供了已制作的 C++ 类库集。这些库使用类去执行一系列功能。C++ 的一些优点, 象可扩展性和可重用性, 可以立即满足你的要求。

2.1.2 VROOMM(覆盖)

《程序员指南》第四章“存储模型、浮点数和覆盖”含有覆盖的进一步内容。

Turbo C++ 的 VROOMM(面向对象的实时虚拟存储管理)提供明智的覆盖, 它不象你可能用过的覆盖技术。如果你已经熟悉其它软件的覆盖(非 Borland 产品), 可能会有些惊奇。首先, VROOMM 能够确定怎样和什么时候去覆盖, 这样就能够减轻你的工作。其次, VROOMM 是建立在高度综合的算术集上, 它比其它的覆盖方案更快和更有效。

2.1.3 Borland 新的集成环境

因为 Borland 的集成环境是完全重复工作的, 我们就要求你通过提供的 TCTOUR 程序指导你巡游, 即使你已经熟悉 Borland 的其它软件。首先把目录转到 TOUR 子目录下(在你的 TC 目录中)。然后键 TCTOUR, 按 ENTER。

进一步, 阅读第三章“学习新的 IDE”。这章给出了在新的集成环境中具有更广特性的继承性经验, 它包括鼠标支持、求助提示系统、剪取板、处理窗口的新方式、编辑多重文件、转入其它程序(然后返回 Turbo C++)等等。《用户手册》的第一章“IDE 指南”介绍了新的集成环境的每一个特性。

2.2 手册安排

Turbo C++ 手册是这样安排的, 在你想找到某一内容的时候, 能够找到和选择对应的书和章节。《入门》和《用户手册》提供了使用 Turbo C++ 的信息, 《程序员指南》和《库函数参考》提供了 C 和 C++ 中的编程技术。

本书的五章，对有趣的问题提供辅助介绍：

■第三章，“学习新的 IDE”

■第四章，“C 简介”

■第五章，“C++ 要素”

■第六章，“掌握 C++”

■第七章，“新的 IDE 调试”

正如以前介绍的，集成环境是一个新的内容，所以即使你已熟悉 Turbo C++ 和 Turbo Pascal，也可能想再浏览一遍。

2.2.1 新程序员或正在学习 C 的程序员

如果你正在学习 C 语言，可从本书的第三章“学习新的 IDE”和第四章“C 简介”开始。这些章介绍了 Turbo C++ 的集成环境并且显示了 C 语言中的编程全貌。第五章“C++ 要素”和第六章“掌握 C++”给出了 C++ 的简洁介绍。根据你对 C++ 的掌握程序，如果你对 C 编程充满信心，读完了第四章后可能会跳过这两章。第七章“新的 IDE 调试”使你通过 Turbo C++ 的集成环境来运行程序。之后，作为参考，应阅读《用户手册》的第一章到第六章。

下一步，你要运用 C 语言编程。在许多地方，要用到《库函数参考》。或者，你可能偏好使用联机提示，它包含着和《库函数参考》一样的信息，并且对你拷贝到程序中的每一个函数，都有程序演示。

2.2.2 熟练的 C 语言程序员

如果你是一个熟练的 C 语言程序员，并且已经安装了 Turbo C++，可能会马上跳过《程序员指南》而去看《库函数参考》，然后，如果你想学到更多的有关新的 Turbo C++ 集成环境，包括集成调试器，请阅读本书的第三章和第七章（“学习新的 IDE”和“新的 IDE 调试”）。

如果你对 C++ 感兴趣，阅读第五和第六章，即“C++ 要素”和“掌握 C++”。

《程序员指南》包含有一些有用的编程内容，象 C++ 流、汇编语言接口、存储模型、视频函数、覆盖和远指针。另外，《程序员指南》的第二章“运行时间库交叉参考”提供了与库函数相关的内容。如果你想知道 C 语言中哪个函数与图形相关，你可翻阅该章并且查看有关“图形”的章节。

第三章 学习新的 IDE

在本章中, 你会逐步熟悉 Borland 新的 Turbo C++ 集成开发环境(IDE)。即使你已经熟悉 Borland 的其它软件, 我们也要求你阅读这一章。Turbo C++ 引入了新的 IDE, 本书给出了许多新特性的全貌。课程由下列组成:

- 第一课 显示怎样启动 Turbo C++, 和怎样装载、编辑、存储文件。
- 第二课 在 IDE 中怎样编译和运行程序。
- 第三课 从 Turbo C++ 中退出。

如果你现在急于在 Turbo C++ 上启动和运行, 请看第一章。否则, 通读本章教材以便掌握使用 IDE 的基础知识。只需 30 分钟就可完成学习。

如果想在菜单或交互对话中得到有关某个特定题目的进一步的信息, 可参阅《用户手册》的第一章“IDE 指南”。如果想通过联机方法试用 IDE, 可运行 TCTOUREXE(在 T 目录的子目录 TOUR 下, 在 DOS 下键入 TCTOUR 即可)。

第七章“新的 IDE 调试”提供了有关调试的长篇教程。

3.1 IDE :

Turbo C++ 包含有窗口、菜单(屏幕下端和上端)、对话框和状态行。这里是一个典型的 IDE 屏幕组成图示:

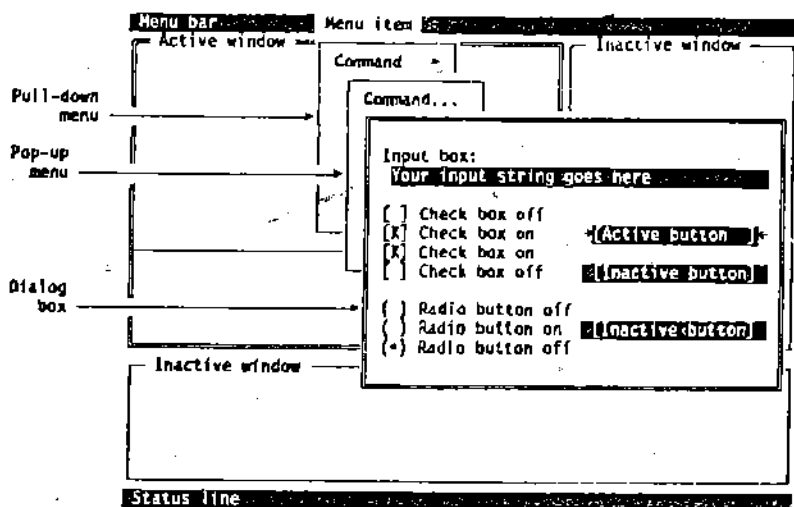


图 3.1 IDE 的组成

为了进入 IDE 菜单系统, 按 F10, 然后在菜单项中按高亮度的字母(你也可按 Alt 和高亮度的字母, 例如, Alt-F 为打开 File 菜单, 它的操作和 F10F 一样)。

为了退出菜单, 按 Esc 直到所有的菜单关闭, 使你返回到只有一个窗口。

Turbo C++ 利用对话框来显示选择。一个对话框包含有一个或多个下表列出的项目。

表 3.1 对话框的内容

条目 (Item)	功能
Action Button	此条是“阴影”显示。如果你选择这条(按ENTER), Turbo C++就立即执行相关的操作。
Check box	此条目是一个触发器开/关按钮。按空格或其它键它就开或关, 当选择为开时(on), x出现在括号中: [x]。
Radio button	此条是两个或两个以上条目组成的触发器, 在一个时刻你只能选择一个Radio按钮。利用键盘上的光标指示键, 可以在这些按钮集中移动, 一旦作出了选择, 按Tab键退出该集去进行新的按钮选择。利用鼠标器, 也能在Radio键中注意选择, 当Radio按钮选定后, 括号中就会出现一圆点: (.)。
Input box	输入框使你快速地键入字母串(例如文件名)。
list box	显示框包含着你所进行的选择的条目(例如一系列打开的文件名)。

当你在对话框的图象下选择了菜单项, 对话框就出现, 象图 3.3 的 File|Open 菜单项。

3.1.1 全菜单开/关(on/off)

Turbo C++有两个菜单系统选择: 全菜单开和全菜单关。全菜单 off(关), 通过提供你需要建立的调试 Turbo C++程序, 给出一个压缩的、易于使用的环境。全菜单 on (开)提供使你能进行更高级的程序任务的外部函数。在本教材中, 我们使用全菜单 off 集(缺省设置)。

图 3.2 对比了两种菜单状态, 作为例子, 它显示了全菜单 off 和 On 下的编译菜单。

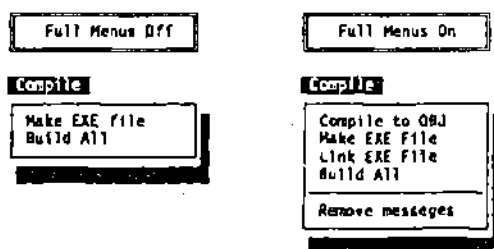


图 3.2 全菜单 on/off

3.1.2 鼠标器、热键和联机提示

在初次涉足 IDE 之前, 你可能想知道基本的 Turbo C++ 鼠标器使用、热键和联机提示。

3.1.2.1 鼠标

Turbo C++ IDE 支持鼠标器, IDE 使用鼠标器的左按钮。Turbo C++也可使你重新定义你的其它鼠标键: 详见《用户手册》第一章“IDE 指南”。鼠标的操作描述如下:

表 3.2 鼠标操作

按一下(Click)	按鼠标键, 然后松开
按两下(Double Click)	快速连续地按两次鼠标键
拖动(Drag)	按着鼠标键移动, 然后松开
按下一拖动(Click-Drag)	按鼠标键, 松开, 然后再按键, 移动鼠标, 松开。
选择(Choose)	按鼠标键选择菜单项, 例如一个命令或一个对话框

3.1.2.2 热键

许多菜单和对话框都有相应的热键: 一个或两个键的操作能够立即激活命令和对话框(在对话框中, 除非你在输入, 其它操作只要按高亮度的字母就可以从一个命令转到另一个命令)。下表列出了最常用的 Turbo C++ 热键, 更全的内容, 请见《用户手册》第一章“IDE 指南”。

当你按其中一个键时, 除了在对话框外, Turbo C++ 就在那个地方执行该键的功能。

表 3.3 Turbo C++ 热键

键	菜单项	功能
F1	求助提示	打开联机提示屏幕
F2	File Save	存储在编辑窗口中的活动文件
F3	File Open	进入对话框, 使你能打开文件
F4	Run Go to Cursor	执行程序到光标指定行
F5	Window Zoom	快速放大和缩小活动窗口
F6	Window Next	循环通过所有打开的窗口
F7	Run Trace Into	在调试状态下执行程序, 并且跟踪进入函数调用
F8	Run Step Over	在调试状态下执行程序, 并且跳过函数调用
F9	Compile Make Exe	编译当前的源文件为可执行的.EXE 文件
F10	(none)	返回主菜单

IDE 屏幕底端的状态行列出了在活动窗口中可以使用的热键。如果有鼠标器, 你可在状态行用鼠标器启动热键, 而不用进行键盘操作。

3.1.2.3 联机求助

象 Borland 其它软件一样, Turbo C++ 提供了内容紧凑的联机求助, 你可用热键(或鼠标器)启动。通过 Turbo C++ 的求助保留键或库函数, 你可得到 Turbo C++ IDE 中的任何项的求助提示。另外, 也能从求助(Help)窗口拷贝例程到你的程序中。

在第二课, 你可使用求助系统。

Turbo C++ 也提供“最短的求助例子”使其便于查看。在 IDE 的底端列出了当前屏幕的热键可用的命令, 在菜单和对话框中也是这样, 并且还提供当前菜单和对话框操作项目简明的描述。

3.2.1 产生一个新文件

Turbo C++ 的 IDE 的一大特征是多窗口性,你能在某个时刻打开多个编辑窗口,可在每个窗口打开不同或是相同的文件,在两个窗口之间切换或是拷贝与粘贴,并且从一个窗口可容易地转到另一个窗口。

如果 BARCHART.C 已在编辑窗口,就可准备开始编辑。但是由于你不想改变原来的程序,用 File|New 打开一个新的编辑窗口。

新的编辑窗口文件是 NONAME01.C(2 号窗口),之后你取名为 MYCHART.C。新的编辑窗口有双线显示,告之你这是活动窗口。

为了直接进入任一个编辑窗口,按 Alt 和编辑窗口编号即可。

现在回到第一个编辑窗口(用鼠标或 Alt-1)。要求你拷贝所有的 BARCHART.C 文件到 NONAME01.C(如果你不熟悉在编辑窗口拷贝,看下表或是在编辑器中按 F1。编辑命令的全部列表,参见《用户手册》第三章“编辑命令”)。

表 3.4 编辑窗移动

移动	键盘操作	鼠标操作
任意方向移动 一个字符或行	按上、下 左、右键	移箭头到卷动处的后面,按键 卷动
返回行首、尾 上、下翻页 返回文件开始或结束	按 Home 或 End 按 PgUp 或 PgDn 按 Ctrl-PgUp 或 Ctrl-PgDn	移动箭头到行首或行尾处,按键 将箭头保持在卷动条之后,按键 在卷动条上按鼠标键移动,移动到 文件头或尾,松开

3.2.1.1 选择块

当作块标记时,你可利用键盘和鼠标来进行。

■利用键盘,把光标置于字符的开头或末尾,然后按 Shift 和箭头键(或 Home、End、PgUp、PgDn)来设置块。(当你利用箭头键时,你必须按 Shift)。

■利用鼠标,在块的头一个字符按键,然后拖动鼠标指示到块尾,按键即可。

文本中设置的块出现反象(高亮度)。

继续本课,在 BARCHART.C 中选择代码。

3.1.1.2 拷贝和粘贴

你在 BARCHART.C 窗口中已选定了文本,就选择 Edit|Copy(或 Ctrl-Ins 键)。它把选择的文本拷贝到内存中称之为剪取板的特殊地方挂起来,一旦文本进了剪取板,你就可在同一个窗口或不同的窗口将剪取板内的文本拷贝到新的位置。

移回到 NONAME01.C: 按 Alt 和窗口号码(或用鼠标键选择编辑窗口)。(如果不知道要移动的那个窗口的号码,使用 Window|List)。

为了把剪取板的文本粘贴到编辑窗口,选择 Edit|Paste(或按 Shift-Ins),在任何地文按鼠标键或按 Ctrl-KH。

现在你已经对文本作了改动, 存盘并且取文件名为 NONAME01.C

3.2.2 修改新文件

BARChart.C 程序对 0 到 50 分之间给出 10 个分数, 计算百分比, 然后在直方图中显示百分比, 并且打印出分数和百分比。

本节, 将对 BARChart.C 作一些改进:

- 改变 BARChart.C(NONAME01.C)版本, 以便它给出 0 到 35 分之间的 5 个分数值
- 利用编辑器的查找与替换特性改变变量名。
- 从求助窗口拷贝和粘贴文本

我们从编辑文件, 在不同的数值域中给出不同的分数值开始。这样做:

1. 把光标移到第七行末。
2. 利用 Backspace 回移到数 50 处, 键入 35
3. 移到下一行, 把数 10 改为 5。

3.2.2.1 查找与替换

变量名 `i` 不是一个具有许多信息的标识符, 所以将它改为 `index`, 因为这个变量名标明了其用途。改变单个字母比第一次输入有一小点麻烦, 每一处的变量 `i` 都要改为 `index`, 如果漏了一个, 将会有有一个未定义的变量而造成程序编译失败, 但又不能把程序中的所有字母 `i` 都进行替换。

很幸运, 查找菜单中包含有查找与替换的操作选择, 你可以利用如下的替换对话框:

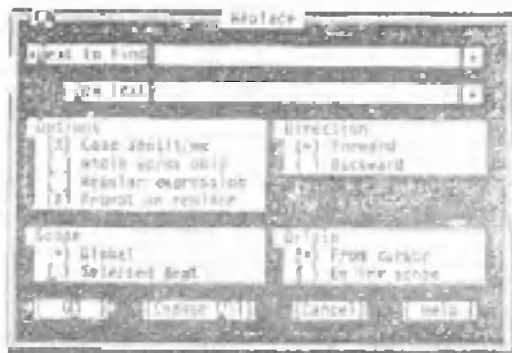


图 3.4 替换对话框

替换对话框包括有两种输入框, 一个收音机按钮, 一个操作检查框和四个控制键, 你要把程序 NONAME01 中所有的变量 `i` 改为 `index`, 所以从文件头开始(按 Ctrl-PgUp 或鼠标键)

1. 选择 Search|Replace 打开替换对话框, 当你第一次打开对话框时, Text to Find 输入框处于活动状态。

2. 键入 `i`, 这个你想要寻找的变量, 然后按 Tab 去激活 New text 输入框。

3. 键入 `index`, 这个你想要替换 `i` 的变量, 再按 Tab 移动状态到 Options 检查框。

为了检查所有字, 只要按 ↓, 然后按 spacebar 或鼠标键即可。

Case Sensitive 和 Prompt on Replace 已经检查了, 这正是你想要做的, 但你也需要去核实 Whole Words Only 以便不至在程序中的每个 `i` 处停止。当选择了这个项时, 按 Tab

或鼠标键离开, 进入 **Direction**。

4. 你要搜寻的方向在前方, 即从头开始, 用 **Tab** 进入 **Scope**。

5. 在 **Scope** 集中已经选择了 **Global**, 这意味着要搜索整个文件, 现在按 **Tab** 进 **Origin**。

6. 在 **Origin** 中, 按 **↓** 指到 **Enter Scope**(或按鼠标键)去进行选择。

7. 用 **Tab** 移到 **Change All** 钮, 并且按 **Enter** 去初始化查找与替换操作(或用鼠标键在每一个 **I** 处, 你要判断是否替换 **i**, 若要替换, 键入 **Y**, 否则键入 **N**。

3.2.2.2 在求助窗口粘贴

假如你要在直方图中加进标题:

Percentages, Exam 1A

在图形状态, 输出文本到屏幕的函数是 **outtextxy**, 描述这个函数的求助屏幕你可拷到剪取板, 命名为 **NONAME01**。这里显示怎样从求助屏拷贝 **outtextxy** 的例子到你的源序。

1. 从 **Help** 菜单中选择 **Index**。
2. 键入 **outtextxy**, 然后按 **Enter** 进入求助屏幕。
3. 本例作为一个块, 已经选中, 所以选择 **Edit|Copy Example** 把它拷贝到剪取板。
4. 按 **Esc** 关闭 **Help** 窗口。
5. 到文件尾, 把光标置于最后一行的开始。
6. 选择 **Edit|Paste** 把内容拷贝到你的文件。

你不需要使用一个整块。其中一部分是非常有用的, 有些部分必须进行修改以适合的要求, 而另有一些部分在这个例子里是没有用的。

一旦把这个例子拷贝到程序中, 就能修改想要保留的行, 然后去掉不用的, 例如:

1. 你可利用行

```
int midx, midy;
```

删去这一行(从加在你程序尾的文件块中), 然后把它拷贝到 **makegraph** 函数变量说段(第 35 行)。

2. 把下面两个拷贝行粘贴到 **makegraph** 的 **for** 循环的结束括弧后(第 49 行)。

```
midx = getmaxx()/2;
```

```
midy = getmaxx()/2;
```

3. 把它们改为读语句

```
midx = (getmaxx()/2) - (textwidth("Percentages,Exam 1A")/2);
```

```
midx = getmaxx() - 10;
```

4. 修改了 **midy** 语句后, 把拷贝行粘贴到 **makegraph** 的正确位置:

```
outtextxy(midx,midy,"This is a test")
```

5. 修改它为读语句

```
outtextxy(midx,midy,"Percentages,Exam 1A");
```

你需要去掉在文件尾已损坏的 **outtextxy** 语句, 方便的是, **Edit** 菜单提供击两次键来置块删除的命令。

1. 选定你想要消除的块: 在 **outtextxy** 中开始设置, 文件尾再设置。
2. 选择 **Edit|Clear**(或按 **Ctrl-Del**)。

3.2.2.3 存储修改过的文件

你已经对程序作了部分改动, 从头开始把它存到磁盘上, 选择 **File|Save**(或按 **F2**), 这要调用存储文件对话框。在输入(input)框内, 键入 **MYCHART.C**, 然后按 **Enter**。

3.3 第二课: 编译和运行

本课花五分钟。

在这一课, 你将

- 用 **Compile|Make EXE** 命令编译 **MYCHART.C**。

- 用 **Run|Run** 命令运行 **MYCHART.EXE**(你能见到输出)

利用 **Compile** 菜单去编译程序。(本教材全菜单选择设置为 **off**, 这个编译菜单仅显示两项。为了查看所有编译菜单选择的操作, 看《用户手册》第一章“IDE 指南”。

注: 如果出现错误信息说 **Turbo C++** 不能找到头文件, 那么可能是 **Turbo C++** 没有安装缺省子目录。详见第一章

1. 选择 **Compile|Mode** 执行文件或按 **F9** 去产生 **MYCHART.EXE**。编译窗口出现, 如果程序有错误, 它会在此处告之(如果这样, 改正并且重新编译)。当程序成功地编译和连接后, 窗口显示闪烁信息:

Success:Press any key

2. 按任意键返回到你的程序

为了运行 **MYCHART**, 可以选择 **Run|Run** 或按 **Ctrl-F9**。(其它在 **Run** 菜单的命令用于调试)。**Turbo C++** 有从 **IDE** 到 **User Screen**(用户屏幕)的开关, 这里 **MYCHART** 有输入语句。

每一次 **MYCHART** 都要求你输入(一共五次), 输入一个 0 至 35 之间的整数。一旦你输入了最后一个数, **MYCHART** 就显示所有的五个数值和它们的百分比。**User Screen** 既显示文本, 也显示图形输出。

按 **Enter**。现在 **MYCHART** 用直方图显示百分比。注意很美观的标题在直方图的中央。如果你对这一切感到惊奇, 查看这行:

```
midx = (getmaxx()/2 - (textwidth("Percentages,Exam 1A")/2);
```

当光标位于函数 **textwidth** 上时, 按 **Ctrl-F1** 以得到指定语言求助提示(看《库函数参考》第一章“运行库”), 然后按 **Enter** 键返回 **IDE**。

3.3.1 关闭编辑窗口

现在你完成了 **MYCHART.C**。关闭它的编辑窗口, 选择 **Window|Close**(或按 **Alt-F3**)。同样也可关闭 **HARCHART.C**。

如果你使用鼠标, 可在关闭框的左上角按鼠键关闭编辑窗口。

3.4 第三章: 退出 Turbo C++

在完成了对文件的工作后, 最后要做两件事:

- 把已改动过的文件存到磁盘。

- 退出 **Turbo C++** 的 **IDE**, 返回到 **DOS**。

你已经把文件存了盘, 所以最后一步工作是退出 **Turbo C++**, 返回到 **DOS**。

■选择 File|Quit, 或按 Alt-X。

3.5 进一步的资料

本教材是为大多数普通用户编的 Turbo C++ IDE 部分的速成介绍; 许多特性刚刚涉及或还未涉及。注意, 在《用户手册》第一章“IDE 指南”中, 你可找到 IDE 每一个部分的进一步信息。

第四章 C 简介

在此之前,如果你从未用 C 语言编过程序(或者有些遗忘),那么请读一下本章内容。本章从简单的例子开始,逐渐引入较复杂的程序,使你能够清楚这些例子是如何用程序实现的。你将学习如何解决一系列的诸如数值、字符和图形在内的问题,而且在设计和组织程序方面,你也可以获得极大的启示。在 EXAMPLES 子目录中,你可以查到更为复杂的源程序代码。这里我们用这种方法提供源程序,使你能够易于装入、编译和运行这些程序,这是学习 C 语言的最佳途径之一。

4.1 回顾

C 是 70 年代为 UNIX 操作系统而开发的,并伴随 UNIX 而发展壮大。当市场上有了足够大容量的微机后,C 编译程序开始在这些机器上运行。1978 年,Brain W.Kernighan 和 Dennis M.Ritchie 在其首版的书《C 编译语言》中给出了 C 的经典定义;五年后,ANSI(美国国家标准协会)开始给出语言的新标准。这个标准解决了经典定义中的二义性,并给出了 C 的新特点,这包括了对引用宏定义原型的功能调用的控制。Kernighan 和 Ritchie 在其再版的书中也详细讨论了 ANSI 的标准。Turbo C++ 是 ANSI 的最新标准,它也可以全部运行由 C++ 调用的 AT&T 的面向对象的 2.0 版本。C++ 面向对象的特征将在第五章“C++要素”和第六章“掌握 C++”中加以讨论。

4.2 基本编程操作

通过装入和运行已设计好的程序,你将领会本章的所有例子。

计算机程序在目的、风格和复杂性上变化很大,但是,几乎所有的程序都要经过如下的三个阶段:

- 描述、收集和存储数据
- 处理数据以获得预期结果
- 显示或存储结果

任何程序使用的数据都必须进行描述,以使得 C++ 知道该如何存储和恢复这些数据;同时,内存也必须进行分配,使程序获得所期望的空间;程序必须通过某种方法获取实际数据放入存储器中——这包括从键盘读入字符,从文件或磁盘恢复数据,从电话线上接收数据,或者使用某种输入设备。

一旦数据用数值变量、字符串、数组、或更为复杂的数据结构存储起来,下一步,它将被处理。处理过程随程序的目的而变化,这是因为:由处理数据的公式展开成的程序要计算出结果,而文字处理需要重排正文以适合新的版面。

一旦进行数据处理,结果必须以某种形式提供给用户。正文可以重排在屏幕上或传输给打印机;程序中的展开量也可以再现其新值。而大多数的数据需要存入磁盘以备后来使用。

下面的一个短小程序,展示了如何使用上述的程序设计步骤:

调试这段程序,装入并运行 INTRO1.C,请记住,这些例子都在 EXAMPLES 子目录

中。

```
/* INTRO1.C--Example from Chapter 4 of getting Started */
#include <stdio.h>
int main()
{
    int bushels;
    float dollars,rate;
    char inbuf [130];
    printf("How many dollars did you get?");
    gets(inbuf);
    sscanf(inbuf,"%f",&dollars);
    printf("por how many bushels?");
    gets(inbuf);
    sscanf(inbuf, "%d", &bushels);
    rate = dollars / bushels;
    printf("You got %f dollars for each bushel\n",rate);
    return 0;
}
```

本段程序的第一行——`main()`，定义一个函数(Function)和一组关联的程序指令。函数是C程序的基本块，如同段落是故事的基本构成一样。每个C程序都有一个称为“main”的函数，大多数程序都有几个其它名字的函数，而语句却定义了当函数main执行时应当做的事；右大括弧表示程序指令组的结束。

如果函数比作是文章段落的话，那么语句就是文章的句子。提请注意的是每个语句的结束符是分号，C允许几个语句写在同一行。如果不引入语句结束符，则程序的阅读将是十分困难的。

一、描述

这部分包括开始的三条语句：

```
int bushels;
float dollars,rate;
char inbuf[130];
```

回忆一下，编写程序的第一个步骤是“描述、收集和存储数据”，在C中，你必须在每个数据项之前先对它加以说明，说明一个数据项，首先列出其数据类型，然后是其名字。在这个例子中，你将看到一个数据项，其类型是int(整型)，且其名字是bushels；第二个和第三个数据项为float类型(浮点数)，其名字分别为dollars和rate。以上这些数据项都称作变量(Variable)，因为它们的值都根据其运行环境而发生改变。第四个数据项是字符数组(Character array)，它可以记入并存储用户的输入。

二、收集和存储

程序中紧接着的下面六条语句读取和存储我们所定义的数据。printf语句提示给bushels、dollars输入其数值，gets和sscanf语句读取和存储这些值到相应的变量名中。Turbo C++程序中做的大部分实际工作是调用Turbo C++的库函数来完成的。你可以在

一条语句中给出函数的名字和必要的参数信息来调用一个函数。printf, gets, sscanf 都是库函数, 它们本身都不是 C 语言的自身组成部分。sscanf 语句中的 %d 和 %f 表示数据项以一个整数和浮点数的形式存放。这些专用格式将在后面介绍。

三、处理

下面一条语句 `rate = dollars/bushels`, 是程序的处理部分, dollars 除上 bushels 就得到每个抽屉中的美元数。

四、打印

最后一个语句, 也调用 printf 函数, 显示计算结果, 请注意, 该 printf 语句在一串字符信息之后跟了一个逗号, 然后才是变量 rate, %f 标明 rate 的值按浮点数格式打出。

当你运行这个程序时, 其输出将如下:

```
How many dollars did you get? 32
For how many bushels? 24
you get 1.33333 dollars for each bushel
```

4.3 C 程序的基本结构

下面一个例子将引入函数、注释和预处理指令 #include 和 #define。

装入并运行 SALESTAG.C

```
/* SALESTAG.C-Example from Chapter 4 of Getting Started.
```

```
SALESTAG.C calculates a sales slip. */
```

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

```
#define RATE 0.065 /* Sales Tax Rate */
```

```
float tax (float amount);
```

```
float purchase, tax_amt, total;
```

```
int main()
```

```
{
    char inbuf[130];
    printf("Amount of purchase: ");
    gets(inbuf);
    sscanf(inbuf, "%f", &purchase);
    tax_amt = tax(purchase);
    total = purchase + tax_amt;
    printf("\nPurchase is: %f", purchase);
    printf("\n      Tax:%f", tax_amt);
    printf("\n      Total:%f", total);
    return 0;
}
```

```
float tax (float amount)
```

```
{
    return(amount * RATE);
}
```

这个程序的第一行由符号`/*`引起, `/*`代表注释行的开始, `*/`表示注释结束。Turbo C++将忽略注释符间的所有字符。而你将用注释来表明程序的目的、函数和语句的作用。合宜的注释将使你的程序易于理解和记忆, 并帮助其它程序员修改你的程序。

由数字或`#`引起的行不是C语句。而是Turbo C++的指令, 这些称为编译指令(或预处理指令), 因为它们来指导编译器的操作。指令`#include <stdio.h>`告诉Turbo C++读入和编译文件`stdio.h`中的内容, `stdio.h`是Turbo C++为你安装的许多头文件(header file), 也称为包含文件(include file)中的一个文件, 这些文件包括库函数的描述(如`printf`, `gets`, `scanf`等)和标准C库的其它组成部分。编译器将利用这些描述把你的程序和库函数一道进行编译。有关这些头文件的信息请参看“程序员指南”的第二章“运行时间库交叉参考”。

程序中的下一行, `#define RATE 0.065`, 定义了一个宏替换: 它将告诉Turbo C++, 当在你的源程序中发现`RATE`时, 就用`0.065`代替。在一个长程序中, 税率`RATE`可能引用多次, 当税率改变之时, 你仅需修改定义语句并重新编译一下程序即可, 所有的引用之处将自动改正。这比自己去查找和修改这些引用之处花费的时间少, 并且提高了精确度。

另外一种指定常量的方法是用`CONST`关键字说明。

下一条语句定义了用户自己定义的函数`tax`, 而`tax`的函数体在随后定义。说明部分表达了这个函数将取到一个浮点数值, 并且返回一个浮点数值, 把说明部分放在此处, 以使Turbo C++确保你的程序不会传递一个错误类型的数值给这个函数。语句

```
float purchase, tax_amt, total;
```

说明了三个浮点变量。

`printf`, `gets`, `scanf`语句提示并读取`purchase`的值, 下面就是实际上的计算:

```
tax_amt = tax(purchase);
```

它调用用户定义的函数`tax`, 将`purchase`的值传递给函数`tax`。为了能知道`tax`做什么, 你可以跳过下面的语句, 找到`tax`的定义:

```
float tax(float amount)
{
    return(amount*RATE);
}
```

这个`tax`函数用它收到的`amount`的值去乘上常量`RATE`, 然后返回这个乘积。这样, 当语句行`tax_amt = tax(purchase);`执行时, 交易额`purchase`应上的税将由函数`tax`传回, 并且存入变量`tax_amt`中。事实上, 下一行中, 这个值将加上`purchase`后得到`total`, 即:

```
total = purchase + tax_amt.
```

在这个程序中, 一个独立的计算税额的函数`tax`似乎是不必要的, 但在更复杂的情形下将十分有用, 例如当存在有多个依据交易所在地而变化税率时。也许你不得不检查一下产品代号以确定该项在第一个地方是否可以征税, 在这种情况下, 分开的计算税收的机构将使得程序的主要部分易于领会, 如果必要的话, 你可在不影响程序其它部分的条件下改变计算税收的那部分程序。

`purchase`的输出值由于浮点运算的原因而与输入值并不完全匹配。

主程序的最后三行是输出`purchase`, `tax`和`total`的值, 例如下面的一组数值:

Amount of purchase:24.95

Purchase is: 24.950001

Tax: 1.621750

Total: 26.571751

很显然,为确定输出的格式,还有许多工作要作,如美元的总值只需要两位小数,并所有值都采用右对齐。这需要你学会如何指定格式和 `printf` 函数之后才能做到。(如果你涉及到分数钱数,在《程序指南》的第四章“存储模型、浮点数和覆盖”中有如何使用 `bcd` 方式获取更高精度值的简要说明)。

4.4 数值运算

数值是计算机使用的基本数据。正如你能知道的一样,计算机存储器实际内容是二进制数字,8位一组构成了一个字节(1bytes = 8bits),16位一组构成两个字节,即一个字(1word = 2bytes = 16bits)。甚至涉及文字和图形的基本处理活动都牵涉存储器中的数字序列。

4.4.1 数值数据类型

存储器的同一部分可以被不同类型的数值所占用,只是依赖于多个字节的数据成组存放。变量名(如 `total`)实际上提供一个或更多字节的内容来追踪内存的一个具体地址——这个地址由 Turbo C++ 在你第一次定义变量时指定。但是你和编译器必须在用给定的变量来表达数字、用多个字节存储和从变量的开始地址读取数字这些方面是一致的,这通过你说明变量的数据类型来做。例如:

```
int total,count,step;
```

```
float cost,profit;
```

每个数据类型代表了不同的数值种类,你必须为你的数据选择一个合适的类型。在这个变量说明中:

■变量 `total` 的类型是 `int`(整型),当你在语句中取用 `total` 的值时,程序将从 `total` 的地址开始取两个字节。

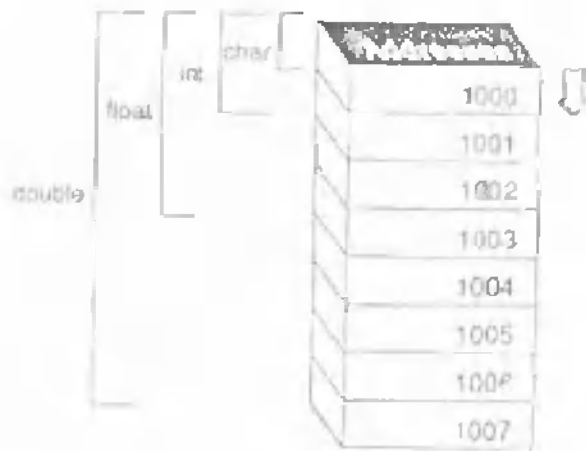


图 4.1 数值类型所占内存位置(增量为1个字节)

■变量 `cost` 的类型是 `float`(浮点型), 当你的程序调用 `cost` 时, 将从 `cost` 的首址开始取四个字节(这是因为一个浮点数需要两个附加字节表示数值的有效数字和以 2 的幂方形式表达的数值大小)。

表 4.1 列出了 Turbo C++ 的基本数据类型及变形类型。注意变形类型必须是可以接受的。这一章将向你说明如何使用这些类型。

表 4.1 数据类型、大小和范围

类型	位数	范围	应用
<code>unsigned char</code>	8	0~255	小数和PC字符
<code>char</code>	8	-128~127	很小数和ASCII字符
<code>enum</code>	16	-32768~32767	有序数
<code>unsigned int</code>	16	0~65535	大数和循环
<code>short int</code>	16	-32768~32767	计数、小数和循环控制
<code>int</code>	16	-32768~32767	计数、小数和循环控制
<code>unsigned long</code>	32	0~4294967295	天文距离
<code>long</code>	32	-2147483648~2147483647	大数、人口
<code>float</code>	32	$3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$	科学计数
<code>double</code>	64	$1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$	科学计数
<code>long double</code>	80	$3.4 \times 10^{-4932} \sim 1.1 \times 10^{4932}$	科学计数
<code>near pointer</code>	16	未提供	存储区操作
<code>far pointer</code>	32	未提供	存储区操作

4.4.1.1 整数

基本的整型是 `int`, 它可以表示负数和正数, 其范围是(-32768~32767)。

下面是应用整型数执行几步操作的例子。

```
#include <stdio.h>
main()
{
    int bags, pounds;
    int total;
    pounds = 50;
    bags = 1000;
    total = bags * pounds;
    printf("there are %d lbs. in 1000 bags of beans\n", total);
    return 0;
}
```

这个程序的输出结果有点让人吃惊:

There are -15536 lbs.in 1000 bags of beans

编译器不会在你存储一个与其数据类型不一致的值时向你提出警告

`bags*pounds` 的乘积为 50,000, 已经超过了正常 `int` 的表示范围, 当程序试图存储 50,000 时, 将会有溢出。如何解决这个问题, 请用 `long int`。

4.4.1.2 长变形

`long int` 通常缩写为 `long`, 它将提供更大范围的整数值, 在说明中用 `long total`; 就可以解决上面例子中的负数问题。

你也必须将 `printf` 中的 `%d` 改为 `%ld`。

这将为你的更多的豆子重量提供存储空间, 因为使用了 `long` 类型, 它将用 32 位来取代通常 `int` 用的 16 位, 其值范围也变为 $(-2,147,483,648 \sim 2,147,483,647)$ 。但 `pounds` 呢? 它仍然用 `int`, 因为每袋的重量不会超过 32,767 磅的范围。变量 `bags` 假定可以超过 32,767, 所以也用 `long`。为什么又不完全用 `long` 取代 `int` 呢? `long` 类型占有 4 个字节空间, 而 `int` 仅占用 2 个字节, 如果有许多变量, 可能会浪费许多空间。

```
int pounds;
```

```
long bags,total;
```

4.4.1.3 有符号和无符号变量

表 4.1 列出的所有数据类型有符号的都省略了 `signed`, 有符号数在存储时用其值的一值来表示该数的正负性质(那些加上 `unsigned` 的显然是无符号类型)。

确定用哪种数据结构, 应当考虑计算或其它操作的可能结果。

在你的工作中遇到的数值既有正数又有负数, 例如温度和银行结算。其它的值也许永远不会出现负数, 如一个行业不可能出现负的雇员数。对任何数据类型加上 `unsigned` 后, 强制其取值在正数范围, 因为其符号位不再使用。而是存储加倍的数值。举例来说, 通常 `int` 的范围是 $(-32,768 \sim 32,767)$, `unsigned int` 的范围则是 $(0 \sim 65,535)$, `unsigned long` 的范围是 $(0 \sim 4,294,967,295)$ 。前面的例子程序中如加入说明语句 `unsigned int total` 后, 其运算将会正确无误, 即使你的结果趋近 `unsigned int` 的极限。

因为省略的数据类型都是有符号的, 所以你不必特意说明 `signed int` 和 `signed long`

4.4.2 浮点数

许多数值包含用小数点隔开的小数部分。诸如用美元和美分的标价, 这些数值称为浮点数(通常也叫实数)。大多数的精确测量都涉及小数, 假如你在一家五金店购买螺钉, 恐怕你不得不用几分之几英寸来表述它的直径。浮点数据类型 `float` 就包括了这些情形。下面将是这样的一段程序。

试一下这个程序, 装入并运行 `INTRO2.C`

```
/* INTRO2.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    char inbuf[130];
    float num,denom;    /* numerator and denominator of fraction */
    float value;        /* value of fraction as decimal */
    printf("Convert a fraction to a decimal\n");
    printf("Numerator:");
```

```

gets(inbuf);
sscanf(inbuf, "%f", &num);
printf("Denominator: ");
gets(inbuf);
sscanf(inbuf, "%f", &denom);
value = num / denom; /* convert fraction to decimal */
printf("\n %f / %f = %f", num, denom, value);
return 0;
}

```

这个程序将提示输入一个小数的分子和分母。然后再将它们转换成小数后打出结果，如：

```

Convert a fraction to a decimal
Numerator:7
Denominator:8
7.000000/8.000000 = 0.875000

```

显然，变量 `value` 为保存一个小数，一定是浮点数据类型，而你可能还没有认识到变量 `num` 和 `denom` 也必须是 `float` 类型，而这又是为了使 `num/denom` 运算正确所必须的。再看下面的例子。

两个整数相除，其结果将用最接近的小于商的整数代替，即使最后结果要传递给一个浮点变量。

```

#include <stdio.h>
int main()
{
    int num = 3, denom = 4;
    float value;
    value = num / denom;
    printf("\n%f", value);
    return 0;
}

```

这段程序的结果为 0，更精确表述是 0.000000，而不是你所期望的 0.75。

4.4.2.1 双精度和长双精度

`double` 和 `long double` 类型相似于 `float`，只不过有更精确的数值范围。精度在科学和金融计算中是很重要的。你可以想到用你的系统打印大面额的支票，如 \$125,375,750.75。

```

#include <stdio.h>
int main()
{
    float amount = 125375750.75;
    printf("\nPay the sum of %f dollars\n", amount);
    return 0;
}

```

当支票打出后, 你将看到:

Pay the sum of 125375752.000000

结果多付了\$1.25, 这看起来不算什么, 但人们期望计算机 100% 的精确, 尤其是在涉及到钱的时候。发生这个错误的原因是 float 类型限制到七位数字的精确度, 而现在给变量 amount 的赋值即是十一位数字, 其中十位非零。如果你将程序中 float 改为 double, 再运行程序, 将看到正确的 amount 值。

4.5 变量

正如你已经知道的。每个变量在引用之前都必须说明。一条说明语句由数据类型和其后的一个或多个变量名组成, 说明语句简洁地告诉 Turbo C++ 你打算引用的特殊变量以及按什么类型来存储这个变量。如:

```
int house;
float total_pay, pay_rate;
long id_number;
```

4.5.1 初始化变量

你必须初始化每个变量, 给它一个特定值, 比如 0。你能想到下面一段程序将显示什么吗?

```
#include <stdio.h>
int main()
{
    int something;
    printf("%d", something);
    return 0;
}
```

这个结果将是不定的, 在我们的机器上, 它将是 -32,417, 你注意到在打印变量 something 之前并没有给它赋值吗? 除了全程和静态变量之外, C 中的所有变量都不能省略其值。(某种语言, 如 BASIC, 特别地给每个数值变量赋初值为 0)。变量 something 的初值是 Turbo C++ 确定其地址时碰巧在该地址单元内的值, 这个值是不可知的。事实上, 当你编译这个程序时, 你可以注意到提示窗口中的一个警告: “函数 main 中, 'something' 在定义之前可能被引用”, 当收到这样的警告后, 请检查命名的变量, 以确保在引用之前给它初始化。

4.5.2 赋值语句

你可以用赋值语句给变量赋值, 赋值语句由变量名、等号和要赋定的值组成, 例如:

```
count = 0;
total = purchase + tax_amt;
tax_amt = tax(purchase);
```

在第一条语句中用一个实际数值或数值常量给变量 count 赋值, 第二条语句用由 purchase 和 tax_amt 之和构成的表达式来给 total 赋值; 一个表达式是由值和操作符组成

的,是能产生一个单一值的组合体。在C中,可以用单一值的地方都可以用一个完整的表达式,可以把它赋给一个变量,可以传给一个函数处理,或者用 `printf` 语句输出。

第三条语句稍微有点复杂:首先它调用函数 `tax`,并把 `purchase` 的值传给该函数,然后函数就依据这个值和其它的信息计算税额,最后函数返回给调用语句一个值,换句话说,函数调用 `tax(purchase)` 被一个实际值取代了,比如 1.14,赋值语句将这个值赋给了 `tax_amt`。在C中,赋值语句使用函数调用是很普遍的。

4.5.3 复合赋值

C 通常允许把两个或更多的不同操作复合在一个单一语句中,你可以在一个语句中说明变量又给它赋值。如:

```
float total_expenses;
```

```
total_expenses = 0;
```

大多数C程序员都写为:

```
float total_expenses = 0;
```

你也可以在一条语句中同时给几个变量赋值,一个文字处理器在正文开始之前设置:

```
page = line = column = 1
```

这个语句能执行是因为赋值语句不仅赋给了一个值,而且为嵌入语句的其它部分提供了一个能用的值。即是 `column = 1` 把 1 赋给 `column`,并且使这个值 1 是可用的。从右到左,我们得到了等效的 `line=1`,依次下去,最后的赋值就是 `page=1`。

但是不要出格,每条语句最好说明和初始化一个变量,以便你能包括一个描述变量目的的注释,如下面的例子:

```
int lines = 0; /* Lines of text, ending in new line char */
```

```
int words = 0; /* Words are groups of characters surrounded */  
/* by space, tabs, or new lines */
```

```
int chars = 0; /* Every character is counted */
```

花时间作注释,也许可以使你警觉到一些可能的问题。比如,把 `chars` 定义成 `int` 真的合适吗?

4.5.4 命名

现在是考虑给变量如何命名的时候了。C 在这个方面是十分灵活的,用户提供的名字(称作为标识符)必须满足下面的规则:

- 所有标识符都必须由一个字母(a~z, 或 A~Z)或下划线(_)开头;
- 标识符的其它部分可以用字母、下划线和数字(0~9)组成,其它字符(如标点符号、控制字符)一律不用;
- 标识符的前 32 个字符有效,这意味着在 Turbo C++ 中, `The_total_amount_of_money_in_checking_account` 和 `The_total_amount_of_money_in_my_charge_account` 被当作同一个变量处理,当然,任何时候使用相同的名字都是很难办的。
- 标识符对大小写字母是有区分的,这意味着 `amount` 和 `Amount` 是完全独立的两个变量。

运用这些规则, `deduction`, `tax_status` 和 `amt_1099` 都是合法标识符,而 `1989_tax` 和

stop!都不是(1989_tax 用一个数字取代了字母或下划线作为开头;而 stop!中包含有惊叹号,而惊叹号既不是字母,又不是下划线,也不是数字。)

除了应用这些规则,还有必要给所命名的变量名赋与一定的含意,下面是一些建议:

- 名字最好能描述变量所包含的内容。a 不能告诉你任何信息,amt 又稍好,而 taxable_amount 最为清楚和专用。
- 用大写字母或下划线来隔开一个长标识符中的各个单词。PricePer100 或 price_per_100 就比变量 priceper100 较为容易阅读。
- 用注释来描述变量的属性和目的,尤其是当它不很明显时。

4.6.5 成组输入: sscanf

你已经看过这样的例子,为了从键盘获取数据而使用函数 gets 和 sscanf。sscanf 把数据送入指定的变量,并且可用不同数据类型(如存储是有差异的,你需要用一种方法来告诉函数 sscanf 你想存放的数据的类型。让我们更严密的剖析一下 sscanf。下面是调用 sscanf 的句法:

```
sscanf(buffer,"format string",(&address,address,...))
```

对比一下前面看到的 sscanf 语句:

```
sscanf(inbuf,"%f", &num);
```

buffer 是 gets 用来临时存放数据的输入数组。格式串(format string)包含一个或更多的格式说明,格式说明由符号%后跟一个代表类型的字母组成,格式说明组安置在一对双引号之间。例如%d说明是int值,%f标明为float,%s指定为字符串,格式串%c%d说明一个单一字符和一个int值。最常用的格式见下表4.2。

表 4.2 sscanf 和 printf 格式说明

功能	sscanf	printf
值大小(修饰数据类型)		
short integer	%hd	%ld
long integer	%ld	%lf
double	%lf	%lf
待读或显示的数据类型		
单个字符	%c	%c
signed integer	%d	%d
signed double或float(指数格式)	%e	%e
signed double或float(小数格式)	%f	%f
字符串	%s	%s
无符号十进制数	%u	%u

这个表也列出了 printf 的格式说明,注意使用格式说明时实际情形是很重要的。

sscanf 的其它必要部分存放数据的地址。C 中的每个变量都有一个指定的内存地址,大多数时间,你可不必考虑变量的地址,只是引用变量名来获取其值。例如 count+1 是

把变量 count 的当前值加 1。而在 sscanf 中，不需要变量值，需要的是变量的地址；地址操作符 & 用来取一个变量的地址。下面的程序就体现了变量地址和变量值的差异。

调试这段程序，装入并运行 INTRO3.C。

```
/* INTRO3.C--Example from chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    char inbuf[130];
    int number = 10;
    printf("Address of variable number = %ld\n", &number);
    printf("Value stored at variable number = %d\n", number);
    printf("Enter a new value for the variable: ");
    gets(inbuf);
    sscanf(inbuf, "%d", &number);
    printf("New value stored at variable number = %d\n", number);
    return 0;
}
```

输出结果如下(地址可变):

Address of variable number = 65498

Value stored at variable number = 10

Enter a new value for the variable: 33

New value stored at variable number = 33

下面程序说明了 sscanf 的更多变化。

调试本程序，装入并运行 INTRO4.C，注意，我们是如何让描述所需值的 printf 和 puts 优先于 sscanf 的。

```
/* INTRO4.C--Example from chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    char inbuf[130];
    long transaction_number;
    int cashier_number;
    char transaction_code;
    float purchase_amount;
    printf("Enter transaction number: ");
    gets(inbuf);
    sscanf(inbuf, "%ld", &transaction_number);
    printf("\nEnter your cashier number: ");
    gets(inbuf);
    sscanf(inbuf, "%2d", &cashier_number);
}
```

```

printf("\nEnter transaction type code: ");
gets(inbuf);
sscanf(inbuf, "%c", &transaction_code);
printf("\nEnter amount of purchase: ");
gets(inbuf);
sscanf(inbuf, "%f", &purchase_amount);
return 0;
}

```

第一个 `sscanf` 将读入一个不超过 8 位的长整数(long int), 第二个 `sscanf` 读入一个不多于 2 位数字的整数(int)。(在实际应用中, 你不得不进行更进一步的错误检查, 一件事是如果你打入超过指定位数的数字, `sscanf` 将把多余的存入下一个指定变量或者忽略不计, 如果后面没有其它变量的话, 正如这个例子程序一样。)

你可以在一个 `sscanf` 语句中申请为多个变量赋值, 例如,

```

gets(inbuf);
sscanf(inbuf, "%8ld%2d", &transaction_number, &cashier_number);

```

把前面的两条 `sscanf` 语句合二为一。因为仅调用一次 `gets`, 所以在一行读取。

缺省地, `sscanf` 假定用户用空格隔开数值变量。在格式中可以包括其它操作字符, 这种情况下, 必须严格敲入每一个值。

```

#include <stdio.h>

int main()
{
    char inbuf[130];
    int hours, minutes, seconds;
    printf("Enter new time in hh:mm:ss ");
    gets(inbuf);
    sscanf(inbuf, "%d:%d:%d", &hours, &minutes, &seconds);
    return 0;
}

```

4.5.6 显示变量值

与 `sscanf` 正好相反, `printf` 是将变量的值输出在显示屏上。`printf` 的调用格式如下:

```
printf("format string", item, item,...)
```

格式串中可以有任何要显示的任何文本数据, 如果要显示多个项(变量、表达式、常量等)的值, 请为每个项提供一个转换说明, 这个转换说明类似于前面讨论的 `sscanf` 的格式说明, 只是增加了控制输出格式的信息, 说明符由百分号%和其后一个类型符组成。

在前面, 已经用到了 `printf` 语句来显示交易额(purchase), 税额(tax)和总额(total)的值:

```

printf("\nPurchase is:%f", purchase);
printf("\n      Tax :%f", tax_amt);
printf("\n      Total :%f", total);

```

而显示的结果如下:

Purchase is: 24.950001

Tax: 1.621750

Total: 26.571751

那么, 如何使用转换说明使得 Turbo C++ 在显示多个变量时做到右边和小数点都对齐呢? 办法之一是在百分号后说明要显示的总位数和小数点位数, 如:

```
printf("\nPurchase is:  $%6.2f", purchase);
```

```
printf("\n      Tax:  $%6.2f", tax_amt);
```

```
printf("\n      Total:  $%6.2f", total);
```

显示结果如下:

Purchase is: \$ 24.95

Tax: \$ 1.62

Total: \$ 26.57

下例说明了 printf 如何以不同的格式显示几种数值类型。

```
/* INTRO5.C--Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
```

```
    int  int_num = 999;
```

```
    float float_num = 99.99895;
```

```
    long double big_num = 1250500750.75;
```

```
    printf("12345678901234567890\n");
```

```
    printf("%d\n", int_num);
```

```
    printf("%6d\n", int_num);
```

```
    printf("%f\n", float_num);
```

```
    printf("%6.3f\n", float_num);
```

```
    printf("%e\n", big_num);
```

```
    printf("%1e\n", big_num);
```

```
    return 0;
```

```
}
```

这段程序输出如下:

12345678901234567890

999

999

99.998947

99.999

-3.55361e-207

1.250501e+09

第一个 printf 语句打出 20 个数字作为其它行数值位数的数值, 它也表明了 printf 对齐非要输出变量或表达式值, 它也能打印出一段你需要的文本字符串。下面一个语句打出 int_num 的值(999); 第三个语句指明了域宽为 6, 因为是右对齐, 数值 999 就显示在三个空

格之后。(如果的标明的域宽过小,不能印下整个数值,则域宽说明将不予考虑而全部显示所有数字。这是说,当用%2d说明时,999不会被印成99)。

下面两条语句显示 float_num 的值,第一条语句显示的值并不严格精确,显示值 99.99847 而不是 99.99895,这是因为 float 类型说明的变量只保证七位数字精确,printf 却保证尽可能印出小数点左面的所有数字,然后再加上右面的六位数字,以致末尾的有些数字不准确。对许多应用而言,这点误差并不要紧;但当必须保留数据所涉及的精度时,使用 printf 显示更多位的数字将产生不精确,并可能使用户误解,认识到这一点是很重要的。

为避免错误数字的显示,用精度说明符来指定小数点后的数字位数。下一条语句用 %6.3f 规定了域宽为 6 并且有 3 位小数,当给定的精度小于可用数字的位数时,后面的数字将进行适当的进位或舍去。

4.5.6.1 printf 中的类型转换

不要让指定的有效数字超过数据类型所能提供的位数

printf 使用转换说明来强制将数值转换成指定的类型,而忽略该值的实际数据类型。例如, float 变量 dollars 的值为 24.95,当使用 %d 说明时,其尾部将为 0 或任意随机值,因为这时 printf 只读 dollars 开始的两个字节(int)而不是四个字节(float);另一方面,如果说明的变量是 int 类型,但试图用 %f 来显示,则会出现错误结果,因为这时候 printf 将从 2 字节的变量中读取 4 字节。

为了处理各种格式,printf 变成了具有大量目标码的复杂函数,仅当你在程序中使用浮点数时, Turbo C++ 才连接处理浮点数的那部分代码,如果程序中未定义浮点类型,而你却对 int 类型使用了 %f,则会出现运行错误,并提示你浮点数格式处理未被连接。printf 和 scanf 更多的协定和清楚的变化将由 C++ 的库提供。

前面的例子用 %f 说明来显示类型为 long double 的变量 big_num,很不幸,这个格式说明试图把它当成普通 double 型来印出,因而又出现了一个其有巨大负数的荒谬值。最后一条语句用 %lf 正确说明了 long double 类型。

4.5.6.2 使用转义符(\)序列的格式

有许多文本显示控制字符存在,例如制表符使光标跳到下一个制表位,换行符使光标移到下一行,换页符使一个新屏或文本从新一页开始。printf 允许在文本打印中包括有由反斜杠(\)前导的字符。反斜杠(\)称为转义符,这由于它将告诉 C++ 不要将下面的字符当成字面上的 n 或 f 或其它什么来解释,而是应当作为一个特殊的专用字符。

确实,你已经看到了用 printf 来显示的字符串中使用 \n 的例子,它的功效就象 BASIC 语言中自动前进光标或把打印头移到下一行开始一样,只是 C 中不作隐含。这就具有更大的灵活性,可以在同一行上显示独立的 printf 语句要输出的正文,而仅当你希望换新行时才进行。表 4.3 列出了 Turbo C++ 的转义符序列。

八进制和十六进制转义符序列不同于 Turbo C 2.0。参见《程序员指南》的“Turbo C++ 语言标准”部分。

■ “换行(NewLine)”在 MS-DOS 系统上等价于回车(CR)再加上换行(LF)。但在其它系统上,这一点并不一定正确。

■ 仅当 Turbo C++ 想给单引号或双引号赋予一个专用意义时,在其前面需要加上反斜杠。例如,“通常定界一个字符串,为了打印带引号的字符串,需用 \" 字符串\"”

的形式。

表4.3 转义符序列

转义序列	名字	意义
<code>\a</code>	响铃	发出一声
<code>\b</code>	退格	退回一个字符
<code>\f</code>	格式反馈	开始一新屏幕或新页
<code>\n</code>	换行	移到下一行开始处
<code>\r</code>	回车	移到当前行开始处
<code>\t</code>	水平制表	到下一制表位置
<code>\v</code>	垂直制表	往下移一定距离
<code>\\</code>	反斜杠	显示反斜杠本身
<code>\'</code>	单引号	显示单引号
<code>\"</code>	双引号	显示双引号
<code>\?</code>	问号	显示问号
<code>\ooo</code>		以八进制方式显示一个字符
<code>\xhhh</code>		以十六进制方式显示一个字符

■八进制或十六进制值通常用作专门的图形字符或打印控制字符。例如 `printf("xDB")` 在 IBM PC 上显示一个固定的方块字符。

除 `\n` 之外，最常使用的转义符序列可能是 `\t`，它常用于对准数表。如下面的程序：

```
/* INTRO6.C-Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
```

```
    int i = 101, j = 59, k = 0;
```

```
    int m = 70, n = 85, p = 5;
```

```
    int q = 39, r = 110, s = 11;
```

```
    printf("\tWon\tLost\tTied\n\n");
```

```
    printf("Eagles\t%d\t%d\t%d\n", i, j, k);
```

```
    printf("Lions\t%d\t%d\t%d\n", m, n, p);
```

```
    printf("Wombats\t%d\t%d\t%d\n", q, r, s);
```

```
    return 0;
```

```
}
```

它按表的格式产生如下结果：

	won	lost	Tied
Eagles	101	59	0

Lions	70	85	5
Wombats	39	110	11

4.5.7 算术运算符

现在已经知道如何获取和显示不同种类变量值的方法，让我们再清楚地看一下 Turbo C++ 提供的各类操作。已经碰到的操作符有：赋值运算符(=)及四则算术运算符(+, -, *, /)。

这些操作符都按期望的运算进行工作，尽管有一些差异，例如两个 int 数相除得到一个整数值，并舍弃任何小数部分。在操作符间，存在有称为优先级的一个特别秩序，如算术运算符中，乘(*)和除(/)的优先级高于加(+)和减(-)。试着预测一下下面这段程序的四个显示结果(装入并运行 INTRO7.C)。

```
/* INTRO7.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    float result;
    result = 1.0 + 2.0 * 3.0 / 4.0;
    printf("%f\n", result);
    result = 1.0 / 2.0 + 3.0;
    printf("%f\n", result);
    result = (1.0 + 2.0) / 3.0;
    printf("%f\n", result);
    result = (1.0 + 2.0 / 3.0) + 4.0;
    printf("%f\n", result);
    return 0;
}
```

使用括弧并没有害处，即使严格说来是不必要的，使用括弧增加了表达式的可读性。这是这段程序的结果，你的预测又是多少？

```
2.500000
3.500000
1.000000
5.666667
```

在第一个表达式中，乘法 2.0×3.0 首先执行得到结果为 6，下一步 $6/4.0$ 得到 1.5，再加上 1.0 得最后结果 2.5。注意当操作符具有相等的优先级(*和/优先级相等，+和-优先级相等)，操作先左后右。

在第二个表达式中，除法首先执行，然后是加法，得结果为 $0.5 + 3$ ，即 3.5。

第三个表达式，括弧中的 $(1.0 + 2.0)$ 首先被运算，得到结果为 3.0，再除 3 后得 1。

最后，括弧中间是 $1.0 + 2./3.0$ ，括弧内的运算规则如下：2 除 3，再加 1，其结果为 1.666667，再加上 4.0 之后为 5.666667。

取模%

模运算符%代表两个数相除，并只保留余数。例如 $5\%2$ 其结果是 1， $18\%3$ 为 0，这是因为 3 正好整除 18。

4.5.8 算术和类型转换

如果用一个 `int` 数加上一个 `float` 数，将会发生什么事呢？你也许期望结果是一个 `float` 数，并保留小数部分，事实上也如此。Turbo C++ 根据一系列规则，实现较低的类型向较高的类型转换(见下表)。从下面的表中，你可以看到 `int` 和 `float` 相加，`int` 将提升为 `float` 类型，然后是两个值相加，得到一个 `float` 值。

在表中，有一部分类型未讨论。

表 4.4 算术运算中的类型转换

Type	Converts to
These types are converted automatically:	
<code>char</code>	<code>int</code>
<code>unsigned char</code>	<code>int</code>
<code>signed char</code>	<code>int</code>
<code>short</code>	<code>int</code> ¹
<code>unsigned short</code>	<code>unsigned int</code> ¹
<code>enum</code>	<code>int</code>
<code>float</code>	<code>double</code>
Then these rules are applied, until both operands have the same type:	
<i>If either operand is...</i>	<i>The other is converted to...</i>
<code>long double</code>	<code>long double</code>
<code>double</code>	<code>double</code>
<code>float</code>	<code>float</code>
<code>unsigned long</code>	<code>unsigned long</code>
<code>long</code>	<code>long</code>
<code>unsigned</code>	<code>unsigned</code>

类型强制转换

有时把一个数据项强制转换成指定类型是很有用且是必要的，例如有程序：

```
#include <stdio.h>
int main()
{
    int a = 5, b = 2;
    printf("%d", a / b);
    return 0;
}
```

其结果为 2，因为整除将舍去小数部分，但如果程序改为：

```
#include <stdio.h>
int main()
```

```

{
    int a = 5, b = 2;
    printf("%f", (float) a / (float) b);
    return 0;
}

```

其中 `a` 和 `b` 的值在执行除法运算之前就被强制转换成括弧内的类型 `float`，所以表达式的值就为 `2.5`。这种转换称为类型强制转换。

4.5.9 算术与赋值联合运算

编程中的一个普通操作是给一个变量添加一个给定量(增值)。例如，一个统计单词的程序，当它每查到一个单词时，就作如下的操作，`total_words = total_words + 1`。在后面，你也将看到涉及重复性增加或减少一个数时是如何循环的，直到达到某一个指定限值为止。

在 C 中有一种快速方法来在一个步骤中执行算术和赋值运算。你可以把任何的二元算术运算同赋值运算操作合二为一。前面的语句因而也可以写为 `total_words += 1`。这条语句是按如下方式读“将 `total_words` 的当前值加 1，将其结果赋给 `total_words`。”类似地，图书平衡检测程序也可以执行如下语句 `balance -= check_amt`(从 `balance` 中减去已检测的数量，然后再作为新值赋给 `balance`)。稍微有点不常见的组合 `*=` 和 `/=` 的工作也采用同样方法。

4.5.10 增量和减量

严格加 1 和减 1 是如此常见，以致为此专门提供了两个运算——增量(`++`)和减量(`--`)，因此 `++total_words` 与 `total_words += 1` 是严格相同的，一个对空短程列车作减量计数的程序可以用语句 `count--` 来进行循环，直到为零。

增量和减量运算符可以放在要影响的变量的前面或后面，放在前面时先将变量值加 1 (或减 1)，然后结果用整个表达式来计算，当操作符放在变量后面时，首先应用变量的当前值进行计算，然后把这个操作作用于变量上。例如：

```

#include <stdio.h>
int main()
{
    int val = 1;
    printf("val is %d and then post-incremented\n", val++);
    printf("val is now %d\n", val);
    printf("val is pre-incremented to %d\n", ++val);
    return 0;
}

```

其结果是：

```

val is 1 and the post-incremented
val is now 2
val is pre-increment to 3

```

在第一个 `printf` 语句, 当打印 `val` 时其值仍然是 1, 但随后变为 2, 这可以在第二个 `printf` 中可以看出; 在第三个 `printf` 语句, `val` 首先进行增量运算, 所以当用 `printf` 显示时, 其值为 3。

4.5.11 位运算

有时, 你发现不得不对内存中构成每一字节的各位进行操作, 而 C 提供了一系列的位运算操作, 见下表 4.5。

表 4.5 位运算操作符

操作	意义
<code>&</code>	与, 只有两者为1, 结果才为1
<code> </code>	或, 只要一个为1, 结果为1
<code>^</code>	异或, 有且只有一个为1, 结果为1
<code>>></code>	右移, 对于 <code>signed int</code> , 左边用符号位填充 对于 <code>unsigned int</code> , 左边用0填充
<code><<</code>	左移, 右边用0填充
<code>~</code>	取反, 每位由0变1, 由1变0

下面的程序说明了这些操作的用法。

```
#include <stdio.h>
int main()
{
    printf("1 & 1 is %d\n", 1 & 1);
    printf("1 | 1 is %d\n", 1 | 1);
    printf("1 ^ 1 is %d\n", 1 ^ 1);
    printf("255 << 2 is %d\n", 255 << 2);
    printf("255 >> 2 is %d\n", 255 >> 2);
    printf("~255 is %u\n", ~255);
    return 0;
}
```

请记住, 整数 1 在内存存储中用 2 字节表达为 0000000000000001, 255 表示成 0000 0000 1111 1111。其输出是:

```
1&1 is 1
1|1 is 1
1 ^ 1 is 0
255 << 2 is 1020
255 >> 2 is 63
~255 is 65280
```

注意, `&` 操作的结果为 1 仅当对应的位都为 1, 这在用位模式对一个值进行位屏蔽选

择时相当有用，而位模式中想滤掉的位置为 0。| 操作在一个或两个为 1 时得到 1，如果你想保证某一位置位，两个相“或”的值对应那位至少有一个为 1。

第四个语句将 255(0000 0000 1111 1111)左移两位得(0000 0011 1111 1100)，因为在二进制数中，每一位都是前一位的两倍，因此这等价于 $255 \times 2 \times 2$ ，即 1020。在下一条语句，255 右移两位，所以(0000 0000 1111 1111)变为(0000 0000 0011 1111)，即 63。最后一条语句把(0000 0000 1111 1111)转成(1111 1111 0000 0000)，这等价于用 65535 减去 255，得到 65280。

4.6 表达式

你已看到了使用表达式的各种语句，现在是回顾一下表达式一般是如何运算的时候了。表达式是一个由变量、常量和数字的任意组合体，它由一个或多个运算符聚合在一起并产生一个单一的结果值，而且尽可能产生对一个方面或多个方面的影响，下面这些都是表达式的例子：

```
purchase * TAX_RATE
dollars / bushels
count + +
STATUS & SWITCH_CN
```

一个表达式可以用赋值运算给变量赋值，也可当成一个单一变量供 printf 显示，前面的程序就用到了如下的语句：

```
printf("255 < < 2 is %d\n", 255 < < 2);
```

4.6.1 表达式求值

Turbo C++ 表达式求值时运用所涉及运算符的优先级，并首先从括号内的表达式开始。表 4.6 列出了 C 的所有运算符的优先级和结合性。结合性是编译器求给定操作符和操作数值时的方向。例如，各种赋值操作(=, +=, *=等)都是从右向左结合的(把右边表达式的值赋予左边的变量)，而算术运算(*, /, +, -和%)则是由左向右结合的。

优先级是求值的先后顺序。例如乘法在加法之前先做，所以语句：

```
count = 5 + 3 * 4
```

的结果是 17，而不是 32。

熟悉这些规则的最好办法是在 printf 中使用表达式，使你能看到结果，然后检查，看是如何得到结果的，(对应于第七章的“新的 IDE 调试”，向你说明如何使用 Turbo C++ 调试程序来求许多表达式的值)。

优先级和结合性的规则如下：

- 一元运算符(仅有一个操作数的运算符，例如++)有比二元运算符(如/)较高的优先级。
- 算术运算先于比较运算
- 大于和小于运算优先于相等和不等运算
- 比较运算先于位操作(左、右移除外)。
- 位操作优先于逻辑运算
- 逻辑与(&&)优先于逻辑或(||)。

■除了逗号(,)运算外,其它都先于赋值运算。

优先级与结合性在《程序员指南》的第一章“Turbo C++语言标准”中有全面的解释(包含C++运算)。

表 4.6 运算符的优先级和结合性

运算符	结合性
0 [] --> .	从左到右
! ~ ++ -- +	从右到左
- * & sizeof	
* / %	从左到右
+ -	从左到右
<< >>	从左到右
== !=	从左到右
&	从左到右
^	从左到右
	从左到右
&&	从左到右
	从左到右
?:	从右到左
= += -= *= %=	从右到左
&= ^= = <<= >>=	从左到右

4.6.2 表达式中赋值

你已经看到过在复合语句中使用赋值操作的例子,例如初始化变量 `total = 0`。赋值语句 `total = count = line = 0` 提醒我们每一重赋值都是一个表达式,你想,下面程序将显示什么结果?

```
#include <stdio.h>
int main()
{
    int val;
    printf("%d\n", val = 7);
    return 0;
}
```

如果预测是 7,那就对了。一个赋值语句作为表达式的值就是要赋定的值。

你已经见到过调用返回一个值的函数保存有一个值。例如, `total += tax(total)` 等价于:

```
tax_amt = tax(total);
total = total + tax_amt;
```


第一种形式消去了使用临时变量，但比较复杂。

4.7 字符和字符串

字符的存在比数字还多，现在来看一下字符和字符串。你知道，一个字符无论是字母表中的大写或小写字母、数字、标点符号、回车及 Ctrl-C 都是以一字节(8bits)存放的，字符值由 ASCII 码给定。各厂家前 128 个值是标准的，但是 IBM-PC 及兼容机用后 128 (128 到 255)个值作为图形和行式图形字符。在 C 中，类型 char(与类型 signed char 同义)的存取值范围是 0 至 127，负值用于专门目的(如指示错误或文件结束等)。如果想存取全部字符，请用类型 unsigned char 进行说明。

4.7.1 单个字符的输入与输出

下面是从键盘读入一个字符并送入变量的程序，(通读，并装入和运行 INTRO8.C):

```
/* INTRO8.C--Example from Chapter 4 of Getting Started 8/  
#include <stdio.h>  
int main()  
{  
    char lbuf[130];  
    char one_char;  
    printf("Enter a character: ");  
    gets(lbuf);  
    sscanf(lbuf, "%c", &one_char);  
    printf("The character you entered was %c\n", one_char);  
    printf("Its ASCII value is %d\n", one_char);  
    return 0;  
}
```

运行结果如下:

```
Enter a character: A  
The character you entered was A  
Its ASCII value is 65
```

记住，需要 one_char 的地址，因此在 one_char 前缀上字符&取地址

one_char 是 char 类型变量，sscanf 语句用 %c 格式说明是一个单一字符，并将键入值存入变量 one_char 中。下一条语句显示 one_char 的值(注意 printf 有 %c 转换说明符)，再下一条 printf 语句则打出键入字符的 ASCII 值，用 %d 来表达这个操作。这完成了把字符转换成 ASCII 值。字符是一个整数，但 C 中的某些设定(如 %c)却把这个值当作字符而不是一个整数来显示。

从键盘读取一个字符的另一种方法是使用库函数 getch 和 getche。先前的程序可以改写为:

```
/* INTRO9.C--Example from Chapter 4 of Getting Started 8/  
#include <stdio.h>  
#include <conio.h>
```

```

int main()
{
    int one_char;
    printf("Enter a character: ");
    one_char = getch();
    printf("\nThe character you entered was %c\n", one_char);
    printf("Its ASCII value is %d\n", one_char);
    return 0;
}

```

执行这个改后的程序，你键入的字符将不再显示在屏幕上。如果打算键入的字符是可见的，请用 `getche` 代替 `getch`。`gets` 读入一完整文本行后程序才继续运行，而 `getch` 和 `getche` 仅须读入一个字符。

4.7.2 显示一个字符

这个例子展现了该如何在屏幕上显示一个字符。`%c` 转换说明用于 `printf` 中显示一个单一字符，显示单个字符的另一种方法是采用库函数 `putch`，它在头文件 `conio.h` 中定义。`putch('>')` 显示给定字符 `>`，如果 `one_char` 是一个 `char` (某些时候是 `int`) 类型变量，`putch(c)` 则把 `one_char` 的值作为一个字符输出显示。

4.7.3 显示字符串

串是字符的序列，在 `printf` 语句中已经用到过串，例如：

```
printf("Enter a character:");
```

调用 `printf` 并告诉它，显示开始于 `E`，终结于空格符的一个字符串。双引号则告之 Turbo C++ 把这个字符组当作串处理。每个字符串连续存放在内存，`printf` 接收到第一个字符的地址。当字符串结束时，`printf` 又是如何知道的呢？当你定义一个字符串时，Turbo C++ 在末尾无形地加上一个空字符，此字符的 ASCII 值为 0，它表示成 `\0`。下面表格展示了串在内存中是如何存放的。

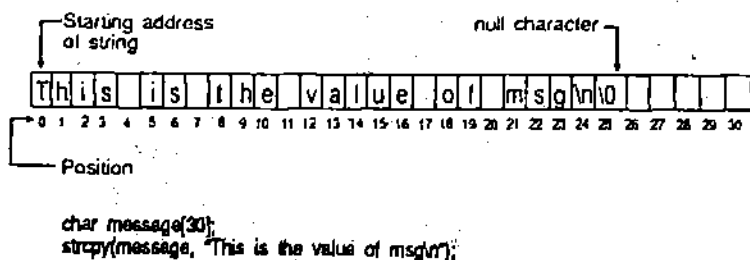


图 4.2 如何在内存中存放一个串

重要

`message` 实际上是定位这个串的指针，因此代码：

```
message="This is the value of msg\n";
```

使得 `message` 指向串开始的地方，它并不将字符串拷贝到 `message` 的存储区域，这就是为什

么你必须用 `strcpy` 来把串中字符拷贝给变量的原因。

为了方便, Turbo C++ 也提供了库函数 `puts`。这个函数把一个串写在标准输出设备上(一般是屏幕)。语句

```
puts("Enter a character:");
```

与 `printf` 几乎是一样的工作方式, 除了有一点外, 那就是 `puts` 在串尾自动放上新行符 `(\n)`, 使光标前进到下一行。如果程序不需要 `printf` 的被动进行格式能力, 可以用类似的 `puts` 显示字符串语句来替代原有程序中的显示部分。

双引号和单引号对串和字符有重要的差异,

■ `"a"` 是一个包含一个字符 `a` 的串, 并且其中还有一个空字符。

■ `'a'` 是单一字符 `a`。

因为串和字符是不同的数据类型, 对串运算的标识符和函数不能用于字符。为了显示 `"a"` 则要用 `printf` 的 `%s` 或 `puts`; 而 `'a'` 用 `printf` 的 `%c` 或 `putch`。

4.8 测试条件和作出选择

我们已经学到了 C 语言的许多元素, 至今为止, 所有程序都是从开始到结束直线运行, 但是, 通常程序都要根据某些值作出选择, 例如下面的代码段:

```
if(bill > credit_limit)
    puts("Consult with the manager");
```

如果 `bill` 的总量大于 `credit_limit` 的量, 则代码显示提请管理者考虑的信息。

4.8.1 使用关系运算符

前面程序中使用的 `大于(>)` 符就是一个关系运算符, 它表达了两个值之间的关系, 在这个例子中是: `bill` 是否大于 `credit_limit`。在计算机中使用两个简单的逻辑值: 如果关系为真, 则值为 1; 如果关系为假, 则值为 0。下表是所有关系运算符。

表 4.7 关系运算符

运算符	意义	示例
<code>></code>	大于	<code>5 > 4</code>
<code>>=</code>	大于或等于	<code>5 >= x</code>
<code><</code>	小于	<code>4 < 5</code>
<code><=</code>	小于或等于	<code>x <= 5</code>
<code>==</code>	等于	<code>5 == 5</code>
<code>!=</code>	不等于	<code>5 != 4</code>

这个简单的程序将告诉我们关系运算符如何使用。

读懂本程序, 装入并运行 `INTRO10.C`:

```
/* INTRO10.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>

int main()
```

```

{
    char inbuf[130];
    int first, second;
    printf("Input two numbers\n");
    gets(inbuf);
    sscanf(inbuf, "%d %d", &first, &second);
    printf("first > second has the value %d\n", first > second);
    printf("first < second has the value %d\n", first < second);
    printf("first == second has the value %d\n", first == second);
    return 0;
}

```

下面是用值 3 和 5 试运行的结果。

```

Input two number:
first > second has the value 0
first < second has the value 1
first == second has the value 0

```

注意，关系测试是一个表达式，因为它将得到一个值，而且能用 printf 显示，也可以赋给一个变量，如使用这样的语句 `hot = (temperature > 90)`，要小心不要把相等关系 `==` 和赋值运算 `=` 相混淆。试把程序中最后一条改为：

```

printf("first == second has the value %d\n",
       first == second)

```

表达式 `first == second` 对 `second` 求值，在这个例子中是 5。if、for 和其它语句的测试条件将任何非 0 条件都视为真。在用 `==` 的地方使用了 `=`，将看到产生不合适的程序行为(例如在循环非结束时停住)。

4.8.2 使用逻辑运算符

可以在一个测试中连接一个以上的条件，为此，用到了下表所列三个逻辑运算符。

表 4.8 逻辑运算符

运算符	意义
&&	AND(与)，两条件必须都为真
	OR(或)，至少有一个条件为真
!	NOT(非)，对结果值取反

例如，条件表达式：

`(employee_type == temporary) && (wage > 6.00)`，仅当雇员是临时性的并且他或她的工资超过每小时 \$6.00 时才为真。C 具有有效处理这种操作的功能：如果第一个条件 `(employee_type == temporary)` 为假，则第二个条件不作测试，因为 AND 操作时只要一个条件为假则结果为假。

表达式

`(employee_type = temporary) || (employee_type = hourly)`

为真仅当其两个条件之一为真，如果第一个条件为真，则不再测试第二个条件。

4.8.3 使用 if 和 if...else 的分支语句

我们已经见到了关系和逻辑运算符，现在来让它们开始工作。简单 if 语句形式为

`if(condition expression)`

`statement or group of statements;`

条件可以是单个关系表达式，也可以是通过逻辑运算符连接的关系表达式组。它必须用圆括弧包含。if 语句将根据条件表达式为真或为假来决定其行动，为真时，紧随其后的语句或语句组将被执行。如果打算让一组语句被执行，则用大括弧封闭这些语句。下面程序将用两个 if 语句来告诉你键入的数字是奇数还是偶数。

读通本程序，装入并运行 `INTRO11.C`：

`/* INTRO11.C-Example from Chapter 4 of Getting Started */`

`#include <stdio.h>`

`int main()`

`{`

`char inbuf[130];`

`int your_number;`

`printf("Enter a whole number:");`

`gets(inbuf);`

`sscanf(inbuf, "%d", &your_number);`

`if(your_number % 2 == 0)`

`printf("Your number is even\n")`

`if(your_number % 2 != 0) {`

`printf("Your number is odd.\n");`

`printf("Are you odd, too?\n");`

`}`

`printf("That's all, folks!\n");`

`return 0;`

`}`

在提示和存放 `your_number` 后面，程序使用一条 if 语句来测试数值是否为偶，采用模 (%) 运算。(因为所有偶数可被 2 整除，或者说模 2 的结果为 0)。如果数值为偶，则执行第一个 printf 语句。第二个 if 语句测试 `your_number` 是否为奇，也即不能被 2 整除，其余数非 0。如果此数为奇，你将看到：

`Your number is odd;`

`Are you odd, too?`

后面的两条 printf 语句都被执行，因为它们是由一对大括弧({})封闭的语句组。最后程序再印出，"That's all folks!"。因为这个 printf 语句不是任何一条 if 语句的一部分，所以无论 `your_number` 为奇或偶都要执行。

4.8.4 使用 if...else 的多重选择

前面的例子也许会难以看懂，如果一个数是偶数，则不可能又是奇数，为什么还要使用两个独立的测试？给 if 再加上 else 分支，上面的例子可简写。if...else 语句的格式是：

```
if(condition expression)
    statement or group of statements;
else
    alternative statement or group of statements;
```

把这个形式应用前面程序，将得到如下程序。

读通本程序，装入并运行 INTRO12.C:

```
/* INTRO12.C--Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
```

```
    char inbuf[130];
```

```
    int your_number;
```

```
    printf("Enter a whole number: ");
```

```
    gets(inbuf);
```

```
    sscanf(inbuf,"%d", &your_number);
```

```
    if(your_number % 2 == 0)
```

```
        printf("Your number is even\n");
```

```
    else
```

```
    {
```

```
        printf("Your number is odd.\n");
```

```
        printf("Are you odd, too?\n");
```

```
    }
```

```
    printf("That's all, folks!\n");
```

```
    return 0;
```

```
}
```

我们可以按照意愿嵌套 if 和 else 子句，尽管在多层嵌套后可能使编译器运行超出内存。假设要写一段把计算机分配给雇员的程序，且符合如下的条件：

■ 如果雇员是一个程序员，并且至少工作了两年，则给一台 80386PC。

■ 如果雇员是一个程序员且工作不超过两年，则给一台 80286PC。

■ 如果雇员至少工作了两年，但不是程序员则给一台 8088PC。

■ 最后，如果雇员不满足上面任何条件，则给一台 MacIntosh。

试拟一下如何用 if 和 else 语句来表达上面的这些条件，下面是一种方法：

```
if (employee_type == PROGRAMMER) {
```

```
    if(years_worked >= 2)
```

```
        give_employee(PC386);
```

```
    else
```

```

        give_employee(PC286);
    }
    else if (years_worked >= 2)
        give_employee(PC88);
    else
        give_employee(Mac);

```

注意，在这里我们使用缩进的方式展现条件间的依赖关系。程序首先判定雇员是否是程序员，如果是则检查其工作年限，配给合适的计算机；如果雇员不是程序，则用外面的 if...else 语句检测工作年限，把机器给不同资历的非程序员。

4.8.5 多重选择: switch

一长列的 if 和 else if 语句书写麻烦，且易混淆，隐伏错误。读下面程序，它将在响应菜单中根据用户键入的字符决定如何图解一系列数据。下面是这段程序。

试读这段程序，装入并运行 INTRO13.C:

```

/* INTRO13.C-Example from Chapter 4 of Getting Started */
#include <conio.h>
#include <stdio.h>
#include <ctype.h>
int main()
{
    char cmd;
    printf("Chart desired: Pie Bar Scatter Line Three-B");
    printf("\nPress first letter of the chart you want: ");
    cmd = toupper(getch());
    printf("\n");
    if(cmd == 'P')
        printf("Doing pie chart\n");
    else if (cmd == 'B')
        printf("Doing bar chart\n");
    else if (cmd == 'S')
        printf("Doing scatter chart\n");
    else if (cmd == 'L')
        printf("Doing line chart\n");
    else if (cmd == 'T')
        printf("Doing 3-D chart\n");
    else printf("Invalid choice.\n");
    return 0;
}

```

本程序显示一个菜单，然后再通过 getch 函数给 cmd 读取一个值，再向前，此值又经过函数 toupper 处理。这就保证程序只需对大写字母进行处理。if 和 else if 序列首先进行

分支，然后测试每一个有效值并执行相关的功能。最后一个 else 如同缺省情形一样，处理无效的值。

而 switch 语句使得这种多重分支易于编码。它采用形式为：

```
switch(value)
{
    case value statement or group of statements
    -----
    default:statement or group of statements
}
```

.....意味着可以有任意多个 case 子句

与第一个 case 子句后的值进行比较测试，如果相同，则冒号(:)后的程序将被执行，直到 switch 语句末尾或碰到 break 语句为止；如果不相同，则与下一个 case 的值进行比较测试，如此下去。子句 default 是可选的，如果没有 default 子句，并且也没有条件满足，则 switch 的语句就不会被执行。

前面的程序可用 switch 语句改写为：

```
/* INTRO14.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    char cmd;
    printf("chart desired: Pie Bar Scatter line Three-D");
    printf("\nPress first letter of the chart you want: ");
    cmd = toupper(getch());
    printf("\n");
    switch (cmd)
    {
        case 'P': printf("Doing pie chart\n"); break;
        case 'B': printf("Doing bar chart\n"); break;
        case 'S': printf("Doing scatter chart\n"); break;
        case 'L': printf("Doing line chart\n"); break;
        case 'T': printf("Doing 3-D chart\n"); break;
        default : printf("Invalid choice.\n");
    }
    return 0;
}
```

试验本程序，装入并运行 INTRO14.C。

每个 case 后面的 break 语句是十分重要的，它将导致程序执行时跳到 switch 语句的尾部。在每个 case 的最后一条语句，通常考虑是 break。例如，移去 case 'L' 后面的 break 语句，再运行程序，如果选择 L，将看到：

Doing Line chart

Doing 3-D char

L 和 T 情况下的语句都被执行了。换句话说, 如果省去 `break`, 程序将一直执行下去, 直到遇见第一个 `break` 或到了 `switch` 的最后一个语句。

有时, 这种行为很有用, 假设打算让程序的使用者能够用 D(delete) 或者 E(erase) 来移去当前文件, 你就可以将程序代码编成:

```
switch (cmd)
{
    case 'I' : insert_file(); break;
    case 'P' : format_file(current_file); break;
    case 'D' :
    case 'E' : erase_file(current_file);
}
```

因为 `case 'D'` 没有 `break`, 所以 `erase_file` 将在这种情况下执行, 同时 `case 'E'` 时也执行。

4.9 循环性重复运行

`if`、`else`、`switch` 语句的最大的特征是执行其测试仅为一次, 执行其后的语句也仅一次。但是许多计算机任务却涉及到重复, 它们也将涉及诸如“对此文件中的每一项使用同一个处理直到文件结束”或“对此数据集中的每一项采用同一个处理”等指令。对于这种任务, 将考虑使用循环。循环将导致一个语句或语句组重复执行, 并监控一个条件以便决定何时停止循环。C 提供了三种形式的循环语句, `while`、`do` 和 `for`。

4.9.1 while 循环

`while` 循环执行一个或多个语句的时间与指定条件为真的时间一样长, 其句法是:

```
while(条件)
    语句或语句组;
```

下面程序将从键盘读入数字, 并求出总和, 当键入 0 时, 程序将给出总和与平均数。

```
/* INTRO15.C--Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
    char inbuf[130];
    int number;          /* Number entered by user */
    int total = 0;        /* Total of numbers entered so far */
    int count = 1;        /* Count of numbers entered */
    printf("Enter 0 to quit\n");
    /* Priming statement puts a value into number */
    gets(inbuf);
    sscanf(inbuf, "%d", &number);
    while(number != 0)
    {
```

```

    total += number;
    gets(inbuf);
    sscanf(inbuf, "%d", &number);
    if(number == 0)
        printf("Thank you. ending routine.\n");
    else count++;
}
printf("Total is %d\n", total);
printf("Average is %d\n", total / count);
return 0;

```

试调本程序，装入并运行 INTRO15.C。

当键入第一个数字时，它将被 while 语句测试，当数字不为 0 时，则作下面三件事。

- 把键入的数值加入 total 中。
- 用 sscanf 读取一个新数字。
- 测试这个数字，如果不为 0，则键入个数递增 1。

4.9.2 do while 循环

do while 循环与 while 循环很相似，其格式为：

do 语句或语句组

while(条件为真)

do while 与 while 之间有什么区别吗？

- while 循环时首先执行测试，然后执行包括的语句，仅当测试结果为真。
- do while 循环先执行包括的语句，然后才执行条件测试，这意味着语句至少要执行一次，直至测试的结果为假。

使用 do while 循环的一个好情形是菜单处理。前面的菜单例子有一个缺陷，即当它仅执行一次后，就会非正式地使用户退出程序。下面是用 do while 语句改写后的程序。

领会这段程序，然后运行 INTRO16.C：

```

/* INTRO16.C--Example from Chapter 4 of Getting Started */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
int main()
{
    char cmd;
    do {
        printf("chart desired: Pie Bar Scatter Line Three-D Exit");
        printf("\nPress first letter of the chart you want:");
        cmd=toupper(getch());
        printf("\n");
    }
}

```

```

switch(cmd)
{
    case 'P': printf("Doing pie chart\n"); break;
    case 'B': printf("Doing bar chart\n"); break;
    case 'S': printf("Doing scatter chart\n"); break;
    case 'L': printf("Doing line chart\n"); break;
    case 'T': printf("Doing 3-D chart\n"); break;
    case 'E': break;
    default: printf("Invalid choice. Try again!\n");
}
} while (cmd != 'E');
return 0;
}

```

这段程序改变了什么？程序的活动部分——包括显示菜单和读取字符的语句，也如同 switch 语句一样，被包括在 do while 循环中，附加的一项情况 E，允许用户退出程序，这个 CASE 简单由一个 break 语句与之相连。如果键入任何其它字符(包括无效字符)，while 条件将导致菜单重显示，但如果键入 E(e)，条件 Cmd != 'E' 为假，控制将会转到 do while 循环末尾并跳出循环，然后中止。

4.9.3 for 循环

for 循环将经过一系列的值，并在每一个值时执行一次指定的操作。这个语句的形式是：

```

for(初始值; 条件; 步长变化)
{
    语句或语句组;
}

```

下面的程序，应用 for 循环显示 PC 字符集的每个可见字符(装入运行 INTRO17.C)：

```

/* INTRO17.C Example from chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    int ascii_val;
    for (ascii_val=32; ascii_val<256; ascii_val++)
    {
        printf("\t%c", ascii_val);
        if (ascii_val % 9 == 0)
            printf("\n");
    }
    return 0;
}

```

变量 `ascii_val` 是 `for` 循环的计数变量, 其初始值为 32; `for` 循环限制的条件是 `ascii_val < 256`, 改变计数变量的步长变化是 `ascii_val++`。(换句话说, 循环的每次增量为 1)。

用大括号 `{}` 封闭起来的 `printf` 和 `if` 语句是循环体, `printf` 语句显示与 `ascii_val` 当前值对应的字符, 并用制表符 `(\t)` 来间隔。`if` 用来在变量 `ascii_val` 可被 9 整除时进行换行处理, 也即确保每一行有 9 个字符。

`for` 循环的题头有许多变化。循环体可以仅有一个语句, 此时大括弧是可选的, 但为了清楚还是选择为好。`for` 循环也可以没有循环体, 所有的工作都在控制语句的变化部分完成, 例如下面的程序从 1 到 10 进行统计(装入运行程序 `INTRO18.C`):

```
/* INTRO18.C—Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    int number, total;
    for (number = 1, total = 0; number < 11; total += number, number++);
    printf("Total of numbers from 1 to 10 is %d\n", total);
    return 0;
}
```

这个例子表明了循环语句的说明部分的赋初始值部分和改变步长部分是可以有多个用逗号隔开的表达式的。循环初始化两个变量 `number` 和 `total`, 每次运行时, 把 `number` 加到 `total` 中, 然后再增加 `number`。

跟在右圆括号后面的分号表示循环体为空, 如果遗漏了它, 循环体将把 `printf` 当作循环体来反复执行。`for` 循环可以用 `while` 循环代替, 前面的例子可翻成如下:

```
/* INTRO19.C—Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    int number = 1, total = 0;
    while number < 11 {
        total += number;
        number++;
    }
    printf("Total of numbers from 1 to 10 is %d\n", total);
    return 0;
}
```

理解这段程序, 装入运行 `INTRO19.C`。

`while` 循环在其循环开始前执行初始化, 而在循环体中改变记数变量; 而 `for` 循环在循环说明中执行这两部分工作。`for` 循环更简洁, 但如果在一对圆括弧间塞入太多表达式, 则阅读是很困难的。

4.9.4 break 和 Continue

有时在又一次测试条件之前从循环的语句中出来是很必要的。break 语句有两种不同用法：在 switch 语句中某个 case 的语句执行完后跳出 switch 语句；立即从 while、do while 和 for 语句中退出，而不再执行循环中的其它语句。例如假设当警告灯亮时，不再发出空列车；程序为：

```
/* INTRO20.C--Example from Chapter 4 of Getting Started */
```

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

```
#define WARNING -1
```

```
int get_status(void)
```

```
{
```

```
    return WARNING;
```

```
}
```

```
int main()
```

```
{
```

```
    int count = 10;
```

```
    while (count-- > 1)
```

```
    {
```

```
        if(get_status() == WARNING)
```

```
            break;
```

```
        printf("%d\n", count);
```

```
    }
```

```
    if(count == 0)
```

```
        printf("Shuttle launched\n");
```

```
    else
```

```
    {
```

```
        printf("Warning received\n");
```

```
        printf("Count down held at 1 - %d", count);
```

```
    }
```

```
    return 0;
```

```
}
```

领会这段程序，装入并运行 INTRO20.C。

用 while 循环进行递减计数，每次检查函数 get_status，看它返回值是否为-1(这里定义为警告 WARNING)，这个函数可假定实时地读警告系统。如果在递减计数过程中，get_status 返回-1，则 break 将停止递减计数。紧跟在 while 后面的 if 语句检测 count 是否为 0，如果是，则没有任何警告发生；如果 count>0，则递减计数一定被中止了，因此，空列车不再发出。

continue 语句如同 break 一样将导致循环中的剩余语句被空过，但是 break 跳出循环，而 continue 只简单地跳至循环的条件测试处。INTRO21.C 将显示到 10 的偶数。

领会此程序，装入并运行 INTRO21.C:

```

/* INTRO21.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    int num = 0;
    while (num++ <= 10)
    {
        if(num % 2 != 0)
            continue;
        printf("%d\n", num);
    }
    return 0;
}

```

如果是奇数，则 `continue` 语句将导致 `printf` 语句被跳过。

`continue` 并不常用。你必须考虑使用 `continue` 和 `break` 的适宜时机问题，把 `continue` 用于前面空列车程序并不是好方法，因为不仅递减计数不能显示(循环中的 `printf` 语句将被空过)，而且 `count-- > 1` 的减量所致，空列车即使是在收到警告后仍然发出。

4.9.5 goto 语句

老资格的 BASIC 和 FORTRAN 程序员对 `goto` 语句都是很熟的，C 也有这个语句，其形式是：

`goto 标号(lable)`

这里标号(lable)是一个标识符，它将与一条特定语句连在一起，当执行 `goto` 语句时，控制将转到被标号的语句处。在早期 BASIC 中，`goto` 语句是必须的，因为它没有其它办法来构造一个循环，由于 C 有三种循环，并且在必要时还使用 `break` 和 `continue` 来空过循环的某些部分，因而很少再需要 `goto` 语句。现代的程序员尽量避免使用 `goto` 语句，因为它将使得程序难于阅读和修改，也容易忘记为什么要转到特定语句处。除此之外，没有其它语言还提供 `wherfrom` 语句。

4.9.6 循环嵌套

循环体中的某条语句又是一个循环，这称作为嵌套。在下面例子中，`while` 循环允许键入字符串，而 `for` 循环在字符串的下面打印连字符。

领会这段程序，装入运行 INTRO22.C:

```

/* INTRO22.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
#include <string.h>
#include <conio.h>
int main()
{
    int post;

```

```

char text[40];
printf("Type 'end' to quit\n");
while(strcmp (gets(text), "end") != 0) {
    for(pos = 1; pos <= strlen(text); pos++)
        putchar('.');
    printf("\n");
}
return 0;
}

```

这是一段具有熟练 C 程序员有效且简洁风格的例子。为了表明 while 循环是如何控制的，从里向外读。gets(text) 获取输入的字符串，并将之存入字符数组 text 中，这个函数也返回一个串给调用语句，以便这个串和串 "end" 通过函数 strcmp 进行比较。函数 strcmp 比较两个字符串。如果比较结果是 0，则串相同，用户键入了 end，然后循环结束，其它情况，while 循环将被执行。循环体的第一条语句是一个 for 循环，它将打印出与字符串长度相等个数的连字符，而函数 strlen 将算出字符串的长度。while 循环体的最后一条语句 printf 将使光标位置进入下一行。

4.9.7 选择合适的循环语句

你已经看到有三种不同形式的循环语句(while, do while 和 for)，用哪种形式的语句更清楚地表达程序的思想，请记住下面几条：

- 如果在条件为假时，不希望循环体被执行，请用 while 循环。
- 如果总是保证循环体至少执行一次，则用 do while 循环。
- 如果循环体执行的次数取决于某个变量或常量的值，则最好使用 for 循环。
- 如果凡是外部条件为真，循环体将被执行(如文件中还有数据)，则用 while 循环。

4.10 使用函数和宏的程序设计

已经知道了如何控制程序的执行，现在可以用 C 做许多工作。我们鼓励你去修改和推敲那些例子程序。我们包括如此多的短小简单的程序并不需要许多的结构。因为你的程序可能变得更大和更复杂，所以需要将它们拆成较小更易管理的逻辑片段和函数。

4.10.1 定义自己的函数

你所看到的程序都只完成部分工作。当使用 gets, puts 或 strcmp 时，不必考虑这些函数是如何工作的。在 Turbo C++ 库中已经定义和编译了这些和另外的 400 多个函数。为了使用它们，只需在程序中包括合适的头文件，查一下联机 Help 或者《库函数参考》的第一章“运行库”，就能使你明白如何调用这些函数，以及它们返回的值是什么。

但是，有时需要编写自己的函数，为此，需要把程序编码拆成零散的章节(函数)，每一部分执行一个单一的可为程序理解的工作。一旦说明和定义了自己的函数，你的程序就能象调用 Turbo C++ 库函数一样调用这些函数。

4.10.1.1 函数原型

函数原型是 ANSI 的 C 语言新标准的主要特征。一个函数原型是用如下形式说明的语

句:

<返回值类型> 函数名称 (<参数类型> 参数名...);

这里是一些例子:

```
int main(void)
float tax(float purchase);
char get_employee_type(int employee_num);
char get_choice(void)
int getch(void)
void show_menu(void)
```

注意, main 作为一个自己的函数, 也必须遵守所有其它函数给出原型的忠告。

通过查看这些原型, 你就可以准确地给出函数所期望的信息类型以及其返回值的类型。

让我们看一下 tax。当调用它时, 需要给它一个浮点数, 其结果也是一个浮点数; 原型也告诉了 Turbo C++ 同样的信息。

一些其它的原型例子使用了关键字 void。void 意味着空或没有。当 void 出现在括号内的参数表位置时, 表明此函数没有参数, 因而, 如果试用语句 show_menu(10), 你将被告知: show_menu 不能传递任何参数。当 void 出现在函数返回类型说明处时, 它又意味着不返回值, 所以也就不可能调用这个函数把返回值赋给一个变量。

早期的 kernighan 和 ritchie 的风格, 仅当函数的返回值不是 int 时才给出类型说明, 类似地, 参数也是在函数体内说明, 而不是在参数表中给出, 用这种老式风格, 函数 get_employee_type 将说明成: char get_employee_type; 并且函数体的定义将如下:

```
char get_employee_type(employee_num)
int employee_num;
{
    /* 函数体 */
}
```

老式风格的程序仍然可在 Turbo C++ 下正确编译, 尽管这样, 由于有关函数参数的信息太少, 所以在函数调用时, 包括参数类型失配的误差将不会自动捕获, 这就是老式风格被 ANSI 舍弃的原因。

4.10.1.2 函数定义

函数定义包括为函数编制实际的可执行代码。函数头部的提出形式如同函数的说明一样, 只是在其末尾没有分号, 紧跟在后面的是由一对大括弧封闭的局部变量说明和可执行代码。例如:

```
float tax(float purchase)
{
    float tax_rate = 0.065;
    return(purchase*tax_rate);
}
```

4.10.1.3 函数内的处理过程

函数把其参数看成已说明的指定类型变量, 因而 tax 函数必须对两个变量存取: 一个

是 float 型的参数 purchase, 另一个是它自己定义的 float 型变量 tax_rate, 如果使用函数调用 tax(amount), 则 tax 将通过其参数 purchase 获得 amount 的一个拷贝值, 函数把这个值记在 purchase 名下, 并能够改变 purchase 的值, 但这并不能改变源变量 amount 的值。在后面将看到如何使用指针来改变用于调用函数的变量值。

4.10.1.4 函数返回值

一个函数并不是必须要返回一个值, 在这种情形, 可以说明其返回类型为 void。为了返回一个值给调用者, 函数使用了 return 语句。(正如函数 tax, 其返回值是 purchase*tax_rate)。下面的图概括了如何传入 tax, 并且如何从中返回。

The value of amount is assigned to corresponding parameter purchase

```
/*function declaration*/
float tax(float purchase)
...
/*calling statement*/
tax_amt = tax(amount);
...
```

The value is now available within function tax under the name purchase

```
return(purchase * tax_rate);
```

The return statement sends the calculated value back to the calling statement, where it replaces the function call tax(amount)

```
tax_amt = tax(amount);
```

In calling statement, returned value is assigned to the variable tax_amt

```
tax_amt = (value returned)
```

图 4.3 信息流入/出 tax 示意图

4.10.1.5 应用返回值

函数的返回值如同表达式的值一样处理, 它可以同其它变量和运算符一起联用, 也可以作为 if 语句或循环语句的条件部分等, 下面是已经见过的使用函数返回值的几个例子:

```
tax_amt = tax(purchase);
if(get_status() == WARNING)
while(strcmp(gets(text),"end") != 0)
```

在第一个语句中, tax 的返回值赋给了变量 tax_amt; 第二个语句, 调用 get_status 返回的值与定义值 WARNING 进行比较, 其结果决定了 if 语句的真值; 在最后一个语句, gets 的返回值经过 strcmp 函数(与串"end"一起), 然后把调用 strcmp 返回的结果与 0 进行比较, 其结果变成 while 语句测试条件的值。

4.10.2 多函数程序

下面程序将画出太阳系的部分示意图, 它表明几个用户自定义函数的使用与 Turbo C++图形库的某些特点一样(要了解如何使用 Turbo C++图形库, 参看《程序员指南》的第五章“视频函数”)。

确保设置的路径，调用 `initgraph` 的目录是 `BGI` 文件存放的子目录。

当写图形时，程序需要 `EGA` 或 `VGA` 图形卡，但它限制了自己的适应能力，如果要改变用到的颜色常数，可以运行 `CGA` 的低级形式，输出将在下一张图中表示出来。

程序表列了函数原型。全程说明、`main` 定义，以及其它各函数的定义。下面这个程序包括在软盘的 `PLANETS.C` 文件中。

通读这个程序的注解部分之后，可以随意修改它，研究图形库，试验不同的颜色和填充模式，经历距离和半径的不同变化值。

```
/* PLANETS.C--Example from chapters 4 and 7 of Getting Started */
#include <graphics.h>          /* For graphics library functions */
#include <stdlib.h>             /* For exit() */
#include <stdio.h>
#include <conio.h>

int set_graph(void);           /* Initialize graphics */
void calc_coords(void);        /* Scale distances onscreen */
void draw_planets(void);       /* Draw and fill planet circles */
/* Draw one planet circle */
void draw_planet(float x_pos, float radius,
                  int color, int fill_style);
void get_key(void);            /* Display text on graphics screen, */
                               /* wait for key */

/* Global variables -- set by calc_coords() */
int max_x, max_y;              /* Maximum x- and y-coordinates */
int y_org;                     /* Y-coordinate for all drawings */
int au1;                        /* One astronomical unit in pixels
                               (inner planets) */
int au2;                        /* One astronomical unit in pixels
                               (outer planets) */
int erad;                       /* One earth radius in pixels */

int main()
{
    /* Exit if not EGA or VGA */
    /* Find out if they have what it takes */
    if(set_graph() != 1) {
        printf("This program requires EGA or VGA graphics\n");
        exit(0);
    }
    calc_coords();              /* Scale to graphics resolution in use */
    draw_planets();             /* Sun through Uranus (no room for others) */
    get_key();                  /* Display message and wait for key press */
}
```

```

closegraph();      /* Close graphics system */
return 0;
}

int set_graph(void)
{
    int graphdriver = DETECT, graphmode, error_code;
    /* Initialize graphics system; must be EGA or VGA */
    initgraph(&graphdriver, &graphmode, "..\\bgi");
    error_code = graphresult();
    if(error_code != grOk)
        return(-1);          /* No graphics hardware found */
    if((graphdriver != EGA) && (graphdriver != VGA))
    {
        closegraph();
        return 0;
    }
    return(1);              /* Graphics OK, so return 'true' */
}

void calc_coords(void)
{
    /* Set global variables for drawing */
    max_x = getmaxx();      /* Returns maximum x-coordinate */
    max_y = getmaxy();      /* Returns maximum y-coordinate */
    Y_org = max_y / 2;     /* Set Y coord for all objects */
    erad = max_x / 200;     /* One earth radius in pixels */
    au1 = erad * 20;        /* Scale for inner planets */
    au2 = erad * 10;        /* Scale for outer planets */
}

void draw_planets()
{
    /* Each call specifies x-coordinate in au, radius, and color */
    /* arc of Sun */
    draw_planet(-90, 100, EGA_YELLOW, EMPTY_FILL);
    /* Mercury */
    draw_planet(0.4 * au1, 0.4 * erad, EGA_BROWN, LTBKSLASH_FILL);
    /* Venus */
    draw_planet(0.7 * au1, 1.0 * erad, EGA_WHITE, SOLID_FILL);

```

```

/* Earth */
draw_planet(1.0 * au1, 1.0 * erad, EGA_LIGHTBLUE, SOLID_FILL);
/* Mars */
draw_planet(1.5 * au1, 0.4 * erad, EGA_LIGHTRED, SOLID_DOT_FILL);
/* Jupiter */
draw_planet(5.2 * au2, 11.2 * erad, EGA_WHITE, LINE_FILL);
/* Saturn */
draw_planet(9.5 * au2, 9.4 * erad, EGA_LIGHTGREEN, LINE_FILL);
/* Uranus */
draw_planet(19.2 * au2, 4.2 * erad, EGA_GREEN, LINE_FILL);
}

```

```

void draw_planet(float x_pos, float radius, int color, int
fill_style)
{
    setcolor(color);          /* This becomes drawing color */
    circle(x_pos, y_org, radius); /* Draw the circle */
    setfillstyle(fill_style, color); /* Set pattern to fill interior
*/
    floodfill(x_pos, y_org, color); /* Fill the circle */
}

```

```

void get_key(void)
{
    outtextxy(50, max_y - 20, "Press any key to exit");
    getch();
}

```

4.10.2.1 函数原型和全局说明

这个程序调用五个程序员自定义的函数。它们的说明正确出现在#include语句之后。这些说明可以出现在任何位置，但是，直到碰到原型时，才能进行类型检查。把它们放在开始部分是最容易使之合适的。void型和非int型的函数原型必须出现在第一个调用它的语句之前。

全局变量保有绘制行星所需要的信息，它们的值由函数calc_coords计算，这些值又由函数draw_planet使用。使这些变量是全局性的(而不是在一个函数中说明)可以使它们为需要使用这些全局变量的函数所存取。稍后，可以看到在两个函数间影响变量值的另一方法。

4.10.2.2 设置图形显示

main的第一条调用是set_graph，它将包容涉及Turbo C++图形库的许多操作。set_graph使用库函数initgraph的DETECT方式自动检测当前使用的图形卡的种类，注意有多个return。第一个If判定初始化图形系统时是否有错(返回码不等于grOk)，则函数

退出, 并返回一个错误码。第二个测试返回一个错误码, 如果既没有 EGA, 也没有 VGA 激活。第二个测试仅当第一个成功后才进行, 使用多个 return 语句避免了 else 语句。

标识符 DETECT 和 grOK 的出现并不需要说明, 事实上, 在 graphics.h 中已把它们定义为常量。除函数原型外, 头文件总是为用户程序提供一些实用的定义。我们建议你浏览一下 graphics.h 和其它头文件, 以便使你熟悉其中的内容。将在后面遇到的颜色和填充模式也在 graphics.h 中定义。每一个标识符都与一个整数值相关联, 但是符号名的使用使得查看当前发生的事更为容易(BLANK 当然比 0 更有意义)。

main 检查 set_graph 返回的值, 如果任何大于 1 的值被传回来, 则程序显示一个信息, 然后退出。

4.10.2.3 计算图形坐标

如果检测通过, 下面就调用 calc_coords。许多初始级程序员在考虑图形屏幕的 x-y 坐标量度时, 通常根据所提供图形适配器如 640×350 的 EGA 来决定, 这种尝试很难把精确的量度编入程序中, 而且当程序在具有不同图形分辨率和可选颜色的机器上运行时, 将会导致麻烦, 正如《程序员指南》的第五章“视频函数”中讨论的一样, 图形库帮助编写可以在较宽范围图形适配器上运行的程序。

为了实现这种可能, 库函数 getmaxx 和库函数 getmaxy 将返回 x 和 y 坐标在当前图形方式下的最大值。(这些依次由 initgraph 设定)。剩下的语句:

- 取最大 y 坐标的一半作为行星的中心。
- 设定地球的半径(衡量其它行星的单位标准)为屏幕宽度(最大 x 值)的 1/200。
- 在 X 轴上设置两个安置圆心的距离度量。因为太阳系中一旦越过火星(Mars)后, 距离就迅速增加。内层和外层的行星将采用不同的大小尺度, 因此获得的结果, 图形在距离上将是不精确的, 但这将精确描述行星的相对大小。(由于屏幕的限制, 不能两者都精确)由于同样的原因, 海王星(Neptune)和冥王星(pluto)忽略了。

4.10.2.4 画出行星

函数 draw_planets 把对实际工作的函数 draw_planet 的一系列调用语句集中在一起, draw_planet 传递四个参数; x_pos, radius, color 和 fill_style。

- x_pos 是行星圆中心的 X 坐标, 它是通过距离单位乘上从太阳到行星轨道的天文单位距离求出的。(一个天文单位等于太阳到地球的距离, 近似为 9300 万英里)。
- radius 是行星圆的半径, 它是通过实际行星半径除上地球的半径(大约 4000 英里)再乘上每单位地球半径的点数 erad 获得的。
- color 是一个来自于 graphics.h 的常量, 它将从 EGA 调色板(对 VGA 同样工作)给出用于画圆(并填充)的颜色。
- fill_style 是来自于 graphics.h 的常量, 给出用于填充圆的式样。

draw_planet 传递这些参数, 并调用 Turbo C++ 图形库例行公事, 画出圆并填充它。注意 y 坐标由 circle 和 floodfill 取用, 不必作为参数提供, 自从它确定后, 由 calc_coords 提前计算的 y_org 量将被用到。

最后, get_key 调用库函数 outtext 显示一个信息, 然后调用 getch, 等待键入一个字符来退出程序。

● 1英里=1609.344米。

4.10.3 头文件、函数和库

在一个小程序中，可以说明和定义许多函数在一个文件中，通过把其它事情联系在一起的 `main` 函数将这些函数聚在一起。下图就是这样一种结构。

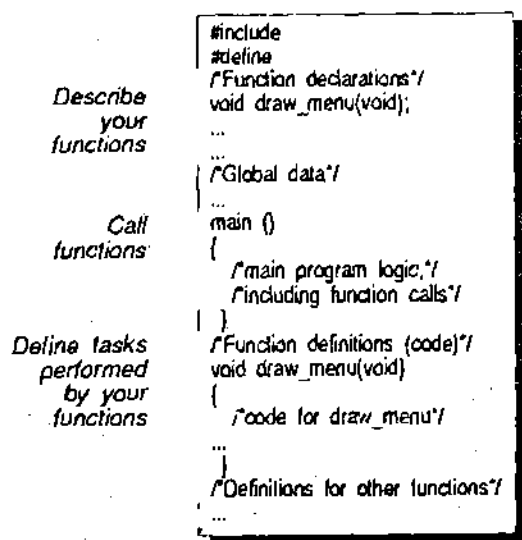


图 4.4 简单程序结构

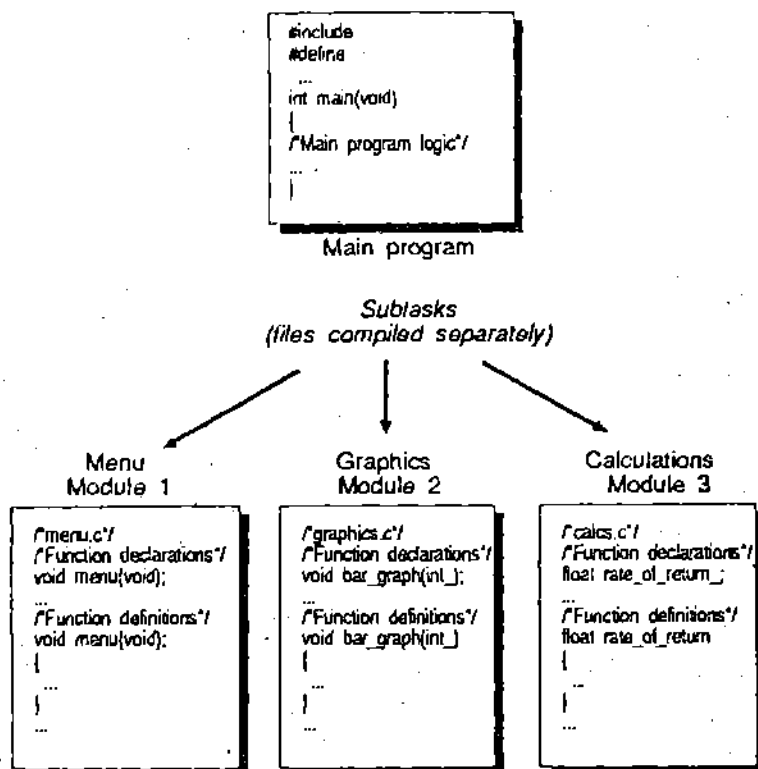


图 4.5 从几个文件产生一段程序

头文件总是只用包含函数说明，所以它们可以包括在不同源文件或模块文件中。

由于程序变得很大，需要把相关的函数定义组织在一个独立文件中。例如处理用户接口的函数可以放入一个文件，数据处理函数放入另一个文件，图形显示的函数则放入第三个。Turbo C++可以执行程序，这种结构在图 4.5 中体现。

一旦你的程序各部分已经稳定，就可以把函数分组编译成库。在每个库中对函数说明将放入一个头文件中，就象在调用 Turbo C++ 自己的库一样使用。主程序包括标头文件，如此插入函数说明在程序文本中。在编译之后，连接程序将把库连接成程序目标码，这个处理过程在下图展示。

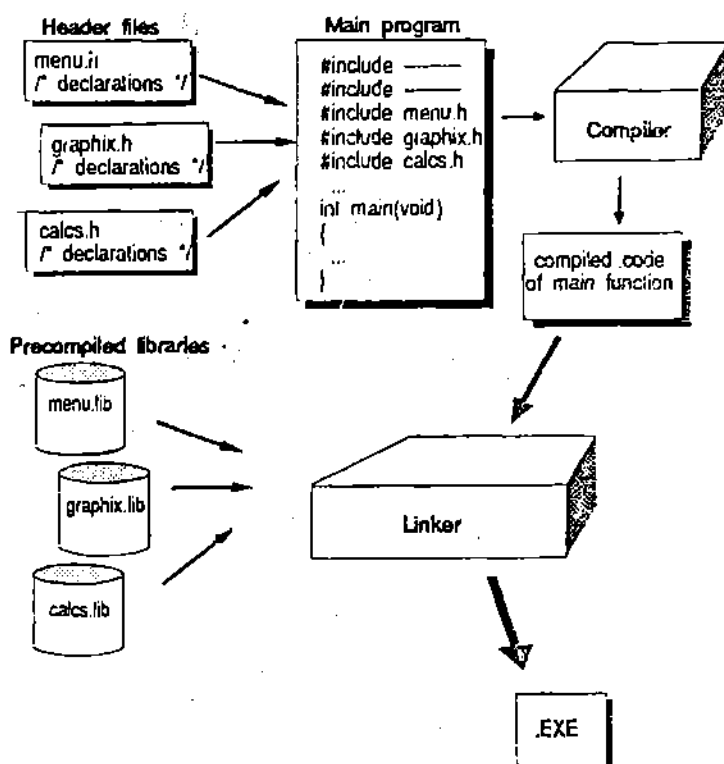


图 4.6 程序使用常规库示意图

4.10.4 变量的范围和持续时间

随着程序变得更为复杂，程序其它部分对变量的存取问题就越易发生。每个变量都有两个特征：范围和持续时间。范围(scope)(有时称为作用域)定义了程序的哪一部分可以存取这个变量；持续时间(duration)说明了变量可存取的保留时间。

4.10.4.1 范围(scope)

范围取决于在何处定义的变量。在函数定义内部说明的变量隐含是局部的，它仅能为同一段函数的代码所存取，例如在下面这段程序中：

```
/* INTRO23.C—Example from Chapter 4 of Getting Started */
#include <stdio.h>
void showval(void);
int main()
```

```

{
    int mainvar = 100;
    showval();
    printf("%d\n", funcvar);
    return 0;
}

void showval (void)
{
    int funcvar = 10;
    printf("%d\n", funcvar);
    printf("%d\n", mainvar);
}

```

函数 `showval` 首先使用 `printf` 语句显示 `funcvar` 的值，这是很好的，因为 `funcvar` 说明定义都在函数 `showval` 之内；而下面一条语句试图显示变量 `mainvar` 的值，将导致编译器抱怨（通过错误信息）变量 `mainvar` 没有定义。这是因为 `mainvar` 是定义在名字为 `main` 的另一个函数中，它不能在 `showval` 内进行存取。

甚至在确定了这些之后，当从 `showval` 返回后，函数 `main` 试图显示 `funcvar` 的值，也将产生一个错误，因为变量 `funcvar` 是在 `showval` 函数内部定义的。

为确保一个变量对当前源文件中的任何函数都是可见的，请在函数定义体之外说明这个变量。它将对源文件中在其定义处之后的函数可见，所以通常定义全局变量的地方是在 `main` 函数开始之前，前面的例子开始部分可以象这样：

```

void showval(void);
int mainvar, funcvar;

```

如果把 `int` 说明从

```

int mainvar = 100 和
int funcvar = 10

```

之前移去，这样就没有抱怨了。请记住，从任何位置都可以存取的变量，在任何位置都是可以改变的，这样就可以定位难以跟踪的故障。改变变量值而产生的这种方法有时称为副作用。

由于可以省略，全局变量对任何文件都是可存取的（尽管在其它文件中没有 `extern` 指示，全局变量的范围遍及整个文件）。如果一个程序使用不止一个源文件，为了使变量在不同源文件间可见，在当前文件中说明它时添上 `extern` 关键字（“外部的”）。这样，如果文件 `main.c` 定义了 `int xscale`，就可以在其它文件中引用用如下方式说明的变量，

```
extern int xscale;
```

当变量原始定义时，将赋给这变量自己的内存地址。`extern` 说明只是提醒编译器有一个外部变量将被提及。

4.10.4.2 持续时间

在一个大程序中，为所有的变量永久保留内存是不可能。既然一个特定函数可能仅调用一次，由于隐含的说明，在函数定义内说明的变量都是 `Auto`（自动）变量，仅当函数开始执行时才分配内存，当函数返回调用者时，内存将被释放，并为其它变量所用。

有时, 你可能希望不考虑这种隐含行为, 使一个变量永久存储, 甚至当说明它的函数并不在运行时也存储它。关键字 `static` 将完成这一任务。例如, 下面这个函数就能统计它被调用的次数。

试验本程序, 装入并运算 `INTRO24.C`。

```
/* INTRO24.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
#include <conio.h>
void tally(void);
int main()
{
    while(getch() != 'q')
        tally();
    return 0;
}
void tally(void)
{
    static int called = 0;
    called++;
    printf("Function tally called %d times\n", called);
}
```

`main` 函数中的 `while` 循环每次通过 `_getch` 函数读取一个非 `q` 的字符, 则调用 `tally` 函数。每次调用, `tally` 将增加静态变量 `called` 的值。当说明静态变量时, `C` 初始化其值为 `0`。

有时使用另一种关键字 `register` 来进行说明, 当把它加在变量说明前时, `register` 请求编译器产生代码时使用微处理的快速内部寄存器来保存这个值, 而不是使用慢速的随机存取存储器。`register` 只能用于可以适于一个寄存器的小数据类型, 诸如 `char` 和 `int` 等。

因为现代的编译器, 如 `Turbo C++` 进行了优化处理, 所以可以充分利用机器的资源, 而这些只有有经验的程序员才知道何时利用这些特点。不适宜的使用将降低程序的速度。通常最好的做法是让编译器弄清楚是否让一个变量使用机器的寄存器, 而且, 使用关键字 `register` 并不保证变量将一定存放在寄存器中, 它仅仅是建议编译器试着这么做。

4.10.5 使用常量值

常量(`constant`)是一个确定的值, 它在程序运行期间不会改变。有两种方法定义常量: 关键字 `constant` 和 `#define` 指示。

1. 产生常量的一个方法是在说明中使用关键字 `constant`。例如说明

```
constant float cm_per_inch = 2.54
```

已告诉编译器这个值将不再改变, 如果试图在程序其它部分赋给它一个新值(包括使用 `++` 或 `--` 的增量与减量), 就会收到一个出错信息。使用这个关键字因此会帮助 `Turbo C++` 捕获编程失误。

2. 另一种在程序中包括常量值的方法是使用 `#define` 指示。你已经有过这样的例子, 如:

#define RATE 0.065

这不是一个说明，而是预处理器中的一个指令。预处理器是 Turbo C++ 的一部分，它将在你的程序编译之前准备程序需要的变换。这里，这个变换等价于使用一个字处理器找出 RATE 的所有场合，并且用字符 0.065 取代它们。

有许多时候，const 和 #define 是很合适。在这两种情况下，只需要简单改变一下定义值，并重新编译就能改变运行值。const 给出的最大有利之处是 Turbo C++ 知道这个常量的数据类型(如 const int...)。另一方面，#define 可以出现在多个模块中而不会出问题，但 const float 则不行。

使用常量的好处是当值发生改变时仅需改变一条语句，不必搜寻每一个该值出现的地方，否则除了麻烦之外，还会潜藏错误。

4.10.6 使用宏来隐含细节

#define 能做简单化的替代。你可以定义一个宏，它传递参数(在这点上类似函数)，并将它们嵌入文本模块中。例如：

装入并运行 INTRO25.C。

程序员在宏定义时常用?:表达式。

```
/* INTRO25.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
#define EVEN(value) ((value) % 2 == 0) ? (value) : ((value) + 1)
int main()
{
    char inbuf[130];
    int num;
    printf("Enter a number:");
    gets(inbuf);
    sscanf(inbuf, "%d", &num);
    printf("\n%d", EVEN(num));
    return 0;
}
```

宏定义中附有圆括弧的 value 处可以是一个表达式

在语句 printf("\n%d", EVEN(num)) 中，定义将导致表达式 EVEN(num) 为 ((num%2==0)?(num):(num+1)) 所代替，换句话说，圆括弧中用什么项，它就取代样板中 value 的出现；最后，结果表达式如果能被 2 整除((num)%2==0)，就变成了对 num 的计算，或者它是奇数，则用 (num)+1 取代 num，使之变成偶数。

宏 EVEN 可以改写为一个函数，如下：

```
int even(int num)
{
    return(num % 2 == 0 ? (num) : (num + 1));
}
```

函数与宏之间的交易是速度对时间，通常函数都较小但比宏慢。

尽管宏和函数采用同样的方法调用，但其工作方式是不同的。对一个宏而言，在编译之前，将用嵌入值的样板去取代源程序中对宏的调用语句；而函数则将它编译成目标码，并且也把函数调用编译成目标码，再通过把合适的值压入堆栈，并弹出给函数的目标码。函数变成目标码只需编译一次，而不管它被调用了多少次。对于宏，文件中的每一次宏调用都要插入一段新的源代码。

如果使用宏，必须确保被替代的值具有合适的类型。

用一个易于记认的名字取代一个复杂表达式，宏使得源程序更加可读。运行库中大量的“函数”是真正的宏；例如，字符分类规则 `isalpha` 将查看一个字符是否是字母之一或者是一个数字，标点符号等等。通过查阅头文件 `ctype.h` 就可以研究这个宏，以及其它的宏定义。

4.11 建立数据结构

数据势必成串进入而不是单个进入。例如，你可能需要保留当年内一个雇员每周工作小时数的曲线，这样，就有了一个具有同样类型的相关数据项集合(总小时数，可能是允许有小数位的 `float`)。C 允许你说明一个数组 `array` 来存储这些同类数据。但是，也许你的业务也必须保存每个雇员的大量信息，诸如姓名、工作年限、薪金、部门等，这些项当然是相关的(都指向同一个雇员)，但其类型却是不同的。姓名是字符串(字符数组)，工作年限可以是 `int` 或 `float`，这要取决于是否允许有小数，薪金可能是 `double`(允许有较好酬金的雇员)，而部门可能是数字代码或字符串。

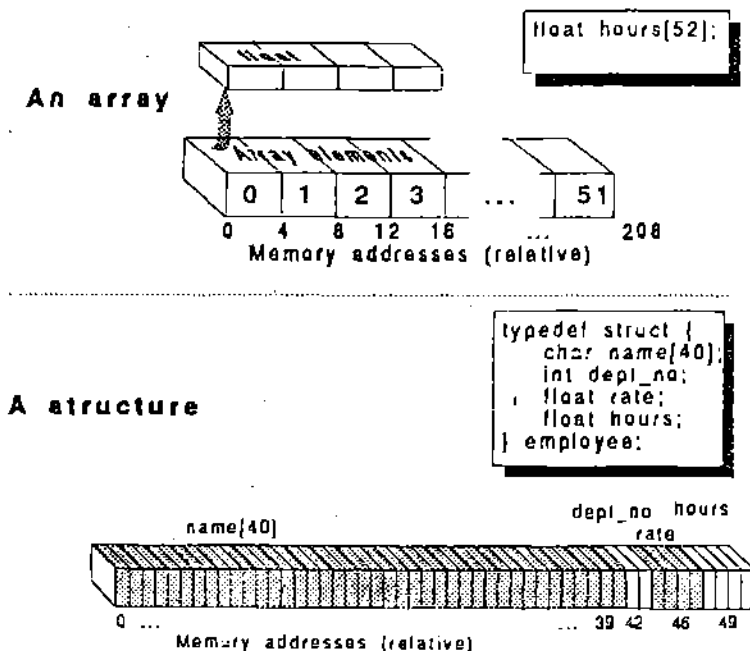


图 4.7 两种处理数据集的方法

注意，结构中的不同成员并不总是在内存中相邻的。

为了组织你的数据，需要有一种易于找出对之操作的指定项的方法。因为这些数据实际存放在成块的内存地址中(这有些象街道上的门牌号)，可以通过使用地址来指定想要的

数据，在这一节中，将学会如何用指针来存取数据结构。

4.12 说明和初始化数组

数组是用来保存一组相同类型的数据项的内存块，例如，int 数组将把指定的一批整数保存在连续的内存单元中。说明一个数组，需要给出它的数据存储类型、数组名和存储的数据项个数，并且用方括弧将这个数括起来。

<类型> <名字> [大小]

下面这个例子说明了一个数组，来存放一个雇员一年内每周工作的总小时数。

```
float hours[52];
```

这个式子可读成“hours，具有 52 个 float 值的数组”。

数组开始于 0 指针，结束位置不超过其给定的大小。

数组的一个给定项——称为元素——可以通过给出数组名及其项的位置来指定。第一项存放在由数组名自己指向的地址内，它也可以用位置 0 来确定，这样第一个星期的总工作小时就可以用 hours[0] 来表示，第十周则用 hours[9]，第五十二周用 hours[51]。

下面这段程序初始化数组 hours 为 0，再赋给开始的四个位置予新值，然后打印出。请看一下它们是如何存取的。

领会本程序，装入运行 INTRO26.C。

```
/* INTRO26.C—Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
```

```
    float hours[52];
```

```
    int week;
```

```
    /* Initialize the array */
```

```
    for (week = 0; week < 52; week++)
```

```
        hours[week] = 0;
```

```
    /* Store four values in array */
```

```
    hours[0] = 32.5;
```

```
    hours[1] = 44.0;
```

```
    hours[2] = 40.5;
```

```
    hours[3] = 38.0;
```

```
    /* Retrieve values and show their addresses */
```

```
    printf("Elements\t\tValue\tAddress\n");
```

```
    for(week = 0; week < 4; week++)
```

```
        printf("hours[%d]\t\t%3.1f\t%p\n", week, hours[week],  
               &hours[week]);
```

```
    return 0;
```

```
}
```

其输出类似下面的结果：

Element	Value	Address
---------	-------	---------

```
hours[0]   = 32.5   FFOE
hours[1]   = 44.0   FF12
hours[2]   = 40.5   FF16
hours[3]   = 38.0   FF1A
```

从一项到另一项，地址将会发生变化，同时从一个机器到另一机器也会改变。

请注意，元素都存储在连续的地址内，四个字节一组(总是 float 类型所占的大小)，地址运算符&将取回指定元素的地址。printf 语句的格式符%p(给指针用)将地址显示成十六进制数，而 for 语句用于一个一个处理数组元素是非常方便的。

也可以在说明数组时，明确地初始化数组，为此，把要赋的值用大括号括起来，每个值可用逗号分开，例如：

```
int quarters[4] = {3,10,7,14};
```

就可以保存一场足球比赛中，一个队在每节比赛中的得分。(当然，这样的数据也许来源于键盘录入或从一个文件读入，但总是可以把提供调试的数据明确赋给一个数组。

当赋给数组字符常量时，把字符放在单引号间，例如：

```
char grades[5] = {'A','B','C','D','F'};
```

如果只提供少于数组长度的值，并且数组全局或静态的，则 Turbo C++ 将使剩下的元素填充 0 或空字符(这种情况数组是字符型)，如果提供的值多于数组长度，将出现错误信息，“太多的初始化值(Too many initializers)”。

数组可以比这样的更为复杂(例如可以省略长度说明，或者用串来初始化字符数组)。不过，这些主题已超过了本章的范畴。

4.12.1 多维数组

有时，一系列的数据集是相当有用的，例如，如果想存储 12 个雇员在 52 周内的每周工作时间，则可以说明成：

```
float hours[12][52];
```

读成“一个包含有 12 个数组的数组，每个数组有 52 个类型为 float 的值。”也可以把这种结构看成是具有 12 列 52 行的平面网格。

下面的游戏将产生一个模拟的棒球积分，用名为 scoreboard[2][9]的数组表示两个队九局比赛的得分。

请装入并运行 GAME.C:

```
/* GAME.C--Example from Chapter 4 of Getting Started */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#define DODGERS 0
#define GIANTS 1
void main()
{
    int scoreboard[2][9]; /* An array two rows by nine columns */
    int team, inning;
```

```

int score, total;
tandomize();          /* Initialize random number generator */
/* Generate the scores */
for (team = DODGERS; team <= GIANTS; team++) {
    for (inning = 0; inning < 9; inning++) {
        score = random(3);
        if (score == 2)      /* 1/3 chance to score at least a run */
            score = random(3) + 1;  /* 1 to 3 runs */
        if (score == 3)
            score = random(7) + 1;  /* Simulates chance of a big
                                     inning of 1 to 7 runs */
        scoreboard[team][inning] = score;
    }
}
/* Print the scores */
printf("\nInning\11 2 3 4 5 6 7 8 9 Total\n");
printf("Dodgers\t");
total = 0;
for (inning = 0; inning <= 8; inning++) {
    score = scoreboard[DODGERS][inning];
    total += score;
    printf("%d ", score);
}
printf("%d, total);
printf("\nGiants\t");
total = 0;
for (inning = 0; inning < 9; inning++) {
    score = scoreboard[GIANTS][inning];
    total += score;
    printf("%d ", score);
}
printf(" %d\n", total);
}

```

毫不奇怪，当涉及到二维数组时，通常总是用两重 for 循环来存取数组元素。内层循环使用 `inning` 来作为计数变量，依次经过九局比赛，而外层则在队 0 (Dodgers) 和队 1 (Giants) 之间转换。

在循环体内，随机函数 `Random` (定义在头文件 `stdlib.h` 中) 产生得分。当第一次调用函数 `random` 时用 `random(3)` 的形式，有 1/3 的机会使 `score` 为 2，(`random` 返回一个介于 0 到小于其参数值间的数)，如果 `score` 是 2，则它将再计算一次，使之取 1 至 3 之间的随机数；最后，如果 `score` 已是 3，则产生一个介于 1 至 7 间的随机得分，这里的 `if` 语句序列

试图仿真一下偶然的大比分。

最后的两个 for 循环打印积分, 以及总分, 每一个语句使用队名常量值来作为数组的第一维下标, 并改变 `inning` 的值, 以便打印出一个队的得分。下面是结果:

inning	1	2	3	4	5	6	7	8	9	Total
Dodgers	0	1	0	1	0	1	1	2	0	6
Giants	1	1	0	2	0	0	4	1	0	9

4.12.2 数组和串

串和数组很相似, 事实上, 一个串是一个在末尾附有空字符的 `char` 型数组。下面的程序说明了一个字符数组(串), 允许给它存入一个值, 并且可以从串中取出一个子串(装入并运行程序 `INTRO27.C`)。

记住, 串数组需要一个标志结束的空字符来作为附加元素。

```
/* INTRO27.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>

int main()
{
    char string[80];           /* Has 79 usable elements */
    char number[10];
    int pos, num_chars;
    printf("Enter a string for the character array:");
    gets(string);
    printf("How many characters do you want to extract?");
    gets(number);
    sscanf(number, "%d", &num_chars);
    for (pos = 0; pos < num_chars; pos++)
        printf("%c", string[pos]);
    printf("\n");
    return 0;
}
```

下面是运行的试样:

```
Enter a string for the character array: The quick brown fox
How many characters do you want to extract? 9
The quick
```

使用库中的例程处理串通常更为方便, 因为它们能自动将空字符放在串尾, 如果用数组说明的方式创建一个串, 且又想用这些例程来处理它, 必须在串尾提供一个空字符。(数组必须足够大, 以便能保存所希望的串和空字符, 而这个字符又必须放在串尾)。

4.12.3 定义串变量

可以定义一个类型为 `char` 的变量保存单个字符, 能否定义一个类型为 `string` 的变量保存一个串呢? 不行, 因为 C 并不把串作为独立的数据类型来处理。请记住一个串是一系

列的字符，一种定义串变量的方法是说明一个字符数组，下面是一段如何处理这个问题的程序。

```
#include <stdio.h>
#include <string.h>
int main()
{
    char message[30];
    strcpy(message, "This is the value of msg\n");
    puts(message);
    return 0;
}
```

行 `char message[30]` 说明了一个可保有 30 个字符的数组(一个串)，但它仅可以用 29 个字符，因为它必须在末尾为空字符留一个单元。这个 30 字节的块开始地址存储在 `message` 中，在 `strcpy` 调用期间，文本中：

This is the value of msg\n

首先编译和存储，然后在末尾加入空字符。

当 `strcpy` 执行时，从 `message` 的起始地址开始，一次一个地从此串中拷贝字符到内存中。

Turbo C++ 库中有许多函数对串执行一些有益的操作，这可以在《程序员指南》第二章“运行时间库交叉参考”中读到。下面是用 `strcat` 联接串的例子。

可以装入并运行 `INTRO28.C`：

```
/* INTRO28.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
#include <string.h>
int main()
{
    char name[60];
    strcpy(name, "Bilbo");
    strcat(name, "Baggins");
    puts(name);
    return 0;
}
```

在此，说明了一个足够存下想要装入的任何串的字符数组，函数 `strcpy` 存储了第一个串，函数 `strcat` 又把第二个附加在 `name` 中。(请注意，`name` 的第一个值以空格作为结尾，所以两个名字将被分开)。

4.12.4 重新命名类型

当碰到更为复杂的数据结构时，给它赋予一个有意义的名字是很有益处的；可以用关键字 `typedef` 来作这件事，`typedef` 给标准 C 数据类型某些联合体一个新名字，例如：

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



```

int main()
{
    typedef unsigned char uchar;
    uchar greek_alpha = 224, greek_beta = 225;
    printf("%c %c", greek_alpha, greek_beta);
    return 0;
}

```

这里，typedef 说明语句把新名字 `uchar` 赋给 `unsigned char` (这是一个能贮 256 个字符，包括扩展 PC 字符集在内的字符类型)。下一条语句说明 `greek_alpha` 和 `greek_beta` 是 `uchar` 型变量，并用 `printf` 语句打印其值。

typedef 并不能真正创建一个新的类型，它只是使程序中处理数据的类型易于记忆罢了，正如看到的一样，它只是对复杂数据类型重新命名特别有用，同时也可用于枚举和结构类型。

4.12.5 枚举类型

有时，数据顺应一个逻辑序列，一个紧跟着另一个。例如，一周的每一天。使用一个循环来逐次经过这些值是很方便的。

```

for(day = mon; day <= fri; day++)
    /* add hours worded that day to total for week */

```

因为循环经过每个值，需要给 `mon` 一个整数值，`tues` 是下一个整数值，如此下去，你可以用 `#define` 语句做这些工作。

```

#define mon 0
#define tues 1
#define wed 2
#define thurs 3
#define fri 4

```

不过，枚举类型(enum)提供了更为简结的方法，这个例子如下(装入运行 `INTRO29.C`):

```

/* INTRO29.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    enum workday {mon, tues, wed, thurs, fri};
    int day;
    for (day = mon; day <= fri; day++)
        printf("%d\n", day);
    return 0;
}

```

第一条说明语句自动实现下列赋值，`mon` 等于 0，`tues` 等于 1，`wed` 等于 2 等。这些名字现在就可以用作 `for` 循环的初始值和终值了。

注意，枚举中的数字可以是不连续的，因此可以不考虑隐含序数而赋值如下：

```
enum scores {touchdown = 6, field_goal = 3, safety = 2, point_after = 1};
```

语句

```
printf("%d\n", touchdown + point_after + field_goal)
```

显示结果为 10。

4.12.6 把数据组合成结构

数组和枚举是两种处理相同类型数据集的有力方法,而结构却是把不同类型的数据聚集在一起(或者至少是把具有不同意的值聚集在一起)。例如,一个雇员的所有信息可以以如下的结构存储:

```
struct employee {  
    char last_name[30];  
    char first_name[20];  
    char initial;  
    double employee_no;  
    double SS_no;  
    char dept_code[3];  
    float annual_salary;  
};
```

现在,已经定义了一个新类型 `employee`,并且指明了它的数据项(元素),而这些数据项是每个 `employee` 型变量都具有的。正如所见,可以使用许多数据类型,在这个例子中,用到了字符数组、单个字符,双倍长类型和浮点型。

4.12.7 使用结构的数据项

下面的例子将前面程序 `PLANETS.C` 画一个行星所需的信息聚合在一起。我们将用它来说明结构的某部分(数据项)是如何初始化和存取的。

结构的域是用结构名后跟一个圆点及元素名表示的,这样, `mars.distance` 就是包含有从太阳到火星天文单位距离的一个元素。

下面是这段程序:

```
/* INTRO30.C--Example from Chapter 4 of Getting Started */  
#include <stdio.h>  
#include <string.h>  
typedef struct {  
    char name[10];  
    float distance;  
    float radius;  
} planet;  
  
planet mars;  
int main()  
{
```

```

strcpy(mars.name, "Mars");
mars.distance = 1.5;
mars.radius = 0.4;
printf("Planetary statistics:\n");
printf("Name: %s\n", mars.name);
printf("Distance from Sun in AU: %4.2f\n", mars.distance);
printf("Radius in Earth radii: %4.2f\n", mars.radius);
return 0;
}

```

`typedef` 关键字将名字 `planet` 赋给一个包含有三个元素(或域): 卫星 `name` 是一个字符数组, 距离 `distance` 和半径 `radius` 为浮点型的结构 `struct`。说明语句 `struct planet mars` 创建了一个类型为 `planet` 的变量 `mars`(换句话说, 它给前面的定义结构制造了一个拷贝)。在 `main` 中, 又给这三个元素赋了值。

正如这里所见, 库函数 `strcpy` 巧妙地把一个串值赋给了保存这个值的结构域(字符数组), 而数值按通常进行简单赋值即可。

程序末尾的 `printf` 语句用 `name.member(名字.元素)` 表示法来指示和打印所赋的值。

4.13 建立合适的说明符

说明符是 C 中用来说明函数、变量、指针和数据类型的语句。而且 C 允许建立更为复杂的说明符。这一节将给一些说明符的例子, 以使你获得设计(阅读)它们的一些体验, 而且也提示你应当避开的一些陷阱。

传统 C 编程允许建立完整的合适的说明符, 并且可以随心嵌套定义。但是很不幸, 这样编出的程序将是难读和难写的。

考虑一下下面表格中的说明符, 假定在小存储模型(小代码、小数据)中进行编译。

表 4.9 不使用 `typedef` 的说明

<code>int f1();</code>	返回 <code>int</code> 的函数
<code>int *p1;</code>	指向 <code>int</code> 的指针
<code>int *p2();</code>	返回指向 <code>int</code> 指针的函数
<code>int far *p2;</code>	指向 <code>int</code> 的远指针
<code>int far *f3();</code>	返回指向 <code>int</code> 远指针的近函数
<code>int * far f4();</code>	返回指向 <code>int</code> 近指针的远函数
<code>int (*fp1)(int)</code>	指向函数的指针, 该函数返回 <code>int</code> 并接受 <code>int</code> 型参数
<code>int (*fp2)(int *ip);</code>	指向函数的指针, 该函数返回 <code>int</code> 并接受指向 <code>int</code> 的指针为参数
<code>int (far *fp3)(int far *ip)</code>	指向函数的指针, 该函数返回 <code>int</code> 并接受指向 <code>int</code> 的远指针为参数
<code>int (far *list[5])(int far *ip);</code>	具有五个元素的指向函数的远指针数组, 该函数返回 <code>int</code> 并接受指向

```
int (far *gopher(int(far *fp[5]))\
    (int far *ip))(int far *ip);
```

int的远指针为参数
有五个元素组成的数组为参数的函数，该数组元素为指向函数的远指针，该指针指向返回int的函数，该函数接受指向int的远指针为参数

这些都是有效的说明，它们变得益加难以理解，不过，审慎地使用 typedef，就会改善这些说明的清晰度。

这里是同样的说明，使用 typedef 语句后重写如下：

表 4.10 使用 typedef 的说明符

int f1();	返回 int 的函数
typedef int *intptr;	
intptr p1;	指向 int 的指针
intptr f2();	返回指向 int 指针的函数
type int far *farptr;	
farptr p2;	指向 int 的远指针
farptr f3();	返回指向 int 的远指针的近函数
intptr far f4();	返回指向 int 的近指针的远函数
typedef int (*fncptr1)(int);	
fncptr1 fp1;	与上表对应
typedef int(*fncptr2)(intptr);	
fncptr2 fp2;	与上表对应
type int (far *ffptr)(farptr);	
ffptr fp3;	与上表对应
typedef ffptr ffplist[5]	
ffplist fp4;	与上表对应

正如所见，用 typedef 说明原 gopher 和前面说明的 gopher 之间，在清晰度和透明性上确实有很大不同，如果能明智地使用 typedef 语句和函数原型，将会发现程序很易于编写、调试和维护。

4.14 指针

每个变量都只有一个内存地址，这个地址指明了变量值占据内存区域的开始位置，而占用的内存总量取决于所涉及的数据类型。在 int 型时，这个区域是 2 个字节，float 占用 4 字节，对一个数组，所占用的区域等于元素个数乘上一个值的说明类型所占空间的大小，对一个结构，所占用区域等于每个结构元素所需区域的总和，再加上某些必要的填充(如果使用 -a 操作)。因为在所有类型的数据中，都是按有序、可预见的方法存储，它使得通过操作包含有相对地址的变量来存取数据成为可能，这样的变量就是指针。

指针为什么会有用？首先，它允许存取和操作结构化数据，而且不必在内存中移动数

据。例如，把一个指针设置为数组连续元素的地址，就可以用指针来对数组进行初始化或者从数组中取回数据，通过增加或减少指针值，就可以指向不同的数据项。

指针也可以用于使一个函数接收和修改变量值，这就能避免说明全局变量。

当程序运行时，存储分配也需要指针，实质上，当申请一块自由内存区时，如经过函数 `malloc` 申请，将会返回一个指向第一个有效地址的指针。

存储分配将在《程序员指南》的第四章“存储模式、浮点数和覆盖”中讲述。

4.14.1 说明和使用指针

指针的说明采用下面形式

`type *name`

`int *intptr; /* Points to an integer */`

`float *fltptr; /* Points to a floating_point value */`

`char *string; /* Points to a character value */`

大多数的程序员使用包含有单词 `pointer` 的缩写在内的名字，例如 `intptr`。

可以给内存中的任意对象说明一个指针，包括数组、结构、函数以及指针。

为了存取由指针指向的值，指针名之前加一个星号(*)作为前缀，例如 `*intptr` 将产生存放在由指针 `intptr` 保存其地址的单元内值，因为这个值是间接取到的(而不同于惯常的变量，其值就存放在变量自己的地址单元中)，这个过程就称为间接(`indirection` or `dereferencing`)引用。

下面说明了一个指针，并用它取回变量值。

装入并运行 `INTRO31.C`:

```
/* INTRO31.C--Example from Chapter 4 of Getting Started */
```

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

```
int main()
```

```
{
```

```
    int intvar = 10;
```

```
    int *intptr;
```

```
    intptr = &intvar;
```

```
    printf("Location of intvar: %p\n", &intvar);
```

```
    printf("Contents of intvar: %p\n", intvar);
```

```
    printf("Location of intptr: %p\n", &intptr);
```

```
    printf("Contents of intptr: %p\n", intptr);
```

```
    printf("The value that intptr points to: %d\n", *intptr);
```

```
    return 0;
```

```
}
```

这儿是输出的结果:

Location of intvar:FFDC

Content of intvar:10

Location of intptr:FFDE

Content of intptr:FFDC

The value that `intptr` points to:10

地址值是可变的。

首先,说明了 `int` 型变量 `intvar` 并赋初值为 10, 下一条语句说明一个 `int*`(整数指针)型的变量 `intptr`, (这条说明语句读作“`intptr` 一个指向整型数的指针”)。再下一条语句将变量 `intvar` 的地址赋给了 `intptr`, (请注意地址操作符 `&`)。必须把试图让指针指向的对象地址赋给该指针, 如果忽略了这一点, 指针中将包含一个废弃的地址, 如果试图通过这个指针来存放某些东西, 必须承担这样的风险, 即破坏部分程序和数据, 甚至挂起整个系统。

正如从 `printf` 语句中所见的一样, `intvar` 和 `intptr` 存放着不同地址, 因为它们是不同的变量。(由于它们的说明语句连在一起, 它们的地址碰巧靠在一起, 但这对解决指针是如何工作的问题并不相干)。`intptr` 包含 `intvar` 的地址在先前已给它赋值)。最后一个 `printf` 语句打印出由 `intptr` 指向的值, 换句话说, 地址单元的内容存放在此。因为那个地址是 `intvar` 的地址, 则指向的值就是 `intvar` 的内容。下面的图将帮助你想象到这一点。

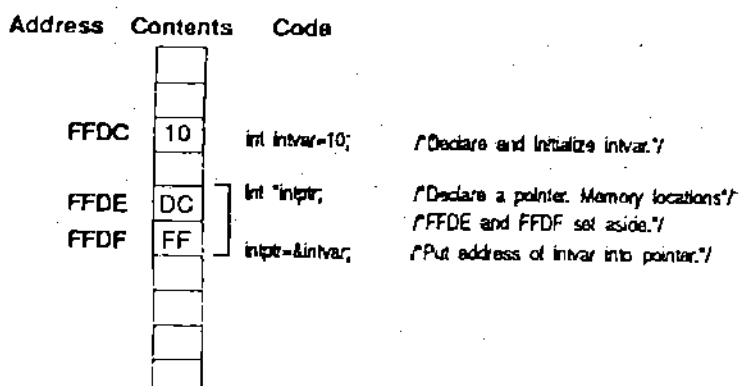


图 4.8 指针及所指内容

4.14.2 指针和串

已经见过, 通过使用下标可以从串中存取单个字符, 如说明了一个串 `char name[20]`, 并赋值“Madama”, 则 `name[2]`的值是字符 d(请记住, 开始数是 0, `name[0]`是 M)。另外一种处理串的方法是说明一个字符指针, 并用它来操作整个串。

这个例子是建立在用字符数组处理串的程序基础上, 只用指针改写了一下, 请试验这段程序, 装入和运行 `INTRO32.C`:

```
/* INTRO32.C-Example from Chapter 4 of Getting Started */
#include <stdio.h>
int main()
{
    char name[40];
    char number[10];
    char *str_ptr = name;
    int pos, num_chars;
    printf("Enter a string for the character array: ;
```

```

gets(name);
printf("How many characters do you want to extract? ");
gets(number);
sscanf(number, "%d", &num_chars);
for(pos = 0; pos < num_chars; pos++)
    printf("%c", *str_ptr++);
printf("\n");
return 0;
}

```

注意, `str_ptr` 被说明成一个字符指针, 并赋给了字符数组 `name` 的地址, 这可以用两条独立语句重写成:

```

char *str_ptr;
str_ptr = name;

```

但现在已知道, C 程序员在可以用一条语句的时候, 极少使用两条语句; 也可以注意到, 赋地址时使用 `name` 而不是 `&name`。提及数组(或结构)的名字, 将给你存放数组元素值的区域的第一个地址, 这等效于 `&name[0]`, (数组 `name` 的第一个元素地址); 有时, 你也会看到后面的这种表示法。

程序剩余部分的工作同前面的程序一样, 直到 `for` 循环取出期望的子串为止。在前面的旧程序中, 通过使用表达式 `name[pos]` 的形式, 每执行循环一次取出一个字符。而在这里使用了指针, 要表示的当前值用 `*str_ptr++` 指向的。(指针每次加 1, 以便使它指向下一个值)。

4.14.3 指针运算

非字符数组使用指针进行处理与串一样, 可以增加指针来指向数组的下面元素; 减少指针值来指向前面的元素, 当然, 必须进行检查确保不会指向数组范围之外的位置, 通常是在循环语句设置合适的终止值来做这件事。

重要的是请记住: 当指针 `ptr` 增值时, 它不一定指向下一个地址, 事实上总是不会的。所指向地址间距等于指针指向的数据类型的大小。一个 `int` 型指针当增值时, 将指向下一个地址单元的头部; `double` 型指针则指向八个字节一组地址单元的头部。C 将自动处理这件事。

4.14.4 指针、结构和列表

你可能还记得用 `structure_name.member.name` 的表示可以获得结构中一个成员域的值, 所以, 结构 `employee` 型说明的 `jim` 的 `salary` 域将表示成 `jim.salary`。怎样用指针来存取一个结构及结构的部分呢? 下面的例子程序将向你展示这个问题(请装入并运行 `SOLAR.C`):

```

/* SOLAR.C--Example from Chapter 4 of Getting Started */
#include <graphics.h>
#include <stdio.h>
#include <string.h>
typedef struct {

```

```

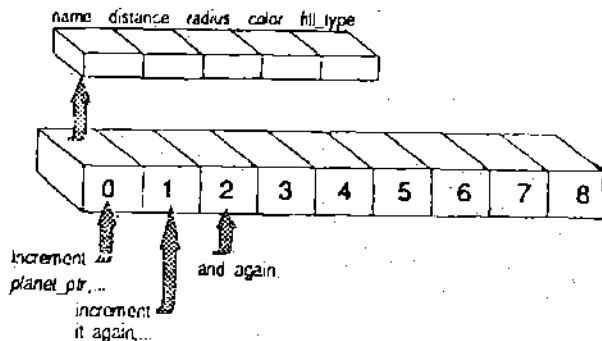
    chr name[10];
    float distance;
    float radius;
    int color;
    int fill_type;
} planet;
planet solar_system[9];
planet *planet_ptr;
int    planet_num;

int main()
{
    strcpy(solar_system[0].name,"Mercury");
    solar_system[0].distance = 0.4;
    solar_system[0].radius = 0.4;
    solar_system[0].color = EGA_YELLOW;
    solar_system[0].fill_type = EMPTY_FILL;
    planet_ptr = solar_system;
    planet_ptr++;          /* Point to second planet structure */
    strcpy (planet_ptr->name," Venus");
    planet_ptr->distance = 0.7;
    planet_ptr->radius = 1.0;
    planet_ptr->color = EGA_BROWN;
    planet_ptr->fill_type = SOLID_FILL;
    planet_ptr = solar_system;      /* Reset to first element */
    for(planet_num = 0; planet_num < 2; planet_num++, planet_ptr++)
        printf("\nPlanetary statistics:\n");
        printf("Name: %s\n", planet_ptr->name);
        printf("Distance from Sun in AU: %4.2f\n",
            planet_ptr->distance);
        printf("Radius in Earth radii: %4.2f\n", planet_ptr->radius);
        printf("Color constant value %d\n", planet_ptr->color);
        printf("Fill pattern constant value %d\n",
            planet_ptr->fill_type);
    }
    return 0;
}

```

程序中的->表示将在下面解释。

下图表示了前面的代码在 for 循环的第一步中是如何表现的。



planet solar_system[9]
Array of nine planet structures

图 4.9 使用提针存取结构中的数组

结构 `planet` 是前面程序中的扩充，增加了 `color` 和 `fill_type` 作为其成员，下一条说明语句：

```
planet solar_system[9];
planet *planet_ptr;
```

说明了一个数组 `solar_system`，它有九个成员，每个都是结构 `planet` 的一个拷贝；同时也说明了一个指向 `planet` 型数据的指针 `planet_ptr`。

`main` 中的第一组语句，使用了数组和结构表示法来为第一个 `planet` 结构的元素赋初值。为了指明在结构数组中的结构元素，使用了

```
array_name[index].member_name
```

的形式。这样第四个行星在太阳系中的距离可以表示成 `solar_system[3].distance`。

因为 `planet_ptr` 是指向结构类型 `planet` 的指针，建立的指针运算将谨慎移动指针，越过足够多的字节以达到 `solar_system` 的下一个元素处。

下一个语句指针来初始化一个 `planet` 型结构，在使用指针之前，必须赋给它一个有效地址，语句：

```
planet_ptr = solar_system
```

使得指针 `planet_ptr` 指向数组 `solar_system` 的第一个元素，这是一个已初始化过的元素，其中含有行星水星的信息。然后，语句 `planet_ptr++` 将指向下一个元素。

通过指针的正确指向，有关第二个行星金星的信息被赋给了 `solar_system` 的第二个元素，在此，是使用指针 `planet_ptr` 来存取结构 `planet` 中的元素，而不是使用下标表示法。其形式是

```
pointer_name->member_name
```

所以，当前指针指向的 `planet` 结构中的元素 `distance` 就表示成 `planet_ptr->distance`。

程序的其余部分显示已初始化的两个 `planet` 结构。首先，`planet_ptr` 通过赋给 `solar_system` 地址的方式设置指回到第一个元素处，`for` 循环增加 `planet_ptr` 的值，并获取了结构的元素值。请注意，表示式 `planet_ptr->distance` 比 `solar_system[index].distance` 少了一些麻烦，这里的 `index` 是当前元素号。

事实上，对数组而言，其下标(`index`)实际上是一种指针，是由数组的首地址加上下标值构成的，它把内部的指针运算转换成了数组地址再迭加上

`index * sizeof(type)`

在此, `type` 是数组说明的类型, `sizeof` 是 C 的一个运算, 它返回被一个类型或变量占据的字节数。

4.14.5 使用指针来从函数返回值

指针可以改变变量的实际值, 或者是用于调用函数的变量值, 至今为止, 看到的函数调用都是使用常量或者变量名, 当使用如下这样的调用语句时, `draw(x_cor, y_cor, size, color)`, 就把指定变量的值传递给了函数 `draw`, 实际上, 函数只获得了这些值的一份拷贝, 并且可以用名字来指向它们并操作它们, 但是, 这对调用语句所用的实际变量毫无影响。

有时, 通过变量名来调用一个函数, 并且让函数能够实际改变这些变量的值是很有用的。下面这个函数是一个例子:

```
void swap(int *a, int *b)
```

`swap` 通过调用函数的方式来交换两个变量的值, 为了让函数存取这些变量, 这就不能是值而要是地址, 所以函数把新的值写回这些变量的内存单元。而且参数是合适变量的指针, 而不是变量自己。因为指针指向一个地址, 所以就用语句 `swap(&x, &y)` 来调用 `swap` 交换 `x` 和 `y` 的值。

下面是 `swap` 函数的定义, 并且用一个 `main` 函数来测试它(装入运行 `INTRO33.C`):

```
/* INTRO33.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
void swap(int *, int *); /* This is swap's prototype */
int main()
{
    int x = 5, y = 7;
    swap(&x, &y);
    printf("x is now %d and y is now %d\n", x, y);
    return 0;
}

void swap(int *a, int *b) /* swap is actually defined here */
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

注意 `swap` 函数使用了间接(*)存取来指向变量 `x` 和 `y` 所含的值, 首先把 `x` 的值存入一个中间变量, 然后把 `y` 的值送入 `x` 中, 存于中间临时变量 `temp` 中的 `x` 先前值则被送入 `y` 中。



图 4.10 在函数中应用指针

当使用指针时，函数中的一个 `return` 语句并不局限于只返回一个值，函数将改变指定让它存取的任何变量的值，虽然这也可以使 `a` 和 `b` 成为全局变量而做到(把变量在函数体外定义即成为全局变量)。偶而改变全局变量的值是很容易的，而指针可管理私有变量间的数据传递。

4.15 利用系统资源

迄今为止所有的例子程序近乎于在真空中操作。一些程序读取从键盘敲入的数据，并且每段程序都在屏幕上显示些东西，但没有一个数据是以永久的方式存放，如果想恢复数据，就必须重新运行一遍程序。在实际应用中，程序通常不得不从磁盘驱动器、通信接口或者其它输入源上读取大量数据；当数据处理时，程序可能需要将之送往打印机或者写入磁盘文件以备后用，这适用于文字处理、页扩充和数据库等例子。

这一节将告诉你如何用 Turbo C++ 来读写磁盘文件。理论上，Turbo C++ 建立了称之为流(stream)的东西，数据可在其间移动。MS-DOS 文件及其处理它们的库函数用较低的方法进行工作，但是流的特色是作为可移植性介绍的，它们将允许在使用文件之时，不用考虑目标机的操作系统。

流代表一个磁盘文件，或者一个放在某些可读数据或传递数据的设备上的文件(许多设备用作输入或输出，但并不两者俱备——通常并不能从打印机上读取数据，也不能完成数据送给键盘的工作)，在你的程序中，不能直接操纵流。取而代之的是定义一个类型为 `FILE`(在 `stdio.h` 中定义)的结构 `struct`，因为这个结构为文件内容管理着一片内存缓冲区，所以你可以说明一个指向 `FILE` 变量的指针，并用之操作整个文件。

为了打开一个流, 将要使用运行库中的一个函数(通常是 `fopen`), 你将指明是否想从流中读取数据, 或者向流写入数据, 或者两者都作, 你也可以表明是否把流当作文本或二进制数进行处理。

文本流应用于通常的 DOS 文件, 标准的 I/O 流假定文本文件由文本行组成, 每个文本行以一个单一的新行符(ASCII 换行字符)作为结束。而 DOS 在每行末尾存入回车和新行符。当说明一个文本流时, Turbo C++ 把输入的 CR/LF 转换成单个的换行字符, 而把文件存入磁盘时, 却把单个换行符转换成与 DOS 操作一致的 CR/LF。

4.15.1 使用文件和流

Turbo C++ 程序使用磁盘文件涉及的基本步骤如下:

1. 使用 `#include <stdio.h>` 包含使用文件所需要的说明。
2. 说明一个类型 `FILE` 的指针
3. 说明用于从文件接收信息的数据对象, 例如, 一个字符数组保存文本的一行, 或者一个数值型数组, 或者一个结构数组。
4. 用 `fopen` 打开文件, 必须包括文件名、存取类型等需要的信息(只读、写、附加等等, 请参见《程序员指南》第二章的“运行时间库交叉参考”中更多的文件存取函数)。
5. 检查 `fopen` 返回值确证文件是否打开。
6. 使用适宜的库函数写或读数据。对文本数据而言, `fprintf` 和 `fscanf` 的工作方式与用于屏幕的 `printf` 和 `scanf` 相似; `fputc` 和 `fgetc` 处理单个字符, 对已知大小的结构化数据块, `fwrite` 和 `fread` 更合适。
7. 当用完之后, 用 `fclose` 关闭文件。

下面的程序打开一个文件, 并存入三行文本数据, 然后再读回。

装入运行 `INTRO34.C`:

```
/* INTRO34.C--Example from Chapter 4 of Getting Started */
#include <stdio.h>
#include <stdlib.h>
FILE *textfile;      /* Pointer to file being used */
char line[81];       /* Char array to hold lines read from file */
int main()
{
    /* Open file, testing for success */
    if (((textfile = fopen("intro34.txt", "w")) == NULL) {
        printf("Error opening text file for writing\n");
        exit(0);
    }
    /* Write some text to the file */
    fprintf(textfile, "%s\n", "one");
    fprintf(textfile, "%s\n", "two");
    fprintf(textfile, "%s\n", "three");
    /* Close the file */
}
```

```

fclose(textfile);
/* Open file again */
if ((textfile = fopen("intro34.txt", "r")) == NULL {
    printf("Error opening text file for reading\n");
    exit(0);
}
/* Read file contents */
while ((fscanf(textfile, "%s", line) != EOF))
    printf("%s\n", line);
/* Close file */
fclose(textfile);
return 0;
}

```

4.15.1.1 打开流

实际上，你始终在使用文件。每个 C 程序可以自动对下面表中的五种流存取：

表 4.11 Turbo C++ 中先期打开的流

名字	功能	相联的设备
stdin	标准输入	键盘
stdout	标准输出	屏幕
stderr	标准出错	屏幕
stdaux	标准辅助设备	串行口
stdprn	标准打印	打印口

前面两个缺省的流能用多种方法再指向它们，由于缺省关系，Turbo C++ 程序将期望从键盘获取输入，把输出送往屏幕。当使用 FILE 类型的指针打开一个文件时，你就打开了一个附加流。

在例子程序中，说明了一个指向流的指针 `textfile`，`fopen` 语句则把这个指针与按写方式打开的磁盘文本文件联结在一起；`if` 语句则检查返回值是否为 `NULL`，而 `NULL` 是预定义的用于标志文件打开错误的指针。

4.15.1.2 写文件

`fprintf` 写三行文本数据给文件，`fprintf` 与 `printf` 工作原理相象，除了 `fprintf` 首先要给出流指针的名字外，其次给出格式说明符和要写的数据。请注意，必须要有换行符(`\n`)隔开文本行，以便后面可以用 `fscanf` 读取。在数据写入文件后，程序就用 `fclose` 关闭文件。

4.15.1.3 读文件

为了从文件读回数据，又用 `fopen` 打开了这个文件，但这次是读方式，`fscanf` 把每行读入字符数组 `line` 中，一个 `printf` 语句把这行数据显示在屏幕上。而整个事情包含在一个 `while` 循环中，而这个 `while` 循环检查 `fscanf` 返回的值是否为预定义值 `EOF`(End Of File, 文件结束标志)。

第五章 C++要素

本章将使你对 C++ 语言的风格有一个感性认识。我们不使用难懂的术语,而是将几个理论结合到简单的说明性的程序中。这些例子中的源码已在你的系统盘中提供。所以你可以研究、编辑、编译并运行它们(对于图形程序例子,当然只有在有一图形适配器和监视器后才可运行。任何 CGA, EGA, VGA 或 Hercules 配置都行)。

Turbo C++ 提供了 AT&T 版本 2.0 的所有特色, C++ 是流行的 C 语言的扩充, 加入了面向对象程序设计(OOP)的特征。

OOP 是一种试图摹仿我们建立现实世界模型方法的程序设计方法。为应付现实生活的复杂性, 我们已逐渐形成了很好的概括、分类和抽象的能力, 在我们词汇中的几乎每一个名词都表示一类对象, 享有一组属性或行为特征。在一个充满个体的狗的世界中, 我们提炼出称作狗的抽象类, 这就允许我们发展和总结有关狗的概念而无须将注意力分散到有关任何一个特殊的狗的细节。在 C++ 中, OOP 的发展利用了我们对于事物分类和抽象这样一种自然的倾向——事实上, C++ 最初就被称为“带类的 C”。

下列三个主要的性质刻画了一种 OOP 语言的特点:

- **封装:** 把一个数据结构同操作数据的函数(行为或方法)组合在一起, 封装是借助于一种新的结构和数据类型机制——类来达到的。
- **继承:** 是建立一个新的派生类。它从一个或多个先前定义的基类中继承函数和数据, 而有可能重新定义或加进了新的数据和行为, 这样就建立了类的等级。
- **多态性:** 给行为取一个名字或符号, 它从上到下共享一个类等级, 在这个等级中的每个类以适合自己的方式实现这个行为。

Borland 公司的 C++ 提供了面向对象程序设计的完整功能:

- 对程序结构和模块性的更多可控性。
- 建立一个有它们自己特殊操作的新数据类型的能力。
- 提供了帮助你建立可重复使用代码的工具。

所有这些特征, 与使用非面向对象语言相比, 可编出更加结构化、可扩充和易维护的程序。

为了获得 C++ 这些重要的益处, 你也许需要改变多年来一直认为是标准的程序设计思维方法, 一旦你这样做了, 那么, C++ 就成了一种简单、直接、高级的解决传统软件设计中许多难题的工具。

你的背景知识将影响你对待 C++ 的态度:

- 如果你不熟悉 C 和 C++:

开始时也许对理解本章讨论的新概念有些困难, 但例子的学习(和上机实验)将有助于使这些概念形象化。在开始之前, 你应该懂得 C 语言的一些基本要素(在继续阅读之前, 你可能需要复习一下第四章的内容), 作为初学者, 你有一个很明显的优势: 你几乎没有要放弃的旧的程序设计习惯。

- 如果你是有经验的 C 程序员:

C++ 是建立在 C 中现有的语法和功能之上的, 这比你不得不学习一门全新的语言要

省时多了,同时也允许你做很小的修改就可把现有的 C 程序移植到 C++ 上,你也并没有失掉 C 的强有力和高效性,且能加进类的表达能力和控制对内部数据访问的保护机制。

■如果你用 Turbo Pascal 5.5 进行程序设计:

Turbo Pascal 5.5 包含了许多在 C++ 中出现的面向对象的特征,当你不得不解决这两种语言之间语法的差异时,会发现, Turbo Pascal 5.5 的对象和 Turbo C++ 的类很相似,你将能识别出 C++ 的成员函数就是 Turbo Pascal 5.5 的方法,并可能注意到许多其它的相似之外,还将发现他们主要的差别是 C++ 对数据访问有更严格的控制。

■如果你很熟悉另外一种面向对象的程序设计语言。

在 C++ 中,你会发现一些差别:

■首先, C++ 的语法是传统的过程语言的语法。

■其次,在编译时, C++ 和 Smalltalk 实际处理对象的方式是不同的, Smalltalk 的联编在运行时刻才彻底完成(迟后联编), C++ 则允许编译时刻先前联编和迟后联编。

在本章,我们首先详细介绍三个关键的 OOP 思想——封装、继承和多态性,首先给出代码片断来说明各个主题,然后,给出完整的可编译的程序例子,主要用于说明面向对象的表示方法在图形上的用途。间或还穿插一些其它有关 C++ 如何处理字符串或其它数据结构的例子。

5.1 封装(Encapsulation)

C++ 是怎样改变你与代码和数据的作用方式的呢? 一个重要的方法就是封装: 代码和数据衔接在一起构成一个具有类类型的对象,例如: 你也许已开发一个诸如数组之类的数据结构,来存储在一个屏幕上画一个字符所需的信息,开发一段代码(函数)来完成显示、缩放和旋转、增强亮度和设置字符颜色。

在传统的 C 语言中,常用的解决办法是,把数据结构和相关的函数放入一个单独编译的源文件中,以图把代码数据作为模块来看待,这当然是朝正确方向迈出的一步,但这还不够好。在数据和代码之间没有明确的关系,你或者其它程序员,不使用提供的函数,仍旧可以直接访问数据。这会导致一些问题,例如,假若你决定用链表来取代数组,另一个干同一工作的程序员则可能认为她有更好的方法来访问这些字形数据,于是她编写了一些自己能直接操纵这些数组的函数,可问题是,那儿再也没有数组了!

C++ 通过扩展 C 的 struct 和 union 关键字的功能,以及加一个 C 中没有的关键字 class 来处理这种情况。在 C++ 中,这三个关键字都用于定义类。

在 C++ 中,一个单一的类实体(由 struct, union 或 class 定义)将函数(称作成员函数)和数据(称作数据成员)组合在一起,你通常给类一个诸如 Font 之类的有意义的名字,该名字成为一个新的类型标识符,你可以用它来说明该类的实例,即对象:

```
class Font {  
    //here you declare your members: both data and functions;  
    //don't worry how for the moment.  
};  
Font Tiffany;           //declares Tiffany to be of type class  
                        //Font.
```

注意在 Turbo C++ 中,可以用两条斜杠//来表示 C 和 C++ 中的一个单行注释,如果

愿意的话,仍可以使用 `/* */` 注释符。实际上,它们对于长的注释特别有用。

警告: `//` 注释的使用通常对 C 编译器是不可移植的,但对其它 C++ 编译器是可移植的。

变量 `Tiffany` 是 `Font` 类的一个实例(有时叫做实例化),你可以象一个正规的 C 数据类型一样使用类名 `Font`,例如:你可以说明数组和指针:

```
Font times[10]; // declare an array of 10 Fonts
```

```
Font *font_ptr; // declare a pointer to Font
```

C++ 的类和 C 的结构之间的主要区别是成员的可访问性,一个 C 结构的成员在其域内对任何表达式或函数都是可用的,对于 C++,你可以通过说明各个成员为 `public`, `private` 或 `protected` 来控制对结构和类成员(代码和数据)的访问,C++ 的联合与 C 中的联合非常相似,所有成员都是 `public`。在后面,我们将更详细地说明这三种访问级别。

C++ 的结构与联合与 C 相比,能提供更多的用处。它们可以包含函数说明和定义,也可以包含数据成员。在 C++ 中,关键字 `struct`、`union` 和 `class` 都可用来定义类。

■ 用 `struct` 定义类是一种缺省认为所有成员都是公有的类(但如果愿意,你可以改变这种安排)。

■ 用 `union` 定义类,所有成员都是公有的(这种访问级别是不能被改变的)。

■ 在一个用 `class` 定义的类中,成员缺省认为是私有的。(也有改变这种访问级别的方法)。

所以,每当我们谈到 C++ 中的类时,我们包括了结构和联合,也包括用关键字 `class` 定义的类。

典型地,可以限制成员函数访问数据成员:通常做法是使数据成员私有,使成员函数公有。

让我们回到处理字形的那个问题上,C++ 的类概念是怎样来帮助我们的呢?

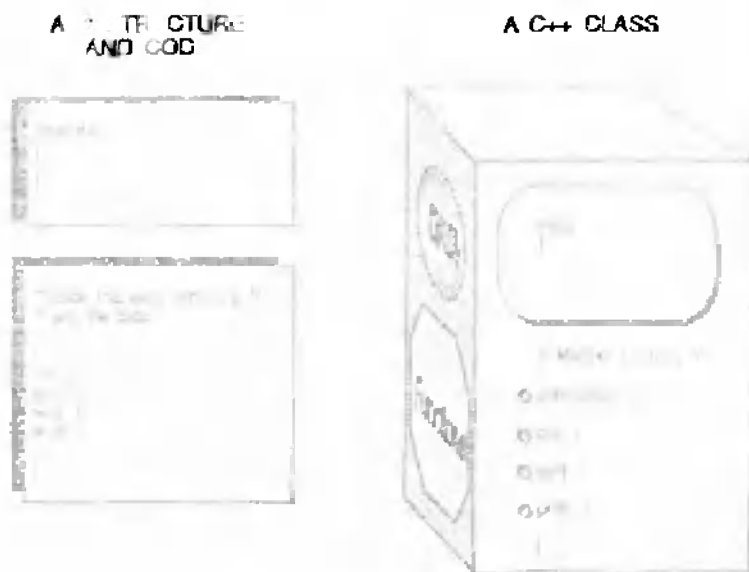


图 5.1 传统的 C 封装与 C++ 的对比

通过建立一个合适的 Font 类, 你就能确保私有的字形数据只有通过你为此目的建立的公有成员函数才能访问和处理。现在, 任何时刻, 你都可以自由地把字形数据结构从数组变成链表。当然, 你还需要重新写成员函数来处理新的字形数据结构。但如果函数名和参数不改变, 那么在你系统其它部分的程序(及程序员)就不会由于你的改动而受到影响。

图 5.1 比较了 C 和 C++ 访问字形数据的方法。

象这样在类中使用封装技术有助于提供模块化的程序, 如同象 Ada 和 Modula-2 这些语言中出现的一样, C++ 的类建立了一个定义良好的接口, 以帮助你设计、实现、维护和重用程序, 由于许多错误可很快追溯到一个特定的类, 所以调试 C++ 程序就很简单了。

类的概念导致了数据抽象的思想。我们的字形数据结构不再束缚于某个特定的物理实现, 相反, 它由允许在其上的操作(成员函数)来定义。同时在 C 中把程序看成是函数集合, 数据是二等公民的传统思想也被改变了, C++ 类把数据和函数同等对待, 互相依赖。

5.2 继承(inheritance)

科学的描述分支(在科学的解释和预测的目标可能带来成果以前是必需的)花费很长时间依据某种特性对对象进行分类, 将你的分类看作一棵家族树, 树根由一个单一的概括性门类组成, 这有助于组织你的分类。

例如, 昆虫学家对昆虫的分类如图 5.2 所示。在昆虫门中有两种分类: 有翅类和无翅类, 在有翅类下是数目众多的种类: 蛾、苍蝇、蝴蝶等。

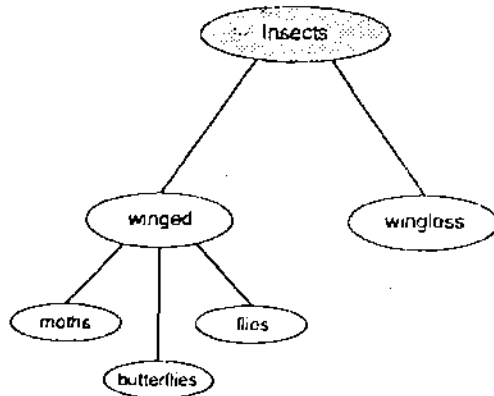


图 5.2 昆虫分类

这个分类的过程叫分类学, 这是对 OOP 继承机制一个很好的比喻。

在试图对一些新的动物或对象进行分类以前, 我们所要问的问题是: 它与其它一般的类有多少相似之处? 差别有多大? 每一个不同的类都由一组用于描述它的行为和特征组成, 我们从一个标本家族树的顶端开始, 然后沿分支而下, 并同时沿此途径提出这些问题, 最高层是最普遍的, 问题也最简单: 有翅还是无翅? 每一层都比它之前的层更具体。

一旦某个特征被定义下来, 所有在它之下的种类都要包含该特征, 所以, 一旦确定一

个昆虫为苍蝇目(flies)中的一个，我们就不必指出一个苍蝇有一对翅膀，该苍蝇从它的目中继承了这个特征。

OOP 是一个建立类的等级的过程，C++ 注入 C 中的一个重要机制是，类可以从较简单普遍的类中继承特征，这种机制就叫做继承(inheritance)。继承提供了通用的功能，允许根据需要进行更具体的定义。若类 D 继承类 B，我们说 D 是派生类(derived class)，而 B 是基类(base class)。为一个特定的应用建立一个理想的类等级决不是没有意义的工作。花费了几百年时间建立起来的昆虫分类学仍在变化和争论之中。在你书写一行 C++ 代码以前，你必须认真考虑在哪级需要哪个类。随着应用的深入，你可能发现需要新类来根本改变整个类等级。同样要记住的一点是，大量的工作正在为 Turbo C++ 提供兼容的类库，所以不要重新发明太多的类库。

偶然你遇到一个类，它可以由以前建立的多于一个的类的特性组合而成。C++ 2.0 版提供了称作多重继承的机制(在 C++ 早期版本中不存在)，这里一个派生类可以继承两个或更多的基类。后面你将看到，这可以通过对单一继承机制进行逻辑扩充来达到。

5.3 多态性(Polymorphism)

多态性一词来源于希腊语：“有多种形态”。在 C++ 中，多态性使用虚函数来实现，虚函数使你在一个类等级中使用相同函数的多个版本，在运行时决定所用的特定版本(称作迟后联编)。

5.4 重载(Overloading)

在 C 中，一个给定的函数名只能对应一个函数。例如，如果你说明并定义下列函数：

```
int cube(int number);
```

你可以得到一个整数的立方，但假如你要对一个浮点数或双精度数求立方呢？为达此目的，你自然又得说明另外的函数，但不能使用名字 cube：

```
float fcube(float float_number);
```

```
double dcube(double double_number);
```

然而，在 C++ 中，你可以重载函数，这意味着你可以有相同名字的函数，但使用时带有不同的数据类型，你可以这样说明：

```
int cube(int number);
```

```
float cube(float float_number);
```

```
double cube(double double_number);
```

只要参数表不同，对于给定参数，C++ 能选定正确的函数。如果你使用 cube(10)，则调用整数版本的 cube；如果你使用 cube(2.5)，则调用双精度版的 cube；如果你使用 cube(2.5F)，传递一个浮点数字，而不是双精度数，则调用浮点版本，即使象“+”这样的运算符也可以重载和重定义，所以它们不仅可以用于对数值而且也可以用于对图形对象、串或对一给定类适合的操作。

5.5 用类来模拟现实世界

C++ 的类为建立现实世界系统的计算机模型提供了一个很自然的方式——事实上，Bjarne Stroustrup 在 AT&T 贝尔实验室设计该语言时，就是为了模型一个大型电话交换系

统。

在汽车行业, C++已有了许多应用, 例如, 模拟汽车时, 你感兴趣的是汽车物理特征的描述(车胎数量、发动机功率、重量等)和行为(加速、制动、驾驶、燃料消耗)。一个 Car 类就可以用很普通的方式来封装这些物理参数(数据)和行为(函数)。利用继承, 你可以派生出专门的 Sports 和 Station wagon 类, 加入新的数据和函数, 及修改(废弃)基类的一些函数。你为基类所编的许多代码可重用或至少可循环使用。

5.5.1 建立类: 一个图形例子

在一个图形环境中, 合适的起点是用平面几何的一个抽象点模拟屏幕上的物理像素的类。首先, 我们试着用一个叫做 Point 的结构类, 它把 x 和 y 坐标放在一起作为其数据成员。

```
struct Point {      // defines a struct class called Point
    int X;  // struct member data are public by default.
    int Y;
};
```

现在你可以说明几个具有 struct point 类的特殊变量(为简单起见, 我们通常不严格地称这些变量具有 Point 类型)。在 C 中, 你要象这样来说明:

```
struct Point Origin, Center, Cur_pos, AnyPoint;
```

但在 C++ 中, 你所作的只是:

```
Point Origin, Center, Cur_pos, AnyPoint;
```

一个 Point 类型的变量(诸如 Origin)是 Point 类型许多可能实例中的一个。要特别注意, 你是给 Point 类的实例赋值(特定的坐标), 而不是 Point 本身, 初学者经常把数据类型 Point 和 Point 类型的实例变量混淆起来, 你可以写成 Center = Origin(把 Origin 的坐标赋给 Center), 但 Point = Origin 这种写法是毫无意义的。

当你需要分开考虑 X 和 Y 坐标时, 可以把它们认为是结构中互不相干的成员(域)X 和 Y。另一方面, 当你要考虑 X 和 Y 坐标共同确定屏幕上一个位置时, 你可以把它们认作是 Point。

假若你要在屏幕上描述的位置显示一个亮点, 除了你已知的 X 和 Y 的位置成员外, 你将要加进一个成员来说明是否在该位置有一个被置亮的点。

这里是包含所有三个成员的新结构类型:

```
enum Boolean { false,true }; // false = 0,true = 1
struct Point{
    int X;
    int Y;
    Boolean Visible;
};
```

这段代码用了一个枚举类型(enum)来建立一个 true/false 测试, 由于枚举类型的值以 0 开始, 所以, Boolean 有以下两个值之一: 0 或者 1(false 或 true)。

5.5.2 说明对象

同其它数据类型一样，也有指向类的指针和类数组：

```
Point Origin; //declare object Origin of type Point
Point Row[80]; //declare an array of 80 objects of type Point
Point *point_ptr; //declare a 'pointer to type Point'
point_ptr = &Origin; //point it to the object Origin
point_ptr = Row; //then point it to Row[0]
```

5.5.3 成员函数

正如你前面所见到的，C++的类既包含有成员函数，又包含有数据成员。一个成员函数就是一个在类定义中说明并紧紧粘附于类的函数(在诸如 Turbo Pascal 和 Smalltalk 的面向对象语言中，成员函数就是所谓的方法(method))。

让我们把一个简单的成员函数 GetX 加入到 Point 类中，往类中加入成员函数有两种方法：

- 在类中定义函数。

- 在类中说明，在类外定义。

这两种方法有不同的语法和技术含义。

第一种方法如下示：

```
struct Point{
    int X, Y;
    Boolean Visible;
    int GetX() (return X;) //inline member function defined
};
```

这种定义方式缺省认为 GetX 为内部函数(inline function)。简单地说，内部函数是一个很小的有用的被编译函数，这和一个宏不一样，但也同样避免了函数调用的开销。

注意，内部成员函数的定义遵循通常 C 对函数定义的语法：函数 GetX 返回一个整型值，没有参数。在 { 和 } 之间的函数体包含着定义函数的语句——在我们的例子中，只有一个简单的句，return X;。

第二种方法，在 Point 结构中，你简单地说明成员函数(使用通常 C 函数说明的语法)，然后，在类定义体之外的其他地方，提供它的完整定义(用函数体中的句子来完成)。

```
struct Point {
    int X,Y;
    Boolean Visible;
    int GetX(); //member function declared
};

int Point::GetX() { //member function defined
    return X; //outside the class
}
```

在类定义之外定义的成员函数仍可以定义为内部的(如果需要的话)，但此时必须显式地使用关键字 inline 来满足这种要求。

请特别注意在函数定义 `Point::GetX` 中作用域冲突解决运算符(`::`)的使用。类名 `Point` 需要告诉编译器 `GetX` 属于哪个类(有可能在附近有属于其它类的 `GetX`)。内部定义不需要 `Point::` 修饰符, 因为 `GetX` 清楚地表明是属于 `Point`。

在 `GetX` 前面的 `Point::` 也用于另一个目的, 它的影响扩展到了函数定义, 以使在 `"return X;"` 中的 `X` 作为对 `Point` 类中成员 `X` 的引用。还要注意, 不管其物理位置如何, `Point::GetX` 的体是在 `Point` 的作用域之内的。

无论我们采用哪种定义方法, 重要的一点是我们现在有了一个依附于类 `Point` 的成员函数 `GetX`。正因为它是一个成员函数, 它才可以存取任何属于 `Point` 数据成员、变量。在我们这个简单的例子中, `GetX` 只存取 `X`, 并返回它的值。

5.5.4 调用一个成员函数

成员函数表示了对类的对象的操作, 所以当我们调用 `GetX` 时, 我们必须设法指明是操作 `Point` 的哪个对象。如果 `GetX` 是一个普通的 C 函数(或一个 C++ 非成员函数), 这个问题就不会出现了——你只须简单地用表达式 `GetX()` 调用函数就行了。对于成员函数, 你必须提供目标对象的名字, 所用语法是在 C 中引用结构成员语法的自然扩展。就如同你对对象 `Origin` 的 `X` 成员的引用采用 `Origin.X`, 对对象 `EndPoint` 的 `Y` 成员的引用采用 `EndPoint.Y` 一样, 你用 `Origin.GetX()` 或 `EndPoint.GetX()` 来调用 `GetX`。"." 运算符用作类成员的选择器, 可用于数据成员和成员函数。一般的调用语法是:

类的对象名 . 成员函数名(参数表);

同样, 如果你有一个指向 `Point` 对象的指针, 你可以使用指针成员选择器 "`->`":
`Point_Pointer -> GetX()`。在本章的例子中, 你可以看到许多这种成员函数调用的例子。

5.5.5 构造函数和析构函数

在 C++ 中, 有两个扮演关键角色的特殊类型的成员函数, 即构造函数和析构函数。为了体会它们的重要性, 有必要将话题说得远点。传统语言的一个共同问题是初始化: 在使用一个数据结构之前, 必须给它分配内存和初始化。考虑对早先定义的结构进行初始化:

```
struct Point {  
    int X;  
    int Y;  
    Boolean Visible;  
};
```

没有经验的程序员也许试图以下列方式给 `X`、`Y` 和 `Visible` 成员赋初值。

```
Point ThisPoint;  
ThisPoint.X = 17;  
ThisPoint.Y = 42;  
ThisPoint.Visible = false;
```

这样做也可以, 但它紧紧地束缚于某个特定的对象, `ThisPoint`。如果不在一个 `Point` 对象需要初始化, 那么就需要更多的赋值语句来做同样的事情。接下来很自然的一步就是建立一个初始化函数, 综合这些赋值语句来处理作为一个参数 传递的任一 `Point` 对象

```

void InitPoint(Point *Target, int NewX; int NewY)
{
    Target->X = NewX;
    Target->Y = NewY;
    Target->Visible = false;
}

```

该函数采用了一个指向 Point 对象的指针并用它给它的成员赋以给定的值(在用指向类成员的指针时, 再次注意→运算符), 你已正确地构造了专门处理 Point 结构的函数 InitPoint, 可为什么必须继续指定 InitPoint 所处理的类类型和特定的对象呢? 答案是, 因为 InitPoint 不是一个成员函数。对于真正的面向对象的环境, 我们真正所需的是能够初始化任一 Point 对象的成员函数, 它就是构造函数的任务之一。

C++的目标是使用户定义的数据类型和语言提供的内部类型对语言本身来讲构成一个完整的统一体(并象内部类型一样容易使用)。为此, C++提供了叫做构造函数的特殊成员函数类型。一个构造函数指定了一个类类型的新对象怎样建立, 也就是说, 怎样分配内存和初始化。它的定义可以包含内存分配, 成员赋值, 从一种类型向另一种类型的转换, 以及其它有用代码。构造函数可以是用户定义的, 或是 C++缺省产生的。构造函数既可以显式调用, 也可以隐含调用。在你定义一个类的新对象时, C++编译器就自动地调用适当的构造函数, 这可发生在一个数据说明、拷贝一个对象或用运算符 new 动态分配一个新对象时。

如其名所示, 析构函数通过清除值来释放内存、撤消以前由构造函数建立的类对象。如同构造函数一样, 析构函数也可以被显式(用 C++运算符 delete)或隐含(例如当一对象退出其作用域时)地调用。如果你没有为一给定类定义一个析构函数, C++产生一个缺省的版本, 后面, 我们再讨论定义析构函数的语法。首先还是让我们看看构造函数是怎样定义的。

下面的 Point 版本中加入了一个构造函数:

```

struct Point {
    int X;
    int Y;
    Boolean Visible;
    int GetX() {return X;}
    Point (int NewX, int NewY); //constructor declaration
};
Point::Point(int NewX, int NewY) //constructor definition
{
    X = NewX;
    Y = NewY;
    Visible = false;
};

```

这里定义构造函数是在类定义之外进行的, 构造函数也可以在类中定义, 这时是内部的; 或者在类定义之外定义而用关键字 inline 使之变成内部的。可是要注意几点, 由构造

函数产生的代码总量并不总是和它的定义中可见的源代码成正比的。

注意,构造函数的名字和类名 `Point` 一样,这使编译器知道它在处理一个构造函数。同时要注意,构造函数也可以和其它函数一样有参数,这里的参数是 `NewX` 和 `NewY`。构造函数体的建立就象其它成员函数体的建立一样,所以构造函数可以调用类中的任一成员函数或存取任一数据成员。不过,构造函数决没有返回类型——甚至也不是 `void` 类型。

现在,你可以这样说明一个新的 `Point` 对象:

```
Point Origin(1, 1);
```

该说明调用了 `Point` 构造函数。如你随后将见到的一个类可以有多于一个的构造函数,与其它 C++ 重载函数一样,根据参数表自动调用适当的版本。你也将看到,如果你没有定义构造函数, C++ 将产生一个没有参数的缺省构造函数。

C++ 中另一个有用的诀窍是函数参数可带有缺省值。

```
Point::Point(int NewX = 0; int NewY = 0) //revised constructor
definition
```

```
{
    //as before
}
```

说明 `Point Origin(5);` 将 `X` 初始化为 5, 而 `Y` 由缺省置为 0。

5.5.6 代码和数据相结合

面向对象程序设计最重要的原则之一是在程序设计阶段,程序员要把代码和数据结合起来考虑。代码或数据都不是存在于真空中的,数据引导代码流,代码处理数据。

当你的数据和代码是分开的实体时,总存在使用错误的数据调用正确的函数和使用正确的数据调用错误的函数的危险。使这两者正确匹配是程序员的责任,与传统的 C 不一样, C++ 提供了较好的类型检查,但最好的情况下,它也只能指出哪些不能一起使用。

通过将代码和数据说明结合在一起, C++ 的类有助于保持它的一致。典型情况,为得到一个类的某一个数据成员,你调用属于这个类的某个成员函数,返回所需要的成员数据的值。为设置一个域的值,你可以调用一个成员函数,将一个新值赋给这个域。

5.5.7 成员访问控制: 私有的(private)、公有的(public)和保护的保护的(protected)

在 C++ 中增强了 `struct` 以允许将数据和函数结合在一起。它不能用于封装或模块化目的。正如我们以前提到的,对一个 `struct`,所有成员函数和数据成员的访问,在缺省时是公有的,也就是说,在相同作用域内的任何语句都可以读或者改变一个 `struct` 类的内部数据。正如我们以前所提到的,这是不理想的,并有可能导致错误,好的 C++ 设计实践使用数据隐藏,即信息隐藏——保持成员数据是私有的或保护的,并提供一个访问它的授权界面。一般的原则是,使所有数据私有以便只有通过公作的成员函数才可访问在它。只有很少情况下,需要公共的而不是私有或保护的数据成员。同样,一些只包含在它内部操作中的成员函数可作为私有或保护的,而不是公有的。

三个关键字提供了对结构或类成员的存取控制,适当的關鍵字(带有冒号)被放在所影响的成员说明前。

私有的(private): 该关键字后的成员只能被在相同类中说明的成员函数访问。

保护的(protected): 该关键字后的成员只能被相同类中的成员函数, 以及从该类派生的类的成员函数访问。

公有的(public): 该关键字后的成员可以与类说明在同一作用域内的任一地方访问。

例如, 下面是重定义 Point 结构, 使数据成员私有, 成员函数公有:

```
struct Point {  
    private:  
        int X;  
        int Y;  
    public:  
        int GetX();  
        Point(int NewX,int NewY);  
};
```

5.5.3 类: 缺省为私有的

一个 struct 类缺省认为是公有的, 所以你不得不用 private 来指定私有部分。而用 public 指定可被一般访问的部分。由于好的 C++ 实践使缺省为私有的, 并认真指定公有部分, 所以 C++ 程序员喜欢使用 class 胜过使用 struct。类和结构之间的唯一区别就在缺省私有这一点上。

Point 可以这样被重新定义为一个类:

```
class Point {  
    int X; //private by default  
    int Y;  
    public: //Needed to override the private default  
        int GetX();  
        Point (int newX, int NewY);  
};
```

对数据成员无需用 private 修饰符——它们缺省认为是私有的, 然而成员函数必须声明为公有的, 以便能在类外用来初始化和访问 Point 对象的值。

你可以在需要时, 反复定义访问控制:

```
enum Boolean {false, true};  
class Employee{  
    double salary; //private by default  
    Boolean permanent;  
    Boolean professional;  
    public:  
        char name[50];  
        char dept_code[3];  
    private:  
        int Error_check (void);  
};
```



```

public:
    Employee(double salary, Boolean permanent,
              Boolean professional, char *name, char *dept_code);
};

```

这里数据成员 salary, permanent 和 professional 缺省是私有的, 数据成员 name 和 dept_code 说明为公有的, 成员函数 Error_check 说明为私有的(为内部使用), 构造函数 Employee 说明为公有的。

5.5.9 运行一个 C++ 程序

现在是你把到此为止所学东西结合在一起, 形成一个完整的可编译程序的时候了。为了在一个集成环境中编译一个 C++ 程序, 象经常做的一样, 输入或装入你的文件到编辑器中。你可以从 IDE 中以两种方式之一运行 C++ 程序。首先, 缺省时, 所有带 .CPP 扩展名的文件采用 C++ 语法来编译, 所有带 .C 扩展名的文件采用 C 语法来编译。然而, 你也可以在 Source Option 对话框中选择 C++ Always 按钮, 把所有文件都当作 C++ 源文件看待而不理会扩展符名。

用命令行编译 C++ 程序, 只要你给出文件的扩展名 .CPP, 或者使用命令行选择项: -P。在这种情况下, Turbo C++ 假定文件具有 .CPP 扩展名。如果文件有不同的扩展名, 你必须随同文件名给出扩展名。如果你对所有 C++ 程序给定 .CPP 扩展名, 所有 C 程序给出 .C 扩展名, 那么“生活”对你来讲就容易多了。

POINT.CPP 程序定义了 Point 类, 并处理其数据的值。

```

/* POINT.CPP illustrates a simple Point class */
#include <iostream.h>    // needed for C++ I/O
class Point {           // define Point class
    int X;               // X and Y are private by default
    int Y;
public:
    Point(int InitX, int InitY) {X = InitX; Y = InitY;}
    int GetX() {return X;} // public member functions
    int GetY() {return Y;}
};
int main()
{
    int YourX, YourY;
    cout << "Set X coordinate: "; // screen prompt
    cin >> YourX;                  // keyboard input to YourX
    cout << "Set Y coordinate: "; // another prompt
    cin >> YourY;                  // key value for YourY
    Point YourPoint(YourX, YourY); // declaration calls constructor
    cout << "X is " << YourPoint.GetX(); // call member function
    cout << "\n";                 // newline
}

```

```

    cout << "Y is " << YourPoint.GetY(); // call member function
    cout << "\n";
    return 0;
}

```

现在 Point 类包含了一个新成员函数, GetY。该函数和我们以前定义的 GetX 函数的作用一样,但是它存取私有数据 Y 而不是 X,由于两个函数都很短,所以选择在类中定义的内部形式较好。

同使用 #define 命令的宏一样,一个内部函数的代码在每次函数使用时都被直接放进你的文件中,所以避免了函数调用的开销但增加了代码的大小。这是在许多程序设计中出现的典型“空间换时间”的问题。作为一个普遍的原则,只有对很短的,有 1~3 个句子的函数,采用内部定义。注意,与宏不一样,一个内部函数在函数调用中没有牺牲有助于防止错误的类型检查,函数中参数数目也与你决定是否使用“内部”有关,因为参数结构影响函数调用的开销。当函数体的全部代码比不采用内部,而调用函数所需代码少时,就应使用内部函数。在决定哪种途径是最适合你的要求之前,你也许要试一下两种方法,检查产生的汇编码。

是否使用构造函数取决于是否包含了基构造函数。派生类构造函数,特别是在层次结构中有虚函数的地方会产生大量“隐藏”代码。

在上面的例子中,Point 构造函数是作为非内部方式定义的,紧随在类说明之后。当你可以以任何次序放置你的定义(甚至放在当前文件的任何其它地方)时,用较少的单一文件的程序按它们被说明的先后次序在类定义之后放置这些定义是合适的。

随着代码变得越来越大,你很可能要把类说明放在头文件中,把类函数定义(实现代码)放入分开编译的 C++ 源文件中,然而内部函数的定义总是在头文件中。

这个程序也介绍了 C++ I/O 流库(注意在程序开始处的语句):

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

cout 表示标准输出流(缺省为屏幕)。数据(例如,变量值和字符串)通过使用“输出到”即插入运算符 << 送到 cout 中。

cin 表示标准输入流(正常情况下是键盘)。从键盘上敲入的值通过用 >> (“从...得到”或取出)运算符存于变量中。对 I/O 流使用 << 和 >> 移位算子是 C++ 中运算符重载的一个典型例子。

流函数节省了你在使用 printf 或 scanf 中需要直接处理各种格式情况的时间,它们也允许 I/O 被结合到特定的类中。

一旦 X 和 Y 的值从键盘上接收到,则用接收到的值作参数来说明类 Point 的对象 YourPoint,该说明自动地调用 Point 类的构造函数,建立和初始化 YourPoint。

试着运行本程序,结果如下所示:

```

Set X coordinate:50
Set Y coordinate:100
X is 50
Y is 100

```

5.6 继承

类不会存在于真空中，一个程序通常不得不与几个不同但又相关的数据结构共同工作。例如，你也许有一个简单的内存缓冲区，在那里你可以存储数据和取到数据。之后，你也许要建立更多的缓冲区，存储从文件中输入或输出到文件中的数据。也许有为打印机存储数据的缓冲区，有为存取从调制解调器中来的，以及到解调器中去的数据的缓冲区。这些特定的缓冲区明显地有许多共性，但是，由于磁盘文件、打印机和调制解调器中包含有工作方式不同的设备，所以每个又有些不同。

C++ 解决这种“相似但又不同”的情况的方法是，允许类从一个或更多的基类中继承其特性和行为，这是一个明显的飞跃。继承也许是 C++ 和 C 之间唯一最大的不同。从基类继承的类叫做派生类，而一个派生类本身也可能是其它由它派生出来的类的基类(回忆一下昆虫家族树)。

5.6.1 重新思考 Point 类

图形的基本单元是屏幕上单个的点(一个像素)。到目前为止，我们已设计了几个由 X 和 Y 位置定义一个真的 Point 类的几个变体。一个建立和初始化点的位置的构造函数和其它能返回当前 X 和 Y 坐标的成员函数。然而在你能画任何东西之前，你必须在像素是“on”(以某种可见颜色画出)和像素是“off”(和背景色一样)之间作出区别。当然以后你要定义一给定点应是许多种颜色中的哪一种，并且也可能有其它属性(例如，闪烁)，不久你可以以一个具有许多数据成员、一个复杂的类系统结束这件工作。

重新思考一下我们的策略，关于 Points 类的两个基本信息类型是什么呢？一种信息描述了点在哪儿(位置)，另一种信息描述了点怎么样(点的当前状态：你既可能看到它也可能看不到。但如果你能看到它，它就带有某种颜色)。两者之中，位置是最基本的，没有位置，你根本不可能有一个点。

因为所有点都要含有一个位置，你可以从一个更基本的基类 Location 中派生一个派生类 Point，在 Location 类中含有关于 X 和 Y 坐标的信息，Point 继承了 Location 中的任何东西，并加进了 Point 类所必需的新内容。

这两个相关的类可以以这种方式定义：

```
/* point.h--Example from Chapter 5 of Getting Started */
// point.h contains two classes:
// class Location describes screen locations in X and Y coordinates
// class Point describes whether a point is hidden or visible
enum Boolean {false, true};
class Location {
protected:          // allows derived class to access private data
    int X;
    int Y;
public:              // these functions can be accessed from outside
    Location(int InitX, int InitY);
    int GetX();
    int GetY();
```

```
};
class Point : public Location {      // derived from class Location
// public derivation means that X and Y are protected within Point
protected:
    Boolean Visible; // classes derived from Point will need access
public:
    Point(int InitX, int InitY);      // constructor
    void Show();
    void Hide();
    Boolean IsVisible();
    void MoveTo(int NewX, int NewY);
};
```

在这里, Location 是基类, Point 是派生类, 这个过程可以无限地继续下去。你可以定义另外从 Location 派生出的类。从 Point 派生出的其它类, 再次从 Point 的派生类派生出来的类, 等等。甚至你可以有一个从不止一个基类中派生出的类: 这叫做多重继承, 以后再讨论。设计一个 C++ 应用程序的很大一部分集中在建立类的等级结构和在程序中表达这个类的家族树中。

继承和访问控制

在我们讨论 point.h 中的各个成员函数之前, 让我们回过头来看看 C++ 的继承和访问控制的机制。

Location 类的数据成员说明为保护的, 这意味着在 Location 类和派生的 Point 类中的成员函数都可访问它们。但是“一般公有的”是不能这么做的。

如下所示, 说明一个派生类:

```
Class D:access_modifier B( //default is private
```

```
...
}
```

或者 struct D:access_modifier B(//default is public

```
...
}
```

D 是派生类的名字, access_modifier 是选择项(public 或 private), B 是基类的名字。

表 5.1 类访问性

Access in base class	Access modifier	Inherited access in base
public	public	public
private	public	not accessible
protected	public	protected
public	private	private
private	private	not accessible
protected	private	private

对类而言,缺省的 `access_modifier` 是 `private`。对结构而言,缺省情况是 `public` (注意联合既不能作为基类也不能作为派生类)。

`access_modifier` 用来修饰被继承成员的可访问性,如表 5.1 所示。

在写一个依赖于已存在类的新类时,你一定要清楚基类和派生类之间的关系。其中最重要的是清楚由说明符 `private`, `protected` 和 `public` 授予的访问级别。存取权要仔细地在父亲——儿子——孙子之间传递或抑制。不需要把你的数据暴露给非家族者或非成员者, C++ 就能做到这一点。由基类所观察到的基类成员的访问级别不必和由导出类所观察到的它的访问级别一样。换句话说,当成员被继承时,你能控制它们的访问级别如何被继承。

一个类可以从其基类私有地或公有地派生。私有派生(class 类型类的缺省情况)可使基类中公共的和保护的成员转变成派生类的私有成员,私有成员仍保持私有(尽管私有派生对类是缺省情况,但它决不是最通用的派生用法——所以我们很少使用缺省情况)。

一个 `public` 派生不改变访问级别。

一个派生的类继承了其基类的所有成员,但只能使用其基类的保护的和共有的成员,基类的私有成员不能由派生类的成员直接使用。

这里采纳的 `Location` 和 `Point` 的特殊定义将允许我们以后为更复杂的图形应用从 `Point` 中派生更进一步的类。

如果你用公有派生,基类的 `Protected` 成员在派生类仍是保护的,除了其它的公有派生类和友元外,不可以从外面访问它。这是一个好主意:不管缺省情况是什么,为避免混淆,总是指定公有的或私有的。好的注解也能改善源程序的可读性。

5.6.2 把类装入模块

象 `Location` 和 `Point` 这样的类,为了能用在更深一层的程序开发中,可以将它们包装在一起。伴随它的内部数据,成员函数和访问控制,类本来就具有模块性。在程序开发中,把每个类或一组有关的类的说明放在一个单独的头文件,而把其非内部成员函数的定义放入一个单独的源文件中,这种做法是很有道理的(关于怎样使用工程管理器来管理由多个文件组成的程序的详细情况,请看《用户手册》第二章“多文件工程管理”)。

你也可以用 `TLIB` 把几个类的目标文件合并到库中(参阅《用户手册》第五章“实用工具”学习怎样建库)。

将类模块化更进一步的好处是你可以以目标代码的形式把你的类分发给其它程序员。其他程序员不用访问你的源代码就可以从你的类中派生出新的特定的类。尽管 C++ 2.0 版是非常新,但第三者的类库已经出现,并期待你的 C++ 程序员伙伴提供更多的你可用于开始程序设计工程的类库。

现在我们可以开发一个单独编译的、含有 `Location` 和 `Point` 类的模块。首先列在前面的两个类(包括其成员函数)的说明,已放在 `point.h` 文件中(在你的源盘里)。

请再注意一下 `Point` 类是怎样从 `Location` 类中派生出来的。

```
class Point: public Location {...
```

在 `Location` 前面,关键字 `public` 是必须的,以确保派生类 `Point` 的成员函数能存取在基类 `Location` 中的保护成员 `X` 和 `Y`。除了 `X` 和 `Y` 位置成员外, `Point` 还从 `Location` 中继承了成员函数 `GetX` 和 `GetY`。 `Point` 类也添加了保护的数据成员 `Visible`(属于枚举类型

Boolean 和五个公共成员函数，其中包括了构造函数 `Point::Point()`。还要注意我们使用 `protected` 而不用 `private` 访问某个元素，是为了使 `Point` 能用于以后从 `Location` 和 `Point` 派生的类中。

文件 `POINT2.CPP` 包含了这两个类的所有成员函数的定义。

```
/* POINT2.CPP—Example from Chapter 5 of Getting Started */
// POINT2.CPP contains the definitions for the Point and Location
// classes that are declared in the file point.h
#include "point.h"
#include <graphics.h>
// member functions for the Location class
Location::Location(int InitX, int InitY) {
    X = InitX;
    Y = InitY;
};
int Location::GetX(void) {
    return X;
};
int Location::GetY(void) {
    return Y;
};
// member functions for the Point class: These assume
// the main program has initialized the graphics system
Point::Point(int InitX, int InitY) : Location(InitX, InitY) {
    Visible = false;           // make invisible by default
};
void Point::Show(void) {
    Visible = true;
    putpixel(X, Y, getcolor()); // uses default color
};
void Point::Hide(void) {
    Visible = false;
    putpixel(X, Y, getbkcolor()); // uses background color to erase
};
Boolean Point::IsVisible(void) {
    return Visible;
};
void Point::MoveTo(int NewX, int NewY) {
    Hide();           // make current point invisible
    X = NewX;         // change X and Y coordinates to new location
    Y = NewY;
};
```

```
Show();           // show point at new location
```

```
};
```

这个例子介绍基类构造函数的重要概念。当定义一个 Point 对象时，我们要使用基类 Location 的构造函数。构造函数 Point::Point 的定义以一个冒号和对基类构造函数 Location (InitX, InitY) 的调用开始。这表明 Point 构造函数首先用参数 InitX 和 InitY 调用 Location 的构造函数，建立和初始化数据成员 X 和 Y，然后，调用 point 的构造函数体，建立和初始化数据成员 Visible。通过显示式地指定一个基类构造函数，我们可以替自己省下一些编程时间(当然在更大的例子中，这种节省也许更可观)。事实上，派生类构造函数总是首先调用基类的构造函数，以保证正确建立和初始化继承的数据成员，如果基类本身也是派生的，则调用基类构造函数的过程将递归进行下去。如果没有为某个特定的类 X 定义一个构造函数，C++ 将产生一个形为 X::X() 的缺省构造函数。

如果派生类构造函数没有显式调用其基类构造函数，或者如果没有定义一个基类构造函数，那就调用缺省的基类构造函数(没有参数)。(在《用户手册》第一章“Turbo C++ 语言标准”中更详细地说明了基类构造函数。

请注意对基类构造函数的调用，Location(InitX, InitY) 出现在派生类构造函数的定义中，而不是出现在说明中。

这里的主程序说明了 point 和 Location 类的作用(在盘中的 PIXEL.CPP 文件中)。

```
/* PIXEL.CPP--Example from Chapter 5 of Getting Started */
// PIXEL.CPP demonstrates the Point and Location classes
// compile with POINT2.CPP and link with GRAPHICS.LIB
#include <graphics.h>    // declarations for graphics library
#include <conio.h>        // for getch() function
#include "point.h"       // declarations for Point and Location classes

int main()
{
    // initialize the graphics system
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "..\\bgi");
    // move a point across the screen
    Point APoint(100, 50); // Initial X, Y at 100, 50
    APoint.Show();         // APoint turns itself on
    getch();               // Wait for keypress
    APoint.MoveTo(300, 150); // APoint moves to 300,150
    getch();               // Wait for keypress
    APoint.Hide();         // APoint turns itself off
    getch();               // Wait for keypress
    closegraph();          // Restore original screen
    return 0;
}
```

5.6.3 扩充类

类的好处之一是可以修改一个类以提供所需的功能。这里的例子使用已定义的 **Location** 和 **Point** 类，派生一个新类 **circle**，同时提供了用于显示、隐藏、放大和缩小圆的函数。

```
/* CIRCLE.CPP--Example from Chapter 5 of Getting Started */
// CIRCLE.CPP  A Circle class derived from Point
#include <graphics.h>    // graphics library declarations
#include "point.h"       // Location and Point class declarations
#include <conio.h>       // for getch() function
// link with point2.obj and graphics.lib
class Circle : Point {  // derived privately from class Point
                        // and ultimately from class Location
    int Radius;        // private by default
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show(void);
    void Hide(void);
    void Expand(int ExpandBy);
    void MoveTo(int NewX, int NewY);
    void Contract(int ContractBy);
};

Circle::Circle(int InitX, int InitY, int InitRadius) : Point(InitX,InitY)
{
    Radius = InitRadius;
};

void Circle::Show(void)
{
    Visible = true;
    circle(X, Y, Radius);    // draw the circle
}

void Circle::Hide(void)
{
    unsigned int TempColor;  // to save current color
    TempColor = getcolor();  // set to current color
    setcolor(getbkcolor());  // set drawing color to background
    Visible = false;
    circle(X, Y, Radius);    // draw in background color to erase
    setcolor(TempColor);     // set color back to current color
};

void Circle::Expand(int ExpandBy)
```



```

{
    Hide();                // erase old circle
    Radius += ExpandBy;    // expand radius
    if (Radius < 0)        // avoid negative radius
        Radius = 0;
    Show();                // draw new circle
};

void Circle::Contract(int ContractBy)
{
    Expand(-ContractBy);   // redraws with (Radius - ContractBy)
};

void Circle::MoveTo(int NewX, int NewY)
{
    Hide();                // erase old circle
    X = NewX;              // set new location
    Y = NewY;
    Show();                // draw in new location
};

main()                    // test the functions
{
    // Initialize the graphics system
    int graphdriver = DETECT, graphmode;
    Initgraph(&graphdriver, &graphmode, "..\\bgi");
    Circle MyCircle(100, 200, 50); // declare a circle object
    MyCircle.Show();           // show it
    getch();                  // wait for keypress
    MyCircle.MoveTo(200, 250); // move the circle (tests hide
                               // and show also)

    getch();
    MyCircle.Expand(50);      // make it bigger
    getch();
    MyCircle.Contract(75);    // make it smaller
    getch();
    closegraph();
    return 0;
}

```

为了理解 Circle 是怎样工作的，你需要考虑在 circle.cpp 中的成员函数，同时你要想在 point.h 中所作的类说明。

首先，Circle 的成员函数必须访问基类 circle，Point 和 Location 中的各种数据。

考虑一下 Circle::Expand，它要存取 int Radius(半径)，这没有什么问题，因为 Radius

在 `Circle` 中被定义为私有的(缺省情况), 所以, `Radius` 对于 `Circle::Expand` 是可访问的——实际上, 它只能被 `Circle` 的成员函数访问(以后, 你将看到一个类的私有成员也能被定义为这个类的友元的函数所访问)。

其次, 来看一看成员函数 `Circle::Hide`, 它要从其基类 `Point` 中访问布尔数据成员 `Visible`。现在, 在 `Point` 中, `Visible` 是保护的, 并且 `Circle` 是从 `Point` 中私有(缺省情况)派生来的, 以上面概括的规则可知, `Visible` 在 `Circle` 中是私有的, 并且就象 `Radius` 一样是可访问的。注意, 如果 `Visible` 在 `Point` 中被定义为私有的, 那么对 `Circle` 的成员函数而言, 它是不可访问的。所以这将诱使你使 `Visible` 变成公共的。可是, 这是一种矫枉过正的作法: `Visible` 对于非成员函数也变得可访问。你也许会说, `protected` 对于派生类是 `private`, 但是有少许 `public` 特点: 一个导出类的成员函数可以访问一个 `protected` 成员, 同时又消除了说明这个成员为 `public` 的弊病。

最后来考虑 `Circle::Show`, 为了画圆, `Circle::Show` 要访问 `Location` 的成员 `X` 和 `Y`, 这是怎样实现的呢? `Circle` 是直接从 `Location` 派生而来, 所以存取权不是马上就可明白的。`Circle` 是从由 `Location` 派生的 `Point` 中派生出来。让我们追溯一下访问说明:

1. 成员 `X` 和 `Y` 在 `Location` 中被说明为保护的。
2. `Point` 指定为从 `Location` 公有派生, 所以 `Point` 也继承了 `X` 和 `Y` 成员, 也是保护的。
3. `Circle` 从 `Point` 使用缺省私有派生。
4. 因此 `Circle` 继承为私有的, `Circle::Show` 可以访问 `X` 和 `Y`。

注意, 在 `Location` 里 `X` 和 `Y` 仍是保护的。

通过整理这条访问权链, 你可能考虑到是否需要一个象 `PieChart` 或 `Arc` 这样的 `Circle` 派生类。是的, 你需要改变从 `Point` 派生出来的 `Circle`——应该为公有派生, 并且 `Radius` 要变成保护的。

现在就很容易看到在 `circle.cpp` 中正在发生什么。一个圆从某种意义上说, 是一个粗点, 它拥有一个点所有的东西(一个 `X`、`Y` 位置和一个可见/不可见状态), 外加上一个半径。`Circle` 类好象只有唯一的成员 `Radius`, 但不要忘了作为 `Point` 的派后类, `Circle` 继承了它的成员, 也具有 `X`、`Y` 和可见性 `Visible`。

编译并连接 `CIRCLE.CPP`、`POINT2.CPP` 和 `GRAPHIC.LIB`, 在你的源盘上的工程文件 `CIRCLE.PRJ` 将帮助你做这件事。当你按下下一个键, 你就可以看到一个圆; 再按下下一个键, 圆移动; 再按, 圆扩大; 再按, 则圆缩小。

5.6.4 多重继承

正如我们前面所述, 一个类可以从不止一个基类中继承, 这种多重继承机制是加进 `C++2.0` 版的主要特征之一。让我们来看一个实际的例子。这个程序是让你在一个圆内显示文本。

一开始, 你的想法也许是简单地把一个字符串数据成员加进 `Circle` 类, 然后给 `Circle::Show` 加段代码, 以便能使它显示由圆围绕着的文本。但是正文和圆是截然不同的两个概念: 当你想到正文时, 你想到的是字体、字符大小和其它可能的属性, 它们之中没有哪个是与圆有关的。当然, 你可以直接从 `Circle` 中派生一个新类, 并给它处理文本的能力。在处理根本不同的功能时, 不管怎么样, 建立新的“基础”基类, 并把适当特征结合起来派生一个特定的类, 通常更好。

MCIRCLE.CPP 将说明这个观点。

我们将定义一个叫做 GMessage 的新类，它在屏幕上从指定的 X 和 Y 坐标开始显示一个字符串，该类将是 MCircle 的另一个父类。MCircle 继承 GMessage::Show 并用它来画正文，所有有关的类之间的关系如图 5.3 所示。

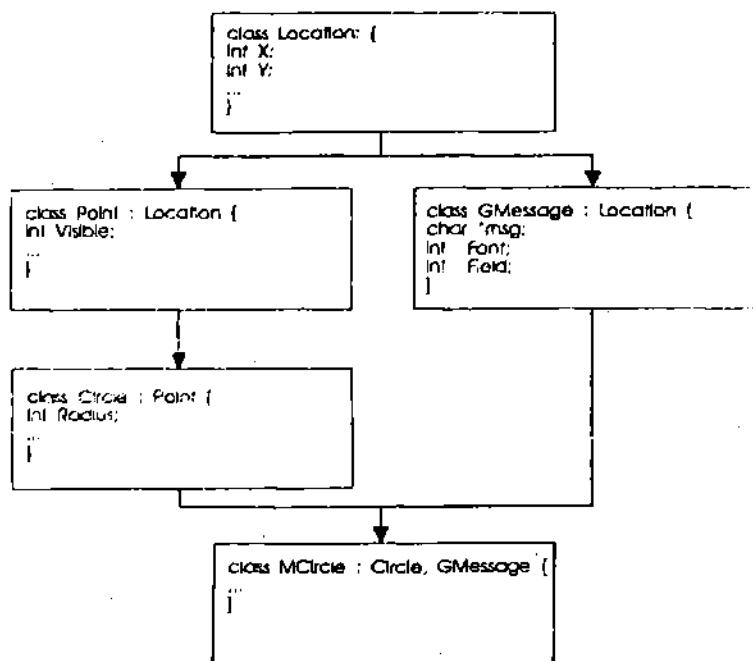


图 5.3 多重继承

```
/* MCIRCLE.CPP--Example for Chapter 5 of Getting Started */
// MCIRCLE.CPP      Illustrates multiple inheritance
#include <graphics.h> // Graphics library declarations
#include "point.h"    // Location and Point class declarations
#include <string.h>    // for string functions
#include <conio.h>     // for console I/O
// link with point2.obj and graphics.lib
// The class hierarchy:
// Location->Point->Circle
// (Circle and CMessage)->MCircle
class Circle : public Point { // Derived from class Point and ultimately
                             // from class Location
protected:
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show(void);
```

```

};
class GMessage : public Location {
// display a message on graphics screen
    char *msg;           // message to be displayed
    int Font;            // BGI font to use
    int Field;           // size of field for text scaling
public:
    // Initialize message
    GMessage(int msgX, int msgY, int MsgFont, int FieldSize,
             char *text);
    void Show(void);     // show message
};

class MCircle : Circle, GMessage { // inherits from both classes
public:
    MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
            char *msg);
    void Show(void);     // show circle with message
};

// Member functions for Circle class
//Circle constructor
Circle::Circle(int InitX, int InitY, int InitRadius) :
    Point (InitX, InitY) // initialize inherited members
//also invokes Location constructor
{
    Radius = InitRadius;
};

void Circle::Show(void)
{
    Visible = true;
    circle(X, Y, Radius); // draw the circle
}

// Member functions for GMessage class
//GMessage constructor
GMessage::GMessage(int msgX, int msgY, int MsgFont,
                   int FieldSize, char *text) :
    Location(msgX, msgY)
//X and Y coordinates for centering message
{
    Font = MsgFont;      // standard fonts defined in graph.h
    Field = FieldSize;   // width of area in which to fit text
}

```

```

    msg = text;          // point at message
};
void GMessage::Show(void)
{
    int size = Field / (8 * strlen(msg));    // 8 pixels per char.
    settextjustify(CENTER_TEXT, CENTER_TEXT); // centers in circle
    settextstyle(Font, HORIZ_DIR, size);      // if size > 1, magnifies
    outtextxy(X, Y, msg);                     // display the text
}
//Member functions for MCircle class
//MCircle constructor
MCircle::MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
                 char *msg) : Circle (mcircX, mcircY, mcircRadius),
                             GMessage(mcircX,mcircY,Font,2*mcircRadius,msg)
{
}
void MCircle::Show(void)
{
    Circle::Show();
    GMessage::Show();
}
main()    //draws some circles with text
{
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "..\\bgi");
    MCircle Small(250, 100, 25, SANS_SERIF_FONT, "You");
    Small.Show();
    MCircle Medium(250, 150, 100, TRIPLEX_FONT, "World");
    Medium.Show();
    MCircle Large(250, 250, 225, GOTHIC_FONT, "Universe");
    Large.Show();
    getch();
    closegraph();
    return 0;
}

```

在读此程序时，检查一下类说明，并注意每个类继承了哪些数据成员和成员函数，说不定还得再看一下 `point.h`，因为 `Location` 和 `Point` 类是在那里定义的，注意，`Mcircle` 和 `GMessage` 都把 `Location` 作为它们最终的基类：`MCircle` 是通过 `Point` 和 `Circle` `GMessage` 则是直接。

在 `MCircle::Show` 定义中，你会看到两个叫做 `MCircle::Show` 和 `GMessage Show` 的函

数, 这个语法说明了::(作用域冲突解决运算符)的另一种用法。当你要调用一个被继承的函数时, 例如 Show, 编译器需要一些帮助: 要调用哪个 Show? 没有作用域冲突解决运算符, Show()将是指当前作用域中的 Show(), 即 MCircle::Show, 为调用另一个作用域中的 Show(假定你获得了访问许可), 你必须提供相应的类名, 后缀::和函数名(如果需要, 也要指定参数)。如果是一个非成员函数调用, 你要调用的 Show 该怎么办? 你就要使用不带有类名的::Show()。

在派生类中, 一个给定名字的成员函数要覆盖基类中同名的成员函数, 但以后你仍可用::得到它, C++的作用域规则与 C 略有不同。

在离开 MCIRCLE.CPP 之前, 简单概述一下 MCircle 的构造函数。首先, 你在前面已看到 Point 构造函数是怎样显式地调用在 Location 中的基构造函数的。由于 MCircle 是从 Circle 和 GMessage 继承过来的, 所以 Mcircle 构造函数可通过调用两个基构造函数, 很方便地进行初始化。

MCircle::MCircle

```
(int mcircX, int mcircY, int mcircRadins, int font, char *msg);  
Circle(mcircX, mcircY, mcircRadius),  
GMessage(mcircX, mcircY, 2*mcircRadius, msg) {  
}
```

构造函数体在这里是空的, 因为所有必要的工作已在成员初始化表中完成了(在:后, 你输入一个由逗号分开的初始化表达式表。在 Point 和 Circle 的类定义中使用了单个基类构造函数, 你已见到过这个语法的简单情形)。当 MCircle 构造函数被调用时(例如通过说明一个 MCircle 对象), 暗中将触发大量的工作。

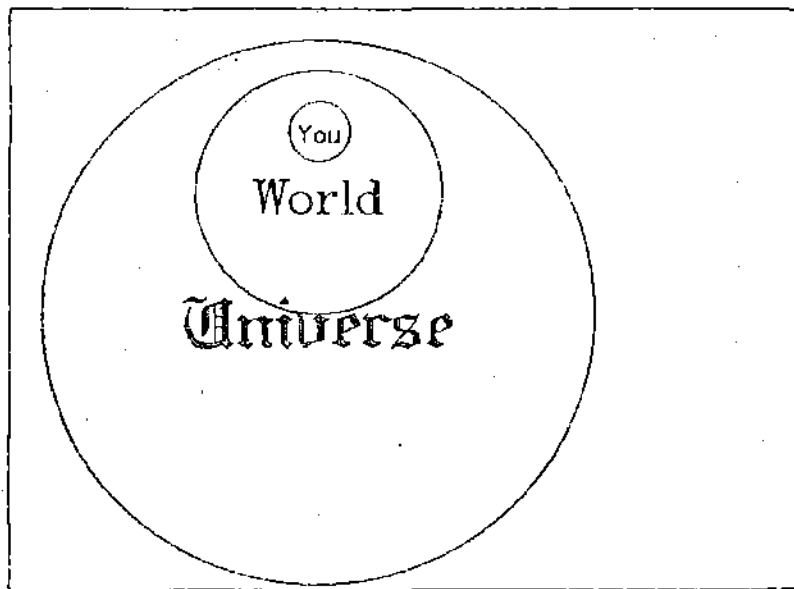


图 5.4 带有文本信息的圆

首先, 调用 Circle 的构造函数, 该构造函数然后调用 Point 的构造函数, 然后 Point 的构造函数又调用 Location 的构造函数, 最后调用 GMessage 的构造函数, 它为建立它的基

类数据成员 X 和 Y 的拷贝要调用 Location 的构造函数。在 MCircle 构造函数中给定的参数被用来初始化基类相应的数据成员。

当调用析构函数时(例如,在对象退出其作用域时),调用顺序与构造函数的顺序相反(虚基类构造函数和析构函数的顺序有点特别,这超出了本章的范围)。

顺便提一下,前面提到的一点:如果你没有提供自己的构造函数或析构函数,C++将产生和调用缺省版本。

图 5.4 是 MCircle 的输出。

5.6.5 虚函数

在我们图形类等级中的每一个类类型代表了屏幕上不同类型的图形:一个点或一个圆。说你在屏幕上可以显示一个点或一个圆当然是有道理的。以后,如果你定义了表示其它诸如线、矩形、弧形等的图形的类,你就可以为它们之中的每一个写一个可以在屏幕上显示其对象的成员函数。面向对象的方法认为,所有这些图形类型具有在屏幕上显示自身的能力。

对于每一个对象类型,其不同之处是其在屏幕上显示自身的方式。用一个绘点例程序画一个点时,只要一个 X、Y 坐标,或许还有一个颜色值就行了。画一个圆则需要更复杂的图形例程序,不仅要考虑到 X 和 Y,还要考虑半径。更深入一步,一个弧需要一个起始角度和一个终止角度,以及一个不同的作图算法。当然,同样的情况适用于隐藏,移动以及基本形状的处理中。

到目前为止,你所见的普通成员函数允许我们给每种形式的类定义一个 Show 函数,但是它们缺乏基本的要素。每当引入一个新类型的类和它的成员函数 Show 时,基于目前已有的类和成员函数的图形模块必须改变源码并重新编译。原因是,到目前为止所披露的 C++ 机制只允许三种方法来解决这个问题:调用哪一个 Show 呢?

1. 根据参数的特征加以区分——例如:

Show(int, char) 与 Show(char*, float) 并不是同一函数。

2. 使用作用域冲突解决运算符,因此,

Circle::Show 有别于 Point::Show 和 ::Show。

3. 根据类对象来解决: ACircle.Show 调用 Circle::Show, 同时, Apoint.Show 调用 Point::Show。类似地,对于指向对象的指针, Apoint_Pointer->Show 调用 Point::Show。到这里为止,所有这些函数的解决都是在编译时进行的——也就是所谓的先前(或静态)联编。

一个典型的图形工具箱将向用户提供在 .H 源文件中定义的、被预先编译好的成员函数所形成的 .OBJ 或 .LIB 代码。由于先前联编的限制,用户不能方便地加入一个新的类,甚至开发者在扩充软件包时也面临着要干额外的乏味的工作。C++ 提供了一种灵活的机制来解决这些问题:借助于叫做虚函数的特殊成员函数的迟后(或称动态)联编。

关键的概念是虚函数调用是在运行时刻解决(因此有术语——迟后联编)。在实际情况中,它表明象调用哪个 Show 函数之类的决定,将推迟到运行期间知道相关的对象的类型以后才可做出。虚函数,如 Show,“隐藏”在预先编译好的工具箱库的类 B 中,并不象 B 中的一般函数那样结合到 B 的对象上。你可以为自己喜爱的形式自由建立一个从 B 中派生的类 D,并写一个合适的函数放在 Show 中,然后编译并把你 .OBJ 或 .LIB 代码与工具

箱的代码连接起来。不管是现存的 B 的成员函数，还是从你为 D 所写的新函数，对 Show 的调用将自动地引用正确的 Show。这种解决方法适用于在调用中包含对象类型的情况。让我们看一个虚函数，在以前为 CIRCLE.CPP 编写的代码中，我们已有了一个潜在的候选者。

5.6.6 虚函数的作用

考查一下在 CIRCLE.CPP 中的成员函数 Circle::MoveTo:

```
Void Circle::MoveTo(Int NewX, Int NewY)
{
    Boolean vis = Visible;
    if(vis) Hide(); //hide only if visible
    X = NewX; Y = NewY; //set new location
    if(vis) Show(); // draw at new location
                    // if previously visible
}
```

注意，该定义与 Circle 的基类的 Point 中出现的 Point::MoveTo 是多么的相似。事实上，返回值、函数名、形参的数量和类型(叫做函数特征)，甚至函数体本身都表现得非常相似！如果 C++ 遇到两个用相同的函数名但特征不同的函数调用，我们已经看到 C++ 编译器很明智地解决由函数名重载引起的潜在的多义性(回忆 C 不象 C++ 那样，它要求唯一的函数名)。在 C++ 中，带有不同特征的成员函数，即使它们享有相同的名字，也是完全不同的函数。

但是，我们的两个 MoveTo，乍看起来并没有给编译器提供任何用以区别的线索——那么它知道你想调用哪个函数吗？正如你所看到的，对于一般的成员函数的答案是：编译器从调用中所含的对象的类型来决定目标函数。

那么，为什么不让 Circle 继承 Point 的 MoveTo，就如同 Circle 继承 Point 的 GetX 和 GetY 呢(经由 Location)？原因是 Circle::MoveTo 中调用的 Hide 和 Show，与在 Point::MoveTo 中调用的 Hide 和 Show 不是同一个，仅仅只是名字和特征相同。从 Point 中继承 MoveTo，在试图移动一个圆时将导致调用错误的 Hide 和 Show。为什么呢？因为两个函数的 Point 的版本在编译时(先前联编)被连接到 Point 的(以后，对 Circle 的也一样) MoveTo。如你已经猜到的答案是：把 Hide 和 Show 说明为虚函数，这将推迟连接，以便在实际调用 MoveTo 来移动一个点或一个圆(或任何其它东西)时，能调用正确的 Hide 和 Show 函数。

请再次注意，如果我们要为一个独立的库预先编译 Point 和 Circle 的类定义和成员函数(实现的源代码，由于商业秘密的原因是不公布的)，我们当然不能预先知道要 MoveTo 移动的对象。虚函数不仅提供了这种技术上的好处，也提供了 OOP 时刻关心的概念上的发展。我们可以集中于开发可重用的类和方法，同时又减少了名字冲突方面的担忧。

随着可以对附加库进行扩充的可能性对许多语言成为现实，C++ 中虚函数和多重继承的使用使扩展性更加自然。你可以继承你所有基类拥有的任何东西，然后加进你所需要的、使新对象以相似的方式工作的新功能，你定义的类及其虚函数版本是有规则的功能等级的真正扩展。由于这是语言设计的一部分而不是事后才有的想法，所以在实现上几乎没

有什么阻力

在介绍了一些虚函数的优点之后,让我们看看怎样来实施贯彻它们以及你必须遵守的一些规则。

5.6.7 定义虚函数

语法很直截了当。在成员函数第一次说明之处,加一个修饰符 **virtual**:

```
virtual void Show();
```

```
virtual void Hide();
```

只有成员函数才能说明为 **virtual**,一旦一个函数说明成 **virtual**,它就不能在任何派生类中重新说明为具有相同的形参、但具有不同的返回值。如果你用相同的特征和相同的返回类型,重新说明 **Show**,那么新的 **Show** 将自动地变成虚函数,不管有没有使用 **virtual** 修饰符,这个新的虚拟的 **Show** 称为重载基类里的 **Show**。

你可以用不同的正式参数署名,随意地重新说明 **Show**(不管有没有改变返回类型)——但虚拟机制对这个版本的 **Show** 是不起作用的。初学者要避免轻率的重载——在某些情况下,一个非虚拟函数可以隐藏一个在它基类中的虚函数。

对于哪个 **Show** 被调用来说,调用特定的 **Show** 只取决于对象的类,即使调用是通过一个指向基类的指针来进行的。

例如:

```
Circle ACircle
```

```
Point* Apoint_pointer = &ACircle;
```

```
// Pointer to Circle
```

```
// assigned to pointer to base class, Point
```

```
Apoint_pointer->Show(); // calls Circle::Show!
```

vpointh 和 **VCIRC.CPP**(已在源盘中提供)是 **Point.h** 和 **CIRCLE.CPP** 将 **Show** 和 **Hide** 改成虚函数后的版本,使用 **VCIRC.PRJ** 编译 **POINT2.CPP** 和 **VCIRC.CPP**,该程序运行起来和 **CIRCLE.CPP** 非常相象。在这里,即使在它们的说明中没有使用案,因为不同点可简单地概括为:

- 在 **vpointh**, 用关键字 **virtual** 说明 **Show** 和 **Hide** 以及基类 **Point** 中的版本。有相同的参数特征和返回值,这就暗示它们也是虚拟的,即使在它们的说明中没有使用关键字 **virtual**。

- 在 **vpointh** **Point**, **Circle** 不再拥有它自己的成员函数 **vpointh**

- 为了允许访问 **MoveTo**,我们现在从 **Point** 中公有地派生 **Circle**。

让我们概述一下这些变动的意义:

Circle 的对象现在可以安全地调用从 **Point** 继承来的 **MoveTo** 了。由 **MoveTo** 调用的 **Show** 和 **Hide** 在运行时刻将连接到 **Circle** 自己的 **Show** 和 **Hide**; **MoveTo** 的任何 **Point** 对象将调用 **Point** 的版本。

5.6.8 开发一个完整的图形模块

作为一个更加完整和实际的虚函数的例子,让我们来建立一个模块,它定义了一些形态类和在屏幕上移动它们的一般方法,该模块 **figures.h** 和 **FIGURES.CPP**(在源盘中)是以

前讨论的图形工具箱的简单实现。

设计 FIGURES 模块的一个主要目标是允许模块的用户扩展模块中定义的类——并且还可以使用模块的所有特征。有趣的挑战是建立几个在屏幕上移动任意图形的方法，以响应用户的输入。

作为一个最初的方法，我们可能考虑一个函数，它把一个对象用作一个参数，然后围绕屏幕移动该对象。

```
void Drag(Point& AnyFigure, int DragBy)
{
    int DeltaX,DeltaY;
    int FigureX,FigureY;
    AnyFigure.Show();           //Display figure to be dragged
    FigureX = AnyFigure.GetX(); //Get the initial X, Y of figure
    FigureY = AnyFigure.GetY();
    //This is the drag loop
    while (GetDelta(DeltaX, DeltaY))
    {
        //Apply delta to figure X,Y
        FigureX = FigureX + (DeltaX * DragBy);
        FigureY = FigureY + (DeltaY * DragBy);
        // And tell the figure to move
        AnyFigure.MoveTo(FigureX, FigureY);
    };
};
```

注意 AnyFigure 被说明为具有 Point&类型，它的意思是“对 Point 类型的一个对象的引用”，这是 C++ 的一个新的特征。如你所知，C 一般通过值来传递参数而不是通过引用，在 C 中，如果你要直接操作一个传递给函数的变量，就不得不传递一个指向变量的指针，这将导致不灵活的语法，因为不得不记住要间接引用这个指针。C++ 通过使用引用，可以让你传递和修改一个实际的变量。为说明一个引用，只须在变量说明中，在数据类型之后附上一个&。

Drag 调用一个这里没有显示的辅助函数 GetDelta，它从用户那里接收各种 X 和 Y 的变化量，这可以从键盘、鼠标或游戏杆接收到(为简单起见，我们的例子从键盘上的箭头键中接收输入)。

对于 Drag 要注意的重要一点是：Point 类型或任何从 Point 派生的类型的任何对象都可作为 AnyFigure 的引用参数传递 Point 或 Circle 类型的对象，或者将来要定义的从 Point 或 Circle 继承来的类型都可毫无困难地传递到 AnyFigure 中。

向一个已存在的类等级中增加一个新的成员函数需要做点思考，这个成员函数应放在等级中的什么地方？考虑一下这个函数提供的功能，然后决定这种功能的应用广度。移动一个图形包括从用户那儿得到响应后变动这个图形的位置。从继承角度而言，它正好处于 MoveTo 的地位——适用于 MoveTo 的任何对象也应该继承 Drag。因此 Drag 应该成为

Point 的成员函数，以便 Point 派生的所有类型可以共享它。

在等级中解决了 Drag 的位置以后，我们就可以仔细研究一下它的定义。作为基类 Point 的一个成员函数，无需对 Point& AnyFigure 作显式引用。我们可以重写 Drag，以便它所调用的函数，例如 GetX, Show, MoveTo 和 Hide，将正确地引用适应于正移动对象的类型的版本。和我们前面见到的一样，函数 Show 和 Hide 需要特殊的和形体有关的代码，因此可说明成 virtual。无需影响到 FIGURE 模块，我们就可以重新定义它们。同样也要注意 MoveTo，因为 MoveTo 调用正确的 Show 和 Hide(你可能记起这里使 Show 和 Hide 为 virtual 的最初移动函数)。GetX 和 GetY 没有问题：作为通过 Location 从 Point 继承而来的一般成员函数，它们只简单地返回现在或将来任何派生类的调用对象的 X 和 Y 数据成员。记住，在 Location 中 X 和 Y 是保护的，所以我们必须象所示那样使用公有派生。

剩下的设计决策是是否使 Drag 为虚拟的。使一个函数成为虚拟的条件是这个函数在下面的等级中是否要改变。不存在通用的原则，但在后面我们将讨论各种情况：扩充性与性能开销(虚函数需要较多的内存和更多的存储访问周期)。我们已经看到将来有些类，例如在 CAD 应用中可能需要一个特别的移动动作。也许移动一个等距图形将需要一些比例操作等。在 figure.h 新的 Point 类定义中，我们使 Drag 成为虚拟的。

```
class Point : public Location {
protected:
    Boolean Visible;
public:
    Point(int InitX, int InitY);
    virtual void Show();           // show and Hide are virtual
    virtual void Hide();
    Boolean IsVisible() {return Visible;}
    void MoveTo(int NewX, int NewY);
    virtual void Drag(int DragBy);
};
```

头文件 figure.h 包含 FIGURES 模块的类说明。这是软件包中唯一需要以源代码形式提供的部分。

```
// figures.h contains three classes.
//
// Class Location describes screen locations in X and Y
// coordinates.
//
// Class Point describes whether a point is hidden or visible.
//
// Class Circle describes the radius of a circle around a point.
//
// To use this module, put #include <figures.h> in your main
// source file and compile the source file FIGURES.CPP together
// with your main source file.
```

```

enum Boolean {false, true};
class Location {
protected:
    int X;
    int Y;
public:
    Location(int InitX, int InitY) {X = InitX; Y = InitY;}
    int GetX() {return X;}
    int GetY() {return Y;}
};
class Point : public Location {
protected:
    Boolean Visible;
public:
    Point(int InitX, int InitY);
    virtual void Show();          // Show and Hide are virtual
    virtual void Hide();
    virtual void Drag(int DragBy); // new virtual drag function
    Boolean IsVisible() {return Visible;}
    void MoveTo(int NewX, int NewY);
};
class Circle : public Point { // Derived from class Point and
                               // ultimately from class Location
protected:
    int Radius;
public:
    Circle(int InitX, int InitY, int InitRadius);
    void Show();
    void Hide();
    void Expand(int ExpandBy);
    void Contract(int ContractBy);
};
// prototype of general-purpose, non-member function
// defined in FIGURES.CPP
Boolean GetDelta(int& DeltaX, int& DeltaY);

```

文件 FIGURES.CPP 包含成员函数定义。这是商品化中以目标或库形式提供的部分。注意我们在类的外面定义了 Circle 构造函数，因为它调用基构造函数。你可能希望试试把它定义成内部函数。非成员函数 GetDelta，如果你对 C 不熟悉，则要做点认真的研究。注意引用参数的使用，这是 C++ 的风格，其余代码很平常。

// FIGURES.CPP: This file contains the definitions for the Point

```

// class (declared in figures.h). Member functions for the
// Location class appear as inline functions in figures.h.
#include "figures.h"
#include <graphics.h>
#include <conio.h>
// member functions for the Point class
//constructor
Point::Point(int InitX, int InitY) : Location (InitX, InitY)
{
    Visible = false;    // make invisible by default
}
void Point::Show()
{
    Visible = true;
    putpixel(X, Y, getcolor()); // uses default color
}
void Point::Hide()
{
    Visible = false;
    putpixel(X, Y, getbkcolor()); // uses background color to erase
}
void Point::MoveTo(int NewX, int NewY)
{
    Hide();           // make current point invisible
    X = NewX;         // change X and Y coordinates to new location
    Y = NewY;
    Show();           // show point at new location
}
// a general-purpose function for getting keyboard
// cursor movement keys (not a member function)
Boolean GetDelta(int& DeltaX, int& DeltaY)
{
    char KeyChar;
    Boolean Quit;
    DeltaX = 0;
    DeltaY = 0;
    do
    {
        KeyChar = getch();    // read the keystroke
        if (KeyChar == 13)    // carriage return

```

```

    return(false);
    if (KeyChar == 0)      // an extended keycode
    {
        Quit = true;      // assume it is usable
        KeyChar = getch(); // get rest of keycode
        switch (KeyChar) {
            case 72: DeltaY = -1; break; // down arrow
            case 80: DeltaY = 1; break;  // up arrow
            case 75: DeltaX = -1; break; // left arrow
            case 77: DeltaX = 1; break;  // right arrow
            default: Quit = false; // bad key
        };
    };
} while (!Quit);
return(true);
}

void Point::Drag(int DragBy)
{
    int DeltaX, DeltaY;
    int FigureX, FigureY;
    Show(); // display figure to be dragged
    FigureX = GetX(); // get initial position of figure
    FigureY = GetY();
    // This is the drag loop
    while (GetDelta(DeltaX, DeltaY))
    {
        // Apply delta to figure at X, Y
        FigureX += (DeltaX * DragBy);
        FigureY += (DeltaY * DragBy);
        MoveTo(FigureX, FigureY); // tell figure to move
    };
}

// Member functions for the Circle class
//constructor
Circle::Circle(int InitX, int InitY, int InitRadius) : Point (InitX, InitY)
{
    Radius = InitRadius;
}

void Circle::Show()
{

```

```

    Visible = true;
    circle(X, Y, Radius);    // draw the circle
}
void Circle::Hide()
{
    unsigned int TempColor;    // to save current color
    TempColor = getcolor();    // set to current color
    setcolor(getbkcolor());    // set drawing color to background
    Visible = false;
    circle(X, Y, Radius);    // draw in background color to
    setcolor(TempColor);    // set color back to current color
}
void Circle::Expand(int ExpandBy)
{
    Hide();                    // erase old circle
    Radius += ExpandBy;        // expand radius
    if (Radius < 0)            // avoid negative radius
        Radius = 0;
    Show();                    // draw new circle
}
void Circle::Contract(int ContractBy)
{
    Expand(-ContractBy);    // redraws with (Radius-ContractBy)
}

```

我们现在准备通过将 FIGURES 用在一个称作 Arc 的新形体的类中测试它，这个新类在 FIGDEMO.CPP 中定义。Arc(很自然地)从 Circle 中导出。回忆一下，Drag 用于平移一个以前从未看到的形体。

```

// FIGDEMO.CPP -- Exercise for Chapter 5
// demonstrates the Figures toolbox by extending it with
// a new type Arc.
// Link with FIGURES.OBJ and GRAPHICS.LIB
#include "figures.h"
#include <graphics.h>
#include <conio.h>
class Arc : public Circle {
    int StartAngle;
    int EndAngle;
public:
    // constructor
    Arc(int InitX, int InitY, int InitRadius, int InitStartAngle, int

```

```

        InitEndAngle) : Circle (InitX, InitY, InitRadius) {
            StartAngle = InitStartAngle; EndAngle = InitEndAngle;
        void Show(); // these functions are virtual in Point
        void Hide();
    };
// Member functions for Arc
void Arc::Show()
{
    Visible = true;
    arc(X, Y, StartAngle, EndAngle, Radius);
}
void Arc::Hide()
{
    Int TempColor;
    TempColor = getcolor();
    setcolor (getbkcolor());
    Visible = false;
    // draw arc in background color to hide it
    arc(X, Y, StartAngle, EndAngle, Radius);
    setcolor(TempColor);
}
int main() // test the new Arc class
{
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "..\\hgi");
    Circle ACircle(151, 82, 50);
    Arc AnArc(151, 82, 25, 0, 190);
    // you first drag an arc using arrow keys (5 pixels per key)
    // press Enter when tired of this!
    // Now drag a circle (10 pixels per arrow key)
    // Press Enter to end FIGDEMO.
    AnArc.Drag(5); // drag increment is 5 pixels
    AnArc.Hide();
    ACircle.Drag(10); // now each drag is 10 pixels
    closegraph();
    return 0;
}

```

5.6.9 普通的还是虚拟的成员函数?

一般来讲, 因为调用一个非虚拟的成员函数比调用一个虚拟成员函数要快点, 因此我

们推荐在不考虑扩充性而性能是主要时使用普通的成员函数，否则使用虚拟函数。

为扼要总结一下我们前面的讨论，让我们假定你正准备定义一个 **Base** 类，在这个类中，你准备说明成员函数 **Action**。你如何决定 **Action** 应是虚拟的还是普通的呢？简单的规则是：有可能将来从 **Base** 派生的类将重载 **Action** 并且你希望将来的代码可以被 **Base** 访问到，那么使其为虚拟的；如果可以确信，对派生类型，**Action** 将完成相同的步骤（即使这包括调用其它的虚函数），或派生类不使用 **Action**，则使 **Action** 成为普通的。

5.7 动态对象

前面所举的所有例子，除了在 **MCIRCLCPP** 中分配消息数组外，所有的类类型的对象都是静态的或自动的，它们以一般的方式进行说明，所需的内存都是在编译时由编译器分配的。这节中我们讨论在运行时建立对象，它们的内存从系统自由贮区中分配。建立动态对象是许多程序设计领域中的一个重要技术，在那里，存储于内存中的大量的数据在程序运行前是不知道的。一个例子是自由形式的数据库程序，这种程序要快速访问在内存中存储各种大小的数据记录。

C++ 可以使用 **C** 的动态内存分配函数，例如 **malloc**，但是 **C++** 包含几个有力的扩充，那就是使对象的动态分配和释放更容易和更可靠。更重要的是，它确保调用构造函数或析构函数。例如：

```
Circle *ACircle = new Circle(151,82,50);
```

这里 **ACircle** 是一个指向类型 **Circle** 的指针，它给出可以容纳类型为 **Circle** 的对象的一块内存区。换句应说，**ACircle** 现在指向在自由存储区分配的对象。**Circle** 的构造函数根据所提供的参数初始化这个对象。

如果你分配的是一个数组而不是一个简单类型，则使用这种语法

```
new object[size]
```

例如，为分配一个称作 **counts** 的含有 50 个元素的数组，使用

```
counts = new int[50];
```

如果你想建立一个动态 **Point** 类的对象，应象这样：

```
// DPOINT.CPP - exercise in Chapter 5, Getting Started
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include "figures.h"
int main()
{
// Assign pointer to dynamically allocated object; call constructor
Point *APoint = new Point(50, 100);
// Initialize the graphics system
int graphdriver = DETECT, graphmode;
initgraph(&graphdriver, &graphmode, "..\\bgi");
// Demonstrate the new object
APoint->Show();
```

```

cout << "Note pixel at (50,100). Now, hit any key...";
getch();
delete APoint;
closegraph();
return(0);
}

```

5.7.1 析构函数和 delete

如同你可以定义一个构造函数，它在一个新对象建立时被自动调用一样，你可以定义一个析构函数，它在一个对象释放时被自动调用，也就是说清除它的值和释放它所占据的内存。

静态对象的空间由编译器分配，构造函数在 main 之前被调用而在 main 之后调用析构函数。对自动对象，当退出说明所在的作用域时(当一个封闭的分程序结束时)释放对象。你定义的任何析构函数在静态或自动对象撤消时被调用(如果你没有定义一个析构函数，C++ 使用一个隐含的由内部生成的析构函数)。

如果使用 new 运算符建立一个动态对象，则你有责任要释放它，因为 C++ 无法知道在什么时候一个对象不再需要。你使用 delete 运算符释放内存，所定义的任何析构函数在执行 delete 时被调用。

delete 运算符的语法是:

```
delete pointer
```

这里 pointer 是一个指向由 new 分配的内存的指针。

你已看见，类 X 的构造函数和类名一样，即 X::X()。类 X 的析构函数的名字是 X::~X()。为了释放内存，析构函数也可以执行别的动作，例如将成员域数据写到磁盘、关闭文件等。

5.7.2 分配动态对象的一个例子

下面这个示例程序提供使用从自由存储区中动态分配对象的实际方法，包括为释放对象析构函数的用法，程序说明一个图形对象的链接表可以在内存中建立并在这些对象不再使用时调用 delete 将它们清除。

建立一个链接表要求表中每个对象包含一个指向表中下一个对象的指针。类型 Point 没有这种指针。一个容易的方法是在 Point 中增加一个指针并确保所有 Point 类型的派生类型也继承这个指针。然而向 Point 中增加任何东西需要你有点 Point 的源代码，但前面我们说过，C++ 的一个优点就是无需重新编译就可扩充已存在的代码的能力。所以对这个例子，我们将认为我们没有 Point 的源代码并说明怎样扩充这个图形工具箱。

许多方法中的一种是无需改变 Point，建立一个不是从 Point 中派生的新类。类型 List 是一个非常简单的类，它的目的是记录下 Point 对象组成的链表的表头。因为 Point 不包含指向表中下一个对象的指针，所以需要使用一个简单的结构 Node 来满足这个要求。Node 比 List 更简单，它没有成员函数，除了一个指向类型 Point 的指针和一个指向表中下一个节点的指针外没有其它数据成员。

List 有一个成员函数以允许向由 Node 记录组成的链接表中增加一个新的图形。Add 成

员函数有一个指向 Point 对象的指针，而不是一个 Point 对象，记住在 C++ 中，类的等级规则允许公有派生的任何类型的指针在 Item 参数中传递到 List::Add 中。

程序 ListDemo 说明了一个静态变量 AList，具有 List 类型，并且用三个节点建立了一个链接表。每个节点指向一个不同的图形，它或者是一个 Point 或者是它的派生类中的一个。自由空间的字节在任何动态对象被分配以前和在所有动态对象被分配以后分别报告给用户。最后，整个数据结构，包括三个 Node 记录 and 三个 Point 对象被清除并从内存中删除，这得感谢 List 类的对象 AList 自动调用析构函数。

```
/* LISTDEMO.CPP—Example from Chapter 5 of Getting Started */
```

```
// LISTDEMO.CPP           Demonstrates dynamic objects
```

```
// Link with FIGURES.OBJ and GRAPHICS.LIB
```

```
#include <conio.h>           // for getch()
```

```
#include <alloc.h>           // for coreleft()
```

```
#include <stdlib.h>          // for ltoa()
```

```
#include <string.h>          // for strepy()
```

```
#include <graphics.h>
```

```
#include "figures.h"
```

```
class Arc : public Circle {
```

```
    int StartAngle, EndAngle;
```

```
public:
```

```
    // constructor
```

```
    Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,  
        int InitEndAngle);
```

```
    // virtual functions
```

```
    void Show();
```

```
    void Hide();
```

```
};
```

```
struct Node {           // the list item
```

```
    Point *Item;        // can be Point or any class derived from Point
```

```
    Node *Next;         // point to next Node object
```

```
};
```

```
class List {           // the list of objects pointed to by nodes
```

```
    Node *Nodes;        // points to a node
```

```
public:
```

```
    // constructor
```

```
    List();
```

```
    // destructor
```

```
    ~List();
```

```
    // add an item to list
```

```
    void Add(Point *NewItem);
```

```
    // list the items
```

```

    void Report();
};
// definitions for standalone functions
void OutTextLn(char *TheText)
{
    outtext(TheText);
    moveto(0, gety() + 12); // move to equivalent of next line
}
void MemStatus(char *StatusMessage)
{
    unsigned long MemLeft; // to match type returned by
                           // coreleft()
    char CharString[12]; // temp string to send to outtext()
    outtext(StatusMessage);
    MemLeft = long (coreleft());
    // convert result to string with ltoa then copy into
    // temporary string
    ltoa(MemLeft, CharString, 10);
    OutTextLn(CharString);
}
// member functions for Arc class
Arc::Arc(int InitX, int InitY, int InitRadius, int InitStartAngle,
         int InitEndAngle) : Circle (InitX, InitY, InitRadius)
                           // calls Circle
                           // constructor
{
    StartAngle = InitStartAngle;
    EndAngle = InitEndAngle;
}
void Arc::Show()
{
    Visible = true;
    arc(X, Y, StartAngle, EndAngle, Radius);
}
void Arc::Hide()
{
    unsigned TempColor;
    TempColor = getcolor();
    setcolor(getbkcolor());
    Visible = false;
}

```

```

    arc(X, Y, StartAngle, EndAngle, Radius);
    setcolor(TempColor);
}
// member functions for List class
List::List () {
    Node *N;
    N = new Node;
    N->Item = NULL;
    N->Next = NULL;
    Nodes = NULL;           // sets node pointer to "empty"
                             // because nothing in list yet
}
List::~List()               // destructor
{
    while (Nodes != NULL) { // until end of list
        Node *N = Nodes;    // get node pointed to
        delete(N->Item);     // delete item's memory
        Nodes = N->Next;     // point to next node
        delete N;           // delete pointer's memory
    };
}
void List::Add(Point *NewItem)
{
    Node *N;                // N is pointer to a node
    N = new Node;           // create a new node
    N->Item = NewItem;       // store pointer to object in node
    N->Next = Nodes;         // next item points to current list pos
    Nodes = N;              // last item in list now points
                             // to this node
}
void List::Report()
{
    char TempString[12];
    Node *Current = Nodes;
    while (Current != NULL)
    {
        // get X value of item in current node and convert to string
        itoa(Current->Item->GetX(), TempString, 10);
        outtext("X = ");
        OutTextLn(TempString);
    }
}

```

```

        // do the same thing for the Y value
        itoa(Current->Item->GetY(), TempString, 10);
        outtext("Y = ");
        OutTextLn(TempString);
        // point to the next node
        Current = Current->Next;
    };
}
void setlist(void);
// Main program
main()
{
    int graphdriver = DETECT, graphmode;
    initgraph(&graphdriver, &graphmode, "..\\bgi");
    MemStatus("Free memory before list is allocated: ");
    setlist();
    MemStatus("Free memory after List destructor: ");
    getch();
    closegraph();
}
void setlist() {
    // declare a list (calls List constructor)
    List AList;
    // create and add several figures to the list
    Arc *Arc1 = new Arc(151, 82, 25, 200, 330);
    AList.Add(Arc1);
    MemStatus("Free memory after adding arc1: ");
    Circle *Circle1 = new Circle(200, 80, 40);
    AList.Add(Circle1);
    MemStatus("Free memory after adding circle1: ");
    Circle *Circle2 = new Circle(305, 136, 35);
    AList.Add(Circle2);
    MemStatus("Free memory after adding circle2: ");
    // traverse list and display X, Y of the list's figures
    AList.Report();
    // The 3 AList nodes and the Arc and Circle objects will be
    // deallocated automatically by their destructors when they
    // go out of scope in main(). Arc and Circle use implicit
    // destructors in contrast to the explicit ~List destructor.
    // However, you could delete explicitly here if you wish:

```

```

// delete Arc1; delete Circle1; delete Circle2;
getch(); // wait for a keypress
return;
}

```

一旦你掌握了 LISTDEMO.CPP, 你就可以依据下面的意见研制一个更满意的解: 定义一个多重继承 Point 和 List 的新类 PointList。

5.8 C++中更多的灵活性

虽然掌握这种新的程序设计风格的细微差异之处需要花费你一些时间, 但你现在已学到了 C++ 的基本要素。还有几个附加的特征, 我们在这里简单浏览一下, 以便你知道它们是些什么和怎样使用它们。

- 在类定义以外的内部函数
- 缺省函数参数
- 重载函数和多构造函数
- 友元函数——提供对一个类进行访问的另一种方法
- 重载运算符以提供新的含义。
- 有关 C++ I/O 和流库的更多说明

5.8.1 在类定义之外的内部函数

你已经看到过你可以在类的说明内包含一个成员函数的内部定义, 如这里列举的 Point 类:

```

class Point: { //define Point class
    int X;      //these are private by default
    int Y;
public:        //public member functions
    Point(int InitX, int InitY) {X = InitX, Y = InitY;}
    int GetX(void) {return X;}
    int GetY(void) {return Y;}
};

```

Point 类的三个成员函数都被定义成内部的, 所以不需要再作单独定义。对只有几行的函数, 这提供了一种更紧凑的、较易阅读的描述方法。

函数也可以说明为内部的。只是不同点在于你必须以关键字 inline 开始函数说明。例如在 LISTDEMO.CPP 中, 有一个操作只是简单地将图形方式下, 由文本输出位置向下移一行(它在函数 OutTextLn 中使用)。如果这个函数在代码的许多地方都被使用, 则有效的方法是将其说明为一个独立的内部函数。

```
inline void graphLn() {moveto(0, gety() + 12); }
```

如果你愿意, 你可以将你的内部定义写成类似于常规函数定义:

```

inline void graphLn()
{
    moveto(0, gety() + 12);
}

```

5.8.2 带有缺省参数的函数

你可以定义一个函数，用比定义要少的参数来调用它，此时未提供的参数使用缺省值。如果打算经常使用这些缺省值，则这种简单的调用方式也节省了打字时间。你没有失去灵活性，因为当想忽略缺省值时，只需简单地定义想使用的值即可。

例如，下面 Circle 的构造函数指定缺省半径为 50，缺省圆心在(X = 200, Y = 200)。当然一个移植性更好的程序必须确定可用的图形硬件，并依此调整这些值。

```
class Circle:public Point { //Derived from class Point and
                           //ultimately from class Location
protected:
    int Radius;
public:
    Circle(int InitX=200, int InitY=200, int InitRadius = 50);
    void Show(void);
    void Hide(void);
    void Expand(int ExpandBy);
    void Contract(int ContractBy);
};
```

说明

```
Circle ACircle;
```

给出一个缺省圆心在(200, 200)，半径是 50 的圆。说明

```
Circle ACircle(50,100);
```

给出一个圆心在 50, 100，半径为 50 的圆。说明

```
Circle ACircle(300);
```

给出一个在 x = 300，使用缺省的 Y=200 和 radius=50 的圆。

任何缺省参数必须是在参数表最右边连续给出。例如，你不能说明

```
void func(int a = 10, int b, int c);
```

因为编译器无法知道正被提供的是哪个值。

5.8.3 有关重载函数的更多说明

重载是 C++ 中重要的和极具影响力的概念。当几个不同的函数(无论是成员函数或非成员函数)，以相同的名字在相同的作用域内被定义时，它们被称为是重载的。你已遇见过在前面定义的称作 cube 的三个函数(早期 C++ 版本需要这种说明之前冠以关键字 overload，但现在已不再使用这种方式)。

基本观点是重载函数的调用通过比较调用中实参类型和函数定义中的形参类型来作区分的。这种实际使用的区分二义性的规则超过了一个初学者的范围，并且很少对初学者有什么影响。其中较复杂的是带有缺省实参或变数目参数情况下的函数调用；要考虑 C 的参数类型转换，同时也要考虑其它对 C++ 是特殊的类型转换。当面临着一个重载函数时，编译器试图找到最好的匹配，如果没有则导致编译错误。

最常用的情况是重载一个构造函数，以便提供几种不同的方式建立一个类的一个新对象。为说明这点，我们将定义一个非常简单的串(string)类。

```
//STRING.CPP—Example from Chapter 5 of Getting Started */
#include <iostream.h>
#include <string.h>
class String {
    char *char_ptr;    // pointer to string contents
    int length;        // length of string in characters
public:
    // three different constructors
    String(char *text);    // constructor using existing string
    String(int size = 80);    // creates default empty string
    String(String& Other_String); // for assignment from another
                                // object of this class
    ~String() {delete char_ptr;};
    int Get_len (void);
    void Show (void);
};
String::String (char *text)
{
    length = strlen(text); // get length of text
    char_ptr = new char[length + 1];
    strcpy(char_ptr, text);
};
String::String (int size)
{
    length = size;
    char_ptr = new char[length+1];
    *char_ptr = '\0';
};
String::String (String& Other_String)
{
    length = Other_String.length;    // length of other string
    char_ptr = new char {length + 1}; // allocate the memory
    strcpy (char_ptr, Other_String.char_ptr); // copy the text
};
int String::Get_len(void)
{
    return (length);
};
```

```

void String::Show(void)
{
    cout << char_ptr << "\n";
};
main ()                                // test the functions
{
    String AString ("Allocated from a constant string.");
    AString.Show();
    String BString;                    // uses default length
    cout << "\n" << BString.Get_len() << "\n" ; //display length
    BString = "This is BString";
    String CString(BString);          // invokes the third constructor
    CString.Show();                   // note its contents
}

```

类 `String` 有三个不同的构造函数。第一个需要一个普通的串常数。例如 `"This is a string"` 并用这些内容初始化一个串。第二个构造函数使用缺省长度 80，并为这个串分配空间，但并未在这空间中存储任何字符(这可以被用来建立临时缓冲区)。注意，你在调用构造函数中使用不同的长度就可忽视这个缺省值。例如，不说明 `String Astring`，而说明成 `String AString(40)`。

第三个构造函数使用一个对类型 `String` 的另一个对象的引用作参数，有了这个构造函数，你现在可以写这样的语句：

```

String AString("This is the first string"); //Create and initialize
String BString = AString; //Create, assign BString from AString

```

注意构造函数被包括在一个对象生命周期的三个相关的但又不同的方面：建立、初始化和赋值。对类赋值的“=”运算符的使用引导我们讨论另一个话题，运算符重载。除非你为某个类定义了一个特殊的“=”运算符，否则 C++ 缺省采用成员到成员的赋值。

5.8.4 重载运算符以提供新的含义

C++ 有一个可在其它语言中找到的特征：已有的运算符，例如 +，可被重新定义，以使它们以一种适当的、用户定义的方式对你自己的类对象进行操作。运算符以一种简洁的方式做一些事情。如果你没有它们，则象 `line*width+pos` 这样的表达式必须写成：`add (mult(line, width), pos)`。幸运的是在 C(和 C++) 中的算术运算符已知道怎样处理所有的数值数据类型——例如，同样的 + 运算符可用于 `int` 值也可用于 `float` 值，使用了相同的运算符，但生成的代码很不相同，因为整数和浮点数在内存中的表示是不同的。换句话说，象 + 这样的运算符即使在 C 中也被重载了。C++ 简单地扩充这个观点，以允许已有运算符用于用户定义的版本中。

为定义一个运算符，你定义一个函数，其名字为关键字 `operator` 后跟运算符的符号(例如，`operator+` 定义了 + 运算符的一个新版本)。所有的运算符函数通过定义被重载：它们使用在 C 中已有意义的运算符，但是它们被重定义为用于一个新的数据类型。例如 + 运算符已具备对任何两个标准数值类型(`int`、`float`、`double` 等)的值加在一起的能力。

现在我们可以向 String 类增加一个+运算符。这个运算符合并两个串对象(和 BASIC 中一样), 并返回适当长度和内容的一个串对象的结果。因为合并是“加到一起”, 所以+符号是最适合于使用的一个符号。BASIC 的支持者经常批评 C 没有这种自然的字符串操作。使用 C++, 你可以比 BASIC 内部提供的串操作做得更好。

文件 XSTRING.CPP, 已放在你的盘上, 对 STRING.CPP 作下述扩充以提供一个简单的+运算符。

```
//XSTRING.CPP-Example from Chapter 5 of Getting Started */
// version of STRING.CPP with overloaded operator +
#include <iostream.h>
#include <string.h>
class String {
    char *char_ptr;    // pointer to string contents
    int length;        // length of string in characters
public:
    // three different constructors
    String(char *text);    // constructor using existing string
    String(int size = 80);    // creates default empty string
    String(String& Other_String); // for assignment from another
                                // object of this class
    ~String() {delete char_ptr;}; // inline destructor
    int Get_len (void);
    String operator+ (String& Arg);
    void Show (void);
};
String::String (char *text)
{
    length = strlen(text); // get length of text
    char_ptr = new char[length + 1];
    strcpy(char_ptr, text);
};
String::String (int size)
{
    length = size;
    char_ptr = new char[length+1];
    *char_ptr = '\0';
};
String::String (String& Other_String)
{
    length = Other_String.length;    // length of other string
    char_ptr = new char [length + 1]; // allocate the memory
```

```

    strcpy (char_ptr, Other_String.char_ptr); // copy the text
};
String String::operator+ (String& Arg)
{
    String Temp( length + Arg.length );
    strcpy(Temp.char_ptr, char_ptr);
    strcat(Temp.char_ptr, Arg.char_ptr);
    return Temp;
}
int String::Get_len(void)
{
    return (length);
};
void String::Show(void)
{
    cout << char_ptr << "\n";
};
main ()                                // test the functions
{
    String AString ("The Quick Brown fox");
    AString.Show();
    String BString(" jumps over Bill");
    String CString;
    CString = AString + BString;
    CString.Show();
}

```

当你运行这个程序时，CString 被置成两个串 AString 和 BString 的合并。所以 CString.Show() 将显示

The Quick Brown Fox jumps over Bill

重载的 + 带有二个显式参数，所以你可能会问它是怎样合并两个串的。编译器处理 AString+BString 表达式为

AString.operator + (BString)

所以 + 运算符确定访问了两个串对象。第一个是当前正被引用的 String 对象，另一个是一个辅助的串对象。这个运算符函数将两个串的长度加在一起，使用库函数 strcat 合并两个串的内容，然后返回合并后的串。这里使用了隐藏指针 this。this 是什么？

对成员函数的每个调用为调用所施于的对象建立一个指针，这个指针可以通过关键字 this 引用(在 OOP 用语中称作自引用或自引用指针)，允许函数访问这个对象。现在 this 具有类型“指向串的指针”，所以返回值必须是 *this，这就是所需的当前对象。也要注意，在一个函数调用中这个对象的每个成员可通过表达式 this -> member 引用。明确一点的是：this 只存在于成员函数，友元函数没有 this。

当重载运算符时，有几个限制：

- C++ 不区分前缀或后缀的++和--。

- 你希望定义的运算符必须在语言中已经存在。例如你不能定义运算符#。

- 你不能重载下列运算符：

., * :: ?:

- 被重载的运算符保持它原先的优先级。

- 如果@表示一个一元运算符，则表达式@x 和 x@可被解释成 x.operator()或 operator@(x)。如果两者都被定义，则编译器试图通过参数匹配来解决二义性。同样对一个被重载的二元运算符@，x@y 可能表示 x.operator@(y)或 operator@(x, y)。如果两种形式都被定义，则编译器需要检查参数。在串版本中你已看到一个二元运算符+的例子，在那里，AString+BString 被解释作 AString(operator+(BString))。

5.8.5 友元函数(Friend function)

一般来讲，访问一个类的私有成员被限制在这个类的成员函数内。偶然有必要允许外面的函数访问这个类的私有数据。在一个类说明中的 friend 说明让你定义外面的函数(甚至外面的类)访问被说明类的私有成员。你有时会看到一个重载的运算符被说明为一个友元(friend)，但一般来讲，使用友元函数要小心——如果在你的项目中必须使用友元，那么这经常表明要修改你的类等级。

但是假定有一个漂亮的格式化打印函数 Fancy_Print，你想让它访问你的类 String 对象的内容。将下面一行加到成员函数说明表中：

```
class String {  
...  
    friend void Fancy_Print(String& AString);  
...  
}
```

在这个假想的例子中，函数 Fancy_Print 可以访问 String 类对象的成员 char_ptr 和 length。即如果 AString 是一个串对象，Fancy_Print 可以访问 AString.char_ptr 和 AString.length。

如果 Fancy_Print 是另一个类(例如 Format 类)的一个成员，在友元说明中使用作用域运算符：

```
friend void Format::Fancy_Print(String& AString);
```

你也可以使整个类成为这个被说明类的友元，在说明中使用 class：

```
friend class Format;
```

现在类 Format 的任何成员函数可以访问类 String 的私有成员。注意在 C++ 中，和生活中一样，友谊不具有可传递性：如果 X 是 Y 的一个友元，Y 是 Z 的一个友元，并不能得出 X 是 Z 的友元。

友元说明只有在确实必要时才使用。在没有它时你必须建立一个复杂的类等级时候。从性质上来说明，友元说明破坏了封装和模块性。特别地，如果你发现你想使整个类成为另一个类的友元，则考虑派生一个公有的派生类并用它访问所需的成员来替代它的可能性。

5.8.6 C++流库

虽然所有的标准 I/O 函数(例如 `print` 和 `scanf`)仍可使用,但 C++ 级提供了一组类和函数用于 I/O 操作,它们定义在 `iostream` 库中。为访问它们,你的程序中必须有命令 `#include <iostream.h>`,正如你已在我们给出的几个例子中注意到的那样。

使用 `iostream` 比使用 `stdio` 有许多优点:语法简单、优美和直观。C++ 的流机制也更有效和更灵活。例如格式化输出通过扩充使用重载而被简化,同样的运算符可被用于预定义的和用户定义的数据类型,避免了 `printf` 参数表的复杂性。

通过将流用作模拟从一个源(或生产者)到一个终点(或消费者)的任何数据流的一个抽象, `iostream` 提供一个丰富的类等级,以处理缓冲的和非缓冲的文件和设备 I/O。

Turbo C++ 也支持较早的(1.X 版)C++ `stream` 库,以帮助程序员向 C++ 2.0 新的 `iostream` 库的过渡。如果你有任何 C++ 代码使用已废弃的 `stream` 类,仍可以在 Turbo C++ 中维护和运行它。然而这给了你一个选择,当你编写新的代码时,应使用更有效的 `iostream` 而避免使用 `stream`。《程序指南》第三章“C++流”解释了 `stream` 和 `iostream` 之间的差异,并提供转换时的建议。同时你也可以参见盘中的 `OLDSTR.DOC` 文件。

在这章中我们只介绍 `iostream` 中简单的类。更多的细节可以阅读《程序员指南》的第三章“C++流”。你也可以浏览一下盘中 `iostream.h`,看一下那里定义的许多类以及它们是怎样使用单一和多重继承派生的。

标准 I/O

C++ 提供四个预定义的流对象,定义如下:

- `cin` 标准输入,通常指键盘,对应于 C 中的 `stdio`。
- `cout` 标准输出,通常指屏幕,对应于 C 中的 `stdout`。
- `cerr` 标准错误输出,通常指屏幕,对应于 C 中的 `stderr`。
- `clog` `cerr` 的一个使用完全缓冲的版本(C 中没有等价的)。

你可以重定向这些标准流来自或到其它的设备和文件(在 C 中你只能重定向 `stdin` 和 `stdout`)。你已在本章的例子中见到 `cin` 和 `cout` 的用途。

简单总结一下 `iostream` 等级,从基本的到特殊的,如下:

- `streambuf` 为内存缓冲器提供方法
- `ios` 处理流状态变量和错误
- `istream` 从一个 `streambuf` 中进行格式化和非格式化的字符转换
- `ostream` 向一个 `streambuf` 中进行格式化和非格式化的字符转换
- `iostream` 组合 `istream` 和 `ostream` 对一个单一流进行双向操作
- `istream_withassign` 为 `cin` 流提供构造函数和赋值运算符
- `ostream_withassign` 为 `cout`, `cerr` 和 `clog` 流提供构造函数和赋值运算符。

`istream` 类包括为标准类型(`int`, `long`, `double`, `float`, `char` 和 `char*(串)`)定义重载的 `>>` 运算符。因此 `cin >> x;` 调用在 `iostream.h` 中定义的 `istream cin` 相应的 `>>` 运算符,将输入流放入由变量 `x` 表示的内存位置。同样 `ostream` 类定义了重载的 `<<` 运算符,它允许语句 `cout << x;` 将 `x` 的值送到用于输出的 `ostream cout`。

这些运算符函数返回对相应流类型的引用(例如 `ostream&`)以传递数据。这允许你将几个这样的运算符写在一起以便输出或输入一系列字符。

```
int i=0, x=243; double d=0;
```

```
cout << "The value of x is " << x << "\n";
cin >> i >> d; // key an int, space, then a double
```

第二行显示"The value of x is 243"后跟一个新行。另一个语句将忽略空白,将键入的字符转换成一个整数放入*i*中,忽略紧随的空白,将下一个键入的字符转换成一个双精度值并放入*d*中。

下面的程序简单地将*cin*拷贝到*cout*。在没有进行重定向情况下,它将你的键盘输入送到屏幕:

```
// COPYKD.CPP    Copies keyboard input to screen
#include <iostream.h>
main()
{
    char ch;
    while (cin >> ch)
        cout << ch;
}
```

注意你怎样将*cin>>ch*当作一个一般的布尔表达式测试的。这是在类*ios*中定义的一个有用技巧。简单说,一个表达式,例如(*cout*)或(*cin>>ch*)被强制为一个指针,它的值取决于流的错误状态。一个空指针(测试为*false*)表明流中的一个错误,而一个非空指针(测试为*true*)说明没有错误。你也可以使用!*cout*改变这种测试,因此(!*cout*)对*cout*流中的一个错误是真(*true*),而一切正常时为假(*false*):

```
if(!cout)errmsg("Output error!");
```

格式化输出

C++中简单的I/O是有效的,因为依据要转换的数据类型只进行最少的转换。对整数,转换和*printf*缺省情况一样。语句

```
int i=5; cout << i;
```

和

```
int i=5; printf("%d",i);
```

结果一样。

格式由在*ios*中枚举的一组格式状态标志决定。对每个打开的流,它们决定转换的基数(十进制、八进制、十六进制)、左对齐或右对齐、浮点数格式(科学的或定点的)以及在输入中空白是否应跳过。你可以改变的其它参数包括域宽度(用于输出)和用于填充空白域的字符。这些标志通过各种成员函数可被测试、设置和清除。下面一段程序说明*ios::width*和*ios::fill*是怎样工作:

```
int previous_width, i = 87;
previous_width = cout.width(7); // set field width to 7
                                //and save previous width
cout.fill('*');                //set fill character to *
cout << i << "\n";              //display *****87 <newline>
//after << the width is cleared to 0
//previous width may have been set without a subsequent <<
```

```
//so you may want to restore it with the following line.
```

```
cout.width(previous_width);
```

设置宽度为零(缺省)表明显示将尽情况所需采用多个屏幕位置。如果给定的宽度不足以正确表达,则假定宽度为零(也就是说不做截断)。对所有的类型,缺省是右对齐的(左填充)。

setw 和 unsetw 是用于设置和清除格式标志的两个常用函数:

```
cout.setf(ios::left, ios::adjustfield);
```

这条语句设置左填充。第一个参数使用用于各种位位置的助记符(可能使用&和|组合在一起),而第二个参数是格式状态的目标域。unsetf 的工作方式一样,但清除所选择的位。(参见《程序员指南》中的第三章“C++流”)。

控制符

一种用于设置格式标志(以及完成其它事务)的更好方法是使用称作控制符的特殊机制。和<<, >>运算符一样,控制符可被嵌入到一个流操作链中:

```
cout<<setw(7)<<"dec"<<i<<setw(6)<<oct<<j;
```

如果没有控制符,这需要六条语句。

参数化的控制符 setw 用一个整型参数设置域宽。

非参数化的控制符,例如 dec, oct 和 hex 设置转换基数为十进制,八进制和十六进制。在上面的例子中, int i 将以十进制方式显示在宽度为 7 的域中; int j 将以八进制显示在宽度为 6 的域中。

其它简单的参数化的控制符包括 setbase, setfill, setprecision, setiosflag 和 resetiosflag。为使用任何一个参数化控制符,你的程序必须包含头文件 iomanip.h 和 ostream.h。非参数化的控制符不需要 iomanip.h。

可用的非参数的控制符包括:

ws(空白选取符):istream >>ws;将忽略 istream 中的任何空白。

endl(结束一行并刷新):ostream <<endl;将在 ostream 中插入一个新行,然后清 ostream 缓冲区。

ends(用 null 结束串):ostream <<ends;将向 ostream 中添加一个 null。

flush(刷新输出流):ostream << flush;清除 ostream。

put, write 和 get:

有两个输出函数值得一提: put 和 write, 在 ostream 中说明成:

```
ostream& ostream::put(char ch);
```

```
//send ch to ostream
```

```
ostream& ostream::write(const char* buff, int n);
```

```
//send n characters from buff to ostream; watch the size of n!
```

put 和 write 让你向一个 ostream 对象输出无格式的二进制数据。put 输出一个单个字符,而 write 可以指定的缓冲区输出任何数目的字符。当你想输出包含许多 null 的一行数据时,write 是有用的。(注意,输出二进制数要求文件以二进制方式打开)。一般的串操作不能使用,因为它要求终止于一个 null 上。

put 的输入版本称作 get:

```
char ch;
```



```
cin.get(ch);
```

```
//grab next char from cin whether whitespace or not
```

另一个版本的 `get` 让你从一个 `istream` 输入任何数目的二进制字符，直到达到预定的最大数目，然后将它放入预定的缓冲区中(和使用 `write` 一样，文件必须以二进制方式打开):

```
istream& istream::get(char *buf, int max, int term='\n');
```

```
//read up to max chars from istream, and place them in buf. Stop if
```

```
//term char is read.
```

你可以设置 `term` 为特定的终止字符(缺省是换行字符)，在 `max` 数目的字符转换进 `buf` 之前，如果遇到这个终止字符则 `get` 终止。

磁盘 I/O

`istream` 库包含从 `streambuf`、`ostream` 和 `istream` 派生的许多类，因此允许对文件 I/O 方法做较宽的选择。例如 `filebuf` 类支持使用文件描述符的 I/O，它带有用于打开、关闭和寻址的成员函数。将它与通过 `stdio` 文件结构支持 I/O 操作的 `stdiobuf` 相比，它允许在混合使用 C 和 C++ 代码时有某些兼容性。

初学者常使用的类是 `ifstream`(从 `istream` 派生)、`ofstream`(从 `ostream` 派生)和 `fstream`(从 `istream` 派生)。这些通过使用 `filebuf` 的对象都支持格式化文件 I/O。这里的一个例子将已存在的一个磁盘文件拷贝另一个特定的文件中:

```
/* DCOPY.CPP -- Example from Chapter 5 of Getting Started */
```

```
/* DCOPY source-file destination-file
```

```
 * copies existing source-file to destination-file
```

```
 * If latter exists, it is overwritten; if it does not
```

```
 * exist, DCOPY will create it if possible
```

```
 */
```

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

```
#include <process.h> // for exit()
```

```
#include <fstream.h> // for ifstream, ofstream
```

```
main(int argc, char* argv[]) // access command-line arguments
```

```
{
```

```
    char ch;
```

```
    if (argc != 3) // test number of arguments
```

```
    {
```

```
        cerr << "USAGE: dcopy file1 file2\n";
```

```
        exit(-1);
```

```
    }
```

```
    ifstream source; // declare input and output streams
```

```
    ofstream dest;
```

```
    source.open(argv[1], ios::nocreate); // source file must be there
```

```
    if (!source)
```

```
    {
```

```

    cerr << "Cannot open source file " << argv[1] <<
        " for input\n";
    exit(-1);
}
dest.open(argv[2]); // dest file will be created if not found
                  // or cleared/overwritten if found
if (!dest)
{
    cerr << "Cannot open destination file " << argv[2] <<
        " for output\n";
    exit(-1);
}
while (dest && source.get(ch)) dest.put(ch);
cout << "DCOPY completed\n";
source.close(); // close both streams
dest.close();
}

```

首先注意 `#include <fstream.h>` 也需要包含 `iostream.h`。DCOPY 使用访问命令行的标准方法来检查用户是否定义了两个文件。当在 `main` 函数中使用这个参数表时，参数 `argc` 包含命令行参数的数目(包括程序名本身)，串 `argv[1]` 和 `argv[2]` 包含输入的两个文件名。调用这个程序的典型命令行是

```
dcopy letter.spr letter.bak
```

为看看 DCOPY 是怎样工作的，检查下面几行

```
ifstream source; //declare an input stream(ifstream object)
```

```
...
```

```
open source(argv[1], ios::nocreate); //source file must be there
```

这个说明调用 `ifstream` 的构造函数(用于处理输入文件流的类)，以建立一个称作 `source` 的流对象。在我们可以使用 `source` 之前，我们必须建立一个文件缓冲区，并将这个流和缓冲区与一个真实的物理文件相联系。这两项任务由 `ifstream` 中的成员函数 `open` 来完成。`open` 函数需要一个文件名串，以及一个或两个可选的用于定义方式和保护权限的参数。这里文件名由 `argv[1]` 给出，即在命令行给出的源文件。

上述说明的一个较简洁的可选方法是：

```
ifstream source(argv[1], ios::nocreate); //source file must be there
```

```
//This creates source and open the file as well
```

方式参数 `ios::nocreate` 告诉 `open`，如果命名的文件未找到则不建立这个文件。对 DCOPY，如果文件在磁盘上未找到，则 `open` 失败。以后你会看到，可以使用其它的方式参数。如果文件 `argv[1]` 由于某个原因不能被打开(通常是文件未找到)，`source` 的值被置为零(false)，以使 `!source` 测试为真，给出一个错误信息，然后退出。

实际上，通过检查 `stream state` 中设置的错误位，我们可以确定源文件不能被成功打开的可能原因。成员函数 `eof`，`fail` 和 `bad` 测试各种错误位，如果它们被设置则返回真。

另外 `rdstate` 以 `int` 形式返回错误状态, 然后你可以测试哪个位被设置。 `eof` (文件结束) 并不一定是一个错误, 但它必须被测试并采取相应的动作, 因为一个流不能访问超过它最后一个字符的位置。注意, 一旦一个流处于错误状态 (包括 `eof`), 不再进行进一步的 I/O 操作。函数 `clear` 用于清除某个或全部错误位, 以便在清除非致命错误以后允许你恢复操作。

回到 `DCOPY`, 如果源文件一切正常, 则我们试图打开具有 `ofstream` 对象的目的地文件 `dest`。对输出文件, 如果它不存在, 缺省状态是建立一个文件。如果它存在, 则它被删除并建立一个空文件。你可以通过增加向 `dest` 说明一个辅助参数 `mode` 来改变这种方式, 例如:

```
ofstream dest(argv[2], ios::app|ios::nocreate);
```

它将试图以添加方式打开 `dest`, 如果 `dest` 不存在则失败。在添加方式, 源文件中的数据将被加到 `dest` 的末端, 以前的内容不改变。其它在类 `ios` 枚举的方式标志是 `ate` (移动到文件尾); `in` (输入打开, 和 `fstream` 一起使用); `trunc` (如果文件存在, 删除文件中的内容); `noreplace` (如果文件存在, 则失败)。

一旦两个文件都被打开, 则真正的拷贝工作以和 C 类似的方式完成。考虑 `while` 循环中的布尔表达式测试:

```
(dest && source.get(ch))
```

我们已知道 `dest` 在一个错误发生之前测试为真。同样调用 `source.get(ch)` 在读错误或到达文件尾发生前测试也为真, 除了硬件错误, 循环从 `source` 读字符, 然后放入 `dest` 中, 直到文件结束导致 `source` 为假为止。

关于 `iostream` 库还有更多的文件 I/O 特征。并且 `iostream` 也可以帮助你进行内存格式化, 这时流是 RAM。象 `stringstream` 这样特殊的类提供对内存流的处理。

5.9 用户定义数据类型的 I/O

使用 C++ 流的一个现实好处是你可以很容易重载 `<<` 和 `>>` 以处理用户定义的数据类型的 I/O。你可能已说明简单的数据结构:

```
struct emp{
    char *name;
    int dept;
    long sales;
};
```

为重载 `<<` 以输出 `emp` 类型的对象, 你可以照下面来定义:

```
ostream& operator << (ostream& str, emp& e)
```

```
{
    str << setw(25) << e.name << ": Department " << setw(6) << e.dept <<
    << lab << "Sales $" << e.sales << "\n";
    return str;
}
```

注意运算符函数 `<<` 必须返回 `ostream&`, 一个对 `ostream` 的引用, 以便你可以象预定义的输出运算符那样将你的新的 `<<` 连接在一起。你现在可以象下面那样输出 `emp` 类型

对象。

```
#include <iostream.h>
#include <iomanip.h>          //don't forget this!
```

```
...
emp jones = ("S. Jones", 25, 1000);
cout << jones;
```

显示结果是:

S.Johnes:Department 25 Sales \$1000

你可以在<<定义中,使用tab控制符吗?这不是一个标准控制符——是用户可以定义的:

```
ostream& tab(ostream& str) {
    return str << '\t';
}
```

当然这很平常,但可以产生比较易读的代码。

对emp的输入过程可以通过重载>>类似完成。这留给读者作为一个练习。

5.10 现在干什么?

建议你的第一个C++开发工程是使用FIGURES模块(它已在磁盘上)并扩充它。点、圆和弧并非已足够使用了,为直线、矩形和正方形建立对象。当你感到有必要时,使用单个馅饼形状图形的一个链接表建立一个馅饼图表对象。

一个更巧妙的挑战是实现一个处理相对位置的类。相对位置是相对于某个基点的偏移,表示为正的或负的偏移。在相对坐标-17, 42的一个点是从基点向左17个像素,向下42个像素。相对位置对将图表有效地组成一个大型图形是必要的,因为多个图形组合以每个图形的绝对点为基础并非总能结合在一起。最好是在绝对点之外再定义一个RX和RY域,对象在屏幕上的最终位置是绝对点和相对值的和。

当你感到已适应于C++以后,就可以开始将它的概念用于日常程序设计任务中。取来你较有用的已有的实用程序,用C++思想重新思考它们。试着观察一下在和你的函数库混在一起的程序中的类——然后以类的形式重写这些函数。你会发现类库在将来的工程项目中更易于重用。你在程序设计努力上的投资很少会被浪费掉,你很少因为漏掉什么而必须重写一个类。如果这个类可用,就使用它。如果它缺少什么,就扩充它。但是如果它工作得很好,就没有理由放弃它。

5.11 小结

C++是对现代程序应用中的复杂性的一个直接回答,复杂性经常使得许多程序员在绝望中放弃了他们手头的工作。继承和封装是用于管理复杂性的一个很有用的手段。C++在软件结构上增加了一些合理的规则,和分类学图表一样,只是增加了规则,而没有增加限制。

在增加对已存在的代码的扩充性和重用性的保证以后,你不仅仅只是拥有了一个工具箱——你拥有了建立新工具的工具。

第六章 掌握 C++

为了给你一种 C++ 的概貌性的和如何用 C++ 完成任务的感性认识, 这章用很少的语言快速浏览一下大量的概念。只要有一台计算机, 你就可以使用它们; 你可以装入和运行每个程序(它们在 Examples 子目录中, 随带的还有所需的任何头文件和其它文件)。如果你想更深入地了解 C++, 尤其是它所支持的面向对象的程序设计概念, 可阅读第五章“C++ 要素”。你也可以参阅《程序员指南》中的第一章“Turbo C++ 语言标准”, 以详细了解 C++ 的语法和使用。

在本章中, 我们假定你熟悉 C 语言, 并且知道如何用 Turbo C++ 编译、连接和运行一个源程序。我们从简单的例子开始, 逐渐增加复杂性, 以便突出新概念。有理由认为这些例子不一定是完美无缺的(换句话说, 它们不检查内存失效等)。本章不是论述数据结构或专业程序设计技术的; 相反, 它是对一个复杂的语言的一般性介绍。

这章分成两部分。第一部分提供你可能已具有的 C 程序设计知识和习惯的 C++ 做法。第二部分简单介绍 C++ 的要点: 使用类和继承进行面向对象的程序设计。

6.1 更好的 C: 从 C 过渡

虽然了解 C 对学习 C++ 是有益的, 但有时这种知识也是有害的, 尤其是在不是特定的针对面向对象的程序设计方面的知识, 而且 C++ 解决问题的方法和 C 不同。由于这个原因, 本章说明你可以用 C 完成的许多事情如何用 C++ 来做: 向屏幕输出文本, 注释你的代码, 建立和使用常量, 使用流 I/O 和内部函数等。

6.1.1 程序 1

```
// ex1.cpp: A First Glance
// from Chapter 6 of Getting Started
#include <iostream.h>
main()
{
    cout << "Frankly, my dear...\n";
    cout << "C++ is a better C.\n";
}
```

输出:

Frankly, my dear...

C++ is a better C.

注意, 这个程序第一行中新注释语法。从遇到的第一个双斜线到这行行末的所有字符被看作注释, 当然你仍旧可以使用传统的 /*...*/ 风格的注释。以 .CPP 为扩展名的文件被假设为 C++ 文件(或使用命令行选择项 -P)。

第三行包含标准头文件 iostream.h, 它代替了许多 stdio.h 的作用。第六和第七行的 cout 是一个输出流, 用作向标准输出发送字符(和 C 中的 stdout 作用一样)。<< 运算符(读

作“输出到...”将它右边数据发送到左边的流中。这里的<<运算符区别于算术左移运算符，它们使用相同的符号(运算符和函数的这种多重用法在 C++ 中很普遍，称作重载)。

6.1.2 程序 2

```
// ex2.cpp: An interactive example
// from Chapter 6 of Getting Started
#include <iostream.h>
main()
{
    char name[16];
    int age;
    cout << "Enter your name: ";
    cin >> name;
    cout << "Enter your age: ";
    cin >> age;
    if (age < 21)
        cout << "You young whippersnapper, " << name << "!\n";
    else if (age < 40)
        cout << name << ", you're still in your prime!\n";
    else if (age < 60)
        cout << "You're over the hill, " << name << "!\n";
    else if (age < 80)
        cout << "I bow to your wisdom, " << name << "!\n";
    else
        cout << "Are you really " << age << ", " << name << "?\n";
}
```

输出:

Enter your name: Don

Enter your age: 40

You're over the hill, Don!

cin 是和标准输入相联系的输入流。它可以正确处理所有标准数据类型。你可能还记得在 C 中向 stdout 输出一个不带有换行字符的提示需要调用 flush(stdout)，这才能显示出提示。然而在 C++ 中，一旦使用 cin 它就清缓冲区，cout 是自动的(如果这是缺省设置，你可以关闭这种自动清除功能)。

6.1.3 程序 3

```
// ex3.cpp: Inline Functions
// from Chapter 6 of Getting Started
#include <iostream.h>
const float Pi = 3.1415926;
```

```

inline float area(const float r) {return Pi * r * r;}
main()
{
    float radius;
    cout << "Enter the radius of a circle: ";
    cin >> radius;
    cout << "The area is " << area(radius) << "\n";
}

```

输出:

Enter the radius of a circle: 3

The area is 28.274334

一个常量标识符作用就象一般变量一样(即它的作用域是定义它的分程序, 并也作类型检查), 只是它不能出现在赋值语句的左边(或任何需要左值的地方)。#define 用法在 C++ 几乎废弃掉了。

关键字 inline 告诉编译器一旦有可能就直接插入代码, 以避免函数调用的开销。在其他方面(作用域等), 一个内部函数和一般函数一样。这种用途被推荐来代替用 #define 定义的宏(当然除了你要依赖于预处理进行宏替换的一些巧妙之处)。这个特征针对简单的只有一行的函数。

对 C 所做的重要的变化是说明可出现在语句出现的任何地方。

6.1.4 程序 4

// ex4.cpp: Default arguments and Pass-by-reference

// from Chapter 6 of Getting Started

```

#include <iostream.h>
#include <ctype.h>
int get_word(char *, int &, int start = 0);
main()
{
    int word_len;
    char *s = " These words will be printed one-per-line ";
    int word_idx = get_word(s, word_len);           // line 13
    while (word_len > 0)
    {
        cout.write(s+word_idx, word_len);
        cout << "\n";
        //cout << form("%.*s\n", word_len, s+word_idx);
        word_idx = get_word(s, word_len, word_idx+word_len);
    }
}

```

```

int get_word(char *s, int& size, int start)
{
    // Skip initial whitespace
    for (int i = start; isspace(s[i]); ++i);
    int start_of_word = i;
    // Traverse word
    while (s[i] != '\0' && !isspace(s[i]))
        ++i;
    size = i - start_of_word;
    return start_of_word;
}

```

输出:

These
words
will
be
printed
one-per-line

第六行(空行也计算在内)中函数 `get_word` 的原型有两个特殊方面。第二个参数被说明为引用参数, 这表明将在调用程序中修改参数的值(这等价于 Pascal 中的 `var` 参数和 C 中通过指针来完成的修改)。借助于这种方法, 变量 `word_len` 在 `main` 中被更新, 并且我们也可以使用函数 `get_word` 返回另一个有用的值。

C++ 中一个令人满意的特征是缺省参数。第三个参数是缺省参数。这表明它可以被省略(如第 13 行), 在这种情况下自动传递值 0。注意缺省值只需定义在函数第一次提及的地方。只有函数的尾部参数可以提供缺省值。

6.2 对象支持

世界是由拥有属性和行为的事物组成。通过将一个结构扩充为既包含函数又包含数据成员, C++ 为此提供了一种模型。一个对象的完全一致性通过一种简单的语言构造来表达。面向对象支持的这种表示方法不仅仅是一种记法上的方便——它是一种思考的工具。

6.2.1 程序 5

假设我们想有一个联机字典, 字典由词的定义组成, 我们首先模拟定义表示方法。

```

// def.h: A word definition class
// from Chapter 6 of Getting Started
#include <string.h>

const int Maxmeans = 5;

class Definition

```



```

{
    char *word;                // Word being defined
    char *meanings[Maxmeans];  // Various meanings of this word
    int nmeanings;

public:
    void put_word(char *);
    char *get_word(char *s) {return strcpy(s,word);} // line 15
    void add_meaning(char *);
    char *get_meaning(int, char *);
};

```

注: 你需要将 DEF.CPP 编译成 OBJ 文件, 然后将 EX6.CPP 或 EX7.CPP 与它连接在一起(或装入 EX5.PRJ)。你也可能需要在调试信息为检查状态下编译它以便跟踪和观察程序流。在传统 C 风格中, 我们将定义放在一个包含文件中。关键字 `class` 引入对象描述。在缺省时, 一个类的成员是私有的(虽然你也可以明显地使用关键字 `private`), 所以在这种情况下, 第 9 行到第 11 行的域只能由这个类的函数访问(在 C++ 中, 类函数称作成员函数)。为使这些函数可以用作用户接口, 在它们前面使用关键字 `public`。注意 `inline` 关键字在类定义(第 15 行)中不再需要。

实现通常保存在一个独立的文件中:

```

// def.cpp: Implementation of the Definition class
// from Chapter 6 of Getting Started
#include <string.h>
#include "def.h"

void Definition::put_word(char *s)
{
    word = new char[strlen(s)+1];
    strcpy(word,s);
    nmeanings = 0;
}

void Definition::add_meaning(char *s)
{
    if (nmeanings < Maxmeans)
    {
        meanings[nmeanings] = new char[strlen(s)+1];
        strcpy(meanings[nmeanings++],s);
    }
}

```

```

char * Definition::get_meaning(int level, char *s)
{
    if (0 <= level && level < nmeanings)
        return strcpy(s, meanings[level]);
    else
        return 0; // line 27
}

```

作用域冲突解决运算符(::)告诉编译器我们为 Definition 类(将类名的第一个字符写成大写是一种好经验,这样可以避免和库中的函数相冲突)定义成员函数。第 8 行的关键字 new 是对动态内存分配函数 malloc 的一种替代。在 C++ 中约定对指针使用 0 而不是 NULL。虽然我们在这里没有这样做,但仍提醒应检查一下 new 是否返回一个非零值。

```

// ex5.cpp: Using the Definition class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "def.h"
main()
{
    Definition d; // Declare a Definition object
    char s[81];
    // Assign the meanings
    d.put_word("class");
    d.add_meaning("a body of students meeting together to \
study the same subject");
    d.add_meaning("a group sharing the same economic status");
    d.add_meaning("a group, set or kind sharing the same attributes");
    // Print them
    cout << d.get_word(s) << "\n\n";
    for (int i = 0; d.get_meaning(i,s) != 0; ++i)
        cout << (i+1) << ": " << s << "\n";
}

```

输出:

class:

- 1: a body of students meeting together to study the same subject
- 2: a group sharing the same economic status
- 3: a group, set, or kind sharing the same attributes

6.2.2 程序 6

我们现在可以定义一个 dictionary 字典为 definition 的一个集合。

```

// dicton.h: The Dictionary class

```

```
// from Chapter 6 of Getting Started
```

```
#include "def.h"
```

```
const int Maxwords = 100;
```

```
class Dictionary
```

```
{
```

```
    Definition *words;        // An array of definitions; line 9
```

```
    int nwords;
```

```
    int find_word(char *);    // line 12
```

```
public:
```

```
    // The constructor is on the next line
```

```
    Dictionary(int n = Maxwords) {nwords = 0; words = new Definition[n];}
```

```
    ~Dictionary() {delete words;};    // This is the destructor
```

```
    void add_def(char *s, char **def);
```

```
    int get_def(char *, char **);
```

在第 12 行的程序 `find_word` 只由 `Dictionary` 类内部使用，所以是私有的。和类具有相同名字的函数称作构造函数(第 16 行)，当一个对象被说明时，它被调用，用于初始化，这里我们为一个 `definition` 数组动态分配空间。当一个对象退出它的作用域时，析构函数(第 17 行)被调用(在这种情况下，`delete` 运算符将释放以前由构造函数分配的内存)。对于一个成员对象数组(第 9 行)，被包含的类必须有一个不带参数的构造函数或根本就没有构造函数(`Definition` 类没有构造函数)。

```
// diction.cpp: Implementation of the Dictionary class
```

```
// from Chapter 6 of Getting Started
```

```
#include "diction.h"
```

```
int Dictionary::find_word(char *s)
```

```
{
```

```
    char word[81];
```

```
    for (int i = 0; i < nwords; ++i)
```

```
        if (strcmp(words[i].get_word(),s) == 0)
```

```
            return i;
```

```
    return -1;
```

```
}
```

```

void Dictionary::add_def(char *word, char **def)
{
    if (nwords < Maxwords)
    {
        words[nwords].put_word(word);
        while (*def != 0)
            words[nwords].add_meaning(*def++);
        ++nwords;
    }
}

int Dictionary::get_def(char *word, char **def)
{
    char meaning[81];
    int nw = 0;
    int word_idx = find_word(word);
    if (word_idx >= 0)
    {
        while (words[word_idx].get_meaning(nw,meaning) != 0)
        {
            def[nw] = new char[strlen(meaning)+1];
            strcpy(def[nw++],meaning);
        }
        def[nw] = 0;
    }

    return nw;
}

```

现在我们无须参照 Definition 类就可以使用 Dictionary 类(输出和前面的例子相同)。

```

// ex6.cpp: Using the Dictionary class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "diction.h"

main()
{
    Dictionary d(5);
    char *word = "class";
    char *indef[4] =
        {"a body of students meeting together to study the same",

```

```

        "subject a group sharing the same economic status",
        "a group, set or kind sharing the same attributes",
        0);
char *outdef[4];

d.add_def(word,ndef);
cout << word << ":\n\n";
int ndef = d.get_def(word,outdef);
for (int i = 0; i < ndef; ++i)
    cout << (i+1) << ": " << outdef[i] << "\n";
}

```

在 Dictionary 的实现中，我们特别地需要访问 Definition 的成员函数。有时希望允许某个函数或整个类访问另一个类的私有成员，我们可以将 Dictionary 类说明为 Definition 类的一个友元(第 18 行)：

```

// def2.h: A word definition class
// from Chapter 6 of Getting Started
#include <string.h>

const int Maxmeans = 5;

class Definition
{
    char *word;                // Word being defined
    char *meanings[Maxmeans];  // Various meanings of this word
    int nmeanings;

public:
    void put_word(char *);
    char *get_word(char *s) {return strcpy(s,word);};
    void add_meaning(char *);
    char *get_meaning(int, char *);
    friend class Dictionary;    // line 18
};

```

find_word 的实现则可以直接访问 Definition 成员(下面代码中的第 5 行)：

```

int Dictionary::find_word(char *s)
{
    char word[81];
    for (int i = 0; i < nwords; ++i)
        if (strcmp(words[i].word,s) == 0)

```

```
return i;
```

```
return -1;
```

```
}
```

8.2.3 程序 7

面向对象程序设计的一个主要特征是继承。一个新类可以继承一个已存在的类的数据和成员函数(新类称作是从基类派生的)。在这个程序中,我们定义 List 是一个处理整数链表的基类,然后派生 Stack 是一个处理栈的类(栈是一种特殊的链表)。首先我们建立头文件:

```
// list.h: A Integer List Class
```

```
// from Chapter 6 of Getting Started
```

```
const int Max_elem = 10;
```

```
class List
```

```
{
```

```
    int *list;        // An array of integers
```

```
    int nmax;         // The dimension of the array
```

```
    int nelem;        // The number of elements
```

```
public:
```

```
    List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
```

```
    ~List() {delete list;};
```

```
    int put_elem(int, int);
```

```
    int get_elem(int&, int);
```

```
    void setn(int n) {nelem = n;};
```

```
    int getn() {return nelem;};
```

```
    void incn() {if (nelem < nmax) ++nelem;};
```

```
    int getmax() {return nmax;};
```

```
    void print();
```

```
};
```

然后我们建立源代码:

```
// list.cpp: Implementation of the List Class
```

```
// from Chapter 6 of Getting Started
```

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

```
#include "list.h"
```

```
int List::put_elem(int elem, int pos)
```

```
{
```

```
    if (0 <= pos && pos < nmax)
```

```

    {
        list[pos] = elem;    // Put an element into the list
        return 0;
    }
    else
        return -1;          // Non-zero means error
}

```

```

int List::get_elem(int& elem, int pos)

```

```

{
    if (0 <= pos && pos < nmax)
    {
        elem = list[pos];    // Retrieve a list element
        return 0;
    }
    else
        return -1;          // non-zero means error
}

```

```

void List::print()

```

```

{
    for (int i = 0; i < nelem; ++i)
        cout << list[i] << "\n";
}

```

最后我们使用这个新类:

```

// ex7.cpp: Using the List class
// from Chapter 6 of Getting Started
#include "list.h"

```

```

main()

```

```

{

```

```

    // ex7.cpp: Using the List class
    // from Chapter 6 of Getting Started
    return -1;

```

```

    List l(5);

```

```

    int i = 0;

```

```

}

```

输出:

1
2
3
4
5

6.2.4 程序 8

```
// stack2.h: A Stack class derived from the List class
// from Chapter 6 of Getting Started
#include "list2.h"
```

```
class Stack : public List           // line 5
{
    int top;

public:
    Stack() {top = 0;};
    Stack(int n) : List(n) {top = 0;}; // line 11
    int push(int elem);
    int pop(int& elem);
    void print();
};
```

为定义一个派生类,必须有一个基类,所以我们包含它的头文件(第3行)。第5行告诉编译器,Stack类是从List类派生。关键字public说明List的公有成员在Stack中也是公有的(这也是通常所必须的)。因List类有一个带有一个参数的构造函数,Stack的构造函数直接调用List的构造函数(第11行)。基类的构造函数在导出类的构造函数之前执行。

```
// stack.cpp: Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack.h"
```

```
int Stack::push(int elem)
{
    int m = getmax();
    if (top < m)
    {
        put_elem(elem,top++);
        return 0;
    }
}
```



```

        else
            return -1;
    }

int Stack::pop(int& elem)
{
    if (top > 0)
    {
        get_elem(elem,--top);
        return 0;
    }
    else
        return -1;
}

void Stack::print()
{
    int elem;

    for (int i = top-1; i >= 0; --i)
    { // Print in LIFO order
        get_elem(elem,i);
        cout << elem << "\n";
    }
}

```

注意 List 类的公有成员函数可以直接被使用，因为一个 Stack 是一个 List。然而一个 Stack 对象的 List 部分的私有成员不能被直接引用。

```

// ex8.cpp: Using the Stack Class
// from Chapter 6 of Getting Started
#include "stack.h"

main()
{
    Stack s(5);
    int i = 0;

    // Insert the numbers 1 through 5
    while (s.push(i+1) == 0)
        ++i;
}

```

```

    a.print();
}
输出:
5
4
3
2
1

```

6.2.5 程序 9

有时允许一个派生类直接访问一个基类的某些私有数据成员是方便的。这种数据成员称作是保护的。

```

// list2.h: A Integer List Class
// from Chapter 6 of Getting Started
const int Max_elem = 10;

class List
{
protected:    // The protected keyword gives subclasses
               // direct access to inherited members
    int *list;    // An array of integers
    int nmax;    // The dimension of the array
    int nelem;    // The number of elements

public:
    List(int n = Max_elem) {list = new int[n]; nmax = n; nelem = 0;};
    ~List() {delete list;};
    int put_elem(int, int);
    int get_elem(int&, int);
    void setn(int n) {nelem = n;};
    int getn() {return nelem;};
    void Incn() {if (nelem < nmax) ++nelem;};
    int getmax() {return nmax;};
    virtual void print();           // line 22
};

```

现在在 Stack 的实现中我们可以用对 List 的数据的直接引用代替对 List 的成员函数的调用。

```

// stack.cpp: Implementation of the Stack class
// from Chapter 6 of Getting Started
#include <iostream.h>

```

```
#include "stack.h"
```

```
int Stack::push(int elem)
```

```
{
    int m = getmax();
    if (top < m)
    {
        put_elem(elem,top++);
        return 0;
    }
    else
        return -1;
}
```

```
int Stack::pop(int& elem)
```

```
{
    if (top > 0)
    {
        get_elem(elem,--top);
        return 0;
    }
    else
        return -1;
}
```

```
void Stack::print()
```

```
{
    int elem;

    for (int i = top-1; i >= 0; --i)
    { // Print in LIFO order
        get_elem(elem,i);
        cout << elem << "\n";
    }
}
```

然后我们可以测试这个程序:

```
// ex9.cpp: Using the print() virtual function
// from Chapter 6 of Getting Started
#include <iostream.h>
#include "stack2.h"
```

```

main()
{
    Stack s(5);
    List l, *lp;
    int i = 0;

    // Insert the numbers 1 through 5 into the stack
    while (s.push(l+1) == 0)
        ++l;

    // Put a couple of numbers into the list
    l.put_elem(1,0);
    l.put_elem(2,1);
    l.setn(2);

    cout << "Stack:\n";
    lp = &s;           // line 22
    lp->print();        // Invoke the Stack print() method; line 23

    cout << "\nList:\n";
    lp = &l;
    lp->print();        // Invoke the List print() method; line 27
}

```

输出:

Stack:

5
4
3
2
1

List:

1
2

上面的例子说明了多态性(也称作“迟后联编”或“动态联编”，在C++中是通过虚函数来完成的)，这说明一个对象的类型只在运行时才能确定下来。通过定义 `print` 成员函数是 `virtual` (见 `List2.h` 的第 22 行)，我们可以通过指向基类的一个指针调用不同的 `print` 成员函数。在上面的第 22 行，`lp` 指向一个 `stack` 对象(记住：一个 `Stack` 是一个 `List`)，所以 `Stack` 的 `Print` 在第 23 行被调用，而 `List` 的 `Print` 成员函数在第 27 行被执行。

6.3 小结

还有比本章所介绍的更多有关 C++ 的内容(例如多重继承)。正如在开头所说的,本章只打算给你有关 C++ 的一种直观和表面上的感性认识,以说明它和 C 怎样不同,以及演示怎样使用 C++ 的大多数基本特征。有关 C++ 基本概念的更多信息,参阅或复习第五章“C++ 要素”和《程序员指南》的第一章“Turbo C++ 语言标准”,以获得对 C++ 的更进一步了解。

第七章 新的 IDE 调试

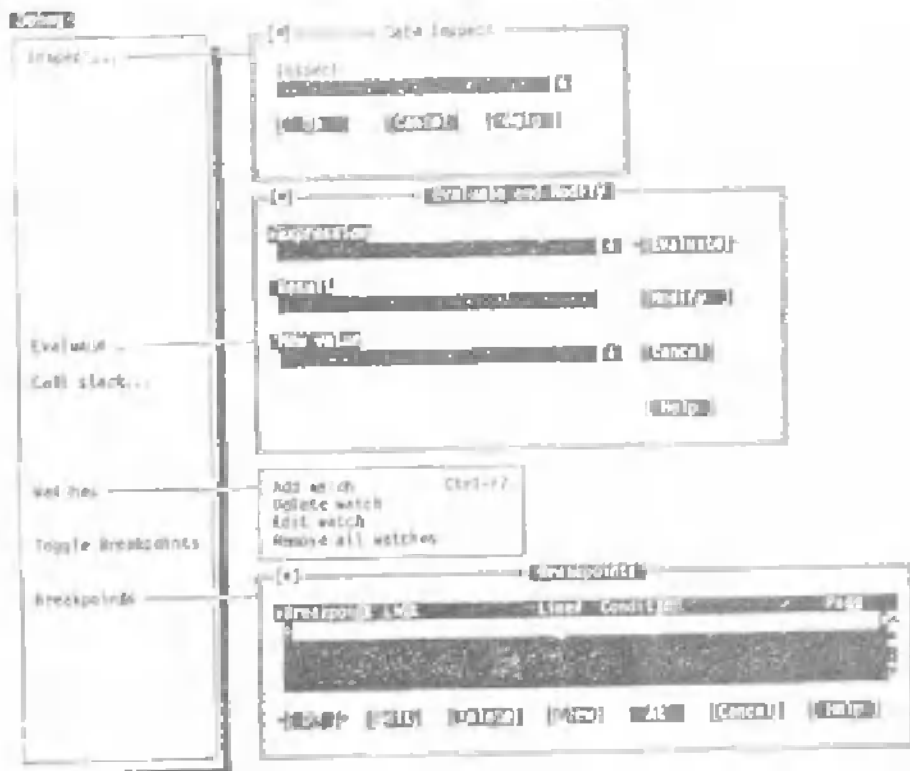
在第四章中，你已经学习了 C 的主要元素，并通过大量的例子，知道如何将这些元素组合起来产生工作程序。用这些知识，也许你已经能够写一些简短的程序了。如果还不能，现在开始也不晚。因为除了自己动手编写程序外，没有别的方法能检查和加深对已讨论概念的理解。

当然，编写程序意味着要处理你的错误。大多数有经验的程序员都承认跟踪程序中的逻辑问题——称为 bug(程序错误)——是程序开发中的一个重要部分。调试，或发现并定位程序中的错误所花费的时间常常比编写程序本身所花的时间长。

Turbo C++ 带有一个源程序级的集成调试程序，它为你在集成环境中的调试提供了方便和速度，同时，它还提供了一个具有许多性能的独立调试程序。

源程序级的含义是你可以跟踪实际程序代码——希望的话，还可以包括你在程序中调用的每个函数。通过设置断点，你可以控制在进行条件检查前运行多少程序，之后，通过用光标选择变量或在 Evaluate 区域中键入变量名，可得到该变量的当前值。你也可以设置 watch 来监视一个或多个变量，在程序运行时显示其值的变化。所有这些的实现都不需很复杂的思考，因为你使用的是编写程序时已使用的同样的变量、表达式和运算符。

键入 TC 启动 Turbo C++ (如果还未运行的话)，然后花几分钟时间检查 Debug 菜单。用光标依次选择菜单上的每一项，并在每一次按 F1，阅读相应的求助正文。这些菜单选择项都是调试工具，在本章后面的示例程序中你将学会如何使用。



7.1 调试与程序开发

当本章集中讨论调试时，很重要的一点是不要把调试和设计、编写程序隔离开来对待。虽然 Turbo C++ 的集成调试程序尽可能地去发现并定位程序中的错误，但如果你设计程序的方法好，同样会使调试变得更容易些。

考虑一个老式的电子管电视机。由于其部件都用导线连入一大堆电子管、电阻和电容等中，因此修理起来很困难。通常很难发现是哪一部分出了故障。而当你找到要替换的那一部分时，又很难从一大堆导线中把那一部分取出来。

自从晶体管和集成电路引入使用后，人们开始设计另一种电视。每块电路放在它自己的硅片上，作为一个模块，便于访问和测试。要替换故障模块时，很容易就能完成替换。

在 C 中，与这些插入模块等价的就是函数。你将设计、实现并测试本章中的示例程序，每次一个函数。通常，人们常常试图编写一个完整的程序，尤其是在这个程序较小时，然后才开始调试。如老式电视一样，它给问题带来一些困难。下面，我们来看这是为什么，假设 main 调用函数 b，而函数 b 中又依次调用函数 c 和 d。如果你在开发每个函数时，没有对其进行测试，则将难以确定函数 d 中出现的错误是由于其间代码出错，还是 b 或 c 中的代码出错。当然，这将比调试一个函数困难得多。同时，看起来不相关的函数 a 也可能修改函数 b 需要用来传递给 c 的全局变量。可见递增的程序开发——每次开发和测试程序中的一部分——可以节省你相当多的时间并使你免遭失败。

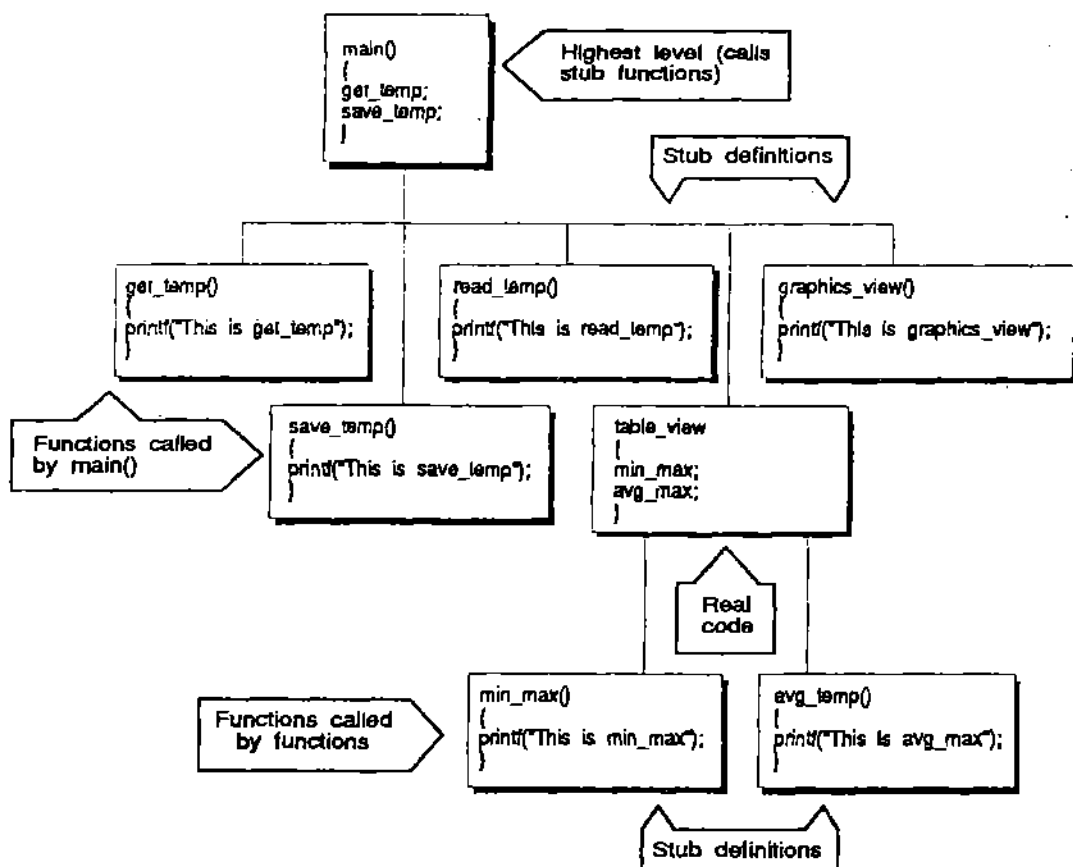


图 7.1 程序开发流程图

7.2 设计示例程序: PLOTEMP.C

本章的示例程序读取收集的温度,并以表格或图形的方式显示这些数据。它显示了 Turbo C++ 各类性能的一个集合:控制板和 I/O 文件,将数组传递给一个函数和一些图形制表函数。之后,如果你愿意,可以修改 PLOJEMP.C,以便处理另一组数据,并给出一组不同的报告和图表。

设计程序最好的办法之一是给出函数原型,显示该函数与用户的接口——要求什么样的信息、对命令的反应和显示信息。在看程序编码前,让我们先从用户角度来考虑该程序应该象什么。

当你运行 PLOTEMP.C 时,它显示菜单:

```
Temperature Plotting Program Menu
E - Enter temperatures for scratchpad
S - Store scratchpad to disk
R - Read disk file to scratchpad
T - Table view of current data
G - Graph view of current data
X - Exit the program
Press one of the above keys:
```

如果按 E,将提示你输入一组温度值。该程序设置为处理 8 个输入值,但你可以很容易地修改该值,只需改变临近程序代码开始的一行:

```
#define READINGS 8
```

即可。

数据输入方式如下:

```
Enter temperatures, one at a time.
Enter reading # 1: 52
Enter reading # 2: 55
Enter reading # 3: 62
Enter reading # 4: 65
Enter reading # 5: 73
Enter reading # 6: 76
Enter reading # 7: 68
Enter reading # 8: 61
```

完成该工作或别的菜单选择(除 X 退出外),菜单将再次被显示。

选取 S 将当前内存中的数据集合存入一磁盘文件(将提示你输入文件名)。选择 R 将从你指定的磁盘文件中读取一组数据并放入该程序的“scratchpad”数组中。

一旦你在 scratchpad 中放入了数据(无论是从键盘输入还是从磁盘中读取),你便可以通过选取 T(即 Table view)来显示这些数据的一个综合表格:

Reading	Temperature (F)
1	52
2	55
3	62
4	65
5	73
6	76
7	68
8	61

Minimum temperature: 52
Maximum temperature: 76
Average temperature: 64.000000

作为另一选择,你也可选取图表方式来显示当前数据,如下图所示:

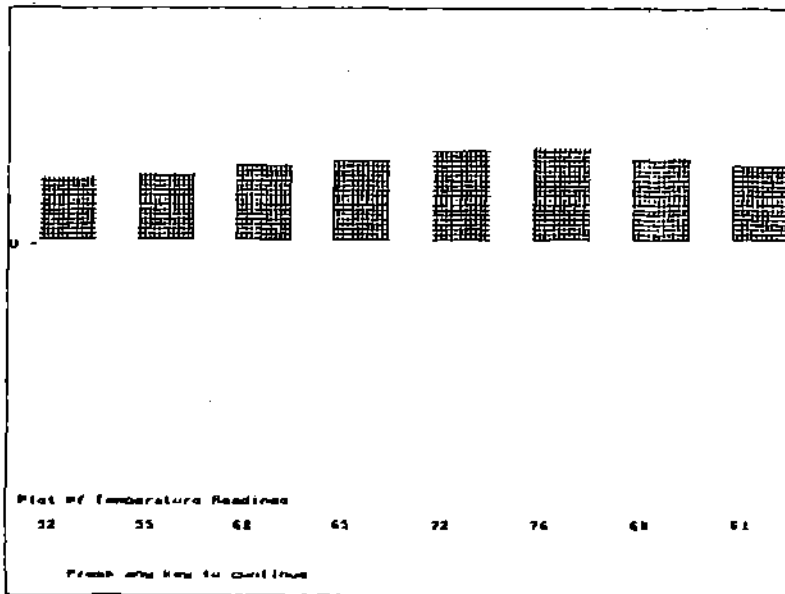


图 7.2 温度数据的图形显示

这里存在一个问题：即被你放在一起的程序不能如愿地完整运行。因此必须用调试程序来查找并定位程序中的错误。

7.3 编写程序原型

决定了该程序将作什么以后，你就可以决定该程序需要的全局数据和一些别的定义，并且编写 main 函数。

```
/* PLOTTEMP1.C--Example from Chapter 7 of Getting Started */
/* This program creates a table and a bar chart plot from a
   set of temperature readings */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
void read_temps(void);
/* Global defines */
#define TRUE 1
```

```

#define READINGS 8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("\tE - Enter temperatures for scratchpad\n");
        printf("\tS - Store scratchpad to disk\n");
        printf("\tR - Read disk file to scratchpad\n");
        printf("\tT - Table view of current data\n");
        printf("\tG - Graph view of current data\n");
        printf("\tX - Exit the program\n");
        printf("\nPress one of the above keys: ")
        switch (toupper(getche()))
        {
            case 'E': get_temps(); break;
            case 'S': save_temps(); break;
            case 'R': read_temps(); break;
            case 'T': table_view(); break;
            case 'G': graph_view();
            case 'X': exit(0);
        }
    }
}

/* Function definitions */
void get_temps(void)
{
    printf("\nExecuting get_temps().\n");
}
void table_view(void)
{
    printf("\nExecuting table_view().\n");
}
void min_max(void)
{
    printf("\nExecuting min_max().\n");
}
void avg_temp(void)

```

```

{
    printf("\nExecuting avg_temp().\n");
}
void graph_view(void)
{
    printf("\nExecuting graph_view().\n");
}
void save_temps(void)
{
    printf("\nExecuting save_temps().\n");
}
void read_temps(void)
{
    printf("\nExecuting read_temps().\n");
}

```

关于程序原型应注意以下几点:

- 全程的 `#define` 和数据结构(数组 `temps`)。
- 函数 `main` 提供了顶层菜单。
- 别的函数说明为返回 `void` 类型, 和带 `void` 类型变量。
- 每个函数包括一条 `printf` 语句, 以便标明它在执行。
- 该程序已经很完整, 可以运行。

为什么用 `void` 说明这些函数呢? 它能使你在顶层运行程序, 并检查执行流程。如果你给出这些函数的完整的 ANSI 原型, 并带上不同数据类型的变量, 就不得不重新开始定义函数并编写代码来使用传递到函数的参数值。否则, 你将收到编译程序关于这些参数未被使用的警告信息。一个好的程序开发原则是“一次做一步, 一次测试一步。”基于这点, 你可以试验该程序的顶层结构是否正确。

7.4 使用集成调试程序

通过选取 `Compile/build All` 编译 `PLOTEMP1.C`。编译程序在显示如下错误信息后停止运行。

Error C:\TC\EXAMPLES\PLOTEMP1.C 43:

statement missing; in function main

Turbo C++ 发现一语法错误。通常你在实际运行程序前就应该检查一遍看有无语法错误, 更不用说开始调试了。幸而, 语法错误通常易于定位。按空格键: 此时, 错误条光标在显示菜单的 `switch` 语句的第一行, 掉一个分号的说明通常是指上一条语句(这里, 是指前面的一组 `printf` 语句中的最后一条)。正如你所回想到的, 你可以按 `Enter` 或 `F6` 回到编译窗口来较正错误。如果有多个语法错误要处理, 按 `F6` 回到信息窗口, 然后使用 `Alt-F8` (或 `↓` 键)移到下一个列出的错误上。

修正了语法错误后, 再次编译该程序。这次编译程序和连接程序的运行无错误信息。表示该程序已无语法错误。

现在选取 **Run/Run**, 运行 **PLOTEMP1.EXE**。你将看到该程序的菜单, 试着按菜单上列出的各个键, 检查该程序。再按一个根本不在菜单中的字母, 会发生什么情况? 还能正常工作吗?

你可能已看到, 对于菜单的每一个选择, 都显示相应函数中的描述行。(记住那些列出的子函数中的 **printf** 语句)。通常, 再次显示该菜单。如果你按(或 **x**), 则程序结束, 返回到 **Turbo C++**。那么, 如果你按 **G**(或 **g**)会发生什么情况呢? 同样也返回到 **Turbo C++**。这是一不正确的——因为你希望此时再得到该菜单。这说明程序中某个地方存在错误。

7.5 跟踪程序的流程

通过使用 **Run** 菜单上的选择项, 和观察运行光标条, 你可以观察到程序中语句的执行顺序, 并能控制跟踪的细节。

选取 **Run/Trace Into**(或按 **F7**), 则调试程序上卷到编辑窗口中 **main** 函数的开始处, 并用高亮度光标条置亮它。该高亮度光标条称为运行光标条, 标明执行位置, 并指出下一条要执行的语句。

1. 跟踪高层的运行

为跟踪程序的高层流程, 选取 **Run/step Over**(或按 **F8**), 则下一行包含的代码被执行(跳过注释行)。当你继续按 **F8** 时, 运行光标条移过一系列的用于显示菜单的 **printf** 语句, 且屏幕出现闪烁。这是因为每当调试程序执行一条在屏幕上显示信息的语句或执行一个函数调用时, 它都要立即转到用户屏幕(**User screen**)。如果你不是在 **Turbo C++** 集成环境中执行的话, 该屏幕显示程序产生的信息(但是屏幕转换进行得非常快, 你无法看清输出)。为了看用户屏幕中的信息, 选取 **window/User Screen**, 或按 **Alt-F5**。依据你执行 **printf** 语句的多少, 你将看到部分或全部 **PLOTEMP** 的菜单。按任意键回到调试程序。

继续按 **F8**, 直到运行到 **switch** 语句的第一行。该行包含一个对 **getche** 函数的调用, 它要求用户输入一个字符。每当程序要求用户输入时, **Turbo C++** 都转到用户屏幕。由于你想观察选择(**G**)图形显示时会发生什么情况, 再按 **F8**, 然后按 **G**。当你给出要求的输入后, 显示返回到调试程序, 并且运行光标移动到下一行语句:

```
Case'G':graph_view();
```

到此, 一切正常。但是, 你再按一次 **F8**, 走到下一步, 下一条语句是 **exit**, 表示你再运行一步将返回编辑程序。按 **F8** 键试验完成。至此, 你也许已注意到在上一行中需要一个 **break** 语句, 修改该行为:

```
Case'G': graph_view(); break;
```

现在来看修改后的工作。重新启动程序, 从头开始每次执行一步, 但这太冗长乏味了。为此, 可以选取 **Run/Go to Cursor**(或按 **F4**; 保证光标在正确的行上)。重建程序(在“**OK to rebuild**”提示符下回答 **yes**); 则程序运行到需要用户输入的一行, 输入 **G** 作为 **PLOTEMP** 菜单的回答后, 程序继续运行到你定位的那一行。改为单步运行, 注意到这次 **break** 语句被执行, 然后执行位置回到 **while** 循环的顶部: 再次显示 **PLOTEMP** 菜单, 操作正确。

2. 跟踪被调函数

当你使用 **Run/Step Over** 时, 仅仅是在程序的高层作单步执行。正如你所见到的, 运行光标条停留在 **main** 函数, 并单步通过 **printf** 语句和 **switch** 语句中的各个 **case**。然而,

你常常需要跟踪被 main 函数调用的函数,有时还有被这些函数调用的函数。跟踪函数调用,选取 Run/Traace into 或按 F7。

现在来试跟踪该程序,在 PLOTEMP 菜单中按 E。这时运行光标条进入 switch 语句中相应的 case,然后进入函数 get_temps 的定义中。单步执行完该函数(这里,仅有一条 printf 语句),执行返回到 main 中大 while 循环的底部,然后返回到再次显示菜单的顶部。

7.6 继续程序的开发

在本章剩余的部分中,将每次向 PLOTEMP 中加入一个函数并通过运行该程序来测试它。代替使你输入代码的全部工作,我们提供了 PLOTEMP 的改进版本,你能装载完成每一步。如果你要调试你的程序之一,按如下步骤进行:

1. 必要的话,用完整的函数代换函数原型。
2. 用需要用来执行任务的实际代码代换已有的函数定义。
3. 通过再次运行该程序测试新函数,注意在菜单中选取相应的项。
4. 定位出现的任一错误,测验到没有错误为止。
5. 以同样的方法完成下一个函数,直到完成一个内容俱全的程序。

为最快且最佳地取得结果,应分别编写、编译、运行和调试每个函数。不要仅在当前工作程序无表面错误时就开始开发下一个函数,该方法不能消除所有错误,因为一些“隐藏”错误仍继续潜伏着,并会产生一些不能预见的情况组合。但该改进方法能使这些不希望破坏机会减少到最小。

从 get_temps 函数开始,它从键盘输入读取一组温度。由于它不带参数,也不直接返回任何值,因此,当前函数原型说明为:

```
void get_temps(void);
```

和前面一样,不需修改。(该程序的完整产品版本可能带有这条语句,且每个函数返回一个值,因此可以作些标志,在此我们忽略它,以减小程序大小和复杂性)。

找到已有的 get_temps 的定义。目前,它是执行时仅仅打印一条信息的存根定义。用下面程序代替已有的编码(该代码已包含修改信息,存入 PLOTEMP2.C):

```
/* PLOTEMP2.C—Example from Chapter 7 of Getting Started */
/* This program creates a table and a bar chart plot from a
   set of temperature readings */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
```

```

void read_temps(void);
/* Global defines */
#define TRUE      1
#define READINGS  8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("\tE - Enter temperatures for scratchpad\n");
        printf("\tS - Store scratchpad to disk\n");
        printf("\tR - Read disk file to scratchpad\n");
        printf("\tT - Table view of current data\n");
        printf("\tG - Graph view of current data\n");
        printf("\tX - Exit the program\n");
        printf("\nPress one of the above keys: ");
        switch (toupper(getche()))
        {
            case 'E': get_temps(); break;
            case 'S': save_temps(); break;
            case 'R': read_temps(); break;
            case 'T': table_view(); break;
            case 'G': graph_view(); break;
            case 'X': exit(0);
        }
    }
}

/* Function definitions */
void get_temps(void)
{
    char inbuf[130];
    int reading;
    printf("\nEnter temperatures, one at a time.\n");
    for (reading = 0; reading < READINGS; reading++)
    {
        printf("\nEnter reading # %d: ", reading + 1);
        gets(inbuf);
        sscanf(inbuf, "%d", temps[reading]);
    }
}

```

```

        /* Show what was read */
        printf("\nRead temps[%d] = %d", reading, temps[reading]);
    }
}

void table_view(void)
{
    printf("\nExecuting table_view().\n");
}

void min_max(void)
{
    printf("\nExecuting min_max().\n");
}

void avg_temp(void)
{
    printf("\nExecuting avg_temp().\n");
}

void graph_view(void)
{
    printf("\nExecuting graph_view().\n");
}

void save_temps(void)
{
    printf("\nExecuting save_temps().\n");
}

void read_temps(void)
{
    printf("\nExecuting read_temps().\n");
}

```

7.7 设置断点

记住，C 中数组从第 0 个元素开始：第一次读处理第 0 个元素，第二次对应于第 1 个元素，等等。

如果 `get_temps` 工作正确，则 `for` 循环在每次读时给出提示，然后使用 `gets` 以字符串形式读取一个值。接下来，用 `sscanf` 将其存入数组 `temps` 的相应元素中，并用另一个 `printf` 语句显示存入数组的值。该 `printf` 语句仅是一个临时方法：用于让我们看见输入的存储。该函数调试通过后应将其删除。

运行 `PLOTEMP1`，并在 `PLOTEMP` 菜单中按 E 以便运行 `get_temps`。在你输入数据时，会出现如下情况：

```

Enter reading # 1: 40
Read temps[0] = 0
Enter reading # 2: 50
Read temps[1] = 0
Enter reading # 3: 55
Read temps[2] = 0
Enter reading # 4: 57
Read temps[3] = 0
Enter reading # 5: 61
Read temps[4] = 0
Enter reading # 6: 64
Read temps[5] = 0
Enter reading # 7: 65
Read temps[6] = 0
Enter reading # 8: 60
Read temps[7] = 0

```

出了什么问题呢？不管你输入什么数据，`temps` 数组中相应元素总保持为 0。有可能是被输入的数据在放入数组时采用的方法不对。

显然，该代码需要进一步的测试。为了将程序运行到感兴趣的区域，设置一个断点。断点是程序中希望运行到此就停止的位置。将光标移到 `get_temps` 函数中循环的开始处：

```
for(reading=0; reading < READINGS; reading++)
```

然后选取 **Debug/Toggle Breakpoint** (或按 **Ctrl-F8**)，则设置该断点。于是带 `for` 的这行被置亮。任何时候，当程序执行到你已设置断点的一行时都被中断，并返回到调试程序。你将看到，它允许你使用别的调试命令来检查和修改变量及别的数据结构。当程序中有多个断点时，可以来检查和修改变量及别的数据结构。也可以选取 **Debug|Breakpoint** 和选择对话框中的 **View** 开关看到下一个断点。

断点设置将保持到你做以下操作之一时消除：

- 离开该集成环境。
- 用 **Ctrl-F8** 开关消去此断点。
- 使 **Debug|Breakpoints|Delete** 删除此断点。
- 删除设置了断点的行。
- 编辑包含断点的文件和不存就放弃该文件。

每当你修改一个错误或编辑当前文件，然后继续使用调试命令时，Turbo C++ 提问 “Source modified, rebuild?” 通常应在该提示下按 **Y**，以便程序被重新组建，以反映你所作的修改。如果按 **N**，则断点和运行光标条都出现在错误的位置上，因为源程序已不再和执行程序匹配了。

1. 用 **Ctrl-Break** 立即中断

也许你知道按 **Ctrl-Break** 可以中断许多正在运行的程序。同样对于 Turbo C++ 和集成调试程序也如此。然而，调试程序常常不会立即停止程序的执行，它要等到你的源程序中一行所对应的机器码被执行完后，然后将程序停在源程序中下一行开始处所对应的机器指令上。运行光标条出现在刚执行完的下一行上。

如果你确实需要立即中断，按两次 **Ctrl-Break**。当检测到第二次按键，调试程序立即终止程序，不再作任何输出或调用退出函数。(这和使用 `_exit` 函数一样)。这种“二次 Break”通常是不使用的，因为这时数据文件的内容是不可预知的，且调试程序也不知道接下来应

执行哪一行。通常仅仅是在程序“挂起”或陷入无限循环时才付第二个 Break。

再次运行程序，显示 PLOTEMP 菜单，按 E 程序运行进入 get_temps 函数，直到设置断点的行。用 F8 单步执行直到运行光标条移过 for 循环体中的语句。到循环的第 1 行。现在来看关键变量以便了解错误的起因。

7.8 检查数据

复杂的程序通常使用了许多种数据结构——数组、结构、联合、表等等。要了解你的程序中某一特定部分将出现什么情况，你需要知道这些数据结构的实际内容。Turbo C++ 提供了一个新的方便的工具，称为检测(inspectors)，它可让你揭开程序的任一部分，检查其内部工作情况。

1. Inspector 窗口

注：它只能检查所给的合法的 C 或 C++ 表达式。不包含 #define 中定义的符号或函数调用。

一个条目打开 Inspector 窗：将光标移到编辑窗口中该条目的位置，并按 Alt-F4(你也可以选取 Debug|Inspect)。尝试检查 PLOTEMP.C 的当前版本(当前应在编辑窗口中)中的 reading 变量。选取 Run|Trace Into，然后选取 Debug|Inspect。

所有 Inspector 窗口都用该项的名字开头。对于变量，下一行是该变量的地址，表示为段地址：偏移地址(被说明为 register 类型的变量，或那些被 Turbo C++ 优化后放入寄存器的变量，没有地址。因为它们被存在 CPU 中而不是 RAM 中。为此，你将看到显示单词 register)。再下一行描述该目的数据类型(例如：unsigned int)。

该条目的实际值被显示在数据类型的右边，Turbo C++ 自动选择合适的格式来显示被调数据的类型。对于非打印字符，用反斜杠(\)跟该字符的十六进制数来代替该字符值的显示。另一方面，一个整型变量可有十进制和十六进制值，但无字符表示。别的数字类型也同样处理。

2. 检查数组和字符串

对于一个数组，在窗口中分行显示每个元素。如果元素太多，在该窗口中不能显示元，可用箭头键滚动窗口。对于一个字符串，其中的字符表作为一个整体显示。图显示了 PLOTEMP.C 中数组 temps 在通过 get_temps 函数时被填满数据后的情况。



图 7.3 检查数组 temps

字符串的显示—数组一样(毕竟它们是相同的数据结构)。但是，字符串中的字符表示方式不是以单个元素的方式。

3. 检查结构和联合

对于结构和联合，显示各个元素的值。为了观察其操作，装入 SOLAR.C(见第 4 章)并

同样, 你也可检查更复杂的数据结构如数组、字符串和结构的值。如果, 想进一步了解数组 `temps` 的情况, 从头开始在求值区中输入 `temps`, 则显示:

```
{0, 0, 0, 0, 0, 0, 0, 0}
```

它表示当前存入数组 `temps` 中的整数值, 通过使用下标你也可以获得单个值: 例如, 指定 `temps[0]`, 便可取得第一个整数。或者, 你也可以打开一个 `Inspector` 窗口。

注意, 我们指的是表达式, 而不仅仅是它们的值(如变量)。回忆一下, 表达式是一些变量, 常量和运算符的组合, 并产生一个值, 例如, `vals1[index]+vals2[index]+1`。你可以显示任何表达式的值, 但要求:

- 不包含函数调用的值, 例如类似 `wqrt(2)+1` 的表达式。

- 不能用 `#define` 的值, 例如当前程序中的 `READINGS`。

作为练习, 在求值区中输入以下表达式, 计算其值:

```
reading+2
```

```
temps[reading+1]
```

1. 指定显示格式

作为可选, 你可在要显示的值得加入一个逗号和一个格式指示符。例如, 输入 `reading`, `h` 可看到 `reading` 当前值的十六进制数(你也可输入 `reading, x`)。缺省时, 整数以十进制形式显示, 且字符数组显示为字符串。

处理数组时指示符 `m` 很有用: 它可从指定地址开始显示内存内容。例如, `temps, m` 可得到从指定位置开始的内存内容(由于 `temps` 是一个数组, 其名字指向被存入数据的起始地址):

```
00 00 00 00 00 00 00
```

这表明数组 `temps` 的所有元素当前被设置为 0。所显示的元素个数依赖于该数组的长度。你也可将 `m` 与别的指示符组合使用:

```
temps, mh
```

以十六进制的格式显示内存转储。

另一个有用的指示符是 `P`, 它显示一指针变量, 给出有关所指内存区域的信息(例如: 中断向量表中 BIOS 数据区, 或用户程序的程序段前缀[PSP])。如果所指向的内存存在程序自己被分配的区域中, 则该段偏移地址处的变量名也被显示。(参见《程序员指南》第四章“存储模式、浮点数和覆盖”)。

2. 指定值的个数

当处理一个数组时, 你可以指定要显示的个数, 例如: `temps, 5` 指定数组 `temps` 的前 5 个元素。你可将计数与格式指示符组合使用, 例如: `temps[2], 3h` 指定以十六进制形式显示从数组 `temps` 的第三个元素开始的 3 个元素。

别的格式指示符和调试的细节见《用户手册》第一章“IDE 指南”(对于集成环境)。练习使用本章中的调试后, 我们建议再读那章中的有关部分。

3. 从光标所在位置拷贝

也许你早已注意到, 当你选取 `Debug|Evaluate` 时, 求值区显示光标所在位置的任何单词, 于是, 你便可以输入键盘值。例如, 如果你将光标移到表达式:

```
temps[reading]
```

的开始处, 则求值区显示 `temps`。当你按 `→` 时显示 `temps` 后的字符。这样便可以将完整的

表达式 `temps[reading]` 拷入求值域，然后和原来一样按 Enter，显示其值。

4. 指定在别的函数中的变量

现在，我们看的是函数 `get_temps` 中的变量 `reading` 和 `temps`。你也可以询问调试程序在别的函数中的静态变量的值，因为静态变量即使在其函数未被执行时，其值也是被保存的。你也可以查看调用你正在执行的函数的函数中的变量。但你不能查看通常在别的函数中说明的自动变量，因为当退出该函数后，这些自动变量便不再存在。被求值的表达式可通过编辑窗口中当前光标位置给出。

要指定当前函数之外的一个变量，你可将光标移入该函数体，或给出该函数名，圆点，变量名。

如果想检查的变量在另一个程序模块中，则必须首先给出该模块名例如：

```
module2.getvals.count
```

5. 修改值

正在你已知道怎样查看不同类型的变量和表达式，及以不同的形式显示其值，练习单步运行 `get_temps` 中的 for 循环，检查 `reading` 和 `temps[reading]` 的值，后者一直保持为 0。

在单步运行几次循环时，试着求表达式 `temps[reading]` 的值。调试程序显示其值为 0，与 `reading` 的值无关。但是该表达式表示什么呢？由于它被指定为 `sscanf` 函数的存储目标，所以它应是一个地址，于是，这意味着所有输入值被存储在地址 0 上。你可以通过使用指针格式 `temps[reading]`，p 验证这一点，发现其值仍为 0。

你需要使用地址运算符 & 使该表达式代表 `temps` 数组的地址。计算 `dtemps[reading]`，p。其结果形如：DS:1278 `temps+1`。实际显示的值依赖于系统结构和 `reading` 的当前值，但你可发现 `&temps[reading]` 指向数据段中的一个实际地址，其偏移值由变量 `temps` 给出。

将 `sscanf` 语句中的表达式 `temps[reading]` 改为 `&temps[reading]`。现在，如果你继续单步运行该函数，调试程序将问你是否 “rebuild the program”，按 Y 回答是。现在，再次单步运行 for 循环，并求 `temps[reading]` 的值，你将发现输入的值都被正确地存入数组了。

这是一次练习用调试程序修改值的好机会：求 `temps[reading]` 的当前值。

用 Tab 键在三个区域中移动 (Evaluate, Result 和 New Value 三个区)。一旦输入光标进入 New Value 区，输入一个值，如 66，并按 Enter。此时，如果在 Evaluate 区中输入 `temps[reading]`，则其新值 66 显示在 Result 区中。你可以修改任何表示一个简单数据元素的表达式，如简单变量、指针或一个数组元素。

用调试程序修改内部值对于临时定位一个错误和继续执行一段程序找下一个错误是很有用的。例如，你可以输入从 `temps[0]` 到 `temps[7]` 的新值，并设置 `reading` 为 8，跳出 for 循环，返回到程序的主菜单中。你也可以强制一个函数返回一个指定的值，或将该值传送给另一个函数。这可帮助你测试导致错误的非常情况，而不必在程序中加入临时的赋值语句。

现在 `get_temps` 函数已能正确工作。下一步来完成 `table_view` 函数，显示输入数据。用下面的 `table_view` 来替代原来的存根函数：

装入 PLOTTEMP3.C:

```
/* PLOTTEMP3.C--Example from Chapter 7 of Getting Started */
/* This program creates a table and a bar chart plot from a
   set of temperature readings */
```

```

#include <conio.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
/* Prototypes */
void get_temps(void);
void table_view(void);
void min_max(void);
void avg_temp(void);
void graph_view(void);
void save_temps(void);
void rend_temps(void);
/* Global defines */
#define TRUE      1
#define READINGS  8
/* Global data structures */
int temps[READINGS];
int main(void)
{
    char choice;
    while (TRUE)
    {
        printf("\nTemperature Plotting Program Menu\n");
        printf("\tE - Enter temperatures for scratchpad\n");
        printf("\tS - Store scratchpad to disk\n");
        printf("\tR - Read disk file to scratchpad\n");
        printf("\tT - Table view of current data\n");
        printf("\tG - Graph view of current data\n");
        printf("\tX - Exit the program\n");
        printf("\nPress one of the above keys: ");
        choice = toupper(getch());
        switch (choice)
        {
            case 'E': get_temps(); break;
            case 'S': save_temps(); break;
            case 'R': read_temps(); break;
            case 'T': table_view(); break;
            case 'G': graph_view(); break;
            case 'X': exit(0);
        }
    }
}

```

```

    }
}
/* Function definitions */
void get_temps(void)
{
    char inbuf[130];
    int reading;
    printf("\nEnter temperatures, one at a time.\n");
    for (reading = 0; reading < READINGS; reading++)
    {
        printf("\nEnter reading # %d: ", reading + 1);
        gets(inbuf);
        sscanf(inbuf, "%d", &temps[reading]);
        /* Show what was read */
        printf("\nRead temps[%d] = %d", reading, temps[reading]);
    }
}

void table_view(void)
{
    int reading;
    clrscr();      /* clear the screen */
    printf("Reading\t\tTemperature(F)\n");
    for (reading = 0; reading <= READINGS; reading++)
        printf("%d\t\t\t%d\n", reading + 1, temps[reading]);
    min_max();
    printf("Minimum temperature: \n");
    printf("Maximum temperature: \n");
    avg_temp();
    printf("Average temperature: \n");
}

void min_max(void)
{
    printf("\nExecuting min_max().\n");
}

void avg_temp(void)
{
    printf("\nExecuting avg_temp().\n");
}

void graph_vlew(void)
{

```

```

    printf("\nExecuting graph_view().\n");
}
void save_temps(void)
{
    printf("\nExecnting save_temps().\n");
}
void read_temps(void)
{
    printf("\nExecnting read_temps().\n");
}

```

该函数打印表头，然后用一个 for 循环取得并打印存入 `temps` 数组中的值。最后，还打印出最小值及平均温度。这些函数还没有完成，因此仅仅打印出信息表示它们正在被执行。

注意到包含未实现的函数调用，可以帮助你检查程序流程，和保持程序的结构。这种设计方法通常称为自顶向下的方法。因为它首先在程序的顶层(main 函数)开始设计，然后进入由 `main` 直接调用的函数的设计，如 `table_view`。当 `table_view` 的顶层正常工作后，便可实现它调用的函数——`min_max` 和 `avg_temp`。

现在来建立和运行 `PLOTEMP3.C`，选取 E，并输入测试数据(10, 20, 30, 40, 50, 60, 70 和 80)。当你再返回到菜单时，选取 T(Table view)，将得到如下显示：

Reading	Temperature (F)
1	10
2	20
3	30
4	40
5	50
6	60
7	70
8	80
9	0

Executing min_max().
 Minimum temperature:
 Maximum temperature:
 Executing avg temp().
 Average temperature:

7.10 通过设置 watches 监视程序

好，你已输入 8 个 `reading` 并返回 9 最后一个值为 0。这里你可能犯了恶劣的“差一”错误——循环的重复次数比期望的少或多一次。

首先可以通过在 `table_view` 中 for 循环的第一行设置一个断点来发现问题存在于何处。为此，将光标移到此行并选取 `Debug|Toggle Breakpoint`(或按 `Ctrl-F8`)。

之后，再次运行程序，输入测试数据，并选取 `Table view`。程序停在 `table_view` 中的断点处，现在，便可以看看 for 循环是如何运行的。

到此为止，已经能通过单步运行程序并使用 `Debug|Evaluate` 检查变量的值，这对于每次只检查一个值是很好的方法，但当处理循环或重复函数的调用时，你还想观察变量值的变化，如果仍重复上面步骤将是很烦人的，这里，调试程序为你提供了设置 `watches` 来自

动监视值的变化。一个 watch 是一个表达式，其值在运行程序中被不断修改。

1. 添加一个 watch

如果你对两个变量感兴趣: reading 在每次 for 循环中加 1, 和 temps[reading] 在每次循环中被打印的值。由于光标已在附近, 因此设置这些 watches 的最简单的方法是将光标移到你要观察的变量名上, 然后选取 Debug|Watches|Add Watch(或按 Ctrl-t7)。把光标移到 reading 试一试; 你将看到一个弹出窗口。和 Debug|Evaluate 一样, 被显示的缺省名是光标所在的变量, 只需按 Enter。用同样的过程为 temps[reading] 设置一个 watch。弹出窗口显示 temps, 和 Debug|Evaluate 一样, 你可以用 → 将表达式的剩余部分拷贝到该窗口中, 然后按 Enter 设置该 watch。

2. 观察 Watches

现在你已设置了两个 watch, 每个 watch 的变量名和值都显示在 watch 窗口中:

reading:177

temps[reading]:92

由于还未运行循环, 所以显示的值(可能在你的系统上是别的值)是没有意义的, 仅代表各内存位置上的情况。

现在开始单步运行循环(用 F8)。运行时, 注意值的变化。第一次循环完成后, 这些值为:

reading:0

temps[reading]:10

假设输入的是前面给出的测试数据, 从 10 开始。则下一次循环结时, watch 显示为:

reading:1

temps[reading]:20

执行完最后一次循环时, 其值为:

reading:8

temps[reading]:0

这表明循环在 reading 达到 8 时结束。应该什么时候结束呢? 由于要输入 8 个 reading, 而 reading 从 0 开始, 因此循环中 reading 的最后值就应为 7, 而不是 8。现在我们来看循环的终止条件:

reading <= READINGS

检查程序开始处的 #define, 发现 READINGS 等于 8。明白问题所在了吗? 要使 reading 等于 7 时(完成 8 个输入后)结束, 则条件应为 reading < READINGS。修改后, 再次运行程序看结果是否正确。

3. 控制调试程序窗口

如果你要设置多个 watches, 而可能无足够空间一次容纳它们。为此, 可以使用 PgUp 和 PgDn 键来滚动 Watch 窗口, 或用 ↑ 和 ↓ 键来一次移动一行。

如果有一个 watch 表达式太长不能全装入该窗口, 可以使用 Home、End、← 和 → 键来移动。(这在 Debug|Evaluate 窗口中也能使用。)

另一种方法是调整窗口, 见《用户手册》的第一章“IDE 指南”有关怎样处理新环境窗口部分。

记住任何时候, 都可按 Alt-F5 看整个程序输出屏幕。按任意键返回环境中。

练习使用这些性能。

4. 编辑和删除 watch.

编辑、添加或删除 watch 都很容易。当 watch 窗口处于活动状态时，当前活动表达式被置亮。要选择另一个表达式，可用 Home, End, ↑ 或 ↓ 键。

要编辑(修改)当前被置亮的 watch，你可选择 Debug|Watches|Edit Watch。当其显示在屏幕底线时，更容易，只需按 Enter，调试程序为被选表达式打开一个弹出窗口，你可以编辑它。练习将 watch 中 temps[reading]改为 temps[reading+1]。(注：这里除了表达式本身外，你不能改变它的值。要改变值，应用 Debug|Evaluate)。

你已经知道如何添加一个 watch，但当 watch 窗口处于活动状态时，还有一更简单的方法，即按 Ins，便出现一个弹出窗口。你可以输入 watch 表达式，用 → 键加入，或从光标所在位置拷入缺省值。

要删除当前 watch，选取 Debug|Watches|Delete Watch，或只按 Del。练习在 watch 中删去 reading。你也可通过选取 Debug|Watches|Remove All Watches 删除所有 watch。

5. 寻找一函数定义

既然 PLOTEMP.C 已开始被充实起来，则要找到欲检查的函数就比较困难了。调试程序提供了一种方法，可将编辑窗口滚动到一指定的函数定义上。

选取 Search|Locate Function，则 Turbo C++ 打开一对话窗口。练习输入 get_temps (不要输入函数名的括号，否则调试程序不能识别)。于是 get_temps 定义显示在编辑窗口中。这在浏览函数的定义时是很有用的，同样也可用于定位设置断点和 watch。

注意 Search|Locate Function 操作只用于在被带调试信息编译的文件中有其源代码的函数。库函数，如 printf 不能用 Search|Locate Function 查找，因为其源代码不在集成环境中。

6. 查找谁调用谁

在一个复杂的程序中，函数调用可能有许多层，你会发现当程序运行到某一断点时，要记住函数的调用顺序是比较困难的。调试程序也能帮助你解决该问题。在程序中你想看调用顺序的地方设一断点，作为练习，可在 min_max 的 printf 处设一断点。

运行程序并选取 Table View，于是程序将停止在断点处。现在来选取 Debug|Call stack，便出现一弹出窗口，列出了所有在该点等待继续执行的函数。Call Stack 中最近调用的函数在顶端，这里为 min_max。它被 table_view 调用，而 table_view 又依次被 main 调用。

你可以用 ↑ 和 ↓ 键置亮 Call stack 窗口中一个指定函数。如果按 Enter，则编辑窗口滚动到该函数中被执行的上一条语句(通常可以通过选取 Call stack 窗口中的第一个函数，返回到当前执行位置，这里为 min_max。现在：

- min_max 中被执行的上一条语句是其定义中的第一行，因为那是你设置断点的地方。

- table_view 中被执行的上一条语句是包含 min_max 调用的一行。

- main 中被执行的上一条语句是执行 table_view 的那行，即：case 'T':table_view;break 换句话说，min_max 正在执行中，而 table_view 和 main 等待完成。

7. 多个源文件

当你处理一个长文件时，会发现我们已讨论过的一个特别有用的性能。实际上许多编程工程都由几个源文件组成。调试程序自动将你要求的源文件装入编辑窗口。例如，如果

你用 Search|Locate Function 来查找一个说明在非当前编辑窗口文件的另一个文件中的函数, 则调试程序装入相应的源文件。若你在此前修改过当前文件, 则将先问你是否将修改过的文件存盘。当你用 Debug|Call stack 检查一个函数已执行的前一行, 而该函数定义在另一个源文件中时, 情况也一样。

虽然调试程序使得对多个源文件的操作变得容易, 但最好一次只调试一个或二个源文件。通常在继续前测试修改一个给定的定位错误, 因为, 可能恰恰是该点工作不正常, 而且还可能引入新的错误。

7.11 预防药

下面将继续 PLOTEMP.C 的开发和测试。为将来调试的需要, 先来看一些减少错误的方法和一些常见的错误类型。

1. 预防性设计

正如你驾驶汽车时多加预防, 便可以避免事故一样, 你可以在设计程序时多加小心以避免错误。你已看到, PLOTEMP.C 的设计就代表了一种自顶向下编程的预防性设计方法。

试一试用一些简单、易于定义的函数建立你的程序, 这可以使测试情况和结果分析变得容易, 同时也使程序易读和易于修改。例如, 如果 PLOTEMP.C 由在同一函数中的 table 和 graph view 组成, 则代码将变得不易控制。

尽量减少每个函数中要求的数据个数和要修改的元素个数。这将使设置测试和分析结果以及阅读和修改程序都变得较容易, 同时它也可以限制错误程序引起的大错误, 允许你在一个调试阶段中多运行几次该函数, 这种程序设计方法称为疏松连接。

2. 清楚地编写

将程序编写得清楚, 用一致的缩进书写形式, 带上一定的注释和变量名的描述。

保持程序的简洁。用多个简单的语句来表示复杂操作, 而不要用一个复杂的语句来显示你知道许多 C 晦涩的表达性能。Turbo C++ 的代码优化提高你的程序效率, 同时也使其易于调试、阅读和修改。

当你编写程序时, 不要去扣最后一点效益, 因为当你尽力使之高效时, 它也正趋于变得难读和难以调试。如果最后你的程序运行太慢的话, 再来考虑哪一部分值得提高速度, 和怎样最好地做到这一点(Turbo Profiler 是完成这一任务的工具)。

应小心编写可能在你的程序中以多种方式使用的函数和可能在别的程序中被调用的函数。编写和调试一个通用函数通常比编写两三个专用函数容易。

7.12 有系统的软件测试

飞机起飞前, 机组人员都要对其做系统的检查, 以保证各部分工作正常。按一定的例程工作可以节省开支。同样, 你也应用标准的方法来做软件测试: 一个经验性的测试步骤可以使你得到一个可靠的程序。

这里, 不存在一个测试程序的“好”方法; 你的测试表依赖于所编的程序、编程时间、水平和你的个人风格。下面的测试表可以作一个起点, 因为它反映较广泛的经验。

■给程序一些简单但不琐碎的输入。试一下不常用的情况——例如, 在 PLOTEMP 中输入负温度值, 用 Debug|Evaluate 和 watch 来检查数据项的值。改正你发现的错误, 一次一个或几个。

■给程序输入另一组数据，检查上一步中未检查的部分。可能的话，找一个不熟悉你的程序的人用键盘使用一组数据。经验表明程序员本人有时难以发现问题，因为他知道哪些数据合适，哪些数据不合适。如果你的程序是为会计编的，则找一个会计。

■测试程序中的每一条语句。你可能在未预计到的地方发现错误。

■将调试程序放于一边，运行整个程序来修改。如果把程序给别人用，他肯定希望运行正常，测验程序对每个可能出现的错误的反映。一个能较好地处理多类错误的程序便可称其是健壮的。

1. 全面测试修改结果

当你修改一个程序后，应重新测试所有受影响的部分，尤其应测试那些未被修改但受到影响的部分。

如果程序较复杂，应保持上次运行的记录。当你修改程序时，该记录可以帮助你重复以前的测试，以便检查是否受修改的影响。如果测试包括一特定的输入文件，存该文件。

2. 仔细观察的部分

当你继续学习 C 和开发程序时，保留一个常见语法错误和代码出错表，以便调试阶段检查这些错误部分。这里有一些许多程序易于出错的地方：

■越界错误

■混淆地址和这些地址中的值。

■增量运算符和减量运算符放错了地方。

■语句测试不彻底。

■用 Pascal 语法代替 C。

下面讨论以上各情况。

特别注意边界条件——尤其是退出循环的条件，填充数组的条件等等。错误常常出现在边界条件的处理上。在前面条件 `reading <= READINGS` 引起的多一个值已可见这一点。由 1 而不是由 0 还会引起别的错误。

通常还应小心你是否指定了一个地址或该地址内的值。例如不要混淆值 `temps[reading]` 和地址 `&temps[reading]`。

小心增量和减量运算符 `++` 和 `--`。是在使用前还是在之后增量？

警惕必须用多种方法测试的单个语句或表达式，如：

```
switch(strcmp(a, b))...
```

`strcmp` 可能返回三个值：0(a 等于 b)，-1(a 小于 b)，或 +1(a 大于 b)。建议你用三组输入值来测试该语句，以便验证 `strcmp` 在各种情况下都工作正常。

```
x = (x > 0) ? func(x) : 0;
```

该语句包含一个“隐含 if”，可产生二个不同的结果。

7.13 完成 PLOTEMP.C

你已经安装和测试了 PLOTEMP.C 原型，并已完成、测试和调试了 `get_temps` 和 `table_view` 函数后，你已经学习了怎样使用调试程序的所有性能。通过 PLOTEMP.C 的实现，可以练习：

■用下面给出代码的特定函数代替原存根程序。我们已为你提供了各段程序，你只需输入，并装入下一版。

- 修改函数原型(需要的话)
- 用适当的调试工具测试函数的执行
- 发现并修改错误
- 移到下一个函数。

答案给在本章末。

1. 完成 table_view

欲完成该函数, 你必须实现下面两个函数:

```
void min_max(int num_vals, int vals[], int *min_val, int *max_val)
{
    int reading;
    *min_val = *max_val = vals[0];
    for (reading = 1; reading < num_vals; reading++)
    {
        if (vals[reading] < *min_val)
            *min_val = (int)&vals[reading];
        else if (vals[reading] > *max_val)
            *max_val = (int)&vals[reading];
    }
}

float avg_temp(int num_vals, int vals[])
{
    int reading, total = 1;
    for (reading = 0; reading < num_vals; reading++)
        total += vals[reading];
    return (float) total/reading; /* reading equals total vals */
}
```

装入 PLOTEMP4.C。记住我们在 PLOTEMP 程序中故意放入一些错误, 以便你可以练习调试技巧。

由于这两个函数带有参数并有返回值, 所以应修改其函数原型:

```
void min_max(int num_vals, int vals[], int *min_val, int *max_val)
float avg_temp(int num_vals, int vals[])
```

这些修改也包含在 PLOTEMP4.C 中。

最后, 修改 table_view, 以便正确使用这些函数的返回值。因此 table_view 应为如下形式:

```
void table_view(void)
{
    int reading, min, max;
    clrscr(); /* clear the screen */
    printf("Reading\t\tTemperature(F)\n");
    for(reading = 0; reading < READINGS; reading++)
```

```

    printf("%d\t\t\t(%d\n", reading + 1, temps[reading]);
min_max(READINGS, temps, &min, &max);
printf("Minimum temperature: %d\n", min);
printf("Maximum temperature: %d\n", max);
printf("Average temperature: %f\n", avg_temp(READINGS, temps));
}

```

该修改也包含在 PLOTEMP4.C 中。

接下来，由你自己进行调试。注意以下几方面：

- 循环的工作是否正确？
- 算术运算是否合适？
- 比较操作要比较什么？

2. 实现 graph_view

回忆一下图 7.2 中 graph_view 函数产生的图表。欲实现该函数，用下面给出的函数定义代替其存根函数(注意应在程序开始处加 #include <graphics.h>)：

```

void graph_view(void)
{
    int graphdriver = DETECT, graphmode;
    int reading, value;
    int maxx, maxy, left, top, right, bottom, width;
    int base; /* zero x-axis for graph */
    int vscale = 1.5; /* value to scale vertical bar size */
    int space = 10; /* spacing between bars */
    char sprintf[20]; /* formatted text for sprintf */
    initgraph(&graphdriver, &graphmode, "..\\bgi");
    if (graphresult() < 0) /* make sure initialized OK */
        return;
    maxx = getmaxx(); /* farthest right you can go */
    width = maxx / (READINGS + 1); /* scale and allow for spacing */
    maxy = getmaxy() - 100; /* leave room for text */
    left = 25;
    right = width;
    base = maxy / 2; /* allow for neg values below */
    for (reading = 0; reading <= READINGS; reading++)
    {
        value = temps[reading] * vscale;
        if (value > 0)
        {
            top = base - value; /* toward top of screen */
            bottom = base;
            setfillstyle(HATCH_FILL, 1);

```

```

    }
    else
    {
        top = base;
        bottom = base - value;      /* toward bottom of screen */
        setfillstyle(WIDE_DOT_FILL, 2);
    }
    bar(left, top, right, bottom);
    left += (width + space);        /* space over for next bar */
    right += (width + space);       /* right edge of next bar */
}
outtextxy(0, base, "0 -");
outtextxy(10, maxy + 20, "Plot of Temperature Readings");
for (reading = 0; reading < READINGS; reading++)
{
    sprintf(fprint, "%d", temps[reading]);
    outtextxy((reading * (width + space)) + 25, maxy + 40, fprint);
}
outtextxy(50, maxy+80, "Press any key to continue");
getch();                          /* Wait for a key press */
closegraph();
}

```

3. save_temps 和 read_temps

函数 `save_temps` 是将当前的“scratchpad”（即数组 `temps` 的内容）存入磁盘文件。在此，你应对数组元素存取的逻辑关系很熟悉了。

用下面程序代替 `save_temps` 函数的存根定义：

```

void save_temps(void)
{
    FILE * outfile;
    char file_name[40];
    printf("\nSave to what filename? ");
    while (kbhit()); /* "eat" any char already in keyboard buffer */
    gets(file_name);
    if ((outfile = fopen(file_name, "wb")) == NULL) {
        perror("\nOpen failed! ");
        return;
    }
    fwrite(temps, sizeof(int), READINGS, outfile);
    fclose(outfile);
}

```

函数 `read_temps` 与 `save_temps` 相反，它从磁盘文件中读出数据并放在数组 `temps` 中。通过用以下程序段替换原存根定义实现 `read_temps`：

```
void read_temps(void)
{
    FILE * infile;
    char file_name[40] = "test";
    printf("\nRead from which file? ");
    while (kbhit()); /* "eat" any char already in keyboard buffer */
    gets(file_name);
    if((infile == fopen(file_name,"rb")) == NULL)
    {
        perror("\nOpen failed! ");
        return;
    }
    fread(temps, sizeof(int), READINGS, infile);
    fclose(infile);
}
```

该变换也包括在 `PLOTEMP5.C` 中。

完成 `read_temps` 后，你便已得到了 `PLOTEMP.C` 的一个完整的工作版本(`PLOTEMP6C` 是该程序是无错版本)。

7.14 调试练习的答案

在 `PLOTEMP5.C` 中的函数仍存在一些错误。

1. `min_max` 和 `avg_temps`

在 `min_max` 中，`if` 语句将 `&vals[reading]` (是一个地址)赋值给 `min` 或 `max`，而不是正确的值 `vals[reading]`。同样，在 `avg_temp` 中，变量 `total` 的值为 0，作为 `reading` 的计数，应加 1 从 1 开始，才能得到正确的平均值。

顺便注意一下，调用函数 `table_view` 时指针的传送和接收——不是该变量的实际值。欲进一步考虑，复习第四章有关指针部分。

2. `graph_view`

该函数中潜在两个错误，在 `for` 循环中：

```
for(reading=0;reading<=READINGS;reading++)
{
```

```
    value=temps[READINGS]*vscale;
```

被取的值是 `temp[READINGS]`——是常量 `READINGS` 而不是变量 `reading`。用这个值绘图的结果是根本不存在的元素 `temps[8]`。第二个错误是条件 `<=READINGS` 应为 `<READINGS`，这样读出的数据个数才正确。注意第一个错误对第二个错误的影响——通常在你修改完另一错误时才能发现该错误，因为第一个错误影响了包含第二个错误的代码执行。

3. `save_temps`

这里，错误不在 for 循环的第一行上。而在下面一段程序中：

```
if((outfile=fopen(file_name, "wb"))==NULL)
    perror("\nOpen failed ");
return;
```

应用 {} 将 perror 语句和 return 语句括在 if 语句下。否则，如果打开文件操作正确，则函数总是在该点返回。

编译程序警告“Unreachable code in function save_temps”的意思是 return 语句后的程序将永远不会被执行。因为没有括号，总是执行 return。

4. read_temps

编译时，将看到编译程序的警告“Possible use of infile before definition”。考虑下面语句：

```
if((infile=fopen(file_name, "rb"))==NULL)
```

当你打开一个文件时，文件柄 infile 从 fopen 中取得一个值，然后测试该值，看它是否为 NULL，以说明文件打开操作是否失败。那为什么在 infile 第一次使用时就说它未定义(取得一个值)呢？原因在于跟在 infile 后的是 ==，而不是 =。因为 == 表示比较而不是赋值，因此 infile 不能取得一个值。

FORM 7-5027-000-1, 7/01

Rev. 7/01