Broadview
www.broadview.com.cn

以任务驱动方式讲解，用实例引导读者学习
只需21天，便可轻松掌握Linux平台C编程

畅销书
新品

21天学通 Linux C编程

马玉军 吴绍兴 梁胜斌 等编著

本书特色

- 基础知识→核心技术→典型实例→综合练习→项目案例
- 218个典型实例、254个练习题
- 一线开发人员全程贴心讲解，上手毫不费力

超值DVD

- 21小时多媒体语音视频教学
- 本书源代码 + 本书电子教案（PPT）
- 1000余页编程参考宝典电子书（免费赠送）

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

21天学通系列

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

本书涵盖主题

- Linux编程平台
- Linux常用命令
- 编译与调试
- C程序的常用语句
- 函数
- 结构体
- 目录与文件
- 网络编程
- 面向连接的套接字通信
- 图形界面编程
- 信号与事件
- 文本编辑器的使用
- Shell编程
- C语言基础
- 数组与指针
- 字符与字符串处理
- 时间函数
- 文件IO
- 无连接的套接字通信
- Linux图形界面开发基础
- 界面布局

精彩内容，尽在21天学编程

21小时多媒体
语音视频教学
DVD

上架建议：Linux

ISBN 978-7-121-10622-4

9 787121 106224 >

定价：49.80元（含DVD光盘1张）

责任编辑：高洪霞
责任美编：李 玲

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

21天学通系列

以任务驱动方式讲解，用实例引导读者学习

21天学通 LinuxC编程

马玉军 吴绍兴 梁胜斌 等编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书由浅入深、通俗易懂地讲解了 Linux 系统下的 C 程序设计。全书共 21 章，从 Linux 系统安装开始，讲解了 Linux 常用命令、Shell 编程、C 语言基础、指针与函数、C 程序网络编程、图形界面程序设计等内容。书中通过大量的实例来讲解知识要点，读者可以通过这些实例和练习系统地学习 Linux 下的 C 程序设计，提高自己的编程能力。

本书共分 7 篇。第一篇为 Linux 基础篇，介绍 Linux 编程平台的搭建及文本编辑器的使用；第二篇为 shell 编程篇，介绍 Linux 下的常用 shell 命令与 shell 编程；第三篇为 C 程序基础篇，介绍 C 程序设计的基础知识；第四篇为指针与函数篇，重点介绍 C 语言中指针与函数的概念和用法；第五篇为文件与目录篇，介绍 C 语言中的文件与目录操作功能；第六篇为网络编程篇，介绍网络编程的基础知识和如何使用 C 语言进行网络编程；第七部分为图形界面开发篇，介绍基于 Linux 的 C 程序图形界面开发及 Glade 的使用。

本书语言通俗易懂，内容丰富，注重理解与实例，知识涵盖面广。非常适合 C 语言的初学者、高校学生、程序开发人员、Linux 工作人员阅读和学习。

本书附赠 DVD 光盘 1 张，内容包括超大容量手把手教学视频、电子教案（PPT）、编程参考宝典电子书、源代码及各章习题答案。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

21 天学通 Linux C 编程 / 马玉军等编著. —北京：电子工业出版社，2010.5
（21 天学通系列）
ISBN 978-7-121-10622-4

Ⅰ. ①2… Ⅱ. ①马… Ⅲ. ①Linux 操作系统－程序设计 ②C 语言－程序设计 Ⅳ. ①TP316.89 ②TP312

中国版本图书馆 CIP 数据核字（2010）第 056105 号

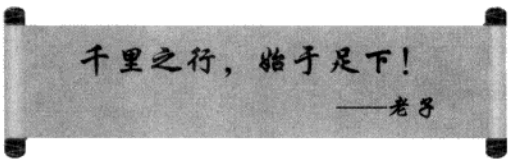
责任编辑：高洪霞
印　　刷：北京市天竺颖华印刷厂
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：26.25　字数：678 千字
印　　次：2010 年 5 月第 1 次印刷
印　　数：4000 册　定价：49.80 元（含 DVD 光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

本书特点



为什么要写这样一本书

Linux 是一款非常优秀的开放源代码操作系统，在服务器解决方案方面和嵌入式领域有着巨大的优势。近年来国内无数大中型企业选择 Linux 系统作为服务器解决方案，对服务器软件开发工作人员需求量巨大，但是相关的开发人员却非常紧缺。又因为 C 语言是 Linux 下最重要的编程语言，所以对 Linux 环境下的 C 程序开发人员的需求更为迫切。C 语言在语法结构和编程风格方面有其自身的特点，较其他编程语言难于学习和掌握。

基于上面的情况，笔者结合自身多年 Linux 下 C 程序开发经验，从一个初学者的角度系统地讲解 Linux 下 C 程序开发知识。书中对有关原理的阐述力求做到通俗易懂，并通过大量的编程实例进行讲解。在这本书中，笔者将通过 21 天的学习规划，让读者快速掌握 Linux 下 C 程序开发技能。

本书有何特色

相比同类图书，本书具有以下明显特色。

1．细致体贴的讲解

为了让读者更快地上手，本书特别设计了适合初学者的学习方式，用准确的语言总结概念➡用直观的图示演示过程➡用详细的注释解释代码➡用形象的比方帮助记忆。效果如下：

① 知识点介绍 准确、清晰是其显著特点，一般放在每一节开始位置，让零基础的读者了解相关概念，顺利入门。

② 范例 书中出现的完整实例，以章节顺序编号，便于检索和循序渐进地学习、实践，放在每节知识点介绍之后。

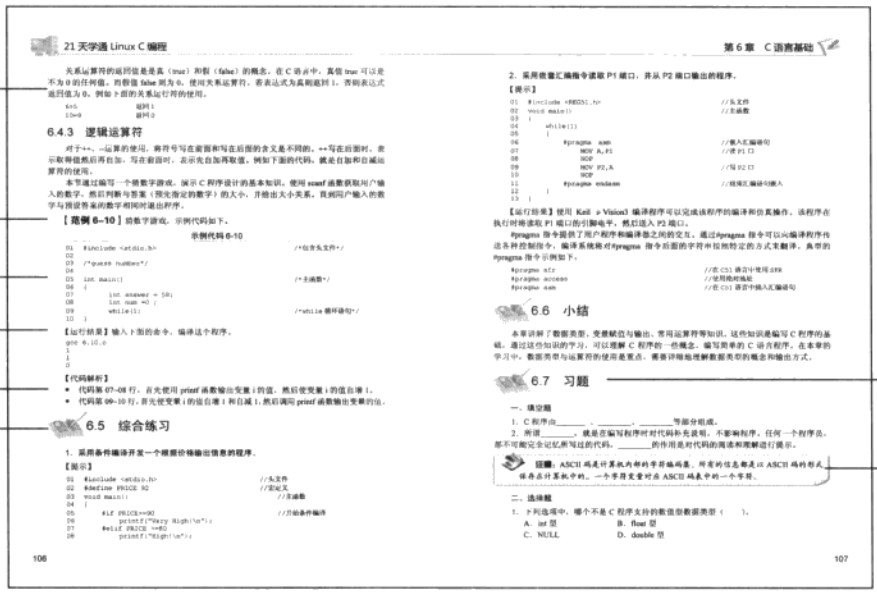
③ 范例代码 与范例编号对应，层次清楚、语句简洁、注释丰富，体现了代码优美的原则，有利于读者养成良好的代码编写习惯。对于大段程序，均在每行代码前设定编号，便于学习。

④ 运行结果 对范例给出运行结果和对应图示，帮助读者更直观地理解范例代码。

⑤ 代码解析 对范例代码中的关键代码进行逐一解释，有助于读者掌握相关概念和知识。

⑥ 综合练习 为了便于读者巩固所学内容，本书每章中均提供了综合练习，并给出了操作提示和结果，配合读者自己动手实践。

⑦ 习题 每章最后提供专门的测试习题，供读者检验所学知识是否牢固掌握，题目的提示或答案放在光盘中。



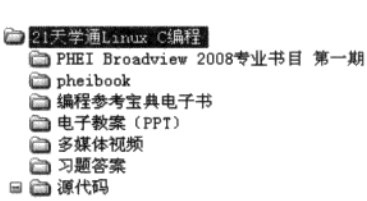
⑧ **贴心的提示** 为了便于读者阅读，全书还穿插着一些技巧、提示等小贴士，体例约定如下：

- 提示：通常是一些贴心的提醒，让读者加深印象或提供建议，或者解决问题的方法。
- 注意：提出学习过程中需要特别注意的一些知识点和内容，或者相关信息。
- 警告：对操作不当或理解偏差将会造成的灾难性后果做警示，以加深读者印象。

经作者多年的培训和授课证明，以上讲解方式是最适合初学者学习的方式，读者按照这种方式，会非常轻松、顺利地掌握本书知识。

2. 实用超值的 DVD 光盘

为了帮助读者比较直观地学习，本书附赠 DVD 光盘，内容包括多媒体视频、电子教案（PPT）、编程参考宝典电子书、各章习题答案和实例源代码等。



● 多媒体视频

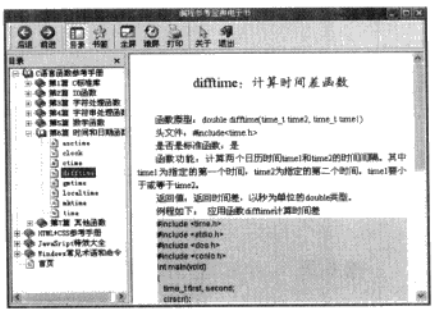
配有长达 21 小时手把手教学视频，讲解关键知识点界面操作和书中的一些综合练习题。作者亲自配音、演示，手把手教会读者使用。

● **电子教案（PPT）**

本书可以作为高校相关课程的教材或课外辅导书，所以笔者特别为本书制作了电子教案（PPT），以方便老师教学使用。

● **编程参考宝典电子书**

为方便广大读者学习，特别制作了编程开发参考电子书，供读者查阅和参考。



3．提供完善的技术支持

本书提供了论坛：http://www.rzchina.net，读者可以在上面提问交流。另外，论坛上还有一些小的教程、视频动画和各种技术文章，可帮助读者提高开发水平。

4．丰富的额外素材下载

相关的开发素材文件在 www.broadview.com.cn 提供下载。

推荐的学习计划

本书从初学者角度出发，结合对学习阶段的认识，归纳了最适合初学者的学习模式，并为读者总结了合理的学习时间分配方式，列表如下：

时间安排		学习内容	教学目标	难度指数
第1周	第1天	熟悉并搭建 Linux 编程平台，了解常见的 Linux 发行版	了解常见的 Linux 发行版及各自的特点，掌握 Linux 操作系统的安装过程，能够使用虚拟机搭建 Linux 编程平台。能够在 Linux 下编写和编译 C 程序，并运行生成文件	★
	第2天	Linux 下文本编辑器的使用	熟练掌握 Linux 下的文本编辑器的使用，重点是掌握纯文本编辑器 vim 的使用。能够使用编辑器进行代码的快速编写	★★
	第3天	常见目录与文件操作命令的使用	熟练掌握各种常见的 Linux 目录和文件操作命令，掌握使用 man 命令和 info 命令获取命令帮助的方法并能够读懂帮助文档	★★

续表

时间安排		学习内容	教学目标	难度指数
第 1 周	第 4 天	Linux 下 bash 的使用，shell 程序的基本结构	掌握 Linux 下 shell 编程的基本方法。能够利用 shell 特性和 Linux 的系统命令编程实现各种系统管理任务	★★★
	第 5 天	C 程序的编译概念，使用 gcc	理解编译的基本概念，能够使用 gcc 和 gdb 编译 C 程序和 C++程序，掌握 gdb 调试 C 程序的过程，能够通过 gdb 的帮助功能查阅常见 gdb 命令的用法，能够判断并纠正常见的编译错误	★★★
	第 6 天	C 程序的基本结构	了解 C 程序的基本结构，能够编写符合规范的简单的 C 程序，掌握 C 程序中变量的数据类型及变量的定义方法，能够为各种类型的变量赋值并输出变量的值，熟练掌握各种运算符的使用	★★★
	第 7 天	C 程序中的循环控制语句	掌握 C 程序中的各种流程控制语句的基本用法，在编程中能够选择合适的流程控制语句实现程序的功能，掌握问号运算符及块语句的用法	★★★
第 2 周	第 8 天	C 程序中的数组和指针	理解数组和指针的概念，掌握数组定义与数组元素的访问方法，掌握指针变量的定义方法，理解指针变量的算术运算的实质，能够使用指针变量的方式访问数组的元素，熟练掌握二维数组的定义与使用方法，了解多维数组	★★★★★
	第 9 天	函数的使用	理解函数的概念，能够使用系统的库函数完成程序的功能，能够自定义函数实现程序的概念，掌握函数的参数使用方法，能够利用指针实现函数参数的引用调用，熟悉各种常见的库函数及需要包含的头文件	★★★★
	第 10 天	字符串函数的使用	掌握各种字符与字符串处理函数的用法，能够使用这些函数灵活地实现字符与字符串的操作	★★★
	第 11 天	结构体的概念和使用	理解结构体的基本概念，能够定义结构体表示复杂的数据类型，掌握结构体数组及结构体指针的定义与访问方法，掌握在函数中使用结构体作为函数参数与返回值的用法	★★★★

续表

时间安排		学习内容	教学目标	难度指数
第 2 周	第 12 天	常用的时间函数	掌握各种获取系统时间的函数的用法，能够利用设置时间函数修改系统的时间	★★
	第 13 天	目录与文件的使用	理解 Linux 下文件的权限，了解常见的文件操作错误处理与错误号，能够利用系统函数实现文件与目录的创建与删除操作，掌握文件操作的整个流程，理解文件锁定的概念及用法，掌握实现文件的移动与复制的方法	★★★★★
	第 14 天	文件的输入和输出	掌握文件的打开与关闭函数的用法，掌握文件的读写函数的用法，能够利用文件操作函数实现常规的读写操作	★★★
第 3 周	第 15 天	网络编程的概念和应用	理解基本的网络概念，理解套接字，掌握套接字类型及相关的数据类型，掌握获取网络协议、网络服务及网络地址转换等相关信息的系统函数的用法，能够判断并处理网络编程中常见的错误	★★★★★
	第 16 天	无连接套接字通信的基本概念和创建方法	理解无连接套接字的基本概念机工作流程，掌握 socket 套接字的建立、状态获取和设置的方法，掌握无连接套接字发送信息与接收信息函数的用法，能够编写基本的面向无连接的网络服务器程序和网络客户端程序	★★★★★
	第 17 天	面向连接的套接字通信工作概念和流程	理解面向连接的套接字通信工作流程，熟练掌握绑定端口函数 bind、等待监听函数 listen、接受连接函数 accept、请求连接函数 connet、数据接收函数 recv、信息发送函数 send、数据传输函数 write 与 read，能够编写基本的面向连接的网络服务器程序和网络客户端程序	★★★★★
	第 18 天	Linux 的桌面环境	理解 Linux 下的 X Window 系统体系结构，了解 KDE 及 GNOME，掌握 GTK 开发库的安装，能够确定和纠正图形界面程序开发的常见错误	★★★
	第 19 天	Linux 图形界面编程	掌握窗口的创建及基本操作方法，了解图形界面编程中常见的界面元素，掌握窗口、标签、按钮及文本框的使用方法，能够利用这些元件编写基本的图形界面应用程序	★★★★

续表

时间安排		学习内容	教学目标	难度指数
第3周	第20天	界面布局的常用方法	熟练掌握使用表格进行界面布局的方法，会用垂直框和水平窗格进行界面布局，掌握界面布局工具Glade的使用方法，能够实现较为复杂的界面布局效果	★★★
	第21天	图形界面编程中信号与事件的基本概念和事件	理解信号与事件的基本概念和运行机制，掌握添加信号与事件的方法，能够编写基于事件驱动的图形界面应用程序	★★★★★

本书适合哪些读者阅读

- Linux下C编程的自学人员和初学者；
- 高校计算机专业学生；
- 相关培训班的培训学员和老师；
- Linux爱好者和工作人员；
- Linux程序员；
- 其他编程爱好者。

本书作者

本书主要由南阳理工学院的马玉军、南阳理工学院的吴绍兴和河南大学的梁胜彬编写，其他参与编写和资料整理的人员有昊燃、方振宇、陈冠佐、傅奎、陈勤、梁洋洋、毕梦飞、陈庆、柴相花、陈非凡、陈华、陈嵩。

编著者

目　录

第一篇　Linux 基础篇

第 1 章　Linux 编程平台（ 教学视频：64 分钟） ····· 25

1.1　Linux 发行版的介绍 ····· 25
1.1.1　Red Hat Linux ····· 25
1.1.2　Debian Linux ····· 25
1.2　安装 Linux ····· 26
1.2.1　Linux 的分区格式 ····· 26
1.2.2　在 Windows 系统中划分 Linux 分区 ····· 26
1.2.3　安装 Red Hat 9.0 ····· 28
1.2.4　GRUB 在 Windows 分区中引导 Linux ····· 35
1.3　VMware 虚拟机安装 Linux 系统 ····· 36
1.3.1　VMware 软件安装 ····· 36
1.3.2　新建虚拟计算机 ····· 38
1.3.3　VMware 的使用 ····· 41
1.3.4　在 VMware 虚拟机中安装 Linux 系统 ····· 42
1.4　第一个 C 程序 ····· 43
1.4.1　在 Linux 中编写 C 程序代码 ····· 43
1.4.2　编译程序 ····· 44
1.5　小结 ····· 44
1.6　习题 ····· 45

第 2 章　使用文本编辑器（ 教学视频：39 分钟） ····· 46

2.1　使用 vim ····· 46
2.1.1　启动 vim ····· 46
2.1.2　在桌面上创建 vim 启动器 ····· 47
2.1.3　vim 的工作模式 ····· 47
2.1.4　保存与打开文件 ····· 47
2.1.5　移动光标 ····· 48
2.1.6　插入操作 ····· 50
2.1.7　删除操作 ····· 50
2.1.8　取消操作 ····· 51
2.1.9　保存操作 ····· 51
2.1.10　退出操作 ····· 52
2.1.11　查找操作 ····· 52
2.1.12　替换操作 ····· 52

2.1.13 选项设置 ····· 52
2.1.14 调用 shell 命令 ····· 53
2.2 vim 使用实例 ····· 53
2.2.1 字符的插入与删除 ····· 53
2.2.2 字符的查找与替换 ····· 54
2.3 gvim ····· 55
2.3.1 文件的新建与保存 ····· 55
2.3.2 查找与替换 ····· 55
2.4 gedit ····· 56
2.4.1 gedit 的启动与打开文本 ····· 56
2.4.2 gedit 编辑文件 ····· 57
2.4.3 gedit 打印文件 ····· 58
2.4.4 gedit 的首选项 ····· 59
2.5 小结 ····· 61
2.6 习题 ····· 61

第二篇 shell 编程篇

第 3 章 Linux 常用命令（ 教学视频：59 分钟） ····· 62
3.1 目录与文件命令 ····· 62
3.1.1 查看目录命令 pwd ····· 62
3.1.2 切换目录命令 cd ····· 62
3.1.3 查看目录命令 ls ····· 63
3.1.4 建立文件命令 touch ····· 64
3.1.5 建立与删除文件夹命令 mkdir、rmdir ····· 65
3.1.6 文件复制命令 cp ····· 65
3.1.7 文件移动和更名命令 mv ····· 66
3.1.8 删除文件与目录命令 rm ····· 66
3.1.9 创建链接命令 ln ····· 67
3.1.10 查看文件信息命令 wc ····· 67
3.1.11 文件查看命令 cat ····· 68
3.1.12 查看文件视图命令 head 与 tail ····· 68
3.1.13 文件权限设置命令 chmod ····· 69
3.1.14 查找文件命令 find ····· 70
3.2 man 系统帮助命令 ····· 70
3.2.1 man 手册的使用方法 ····· 71
3.2.2 帮助文档的结构 ····· 71
3.3 小结 ····· 72
3.4 习题 ····· 72

第 4 章　shell 编程（ 教学视频：122 分钟）……74
4.1　bash 的使用……74
4.1.1　bash 的内部命令与外部命令……74
4.1.2　命令补齐功能……74
4.1.3　命令通配符……75
4.1.4　使用命令的历史记录……75
4.1.5　命令的别名……76
4.2　shell 程序的基本结构……76
4.3　局部变量……77
4.4　环境变量……78
4.4.1　环境变量的查看……78
4.4.2　访问环境变量……78
4.4.3　定义环境变量……78
4.4.4　在系统配置文件中定义环境变量……79
4.4.5　位置变量……79
4.5　shell 的运算符……80
4.5.1　变量赋值……80
4.5.2　算术运算符……80
4.6　输入和输出……82
4.6.1　echo 命令输出结果……82
4.6.2　read 命令读取信息……83
4.6.3　文件重定向……84
4.7　引号的使用方法……85
4.7.1　双引号……85
4.7.2　单引号……85
4.7.3　反引号……86
4.7.4　反斜线……86
4.8　测试语句……86
4.8.1　文件状态测试……87
4.8.2　数值测试……87
4.8.3　字符串测试……88
4.8.4　逻辑测试……89
4.9　流程控制结构……89
4.9.1　if 语句……89
4.9.3　for 语句……90
4.9.5　until 语句……91
4.10　综合练习……92
4.10.1　程序的功能……92
4.10.2　编写程序的代码……92
4.11　小结……94
4.12　习题……94

第三篇　C 程序基础篇

第 5 章　编译与调试（ 教学视频：49 分钟） ……95

5.1　编译的概念和理解 ……95
5.1.1　程序编译的过程 ……95
5.1.2　编译器 ……95
5.2　gcc 编译器 ……96
5.2.1　gcc 编译器简介 ……96
5.2.2　gcc 对源程序扩展名的支持 ……96
5.3　C 程序的编译 ……97
5.3.1　编写第一个 C 程序 ……97
5.3.2　用 gcc 编译程序 ……98
5.3.3　查看 gcc 的参数 ……99
5.3.4　设置输出的文件 ……100
5.3.5　查看编译过程 ……100
5.3.6　设置编译的语言 ……101
5.3.7　设置 ANSIC 标准 ……101
5.3.8　编译 C++程序 ……101
5.4　编译过程的控制 ……102
5.4.1　编译过程简介 ……102
5.4.2　控制预处理过程 ……103
5.4.3　生成汇编代码 ……103
5.4.4　生成目标代码 ……104
5.4.5　链接生成可执行文件 ……104
5.5　gdb 调试程序 ……105
5.5.1　gdb 简介 ……105
5.5.2　在程序中加入调试信息 ……105
5.5.4　启动 gdb ……105
5.5.5　加载需要调试的程序 ……106
5.5.6　查看代码 ……106
5.5.7　在程序中加入断点 ……106
5.5.8　查看断点 ……107
5.5.9　运行程序 ……107
5.5.10　变量的查看 ……108
5.6　gdb 的帮助与常用命令 ……109
5.6.1　使用 gdb 帮助 ……109
5.6.2　gdb 常用命令 ……109
5.7　编译程序常见的错误与问题 ……110
5.7.1　逻辑错误与语法错误 ……110
5.7.2　C 程序中的错误与异常 ……110

5.7.3 编译中的警告提示 111
5.7.4 找不到包含文件的错误 111
5.7.5 错误地使用逗号 111
5.7.6 括号不匹配错误 111
5.8 综合练习 112
5.8.1 编写程序 112
5.8.2 编译文件 113
5.8.3 调试程序 113
5.9 小结 115
5.10 习题 115
第 6 章 C 语言基础（ 教学视频：70 分钟） 117
6.1 C 程序的基本概念 117
6.1.1 C 程序的基本结构 117
6.1.2 C 程序的一般格式 118
6.1.3 C 程序中的注释 119
6.2 数据类型 119
6.2.1 整型（int） 119
6.2.2 浮点型（float） 120
6.2.3 字符型（char） 121
6.2.4 变量名 122
6.2.5 字符 NULL 123
6.3 变量的赋值与输出 123
6.3.1 变量的赋值 123
6.3.2 printf 函数输出变量 123
6.3.3 scanf 函数从键盘读入变量 125
6.4 运算符 126
6.4.1 算术运算符 126
6.4.2 关系运算符 127
6.4.3 逻辑运算符 128
6.5 综合练习 129
6.6 小结 130
6.7 习题 130
第 7 章 C 程序的常用语句（ 教学视频：87 分钟） 132
7.1 流程控制语句 132
7.1.1 if 条件语句 132
7.1.2 if 语句的嵌套 133
7.1.3 switch 选择执行语句 135
7.1.4 for 循环语句 137
7.1.5 for 循环的嵌套 138

7.1.6 for 循环应用实例：输出九九乘法口诀表 ····· 140
7.1.7 while 循环语句 ····· 140
7.1.8 do while 循环语句 ····· 141
7.1.9 转移控制语句：continue ····· 142
7.1.10 转移控制语句：break ····· 143
7.1.11 转移控制语句：return ····· 144
7.2 两种特殊语句结构 ····· 144
7.2.1 ?三元操作符 ····· 145
7.2.2 块语句 ····· 145
7.3 综合练习 ····· 146
7.4 小结 ····· 147
7.5 习题 ····· 147

第四篇 指针与函数篇

第 8 章 数组与指针（教学视频：66 分钟） ····· 149

8.1 数组的理解与操作 ····· 149
8.1.1 什么是数组 ····· 149
8.1.2 数组的定义与访问 ····· 150
8.1.3 数组使用实例 ····· 151
8.2 指针 ····· 152
8.2.1 指针的理解 ····· 152
8.2.2 指针操作符 ····· 152
8.2.3 指针的定义与访问 ····· 152
8.2.4 指针使用实例 ····· 153
8.3 数组与指针 ····· 154
8.3.1 数组与指针的关系 ····· 154
8.3.2 指针的算术运算 ····· 155
8.3.3 字符数组与字符串 ····· 156
8.4 二维数组与多维数组 ····· 157
8.4.1 二维数组的理解 ····· 157
8.4.2 二维数组与指针 ····· 158
8.5 实例 ····· 159
8.5.1 学生成绩统计实例 ····· 159
8.5.2 冒泡法排序实例 ····· 160
8.5.3 统计字符串中字符 ····· 162
8.6 小结 ····· 164
8.7 习题 ····· 164

第 9 章 函数（教学视频：67 分钟） ····· 166

9.1 函数的理解 ····· 166

9.1.1 什么是函数 166
9.1.2 系统函数与库函数 167
9.1.3 函数的返回值 168
9.1.4 无参函数 169
9.1.5 有参函数 169
9.1.6 函数参数实例 170
9.2 自定义函数 171
9.2.1 自编写定义函数 171
9.2.2 函数中调用函数 172
9.2.3 函数的声明 174
9.2.4 递归函数 175
9.2.5 main 函数的参数 argc 与 argv 176
9.3 函数与指针 177
9.3.1 值调用与引用调用 177
9.3.2 引用调用与指针 178
9.3.3 指针参数简单实例 179
9.3.4 自定义函数中指针使用实例 180
9.3.5 数组作为参数 181
9.3.6 数组作为函数参数实例 182
9.4 返回值 183
9.4.1 函数返回值的类型 183
9.4.2 函数返回指针 184
9.4.3 函数返回指针实例 185
9.4.4 函数返回结果到参数 186
9.5 综合练习 186
9.6 小结 187
9.7 习题 188
第 10 章 字符与字符串处理（ 教学视频：66 分钟） 190
10.1 字符测试函数 190
10.1.1 数字或字母测试函数 isalnum 190
10.1.2 字母测试函数 isalpha 191
10.1.3 可打印字符测试函数 isgraph 192
10.1.4 测试大小写字母函数 islower 和 isupper 193
10.1.5 测试是否是数字 isxdigit 193
10.1.6 测试字符是否为符号函数 ispunct 194
10.1.7 其他字符测试函数 195
10.2 字符测试函数综合实例 195
10.3 字符串转换 197
10.3.1 C 程序中的字符串 197
10.3.2 字符串转换成浮点型函数 atof 197

10.3.3 字符串转换成整型函数 atoi……198
10.3.4 字符串转换成长整型函数 atol……199
10.3.5 将浮点型转换成字符串函数 ecvt……200
10.3.6 字母的大写小转换函数 tolower 和 toupper……201
10.3.7 其他字符串转换函数……202
10.4 字符串比较……203
10.4.1 字符串比较函数 bcmp……203
10.4.2 字符串大小比较函数 memcmp……204
10.4.3 忽略大小写比较字符串函数 strncasecmp……204
10.5 字符串复制……205
10.5.1 字符串复制函数 bcopy……205
10.5.2 字符串复制函数 memccpy……206
10.5.3 复制字符串函数 strcpy……207
10.5.4 字符串复制函数 strncpy……208
10.6 字符串的清理与填充……208
10.6.1 字符串清理函数 bzero……209
10.6.2 字符串填充函数 memset……209
10.7 字符串查找……210
10.7.1 字符查找函数 index 与 rindex……210
10.7.2 字符查找函数 memchr……211
10.7.3 字符查找函数 strchr 与 strrchr……211
10.8 字符串的连接与分割……212
10.8.1 字符串连接函数 strcat……212
10.8.2 字符串分割函数 strtok……213
10.9 综合练习……214
10.10 小结……216
10.11 习题……216
第 11 章 结构体（教学视频：49 分钟）……218
11.1 结构体的操作……218
11.1.1 结构体的理解……218
11.1.2 结构体的定义……218
11.1.3 结构体的访问……220
11.1.4 结构体数组……222
11.1.5 结构体使用实例……222
11.2 结构体与指针……224
11.2.1 结构体指针的定义……225
11.2.2 结构体指针的访问……225
11.2.3 结构体作为函数的参数……227
11.2.4 结构体指针作为函数的参数……228
11.2.5 结构体作为函数的返回值……230

11.2.6 结构体指针作为函数的返回值 …… 231
11.3 综合练习 …… 233
11.3.1 程序的需求分析 …… 233
11.3.2 程序中的函数 …… 234
11.3.3 程序中的结构体与全局变量 …… 234
11.3.4 头文件和函数声明 …… 234
11.3.5 定义结构体和全局变量 …… 235
11.3.6 显示学生信息的函数 …… 235
11.3.7 程序的选择菜单 …… 235
11.3.8 学生信息输入函数 …… 237
11.3.9 按姓名查找函数 …… 237
11.3.10 删除学生函数 …… 238
11.3.11 按年龄查找函数 …… 238
11.3.12 按身高查找函数 …… 239
11.3.13 按性别查找函数 …… 240
11.3.14 程序的主函数 …… 240
11.3.15 程序的运行和调试 …… 242
11.4 小结 …… 243
11.5 习题 …… 243

第 12 章 时间函数（教学视频：23 分钟） …… 245

12.1 常用时间函数 …… 245
12.1.1 返回时间函数 time …… 245
12.2.2 取当前时间函数 gmtime …… 246
11.2.3 字符串格式时间函数 ctime …… 248
12.2.4 字符串格式时间函数 asctime …… 249
12.1.5 取得当地时间函数 localtime …… 250
12.1.6 将时间转换成秒数函数 mktime …… 251
12.1.7 取得目前的时间函数 gettimeofday …… 251
12.1.8 设置当前时间函数 settimeofday …… 253
12.2 综合练习 …… 254
12.3 小结 …… 254
12.4 习题 …… 255

第五篇 文件与目录篇

第 13 章 目录与文件（教学视频：115 分钟） …… 256

13.1 文件操作的权限 …… 256
13.2 错误处理与错误号 …… 257
13.2.1 错误定义的理解 …… 257
13.2.2 用错误常数显示错误信息 …… 258

13.2.3 用错误序号显示错误信息……259
13.3 创建与删除目录……259
13.3.1 创建目录函数 mkdir……260
13.3.2 删除目录函数 rmdir……261
13.4 文件的创建与删除……262
13.4.1 创建文件函数 creat……262
13.4.2 删除文件函数 remove……263
14.4.3 建立临时文件函数 mkstemp……264
13.5 文件的打开与关闭……265
13.5.1 打开文件函数 open……266
13.5.2 文件打开方式的设置……266
13.5.3 打开文件的权限……266
13.5.4 文件打开实例……266
13.5.5 关闭文件函数 close……267
13.6 文件读写……269
13.6.1 在文件中写字符串函数 write……269
13.6.2 读取文件函数 read……270
13.6.3 文件读写位置的移动……272
13.6.4 将缓冲区数据写入到磁盘函数 sync……273
13.6.5 将缓冲区数据写入到磁盘函数 fsync……275
13.7 文件锁定……276
13.7.1 文件锁定的理解……276
13.7.2 文件锁定函数 flock……277
13.7.3 文件锁定函数 fcntl……278
13.7.4 文件锁定函数 fcntl 使用实例……279
13.8 文件的移动与复制……281
13.8.1 文件的移动函数 rename……281
13.8.2 rename 函数使用实例……281
13.8.3 文件复制实例……282
13.9 综合练习……283
13.9.1 程序功能分析……284
13.9.2 程序的函数……284
13.9.3 包含文件……284
13.9.4 数据的定义……285
13.9.5 菜单函数……285
13.9.6 显示电话信息函数……286
13.9.7 添加电话号码函数……286
13.9.8 按姓名查找函数……287
13.9.9 删除电话号码函数……287
13.9.10 保存到文件函数……288
13.9.11 从文件导入信息函数……288

13.9.12 主函数 …… 289
13.10 小结 …… 291
13.11 习题 …… 291
第 14 章 文件 I/O（教学视频：42 分钟） …… 293
14.1 文件的打开与关闭 …… 293
14.1.1 文件打开函数 fopen …… 293
14.1.2 文件打开函数 fdopen …… 295
14.1.3 打开文件函数 freopen …… 296
14.1.4 关闭文件函数 fclose …… 297
14.2 文件的读写 …… 298
14.2.1 字符写入函数 putc 与 fputc …… 298
14.2.2 向文件中写入字符串函数 fputs …… 299
14.2.3 数据写入函数 fwrite …… 300
14.3 综合练习 …… 300
14.4 小结 …… 302
14.5 习题 …… 302

第六篇 网络编程篇

第 15 章 网络编程（教学视频：71 分钟） …… 303
15.1 网络编程的基本概念 …… 303
15.1.1 IP 地址 …… 303
15.1.2 端口 …… 304
15.1.3 域名 …… 304
15.1.4 TCP 与 UDP …… 304
15.2 套接字 …… 305
15.2.1 什么是套接字 …… 305
15.2.2 套接字相关的数据类型 …… 305
15.2.3 套接字类型 …… 306
15.3 域名与 IP 地址 …… 306
15.3.1 用域名取得主机的 IP 地址 …… 306
15.3.2 用 IP 地址返回域名 …… 308
15.4 网络协议 …… 309
15.4.1 由协议名取得协议数据 …… 309
15.4.2 由协议编号取得协议信息 …… 311
15.4.3 取得系统支持的所有协议 …… 312
15.5 网络服务 …… 313
15.5.1 取得系统支持的网络服务 …… 313
15.5.2 用名称取得系统所支持的服务 …… 314
15.5.3 由端口取得服务名称 …… 316

15.6 网络地址的转换 ····317
15.6.1 将网络地址转换成整型 ····317
15.6.2 将长整型数地址转换成网络地址 ····318
15.6.3 主机字符顺序与网络字符顺序的转换 ····319
15.7 错误处理 ····319
15.7.1 herror 函数显示错误 ····319
15.7.2 捕获错误编号 ····320
15.8 综合练习 ····321
15.9 小结 ····321
15.10 习题 ····322
第 16 章 无连接的套接字通信（ 教学视频：22 分钟） ····323
16.1 socket 套接字 ····323
16.1.1 建立 socket ····323
16.1.2 取得 socket 状态 ····325
16.1.3 设置 socket 状态 ····326
16.2 无连接的套接字通信 ····327
16.2.1 工作流程 ····328
16.2.1 信息发送函数 sendto ····328
16.2.2 信息接收函数 recvfrom ····329
16.3 综合练习 ····329
16.3.1 无连接套接字通信客户端 ····329
16.3.2 无连接套接字通信服务器 ····331
16.3.3 测试 UDP 通信程序 ····332
16.4 小结 ····333
16.5 习题 ····333
第 17 章 面向连接的套接字通信（ 教学视频：76 分钟） ····335
17.1 面向连接的套接字通信工作流程 ····335
17.2 绑定端口 ····336
17.2.1 绑定端口函数 bind ····336
17.2.2 bind 函数绑定端口实例 ····337
17.3 监听与连接 ····338
17.3.1 等待监听函数 listen ····338
17.3.2 listen 函数使用实例 ····338
17.3.3 接受连接函数 accept ····340
17.3.4 accept 函数使用实例 ····340
17.3.5 请求连接函数 connet ····342
17.3.6 connet 函数使用实例 ····342
17.4 数据的发送与接收 ····343
17.4.1 数据接收函数 recv ····343

17.4.2 recv 函数使用实例 …… 344
17.4.3 信息发送函数 send …… 345
17.4.4 数据传输函数 write 与 read …… 346
17.4.5 read 函数接收数据实例 …… 346
17.5 综合练习 …… 348
17.5.1 服务器程序 …… 348
17.5.2 客户端程序 …… 350
17.5.3 实例程序测试 …… 352
17.6 小结 …… 352
17.7 习题 …… 353

第七篇 图形界面开发篇

第 18 章 Linux 图形界面开发基础（ 教学视频：28 分钟） …… 354

18.1 X Window 系统简介 …… 354
18.1.1 X Window 的历史 …… 354
18.1.2 X Window 体系结构 …… 355
18.2 Linux 桌面环境 …… 355
18.2.1 KDE …… 355
18.2.2 GNOME …… 356
18.3 Linux 下的图形程序开发环境 …… 357
18.3.1 GTK+简介 …… 357
18.3.2 yum 安装 GTK …… 357
18.3.3 用软件包管理器安装 GTK …… 358
18.4 常见问题 …… 359
18.4.1 找不到 gtk.h 包含文件的错误 …… 359
18.4.2 pkg-config 参数有什么作用 …… 359
18.5 综合练习 …… 359
18.5.1 编写程序代码 …… 359
18.5.2 编译程序 …… 360
18.5.3 运行程序 …… 361
18.6 小结 …… 361
18.7 习题 …… 361

第 19 章 图形界面编程（ 教学视频：35 分钟） …… 363

19.1 窗口 …… 363
19.1.1 最简单的窗口程序 …… 363
19.1.2 窗口的构建过程 …… 364
19.1.3 新建窗口函数 gtk_window_new …… 364
19.1.4 设置窗口的标题 …… 365
19.1.5 设置窗口的大小与位置 …… 366

19.2 标签……367
19.2.1 建立一个标签……367
19.2.2 将标签添加到窗口中……367
19.2.3 设置与获取标签的文本……368
19.3 按钮……369
19.3.1 添加一个按钮……370
19.3.2 取得与设置按钮的标签……371
19.4 文本框……372
19.4.1 建立文本框……372
19.4.2 获取与设置文本框数据……373
19.5 综合练习……374
19.6 小结……375
19.7 习题……375
第 20 章 界面布局（ 教学视频：67 分钟）……377
20.1 表格……377
20.1.1 表格的建立……377
20.1.2 表格中添加元件……378
20.1.3 表格使用实例……379
20.1.4 单元格的合并……380
20.1.5 表格中添加表格……381
20.2 垂直框……383
20.2.1 添加垂直框……383
20.2.2 垂直框中添加元件……384
20.3 水平窗格……385
20.3.1 新建一个窗格……385
20.3.2 窗格中添加元件……387
20.4 Glade……388
20.4.1 Glade 简介……388
20.4.2 使用 Glade 设计程序界面……388
20.4.3 Glade 设计窗口……389
20.4.4 添加表格……392
20.4.5 添加标签……392
20.4.6 添加文本框……395
20.4.7 添加按钮……396
20.4.8 添加信号与事件……397
20.4.9 Glade 实例……398
20.5 综合练习……399
20.6 小结……400
20.7 习题……401

第 21 章　信号与事件（ 教学视频：18 分钟）……403
21.1　信号……403
21.1.1　添加信号……403
21.1.2　事件函数的参数……404
21.1.3　在函数中使用全局变量……405
21.2　GTK 事件……407
21.3　实例：加法计算器……407
21.3.1　程序功能的分析……407
21.3.2　程序实现……407
21.4　综合练习……409
21.4.1　程序的功能分析……409
21.4.2　程序的功能模块……410
21.4.3　包含头文件与定义全局变量……411
21.4.4　单击按钮输入函数……411
21.4.5　加法运算函数……411
21.4.6　减法运算函数……412
21.4.7　乘法运算函数……413
21.4.8　除法运算函数……413
21.4.9　输入小数点函数……414
21.4.10　清除数据函数……414
21.4.11　界面构建函数……414
21.4.12　添加事件函数……416
21.4.13　主函数……417
21.4.14　程序的编译和运行……417
21.5　小结……418
21.6　习题……418

第一篇 Linux 基础篇

第 1 章 Linux 编程平台

Linux 操作系统为开发者提供了非常丰富的开发工具和友好易用的开发环境，使 Linux 成为最优秀的开发平台。本章将介绍如何快速搭建 Linux 下的软件开发平台，内容包括 Linux 发行版的介绍、Linux 操作系统的安装，以及如何编写和运行第一个简单的 C 程序。通过本章学习，将使读者熟悉 Linux 开发平台及 Linux 下 C 程序的开发过程。本章的主要内容如下：

- Linux 发行版的基本概念；
- Linux 操作系统的安装；
- VMware 虚拟机的应用；
- 编写第一个 C 程序。

1.1 Linux 发行版的介绍

所谓 Linux 发行版，指的是以 Linux 内核为中心，再集成开发各种系统管理软件或应用软件，组成一套完整的操作系统。这种在内核上开发出的可供用户使用的 Linux 系统就是 Linux 发行版。Linux 内核指的是可提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。内核虽然提供了操作系统的最核心功能，但是，它并不是一套完整的操作系统，在计算机上无法单独使用，必须利用 Linux 发行版中的附加软件才能够正常工作。

1.1.1 Red Hat Linux

Red Hat Linux 是最成熟的一种 Linux 发行版，在服务器和个人桌面、嵌入式开发方面有着广泛的应用。Red Hat Linux 的第一个版本发布于 1994 年，目前已经发布了数十个版本。目前，RedHat Linux 分为企业版和社区版（Fedora Linux），企业版的用户必须进行订阅才能得到升级等服务，而社区版则可以完全免费使用。本书采用 Red Hat Linux 9.0 和 Fedora 8 为蓝本进行开发和讲解。

Red Hat Linux 不但具有传统的纯文本命令行操作界面，而且有着非常友好的图形用户界面。在应用软件和开发工具等方面，Red Hat Linux 9 提供了 OpenOffice 办公软件、Mozilla 浏览器、网站和数据库服务器、QT 开发平台等开源工具。

1.1.2 Debian Linux

Debian Linux 是由 GNU 组织开发的一套 Linux 发行版，在个人计算机和商业领域有着广泛的应用。Debian Linux 最大的特点是集成了数万个软件包，用户可以在网络中非常方便地管理和升级所需要的软件。Debian Linux 的软件管理有着良好集成度，所有的软件都是由官方编译成 deb 包后发布到升级服务器中的。deb 包的管理方式很好地解决了软件包之间的复杂依赖性问题，使 Debian Linux 更加稳定强健。apt-get 是 deb 包的管理指令，利用此工具使软件的下载和安装变得非常简单，一个简单的命令就可以完成软件的下载与安装等全部操作。

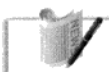
1.2 安装 Linux

Linux 的系统安装与 Windows 安装相似，用安装光盘可以方便地安装 Linux 系统。与 Windows 系统安装不同的是，需要注意系统的引导与分区。特别是与 Windows 一起安装双系统时，需要注意分区和引导的问题。本节将讲解 Linux 系统的分区和安装等内容。

1.2.1 Linux 的分区格式

Linux 的分区格式与 Windows 是完全不同的。Windows 使用的是 FAT32 或 NTFS 分区格式，而 Linux 使用 Ext2 或 Ext3 分区格式。Linux 至少需要两个硬盘分区，即除了根分区以外，还需要一个 swap（交换）分区。下面给出 Linux 下常见分区格式的概要说明。

- Ext2：Ext2 是 Linux 系统中标准的文件系统，是 Linux 中使用最多的一种文件系统。Ext2 是专门为 Linux 设计的分区格式，有极快的访问速度和极小的 CPU 占用率。Ext2 既可以用于硬盘，也可以应用在软盘、移动硬盘、闪存等移动存储设备上。
- Ext3：Ext3 是一种日志式文件系统（Journal File System），是 Ext2 的下一代分区格式，在 Ext2 的基础上增加了日志功能。Ext3 的最大特点是将整个磁盘的写入动作以日志的形式完整地记录在磁盘的某个区域，以便在需要时回溯追踪。当文件系统中的某个操作过程出现异常时，可以根据记录的日志直接找回被中断的部分。这种日志式的分区格式使文件的存储非常稳定，被广泛应用在 Linux 系统中。
- swap：swap 是 Linux 系统专门用于内存交换的硬盘分区，相当于 windows 系统中的虚拟内存。swap 分区的大小一般是内存的 1~2 倍。在内存不够用时，Linux 会将不活动的内存数据暂时保存到交换分区中。



提示：Linux 支持多种分区格式，Linux 默认使用的分区格式为 ext2 或 ext3。

1.2.2 在 Windows 系统中划分 Linux 分区

在安装 Linux 时，一般需要将 Linux 与 Windows 系统安装在同一台计算机中。这就需要在已经安装好的 Windows 系统中为 Linux 划分分区。使用 Norton PartitionMagic 分区软件，可以方便地在 Windows 系统中的图形界面上对硬盘进行分区。下面的步骤将描述使用 Norton PartitionMagic 在硬盘中划分 Linux 分区，用于安装 Linux 系统。

① 在网络上下载“Norton PartitionMagic”分区软件并且安装。然后，打开 Norton PartitionMagic 进行硬盘分区。Norton PartitionMagic 的界面如图 1-1 所示。

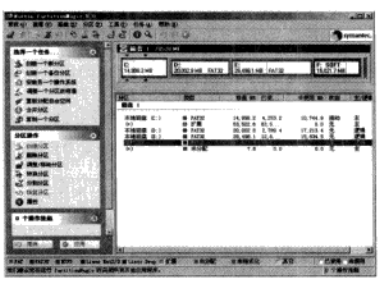


图 1-1 使用 Norton PartitionMagic 进行硬盘分区

注意：在进行硬盘分区时，需要关闭其他正在运行的软件。

② 查看计算机中的现有分区。在 Norton PartitionMagic 的分区列表中，显示了计算机中当前的硬盘分区情况和分区使用情况，如图 1-2 所示。

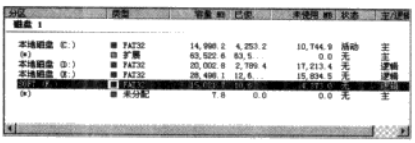


图 1-2　在 Norton PartitionMagic 查看计算机的分区

③ 删除最后一个分区。

④ 在分区列表中，单击选择最后一个分区。在如图 1-1 所示的界面中，单击“分区操作”标签中的“删除分区”工具，出现的删除分区对话框如图 1-3 所示。

⑤ 单击“确定”按钮，确定删除分区。

警告：删除分区时，会删除分区上的所有数据。重新分区后，分区上的原有数据将不能再恢复。所有的操作需要谨慎，确认不会出现错误。

⑥ 如图 1-4 所示，删除一个分区以后，硬盘中有了 15GB 的未分配空间。

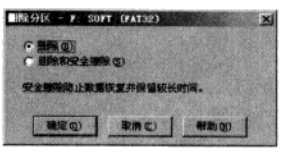


图 1-3　“删除分区”对话框

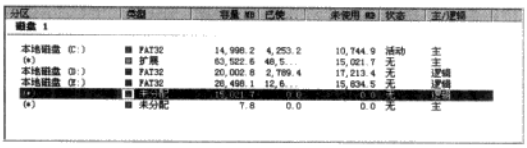


图 1-4　硬盘删除分区以后的分区列表

⑦ 新建一个 swap 交换分区。在分区列表中，单击选择未分区的硬盘空间。在如图 1-1 所示的界面中，单击“分区操作”标签中的“创建分区”工具，“创建分区”工具如图 1-5 所示。

⑧ 在“创建为”下拉菜单中，单击选择“逻辑分区”选项。在“分区类型”下拉菜单中，单击选择“Linux Swap”选项。在“大小”下拉菜单中输入分区的大小 1024。swap 交互分区的大小应该是内存大小的 2 倍。然后单击“确定”按钮完成设置。

⑨ 创建一个引导区。在 Linux 系统中，需要有引导程序来引导一个系统。引导程序通常存放在独立的分区中，大小一般是 100MB~200MB。如步骤 7 所示，新建一个分区，“创建分区”对话框如图 1-6 所示。

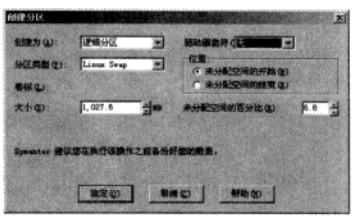


图 1-5　新建一个 swap 分区

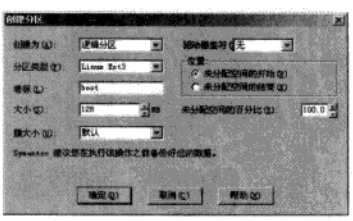


图 1-6　创建一个引导分区

⑩ 如图 1-6 所示，在“创建为”下拉菜单中，单击选择“逻辑分区”选项。在“分区类

型”下拉菜单中，单击选择“Linux Ext3”选项。在“卷标”文本框中输入卷标“boot”。卷标是在文件管理中可以看到的分区名称。在“大小”文本框中输入这个分区的大小 128。然后单击“确定”按钮完成设置。

⑪ 用上面的两个步骤，新建一个主分区。分区的类型为“Linux Ext3”，卷标为“linux”。分区大小为硬盘中的所有剩余空间。

⑫ 在 Norton PartitionMagic 的分区列表中，创建的 Linux 分区如图 1-7 所示。这样，硬盘的前三个分区是 Windows 分区，后三个分区是 Linux 分区，实现了同一个硬盘安装 Windows 分区与 Linux 分区双系统。

⑬ 上面步骤中的分区操作，只是进行分区设置，并没有执行分区。在 Norton PartitionMagic 左下角的操作列表中有已经设置的操作列表。如图 1-8 所示。单击“确定”按钮，完成分区的操作。

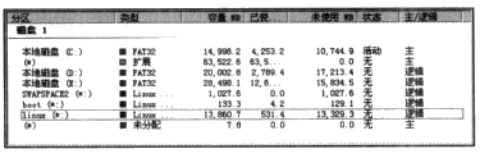


图 1-7 分区列表

图 1-8 分区的操作列区

⑭ Norton PartitionMagic 会自动把所有的操作应用到硬盘，并自动格式化分区。然后 Norton PartitionMagic 会提示重启计算机。

经过上面的操作步骤，在已经安装 Windows 系统的硬盘中又另外划分出三个 Linux 分区。

1.2.3 安装 Red Hat 9.0

Red Hat Linux 是使用最广泛的 Linux 发行版。可以将 Red Hat Linux 与 Windows 一起以双系统的形式安装到计算机中。本节将介绍用安装光盘进行 Linux 系统的安装与设置的详细过程。

① 进行上一节的分区操作，在硬盘中划分三个分区以供 Linux 安装。

② 将 Red Hat Linux 的安装光盘装入光驱。重启计算机，在启动时按下“Del”键进入“BIOS”，设置计算机从光盘启动。然后保存设置，重启计算机。

注意：不同的计算机进入“BIOS”时，使用的按键可能有所不同，常用的键有 Del 键和 F2 键。

图 1-9 Rad Hat Linux 9.0 的安装启动界面

③ 计算机从光盘启动，进行安装，如图 1-9 是 Rad Hat Linux 9.0 的安装启动界面。

④ 安装会停在如下所示的一个提示上面。提示用户如果要安装系统则按“Enter”键。这时，需要按“Enter”键确认继续 Linux 的安装。

```
To install or upgrade Red Hat Linux in Graphical mode,
Press the <ENTER> key.
```

⑤ 检查光盘的完整性。在 Red Hat Linux 9.0 安装之前，为了防止光盘错误中断安装，需要对光盘的完整性进行检查。安装过程会停止在如图 1-10 所示的界面上。这时用“Tab”键可以在按钮之间移动，按“Enter”键确认这个按钮。在图 1-10 所示的界面中，“OK”按钮是进行光盘完整性检查，“Skip”按钮是跳过完整性检查。按“Tab”键将光标移动到“OK”按钮上，然后按“Enter”键确认。

提示：如果使用光盘安装 Linux 操作系统，最好进行本步骤的光盘完整性检查，防止由于光盘损坏导致数据无法读出，而最终导致安装失败。

⑥ 完成光盘完整性检查以后，Red Hat Linux 9.0 的图形安装界面如图 1-11 所示。然后单击“NEXT”（下一步）按钮。在这一步以后的步骤都可以用鼠标来操作。如果还没有正常加载鼠标，可以用“Tab”键在窗口上移动选择的内容，按“Enter”键激活选择的内容。按“Space”键选择单选按钮或复选框。

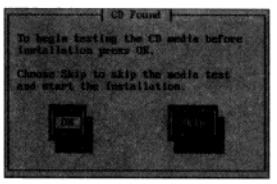


图 1-10　光盘完整性检查

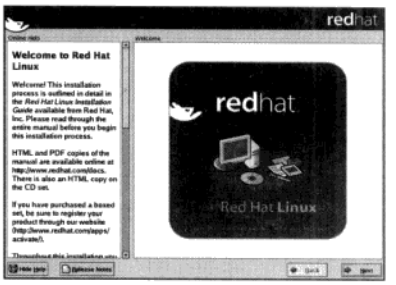


图 1-11　Red Hat Linux 9.0 的图形安装界面

⑦ 选择系统语言。Red Hat Linux 9.0 安装界面的默认语言是英文。在这以前的步骤都是英文界面。如图 1-12 所示是“Language Selection”窗口。在语言列表中单击选择“Chinese (Simplified)（简体中文）”选项。然后单击“Next”按钮。后面的安装会在中文界面中完成。

⑧ 选择键盘。如图 1-13 所示是选择键盘界面。Red Hat Linux 9.0 是一个支持多种语言的系统，在不同国家中可能使用不同的键盘。在列表中单击选择“U.S. English”选项。然后单击“下一步”按钮。

⑨ 选择鼠标。Red Hat Linux 9.0 在安装时，可能不能正确检测到用户使用的鼠标类型，需要选择自己的鼠标类型。如图 1-14 所示，选择正在使用的鼠标类型。鼠标可能是串口、USB、PS/2 等接口，可能有多个键，需要对这些类型进行选择。然后单击“下一步”按钮。

图 1-12　选择安装语言

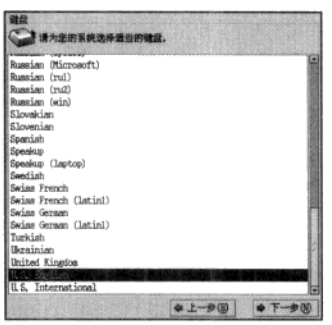


图 1-13　选择键盘类型

图 1-14　选择鼠标类型

⑩ 选择安装类型。Red Hat Linux 9.0 可以安装个人桌面、服务器、工作站等不同的环境。如图 1-15 所示，单击“定制”单选按钮，在安装时选择开发所必需的软件。然后单击“下一步”按钮。

⑪ 选择分区方式。在上一节中已经为 Linux 划分了三个分区，在安装时需要手动挂载已经划分的 Linux 分区。单击选择“用 Disk Druid 手工分区”单选按钮，如图 1-16 所示。然后单

击“下一步”按钮。

⑫ 如图 1-17 所示是硬盘的分区列表。上方以列表的方式显示各个硬盘的分区情况，下方显示分区的列表。

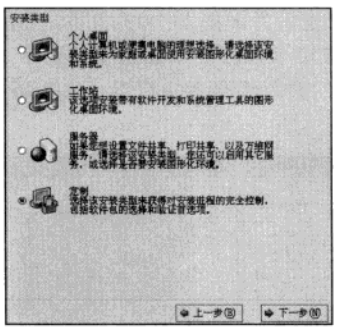


图 1-15 选择安装环境

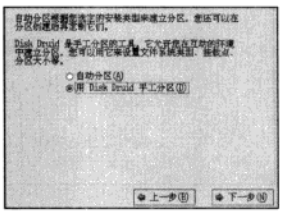


图 1-16 选择分区方法

图 1-17 分区列表

⑬ 在分区列表中，单击选择“/dev/hda1”分区，然后单击“编辑”按钮，挂载这一个分区。如图 1-18 是“编辑分区”对话框，单击“挂载点”下拉菜单，选择挂载点“/boot”，“文件系统类型”下拉菜单中单击选择“ext3”选项。然后单击“确定”按钮完成设置。

注意：在 Linux 系统中，分区并不像 Windows 系统一样排列成盘符，而是挂载到不同的挂载点上。最高一级挂载点为“/”。可以把一个分区挂载到任何一个文件夹下。

⑭ 用与上一步骤同样的方法，将 swap 分区的文件系统类型设置为“swap”，此时挂载点不可设置，如图 1-19 所示。将最大分区的文件系统类型设置为“ext3”，挂载点设置为“/”。

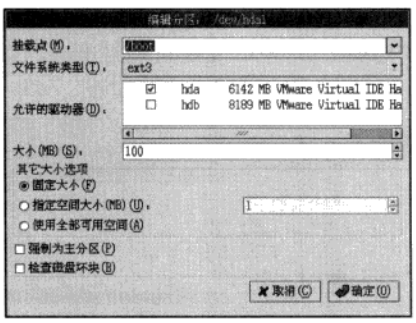


图 1-18 挂载引导分区

图 1-19 挂载 swap 交换分区

⑮ 设置 GRUB 引导。Red Hat Linux 9.0 安装以后在引导时，需要一个软件来进行引导。GRUB 是一个系统启动引导软件。如图 1-20 所示是 GRUB 的引导设置，在引导列表中单击“默认”复选框中的一个系统作为默认启动的系统。然后单击“下一步”按钮。

提示：在 Linux 下的引导程序还有 Lilo（Linux Loader），但是由于存在一些缺点，在新版的 Linux 发行版中默认都采用 GRUB 作为引导程序。

⑯ 设置网络。“网络设备”列表显示的是计算机中可以使用的网卡。单击一个网卡，设置网络属性，如图 1-21 所示。如果网络支持“DHCP”自动获取 IP，可以选择“通过 DHCP

自动被设置”单选按钮。否则需要单击选择“手工设置”单选按钮，然后在 IP 列表中分别设置可以使用的 IP 地址、网关、DNS 等信息。然后单击“下一步”按钮。

⑰ 配置防火墙。如图 1-22 所示，在“选择系统的安全级别”单选按钮中单击选择“中级”选项。然后在“允许进入”列表中单击选择系统中允许访问的服务。然后单击“下一步”按钮。

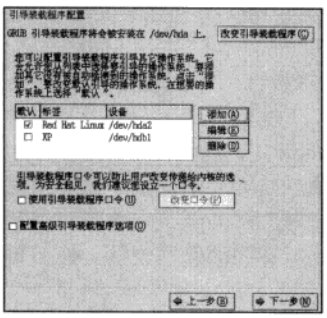


图 1-20　设置 GRUB 引导

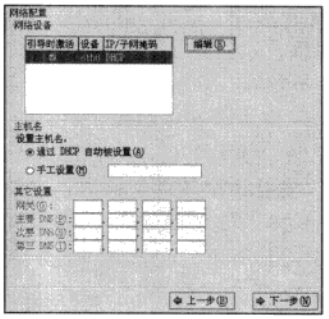


图 1-21　设置网络

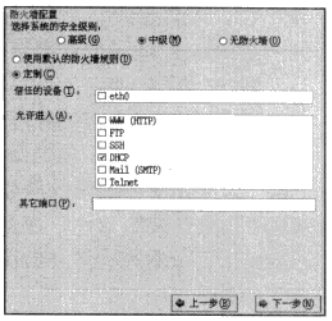


图 1-22　配置防火墙

提示：如果以学习为目的，对网络服务和防火墙不太了解的话，推荐使用“无防火墙”设置，以免防火墙将一些服务屏蔽掉，导致服务无法被外界访问。

⑱ 选择系统默认语言。Red Hat Linux 9.0 在系统安装时，可以安装多种可供选择的语言。在如图 1-23 所示的语言列表中，单击选择“Chinese （P.R. of China）”选项作为系统的默认语言。然后单击“下一步”按钮。

⑲ 选择所在的时区。Linux 系统中使用了 UTC 时间，需要设置当地的时区转换成本地时间。如图 1-24 所示，在位置列表中单击选择“亚洲/上海”选项，然后单击“下一步”按钮。

注意：UTC 时间（Universal Time Coordinated）是协调世界时间，以英国格林威治平均时作为零时区，北京时间比 UTC 时间早 8 小时。在编程时常用到 UTC 时间。

⑳ 设置根用户口令。在 Linux 系统中，“root”用户是最高权限用户，被称为根用户。如图 1-25 所示，在“根口令”文本框中输入密码，然后在“确认”文本框中再次输入相同的密码。然后单击“下一步”按钮。

图 1-23　选择默认语言

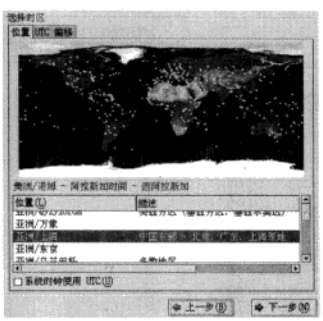


图 1-24　选择所在的时间

图 1-25　设置根用户口令

注意：此处的密码必须为 6 个字符以上，否则无法继续往下进行。

㉑ 选择软件包组。如图 1-26 所示，根据需要选择需要安装的软件包组。在 Red Hat Linux 9.0 中需要使用的软件包组如下所示。

- X 窗口系统：Linux 系统的图形桌面是一个应用软件，如果不安装 X 窗口系统，则不能进入桌面。
- GNOME 桌面环境：Linux 系统中，桌面、任务栏、开始菜单等工具都是桌面环境软件。GNOME 是一个桌面管理软件。
- KDE 桌面环境：KDE 是一个与 GNOME 类似的桌面管理软件。
- 编辑器：如文本编辑器、记事本等文本编辑工具。
- 图形化互联网：包括浏览器、聊天工具、FTP 工具、Email 工具等网络常用软件。

㉒ 选择软件包组中的软件。安装一个软件包组时，会默认安装需要的工具。需要根据使用的需要选择安装的软件。在图 1-26 所示的窗口中，在“编辑器”软件包组中单击“细节”按钮选择需要安装的软件。如图 1-27 所示，在软件列表中选择所有的编辑器软件。然后单击“确定”按钮。

图 1-26　设置需要安装的工具

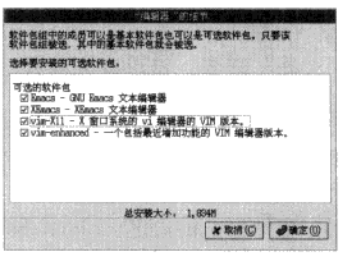


图 1-27　详细选择一组工具中的软件

㉓ 向下拖动滚动条，在软件包组列表中，单击选择编程开发所需要的“开发工具”、“X 软件开发”、“GNOME 软件开发”、“KDE 软件开发”等软件包组。如图 1-28 所示，然后单击“下一步”按钮。

㉔ 经过这些设置以后，Red Hat Linux 9.0 会自动安装已经选择的软件。如图 1-29 所示，进度条和进度列表显示了当前的安装进度和所需的时间。“正在安装的软件包”标签中显示当前正在安装的软件包信息。这一过程大约需要 10 分钟。

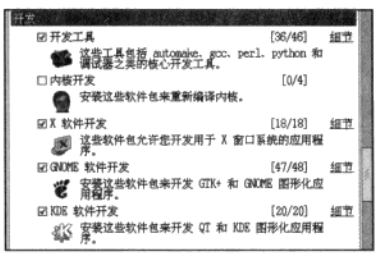


图 1-28　选择需要使用编程开发工具

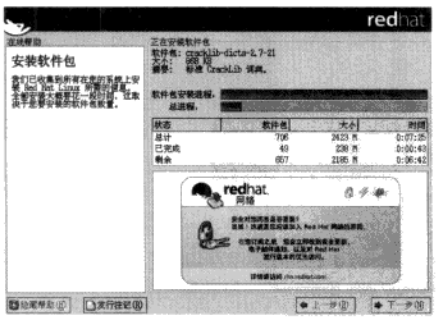


图 1-29　Red Hat Linux 9.0 正在安装系统

㉕ 设置引导盘。Red Hat Linux 9.0 安装完毕以后，会提示制作引导盘。引导盘是在一张

软盘上写入 Red Hat Linux 9.0 的引导信息，当系统无法引导时可以使用引导盘来启动系统。如果计算机中没有软驱，可以单击选择“否，我不想创建引导盘”单选按钮，如图 1-30 所示。然后单击“下一步”按钮。

㉖ 选择和设置显卡。Linux 系统在安装时，可能无法正确检测到显卡。如图 1-31 所示，在显卡列表中单击选择计算机的显卡类型。在“视频卡内存”下拉菜单中单击选择显卡显存的大小。然后单击“下一步”按钮。

㉗ 设置显示器类型。如图 1-32 所示，在显示器列表中单击选择显示器的品牌和型号，然后单击“下一步”按钮。

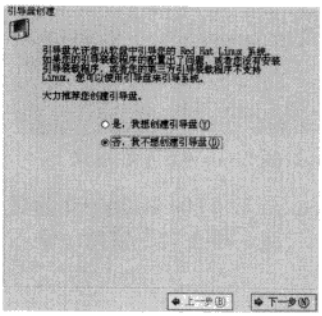


图 1-30　选择是否创建启动盘

图 1-31　选择和设置显卡

图 1-32　选择和设置显示器

㉘ 设置显示器分辨率。如图 1-33 所示，在“色彩深度”下拉菜单中，单击选择“高色(16 位)”选项，在“屏幕分辨率”下拉菜单中单击选择适合自己显示器的分辨率。然后单击“下一步”按钮。

㉙ 完成安装。系统完成安装的界面如图 1-34 所示。这时单击“退出”按钮，会自动重启计算机进入 Red Hat Linux 9.0 系统。

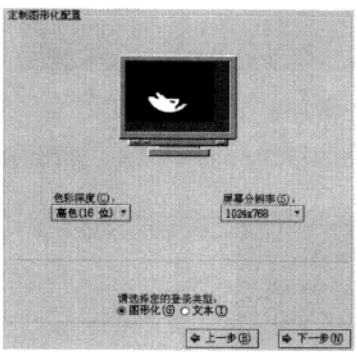


图 1-33　设置显示器的分辨率

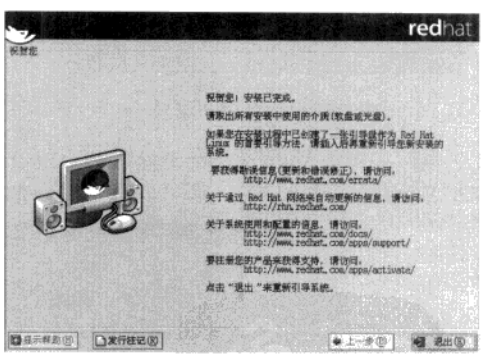


图 1-34　完成 Red Hat Linux 9.0 的安装

㉚ 选择需要启动的系统。安装了 GRUB 来引导双系统的计算机，会在启动时等待几秒时间，让用户选择需要启动的系统。如图 1-35 所示。按“向上”或“向下”键可以选择系统，然后按“Enter”键启动选择的系统。

㉛ Linux 系统在启动时，会详细显示硬件和软件的启动情况。如图 1-36 所示，是 Red Hat Linux 9.0 的启动界面。

㉜ 在第一次登录时，Red Hat Linux 9.0 需要设置一个用户。如图 1-37 所示，在“用户名”

和“全名”文本框中输入一个用户名。在“口令”和“确认口令”文本框中两次输入密码。然后单击“前进”按钮。这时提示设置“声卡”、“用户信息”等内容，使用默认设置单击“前进”按钮即可。

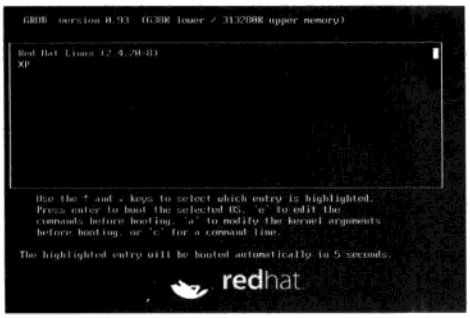


图 1-35　GRUB 引导系统

图 1-36　Red Hat Linux 9.0 正在启动

33 完成用户设置以后，就可以登录系统了。如图 1-38 所示，是 Red Hat Linux 9.0 的登录界面。在“用户名”文本框中输入用户名“root”，然后按“Enter”键。再在口令窗口中输入这个用户的口令。然后按“Enter”键进入系统。

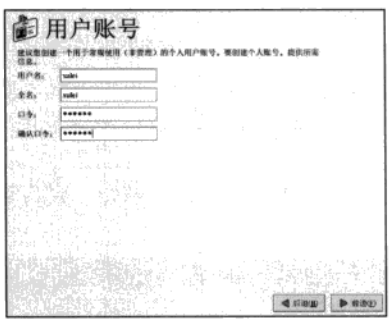


图 1-37　设置用户账号

图 1-38　用户登录界面

34 如图 1-39 所示，是 Red Hat Linux 9.0 的用户桌面。在 Red Hat Linux 9.0 中，常用的操作和 Windows 是相似的，都可以用鼠标来完成。

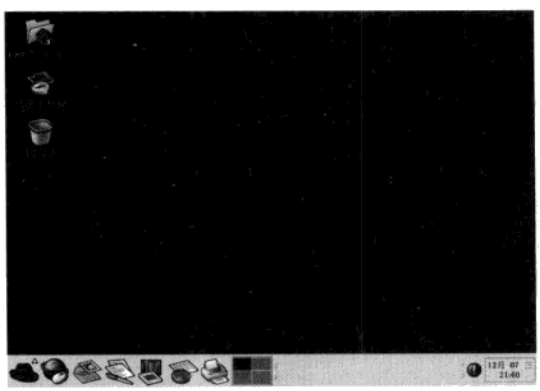


图 1-39　Red Hat Linux 9.0 的桌面

1.2.4　GRUB 在 Windows 分区中引导 Linux

安装完成 Windows 与 Linux 的双系统以后，GRUB 软件可以实现系统启动时两个操作系统的选择和引导。但重新安装 Windows 系统以后，会删除硬盘中的引导信息。这时只能默认启动 Windows 系统，而无法对 Linux 系统进行引导。需要在 Windows 系统中安装 GRUB 对 Linux 系统进行引导。

① 在 Windows 系统中安装 GRUB。GRUB 是一个 DOS 下的系统引导软件，很多常用的系统备份工具中包含有 GRUB 引导工具。例如一键 GHOST、矮人 DOS 工具箱等软件中的 GRUB 可以用来引导 Linux 系统。可以从网络上下载一键 GHOST 软件，并且按照默认的方式进行安装。

② 查看 Red Hat Linux 9.0 的引导信息。启动 Red Hat Linux 9.0。在如图 1-39 所示的桌面上，双击打开“root 的主目录”图标。如图 1-40 所示是 Red Hat Linux 9.0 的文件管理。在“位置”文本框中输入 GRUB 的目录“/boot/grub”。然后按“Enter”键进入。

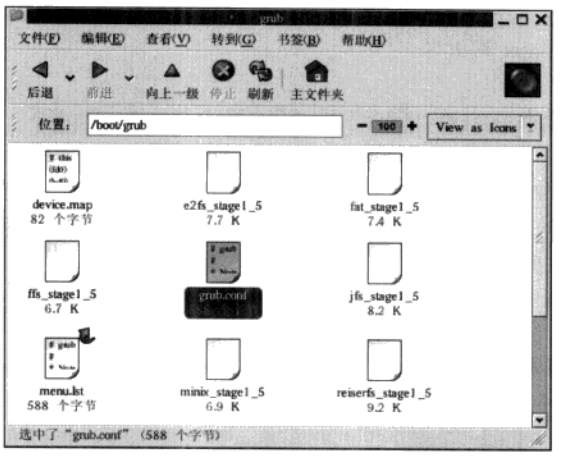


图 1-40　在 Red Hat Linux 9.0 中打开 GRUB 的文件夹

③ 双击打开文件“grub.conf”，查看 GRUB 的引导信息。如以下代码所示，GRUB 分别标记了每个系统所在的分区和启动方式。

```
default=0                                    #注释：默认启动的信息。
timeout=10
splashimage=(hd0,0)/grub/splash.xpm/gz  #注释：GRUB 的安装目录。
title Red Hat Linx (2.4.20-8)               #注释：下面是 Red Hat Linux 9.0 的启动信息。
  root (hd0,5)
  kernel /vmlinuxz-2.4.20-8 ro root=LABEL=/
  initrd /initrd-2.4.20-8.img
title XP                                     #注释：下面是 Windows XP 系统的启动信息。
  rootnoverify (hd0,0)
  chainloader +1
```

④ 打开一键 GHOST 的菜单文件。进入 Windows 系统，打开“C:\”盘下面的“dosh”文件夹。再双击打开“ghost”文件夹。

⑤ 用记事本打开文件“menu.lst”，查看 GRUB 引导信息。可以发现，Windows 下面的 GRUB 有与 Red Hat Linux 9.0 中 GRUB 相似的引导信息。找到“menu.lst”文件中的引导信息，代码如下所示。

```
title 1KEY GHOST                             #注释：这是一键 GHOST 的启动信息。
```

```
map --mem /dosh/ghos/ghost.img (fd0)
map --hook
chainloader (fd0)+1
rootnoverify (fd0)
```

⑥ 将 Red Hat Linux 9.0 的引导信息加入到打开的“menu.lst”文件中上面找到的一键 GHOST 启动项的下面。需要添加的代码如下所示。

```
title Red Hat Linx (2.4.20-8)
  root (hd0,5)
  kernel /vmlinuxz-2.4.20-8 ro root=LABEL=/
  initrd /initrd-2.4.20-8.img
```

⑦ 在记事本中，单击“文件”|“保存”菜单命令保存文件。

⑧ 重启计算机。当进入 Windows 系统和启动菜单时，按向下方向键进入一键 GHOST，然后再按向下方向键选择需要启动的系统。选择“Red Hat Linx (2.4.20-8)”菜单，然后按“Enter”键，即可启动 Red Hat Linux 9.0 系统。

用这种方式，可以实现 Windows 系统下 Linux 系统的引导。Windows 重装系统以后，可以用这种方式恢复对 Linux 的引导。

1.3 VMware 虚拟机安装 Linux 系统

VMware 是一款可以在一台电脑上同时安装多个操作系统的软件，可以在一台电脑上虚拟若干个计算机硬件环境。每个虚拟的计算机可以运行单独的操作系统而互不干扰。因为所有的硬件都是虚拟的，在虚拟机上进行分区、格式化、安装系统等操作不会对原有的系统产生任何影响。在学习安装 Linux 系统时，可以在 VMware 虚拟计算机上操作。

注意：使用虚拟机虽然方便了学习，但是会损失部分性能。

1.3.1 VMware 软件安装

使用 VMware 虚拟机可以在 Windows 系统内再安装其他的操作系统，而不会对原有的操作系统造成任何影响。本节将讲述 VMware 虚拟软件在 Windows XP 上的安装。

① 获取 VMware 安装包。可以购买光盘或网络下载获得安装包。本节中使用的是 VMware workstation 5.5.3 版本的安装包。

② 双击打开安装包开始 VMware 的安装。如图 1-41 所示，单击“Next”按钮。

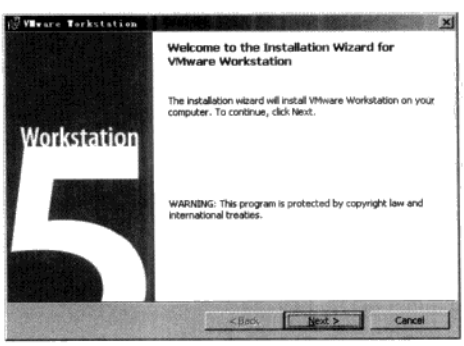


图 1-41 VMware 的安装界面

③ 选择安装目录。VMware 的默认安装目录是“C:\Program Files\VMware-workstation-5.5.3”。如图 1-42 所示，单击“Change”按钮，选择一个文件夹，然后单击“Next”按钮。

④ 设置快捷方式。如图 1-43 所示，VMware 在安装时，会默认创建快捷试。这些快捷方式的设置含义如下所示。

- Desktop：将在桌面创建快捷方式。
- Start Menu Programs folder：将在“开始”菜单中创建快捷方式。
- Quick Launch toolbar：将在任务栏的快速启动中创建快捷方式。

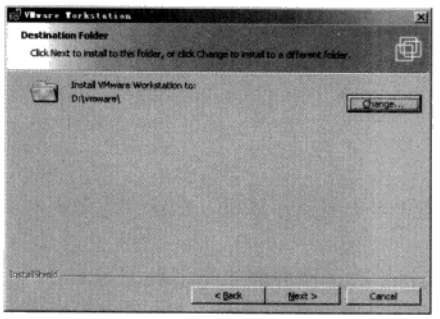


图 1-42　设置 VMware 的安装文件夹

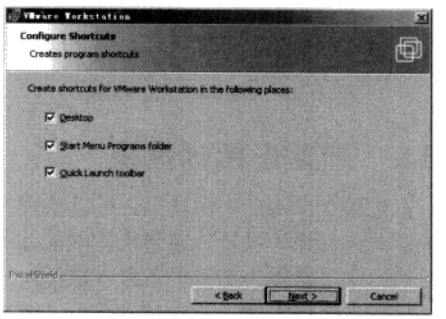


图 1-43　设置快捷方式

⑤ 单击“Next”按钮。如图 1-44 所示，“Yes disable autorun”复选框表示取消光驱的自动运行。如果光驱自动运行，虚拟机会无法加载光驱。在安装时默认取消光驱自动运行。然后单击“Next”按钮。

⑥ 完成了这些设置以后，单击“Install”按钮开始软件的安装，如图 1-45 所示。

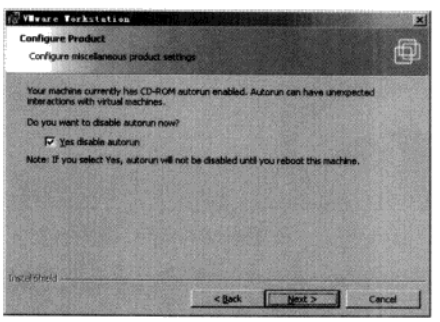


图 1-44　设置光驱的快速启动

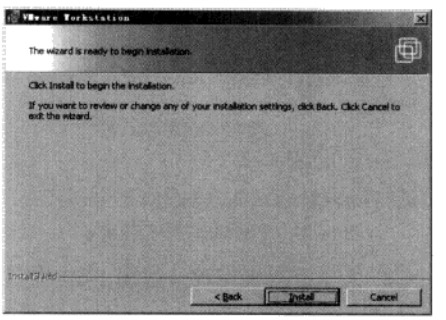


图 1-45　完成安装设置

⑦ 然后 VMware 会自动完成安装，进度条会显示安装的进度，如图 1-46 所示。

⑧ 安装结束以后，单击“Finish”按钮完成安装。

⑨ 汉化 VMware。这里安装的 VMware 是英文版的。如果读者对英文软件的操作不习惯，可以安装相应的汉化包。汉化包可以从网络上下载。

⑩ 双击运行汉化包，单击“下一步”按钮。如图 1-47 是选择 VMware 目录的界面。单击“浏览”按钮，在“浏览”对话框中选择 VMware 的安装目录“D:\vmware”。然后单击“下一步”按钮。

⑪ 汉化包安装以后，单击“确定”按钮结束。在“开始”菜单中单击“VMware Workstation”命令，VMware 的运行界面如图 1-48 所示。

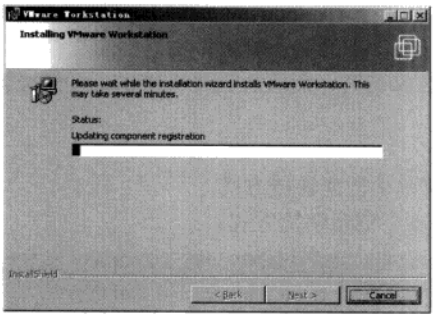


图 1-46　VMware 正在安装

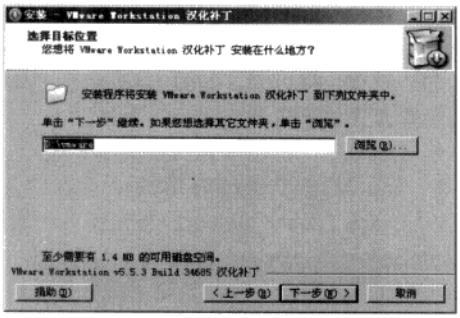


图 1-47　汉化包的安装

1.3.2　新建虚拟计算机

安装好 VMvare 软件以后，可以使用 VMware 建立一个虚拟计算机。在建立虚拟机时，需要注意虚拟硬件的设置。新建虚拟机的步骤如下：

① 在“开始”菜单中打开 VMware 软件。VMware 的工作界面如图 1-48 所示。

② 新建虚拟机。单击“文件”|“新建”|“虚拟机”菜单命令，“新建虚拟机向导”如图 1-49 所示。然后单击“下一步”按钮。

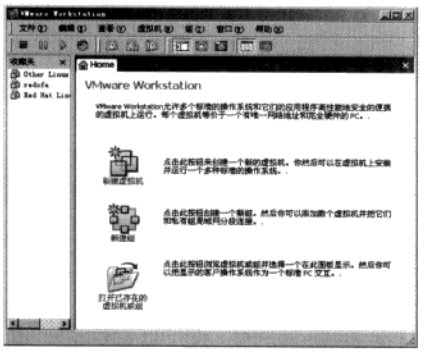


图 1-48　VMware 的工作界面

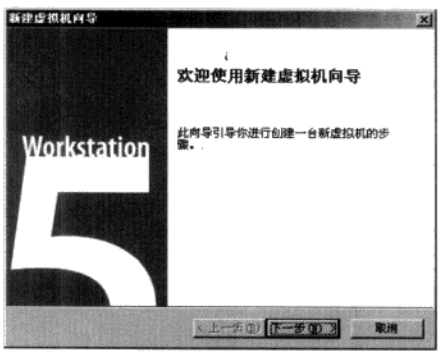


图 1-49　“新建虚拟机向导”对话框

③ 选择虚拟机配置。如图 1-50 所示，虚拟机可以选择“典型”配置和“自定义”配置。“典型”配置是创建一个有普通设备和配置选项的虚拟机。单击“典型”单选按钮，然后单击“下一步”按钮。

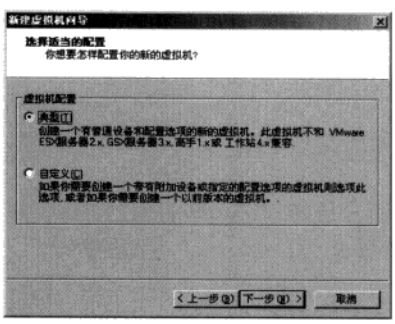


图 1-50　选择“虚拟机配置”

④ 选择用户操作系统。在 VMware 虚拟机中，可以安装多种操作系统。如图 1-51 所示，在“用户操作系统”单选按钮中，单击选择“Linux”选项。然后在“版本”下拉菜单中单击选择“Red Hat Linux”。在本书中，使用的 Linux 发行版是 Red Hat Linux 9.0。然后单击“下一步”按钮。

⑤ 设置虚拟机的名称和存储位置。如图 1-52 所示，在“虚拟机名称”文本框中输入虚拟机的名称“Red Hat Linux”，单击“位置”文本框后面的“浏览”按钮，选

择一个文件夹存储虚拟机文件。然后单击“下一步”按钮。

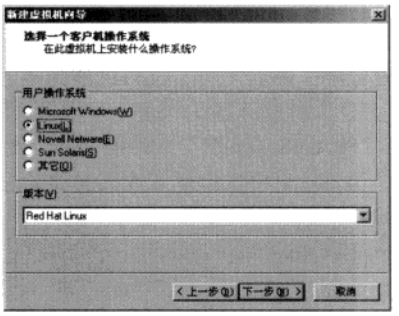


图 1-51　选择用户操作系统

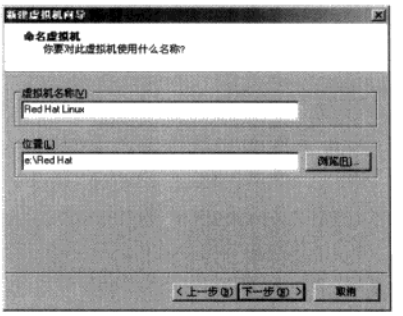


图 1-52　设置虚拟机的名称和存储位置

⑥ 设置虚拟机的网络类型。如图 1-53 所示，在“网络连接”单选按钮中单击“使用桥接网络”。这时虚拟机可以通过客户机的操作系统直接访问网络。然后单击“下一步”按钮。

⑦ 设置“磁盘”容量。如图 1-54 所示，可以设置虚拟机的最大磁盘容量。在“磁盘大小”文本框中输入“8”，表示为虚拟机分配 8GB 硬盘容量，然后单击“完成”按钮。

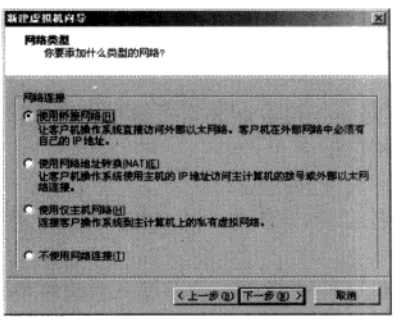


图 1-53　设置网络连接

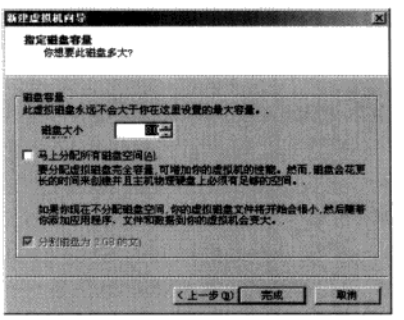


图 1-54　设置虚拟机的硬盘容量

⑧ 新建的虚拟机会显示在“收藏夹”中，如图 1-55 所示。

⑨ 设置虚拟机。在“收藏夹”中单击选择一个虚拟机，在主界面中会显示这个虚拟机的信息和工具，如图 1-56 所示。

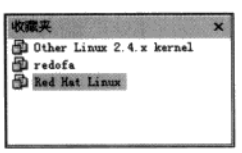


图 1-55　虚拟机收藏夹

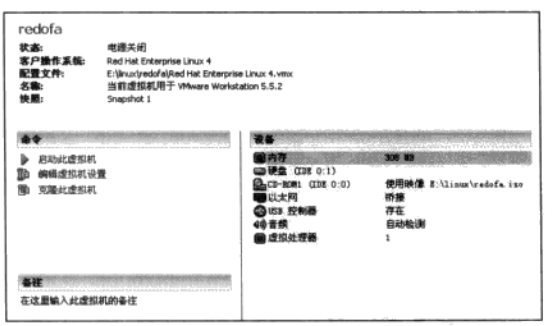


图 1-56　虚拟机的信息

⑩ 在虚拟机窗口中，单击“编辑虚拟机设置”工具，可以对一个虚拟机进行配置。如图 1-57 所示，在“虚拟机设置”对话框中设置虚拟机的内存大小。在“此虚拟机的内存”文本框

中输入“308”，表示虚拟机最多可以占用主机中的 308MB 内存。如果主机的内存配置较高，可以为虚拟机分配较多的内存，以提高虚拟机的性能。

⑪ 配置光驱。在“虚拟机设置”对话框中，单击“设备”列表中的“CD-ROM1”选项，如图 1-58 所示。在“连接”单选按钮中，“使用物理驱动器”表示使用主机的物理光驱。“使用 ISO 镜像”表示用一个 ISO 文件作为虚拟机的光驱。ISO 文件是光盘的镜像，可以从网络上下载 Linux 系统的 ISO 镜像在虚拟机上安装系统。单击“使用 ISO 镜像”文本框后的“浏览”按钮，选择一个从网络上下载的“ISO”镜像。“虚拟设备节点”中可以选择虚拟机光驱的总线。

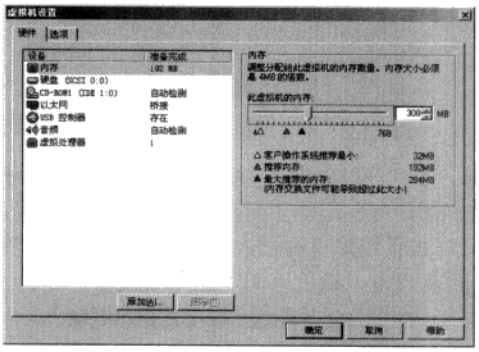


图 1-57　设置虚拟机的内存大小

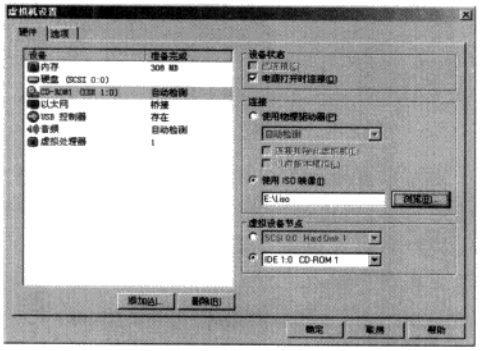


图 1-58　设置虚拟机的光驱

⑫ 配置网络。单击“设备”列表中的“以太网”选项，可以对虚拟机的网络进行设置，如图 1-59 所示。虚拟机的网络可以设置成下面这些网络类型。

- 桥接：直接使用主机的物理网卡，这时虚拟机并没有虚拟网卡。
- NAT：可以共享主机的 IP 地址，在虚拟机安装以后的系统中，需要将网络 IP 地址设置成与虚拟机相同的 IP 地址。
- 仅主机：虚拟机和主机共享一个本地连接，虚拟机的网络功能是通过主机的本地连接实现的。
- 自定义：可以在计算机中设置虚拟路由和虚拟局域网。虚拟机通过虚拟局域网连接到网络中。

⑬ 设置虚拟机的声卡。如图 1-60 所示，单击“设备”列表中的“音频”选项。可以对虚拟机指定一个声卡。然后单击“确定”按钮完成虚拟机的设置。

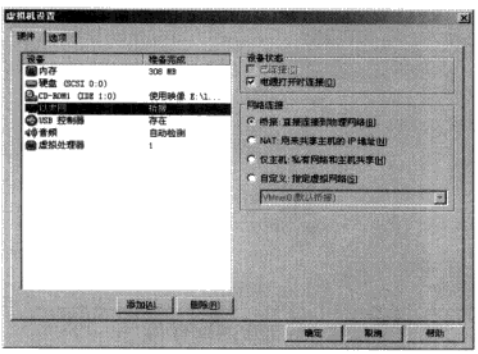


图 1-59　设置虚拟机的网络

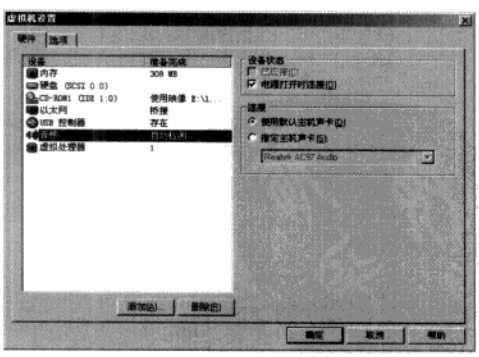


图 1-60　设置虚拟机的声卡

1.3.3　VMware 的使用

新建虚拟机后，需要注意 VMware 的使用方法。本节将介绍 VMware 中常用工具的使用方法。利用这些常用工具可以有效地对虚拟机进行管理。当启动虚拟机系统以后，在虚拟机中的操作与本地计算机上的操作是完全相同的。

① 在“开始”菜单中单击“VMware Workstation”命令，启动 VMware 软件。

② 在 VMware 的收藏夹中，单击选择上一节新建的虚拟机“Red Hat Linux”，VMware 中显示的虚拟机信息如图 1-61 所示。

③ 使用虚拟机工具控制虚拟机。VMware 的常用工具栏如图 1-62 所示。

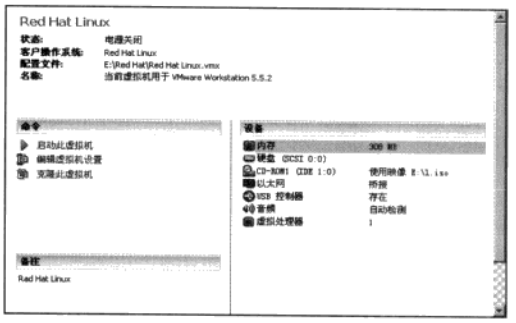


图 1-61　虚拟信息界面

图 1-62　VMware 的常用工具栏

④ 常用工具栏中的工具和使用方法如下所示。

- 关闭电源：启动虚拟以后，可以用这个工具关闭虚拟机。相当于强制关闭计算机的电源。
- 挂起：暂停虚拟机的运行。相当于使计算机处于待机状态。
- 打开电源：运行虚拟机。相当于按下计算机的电源键。
- 重启：重启虚拟机。相当于按下计算机的重启键。
- 制作虚拟机快照：将虚拟机的当前状态保存，以后可以恢复到这个保存的状态。
- 恢复虚拟机到父快照：将虚拟机恢复到最后一次保存的快照。
- 管理快照：当虚拟机有多个快照时，单击这个工具可以对虚拟机的快照进行管理。
- 全屏：以全屏的状态显示一个虚拟机。这时在虚拟中操作和真正安装的系统是相同的。
- 快速切换：当 VMware 启动了多个虚拟机时，单击这个工具可以快速切换工作的虚拟机。

⑤ 启动虚拟机。在收藏夹中单击选择已经建立的虚拟机“Red Hat Linux”。然后单击工具栏中的“打开电源”工具，即可启动这个虚拟机。虚拟机的启动界面如图 1-63 所示。

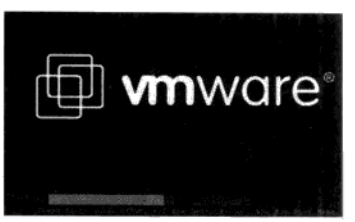


图 1-63　虚拟机的启动界面

⑥ 单击“制作虚拟机快照”工具，为虚拟机的当前运行状态制作一个快照。用同样的方法，再次制作一个快照。

⑦ 管理和恢复快照。单击“管理快照”工具。对话框如图 1-64 所示，在列表中单击选择一个快照，在“名称”文本框中输入一个名称可以对这个快照进行重命名。单击“删除”按钮可以删除一个快照。单击“转到”按钮可以恢复到这个快照。

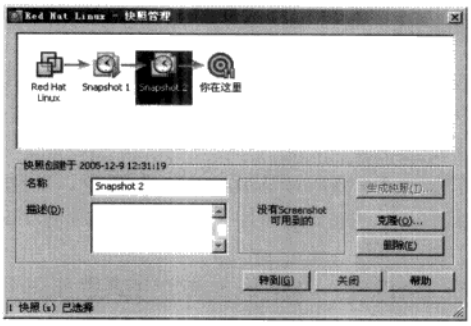


图 1-64　管理虚拟机的快照

⑧ 单击“关闭电源”工具，可以关闭已经启动虚拟机的电源。

1.3.4　在 VMware 虚拟机中安装 Linux 系统

在 VMware 的虚拟机中安装 Linux 系统时，对硬盘的分区、格式化操作，都是在虚拟硬盘上进行的。对硬盘上的数据没有任何影响。在学习 Linux 安装时，可以使用 VMware 虚拟机进行练习。本节将在 VMware 虚拟机上安装 Red Hat Linux 9.0 系统。

① 打开本章 1.3.2 节中创建的 Red Hat Linux 虚拟机。

② 设置光驱。如果是已经有 Red Hat Linux 9.0 的安装光盘，将光盘插入安驱中。然后在虚拟机界面中单击“编辑虚拟机设置”工具。在“虚拟机设置”对话框中，单击“设备”列表的“CD-ROM1”工具，设置虚拟机的光驱。如图 1-65 所示，在“连接”单选按钮中，单击选择“使用物理驱动器”选项。然后单击“确定”按钮。这时，系统在启动时会使用光驱中的光盘进行系统安装。

③ 设置 ISO 镜像作为 VMware 的光驱。如果没有安装光盘，可以从网络上下载 Red Hat Linux 9.0 的 ISO 光盘镜像。在图 1-65 所示的对话框中，单击选择“连接”单选按钮组中的“使用 ISO 镜像”选项。然后单击“浏览”按钮，选择已经下载的 Red Hat Linux 9.0 光盘 ISO 镜像文件，如图 1-66 所示。然后单击“确定”按钮。

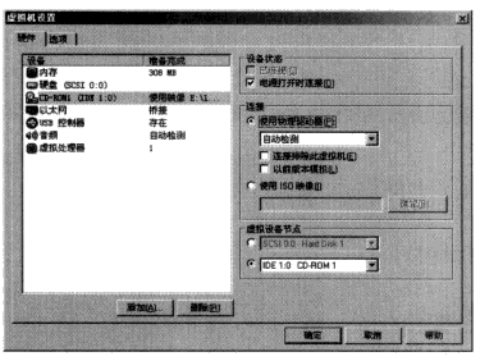


图 1-65　将光驱设置成物理光驱

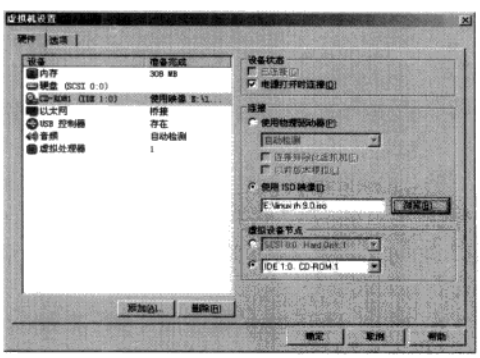


图 1-66　设置 ISO 光盘镜像作为光驱

④ 在工具栏中，单击“打开电源”工具，启动虚拟机。虚拟机会自动从 ISO 光盘镜像启动，进入 Red Hat Linux 9.0 的安装，如图 1-67 所示。

⑤ 在虚拟机的显示窗口中单击鼠标左键，虚拟机就会捕获鼠标和键盘的操作。这时鼠标

只能在虚拟机的显示窗口中移动，键盘的输入对原有系统不再有效。按“Ctrl”+“Alt”快捷键可以将鼠标和键盘切换到原来的系统。

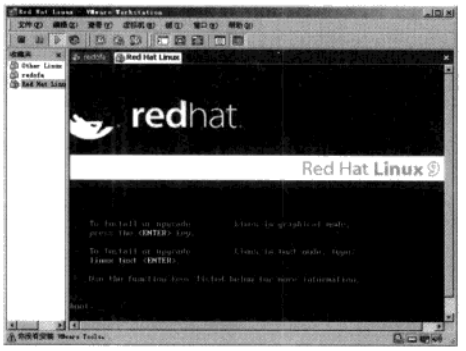


图 1-67　在虚拟机中安装 Red Hat Linux 9.0 系统

⑥ 按“Ctrl”+“Alt”+“Enter”快捷键可以使虚拟机进入全屏操作。再次按“Ctrl”+“Alt”+“Enter”快捷键可恢复以前的窗口。

⑦ 在虚拟机中，按下“Ctrl”+“Alt”+“Delete”组合键，将只对原系统有效。如果需要重启虚拟机，需要按“Ctrl”+“Alt”+“Insert”组合键。

⑧ 用本章 1.2.3 节中的步骤在虚拟机中完成 Red Hat Linux 9.0 的安装。然后启动 Red Hat Linux 9.0 系统。

1.4　第一个 C 程序

典型的 C 程序设计包括源代码的编写和编译两个步骤。C 语言源程序只有经过编译后才能够在 Linux 系统中运行。本节将讲解在 Linux 系统中进行 C 程序设计的一般方法。

1.4.1　在 Linux 中编写 C 程序代码

C 程序由 C 语言源代码构成。编译后的 C 程序为普通的二进制程序，可在 Linux 系统上直接运行。通常，借助于文本编辑器编写C程序源代码。本节将介绍在Red Hat Linux 9.0 系统上编写 C 程序源代码代码及编译运行的整个过程。

① 打开文本编辑器。在图形界面下，单击“主菜单”。在弹出的菜单中选择“附件”中的“文本编辑器”。启动的文本编辑器如图 1-68 所示。

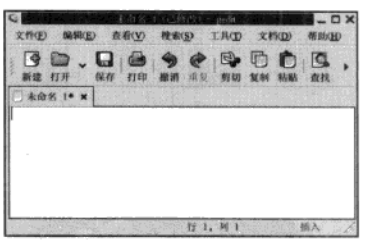


图 1-68　Red Hat Linux 9.0 中的文本编辑器

② 在文本编辑器中输入如下代码，实现输出“Hello , Linux .”的文本功能。

注意：注意程序代码的大小写。C 程序严格区分大小写。

```
01  int main()                                  /*主函数*/
02  {
03    printf("Hello , Linux .\n");              /*打印字符串到终端上*/
04  }
```

③ 保存文件。单击文本编辑器的“文件”|“另存为”菜单命令，“另存为”对话框如

图 1-69 所示。文本编辑器默认的保存目录为根目录“/root”。在“选中”下面的文本框中输入文件名“hello.c”。然后单击“确定”按钮保存文件。

④ 查看代码文件。在桌面上双击“root 的主目录”图标，打开 root 用户的主目录，可以发现目录中有一个文件“hello.c”，如图 1-70 所示。

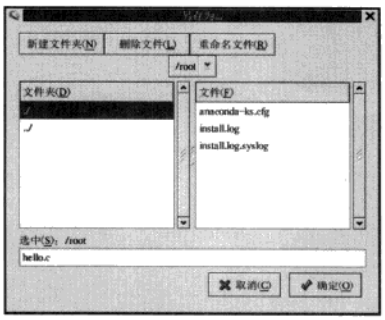


图 1-69 保存代码

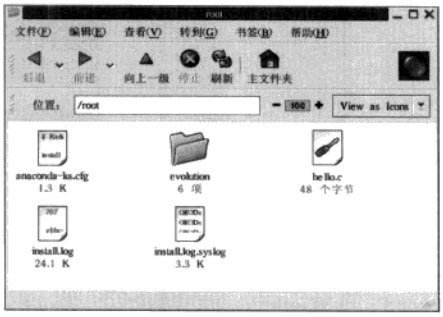


图 1-70 root 根目录中的代码文件

1.4.2 编译程序

所谓编译，就是把高级语言编写的程序源代码，翻译成计算机可以执行的二进制程序。上一节中编写的 C 程序代码是不能被计算机执行的，需要用编译程序编译成二进制程序才可以执行。本节将用 gcc 编译上一节的程序代码。gcc 是 Red Hat Linux 9.0 中的 C 程序编译器。

① 在“主菜单”中选择“系统工具”。在“系统工具”的菜单中单击“终端”命令，打开 Linux 的命令行。Linux 命令行界面如图 1-71 所示。可以在命令行中输入系统命令。

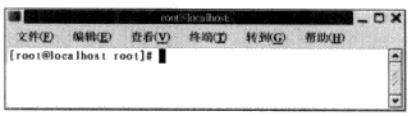


图 1-71 Linux 的命令行

② 编译程序。在 Linux 命令行中输入下面的命令，然后按“Enter”键。

```
gcc hello.c
```

这时，gcc 编译器会自动对程序 hello.c 进行编译，在同一目录下生成可执行文件 a.out。

注意： Linux 系统与 Windows 系统的扩展名是不同的。Linux 系统下的可执行文件可以是任何扩展名，一个文件必须具有可执行权限才能够运行。而 Windows 系统依据扩展名来判断文件是否可执行，常见的可执行文件扩展名为“.exe”、“.com”和“.bat”等。

③ 运行可执行程序。在命令行中输入下面的代码，然后按“Enter”键。

```
./a.out
```

这个程序只输出了一行文本，运行结果如下所示。

```
Hello , Linux .
```

1.5 小结

本章讲述了 Red Hat Linux 9.0 的安装、VMware 虚拟机的使用和一个简单 C 程序的编写等问题。重点将解了 Linux 系统分区，在分区时需要正确设置各个分区的格式，正确将分区持载

到系统中，并处理好与 Windows 系统的引导问题。通过 C 程序实例的练习，可以初步了解 C 程序的开发过程。

1.6　习题

一、填空题

1．所谓_____，指的是以 Linux 内核为中心，再集成开发各种系统管理软件或应用软件，组成一套完整的操作系统。

2．_____指的是可提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件。

3．Linux 默认使用的分区格式为_____或_____分区格式。

4．VMware 是一款可以在一台电脑上同时安装多个操作系统的软件，可以在一台电脑上_____。

5．典型的 C 程序设计包括_____和_____两个步骤。C 语言源程序只有经过后才能够在 Linux 系统中运行。

二、选择题

1．关于 Linux 发行版的说法中正确的是（　　）。

A．Linux 发行版就是 Linux 内核

B．Linux 发行版只有一种

C．所谓 Linux 发行版，指的是以 Linux 内核为中心，再集成开发各种系统管理软件或应用软件，组成一套完整的操作系统

D．Linux 发行版中可以不包含 Linux 内核

2．关于 GRUB 的说法中正确的是（　　）。

A．GRUB 是一种分区工具

B．GRUB 是一款多重引导程序

C．GRUB 只能引导 Linux 操作系统

D．GRUB 不能设置引导密码

3．关于 Vmware 的说法中错误的是（　　）。

A．Vmware 内置了很多操作系统，无须安装操作系统就可以直接使用

B．Vmware 是一款虚拟化产品，可以同时运行多个操作系统

C．使用 Vmware 来安装 Linux 操作系统不会对宿主操作系统产生任何危害

D．在 Vmware 中安装 Linux 操作系统与在物理机器上安装 Linux 操作系统的过程是相同的

三、简答题

1．简述 Linux 操作系统的安装过程。

2．简述 vmware 虚拟机的使用方法，以及在 vmware 虚拟机中安装 Linux 操作系统的过程。

四、编程题

编写一个 C 程序，用于在屏幕上输出“Hello World!”。

第 2 章　使用文本编辑器

vi 是 UNIX 世界最通用的纯文本编辑器，它不但是一款功能强大的文本编辑，同样也可以作为一款优秀的 C 程序代码编辑器。几乎所有的 Linux 发行版都自带了 vi 的增强版本 vim，vim 与 vi 完全兼容。本书推荐读者使用 vim 作为 C 程序的代码编辑器，熟练掌握 vim 的使用将极大地提升 C 程序的代码编写速度。本章的主要内容如下：

- 文本编辑器 vim 的使用；
- 图形化文本编辑器 gedit 的使用。

2.1　使用 vim

vim 是“Visual Interface Improved”的简称，是 Linux 下最常用的文本编辑器。vim 可以完成文本的输入、删除、查找、替换、块操作等功能。用户还可以根据需要对其进行定制，使用插件扩展 vim 的功能。本节将讲述 vim 的使用方法。

提示：vim 是 vi 的增强版本，与 vi 兼容。

2.1.1　启动 vim

在使用 vim 之前，需要从终端中输入 vim 命令启动 vim。下面的步骤可以启动 vim 文本编辑器。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。

② 在终端界面中输入“vim”命令，然后按“Enter”键，系统会启动 vim。vim 的工作界面如图 2-1 所示。

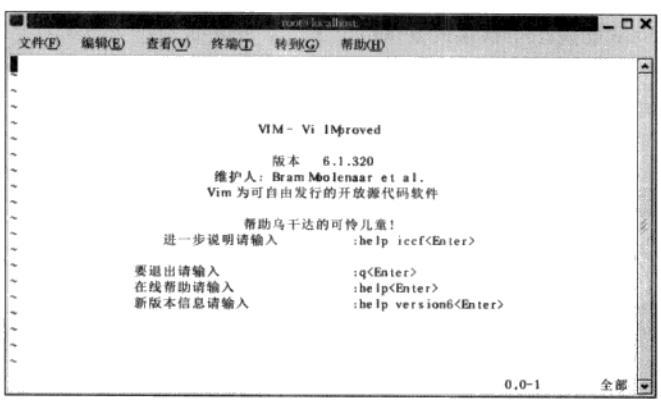


图 2-1　vim 的工作界面

2.1.2　在桌面上创建 vim 启动器

Linux 系统中的启动器相当于 Windows 系统中的快捷方式。除了可以在终端中用命令启动 vim，也可以在桌面新建一个 vim 启动器。双击启动器的图标也可以启动 vim。

① 右键单击桌面上的空白部分，然后单击“新建启动器”右键菜单。“Create Launcher”对话框如图 2-2 所示。

② 在“名字”文本框中输入启动器的名称“vim”。在“命令”文本框中输入启动命令“vim”。然后单击选择“在终端中运行”复选框。单击“图标”按钮，为启动器选择一个图标。然后单击“确定”按钮。

③ 桌面上创建的启动器如图 2-3 所示。双击这个启动器图标可以启动 vim。

图 2-2　新建启动器

图 2-3　桌面上的启动器

2.1.3　vim 的工作模式

vim 的工作模式指的是 vim 不同的使用方式。vim 有普通（Normal）模式、插入（insert）模式和可视（visual）模式三种工作模式。三种模式的表现形式与功能如下所示。

- 普通模式：在进入 vim 后，默认是普通模式，这时可以输入一些命令。如图 2-1 所示就是 vim 的普通模式。在普通模式中，可以在一个冒号的后面输入一个命令，按“Enter”键执行这一个命令。
- 插入模式：在普通模式下，按“i”键或者“a”键，则进入插入模式。这时所有的输入是 vim 的编辑内容。输入结束后，按“Esc”键，可返回到普通模式。
- 可视模式：在普通模式下，按“v”键，则进入可视模式。可视模式下主要进行复制粘贴的操作。按“Esc”键可以返回到普通模式。

vim 的三种工作模式与切换方法如图 2-4 所示。

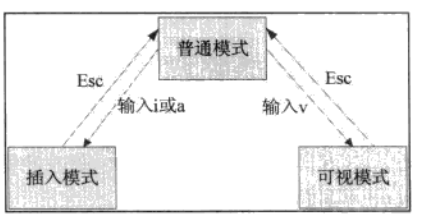


图 2-4　vim 的三种工作模式

2.1.4　保存与打开文件

vim 中保存文件的命令是“:w”。打开文件的命令是“:r”。本节将讲解 vim 中用命令保存与打开文件。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。。在终端中输入“vim”命令再按“Enter”键打开 vim。

注意：vim 是在终端中打开的。vim 上面的菜单命令都是终端的菜单命令。单击 vim 上的菜单命令都是对终端进行操作。

② 这时 vim 是普通模式。按 i 键进入插入模式。然后在 vim 中输入下面的句子。

```
godd moring ,everyone .
I'm learning about vim .
```

vim 的工作界面如图 2-5 所示。

③ 按“Esc”键进入普通模式。这时输入下面的命令，将文件保存到“/root/”文件夹下，文件名为“vim1.txt”。

```
w /root/vim1.txt
```

vim 会显示下面的提示信息。

```
"~/vim1.txt" [新] 3L ,48C 已写入
```

表示新建了文件“vim1.txt”。文本中有 3 行，48 个字符。

④ 如果再次输入“w /root/vim1.txt”命令，则再次以同一文件名保存文件。vim 会有以下提示。

图 2-5 在 vim 中输入文本

```
E13: File exists (add ! to override)
```

vim 提示有重名文件。可以命令后面添加!覆盖以前的文件。命令如下所示。

```
w! /root/vim1.txt
```

⑤ 按“Esc”键进入普通模式。输入“:q”命令退出 vim。vim 显示信息如下所示。信息的含义是最后的编辑没有保存，在命令后面加!强制退出。

```
E37 No write since last change (add ! to override)
```

⑥ 保存并退出 vim。按“Esc”键进入普通模式。输入“:wq”命令，再按“Enter”键。vim 会保存文件并退出 vim。

⑦ 再次在终端中输入“vim”命令，按“Enter”键打开 vim。

⑧ 打开的 vim 默认是普通模式。输入下面的命令打开前面的步骤编辑的文件。

```
-r /root/vim1.txt
```

⑨ 这时 vim 会读入一个文件并显示。按 i 键进入编辑模式，编辑以前的文件。

⑩ 按“Esc”键进入普通模式。输入“:q”命令退出 vim。这时 vim 提示文件没有保存。输入“:q!”命令可以不保存文件退出 vim。

⑪ 在终端中，可以输入“vim /root/vim1.txt”命令来启动 vim 并打开一个文本。

2.1.5 移动光标

vim 中的移动指的是在 vim 移动光标的位置。在三种模式下，都可以按键盘上的上、下、左、右方向键进行移动。vim 在方向键的基础上提供了更多更快的移动方式，这些移动方式可以分为下面几类。

- 字符移动：每次向前或向后移动一个字符的位置。
- 单词移动：每次向前或向后移动一个单词的位置。
- 行移动：每次向上或向下移动一整行。
- 页面移动：每次向上或向下移动一页。

1．光标移动一个字符

在普通模式下，可以使用下面的命令来移动光标。

注意：这些命令都是小写字母。

- h：向左移动光标。
- j：向下移动光标。
- k：向上移动光标。
- l：向右移动光标。

这四个命令的字符是键盘上“H”右边的四个字母，非常便于使用。这几个键可以替代方向键进行光标移动。

2．移动单词

在普通模式下，使用 w 命令可以将光标向后移动一个单词。在 w 前面指定一个数字前辍，光标会向后移动指定数目的单词。例如，3w 命令可以将光标向后移动 3 个单词。b 命令的作用和 w 的作用相反，可以将光标向前移动一个单词。也可以加一个数字前缀一次向前移动多个单词。

e 命令可将光标移动到下一个单词的最后一个字符。与之相对应的是 be 命令可以将光标移动到前一个单词的最后一个字符。相当于用 b 命令向前移动一个单词，然后再用 e 命令将光标移动到最后一个字符。

3．移动行

vim 中有着丰富的行移动功能。这些行移动功能可以取代图形界面中的滚动条。行移动的命令如下所示。

- $命令：$命令可以将光标移动到当前行的行尾，作用类似于键盘上的“End”键。该命令可接受一个数字前缀，表示向后移动若干行的行尾。命令 1$会将光标移动到当前行行尾，2$则会移动到下一行的行尾。
- 0 命令：和$命令对应的命令是 0 命令，可将光标移动到当前行的第一个字符上。相当于“Home”的功能。该命令不能接受数字前缀。
- ^命令：^命令可将光标移动到当前行的第一个非空白字符上。该命令前面加上计数没有任何效果。
- 冒号命令：行移动中最简单方法是使用冒号:加上具体的行号，这样光标就会移动到指定的行号。
- j 命令：使用 j 命令可以向下转跳若干行。在前面加上数目，就可以跳转相应的行数。
- G 命令：G 命令把光标定位到指定的行上。例如"15G"就会把光标定位到 15 行。如果没有指定命令数字，则会把光标定位到最后一行。
- gg 命令：gg 命令是跳转到第一行，与命令 1G 效果一样。
- 百分比命令：在命令%之前指定一个命令计数，可以将文件定位到这个指定百分比的位置上。例如 90%会跳到接近文件尾的地方，50%将会把光标定位在文件的中间。

显示当前屏幕的行，可以使用命令 H，M 和 L。命令的功能如表 3.1 所示的命令。

4．页滚动

vim 可以实现所显示页面的向上向下滚动。查当于图形界面中的拖运滚动条。常用的页滚动命令如下所示。

- Ctrl+u 命令：Ctrl+u 命令可使文本向上滚动半屏。相对应的命令是 CTRL-D 命令，将窗口向下移动半屏。
- Ctrl+e 命令：如果一次滚动一行可以使用 Ctrl+e（向上滚动）和 Ctrl+y（向下滚动）命令。
- Ctrl+f 命令：向前滚动一整屏命令是 Ctrl+f。相反的是 Ctrl+b 是向下滚动一屏。
- %命令：使用%命令可以用来匹配括号。在书写程序或者阅读代码时，用%命令可跳转到与当前光标下的括号相匹配的那一个括号上去。可能是向前或向后转跳。这里的括号匹配可以匹配小括号、中括号、花括号三种。
- zz 命令：zz 命令把光标所在的行滚动到屏幕正中央。相似功能的有 zt 命令把光标所在的行滚动到屏幕顶端，zb 则把光标所在的行滚动到屏幕底端。

表 3.1　移动操作

命　令	意　义	作　用
H	Home	移动到当前屏幕的第一行
M	Middle	移动到当前屏幕的中间一行
L	Last	移动到当前屏幕的最后一行

2.1.6　插入操作

插入指的是在光标位置的前后行、前后字符处插入新行或新字符。也可能是删除指定数目的行和字符，然后输入新的内容。插入命令如表 3.2 所示。需要强调的是，这里的命令操作都是在普通模式下的进行的。

表 3.2　插入操作

命　令	意　义
i	在光标前插入
I	在当前行首插入
a	在当前光标后插入
A	在当前行尾插入
o	在当前行之下新开一行
O	在当前行之上新开一行
r	替换当前字符
R	替换当前字符及其后的字符，直至按 Esc 键
s	从当前光标位置处开始，以输入的文本替代指定数目的字符
S	删除指定数目的行，并以所输入文本代替
ncw/nCW	修改指定数目的字符
nCC	修改指定数目的行

在普通模式下键入 i 命令以后，vim 就会在窗口底部显示“--Insert—”提示，这表明用户可以在光标处输入内容。此时按“Esc”键，会返回到正常模式。a 命令用来在当前光标处追加内容。o 命令可以在当前行的下面新起一行，在新行中输入内容。

2.1.7　删除操作

vim 可以使用命令对光标处字符进行删除，也可以对单词、整行进行删除。删除命令如表 3.3 所示。

表 3.3　删除操作

命　令	意　义
x	删除光标当前字符
ndw 或 ndw	删除光标处开始及其后的 *n*-1 个单词
do	删除当前行光标以前的所有字符
d$	删除当前行光标以后的所有字符
dd	删除光标所在的行
ndd	删除当前行及其后 *n*-1 行
X	删除光标前的一个字符。而 x 删除光标后的一个字符
ctrl+u	删除当前输入方式所输入的文本

x 命令可以删除光标处的一个字符，使用一个命令参数 4x 可以删除 4 个字符。dw 命令可以删除一个单词，其中的 w 可以看做是向右移动一个单词的距离。d 命令后面可以跟任何一个位移命令，它将删除从当前光标起到位移的终点处的文本内容。于是，do 就是删除当前行光标以前所有的内容，d$就是删除当前行光标以后的内容，ndd 就是删除 *n*-1 行的内容。

2.1.8　取消操作

在编辑时，如果错误操作而修改了原有的文本，可以使用取消命令来取消之前的修改操作。vim 也可以多次取消以前的操作。常用的取消命令如表 3.4 所示。

表 3.4　修改操作

命　令	意　义
.（英文句号）	重复上一次修改
u	取消上一次修改
U	将当前行恢复到修改前的状态

U 命令会一次撤销对一行的全部操作。第 2 次使用 U 命令则会撤销前一个 U 命令的操作。连接按 u 或句点可以多次执行取消或重复上一次操作。

2.1.9　保存操作

vim 可实现文件保存、另存、覆盖保存、追加保存等文件保存操作。文本保存命令如表 3.5 所示。

表 3.5　保存操作

命　令	意　义
:w	保存文件。这个命令需要文件已经保存过
:x	保存文件并退出。这个命令需要文件已经保存过
:w file2	将内容写入文件 file2，覆盖以前的文件
:w >>file2	将缓冲区内容附加保存到文件 file2 的后面

注意：使用文件保存命令时，需要先输入冒号。用 w 命令表示将内容写到某个文件，需要文件已经保存过。如果不指定文件名，则默认保存到正在编辑的文件。指定文件名则会另存为新文件。

2.1.10 退出操作

vim 在结束工作时需要退出。在退出之前需要对当前编辑的文件进行处理。退出操作命令如表 3.6 所示。

表 3.6 退出操作

命　令	意　义
: q	退出 vi。如果文件没有保存则不会退出
:q!	不保存文件，强制退出 vim
ZZ	保存并退出

q 命令直接退出 vim 而不保存任何修改。这时，如果用户已经修改了文本，则 vim 则会提示文件没有保存，不会退出。q!命令可以放弃保存退出编辑。命令 ZZ 可以保存当前文件并退出。

注意：ZZ 命令前面不需要输入冒号，直接在普通模式下，按着 shift 键的同时按两次 Z 键即可。

2.1.11 查找操作

命令/string 用于搜索一个字符串 string，会从光标开始处向文件尾搜索所有的 string。命令 ? string 从光标开始处向文件首搜索所有的 string。

注意：字符.*[]^%/?～$有特殊意义，如果需要查找的内容中包含这些字符，要在这些字符前加一个反斜杠\对字符进行转义。

命令 n 在同一方向重复上一次搜索命令。N 在反方向上重复上一次搜索命令。常用的特殊字符匹配有以下两个。

- *：在查找的字符串中匹配任意字符。
- ?：在查找的字符串中匹配一个字符。

2.1.12 替换操作

vim 有着强大的替换功能。除了进行字符串替换以外，还可以使用正则表达式进行替换。常用的替换命令如下所示。

- s/p1/p2/g：将当前行中所有字符串 p1 用字符串 p2 替代。
- n1,n2s/p1/p2/g：将第 n1 至 n2 行中所有字符串 p1 用字符串 p2 替代。
- g/p1/s//p2/g：将文件中所有 p1 均用 p2 替换。

2.1.13 选项设置

vim 编辑器可以用 set 命令来设置特定的选项来定制编辑环境。表 3.7 列出了 set 命令的部分选项。

表 3.7 vim 部分选项

选　项	作　用
all	列出所有选项设置情况
term	终端类型

续表

选　项	作　用
ignorance	在搜索中忽略大小写
list	显示制表位（Ctrl+I）和行尾标志（$）
number	显示行号
nomagic	允许在搜索模式时使用前面不带“\”的特殊字符
nowrapscan	禁止 vi 在搜索到达文件两端时，又从另一端开始
mesg	允许 vi 显示其他用户用 write 写到自己终端上的信息

如果要查看所有选项的设置，在普通模式下输入命令:set all，vim 会显示 vim 的详细配置列表。如果要改变某个设置，可以输入:set option(= value)”。其中“option”就是列表中的选项名，选项的值“(=value)”根据选项不同是可选的设置。每次进入 vim，所有的选项将会被设置为默认值。进入 vim 之后对选项的修改，只在当前窗口有效。

提示：在进行软件开发时，经常需要查看行号，此时在普通模式下输入命令:set number（等价于:set nu）即可显示文件中文本的行号。如果需要取消查看行号，此时在普通模式下输入命令:set nonumber（等价于:set nonu）即可。

2.1.14　调用 shell 命令

在使用 vim 编辑文本时，有时需要执行一些 shell 命令。vim 中使用 shell 命令的方法如表 3.8 所示。

表 3.8　执行 shell 命令

命令格式	说　明
:!cmd	执行 cmd 命令
:m,n w!cmd	执行 cmd 命令，文本中 *m* 到 *n* 行的内容作为 cmd 的参数
:r!cmd	执行 cmd 命令，cmd 命令的结果插入到当前文本中

例如，在插入状态下需要查看用户目录下的文件，可以按“Esc”键切换到普通模式。然后输入命令“:ls /root”，vim 中就会显示 root 目录下的文件列表。如果要把文件列表插入当前编辑的文本中，可以使用:r!ls /root 命令。

2.2　vim 使用实例

vim 是一个功能强大的文本编辑器，字符模式下完成文本编辑的各种操作。本节将以操作实例讲解 vim 的文本编辑操作。

2.2.1　字符的插入与删除

文本编辑的最基本操作是文本的插入与删除。vim 有各种字符插入与删除的命令。本节将进行插入与删除的练习。

① 在 Red Hat Linux 9.0 的桌面上，单击“主菜单”|“系统工具”|“终端”命令，打开系统终端。在终端中输入 vim 命令，然后按“Enter”键启动 vim。

② 进行 vim 默认的模式是普通模式。按 i 键进入插入模式。在 vim 中输入下面的程序。

vim 中的文本如图 2-6 所示。

```
01  void main()                    /*主函数*/
02  {
03  printf("hello , Linux .");     /*打印字符串到终端*/
04  }
```

③ 当前光标停留在最后一行。按“Esc”键返回到普通状态。然后按“O”键在当前光标的上一行插入一行。

注意：这里使用的是大写“O”，小写表示在当前行的下一行插入一行。

④ 当前光标停留在插入的新行中，已经进入了插入模式。然后输入下面一行文本。

```
getch();
```

⑤ 移动光标。按“Esc”键返回到普通状态。按“k”键向上移动光标。按“h”键向左移动光标。把光标移动到文本的第一个字符上。

⑥ 删除字符。按“Esc”键返回到普通状态。按“x”键删除当前光标的字符。多次按“x”键删除文本中的“void”。

⑦ 按“i”键进入插入模式。然后在光标处输入文本“int”。

⑧ 按“Esc”键返回到普通状态。按“j”键向下移动光标，按“h”键向左移动光标。按“i”键进入插入状态。在两行代码前对代码输入空格进行缩进。编辑以后的文本如图 2-7 所示。

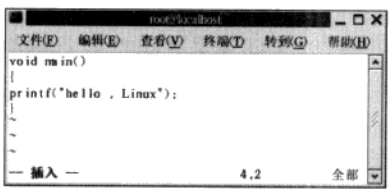


图 2-6　在 vim 中输入文本

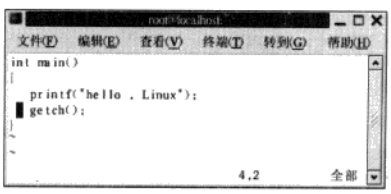


图 2-7　编辑以后的文本

⑨ 保存文件。按“Esc”键进入到普通状态。然后输入命令“:w /root/01.c”，再按“Enter”键保存文件。

⑩ 退出 vim。按“Esc”键进入到普通状态。然后输入命令“q”，再按“Enter”键退出 vim 回到终端。

2.2.2　字符的查找与替换

① 用上一节第 1 个步骤相同的方法打开终端，然后在终端中打开 vim。

② 打开文件。打开 vim 时，默认的是普通状态。这时输入命令“r /root/01.c”，可以打开上一节输入的文本。

③ 查找字符。按“Esc”键进入到普通状态。然后输入命令“/i”，表示在文本中查找字符串“i”。然后按“Enter”键。这时 vim 会以红色显示在文本中所有的字符串“i”，如图 2-8 所示。

④ 字符串替换。按“Esc”键进入到普通状态。然后输入命令“:g/i/s//z/g”。再按“Enter”键，会将文本中所有的“i”替换为“z”。然后用命令“:g/z/s//i/g”将文本中的“z”替换回“i”。

⑤ 使用外部命令。按“Esc”键进入到普通状态。然后输入命令“:!ls”，这时会执行外部命令查看当前的目录，命令的执行结果显示在 vim 中，如图 2-9 所示。这时按任意键可以回到原来的 vim 编辑窗口中。

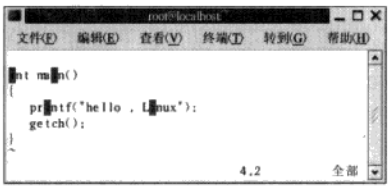


图 2-8　查找字符串

图 2-9　vim 执行外部命令的结果

⑥ 不保存退出 vim。按“Esc”键进入到普通状态。然后输入命令“:q!”，再按“Enter”键，可以不保存文件退出 vim。

2.3　gvim

gvim 是一个有着菜单操作功能的 vim 文本编辑器。除了包含 vim 所有的功能以外，还可以使用鼠标用菜单命令进行操作。gvim 可以使用命令的强大功能，如果不熟悉命令，可以使用鼠标完成各种操作。

2.3.1　文件的新建与保存

gvim 可以使用与 vim 完全相同的命令和操作。不同的是，可以使用菜单命令和对话框代替命令的输入。本节将讲述 gvim 的文件新建与保存操作。

① 打开 gvim。单击“主菜单”|“系统工具”|“终端”命令，打开系统终端。然后在终端中输入命令“gvim”，再按“Enter”键启动 gvim。gvim 的工作界面如图 2-10 所示。

② 新建文件。在 gvim 中，单击“File”|“New”菜单命令，新建一个文件。

③ 输入文本。在 gvim 的工具界面中输入一段文本，如图 2-11 所示。

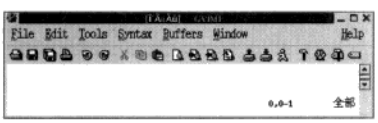


图 2-10　gvim 的工具界面

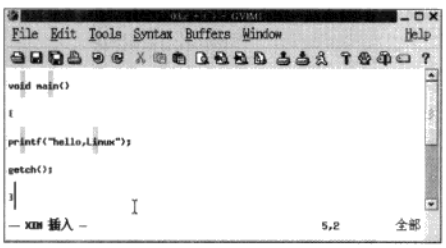


图 2-11　在 gvim 中输入一段文本

④ 移动光标。在 gvim 的文本中，单击可以移动光标的位置。

⑤ 删除字符。按退格键可以删除光标前面的一个字符。按“Delete”可以删除光标后面的一个字符。按住鼠标左键在文本上拖动选择文本，然后按“Delete”键可以删除选择的文本。

⑥ 保存文件。单击“File”|“Save”菜单命令，显示的“另存为”对话框如图 2-12 所示。单击“目录”列表中的目录，可以选择保存目录。当前的默认目录是用户主目录/root/。在“选择”文本框中输入文件名“03.c”，然后单击“确定”按钮保存文件。

⑦ 退出 gvim。单击 gvim 右上角的“关闭窗口”工具，退出 gvim。

2.3.2　查找与替换

gvim 可以使用菜单命令，在对话框中设置内容以后对文本进行查找和替换操作。这种查找与替换与 vim 中的命令是相同的，只是用菜单命令与对话框代替了命令的输入。

① 用上一节第 1 步骤相同的方法打开 gvim。

② 打开文件。在 gvim 中，单击“File” | “Open”菜单命令，显示的“编辑文件”对话框如图 2-13 所示。在“目录”列表中可以单击选择一个目录。默认目录是用户主目录。在“文件”列表中单击选择上一节保存的文件“03.c”。然后单击“确定”按钮打开文件。

③ 在文本中查找。单击“Edit” | “Find”菜单命令，显示的“查找”对话框如图 2-14 所示。

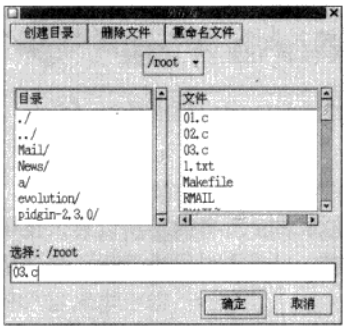


图 2-12 保存文件

图 2-13 打开一个文件

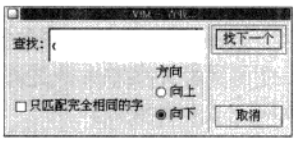


图 2-14 在文本中查找

④ 如图 2-14 所示，在“查找”文本框中输入要查找的内容“(”，然后单击“找下一个”按钮。gvim 会查找出所有的“(”。查找出的内容以黄色显示，如图 2-15 所示。光标会移动到当前光标的下一个要查找的字符上。

⑤ 替换文本。单击“Edit” | “Find and Replace”菜单命令，显示的“查找与替换”对话框如图 2-16 所示。

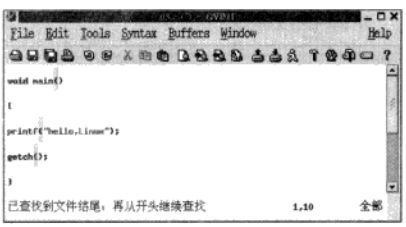


图 2-15 在文本中查找

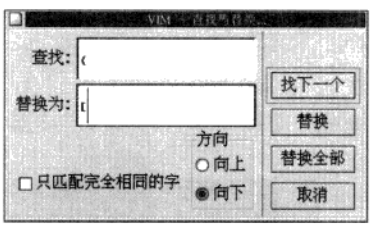


图 2-16 在 gvim 替换文本

⑥ 如图 2-16 所示，在“查找”文本框中输入“(”，在“替换为”文本框中输入“[”。单击“找下一个”按钮可以查找出下一个要替换的内容，但是不替换。单击“替换”按钮可以替换光标下面的一个字符串。单击“全部替换”按钮可以替换文本中所有匹配的字符串。

2.4 gedit

gedit 是 Red Hat Linux 9.0 下最常用的图形界面编辑器。与 vim 或 gvim 最大的不同是 gedit 采用了方便的图形界面。用户不需要输入命令就可以完成文本的编辑。gedit 的功能与操作与 Windows 系统下的记事本相似。本节将讲解 gedit 的使用。

2.4.1 gedit 的启动与打开文本

gedit 可以用命令或主菜单的方式两种方式启动。打开文件可以在终端中使用命令行，也

可以使用 gedit 中的菜单工具。

① 终端中启动 gedit。单击“主菜单”|“系统工具”|“终端”命令，打开系统终端。在终端中输入 gedit 命令，然后按“Enter”键，启动 gedit。gedit 的工具界面如图 2-17 所示。

② 在主菜单中启动 gedit。单击“主菜单”|“附件”|“文本编辑器”命令，打开 gedit。

③ 在 gedit 中打开文件。单击“文件”|“打开”菜单命令，显示的“打开文件”对话框如图 2-18 所示。在“文件夹”列表中可以选择一个目录。在“文件”列表中单击选择上一节保存的文件“03.c”。然后单击“确定”按钮打开文件。

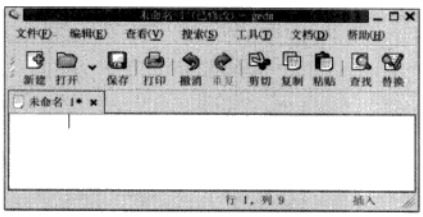


图 2-17　gedit 工具界面

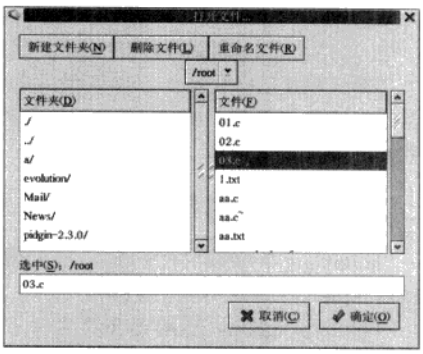


图 2-18　在 gedit 中打开一个文件

④ 可以在终端中直接用命令直接打开一个文件。在终端中输入下面的命令。

```
gedit /root/03.c
```

gedit 命令后面有一个文件名，gedit 打开时会自动打开这个文件。

⑤ 单击工具栏上的“打开文件”工具，在显示的“打开文件”对话框中打开一个文件。

2.4.2　gedit 编辑文件

gedit 可以方便地使用各种工具进行文本文件编辑。在进行文件编辑时，不需要输入命令，可使用菜单命令、工具、右键菜单等方法进行文件编辑。本节讲述在 gedit 中如何进行文件编辑操作。

① 用上一节的方法，打开 gedit。

② 在 gedit 中输入下面一段 C 语言程序代码。

```
void main()
{
   printf("hello ,Linux. ");
   getch();
}
```

③ 保存文件。单击“文件”|“另存为”菜单命令，显示的“另存为”对话框如图 2-19 所示。“另存为”对话框中已经选择当前用户的主目录。在“选中”文本框中输入文件名“04.c”。然后单击“确定”按钮保存文件。

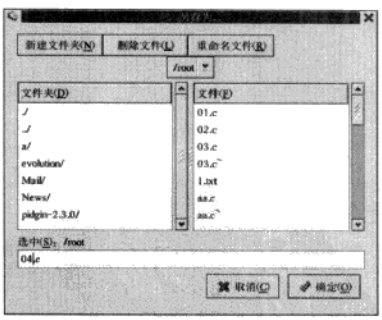


图 2-19　保存文件

④ 也可以单击工具栏中的“保存当前文件”工具保存正在编辑的文件。

⑤ 移动光标。在 gedit 的文本中单击，即可移动光标所在的位置。

⑥ 选择文本。按住鼠标左键在需要选择的文本上拖动，即可选择一段文本。

⑦ 选择一个单词。在需要选择的单词上单击，即可选择一个单词。

⑧ 文本的复制和粘贴。选择需要复制的文本以后，然后单击“编辑”|“复制”工具，可以复制选中的文本。这些基本的文件编辑操作与 Windows 系统是相同的。还有三种方法可以完成文本的复制。

- 单击工具栏中的“复制”工具。
- 按“Ctrl”+“V”快捷键复制文本。
- 右键单击选中的文本，再单击“复制”命令进行复制。

可以用这些方法进行文本的粘贴、剪切操作。

⑨ 代码缩进。如果编写的文本是程序代码，需要在文本中进行代码缩进。在需要缩进的行单击，或者选中需要缩进的多行。然后单击“编辑”|“缩进”菜单命令。

⑩ 撤销与重复操作。如果最后的编辑步骤错误，可以撤销最后的操作。单击“编辑”|“撤销”菜单命令可以撤销上一次的操作。如果需要取消上一次的撤销操作，可以单击“编辑”|“重做”菜单命令。

2.4.3 gedit 打印文件

在编辑文本或编写代码时，常常需要打印文件。gedit 提供了基本的文件打印功能。本节将讲解 gedit 如何进行文件打印。

① 打开 gedit。然后单击“文件”|“打开”命令，在“打开文件”对话框中打开上一节编辑的文件“04.c”。

② 打印预览。打印预览指的是查看当前文件的打印效果。单击“文件”|“打印预览”，显示的“打印预览”对话框如图 2-20 所示。

③ 打印设置。单击“文件”|“打印”菜单命令。显示的“打印”对话框如图 2-21 所示。在“打印”对话框的“任务”选项卡中，“打印范围”可以设置当前文件的页数打印范围。“打印份数”文本框可以设置文件文件的打印份数。默认是打印一份。

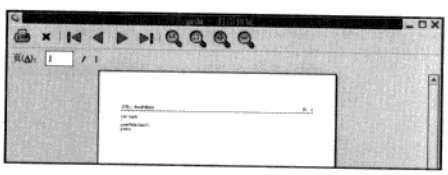


图 2-20 打印预览

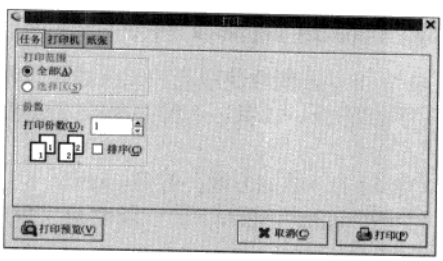


图 2-21 设置打印任务

④ 设置打印机。如果计算机上连接有多个打印机，需要在“打印机”下拉菜单中选择需要使用的打印机，如图 2-22 所示。单击“设置”下拉菜单可以选择打印机的工作方式。

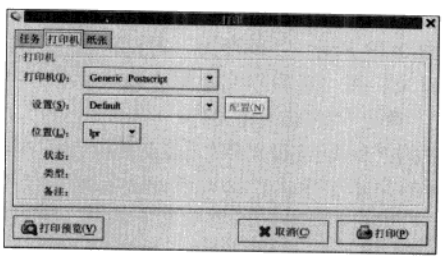


图 2-22 选择与设置打印机

⑤ 设置打印纸张。如图 2-23 所示，在“纸张大小”下拉菜单中单击选择所使用的纸张型号。然后根据打印机的情况选择“进纸方向”和“纸张方向”。然后单击“确定”按钮打印当前的文件。

2.4.4　gedit 的首选项

gedit 可以对软件的很多参数进行设置，这些设置有利于文本编辑和软件使用。本节将讲述 gedit 的首选项设置。

① 打开 gedit。在 gedit 中单击“编辑”|“首选项”菜单命令，显示的“首选项”对话框如图 2-24 所示。

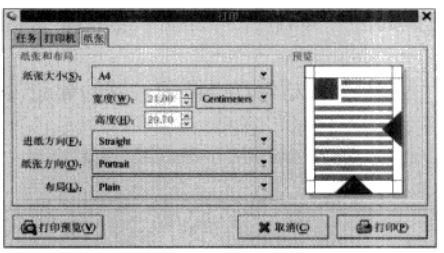


图 2-23　设置打印纸张

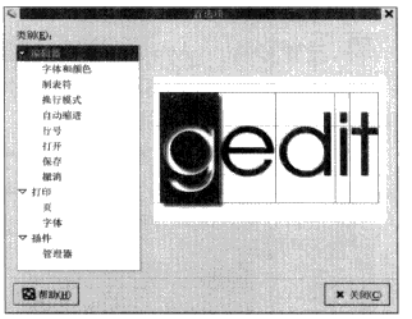


图 2-24　gedit 的“首选项”对话框

② 设置 gedit 的字体和颜色。在“首选项”对话框中，单击“类别”列表中的“字体与颜色”选项，“首选项”对话框如图 2-25 所示。

③ 如图 2-25 所示，单击“编辑器字体”后面的字体，显示的对话框如图 2-26 所示。在“字体族”列表中单击选择一个字体。在“大小”列表中单击选择一个文字大小。然后单击“确定”按钮。

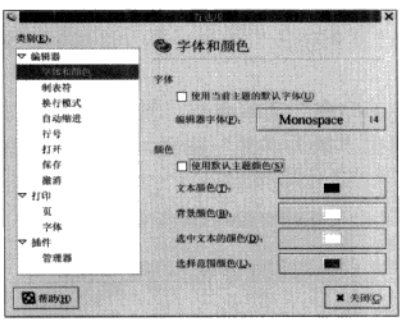


图 2-25　设置 gedit 的字体

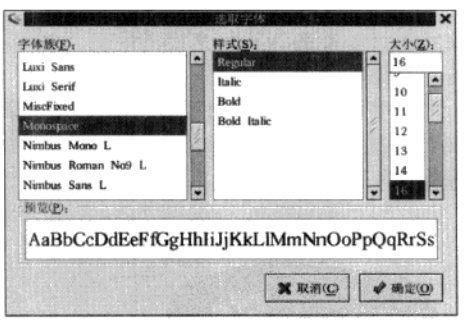


图 2-26　设置字体的大小

④ 设置显示行号。在编写代码时，需要查看代码的行号。在“类别”列表中，单击“行号”选项。然后单击选择“显示行号”复选框，如图 2-27 所示。

⑤ 设置保存选项。如图 2-28 所示，在“类别”列表中，单击“保存”选项。单击选择“在保存之前创建一个备份文件”复选框。这时，文件在保存时，会对以前的文件自动创建一个文件。如果出现错误，可以用备份文件来恢复所保存的文件。单击选择“自动保存当前文件”，并且选择一个保存时间，则在文件编辑时会定期保存当前文件。

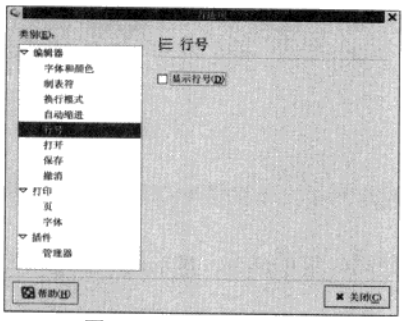


图 2-27　设置显示行号

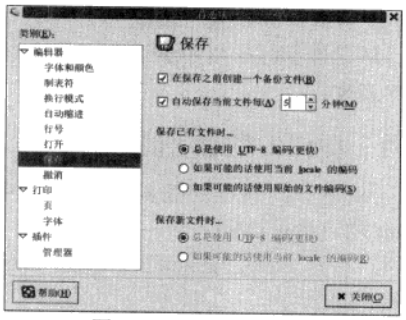


图 2-28　设置保存选项

⑥ 设置插件。gedit 的某些功能是通过插件来完成的，使用插件可以增强 gedit 的功能。如图 2-29 所示，单击“类别”列表中的“管理器”选项。在“插件”列表中单击选择“ASCII 码表”和“文档统计”两个插件。

⑦ 单击“首选项”对话框中的“关闭”按钮，gedit 会自动保存当前的设置。如图 2-30 所示，gedit 的配置改变了文本的字体大小，编辑的文本前面显示了行号。

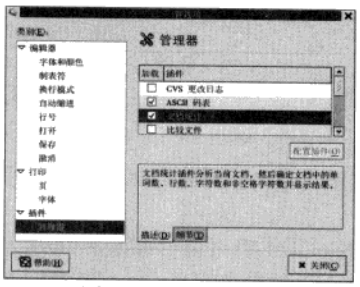


图 2-29　设置插件

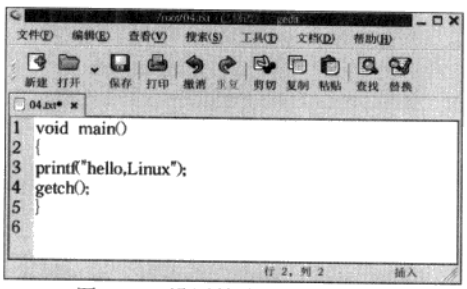


图 2-30　设置首选项以后的 gedit

⑧ 使用文档统计插件。在“首选项”中启用“文档统计”插件以后，可以使用 gedit 中的文档统计功能统计当前文档的信息。单击“工具”|“文档统计”菜单命令，显示的“文档统计”对话框如图 2-31 所示。

⑨ 使用 ASCII 码表工具。单击“查看”|“ASCII 码表”菜单命令，显示的“ASCII”码表对话框如图 2-32 所示。单击选择一个字符，然后单击“插入字符”按钮，可以在 gedit 的光标处插入这一个字符。

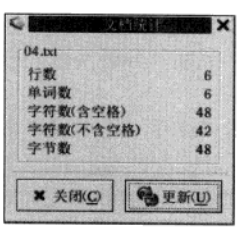


图 2-31　文档统计

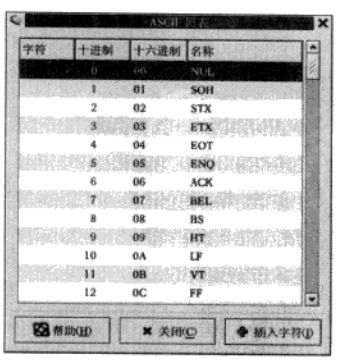


图 2-32　ASCII 码表工具

2.5　小结

本章讲述了 vim、gvim、gedit 三种 Linux 系统下常用的文本编器软件。其中 vim 是一个全字符操作的文本编辑器，学习的重点是各种命令的记忆和使用。gedit 是一个图形界面的文本编辑器，操作非常简单。gvim 结合了 vim 命令操作与 gedit 界面操作的两种功能。vim 是本章学习的难点，在 Linux 全字符界面的管理中，需要灵活运用 vim 进行文本编辑。

2.6　习题

一、填空题

1．vim 的工作模式指的是 vim 不同的使用方式。vim 有_____模式、_____模式和_____三种工作模式。

2．在编辑时，如果操作错误而修改了原有的文本，可以使用取消命令来取消之前的修改操作。其中_____用于取消上一次修改，_____将当前行恢复到修改前的状态。

3．vim 有着强大的替换功能。除了进行字符串替换以外，还可以使用正则表达式进行替换。_____将当前行中所有字符串 p1 用字符串 p2 替代。_____将第 n1 至 n2 行中所有字符串 p1 用字符串 p2 替代。将文件中所有 p1 均用 p2 替换的命令是_____。

4．vim 编辑器可以用 set 命令来设置一些特定的选项来定制编辑环境。使用_____列出所有选项设置情况。使用_____显示行号。

二、选择题

1．关于 vim 的说法中正确的是（　　）。

A．vim 是一款 C 语言编译器

B．vim 是 vi 的增强版，但是与 vi 不兼容

C．vim 使文本界面下全屏幕编辑器

D．vim 可以支持鼠标和菜单指令操作，不用 vim 的命令同样可以正常工作

2．下面哪些指令不能实现保存并退出 vim 的功能（　　）。

A．使用“:wq”指令

B．使用“ZZ”指令

C．使用“:q!”指令

D．先使用“:w”指令，在使用“:q”指令

3．vim 中从编辑模式切换到命令模式使用（　　）键。

A．Del　　B．Ctr+Alt

C．Esc　　D．Tab

三、简答题

1．简述使用 vim 编辑器的工作模式及其切换方法。

2．简述使用 vim 进行基本的文本编辑的常用命令及其使用方法。

第二篇　shell 编程篇

第 3 章　Linux 常用命令

Linux 系统虽然提供了友好的图形用户界面，但是，在进行软件开发和系统维护过程中，经常使用命令行的工作方式。所以，软件开发人员和系统管理员还必须掌握 Linux 下的常用命令。本章将重点介绍 Linux 下的目录与文件管理命令和帮助类命令，主要内容如下：

- Linux 下的常用命令介绍；
- Linux 获得系统帮助的命令。

3.1　目录与文件命令

在软件开发过程中，开发者经常需要进行大量的文件与目录操作，本节将介绍 Linux 下常用的文件与目录操作命令。

3.1.1　查看目录命令 pwd

pwd 命令的作用是查看当前所在的目录。这个命令的使用很简单，没有参数，输入命令后按“Enter”键，会显示当前所在目录的绝对路径。

① 单击“主菜单”|“系统工具”|“终端”命令，打开命令行终端。

② 在终端中输入 pwd 命令，然后按“Enter”键。需要注意的是，Linux 命令都是区分大小写的。命令的输出结果如下所示。

```
/root
```

③ 显示结果为“/root”，表明当前目录是 root 用户的主目录。

注意：当打开命令行终端时，默认的目录是当前登录用户的主目录（Home Directory）。

3.1.2　切换目录命令 cd

cd 命令用于切换用户的当前工作目录。此命令的使用方法与 DOS 中的 cd 命令相似。表 3.1 列出了几种常用的 cd 命令使用方法。

表 3.1　cd 命令的使用方法

命　令	含　义
cd /	切换到根目录
cd /bin	切换到根目录下的 bin 子目录
cd bin	切换到当前所在目录下的 bin 子目录
cd ..	切换到当前目录上一级目录（父目录）
cd ../..	切换到当前目录的上一级目录的父目录
cd /local/bin	切换到根目录下 local 子目录下的 bin 子目录

续表

命　令	含　义
cd ~	切换到用户的主目录
cd	切换到用户的主目录
cd $HOME	切换到用户的主目录

注意：在 Linux 系统中，目录、计算机名、域名要用/隔开字段，而不能用\。这种规则和 Windows 系统是不同的。

3.1.3　查看目录命令 ls

查看目录指的是显示指定目录下面的子目录或文件列表。ls 命令查看指定目录时，使用选项可以设置列表的显示方式、排列顺序等。如表 3.2 所示，是 ls 命令的常用选项。

表 3.2　ls 命令的参数

选　项	含　义
-a	显示所有文件及目录，包括隐藏文件（文件名以点开头的文件或目录）
-l	显示文件的详细列表，显示文件名、文件型态、权限、拥有者、文件大小等信息
-r	将文件以字母的倒序排列显示
-t	将文件依建立时间先后排列显示
-A	同-a，但不列出“.”（当前目录）及 “..”（父目录）
-F	在列出的文件名称后加表示文件类型的符号。例如：可执行文件后面添加“*”，目录则添加“/”
-R	递归列出目录下的所有文件

下面的操作，是使用 cd 命令打开目录和 ls 命令查看目录的实例。

① 单击“主菜单”|“系统工具”|“终端”命令，打开命令行终端。

② 在终端中输入 ls 命令，然后按“Enter”键。显示的结果如下所示。

```
01.c     4.1.txt        a.out          Makefile
02.c     4.2.sh         a.sh           movie_huayudianying01.jpg
```

③ 显示隐藏文件，在终端中输入命令“ls –a”，显示的结果如下所示。

```
.              .bash_profile                .metacity
..             .bashrc                      .mime.types
01.c           b.sh                         .moc
02.c           b,txt                         movie_huayudianying01.jpg
```

提示：上面的结果“..”表示上一级目录，“.”表示当前目录。以“.”开头的文件表示隐藏文件。

④ 显示文件详细列表，在终端中输入命令“ls –l”。显示的结果如下所示。在列表中显示了文件的权限、所有者、创建时间、文件名等信息。

```
-rw-r--r--   1 root    root         55 12 月 10 04:04 01.c
-rw-r--r--   1 root    root         47 12 月 10 04:29 02.c
-rw-r--r--   1 root    root         48 12 月 10 05:57 03.c
-rw-r--r--   1 root    root         48 12 月 10 04:49 03.c~
-rw-r--r--   1 root    root         48 12 月 10 06:00 04.txt
-rw-r--r--   1 root    root         17 2005-12-09  1.txt
```

⑤ 在文件列表中显示一个标志，在终端中输入命令“ls –F”，显示结果如下所示。在列表中，可执行文件添加了 “*”，对目录则添加“/”标志。

```
01.c      4.1.txt*       a.out*          Makefile*
02.c      4.2.sh*        a.sh*           movie_huayudianying01.jpg
03.c      4.2.txt*       a.txt           movie_huayudianying02.jpg
```

说明：多个参数可以组合的。例如：“ls –aF”，表示显示所有的文件，并且添加一个标志。“ls –alF”表示以列表的形式显示所有文件，同时添加一个标志。

⑥ 用“*”可以匹配文件名中的任意字符串。例如：“ls *.c”表示列出扩展名为“.c”的文件，即列出 c 语言文件。同时可以在 ls 的后面添加参数。例如：输入命令“ls –l *.c”，输出的结果如下所示。

```
-rw-r--r--   1 root    root         55 12 月 10 04:04 01.c
-rw-r--r--   1 root    root         47 12 月 10 04:29 02.c
-rw-r--r--   1 root    root         48 12 月 10 05:57 03.c
-rw-r--r--   1 root    root         36 2005-12-10  aa.c
-rw-r--r--   1 root    root         42 2005-12-09  h.c
-rw-r--r--   1 root    root         47 2005-12-09  hello.c
```

⑦ 用“?”可以匹配一个字符。例如：输入命令“ls 0?.c”，表示第一个字符为“0”，扩展名为“.c”，文件名有两个字符的文件。结果如下所示。

```
01.c 02.c 03.c
```

⑧ 输入命令“cd /”，表示进入到根目录。输入“ls”，显示结果如下所示。

```
bin   dev  home   lib        misc  opt   root  tmp  var
boot  etc  initrd  lost+found  mnt   proc  sbin  usr
```

⑨ 输入命令“cd ~”，表示进入到用户的主目录。如果是 root 用户，则表示进入到目录“/root”。

3.1.4 建立文件命令 touch

touch 命令用于建立空文件。Linux 系统中的文件所注重的是权限和文件内容，而不注重扩展名，所以可以使用随意的扩展名。如果需要建立的文件已经存在，touch 命令可以使文件的最后访问时间变成当前时间。下面的操作，是使用 touch 命令建立空文件的例子。

① 单击“主菜单” | “系统工具” | “终端”命令，打开一个终端。

② 在终端中输入命令 touch file1，然后按“Enter”键，建立一个文件 file1。

③ 以列表的方式查看这个文件，输入命令 ls –l file1。显示的结果如下所示。

```
-rw-r--r--   1 root    root          0 12 月 10 20:14 file1
```

④ 用 touch 命令可以更改文件的最后访问时间。输入命令“touch file1”。

⑤ 查看这个文件，输入命令“ls –l file1”，显示结果如下所示。可以发现，文件的最后访问时间已经改变。

```
-rw-r--r--   1 root    root          0 12 月 10 20:16 file1
```

⑥ 在 Linux 系统中，文本文件的扩展名并非必须是“.txt”。任意的空文件都可以作为文本文件进行编辑。在终端中输入命令“vim file1”，可以对这个文件进行编辑。

提示：touch 指令还可以用来将文件的时间属性设置为当前的系统时间。

3.1.5　建立与删除文件夹命令 mkdir、rmdir

命令 mkdir 用于建立一个目录。建立的目录不一定是在当前目录，可以在任意一个目录中建立一个目录。参数-p 可以指定建立多级目录。命令 rmdir 用于删除一个空目录，也可以使用-p 命令表示删除多级目录。下面的操作是目录建立与删除的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 输入命令 mkdir mydir，然后按“Enter”键。这时会在用户的主目录中建立一个目录 mydir。

③ 输入命令 rmdir mydir，可以删除目录 mydir。

④ 输入命令 mkdir –p mydir1/mydir2。表示在目录 mydir1 中建立目录 mydir2。但是 mydir1 是不存在的，用参数-p 可以同时建立目录 mydir1。

⑤ 输入命令 rmdir mydir1，目录 mydir1 是非空的，不可删除。输出的结果如下所示。

```
'mydir1': 目录非空
```

⑥ 输入命令 rmdir –p mydir1/mydir2。会删除目录 mydir2。同时，mydir1 是空目录，则会同时删除目录 mydir1。

注意：rmdir 指令只能删除空目录，对于非空目录可以使用带“-R”选项的 rm 指令删除。

3.1.6　文件复制命令 cp

cp 命令可将文件或目录复制到另一文件或目录，该命令有多个选项，可以实现很多种功能。cp 命令的使用方法如下所示。

```
cp [参数] 源文件或目录 目标文件或目录
```

cp 命令的可选参数如下所示。

- -a：-a 选项通常在复制目录时使用。它保留链接、文件属性，并复制目录下的所有内容。其作用与组合参数 dpR 等价。
- -d：复制时保留链接。这里所说的链接相当于 Windows 系统中的快捷方式。
- -f：覆盖已经存在的目标文件而不给出提示。
- -i：与 f 选项相反，在覆盖目标文件之前给出提示，要求用户确认是否覆盖。回答 y 时目标文件将被覆盖。
- -p：除复制文件的内容外，同时把修改时间和访问权限也复制到新文件中。
- -r：若给出的源文件是一个目录文件，此时将复制该目录下所有的子目录和文件。
- -l：不复制文件，只是生成链接文件。

需要注意的是，默认情况下用 cp 命令复制文件时，如果目标文件已经存在，则会覆盖目标文件而不给出提示。被覆盖的文件是不能恢复的。为防止破坏另一个文件，复制文件时最好使用-i 选项。例如下面的操作，是文件复制的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 输入命令 cp 01.c 11.c，然后按“Enter”键。表示将主目录下的文件 01.c 复制一份放到当前目录，文件名为 11.c。

③ 建立一个文件夹 c，输入命令 mkdir c。

④ 复制多个文件。输入命令如下所示，表示将当前目录下的所有扩展名为.c 的文件复制

到文件夹 c 中，文件名不变。

```
cp *.c c/
```

⑤ 提示是否覆盖文件。将文件 01.c 复制到目录 c 中，因为目录中已经有该文件，需要加 -i 选项确认是否覆盖目标文件。输入命令如下所示。

```
cp -i 01.c /c
```

⑥ 终端中会输出下面的提示，用户输入 y 表示确认覆盖。

```
cp: '是否覆盖/c? '
```

⑦ 输入命令 cp –r /c /d。将目录 c 中所有的文件和目录复制到目录 d 中。如果目录 d 不存在，则建立这个目录。

⑧ 输入命令 cp d/01.c .。将目录 d 中的文件 01.c 复制到当前目录中，文件名不变。

注意：大多数的 Linux 发行版都为 cp 命令设置了命令别名，当用户在命令行中输入 cp 指令时，等价于输入了“cp -i”。

3.1.7 文件移动和更名命令 mv

在 Linux 系统中，除非把文件移动到不同的分区中，否则文件在移动时依旧是以前的文件。文件的重命名相当于文件的移动。文件移动命令的参数如下所示。

- -b：若需覆盖文件，则覆盖前先行备份。
- -f：若目标文件或目录与现有的文件或目录同名，则直接覆盖已有的文件或目录。
- -i：覆盖前询问用户。
- -S：与-b 参数一并使用，可指定备份文件所要附加的后缀。
- -u：在移动或文件更名时，若目标文件已存在，且其文件日期比源文件新，则不覆盖目标文件。
- -v：执行时显示详细的信息。
- -V：与-b 参数一并使用，指定备份的方法。

下面的操作，是使用 mv 命令移动和重命名文件的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 输入下面的命令，将用户主目录下面的文件 01.c 更名为文件 001.c。

```
mv 01.c 001.c
```

③ 输入下面的命令，将用户主目录下面所有扩展名为.c 的文件移动到目录 c 中。

```
mv *.c /c
```

④ 输入下面的命令，将用户主目录下的文件 001.c 移动到目录 c 中。覆盖文件时不提示信息。

```
mv -f 001.c c/
```

注意：当使用 mv 指令同时移动多个文件或目录时，最后一个参数必须是一个存在的目录。

3.1.8 删除文件与目录命令 rm

命令 rm 可以删除文件或目录。如果要删除目录，需要添加参数-r。在 Red Hat Linux 9.0

中，删除文件时默认添加了-i 参数，会提示用户是否删除。rm 的参数如下所示。

- -d：把删除的目录的硬连接数据删除，删除该目录。
- -f：强制删除文件或目录。
- -i：删除文件或目录之前询问用户。
- -r：递归处理，将指定目录下的所有文件及子目录全部删除。
- -v：显示命令执行过程。

例如下面的操作，是使用 rm 删除文件和目录的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入下面的命令，删除目录 c 下所有以.c 为扩展名的文件。

```
rm c/*.c
```

③ 终端会依次提示是否删除目录中的文件。输入 y 删除文件，输入 n 取消删除。

④ 输入下面的命令删除目录 d 中所有的文件，并且不提示信息。

```
rm -rf d/
```

3.1.9　创建链接命令 ln

Linux 中的链接相当于 Windows 系统中的快捷方式。打开这个链接时，可以打开相应的文件。ln 命令被用来创建链接文件。链接有软件链接和硬链接两种，这两种链接的含义是不同的。

- 硬链接：链接文件与原文件一样，指文件系统中的文件。建立硬链接时，链接文件和被链接文件必须位于同一个分区中，并且不能建立指向目录的硬链接。
- 软链接：软链接即符号链接，只是建立一个简单的快捷方式。用户打开这个快捷方式时，会自动打开被链接的文件。如果 ln 命令加上-s 选项，则建立符号链接。

下面的操作，是使用 ln 命令建立链接的实例。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 输入下面的命令，对文件 01.c 建立一个 001.c 的硬链接。

```
ln 01.c 001.c
```

③ 输入下面的命令，对文件 01.c 建立一个 002.c 的软链接。

```
ln -s 01.c 002.c
```

④ 文件同名时提示信息。输入下面的命令，对文件 01.c 建立一个 002.c 的软链接。这时会提示是否覆盖已有文件。

```
ln -si 01.c 002.c
```

提示：使用 ln 指令建立硬链接时，源文件和目标文件的索引节点相同。删除任何一个文件，另一个文件不受影响。修改任何一个文件的内容，另一个文件的内容也产生相同的变化。

3.1.10　查看文件信息命令 wc

wc 命令可以查看一个文件的行数、字数、字符数的信息。该命令的使用方法比较简单，可以使用下面的参数选择输出的信息。

- -l：显示文件的文本行数。
- -w：显示文件的字数。
- -c：显示文件中的字符数。

例如查看当前目录下的文件 01.c 的信息，可以输入下面的命令。

```
wc 01.c
```

命令的显示结果如下所示。

```
5   8   55  01.c
```

显示的结果表示文本中有 5 行，共有 8 个单词，55 个字符。如果要显示某一项的信息，需要在 wc 命令后面加入一个参数。下面的命令可以显示这个文件的行数。

```
wc -l 01.c
```

命令的运行结果如下所示，表示这个文件有 5 行。

```
5
```

3.1.11 文件查看命令 cat

cat 命令可以输出文件的内容。该命令可选的参数如下所示。

- -n：由 1 开始对所有输出的行进行编号。
- -b：和-n 相似，但对空白行不编号。
- -s：当遇到有连续两行以上的空白行，就替换为一行。

下面的操作，是使用 cat 命令查看文件的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入命令 cat 01.c，然后按“Enter”键，终端中会显示文件 01.c 所有的内容。

③ cat 命令可以显示多个文件的内容。在终端中输入下面的命令。

```
cat 01.c 02.c 03.c
```

④ 上面的命令可以连续显示三个文件的内容。可以将输出结果定向到一个文件中。在终端中输入下面的命令，将三个文件所有的内容输出到文件 123.c 中。

```
cat 01.c 02.c 03.c >123.c
```

⑤ 显示文件时输出行号，在终端中输入下面的命令。

```
cat -n 01.c
```

⑥ 把一个文件添加行号以后保存到另一个文件中。在终端中输入下面的命令。

```
cat -n 01.c >003.c
```

⑦ 在终端中输入命令 cat 003.c，这个文件的显示结果如下所示。文件中的行号是上一命令产生的行号。

```
1  int main()
2  {
3     printf("hello , Linux");
4     getch();
5  }
```

3.1.12 查看文件视图命令 head 与 tail

head 命令用来查看文件前面的 *n* 行。如果没有设定 *n* 的值，则默认显示文件的前 10 行。tail 命令与 head 命令的使用方法相同，用来查看文件末尾的内容。如表 3.3 所示是这两个命令的使用方法。

表 3.3 ls 命令的参数

参　数	含　义
head –n 5 a.txt	查看文件的前 5 行

续表

参　数	含　义
head 400b a.txt	查看文件的前 400 个字节
head 4k a.txt	查看文件的前 4KB 的文件内容
head 3m a.txt	查看文件的前 3MB 的文件内容
tail –n5 a.txt	查看文件的后 5 行
tail 400b a.txt	查看文件的后 400 个字节的内容
tail 4k a.txt	查看文件的后 4KB 的文件内容
tail 3m a.txt	查看文件的后 3MB 的文件内容

提示：使用 tail 指令的“-f”选项可以监视文本文件的实时变化。

3.1.13　文件权限设置命令 chmod

文件权限指的是文件是否可以进行执行、写入、读取等操作。在 Linux 系统中，用户可以对自己的文件设置权限，不同的用户和组具有不同的文件权限。chmod 命令可以设置和更改权限的方式。在设置权限之前，需要理解文件权限的种类。

文件的权限分为三级，三级的用户可以进行不同的权限设置。这三种用户和表示方法如下所示。

- 所有者：用 u 表示，创建这个文件的用户。
- 群组：用 g 表示，是所有者同一群组中的用户。
- 其他：用 o 表示，除了所有者和同组用户以外的用户。

除此之外，可以用 a 表示所有的用户。权限需要用权限字符表示，r 表示可读取，w 表示可写入，x 表示可执行，权限的更改需要使用权限设定符号。+表示增加权限、-表示取消权限、=表示唯一设定权限。chmod 命令的可选参数如下所示。

- -c：若文件权限已经更改，显示其更改动作。
- -f：若文件权限无法被更改，则不显示错误信息。
- -v：显示文件变更的详细资料。
- -R：对目录下的所有文件与子目录进行相同的权限设置。

例如下面的操作，是用 chmod 命令对文件的权限进行设置的例子。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入下面的命令，对文件 01.c 的所有者添加可执行权限。

```
chmod +x 01.c
```

③ 在终端中输入下面的命令，对文件 01.c 的同组用户删除可写权限。经过这种设置以后，同组用户就无法编辑文件。

```
chmod g-x 01.c
```

④ 在终端中输入下面的命令，对文件 01.c 的所有用户添加可读权限。

```
chmod a+r 01.c
```

⑤ 在终端中输入下面的命令，对文件 01.c 的其他用户和同组用户添加可读权限，同时取消可写权限。

```
chmod go+r-w 01.c
```

⑥ 在终端中输入 vim 命令，打开 vim。

⑦ 在 vim 中按 i 键，进入插入模式，在文件中输入下面的内容。

```
echo "hello ,Linux"
```

⑧ 按“Esc”键进入普通模式。然后输入:w 3.1.sh 命令保存文件。输入:q 命令退出 vim。

⑨ 运行已经编辑的文件。在终端中输入如下所示的命令。

```
./3.1.sh
```

⑩ 终端显示权限不够，文件不能运行。在终端中输入下面的命令，对用户添加可执行权限。

```
chmod u+x 3.1.sh
```

⑪ 输入命令./3.1.sh，这个文件的运行结果如下所示。上一步骤的命令对这个文件添加了可执行权限。

```
hello ,Linux
```

⑫ 查看文件权限。ls –l 命令可以显示一个文件的权限。在终端中输入下面的命令，可以显示文件 3.1.sh 的权限。

```
ls -l 3.1.sh
```

⑬ 命令的运行结果如下所示。

```
-rwxr-xr-x   1 root    root         20 12月 10 22:49 3.1.sh
```

⑭ 显示的-rwxr-xr-x 表示文件的权限。第一个字符如果不是-，则表示这个文件不是普通文件。后面的字符串三个字符为一组。rwx 表示所有者的权限是可读、可写、可执行。r-x 表示同组用户可读可执行不可写。最后的 r-x 表示其他用户可读、可写、可执行。

提示：使用 chmod 指令的“-r”选项可以递归地改变指定目录下所有文件与子目录的权限。

3.1.14 查找文件命令 find

find 命令可以从目录和子目录中搜索文件。find 命令的最简单使用方式如下所示。

```
fine 目录 -name 文件名
```

例如，要查找计算机中的一个名为 httpd.sh 的文件，而不知道相应的目录，可以在终端中输入下面的命令。

```
find / -name httpd.sh
```

终端搜索出硬盘中所有文件名为 httpd.sh 的文件。如果缩小查找范围，可以输入下面的命令。

```
find /usr -name httpd.sh
```

在这个命令中，文件名支持*与?等通配符。例如下面的命令，可以查找 root 目录中所有扩展名为.c 的文件。

```
fine /root -name *.c
```

3.2 man 系统帮助命令

Linux 系统中有着大量的命令操作，用户不一定完全掌握这些命令的使用方法。man 命令可以对系统中的命令提供详细的帮助信息。本节将讲述如何使用 man 命令查看系统帮助文档。

3.2.1　man 手册的使用方法

man 是 manual 的缩写，意思是在线帮助手册。如果需要查找某个命令的帮助信息，可以在终端中输入 man，然后输入这个命令即可。

默认情况下，终端会使用一个名为 less 的文件阅读器来查阅帮助手册。less 的使用非常简单。q 命令退出手册页。键盘向上和向下方向键可以一次一行地上下移动手册。翻页键 Page Up 和 Page Down 可以向上或向下翻页。空格键可以一次向下翻一页。“Enter”键可以一次向下移动一行。/加上需要搜索的文本可以在帮助文档中查找需要的内容。例如下面的操作，可以查看 ls 命令的帮助手册。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入 man ls 命令，然后按“Enter”键。终端中显示的 ls 命令帮助文本如下所示。

```
LS(1)          FSFLS(1)
NAME
 ls - list directory contents
SYNOPSIS
 ls [OPTION]... [FILE]...
DESCRIPTION
 Listinformationaboutthe FILEs (the current directory by default).
 Sort entries alphabetically if none of -cftuSUX nor --sort.
 Mandatory arguments to long options aremandatoryforshortoptions too.
 -a, --all
do not hide entries starting with .
 -A, --almost-all
do not list implied . and ..
 --author
print the author of each file
 -b, --escape
print octal escapes for nongraphic characters
```

③ 翻页键“Page Up”和“Page Down”可以对帮助文档向上或向下翻页。查看文档以后，按“q”键可以退出帮助文档。

注意：man 命令可以显示多本手册页，默认情况下显示的为命令手册。

3.2.2　帮助文档的结构

从上面的实例可知，帮助文档可分为不同的小节。每个小节讲述命令不同方面的内容，这些小节的标志和含义如表 3.4 所示。在使用 man 手册时需要识别这些小节，在正确的小节中找到需要的内容。

表 3.4　man 手册的小节和含义

小节标题	内　容
NAME	该命令的简单说明
SYNOPSIS	大致说明。对于命令来说是命令的语法，对于函数来说是函数的定义
AVAILABILITY	可用性说明
DESCRIPTION	该段内容的一个简明介绍，对内容进行概括
OPTIONS	命令参数的详细解释
RETURN VALUE	返回值

续表

小节标题	内　容
ERRORS	运行出错后，可能出现的出错值及引起错误的原因
FILES	涉及的文件
ENVIROMENT	相关的环境变量
NOTES	需要提醒的细节
BUGS	该命令或函数存在的问题，以及这些问题可能会造成怎样的影响
HISTORY	该命令和函数的历史和版本
SEE ALSO	参考资料
AUTHOR	作者
OTHERS	与这一命令和函数相关的内容

3.3　小结

本章主要介绍了 Linux 中常用的文件操作命令和系统帮助文档的使用。在以后的编程学习中，需要使用这些命令对文件进行各操作，其中文件权限的设置是一个难点，在使用这个命令时，需要注意命令参数的作用与含义。读者在学习中遇到问题，可以使用 man 帮助手册查看相关的帮助信息。

3.4　习题

一、填空题

1．_____命令的作用是查看当前所在的目录。_____命令用于切换用户的当前工作目录。

2．_____命令用于建立空文件和修改文件的时间属性。

3．_____命令用于实现移动文件和为文件改名操作。_____命令用于删除文件和目录。

4．如果文件的内容比较长，希望查看文件的尾部内容，使用_____命令比较合适。

5．Linux 操作系统中用于设置和修改文件或者目录权限的指令是_____。

二、选择题

1．关于 ls 指令的说法错误的是（　　）。

A．ls 指令默认不显示隐藏文件

B．默认情况下，ls 指令仅显示文件名和目录名称

C．ls 指令的全程为 list，用于列出指定目录下的文件列表

D．使用 ls 指令的-l 选项可以显示所有以点开头的文件和目录

2．关于 rmdir 的说法正确的是（　　）。

A．rmdkir 指令可以删除任何空目录

B．rmdir 指令默认情况下会将指定目录下的所有文件及子目录全部删除

C．rmdir 指令不但能够删除目录，还能够删除普通文件

D．rmdir 只能删除空目录，如果目录不为空，必须先将目录下的文件及子目录删除后才能删除此目录

3．关于 find 指令的说法错误的是（　　）。

A．find 指令不但可以查找文件，而且能够指定对找到的文件执行某种操作（例如：删除）

B．find 指令无须指定搜索的起始目录，默认情况下 find 从根目录下开始搜索

C．find 指令可以指定不同的搜索条件，例如：按照文件名搜索，按照文件的所有者搜索

D．find 指令支持通配符操作

4．Linux 操作系统中获得指令帮助的方法错误的是（　　）。

A．使用 man 指令可以获得给定指令的使用帮助

B．使用 man 指令无法获得 Linux 中配置文件的语法帮助

C．info 是系统中的另一个帮助文档，和 man 帮助文档的功能相似

D．linux 的绝大多数指令都有“--help”选项，可以显示指令的快速使用参考

三、简答题

1．Linux 下的文件和目录操作类命令有哪些，如何使用？

2．如何获得 Linux 下指令的帮助信息？

第 4 章　shell 编程

shell 是 Linux 中的命令解释程序，它负责将用户输入的命令传递给内核执行并将执行结果反馈给用户。熟练掌握 shell 的使用将提高用户在命令行中的工作效率。利用 shell 程序还可以编写 shell 脚本，把经常性的工作转换为 shell 程序，方便以后使用。本章介绍 Linux 下的默认 shell 程序 bash 的使用及脚本编程，主要内容如下：

- shell 的基本使用；
- shell 程序的基本结构；
- shell 中的局部变量和环境变量；
- shell 中的运算符；
- shelll 中的输入与输出；
- shell 中的测试语句与流程控制语句。

4.1　bash 的使用

在 Linux 操作系统中有多种 shell 可供使用，在 Red Hat Linux 9.0 中默认使用的 shell 是 bash。用户成功登录系统后，shell 将成为用户与系统内核进行交互的桥梁，直至用户退出系统。Linux 系统中所有用户都有一个默认的 shell。用户的默认 shell 在系统文件/etc/passwd 中指定。

4.1.1　bash 的内部命令与外部命令

Linux 的命令可以分为内部命令和外部命令。内部命令在 shell 启动时就调入内存，是常驻内存的。而外部命令是系统的软件功能，用户需要时才从硬盘中读入内存。例如下面的命令就是几个常用的内部命令。

- exit：终止当前 shell 的执行。
- export：设置一个环境变量，当前 shell 的所有子进程都可以访问此环境变量。
- kill：终止某个进程的执行。需要传递进程 PID 参数，终止对应进程的执行。

提示：外部命令的查找路径由环境变量 PATH 定义。

4.1.2　命令补齐功能

命令补齐可以自动补齐没有输入完整的命令。当用户不能拼写整个命令时，只需要输入开头的几个字符，然后按下“Tab”键即可调用 bash 的命令补齐功能。如果前面几个字符输入没有错误，并且以这几个字符开头的命令只有一个，系统会自动补齐整个命令。如果以这几个字符开头的目录不唯一，则再次按下“Tab”键时，会显示所有以这几个字符开头的命令。命令补齐不但能够对命令输入进行提示，还可以加快命令输入的速度。例如下面的操作使用了 bash 的命令补齐功能。

（1）单击“主菜单”|“系统工具”|“终端”命令，打开系统终端。

（2）在终端中输入“ifco”，然后按“Tab”键，这时会自动补齐为“ifconfig”命令。

（3）在终端中输入“ch”，然后按两次“Tab”键，这时会列出所有以 ch 开头的命令，这些命令如下所示。这时可以使用这些提示书写相关命令。

```
chacl    cheatmake    chinput      chmoddic     chroot
chage    checkXML     chkconfig    chooser      chsh
chat     chfn         chkfontpath  chown        chvt
chattr   chgrp        chmod        chpasswd
```

4.1.3　命令通配符

所谓通配符，就是指可以在命令中用一个字符来代替一系列字符或字符串。bash 中有三种通配符，其中?和[]代表单个字符。*可以代表一个或多个字符，也可以是空字符串。

- *：匹配任何字符和字符串，包括空字符串。
- ?：匹配任意一个字符。例如，?abc，可以匹配任何以 abc 结束，并且以任意字符开头的含有四个字符的字符串。
- [...]：匹配括号里列出的任何单字符。比如 abc[def]，可以匹配以 abc 开头，以 def 中任意一个字符结尾的字符串。

例如下面的例子在命令中使用通配符。

（1）从主菜单中打开一个终端。

（2）查看主目录下所有的 C 程序文件。C 程序文件的扩展名都是.c。所以输入以下的命令。

```
ls *.c                                    注意：用*匹配一个任意字符串。
```

命令的运行结果如下所示。

```
01.c  02.c  03.c  aa.c  h.c  hello.c
```

（3）列出用户主目录中以 0 开头的 C 程序文件，输入的命令如下所示。

```
ls 0*.c                                   注意：用*匹配一个任意字符串。
```

命令的运行结果如下所示。

```
01.c  02.c  03.c
```

（4）列出用户主目录中文件名只有 2 个字符的 C 程序文件。输入的命令如下所示。

```
ls ??.c                                   注意：用?匹配一个单一字符。
```

命令的运行结果如下所示。

```
01.c  02.c  03.c  aa.c
```

4.1.4　使用命令的历史记录

在终端中，如果需要再次使用已经输入过的命令，按向上方向键可以依次显示以前的命令。查找到需要的命令以后，按“Enter”键即可执行这个命令。用此方法，可以向前查找已经输入过的命令。history 命令可以显示所有的命令记录列表，命令的用法如下所示。

```
history [n]
```

参数 *n* 是一个可选的整数。当没有参数时，会列出以前执行过的所有命令。有参数 *n* 时，会列出最后执行的 *n* 个命令。例如用下面的命令来查看已经执行过的操作。

```
history 5
```

这个命令会显示最后 5 个命令的操作列表，结果如下所示。

```
152  ls ??.c
```

```
153  fc
154  ls ??.c
155  ls
156  history 5
```

4.1.5 命令的别名

命令别名指的是自定义命令代替其他命令，可以作为其他命令的缩写，用来减少键盘输入。别名的定义命令为 alias，用法如下所示。

```
alias list='ls -l'                    注释：定义一个文件列表的别名。
alias allfile='ls -a'                 注释：定义显示所有文件命令的别名。
alias lsc='ls *.c'                    注释：定义显示所有 c 程序文件命令的别名。
```

注意：定义别名时，等号的两边不能有空格，命令别名使用单引号括起来。

如果想取消别名，可以使用 unalias 命令。例如下面的命令可取消上面的代码定义的 lsc 别名。

```
unalias lsc
```

4.2 shell 程序的基本结构

shell 程序就是指将一系列的 Linux 程序通过流程控制写在一个文本文件中，由 shell 读取此文本文件并且依次执行文件中的这些程序。本节将以一个简单的 shell 程序为例讲解 shell 程序的结构。

① 打开终端，在终端中输入 vim 命令，按“Enter”键进入 vim。

② 在 vim 中按“i”键进入插入模式。然后输出下面的文本。

```
01  #!/bin/bash
02  #hello                            #注意：#后面的内容是 shell 程序的注释。
03  echo 'hello Linux'                #打印字符串到终端
04  echo 'this is a shell file.'
```

注意：在 shell 程序中，“#”后边的内容为注释信息，不影响程序的执行。

③ 在 vim 中，按“Esc”键进入命令模式。然后输入命令:w a.sh，保存这个文件到用户主目录下，文件名为 a.sh。

④ 输入命令:q，退出 vim。

⑤ 输入下面的命令对文件 a.sh 添加可执行的权限。文本文件是没有执行权限的。

```
chmod +x a.sh
```

注意：在 Linux 系统中，不论是 shell 程序还是二进制程序，都必须具有可执行权限才能够运行。

⑥ 输入下面的命令运行这个 shell 程序。这个程序执行了两次字符串输出。

```
./a.sh
```

⑦ 程序的运行结果如下所示。

```
hello Linux
this is a shell file.
```

本例中的程序虽然简单，但包含了 shell 程序的下面一些基本特征：

① 所有 shell 程序的第一行都是以#!开头。后面跟上执行此 shell 程序的 shell 解释器完整路径。Red Hat Linux 9.0 的默认 shell 解释器是 bash。本书中所有的 shell 程序都使用 bash 来解释执行。

② 程序的第 2 行以注释的方式写出程序的名称，这是 shell 编程的一种习惯。

③ 最简单的 shell 程序就是一组 shell 命令的堆砌。在这个程序中，使用两个 echo 命令来输出两个字符串。

④ shell 程序是一个普通的文本文件，需要添加可执行权限以后才可以执行。如果试图执行一个没有可执行权限的 shell 程序，得到的输出结果如下所示。

```
bash: ./c.sh 权限不够
```

4.3　局部变量

shell 程序中需要用变量来存储程序的数据。shell 中的变量可分为局部变量、环境变量、位置变量三种。本将讲述 shell 程序中变量的使用方法。shell 语言是一种解释型语言，它与 C 或 Java 语言不同，编程时无须事先声明变量。当对一个变量进行赋值时，也就定义了变量。局部变量指的是只在当前的进程和程序中有效的变量。

shell 程序的变量是无数据类型的，可以使用同一个变量存放不同数据类型的值。变量赋值之后，只需在变量前面加一个$符号即可访问变量的值。可以用赋值符号（=）为变量赋值。变量的值如果是不带空格的字符串，则可以不加引号。变量输出使用的命令是 echo。例如下面的步骤建立一个 shell 程序，在程序中定义变量并输出变量的值。

【范例 4-1】 shell 脚本中变量的定义与赋值，示例代码如下所示。

示例代码 4-1

```
01  #!/bin/bash
02  #bianliang 4.1.sh
03  a=123                          #直接为变量赋值
04  b=1.23
05  c=xyz
06  d=efgh xyz
07  e='efgh xyz'
08  echo $a                        #打印变量的值
09  echo $b
10  echo $c
11  echo $d
12  echo $e
```

【运行结果】在终端中输入下面的命令：

```
chmod +x 4.1.sh
```

在终端中输入命令“./4.1.sh”，然后按“Enter”键运行这个 shell 程序。程序的运行结果如下：

```
123                            $a 是一个整数赋值。
1.23                           $b 是一个小数赋值。
xyz                            $c 是一个字符串赋值。
                               $d 赋值时出现空格，赋值错误。
efgh xyz                       $e 用引号将一个含空格的字符串引起来再赋值。
```

【代码解析】

- 代码第 03~07 行在赋值时完成变量的定义。
- 代码第 08~12 行依次打印变量的值。

提示：在本章后面的例子中，如果没有特殊说明，都将在 vim 中输入代码。

4.4 环境变量

环境变量是在一个用户的所有进程中都可以访问的变量。系统中常常使用环境变量来存储常用的信息。本节将讲述环境变量的查看、访问和定义等操作。

4.4.1 环境变量的查看

使用 export 命令可以查看系统的环境变量列表。打开一个终端，在一个终端中输入命令 export，然后按“Enter”键。在终端中显示的环境变量列表如下所示。

```
declare -x BASH_ENV="/root/.bashrc"
declare -x COLORTERM="gnome-terminal"
declare -x DESKTOP_STARTUP_ID=""
declare -x DISPLAY=":0.0"
declare -x GDMSESSION="Default"
declare -x GNOME_DESKTOP_SESSION_ID="Default"
declare -x GTK_RC_FILES="/etc/gtk/gtkrc:/root/.gtkrc-1.2-gnome2"
declare -x G_BROKEN_FILENAMES="1"
declare -x HISTSIZE="1000"
declare -x SHELL="/bin/bash"
declare -x SHLVL="2"
declare -x SSH_AGENT_PID="1858"
declare -x SSH_ASKPASS="/usr/libexec/openssh/gnome-ssh-askpass"
declare -x SSH_AUTH_SOCK="/tmp/ssh-XXEMD4KZ/agent.1804"
declare -x TERM="xterm"
declare -x USER="root"
declare -x USERNAME="root"
declare -x WINDOWID="29512886"
declare -x XAUTHORITY="/root/.Xauthority"
declare -x XMODIFIERS="@im=Chinput"
```

4.4.2 访问环境变量

环境变量的访问，需要在一个环境变量的前面加一个$符号。例如在终端中执行下面的命令可以输出环境变量的值。

```
echo $SSH_ASKPASS
```

程序的运行结果如下所示。

```
/usr/libexec/openssh/gnome-ssh-askpass
```

4.4.3 定义环境变量

可以用 export 命令来定义一个环境变量。环境变量的命令一般都是大写的。在终端中输入下面的命令来定义一个环境变量。

```
export XX=1234
```

可以用下面的语句来访问输出这个环境变量。

```
echo $XX
```

命令的运行结果如下所示。

```
1234
```

4.4.4　在系统配置文件中定义环境变量

上一节定义的环境变量，只在当前 shell 中运行的所有进程中有效，并没有保存到系统文件中。所以系统重启后就没法再访问这些环境变量。为了实现重启系统后，仍然能够访问定义的环境变量，可以在系统配置中加入这些环境变量的定义。环境变量的系统配置文件是/etc/profile。用下面的步骤在系统文件中查看和配置环境变量。

① 打开一个终端。在终端中输入并执行下面的命令。

```
gedit /etc/profile
```

② 文本编辑器会打开系统环境变量配置文件/etc/profile。可以在文件中发现，有下面这样的语句输出了环境变量。

```
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC
```

③在文件的最后一行输入下面的代码，添加两个环境变量。

```
export A1=hello
export A2=12345
```

④ 单击文本编辑器的“文件”|“保存”菜单命令，保存这个文件。

⑤ 重新启动计算机以后，系统中会产生上面所设置的两个环境变量。

4.4.5　位置变量

位置变量指的是 shell 程序在运行时传入的参数。程序中可以用变量的形式来调用这些参数。这些参数被存放在 1~9 的 9 个特殊变量名中，被形象地称为位置变量。同普通变量一样，位置变量用$前缀加数字来表示。例如，第 5 个参数，表示为$5。例如要向 shell 程序传递参数“Beijing is a beautiful city”，用表 4.1 来说明如何访问每一个参数。

表 4.1　shell 中的位置变量

$0	$1	$2	$3	$4	$5	$6	$7	$8	$9
bash	Beijing	is	a	Beautiful	city				

在位置变量中，$0 的值为 bash。$1 以后的变量是输入参数的列表。例如下面的 shell 程序是访问位置变量的实例。

【范例 4-2】输出 shell 中的位置变量，示例代码如下所示。

示例代码 4-2

```
01  #!/bin/bash
02  #4.2.sh;
03  echo $1;
04  echo $2;
05  echo $3;
06  echo $4;
```

【运行结果】输入下面的命令对这个文件添加可执行的权限。

```
chmod +x 4.2.sh
```

输入下面的命令执行这个程序，在命令中输入程序的参数。

```
./4.2.sh Beijing is a beautiful city
```

程序输出了$1 到$4 位置变量中的 4 个参数，运行结果如下所示。

```
Beijing
is
a
beautiful
```

【代码解析】这个程序的内容很简单，用 echo 语句输出 shell 运行时输入的 4 个参数。

4.5 shell 的运算符

shell 中的运算符可以实现变量的赋值、算术运算、测试、比较等功能，运算符是构成表达式的基础。本章将讲述 shell 中运算符的使用。

4.5.1 变量赋值

shell 中使用“=”进行变量赋值，也可以用等号来改变或初值化一个变量的值。在进行赋值时不用考虑变量的数据类型，这是由 shell 中变量数据类型的特点决定的。例如下面是使用等号进行赋值的例子。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令对一个变量进行赋值。

```
STR=123
```

③ 输入下面的命令输出这一个变量的值。

```
echo $STR
```

④ shell 会输出这一个变量的值，结果如下所示。

```
123
```

⑤ 可以对同一个变量赋于不同的值，命令如下所示。

```
STR=asdf
```

⑥ 输入下面的变量输出变量的值。

```
echo $STR
```

⑦ 变量值的输出结果如下所示。

```
asdf
```

4.5.2 算术运算符

算术运算符指的是可以在程序中实现加、减、乘、除等数学运算的运算符。shell 中常用的数学运算符如下所示。

- +：对两个变量做加法。
- -：对两个变量做减法。
- *：对两个变量做乘法。
- /：对两个变量做除法。
- **：对两个变量做幂运算。
- %：取模运算，第一个变量除以第二个变量求余数。
- +=：加等于，在自身基础上加第二个变量。
- -=：减等于，在第一个变量的基础上减去第二个变量。
- *=：乘等于，在第一个变量的基础上乘以第二个变量。

- /=：除等于，在第一个变量的基础上除以第二个变量。
- %=：取模赋值，第一个变量对第二个变量取模运算，再赋值给第一个变量。

在使用这些运算符时，需要注意到运算顺序的问题。例如输入下面的命令，输出 1+2 的结果。

```
echo 1+2
```

shell 并没有输出结果 3，而是输出了 1+2。在 shell 中有 3 种方法可以更改运算顺序。

- 用 expr 改变运算顺序。可以用 echo `expr 1 + 2`来输出 1+2 的结果，用 expr 表示后面的表达式为一个数学运算。需要注意的是，`并不是一个单引号，而是“Tab”键上面的那个符号。
- 用 let 指示数学运算。可以先将运算的结果赋值给变量 b，运算命令是 let b=1+2。然后用 echo $b 来输出 b 的值。如果没有 let，则会输出 1+2。

注意：在“let b=1+2”中，加号的前后不能有空格。

- 用$[]表示数学运算。将一个数学运算写到$[]符号的中括号中，中括号中的内容将先进行数学运算。例如命令 echo $[1+2]，将输出结果 3。

下面是一个 shell 程序实例，实现数学函数 $S=3(x^y)+4x^2+5y+6$ 的运算。在程序中以位置变量的方式输入 x 与 y 的值。程序的编写步骤如下所示。

【范例 4-3】用 shell 脚本进行数学运算实例，示例代码如下所示。

示例代码 4-3

```
01  #!/bin/bash
02  #4.3.sh
03  s=0                                  #定义一个求和变量，初值为 0。
04  t=`expr $1**$2`                      #用 expr 改变运算顺序，求 x 的 y 次方。
05  t=$[t*3]                             #t 乘以 3。
06  s=$[s+t]                             #结果相加。
07  t=$[$1**2]                           #求 x 的平方。
08  t=$[t*4]                             #结果乘以 4。
09  s=$[s+t]                             #结果相加。
10  t=`expr $2*5`                        #求 5y 的值。
11  s=$[s+t]                             #结果相加。
12  s=$[s+6]                             #结果加上 6。
13  echo $s                              #输出结果。
```

【运行结果】在终端中，输入下面的命令为 4.3.sh 文件添加可执行权限。

```
chmod +x 4.3.sh
```

输入下面的命令运行程序。在命令中需要输入两个参数。

```
./4.3.sh 2 4
```

结果如下所示。

```
90
```

【代码解析】

- 代码第 04~10 行代码首先调用 expr 指令计算表达式的值，然后使用命令替换将运算结果赋值给相应的变量。其他代码则使用方括号的方式完成表达式的计算和赋值。
- 程序实际完成 $S=3(x^y)+4x^2+5y+6$ 的数学表达式运算并输出结果。

4.6 输入和输出

输入指的是 shell 程序读入数据。有从文件读取、从用户输入读取等方式读入数据。输出指的是 shell 程序运行结果的处理，可以显示到屏幕或保存到文件。本节将讲述 shell 程序的输入和输出。

4.6.1 echo 命令输出结果

echo 命令可以输出文本或变量的值，是 shell 中最常用的输出方式。结果可以输出到终端，也可以写入到文件。该命令的用法如下所示。

```
echo $str                    #将结果输出到终端。
echo $str >file              #将结果保存到文件 file 中，如果没有文件会新建一个文件。如果已经有文件则会覆盖以前的文件。
echo $str >>file             #将结果追加到文件 file 中。
```

echo 输出的内容可以有下面的格式控制字符。

- \c：末尾加上\c 表示这一行输出完毕以后不换行。
- \t：输出一个跳格，相当于按下“Tab”键。
- \n：输出一个换行。

需要注意的是，如果要输出特殊字符，必须加-e 选项，否则输出的结果中会直接输出字符。加-n 选项可以禁止 echo 输出后的换行。例如下面是使用 echo 进行输出的例子。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令，输出一行文本。

```
echo "hello ,Beijing"
```

③ 按“Enter”键，运行结果如下所示。

```
hello ,Beijing
```

④ 在输出内容中加入一个换行，输入如下所示命令。

```
echo "hello ,\nBeijing"
```

⑤ 按“Enter”键，运行结果如下所示，换行符\n 被直接输出，并没有换行。

```
hello ,\nBeijing
```

⑥ 如果需要在输出内容中显示换行，需要在 echo 后加-e 选项。输入下面的命令。

```
echo  -e "hello ,\nBeijing"
```

⑦ 按“Enter”键，运行结果如下所示。换行符\n 处输出了一个换行。

```
hello ,
Beijing
```

⑧ 在文本中输出几个跳格，输入下面的命令。

```
echo  -e "hello ,\t\t\tBeijing"
```

⑨ 按“Enter”键，运行结果如下所示。

```
hello ,          Beijing
```

⑩ 将结果输出到文件。在终端中输入下面的命令。

```
echo "hello ,Beijing ." >a.txt
```

⑪ 按“Enter”键。在终端中没有输出显示。输入 vim a.txt 命令，用 vim 查看 a.txt 文件。

可以发现文件 a.txt 中有下面的文本。

```
hello ,Beijing .
```

⑫ 再次向这个文本中输出结果，在终端中输入如下的命令。

```
echo "abcde" >a.txt
```

⑬ 按“Enter”键。在终端中没有输出显示。输入 vim a.txt 命令查看文件 a.txt。可以发现文件中有下面的文本。第二次的输出覆盖了第一次输出时建立的文件。

```
abcde
```

⑭ 用追加的方法向这个文本中输出结果，在终端中输入如下的命令。

```
echo "hijkl" >>a.txt
```

⑮ 按“Enter”键。在终端中没有输出显示。输入 vim a.txt 命令查看文件 a.txt。文本的内容如下所示，第二次的输出追加到了第一次输出结果的末尾。

```
abcde
hijkl
```

4.6.2　read 命令读取信息

read 命令可以从键盘或文件中读入信息，并赋给一个变量。read 命令读取信息的各种方法如下所示。

- 如果只读入一个变量，会把键盘上输入的所有信息赋值给这个变量。按“Enter”键结束输入。
- 如果输入多个变量，用“Space”键将输入的变量隔开。如果输入变量的个数多于需要读取变量的个数，将会把剩余的变量赋值给最后一个变量。
- 在读取语句后面添加<filename，表示从文件中读取数据，并且赋值给变量。

例如下面的操作，就是从键盘或文本文件读取变量。

① 从主目录中打开一个终端。

② 在主目录中输入下面的命令，读取一个变量赋值给 A。

```
read A
```

③ 按“Enter”键，终端会等待用户的输入。在终端中输入下面的字符。

```
asdf
```

④ 按“Enter”键，再输入 echo $A 命令输出变量的值。显示的结果如下所示。

```
asdf
```

⑤ 读取多个字符串的变量。在终端中输入 read A 命令。然后按“Enter”键，在光标后面输入下面的字符串。

```
asd fgh jkl
```

⑥ 在终端中输入 echo $A 命令显示这个变量，按“Enter”键后显示的结果如下所示。

```
asd fgh jkl
```

⑦ 读取多个变量。在终端中输入下面的命令，然后按“Enter”键。

```
read A B C
```

⑧ 在终端中等待光标后面输入下面的字符串。

```
aaa sss ddd
```

⑨ 在终端中分别输入下面的命令输出变量的值。然后按“Enter”键。

```
echo $A
echo $B
echo $C
```

⑩ 三次命令的运行结果分别如下所示。

```
aaa
sss
ddd
```

⑪ 如果输入的数据多于需要读取的字符串，将会把多余的输入信息赋值给最后一个变量。在终端中输入下面的命令。

```
read A B
```

⑫ 按“Enter”键执行命令，在光标处输入下面的字符串。

```
aaa bbb ccc ddd
```

⑬ 输出变量。在终端中输入 echo $A 命令，然后按“Enter”键，会输出 aaa。再输入 echo $B 命令，按“Enter”键执行，终端中输出的结果如下所示。

```
bbb ccc ddd
```

⑭ 从文件中读取信息。在终端中输入 vim 命令，打开 vim。

⑮ 在 vim 中按“i”键进入插入模式。然后输入下面的字符串。

```
aaa bbb
```

⑯ 按“Esc”键进入命令模式。输入:w a.txt 命令，再按“Enter”键，保存这个文件。再输入:q 命令，退出 vim。

⑰ 在终端中输入下面的命令，从文本中读取字符串并赋值给变量。

```
read A B <a.txt
```

⑱ 输出变量。在终端中输入命令 echo $A 命令，然后按“Enter”键，终端中会显示 aaa。输入 echo $B 命令，然后按“Enter”键，终端中会显示 bbb。

4.6.3 文件重定向

文件重定向指的是，在执行命令时指定命令的输入、输出和错误的输出和输入方式。例如，可以命令的结果输出到一个文件中。表 4.2 列出了文件重定向的常见使用方法。

表 4.2 文件重定向

command > filename	把标准输出重定向到一个文件中
command >> filename	把标准输出以追加的方式重定向到一个文件中
command 1 > fielname	把标准输出重定向到一个文件中
command > filename 2>&1	把标准输出和标准错误一起重定向到一个文件中
command 2 > filename	把标准输出的错误重定向到一个文件中
command 2 >> filename	把标准输出以追加的方式重定向到一个文件中
command >> filename 2>&1	把标准输出和标准错误一起追加到一个文件中
command < filename >filename2	把 command 命令以 filename 文件作为标准输入，以 filename2 文件作为标准输出
command < filename	把 command 命令以 filename 文件作为标准输入

下面是在命令中使用文件重定向的例子。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令，查看当前的文件夹，把结果保存到文件 a.txt 中。

```
ls > a.txt
```

③ 按“Enter”键以后，命令执行没有显示结果。在终端中输入 vim a.txt 命令，可以发现文件 a.txt 中有上一命令的文件列表。

④ 将上一步骤输出的结果作为命令的输入。在终端中输入下面的命令。

```
read A <a.txt
```

⑤ 按“Enter”键执行命令。这时 read 命令会从文件 a.txt 中读取一个字符串赋值给 A。显示这个变量，输入 echo $A 命令，然后按“Enter”键。这是会显示文件 a.txt 中的第一个字符串，也就是当前用户目录下的第一个文件名，结果如下所示。

```
01.c
```

⑥ 输入和输出都是文件。例如可以用下面的命令，将文件 a.txt 中的所有小写字母转换成大写字母，然后保存到文件 b.txt 中。

```
tr "[a-z]" "[A-Z]" <a.txt >b.txt
```

⑦ 按“Enter”键执行这个命令。然后输入 vim b.txt 命令，查看文件 b.txt 中的内容，可以发现文件中所有的字母都是大写。

⑧ 在 vim 中输入:q 命令，退出 vim。

4.7　引号的使用方法

shell 程序中的单引号、双引号、反引号、反斜线在命令中有特殊含义。本节将讲述这些特殊符号的用法。

注意：反引号指的是“Tab”键上面的那个符号，在 Linux 系统中常常使用到这个符号。

4.7.1　双引号

双引号表示引用一个字符串。字符串里不能直接使用$、引号、双引号、反斜线、反引号这些特殊符号。如果字符串里没有空格，使用双引号将赋值的字符串引起来，和不使用双引号效果是一样的。当字符串中有空格时，用双引号表示引号中的内容为一个字符串。下面的操作是在字符串中使用双引号。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令，对变量 A 进行赋值。

```
A="asd fgh jkl"
```

③ 在赋值时，可以用双引号把字符串引起来。输入命令 echo $A，显示变量$A 的结果，终端的结果如下所示。

```
asd fgh jkl
```

④ 在终端中输入下面的命令，对变量 A 进行赋值。字符串中有空格，但是没有引号。

```
A=asd fgh jkl
```

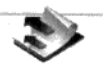
⑤ 按“Enter”键执行这个命令。终端显示命令错误，终端会认为 A=asd 是一个完整的命令，而后面的 fgh jkl 是一个不可识别的命令。

4.7.2　单引号

单引号表示引用一个字符串，用法和双引号是相似的。如果双引号中再使用单引号，则输

出时会输出这个单引号。例如输入命令 A="asd'fgh'"对变量 A 赋值。再输入命令 echo $A。命令的显示结果如下所示。

```
asd'fgh'
```



注意：单引号与双引号的区别是：双引号中出现的 shell 变量会被替换为变量的值，而单引号则不会发生变量替换。例如：echo "my home directory is $HOME"会将 $HOME 替换为当前用户的主目录的路径，而 echo 'my home directory is $HOME'打印的结果为：my home directory is $HOME。

4.7.3 反引号

反引号用于执行引号中的系统命令，然后将命令的结果返回。这个返回的结果可以赋值给一个变量。例如，下面的例子是使用反引号将命令的执行结果赋值给变量。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令。需要注意的是，命令中的是反引号而不是单引号。

```
echo `date`
```

③ 命令的运行结果如下所示。

```
一 12 月 10 15:06:08 CST 2007
```

④ 在终端中输入下面的命令。表示执行 ls 命令，再把结果赋值给变量 A。

```
A=`ls`
```

⑤ 输入 echo $A 命令，输出变量 A 的值，结果如下所示。

```
01.c 02.c 03.c 03.c~ 04.txt 1.txt 4.1.sh 4.2.sh 4.3.sh 4.4.sh
```

提示：反引号经常用于将一个命令的输出结果当做另一个命令的参数。

4.7.4 反斜线

反斜线用于对特殊字符进行转义。如果字符串中含有&、*、+、^、$、`、"、|、?这些特殊符串，shell 会认为这些符串代表着相应的运算。可以使用反斜线对这些符串进行转义。例如下面的例子是在字符串中用反斜线处理特殊符号。

① 从主菜单中打开一个终端。

② 定义含有三个引号的字符串。引号是一个特殊字符，如果直接对一个变量赋值 3 个引号，命令会发生错误。需要用反斜线对输入的引号进行转义。输入的命令如下所示。

```
A=\"\"\"
```

③ 输入命令 echo $A，再按“Enter”键执行，会输出三个引号，结果如下所示。

```
"""
```

④ 输出一个字符串$$。在命令中输入 echo $$，终端会显示当前进程的 ID。需要使用\进行转义，输入命令 echo \$$。

4.8 测试语句

这里所说的测试是指对变量的大小、字符串、文件属性等内容进行判断。test 命令可以用

于字符串、数字、文件状态等内容的测试。本节将讲述 shell 的测试语句。

提示：test 命令经常用于在 shell 脚本中与 if 语句连用进行条件判断。

4.8.1　文件状态测试

文件状态测试指的是对文件的权限、有无、属性、类型等内容进行判断。与其他语言不同的是，test 命令的测试结果，返回 0 时表示测试成功。返回 1 时表示测试失败。表 4.3 是文件测试的参数列表。

表 4.3　文件状态测试表

-d 测试文件是否是目录文件	-s 测试文件是否非空
-f 测试文件是否是正规文件	-w 测试是否可写
-L 测试文件是否是符号链接	-u 测试文件是否有 suid 位设置
-x 测试文件是否可执行	-r 测试文件是否可读

例如下面的例子使用这些测试参数，对文件的属性进行测试。

① 从主菜单中打开一个终端。

② 测试文件/windows 是否是一个目录。在终端中输入下面的命令，然后按“Enter”键。

```
test -d /windows
```

③ 输出测试结果。$?用于保存上一个命令的结果，可以用下面的命令进行输出。

```
echo $?
```

④ 终端中输出结果 1，表明/windows 不是一个目录。

⑤ 测试当前目录下的文件 a.txt 是否可以执行。从表 4.3 中可知，测试是否可以执行的参数是-x，所以输入的命令如下所示。

```
test -x a.txt
```

⑥ 按“Enter”键执行。输入 echo $?命令输出结果。

⑦ 终端中显示的结果是 1。表明文件 a.txt 是不能执行的。

提示：特殊变量$?表示上一个命令的返回值，在 shell 脚本中，进程使用$?判断上一个命令执行是否成功。如果返回值为 0，则表明上一个命令执行成功，否则执行失败。

4.8.2　数值测试

数值测试指的是比较两个数值的大小或相等关系，相当于 C 语言中的比较运算符。shell 程序中的数值测试有下面两种形式。

- 用 test 命令。test 命令和相应的参数可以对两个数值的关系进行测试，使用方法如下所示。

```
test  第一个操作数  数值比较符 第二个操作数
```

- 用中括号代替 test 命令。这种方法和 test 命令的原理相同，使用方法如下所示。

```
[ 第一个操作数  数值比较符 第二个操作数  ]
```

注意：左方括号[后面和右方括号]前面分别有一个空格。

数值比较符相当于 C 中的数据比较符号，不同的是需要使用字符串写出。数值比较符如表 4.4 所示。

表 4.4 数值比较符

-eq：两个数是否相等	-ne：两个数是否不相等
-le：第一个数是否不大于第二个数	-ge：第一个数是否不小于第二个数
-gt：第一个数是否大于第二个数	-lt：第一个数是否小于第二个数

例如下面的例子是使用数值比较符进行数字测试。

① 从主菜单中打开一个终端。

② 测试 3 和 5 是否相等。在终端中输入下面的命令。

```
test 3 -eq 5
```

③ 按“Enter”键运行这个命令。然后输入$?命令输出测试结果。终端中显示的结果为 1，表明测试结果是 3 和 5 不相等。

④ 测试 10 是否小于 12。从表 4.4 中可知，小于测试的参数是-lt。在终端中输入下面的命令。

```
test 10 -lt 12
```

⑤ 按“Enter”键运行这个命令。然后输入$?命令输出上一次测试的结果。终端中显示的结果为 0，表明 10 小于 12 为真。

⑥ 上面的测试也可以写成下面的命令。需要注意的是，[后面一定要有个空格。

```
[ 10 -lt 12 ]
```

4.8.3 字符串测试

所谓字符串测试，指的是比较两个字符串是否相等，或者判断一个字符串是否为空。这种判断常用来测试用户输入是否符合程序的要求。字符串测试有下面 4 种常用的方法。

```
test 字符串比较符 字符串
test  字符串 1  字符串比较符  字符串 2
[ 字符串比较符 字符串 ]
[ 字符串 1  字符串比较符  字符串 2 ]
```

字符串比较符有如下所示的 4 种。

- =：测试两个字符串是否相等；
- !=：测试两个字符串是否不相等；
- -z：测试字符串是否是空字符串；
- -n：测试字符串是非否是空字符串。

下面的例子是进行字符串测试。

① 从主菜单中打开一个终端。

② 用 read 命令读入两个变量，在终端中输入下面的命令。

```
read A B
```

③ 按“Enter”键运行命令，在终端中输入下面的字符串。

```
aaa bbb
```

④ 测试变量 A 与变量 B 是否相等。在终端中输入下面的命令。

```
test $A = $B
```

⑤ 按“Enter”键运行这个命令。然后输入 echo $?命令输出结果。显示的结果为 1，表明

两个字符串不相等。

4.8.4　逻辑测试

逻辑测试指的是将多个条件进行逻辑运算，常用做循环语句或判断语句的条件。shell 程序中有下面三种逻辑测试。

- -a：逻辑与，操作符两边均为真时结果为真，否则为结果为假。
- -o：逻辑或，操作符两边至少一个为真时，结果为真，否则为假。
- ！：逻辑否，只有条件为假时，返回结果为真。

例如下面的例子，可以判断主目录中的文件 a.txt，是否同时可写并且可执行。

① 从主菜单中打开一个终端。

② 在终端中输入下面的命令，用-a 表示需要同时满足两个测试条件。

```
[ -w a.txt -a -x.txt ]
```

③ 按“Enter”键执行这个命令。然后输入 echo $?命令输出测试结果，终端中显示 1。表明了测试结果为假，因为文件 a.txt 是不可写的。

4.9　流程控制结构

所谓流程控制，指的是使用逻辑判断，针对判断的结果执行不同语句或不同的程序部分。这种结构是所有编程语言的重要组成部分。本节将讲述判断、循环等流程控制结构。

注意：shell 中的流程控制结构，实际上是使用 shell 的内部命令来实现的。使用方法与其他编程语言中的流程控制语句类似。

4.9.1　if 语句

if 语句是最常用的条件判断语句，是通过一个条件的真假来决定后面的语句是不是执行。最简单的 if 条件语句如下所示。

```
if 条件
  then 命令 1
fi
```

在这种结构中，先执行条件判断，如果条件结果为真，则执行 then 后面的语句，一直到 fi。如果条件为假，则跳过后面的语句，执行 fi 后面的语句。

如果条件判断的结果，只可能是真或假两种值，可以使用下面的结构。

```
if 条件
then 命令 1
else 命令 2
f i
```

在这种结构中，先对条件进行判断，如果判断结果为真，则执行 then 后面的语句。如果判断结果为假，则执行 else 后面的语句。如果判断的结果有多种可能，则使用下面的 if 语句嵌套结构。需要注意的是，if 结构必须要有 fi 进行结束。

```
if 条件 1
then 命令 1
elif 条件 2
then 命令 2
```

```
else 命令3
f i
```

if语句也可以将 then 写在 if 条件之后，中间用分号隔开。这种语句的形式如下所示。

```
if 条件1;then 命令1
elif 条件2 ;then 命令2
else 命令3
fi
```

提示：if 语句的结尾时 fi，即 if 的反序。

例如下面的例子，是编写一个 shell 程序。程序从参数中读取一个数字，然后判断这个数字是奇数还是偶数。

【范例 4-4】shell 脚本中使用 if 语句进行条件判断，示例代码如下所示。

示例代码 4-4

```
01  #!/bin/bash
02  #4.4.sh
03  i=$[  $1 % 2 ]
04  if  test  $i -eq 0 ; then
05    echo oushu
06  else
07    echo jishu
08  fi
```

【运行结果】在终端中输入下面的命令，对这个文件添加可执行权限：

```
chmod +x 4.4.sh
```

在终端中输入如下所示的命令运行这个程序，在命令中加入一个参数。

```
./4.4.sh 4
```

程序的执行结果如下所示。

```
0
oushu
```

【代码解析】

- 代码第 03 行完成对命令行中输入的参数与 2 取模运算。
- 代码第 04~08 行判断变量$i 的值，并打印第一个命令行参数是奇数还是偶数。

4.9.3 for 语句

for 语句是一种常用的循环语句，实现在一个值列表中的循环功能。下面是 for 语句的使用方法。

```
for 变量名i n列表
do
命令1
命令2…
done
```

例如下面是一个简单的 for 循环程序，作用是用循环的方法输出列表中的数值。

【范例 4-5】使用 for 循环语句输出字符列表中的字符，示例代码如下所示。

示例代码 4-5

```
01  #!/bin/bash
02  #4.7.sh for
```

```
03   for char in a s d f g                    #开始 for 循环。
04   do                                       #循环体。
05      echo $char
06   done                                     #结束 for 循环,
```

【运行结果】在终端中输入下面的命令，运行这个程序。

```
./4.7.sh
```

程序的运行结果如下所示。

```
a
s
d
f
g
```

【代码解析】

- 代码第 03 行中的“a s d f”表示在进行循环时变量 i 依次取值。
- 被“do”和当“done”包围的命令是 for 语句的循环体。

注意：当 for 语句省略后边的 in 关键字时，将接受输入命令时的参数作为循环变量集。

例如下面的一个 for 循环程序，可以输出程序中所有的参数。

【范例 4-6】使用 for 循环语句输出所有的命令行参数，示例代码如下所示。

示例代码 4-6

```
01   #!/bin/bash
02   #4.8.sh for
03   for  str                                 #开始 for 循环。
04   do                                       #循环体。
05      echo $str
06   done                                     #结束 for 循环。
```

【运行结果】在终端中输入下面的命令运行这个程序。

```
./4.8.sh a s d f
```

程序会依次列出所输入的参数，运行结果如下所示。

```
a
s
d
f
```

【代码解析】代码第 03 行代码省略了 in，则 for 循环的变量值来自命令行的输入。

4.9.5　until 语句

until 循环用于执行一个循环体，直至条件为真时停止。这种语言结构如下所示。

```
until 条件
do
命令 1
…
done
```

例如，下面的实例是用一个 until 循环求出 1 至 100 之间所有整数的和，然后输出结果。在程序中，需要有一个变量用来存放求和的结果，一个变量用于循环的计数。

【**范例 4-7**】使用 until 循环求 1 到 100 相加的和，示例代码如下所示。

示例代码 4-7

```
01  #!/bin/bash
02  #4.11.sh
03  sum=0
04  i=1
05  until [ $i -gt 100  ]
06  do
07      sum=$[$sum+$i]
08      i=$[$i+1]
09  done
10  echo $sum
```

【运行结果】对这个文件添加可执行权限，在终端中输入下面的命令。

```
chmod +x 4.11.sh
```

输入下面的命令运行这个程序。

```
./4.11.sh
```

程序求出了 1 到 100 之间的整数和，结果如下所示。

```
5050
```

【代码解析】

- 代码第 03 行和代码第 04 行分别定义了两个 shell 变量，“sum”用于保存计算结果，而变量“i”则为循环标志变量。
- 代码第 07 行和代码第 08 行分别实现求和与循环标志变量的自动操作。

4.10 综合练习

shell 程序以简单的代码调用各种系统命令实现强大的功能，在系统设置、服务管理方面有着重要的应用。本节讲述一个完整的 shell 程序实例，自动完成计算机上 U 盘的挂载与文件复制的功能。

4.10.1 程序的功能

在 Linux 系统中，如果计算机上插入一个 USB 设置，需要用挂载命令才能实现这个设备的加载。可以把 USB 的挂载与文件的复制写成一个 shell 程序，这样可以通过程序的运算自动完成很多步骤的操作，这个程序的功能如下所示。

（1）运算程序时，提示用户输入“y”或“n”，确定是不是挂载 USB 设备。

（2）如果用户输入“Y”，则挂载这个 USB 设备。

（3）提示用户输入“y”或“n”，确定是不是复制文本。

（4）如果用户输入“Y”，则显示文件的列表，然后提示用户是否复制文件。

（5）程序根据用户输入的文件名复制相应的文件，然后提示是否将计算机中的文件复制到 USB 中。

（6）完成文件的复制以后，提示用户是否卸载 USB 设备。

4.10.2 编写程序的代码

本节根据上一节的功能分析，编写出这个程序的代码。在编程时，需要注意判断语句、循环语句的使用，对用户的输入做出正确的判断。

【范例 4-8】使用 shell 脚本实现 USB 设备加载与文件复制，示例代码如下所示。

示例代码 4-8

```
#!/bin/bash
#autousb

echo  "welcome to use AUTOUSB"
echo  "do you want load usb(y/n)?"
read ANSWER

if [ $ANSWER = "Y" -o $ANSWER = "y" ]
  then mount -t vfat /dev/sda1 /mnt/usb
  echo  "do you want copy files to /root(y/n)?"
  read ANSWER
  while [ $ANSWER = "y" -o $ANSWER = "Y" ]
  do
    ls -a /mnt/usb
    echo "type the filename you want to copy"
    read FILE
    cp /mnt/usb/"$FILE" /root
      if [ $? -qe 0 ];then
        echob " copy finished"
      else
        echob " copy errored "
      fi
    echo "any other files(y/n)?"
    read ANSWER
  done
fi

echo   "do you want to copy files to usb(y/n)?"
read ANSWER
while [ $ANSWER = "y" -o $ANSWER = "Y" ]
do
  ls -a /root
  echo "type the filename you want to copy to usb"
  read FILE
  cp "/root/$FILE" /mnt/usb
    if [ $? -qe 0 ];then
       echob " copy file finished"
      else
       echob " copy file errored "
    fi
  echo "any other files(y/n)?"
  read ANSWER
done

echo "do you want to umount usb(y/n)?"
read ANSWER
if [ $ANSWER = " y" -o  $ANSWER = " Y" ] ;then
  umount /mnt/usb
  else
echo "Haven't umount!"
fi
echo "GoodBye!"
```

【运行结果】对这个文件添加可执行权限，在终端中输入下面的命令。

```
chmod +x 4.12.sh
```

输入下面的命令运行这个程序。

```
./4.12.sh
```

4.11 小结

本节讲述了 shell 程序的基本语法。通过本章的学习，可以编写出简单的 shell 程序完成各种系统功能。在 shell 程序的学习中，需要注意局部变量和环境变量和使用、各种符号的使用方法。测试语句与流程控制语句是本章的难点，需要对程序的各种变量进行逻辑判断，执行不同的程序。

4.12 习题

一、填空题

1．Linux 的命令可以分为_____和_____。_____在 shell 启动时就调入内存，是常驻内存的。而_____是系统的软件功能，用户需要时才从硬盘中读入内存。

2．使用_____命令可以查看系统的环境变量列表。打开一个终端，在一个终端中输入命令_____，然后按“Enter”键。

3．_____命令可以从键盘或文件中读入信息，并赋给一个变量。

4．_____指的是，在执行命令时指定命令的输入、输出和错误的输出和输入方式。

5．文件状态测试指的是对文件的权限、有无、属性、类型等内容进行判断。与其他语言不同的是，test 命令的测试结果，返回_____时表示测试成功。返回_____时表示测试失败。

二、选择题

1．test 指令支持的测试包括（　　）。

A．文件状态测试，数值测试，字符串测试，逻辑测试

B．文件状态测试，数值测试，流程测试，逻辑测试

C．输入测试，数值测试，字符串测试，逻辑测试

D．文件状态测试，数值测试，字符串测试，输出测试

2．关于 shell 中的算术运算符中说法错误的是（　　）。

A．“+”：对两个变量做加法　　B．“-”：对两个变量做减法

C．“x”：对两个变量做乘法　　D．“/”：对两个变量做除法

3．shell 编程中单引号与双引号的说法中正确的是（　　）。

A．单引号和双引号没有任何区别

B．单引号中的变量名不会被转换为变量的值。双引号中的变量名会被转换为变量的值

C．双引号中的变量名不会被转换为变量的值。单引号中的变量名会被转换为变量的值

D．单引号中的变量名不会被转换为变量的值。双引号中的变量名也不会被转换为变量的值

三、简答题

1．简述 Linux 下的 shell 编程中使用 test 指令进行条件测试的详细用法。

2．简述 Linux 下的 shell 编程中的流程控制方法。

四、编程题

1．编写 shell 脚本程序实现求 1~100 相加的和。

2．编写 shell 脚本程序，提示用户输入两个数字，在程序中比较这两个数字的大小，并且输出较大者。

第三篇　C 程序基础篇

第 5 章　编译与调试

编译和调试是软件开发人员最频繁的工作，本章介绍 Linux 下 C 程序开发时所要使用的编译器 gcc 和调试器 gdb 的使用情况。本章内容是进行后续学习 Linux 下 C 程序开发的基础，本章的主要内容如下：

- 编译的概念和理解；
- gcc 编译器的使用；
- 编译过程的控制；
- 使用 gdb 调试程序。

5.1　编译的概念和理解

在进行 C 程序开发时，编译就是将编写的 C 语言代码变成可执行程序的过程。这一过程是由编译器来完成的。编译器就是完成程序编译工作的软件。编译器进行程序编译时完成了一系列复杂的过程。

5.1.1　程序编译的过程

在进行编译操作时，程序完成了复杂的过程。一个程序的编译，需要完成词法分析、语法分析、中间代码生成、代码优化、目标代码生成。本章将讲解这些步骤的作用与原理。

（1）词法分析，指的是对由字符组成的单词进行处理，从左至右逐个字符地对源程序进行扫描，产生一个个的单词符号。然后把字符串的源程序改造成为单词符号串的中间程序。在编译程序时，这一过程是自动完成的。编译程序会对代码的每一个单词进行检查。如果单词发生错误，编译过程就会停止并显示错误。这时需要对程序中的错误进行修改。

（2）语法分析。语法分析器以单词符号作为输入，分析单词符号串是否形成符合语法规则的语句。例如，需要检查表达式、赋值、循环等结构是否完整和符合使用规则。在语法分析时，会分析出程序中错误的语句，并显示出结果。如果语法发生错误，编译任务是不能完成的。

（3）中间代码生成。中间代码是源程序的一种内部表示，或称中间语言。程序进行词法分析和语法分析以后，将程序转换成中间代码。这一转换的作用是使应用程序的结构更加简单和规范。中间代码生成操作是一个中间过程，与用户是无关的。

（4）代码优化。代码优化是指对程序进行多种等价变换，使得从变换后的程序能生成更有效的目标代码。用户可以在编译程序时设置代码优化的参数，可以针对不同的环境和设置进行优化。

（5）目标代码生成。目录代码生成指的是产生可以执行的应用程序，这是编译的最后一个步骤。生成的程序是二进制的机器语言，用户只能运行这个程序，而不能打开这个文件查看程序的代码。

5.1.2　编译器

所谓编译器，是将编写出的程序代码转换成计算机可以运行的程序的软件。在进行 C 程序

开发时，编写出的代码是源程序代码，是不能直接运行的。需要用编译器编译后可以运行的二进制程序。

在不同的操作系统下面有不同的编译器。C 程序是可以跨平台平行的。但并不是说 Windows 系统下 C 语言编写的程序可以直接在 Linux 下面运行。Windows 下面 C 语言编写的程序，被编译成 EXE 文件。这样的程序只能在 Windows 系统下运行。如果需要在 Linux 系统下运行，需要将这个程序的源代码在 Linux 系统下面重新编译。不同的操作系统下面有不同的编译器。Linux 系统下面编译生成的程序是不能在 Windows 系统上运行的。

提示：Linux 和 windows 下的二进制程序是不兼容的，所以要想在 Linux 和 Windows 之间移植 C 程序，必须进行重新编译。

5.2 gcc 编译器

gcc 是 Linux 下通用的 C 程序编译器，具有非常强大的程序编译功能。在 Linux 系统下，C 语言编写的程序代码一般需要通过 gcc 来编译成可执行程序。

5.2.1 gcc 编译器简介

Linux 系统下的 gcc 编译器（GNU C Compiler）是一个功能强大、性能优越的编译器。gcc 支持多种平台的编译，是 Linux 系统自由软件的代表作品。gcc 本来只是 C 编译器，但是后来发展为可在多种硬件平台上编译出可执行程序的超级编译器。各种硬件平台对 gcc 的支持使得其执行效率与一般的编译器相比平均效率要高 20%~30%。gcc 编译器能将 C、C++源程序、汇程语言和目标程序进行编译连接成可执行文件。通过支持 make 工具，gcc 可以实施项目管理和批量编译。

经过了多年的发展，gcc 已经发生了很大的变化。gcc 已经不仅仅能支持 C 语言，还支持 Ada 语言、C++语言、Java 语言、Objective C 语言、Pascal 语言、COBOL 语言等更多的语言集的编译。gcc 几乎支持所有的硬件平台，使得 gcc 对于特定的平台可以编译出更高效的机器码。

gcc 在编译一个程序时，一般需要完成预处理（preprocessing）、编译（compilation）、汇编（assembly）和连接（linking）过程。通常使用 gcc 编译 C 程序时，这些过程是使用默认的设置自动完成的，但是用户可以对这些过程进行设置，控制这些操作的详细过程。

提示：默认情况下，gcc 自动完成预处理、编译、汇编和连接等操作。使用适当的编译选项可以分步进行编译操作。

5.2.2 gcc 对源程序扩展名的支持

扩展名指的是文件名中最后一个点以后的部分。例如下面是一个 C 程序源文件的扩展名。

```
5.1.c
```

那么这个文件的文件名是“5.1.c”，扩展名是“.c”。通常来说，源文件的扩展名标识源文件所使用的编程语言。例如 C 程序源文件的扩展名一般是“.c”。对编译器来说，扩展名控制着默认语言的设定。在默认情况下，gcc 通过文件扩展名来区分源文件的语言类型。然后根据这种语言类型进行不同的编译。gcc 对源文件的扩展名约定如下所示。

- .c 为扩展名的文件，为 C 语言源代码文件。

- .a 为扩展名的文件，是由目标文件构成的库文件。
- .C、.cc 或.cpp 为扩展名的文件，标识为 C++源代码文件。
- .h 为扩展名的文件，说明文件是程序所包含的头文件。
- .i 为扩展名的文件，标识文件是已经预处理过的 C 源代码文件，一般为中间代码文件。
- .ii 为扩展名的文件，是已经预处理过的 C++源代码文件，同上也是中间文件。
- .o 为扩展名的文件，是编译后的目标文件，源文件生成的中间目标文件。
- .s 为扩展名的文件，是汇编语言源代码文件。
- .S 为扩展名的文件，是经过预编译的汇编语言源代码文件。
- .O 为扩展名的文件，是编译以后的程序目标文件（Object file），目标文件经过连接成可执行文件。

此外，对于 gcc 编译器提供两种显示的编译命令，分别对应于编译 C 和 C++源程序的编译命令。

提示：严格来说，Linux 下只有文件名的概念而没有扩展名的概念。使用句点加后缀的方式以方便获得文件的类型。

5.3　C 程序的编译

本章以一个实例讲述如何用 gcc 编译 C 程序。在编译程序之前，需要用 vim 编写一个简单的 C 程序。在编译程序时，可以对 gcc 命令进行不同的设置。

5.3.1　编写第一个 C 程序

本节将编写第一个 C 程序。程序实现一句文本的输出和判断两个整数的大小关系。本书中编写程序使用的编辑器是 vim。程序编写步骤如下所示。

【范例 5-1】使用 gcc 编译第一个 C 程序。打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。在终端中输入下面的命令，在用户主目录“root”中建立一个目录。

```
mkdir c
```

在终端界面中输入“vim”命令，然后按“Enter”键，系统会启动 vim。在 vim 中按“i”键，进入插入模式。然后在 vim 中输入下面的程序代码。

示例代码 5-1

```
01  #include <stdio.h>                                    /*包含头文件*/
02
03  int max(int i,int j )                                /*定义比较两个整数大小的函数*/
04  {
05      if(i>j)                                          /*比较两个整数的大小*/
06      {
07          return(i);
08      }
09      else
10      {
11          return(j);
12      }
13  }
14
```

```
15  void main()                                    /*主函数*/
16  {
17     int i ,j,k;
18     i=3;
19     j=5;
20     printf("hello ,Linux.\n");
21     k=max(i,j);                                 /*调用函数比较两个整数的大小*/
22     printf("%d\n",k);
23  }
```

【运行结果】代码输入完以后，按“Esc”键，进入到普通模式。然后输入下面的命令，保存文件。

```
:w /root/c/a.c
```

vim 会把输入的程序保存到 c 目录下的文件 a.c 中。再输入“:q”命令，退出 vim。这时，已经完成了这个 C 程序的编写。

【代码解析】

- 第 03~13 行代码定义函数 max，用于比较两个数字的大小，并返回较大的数字。
- 第 15~23 行代码是 C 程序的主函数，调用 max 函数的功能，并打印 max 函数的返回值。

提示：从上面的操作可知，用 gcc 可以将一个 C 程序源文件编译成一个可执行程序。编译以后的程序需要添加可执行的权限才可以运行。在实际操作中，还需要对程序的编译进行各种设置。

5.3.2 用 gcc 编译程序

上面编写的 C 程序，只是一个源代码文件，还不能作为程序来执行。需要用 gcc 将这个源代码文件编译成可执行文件。编译文件的步骤如下所示。

① 打开系统的终端。单击“主菜单”|“系统工具”|“终端”命令，打开一个系统终端。这时进入的目录是用户主目录“/root”。然后输入下面的命令，进入 c 目录。

```
cd c
```

② 上一节编写的程序就存放在这个目录中。输入“ls”命令可以查看这个目录下的文件。显示的结果如下所示。

```
a.c
```

③ 输入下面的命令，将这个代码文件编译成可执行程序。

```
gcc a.c
```

④ 查看已经编译的文件。在终端中输入“ls”命令，显示的结果如下所示。

```
a.c a.out
```

⑤ 默认情况下，编译生成的二进制程序 a.out 具有可执行权限，可以直接运行。如果无法运行，请输入下面的命令对这个程序添加可执行权限。

```
chmod +x a.out
```

⑥ 输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
hello ,Linux.
5
```

5.3.3　查看 gcc 的参数

gcc 在编译程序时可以有很多可选参数。在终端中输入下面的命令，可以查看 gcc 的这些可选参数。

```
gcc --help
```

在终端中显示的 gcc 的可选参数如下所示。进行程序编译时，可以设置下面的这些参数。

```
用法：gcc [选项] 文件...
选项：
-pass-exit-codes：在某一阶段退出时返回最高的错误码
--help：显示此帮助说明
--target-help：显示目标机器特定的命令行选项
-dumpspecs：显示所有内建 spec 字符串
-dumpversion：显示编译器的版本号
-dumpmachine：显示编译器的目标处理器
-print-search-dirs：显示编译器的搜索路径
-print-libgcc-file-name：显示编译器伴随库的名称
-print-file-name=<库>：显示 <库> 的完整路径
-print-prog-name=<程序>：显示编译器组件 <程序> 的完整路径
-print-multi-directory：显示不同版本 libgcc 的根目录
-print-multi-lib：显示命令行选项和多个版本库搜索路径间的映射
-print-multi-os-directory：显示操作系统库的相对路径
-Wa,<选项>：  将逗号分隔的 <选项> 传递给汇编器
-Wp,<选项>：将逗号分隔的 <选项> 传递给预处理器
-Wl,<选项>：将逗号分隔的 <选项> 传递给链接器
-Xassembler <参数>：将 <参数> 传递给汇编器
-Xpreprocessor <参数>：将 <参数> 传递给预处理器
-Xlinker <参数>：将 <参数> 传递给链接器
-combine：将多个源文件一次性传递给汇编器
-save-temps：不删除中间文件
-pipe：使用管道代替临时文件
-time：为每个子进程计时
-specs=<文件>：  用 <文件> 的内容覆盖内建的 specs 文件
-std=<标准>：指定输入源文件遵循的标准
--sysroot=<目录>：将 <目录> 作为头文件和库文件的根目录
-B <目录>：将 <目录> 添加到编译器的搜索路径中
-b <机器>：为 gcc 指定目标机器(如果有安装)
-V <版本>：运行指定版本的 gcc(如果有安装)
-v：显示编译器调用的程序
-###：与 -v 类似，但选项被引号括住，并且不执行命令
-E：仅作预处理，不进行编译、汇编和链接
-S：编译到汇编语言，不进行汇编和链接
-c：编译、汇编到目标代码，不进行链接
-o <文件>：输出到 <文件>
-x <语言>：指定其后输入文件的语言。允许的语言包括 c、c++、assembler 等。
```

以 -g、-f、-m、-O、-W 或 --param 开头的选项将由 gcc 自动传递给其调用的不同子进程。若要向这些进程传递其他选项，必须使用 -W<字母> 选项。

提示：gcc 的选项非常多，初学者不需要知道所有的选项的用法，只需要能够进行最基本的编译即可，其他的参数可以以后慢慢掌握。

5.3.4 设置输出的文件

在默认情况下，gcc 编译出的程序为当前目录下的文件 a.out。-o 参数可以设置输出的目标文件。例如下面的命令，可以设置将代码编译成可执行程序 do。

```
gcc a.c -o do
```

也可以设置输出目录文件为不同的目录。例如下面的命令，是将目录文件设置成/tmp 目录下的文件 do。

```
gcc a.c -o /tmp/do
```

输入下面的命令，查看生成的目录文件。结果如下所示，在编译程序时生成的目录为/tmp 目录下的文件 do。

```
-rwxrwxr-x 1 root root 5109 12-28 13:33 /tmp/do
```

5.3.5 查看编译过程

参数-v 可以查看程序的编译过程和显示已经调用的库。输入下面的命令，在编译时输出编译过程。

```
gcc -v a.c
```

显示的结果如下所示。

```
使用内建 specs。
目标：i386-redhat-linux
配置为：../configure --prefix=/usr --mandir=/usr/share/man
 --infodir=/usr/share/info --enable-shared --enable-threads=posix
 --enable-checking=release --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions
--enable-languages=c,c++,objc,obj-c++,java,fortran,ada
--enable-java-awt=gtk --disable-dssi --enable-plugin
--host=i386-redhat-linux
线程模型：posix
gcc 版本 4.1.2 20070925 (Red Hat 4.1.2-33)
/usr/libexec/gcc/i386-redhat-linux/4.1.2/cc1
-quiet -v a.c -quiet -dumpbase a.c -mtune=generic -auxbase a -version
-o /tmp/cc8P7rzb.s
忽略不存在的目录“/usr/lib/gcc/i386-redhat-linux/4.1.2/../../../../i386-redhat-
linux/include”
#include "..." 搜索从这里开始：
#include <...> 搜索从这里开始：
 /usr/local/include
 /usr/lib/gcc/i386-redhat-linux/4.1.2/include
 /usr/include
搜索列表结束。
GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) (i386-redhat-linux)
由 GNU C 版本 4.1.2 20070925 (Red Hat 4.1.2-33) 编译。
GGC 准则：--param ggc-min-expand=64 --param ggc-min-heapsize=64394
Compiler executable checksum: ab322ce5b87a7c6c23d60970ec7b7b31
a.c: In function 'main':
a.c:16: 警告：‘main’ 的返回类型不是 ‘int’
 as -V -Qy -o /tmp/ccEFPrYh.o /tmp/cc8P7rzb.s
GNU assembler version 2.17.50.0.18 (i386-redhat-linux) using BFD version version
2.17.50.0.18-1 20070731
/usr/libexec/gcc/i386-redhat-linux/4.1.2/collect2
--eh-frame-hdr --build-id -m elf_i386 --hash-style=gnu -dynamic-linker
/lib/ld-linux.so.2 /usr/lib/gcc/i386-redhat-linux/4.1.2/../../../crt1.o
```

从编译过程可知，gcc 自动加载了系统的默认配置，调用系统的库函数完成了程序的编译过程。

提示：通过查看 gcc 的编译过程可以了解 gcc 的内部工作机制，加深对 gcc 编译 C 程序过程的理解。

5.3.6　设置编译的语言

gcc 可以对多种语言编写的源代码进行编译。如果源代码的文件扩展名不是默认的扩展名，gcc 就无法编译这个程序。可以用-x 选项来设置程序的语言。可用下面的步骤来练习这一操作。

① 输入下面的命令，将 C 程序文件复制一份。

```
cp a.c a.u
```

② 复制出的文件 a.u 是一个 C 程序文件，但扩展名不是默认的扩展名。这时输入下面的命令编译这个程序。

```
gcc a.u
```

③ 显示的结果如下所示，表明文件的格式不能识别。

```
a.u: file not recognized: File format not recognized
collect2: ld 返回 1
```

④ 这时，用-x 参数设置编译的语言，命令如下所示。这样就可以正常地编译文件 a.u。

```
gcc -x 'c' a.u
```

注意：这里的语言 C 需要用单引号扩起来。当编译扩展名不是.c 的 C 程序时，需要使用-x 参数。

5.3.7　设置 ANSIC 标准

ANSIC 是 American National Standards Institute（ANSI：美国标准协会） 出版的 C 语言标准。使用这种标准的 C 程序可以在各种编译器和系统下运行通过。gcc 可以编译 ANSIC 的程序，但是 gcc 中的很多标准并不被 ANSIC 所支持。在 gcc 编译程序时，可以用-ansic 来设置程序使用 ANSIC 标准。例如下面的命令，就是设置程序编译时，用 ANSIC 的标准进行编译。

```
gcc -asci a.out
5.3.8
```

5.3.8　编译 C++程序

gcc 可以编译 C++程序。编译 C 程序和 C++程序时，使用的是不同的命令。编译 C++程序时，使用的命令是 g++。该命令的使用方法与 gcc 是相似的。下面是使用 g++命令编译 C++程序的实例。下面是一个 C++程序的代码，实现与前面章节中介绍的程序同样的功能。C++程序的代码与 C 程序的代码非常相似。

【范例 5-2】使用 g++编译 C++程序，示例代码如下所示。

示例代码 5-2

```
01  #include <iostream>                          /*包含头文件*/
02  int max(int i,int j )                        /*自定义函数，比较两个整数的大小*/
03  {
```

```
04        if(i>j)                                    /*使用 if 语句判断两个变量的大小*/
05        {
06            return(i);
07        }
08        else
09        {
10            return(j);
11        }
12    }
13
14    int main()                                     /*主函数*/
15    {
16
17      int i ,j,k;
18      i=3;
19      j=5;
20      printf("hello ,Linux.\n");
21      k=max(i,j);                                  /*调用比较大小的函数 max*/
22      printf("%d\n",k);
23      return(0);
24    }
```

【运行结果】输入下面的命令，编译这个 C++程序。

```
g++ 5.2.cpp -o 5.2.out
```

输入下面的命令，运行这个程序。

```
./5.2.out
```

程序的运行结果如下所示。

```
hello ,Linux.
5
```

【代码解析】示例代码 5-2 与示例代码 5-1 的功能完全相同（包含的头文件不同），示例代码 5-2 使用了 C++的编译器进行编译。

提示：C++的语法向下兼容 C 语言的语法，故可用 C++编译器编译 C 程序（不用改动代码或者进行较小的改动即可）。

5.4 编译过程的控制

编译过程指的是 gcc 对一个程序进行编译时完成的内部处理和步骤。编译程序时会自动完成预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）和链接（Linking）四个步骤。本节将讲解如何对这四个步骤进行控制。

5.4.1 编译过程简介

gcc 把一个程序的源文件，编译成一个可执行文件，中间包括很多复杂的过程。可以用图 5-1 来表示编译中四个步骤的作用和关系。

在四个过程中，每一个操作都完成了不同的功能。编译过程的功能如下所示。

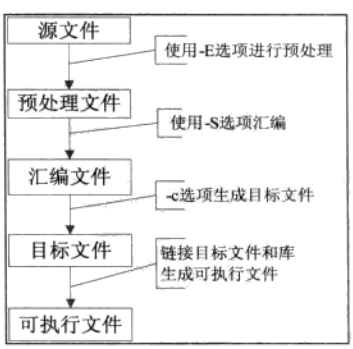


图 5-1 gcc 编译源文件到可执行文件的过程

- 预处理：在预处理阶段，主要完成对源代码中的预编译语句（如宏定义 define 等）和文件包含进行处理。需要完成的工作是对预编译指令进行替换，把包含文件放置到需要编译的文件中。完成这些工作后，会生成一个非常完整的 C 程序源文件。
- 编译：gcc 对预处理以后的文件进行编译，生成以.s 为后缀的汇编语言文件。该汇编语言文件是源代码编译得到的汇编语言代码，接下来交给汇编过程进行处理。编语言是一种比 C 语言更低级的语言，直接面对硬盘进行操作。程序需要编译成汇编指令以后再编译成机器代码。
- 汇编：汇编过程是处理汇编语言的阶段，主要调用汇编处理程序完成将汇编语言会变成二进制机器代码的过程。通常来说，汇编过程是将.s 的汇编语言代码文件汇编为.o 的目标文件的过程。所生成的目标文件作为下一步连接过程的输入文件。
- 链接：链接过程就是将多个汇编生成的目标文件及引用的库文件进行模块链接生成一个完整的可执行文件。在链接阶段，所有的目标文件被安排在可执行程序中的适当的位置。同时，该程序所调用到的库函数也从各自所在的函数库中链接到程序中。经过了这个过程以后，生成的文件就是可执行的程序。

5.4.2　控制预处理过程

参数-E 可以完成程序的预处理工作而不进行其他的编译工作。下面的命令可以将本章编写的程序进行预处理，然后保存到文件 a.cxx 中。

```
gcc -E -o a.cxx a.c
```

输入下面的命令，查看经过预处理以后的 a.cxx 文件。

```
vim a.cxx
```

可以发现。文件 a.cxx 约有 800 行代码。程序中默认包含的头文件已经被展开写到这个文件中。显示的文件 a.cxx 前几行代码如下所示。可见，在程序编译时，需要调用非常多的头文件和系统库函数。

```
# 1 "a.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "a.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 335 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 360 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 361 "/usr/include/sys/cdefs.h" 2 3 4
# 336 "/usr/include/features.h" 2 3 4
# 359 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 5 "/usr/include/gnu/stubs.h" 2 3 4
# 1 "/usr/include/gnu/stubs-32.h" 1 3 4
# 8 "/usr/include/gnu/stubs.h" 2 3 4
# 360 "/usr/include/features.h" 2 3 4
# 29 "/usr/include/stdio.h" 2 3 4
```

5.4.3　生成汇编代码

参数-S 可以控制 gcc 在编译 C 程序时只生成相应的汇编程序文件，而不继续执行后面的编译。下面的命令，可以将本章中的 C 程序编译成一个汇编程序。

```
gcc -S -o a.s a.c
```

输入下面的命令，查看汇编文件 a.s。可以发现这个文件一共有 60 行码。这些代码是这个程序的汇编指令。部分汇编程序代码如下所示。

```
01        .file "a.c"
02        .text
03   .globl max
04        .type    max, @function
05   max:
06        pushl    %ebp
07        movl %esp, %ebp
08        subl $4, %esp
09        movl 8(%ebp), %eax
10        cmpl 12(%ebp), %eax
11        jle .L2
12        movl 8(%ebp), %eax
13        movl %eax, -4(%ebp)
14        jmp .L4
15   .L2:
16        movl 12(%ebp), %eax
17        movl %eax, -4(%ebp)
18   .L4:
19        movl -4(%ebp), %eax
20        leave
21        ret
22        .size    max, .-max
23        .section .rodata
24   .LC0:
25        .string  "hello ,Linux."
26   .LC1:
27        .string  "%d\n"
28        .text
29   .globl main
30        .type    main, @function
31   main:
32        leal 4(%esp), %ecx
33        andl $-16, %esp
34   .......
35        popl %ebp
36        leal -4(%ecx), %esp
37        ret
38        .size    main, .-main
39        .ident   "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
40        .section .note.GNU-stack,"",@progbits
```

5.4.4 生成目标代码

参数-c 可以使用 gcc 在编译程序时只生成目录代码而不生成可执行程序。输入下面的命令，将本章中的程序编译成目录代码。

```
gcc -c -o a.o  a.c
```

输入下面的命令，查看这个目录代码的信息。

```
file a.o
```

显示文件 a.o 的结果如下所示，显示文件 a.o 是一个可重定位的目标代码文件。

```
a.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

5.4.5 链接生成可执行文件

gcc 可以把上一步骤生成的目录代码文件生成一个可执行文件。在终端中输入下面的命令。

```
gcc a.o -o aa.out
```

这时生成一个可执行文件 aa.out。输入下面的命令查看这个文件的信息。

```
file aa.out
```

显示的结果如下所示，表明这个文件是可在 Linux 系统下运行的程序文件。

```
aa.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.9, not stripped
```

注意：通常情况下 gcc 自动完成上述的四个编译过程，无须单独执行。当需要了解编译过程时，可以使用相应的参数进行分步骤编译。

5.5　gdb 调试程序

所谓调试，指的是对编好的程序用各种手段进行查错和排错的过程。进行这种查错处理时，并不仅仅是运行一次程序检查结果，而是对程序的运行过程、程序中的变量进行各种分析和处理。本节将讲解如何使用 gdb 进行程序的调试。

5.5.1　gdb 简介

gdb 是一个功能强大的调试工具，可以用来调试 C 程序或 C++程序。在使用这个工具进行程序调试时，主要使用 gdb 进行下面几个方面的操作。

- 启动程序：在启动程序时，可以设置程序运行环境。
- 设置断点：断点就是可以在程序设计时暂停程序运行的标记。程序会在断点处停止，便于用户查看程序的运行情况。这里的断点可以是行数、程序名称或条件表达式。
- 查看信息：在断点停止后，可以查看程序的运行信息，显示程序变量的值。
- 分步运行：可以程序程序一个语句一个语句的执行，这时可以及时地查看程序的信息。
- 改变环境：可以在程序运行时改变程序的运行环境和程序变量。

5.5.2　在程序中加入调试信息

为了使用 gdb 进行程序调试，需要在编译程序中加入供 gdb 使用的调试信息。方法是在编译程序时使用一个-g 参数。在终端中输入下面的命令，在编译程序时加入调试信息。

```
gcc -g -o a.debug a.c
```

这时，编译程序 a.c，生成一个 a.bedug 的可执行程序。这个可执行程序中加入了供调试所用的信息。

注意：使用-g 生成的目标代码比不使用此参数生成的目标代码要大，通常在开发过程中使用-g 参数，当发布程序时则不使用-g 参数。

5.5.4　启动 gdb

在调试文件以前，需要启动 gdb。在终端中输入下面的命令。

```
gdb
```

这时，gdb 的启动信息如下所示。这些提示显示了 gdb 的版本和版权信息。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
(gdb)
```

5.5.5 加载需要调试的程序

使用 gdb 调试一个程序之前，需要加载这个程序。加载程序的命令是 file。在（gdb)提示符后面输入下面的命令加载程序 a.debug。

```
file a.debug
```

命令的运行结果如下所示，显示已经加载了这个文件，并且使用了系统库文件。

```
Reading symbols from /root/c/a.debug...done.
Using host libthread_db library "/lib/libthread_db.so.1".
```

5.5.6 查看代码

用 gcc 命令编译程序加入了-g 命令以后，编译后的 a.debug 程序中加入了断点。可以用 list 命令显示程序的源代码和断点。下面的步骤是查看加入断点以后的代码。

① 在 gdb 提示符后面输入下面的命令。

```
list 1
```

② gdb 显示第一个断点以前的代码。显示的代码如下所示。

```
1       #include <stdio.h>
2
3       int max(int i,int j )
4       {
5               if(i>j)
6               {
7                       return(i);
8               }
9               else
10              {
(gdb)
```

（3）按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
11                      return(j);
12              }
13      }
14
15      void main()
16      {
17        int i ,j,k;
18        i=3;
19        j=5;
20        printf("hello ,Linux.\n");
(gdb)
```

（4）按“Enter”键，显示下一个断点以前的代码。结果如下所示。

```
21        k=max(i,j);
22        printf("%d\n",k);
23      }
(gdb)
```

5.5.7 在程序中加入断点

程序会运行到断点的位置停止下来，等待用户处理信息或查看中间变量。如果自动设置的

断点不能满足调试要求，可以用 break 命令增加程序的断点。例如需要在程序的第六行增加一个断点，可以输入下面的命令。

```
break 6
```

这时 gdb 显示的结果如下所示。

```
Breakpoint 1 at 0x8048402: file a.c, line 6.
```

输入下面的命令，在程序的第 18 行、第 19 行、第 21 行增加断点。

```
break 18
break 19
break 21
```

5.5.8　查看断点

命令“info breakpoint”可以查看程序中设置的断点。输入“info breakpoint”命令，结果如下所示。显示程序中所有的断点。

```
1   breakpoint     keep y   0x08048402 in max at a.c:6
2   breakpoint     keep y   0x08048426 in main at a.c:18
3   breakpoint     keep y   0x0804842d in main at a.c:19
4   breakpoint     keep y   0x08048440 in main at a.c:21
```

加上相应的断点编号，可以查看这一个断点的信息。例如下面的命令就是查看第二个断点。

```
info breakpoint 2
```

显示的结果如下所示。

```
2   breakpoint     keep y   0x08048426 in main at a.c:18
```

5.5.9　运行程序

gdb 中的 run 命令可以使这个程序以调试的模式运行。下面的步骤对程序进行调试。

① 在 ddb 提示符后输入“run”命令，显示的结果如下所示。

```
Starting program: /root/c/a.debug
warning: Missing the separate debug info file: /usr/lib/debug/.build-id/ac/
2eeb206486bb7315d6ac4cd64de0cb50838ff6.debug
warning: Missing the separate debug info file: /usr/lib/debug/.build-id/ba/
4ea1118691c826426e9410cafb798f25cefad5.debug
Breakpoint 2, main () at a.c:18
18          i=3;
```

② 结果显示了程序中的异常，并将异常记录到了系统文件中。然后在程序的第二个断点的位置第 18 行停下。

③ 输入“next”命令，程序会在下一行停下，结果如下所示。

```
19          j=5;
```

④ 输入“continue”命令，程序会在下一个断点的位置停下。结果如下所示。

```
Continuing.
Breakpoint 3, main () at a.c:19
21          k=max(i,j);
```

⑤ 输入 continue”命令，程序运行到结束。结果如下所示，表明程序已经运行完毕正常退出。

```
5
Program exited with code 02.
```

⑥ “step”命令与“next”命令的作用相似，对程序实现单步运行。不同之处是，在遇上

函数调用时，“step”函数可以进行到函数内部。而“next”函数只是一步完成函数的调用 。

5.5.10 变量的查看

“print”命令可以在程序的运行中查看变量的值。本节将用下面的步骤来讲解变量的查看方法。

① 输入下面的命令，运行程序。

```
run
```

② 程序在第一个断点位置停下。显示的结果如下所示。

```
Breakpoint 2, main () at a.c:18
18        i=3;
```

③ 程序进入第 18 行之前停下，并没有对 i 进行赋值。可以用下面的命令来查看 i 的值。

```
print i
```

④ 显示的结果如下所示，表示 i 现在只是一个任意值。

```
$5 = -1076190040
```

⑤ 输入下面的命令，使程序运行一步。

```
step
```

⑥ 显示的结果如下所示。

```
19        j=5;
```

⑦ 这时程序在第 19 行以前停下，这时输入下面的命令，查看 i 的值。

```
print i
```

⑧ 这时显示的 i 的结果如下所示。表明 i 已经被赋值为 3。

```
$6 = 3
```

⑨ 这时输入“step”命令，再次输入“step”命令，显示的结果如下所示。

```
21        k=max(i,j);
```

⑩ 这时输入“step”命令，会进入子函数中，结果如下所示。这时，显示了传递该函数的变量和值。

```
max (i=3, j=5) at a.c:5
5             if(i>j)
```

⑪ 这时，输入“step”命令，显示的结果如下所示，表明函数会返回变量 j。

```
11                  return(j);
```

⑫ 输入下面的命令，查看 j 的值。

```
print j
```

⑬ 显示的结果如下所示，表明 j 的值为 5。

```
$7 = 5
```

⑭ 这时再运行两次“step”命令，显示的结果如下所示。

```
22        printf("%d\n",k);
```

⑮ 这时，输入下面的命令，查看 k 的值。

```
print k
```

⑯ 显示的结果如下所示，表明 k 的值为 5。

```
$8 = 5
```

⑰ 完成了程序的调试运行以后，输入“q”命令，退出 gdb。

5.6　gdb 的帮助与常用命令

在本节中，将详细讲解 gdb 的帮助信息及常用的命令。在使用 gdb 的时候，如果用户对使用过程有疑惑，可以使用 gdb 的帮助信息。

5.6.1　使用 gdb 帮助

gdb 有很多的命令。输入“help”命令，显示这些命令的帮助信息。本节将讲解帮助信息的使用。

（1）在 gdb 输入“help”命令，显示的帮助信息如下所示。

```
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

（2）上面的帮助信息显示，输入“help all”会输出所有帮助信息。

（3）在“help”命令后面加上一个命令名称，可以显示这个命令的帮助信息。例如输入“help file”，显示的 file 命令帮助信息如下所示。

```
Use FILE as program to be debugged.
It is read for its symbols, for getting the contents of pure memory,
and it is the program executed when you use the `run' command.
If FILE cannot be found as specified, your execution directory path
($PATH) is searched for a command of that name.
No arg means to have no executable file and no symbols.
```

5.6.2　gdb 常用命令

除了前面讲述的 gdb 命令以外，gdb 还有很多种命令。这些命令可以完成程序调试的各种功能。其他的常用命令含义如下所示。

- backtrace：显示程序中的当前位置和表示如何到达当前位置的栈跟踪；
- breakpoint：在程序中设置一个断点；
- cd：改变当前工作目录；
- clear：删除刚才停止处的断点；
- commands：命中断点时，列出将要执行的命令；
- continue：从断点开始继续执行；
- delete：删除一个断点或监测点，也可与其他命令一起使用；

- display：程序停止时显示变量和表达时；
- down：下移栈帧，使得另一个函数成为当前函数；
- frame：选择下一条 continue 命令的帧；
- info：显示与该程序有关的各种信息；
- info break：显示当前断点清单，包括到达断点处的次数等；
- info files：显示被调试文件的详细信息；
- info func：显示所有的函数名称；
- info local：显示当前函数中的局部变量信息。

5.7 编译程序常见的错误与问题

在编写程序时，无论是逻辑上还是语法上，不可能一次做到完全正确。于是在编译程序时，就会发生编译错误。本节将讲述程序编译时常见的错误类型与处理方法。

5.7.1 逻辑错误与语法错误

在编程时，出现的错误可能有逻辑错误与语法错误两种。这两种错误的发生原因和处理方法是不同的。本节将讲述这两种错误的处理方法。

- 逻辑错误指的是程序的设计思路发生了错误。这种错误在程序中是致命的，程序可能正常编译通过，但是结果是错误的。当程序正常运行而结果错误时，一般都是编程的思路错误。这时，需要重新考虑程序的运算方法与数据处理流程是否正确。
- 语法错误：语法错误指的是程序的思路正确，但是在书写语句时，发生了语句错误。这种错误一般是编程时不小心或是对语句的错误理解造成的。在发生语句错误时，程序一般不能正常编译通过提示错误。这时，会提示错误的类型和错误的位置，按照这些提示改正程序的语法错误即可完成错误的修改。

5.7.2 C 程序中的错误与异常

C 程序中的错误，根据严重程序的不同，可以分为异常与错误两类。在编译程序时，这两种情况对编译的影响是不同的，对错误与异常的处理方式是不同的。

1. 什么是异常

异常指的是代码中轻微的错误，这些错误一般不会影响程序的正常运行，但是不完全符合编程的规范。在编译程序时，会产生一个“警告”，但是程序会继续编译。下面的程序会使程序发生异常，在编译时产生一个警告错误。

- 在除法中，0 作除数。
- 在开方运算时，对负数开平方。
- 程序的主函数没有声明类型。
- 程序的主函数没有返回值。
- 程序中定义了一个变量，但是没有使用这个变量。
- 变量的存储发生了溢出。

2. 什么是错误

错误指的是程序的语法出现问题，程序编译不能正常完成，产生一个错误信息。这时会显示错误的类型与位置。根据这些信息可以对程序进行修改。

5.7.3　编译中的警告提示

在编译程序时，如果发生了不严重的异常，会输出一个错告错误，然后完成程序的编译。例如下面的内容是一个程序在编译时产生的警告。

```
5.1.c: In function 'main':
5.1.c:16: 警告：‘main’ 的返回类型不是 ‘int’
5.1.c:18: 警告：被零除
```

这些的含义如下所示。

（1）“In function ‘main’:”表示发生的异常在 main 函数内。

（2）“5.1.c:16:”表示发生异常的文件是 5.1.c，位置是第 16 行。

（3）下面的信息是第 16 行的异常，表明程序的返回类型不正确。

```
‘main’ 的返回类型不是 ‘int’
```

（4）下面的警告信息表明程序的第 18 行有除数为 0 的错误。

```
5.1.c:18: 警告：被零除
```

5.7.4　找不到包含文件的错误

程序中的包含文件在系统或工程中一定要存在，否则程序编译时会发生致命错误。例如下面的语句包含了一个不正确的头文件。

```
#include <stdio1.h>
```

编译程序时，会发生错误。错误信息如下所示。

```
5.1.c:2:20: 错误：stdio2.h：没有那个文件或目录
```

5.7.5　错误地使用逗号

程序中逗号的含义是并列几个内容，形成某种算法或结构。程序中如果错误地使用逗号，会使用程序在编译时发生致命错误。例如下面的代码，是程序中的 if 语句后面有一个错误的逗号。

```
01  int max(int i,int j )
02  {
03      if(i>j),
04      {
05          return(i);
06      }
07      else
08      {
09          return(j);
10      }
11  }
```

程序编译时输出的错误信息如下所示。表明 max 函数中逗号前面的表达式有错误。实际上的错误是多一个逗号。

```
5.1.c: In function 'max':
5.1.c:4: 错误：expected expression before ‘,’ token
5.1.c: In function 'max':
```

5.7.6　括号不匹配错误

程序中的引号、单引号、小括号、中括号、大括号等符号，必须是成对出现的。这方面的错误会使程序发生符号不匹配的错误。发生这种错误后，编译程序往往不能理解代码的含义，

也不能准确显示错误的位置，而是显示表达式错误。例如下面的代码，在最后一行少了一个花括号。

```
01  int max(int i,int j )
02  {
03      if(i>j)
04      {
05          return(i);
06      }
07      else
08      {
09          return(j);
10      }
```

编译程序时，会显示下面的错误信息。

```
5.1.c:22: 错误: expected declaration or statement at end of input
```

提示：为了避免在编程时出现括号不匹配的错误，可以在写代码时先将左右括号输入，然后再输入括号内的内容。另外，注意代码的缩进也可以有效地避免括号不匹配错误或者容易发现括号不匹配的错误。

5.8 综合练习

本节讲解一个程序调试实例。先编写一个程序，在程序运行时，发现结果与预想结果不同。然后用 gdb 工具进行调试，通过单步运行和对变量的查看，查找出程序的错误。

5.8.1 编写程序

本节将编写一个程序，要求程序运行时可以显示下面的结果。

```
1+1=2
2+1=3 2+2=4
3+1=4 3+2=5 3+3=6
4+1=5 4+2=6 4+3=7 4+4=8
```

很明显，这个程序是通过两次循环与一次判断得到的。程序中需要定义三个变量。下面用这个思路来编写这个程序。

① 打开一个终端。在终端中输入“vim”命令，打开 vim。

② 在 vim 中按“i”键，进入插入模式。然后在 vim 中输入下面的代码。

```
01  #include <stdio.h>                                  /*包含头文件*/
02  main()                                              /*主函数*/
03  {
04      int i,j,k;
05      for(i=1;i<=4;i++)                               /*使用嵌套循环语句输出数据*/
06      {
07          for(j=1;j<=4;j++);                          /*内部循环语句输出数据*/
08          {
09              if(i>=j)                                /*判断两个整数的大小*/
10              {
11                  k=i+j;
12                  printf("%d+%d=%d ",i,j,k);
13              }
01          }
02          printf("\n");
03      }
```

```
04  }
```

③ 在 vim 中按“Esc”键，进入普通模式。然后输入下面的命令，保存这个文件。

```
:w /root/c/test.c
```

④ 输入“:q”命令退出 vim。很容易发现，在第二个循环后多了一个分号。

5.8.2　编译文件

本节将对上一节编写的程序进行编译和运行。在运行程序时，会发现程序有错误。

① 在终端中输入下面的命令，编译这个程序。

```
gcc /root/c/test.c
```

② 程序可以正常编译通过，输入下面的命令，运行这个程序。

```
/root/c/a.out
```

③ 程序的显示结果是四个空行，并没有按照预想的要求输出结果。

④ 输入下面的命令，对这个程序进行编译。在编译加入-g 参数，为 gdb 调试做准备。

```
gcc -g -o test.debug 6.2.c
```

⑤ 这时，程序可以正常编译通过。输出的文件是 test.debug。这个文件中加入了文件调试需要的信息。

5.8.3　调试程序

本节将讲述如何使用 gdb 对上一节编写的程序进行调试，查找出程序中的错误。

① 在终端中输入“gdb”命令，进入到 gdb，显示的结果如下所示。

```
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu".
```

② 导入文件。在 gdb 中输入下面的命令。

```
file /root/c/test.debug
```

③ 这时显示的结果如下所示。表明已经成功加载了这个文件。

```
Reading symbols from /root/c/test.debug...(no debugging symbols found)...done.
Using host libthread_db library "/lib/libthread_db.so.1".
```

④ 查看文件。在终端中输入下面的命令。

```
list
```

⑤ 显示的文件查看结果如下所示。

```
1       #include <stdio.h>
2
3       main()
4       {
5               int i,j,k;
6               for(i=1;i<=4;i++)
7               {
8                       for(j=1;j<=4;j++);
9                       {
10                              if(i>=j)
(gdb)
11                              {
```

```
12                              k=i+j;
13                              printf("%d+%d=%d ",i,j,k);
14                          }
15                      }
16                      printf("\n");
17              }
18      }
(gdb)
Line number 19 out of range; 6.2.c has 18 lines.
```

⑥ 在程序中加入断点。从显示的代码可知，需要在第 6 行、第 11 行、第 12 行、第 13 行加入断点。在 gdb 中输入下面的命令。

```
break 6
break 11
break 12
break 13
```

⑦ gdb 显示的添加断点的结果如下所示。

```
Breakpoint 1 at 0x8048405: file 6.2.c, line 6.
Breakpoint 2 at 0x8048429: file 6.2.c, line 11.
Breakpoint 3 at 0x8048429: file 6.2.c, line 12.
Breakpoint 4 at 0x8048432: file 6.2.c, line 13.
```

⑧ 输入下面的命令，运行这个程序。

```
run
```

⑨ 运行到第一个断点显示的结果如下所示。

```
Breakpoint 1, main () at 6.2.c:6
6           for(i=1;i<=4;i++)
```

⑩ 输入“step”命令，程序程序运行一步，结果如下所示。

```
8                   for(j=1;j<=4;j++);
```

⑪ 这说明程序已经进入了 for 循环。这时输入下面命令，查看 i 的值。

```
print i
```

⑫ 显示的结果如下所示。

```
$2 = 1
```

⑬ 这时再输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑭ 这时再输入“step”命令，显示的结果如下所示。

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

⑮ 这表明，在进行 j 的 for 循环时，没有反复执行循环体。这时再输入“step”命令，显示的结果如下所示。

```
 for(i=1;i<=4;i++)
```

⑯ 这表明，程序正常地进行了 i 的 for 循环。这是第二次执行 for 循环。

⑰ 输入“step”命令，显示的结果如下所示。

```
8                 for(j=1;j<=4;j++);
```

⑱ 这表明，程序执行到 for 循环。这时再次输入“step”命令，显示的结果如下所示。

```
10                          if(i>=j)
```

⑲ 输入“step”命令，显示的结果如下所示。

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

㉑ 输入“step”命令，显示的结果如下所示。

```
6               for(i=1;i<=4;i++)
```

㉒ 这说明，程序正常地进行了 i 的 for 循环，但是没有执行 j 的 for 循环。这一定是 j 的 for 循环语句有问题。这时就不难发现 j 的 for 循环后面多了一个分号。

㉓ 输入“q”命令，退出 gdb。

5.9　小结

程序的编译和调试是编程的一个重要环节。本节讲解了 Linux 系统中 C 编程的编译器 gcc 和编译器 gdb 的使用。使用 gcc 时，需要对编译进行各种设置，需要理解 gcc 各项参数的作用。gdb 的学习重点是对 gdb 单步运行程序的理解，通过程序的单步运行发现程序中的问题。

5.10　习题

一、填空题

1．_____就是完成程序编译工作的软件。

2．_____是 Linux 下通用的 C 程序编译器，具有非常强大的程序编译功能。

3．所谓_____，指的是对编好的程序用各种手段进行查错和排错的过程。进行这种查错处理时，并不仅仅是运行一次程序检查结果，而是对程序的运行过程、程序中的变量进行各种分析和处理。

4．_____是一个功能强在的调试工具，可以用来调试 C 程序或 C++程序。它有非常多的命令。输入_____命令可以显示这些命令的帮助信息。

5．在进行编译操作时，程序完成了复杂的过程。一个程序的编译，需要完成_____、_____、_____、_____、_____。

6．编译过程指的是 gcc 对一个程序进行编译时完成的内部处理和步骤。编译程序时会自动完成_____ 、_____、_____和_____四个步骤。

7．_____指的是代码中轻微的错误，这些错误一般不会影响程序的正常运行，但是不完全符合编程的规范。在编译程序时，会产生一个“警告”，但是程序会继续编译。_____指的是程序的语法出现问题，程序编译不能正常完成，产生一个错误信息。

二、选择题

1．关于 gcc 说法错误的是（　　）。

A．gcc 是一个 C 语言编译器

B．gcc 是一款 C 语言编辑器

C．gcc 编译时自动完成编译的四个步骤

D．gcc 的全程为“gnu C Compiler”

2．关于 gdb 说法正确的是（　　）。

A．gdb 是 gnu 组织的 C 程序调试器

B．使用 gdb 调试 C 程序时，无须对编译做特殊处理

C．gdb 调试 C 程序时支持加入断点和查看代码功能

D．使用 gdb 调试 C 程序时可以查看 C 程序中变量的值

3．下列说法中错误的是（　　）。

A．逻辑错误指的是程序的设计思路发生了错误。这种错误在程序中是致命的，程序可能正常编译通过，但是结果是错误的。当程序正常运行而结果错误时，一般都是编程的思路错误。这时，需要重新考虑程序的运算方法与数据处理流程是否正确

B．语法错误：语法错误指的是程序的思路正确，但是在书写语句时，发生了语句错误。这种错误一般是编程时不小心或是对语句的错误理解造成的。在发生语句错误时，程序一般不能正常编译通过提示错误。这时，会提示错误的类型和错误的位置，按照这些提示改正程序的语法错误即可完成错误的修改

C．错误指的是代码中轻微的错误，这些错误一般不会影响程序的正常运行，但是不完全符合编程的规范。在编译程序时，会产生一个“警告”，但是程序会继续编译

D．错误指的是程序的语法出现问题，程序编译不能正常完成，产生一个错误信息。这时会显示错误的类型与位置。根据这些信息可以对程序进行修改

三、简答题

1．简述使用 gcc 编译 C 程序的整个过程，以及常用的编译选项所代表的含义。

2．简述使用 gdb 调试 C 程序的整个过程。

四、编程题

编写一个简单的 C 程序，并用 gdb 调试运行此程序。

第 6 章　C 语言基础

C 语言有着严格的格式和语法规则，用户必须按照这些要求来编写程序。本章将讲解 C 程序的组成、语句、注释等基本概念，本章的主要内容如下：

- C 程序的基本概念；
- C 语言中的数据类型；
- 变量的赋值与输出；
- C 语言中的运算符。

6.1　C 程序的基本概念

本节通过最简单的 C 程序来介绍 C 程序的基本结构、一般格式和注释的用法。希望用户通过本节的内容，了解 Linux 下 C 语言编程的主要内容。

6.1.1　C 程序的基本结构

C 程序由语句、函数、包含文件等部分组成。本节将以一个简单的 C 程序实例来讲解 C 程序的基本结构问题。这个程序实现两个整数的大小判断功能。

【范例 6-1】编写比较两个整数大小的简单 C 程序，示例代码如下。

示例代码 6-1

```
01  #include <stdio.h>          /*包含文件。*/
02  int max(int i , int j)
03  {
04                              /*注释：这是一个自定义函数，实现两个整数的大小比较功能。 */
05  if(i>=j)
06    return(i);                /*返回较大的数。*/
07  else
08    return(j);
09  }
10  int main()                  /*主函数。*/
11  {
12    int i =3;                 /*定义三个变量，并且赋值。*/
13    int j =5;
14    int t;
15    t=max(3,5);               /*将较大的数赋值给 t。*/
16    printf("the big number is %d\n",t);   /*输出文本和结果。*/
17  }
```

【运行结果】在终端中输入下面的命令，编译这个程序。

```
gcc -o 6.1.out 6.1.c
```

输入下面的命令，运行这个程序。

```
./6.1.out
```

程序的运行结果如下所示，显示较大值是 5。

```
the big number is 5。
```

【代码解析】上面的代码演示了 C 程序设计的基本结构，包括函数的定义调用，包含头文件、注释，以及变量的定义与使用等内容。

一个较完整的程序大致包括下面这些内容。

- 包含文件：一组#include <*.h>语句。
- 用户自定义函数：用户已编写的完成特定功能的模块。
- 主函数：程序自动执行的程序体。
- 变量定义：定义变量以存储程序中的数据。
- 数据运算：程序通过运算完成各种逻辑功能。这些运算是由各种语句和函数实现的。
- 注释：注释写在/* */符号之中，不是程序的必需部分，但是可以增强程序的可读性。

需要注意的是，C 程序是严格区分大小写的，程序中不同的大写小字母代表不同的内容。可以将多个语句写在一行上，但是每个语句必须有一个分号结束。例如本节代码中的 max 函数，也可以写成下面的形式。

```
01   int max(int i , int j){  if(i>=j)  return(i);    else  return(j);}
```

这段程序也是可以正常执行的，但是不便于程序的阅读与修改。在编程中需要使用正确的格式，每个语句写在一行上，并且正确的注释与缩进，使代码的层次清晰明了。

提示：编程中的代码缩进可以提高代码的可读性，初学者应该养成代码缩进的习惯。

6.1.2 C 程序的一般格式

通过上一节的例子，可知程序的主要部分是主函数和自定义函数。C 程序的一般结构如下所示。

```
程序的包含文件
子函数类型声明
全局变量定义
main 主函数()
{
   局部变量定义
   <程序体>
}
自定义函数 1()
{
   局部变量定义
   <程序体>
}
自定义函数 2()
{
   局部变量定义
   <程序体>
}
……
自定义函数 N()
{
   局部变量定义
   <程序体>
}
```

提示：主函数与自定义函数的位置是随意的。自定义函数可以写在主函数的前面或后面，如果放在后面，需要在程序的前面声明这个函数。

6.1.3　C 程序中的注释

所谓注释，就是在编写程序时对代码的补充说明，不影响程序。任何一个程序员，都不可能完全记忆所写过的代码，注释的作用是对代码的阅读和理解进行提示。程序在编译时，会自动去除注释的内容。在编写程序时，需要正确地书写注释，养成良好的注释习惯。例如下面的代码，就是常用的注释方法。

```
01  /*   作者：小明。
02       内容：变量的输入与输出。
03       时间：2007.11.18
04       程序的最前面注释出代码的信息。*/
05  #include <stdio.h>                    /*包含文件。*/
06  int main()                            /*主函数。可以按“Tab”键使注释对齐。*/
07  {
08  int i ;                               /*定义一个变量。*/
09  scanf("%d",&i);                       /*输入一个变量。*/
10  printf("%d",i)                        /*输出变量。*/
11  }
```

提示：程序中的注释不需要长篇大论，只需要精简地说明程序的功能和设计思想即可。

6.2　数据类型

数据类型指的是一类数据的集合，是对数据的抽象描述。数据类型的不同决定所占存储空间的大小不同。每个变量在使用之前必须定义其数据类型。C 程序有整型（int）、浮点型（float）、字符型（char）、指针型（*）、无值型（void）这些常用数据类型。还有结构体（struct）和联合体（union）两种自定义数据类型。本章将讲解前三种基本的数据类型。

提示：C 语言是一种强类型语言，在使用变量前必须定义变量和指明变量的数据类型。

6.2.1　整型（int）

整型可以简单理解为整数。与整数不同的是，一个整型变量有一定的字节长度和存储数据范围。整型变量是有正负的，在定义整型变量时，需要注意变量的正负问题。如表 6.1 所示是不同长度与正负的整型变量。

表 6.1　整型变量

类　型	说　明
signed short int	有符号短整型数。简写为 short 或 int，字长为 2 字节。范围是-32768~32767
signed long int	有符号长整型数。简写为 long，字长为 4 字节。数的范围是-2147483648~2147483647

续表

类　型	说　明
unsigned short int	无符号短整型数。简写为 unsigned int，字长为 2 字节，数的范围是 0~65535
unsigned long int	无符号长整型数。简写为 unsigned long，字长为 4 字节，范围是 0~4294967295

例如下面的代码，就是定义和使用整型变量。

【范例 6-2】整型变量的定义与输出，示例代码如下。

示例代码 6-2

```
01  #include <stdio.h>
02  int main()
03  {
04    int i ;                          /*定义一个 signed short int 变量 i。*/
05    long j;                          /*定义一个 signed long int 变量 i。*/
06    unsigned short k;                /*定义一个 unsigned short int 变量 i。*/
07    i=123;                           /*i 赋值为 123。*/
08    j=1234L;                         /*j 赋值为 1234。*/
09    k=0x5e;                          /*k 赋值为十六进制的 5e。*/
10    printf("%d\n",i);
11    printf("%ld\n",j);
12    printf("%d\n",k);
13  }
```

【运行结果】用下面的命令编译这段代码。

```
gcc 6.2.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
123
1234
94
```

【代码解析】

- 代码第 04 行~09 行分别定义了整型、长整型和无符号短整型变量，然后对变量赋值。
- 代码第 10 行~13 行输出三个不同类型的变量的值。

在程序中，整型变量有以下三种表示方法。

- 普通十进制数，在程序中除了 0 以外不以 0 开始。如 0，123，-123 等。
- 八进制数，程序中以 0 开始，如 013，-013 等。
- 十六进制数，程序中以 0x 或 0X 开始，如 0x1E、0XEF 等。

另外，可在整型常数后添加一个"L"或"l"字母表示该数为长整型数，如代码中的 j=1234L，表示 j 为长整型数。

6.2.2　浮点型（float）

浮点型数指的是数值的小数点在存储空间中可以移动，可以用来表示不同精度与大小的数值。小数、科学计数法的值都是用浮点型变量来存储的。根据所占的存储空间与表示范围不同，浮点型数可以分为下面两种。

- float 单精度浮点数，字长为 4 个字节共 32 位。可表示数的范围是 3.4x10-38E~3.4x10+38E。
- double 双精度浮点数，字长为 8 个字节共 64 位二进制数，数的范围是 1.7x10-308E~1.7x10+308E。

【范例 6-3】浮点型变量的定义与输出，示例代码如下。

示例代码 6-3

```
01  #include <stdio.h>
02  int main()
03  {
04    float m;                          //定义一个 float 变量。
05    double n ;                        //定义一个 double 变量。
06    m=1.234;                          //对 m 赋值。
07    n=.1234;
08    printf("%f\n",m);                 //输出变量。
09    printf("%f\n",n);
10  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 6.3.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1.234000
0.123400
```

【代码解析】上面的代码分别定义了单精度和双精度变量，并且对变量赋值，最后输出变量值时使用了转移字符“%f”。

与整型变量相比，浮点型变量在下面这些方面是不同的。

- 浮点常数只有十进制一种进制。
- 绝对值小于 1 的浮点数，其小数点前面的零可以省略。如 0.123 可写为.123。
- 用默认格式输出浮点数时，都是保留 6 位小数，没有小数的后面加零。

6.2.3　字符型（char）

字符型变量在计算机中以 ASCII 码方式表示，长度为 1 个字节。各种字母、符号都可以是一个字符型的变量。数字 0~9 也可以存储为一个字符型变量。当数字存储为字符型变量时，保存的是这个数字的 ASCII 码的值，与 int 型保存数值是不相同的。

注意：ASCII 码是计算机内部的字符编码集。所有的信息都是以 ASCII 码的形式保存在计算机中的。一个字符变量对应 ASCII 码表中的一个字符。

在程序中，字符可直接用单引号引起来。如'A'、'b'、'9'、'@'都是一个字符型变量。也可以用这一字符的 ASCII 码值来表示一个字符。例如 87 表示大写字母'W'。除了屏幕上可以直接显示的字符以外，还有一些字符是用来进行格式控制的，并不能直接显示在屏幕上。例如\n 表示一个换行符，ASCII 码为八进制 010，而不是一个字符\加一个字符 n。C 程序中常用的转义字符如表 6.2 所示。

表 6.2 转义字符

字　符	名　称	ASCII 码	字　符	名　称	ASCII 码
\a	响铃（BEL）	007	\v	垂直制表（VT）	011
\b	退格（BS）	008	\\	反斜杠	092
\f	换页（FF）	012	\?	问号字符	063
\n	换行（LF）	010	\'	单引号字符	039
\r	回车（CR）	013	\"	双引号字符	034
\t	水平制表（HT）	009	\0	空字符（NULL）	000

在程序中字符变量需要用 char 来定义。例如下面的代码，是字符变量的定义与使用实例。

【范例 6-4】字符变量的定义与输出，示例代码如下。

示例代码 6-4

```
01  #include <stdio.h>                    /*包含头文件*/
02  int main()                            /*主函数*/
03  {
04    char m;                             /*定义字符型变量*/
05    char n;
06    char t;
07    m='a';                              /*为字符变量赋值*/
08    n='!';
09    t='\n';
10    printf("%c\n",m);                   /*输出 a 再输出一个换行。*/
11    printf("%c\n",n);                   /*输出!再输出一个换行。*/
12    printf("%c\n",t);                   /*输出的 t 为一个换行.*/
13  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 6.4.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a
!
```

【代码解析】上面的代码中使用关键字“char”定义字符变量，为变量赋值时使用单引号将字符括起来，最后输出了变量的值。

6.2.4 变量名

变量名只能由字母、数字、下画线组成，且头一个字母只能是字母或下画线，变量名之间不能有空格或其他字符。这样的变量才是合法的。例如下面这些变量名是合法变量名。

```
ab   _a   a2  AAA   A_B   __AB
```

下面这些变量名，不符合变量名的要求，是不合法的。

```
3a                  /*不能以数字开头。*/
a.b                 /*变量名之间不能有特殊符号。*/
$ab                 /*不能以特殊符号开头。*/
a b                 /*变量名中不能有空格。*/
```

不合法的变量名会引起程序错误。除了这些变量名以外，系统关键字也不能作为变量名。

例如下面这些词是系统关系字，作为变量名时会引发生错误。

```
int   printf   float   exit
```

提示：在编写代码时变量名的选择要遵循“见名知义”的原则，即变量名要尽量和变量表示的内容相关，看到变量名字时即可知道变量的功能。

6.2.5　字符 NULL

NULL 是一个特殊的字符，在 C 程序中有着重要的功能。NULL 在 ASCII 码表中是 0 号，在程序中表示空字符，或者没有这个数值。在定义变量时，如果要对变量赋值为空值，可将这个变量赋值为 NULL，如下面的代码所示。

```
int a=NULL;
char b=NULL;
float c=NULL;
```

提示：ASCII 码是计算机字符的内部编码，每一个字符对应 ASCII 码表中的一个编号。详细介绍见本书 8.5.3 中的内容。

6.3　变量的赋值与输出

变量的赋值指的是对一个变量指定一个值，这个值可以用于程序的运算。变量的输出指的是将变量的值显示在屏幕上，程序执行完毕以后需要将结果输出。

6.3.1　变量的赋值

赋值符号是等号，含义是将等号右边的值或表达式结果传递给等号左边的变量。例如下面的代码就是定义变量和赋值的例子。

```
01  #include <stdio.h>                    /*包含头文件*/
02  int main()                            /*主函数*/
03  {
04  int i =0,j=1,k=123;                   /*定义三个整型变量，定义变量同时赋值。*/
05  int s;
06  char a,b,c;                           /*定义三个字符型变量。*/
07  s=j;                                  /*将 j 赋值给 s。*/
08  a='a';                                /*将字符 a 赋值给变量 a。*/
09  b=a;                                  /*把变量 a 的值赋值给 b。*/
10  c=b;                                  /*把变量 b 的值赋值给 c。*/
11  }
```

6.3.2　printf 函数输出变量

在前面的代码中，都是使用 printf()函数进行变量的输出。printf()函数称为格式输出函数，功能是按用户指定的格式，把指定的数据显示到屏幕上。printf 函数是一个标准库函数，它的函数原型在头文件“stdio.h”中，该函数的使用方法如下所示。

```
printf("格式控制字符串",输出变量列表)
```

输出变量列表中给出了各个输出变量。格式控制字符串用于指定输出格式，由格式字符串和非格式字符串两种组成。

- 格式字符串是以%开头的字符串，在%后面跟有各种格式字符。不同格式字符串的作用

是说明输出数据的类型、形式、长度、小数位数等。如“%d”表示按十进制整型输出，“%ld”表示按十进制长整型输出。这些格式字符串与输出变量列表中的变量一一对应。

- 格式字符串以外的内容是非格式字符串，在输出时按照原有的字符直接输出。

格式字符串的内容和格式字符的意义如下所示。

- %c：输出单个字符，参数为该字符的 ASCII 码。
- %d：以十进制形式输出带符号整数（正数不输出符号）。
- %e 或%E：以指数形式输出单、双精度实数，默认 6 位小数。
- %f：以小数形式输出单、双精度实数，默认 6 位小数。
- %g 或%G：以%f 或%e 中较短的输出宽度输出单、双精度实数。如果指数小于-4 或大于等于默认精度，则使用%e 或%E 格式输出。否则用%f 格式输出，省略末尾多余的 0。
- %i：以十进制形式输出带符号整数，同%d。
- %o：以八进制形式输出无符号整数（不输出前缀 O）。
- %s：输出字符串，参数为 char 指针，显示字符串中所有的字符。
- %u：以十进制形式输出无符号整数。
- %x 或%X：以十六进制形式输出无符号整数，%x 输出小写，%X 输出大写。

例如下面的代码，是 printf()函数格式化输出的实例。

【范例 6-5】使用 printf 函数输出变量的值，示例代码如下。

示例代码 6-5

```
#include <stdio.h>
int main()
{
  int i=117;                          /*定义整型变量 i=117。*/
  char a='m';                         /*定义字符定量 a。*/
  float m=123.1234567 ;               /*定义一个浮点型变量。*/
  printf("%d\n",i);                   /*以整型输出 i。*/
  printf("%c\n",i);                   /*以字符型输出 i。*/
  printf("%o\n",i);                   /*以八进制输出 i。*/
  printf("%x\n",i);                   /*以十六进制小写输出 i。*/
  printf("%X\n",i);                   /*以十六进制大写输出 i。*/
  printf("%c\n",a);                   /*以字符型输出 a。*/
  printf("%d\n",a);                   /*以整型输出 a。*/
  printf("%o\n",a);                   /*以八进制输出 a。*/
  printf("%x\n",a);                   /*以十六进制输出 a。*/
  printf("%X\n",a);                   /*以十六进制大写输出 a。*/
  printf("%f\n",m);                   /*以浮点型输出 f。*/
  printf("%e\n",m);                   /*以科学计数法输出 m。*/
}
```

【运行结果】输入下面的命令，编译这个 C 程序文件。

```
gcc 6.5.c
```

输入下面的命令运行这一个程序。

```
./a.out
```

程序的运行结果如下所示。

```
117
u
165
75
75
```

```
m
109
155
6d
6D
123.123459
1.23123e+02
```

【代码解析】

- 代码第 04~06 行定义变量。
- 代码第 07~18 行调用 printf 函数输出变量的值，需要注意变量输出格式的变化。

6.3.3　scanf 函数从键盘读入变量

scanf()函数是键盘格式化输入函数，可以从键盘输入读取信息，这个函数的使用方法如下所示。

```
scanf("<格式化字符串>", <地址表>);
```

这里的格式化字符串与上一节中的格式化字符串含义是相同的。当有多个要读取的变量时，可以用空格或其他符号隔开，在键盘输入时也相应地输入这些符号。例如下面的代码，就是使用 scanf()函数从键盘读取输入，再输出变量的例子。

【范例 6-6】使用 scanf 函数获取用户输入的内容，示例代码如下。

示例代码 6-6

```
01  #include <stdio.h>                                  /*包含头文件*/
02  int main()                                          /*主函数*/
03  {
04      int i ,j ;                                      /*定义程序中的变量。*/
05      int m;
06      printf("please enter a char:\n");               /*提示信息。*/
07      scanf("%c",&m);                                 /*输入一个字母。*/
08      printf("the char is %c.\n",m);                  /*输出一个字母。*/
09      printf("please enter a number:\n");
10      scanf("%d",&i);                                 /*输入一个整数。*/
11      printf("please enter another number:\n");
12      scanf("%d",&j);
13      printf("%d + %d = %d \n",i ,j ,i+j);            /*输出两个整数和这两个数的和。*/
14  }
```

【运行结果】输入下面的命令，编译这个文件。

```
gcc 6.6.c
```

输入下面的命令运行这个程序。

```
./a.out
```

运行程序时，光标等待输入内容。输入内容以后，按“Enter”键继续向后执行。程序的运行结果如下所示。

```
please enter a char:
```

这时从键盘输入一个这母 a，然后按“Enter”键。这时程序的结果如下所示。

```
the char is a.
please enter a number:
```

这时输入一个数字 5，然后按“Enter”键，程序显示如下所示。

```
please enter another number:
```

这时输入一个数字 3，然后按“Enter”键，程序显示结果如下所示。

```
5 + 3 = 8
```

【代码解析】上面的代码中使用 printf 函数输出程序的提示信息，使用 scanf 函数获得用户输入的值，并把值保存到变量中，然后又使用 printf 函数输出变量的值。

6.4 运算符

运算符指的是用来完成各种运算的符号。C 程序中有着丰富的运算符，很多运算都是通过运算符来实现的。本节将讲解常用运算符的操作。

6.4.1 算术运算符

算术运算符是指进行简单数学计算的运算符，是程序中最简单、最常用的运算符。算术运算符的种类和使用方法如表 6.3 所示。

表 6.3 算术运算符

运 算 符	含 义	运 算 符	含 义
+ -	加号和减号	+= -=	相加赋值和相减赋值，对原有值加或者减去一个值
*/	乘号和除号	*= /=	相乘赋值或相除赋值，对原有值乘或除一个值
++	自加，在原来的值上面加 1	%	求余运算，第一个数除以第二个数求余数
--	自减，在原来的值上面减 1		

例如下面的代码，是使用这些运算符进行简单的数学运算。

【范例 6-7】C 语言中算术运算符的用法，示例代码如下。

示例代码 6-7

```
01  #include <stdio.h>
02  int main()
03  {
04     int i =3 ,j = 7 ,k;                    /*定义变量和赋值。*/
05     float m=2.5 , n = 3.8,t;
06     k= 3+7;                                /*做加法运算。*/
07     printf("%d\n",k);
08     k*=5;                                  /* *=运算，表示把 k 乘以 5 再赋值给 k。*/
09     printf("%d\n",k);
10     k=k%j;                                 /*求余运算。*/
11     printf("%d\n",k);
12     printf("%d\n",i+j);                    /*可以直接在参数列表中进行运算。*/
13     t= 3.8/2.5;
14     printf("%f\n",t);                      /*以浮点数的形式输出。*/
15     printf("%f\n",m*n+5);                  /*直接在参数列表中进行运算。*/
16  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 6.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
10
```

```
50
1
10
1.520000
14.500000
```

【代码解析】代码第 12 行和第 15 行，在调用 printf 函数输出变量值之前完成算术表达式的求值。

对于++、--运算的使用，将符号写在前面和写在后面的含义是不同的。++写在后面时，表示取得值然后再自加，写在前面时，表示先自加再取值。例如下面的代码，就是自加和自减运算符的使用。

【范例 6-8】++与--运算符的应用，示例代码如下。

示例代码 6-8

```
01  #include <stdio.h>
02  int main()
03  {
04     int i =3 ,j = 7;
05     i++;                                  /*i 的值为 4。*/
06     printf("%d\n",i);
07     printf("%d\n",i++);                   /*输出 i 的值为 4，然后自加得 5。*/
08     printf("%d\n",i++);                   /*输出 i 的值为 5，然后自加得 6。*/
09     printf("%d\n",++i);                   /*自加得 7，输出结果得 7。*/
10     printf("%d\n",--i);                   /*自减得 6，输出结果为 6。*/
11     printf("%d\n",i++ + j++);             /*先取值相加得 13，然后分别自加。*/
12  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 6.8.c
```

输入下面的命令，运算这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
4
4
5
7
6
13
```

【代码解析】

- 代码第 07~08 行，首先使用 printf 函数输出变量 i 的值，然后使变量 i 的值自增 1。
- 代码第 09~10 行，首先使变量 i 的值自增 1 和自减 1，然后调用 printf 函数输出变量的值。

6.4.2　关系运算符

所谓关系运算符，指的是比较两个数的大小关系或相等关系的符号。关系运算符的种类与使用方法如表 6.4 所示。

表 6.4　关系运算符

运 算 符	含　义	运 算 符	含　义
>	大于。第一个数大于第二个数时返回真	<=	小于等于
>=	大于等于	==	等于。两个数相等时返回真
<	小于	!=	不等于。两个数不相等时返回真

关系运算符的返回值是是真（true）和假（false）的概念。在 C 语言中，真值 true 可以是不为 0 的任何值。而假值 false 则为 0。使用关系运算符，若表达式为真则返回 1，否则表达式返回值为 0。例如下面的关系运行符的使用。

```
6>5                返回 1
10=9               返回 0
```

6.4.3 逻辑运算符

所谓逻辑运算符，指的是用形式逻辑原则来建立数值间关系运算的符号。逻辑运算有与、或、非三种，使用方法如表 6.5 所示。

表 6.5 逻辑运算符

运 算 符	含 义	返 回 值
&&	逻辑与	所有的条件为真则返回真
||	逻辑或	只要有一个条件为真则返回真
!	逻辑非	真值取非返回假，假值取反返回真

例如下面的代码，是关系运算符与逻辑运算符的使用实例。

【范例 6-9】关系运算符与逻辑运算符的应用示例代码如下。

示例代码 6-9

```
01  #include <stdio.h>
02  int main()
03  {
04     int i=5,j=3;
05     printf("%d\n",i>j);                    /*判断 i 是否大于等于 j，返回真，输出 1。*/
06     printf("%d\n",i==j);                   /*判断 i 是否等于 j，返回假，输出 0。*/
07     printf("%d\n",i<j);
08     printf("%d\n",i!=j);
09     printf("%d\n",i!=j);
10     printf("%d\n",(i>j)&&(i>=j));          /*判断两个条件，然后做逻辑与运算。*/
11     printf("%d\n",(i>j)&&(i<j));
12     printf("%d\n",1&&1&&1);                /*所有条件为真，则返回真。*/
13     printf("%d\n",1&&0&&1);                /*有一个条件为假则返回假。*/
14     printf("%d\n",0||0||1);                /*有一个条件为真则返回真。*/
15     printf("%d\n",!0);                     /*非假返回真。*/
16  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 6.9.c
```

输入下面的命令，运算这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1
0
0
1
1
1
0
1
0
```

```
1
1
```

【代码解析】

- 代码第 05~09 行分别演示了关系运算符“>”、“==”、“<”、“!=”的使用实例。
- 代码第 10~15 行分别演示了逻辑运算符“&&”、“|”、“!”的使用实例。

6.5　综合练习

本节通过编写一个猜数字游戏，演示 C 程序设计的基本知识。使用 scanf 函数获取用户输入的数字，然后判断与答案（预先指定的数字）的大小，并给出大小关系。直到用户输入的数字与预设答案的数字相同时退出程序。

【范例 6-10】猜数字游戏，示例代码如下。

示例代码 6-10

```
01  #include <stdio.h>                                          /*包含头文件*/
02
03  /*guess number*/
04
05  int main()                                                  /*主函数*/
06  {
07        int answer = 58;
08        int num =0 ;
09        while(1)                                              /*while 循环语句*/
10        {
11              printf("Please input a number:\n");
12              scanf("%d", &num);                              /*获取用户输入的数字*/
13              if(num == answer){                              /*如果答案正确则退出循环*/
14                    printf("OK! The answer is %d \n",num);
15                    break;
16              }
17              if(num > answer)                                /*给出提示信息*/
18                    printf("greater than answer!Retry! \n");
19              else
20                    printf("lower than answer!Retry! \n");
21        }
22  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 6.10.c
```

执行这个程序。

```
./a.out
```

程序提示输入一个数字，然后按“Enter”键后，提示输入的数字与答案的关系，按照提示输入接下来的数字，直到输入的数字与答案相同为止。输出的结果如下所示。

```
Please input a number:
50
lower than answer!Retry!
Please input a number:
60
greater than answer!Retry!
Please input a number:
55
lower than answer!Retry!
Please input a number:
```

```
57
lower than answer!Retry!
Please input a number:
58
OK! The answer is 58
```

6.6 小结

本章讲解了数据类型、变量赋值与输出、常用运算符等知识。这些知识是编写 C 程序的基础，通过这些知识的学习，可以理解 C 程序的一些概念，编写简单的 C 语言程序。在本章的学习中，数据类型与运算符的使用是重点，需要详细地理解数据类型的概念和输出方式。

6.7 习题

一、填空题

1．C 程序由_____ 、_____、_____等部分组成。

2．所谓_____，就是在编写程序时对代码补充说明，不影响程序。任何一个程序员，都不可能完全记忆所写过的代码，_____的作用是对代码的阅读和理解进行提示。

3．_____指的是一类数据的集合，是对数据的抽象描述。

4．_____是一个特殊的字符，在 C 程序中有着重要的功能。它在 ASCII 码表中是 0 号，在程序中表示空字符，或者没有这个数值。

5．变量的_____指的是对一个变量指定一个值，这个值可以用于程序的运算。变量的_____指的是将变量的值显示在屏幕上，程序执行完毕以后需要将结果输出。

6．_____指的是用来完成各种运算的符号。C 程序中有着丰富的_____，很多运算都是通过运算符来实现的。

二、选择题

1．下列选项中，哪个不是 C 程序支持的数值型数据类型（　　）。

A．int 型　　B．float 型

C．NULL　　D．double 型

2．关于 C 程序编程描述中正确的是（　　）。

A．C 程序不支持多行注释“/*.....*/”

B．C 程序支持的浮点型数据类型包括 float 型和 double 型

C．字符 NULL 表示变量的值为‘’

D．printf 函数用户从键盘读取数据，scanf 函数用户将内容打印大屏幕上

3．C 程序的基本结构不包括（　　）。

A．语句　　B．函数

C．包含文件　　D．注释

4．关于 C 程序中的算术运算符说法中错误的是（　　）。

A．+，-表示加法和减法

B．++，--表示自增 1 和自减 1

C．%表示除法运算，返回精确的运算结果（带小数点）

D．*，/表示乘法和除法运算

三、简答题

1．简述 C 程序的基本结构和一般格式。
2．简述 C 程序支持的数据类型及用途。
3．简述 C 程序支持的运算符及其用法。

四、编程题

1．编写一个简单的 C 程序，实现在屏幕上打印一行文本，使用注释标注程序的功能。
2．编写程序实现读取用户输入的两个数字，比较这两个数字的大小，并输出其大小关系。

第 7 章　C 程序的常用语句

流程控制语句控制着程序的执行过程，本章介绍 C 程序中常用的流程控制语句的用法及两种特殊的语句结构，这些知识是进行软件开发所必备的。本章的主要内容如下：

- 流程控制语句；
- 两种特殊的语句结构。

7.1　流程控制语句

在程序中，需要对语句的执行进行分支选择，或者重复执行某些语句，这些实现程序逻辑功能或多次循环执行运算的语句就是流程控制语句。流程控制语句有条件语句与循环语句两种。条件语句实现程序的逻辑功能，循环语句实现程序的重复执行功能。

7.1.1　if 条件语句

if 条件语句的作用，是对一个条进行判断。如果判断的结果为真，则执行条件后面的语句。如果执行判断的结果为假，则跳过后面的语句。最基本的 if 条件语句结构如下所示。

```
if(条件)
{
   条件成立时需要执行的内容。
}
```

这种条件语句的执行流程如图 7-1 所示。

如果 if 后面的判断结果，是指两种情况，第一种条件不成立时一定是第二种情况。则条件语句可以使用 if else 结构，这种条件语句的使用方法如下所示。

```
if(条件)
{
   条件成立时需要执行的内容。
}
else
{
   条件不成立时执行的内容。
}
```

这种条件语句中，先判断 if 的条件内容。如果返回的结果为真，则执行 if 后面的语句。否则就执行 else 后面的语句。程序的执行流程如图 7-2 所示。

例如下面的实例，实现从键盘输入一个整数。用 if 语句判断这个整数是奇数还是偶数，然后输出判断结果。

【范例 7-1】使用 if 语句判断数字是否为偶数，示例代码如下所示。

示例代码 7-1

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

```
03  {
04     int i ,j ;                                  /*定义两个整型变量。*/
05     printf("please input a number:\n");         /*提示输入。*/
06     scanf("%d",&i);                             /*从键盘读取一个整数。*/
07     j =i %2;                                    /*输入的数对 2 求余数。*/
08     if(j==0)                                    /*如果余数为 0 则输入是偶数。*/
09     {
10        printf("%d oushu.\n",i);
11     }
12     else                                        /*否则输出为奇数。*/
13     {
14        printf("%d jishu.\n",i);
15     }
16  }
```

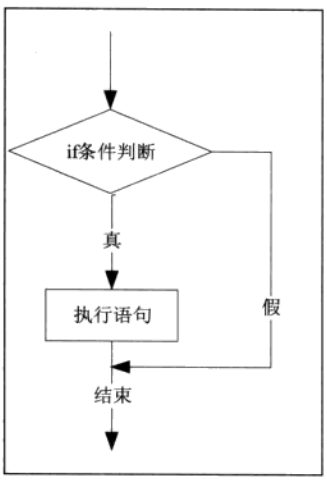


图 7-1　if 条件语句的流程

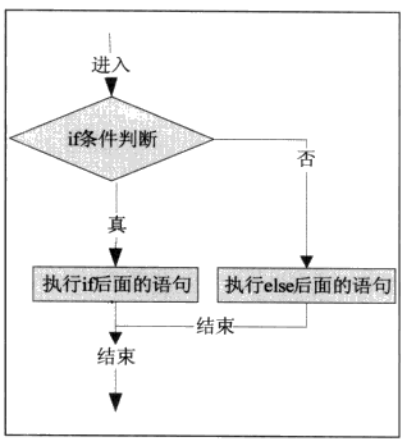


图 7-2　if else 条件语句的流程

【运行结果】编译程序，在终端中输入下面的命令，然后按“Enter”键。

```
gcc 7.1.c
```

在终端中输入下面的命令，运行这个程序。

```
./a.out
```

程序在运行时，显示下面的信息，提示输入一个整数。

```
please input a number:
```

在键盘上输入任意一个整数 12，程序的显示结果如下所示。

```
12 oushu
```

【代码解析】

- 代码第 04~07 行首先定义两个整型变量，然后输出提示信息，接下来用 2 除以输入的整数，最后将余数赋值给变量 j。
- 代码第 08~15 行通过判断变量 j 的值，而得出输入的整数是偶数还是偶数，并输出提示信息。

7.1.2　if 语句的嵌套

所谓 if 语句嵌套，指的是在一个 if 语句的执行内容中再进行 if 判断语句。当判断情况很

多时，需要使用 if 语句嵌套。例如下面就是一种简单的 if 语句嵌套。

```
if(条件)
{
   if(条件)
   {
      执行的内容}
   else
   {
      执行的内容
   }
}
else
{
   if(条件)
   {
      执行的内容}
   else
   {
      执行的内容
   }
}
}
```

在这个嵌套结构中，先进行一次条件判断，然后判断的结果进行另外一次条件判断。这种结构的执行流程可用图 7-3 来表示。

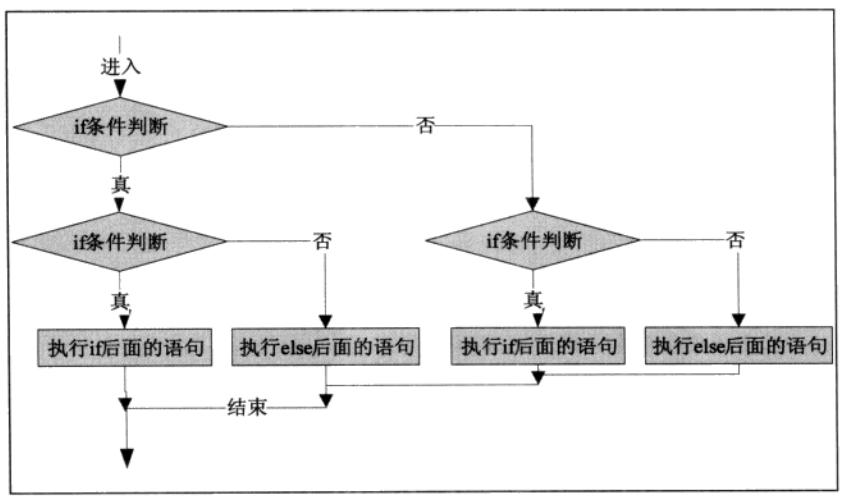


图 7-3　if 嵌套语句的流程

【范例 7-2】使用 if 语句的嵌套实现分数评级。假设对考试分数进行 A、B、C、D 评级，85 分以上为 A，70 分~85 分为 B，60 分~70 分为 C，60 分以下为 D。要实现这个功能，需要对分数进行多次 if 判断，可以使用如图 7-4 所示的 if 嵌套结构。程序的代码如下所示。

示例代码 7-2

```
01  #include <stdio.h>
02  int main()
03  {
04     int i;                                    /*定义一个变量。*/
```

```
05      printf("please input a number:\n");   /*提示输入信息。*/
06      scanf("%d",&i);                       /*从读取一个变量。*/
07      if(i>=70)                             /*第一次 if 判断。*/
08      {
09        if(i>=85)                           /*85 分以上的输出 A。*/
10        {
11          printf("A");
12        }
13        else                                /*其他的分数就是 70~85 分的，输出 B。*/
14        {
15          printf("B");
16        }
17      }
18      else                                  /*这里其他的就是 70 分以下的。*/
19      {
20         if(i>=60)                          /*60 分~70 分的输出 C。*/
21        {
22          printf("C");
23        }
24        else
25        {
26          printf("D");                      /*其他的一定是 60 分以下的，输出 D。*/
27        }
28      }
29  }
```

【运行结果】编译并运行这个程序，在提示后面输入一个数字 77，然后按“Enter”键。程序的 if 嵌套结构会判断分数的级别，显示的结果如下：

```
B
```

【代码解析】

- 代码第 03~06 行定义变量，给出提示信息，最后使用 scanf 函数获得用户输入的分数。
- 代码第 07~29 行是 if 语句嵌套的用法，用于判断输入的分数的等级，并打印出分数的等级。

7.1.3　switch 选择执行语句

当 if 语句的判断情况很多时，可以使用 switch 选择执行语句。在这种结构中，switch 对一个条件进行判断，然后分别列出可能的结果，每个结果执行不同的语句。switch 语句的使用方法如下所示。

```
switch(条件)
{
  case 结果 1 :
       执行内容 1; break ;
  case 结果 2 :
       执行内容 2; break ;
.....
  case 结果 n :
       执行内容 n; break ;

}
```

这种结构的运行流程如图 7-4 所示。

下面是一个 switch 语句的使用实例。用 scanf()函数从键盘读取一个数字，如果数字在 0~6 之间，输出对应的星期一~星期六，否则提示错误。

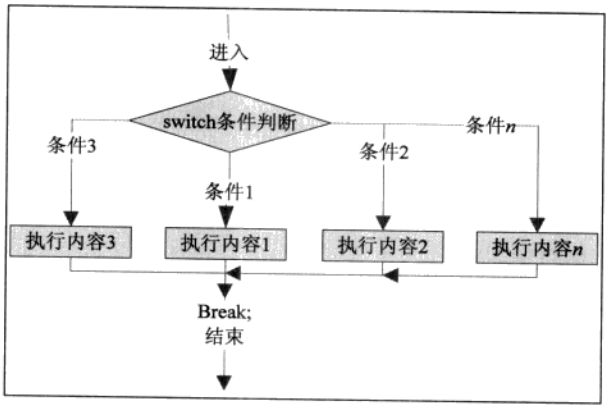


图 7-4　switch 语句的流程

【范例 7-3】使用 switch 语句判断输入的数字代表星期几，示例代码如下所示。

示例代码 7-3

```
#include <stdio.h>
int main()
{
   int i;                                          /*定义一个变量。*/
   printf("please input a number:\n");              /*提示输入信息。*/
   scanf("%d",&i);                                 /*从键盘读取一个数字。*/
   if(i<0 || i >6)              /*如果这个数字小于 0 或大于 6 则输出错误提示信息。*/
   {
     printf("input error.\n");
   }
   else                                            /*否则执行判断。*/
   {
     switch(i)                                     /*执行判断。*/
     {
       case 0:
         printf("Sunday\n") ; break;               /*7 个判断条件和结果。*/
       case 1:
         printf("Monday\n") ; break;
       case 2:
         printf("Tuesday\n") ; break;
       case 3:
         printf("Wednesday\n") ; break;
       case 4:
         printf("Thursday\n") ; break;
       case 5:
         printf("Friday\n") ; break;
       case 6:
         printf("Saturday\n") ; break;
     }
   }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.3.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序运行时，会输出下面的提示信息要求输入一个数字。

```
please input a number:
```

这时输入一个数字 5，然后按“Enter”键。程序显示的结果如下所示。

```
Friday
```

【代码解析】

- 代码第 06 行从键盘读取一个数字，代码第 07~31 行判断用户输入的数字是否大于等于 0 和小于等于 6。
- 代码第 13~29 行使用多重分支语句判断用户输入的数字是 0~6 中的哪一个，并输出其代表的星期。

7.1.4　for 循环语句

循环语句指的是，使用循环结构使一个程序过程多次执行。for 循环语句是最常用的循环语句，这种结构的用法如下所示。

```
for(起始条件;循环条件;循环变量变化)
{
  循环体执行的内容;
}
```

例如下面的程序，实现从 1~6 之间 6 个数字的输出。

【范例 7-4】使用 for 循环语句输出 1~6 之间的数字，示例代码如下所示。

示例代码 7-4

```
01  #include <stdio.h>
02  int main()
03  {
04     int i;                                  /*定义循环变量。*/
05     for(i=1;i<=6;i++)                       /*开始 for 循环*/
06     {
07       printf("%d\n",i);                     /*循环体，输出 i 的值和换行。*/
08     }
09  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.4.c
```

输入下面的命令，执行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1
2
3
4
5
6
```

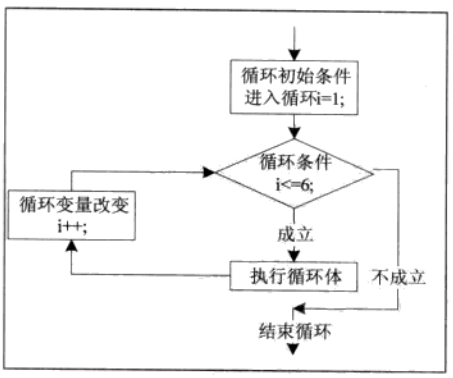


图 7-5　for 循环的程序流程

【代码解析】这个程序的运行流程可用图 7-5 来表示。

根据图 7-6 可知，for 循环是用反复判断循环条件、执行循环体、改变循环变量的方法来实现程序的循环流程的。代码 7-4 的执行过程如下所示。

① 第 4 行定义一个循环变量 i。

② 第 5 行，先执行循环初始条件，对 i 赋值为 1。

③ 进行第一次条件判断 i 是否小于等于 6。判断的结果为真则执行下面的循环体，为假时则跳出循环。

④ 循环体执行完毕以后，进入循环条件改变运算 i++。然后再次进行条件判断，从而完成程序的循环。

⑤ 当 i 等于 7 时，条件判断不成立，这时跳出循环，结束程序。

注意：循环变量的改变和条件判断要确保 for 循环可以中断，否则就会构成死循环。死循环指的是循环语句无法达到停止条件，永远执行下去，可能引起计算机的死机。

下面是一个 for 循环实例，求出 100 以内的整数和。

【范例 7-5】使用 for 循环语句实现求和运算，示例代码如下所示。

示例代码 7-5

```
01  #include <stdio.h>
02  int main()
03  {
04    int i ;                                  /*定义循环变量。*/
05    int sum = 0;                             /*定义求和的结果。*/
06    for(i=0;i<=100;i++)                      /*循环语句。*/
07    {
08       sum =sum +i ;                         /*循环体。*/
09    }
10    printf("sum = %d",sum);                  /*输出结果。*/
11  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.5.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序会从 1~100 进行循环，每次循环时，与 sum 相加的结果赋值给 sum。这样就得到了 100 以内整数的和。

程序的运行结果如下所示。

```
sum = 5050
```

【代码解析】代码第 06~09 行利用 for 循环实现求和运算。第 06 行代码给定了循环的初始条件、循环条件和步进方式。

注意：定义求和结果 sum 时，需要对 sum 赋初值 0，这样可以与后面的数相加而结果不变。进行求积运算时，需要对结果赋初值 1。

7.1.5 for 循环的嵌套

在 for 循环结构的循环体中，再次执行一个循环语句，这种结构就是 for 循环嵌套。在循环嵌套结构中，每执行一次外层循环，就会多次执行内层循环。for 循环嵌套的结构如下所示。

```
for(起始条件;循环条件;循环变量变化)
{
```

```
    for(起始条件;循环条件;循环变量变化)
    {
       循环体。
    }
}
```

这种结构的运行流程图如图 7-6 所示。

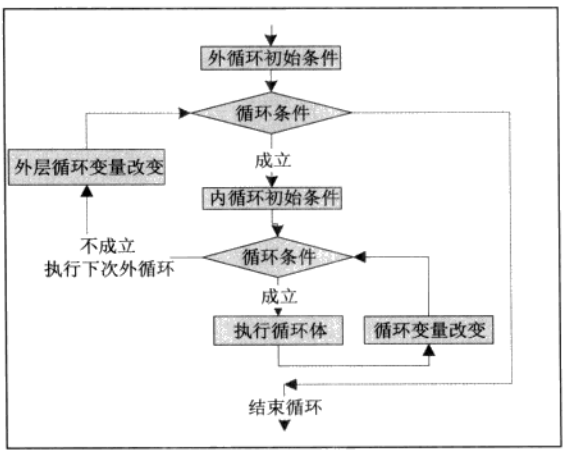


图 7-6　for 循环的程序流程

例如下面是一个循环嵌套结构实例，输出一个星号图案。在外层循环中，实现个数和行数的计数，在内层循环中实现星号的输出。

【范例 7-6】使用 for 循环的嵌套实现统计个数和行数，示例代码如下所示。

示例代码 7-6

```
01  #include <stdio.h>
02  int main()
03  {
04     int i ,j;                                    /*定义程序中的变量。*/
05     for(i=1;i<7;i++)                             /*外层循环。*/
06     {
07        for(j=1;j<=i;j++)                         /*内层循环。*/
08        {
09         printf("* ");                           /*输出星号和一个空格。*/
10        }
11         printf("\n");                           /*输出一个换行。*/
12     }
13  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.6.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
*
* *
* * *
* * * *
```

```
* * * * *
* * * * * *
```

【代码解析】代码第 05 行尾外层 for 循环，第 07 行为内层 for 循环。

7.1.6 for 循环应用实例：输出九九乘法口诀表

【范例 7-7】使用 for 语句打印乘法口诀表。乘法口诀表需要用 for 循环嵌套来实现，一个循环变量作为被乘数，另一个循环变量作为乘数。在输出结果时，需要用一个 if 判断来实现排列。每一行的循环需要输出一个换行。这种程序的代码如下所示。

示例代码 7-7

```
01  #include <stdio.h>
02  int main()
03  {
04      int i,j;                                    /*定义两个循环变量。*/
05      for(i=1;i<=9;i++)                           /*外层循环。*/
06      {
07          for(j=1;j<=i;j++)                       /*内层循环。*/
08          {
09            if (j<=i)                             /*判断 i 与 j 的大小，实现排列。*/
10              {
11                 printf("%d*%d=%d  ",j ,i , i*j );  /*输出乘法式。*/
12              }
13          }
14          printf("\n");                           /*输出一个换行。*/
15       }
16  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12  4*4=16
1*5=5  2*5=10  3*5=15  4*5=20  5*5=25
1*6=6  2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7  2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8  2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9  2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

【代码解析】代码第 05~07 行为两层 for 循环语句的嵌套。在这个程序中，需要注意乘法算式的输出，下面是代码中的输出语句。

```
printf("%d*%d=%d  ",j ,i , i*j );
```

输出格式字符串中，三个%d 分别对应输出列表中的 j、i、i*j。而字符*、=与空格按照原来的格式输出，这就实现了乘法式的输出。

7.1.7 while 循环语句

while 语句是另一种循环语句，可以实现与 for 循环一样的功能。while 循环的使用方法如下所示。

```
while(循环条件)
{
   循环体语句;
   循环条件改变;
}
```

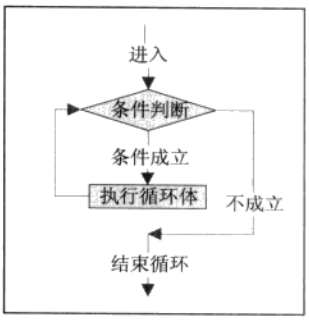


图 7-7　while 循环的执行流程图

在 while 循环结构中，先判断循环的条件是否成立。如果条件成立则执行循环体中的语句，然后继续判断条件是否成立。如果条件不成立则跳出循环，执行后面的语句。while 循环的执行流程可用图 7-7 来表示。

因为 while 循环直接进入循环，而没有对循环变量赋值，所以需要在执行循环以前对循环变量赋值。在循环体中也需要更改循环变量。下面的实例，是用 while 循环求出 1~100 之间的整数和。

【范例 7-8】使用 while 语句计算 1~100 之间整数累加的和，示例代码如下所示。

示例代码 7-8

```
01  #include <stdio.h>
02  void main()
03  {
04    int i,sum;                          /*定义程序的变量。*/
05    i=0;                                /*变量赋初值。*/
06    sum=0;                              /*求和变量赋初值。*/
07    while(i<=100)                       /*进入 while 循环。*/
08    {
09      sum=sum+i;                        /*sum 与 i 求和赋值给 sum。*/
10      i++;                              /*i 自加。*/
11    }
12    printf("sum = %d \n",sum);          /*输出结果。*/
13  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.8.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 5050
```

【代码解析】代码第 07~11 行为 while 循环体，第 09 行为求和语句，第 10 行实现循环条件变量 *i* 自增 1 操作。

注意：在这个程序中需要注意下面两个问题。

（1）需要对循环变量与求和变量赋初值。while 循环在进入循环时没有赋值语句，如果在循环以外没有对循环变量赋初值，则程序会发生异常。

（2）在循环体中一定要改变循环变量的值，否则 while 循环的条件就始终成立，无法跳出循环。这种情况可能造成计算机死机。

7.1.8　do while 循环语句

do while 循环与 while 循环相似，实现程序对循环体的多次执行。与 while 循环不同的是，

do while 循环先执行一次循环体，然后判断条件是否成立。如果条件成立则执行下一次循环，否则结束循环。这种循环结构的使用方法如下所示。

```
do
{
   循环体;
}
while(循环条件);
```

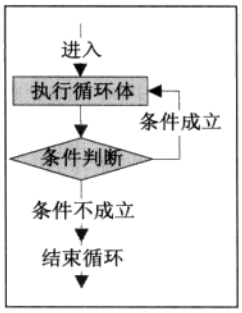


图 7-8　do while 循环的执行流程图

do while 循环的程序流程如图 7-8 所示。

与 while 循环一样，do while 循环在进入循环以前，需要对循环变量赋初值。在循环体中需要更改循环变量的值。下面是一个使用 do while 循环的实例，可以完成 0~100 之间偶数的求和。

【范例 7-9】使用 do-while 语句计算 0~100 之间的偶数的和，示例代码如下所示。

示例代码 7-9

```
01  #include <stdio.h>
02  void main()
03  {
04    int i,sum;                          /*定义两个程序变量。*/
05    i=0;                                /*循环变量赋初值。*/
06    sum=0;                              /*循环结果赋初值。*/
07    do                                  /*开始 do 循环。*/
08    {
09       sum=sum+i;                       /*求和。*/
10       i=i+2;                           /*变量加 2，使 i 是 100 以内的偶数。*/
11    }while(i<=100);                     /*判断条件。*/
12    printf("sum = %d \n",sum);
13  }
```

【运行结果】

输入下面的命令，编译这个程序。

```
gcc 7.9.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 2550
```

【代码解析】代码第 07~11 行是 do-while 循环体，首先执行一次第 09 行和第 10 行代码后才进行循环条件的判断。

注意：使用 do-while 循环时，循环体内的代码至少执行一次。

7.1.9　转移控制语句：continue

有时需要在判断、循环、程序执行过程中实现程序流程的强制转移，这就需要使用转移控制语句。C 程序中有 continue、break 和 return。在循环语句中，continue 语句可以中断循环体后面语句的执行，直接进入下一次循环。下面是一个在 for 循环中使用 continue 语句的实例，

实现 100 以内偶数的求和。

【范例 7-10】判断数字是否为偶数。在程序中，用对 2 求余判断结果的方法，来判断是不是一个偶数。如果不是偶数，则用 continue 语句进行下一次循环。

示例代码 7-10　\源文件\07\7.10.c

```
01  #include <stdio.h>
02  void main()
03  {
04    int i,sum;                          /*定义程序的变量。*/
05    sum=0;                              /*对求和变量赋初值为 0。*/
06    for(i=0;i<=100;i++)                 /*for 语句实现 1 到 100 的循环。*/
07    {
08      if(i%2==1)                        /*判断是否为奇数。*/
09      {
10        continue;                       /*如是是奇数则跳过下面的语句，进入下一次循环。*/
11      }
12      sum = sum +i;                     /*sum 与 i 求和的结果赋值给 sum。*/
13    }
14    printf("sum = %d \n",sum);          /*输出结果。*/
15  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.10.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 2550
```

【代码解析】代码第 06~13 行为 for 循环体，程序进入 for 循环以后，从 1~100 执行 100 次循环。在每次循环中，变量 *i* 对 2 求余。如果结果为 1，则表明 *i* 为奇数，则用 continue 语句跳过后面循环体中语句的执行，进入下一次循环。否则进行求和运算。

7.1.10　转移控制语句：break

break 语句可以中断循环语句的执行。例如在 for 循环、while 循环的语句中，可以不设置循环条件，让循环自动执行。在循环体中对循环变量进行判断，当符合循环中止条件时，使用 break 语句使循环中止。例如下面用 for 循环与 break 语句实现 100 以内偶数的求和。

【范例 7-11】使用 for 语句与 break 语句实现 100 以内偶数的求和，示例代码如下所示。

示例代码 7-11

```
01  #include <stdio.h>
02  void main()
03  {
04    int i,sum;                        /*定义程序的变量。*/
05    sum=0;                            /*求和变量赋初值为 0。*/
06    for(i=0;  ;i++)                   /*进入 for 循环，不设置结束条件。*/
07    {
08      if(i>100)                       /*如果 i 的值大于 100，则用 break 语句中断循环。*/
09      {
10        break;                        /*break 语句跳出循环。*/
11      }
12      sum = sum +i;                   /*求和。*/
```

```
13    }
14    printf("sum = %d \n",sum);                    /*输出结果。*/
15  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.11.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
sum = 5050
```

【代码解析】代码第 06~13 行是 while 循环体，代码第 08~11 行使用 if 语句实现当变量 i 的值大于 100 时退出 while 循环。

提示：注意区分 continue 和 break 的区别，前者可以退出本次循环而直接进入下次循环，后者则彻底退出循环语句。

7.1.11 转移控制语句：return

程序中的 return 语句可以中止 main 主函数或自定义函数的执行。return 语句在 main 主函数与自定义函数中的使用是不同的。

- main 主函数中使用 return 语句，会中断当前程序的执行。程序中后面的所有部分都不再执行。
- 在自定义函数中使用 return 语句时，会中断函数的执行，返回到主函数的执行。如果 return 后面有参数，则参数就是函数的返回值。

例如下面的程序，使用了 return 语句来中断程序的执行。

【范例 7-12】 return 语句中断程序的执行，示例代码如下所示。

示例代码 7-12

```
01  #include <stdio.h>
02  void main()
03  {
04    printf("good morning.\n");
05    return ;
06    printf("good evening.\n");
07  }
```

【运行结果】编译并运行这个程序，结果如下所示。

```
good morning.
```

【代码解析】代码第 05 行使用 return 语句中断了程序的执行，return 以后的语句都是不执行的。

提示：通常情况下，程序中出现异常或错误，可以用 return 语句来中止程序。

7.2 两种特殊语句结构

在 C 程序中，有一些特殊结构的用法。本节将讲解三元操作符?和块语句的用法。?结构相当于一个 if else 条件语句。块语句是作为一个单元进行处理的语句组。

注意：三元运算符指的是一个运算符号有三个参数。

7.2.1　?三元操作符

问号（?）操作符是 C 程序中的一个强有力的操作符，实现逻辑判断的功能，可以代替程序中的 if else 语句。这种结构的用法如下所示。

条件表达式 1?结果表达式 1:结果表达式 2

?操作符在执行时实现了一个逻辑功能。先求条件表达式的值。如果条件表达式的结果为真，则求出结果表达式 1 的值并返回。如果条件表达式的值为假，则求出结果表达式 2 的值并返回。两个结果表达式之间用冒号（:）隔开。例如下面是一个简单的?操作符使用。

```
01  int x ,y;
02  x=15;
03  y=x>10?20:30;
```

?操作符的含义是，如果 x 大于 9，则 y 被赋值为 20，否则 y 被赋值为 30。在这个语句中，条件表达式的值为真，所以 y 被赋值为 20。这个语句结构也可以改写成如下所示的 if 语句结构。

```
01  int x ,y;
02  x=15;
03  if(x>10)
04     y=20;
05  else
06     y=30;
```

下面是一个?操作符使用实例，求出一个数的绝对值。

【范例 7-13】使用？操作符输出一个数字的绝对值。在程序中，需要对输入的值进行判断，如果大于等于 0，则输出原来的变量。如果小于 0，则输入这个数的相反数。程序的代码如下所示。

示例代码 7-13

```
01  #include <stdio.h>
02  void main()
03  {
04     float x,y;                                    /*定义变量。*/
05     printf("please input a number:\n");           /*输出提示信息。*/
06     scanf("%f", &x);                              /*从键盘读取一个数。*/
07     y=x>=0?x:(-x);                                /*用?操作符判断和赋值。*/
08     printf("number: %f \nvalue: %f\n",x,y);       /*输出原来变量和绝对值。*/
09  }
```

【运行结果】编译并运行这个程序，提示输入一个数字，输入 3 后按“Enter”键，程序执行结果如下所示。

```
please input a number:
3
number: 3.000000
value: 3.000000
```

【代码解析】代码第 07 行使用？操作符实现，当变量 x 的值大于等于 0 时，将变量 x 的值赋给变量 y，当变量 x 的值小于 0 时，将变量 x 的相反数赋给变量 y。

7.2.2　块语句

块语句是使用两个花括号（{}）括起来的一个语句集，形成一个语句体。循环语句、判断

语句要执行的内容，都是一个块语句。在程序中，块语句是作为一个整体来执行的，要么全部执行，要么都不执行。可以将块语句看能一个语句。块语句的格式如下所示。

```
{
  语句列表;
}
```

例如下面的程序就是使用块语句作为循环语句的循环体。

```
01  #include <stdio.h>
02  void main()
03  {
04    int i,sum;                          /*定义程序的变量。*/
05    i=0;                                /*变量赋初值。*/
06    sum=0;                              /*求和变量赋初值。*/
07    while(i<=100)                       /*进入 while 循环。*/
08    {
09      sum=sum+i;                        /*sum 与 i 求和赋值给 sum。*/
10      i++;                              /*i 自加。*/
11    }
12    printf("sum = %d \n",sum);          /*输出结果。*/
13  }
```

在这个程序中，while 语句的循环体是两个语句，这两个语句用花括号构成块语句。

7.3 综合练习

流程控制语句可以在程序中实现复杂的功能。判断语句可以实现程序的逻辑功能，循环语句可以完成程序的多次运算。本节将讲解流程控程语句的使用实例。用 if 语句可以比较两个变量的大小，可以用一个中间变量来实现两个数字的排列。用同样的方法，对三个数字每两个进行三次从小到大的排列，就可以实现三个数字的从小到大排列。这个程序的算法如下所示。

① 定义 4 个变量 x、y、z、t。

② 输出提示，然后用 scanf()函数分别从键盘读取 3 个数字，分别赋值给 x、y、z。

③ 取出 x 与 y 进行比较，如果 x 大于 y，则 x 赋值给 t，y 赋值给 x，t 再赋值给 y。三次赋值操作实现了 x 与 y 的交换。这一步骤的操作实现了 x 与 y 的从小到大排列。

④ 用与上一步骤相同的办法，实现 x 与 z 的从小到大排列，再实现 y 与 z 的从小到大排列。从而实现了 3 个数字的从小到大排列。

⑤ 输出排列的结果。

【范例 7-14】 3 个数字从小到大排列，示例代码如下所示。

示例代码 7-14

```
01  #include <stdio.h>
02  void main()
03  {
04    int x, y, z, t;                          /*定义 4 个变量。*/
05    printf("please input x:\n");             /*分别提示和输入 3 个变量。*/
06    scanf("%d", &x);
07    printf("please input y:\n");
08    scanf("%d", &y);
09    printf("please input x:\n");
10    scanf("%d", &z);
11    if(x > y)                                /*判断 x 是否大于 y。*/
12    {
```

```
13        t = x;                          /*3 次赋值操作实现 x 与 y 的交换。*/
14        x = y;
15        y = t;
16      }
17      if(x > z)                         /*用同样的方法实现 x 与 z 的排列。*/
18      {
19        t = x;
20        x = z;
21        z = t;
22      }
23      if(y > z)                         /*实现 y 与 z 的排列。*/
24      {
25        t = y;
26        y = z;
27        z = t;
28      }
29      printf("%d  %d  %d",x,y,z);       /*输出结果。*/
30  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 7.14.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序提示输入一个数字，结果如下所示。

```
please input x:
```

输入一个数字 5，然后按“Enter”键。在后面的提示中再输入两个数字 8 和 2。按“Enter”键以后，显示的结果如下所示。

```
2  5  8
```

7.4　小结

本章讲述了 C 程序中的流程控制语句。流程控制语句实现了程序的逻辑功能，可以完成各种复杂的程序运算，是编程开发的基础和重点。其中，循环语句和判断语句是本章的重点，需要理解这些语句的结构和运行流程。

7.5　习题

一、填空题

1. 流程控制语句有_____与_____两种。_____实现程序的逻辑功能，_____实现程序的重复执行功能。

2. _____指的是循环语句无法达到停止条件，永远执行下去，可能引起计算机的死机。

3. 在_____循环结构中，先判断循环的条件是否成立。如果条件成立则执行循环体中的语句，然后继续判断条件是否成立。如果条件不成立则跳出循环，执行后面的语句。

4. _____循环先执行一次循环体，然后判断条件是否成立。如果条件成立则执行下一次循环，否则结束循环。

5. 有时需要在判断、循环、程序执行过程中实现程序流程的强制转移，这就需要使用转移控制语句。C 程序中流程的强制转移有_____、_____和 _____。

6．C 语言中的三元操作符是_____。

7．_____是使用两个花括号（{}）括起来的一个语句集，形成一个语句体。

二、选择题

1．C 语言中的流程控制语句包括（　　）。

A．分支语句和循环语句

B．while 语句和 do while 语句

C．for 语句和 while 语句

D．for 语句和 if 语句

2．C 语言中的转移控制语句不包括（　　）。

A．continue

B．break

C．return

D．goto

3．下列关于 C 语言中的流程控制语句说法正确的是（　　）。

A．while 语句与 do while 语句完全相同

B．if 语句属于分支结构，if 语句不能嵌套使用，当条件较多时只能选择使用 switch 语句

C．使用 for 语句完全实现 while 语句的功能

D．switch 语句属于多重分支语句

三、简答题

1．简述 while 语句与 do while 语句的区别。

2．简述 continue、break 和 return 语句的作用域使用方法及区别。

3．简述 C 语言中的三元运算符及其使用方法。

四、编程题

1．编写程序实现输出九九乘法表（分别使用 while 和 for 循环语句实现）。

2．使用 if 语句实现三个数字的排序（用户从键盘随机输入三个数字）。

第四篇　指针与函数篇

第 8 章　数组与指针

数组在进行软件开发时的使用频率很高，指针则是 C 语言的精华所在，同时也是比较难以掌握的知识点。本章首先分别介绍数组和指针的基本概念和使用方法，然后介绍数组与指针之间的关系，最后通过实例介绍在实际开发中如何灵活地使用数组与指针。本章的主要内容如下：

- 数组与指针的概念与基本操作；
- 数组与指针的关系；
- 二维数组与多维数组；
- 指针与数组的常见问题。

8.1　数组的理解与操作

数组实际上就是一组相同数据类型的变量。如果一个数组中的一个变量也是一个数组，就构成了二维数组。同样的方法，可以构成多维数组。在使用数组以前需要建立一个数组。

8.1.1　什么是数组

可以用一个实例来理解数组。假设在一个程序中要存储 100 个人的年龄，可以写成下面的代码来定义 100 个整型变量。

```
int age1=12;
int age2=18;
int age3=15;
……
int age100=21;
```

如果这样编写程序，程序中会产生大量相似的语句，在调用变量时，变量名没有统一的规则。可以将所有同类的数据存放在一个数组变量中。所有人的年龄可以建立一个数组，然后用编号来表示数组的一个变量。可以用下面的代码来表示这些数据。

```
int a[100]
age[0]=12;
age[1]=18;
age[2]=15;
……
age[99]=21;
```

这样就可以用一个变量名和一个下标来表示数组中的一个变量。例如 age[0]表示数组中的第一个变量，age[1]表示数组中的第二个变量。age 是这个数组的变量名，后面的数字是这个数组的下标，下标需要使用中括号括起来。例如下面的程序，就是定义一个 10 个整型变量的数组，然后用循环语句对数组赋值 10~19。再用循环语句输出这些数据变量。

【范例 8-1】使用循环语句为一个包含 10 个元素的数组赋值，示例代码如下所示。

示例代码 8-1

```
01  #include <stdio.h>
02  void main()
```

```
03  {
04     int a[10];
05     int i;
06     for(i=0;i<=9;i++)
07     {
08        a[i]=i+10;
09     }
10     for(i=0;i<=9;i++)
11     {
12        printf("%d  ",a[i]);
13     }
14  }
```

【执行结果】编译并运行这个程序，结果如下所示。

```
10  11  12  13  14  15  16  17  18  19
```

【代码解析】示例代码 8-1 的代码解析如下：

- 代码第 04 行定义了一个具有 10 个元素的数组 a。
- 代码第 06~09 行使用 for 循环为数组元素赋值。
- 代码第 10~13 行输出数组 a 中所有元素的值。

在使用数组时，需要注意下面这些下标和数据类型的问题。

- 在定义数组时，已经定义了数组的数据类型，在访问时需要按照数组的数据类型进行访问和赋值。
- 数组的下标是从 0 开始的向后排列整数，不能是其他的下标。第一个下标是 0 而不是 1。而下标 n 表示第 n+1 个变量。
- 定义数组 a[n]以后，有 n 个元素，但是没有 a[n]这个元素。访问不存在的数组变量时，程序就会发生溢出错误。

注意：溢出指的是程序中的变量不够存储空间，或者访问了不存在的变量，从而使程序发生错误。

在本质上，数组是内存上一组同类信息列出的一个表。假设上面例子中的数组 a[10]是从内存编号为 1000 的存储单元开始存储的，数组的内容和值可用表 8.1 表示。

表 8.1　数组的存储与值

变量	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
值	10	11	12	13	14	15	16	17	18	19
存储单元	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009

8.1.2　数组的定义与访问

数组的定义指的是在内存中开辟一块存储空间，生成一个空数组。数组的定义与变量的定义相似，需要指定数据类型的变量的多少，变量数目用中括号括起来。例如下面的代码就是定义不同的数组。

```
01  int a[10];                  /*定义 10 个变量的整型数组。*/
02  float f[20];                /*定义有 20 个变量的浮点型数组。*/
03  char s[5];                  /*定义有 5 个变量的字符型数组。*/
```

提示：数组在定义时，可以不指定变量的个数，在访问变量时可以动态改变数组中变量的个数。例如下面的代码定义一个有不定变量的字符型数组。

```
01 char a[];                   /*定义一个含有不定个数变量的字符数组。*/
```

数组的访问很简单。用数组变量的名称与下标就可以访问这个数组。例如下面的代码输出数组元素的值和对数组进行赋值。

```
01  printf("%d",a[2]);      /*输出数组 a 中第 3 个元素的值。*/
02  a[3]=5;                 /*对数组中的第 4 个变量进行赋值。*/
```

8.1.3　数组使用实例

数组可以实现变量的循环访问功能，和循环语句一起用可以方便地实现数组的访问与输出。下面的实例，定义一个字符型数组，然后从键盘上输入字符存储到这个数组中。当输入一个“#”时停止输入。然后用循环语句输出这些字符。

【范例 8-2】将输入信息保存到数组中，遇到“#”停止输入，示例代码如下所示。

示例代码 8-2

```
01  #include <stdio.h>
02  void main()
03  {
04     char a[100];                    /*定义一个字符型数组。*/
05     char c='a';                     /*定义一个字符变量，赋一个初值。*/
06     int i=0 ;                       /*定义一个计数变量，实现下标的计数。*/
07     while(1)                        /*进入一个循环。*/
08     {
09       scanf("%c",&c);               /*输入一个字符。*/
10       if(c=='#')                    /*如果字符为#则中断循环。*/
11       {
12         break;                      /*结束循环。*/
13       }
14       a[i]=c;                       /*把输入的变量赋值给数组中的一个元素。*/
15       i++ ;                         /*计数变量自加。*/
16     }
17     i=0;                            /*输出时，计数变量赋值为 0。*/
18     while(a[i]!=NULL)
19     {
20       printf("%c",a[i]);            /*输出数组的中字符。*/
21       i++ ;                         /*下标自加。*/
22     }
23  }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.2.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序中输入一个字符，然后按“Enter”键。输入多个字符以后，输入“#”结束输入。程序会输出这些字符。

【代码解析】这个程序是用井号“＃”来结束循环输入的。每次输入一个字符，判断这个字符是不是“#”。如果是“#”则用 break 语句来中断循环。在输出时，判断输出的字符是不是存在。在定义数组时，数组中的字符都是空字符 NULL。

注意：由于数组定义后将无法修改数组的大小，所以，在定义数组时要充分考虑数组能够存储的元素个数。

8.2 指针

指针是一种特殊的数据类型，用来存储一个变量的内存地址。通过一个指针，可以访问这个指针的变量。在使用指针时，需要考虑到变量的存储关系。

8.2.1 指针的理解

程序中的变量都是以字节的形式存储在内存单元中的，这些内存单元都有一个编号。这个编号就是程序中的指针。如表 8.2 所示，是一个程序中的内存地址和变量值的关系。

在这个内存区域中，每一个变量都有一个内存存储单元编号，这个编号就是变量的指针。通过这个指针可以访问这个变量的值。

提示： 指针即内存地址，通过指针可以直接操作内存地址。

8.2.2 指针操作符

表 8.2 指针和变量

内存地址	内存中的变量
1005	1
1006	2
1007	3
1008	b
1009	c
1010	d

指针操作符有*与&两个，分别实现取变量和取地址的操作。程序中就是通过这两个操作符实现指针的定义与访问的。“&”可以实现取一个变量的地址的功能。取出的变量地址可能是一个很复杂的数据类型，但是操作时并不关心数值的多少，只需要保存到一个变量上面。例如下面的代码是取指针操作。

```
01  int *p;                 /*定义一个指针变量。*/
02  int i=5;                /*定义一个整变量。*/
03  p=&i;                   /*取变量 i 的地址赋值给 p。*/
```

“*”实现取一个指针所指向的变量的功能。例如下面的代码，就是通过一个变量的指针来访问变量。

```
01  int *p;                 /*定义一个指针变量。*/
02  int i=5 , j;            /*定义一个整变量。*/
03  p=&i;                   /*取变量 i 的地址赋值给 p。*/
04  j=*p;                   /*取指针 p 的值赋值给 j。*/
```

注意： 在定义指针变量时，*表示定义的变量为指针变量，而类似 j=*p 的操作中，*则表示取地址空间的变量值。

8.2.3 指针的定义与访问

指针虽然是一种特殊类型，但定义指针变量时，需要考虑到这个指针所指向变量的数据类型。定义指针的方法是用定义变量的方法，在指针类型名称前面加一个“*”。例如下面的代码是指向不同数据类型的指针。

```
01  int *p1;                        /*定义一个指向整型变量的指针。*/
02  char *p2;                       /*定义指向一个字符型变量的指针。*/
03  float *p3;                      /*定义一个指向浮点型变量的指针。*/
```

一个变量是用“*”取地址的。一个指针变量可以用“&”来取这个变量指向的数值。例

如下面的代码是使用指针访问变量的实例。

【范例 8-3】指针的使用案例，示例代码如下所示。

示例代码 8-3

```
01  #include <stdio.h>
02  void main()
03  {
04    int i ,j;                                  /*定义两个整型变量。*/
01    int *p;                                    /*定义一个指向整型的指针变量。*/
02    char a ,b;                                 /*定义两个字符变量。*/
03    char *q;                                   /*定义一个指向字符变量的指针。*/
04    i=5;                                       /*i 赋值为 5。*/
05    p=&i;                                      /*取 i 的地址赋值给 p。*/
06    j=*p;                                      /*取指针 p 指向的变量值赋值给 j。*/
07    a='w';                                     /*a 赋值为 w。*/
08    q = &a;                                    /*取 a 的地址赋值给指针变量 q。*/
09    b=*q;                                      /*取指针 q 指向的变量值赋值给 b。*/
10    printf("%d  %d  %d \n",i ,j ,*p);          /*输出结果。*/
11    printf("%c  %c  %c \n",a ,b ,*q);          /*输出结果。*/
12  }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.3.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
5  5  5
w  w  w
```

从结果可知，通过一个变量的指针，可以访问这个变量的值。在程序中，常常通过指针来访问一个变量的值。

【代码解析】示例代码 8-3 的代码解析如下：

- 代码第 01 行和第 03 行分别定义了两个指针变量。
- 代码第 05 行和第 08 行将变量的地址赋值给指针变量。
- 代码第 10 行和第 11 行输出指针变量和非指针变量的值。

8.2.4　指针使用实例

在程序中可以使用指针来访问变量的值。本节将讲述一个指针使用实例，通过指针所指向的变量比较两个整数的大小，然后从小到大输出两个整数。

【范例 8-4】通过指针所指向的变量比较整数的大小，示例代码如下所示。

示例代码 8-4

```
01  #include <stdio.h>
02  void main()
03  {
04    int i ,j;                                  /*定义两个整型变量。*/
05    int *p,*q,*temp;                           /*定义指向两个整型变量的指针变量。*/
06    printf("please input the first number: \n");      /*提示输入。*/
07    scanf("%d",&i);                                   /*输入一个数值。*/
08    printf("please input the second number: \n") ;
```

```
09      scanf("%d",&j);
10      p=&i;                                   /*取 i 的地址赋值给指针 p。*/
01      q=&j;                                   /*取 j 的址址赋值给指针 q。*/
02      if(*p>*q)                               /*判断这两个指针指向值的大小。*/
03      {                                       /*用一个中间指针变量交换两个指针。*/
04        temp=p;
05        p=q;
06        q=temp;
07      }
08      printf("%d  %d\n",*p,*q);               /*输出结果。*/
09    }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.4.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序运行时，输出提示，这时输入一个变量 5，然后按“Enter”键。然后再输入一个变量 3，然后按“Enter”键。这时程序显示的结果如下所示。

```
3  5
```

【代码解析】本程序是通过指针所指向的值来实现两个整数的排列的。通过比较指针所指向变量的大小，用一个中间指针变量交换两个指针变量的值。

8.3 数组与指针

数组与指针的联系非常紧密。除了可以用下标访问数组元素以外，也可以用数组的指针访问数组变量。本节将讲解数组与指针的关系。

8.3.1 数组与指针的关系

定义一个数组的时候，定义这个数组的头指针，然后分配若干个存储单元。定义的数组名称是可以直接赋值给一个指针的，而这个指针可以指向这个数组的第一个元素。下面的程序可以说明数组与指针的这种关系。

【范例 8-5】数组与指针的关系示例代码如下所示。

示例代码 8-5

```
01  #include <stdio.h>
02  void main()
03  {
04     int *p;                              /*定义一个指向整型的指针。*/
05     int a[3];                            /*定义一个整型数组。*/
06     a[0]=10;                             /*对数组的变量赋值。*/
07     a[1]=11;
08     a[2]=12;
09     p=a;                                 /*将数组赋值给一个指针。*/
10     printf("%d\n",*p);                   /*输出这个指针指向的内容。*/
11     p++;                                 /*指针向后移动一个单元。*/
12     printf("%d\n",*p);                   /*输出指针指向的内容。*/
13     p++;                                 /*指针向后移动一个单元。*/
14     printf("%d\n",*p);                   /*输出指针指向的内容。*/
15  }
```

【执行结果】编译并运行这个程序，结果如下所示。

```
10
11
12
```

从程序的结果可知，数组的定义就是返回了这个数组的指针。用指针来访问数组的元素与下标访问量等效。

【代码解析】

- 代码第 09 行将数组的首地址赋值给指针变量 p。
- 代码第 10 行、第 12 行和第 14 行分别输出数组的第 0 个元素的值、第 1 个元素的值和第 2 个元素的值。

提示： 数组名即为数组的头指针。

8.3.2　指针的算术运算

指针可以做加法和减法的算术运算，相当于指针的位置从这一位置向后或向前移动若干个单元。这里移动的单元，指的是可以存储相应的变量的内存空间，而不一定是这么多个字节。下面的实例可以说明指针算术运算的作用。

提示： 指针的加减法运算是基于指针所代表的存储单元为单位，而不是以具体的字节为单位。

【范例 8-6】指针的算术运算，示例代码如下所示。

示例代码 8-6

```
01  #include <stdio.h>
02  void main()
03  {
04     int i;                              /*定义一个循环变量 i。*/
05     int *p;                             /*定义一个指向整型变量的指针。*/
06     int a[10];                          /*定义一个整型的数组。*/
07     for(i=0;i<10;i++)                   /*变量 i 从 0~9 执行循环。*/
08     {
09        a[i]= i +10;                     /*对数组进行赋值。*/
10     }
11     p=a;                                /*将数组 a 的指针赋值给 p。*/
12     for(i=0;i<10;i++)                   /*进行 10 次循环。*/
13     {
14        printf("%d  ",*p);               /*输出指针 p 指向的值。*/
15        p++;                             /*指针 p 向后移动一个单元。*/
16     }
17     printf("\n",*p);                    /*输出一个换行。*/
18     p=p-4;                              /*指针向前移动 4 个单元。*/
19     printf("%d  ",*p);                  /*输出指针 p 指向的值。*/
20     p=p-3;                              /*指针向前移动 3 个单元。*/
21     printf("%d  ",*p);                  /*输出指针 p 指向的值。*/
22     p=p+5;                              /*指针向后移动 5 个单元。*/
23     printf("%d  ",*p);                  /*输出指针 p 指向的值。*/
24  }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.6.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
10  11  12  13  14  15  16  17  18  19
16  13  18
```

【代码解析】代码第 18 行、第 20 行和第 22 行通过指针的算术运算来访问对应的数组元素。这种访问数组元素的方式与使用下标的方式效果相同。

8.3.3 字符数组与字符串

字符数组是一种特殊的数组。定义一个字符数组以后，这个字符数组会返回一个头指针。可以根据这个头指针来访问数组中的每一个字符。而 scanf()输入函数和 printf()输出函数可以输入或输出一个字符串。下面的实例是字符数组的使用。

【范例 8-7】字符数组与指针的关系，示例代码如下所示。

示例代码 8-7

```
01  #include <stdio.h>
02  void main()
03  {
04    char a[30];                                  /*定义一个字符数组。*/
05    char *p;                                     /*定义一个指向字符变量的指针。*/
06    int i;                                       /*定义一个计数器变量。*/
07    printf("please input a string:\n") ;         /*提示输出。*/
08    scanf("%s",a) ;                              /*输入一个字符数组。*/
09    printf("result:\n") ;
10    printf("%s\n",a) ;            /*输出一个字符数组。*/
11    i=0;                          /*循环计数器赋初值。*/
12    while(a[i]!=NULL)             /*判断相对应的字符是不是为空。*/
13    {
14      printf("%c",a[i]) ;         /*输出这个字符。*/
15      i++;                        /*计数器自加。*/
16    }
17    printf("\n") ;                /*输出一个换行。*/
18    p=a;                          /*数组的头指针赋值给指针 p。*/
19    printf("%s\n",p) ;            /*输出变量 p 的内容。*/
20    while(*p!=NULL)               /*判断 p 指向的内容是不是为 NULL，进行 while 循环。*/
21    {
22      printf("%c",*p) ;           /*输出指针 p 指向的一个字符。*/
23      p++;                        /*指针 p 指向下一个变量。*/
24    }
25  }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序输出下面的提示，要求输入一个字符串。

```
please input a string:
```

这时输入下面的字符串，然后按“Enter”键。

```
asdfgh
```

程序的运行结果如下所示。

```
please input a string:
asdfgh
result:
asdfgh
asdfgh
asdfgh
asdfgh
```

【代码解析】上面的代码使用三种情况演示了如何访问数组中的元素。代码第 12~16 行使用数组下标访问数组，代码第 19 行使用字符串的方式打印字符数组中的所有元素，代码第 20~24 行使用指针访问数组元素。

注意：在 C 语言中没有字符串数据类型，对字符串的存储可以通过字符数组来实现。

8.4　二维数组与多维数组

如果数组的每一个元素是一个一维数组，则这个数组就是一个二维数组。在二维数组中，变量可以理解成行和列的关系。用同样的方法，可以构造出多维数组。

8.4.1　二维数组的理解

二维数组可以理解成一个多行多列的表格，每一个单元格中存储了一个变量。可以同一维数组一样定义二维数组。下面的代码定义了一个 4 行 5 行的整型数组。

```
01   int a[4][5];
```

可用表 8.3 来表示这个数组的行和列存储关系。

表 8.3　二维数组

行 和 列	0	1	2	3	4
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

根据这个表格，可以用数组的下标访问每一个元素。下面的代码使用 for 循环对这个二维数组进行赋值与输出。

提示：二维数组可以认为是数组的数组，即一维数组的元素又是一个一维数组。

【范例 8-8】使用 for 语句为二维数组赋值并输出，示例代码如下所示。

示例代码 8-8

```
01  #include <stdio.h>
02  void main()
03  {
04      int i=10 , m , n ;                    /*定义程序中的变量。*/
05      int a[4][5];                          /*定义一个四行五列的二维数组。*/
```

```
06      for(m=0;m<4;m++)                                    /*用变量 m 从 0 到 3 进行行循环。*/
07      {
08         for(n=0;n<5;n++)                                 /*变量 n 从 0 到 4 进行列循环。*/
09         {
10            a[m][n]=i;                                    /*数组中变量赋值。*/
11            i++;                                          /*i 自加。*/
12         }
13      }
14      for(m=0;m<4;m++)                                    /*变量 m 的行循环。*/
15      {
16         for(n=0;n<5;n++)                                 /*变量 n 的列循环。*/
17         {
18           printf("a[%d][%d]=%d  ",m,n,a[m][n]);          /*输出变量名变量值。*/
19         }
20         printf("\n");                                    /*每行输出一个换行。*/
21      }
22   }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.8.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a[0][0]=10  a[0][1]=11  a[0][2]=12  a[0][3]=13  a[0][4]=14
a[1][0]=15  a[1][1]=16  a[1][2]=17  a[1][3]=18  a[1][4]=19
a[2][0]=20  a[2][1]=21  a[2][2]=22  a[2][3]=23  a[2][4]=24
a[3][0]=25  a[3][1]=26  a[3][2]=27  a[3][3]=28  a[3][4]=29
```

【代码解析】代码第 05 行定义二维数组，代码第 06~13 行用 for 循环的嵌套实现为二维数组元素赋值，代码第 14~22 行用 for 循环的嵌套打印二维数组的元素值。

8.4.2 二维数组与指针

同一维数组一样，二维数组在定义时也返回了一个指向第一个元素的指针。而指针向后移动存储单元，可以访问数组中的元素。下面的代码定义了一个二维数组，然后将二维数组的头指针赋值给一个指针变量。

```
01   int a[4][5];
02   int *p;
03   p=a;
```

根据数组与指针的关系，可以用表 8.4 中的指针变量来访问数组中的所有变量。

表 8.4 二维数组与指针

行和列	0	1	2	3	4
0	p	p+1	p+2	p+3	p+4
1	p+5+0	p+5+1	p+5+2	p+5+3	p+5+4
2	p+5+5+0	p+5+5+1	p+5+5+2	p+5+5+3	p+5+5+4
3	p+15+0	p+15+1	p+15+2	p+15+3	p+15+4

从表 8.4 可知，数组中变量的指针地址与首指针存在着下面的关系。

```
q=p+(行数*总列数)+列数
```

根据这个关系，可以用下面的程序用指针的方法来访问一个数组。

【范例 8-9】使用指针访问二维数组元素，示例代码如下所示。

示例代码 8-9　\源文件\08\8.9.c

```
#include <stdio.h>
void main()
{
    int i=10 , m , n ;                                  /*定义 3 个变量。*/
    int a[4][5];                                        /*定义 1 个 4 行 5 列的二维数组。*/
    int *p;                                             /*定义一个指针。*/
    for(m=0;m<4;m++)                                    /*行的循环。*/
    {
      for(n=0;n<5;n++)                                  /*列的循环。*/
      {
        a[m][n]=i;                                      /*对数组变量进行赋值。*/
        i++;                                            /*i 自加。*/
      }
    }
    p=a;                                                /*把数组的头指针赋值给 p。*/
    for(m=0;m<4;m++)                                    /*行循环。*/
    {
      for(n=0;n<5;n++)                                  /*列循环。*/
      {
        printf("a[%d][%d]=%d  ",m,n,*(p+(5*m)+n));/*用指针来访问数组的变量。*/
      }
      printf("\n");                                     /*每行输出一个换行。*/
    }
}
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.9.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a[0][0]=10  a[0][1]=11  a[0][2]=12  a[0][3]=13  a[0][4]=14
a[1][0]=15  a[1][1]=16  a[1][2]=17  a[1][3]=18  a[1][4]=19
a[2][0]=20  a[2][1]=21  a[2][2]=22  a[2][3]=23  a[2][4]=24
a[3][0]=25  a[3][1]=26  a[3][2]=27  a[3][3]=28  a[3][4]=29
```

从程序的结果可知，用指针来访问数组，与用下标访问数组的效果是一样的。

【代码解析】代码第 20 行中表达式“*(p+5*m)+n)”的功能是计算数组元素的地址，“p”数组的首地址，“5*m”为一个含有 5 个元素的数组所占用的内存但与数目，则“p+{5*m}”可以计算出第 *m* 行的首地址，再加上 *n* 则表示第 *m* 行的第 *n* 个元素。

8.5　实例

数组可以在程序中实现数据存储，程序可以使用循环的方法对数组进行访问。本节将讲解数组与指针的编程使用实例。

8.5.1　学生成绩统计实例

本节编写一个成绩输入与统计程序。在程序中输入 6 个人的姓名、各科成绩。然后程序输出每个人的各科成绩、总成绩和平均成绩。程序的代码如下所示。

【范例 8-10】学生成绩统计程序，示例代码如下所示。

示例代码 8-10

```
01  #include <stdio.h>
02  void main()
03  {
04     char name[6][10];                              /*定义 6 个人的姓名，6 个字符数组。*/
05     int score[6][5];                               /*定义一个二维数组存放成绩。*/
06     int i;                                         /*定义一个计数器变量。*/
07     for(i=0;i<6;i++)                               /*进行 6 次循环。*/
08     {
09       printf("please input the name:\n");          /*提示输入姓名。*/
10       scanf("%s",name[i]);                         /*输入一个人的姓名。*/
11       printf("score ,math:\n");
12       scanf("%d",&score[i][0]);                    /*输入数学成绩。*/
13       printf("score ,English:\n");
14       scanf("%d",&score[i][1]);                    /*输入外语成绩。*/
15       printf("score ,Chinese:\n");
16       scanf("%d",&score[i][2]);                    /*输入汉语成绩。*/
17       score[i][3] = score[i][0]+score[i][1]+score[i][2]; /*求出这个人的总分。*/
18       score[i][4]= (int)(score[i][3] /3);          /*求出这个人的平均分。*/
19     }
20      printf("result:\n math  English Chinese  total  average :\n");
21     for(i=0;i<6;i++)                               /*循环输出成绩。*/
22     {
23      printf("%s: %d  %d   %d   %d  %d \n",name[i],score[i][0],score[i][1],
score[i][2],
24   score[i][3],score[i][4]);                        /*输出一个人各科的成绩。*/
25     }
26  }
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.10.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序运行时，在文本提示后面输入每个人的姓名和各科分数。程序显示的结果如下所示。

```
math  English Chinese  total  average :
jim: 45   76    87    208    69
tom: 56   76    45    177    59
bill: 56   76    87    219    73
lucy: 45   65    87    197    65
bod: 56   76    46    178    59
lily: 34   65    67    166    55
```

【代码解析】示例代码 8-9 的代码示例如下：

代码第 05 行定义了一个 6 乘 5 的二维数组，共 6 行（每行表示一个学生的各科成绩），每行 5 个元素（分别表示前三门课的成绩、总成绩和平均成绩）。

8.5.2 冒泡法排序实例

冒泡法排序实例是 C 语言中的一个经典算法，实现多个数值的排序。方法是多次循环进行比较，将大数。每次循环时，找出剩余变量里的最大值，然后减小查询范围。这样经过多次循环以后，就完成了对这个数组的排序。这种排序的算法可用图 8-1 来表示。

冒泡法排序使用了反复循环和比较的算法，执行了下面这些步骤。

① 在第一次循环时，拿第 1 个数与第 2 个数进行比较，如果第 1 个数小于第 2 个数，就

用一个中间变量使这两个数交换。这样就使第 1 个和第 2 个数从大到小排列。

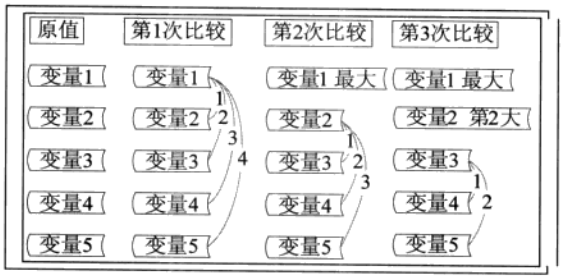


图 8-1　冒泡法排序的算法

② 再用第 1 个数与第 3 个数比较，使这两个数从大到小排列。用同样的方法，用第 1 个数与后面所有的数进行比较。

③ 经过了第 1 轮循环比较以后，第 1 个数一定是所有数里面最大的数。

④ 进行第 2 轮比较。这时让第 2 个数与后面所有的数比较，使用这个数是这些数里面的最大数。

⑤ 用循环的方法，依次用后面的一个数与这个数后面的所有数进行比较，这样就完成了这些数的从大到小排序。

【范例 8–11】用 for 语句实现冒泡法排序。下面的程序是使用冒泡法排序的例子。在这个程序中，需要注意两次循环比较。

示例代码 8-11

```
#include <stdio.h>
main()
{
   int a[10];                          /*定义一个整型数组。*/
   int i,j,temp;                       /*定义循环变量和中间变量。*/
   for(i=0;i<10;i++)                   /*进行循环输入变量。*/
   {
      printf("please enter a number:\n");     /*输出提示。*/
      scanf("%d",&a[i]);               /*输入变量赋值给数组变量。*/
   }
   for(i=0;i<10;i++)                   /*进行 10 次循环。*/
   {
      for(j=i+1;j<10;j++)              /*循环比较剩余的变量。*/
      {
         if(a[i]<a[j])                 /*如果前面一个数比后面数小，交换两个数的值。*/
         {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
         }
      }
   }
   for(i=0;i<10;i++)                   /*循环输出排序以后的结果。*/
   {
      printf("%d  ",a[i]);
   }
}
```

【执行结果】编译并运行这一个程序，根据程序的提示输入 10 个变量，程序会对这些变量

进行排序，然后输出排序以后的变量。运行结果如下所示：

```
please enter a number:
9
please enter a number:
10
please enter a number:
3
please enter a number:
4
please enter a number:
8
please enter a number:
6
please enter a number:
0
please enter a number:
1
please enter a number:
7
please enter a number:
6
0  1  3  4  6  6  7  8  9  10
```

【代码解析】

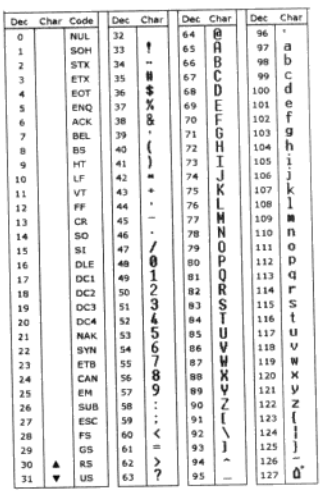
- 代码第 11~22 行使用冒泡法进行排序。读者需要仔细体会冒泡法排序的算法实现原理。
- 代码第 23~26 行打印排序后的结果。

8.5.3 统计字符串中字符

本节将讲解一个统计字符串中字符的例子。文本中的字符串，是由不同的大写字母、小写字母、标点符号等不同字符构成的。在程序中可以把文本存储在字符串中，然后分别统计出文本中的各种字符个数然后输出。为了识别这些字符，需要学习 ASCII 码。计算机中的所有字符，都是以 ASCII 码的形式保存在计算机中的。每一个 ASCII 码在内存中占据一个字节。ASCII 码与相对应的序号如图 8-2 所示。

字符与整型变量是等同的，这就很容易把字符与整型变量联系起来，通过序号的比较可以判断这些字符的种类。从图 8-2 可知，字符与序号的分布可以总结成下面的分段。

- 48-57：数字字符。
- 65-90：大写字母。
- 97-122：小写字母。
- 其他字符：特殊符号或标点符号。



Dec	Char	Code
0		NUL
1		SOH
2		STX
3		ETX
4		EOT
5		ENQ
6		ACK
7		BEL
8		BS
9		HT
10		LF
11		VT
12		FF
13		CR
14		SO
15		SI
16		DLE
17		DC1
18		DC2
19		DC3
20		DC4
21		NAK
22		SYN
23		ETB
24		CAN
25		EM
26		SUB
27		ESC
28		FS
29		GS
30	▲	RS
31	▼	US

Dec	Char
32	
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?

Dec	Char
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_

Dec	Char
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}
126	~
127	⌂

图 8-2　ASCII 码表

要完成字符的统计，需要在程序中完成下面这些步骤。

① 定义一个字符数组，用来存储字符串。
② 定义相关的计数变量。变量要赋初值为 0。
③ 从键盘输入一个字符串，这一操作可以用 scanf()函数实现。
④ 用循环的方法，访问字符串中的每一个字符。
⑤ 对每一个字符判断字符的种类，然后改变计数的值。
⑥ 用 printf()函数输出统计结果。

根据这些步骤，编写的程序代码如下所示。

【范例 8-12】统计输入的字符，示例代码如下所示。

示例代码 8-12

```
#include <stdio.h>
int main()
{
  char a[100];                                  /*定义一个字符数组保存字符串。*/
  int i,uper,lower,marks,num;                   /*定义计数的变量。*/
  i=0;                                          /*计数变量赋初值为 0。*/
  uper=0;
  lower=0;
  marks=0;
  num=0;
  printf("please input a string:\n") ;          /*提示输入信息。*/
  scanf("%s",a);                                /*输入一个字符串。*/
  while(a[i]!=NULL)                             /*循环访问字符串中的第一个字符。*/
  {
   if(a[i]<=57&&a[i]>=48)                       /*判断是否是数字。*/
   {
     num++;                                     /*是数字则数字计数加 1。*/
   }
   else                                         /*其他的可能。*/
   {
     if(a[i]<=90&&a[i]>=65)                     /*判断是否是大写字母。*/
     {
       uper++;                                  /*是小写字母则大写字母的记数加 1。*/
     }
     else                                       /*其他的情况。*/
     {
       if(a[i]<=122&&a[i]>=97)                  /*判断是否是小写字母。*/
       {
         lower++;                               /*是小写字母则小写字母的记数加 1。*/
       }
       else
       {
         marks++;                               /*其他的情况一定是特殊字符加 1。*/
       }
     }
   }
   i++;                                         /*字符总数加 1，进行下一次循环。*/
  }
  printf("total chars: %d \n",i) ;              /*输出总字符数。*/
  printf("uper chars: %d \n",uper) ;            /*输出大写字符数。*/
  printf("lower chars: %d \n",lower) ;          /*输出小写字符数。*/
  printf("number chars: %d \n",num) ;           /*输出数字字符数。*/
  printf("other chars: %d \n",marks) ;          /*输出其他字符数。*/
}
```

【执行结果】输入下面的命令，编译这个程序。

```
gcc 8.12.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序提示输入一个字符串，这时输入下面的字符串，然后按“Enter”键。需要注意的是，字符串中不要有空格。

```
asdfghJKLMNQWERT123789123;',./ASDJjhjad
```

程序显示出的统计结果如下所示。

```
total chars: 39
uper chars: 14
lower chars: 11
number chars: 9
other chars: 5
```

【代码解析】代码第 15 行、第 21 行和第 27 行判断输入的字符类型，它们是利用字符在计算机内保存的是其 ASCII 值的特点来实现的。

8.6 小结

数组与指针的知识是 C 程序中的重要内容，使用它们可以实现很多复杂的功能和算法。在数组与指针的知识中，这两个概念的理解是个难点，需要建立起直观形象的空间概念。在实际开发时，各种对象与算法常常是通过指针与数组建立起来的，需要使用各种复杂的指针和数组。在程序设计练习时，需要注意数组与指针的应用技巧。

8.7 习题

一、填空题

1. _____实际上就是一组相同数据类型变量。
2. 如果一个数组中的一个变量也是一个数组，就构成了_____。
3. _____是一种特殊的数据类型，用来存储一个变量的内容地址。
4. 定义一个数组的时候，就是定义这个数组的_____，然后分配若干个存储单元。定义的数组名称是可以直接赋值给一个_____的，而这个_____可以指向这个数组的第一个元素。
5. _____可以理解成一个多行多列的表格，每一个单元格中存储了一个变量。

二、选择题

1. 下列关于数组的说法中错误的是（　　）。
   A. 数组实际上就是一组相同数据类型变量。如果一个数组中的一个变量也是一个数组，就构成了二维数组
   B. 数组的定义指的是在内存中开辟一块存储空间，生成一个空数组
   C. 数组的定义与变量的定义相似，需要指定数据类型的变量的多少，变量数目用中括号括起来
   D. 数组的下标从 1 开始
2. 下列关于指针的说法中错误的是（　　）。
   A. 程序中的变量都是以字节的形式存储在内存单元中的，这些内存单元都有一个编号。这个编号就是程序中的指针
   B. 指针操作符有*与&两个，分别实现取变量和取地址的操作。程序中就是通过这两个操作符实现指针的定义与访问的
   C. 指针虽然是一种特殊类型，但定义指针变量时，需要考虑到这个指针所指向变量的数据类型。定义指针的方法是用定义变量的方法，在指针类型名称前面加一个“&”

D．一个变量是用“*”取地址的。一个指针变量可以用“&”来取这个变量指向的数值

3．下列关于数组与指针的关系的说法中错误的是（　　）。

A．定义一个数组的时候，就是定义这个数组的头指针，然后分配若干个存储单元

B．定义的数组名称是可以直接赋值给一个指针的，而这个指针默认指向这个数组的最后一个元素

C．指针的加减法运算是基于指针所代表的存储单元为单位，而不是以具体的字节为单位

D．字符数组是一种特殊的数组。定义一个字符数组以后，这个字符数组会返回一个头指针。可以根据这个头指针来访问数组中的每一个字符

三、简答题

1．什么是数组？数组变量如何定义与访问？

2．什么是指针？指针变量如何定义与访问？

3．简述数组与指针的关系。

四、编程题

1．编写一个成绩输入与统计程序。在程序中输入 5 个人的姓名、各科成绩。然后程序输出每个人的各科成绩、总成绩和平均成绩。

2．编写小写字母转换的程序。在一个数组中输入一个字符串，把字符串中的小写字母转换成大写字母，而原有的大写字母和特殊字符保存不变。

第9章 函 数

C语言被称为“函数式的编程语言”，在C语言中所有的功能都是通过函数来实现的。开发者需要调用众多的库函数和自定义各种各样的函数以实现特定的功能。本章介绍C语言中函数的相关概念及如何定义新的函数和使用库函数，主要内容如下：

- 函数的理解；
- 自定义函数；
- 函数与指针的关系；
- 函数的返回值；

9.1 函数的理解

在程序中，各种功能都是通过函数实现的。在学习函数之前，需要理解函数的作用与特点。一个函数可能有返回值，可能有不同的参数，完成不同的功能。本节将讲解这些函数的基本概念。

9.1.1 什么是函数

简单地说，函数就是把一个程序功能封装成一个整体。函数由类型名、函数名、参数列表、函数体等部分组成，一般形式如下所示。

```
类型名 函数名(参数列表)
{
   函数体;
}
```

函数这些部分的功能如下所示。

- 类型名：函数一般会返回一个数据，这个数据就是函数返回值。返回值的数据类型就是函数的数据类型。函数的类型可以是所有的数据类型。如果没有返回值，则声明函数的类型为void。
- 函数名：函数的名称。函数名的规则与变量命令的规则是相同的，函数名需要有效。
- 参数列表：函数输入的参数，函数需要使用这些参数进行运算。
- 函数体：完成函数运算功能的程序。在函数体中可以调用其他的函数。

例如下面是一个求出两个数中最大数的函数。

```
01  int max(int x ,int y)          /*定义一个函数。*/
02  {
03    if(x>y)                      /*如果x大于y就返回x。*/
04    {
05      return(x);
06    }
07    else                         /*否则就返回y。*/
08    {
09      return(y);
10    }
```

```
11  }
```

这个函数可以求出两个数中最大的一个数，函数的各部分作用如下所示。

- int 是函数的类型名，表示这个函数可以返回一个整型变量。
- 函数的名称是 max，用这个名称可以调用这个函数。
- 参数列表是“int x ,int y”，表示这个函数有 x 和 y 两个整型参数，函数体可以调用这两个参数进行程序运算。
- 函数体中判断两个参数的大小，用 return 语句返回较大的一个变量。

例如下面的程序，是用这个函数判断两个数的大小。

【范例 9-1】自定义函数比较两个数的大小，示例代码如下所示。

示例代码 9-1

```
01  #include <stdio.h>
02  int main()
03  {
04     int a,b,s;                                  /*定义一个变量。*/
05     printf("please input a:\n");                /*提示输入。*/
06     scanf("%d",&a);                             /*输入一个变量。*/
07     printf("please input b:\n");                /*提示输入。*/
08     scanf("%d",&b);                             /*输入另一个变量。*/
09     s=max(a,b);                                 /*调用函数 max 求得较大的一个数。*/
10     printf("the max number is %d\n",s);         /*输出结果。*/
11  }
12  int max(int x ,int y)                          /*定义一个函数。*/
13  {
14     if(x>y)                                     /*如果 x 大于 y 就返回 x。*/
15     {
16        return(x);
17     }
18     else                                        /*否则就返回 y。*/
19     {
20        return(y);
21     }
22  }
```

【运行结果】用下面的命令编译这段代码。

```
gcc 9.1.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序提示输入数字，输入一个数字 5，再按“Enter”键。然后用同样的方法输入另一个数字 1。程序的运行结果如下所示。

```
the max number is 5
```

【代码解析】代码第 12~22 行定义了函数 max，代码第 09 行调用自定义函数的功能实现数字的比较并输出较大的数字。

9.1.2　系统函数与库函数

按照函数的来源，可以将函数分为库函数与用户自定义函数。系统函数是编程平台提供的函数。用户自定义函数是用户自己按编程的需要编写的函数。这两种函数的作用与区别如下所示。

- 库函数：由系统提供，用户无须定义，也不必在程序中声明的函数。在程序需要使用库函数时，需在程序前包含该函数的头文件。在前面章节里反复使用的 printf、scanf

等函数都是库函数。

- 用户定义函数：用户按需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。上一节的函数 max()就是一个用户自定义函数。

提示：在进行程序设计时，要善于使用系统的库函数，避免自己编写功能重复的函数，以提高开发效率。

9.1.3 函数的返回值

所谓返回值，指的是函数运行以后会产生一个数据，这个数据用 return 语句返回给主程序。有些函数是没有返回值的，只执行了一个运行或操作过程。从这个角度，可以把函数分为有返回值函数和无返回值函数两种。

1. 有返回值函数

这种函数被调用执行以后，会向调用语句返回一个执行结果，这个结果就是返回值。这种函数在定义时，需要指定函数的类型名。调用函数时，需要有一个变量来接收函数的返回值。例如 9.1.1 节中的下列语句。

```
01   s=max(a,b);
```

函数 max 会返回一个整型的结果，这个结果需要赋值给变量 m。

2. 无返回值函数

此类函数用于完成一定的处理过程，执行完成后不向调用者返回结果。由于函数没有返回值，用户在定义此类函数时需要指定返回为“空类型”。空类型的说明符为“void”。例如下面的自定义函数，就是用一个自定义函数输出 n 个指定字符。

【范例 9-2】使用无返回值的函数，示例代码如下所示。

示例代码 9-2

```
01  #include <stdio.h>
02  void show(char a ,int x)                    /*定义一个无返回值函数。*/
03  {
04     int i;                                   /*定义一个变量作为计数器。*/
05     for(i=0;i<x;i++)                         /*for 循环输出字符。*/
06     {
07        printf("%c",a);                       /*输出这个字符。*/
08     }
09  }
10
11  int main()                                  /*主函数。*/
12  {
13     char c='a';                              /*定义一个变量 c，赋值为 a。*/
14     show(c,6)  ;                             /*调用函数，将变量 c 输出 6 次。*/
15     printf("\n") ;                           /*输出一个换行。*/
16     show('*',9)  ;                           /*调用函数，将星号输出 9 次。*/
17  }
```

【运行结果】用下面的命令编译这段代码。

```
gcc 9.2.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
aaaaaa
******
```

【代码解析】代码中首先定义 show 函数，然后在 main 函数中调用 show 函数的功能。

9.1.4　无参函数

参数指的是调用语句向被调用函数传送的数据。从调用语句和被调用函数之间数据传送的角度，又可将函数分为无参函数和有参函数两种。无参函数指的是函数在定义、函数说明及函数调用中均不带参数。调用语句和被调用函数之间不进行参数传送。

提示：无参函数通常用来完成一组指定的功能，不需要给函数传递数据，但是函数可以向调用者返回一个结果。

例如下面的程序中使用一个无参函数输出一行文本。

【范例 9-3】使用无参函数打印一行文本，示例代码如下所示。

示例代码 9-3

```
01  #include <stdio.h>
02
03  void hello()                          /*定义一个无参函数。*/
04  {
05    printf("good morning .\n") ;        /*函数中输出一行文本。*/
06  }
07
08  int main()                            /*主函数。*/
09  {
10    hello();                            /*调用函数输出信息，调用时没有参数。*/
11    hello();                            /*再次调用这个函数。*/
12  }
```

【运行结果】用下面的命令编译这段代码。

```
gcc 9.3.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
good morning .
good morning .
```

【代码解析】代码第 03~06 行定义了一个无参数的函数 hello。在主函数中两次调用 hello 函数。

9.1.5　有参函数

与无参函数相对应的是有参函数。有参函数在定义、调用、声明时都有参数。有参函数也称为带参函数。在使用这些参数时，应该注意以下内容。

- 在定义函数时，每一个参数都有一个具体的数据类型。
- 在调用函数时，参数列表中的参数要与函数定义时的参数个数相同，所有的参数一一对应。所给的参数数值应该有效。如果调用时，所给的参数不合法，或者类型不匹配、个数不相同，程序就会发生错误。

函数定义时的参数被称为形式参数，只在函数体中有效。函数调用时的参数被称为实际参数，在主程序中是有效的变量。

提示：形式参数仅在定义函数时有用，在程序运行时，形式参数将由实际参数代替。

9.1.6 函数参数实例

本节讲解一个函数使用实例。自定义函数有 4 个变量，前 3 个变量分别为整型数字，第 4 个变量为一个字符。函数对 3 个变量进行从小到大排列，然后用给定的字符连接输出。前面的章节已经讲解过 3 个数字排序的问题。程序的代码如下所示。

【范例 9-4】函数参数的使用，示例代码如下所示。

示例代码 9-4

```
#include <stdio.h>
void order(int a ,int b,int c,char s)        /*定义一个函数，有 4 个参数。*/
{
   int temp;                                 /*函数中定义一个中间变量。*/
   if(a>b)                                   /*如果 a 大于 b，则交换 a 与 b 的值。*/
   {
      temp=a;                                /*a 的值给一个中间变量。*/
      a=b;                                   /*b 赋值给 a。*/
      b=temp;                                /*中间变量的值赋值给 b。*/
   }
   if(a>c)                                   /*如果 a 大于 c，则交换 a 与 c 的值。*/
   {
      temp=a;
      a=c;
      c=temp;
   }
   if(b>c)                                   /*如果 b 大于 c，则交换 b 与 c 的值。*/
   {
      temp=b;
      b=c;
      c=temp;
   }
  printf("%d%c%d%c%d",a, s, b, s, c); /*输出排序以后的值，并用参数中的字符 s 连接。*/
}

int main()
{
   order(5,4,7,'-');                    /*调用函数。*/
   printf("\n");                        /*输出一个换行。*/
   order(7,55,32,'<');
   printf("\n");
   order(9,3,6,'.');
}
```

【运行结果】用下面的命令编译这段代码。

```
gcc 9.4c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
4-5-7
```

```
7<32<55
3.6.9
```

【代码解析】示例代码 9-4 中的自定义函数 order 的类型如下所示。

```
01   void order(int a ,int b,int c,char s)
```

这个函数没有返回值，有 4 个参数。其中前 3 个参数是整型，后 1 个参数是字符类型。所以需要用下面的语句来调用这个函数。调用时，参数列表中的参数需要与定义的参数列表相同。

```
01   order(5,4,7,'-');
02   order(7,55,32,'<');
03   order(9,3,6,'.');
```

9.2　自定义函数

在程序中需要把各种模块封装为自定义函数，在使用这个函数时，可以不考虑函数的内部执行过程。这样可以使用简化程序的设计，使代码和模块得到有效的重用。

提示：C 语言又称为函数式的编程语言，除了应用系统的众多库函数外，经常还要定义满足自己需求的自定义函数，函数构成了 C 程序的主体。

9.2.1　自编写定义函数

编写复杂程序时，需要将不同的功能分解为多个函数，这样可以简化程序的执行过程。在编写自定义函数时，需要考虑函数的参数与返回值。例如要编写一个分数等级判断的函数。用户从键盘输入一个分数，程序调用一个函数来判断成绩的等级，并返回表示等级的字符。当输入数字 0 时，退出这个程序。需要用下面的方法来实现这个程序。

（1）需要用 while 循环语句，构成一个无限循环。

（2）输入一个数值，判断这个数值是不是 0。如果数值为 0，则进行判断和循环，则否中止循环。

（3）如果数值不为 0，则调用函数判断成绩。成绩作为函数调用的参数，返回一个字符。

（4）自定义函数可以用 if 语句的嵌套结构来实现分数等级的判断。

根据这个思路，编写出的程序如下所示。

【范例 9-5】自定义函数实现判断分数等级，示例代码如下所示。

示例代码 9-5

```
01  #include <stdio.h>
02  char result(int x)                          /*定义一个分数评定函数。*/
03  {
04    char s;                                   /*定义一个字符变量存储结果。*/
05    if(x>=90)                                 /*90 分以上的结果为 A。*/
06    {
07      s='A';
08    }
09    else                                      /*其他的值 80 分以上的结果为 B。*/
10    {
11      if(x>=80)
12      {
13        s='B';
14      }
15      else
```

```
16        {
17          if(x>=70)                          /*其他的值 70 分以上的结果为 C。*/
18          {
19            s='C';
20          }
21          else
22          {
23            if(x>=60)                        /*其他的值 60 分以上的结果为 D。*/
24            {
25              s='D';
26            }
27            else                             /*其他的值一定小于 60，结果为 E。*/
28            {
29              s='E';
30            }
31          }
32        }
33      }
34      return(s);                             /*将变量 s 作为返回值返回。*/
35    }
36    int main()                               /*主函数。*/
37    {
38      char r;                                /*定义一个存放结果的变量。*/
39      int i=1 ;                              /*定义存放分数的变量，赋初值为 1。*/
40      while(i!=0)                            /*i 的值不为 0 则提示输入和判断结果。*/
41      {
42        scanf("%d",&i);                      /*输入一个数值。*/
43        r=result(i);                         /*调用函数判断等级，返回值赋值给 r。*/
44        printf("%c\n",r);                    /*输出结果。*/
45      }
46    }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.5c
```

输入下面的命令运行这个程序。

```
./a.out
```

在程序中输入一个数字 55，然后按“Enter”键。程序会输出一个结果 E。输入相应的数值，程序能正确输出判断结果。输入数字 0 以后，程序结束。

【代码解析】代码第 02~33 行定义分数评级函数 result，当用户输入一个分数并按“Enter”键确认后，立即打印分数等级。在主函数中使用 while 循环调用 result 函数，实现分数评级，如果用户输入的数字为 0，则退出主程序。

9.2.2 函数中调用函数

在自定义函数中，可以调用另外一个自定义函数。这样，就可以把复杂的程序功能分解为多个简单的函数。例如上一节中的程序，没有对输入值的有效性进行判断。可以进行分数评级之前，用一个函数判断输入的值是否合理。然后这个函数再调用分数评级函数 result。程序的代码如下所示。

【范例 9-6】函数中调用函数实例，示例代码如下所示。

示例代码 9-6

```
01  #include <stdio.h>
02  char result(int x)                         /*定义一个分数评定函数。*/
```

```
{
   char s;                    /*定义一个字符变量存储结果。*/
   if(x>=90)                  /*90 分以上的结果为 A。*/
   {
     s='A';
   }
   else                       /*其他的值 80 分以上的结果为 B。*/
   {
      if(x>=80)
      {
        s='B';
      }
      else
      {
        if(x>=70)             /*其他的值 70 分以上的结果为 C。*/
        {
          s='C';
        }
        else
        {
          if(x>=60)           /*其他的值 60 分以上的结果为 D。*/
          {
            s='D';
          }
          else                /*其他的值一定小于 60，结果为 E。*/
          {
            s='E';
          }
        }
      }
   }
   return(s);                 /*将变量 s 作为返回值返回。*/
}
void myresult(int x)          /*定义一个有数值有效性判断功能的函数，无返回值。
*/
{
   char s;                    /*存放结果的变量。*/
   if(x>0&&x<=100)            /*数值在这个范围内有效。*/
   {
      s=result(x);            /*调用函数 result()判断分数等级。*/
      printf("%c\n",s);       /*输出结果。*/
   }
   else                       /*其他值的分数。*/
   {
      printf("error.\n",s); /*输出错误提示。*/
   }
}
int main()                    /*主函数。*/
{
   int i=1 ;                  /*定义存放分数的变量，赋初值为 1。*/
   while(i!=0)                /*i 的值不为 0 则提示输入和判断结果。*/
   {
     scanf("%d",&i);          /*输入一个数值。*/
     myresult(i);             /*调用函数判断等级，返回值赋值给 r。*/
   }
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.6c
```

输入下面的命令运行这个程序。

```
./a.out
```

在程序中输入一个数字 123，然后按“Enter”键。程序会输出提示“error.”。输入 1~100 的数值，程序会判断并显示出这个分数的级别。输入数值 0，程序结束运行。

【代码解析】

- 代码第 02~35 行定义分数评级函数 result。
- 代码第 36~49 行定义函数 myresult，在 myresult 函数内部又调用了自定义函数 result。
- 主函数中通过 while 循环来调用 myresult 函数，实现分数等级的判断。如果用户输入的值为 0，则退出循环，程序结束运行。

提示：在 C 程序中所有的功能都封装在函数内，整个程序的功能就是在函数中调用其他函数来实现的。读者必须习惯 C 语言的这种函数式编程风格。

9.2.3 函数的声明

在 C 程序中，自定义函数需要写在被调用函数的前面。如果前面的函数调用了后面的函数，程序就会发生错误，提示程序中找不到相应的函数。如果需要调用后面的自定义函数，需要在程序最前面声明函数。

提示：如果函数定义在主函数之前，则不需要进行事先声明；如果函数定义在主函数之后，则必须在调用函数之前声明函数。

函数的声明方法，是在程序的最前面，列出程序中可能调用的自定义函数。需要列出与函数定义相同的类型名、函数名、参数列表。下面的程序将上一节中的自定义函数写在后面，并在程序最前面声明函数。

【范例 9-7】函数的声明实例，示例代码如下所示。

示例代码 9-7

```
01  #include <stdio.h>
02  void myresult(int x); /*声明程序中的函数 myresult。*/
03  char result(int aax); /*声明函数 result。*/
04                        /*函数声明结束，下面是主函数。*/
05  int main()            /*主函数将调用后面的函数。*/
06  {
07    int i=1 ;           /*定义存放分数的变量，赋初值为 1。*/
08    while(i!=0)         /*i 的值不为 0 则提示输入和判断结果。*/
09    {
10      scanf("%d",&i);   /*输入一个数值。*/
11      myresult(i);      /*调用函数判断等级，返回值赋值给 r。*/
12    }
13  }
14  void myresult(int x)  /*定义一个有数值有效性判断功能的函数，无返回值。
15  */
16  {
17    char s;             /*存放结果的变量。*/
18    if(x>0&&x<=100)     /*数值在这个范围内有效。*/
19    {
20      s=result(x);      /*调用函数 result()判断分数等级，调用是后面定义的函
21  数。*/
```

```
22        printf("%c\n",s);/*输出结果。*/
23     }
24     else                    /*其他值的分数。*/
25     {
26        printf("error.\n",s);                    /*输出错误提示。*/
27     }
28  }
29  char result(int x)                             /*定义一个分数评定函数。*/
30  {
31     char s;                                     /*定义一个字符变量存储结果。*/
32     if(x>=90)                                   /*90 分以上的结果为 A。*/
33     {
34       s='A';
35     }
36     else                                        /*其他的值 80 分以上的结果为 B。*/
37     {
38        if(x>=80)
39        {
40           s='B';
41        }
42        else
43        {
44           if(x>=70)                             /*其他的值 70 分以上的结果为 C。*/
45           {
46              s='C';
47           }
48           else
49           {
50              if(x>=60)                          /*其他的值 60 分以上的结果为 D。*/
51              {
52                 s='D';
53              }
54              else                               /*其他的值一定小于 60，结果为 E。*/
55              {
56                 s='E';
57              }
58           }
59        }
60     }
61     return(s);                                  /*将变量 s 作为返回值返回。*/
62  }
```

【运行结果】这个程序与上一节中的程序运行结果是相同的。

【代码解析】由于将函数的定义放在了主函数的后面，所以在程序的最前面（代码第 02 行与代码第 03 行）进行函数的声明，否则将导致编译出错。

9.2.4　递归函数

所谓递归函数，指的是函数调用自身，用循环的方法实现运算。例如一个程序求出 100 以内的整数和，可以用下面的思想。

```
100 以内的整数和 = 100 + 99 以内的整数和;
99 以内的整数和 = 99 +98 以内的整数和;
……
2 以内的整数和 = 2 +以 1 以内的整数和;
1 以内的整数和为 1。
```

用这种思路，可以用反复调用身函数的方法求出 100 以内的整数和。这个程序的代码如下所示。

【范例 9-8】使用递归函数计算 100 以内整数相加之和，示例代码如下所示。

示例代码 9-8

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

```
02  int addn(x)                          /*定义一个函数，实现 0 到 x 之间整数的相加。*/
03  {
04     int s;                            /*定义一个变量保存结果。*/
05     if(x==1)                          /*结果 x 为 1，则结果为 1。*/
06     {
07        s=1;
08     }
09     else                              /*x 不为 1 的情况。*/
10     {
11        s=x + addn(x-1);               /*递归调用自身函数。*/
12     }
13     return(s);                        /*输出一个结果。*/
14  }
15
16  int main()
17  {
18     int i=100;                        /*定义一个变量，赋值为 100。*/
19     int j;                            /*定义一个变量存储结果。*/
20     j=addn(i);                        /*调用函数求出结果。*/
21     printf("result: %d",j);           /*输出结果。*/
22  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.8c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的结果如下所示。

```
result: 5050
```

【代码解析】代码第 05~12 行用于实现函数的递归调用。代码第 07 行是递归返回时为变量 s 赋值。

提示：编写递归调用的函数时，需要注意指定递归返回的条件与返回值，否则可能导致出现死循环等错误，而达不到递归的效果。

9.2.5 main 函数的参数 argc 与 argv

前面的例子中，main()函数都是没有参数的。但编写的程序常常需要从命令行中输入参数。例如，Linux 命令中复制文件的命令如下所示。

```
cp /root/1.txt /root/2.txt
```

“cp”后面的两个字符串就是这个程序的参数。在 main()函数中，默认有 argc 与 argv 两个参数。main 函数的原型如下所示。

```
01  int main(int argc , char *argv[])
```

main 函数是有一个返回值的，如果程序执行成功，则返回一个整型值 1。程序中常常省略这个返回值。argc 是一个整型变量，表示参数的个数。*argv[]是一个指针型数组，数组中的每一个指针指向一个参数的字符数组。下面的程序可以输出程序启动时后面的参数。

【范例 9-9】 main 函数的参数应用举例，示例代码如下所示。

示例代码 9-9

```
01  #include <stdio.h>                   /*包含头文件*/
```

```
02  int main(int argc, char *argv[])                    /*带参数的额主函数*/
03  {
04     int i ;                                          /*定义一个计数器变量。*/
05     for(i=0;i<argc;i++)                              /*用变量 i 进行 for 循环。*/
06     {
07        printf("%s\n",argv[i]);                       /*输出一个指针所指向的字符串。*/
08     }
09  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.9c
```

输入下面的命令运行这个程序。在程序的后面加上若干个参数。

```
./a.out hello good evening
```

程序的结果如下所示。

```
hello
good
evening
```

【代码解析】代码第 05~08 行利用 for 循环语句实现打印所有的命令行参数的功能。

提示：代码第 02 行，与代码“int man(int argc, char ** argv)”等效。

9.3　函数与指针

函数的参数与返回值可以是指针。使用指针变量可以在函数中对其他函数的变量进行调用。利用数组的头指针，可以用指针移动的方法访问数组中的每个元素，所以可用指针作为参数使函数访问一个数组。本节将讲述函数中的指针应用。

9.3.1　值调用与引用调用

函数对参数的调用，有值调用和引用调用两种方法。前面的例子都是值调用。函数对这两种参数调用的处理方法是不同的。

- 值调用：进行函数调用时候，给形式参数分配内存空间，并把实际参数的值直接传递给形式参数。这一过程是参数值的直接传递过程。一旦形式参数获得了值，便于实际参数没有关系。形式参数的变化对实际参数没有任何影响。
- 引用调用：当用指针或数组作为函数的参数时，可以用指针来访问实际参数的值。形式参数的变化，将直接引起实际参数的变化。

注意：函数调用语句中的参数是实际参数，被调用函数中的参数是形式参数。这一概念见本章 9.1.5 节所述。

例如下面的程序，形式参数的改变不会影响实际参数的值。

【范例 9-10】函数参数的值调用举例示例代码如下所示。

示例代码 9-10　\源文件\09\9.10.c

```
01  #include <stdio.h>
02  int aa(int x)                                       /*定义一个函数，参数为 x。*/
03  {
```

```
04      x = x*x;                                    /*更改 x 的值。*/
05      return(x);                                  /*返回 x。*/
06    }
07    main()
08    {
09      int x,y;                                    /*定义两个整型变量 x 与 y。*/
10      x=5;                                        /*x 赋值为 5。*/
11      y=aa(x);                                    /*调用函数，返回值赋值给 y。*/
12      printf("X: %d Y: %d\n",x,y);                /*输出 x 与 y。*/
13    }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.10.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的结果如下所示。

```
X: 5 Y: 25
```

【代码解析】从结果可知，函数中更改形式参数的值，并不会对实际参数造成影响。实际参数的值复制给形式参数以后，形式参数无法访问实际参数。

提示： 函数参数的调用默认情况下是值调用。

9.3.2 引用调用与指针

虽然函数调用时默认是值调用，但是传入的参数如果是指针，函数可以通过这个指针改变函数外部变量的值。参数的引用调用，就是通过指针对外部变量的访问来实现的。指针可以同普通变量一样，作为一个参数传递给一个函数。函数可以更改这一个指针和访问指针指向的变量。下面的程序，是通过传入的指针来更改函数以外变量的值。

【范例 9-11】函数参数的引用调用与指针举例，示例代码如下所示。

示例代码 9-11　\源文件\09\9.11.c

```
01  #include <stdio.h>
02  void change(int *x,int *y);                     /*声明一个函数。*/
03
04  main()                                          /*主函数。*/
05  {
06    int m,n;                                      /*定义两个变量。*/
07    m=3;                                          /*对两个变量赋值。*/
08    n=5;
09    printf("m:%d  n:%d\n",m,n);                   /*输出两个变量。*/
10    change(&m,&n);                                /*用函数交换两个变量。*/
11    printf("m:%d  n:%d\n",m,n);                   /*输出两个变量。*/
12    change(&m,&n);                                /*用函数交换两个变量。*/
13    printf("m:%d  n:%d\n",m,n);                   /*输出两个变量。*/
14  }
15
16  void change(int *x,int *y)                      /*定义一个函数，指针作为函数的参数。*/
17  {
18    int temp;                                     /*定义一个中间变量。*/
19    temp=*x;                                      /*指针指向的值赋值给中间变量。*/
20    *x=*y;                                        /*指针 y 指向的值赋值给指针 x 指向的值。*/
```

```
    *y=temp;                                    /*中间变量的值赋值给指针 y 指向的值。*/
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.11.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果如下所示。

```
m:3  n:5
m:5  n:3
m:3  n:5
```

【代码解析】从结果可知，自定义函数通过参数中的指针，访问指针所指向的变量，从而交换了两个变量的值。这种指针访问变量的方法，就是函数对参数的值引用。

提示：函数参数的引用调用可以实现将函数内部变量值的变化传递到函数的调用者。

9.3.3　指针参数简单实例

本节将讲解一个指针参数实例，通过自定义函数对参数的值引用，对三个变量进行排序。函数有三个参数，分别为三个变量的指针。函数通过指针来访问主函数中的数值。程序的代码如下所示。

【范例 9-12】函数参数为指针实例，示例代码如下所示。

示例代码 9-12

```
#include <stdio.h>                              /*包含头文件*/
void order(int *x ,int *y,int *z);              /*声明一个函数。*/

main()                                          /*主函数。*/
{
   int a,b,c;                                   /*定义三个变量。*/
   a=5;                                         /*对三个变量赋值。*/
   b=3;
   c=8;
   printf("%d %d %d\n",a,b,c);                  /*输出三个变量。*/
   order(&a,&b,&c);                             /*调用函数对三个变量进行排序。*/
   printf("%d %d %d\n",a,b,c);                  /*输出三个变量。*/
}

void order(int *x ,int *y,int *z)               /*自定义函数对三个变量进行排序。*/
{
   int temp;                                    /*定义一个中间变量。*/
   if(*x > *y)                                  /*对指针指向的值进行比较。*/
   {
      temp=*x;                                  /*用一个中间变量存储指针变量 x 指向的值。*/
      *x=*y;                                    /*指针 y 指向的值赋值给指针 x 指向的值。*/
      *y=temp;                                  /*中间变量的值赋值给指针 y 指向的值。*/
   }
   if(*x > *z)              /*同样的方法，对 x 和 z 两个指针指向的值进行排序。*/
   {
      temp=*x;
      *x=*z;
      *z=temp;
```

```
    }
    if(*y > *z)                    /*同样的方法，对 y 和 z 两个指针指向的值进行排序。*/
    {
      temp=*y;
      *y=*z;
      *z=temp;
    }
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.12.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果如下所示。

```
5 3 8
3 5 8
```

【代码解析】与前面章节的三个数字进行排序不同的是，这个程序调用一个函数，将三个变量的指针作为参数。函数通过指针来访问程序中的变量。

9.3.4 自定义函数中指针使用实例

在前面的章节中，用指针实现了两个变量的交换。在前面的章节中，对两个变量进行排序时，需要对两个变量使用指针访问进行交换。实际上，在自定义函数中，可以用函数中的指针作为参数访问另外一个函数。

【范例 9-13】用函数中的指针作为参数访问另外一个函数，示例代码如下所示。

示例代码 9-13

```
#include <stdio.h>
void order(int *x ,int *y,int *z); /*声明一个函数。*/
void change(int *x,int *y);        /*声明一个函数。*/

main()                             /*主函数。*/
{
   int a,b,c;                      /*定义三个变量。*/
   a=5;                            /*对三个变量赋值。*/
   b=3;
   c=8;
   printf("%d %d %d\n",a,b,c);     /*输出三个变量。*/
   order(&a,&b,&c);                /*调用函数对三个变量进行排序。*/
   printf("%d %d %d\n",a,b,c);     /*输出三个变量。*/
}

void order(int *x ,int *y,int *z)  /*自定义函数对三个变量进行排序。*/
{
   if(*x > *y)                     /*对指针 x 和 y 指向的值进行比较。*/
   {
     change(x,y);                  /*用函数 change 交换两个指针所指向的变量。*/
   }
   if(*x > *z)                     /*对指针 y 和 z 指向的值进行比较。*/
   {
      change(x,z);                 /*用函数 change 交换两个指针所指向的变量。*/
   }
   if(*y > *z)                     /*对指针 y 和 z 指向的值进行比较。*/
   {
```

```
28        change(y,z);                    /*用函数 change 交换两个指针所指向的变量。*/
29     }
30  }
31
32  void change(int *x,int *y)            /*定义一个函数，指针作为函数的参数。*/
33  {
34     int temp;                          /*定义一个中间变量。*/
35     temp=*x;                           /*指针指向的值赋值给中间变量。*/
36     *x=*y;                             /*指针 y 指向的值赋值给指针 x 指向的值。*/
37     *y=temp;                           /*中间变量的值赋值给指针 y 指向的值。*/
38  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.13c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果如下所示。

```
5 3 8
3 5 8
```

【代码解析】本程序的功能与上一节的功能是相同的，实现对三个数的排序。在排序时，使用了两次值引用函数。

9.3.5　数组作为参数

既然指针可以作为参数传递给一个函数，函数可以通过指针访问外部变量。数组在定义时，会返回一个头指针。可以把这个头指针作为函数的参数，函数可以通过指针访问数组中所有的变量。例如下面的代码，就是把数组的指针作为函数的参数，函数用指针来访问一个数组。

【范例 9-14】数组作为函数参数简单示例，示例代码如下所示。

示例代码 9-14

```
01  #include <stdio.h>                    /*包含头文件*/
02  void myarray(int *a)                  /*定义一个函数，用数组作为参数。*/
03  {
04    int i;                              /*定义一个循环变量。*/
05    for(i=0;i<10;i++)                   /*for 循环，处理数组中的值。*/
06     {
07        *a=*a+15;                       /*指针指向的变量加上 15。*/
08        a++;                            /*指针向后移动 1 个变量。*/
09     }
10  }
11
12  int main()                            /*主函数。*/
13  {
14     int i ;                            /*定义一个变量作为计数器。*/
15     int m[10];                         /*定义一个数组。*/
16     for(i=0;i<10;i++)                  /*用 for 循环对数组赋值。*/
17     {
18        m[i]=i+10;                      /*数组赋值。*/
19     }
20     for(i=0;i<10;i++)                  /*for 循环输出结果。*/
21     {
22        printf("%d ",m[i]);             /*输出结果。*/
23     }
24     printf("\n");                      /*输出一个换行。*/
```

```
25      myarray(m);                 /*把数组的头指针作为参数，访问数组中的每一个值。
26   */
27      for(i=0;i<10;i++)           /*for 循环输出结果。*/
28      {
29        printf("%d ",m[i]);       /*输出结果。*/
30      }
31   }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.14c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
10 11 12 13 14 15 16 17 18 19
25 26 27 28 29 30 31 32 33 34
```

【代码解析】本程序中，将数组的头指针作为函数的参数。函数通过这个指针来访问数组中的每一个变量，将每个变量的值加 15。

提示： 数组名即为数组的首地址，可以作为指针变量来使用。

9.3.6 数组作为函数参数实例

数组作为函数参数，可以方便地访问函数外部的数组，这样函数就可以对函数外的多个变量发生作用。本节将讲解一个数组作为函数参数的实例，实现下面这些功能。

- 定义三个字符串数组，前两个字符串数组分别赋值。
- 自定义一个函数，将三个字符串数组的指针作为参数。函数将第 2 个数组附加到第 1 个数组后面，然后保存到第 3 个数组上。
- 对数组的访问是通过数组的指针完成的，函数没有参数。

【范例 9-15】数组作为函数参数实例，示例代码如下所示。

示例代码 9-15

```
01  #include <stdio.h>
02  void stradd(char *a,char *b,char *c)/*定义一个连接字符串的函数,参数是 3 个指针。*/
03  {
04     while(*a!=NULL)             /*如果指针 a 对定的字符不是空字符则进行循环。
05  */
06     {
07        *c=*a;                   /*指针 a 指向的字符赋值给指针 c 指向的字符。*/
08        c++;                     /*指针 c 向后移动一个元素。*/
09        a++;                     /*指针 a 指向下一个变量。*/
10     }
11     while(*b!=NULL)             /*如果指针 b 指向的变量不为空字符则进行循环。
12  */
13     {
14        *c=*b;                   /*指针 b 指向的字符赋值给指针 c 指向的字符。*/
15        c++;                     /*指针 c 指向下一个字符。*/
16        b++;                     /*指针 b 指向下一个字符。*/
17     }
18  }
19
20  int main()
21  {
```

```
22      char a[30]="abcdefg";              /*定义一个字符串数组，赋初值。*/
23      char b[30]="HIJKLMN";              /*定义一个字符串数组 b，赋初值。*/
24      char c[30]="";                     /*定义一个字值串数组，赋值为空。*/
25      printf("a:  %s\n",a);              /*输出第 1 个字符串数组。*/
26      printf("b:  %s\n",b);              /*输出第 2 个字符串数组。*/
27      stradd(a,b,c);                     /*调用函数，连接两个字符串数组。*/
28      printf("c:  %s\n",c);              /*输出字符串 c。*/
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.15c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a:  abcdefg
b:  HIJKLMN
c:  abcdefgHIJKLMN
```

【代码解析】在这个程序的 stradd 函数中，是通过指针的移动和判断字符来实现字符串的连接的。下面的代码实现了字符串 a 复制到字符串 c 上。

```
01     while(*a!=NULL)                    /*如果指针 a 对定的字符不是空字符则进行循环。
02   */
03     {
04        *c=*a;                          /*指针 a 指向的字符赋值给指针 c 指向的字符。*/
05        c++;                            /*指针 c 向后移动一个元素。*/
06        a++;                            /*指针 a 指向下一个变量。*/
07     }
```

程序用下面的方法来执行字符串的复制。

（1）先判断指针 a 指向的字符是不是 NULL，如果不是 NULL，则进行 while 循环。

（2）将指针 a 当前的字符赋值给指针 c 当前的字符。在这个过程中，是用 while 循环来逐个进行字符复制的。

（3）指针 c 指向后一个字符。

（4）指针 a 指向后一个字符。然后进行下次循环。结果指针 a 指向的字符为 NULL，表示数组 a 已经结束。

（5）用同样的步骤，将字符串 b 上所有的字符复制到字符串 c 后面。

9.4　返回值

函数执行一个操作过程以后，一般需要返回一个处理结果。这个结果就是函数的返回值。函数可以用一个数值返回、用指针返回，也可以用指针直接访问程序中的变量。本节将讲解函数的返回值问题。

9.4.1　函数返回值的类型

函数的返回值根据数据类型和处理方式的不同，一般有下面的几种形式。函数对这几种类型返回值的处理是不同的。

- 无返回值：这种函数只是完成了一个处理过程。完成函数的运行以后，程序会自动中止运行。无返回值的函数需要用 void 来声明。
- 返回一个具体值：函数返回运算的结果。函数执行的是一个运算，对参数进行处理以

后，将结果作为一个具体数据类型的变量值来返回。这类函数，在定义时需要声明函数的数据类型。

- 用指针来访问主函数中的变量：9.3.3 节与 9.3.4 节讲解的内容，函数可以通过指针来访问函数外部的变量。这种函数虽然没有返回值，但是对外部变量产生了影响。指针访问外部变量的方式，可以看做是一种特殊的返回值。
- 返回一个指针：指针是一种特殊的数据类型。函数中的指针可以作为返回值返回给主函数。这类函数需要用指针数据类型来声明。
- 返回结果到参数：如果参数中有一个指针，可以用这个指针直接访问这个指针所指向的变量。这种方式可以看做是将结果返回到参数中。

前三种情况前面的章节已经讲解。后面的章节将重点讲解后面的两种返回类型。

9.4.2 函数返回指针

指针是一种特殊的数据类型，表示指向一个变量的内存地址。函数也可以返回一个指针。在主函数中，可以用这个指针来访问所指向的变量。例如下面的程序，是用一个函数来选择两个数中较大的一个数，返回较大数的指针。然后主函数通过这个指针输出结果。

【范例 9-16】用函数返回较大数的指针，示例代码如下所示。

示例代码 9-16

```
#include <stdio.h>                          /*包含头文件*/
int *max(int *i ,int *j)                    /*定义一个函数求两个数的最大值，返回一个指针。*/
{
    int *p;                                 /*定义一个指针，存储结果。*/
    if(*i>*j)                               /*比较两个指针所指向变量的大小。*/
    {
      p=i;                                  /*指针 i 赋值给指针 p。*/
    }
    else
    {
      p=j;                                  /*另一种情况是指针 j 赋值给指针 p。*/
    }
    return(p);                              /*返回指针 p。*/
}

int main()                                  /*程序的主函数。*/
{
   int i ,j ;                               /*定义两个变量。*/
   int *q;                                  /*定义一个指向整型变量的指针。*/
   i=5;                                     /*变量赋初值。*/
   j=3;
   q=max(&i,&j);                            /*取两个数的指针作为参数，求两个数的较大值。
                                            函数会返回较大数的指针。*/
   printf("max = %d" ,*q);                  /*输出指针 q 指向的变量值。*/
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.16c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
max = 5
```

【代码解析】代码第 02~14 行定义了返回值为整型指针的 max 函数，在主函数中调用 max 函数，并将函数返回值赋给了整型指针变量。

9.4.3　函数返回指针实例

本节将讲解一个函数返回指针实例。自定义一个函数，连接两个字符串，两个字符串的指针作为函数的参数。函数连接两个字符串以后，返回连接以后字符串的指针。

【范例 9-17】用函数实现连接两个字符串，并返回连接后字符串的指针，示例代码如下所示。

示例代码 9-17　\源文件\09\9.17.c

```
#include <stdio.h>
char *stradd(char *a,char *b)  /*定义一个连接字符串函数,参数是两个指针，返回一个指针。*/
{
   char *p;                    /*定义一个指针 p。*/
   p=a;                        /*把指针 p 赋值给指针 a。*/
   while(*a!=NULL)             /*如果指针 a 对定的字符不是空字符则进行循环。*/
   {
      a++;                     /*指针 a 指向下一个变量。*/
   }
   while(*b!=NULL)             /*如果指针 b 指向的变量不为空字符则进行循环。*/
   {
      *a=*b;                   /*指针 b 指向的字符赋值给指针 c 指向的字符。*/
      b++;                     /*指针 b 指向下一个字符。*/
      a++;                     /*指针 a 指向下一个字符。*/
   }
   return(p);                  /*返回指针 p，也就是返回原来的指针 a。*/
}

int main()
{
    char a[30]="abcdefg";      /*定义一个字符串数组 a，赋初值。*/
    char b[30]="HIJKLMN";      /*定义一个字符串数组 b，赋初值。*/
    char *c;                   /*定义一个指向字符的指针 c。*/
    printf("a:  %s\n",a);      /*输出第 1 个字符串数组。*/
    printf("b:  %s\n",b);      /*输出第 2 个字符串数组。*/
    c= stradd(a,b);            /*调用函数，连接两个字符串数组，然后返回一个指针。*/
    printf("c:  %s\n",c);      /*输出这个指针所指向的字符数组。*/
}
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.17c
```

输入下面的命令运行这个程序。

```
./a.out
```

在这个程序中，函数把字符串 b 添加到字符串 a 上，实际上返回的是 a 的指针。程序的运行结果如下所示。

```
a:  abcdefg
b:  HIJKLMN
c:  abcdefgHIJKLMN
```

【代码解析】本程序与示例代码 9-15 的功能相同，函数定义时返回值类型为字符指针。

9.4.4 函数返回结果到参数

如果函数的参数是指针，函数可以访问这个指针所指向的变量的值。这时函数就可以把结果返回给参数中的指针变量。如果函数的参数不是指针，函数是不能把结果返回给参数的。例如下面的代码实现两个数的加法，将结果返回给参数的指针。

【范例 9-18】用函数实现两个数加法运算，并将结果返回给函数的参数，示例代码如下所示。

示例代码 9-18

```
01  #include <stdio.h>
02  void  add(int a,int b,int *c)   /*定义一个自定义函数，参数是两个整型变量和一个指针。*/
03  {
04     *c=a+b;                      /*将相加的结果赋值给参数中指针指向的值。*/
05  }
06
07  int main()                      /*程序的主函数。*/
08  {
09     int a,b,c;                   /*定义 3 个整型变量。*/
10     a=5;                         /*变量赋值。*/
11     b=8;
12     add(a,b,&c);                 /*调用函数求和，c 取地址作为参数。*/
13     printf("reslut is %d",c);    /*输出 c 的值。*/
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 9.18.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
reslut is 13
```

【代码解析】本程序中，代码第 12 行将函数的返回值通过参数 c 传递回主函数。

9.5 综合练习

C 语言是一种函数式的编程语言，熟练掌握函数的相关知识是学好 C 语言的基础。本节的代码将计算出用户给定的两个日期之间相差的天数。

【范例 9-19】计算两个给定日期相差的天数，示例代码如下所示。

示例代码 9-19

```
01  #include "stdio.h"              /*包含头文件*/
02
03  int Mon[13]={ 0,31,28,31,30,31,30,31,31,30,31,30,31};
                                    /*定义一个一维数组，记录每月的天数*/
04
05  int funy(int y)                 /*自定义函数，判断是否是闰年*/
06    {
07     return(y%4==0&&y%100!=0||y%400==0);
08    }
09
```

```
int funa(int y,int m,int d)          /*自定义函数，计算给定日期到年底的天数*/
  {
    int i,days=0;
    for(i=m+1;i<13;i++) days+=Mon[i];
    days+=(Mon[m]-d);
 if(m<3) days+=funy(y);
    return days;
  }

int funb(int y,int m,int d)          /*自定义函数，计算给定日期是该年的第多少天*/
  {
    int i,days=0;
    for(i=1;i<m;i++) days+=Mon[1];
    days+=d;
    if(m>2) days+=funy(y);
    return days;
  }

int func(int y1,int y2)              /*自定义函数，计算整年的天数*/
{
  int i,days=0;
  for(i=y1+1;i<y2;i++)
   days+=(365+funy(i));
  return days;
}

main()                               /*主函数*/
{
   int y1,m1,d1,y2,m2,d2,days=0;     /*对应变量，用于接收用户输入的日期*/
   printf("please input begin date(yyyy-mm-dd)\n");
   scanf("%d-%d-%d",&y1,&m1,&d1);                    /*输入第一个日期*/
   printf("please input end date(yyyy-mm-dd)\n");
   scanf("%d-%d-%d",&y2,&m2,&d2);                    /*输入第二个日期*/
   if(y1==y2) days=funa(y1,m1,d1)-funa(y2,m2,d2);/*计算两个给定日期相差的天数*/
      else days=funa(y1,m1,d1)+funb(y2,m2,d2)+func(y1,y2);
   printf("%d-%d-%d~%d-%d-%d:all days is %d !\n",y1,m1,d1,y2,m2,d2,days);
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 9.20.c
```

执行这个程序。

```
./a.out
```

程序提示输入一个日期，然后按“Enter”键后，提示输入第二个日期，最后输出两个给定日期相差的天数。输出的结果如下所示。

```
please input begin date(yyyy-mm-dd)
2009-1-1
please input end date(yyyy-mm-dd)
2010-10-10
2009-1-1~2010-10-10:all days is 653 !
```

9.6 小结

本章讲述了C程序中的函数操作。学习难点是自定义函数的理解和使用。函数可以把一个功能或模块封装起来，程序可以方便地使用这个已经封装的功能。在自定义函数中，需要注意函数的参数与返回值的理解。指针与数组作为一种特殊的数据类型，可以作为函数的参数和返回值，这一知识比较难以理解，读者在学习时需要根据实例理解函数中指针和数组的作用。

9.7 习题

一、填空题

1. 简单地说，_____就是把一个程序功能封装成一个整体。它由_____、_____、_____、_____等部分组成。

2. 按照函数的来源，可以将函数分为库函数与用户自定义函数。_____是编程平台提供的函数。_____是用户自己按编程的需要编写的函数。

3. 所谓_____，指的是函数运行以后会产生一个数据，这个数据用 return 语句返回给主程序。

4. 参数指的是调用语句向被调用函数传送的数据。从调用语句和被调用函数之间数据传送的角度，又可将函数分为_____和_____两种。

5. 所谓_____，指的是函数调用自身，用循环的方法实现运算。

6. 函数对参数的调用，有_____和_____两种方法。

二、选择题

1. 下面关于函数的参数的说法中错误的是（　　）。
   A. 参数指的是调用语句向被调用函数传送的数据
   B. 从调用语句和被调用函数之间数据传送的角度，又可将函数分为无参函数和有参函数两种
   C. 无参函数指的是函数在定义、函数说明及函数调用中均不带参数。调用语句和被调用函数之间不进行参数传送
   D. 有参函数在定义、调用、声明时可以有参数也可以没有参数。有参函数也称为带参函数

2. 下面关于函数的返回值的说法中正确的是（　　）。
   A. 无返回值：这种函数只是完成了一个处理过程。完成函数的运行以后，程序会自动中止运行。无返回值的函数需要用 void 来声明
   B. 指针是一种特殊的数据类型，表示指向一个变量的内存地址。函数不可以返回一个指针
   C. 返回一个具体值：函数返回运算的结果。函数执行的是一个运算，对参数进行处理以后，将结果作为一个具体数据类型的变量值来返回。这类函数，在定义时不需要声明函数的数据类型
   D. C 语言中不支持将函数结果返回到参数中

3. C 语言中常用的库函数描述错误的是（　　）。
   A. 字符类型分类函数：对字符按照字母、数字、控制字符、分隔符、大小写字母等方面进行分类测试
   B. 转换函数：用于字符或字符串的转换。可以在字符变量和各类数字变量之间进行转换；在大、小写之间进行数据类型转换
   C. 目录路径函数：用于文件目录和路径操作，但是不能对文件或目录进行创建、删除、移动等操作
   D. 诊断函数：用于系统内部错误检测，完成系统管理、系统诊断的功能

4．关于 main 函数的参数说法中正确的是（　　）。

A．main()函数都是没有参数的

B．在 main()函数中，默认有 argc 与 argv 两个参数

C．main()函数中参数 argc 表示命令行参数的内容，argv 表示命令行参数的个数

D．main()函数必须提供 argc 和 argv 两个参数

三、简答题

1．什么是函数、函数的参数、函数的返回值？

2．简述函数的值调用和引用调用的区别。

2．简述系统库函数的作用与使用方法。

四、编程题

1．编写程序，要求程序从命令行参数中读取两个整数，并输出这两个整数相加后的和。

2．定义一个排序函数，函数的形式参数是三个整型数。在主函数中使用 scanf 函数从终端中读取三个整型数据，并调用排序函数完成三个整数的排序，最后输出排序结果。

第 10 章　字符与字符串处理

在程序设计中经常需要对字符和字符串进行处理，C 语言中内置了许多与字符和字符串操作相关的库函数，本章将分别介绍字符测试函数，字符串转换函数，字符串比较函数，字符串复制函数，字符串清理与填充函数，字符串查找函数及字符串的连接与分割函数，具体内容如下：

- 字符测试函数；
- 字符串转换函数；
- 字符串比较函数；
- 字符串复制函数；
- 字符串清理与填充函数；
- 字符串查找函数；
- 字符串连接与分割函数。

10.1　字符测试函数

所谓字符测试，是指对一个字符进行大小写、是否可以打印、是否可以显示、是否是数字等方面的判断。C 程序提供了丰富的字符测试函数，这些函数原型在头文件“ctype.h”中定义，使用这些函数之前需要包含此头文件。

提示：使用某一类功能的系统库函数时，必须在程序开头包含对应的头文件，否则将出现函数未定义之类的错误，导致无法正常编译。

10.1.1　数字或字母测试函数 isalnum

函数 isalnum 的作用，是检查参数 c 是否为英文字母或阿拉伯数字。若参数 c 是一个字母或数字，则返回真值，否则返回假。这个函数包含于头文件“ctype.h”中，使用方法如下所示。

```
01  int isalnum (int c)
```

提示：参数 c 是一个字符变量。但在 C 程序中，字符变量等同于这个变量所对应的 ASCII 码的值，所以参数也可以是一个 ASCII 码值的整型数值。

下面是这个函数的使用实例，测试一个字符数组中所有的字符，如果是字母或数字，则输出结果。

【范例 10-1】使用 isalnum 函数测试字符数组中的字符是否为字母或数字，示例代码如下所示。

示例代码 10-1

```
01  #include <stdio.h>
02  #include <ctype.h>                    /*包含头文件 ctype.h。*/
```

```
03  main()
04  {
05      char s[]="12as056;^*&";          /*定义一个字符串数组。*/
06      int i;                           /*定义一个整型变量作为循环的计数器。*/
07      for (i=0;s[i]!=NULL;i++ )        /*for 循环，判断 s[i]是否为空作为循环条件。*/
08      {
09          if (isalnum(s[i]))           /*判断当前的字符是不是一个字母或数字。*/
10          {
11              printf("%c is a number or character.\n",s[i]); /*输出结果。*/
12          }
13      }
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.1.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1 is an number or character.
2 is an number or character.
a is an number or character.
s is an number or character.
0 is an number or character.
5 is an number or character.
6 is an number or character.
```

【代码解析】程序用循环的方法，对字符数组中的每一个字符进行测试，如果是字母或数字，则输出结果。

10.1.2　字母测试函数 isalpha

函数 isalpha 可以测试一个字符是不是英文字母。这个函数的使用方法如下所示。

```
01  int isalpha (int c)
```

函数的参数 c 是一个字母或是表示一个字母的 ASCII 码值。如果这个参数是一个英文字母，则返回真值，否则返回假。这里所说的英文字母，指的是 26 个大写字母和 26 个小写字母，而不包括其他的任何符号。下面的程序对一个字符数组中的每一个字符进行测试，如果是字母则输出结果。

【范例 10-2】使用 isalpha 函数测试字符是否为英文字母，示例代码如下所示。

示例代码 10-2

```
01  #include <stdio.h>
02  #include <ctype.h>                   /*包含头文件 ctype.h。*/
03  main()
04  {
05      char s[]="12as056;^*&";          /*定义一个字符串数组。*/
06      int i;                           /*定义一个整型变量作为循环的计数器。*/
07      for (i=0;s[i]!=NULL;i++ )        /*for 循环，判断 s[i]是否为空作为循环条件。*/
08      {
09          if (isalpha(s[i]))           /*判断当前的字符是不是字母。*/
10          {
11              printf("%c is a character.\n",s[i]);          /*输出结果。*/
12          }
13      }
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.2.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a is a character.
s is a character.
Q is a character.
W is a character.
```

【代码解析】程序使用循环语句依次测试字符数组中的每一个字符，如果是字母就输出结果。

10.1.3 可打印字符测试函数 isgraph

所谓可打印字符，指的是这个字符可以在屏幕上显示，或是可以在打印机中打印出这个字符。ASCII 码中的有些字符用于格式控制或特殊作用，是不可以打印的。可打印字符包括字母、数字、标点符号、键盘上可打印符号等。函数 isgraphis 的作用是判断一个字符是否是可打印字符，使用方法如下所示。

```
01  int isgraph (int c)
```

参数 c 是一个字符，或是表示一个字符的 ASCII 码值。如果参数是一个可打印字符，则返回一个真值，否则返回假。下面的代码，是使用这个函数判断字符可否打印的实例。

【范例 10-3】使用 isgraph 函数测试是否是可打印字符，示例代码如下所示。

示例代码 10-3

```
01  #include <stdio.h>
02  #include <ctype.h>                  /*包含头文件 ctype.h。*/
03  main()
04  {
05      char s[]="12 0Q\n*&";           /*定义一个字符串数组。*/
06      int i;                          /*定义一个整型变量作为循环的计数器。*/
07      for (i=0;s[i]!=NULL;i++ )       /*for 循环，判断 s[i]是否为空作为循环条件。*/
08      {
09          if (isgraph(s[i]))          /*判断当前字符是否是可打印字符。*/
10          {
11              printf("%c is a printable character.\n",s[i]);      /*如果是可打印字
符则输出结果。*/
12          }
13      }
14  }
```

输入下面的命令编译这段代码。

```
gcc 10.3.c
```

输入下面的命令运行这个程序。

```
./a.out
```

【运行结果】运行结果如下所示。

```
1 is a printable character character.
2 is a printable character character.
0 is a printable character character.
Q is a printable character character.
```

```
* is a printable character character.
& is a printable character character.
```

【代码解析】程序循环测试字符数组中的每一个字符，如果是可打印字符就输出结果。

10.1.4　测试大小写字母函数 islower 和 isupper

函数 lower 用于测试一个字符是不是小写字母，isupper 用于测试一个字符是不是大写字母。这两个函数的使用方法如下所示。

```
01   int islower (int c)
02   int isupper (int c)
```

在函数 islower 中，参数 c 是一个字符，如果这个参数是一个小写字母，函数就返回真值，否则返回为假。函数 isupper 的用法与 islower 相似。下面是一个判断字符大小写的实例，判断一个字符数组中有哪些小写字母与大写字母。

【范例 10-4】使用 islower 函数和 isupper 函数测试字符的大小写，示例代码如下所示。

示例代码 10-4

```
01  #include <stdio.h>
02  #include <ctype.h>                          /*包含一个头文件。*/
03  main()
04  {
05     char s[]="12asSHDqw^i&*";                /*定义一个字符串数组。*/
06     int i;                                   /*定义一个变量作为循环计数器。*/
07     for (i=0;s[i]!=NULL;i++ )                /*如果数组中当前的字符存在则进行循环。 */
08     {
09        if (islower(s[i]))                    /*如果是小写字母则输出一个结果。*/
10        {
11          printf("%c is a islower character.\n",s[i]);     /*输出结果。 */
12        }
13        if (isupper(s[i]))                    /*如果是大写字母则输出结果。 */
14        {
15          printf("%c is a upper character.\n",s[i]);       /*输出结果。*/
16        }
17     }
18  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.4.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a is a islower character.
 s is a islower character.
S is a upper character.
H is a upper character.
D is a upper character.
q is a islower character.
w is a islower character.
```

【代码解析】程序判断字符数组中的每一个字符，然后输出结果。

10.1.5　测试是否是数字 isxdigit

函数 isxdigit 可以测试一个字母是不是 0～9 之间的阿拉伯数字。这个函数的使用方法如下

所示。

```
01  int isdigit(int c)
```

这个函数的参数 c 是一个字符，或者是 ASCII 码表中的一个编号。函数对这个字符进行判断，如果是一个阿拉伯数字则返回一个真假，否则返回值为假。下面是一个 isxdigit 函数的使用实例，判断一个字符数组中的字符，如果是数字则输出结果。程序的代码如下所示。

【范例 10-5】使用 isxdigit 函数判断是否是数字，示例代码如下所示。

示例代码 10-5

```
01  #include <ctype.h>                    /*包含 ctype.h 头文件。*/
02  #include <stdio.h>
03  main()                                /*程序的主函数。*/
04  {
05      char s[]="123asd0ASD$%^";         /*定义一个字符串数组。*/
06      int i;                            /*定义一个变量作为循环计数器。*/
07      for(i=0;s[i]!=0;i++)              /*for 循环，当前的有没有字符作为循环条件。*/
08      {
09          if(isdigit(s[i]))                          /*判断当前字符是不是一个数字。*/
10          {
11              printf("%c is a number.\n",s[i]);      /*输出判断结果。*/
12          }
13      }
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.5.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1 is a number.
2 is a number.
3 is a number.
0 is a number.
```

【代码解析】程序第 09 行调用 isdigit 函数判断出数组中的数字，然后输出结果。

10.1.6 测试字符是否为符号函数 ispunct

函数 ispunct 可以测试一个字符是否为标点符号或特殊符号。这个函数的使用方法如下所示。

```
01  int ispunct(int c)
```

函数的参数 c 是需要测试的字符，或是 ASCII 码表中用来表示这个字符的编号。函数对这个字符进行测试，如果是一个标点符号或特殊字符，则返回一个真值。否则返回值为假。下面是这个函数的使用实例，对一个字符数组中的特殊字符进行测试并输出。

【范例 1-6】使用 ispunct 函数测试字符是否为一个标点符号或特殊符号，示例代码如下所示。

示例代码 10-6

```
01  #include <ctype.h>                    /*包含 ctype.h 头文件。*/
02  #include <stdio.h>
03  main()                                /*程序的主函数。*/
04  {
```

```
05      char s[]="123a,.;sd0ASD$^";              /*定义一个字符串数组。*/
06      int i;                                   /*定义一个变量作为循环计数器。*/
07      for(i=0;s[i]!=0;i++)                     /*for 循环，当前的有没有字符作为循环条件。
08   */
09      {
10        if(ispunct(s[i]))                            /*判断当前的字符是否是一个符号。*/
11        {
12        printf("%c is a is a punct.\n",s[i]);  /*如果是符号则输出结果。*/
13        }
14      }
15   }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.6.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
, is a is a punct.
. is a is a punct.
; is a is a punct.
$ is a is a punct.
^ is a is a punct.
```

【代码解析】程序第 10 行调用 ispunct 函数判断出字符数组中的符号，然后输出结果。

10.1.7　其他字符测试函数

除了上面讲解的这些字符测试函数以外，还有空格测试、可否打印测试、控制字符测试等函数。这些函数的使用方法和上面这些函数的使用方法是相似的。如果需要对程序中的字符进行类型测试，需要在程序的最前面包含头文件 ctype.h。这些函数的功能与使用方法如表 10.1 所示。

表 10.1　字符测试函数

函 数 名	作　　用	使用方法
isalnum	测试字符是否为英文或数字	int isalnum (int c)
isalpha	测试字符是否为英文字母	int isalpha (int c)
isascii	isascii（测试字符是否为 ASCII 码字符）	int isascii(int c)
iscntrl	测试字符是否为 ASCII 码的控制字符	int iscntrl(int c)
isdigit	测试字符是否为阿拉伯数字	int isdigit(int c)
isgraphis	测试字符是否为可打印字符	int isgraph (int c)
islower	测试字符是否为小写字母	int islower (int c)
isprint	测试字符是否为可打印字符	int isprint (int c)
isspace	测试字符是否为空格	int isspace (int c)
ispunct	测试字符是否为标点符号或特殊符号	int ispunt (int c)
isupper	测试字符是否为大写字母	int isupper (int c)
isxdigit	测试字符是否为十六进制字符	int isxdigit (int c)

10.2　字符测试函数综合实例

在程序中常常需要判断输入的内容是否有效，这就需要使用字符测试函数。本节讲解一个

字符测试实例。程序提示用户从键盘输入一行字符，这行字符保存在一个字符数组中，然后程序统计字符中大写字母、小写字母、数字、符号的个数，然后输出结果。程序需要用循环的方法来访问字符数组中的每一个字符，然后判断当前字符的类别。编写的代码如下所示。

【范例 10-7】统计给定字符串中各类字符的个数，示例代码如下所示。

示例代码 10-7

```
01  #include <ctype.h>                                  /*包含 ctype.h 头文件。*/
02  #include <stdio.h>
03  main()                                              /*程序的主函数。*/
04  {
05    char s[100];                                      /*定义一个字符数组。*/
06    int i,c_num,c_lower,c_upper,c_mark ;               /*定义几个计数变量。*/
07    c_num=0;                                          /*对计数变量赋初值为 0。*/
08    c_lower=0;
09    c_upper=0;
10    c_mark=0;
11    printf("please input a string:\n");               /*输出一个提示信息。*/
12    scanf("%s",s);                                    /*从键盘输入一个字符串。*/
13    for(i=0;s[i]!=NULL;i++)                           /*for 循环，访问字符数组中的所有字符。*/
14    {
15      if(isdigit(s[i]))                               /*判断当前字符是不是一个数字。*/
16      {
17        c_num++;                                      /*数字字符的计数变量加 1。*/
18      }
19      if(islower(s[i]))                               /*判断当前字符是不是一个小写字母。*/
20      {
21        c_lower++;                                    /*小写字母的计数加 1。*/
22      }
23      if(isupper(s[i]))                               /*判断当前字符是不是一个大写字母。*/
24      {
25        c_upper++;                                    /*大写字母的计数加 1。*/
26      }
27      if(ispunct(s[i]))                               /*判断当前字符是不是一个特殊符号。*/
28      {
29        c_mark++;                                     /*符号的计数加 1。*/
30      }
31    }
32    printf("upper: %d\n",c_upper);                    /*输出大写字母的统计结果。*/
33    printf("lower: %d\n",c_lower);                    /*输出小写字母的统计结果。*/
34    printf("mark: %d\n",c_mark);                      /*输出符号的统计结果。*/
35    printf("number: %d\n",c_num);                     /*输出数字的统计结果。*/
36    printf("total: %d\n",i);                          /*输出字符的总数。*/
37  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.7.c
```

输入下面的命令运行这个程序。

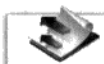
```
./a.out
```

程序输出下面的语句，提示输入一行文本。

```
please input a string:
```

这时输入下面的字符，然后按“Enter”键。

```
asd^*2749#$%YIDHioprt
```



注意：字符之间不可以有空格。

程序对输入字符进行统计，显示的结果如下所示。

```
upper: 4
lower: 8
mark: 5
number: 4
total: 21
```

【代码解析】代码第 13~31 行使用 for 循环统计输入的字符串中各种字符的个数，在 for 循环内部通过调用函数 isdigit、islower、isupper 和 ispunct 判断用户输入的字符的类型，并统计个数。

10.3　字符串转换

所谓字符串转换，指的是把字符串转换成整型、浮点型等数据类型，或者进行大小写转换。本节将讲解常用的字符串操作与相关函数的使用。

10.3.1　C 程序中的字符串

在 C 程序中，并没有字符串这一数据类型。这里所说的字符串，实际上是一个字符数组。字符串通常是保存在一个字符数组中的，可以用这个数组的名称或这个数组的头指针定义一个字符串。可以用下面的方法来定义一个字符串。

```
01   char a[50];
02   char b[];
```

用后面一种方法定义字符串，没有设置数组的长度，这种数组的长度是可以根据需要改变的。可以用下面的方法对字符串进行赋值。

```
01   char a[50]="asdfgh";
02   char b[]="ASDFG";
```

也可以直接定义一个指向字符的指针，对这个指针赋值为一个字符串，这种方法如下所示。

```
01   char *a="asdfg";
02   char *b;
03   *b="SDFG";
```

但是下面这种方法，先定义一个数组，然后用另一个语句赋值为字符串，这种方法是错误的。

```
01   char a[50];
02   a="asdfg";
```

如果需要在数组定义以后，在后面的程序中进行赋值，可以用键盘输入的方法，代码如下所示。

```
01   char a[50];
02   scanf("%s",a);
```

10.3.2　字符串转换成浮点型函数 atof

函数 atof 的作用是将一个字符串转换成一个浮点型变量。函数的使用方法如下所示。

```
01   double atof(char *nptr);
```

函数的参数*nptr 是一个字符串，函数可以把字符串转换成一个浮点型数，然后返回。在处理字符串时，跳过前面的空格，遇上数字或正负符号才开始做转换，一直到字符串的结尾。如果字符串中有字母或其他符号，函数会去除这个字符和这个字符以后的内容。下面是这个函数的使用实例，需要注意的是字符串中字母的处理情况。

【范例 10-8】使用 atof 函数将字符串转换成浮点数，示例代码如下所示。

示例代码 10-8

```
01  #include <stdlib.h>                                  /*包含 stdlib.h 头文件。*/
02  #include <stdio.h>
03  int main()
04  {
05     char a[8]="-123.45";                              /*定义字符串然后赋值。*/
06     char b[6]="23.456";
07     char c[6]="-1234";
08     char d[6]="123asd";                               /*字符串中含有字母。*/
09     char e[6]="12a12";                                /*字符串中含有字母。*/
10     float x,y,z,m,n ;                                 /*定义几个浮点型变量。*/
11     x=atof(a);                                        /*用 atof 函数转换成浮点型。*/
12     y=atof(b);
13     z=atof(c);
14     m=atof(d);                                        /*注意结果中字母的处理。*/
15     n=atof(e);
16     printf("string: %s  float: %f",a,x);              /*输出字符串和转换结果。*/
17     printf("string: %s  float: %f",b,y);
18     printf("string: %s  float: %f",c,z);
19     printf("string: %s  float: %f",d,m);
20     printf("string: %s  float: %f",e,n);
21  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.8.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: -123.45  float: -123.449997
string: 23.456-1234  float: 23.455999
string: -1234  float: -1234.000000
string: 123asd12a12  float: 123.000000
string: 12a12  float: 12.000000
```

【代码解析】代码第 11~15 行调用函数 atof 将 5 种类型的字符串转换为浮点型数据，最后打印转换结果。

10.3.3 字符串转换成整型函数 atoi

函数 atoi 的作用是将一个字符串转换成一个整型数，使用方法如下所示。

```
01  int atoi(char *nptr);
```

参数 nptr 是一个字符串的指针。函数会扫描这个字符串，如果字符串前面有空格则跳过空格。从每一个字符直到字符串结束，如果字符串中有其他字符，则去除这个字符与这个字符以后的内容。如果字符串中有小数点，则去除小数点以后的内容。下面是使用 atoi 函数将字符串转换成整型变量的实例。

【范例 10-9】使用 atoi 函数将字符串转换成整型数，示例代码如下所示。

示例代码 10-9

```
01  #include <stdlib.h>                          /*包含 stdlib.h 头文件。*/
02  #include <stdio.h>
03  int main()
04  {
05     char a[8]="-1234";                        /*定义需要转换的字符串。*/
06     char b[6]="123";
07     char c[8]="-12.24";
08     char d[6]="0.135";
09     int x,y,z,n ;                             /*定义 4 个整型变量，存储转换后的结果。*/
10     x=atoi(a);                                /*用 atoi 函数将字符串转换成整型。*/
11     y=atoi(b);
12     z=atoi(c);
13     n=atoi(d);
14     printf("string: %s  int: %d\n",a,x);      /*输出字符串和转换的结果。*/
15     printf("string: %s  int: %d\n",b,y);
16     printf("string: %s  int: %d\n",c,z);
17     printf("string: %s  int: %d\n",d,n);
18  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.9.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: -1234  int: -1234
string: 123  int: 123
string: -12.24  int: -12
string: 0.135  int: 0
```

【代码解析】代码第 10~13 行调用函数 atoi 将 4 种类型的字符串转换为整型数据，最后打印转换结果。

10.3.4　字符串转换成长整型函数 atol

函数 atol 的作用是将一个字符串转换成长整型数，使用方法如下所示。

```
01  long atol(char *nptr);
```

参数 nprt 字符串的指针，函数 atol 会把这个字符串转换成一个长整型数返回。对 0 和字母的处理方式与 10.3.2 节中的 atof 函数是相同的。下面是使用函数 atol 将一个字符串转换成整型数的实例。

【范例 10-10】使用 atol 函数将字符串转换成长整型数，示例代码如下所示。

示例代码 10-10

```
01  #include <stdlib.h>
02  #include <stdio.h>
03  int main()
04  {
05     char a[8]="123567";                       /*定义需要转换的字符串并且赋值。*/
06     char b[8]="-45645";                       /*字符串中可以有负号。*/
07     char c[8]="5645.234";                     /*需要注意小数点的处理方式。*/
08     char d[8]="0.135";                        /*需要注意小数点的处理方式。*/
```

```
09      char e[8]="5675asd";                    /*需要注意字母的处理方式。*/
10      long x,y,z,n,m ;                        /*定义几个长整型变量存储转换结果。*/
11      x=atol(a);                              /*用 atol 函数将字符串转换成长整型数。*/
12      y=atol(b);
13      z=atol(c);
14      n=atol(d);
15      m=atol(e);
16      printf("string: %s  long: %ld\n",a,x);    /*输出字符串和转换以后的长整型数。*/
17      printf("string: %s  long: %ld\n",b,y);
18      printf("string: %s  long: %ld\n",c,z);
19      printf("string: %s  long: %ld\n",d,n);
20      printf("string: %s  long: %ld\n",e,m);
21  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.10.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
string: 123567  long: 123567
string: -45645  long: -45645
string: 5645.2340.135  long: 5645
string: 0.135  long: 0
string: 5675asd  long: 5675
```

【代码解析】代码第 11~15 行调用函数 atol 将 5 种类型的字符串转换为长整型数据，最后打印转换结果。

10.3.5 将浮点型转换成字符串函数 ecvt

函数 ecvt 可以将一个浮点型数转换成一个字符串，这个函数的使用方法如下所示。

```
01  char *ecvt(double number,int ndigits,int *decpt,int *sign)
```

这个函数参数与返回值的作用如下所示。

- number：是一个 double 型的浮点数，函数需要对这个浮点数进行转换。
- ndigits：在浮点数中从左向右取的位数。
- decpt：是一个整型数的指针，显示浮点数中小数点在第几位。
- sign：是一个整型数的指针。代表数值的正与负，如果为正则返回 0，否则返回 1。

函数的返回值是指向字符串的指针。下面的代码，是使用函数 ecvt 将浮点型数转换成整型的实例。

【范例 10-11】使用 ecvt 函数将浮点数转换成字符串，示例代码如下所示。

示例代码 10-11

```
01  #include <stdlib.h>                         /*包含头文件 stdlib.h。*/
02  #include <stdio.h>
03  int main()
04  {
05     float a,b ;                              /*定义两个浮点型数。*/
06     char  *p;                                /*定义一个指向浮点数的指针。*/
07     int po,sign;                             /*定义两个整型变量用于结果的返回。*/
08     a=123.45;                                /*需要转换的浮点数。*/
09     b=-2345.754;
10     p=ecvt(a,4,&po,&sign);                   /*转换成字符串。*/
```

```
11      printf("float:%f  string :%s  howmany:4 dot:%d sign:%d\n",a,p,po,sign);
12      p=ecvt(b,4,&po,&sign);                  /*转换成字符串。*/
13      printf("float:%f  string :%s  howmany:4 dot:%d sign:%d\n",b,p,po,sign);
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.11.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。howmany 表示在浮点型数中取的位数，dot 表示浮点数中小数点出现在第几位的后面，sign 表示浮点数正负。

```
float:123.449997  string :1234  howmany:4 dot:3 sign:0
float:-2345.753906  string :2346  howmany:4 dot:4 sign:1
```

【代码解析】代码第 10 行和第 12 行调用 ecvt 函数将指定的浮点型数据转换为字符串，然后打印转换结果。

10.3.6　字母的大写小转换函数 tolower 和 toupper

函数 tolower 可以把一个大写字母转换成小写字母，函数 toupper 可以把一个小写字母转换成大写字母。这两个函数的使用方法如下所示。

```
01  int tolower(int c)
02  int toupper(int c)
```

参数 c 是需要进行转换的字母。函数 tolower 可以把一个大写字母转换成小写字母。如果这个字母是小写字母或其他符号，则不进行转换，直接返回。返回值是一个字符。下面的实例用这两个函数进行大小写转换。

【范例 10-12】使用 tolower 函数和 toupper 函数转换字母的大小写，示例代码如下所示。

示例代码 10-12

```
01  #include <stdio.h>
02  #include <stdlib.h>                     /*包含 stdlib.h 头文件。*/
03  main()
04  {
05     char a[20]="asFG$%";                 /*定义一个字符串并且赋值。*/
06     int i;                               /*定义一个整型变量作为计数器。*/
07     char c;                              /*定义一个字符变量存储结果。*/
08     printf("tolower:\n");                /*输出提示。*/
09     for(i=0;a[i]!=NULL;i++)              /*for 循环访问数组中的每一个字符。*/
10     {
11        c=tolower(a[i]);                  /*将当前的字符转换成小写。*/
12        printf("  %c  %c\n",a[i],c);      /*输出结果。*/
13     }
14     printf("toupper:\n");                /*输出提示。*/
15     for(i=0;a[i]!=NULL;i++)              /*用 for 循环访问字符串中的每一个字符。*/
16     {
17        c=toupper(a[i]);                  /*将当前字符转换成大写。*/
18        printf("  %c  %c\n",a[i],c);      /*输出结果。*/
19     }
20  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.12.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
tolower:
  a  a
  s  s
  F  f
  G  g
  $  $
  %  %
toupper:
  a  A
  s  S
  F  F
  G  G
  $  $
  %  %
```

【代码解析】:

（1）代码第 9~13 行在 for 循环内调用 tolower 函数将指定字符数组中的字符转换为小写字符，并打印转换结果。

（2）代码第 15~19 行在 for 循环内调用 toupper 函数将指定字符数组中的字符转换为大写字符，并打印转换结果。

10.3.7 其他字符串转换函数

除了上面的字符串转换函数以外，还有二进制转换、十六进制转换等函数。这些函数的使用方法和上面这些函数的使用方法是相似的。

提示：如果需要对程序中的字符串与其他类型进行转换，需要在程序的最前面包含头文件 stdlib.h。

这些函数的功能与使用方法如表 10.2 所示。

表 10.2 字符串转换函数

函 数	作 用	使用方法
atof	将字符串转换成浮点型数	double atof(const char *nptr)
atoi	将字符串转换成整型数	int atoi(const char *nptr)
atol	将字符串转换成长整型数	long atol(const char *nptr)
gcvt	将浮点型数转换为字符串	char *gcvt(double number，size_t ndigits，char *buf)
ecvt	将浮点型数转换为字符串	char *ecvt(double number，size_t ndigits，char *buf)
fcvt	将浮点型数转换为字符串	char *fcvt(double number，size_t ndigits，char *buf)
strtod	将字符串转换成浮点数	double strtod(const char *nptr,char **endptr)
strtol	将字符串转换成长整型数	long int strtol(const char *nptr,char **endptr,int base)
strtoul	将字符串转换成无符号长整型数	unsigned long int strtoul(const char *nptr,char **endptr,int base)
toascii	将整型数转换成合法的 ASCII 码字符	int toascii(int c)
tolower	将大写字母转换成小写字母	int tolower(int c)
toupper	将小写字母转换成大写字母	int toupper(int c)

10.4　字符串比较

字符串处理指的是对字符串进行比较、复制、连接、查找、替换等操作。这些字符串处理操作是通过字符串处理函数实例的，这些函数是头文件 string 定义和声明的。本章将讲解这些函数的使用，学习的重点是内存和地址的理解。

本节将讲解字符串比较的各种操作。字符串比较指的是比较两个字符串的大小、是否相同等操作。常用函数有 bcmp、memcmp、strcmp、strncasecmp 等。

10.4.1　字符串比较函数 bcmp

函数 bcmp 用来比较两个字符串的前 *n* 个字节是否相同。如果相同则返回 0 值，否则返回非 0 值。函数的使用方法如下所示。

```
01   int bcmp ( const void *s1,const void * s2,int n);
```

注意：返回的 0 值作为 if 判断的条件时，相当于假，非 0 值为真。

参数 s1 与 s2 是需要进行内容比较的两个字符串，*n* 是需要比较的两个字符串前 *n* 个字符。下面是这个函数的使用实例。程序中先定义两个字符串数组，然后比较前几个字节是否相同并且输出结果。

【范例 10–13】使用 bcmp 函数比较字符串中指定字符是否相同，示例代码如下所示。

示例代码 10-13

```
01  #include <stdio.h>
02  #include <string.h>                          /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfgQWERT!@$%";              /*定义两个字符串并且赋值。*/
06     char b[20]="asdfgqwERT!@$%";
07     if(bcmp(a,b,5)==0)            /*比较这两个字符串的前 5 个字符，前 5 个字符是相同的。
08  */
09     {
10        printf("same.\n");         /*结果为真则输出相同。*/
11     }
12     else
13     {
14        printf("not same.\n");     /*结果为假则输出不同。*/
15     }
16     if(bcmp(a,b,7)==0)            /*比较两个字符串的前 7 个字符，前 7 个字符是相同的。
17  */
18     {
19        printf("same.\n");         /*如果结果为真则输出相同。*/
20     }
21     else
22     {
23        printf("not same.\n");     /*结果为假则输出不同。*/
24     }
25  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.13.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
same.
not same.
```

【代码解析】在这个程序中，需要注意函数 bcmp 的返回值。像这种进行判断或比较的函数，返回值如果是真或假，可以直接作为判断语句的条件。如果返回值是数字，则需要根据返回值的情况，把返回值与某一数值进行比较，然后作为判断语句的条件。bcmp 判断结果为相同时，返回值是 0，如果直接作为 if 语句的条件，则表示为假。所以需要同 0 进行比较。

10.4.2 字符串大小比较函数 memcmp

函数 memcmp 用来比较两个字符串的大小是否相同，并且返回第 1 个不相同字符的差值。函数的使用方法如下所示。

```
01   int memcmp (const void *s1,const void *s2,size_t n)
```

参数 s1 与 s2 是需要进行比较的两个字符串，*n* 是需要进行比较的前 *n* 个字符。这里的字符串大小比较是以 ASCII 码表上的顺序来决定的。member 函数首先将字符串 s1 第一个字符值减去 s2 第一个字符的值，若差为 0 则继续比较下个字符，差值不为 0 则返回这个差值。例如字值串“sA”与“sb”，第 1 个字母相同，则比较下一个字母。字符'A(65)'和'b(98)'的差值是-33，函数则返回-33。如果两个字符串的前 *n* 个字母相同，函数返回 0，字符串 s1 小于 s2 则返回小于 0 的值。s1 大于 s2 则返回大于 0 的值。下面是使用这个函数进行字符串比较的实例。

【范例 10-14】使用 memcmp 函数比较两个字符串的大小，示例代码如下所示。

示例代码 10-14

```
01  #include <stdio.h>
02  #include <string.h>                          /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfgQWERT";                  /*定义两个字符串并且赋值。*/
06     char b[20]="asdfgqwERT";
07     int i,j;                                  /*定义两个整型变量存储比较结果。*/
08     i=memcmp(a,b,5);                          /*比较两个字符串的前 5 个字符。*/
09     j=memcmp(a,b,7);                          /*比较两个字符串的前 7 个字符。*/
10     printf("memcpy %s , %s ,5 charactor ,result : %d \n",a,b,i); /*输出结果。*/
11     printf("memcpy %s , %s ,7 charactor ,result : %d \n",a,b,j);
12  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.14.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
memcpy asdfgQWERT , asdfgqwERT ,5 charactor ,result : 0
memcpy asdfgQWERT , asdfgqwERT ,7 charactor ,result : -32
```

【代码解析】两个字符串中第 1 个不相同的字母是 Q 和 q，Q 比 q 的 ASCII 码值小 32。

10.4.3 忽略大小写比较字符串函数 strncasecmp

函数 strncasecmp 可以忽略大小写比较两个字符串，这个函数的使用方法如下所示。

```
01    int strncasecmp(const char *s1,const char *s2,size_t n)
```

注意：此函数的参数与作用与上一节中的 memcmp 函数是相同的，不同点是比较字母时忽略字母的大小写差异。

【范例 10-15】使用 strncasecmp 忽略大小写比较两个字符串的大小，示例代码如下所示。

示例代码 10-15

```
01  #include <stdio.h>
02  #include <string.h>                                  /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfgQWERT";                          /*定义两个字符串并且赋值。*/
06     char b[20]="asdfgqwFRT";
07     int i,j,k;                                        /*定义几个变量存储比较结果。*/
08     i=strncasecmp(a,b,5);                             /*忽略大小写比较前 5 个字符。*/
09     j=strncasecmp(a,b,7);                             /*忽略大小写比较前 7 个字符。*/
10     k=strncasecmp(a,b,8);                             /*忽略大小写比较前 8 个字符。*/
11     printf("strncasecmp %s , %s ,5 charactor ,result : %d \n",a,b,i);   /* 输
出结果。*/
12     printf("strncasecmp %s , %s ,7 charactor ,result : %d \n",a,b,j);
13     printf("strncasecmp %s , %s ,8 charactor ,result : %d \n",a,b,k);
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.16.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
strncasecmp asdfgQWERT , asdfgqwFRT ,5 charactor ,result : 0
strncasecmp asdfgQWERT , asdfgqwFRT ,7 charactor ,result : 0
strncasecmp asdfgQWERT , asdfgqwFRT ,7 charactor ,result : -1
```

【代码解析】忽略大小写的情况下，前 7 个字符是相同的。第 8 个字符 E 与 F 的 ASCII 码差值为 1。

10.5　字符串复制

字符串复制指的是将一个字符串的全部或其中的若干字符，写入另一个字符串中。本节将讲述字符串复制的相关操作，可以进行字符串复制的函数主要有 bcopy、memccpy、strcpy、strncpy 等。

10.5.1　字符串复制函数 bcopy

函数 bcopy 的作用是将一个字符串的前 *n* 个字符复制到另一个字符串中，使用方法如下所示。

```
01   void bcopy ( const void *src,void *dest ,int n);
```

函数的参数 src 是需要复制字符的字符串，dest 是复制到的字符串。*n* 表示需要在字符串中复制的字符数目。函数会改变字符串 dest 的值，没有返回值。下面是这个函数的使用实例。

【范例 10-16】使用 bcopy 函数进行字符串复制，示例代码如下所示。

示例代码 10-16

```
01  #include <stdio.h>
02  #include <string.h>                  /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfgh";              /*定义两个字符串并且赋值。*/
06     char b[20]="ijklmn";
07     printf("1 :%s \n %s\n",a,b);      /*输出原有的字符串。*/
08     bcopy(a,b,4);                     /*第 1 个字符串的前 4 个字符复制到第 2 个字符串
09  中。*/
10     printf("2 :%s \n %s\n",a,b);      /*输出复制以后的结果。*/
11  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.16.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
1 :asdfgh
ijklmn
2 :asdfgh
asdfmn
```

【代码解析】函数 bcopy 会把第 1 个字符串中的前 4 个字母复制到第 2 个字符串的前 4 个字母。第 2 个字符串中的前 4 个字母将会被替换。

10.5.2 字符串复制函数 memccpy

函数 memccpy 可以将一个字符串中的前 *n* 个字节复制到另一个函数中。与函数 bcopy 不同的是，memccpy 可以检查字符串里是不是有某一个字符。该函数的使用方法如下所示。

```
01  void * memccpy(void *dest, const void * src, int c,size_t n)
```

函数参数中的原字符串与目标字符串，与函数 bcopy 的参数相反。dest 是复制的目录字符串的指针，src 需要复制的字符串指针。c 是需要在字符串 dest 中赋值为 c 的字符。如果查找到这个字符，则返回下一个字符的指针。*n* 是需要在字符串 src 中复制的字符的个数。下面的程序是使用这个函数进行字符串替换的实例。

【范例 10-17】使用 memcopy 函数进行字符串复制，示例代码如下所示。

示例代码 10-17

```
01  #include <stdio.h>
02  #include <string.h>                  /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfgh";              /*定义两个字符串并且赋值。*/
06     char b[20]="ijklmn";
07     char *s;                          /*定义一个指向字符的指针。*/
08     printf("%s\n%s\n",a,b);           /*输出两个原字符串。*/
09     s=memccpy(a,b,'k',3);             /*进行字符串替换，查找赋值为 k 的字符。返回
赋 10 值为 k 的下一个指针。*/
11     printf("%c\n",*s);                /*输出返回指针指向的字符。*/
12     printf("%s\n%s\n",a,b);           /*输出替换以后的字符串和原字符串。*/
13  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.17.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfgh
ijklmn
f
ijkfgh
ijklmn
```

【代码解析】代码第 09 行调用 memcopy 完成字符串复制操作，其中参数 a 是复制的目标字符串的指针，参数 b 是需要复制的字符串指针。'k'是需要在字符串 a 中赋值为'k'的字符。如果查找到这个字符，则返回下一个字符的指针。3 是需要在字符串 b 中复制的字符的个数。

10.5.3　复制字符串函数 strcpy

函数 strcpy 可以将一个字符串复制到另一个字符串，函数的使用方法如下所示。

```
01   char *strcpy(char *dest,const char *src)
```

在参数列表中，dest 是复制字符串的目标指针，src 是原字符串指针。函数将返回字符串 dest 的指针。

注意：使用这个函数时，字符串 dest 需要有足够的空间来存储字符串 src，否则将会发生溢出错误。

【范例 10-18】使用 strcpy 函数进行字符串复制，示例代码如下所示。

示例代码 10-18

```
01   #include <stdio.h>
02   #include <string.h>                          /*包含 string.h 头文件。*/
03   main()
04   {
05      char a[20]="asdfgh";                      /*定义一个字符串进行赋值。*/
06      char b[20];                               /*定义一个字符串。*/
07      char *p;                                  /*定义一个指向字符的指针。*/
08      p=strcpy(b,a);                            /*将字符串 a 复制到字符串 b 中。*/
09      printf("%s\n",a);                         /*输出原字符串 a。*/
10      printf("%s\n",b);                         /*输出复制后的字符串 b。*/
11      printf("%s\n",p);                         /*输出指针 p 指向的内容。*/
12   }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.18.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfgh
asdfgh
asdfgh
```

【代码解析】代码第 08 行调用 strcopy 完成字符串复制操作，其中参数 b 是复制字符串的目标指针，a 是原字符串指针。函数将返回字符串 b 的指针。

注意：使用这个函数时，目标字符串 dest 需要有足够的空间来存储字符串 src，否则将会发生溢出错误。

10.5.4 字符串复制函数 strncpy

函数 strncpy 可以将字符串中的若干个字符复制到另一个字符串中，该函数的使用方法如下所示。

```
01  char * strncpy(char *dest,const char *src,size_t n);
```

在参数列表中，dest 是目标字符串的指针，src 是需要复制的字符串的指针。*n* 是需要在字符串 src 中复制的字符个数。返回值是字符串 dest 的头指针。下面是使用 strncpy 进行字符串复制的实例。

【范例 10-19】使用 strncpy 函数进行字符串复制，示例代码如下所示。

示例代码 10-19

```
01  #include <stdio.h>
02  #include <string.h>                  /*包含 string.h 头文件。*/
03  main()
04  {
05     char a[20]="asdfghijk";           /*定义一个字符串并且赋值。*/
06     char b[20]="";                    /*定义一个空字符串。*/
07     char *p;                          /*定义一个指向字符的指针。*/
08     p=strncpy(b,a,5);                 /*将字符串 a 的内容取 5 个字符复制到字符串 b 中。*/
09     printf("%s\n",a);                 /*输出字符串 a。*/
10     printf("%s\n",b);                 /*输出字符串 b。*/
11     printf("%s\n",p);                 /*输出指针 p 指向的字符串。*/
12  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.19.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfghijk
asdfg
asdfg
```

【代码解析】代码第 08 行调用 strncpy 完成字符串复制操作，其中参数 b 是目标字符串的指针，a 是需要复制的字符串的指针。5 是需要在字符串 a 中复制的字符个数。返回值是字符串 b 的头指针。

10.6 字符串的清理与填充

字符串的清理指的是删除一个字符串的部分内容，将所有字节写为 NULL。字符串填充指的是把一个字符串的部分字节写为某一个字符。本节将讲述字符串的这两种操作。

10.6.1　字符串清理函数 bzero

函数 bzero 的作用是将字符串中的部分字节写为 0，即写入 NULL 值。函数的使用方法如下所示。

```
01   void bzero(void *s,int n)
```

参数列表中，s 表示一个字符串的头指针。*n* 表示需要在字符串中清理前 *n* 个字符。函数没有返回址，操作时已经改变了字符串 s 的值。

【范例 10-20】使用 bzero 函数将字符串中的指定字节清零，示例代码如下所示。

示例代码 10-20

```
01   #include <stdio.h>
02   #include <string.h>                    /*包含 string.h 头文件。*/
03   main()
04   {
05      int i;                              /*定义一个循环变量 i。*/
06      char a[20]="asdfghijk";             /*定义一个字符串，赋初值。*/
07      bzero(a,3);                         /*用 bzero 函数将字符串 a 的前 3 个字符清除。*/
08      for(i=0;i<6;i++)                    /*用 for 循环执行 6 次输出。*/
09      {
10         printf("%c\n",a[i]);             /*输出数组中当前的字符。*/
11      }
12   }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.20.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
f
g
h
```

【代码解析】代码第 06 行调用 bzero 函数将前 3 个字符清零，将没有输出内容。后 3 次输出分别为 f、g、h 三个字符。

10.6.2　字符串填充函数 memset

函数 memset 的作用是将一个字符的前 *n* 个字符填充为某一个字符。该函数的使用方法如下所示。

```
01   void * memset (void *s ,int c, size_t n)
```

参数列表中，s 指的是需要处理的字符串头指针。c 是需要写入的字符，*n* 是从字符串第 1 个字符开始需要填充的多少个字符。函数没有返回值。

【范例 10-21】使用 memset 函数填充字符串，示例代码如下所示。

示例代码 10-21

```
01   #include <stdio.h>
02   #include <string.h>                    /*包含 string.h 头文件。*/
03   main()
04   {
05      char a[20]="asdfghijk";             /*定义一个字符串，赋初值。*/
```

```
06      printf("%s\n",a);                    /*输出这个字符串。*/
07      memset(a,'W',3);                     /*将字符串中的前 3 个字符替换成 W。*/
08      printf("%s\n",a);                    /*输出替换以后的字符。*/
09      memset(a,'x',6);                     /*将字符串中的前 3 个字符替换成 x。*/
10      printf("%s\n",a);                    /*输出替换以后的字符。*/
11  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.21.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfghijk
WWWfghijk
xxxxxxijk
```

【代码解析】代码第 07 行调用 memset 函数将字符数组 a 的前三个字符替换为字符 W。

10.7 字符串查找

字符串查找指的是在一个字符串中检索另外一个字符串或字符。如果发现这个字符或字符串，则返回这个字符的指针。本节将讲解字符串查找相关的 index、rindex、strchr、strrchr 等函数。

10.7.1 字符查找函数 index 与 rindex

函数 index 用来在字符串中找出第一个需要查找字符的出现位置，然后将该字符地址返回。rindex 的使用方法与 index 相似，但作用是找出字符串中最后一次某字符的出现位置。这两个函数的使用方法如下所示。

```
01  char *index( const char *s, int c);
02  char *rindex( const char *s, int c);
```

参数列表中，s 表示一个字符串的头指针。c 表示需要在字符串 s 里面查找的字符。返回值是一个指针，指向找到的这个字符的地址。下面是这两个函数的使用实例。

【范例 10-22】使用 index 函数和 rindex 函数进行字符串查找，示例代码如下所示。

示例代码 10-22

```
01  #include <stdio.h>
02  #include <string.h>                      /*包含 string.h 头文件。*/
03  main()
04  {
05      char *p,*q;                          /*定义两个指向字符的指针。*/
06      char a[20]="asdfghasdf";             /*定义一个字符串并且赋值。*/
07      p=index(a,'g');                      /*在字符串中查找字符 g 第一次出现的位置。*/
08      q=rindex(a,'d');                     /*在字符串中查找字符 d 最后一次出现的位置。*/
09      printf("%c\n",*p);                   /*输出指针 p 指向的内容。*/
10      printf("%c\n",*q);                   /*输出指针 q 指向的内容。*/
11  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.22.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
g
d
```

【代码解析】代码中第 07 行与第 08 行调用函数 index 与函数 rindex 在字符数组 a 中搜索指定字符，前者搜索指定字符第一次出现的位置，后者搜索指定字符最后一次出现的位置。

10.7.2　字符查找函数 memchr

函数 memchr 的作用，是在一个字符串的前 *n* 个字符中查找另一个字符，返回这个字符的指针地址。函数的使用方法如下所示。

```
01   void * memchr(const void *s,int c,size_t n)
```

参数列表中，s 是需要查找的字符串，c 是需要查找的字符。*n* 表示在字符串 s 中的前 *n* 个字符里查找。如果找到了这个字符，则会返回这个字符的指针，如果没有这个字符，则返回 0。下面是函数 memchr 的使用实例。

【范例 10-23】使用 memchr 函数进行字符串查找，示例代码如下所示。

示例代码 10-23

```
01   #include <stdio.h>
02   #include <string.h>                    /*包含 string.h 头文件。*/
03   main()
04   {
05      char *p;                            /*定义一个指针字符的指针。*/
06      char a[20]="asdfghasdf";            /*定义一个字符串并且赋值。*/
07      p=memchr(a,'f',10);                 /*在字符串 a 的前 10 个字符中查找字符 f。*/
08      printf("%c\n",*p);                  /*输出指针 p 指向的字符。*/
09   }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.23.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
f
```

【代码解析】代码第 07 行调用函数 memchr 在字符数组 a 的前 10 个字符位置搜索字符 f 出现的位置，并返回其地址，最后打印 memchr 返回地址的内容，由于 memchr 找到字符 f 的地址，在这个地址上的字符一定是 f。

10.7.3　字符查找函数 strchr 与 strrchr

函数 strchr 的作用是在一个字符串中查找另一个字符第一次出现的位置。函数 strrchr 的作用是在一个字符串中查找一个字符最后一次出现的位置。这两个函数的使用方法如下所示。

```
01   char * strchr (const char *s,int c)
02   char * strrchr (const char *s,int c)
```

参数列表中，s 是需要查找的字符串头指针，c 是需要查找的字符。如果查找到这个字符，将返回这个字符指针，如果没有找到这个字符则返回 0。下面是这两个函数的使用实例。

【范例 10-24】使用 strchr 函数和 strrchr 函数进行字符串查找，示例代码如下所示。

示例代码 10-24

```
01  #include <stdio.h>
02  #include <string.h>                          /*包含 string.h 头文件。*/
03  main()
04  {
05     char *p,*q;                               /*定义两个指向字符的指针。*/
06     char a[20]="asdfghasdf";                  /*定义一个字符串并且赋初值。*/
07     p=strchr(a,'s');                          /*在字符串 a 中查找 s 第一次出现的位置。*/
08     q=strrchr(a,'d');                         /*在字符串 a 中查找 d 最后一次出现的位置。*/
09     printf("%c\n",*p);                        /*输出指针 p 指针的字符。*/
10     printf("%c\n",*q);                        /*输出指针 q 指向的字符。*/
11  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.24.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
s
d
```

【代码解析】代码中第 07 行与第 08 行调用函数 strchr 与函数 strrchr 在字符数组 a 中搜索指定字符，前者搜索指定字符第一次出现的位置，后者搜索指定字符最后一次出现的位置。

10.8 字符串的连接与分割

字符串的连接指的是将一个字符串添加到另一个字符串的后面。字符串的分割指的是把一个字符串按照一定的标记切分成多个字符串。本节将讲解字符串的这两种操作。

10.8.1 字符串连接函数 strcat

函数 strcat 的作用，是将一个字符串连接到另一个字符串后面。下面是这个函数的使用方法。

```
01  char *strcat (char *dest,const char *src)
```

参数列表中，dest 和 src 是两个字符串的头指针。函数会把字符串 src 的内容添加到字符串 dest 上面，并返回指针 dest。

注意：字符串 dest 需要有足够的空间来存储字符串 src 的数据。下面是这个函数的使用实例。

【范例 10-25】使用 strcat 函数进行字符串连接，示例代码如下所示。

示例代码 10-25

```
01  #include <stdio.h>
02  #include <string.h>                          /*包含 string.h 头文件。*/
03  main()
04  {
05     char *p;                                  /*定义 1 个指向字符的指针。*/
06     char a[20]="asdfg";                       /*定义 1 个字符串并且赋初值。*/
```

```
07      char b[20]="hjklm" ;
08      printf("%s\n",a);                    /*输出原来的字符串。*/
09      printf("%s\n",b);
10      p=strcat(a,b);                       /*将字符串 b 连接到字符串 a 后面。*/
11      printf("%s\n",a);                    /*输出连接以后的字符串。*/
12      printf("%s\n",b);
13      printf("%s\n",p);                    /*输出指针 p 指向的字符串。*/
14  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.25.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
asdfg
hjklm
asdfghjklm
hjklm
asdfghjklm
```

【代码解析】代码第 10 行调用 strcat 函数将字符串 a 与 b 进行连接，b 的内容为连接后新字符串的值。

10.8.2　字符串分割函数 strtok

函数 strtok 的作用，是将字符串分割成一个个片段。函数的使用方法如下所示。

```
01  char * strtok(char *s,const char *delim);
```

参数列表中，s 是需要分割的字符串，delim 为分割字符串的标记。在第一次调用时，strtok 在参数 s 字符串中发现参数 delim 的分割字符时，将该字符改为 NULL 字符。然后返回更改以后的字符串。再次调用时，将参数 s 设置成 NULL。每次调用成功则返回下一个分割后的字符串指针。下面是这个函数的使用实例，在一个字符串中以 a 为标记，将字符串分割为多个字符串然后输出。

【范例 10-26】使用 strtok 函数进行字符串分割，示例代码如下所示。

示例代码 10-26

```
01  #include <stdio.h>
02  #include <string.h>                      /*包含 string.h 头文件。*/
03  main()
04  {
05     char *p;                              /*定义 1 个指向字符的指针。*/
06     char a[20]="qweaQWEa^*&aIOP";         /*定义 1 个字符串并且赋初值。*/
07     char s[]="a";                         /*定义分割字符串的标记。*/
08     printf("%s\n",a);                     /*输出原字符串。*/
09     p=strtok(a,s);                        /*调用 strtok 函数分割字符串。*/
10     printf("%s\n",p);                     /*输出指针 p 指向的字符串。*/
11     while(p=strtok(NULL,s))               /*用 while 循环来循环输出字符串。*/
12     {
13        printf("%s\n",p);                  /*每次循环输出字符串的结果。*/
14     }
15  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.26.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
qweaQWEa^*&aIOP
qwe
QWE
^*&
IOP
```

【代码解析】程序中函数 strtok 将一个长字符串分割为短字符串分多次输出。

10.9 综合练习

本节讲解一个字符测试编程实例。在程序中常常需要测试输入的数据是否正确，对数据的测试常常是使用字符测试功能来实现的。需要构造一个函数，用循环的方法测试字符串中的每一个字符，然后返回一个测试结果。在这个实例中，构造两个函数，分别测试输入的姓名与电话号码是否正确，然后输出结果。要测试姓名与电话号码，需要考虑以下几个方面的测试内容。

- 姓名的长度，至少需要 3 个字符，一般不会超过 20 个字符。
- 姓名只可以由字母构成，不能出现符号与数字。
- 电话号码的长度，最少需要 3 位，最多不超过 13 位。
- 电话号码中所有的字符必须为数字，出现其他的字符即返回错误。

根据这些要求，编写的程序代码如下所示。

【范例 10-27】测试输入的姓名与电话号码的正确性，示例代码如下所示。

示例代码 10-27

```
#include<ctype.h>          /*包含 ctype.h 头文件。*/
#include<stdio.h>

int istel(char *p)         /*自定义电话号码测试函数，参数为数组的头指针。
*/
{
   int i ;                 /*定义一个变量作为计数器。*/
   i=0 ;                   /*计数器赋初值为 1。*/
   while(*p!=NULL)         /*判断当前指针所指向的字符是不是 NULL。*/
   {
    if(!isdigit(*p))       /*判断指针所指向的字符是不是数字，然后用!取反。
*/
    {
      return 0;            /*如果不是一个数字则返回 0。*/
    }
    i++;                   /*计数加 1。*/
    if(i>13)               /*如果字符总数大于 13 则返回 0。*/
    {
      return 0;            /*如果字符总数大于 13 则返回 0。*/
    }
    p++;                   /*指针向后移动一个字符。*/
   }
   if(i<3)                 /*如果字符的总数小于 3 则返回 0。*/
   {
     return 0;             /*如果字符的总数小于 3 则返回 0。*/
   }
   return 1;               /*如果能执行到这一步，说明是一个电话号码。*/
}
```

```
29
30  int isname(char *p)          /*自定义姓名测试函数，参数为姓名数组的头指针。
31  */
32  {
33     int i ;                   /*定义一个整型变量作为计数器。*/
34     i=0 ;                     /*计数器赋初值为 0。*/
35     while(*p!=NULL)           /*如果当前指向的字符不为 NULL 则向下循环。*/
36     {
37       if(!isalpha(*p))        /*测试当前的字符是不是字母，然后用!取反。*/
38       {
39          return 0;            /*如果当前指向的字符不是字母则返回 0。*/
40       }
41       i++;                    /*指针指向下一个字符。*/
42       if(i>20)                /*如果字符的总数大于 20，则返回 0。*/
43       {
44          return 0;            /*如果字符的总数大于 20，则返回 0。*/
45       }
46       p++;                    /*指针指向下一个字符。*/
47     }
48     if(i<3)                   /*如果字符的总数小于 3，则返回 0。*/
49     {
50       return 0;               /*如果字符的总数小于 3，则返回 0。*/
51     }
52     return 1;                 /*如果能执行到这一步，说明测试正确，返回 1。*/
53  }
54
55  main()                       /*程序的主函数。*/
56  {
57     char name[50];            /*定义一个字符数组存储姓名。*/
58     char tel[50];             /*定义一个字符数组，存储电话。*/
59     char *q;                  /*定义一个指向字符的指针。*/
60
61     printf("please input a telephone number:\n");      /*输出一行提示。*/
62     scanf("%s",tel);          /*从键盘输入一行文本，赋值给数组 tel。*/
63     q=tel;                    /*把数组的头指针赋值给指针 q。*/
64     if(istel(q))              /*把指针作参数，调用函数判断是不是一个电话号
65  码。*/
66     {
67       printf("%s is a telephoe number.\n ",tel);       /*如果为真，输出结果。*/
68     }
69     else
70     {
71       printf("%s is not  a telephoe number.\n ",tel);  /*如果为假，输出错误提示。*/
72     }
73
74     printf("please input a name:\n");                   /*提示输入一个姓名。*/
75     scanf("%s",name);          /*从键盘读取一行文本，赋值给数组 name。*/
76     q=name;                    /*把数组的头指针赋值给指针 q。*/
77     if(isname(q))              /*把指针作参数，调用函数判断是不是一个姓名。
78   */
79     {
80       printf("%s is a name.\n ",name);    /*如果判断结果为真值，则输出结果正确。*/
81     }
82     else
83     {
84       printf("%s is not  a name.\n ",name);  /*另一种情况，输出结果错误。*/
85     }
86  }
```

【运行结果】输入下面的命令编译这段代码。

```
gcc 10.30.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出下面的语句，提示输入一行电话号码。

```
please input a telephone number:
```

这时输入下面的字符，然后按“Enter”键。

```
234456a
```

程序判断这行文本是不是一个电话号码，输出的结果如下所示。因为电话号码中的字符只能是数字，所以判断结果为错误。

```
234567a is not  a telephoe number.
```

用同样的方法，可以对姓名判断进行测试。

10.10 小结

本章讲述了 C 程序中的字符测试、字符串转换、字符串操作等三方面的知识。在 C 程序中常常需要使用这些知识对字符和字符串进行测试和操作。这些知识的使用方法和要点如下所示。

- 字符测试：如果需要对某一个字符进行类型测试，需要用到字符测试函数。字符测试的函数主要有 isalnum、isalpha、isascii、iscntrl、isdigit、isgraphis、islower、isprint 等。
- 字符串转换：本章所讲述的字符串转换，主要是字符串到整型、浮点型、长整型的转换，整型、浮点型、长整型到字符串的转换。常用的函数有 atof、atoi、atol、gcvt、strtod、strtol、strtoul、toascii、tolower 等。
- 字符串处理：本章讲述了字符串连接、分割、复制、查找、比较等字符串操作。主要的函数有 bcopy、bzero、index、strcat、strchr、strncat、strncpy、strrchr、strtok 等。这些函数的参数比较复杂，使用时需要注意各个参数的含义和作用。

10.11 习题

一、填空题

1．所谓_____，是指对一个字符进行大小写、是否可以打印、是否可以显示、是否是数字等方面进行判断。C 程序提供了丰富的字符测试函数，这些函数原型在头文件“ctype.h”中定义，使用这些函数之前需要包含此头文件。

2．_____指的是对字符串进行比较、复制、连接、查找、替换等操作。

3．函数_____ 用来在字符串中找出第一个需要查找字符的出现位置，然后将该字符地址返回。 _____的使用方法与前者相似，但作用是找出字符串中最后一次出现某字符的位置。

4．函数_____的主要作用是将字符串中的部分字节写为 0，即写入 NULL 值。函数_____的作用是将一个字符的前 *n* 个字符填充为某一个字符。

二、选择题

1．常用的字符串复制函数不包括（　　）。

A．bcopy　　B．copy

C．strcpy　　D．strmcpy

2．常用的字符串比较函数不包括（　　）。

A．bcmp　　B．memcmp　　C．strncase　　D．compress

3．字符串清理函数时（　　）。

A．bzero　　B．bcmp　　C．bcopy　　D．memcmp

4．字符串连接与分割函数为（　　）。

A．strchr 与 strrchr　　B．strcat 与 strtok

C．strlen 与 strspn　　D．strcpy 与 strncpy

三、简答题

1．简述常见的字符测试函数及其用法。

2．简述常见的字符串查找函数及其用法。

四、编程题

编程实现一个简单的通讯录管理程序（包括姓名，性别，出生年月等），要求使用本章的字符与字符串函数校验输入内容的合法性。

第 11 章　结　构　体

在编程中经常遇到表示诸如学生基本信息（包括姓名，性别，出生日期等信息）这样的数据的情形，这样的数据是无法用基本数据类型表示的。在 C 语言中可以使用结构体来表示类似这种复杂的数据类型。本章介绍 C 语言中的结构体的定义与基本使用方法，以及结构体与指针的关系，主要内容如下：

- 结构体的概念、定义与访问；
- 结构体数组的使用；
- 结构体与指针。

11.1　结构体的操作

结构体是一种自定义数据类型。结构体有定义、新建、访问等其本操作。本节将讲解结构体的这些操作。在进行本节的学习时，需要正确理解结构体的作用和特点。

11.1.1　结构体的理解

可以用一个实例来理解什么是结构体。假设用下面的变量来描述一个学生的信息。

```
01   char name[15];
02   int age;
03   char address[50];
04   int height;
```

如果有多个学生时，这些数据就无法体现出各自的联系和归属关系。因为每个学生都有相同类别的信息，可以把这些信息封装成一个整体，作为一个新的数据类型。可以用下面的新数据类型来描述一个学生。

```
学生数据类型{
     char name[15];
     int age;
     char address[50];
     int height;
}
```

这样，当用这个类型定义一个学生后，这个学生就有了这个类型里面所有的数据。学生的各个变量与这个学生产生了一对一的联系。从这方面来说，结构体的作用，是把若干个简单的数据类型封装成一个整体，作为一个新的数据类型。这个数据类型可以和普通数据类型一样参与程序的运算。

提示：可以把结构体理解为复合数据类型，它由 C 语言的基本数据类型组合而成，用来在程序中表示更加复杂的数据。

11.1.2　结构体的定义

所谓结构体的定义，是指列出一个结构体中所包括的多个变量和数据类型，建立一个新的

数据类型。结构体的定义方法如下所示。

```
struct 结构名
{
  类型 1  变量名 1;
  类型 2  变量名 2;
  ...
};
```

结构名是新定义的数据类型的名称，结构体内部的变量名被称为这个结构体的成员。例如对上一节中的学生定义一个结构体，代码如下所示。

```
01  struct student                          /*定义一个结构体 student。*/
02  {
03    char name[15];                        /*姓名。*/
04    int age;                              /*年龄。*/
05    char address[50];                     /*住址。*/
06    int height;                           /*身高。*/
07  }
```

如果把这个结构体作为一个新的数据类型，可以用这个数据类型声明新的变量。用结构体 student 声明几个学生，代码如下所示。

```
01  struct  student A;                      /*声明 student 结构体变量 A。*/
02  struct  student Jim,Lily;               /*声明 student 结构体变量 Jim 和 Lily。*/
```

也可以用下面这种方法，直接在定义结构体时声明两个结构变量。

```
01  struct student                          /*定义一个结构体 student。*/
02  {
03    char name[15];                        /*姓名。*/
04    int age;                              /*年龄。*/
05    char address[50];                     /*住址。*/
06    int height;                           /*身高。*/
07  }Jim,Lily;                              /*声明结构体变量 Jim 和 Lily。*/
```

结构体可以作为一个普通的数据类型，于是可以用结构体声明一个数组。例如下面的代码，就是用上面定义的结构体 student 声明 10 个学生的数组。

```
01  struct student stu[10];    /*声明一个含有 10 个元素的 student 结构体数组 stu[10]。*/
```

在一个结构体中，可以引用另外一个结构体，构成更复杂的数据类型。假设需要定义一个班级的结构体，每个班有 1 个班主任和 50 个学生。班主任需要保存姓名，50 个学生可以用 student 结构体定义一个结构体数组，代码如下所示。

```
01  struct class
02  {
03    char teacher[20];                     /*定义一个字符串，表示班主任的姓名。*/
04    struct student stu[50];               /*这是一个结构体数组，表示 50 个学生。*/
05  }
```

可以用这个班级结构体来定义 1 个班，代码如下所示。

```
struct class cla1;
```

这样，班级就是一个数据类型。定义了一个 cla1 班级以后，就可以访问这个班下面的老师、每个学生的变量。例如下面的代码，使用此方法定义学生与班级的结构体。在程序中，需要注意结构体数组的使用。

【范例 11-1】使用结构体表示学生与班级，示例代码如下所示。

示例代码 11-1

```
01  #include <stdio.h>
02
03  struct student                                  /*定义一个结构体。*/
04  {
05     char name[15];                               /*姓名。*/
06     int age;                                     /*年龄。*/
07     char address[50];                            /*住址。*/
08     int height;                                  /*身高。 */
09  };
10
11  struct class                                    /*定义一个班结构体。*/
12  {
13     char teacher[20];                            /*班主任姓名。*/
14     struct student stu[50];                      /*用结构体定义 50 个学生。*/
15  };
16
17  main()                                          /*程序的主函数。*/
18  {
19      struct class A;                             /*用 class 结构体声明一个班。*/
20  }
```

【运行结果】由于本例仅演示了结构体的定义和结构体变量的声明，没有具体的实际意义，所以编译并运行这个程序，不会有运行结果输出。

【代码解析】

- 代码第 03~09 行定义 student 结构体类型，共包含四个成员。
- 代码第 11~14 行定义 class 结构体类型，其中第 14 行定义了一个包含 50 个元素的 student 结构体数组。
- 代码第 19 行声明了一个 class 结构体变量 A。

注意：定义完结构体类型后，就可以像其他的基本数据类型那样，定义新的结构体变量了。在定义新的结构体变量时的格式为“struct 结构体类型名称 结构体变量名”。注意“struct”不能省略。

11.1.3 结构体的访问

结构体是自定义的数据类型，因此结构体变量也可以和其他类型的变量一样赋值、运算。不同的是，结构体本身并不能作为一个变量进行访问，结构变量以成员作为基本变量。结构体成员的表示方式如下所示。

```
结构变量.成员名
```

如果将“结构变量.成员名”看成一个整体，则这个整体的数据类型与结构中该结构成员的数据类型相同，可以同前面所讲的基本变量一样访问。例如下面的代码是结构体的使用实例。先定义一个学生结构体 student，然后用这个结构体新建两个学生变量 Jim 和 Lily，并且赋值输出。

【范例 11-2】结构体成员的访问举例，示例代码如下所示。

示例代码 11-2

```
01  #include <stdio.h>
02
03  struct student                                  /*定义一个学生结构体。*/
04  {
```

```
05      char name[20];                          /*姓名。*/
06      int age;                                /*年龄。*/
07      char sex ;                              /*性别。*/
08      int height;                             /*身高。*/
09  }
10
11  main()
12  {
13      struct student Jim,Lily;                /*用结构体声明两个学生。*/
14      Jim.age=13;                             /*对 Jim 的年龄赋值。*/
15      strcpy(Jim.name,"Jim Green");           /*用 strcpy 函数对 Jim 的姓名进行赋值。*/
16      Jim.sex='m';                            /*对 Jim 的性别进行赋值。*/
17      Jim.height=168;                         /*对 Jim 的身高进行赋值。*/
18
19      Lily.age=13;
20      strcpy(Lily.name,"Lily Kate");
21      Lily.sex='f';
22      Lily.height=174;
23
24      printf("Jim\n");                        /*输出姓名。*/
25      printf(" Name:  %s\n",Jim.name);        /*输出姓名。*/
26      printf(" Age:   %d\n",Jim.age);         /*输出年龄。*/
27      printf(" Sex:   %c:\n",Jim.sex);        /*输出性别。*/
28      printf(" Height:%d:\n",Jim.height );/*输出身高。*/
29
30      printf("Lily\n");                       /*输出 Lily 的信息。*/
31      printf(" Name:  %s\n",Lily.name);
32      printf(" Age:   %d\n",Lily.age);
33      printf(" Sex:   %c\n",Lily.sex);
34      printf(" Height:%d\n",Lily.height );
35  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.2.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Jim
 Name:  Jim Green
 Age:   13
 Sex:   m:
 Height:168:
Lily
 Name:  Lily Kate
 Age:   13
 Sex:   f
 Height:174
```

【代码解析】

- 代码第 03~09 行定义结构体类型 student。
- 代码第 13~12 行完成两个 student 结构体变量的声明和成员的初始化。
- 代码第 24~34 行将两个 student 结构体变量的成员值打印到屏幕上。

注意：结构体成员的访问格式为“结构体变量名.成员名”。

11.1.4 结构体数组

所谓结构体数组，指的是把结构体作为一种基本的数据类型，用这种数据类型定义一个数组。结构体数组的定义和基本数据类型的数组定义方法是相同的。在定义结构体数组之前，需要定义一个结构体。结构体数组在访问时，需要和数组一样用下标表明使用哪一个变量，然后再访问这个结构变量的成员。下面的代码是一个结构体数组的定义与使用实例。

【范例 11-3】结构体数组的定义与使用示例代码如下所示。

示例代码 11-3

```
01  #include <stdio.h>
02
03  struct student                          /*定义一个结构体。*/
04  {
05    char name[15];                        /*姓名。*/
06    int age;                              /*年龄。*/
07    char address[50];                     /*住址。*/
08    int height;                           /*身高。 */
09  };
10
11  main()                                  /*主程序。*/
12  {
13    struct student stu[3];                /*定义一个结构体变量表示 3 个学生。*/
14    stu[1].age=15;                        /*第 2 个学生的年龄为 15。*/
15    stu[2].height=167;                    /*第 3 个学生的身高为 167。*/
16    stu[0].sex='m';                       /*第 1 个学生的性别为 m。*/
17  }
```

【运行结果】由于本例仅演示了结构体的定义和结构体变量成员的赋值，没有具体的实际意义，所以编译并运行这个程序，不会有运行结果输出。

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 13 行声明了包含三个元素的 student 结构体数组。
- 代码第 14~16 行对 student 结构体数组元素的成员赋值。

提示：结构体数组与普通的数组的概念和使用方法完全相同，只不过数组元素为结构体变量而已。

11.1.5 结构体使用实例

在程序中使用结构体以后，可以使程序复杂的数据关系构成逻辑联系。使用这种联系可以方便地访问一组数据。本节将讲解一个结构体使用实例。先定义一个表示学生信息的结构体，然后用这个结构体数组定义若干个学生。从键盘输入信息以后，输入学生信息。然后用循环的方法输出学生的信息。要实现这个程序，需要在程序中完成下面这些操作。

（1）定义一个用来表示学生的结构体。

（2）定义一个结构体数组，用来存放若干学生的信息。

（3）定义一个字符变量。用户输入“y”或“Y”时，继续输入学生数据。用户输入“n”或“N”时，结束输入信息。

（4）定义一个整型变量，用来存储当前学生的序号。

（5）用 while 语句完成输入的循环。
（6）用 for 语句循环输出学生的信息。
根据上面的思路，编写的程序代码如下所示。

【范例 11-4】用结构体数组保存录入的学生基本信息，示例代码如下所示。

示例代码 11-4

```
#include <stdio.h>

struct student                              /*定义一个结构体。*/
{
   char name[20];                           /*姓名。*/
   int age;                                 /*年龄。*/
   int sex;                                 /*性别。*/
   int height;                              /*身高。 */
};

main()
{
  struct student stu[50];                   /*定义一个结构体，表示有 50 个学生。*/
  int i=0;                                  /*输入学生的计数。*/
  int j ;                                   /*输出的计数。*/
  char s;                                   /*用户输入数据。*/
  char name[20];                            /*姓名。*/
  while(1)                                  /*进入一个 while 循环。*/
  {
    printf("input a student?\n?Y/N:");      /*提示用户输入选择。*/
    scanf("%c",&s);                         /*用户输入一个字符。*/

    if(s=='N'||s=='n')                      /*输入 N 或 n 则中断循环。*/
    {
      break;                                /*中止当前循环。*/
    }

    if(s!='n'&&s!='N'&&s!='y'&&s!='Y')      /*如果不是 N、n、Y、y 则进入下一次循环。*/
    {
      printf("error.\n");                   /*输出提示。*/
      continue;                             /*进入下一次循环。*/
    }

    printf("please input age:\n");          /*提示输入年龄。*/
    scanf("%d",&stu[i].age);                /*输入年龄。*/
    printf("please input height:\n");       /*提示身高。*/
    scanf("%d",&stu[i].height);             /*输入身高。*/
    printf("please input sex:\n");          /*提示性别。*/
    scanf("%d",&stu[i].sex);                /*输入性别。*/
    printf("please input name:\n");         /*提示姓名。*/
    scanf("%s",name);                       /*输入姓名到 name 字符串。*/
    strcpy(stu[i].name,name) ;              /*将字符串复制到学生姓名上。*/
    i++;                                    /*计数器自加。*/
  }

  for(j=0;j<i;j++)                              /*for 循环输出。*/
  {
    printf("Student[%d]\n",j);                  /*提示第几个学生。*/
    printf("  Name  :%s  \n",stu[j].name);      /*输出姓名。*/
```

```
50      printf("  Age   :%d  \n",stu[j].age);          /*输出年龄。*/
51      printf("  Sex   :%d  \n",stu[j].sex);          /*输出性别。*/
52      printf("  Height:%d  \n",stu[j].height);       /*输出身高。*/
53    }
54  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.4.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序输出下面的提示，询问是不是输入一个学生。

```
input a student?
?Y/N:
```

这时输入一个“Y”，然后在各项提示的后面输入学生的信息。输入完一个学生的信息后，输入“Y”继续输入下一个学生的信息。输入“N”则停止输入信息，然后循环输出已经输入的内容。显示的结果如下所示。

```
Student[0]
  Name  :Jim
  Age   :16
  Sex   :1
  Height:168
Student[1]
  Name  :Lily
  Age   :18
  Sex   :0
  Height:175
Student[2]
  Name  :Bush
  Age   :15
  Sex   :1
  Height:178
Student[3]
  Name  :Lucy
  Age   :19
  Sex   :0
  Height:175
```

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 18~44 行使用 while 循环实现输入多个学生信息并保存在 student 结构体数组中。其中代码第 23 ~26 行使用 if 判断用户的输入是否符合退出循环的条件，如果输入的是“N”或“n”，则退出 while 循环。
- 代码第 46~53 行打印 student 结构体数组所有元素值。

提示：当程序中要表达复杂的事物时（例如：学生基本信息，员工基本信息等），使用结构体是一种比较简单和方便的途径。

11.2 结构体与指针

结构体是一种特殊的数据类型，可以定义一个指向结构体的指针。这种指向结构体的指针就是结构体指针。结构体指针和普通指针一样，可以访问结构体的成员，也可以作为函数的参

数和返回值。本节将讲解结构体指针的操作。

11.2.1　结构体指针的定义

所谓结构体指针的定义，指的是定义一个指针，这个指针指向一个结构体类型的变量。用这个指针可以访问一个结构体中的成员。结构体指针的定义和普通指针的定义方法是相同的，在变量名前面加一个星号即可。下面的代码利用前面的学生结构体定义一个结构体指针。

```
01  struct student                          /*定义一个结构体。*/
02  {
03    char name[20];                        /*姓名。*/
04    int age;                              /*年龄。*/
05    int sex;                              /*性别。*/
06    int height;                           /*身高。 */
07  };
08  struct student *stu;                    /*定义一个结构体指针 stu。*/
```

注意：结构体指针与普通的指针概念完全相同，只不过它所指向的内存地址空间类型为结构体。

11.2.2　结构体指针的访问

所谓结构体指针的访问，是指用结构体指针访问这个指针所指向的变量和成员，对这些变量和成员进行赋值或读取的操作。有两种方法访问结构体指针。可以用下面的方法来理解。

（1）假设使用上一节定义的结构体，定义一个结构体指针，代码如下所示。

```
01  struct student *stu;
```

（2）用星号（*）可以访问这一个结构体，代码如下所示。

```
01  *p;
```

（3）既然*p 是一个结构体，则可以使用这个结构体直接访问成员变量，代码如下所示。

```
01  (*p).age = 15;
02  (*p).sex = 0 ;
```

（4）(*p).age 这个成员变量可以写成另一种形式，用一个箭头表示指针所指向的变量，代码如下所示。这些代码和上面的代码是等价的。

```
01  p->age=15;
02  p->sex=0;
```

根据所述的结构体变量的访问方法，可以在程序中用结构体指针来访问一个结构体。下面是指针访问结构体的实例。

注意：当使用星号访问结构体指针变量所表示的结构体的成员时必须使用括号，例如：(*p).age=15。

【范例 11-5】使用指针访问结构体，示例代码如下所示。

示例代码 11-5

```
01  #include <stdio.h>
02  #include <stdlib.h>              /*因为要用 strcpy 函数，所以要包含头文件 stdlib.h。
03  */
04
05  struct student                   /*定义一个结构体。*/
```

```
{
   char name[20];                          /*姓名。*/
   int age;                                /*年龄。*/
   int sex;                                /*性别。*/
   int height;                             /*身高。 */
};

main()
{
   struct student stu;                     /*定义一个结构体变量。*/
   struct student *p;                      /*定义一个指向结构体变量的指针。*/

   p=&stu;                                 /*结构体取地址赋值给结构体指针。*/
   stu.age=15;                             /*访问结构体的成员，对年龄赋值。*/
   strcpy(stu.name,"Jim");                 /*用 strcpy 函数对姓名赋值。*/

   (*p).sex =1;                            /*用结构体指针访问性别成员变量。*/
   (*p).height=168;                        /*用结构体指针访问身高成员变量。*/

   printf("Name  : %s \n",stu.name);       /*输出这些成员变量的值。*/
   printf("Age   : %d \n",stu.age);
   printf("Sex   : %d \n",stu.sex);
   printf("Height: %d \n\n",stu.height);

   p->height=169;                          /*用箭头方法访问成员变量。*/
   p->sex=0;

   printf("Name  : %s \n",stu.name);       /*输出结构体的成员。*/
   printf("Age   : %d \n",stu.age);
   printf("Sex   : %d \n",stu.sex);
   printf("Height: %d \n\n",stu.height);

   printf("Name  : %s \n",(*p).name); /*用指针访问结构体变量。*/
   printf("Age   : %d \n",(*p).age);
   printf("Sex   : %d \n",p->sex);     /*用箭头符号代替结构体指针访问结构体成员。*/
   printf("Height: %d \n\n",p->height);
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.5.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。从结果可知，结构体指针对成员的访问与结构体直接访问成员的作用是相同的。

```
Name  : Jim
Age   : 15
Sex   : 1
Height: 168

Name  : Jim
Age   : 15
Sex   : 0
Height: 169

Name  : Jim
Age   : 15
Sex   : 0
Height: 169
```

提示：箭头符号和指针星号对结构体成员的访问作用是相同的。

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 22~23 行和代码第 30~31 行分别使用结构体指针和箭头符号访问结构体变量，为结构体变量赋值。
- 代码第 25~28 行和代码第 33~36 行分别使用结构体指针和箭头符号打印结构体变量的值。代码第 38~41 行则混合使用结构体指针和箭头符号打印结构体变量的值。

11.2.3　结构体作为函数的参数

结构体作为一个特殊的数据类型，是可以作为函数的参数的。使用结构使用变量作为函数参数时，需要在参数列表中声明这个函数的参数是一个结构体。在函数中可以直接调用这个结构体和成员变量。调用函数时，需要将结构体变量作为函数的参数。下面的程序是结构体作为函数参数的实例，实现一个学生的输出功能。

【范例 11-6】结构体作为函数的参数输出学生信息，示例代码如下所示。

示例代码 11-6

```
01  #include <stdio.h>
02
03  struct student                          /*定义一个结构体。*/
04  {
05     char name[20];                       /*姓名。*/
06     int age;                             /*年龄。*/
07     int sex;                             /*性别。*/
08     int height;                          /*身高。 */
09  };
10
11  void showstu(struct student s)          /*定义一个自定义函数，结构体变量作为参数。*/
12  {
13     printf("A student:\n");                  /*输出信息。*/
14     printf("  Name   : %s \n",s.name);       /*访问和输出参数中结构体的变量成员。*/
15     printf("  Age    : %d \n",s.age);        /*访问和输出参数中结构体的变量成员年龄。*/
16     printf("  Sex    : %d \n",s.sex);        /*访问和输出参数中结构体的变量成员性别。*/
17     printf("  Height: %d \n\n",s.height);/*访问和输出参数中结构体的变量成员身高。*/
18  }
19
20  void main()                                 /*主函数。*/
21  {
22     struct student stu1,stu2;                /*定义两个结构体变量，表示两个学生。*/
23
24     stu1.age=17;                             /*对第一个结构体变量的成员进行赋值。*/
25     stu1.sex=1;
26     stu1.height=176;
27     strcpy(stu1.name,"Jim");                 /*strcpy 函数对字符串进行赋值。*/
28     stu2.age=23;                             /*对第二个结构体变量的成员进行赋值。*/
29     stu2.sex=0;
30     stu2.height=171;
31     strcpy(stu2.name,"Lily");                /*strcpy 函数对字符串进行赋值。*/
32
33     showstu(stu1);                           /*调用函数输出结构体 stu1 的信息。*/
34     showstu(stu2);                           /*调用函数输出结构体 stu2 的信息。*/
```

```
35  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.6.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 176
A student:
  Name  : Lily
  Age   : 23
  Sex   : 0
  Height: 171
```

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 11~18 行定义函数 showstu，形式参数为 student 结构体变量，函数体则打印传递进来的 student 结构体的成员。
- 代码第 22 行定义 student 结构体变量 stu1 和 stu2，代码第 24~31 行则为这两个结构体变量的成员赋值。
- 代码第 33~34 行调用 showstu 函数打印结构体变量 stu1 和 stu2 成员的值。

11.2.4 结构体指针作为函数的参数

既然结构体可以作为函数的参数，结构体指针可以访问一个结构体变量，也可以作为函数的参数。当结构体指针作为函数的参数时，这个指针可以访问结构体的成员，对成员进行赋值或读取操作。下面的程序是使用结构体指针作为函数参数的实例。在程序中，自定义一个函数，结构体指针作为这个函数的参数，函数输出这个指针的信息。

【范例 11-7】结构体指针作为函数参数输出学生信息，示例代码如下所示。

示例代码 11-7

```
01  #include <stdio.h>
02
03  struct student                          /*定义一个结构体。*/
04  {
05    char name[20];                        /*姓名。*/
06    int age;                              /*年龄。*/
07    int sex;                              /*性别。*/
08    int height;                           /*身高。 */
09  };
10
11  void showstu(struct student *p)         /*定义一个自定义函数，结构体指针作为参数。*/
12  {
13    printf("A student:\n");               /*输出信息。*/
14    printf("  Name  : %s \n",s.name);     /*访问和输出参数中结构体指针指向的成员。*/
15    printf("  Age   : %d \n",s.age);  /*访问和输出参数中结构体指针指向的成员年龄。
16  */
17    printf("  Sex   : %d \n",s.sex);  /*访问和输出参数中结构体指针指向的成员性别。
18  */
```

```
19      printf("  Height: %d \n\n",s.height);/*访问和输出参数中结构体指针指向的成员身高。
20    */
21    }
22
23    void main()                                    /*主函数。*/
24    {
25      struct student stu1,stu2;                    /*定义两个结构体变量，表示两个学生。*/
26      struct student *p1,*p2;                      /*定义两个结构体指针，指向学生结构体。*/
27
28      p1=&stu1;                                    /*第一个学生取地址各个领域给指针 p1。*/
29      p2=&stu2;                                    /*第二个学生取地址各个领域给指针 p2。*/
30      stu1.age=17;                                 /*对第一个结构体变量的成员进行赋值。*/
31      stu1.sex=1;
32      stu1.height=176;
33      strcpy(stu1.name,"Jim");                     /*strcpy 函数对字符串进行赋值。*/
34      stu2.age=23;                                 /*对第二个结构体变量的成员进行赋值。*/
35      stu2.sex=0;
36      stu2.height=171;
37      strcpy(stu2.name,"Lily");                    /*strcpy 函数对字符串进行赋值。*/
38
39      showstu(p1);                                 /*调用函数输出指针 p1 的信息。*/
40      showstu(p2);                                 /*调用函数输出指针 p2 的信息。*/
41    }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，然后程序显示的结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 176
A student:
  Name  : Lily
  Age   : 23
  Sex   : 0
  Height: 171
```

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 11~21 行定义函数 showstu，形式参数为 student 结构体指针变量，函数体则打印传递进来的 student 结构体指针变量所指向结构体的成员的值。
- 代码第 25 行和第 26 行分别定义 student 结构体变量 stu1 和 stu2 和结构体指针变量 p1 和 p2，代码第 28~29 行将结构体变量 stu1 和 stu2 的地址赋给结构体指针变量 p1 和 p2。代码第 30~37 行为结构体变量 stu1 和 stu2 的成员赋值。
- 代码第 33~34 行调用 showstu 函数打印结构体变量 stu1 和 stu2 成员的值。

提示：在代码 11-7 中由于将结构体变量的地址赋给了结构体指针，所以分别通过结构体变量 stu1 和 stu2 访问结构体成员与分别通过结构体指针 p1 和 p2 访问结构体成员是等价的。

11.2.5 结构体作为函数的返回值

自定义函数中的结构体，可以作为返回值返回给主函数。结构体返回值的处理方法，与普通变量的处理方法是相同的。在主函数中，需要有一个结构体来接收函数返回的结构体。下面的代码是自定义一个函数，函数要求用户输入一个用户的信息。然后将这个学生以结构体的形式返回到主函数中。主函数调用前面章节中的显示学生信息函数输出这个学生。程序代码如下所示。

提示：如果函数的返回值为结构体，则可以直接使用此函数为结构体变量赋值。

【范例 11-8】结构体作为函数的返回值返回学生信息，示例代码如下所示。

示例代码 11-8

```
#include <stdio.h>

struct student                          /*定义一个结构体。*/
{
   char name[20];                       /*姓名。*/
   int age;                             /*年龄。*/
   int sex;                             /*性别。*/
   int height;                          /*身高。 */
};

void showstu(struct student s)          /*定义一个自定义函数，结构体变量作为参数。*/
{
   printf("A student:\n");                  /*输出信息。*/
   printf("  Name  : %s \n",s.name);        /*访问和输出参数中结构体的变量成员。*/
   printf("  Age   : %d \n",s.age);         /*访问和输出参数中结构体的变量成员年龄。*/
   printf("  Sex   : %d \n",s.sex);         /*访问和输出参数中结构体的变量成员性别。*/
   printf("  Height: %d \n\n",s.height);    /*访问和输出参数中结构体的变量成员身高。*/
}

struct student getstu(void)                 /*定义一个函数，结构体作为函数的返回值。*/
{
    struct student stu;                     /*定义一个结构体。*/
    char name[20];                          /*定义一个字符串，存储姓名。*/
    printf("please input age:\n");          /*提示输入年龄。*/
    scanf("%d",&stu.age);                   /*输入年龄。*/
    printf("please input height:\n");       /*提示身高。*/
    scanf("%d",&stu.height);                /*输入身高。*/
    printf("please input sex:\n");          /*提示性别。*/
    scanf("%d",&stu.sex);                   /*输入性别。*/
    printf("please input name:\n");         /*提示姓名。*/
    scanf("%s",name);                       /*输入姓名到 name 字符串。*/
    strcpy(stu.name,name) ;                 /*将字符串复制到学生姓名上。*/
    return(stu);                            /*返回这个结构体。*/
}

void main()                                 /*主函数。*/
{
   struct student stu1,stu2,stu3;           /*定义三个结构体，表示三个学生。*/
   stu1=getstu();                           /*用函数 getstu()输入 stu1 的信息。*/
   stu2=getstu();                           /*用函数 getstu()输入 stu2 的信息。*/
```

```
41      stu3=getstu();              /*用函数 getstu()输入 stu3 的信息。*/
42      showstu(stu1);              /*结构体作为参数，调用 showstu 输出 stu1 的信息。
43  */
44      showstu(stu2);              /*结构体作为参数，调用 showstu 输出 stu2 的信息。
45  */
46      showstu(stu3);              /*结构体作为参数，调用 showstu 输出 stu3 的信息。
47  */
48  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.8.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，然后程序显示的结果如下所示。

```
A student:
  Name  : Jim
  Age   : 17
  Sex   : 1
  Height: 178
A student:
  Name  : Lily
  Age   : 21
  Sex   : 1
  Height: 174
A student:
  Name  : Lucy
  Age   : 22
  Sex   : 0
  Height: 171
```

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 11~18 行定义函数 showstu，形式参数为 student 结构体变量，函数体则打印传递进来的 student 结构体的成员。
- 代码第 20~34 行定义了返回值为 student 结构体的函数 getstu，在函数内部则调用 scanf 函数用于获取从命令行中输入结构体成员的值。
- 在主函数中，首先定义了三个 student 结构体类型的结构体变量，然后调用 getstu 函数为这三个结构体变量的成员赋值，最后调用 showstu 函数打印这三个结构体变量的成员的值。

11.2.6　结构体指针作为函数的返回值

既然结构体可以作为函数的返回值，结构体指针作为一种特殊的数据类型也可以作为函数的返回值。当一个函数需要用结构体作为返回值时，需要在函数中输入一个指针，函数对指针进行处理，然后返回这个指针。在函数中，可以用这个指针来访问指针指向的主函数中的结构体。下面是一个使用结构体指针作函数返回值的实例。函数利用这个指针输入学生的信息，然后返回该学生结构体的指针。程序通过这个指针取值调用学生显示函数输出学生的信息。

提示：如果函数的返回值为结构体指针，则可以直接使用此函数为结构体指针变量赋值。

【范例 11-9】结构体指针作为函数的返回值返回学生信息，示例代码如下所示。

示例代码 11-9

```
#include <stdio.h>

struct student                          /*定义一个结构体。*/
{
   char name[20];                       /*姓名。*/
   int age;                             /*年龄。*/
   int sex;                             /*性别。*/
   int height;                          /*身高。 */
};

void showstu(struct student s)          /*定义一个自定义函数，结构体变量作为参数。*/
{
   printf("A student:\n");              /*输出信息。*/
   printf("  Name  : %s \n",s.name);        /*访问和输出参数中结构体的变量成员。*/
   printf("  Age   : %d \n",s.age);         /*访问和输出参数中结构体的变量成员年龄。*/
   printf("  Sex   : %d \n",s.sex);         /*访问和输出参数中结构体的变量成员性别。*/
   printf("  Height: %d \n\n",s.height);/*访问和输出参数中结构体的变量成员身高。*/
}

struct student *getstu(struct student  *s)  /*自定义一个函数，*/
{
    char name[20];                          /*定义一个字符串，存储姓名。*/
    printf("please input age:\n");          /*提示输入姓名。*/
    scanf("%d",&(*s).age);                  /*用指针访问结构体的变量。*/
    printf("please input height:\n");       /*提示输入身高。*/
    scanf("%d",&(*s).height);               /*用指针访问结构体变量的成员。*/
    printf("please input sex:\n");          /*提示输入性别。*/
    scanf("%d",&(*s).sex);
    printf("please input name:\n");         /*提示输入姓名。*/
    scanf("%s",name);                       /*输入一个字符串，赋值给 name。*/
    strcpy(s->name,name) ;          /*用 strcpy 函数把字符串复制到结构体 name 成员
上。*/
    return(s);                      /*返回这个指针。*/
}

void main()                         /*主函数。*/
{
   struct student s1,s2,s3;             /*用结构体定义三个学生。*/
   struct student *stu1,*stu2,*stu3;    /*定义三个结构体类型的指针。*/
   stu1=getstu(&s1);                    /*对第一个学生取地址作为参数，调用函数。*/
   stu2=getstu(&s2);                    /*对第二个学生取地址作为参数，调用函数。*/
   stu3=getstu(&s3);
   showstu(*stu1);                      /*用第一个指针取结构体变量作为参数显示。*/
   showstu(*stu2);                      /*用第二个指针取结构体变量作为参数显示。*/
   showstu(*stu3);                      /*用第三个指针取结构体变量作为参数显示。*/
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 11.9.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

在程序的提示信息后面输出学生信息，然后程序显示的结果如下所示。

```
A student:
  Name  : Jim
```

```
  Age   : 17
  Sex   : 1
  Height: 178
A student:
  Name  : Lily
  Age   : 21
  Sex   : 0
  Height: 172
A student:
  Name  : Brucs
  Age   : 23
  Sex   : 0
  Height: 177
```

【代码解析】

- 代码第 03~09 行定义 student 结构体类型。
- 代码第 11~18 行定义函数 showstu，形式参数为 student 结构体变量，函数体则打印传递进来的 student 结构体的成员。
- 代码第 20~34 行定义了返回值为 student 结构体指针的函数 getstu，形式参数为 student 结构体指针，在函数内部则调用 scanf 函数用于获取从命令行中输入结构体成员的值。
- 在主函数中，首先定义了三个 student 结构体类型的结构体变量 s1、s2 和 s3，以及三个 student 结构体指针变量 stu1、stu2 和 stu3，然后调用 getstu 函数为这三个结构体变量的成员赋值（参数为 s1、s2 和 s3 的地址），将 getstu 的返回值分别赋给结构体指针 stu1、stu2 和 stu3，最后调用 showstu 函数打印这三个结构体变量的成员的值。

11.3　综合练习

结构体在程序中有着非常重要的应用。在复杂的程序中，需要定义结构体作为新的数据类型，用结构体数据类型作为函数的参数与返回值。本节将讲解一个结构体运用实例，在程序中用结构体进行各种数据的存储。

11.3.1　程序的需求分析

本节将讲解一个学生管理程序。在程序中，用结构体进行学生数据的存储，结构体作为函数的参数与返回值。程序需要完成下面的功能。

（1）菜单选择功能。进入程序以后，显示一个菜单，用户在键盘上输入一个数字选择相应的功能。

（2）学生添加功能。用户选择学生添加菜单项以后，可以在文字提示下输入一个学生的信息。

（3）学生姓名列表功能。这个功能是列出所有学生的编号和姓名。

（4）所有学生详细列表功能。

（5）按姓名查询功能。

（6）按年龄查询功能。输入一个年龄段以后，显示这个年龄段所有的学生信息。

（7）按性别查询功能。输入一个性别以后，显示这个性别所有的学生信息。

（8）按身高查询功能。输入一个身高区域以后，显示这个区域中所有学生的信息。

（9）删除学生功能。输入一个学生的姓名，在结构体数组中删除这个学生。

（10）程序退出功能。选择这个选项以后，结束程序。

11.3.2 程序中的函数

实例程序中的每个功能是分成不同的函数来完成的，每一个函数根据需要完成一定的功能。这样就可以把一个复杂的程序分为不同的函数模块。各个功能不同的函数在主函数的调用下，统一地完成一个复杂的功能。下面是这个程序中的各个自定义函数。

- 显示一个学生的函数，参数是一个结构体。

```
void showstu(struct student s)
```

- 删除一个学生的函数，没有参数。

```
void delestu();
```

- 菜单函数，完成菜单的显示和选择。

```
int menu();
```

- 读取学生函数，完成一个学生信息的输入，返回一个结构体。

```
struct student getstu(void);
```

- 按学生姓名查找函数。

```
void selectbyname();
```

- 按年龄查找函数。

```
void selectbyage();
```

- 按性别查找函数。

```
void selectbysex();
```

- 按身高查找函数。

```
void selectbyheight();
```

- 主函数，完成程序中各个函数的组织和调用。

```
int  main()
```

11.3.3 程序中的结构体与全局变量

所谓全局变量，指的是程序中所有的函数都可以访问的变量。可以把变量定义语句写在程序最前面，这样在所有的函数中都可以调用这些变量。在本程序中，学生的结构体数组与学生人数的计数都是全局变量。结构体变量是这样定义的。

```
01   int i;
02   struct student stu[100];
```

11.3.4 头文件和函数声明

本节以后的多个章节，讲解程序不同的模块。本程序在光盘中的目录是“/源文件/11/11.10.c”。C 程序中的代码可以分解为多个函数。在学习时把这些函数写在一种程序中即可运行。

在程序的最前面，需要包含程序中需要使用的头文件。程序中自定义的函数，当前面的函数调用后面的函数时，可能找不到这个函数。于是需要在程序的最前面声明这些函数。包含文件和声明函数的代码如下所示。

提示：在进行 C 程序开发时，通常都将所要使用的头文件包含指令和自定义函数的原型写在程序的开头。

【范例 11-10】模块 1：包含文件与函数声明，示例代码如下所示。

示例代码 11-10

```
#include <stdio.h>
#include <stdlib.h>                                    /*包含 stdlib.h 头文件。*/
void delestu();                                        /*声明程序中所有的自定义函数。*/
int menu();
struct student getstu(void);
void selectbyname();
void selectbyage();
void selectbysex();
void selectbyheight();
void showstu(struct student s);
```

11.3.5　定义结构体和全局变量

声明变量以后，需要在程序中定义表示学生的结构体。然后用这个结构体类型定义一个结构体数组，保存若干个学生。需要有一个变量 i 来保存当前一共有多少个学生。这些变量是在主函数以外定义的，所有的函数都可以访问。这一模块的代码如下所示。

【范例 11-11】模块 2：结构体与变量的定义，示例代码如下所示。

示例代码 11-11

```
struct student                  /*定义一个结构体。*/
{
   char name[20];               /*姓名。*/
   int age;                     /*年龄。*/
   int sex;                     /*性别。*/
   int height;                  /*身高。 */
};
int i;                          /*进行一个全局变量，用于学生的计数。*/
struct student stu[100];        /*定义一个结构体数组，最多可以存储 100 个学生。10    */
```

11.3.6　显示学生信息的函数

这个函数的功能是显示一个学生的详细信息。函数的参数是一个学生类型的结构体。函数中通过参数中的结构体，访问结构体的成员，输出详细信息。函数完成输出以后没有返回值。这一模块的代码如下所示。

【范例 11-12】模块 3：显示学生信息函数，示例代码如下所示。

示例代码 11-12

```
void showstu(struct student s)          /*定义一个自定义函数，结构体变量作为参数。*/
{
   printf("A student:\n");              /*输出信息。*/
   printf("  Name   : %s \n",s.name);   /*访问和输出参数中结构体的变量成员。*/
   printf("  Age    : %d \n",s.age);    /*访问和输出参数中结构体的变量成员年龄。*/
   printf("  Sex    : %d \n",s.sex);    /*访问和输出参数中结构体的变量成员性别。*/
   printf("  Height: %d \n\n",s.height);/*访问和输出参数中结构体的变量成员身高。*/
}
```

11.3.7　程序的选择菜单

这个函数的功能是显示所有的功能作为菜单，用户从键盘输入一个编号。如果这个编号有效，则返回这个编号。如果编号为 9，则退出程序。如果编号无效，则显示错误，再次显示菜

单。函数的运行流程如图 11-1 所示。

注意：菜单决定了用户如何使用程序的功能，所以菜单的定义要清晰明了，方便用户使用。

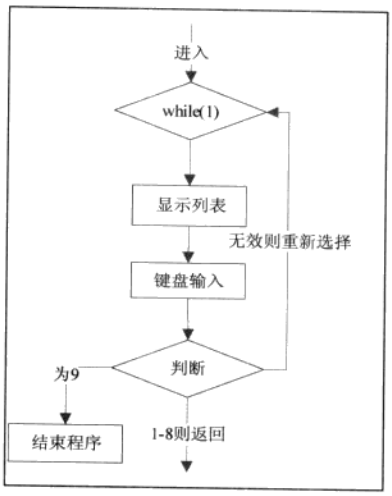


图 11-1　menu 函数的执行流程

【范例 11-13】模块 4：菜单函数，示例代码如下所示。

示例代码 11-13

```
int menu()
{
  int i;                                    /*定义变量，这个就是只有这个函数有 04    效。*/
  i=0;                                      /*赋初值。*/
  while(1)                                  /*进入一个循环。*/
  {
    printf("Please select a menu:\n");                          /*提示输入。*/
    printf("      1: add a student.\n");                        /*添加学生选项。*/
    printf("      2: list the name all the student.\n");/*显示所有学生选项。*/
    printf("      3: list informations of all the student.\n");  /*显示学生
详细信息选项。*/
    printf("      4: select a student by name .\n");    /*按姓名查找选项。*/
    printf("      5: select students  by age .\n");     /*按年龄查找。*/
    printf("      6: select students  by sex .\n");     /*按性别查找。*/
    printf("      7: select students  by height .\n");  /*按身高查找。*/
    printf("      8: delete a student .\n");    /*删除一个学生。*/
    printf("      9: exit .\n");                /*退出。*/
    scanf("%d",&i);                             /*输入一个变量。*/
    if(i==9)                                    /*如果输入为 9 则退出程序。*/
    {
      printf("Byebye.\n");                      /*输出信息。*/
      exit(1);                                  /*用 exit 函数退出程序。*/
    }
    if(i<1 || i>9)                              /*输入的数字不正确。*/
    {
      printf("error.\n");                       /*提示错误。*/
```

```
27          continue;                                   /*进行下一次选择。*/
28        }
29        else
30        {
31          return(i);                                  /*输入正确则返回这个变量。*/
32        }
33      }
34  }
```

11.3.8　学生信息输入函数

这个函数的功能是提示用户从键盘输入一个学生的信息。然后把这个学生信息的结构体返回。这个函数没有输入参数，代码如下所示。需要注意的是，学生姓名的输入是先把学生姓名输入到另一个字符串上，然后用 strcpy 函数把这个字符串复制到结构体的姓名成员变量中。

【范例 11-14】模块 5：输入学生信息，示例代码如下所示。

示例代码 11-14

```
01  struct student getstu(void)                        /*定义一个函数，结构体作为函数的返回值。*/
02  {
03      struct student stu;                             /*定义一个结构体。*/
04      char name[20];                                  /*定义一个字符串，存储姓名。*/
05      printf("please input age:\n");                  /*提示输入年龄。*/
06      scanf("%d",&stu.age);                           /*输入年龄。*/
07      printf("please input height:\n");               /*提示身高。*/
08      scanf("%d",&stu.height);                        /*输入身高。*/
09      printf("please input sex:\n");                  /*提示性别。*/
10      scanf("%d",&stu.sex);                           /*输入性别。*/
11      printf("please input name:\n");                 /*提示姓名。*/
12      scanf("%s",name);                               /*输入姓名到 name 字符串。*/
13      strcpy(stu.name,name) ;                         /*将字符串复制到学生姓名上。*/
14      return(stu);                                    /*返回这个结构体。*/
15  }
```

11.3.9　按姓名查找函数

这个函数的功能是按照学生的姓名查找出一个学生的详细信息。函数中需要输入一个学生的姓名，然后用 strcmp 函数比较这个字符串与学生结构体数组中的学生姓名是否相同。如果姓名相同，则调用 showstu 函数输出这个学生的信息。

【范例 11-15】模块 6：按姓名查找学生，示例代码如下所示。

示例代码 11-15

```
01  void selectbyname()
02  {
03    int j,n;                                          /*定义两个计数器变量。*/
04    char name[20];                                    /*定义一个字符串。*/
05    n=0;                                              /*计数变量 n 赋值为 0。*/
06    printf("please input a name;\n");                 /*提示输入姓名。*/
07    scanf("%s",name);                                 /*输入一个姓名。*/
08    for(j=0;j<i;j++)                                  /*for 循环，访问学生数组中所有的学生。*/
09    {
10      if(strcmp(stu[j].name,name))    /*比较当前输入的姓名与数组中的学生姓名是否相
11  同。*/
12      {
```

```
13          showstu(stu[j]);                    /*相同则调用 showstu 函数输出这个学生。*/
14          n++;                                /*然后计数自加。*/
15        }
16      }
17      if(n==0)                                /*如果输出学生的个数为 0。*/
18      {
19        printf("there is no such a student.\n");    /*提示没有这个学生。*/
20      }
21  }
```

提示：程序中通过判断函数 strcmp 的返回值来确定两个字符串是否相等。

11.3.10 删除学生函数

这个函数的作用是删除数组中的一个学生。实现方法是从键盘中输入一个学生的姓名，然后用 for 循环访问学生数组中所有的学生。把这个姓名与学生数组中的进行比较，如果相同则结束循环。

然后再次执行循环，把下一个学生赋值给这个学生，相当于删除了当前的学生。后面所有的学生都向前移动了一个位置。最后将学生的计数减 1。这一功能的代码如下所示。

【范例 11-16】模块 7：删除一个学生，示例代码如下所示。

示例代码 11-16

```
01  void delestu()
02  {
03    int j,n;                                  /*定义两个计数器变量。*/
04    char name[20];                            /*定义一个数组。*/
05    n=0;                                      /*定义一个变量作为计数器。*/
06    printf("delete a student:\n");            /*提示信息。*/
07    printf("please input a name:\n");         /*提示输入学生的姓名。*/
08    scanf("%s",name);                         /*从键盘输入学生的姓名。*/
09    for(j=0;j<i;j++)                          /*for 循环访问学生数组中的变量。*/
10    {
11      if(strcmp(stu[j].name,name))        /*将学生数组中的姓名与当前输入的姓名进行比
12  较。*/
13      {
14        for( ;j<i-1;j++)                  /*如果姓名相同，则执行这个循环，*/
15        {
16          stu[j]=stu[j+1];                /*将下一个学生赋值给当前的学生。*/
17        }
18        n++;                                    /*计数加 1。*/
19        i--;                                    /*学生计数减 1。*/
20        break;                                  /*结束当前循环。*/
21      }
22    }
23    if(n==0)                                    /*如果输出学生的个数为 0。*/
24    {
25      printf("there is no such a student.\n");    /*提示没有这个学生。*/
26    }
27  }
```

11.3.11 按年龄查找函数

这个函数的功能，是提示用户输入一个年龄区域。程序用 for 循环的方法，查找出这个年

龄段中所有的学生，然后列表显示出这些学生的详细信息。函数没有参数和返回值，代码如下所示。

【范例 11-17】模块 8：按年龄查找学生，示例代码如下所示。

示例代码 11-17

```
void selectbyage()
{
   int j,k,t,n;                                          /*定义函数中的变量。*/
   n=0;                                                  /*定义一个计数变量。*/
   printf("please input a top age:\n");                  /*提示输出一个年龄。*/
   scanf("%d",&j);                                       /*输入年龄。*/
   printf("please input a bottom age:\n");
   scanf("%d",&k);
   for(t=0;t<i;t++)                          /*进行循环，查找所有的学生。*/
   {
     if(stu[t].age<=j &&stu[t].age>k)        /*数组中学生的年龄与当前输入的两个年龄作比
较。*/
     {
       showstu(stu[t]);                                  /*显示这个学生。*/
       n++;                                              /*计数变量加 1。*/
     }
   }
   if(n==0)                                              /*如果计数变量为 0。*/
   {
     printf("there is no such a student.\n");            /*显示没有要查找的学生。*/
   }
}
```

11.3.12　按身高查找函数

这个函数的作用，是提示用户输入一个身高区域。程序用循环的方法，把学生数组中的年龄与这个年龄进行比较。如果学生的年龄在这个年龄区间，则输出这个学生的详细信息。函数没有参数和返回值，代码如下所示。

【范例 11-18】模块 9：按身高查找学生，示例代码如下所示。

示例代码 11-18

```
void selectbyheight()
{
   int j,k,t,n;                                          /*定义函数中的变量。*/
   n=0;                                                  /*定义一个计数器。*/
   printf("please input a top height:\n");               /*输入一个较大值。*/
   scanf("%d",&j);                                       /*从键盘读取一个变量。*/
   printf("please input a bottom  height:\n");           /*提示输入一个较小值。*/
   scanf("%d",&k);                                       /*从键盘读取一个变量。*/
   for(t=0;t<i;t++)                                  /*用 for 循环访问数组中的每一个变量。*/
   {
     if(stu[t].height<=j &&stu[t].height>k)  /*把输入的年龄和当前的年龄做比较。*/
     {
       showstu(stu[t]);                                /*如果在输入的年龄段则输出这个学生。*/
       n++;                                            /*计数变量加 1。*/
     }
   }
   if(n==0)                                              /*如果计数变量为 0。*/
   {
```

```
19        printf("there is no such a student.\n");    /*输出没有要查找的学生。*/
20     }
21  }
```

11.3.13 按性别查找函数

这个函数的功能是提示输入一个性别。程序用循环来访问学生数组中的每一个学生，把输入的性别与学生数组的性别进行比较，如果相同则输出这个学生的信息。函数没有参数和返回值，代码如下所示。

【范例 11-19】模块 10：按性别查找学生，示例代码如下所示。

示例代码 11-19

```
01  void selectbysex()
02  {
03     int j,t,n;                                          /*定义函数中的变量。*/
04     n=0;
05     printf("please input the sex:\n");                  /*提示输入性别。*/
06     scanf("%d",&j);                                     /*输入性别。*/
07     for(t=0;t<i;t++)                    /*用 for 循环访问学生数组中的每一个结构体。*/
08     {
09       if(stu[t].sex==j)                 /*结构体的成员变量与输入的性别作比较。*/
10       {
11          showstu(stu[t]);               /*显示这个学生。*/
12          n++;                           /*计数变量自加。*/
13       }
14     }
15     if(n==0)                            /*如果计数变量为 0。*/
16     {
17        printf("there is no such a student.\n");    /*提示没有这个学生。*/
18     }
19  }
```

11.3.14 程序的主函数

程序主函数的功能，是调用上面这些自定义的函数和模块，使这些功能能够成为一个统一的整体。函数的主要内容，是用 while 语句进入一个循环，然后用 menu 要求用户输入一个菜单选项。根据不同的返回结果，调用不同的函数执行不同的功能。在 menu 函数中，输入 9 时会结束这个程序的运行。主函数的流程可用图 11-2 表示。

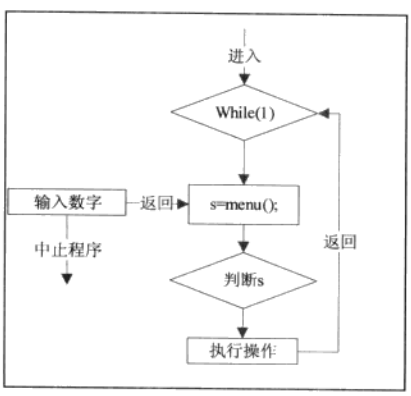


图 11-2 主函数的运行流程

根据上面的程序流程图，编写出的主函数代码如下所示。

提示：因为事先并不知道循环次数，使用 while 循环实现菜单操作界面是比较合适的选择。

【范例 11-20】模块 11：程序的主函数，示例代码如下所示。

示例代码 11-20

```
int  main()                                  /*程序的主函数。*/
{
   int s,j;                                  /*定义局部变量。*/
   i=0;                                      /*对全局变量 i 赋初值为 0。*/
   while(1)                                  /*进入一个 while 循环。*/
   {
      s=menu();                              /*从菜单返回一个选项。*/
      if(s==1)                               /*如果选项为 1，则调用学生输入函数，添加一个学生。
*/
      {
         stu[i]=getstu();                    /*添加一个学生。*/
         i++;                                /*学生计数加 1。*/
      }
      if(s==2)                               /*选项为 2。*/
      {
         printf("All the students:\n");                  /*列出所有学生。*/
         for(j=0;j<i;j++)                                /*for 循环访问所有的学生。*/
            {
               printf("  %d:  %s\n",j,stu[j].name);     /*输出学生的姓名与编号。*/
            }
      }
      if(s==3)                                           /*选项为 3。*/
      {
         for(j=0;j<i;j++)                                /*循环访问所有的学生。*/
         {
            showstu(stu[j]);                             /*显示一个学生的详细信息。*/
         }
      }
      if(s==4)                                           /*选项为 4。*/
      {
         selectbyname();                                 /*按姓名查找一个学生。*/
      }
      if(s==5)                                           /*如果选项为 5。*/
      {
         selectbyage();                                  /*按年龄查找。*/
      }
      if(s==6)                                           /*选项为 6。*/
      {
         selectbysex();                                  /*按性别查查。*/
      }
      if(s==7)                                           /*选项为 7。*/
      {
         selectbyheight();                               /*按身高查找。*/
      }
      if(s==8)                                           /*选项为 8。*/
      {
         delestu();                                      /*删除一个学生。*/
      }
```

```
49      }
50  }
```

11.3.15 程序的运行和调试

本节讲解上面编写程序的运行和调试。输入下面的命令，编译这个程序。

输入下面的命令，编译这个程序。

```
gcc 11.10.c -o stu
```

下面对这个程序进行执行和测试。

（1）输入下面的命令，运行这个程序。

```
./stu
```

（2）程序的运行结果如下所示。程序显示出一个菜单，提示用户输入一个选项。

```
Please select a menu:
      1: add a student.
      2: list the name all the student.
      3: list informations of all the student.
      4: select a student by name .
      5: select students  by age .
      6: select students  by sex .
      7: select students  by height .
      8: delete a student .
      9: exit .
```

（3）这时输入“1”，添加一个学生。然后再次用同样的方法一共输入三个学生。

（4）输入“2”，显示所有的学生。结果如下所示。

```
All the students:
  0:  Tom
  1:  Lily
  2:  Lucy
```

（5）输入“3”，显示所有学生的详细信息，结果如下所示。

```
A student:
  Name  : Tom
  Age   : 19
  Sex   : 1
  Height: 176

A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
  Height: 166

A student:
  Name  : Lucy
  Age   : 23
  Sex   : 0
  Height: 179
```

（6）输入“4”，按姓名查找一个学生。程序输出下面的提示。

```
please input a name;
```

（7）这时输入一个姓名“Lily”。程序的显示结果如下所示。

```
A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
```

```
  Height: 166
```

（8）输入“7”，按身高查找学生。程序输入下面的提示，要求输入一个上限值。

```
please input a top height:
```

（9）这时输入“170”。然后在另一个提示后面输入一个下限值“160”。程序会查找出身高在“160”与“170”之间所有学生，显示结果如下所示。

```
A student:
  Name  : Lily
  Age   : 17
  Sex   : 0
  Height: 166
```

（10）输入“8”，删除一个学生。在提示后面输入一个学生的姓名“Lily”，程序会删除这个学生。

（11）然后输入“3”，显示学生的详细信息。显示的结果如下所示。可见学生“Lily”的信息已经被删除。

```
A student:
  Name  : Tom
  Age   : 19
  Sex   : 1
  Height: 176

A student:
  Name  : Lucy
  Age   : 23
  Sex   : 0
  Height: 179
```

（12）输入“9”，退出这个程序。

11.4　小结

本章讲解了 C 程序的结构体操作。结构体是对现有数据类型的扩充，可以把有实际含义的事物定义为一个结构体数据类型。程序中使用结构体数据类型可以方便地处理有实际含义的数据。本章的内容中，结构体指针和结构体相关的函数是个难点。要掌握用结构体指针访问结构成员变量的方法。需要仔细理解结构体指针与结构体函数的作用。本章中的最后一个实例，是结构体的一个综合运用，处理了复杂的数据与功能，建议学习时完成这个程序的编写。

11.5　习题

一、填空题

1．_____是一种自定义数据类型。它有定义、新建、访问等基本操作。

2．所谓_____，是指列出一个结构体中所包括的多个变量和数据类型，建立一个新的数据类型。

3．结构体本身并不能作为一个变量进行访问，结构变量以_____作为基本变量。

4．所谓_____，指的是把结构体作为一种基本的数据类型，用这种数据类型定义一个数组。结构体数组的定义和基本数据类型的数组定义方法是相同的。

5．所谓_____，指的是定义一个指针，这个指针指向一个结构体类型的变量。用这个

指针可以访问一个结构体中的成员。

6．所谓_____，是指用结构体指针访问这个指针所指向的变量和成员，对这些变量和成员进行赋值或读取的操作。

二、选择题

1．结构体不可以作为（　　）。

A．函数的参数　　B．函数的返回值

C．函数的名称　　D．函数的实参

2．关于结构体的理解错误的是（　　）。

A．结构体是一种非自定义数据类型，由系统默认提供

B．结构体是一种自定义数据类型。结构体有定义、新建、访问等基本操作

C．结构体是自定义的数据类型，因此结构体变量也可以和其他类型的变量一样赋值、运算。不同的是，结构体本身并不能作为一个变量进行访问，结构变量以成员作为基本变量

D．所谓结构体数组，指的是把结构体作为一种基本的数据类型，用这种数据类型定义一个数组

三、简答题

1．什么是结构体，结构体如何定义与访问？

2．结构体和结构体指针如何被作为函数的参数？

四、编程题

编写一个学生管理程序。在程序中，用结构体进行学生数据的存储，结构体作为函数的参数与返回值。

第 12 章　时 间 函 数

本章介绍 C 语言中与时间有关的系统函数，包括获取当前系统的时间函数、时间的格式转换函数，以及设置当前系统时间函数。本章的主要内容如下：

- 获取当前系统时间函数；
- 有关时间格式的函数；
- 设置当前系统时间函数。

12.1　常用时间函数

在调用系统时间处理时间问题时，需要使用时间函数。本节将讲解 time、gmtime、ctime、asctime、mktime 等常用时间函数的用法。

12.1.1　返回时间函数 time

函数 time 可以返回一个时间值。该函数的使用方法如下所示。

```
01   time_t time(time_t *t);
```

time 函数会返回从公元 1970 年 1 月 1 日的 UTC 时间从 0 时 0 分 0 秒算起到现在所经过的秒数。参数 t 是一个指针，如果不是一个空指针，函数也会将返回值存到 t 指针所指的内存单元中。time_t 是“time.h”头文件中定义的一个数据类型，表示一个时间的秒数，相当于一个长整型变量。如果 t 是一个空指针，函数会返回一个 time_t 型长整型数。

注意：UTC（Universal Time Coordinated）指的是协调世界时，相当于格林威治平均时，和英国时间是相同的。中国北京时间比 UTC 时间早 8 小时。

下面是使用 time 函数返回当前时间的秒数，然后输出。

【范例 12-1】使用 time 函数返回当前时间的秒数，示例代码如下所示。

示例代码 12-1

```
01   #include <stdio.h>
02   #include <time.h>                          /*包含“time.h”头文件。*/
03
04   int main()                                 /*程序主函数。*/
05   {
06     time_t s;                                /*定义一个 time_t 型时间变量。*/
07     s= time((time_t*)NULL);                  /*取当前的时间，参数是一个空指针。*/
08     printf("Now  :%ld\n",s);                 /*输出时间。*/
09   }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.1.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Now  :1198607050
```

再次运行这个程序，结果如下所示。

```
Now  :1198607162
```

从结果可知，第二次运行程序比第一次运行程序经过了 112 秒。

【代码解析】由于使用了预定义类型 time_t，所以在代码第 02 行引入头文件 time.h。在代码第 07 行使用强制类型转换将 NULL 转换为 time_t 类型后，作为 time 函数的参数。

下面的代码，是在 time 函数的参数中使用一个指针，用这个指针接收 time 函数返回的结果。

【范例 12-2】用指针接收 time 函数返回的秒数，示例代码如下所示。

示例代码 12-2　\源文件\12\12.2.c

```
01  #include <stdio.h>
02  #include <time.h>                    /*包含“time.h”头文件。*/
03
04  int main()
05  {
06    time_t p;                          /*定义一个 time_t 类型变量。*/
07    time(&p);                          /*取时间，参数是指针 p，返回结果到指针的内存单元。
08  */
09    printf("Now  :%ld\n",*p);          /*输出时间。*/
10  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.2.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。可见使用指针接收时间结果的作用与返回时间方法是相同的。

```
Now  :1198607309
```

【代码解析】代码第 07 行将 time_t 类型变量 p 的地址传递给 time 函数用于保存获得的时间。

12.2.2　取当前时间函数 gmtime

函数 gmtime 的作用是将 time_t 表示秒数的时间转换为人可以理解的时间。这个函数的使用方法如下所示。

提示：可以使用 gmtime 函数将人们无法理解的秒数时间转换为人们容易理解的时间格式。

```
struct tm *gmtime(time_t *timep);
```

从上面的使用方法可知，函数的参数是一个表示当前时间秒数的指针。返回值是一个 tm 类型的结构体指针。tm 结构体是“time.h”头文件中定义的，定义方法和成员如下所示。

```
01  struct tm
02  {
03    int tm_sec;
```

```
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

这些成员表示的含义和范围如下所示。

- int tm_sec：代表当前秒数，正常范围是 0～59。
- int tm_min：代表当前分钟数，正常范围是 0～59。
- int tm_hour：从午夜算起的小时数，范围是 0～23。
- int tm_mday：当前月份的日数，范围是 1～31。
- int tm_mon：代表当前月份，从一月算起，范围是 0～11。
- int tm_year：从 1900 年算起至今的年数。
- int tm_wday：一星期的日数，从星期一算起，范围是 0～6。
- int tm_yday：从本年 1 月 1 日算起至今的天数，范围为 0～365。
- int tm_isdst：是否使用了夏令时间。如果为 1 表示使用了夏令时，为 0 则表示没有使用夏令时。

注意：夏令时间指的是某些国家在夏季将时间调整若干个小时。我国过去曾使用过夏令时间，现在已经中止。计算机中可以设置夏令时间。

需要注意的是，这里的时间返回的是 UTC 时间，即英国零时区时间。如果计算机中使用了 UTC 时间并且设置了时间，则结果与计算机上显示的时间有一些差异。

【范例 12-3】使用 gmtime 函数获取容易理解的当前系统时间，示例代码如下所示。

示例代码 12-3

```
#include <stdio.h>
#include <time.h>                          /*包含“time.h”头文件。*/

main(){
   time_t timep;                           /*定义一个 time_t 型变量。*/
   struct tm *p;                           /*定义一个 tm 型结构体指针。*/
   time(&timep);                           /*取当前时间，返回到 timep 的值中。*/
   p=gmtime(&timep);                       /*取当前时间，返回到结构体指针 p 上。*/
   printf("Year  :%d\n",1900+p->tm_year);          /*输出年。*/
   printf("Month :%d\n",1+p->tm_mon);              /*输出月。*/
   printf("Day   :%d\n",p->tm_mday);               /*日。*/
   printf("Hour  :%d\n",p->tm_hour);               /*小时。*/
   printf("Minute:%d\n",p->tm_min);                /*分。*/
   printf("Second:%d\n",p->tm_sec);                /*秒。*/
   printf("Weekday:%d\n",p->tm_wday);              /*星期几。*/
   printf("Days  :%d\n",p->tm_yday);               /*一年的第几天。*/
   printf("Isdst :%d\n",p->tm_isdst);              /*是否使用了夏令时。*/
}
```

注意：tm 结构体中的年是从 1900 年到现在的第几年，所以在显示时需要加 1900。月份的返回值是 0 至 11，所以加 1 以后才可以表示当前的月份。

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.3.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。时间这样显示时，可以表示为可以理解的时间。表示成秒数的时间是无法被识别的。

```
Year  :2007
Month :12
Day   :25
Hour  :18
Minute:47
Second:52
Weekday:2
Days  :358
Isdst :0
```

再次运行这个程序，结果如下所示。可见这个时间的返回值是系统时间，是随时变化的。

```
Year  :2007
Month :12
Day   :25
Hour  :18
Minute:54
Second:49
Weekday:2
Days  :358
Isdst :0
```

【代码解析】代码第 08 行是程序的关键代码，它通过 gmtime 函数将使用 time 函数获得的时间转换为 tm 结构体指针，并且赋值给 tm 结构体变量 p。

11.2.3 字符串格式时间函数 ctime

函数 ctime 的作用是将一个时间返回成一个可以识别的字符串格式。这个函数的使用方法如下所示。

```
01   char *ctime(time_t *timep);
```

从上面的使用方法可知，函数 ctime 的参数是一个指向 timep 类型的指针。函数会把这个指针转换成一个字符串，然后返回为一个字符串的头指针。这里返回的时间已经转换成本地时区的时间，与计算机上显示的时间相同。字符串的显示格式为“Feb Jun 14 12:56:08 1999”这种形式。下面是这个函数的使用实例。

【范例 12-4】使用 ctime 函数返回当前时间字符串，示例代码如下所示。

示例代码 12-4

```
01   #include <stdio.h>
02   #include <time.h>                    /*包含“time.h”头文件。*/
03   #include <stdlib.h>                  /*包含“stdlib.h”头文件。*/
04
05   int main()
06   {
07     time_t p;                          /*定义一个 time_t 类型变量。*/
08     char s[30];                        /*定义一个字符串 s。*/
09     time(&p);                          /*取时间，参数是指针 p，返回结果到指针的内存单元。
10   */
11     strcpy(s,ctime(&p)) ;              /*将 ctime 返回的结果复制到字符串 s 上。*/
```

```
12    printf("%s\n",s);                    /*输出字符串 s。*/
13  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.4.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Tue Dec 25 14:03:42 2007
```

【代码解析】代码第 11 行调用 ctime 函数将当前时间转换为字符串，并通过 strcp 函数将此字符串赋值给字符数组 s。

12.2.4　字符串格式时间函数 asctime

函数 asctime 的作用是将一个 tm 格式的时间转换为一个字符串格式。这个函数的使用方法如下所示。

```
char * asctime(struct tm * timeptr);
```

函数的参数是一个 tm 格式的时间结构体。返回值是一个字符串。与函数 ctime 不同的是，ctime 的参数是一个表示秒数的时间指针。返回的字符串格式与 ctime 的返回格式是相同的。下面是 asctime 的使用实例。

注意： ctime 函数的参数为秒数时间指针，而 asctime 函数的参数为 tm 结构体指针。

【范例 12-5】使用 asctime 函数返回当前时间字符串，示例代码如下所示。

示例代码 12-5

```
01  #include <stdio.h>
02  #include <time.h>                    /*包含“time.h”头文件。*/
03  #include <stdlib.h>                  /*包含“stdlib.h”头文件。*/
04
05  int main()
06  {
07    time_t p;                          /*定义一个 time_t 类型变量。*/
08    struct tm *q;                      /*定义一个 tm 类型的指针。*/
09    char s[30];                        /*定义一个字符串 s。*/
10    time(&p);                          /*取时间，参数 p 的地址，返回结果到变量的内存单元。
11  */
12    q=gmtime(&p);                      /*用 gmtime 函数返回一个 tm 格式的时间指针。*/
13    strcpy(s,asctime(q)) ;             /*将 asctime 返回的结果复制到字符串 s 上。*/
14    printf("%s\n",s);                  /*输出字符串 s。*/
15  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.5.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Tue Dec 25 19:24:43 2007
```

【代码解析】程序中代码第 10~13 行分别调用了 time 函数获得当前时间，gmtime 函数获得 tm 结构体指针，asctime 函数将 tm 结构体指针的内容转换为字符串类型。

12.1.5 取得当地时间函数 localtime

函数 localtime 的作用是返回 tm 格式的当地时间。与 gmtime 函数不同的是，gmtime 函数返回的是一个 UTC 时间。localtime 时间的使用方法如下所示。

```
01  struct tm *localtime(time_t * timep);
```

注意：gmtime 函数返回 UTC 时间，localtime 函数则返回本地时间。

从上面的使用方法可知，localtime 函数的参数是一个 time_t 型时间的指针，返回值是一个 tm 型的结构体时间。这个函数的使用实例如下所示。

【范例 12-6】使用 localmtime 函数获取当地时间，示例代码如下所示。

示例代码 12-6

```
01  #include <stdio.h>
02  #include <time.h>                          /*包含“time.h”头文件。*/
03
04  main(){
05     time_t timep;                           /*定义一个 time_t 型变量。*/
06     struct tm *p;                           /*定义一个 tm 型结构体指针。*/
07     time(&timep);                           /*取当前时间，返回到 timep 的值中。*/
08     p=localtime(&timep);                    /*取本地时间，返回到结构体指针 p 上。*/
09     printf("Year  :%d\n",1900+p->tm_year);          /*输出年。*/
10     printf("Month :%d\n",1+p->tm_mon);              /*输出月。*/
11     printf("Day   :%d\n",p->tm_mday);               /*日。*/
12     printf("Hour  :%d\n",p->tm_hour);               /*小时。*/
13     printf("Minute:%d\n",p->tm_min);                /*分。*/
14     printf("Second:%d\n",p->tm_sec);                /*秒。*/
15     printf("Weekday:%d\n",p->tm_wday);              /*星期几。*/
16     printf("Days  :%d\n",p->tm_yday);               /*一年的第几天。*/
17     printf("Isdst :%d\n",p->tm_isdst);              /*是否使用了夏令时。*/
18  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.6.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。这次显示时间的结果，是转换成本地时区以后的时间，和计算机上显示的时间是相同的。

```
Year  :2007
Month :12
Day   :25
Hour  :14
Minute:29
Second:14
Weekday:2
Days  :358
Isdst :0
```

【代码解析】代码第 08 行调用 localtime 函数时，实际参数为 time_t 类型变量的地址。

localtime 函数的返回值赋给了 tm 结构体指针。

12.1.6　将时间转换成秒数函数 mktime

函数 mktime 的作用是将一个 tm 结构类型的时间转换成秒数时间。

注意：mktime 函数与 gmtime 的作用刚好相反。

该函数的使用方法如下所示。

```
01   time_t mktime(tm * timeptr);
```

从上面的使用方法可知，函数的参数是一个 tm 类型的指针，返回一个 time_t 类型的数字表示当前的秒数。例如下面是这个函数的使用实例。先取得当前时间的秒数，然后用 ctime 函数转化为字符串输出。对这个时间转换成 tm 结构体时间以后，再用 mktime 转换成 time_t 格式的时间，然后用 ctime 将时间转换为字符串输出。程序的代码如下所示。

【范例 12-7】使用 mktime 函数构造时间，示例代码如下所示。

示例代码 12-7

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include <sys/time.h>
04  #include <unistd.h>
05  #include <time.h>                          /*程序的包含文件。*/
06
07  main()
08  {
09     time_t t;                               /*定义一个 time_t 型的时间。*/
10     struct tm *p;                           /*定义一个 tm 型的时间指针。*/
11     char s[30];                             /*定义一个字符串。*/
12     time(&t);                               /*取当前时间。*/
13     strcpy(s,ctime(&t));                    /*将时间转换成字符串。*/
14     printf("%s\n",s);                       /*输出时间。*/
15     p=gmtime(&t);                           /*将时间转换成 tm 格式的结构体。*/
16     t=mktime(p);                            /*将时间转换成 time_t 型的时间。*/
17     strcpy(s,ctime(&t));                    /*将时间转换成字符串。*/
18     printf("%s\n",s);                       /*输出时间。*/
19  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.7.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序输出的结果是相同的。两次输出的是同一个时间值，第二次输出时经过了一次转换。

```
Wed Dec 26 08:41:07 2007
Wed Dec 26 08:41:07 2007
```

【代码解析】代码第 15~17 行对使用 time 函数获得的时间，调用 gmtime 函数和 mktime 完成一次转换。代码第 14 行和第 18 行的输出结果是相同的。

12.1.7　取得目前的时间函数 gettimeofday

前面所讲到的时间函数只能把时间精确到秒。如果对时间的处理精度为微秒级，需要使用

函数 gettimeofday。一微秒等于百万分之一秒。这个函数的使用方法如下所示。

```
01  int gettimeofday ( struct timeval * tv , struct timezone * tz )
```

这个函数的参数是两个结构体指针。这两个结构体的定义如下所示。

```
01  struct timeval
02  {
03    long tv_sec;
04    long tv_usec;
05  };
```

结构体成员的含义如下所示。

- tv_sec：当前时间的秒数。
- tv_usec：当前时间的微秒数。

```
01  struct timezone{
02    int tz_minuteswest;
03    int tz_dsttime;
04  };
```

结构体成员的含义如下所示。

- tz_minuteswest：与 UTC 时间相差的分钟数。
- tz_dsttime：与夏令时间相差的分钟数。

函数 gettimeofday 会把当前时间的这些参数返回到这两个结构体指针上。如果处理成功，则返回真值 1，否则返回 0。这个函数的使用实例如下所示。

【范例 12-8】使用 gettimeofday 函数取得当前时间，示例代码如下所示。

示例代码 12-8

```
01  #include <stdio.h>
02  #include <sys/time.h>
03  #include <unistd.h>
04  #include <time.h>                    /*包含头文件 time.h。*/
05
06  main()
07  {
08    struct timeval tv;                 /*定义一个 timeval 型的结构体。*/
09    struct timezone tz;                /*定义一个 timezone 型的结构体。*/
10    gettimeofday (&tv , &tz);          /*取得当前时间。两个结构体的指针作为参数。*/
11    printf("tv_sec      : %d\n", tv.tv_sec);      /*输出秒。*/
12    printf("tv_usec     : %d\n",tv.tv_usec);      /*输出微秒。*/
13    printf("tz_minuteswest: %d\n", tz.tz_minuteswest);     /*输出 UTC 时间相差的
分钟数。*/
14    printf("tz_dsttime    : %d\n",tz.tz_dsttime);          /*输出与夏令时间相差
的分钟数。*/
15  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.8.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
tv_sec      : 1198594465
tv_usec     : 462591
tz_minuteswest: 0
tz_dsttime    : 0
```

再次运行这个程序，结果如下所示。秒数与微秒数已经改变。

```
tv_sec      : 1198598472
tv_usec     : 726880
tz_minuteswest: 0
tz_dsttime  : 0
```

【代码解析】程序的主函数中使用 gettimeofday 函数取得当前的时间，并将取得的时间赋值给 timeval 结构体和 timezone 结构体。

提示：使用 gettimeofday 函数可以得到比 time 函数更加精确的时间。

12.1.8　设置当前时间函数 settimeofday

函数 settimeofday 的作用是设置当前的系统时间。只有 root 用户登录以后才有权限进行这个操作。该函数的使用方法如下所示。

```
01  int settimeofday (struct timeval *tv, struct timezone *tz);
```

这个函数的参数是 timeval 类型的结构体指针和 timezone 类型的结构体指针。这两个结构体类型的定义在上一节中已经讲过。下面是这个函数的使用实例，取当前的时间输出，然后将当前的时间向前调整 4000 秒。

【范例 12-9】使用 settimeofday 函数设置当前时间，示例代码如下所示。

示例代码 12-9

```
01  #include <stdio.h>
02  #include <sys/time.h>
03  #include <unistd.h>
04  #include <time.h>                                   /*程序的包含文件。*/
05
06  main()
07  {
08     struct timeval tv;                               /*定义一个 timeval 类型的结构体。*/
09     struct timezone tz;                              /*定义一个 timezone 型的结构体。*/
10     gettimeofday (&tv , &tz);                        /*取当前的时间。*/
11     printf("tv_sec      : %d\n", tv.tv_sec); /*输出秒。*/
12     printf("tv_usec     : %d\n",tv.tv_usec); /*输出微秒。*/
13     printf("tz_minuteswest: %d\n", tz.tz_minuteswest);     /*与 UTC 时间相差的分
钟数。*/
14     printf("tz_dsttime   : %d\n",tz.tz_dsttime);  /*夏令时间相差的分钟数。*/
15     tv.tv_sec=tv.tv_sec- 4000;                       /*当前的时间减去 4000 秒。*/
16     settimeofday (&tv , &tz);                        /*设置当前时间。*/
17  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.9.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示，这个时间是当前时间。程序同时将计算机的时间向前设置了 4000 秒，可以看到计算机的时间已经改变。

```
tv_sec      : 1198599321
tv_usec     : 264551
tz_minuteswest: 0
```

```
tz_dsttime    : 0
```

【代码解析】代码第 10~14 行取得当前时间并输出，第 15~16 行将取得的时间减去 4000 秒后，重新设置为当前时间。

12.2 综合练习

本节将通过综合练习讲解时间函数在实际开发中的应用。在处理时间时，需要用到上一节所讲的各种时间函数。在进行计时的程序中，需要用到 gettimeofday 函数，这个函数可以返回精确到微秒的时间值。计算机运行一个程序时，需要占用一定的时间。用时间函数可以对程序开始和结束的时间进行计时。程序结束时可以处理这两个时间，显示出结果。因为计算机运行程序的时间常常很短，需要用 gettimeofday 函数来取得精确到微秒的时间值。程序的代码如下所示。

【范例 12-10】使用 gettimeofday 函数获取精确时间，用于计算程序运行花费的时间。

示例代码 12-10

```
01  #include <stdio.h>
02  #include <sys/time.h>
03  #include <unistd.h>
04  #include <time.h>                          /*包含头文件。*/
05
06  main()
07  {
08     struct timeval tv,tv2;                  /*定义两个 timeval 类型的结构体。*/
09     struct timezone tz;                     /*定义一个 timzone 型的结构体。*/
10     long sec,usec,i;                        /*定义三个长整形数。*/
11
12     gettimeofday (&tv , &tz);               /*取得当前时间。*/
13     for(i=0;i<100000000;i++)                /*程序进行多次循环。*/
14     {
15          ;;;;;;;;;;;;;;;;;;;;               /*执行一段空语句。分号表示空语句。*/
16     }
17     gettimeofday (&tv2, &tz);               /*取得结束时的时间。*/
18     sec=tv2.tv_sec-tv.tv_sec;               /*用减法求出秒数。*/
19     usec=tv2.tv_usec-tv.tv_usec;            /*用减法求得微秒数。*/
20     printf("%ld\n",sec*1000000 + usec);     /*把秒转换成微秒，再与微秒相加，输出结果。*/
21  }
```

【运行结果】输入下面的命令编译这个程序。

```
gcc 12.10.c
```

输入下面的命令运行这个程序。

```
./a.out
```

程序的运行结果如下所示。说明进行这些循环运算，占用的时间是 0.64 秒。

```
647834
```

12.3 小结

本章讲解了 C 函数中的常用时间函数。在本章的学习中，time_t 时间类型与 gm 时间结构体的理解是个难点。相关的函数都需要使用这两种时间或指针作为参数。其中 time、gmtime、

ctime、asctime、gettimeofday 这几个函数需要重点掌握。三个实例演示了时间函数的使用方法与技巧。

12.4　习题

一、填空题

1．函数_____的作用是将 time_t 表示秒数的时间转换为人可以理解的时间。

2．函数_____的作用是将一个时间返回成一个可以识别的字符串格式。

3．函数_____的作用是将一个 tm 格式的时间转换为一个字符串格式。

4．函数_____的作用是返回 tm 格式的当地时间。与 gmtime 函数不同的是，gmtime 函数返回的是一个 UTC 时间。

5．函数_____的作用是将一个 tm 结构类型的时间转换成秒数时间。

6．函数_____的作用是设置当前的系统时间。只有 root 用户登录以后才有权限进行这个操作。

二、选择题

1．取得当前时间的函数是（　　）。

A．gmtime　　B．ctime　　C．asctime　　D．localtime

2．将时间转换为秒数的函数是（　　）。

A．gmtime　　B．ctime　　C．asctime　　D．mktime

3．gmtime 函数与 localtime 函数的区别是（　　）。

A．gmtime 函数返回 UTC 时间，localtime 函数则返回本地时间

B．localtime 函数返回 UTC 时间，gmtime 函数则返回本地时间

C．localtime 函数与 gmtime 函数完全等价，没有区别

D．gmtime 函数与 localtime 函数都能取得 UTC 时间和本地时间

三、简答题

1．简述常用的时间函数及其用法。

2．简述 gmtime 函数与 localtime 函数的区别与联系。

3．C 程序中如何设置系统的时间。

四、编程题

编写程序，用时间函数对程序开始和结束的时间进行计时。程序结束时可以处理这两个时间，显示出结果。因为计算机运行程序的时间通常很短，需要用 gettimeofday 函数来取得精确到微秒的时间值。

第五篇　文件与目录篇

第 13 章　目录与文件

本章介绍了 C 语言中关于文件和目录操作的最基本概念。包括文件操作的权限，文件与目录的创建与删除，文件的打开与关闭，文件的读写与锁定，文件的移动与复制等内容。本章的主要内容如下：

- 文件操作的权限；
- 文件操作的错误处理与错误号；
- 创建与删除目录、文件；
- 文件的打开与关闭；
- 文件的读写、锁定、移动与复制。

13.1　文件操作的权限

所谓权限，指的是文件系统为了进行安全管理，需要在操作时进行用户身份认证。合法的用户可以进行操作。而没有权限的用户不能进行文件操作。用 C 程序进行目录与文件操作时，需要设置目录与文件的权限。本节将讲解权限的表示方法。前面章节讲解的 Linux 命令中，可以用 chmod 命令更改文件的权限。例如下面的命令，是对一个文件添加可执行权限。

```
chmod +x a.out
```

在 C 编程中，需要用三个八进制数字来表示文件的权限。例如 777 表示这个用户、同组成员、其他成员对这个文件都有执行、写、读的权限。数字 abc 表示的文件权限如下所示。

- 第一个数字表示本用户的权限。相当于 User 的权限。
- 第二个数字表示同组的权限。相当于 Group 的权限。
- 第三个数字表示其他用户的权限。相当于 Other 的权限。

提示：Linux 将所有用户（简写为 a）分成三类 User（简写为 u）、Group（简写为 g）和 Other（简写为 o）。

每一个数字不同的值代表不同的权限。数字和含义如下所示。

- 4 表示可读权限，相当于 r 权限。
- 2 表示可写权限，相当于 w 权限。
- 1 表示可执行权限，相当于 x 权限。

如果有多种权限，可把几个权限加起来。例如下面不同的权限组合。

- 7 等于 4+2+1，表示这个文件有可读可写可执行的权限。
- 5 等于 4+1，表示这个文件有可读可执行的权限。
- 6 等于 4+2，表示这个文件有可读可写权限，但是不可以执行。

chmod 命令可以使用这些数字作为参数对一个文件更改权限。例如下面的命令，就是把一个文件的权限设置为 766。

```
chmod 766 a.out
```

13.2　错误处理与错误号

在进行文件操作时，可能会遇到权限不足、找不到文件等错误，这时需要在程序中设置错误捕捉语句并显示错误。错误捕捉和错误输出是应用错误号和 strerror 函数来实现的。

13.2.1　错误定义的理解

在 Linux 系统中，已经把所有的错误定义为不同的错误号和错误常数。程序如果发生了异常，会返回错误的常数，这个常数可以显示为整型数字，也可以用 strerror 函数显示为已经定义的错误信息。可以打开包含文件查看这些错误号的错误信息。在终端中输入下面的命令，打开错误定义文件。

```
vim /usr/include/asm-generic/errno-base.h
```

显示的错误定义代码如下所示。

```
01  #ifndef _ASM_GENERIC_ERRNO_BASE_H
02  #define _ASM_GENERIC_ERRNO_BASE_H
03  #define  EPERM          1   /* Operation not permitted */
04  #define  ENOENT         2   /* No such file or directory */
05  #define  ESRCH          3   /* No such process */
06  #define  EINTR          4   /* Interrupted system call */
07  #define  EIO            5   /* I/O error */
08  #define  ENXIO          6   /* No such device or address */
09  #define  E2BIG          7   /* Argument list too long */
10  #define  ENOEXEC        8   /* Exec format error */
11  #define  EBADF          9   /* Bad file number */
12  #define  ECHILD         10  /* No child processes */
13  #define  EAGAIN         11  /* Try again */
14  #define  ENOMEM         12  /* Out of memory */
15  #define  EACCES         13  /* Permission denied */
16  #define  EFAULT         14  /* Bad address */
17  #define  ENOTBLK        15  /* Block device required */
18  #define  EBUSY          16  /* Device or resource busy */
19  #define  EEXIST         17  /* File exists */
20  #define  EXDEV          18  /* Cross-device link */
21  #define  ENODEV         19  /* No such device */
22  #define  ENOTDIR        20  /* Not a directory */
23  #define  EISDIR         21  /* Is a directory */
24  #define  EINVAL         22  /* Invalid argument */
25  #define  ENFILE         23  /* File table overflow */
26  #define  EMFILE         24  /* Too many open files */
27  #define  ENOTTY         25  /* Not a typewriter */
28  #define  ETXTBSY        26  /* Text file busy */
29  #define  EFBIG          27  /* File too large */
30  #define  ENOSPC         28  /* No space left on device */
31  #define  ESPIPE         29  /* Illegal seek */
32  #define  EROFS          30  /* Read-only file system */
33  #define  EMLINK         31  /* Too many links */
34  #define  EPIPE          32  /* Broken pipe */
35  #define  EDOM           33  /* Math argument out of domain of func */
36  #define  ERANGE         34  /* Math result not representable */
37  #endif
```

还有一个错误定义文件，可以用下面的命令打开查看这个文件。

```
vim /usr/include/asm-generic/errno.h
```

从这两个文件可知，Linux 系统一共定义了 131 种错误常数。

提示：利用这些错误常数可以返回程序的错误信息。

13.2.2 用错误常数显示错误信息

程序错误返回的错误常数是不容易理解的。需要转换成有效的识别语句。函数 strerror 可以把一个错误常数输出成一个错误提示语句。例如下面的实例，是把 ENOENT、EIO、EEXIST 三个错误的序号和错误信息显示出来。

【范例 13-1】使用 strerror 函数把错误常数转换为错误提示语句，示例代码如下所示。

示例代码 13-1

```
#include <stdio.h>
#include <string.h>
#include <errno.h>                              /*包含错误处理头文件。*/
int main(void)
{
   printf("ENOENT:\n");                         /*输出提示。*/
   char *mesg = strerror(ENOENT);               /*将错误号转换成一个字符串。*/
   printf("  Errno  :%d\n",ENOENT);             /*输出错误号。*/
   printf("  Message:%s\n",mesg);               /*输出错误信息。*/

   printf("EIO    :\n");                        /*EIO 错误。*/
   char *mesg1 = strerror(EIO);
   printf("  Errno  :%d\n",EIO);
   printf("  Message:%s\n",mesg1);

   printf("EEXIST :\n");                        /*文件重名错误。*/
   char *mesg2 = strerror(EEXIST);
   printf("  Errno  :%d\n",EEXIST);
   printf("  Message:%s\n",mesg2);
}
```

这三个错误的含义如下所示。

- ENOENT：没有相关的文件或文件夹。
- EIO：I/O 出现错误。
- EEXIST：文件重名。

注意：I/O 指的是计算机的输入和输出，包括一切操作、程序或设备与计算机之间发生的数据流通。最常见的 I/O 设备有打印机、硬盘、键盘和鼠标。

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.err.1.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
ENOENT:
  Errno  :2
  Message:No such file or directory
EIO    :
  Errno  :5
  Message:Input/output error
EEXIST :
```

```
Errno  :17
Message:File exists
```

【代码解析】本程序使用 strerror 函数（代码第 07 行、第 12 行和 17 行）将给定的错误号转换为人们容易理解的错误字符串。

13.2.3　用错误序号显示错误信息

函数 strerror 可以把错误常数返回成一个表示错误信息的字符串。这个函数的使用方法如下所示。

```
01   char *strerror(int errnum);
```

函数的参数是一个表示错误信息的整型数，返回值是一个字符串。上一节程序中的错误常数，实际上也是一个整型数。下面的实例输出前 15 个序号的错误信息。

【范例 13-2】使用 strerror 函数把错误号转换为错误提示语句，示例代码如下所示。

示例代码 13-2

```
01   #include <stdio.h>
02   #include <string.h>
03   #include <errno.h>                                  /*包含错误处理头文件。*/
04   int main(void)
05   {
06      int i;                                          /*定义一个循环变量 i。*/
07      for(i=1;i<=15;i++)                              /*循环输出。*/
08      {
09         printf("Errno:%d  ",i);                      /*输出错误号。*/
10         printf("Message:%s\n",strerror(i));          /*输出错误信息。*/
11      }
12   }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.err.2.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
Errno:1  Message:Operation not permitted
Errno:2  Message:No such file or directory
Errno:3  Message:No such process
Errno:4  Message:Interrupted system call
Errno:5  Message:Input/output error
Errno:6  Message:No such device or address
Errno:7  Message:Argument list too long
Errno:8  Message:Exec format error
Errno:9  Message:Bad file descriptor
Errno:10  Message:No child processes
Errno:11  Message:Resource temporarily unavailable
Errno:12  Message:Cannot allocate memory
Errno:13  Message:Permission denied
Errno:14  Message:Bad address
Errno:15  Message:Block device required
```

【代码解析】代码第 07~11 行利用 for 循环语句输出编号为 1~15 的错误编号和字符串解释。

13.3　创建与删除目录

在 Linux 系统中，目录就是一个文件夹，文件可以存放在目录中。目录是一种特殊的文件，

需要对目录设置权限。本节将讲解 C 程序的目录操作。

13.3.1 创建目录函数 mkdir

函数 mkdir 可以在硬盘中建立一个目录，相当于 mkdir 命令。但与 mkdir 命令不同的是，这里的操作是用 C 语言的函数完成目录创建的。函数的使用方法如下所示。

```
01   int mkdir(char *pathname,mode_t mode);
```

在参数列表中，pathname 是一个字符串，表示需要创建的目录路径。mode 是表示权限的八进制数字。如果目录创建成功，则返回整型数 0，否则返回整型数-1。要使用这个函数，需要在程序中包含“sys/types.h”与“sys/stat.h”两个头文件。

注意：使用 mkdir 函数创建目录时需要指定创建的目录的权限模式，权限模式以八进制的方式给出（以 0 开头的四个数字）。

在创建目录时，可能有权限、目录名、硬盘空间等问题，可能返回的错误代码如下所示。需要在程序中捕捉这些错误输出错误信息。

- EPERM：目录中有不合规则的名字。
- EEXIST：参数 pathname 所指的目录已存在。
- EFAULT：pathname 指向了非法的地址。
- EACCESS：权限不足，不允许创建目录。
- ENAMETOOLONG：参数 pathname 太长。
- ENOENT：所指的上级目录不存在。
- ENOTDIR：参数 pathname 不是目录。
- ENOMEM：核心内存不足。
- EROFS：欲创建的目录在只读文件系统内。
- ELOOP：参数 pathname 多个符合的链接。
- ENOSPC：磁盘空间不足。

下面的实例，是在目录/root 下面创建一个 tmp11 目录。程序中设置一个错误号变量，捕捉程序发生的异常，如果创建不成功则输出这个错误。

【范例 13-3】使用 mkdir 函数创建一个目录，示例代码如下所示。

示例代码 13-3

```
01  #include <stdio.h>
02  #include <sys/types.h>
03  #include <sys/stat.h>                    /*包含头文件。*/
04  #include <errno.h>                       /*包含头文件处理程序的错误。*/
05  main()
06  {
07     extern int errno;                     /*设置一个错误。*/
08     char *path="/root/tmp11";             /*定义一个字符串表示需要创建的目录。*/
09
10     if(mkdir(path,0766)==0)               /*创建一个目录。*/
11       /*需要注意这里的权限设置参数，第一个 0 表示这里的是八进制数，766 的含义如本章第 1 节所述。*/
12       /*如果目录创建成功，则会返回 0，返回值与 0 进行比较。*/
13     {
14        printf("created the directory %s.\n",path);    /*输出目录创建成功。*/
15     }
16     else                                  /*如果不成功则输出提示信息。*/
```

```
17      {
18          printf("cant't creat the directory %s.\n",path);    /*输出信息。*/
19          printf("errno: %d\n",errno);                        /*输出错误号。*/
20          printf("ERR  : %s\n",strerror(errno));              /*输出错误信息。*/
21      }
22  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.1.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表明已创建这个目录。可以在终端中输入“ls /root”命令查看这个目录。

```
created the directory /root/tmp11
```

当再次运行这个程序时，需要创建的目录与上一次创建的目录重名，则不能创建目录。显示的结果如下所示。这时返回了一个错误号，输出了错误信息为“File exists”。

```
cant't creat the directory /root/tmp11.
errno: 17
ERR: File exists
```

【代码解析】代码第 10 行完成目录的创建操作，需要注意新创建的目录的权限的格式。

13.3.2　删除目录函数 rmdir

rmdir 函数的作用是删除一个空目录。该函数的使用方法如下所示。

```
01  int rmdir(char *pathname);
```

参数 pathname 是需要删除的目录字符串指针。如果删除成功，则返回一个整型 0，否则返回-1。可以设置一个 errno 来捕获发生的错误。

注意：使用 rmdir 函数删除目录时，执行程序的用户必须具有相应的权限，否则将导致删除失败。

下面使用这个函数来删除一个目录的实例，用来删除上一节创建的目录/root/tmp11。

【范例 13-4】使用 rmdir 函数删除指定目录，示例代码如下所示。

示例代码 13-4

```
01  #include <stdio.h>
02  #include <sys/types.h>
03  #include <sys/stat.h>
04  #include <errno.h>                              /*包含头文件。*/
05  main()
06  {
07    extern int errno;                             /*设置错误编号。*/
08    char *path="/root/tmp11";                     /*设置需要删除的目录。*/
09
10    if(rmdir(path)==0)                            /*删除文件，返回值与 0 比较。*/
11    {
12      printf("deleted the directory %s.\n",path);     /*显示删除成功。*/
13    }
14    else                                                /*如果删除不成功。*/
15    {
```

```
16      printf("cant't delete the directory %s.\n",path);     /*显示删除错误。*/
17      printf("errno: %d\n",errno);                          /*显示错误号。*/
18      printf("ERR  : %s\n",strerror(errno));                /*显示错误信息。*/
19    }
20  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.2.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。表明已经删除了这个目录。

```
deleted the directory /root/tmp11.
```

再次运行这个程序，因为没有目录/root/tmp11，所以显示错误信息。程序的运行结果如下所示。

```
cant't delete the directory /root/tmp11.
errno: 2
ERR  : No such file or directory
```

【代码解析】代码第 08 行指定要删除的目录的路径字符串，代码第 10 行执行目录的删除操作。

13.4 文件的创建与删除

所谓创建文件，是指在一个目录中建立一个空文件，文件可供其他程序的写入操作。删除文件指将无用的文件从磁盘中删除。本节将讲解文件的建立与删除操作。

13.4.1 创建文件函数 creat

函数 creat 的作用是在目录中建立一个空文件，该函数的使用方法如下所示。

```
01  int creat(char * pathname, mode_t mode);
```

函数的参数 pathname 表示需要建立文件的文件名和目录名。mode 表示这个文件的权限。文件权限的设置见本章第一节所述。文件创建成功时返回创建文件的编号，否则返回-1。

注意：使用 create 函数创建文件时需要指定创建的文件的权限模式，权限模式以八进制的方式给出（以 0 开头的四个数字）。

使用这个函数时，需要在程序的前面包含下面三个头文件。

```
01  #include<sys/types.h>
02  #include<sys/stat.h>
03  #include<fcntl.h>
```

如果创建文件不成功，可用 errno 捕获错误编号然后输出。下面的实例使用 creat 函数在/root 目录下面建立一个文件 tmp.txt。

【范例 13-5】使用 create 函数创建文件，示例代码如下所示。

示例代码 13-5

```
01  #include <stdio.h>
02  #include <sys/types.h>
03  #include <sys/stat.h>
```

```
04  #include <fcntl.h>
05  #include <errno.h>                                    /*包含头文件。*/
06  main()
07  {
08    extern int errno;                                  /*定义错误号。*/
09    char *path="/root/tmp.txt";
10
11    if(creat(path,0766)==-1)                           /*用 creat 函数创建一个文件。*/
12    {
13      printf("cant't create the file %s.\n",path);     /*不能创建文件。*/
14       printf("errno: %d\n",errno);                    /*显示错误号。*/
15      printf("ERR  : %s\n",strerror(errno));           /*显示错误信息。*/
16    }
17    else                                               /*另一种情况。*/
18    {
19       printf("created file %s.\n",path);              /*显示创建成功。*/
20    }
21  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.3.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
created file /root/tmp.txt .
```

提示：与创建目录不同的是，当再次运行这个程序时，也能创建同名的文件。这时新创建的文件会覆盖以前的文件。

【代码解析】代码第 09 行指定要创建的文件的路径字符串，代码第 11 行则调用 create 函数执行创建文件操作。

13.4.2　删除文件函数 remove

函数 remove 的作用是删除一个文件。这个函数的使用方法如下所示。

```
01  int remove(char *pathname);
```

参数 remove 是一个字符串，表示需要删除的目录。文件删除成功则返回 0，则否返回-1。要使用这个函数，需要在程序的最前面包含下面的头文件。

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

函数 remove 在删除文件时，可能发生权限不足或目录不存在的问题。需要在程序中设置 errno 来捕获错误信息。下面用 remove 函数删除上一节创建的文件/root/tmp.txt。

【范例 13-6】使用 remove 函数删除指定文件，示例代码如下所示。

示例代码 13-6

```
01  #include <stdio.h>
02  #include <sys/types.h>
03  #include <sys/stat.h>
04  #include <fcntl.h>
05  #include <errno.h>                                   /*包含头文件。*/
06  main()
07  {
```

```
08     extern int errno;                                    /*定义一个错误号。*/
09     char *path="/root/tmp.txt";                          /*定义要删除的文件名。*/
10
11     if(remove(path)==0)                                  /*删除文件。*/
12     {
13       printf("Deleted file %s.\n",path);                 /*显示删除成功。*/
14     }
15     else                                                 /*另一种情况。*/
16     {
17       printf("cant't delete the file %s.\n",path);      /*显示删除失败。*/
18       printf("errno: %d\n",errno);                       /*显示错误号。*/
19       printf("ERR  : %s\n",strerror(errno));             /*显示错误信息。*/
20     }
21   }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.4.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示已经删除了这个文件。

```
Deleted file /root/tmp.txt.
```

当再次运行这个程序时，显示的结果如下所示。因为需要删除的目录在上一次程序运行时已经删除，所以找不到要删除的文件。

```
cant't delete the file /root/tmp.txt.
errno: 2
ERR  : No such file or directory
```

注意：如果要删除的文件不存在，将产生编号为 2 的错误，在编程时需要注意捕捉此错误。

【代码解析】代码第 09 行指定要删除的文件的路径字符串，代码第 11 行则调用 remove 函数执行删除文件操作。

14.4.3 建立临时文件函数 mkstemp

所谓临时文件，指的是程序运行时为了存储中间数据建立的文件。计算机重启时，这些文件会自动删除。如果在程序运行时，需要把文件短时间地写到磁盘上，可以使用 mkstemp 函数创建一个临时文件。mkstemp 函数的使用方法如下所示。

```
01   int mkstemp(char *template);
```

参数 template 表示需要建立临时文件的文件名字符串。文件名字符串中最后 6 个字符必须是 XXXXXX。mkstemp 函数会以可读写模式和 0600 权限来打开该文件。如果文件不存在，则会建立这个文件。返回值是打开文件的编号，如果文件建立不成功，则返回-1。该函数可能发生的错误和产生的错误代码如下所示。可以用 errno 来捕获错误信息。

- EEXIST：文件同名错误。
- EINVAL：参数 template 字符串最后 6 个字符非 XXXXXX。

注意：参数 template 所指的文件名称字符串必须声明为数组，用下面这种声明数组的方法声明。

```
01 char template[] ="template-XXXXXX";
```

使用下面这种声明字符串的方法声明的 template 是不能运行的。

```
01 char *template = "template-XXXXXX";
```

下面是使用 mkstemp 函数建立文件的实例。

【范例 13–7】使用 mkstemp 函数创建临时文件，示例代码如下所示。

示例代码 13-7

```
01  #include <stdio.h>
02  #include <stdlib.h>                                    /*包含头文件。*/
03  main()
04  {
05    extern int errno;                                    /*设置一个错误号。*/
06    char path[]="mytemp-XXXXXX";                         /*定义文件名。*/
07
08    if(mkstemp(path)!=-1)                                /*建立一个临时文件。*/
09    {
10      printf("created temp file %s.\n",path);            /*显示文件已经建立。*/
11    }
12    else                                                 /*另一种情况。*/
13    {
14      printf("cant't create temp file %s.\n",path);      /*显示不能创建临时文件。*/
15      printf("errno: %d\n",errno);                       /*显示错误号。*/
16      printf("ERR  : %s\n",strerror(errno));             /*显示错误信息。*/
17    }
18  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.5.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示已经建立了一个临时文件 mytemp-USfaDc。

```
created temp file mytemp-USfaDc.
```

再次运行这个程序，显示的结果如下所示。显示已经建立了一个临时文件 mytemp-hL1La9。系统会自动生成一个不同名的文件名来建立临时文件。

```
created temp file mytemp-hL1La9.
```

提示：临时文件的文件名由系统随机生成。

在终端中输入“ls”命令，显示的结果如下所示。可以在当前文件夹中查看到这两个临时文件。

```
13.1.c  13.3.c  13.eer.1.o  a.out  mytemp-hL1La9  tmp.c
13.2.c  13.4.c  13.eer.2.c  a.txt  mytemp-USfaDc  tmp.o
```

【代码解析】代码第 08 行调用 mktemp 函数创建临时文件，临时文件的路径字符串将由 mktemp 函数随机产生。

13.5　文件的打开与关闭

文件的打开指的是从磁盘中找到一个文件，返回一个整型的打开文件顺序编号。打开的文

件处于可读可写状态。文件的关闭指的是释放打开的文件，使文件处于不可读写的状态。

13.5.1 打开文件函数 open

函数 open 的作用是打开一个文件，使文件处于可读写的状态。这个函数的使用方法如下所示。

```
01  int open(char *pathname, int flags);
02  int open(char *pathname,int flags, mode_t mode);
```

13.5.2 文件打开方式的设置

在上一节所示函数的参数列表中，pathname 表示要打开文件的路径字符串。参数 flags 是系统定义的一些整型常数，表示文件的打开方式。Flags 的可选值如下所示。

- O_RDONLY：以只读方式打开文件。
- O_WRONLY：以只写方式打开文件。
- O_RDWR：以可读写方式打开文件。

上述三种旗标是互斥的，也就是不可同时使用，但可与下列的旗标利用“|”运算符组合。

- O_CREAT：若要打开的文件不存在则自动建立该文件。
- O_EXCL：如果 O_CREAT 已被设置，此指令会去检查文件是否存在。文件若不存在，则建立该文件，否则将导致打开文件错误。此外，若 O_CREAT 与 O_EXCL 同时设置时，如果要打开的文件为一个链接，则会打开失败。
- O_NOCTTY：如果要打开的文件为终端机设备，则不会将该终端机当成进程控制终端机。

13.5.3 打开文件的权限

打开文件时，如果没有这个文件，则会自动新建一个文件。在新建文件时需要设置新建文件权限。系统为参数 mode 定义了下面这些常数，可以直接使用这些常数来设置文件的权限。这些权限设置只有在建立新文件时才会有效。

- S_IRWXU：00700 权限，该文件所有者具有可读可写可执行的权限。
- S_IRUSR 或 S_IREAD：00400 权限，该文件所有者具有可读取的权限。
- S_IWUSR 或 S_IWRITE：00200 权限，该文件所有者具有可写入的权限。
- S_IXUSR 或 S_IEXEC：00100 权限，该文件所有者具有可执行的权限。

13.5.4 文件打开实例

本节讲解一个文件打开实例，用 open 函数打开一个文件。调用这个函数时，如果正确地打开了这个文件，则返回这个文件的打开编号。如果打开失败，则返回 0 。使用这个函数之前需要在程序的最前面包含下面这些头文件。

```
01  #include<sys/types.h>
02  #include<sys/stat.h>
03  #include<fcntl.h>
```

open 函数可能发生下面这些错误。可用 errno 捕获打开文件时发生的错误。下面的程序，是使用 open 函数打开一个文件的实例。

【范例 13-8】使用 open 函数打开文件，示例代码如下所示。

代码 13-8

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

```
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                                  /*包含头文件。*/
07  main()
08  {
09    int fd,fd1;                                      /*定义打开文件的编号。*/
10    char path[]="/root/txt1.txt";                    /*定义需要打开的文件。*/
11    extern int errno;                                /*定义错误号。*/
12    fd=open(path,O_WRONLY,0766);                     /*打开文件，只读，不能自动新建。*/
13    if(fd!=-1)                                       /*如果返回值不为-1。*/
14     {
15       printf("opened file %s  .\n",path);           /*显示已经打开文件。*/
16     }
17    else                                             /*另一种情况。*/
18     {
19       printf("cant't open file %s.\n",path);        /*显示不能打开文件。*/
20       printf("errno: %d\n",errno);                  /*显示错误号。*/
21       printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
22     }
23
24    fd1=open(path,O_WRONLY|O_CREAT,0766);            /*打开文件，如果没有文件则自动新建。*/
25    if(fd1!=-1)                                      /*如果返回的文件号不为-1。*/
26     {
27       printf("opened file %s  .\n",path);           /*显示文件新建成功。*/
28     }
29    else                                             /*另一种情况。*/
30     {
31       printf("cant't open file %s.\n",path);        /*显示不能打开文件。*/
32       printf("errno: %d\n",errno);                  /*显示错误号。*/
33       printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
34     }
35  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.6.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是以只读的方式打开文件，没有这个文件时显示打开错误。第二次在打开的 Flags 参数中加入了 O_CREAT，表示没有文件时新建这个文件。显示文件打开成功。

```
cant't open file /root/txt1.txt.
errno: 2
ERR  : No such file or directory
opened file /root/txt1.txt  .
```

输入“cd ~”命令进行到用户的根目录。然后输入“ls”命令，可以发现目录中有一个文件 txt1.txt。

【代码解析】代码第 12 行与代码第 24 行分别调用 open 函数打开文件，前者为只读方式，后者则以可写入方式打开（如果文件不存在，则被自动创建）。

13.5.5　关闭文件函数 close

函数 close 的作用是关闭一个已经打开的文件。使用完文件后则可使用 close 函数关闭该文

件，这个操作会让数据写回磁盘，并释放该文件所占用的资源。该函数的使用方法如下所示。

```
01  int close(int fd);
```

参数 fd 是用 open 函数打开文件时返回的打开序号。如果成功关闭文件则返回 0，发生错误则返回-1。在进程结束时，系统会自动关闭已打开的文件，但仍建议在程序中关闭文件，并检查返回值是否正确。在关闭文件时，可能返回 EBADF 错误，表示需要关闭的文件号不存在。在使用这个函数时，需要在程序的前面包含下面的头文件。

```
01  #include<unistd.h>
```

下面是一个实例，用 open 函数打开一个文件以后，再用 close 函数关闭这个文件。

【范例 13-9】使用 open 函数打开文件，使用 close 函数关闭已打开的文件，示例代码如下所示。

示例代码 13-9

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                                  /*包含头文件。*/
07  main()
08  {
09    int fd;                                          /*定义一个文件编号。*/
10    char path[]="/root/txt1.txt";                    /*定义需要打开的文件名。*/
11    extern int errno;                                /*定义一个错误号。*/
12
13    fd=open(path,O_WRONLY|O_CREAT,0766);      /*打开文件，如果没有这个文件则新建。*/
14    if(fd!=-1)                                       /*打开成功。*/
15    {
16      printf("opened file %s  .\n",path);            /*显示打开成功。*/
17    }
18    else                                             /*打开不成功。*/
19    {
20      printf("cant't open file %s.\n",path);         /*显示打开不成功。*/
21      printf("errno: %d\n",errno);                   /*显示错误号。*/
22      printf("ERR  : %s\n",strerror(errno));         /*显示错误信息。*/
23    }
24
25    if(close(fd)==0)                                 /*关闭打开的文件。*/
26    {
27      printf("closed.\n");                           /*显示关闭成功。*/
28    }
29    else                                             /*另一种情况。*/
30    {
31      printf("close file %s error.\n",path);         /*显示关闭不成功。*/
32      printf("errno: %d\n",errno);                   /*显示错误号。*/
33      printf("ERR  : %s\n",strerror(errno));         /*显示错误信息。*/
34    }
35
36     if(close(1156)==0)                              /*关闭一个不存在的文件号。*/
37    {
38      printf("closed.\n");                           /*如果关闭成功则显示。*/
39    }
40    else
41    {
42      printf("close file %s error.\n",path);         /*关闭不成功则显示错误。*/
```

```
43          printf("errno: %d\n",errno);                    /*显示错误号。*/
44          printf("ERR  : %s\n",strerror(errno));          /*显示错误信息。*/
45      }
46  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次关闭的是已经打开的一个文件，可以正常关闭。第二次关闭的是一个不存在的文件号，显示没有这个文件。

```
opened file /root/txt1.txt  .
closed.
close file /root/txt1.txt error.
errno: 9
ERR  : Bad file descriptor
```

【代码解析】本程序通过 open 函数打开文件，close 函数关闭函数。在关闭文件时使用了错误的参数，演示了关闭一个不存在的文件时可能遇到的问题。

13.6　文件读写

文件读写指的是从文件中读出信息或将信息写入到文件中。文件读取是使用 read 函数来实现的。文件保存是通过 write 函数来实现的。在进行文件保存的操作时，只是在文件的缓冲区中操作，可能没有立即写入到文件中。需要使用 sync 或 fsynx 函数将缓冲区中的数据写入到文件中。

13.6.1　在文件中写字符串函数 write

函数 write 可以把一个字符串写入到一个已经打开的文件中。这个函数的使用方法如下所示。

```
01  ssize_t write (int fd,void * buf,size_t count);
```

在参数列表中，fd 是已经打开文件的文件号。buf 是需要写入的字符串。count 是一个整型数，表示需要写入的字符个数。size_t 是一个相当于整型的数据类型，表示需要写入的字节的数目。将字符串写入文件以后，文件的读写位置也会随之移动。如果写入成功，write 函数会返回实际写入的字节数。发生错误时则返回-1，可以用 errno 来捕获发生的错误。可能发生的错误如下所示。

- EINTR：此操作被其他操作中断。
- EAGAIN：当前打开文件是以不可写的方式打开时，不能写入文件。
- EADF：参数 fd 不是有效的文件编号，或该文件已关闭。

下面是这个函数的使用实例。用 open 函数打开一个文件，将一个字符串写入到这个文件中，然后关闭文件。

【范例 13-10】使用 write 函数向已打开的文件中写入字符串，示例代码如下所示。

示例代码 13-10

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
```

```
06  #include <errno.h>                                        /*包含头文件。*/
07  main()
08  {
09    int fd;                                                 /*定义打开的文件号。*/
10    char path[]="/root/txt1.txt";                           /*定义要写入文件的路径。*/
11    char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";    /*  定
立要写入的字符串。*/
12    extern int errno;                                       /*使用错误号。*/
13
14    fd=open(path,O_WRONLY|O_CREAT);                         /*打开文件。*/
15    if(fd!=-1)                                              /*打开文件是否成功。*/
16    {
17      printf("opened file %s  .\n",path);                   /*文件打开成功。*/
18    }
19    else
20    {
21      printf("cant't open file %s.\n",path);                /*文件打开不成功。*/
22       printf("errno: %d\n",errno);                         /*显示错误号。*/
23       printf("ERR  : %s\n",strerror(errno));               /*显示错误信息。*/
24    }
25
26    write(fd,s,sizeof(s));                                  /*将内容写入到文件。*/
27    close(fd);                                              /*关闭文件。*/
28    printf("Done");                                         /*显示信息。*/
29  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.8.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt .
Done
```

这时，在终端中输入下面的命令，查看写入文件的信息。

```
vim  /root/txt1.txt
```

vim 显示的/root/txt1.txt 文件内容如下所示。

```
hello ,Linux.
I've leart C program for two weeks.
```

【代码解析】代码第 14 行调用 open 函数打开文件，代码第 26 行调用 write 函数向文件中写入字符串 s（使用 sizeof 可以返回字符串 s 的长度）。

13.6.2 读取文件函数 read

函数 read 可以从一个打开的文件中读取字符串。这个函数的使用方法如下所示。

```
01  ssize_t read(int fd,void *buf ,size_t count);
```

在参数列表中，fd 表示已经打开文件的编号。buf 是空指针，读取的内容会返回到这个指针指向的字符串。count 表示需要读取字符的个数。返回值表示实例读取到的字符的个数。如果返回值为 0，表示已经到达文件末尾或文件中没有内容供读取。在读文件时，文件的读写位置会随着读取到的字节移动。

当有错误发生时，返回值为-1，可以用 errno 来捕获错误编号。可能返回的错误编号如下

所示。

- EINTR：此操作被其他操作中断。
- EAGAIN：当前打开文件是以不可写的方式打开时，不能写入文件。
- EADF：参数 fd 不是有效的文件编号，或该文件已关闭。

下面是使用 read 函数进行文件读取的实例，可以读取上一节中写入文件的内容。然后显示出读取的字符串和实际读取到的字符数。

【范例 13-11】使用 read 函数读取文件内容，示例代码如下所示。

示例代码 13-11

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                              /*包含头文件。*/
07
08  main()
09  {
10    int fd;                                      /*定义文件号。*/
11    char path[]="/root/txt1.txt";                /*定义打开的文件名。*/
12    int size;
13    char s[100];                                 /*定义一个字符串。*/
14    extern int errno;                            /*使用错误号。*/
15
16    fd=open(path,O_RDONLY);                      /*打开文件。*/
17    if(fd!=-1)                                   /*打开成功。*/
18    {
19      printf("opened file %s  .\n",path);
20    }
21    else
22    {
23      printf("cant't open file %s.\n",path);     /*打开不成功。*/
24      printf("errno: %d\n",errno);               /*显示错误号。*/
25      printf("ERR  : %s\n",strerror(errno));     /*显示错误信息。*/
26    }
27
28    size=read(fd,s,sizeof(s));                   /*读取文件内容。*/
29    close(fd);                                   /*关闭文件。*/
30    printf("%s\n",s);                            /*输出。*/
31    printf("%d\n",size);                         /*输出返回的字节数。*/
32  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.9.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序显示了文件打开成功和上一节写入的字符串。最后一行显示了实际读取到的字符数。

```
opened file /root/txt1.txt  .
hello ,Linux.
I've leart C program for two weeks.
51
```

【代码解析】代码第 16~26 行打开指定的文件，代码第 28 行则从打开的文件中读取数据到

字符数组中。代码第 29 行关闭打开的文件，最后输出从文件中读取的内容和读取的字节数。

13.6.3 文件读写位置的移动

每一个已打开的文件都有一个读写位置。当打开文件时通常读写位置是指向文件开头，若是以附加的方式打开文件，则读写位置会指向文件末尾。read 或 write 函数读写文件时，读写位置会随之相应移动。可以用 lseek 函数在文件内容中的位置上面移动，这样就可以在文件中不同的位置上进行读写。这个函数的使用方法如下所示。

```
01  off_t lseek(int fd,off_t offset ,int whence);
```

在参数列表中，fd 表示用 open 函数打开的文件返回编号。参数 offset 是根据参数 whence 来移动读写位置的位移数。whence 为系统定义的常量，可能有下面这些可能的赋值。

- SEEK_SET：参数 offset 即为新的读写位置。
- SEEK_CUR：以目前的读写位置往后增加 offset 个位移量。
- SEEK_END：将读写位置指向文件尾后再增加 offset 个位移量。

当 whence 值为 SEEK_CUR 或 SEEK_END 时，参数 offet 允许负值出现。这时表示在当前位置上向前移动若干个字节。下面是一些常用的文件位置移动方式。

- lseek（int fd，0,SEEK_SET）：将读写位置移到文件开头。
- lseek（int fd，0,SEEK_END）：将读写位置移到文件结尾。
- lseek（int fd，0,SEEK_CUR）：取得当前的文件位置。

函数调用成功时返回这个文件当前的读写位置，即距文件开头多少个字节。若有错误则返回-1，errno 会存放错误代码。这个函数可能产生的错误代码如下所示。

- EBADF：传入的参数不是一个已经打开的文件。
- EINVAL：给入的 whence 参数不合理。
- EOVERFLOW：给入的移动参数导致文件头指针指向了文件头以前，产生了溢出错误。

要使用这个函数，需要在程序的前面包含下面的头文件。

```
01  #include<sys/types.h>
02  #include<unistd.h>
```

下面是 lseek 函数的使用实例。打开上一节所使用的文本文件，在文件读取时，用 lseek 函数移动文件的读写位置。

【范例 13-12】使用 lseek 函数移动文件的读写位置，示例代码如下所示。

示例代码 13-12

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                              /*包含头文件。*/
07  main()
08  {
09     int fd;                                     /*定义文件号。*/
10     char path[]="/root/txt1.txt";               /*定义文件路径。*/
11     int size;
12     char s[100]="";                             /*定义一个字符串。*/
13     extern int errno;                           /*使用错误号。*/
14
15     fd=open(path,O_RDONLY);                     /*打开文件。*/
16     if(fd!=-1)                                  /*打开成功。*/
```

```
17      {
18        printf("opened file %s  .\n",path);
19      }
20      else
21      {
22        printf("cant't open file %s.\n",path);        /*打开失败。*/
23         printf("errno: %d\n",errno);                 /*显示错误号。*/
24         printf("ERR  : %s\n",strerror(errno));       /*显示错误信息。*/
25      }
26
27      size=read(fd,s,3);                              /*读取 3 个字符。*/
28      printf("%d  :",size);                           /*输出返回的字符数。*/
29      printf("%s\n",s);                               /*输出 3 个读取的字符。*/
30
31      size=read(fd,s,3) ;                             /*读取后面的 3 个字符。*/
32      printf("%d  :",size);                           /*输出返回的字符数。*/
33      printf("%s\n",s);                               /*输出 3 个读取的字符。*/
34
35      lseek(fd,8,SEEK_SET);                           /*移动到第 8 个字符。*/
36      size=read(fd,s,3);                              /*读取后面的 3 个字符。*/
37      printf("%d  :",size);                           /*输出返回的字符数。*/
38      printf("%s\n",s);                               /*输出 3 个读取的字符。*/
39
40      lseek(fd,0,SEEK_SET);                           /*移动到第 0 个字符。*/
41      size=read(fd,s,3);                              /*读取后面的 3 个字符。*/
42      printf("%d  :",size);                           /*输出返回的字符数。*/
43      printf("%s\n",s);                               /*输出 3 个读取的字符。*/
44
45      close(fd);                                      /*关闭文件。*/
46  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.10.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

注意：文件读写时，空格也是一个字符，但是输出时看不到这个字符。

```
3  :hel
3  :lo
3  :inu
3  :hel
```

【代码解析】代码第 15 行调用 open 函数打开文件，代码第 35 行与代码第 40 行对文件的读写位置进行重定位，紧接着输出文件中的指定内容。

13.6.4　将缓冲区数据写入到磁盘函数 sync

所谓缓冲区，是 Linux 系统对文件的一种处理方式。在对文件进行写操作时，并没有立即把文件写入到磁盘，而是把文件写入到缓冲区中。如果需要把文件立即写入到磁盘，可以用 sync 函数。用这个函数强制写入缓冲区数据的好处是保证数据有同步。这个函数的使用方法如下所示。

```
01  int sync(void)
```

这个函数会对当前程序打开的所有文件进行处理，将缓冲区中的内容写入到文件。函数没

有参数，返回值为 0。这个函数一般不会产生错误。要使用这个函数，需要在程序中包含下面的头文件。

```
01  #include<unistd.h>
```

提示： Linux 系统下还有一个 sync 指令能达到同样的效果，可以在命令行中调用，用于将缓冲区中的数据立即写入磁盘中。

下面是这个函数的使用实例。

【范例 13–13】 使用 sync 函数将缓冲区数据写回磁盘，示例代码如下所示。

示例代码 13-13

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                                    /*包含头文件。*/
07
08  main()
09  {
10     int fd;                                           /*定义文件号。*/
11     char path[]="/root/txt1.txt";                     /*要打开的文件。*/
12     char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";   /* 要
写入的字符串。*/
13     extern int errno;                                 /*定义错误号。*/
14
15     fd=open(path,O_WRONLY|O_CREAT);                   /*打开文件。*/
16     if(fd!=-1)                                        /*文件打开成功。*/
17     {
18       printf("opened file %s  .\n",path);
19     }
20     else
21     {
22       printf("cant't open file %s.\n",path);         /*文件打开失败。*/
23        printf("errno: %d\n",errno);                  /*显示错误号。*/
24        printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
25     }
26     write(fd,s,sizeof(s));                            /*写入文件。*/
27     sync();                                           /*将缓冲区数据写入磁盘。*/
28     printf("sync function done. \n");                 /*输出信息。*/
29     close(fd);                                        /*关闭文件。*/
30  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.11.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt  .
sync function done.
```

【代码解析】代码第 26 行调用 write 函数将指定的内容写入文件（缓冲区）中，代码第 27 行调用 sync 函数立即将缓冲区中的数据写入磁盘。

提示：Linux 下的这种缓冲区机制可以提高磁盘的写入效率，但是，也有可能导致数据丢失（例如：主机突然掉电，在缓冲区中的数据将会丢失。）

13.6.5　将缓冲区数据写入到磁盘函数 fsync

函数 fsync 的作用是将缓冲区的数据写入磁盘。这个函数的使用方法如下所示。

```
01   int fsync(int fd);
```

参数 fd 是 open 函数打开文件时返回的编号。函数如果执行成功，则返回 0，否则返回-1。

提示：与 sync 函数不同的是，函数 fsync 可以指定打开文件的编号，执行以后会返回一个值。

在使用这个函数时，需要在文件前面包含下面的头文件。

```
01   #include<unistd.h>
```

函数可能发生下面这些错误，可以用 errno 捕获错误。

- EBADF：参数 fd 不是一个正确的文件打开编号或者文件不能写入。
- EIO：发生了 I/O 错误。前面的章节已经讲述过 I/O 错误。
- EROFS 或 EINVAL：fd 是一个特殊的文件，不能够写入内容。

下面的程序是 fsync 函数的使用实例。在文件中写入内容以后，用 fsync 函数将文件写入到缓冲区。

【范例 13-14】使用 fsync 函数将缓冲区数据写回磁盘，示例代码如下所示。

示例代码 13-14

```
01   #include <stdio.h>
02   #include <unistd.h>
03   #include <sys/types.h>
04   #include <sys/stat.h>
05   #include <fcntl.h>
06   #include <errno.h>                                    /*包含头文件。*/
07
08   main()
09   {
10     int fd;                                            /*定义文件号。*/
11     char path[]="/root/txt1.txt";                      /*要打开的文件。*/
12     char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";   /* 要
写入的字符串。*/
13     extern int errno;                                  /*定义错误号。*/
14
15     fd=open(path,O_WRONLY|O_CREAT);                    /*打开文件。*/
16     if(fd!=-1)                                         /*文件打开成功。*/
17     {
18       printf("opened file %s  .\n",path);
19     }
20     else
21     {
22       printf("cant't open file %s.\n",path);          /*文件打开失败。*/
23        printf("errno: %d\n",errno);                    /*显示错误号。*/
24        printf("ERR  : %s\n",strerror(errno));          /*显示错误信息。*/
25     }
26
```

```
27      write(fd,s,sizeof(s));                          /*写入文件。*/
28      if(fsync(fd)==0)                                /*将缓冲区数据写入磁盘。*/
29      {
30          printf("fsync function done.\n");           /*写入成功。*/
31      }
32      else
33      {
34        printf("fsync function failed. \n");          /*写入失败。*/
35      }
36      close(fd);                                      /*关闭文件。*/
37  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.12.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt  .
fsync function done.
```

【代码解析】代码第 27 行调用 write 函数将指定的内容写入文件（缓冲区）中，代码第 28 行调用 fsync 函数将已打开文件指针 fp 所代表的文件的缓冲区内容立即写入磁盘。

注意： 函数 sync 与函数 fsync 的不同之处在于，函数 sync 是将操作系统的缓冲区数据都写入磁盘，而函数 fsync 则仅将指定文件的缓冲区数据写入磁盘。

13.7 文件锁定

所谓文件锁定，指的是以独占的方式打开文件。一个程序打开文件以后，其他的程序不能读取或写入文件。文件锁定有利于文件内容的一致性。本节将讲解文件的锁定权限问题。

13.7.1 文件锁定的理解

当多个应用程序同时访问同一个文件时，可能导致文件的内容不一致。例如一个文件中的数据是一个账户金额。用户打开这个文件读取数据，进行处理以后将写结果写入到文件。如果文件没有进行锁定，可以发生下面这种错误。

- 假设置文件中的数据为 10000。用户 A 打开文件读取这个数据，读取了这个数据。但是还没有及时写入。
- 这时，用户 B 读取这个数据，将结果加 10000。然后将结果 20000 写入到这个文件。
- 用户 A 的处理时间较长，比用户 B 早读入数据，但后写入数据。A 将数据加 1000，得到 11000，然后将数据写入文件。

这时就发生了一个错误，用户 B 的数据丢失。这时需要在打开文件时进行文件锁定，正确的做法如下所示。

（1）假设置文件中的数据为 10000。用户 A 打开文件读取这个数据，读取了这个数据。但是还不能及时写入，于是将文件加一把锁，不允许别的用户访问。

（2）用户 B 访问这个文件时，文件已经加锁，无法访问。于是等待用户 A 完成数据访问。

（3）用户 A 完成数据处理以后，将信息写入到文件，这时解除对文件的锁定。

（4）这时用户 B 可以对文件进行访问。访问的同时，对文件添加一个锁定。在解除锁定以前，其他用户不能访问这个文件。

13.7.2　文件锁定函数 flock

在访问一个文件时，可以用 flock 函数对文件进行锁定。防止其他用户同时访问这个文件时发生数据不一致的情况。这个函数的使用方法如下所示。

```
01   int flock(int fd,int operation);
```

在参数列表中，fd 是 open 函数打开文件时返回的打开序号。operation 是系统定义的一些整型常量，可能的取值和含义如下所示。

- LOCK_SH：建立共享锁定，其他的程序可以同时访问这个文件。多个程序可同时对同一个文件作共享锁定。
- LOCK_EX：建立互斥锁定，其他用户不能同时访问这个文件。

注意：一个文件同时只有一个互斥锁定。单一文件不能同时建立共享锁定与互斥锁定。

- LOCK_UN：解除文件锁定状态。
- LOCK_NB：无法建立锁定时，此操作可不被阻断，马上返回进程。通常与 LOCK_SH 或 LOCK_EX 做 OR（|）组合。

当文件锁定成功时，会返回 0，否则返回-1。可以用 errno 来捕获发生的错误。下面的程序，是 flock 函数的使用实例。打开文件以后，对文件建立一个锁定。当其他程序打开这个文件时，则无法打开锁定的文件。

【范例 13-15】使用 flock 函数锁定指定文件，防止其他程序再次打开文件，示例代码如下所示。

示例代码 13-15

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
06  #include <errno.h>                              /*包含头文件。*/
07
08  main()
09  {
10    int fd,i;                                     /*定义变量。*/
11    char path[]="/root/txt1.txt";                 /*要访问的文件。*/
12    extern int errno;                             /*定义错误号。*/
13
14    fd=open(path,O_WRONLY|O_CREAT);               /*打开文件。*/
15    if(fd!=-1)                                    /*打开成功。*/
16    {
17      printf("opened file %s  .\n",path);
18      printf("please input a number to lock the file.\n");  /*输入一个数字。*/
19      scanf("%d",&i);                             /*输入。*/
20      if(flock(fd,LOCK_EX)==0)                    /*锁定文件。*/
21      {
22        printf("the file was locked.\n");         /*输出信息。*/
23       }
```

```
24          else
25          {
26            printf("the file was not locked.\n");   /*文件锁定失败。*/
27          }
28        printf("please input a number to unlock the file.\n");  /*提示输入。*/
29        scanf("%d",&i);                                         /*输入。*/
30        if(flock(fd,LOCK_UN)==0)                                /*解除文件锁定。*/
31        {
32          printf("the file was unlocked.\n");       /*输出解除锁定成功。*/
33          }
34          else
35          {
36            printf("the file was not unlocked.\n"); /*输出解除锁定失败。*/
37          }
38        close(fd);                                  /*关闭文件。*/
39      }
40      else
41      {
42        printf("cant't open file %s.\n",path);      /*不能打开文件的情况。*/
43          printf("errno: %d\n",errno);              /*显示错误号。*/
44          printf("ERR  : %s\n",strerror(errno));    /*显示错误信息。*/
45      }
46  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.13.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示输入一个数字锁定文件。输入一个数字以后，显示文件锁定。这时输入一个数字可以解除文件锁定。

```
opened file /root/txt1.txt  .
please input a number to lock the file.
```

【代码解析】代码第 14 行调用 open 函数打开文件，代码第 20 行则调用 flock 函数对文件进行互斥锁定，当锁定成功后，其他用户则无法访问被锁定的文件。

13.7.3 文件锁定函数 fcntl

函数 fcntl 的作用是对一个文件进行锁定。

提示：与 flock 函数不同的是，fcntl 可以设定对文件的某一部分进行锁定。

该函数的使用方法如下所示。

```
01  int fcntl(int fd , int cmd);
02  int fcntl(int fd,int cmd,long arg);
03  int fcntl(int fd,int cmd,struct flock *lock);
```

这个函数的参数列表有多种形式。在编译时，会自动与参数进行匹配。参数 fd 是 open 函数打开文件时返回的文件编号。cmd 是系统定义的整型常量，用来设置文件的打开方式，有下面几种可选方式。

- F_DUPFD：用来查找大于或等于参数 arg 的最小且仍未使用的文件编号，并且复制参数 fd 的文件编号。执行成功则返回新复制的文件编号。
- F_GETFD：取得 close-on-exec 标志。若此标志的参数 arg 的 FD_CLOEXEC 位为 0，

代表在调用 exec()相关函数时文件将不会关闭。

- F_SETFD：设置 close-on-exec 旗标。该旗标以参数 arg 的 FD_CLOEXEC 位决定。
- F_GETFL：取得文件描述词状态旗标，此旗标为 open 的参数返回序号。
- F_SETFL：设置文件描述词状态旗标，参数 arg 为新旗标，但只允许 O_APPEND、O_NONBLOCK 和 O_ASYNC 位的改变，其他位的改变将不受影响。
- F_GETLK：取得文件锁定的状态。
- F_SETLK：设置文件锁定的状态。此时 flcok 结构的 l_type 值必须是 F_RDLCK、F_WRLCK 或 F_UNLCK。如果无法建立锁定，则返回-1，错误代码为 EACCES 或 EAGAIN。
- F_SETLKW 或 F_SETLK ：这两个参数的作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。

参数 lock 指针为 flock 类型的结构体指针，用来设置锁定文件的一个区域。flock 结构体的定义方式如下所示。

```
01  struct flock
02  {
03     short int l_type;        /*文件的锁定状态。*/
04     short int l_whence;      /*设定 l_start 位置。*/
05     off_t l_start;           /*锁定区域的开头位置，从这个地方开始锁定文件的内容。*/
06     off_t l_len;             /*锁定文件区域的大小。*/
07     pid_t l_pid;             /*锁定文件的进程。*/
08  };
```

l_type 有下面三种可选的参数。

- F_RDLCK：建立一个供读取用的锁定。
- F_WRLCK：建立一个供写入用的锁定。
- F_UNLCK：删除之前建立的锁定方式。

l_whence 有下面三种可选设置方式。

- SEEK_SET：以文件开头为锁定的起始位置。
- SEEK_CUR：以文件当前的读写位置为锁定的起始位置。
- SEEK_END：以文件结尾为锁定的起始位置。

fcntl 函数执行成功则返回 0，失败则返回-1。这个函数可能发生下面的错误，可以用 errno 来捕获发生的错误。在使用这个函数时，需要在程序的最前面包含下面的头文件。

```
01  #include <unistd.h>
02  #include <fcntl.h>
```

13.7.4　文件锁定函数 fcntl 使用实例

本节讲解 fcntl 函数的使用实例。打开一个文件后，用 fcntl 函数来锁定文件的一部分字节。在程序中，需要定义一个 flock 类型的结构体，然后设置这个结构体各个成员的变量。用这个结构体的指针作为 fcntl 函数的参数。

【范例 13-16】使用 fcntl 函数锁定文件，示例代码如下所示。

示例代码 13-16

```
01  #include <stdio.h>
02  #include <unistd.h>
03  #include <sys/types.h>
04  #include <sys/stat.h>
05  #include <fcntl.h>
```

```
06  #include <errno.h>                                  /*包含头文件。*/
07
08  main()
09  {
10    int fd;                                          /*定义打开的文件名。*/
11    char path[]="/root/txt1.txt";                    /*定义要打开的文件名字符串。*/
12    struct flock fl;                                 /*定义一个 flock 结构体。*/
13    char s[]="hello ,Linux.\nI've leart C program for two weeks.\n";
14    extern int errno;                                /*定义一个错误号。*/
15
16    fd=open(path,O_WRONLY|O_CREAT);                  /*打开文件。*/
17    if(fd!=-1)                                       /*文件打开成功。*/
18    {
19      printf("opened file %s  .\n",path);            /*输出文件打开成功。*/
20    }
21    else
22    {
23      printf("cant't open file %s.\n",path);         /*文件打开失败。*/
24       printf("errno: %d\n",errno);                  /*显示错误号。*/
25       printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
26    }
27
28    fl.l_type=F_RDLCK;                               /*构造 fl 结构体的内容。*/
29    fl.l_whence=SEEK_SET;                            /*文件位置方式。*/
30    fl.l_start=2;                                    /*从第二个字节开始。*/
31    fl.l_len=10;                                     /*一共锁定 8 个字节。*/
32    fl.l_pid=15;                                     /*进程号。*/
33
34    if(fcntl(fd,F_SETLKW,&fl)==0)                    /*锁定文件的这个区域。*/
35    {
36      printf("some string of the file was locked.\n");     /*显示锁定成功。*/
37    }
38    else
39    {
40       printf("locked error.\n");                    /*锁定不成功则显示错误。*/
41       printf("errno: %d\n",errno);                  /*显示错误号。*/
42       printf("ERR  : %s\n",strerror(errno));        /*显示错误信息。*/
43     }
44     close(fd);                                      /*关闭文件。*/
45  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.14.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表示文件从第二个字节开始以后的 10 个字节已经被锁定。

```
some string of the file was locked.
```

【代码解析】

- 代码第 16 行调用 open 函数打开文件。
- 代码第 28~32 行为 flock 结构体类型变量 fl 的成员赋值，以便在调用函数 fcntl 时使用。
- 代码第 34 行调用 fcntl 函数对文件进行锁定操作。

13.8　文件的移动与复制

文件的移动指的是把文件中一个目录转移到另一个目录中。C 程序提供了方便的文件移动函数。文件的复制指的是将文件作一个备份。C 程序没有提示文件复制函数。需要新建一个文件，从原文件中读取内容一次次地写入到新文件中。本节将讲解文件的移动与复制操作。

13.8.1　文件的移动函数 rename

在 Linux 系统中，移动文件有两种方式。一种方式是在同一个分区中移动文件。这种文件移动方式相当于把文件进行重命名。另一种方式是在不同分区之间移动文件。本节只讲前一种文件移动方式。

在同一个分区中移动文件可以用 rename 函数。该函数的使用方式如下所示。

```
01  int rename(char *oldpath, char *newpath);
```

在参数列表中，oldpath 是原文件的路径，newpath 是文件的新路径。rename 函数可以把文件从原路径移动到新路径中。如果文件移动成功，将返回 0，不成功返回-1。文件移动不成功时，可能产生错误。可以用 errno 捕获程序中的错误。

使用这个函数时，需要在程序的最前面包含下面的头文件。

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

13.8.2　rename 函数使用实例

本节讲述一个实例，使用 rename 函数移动一个文件。将/root/a.txt 文件移动到/tmp 目录中，文件名为 b.txt。在使用这个函数时，需要注意对错误信息的处理。

【范例 13-17】是一个 rename 函数移动文件，示例代码如下所示。

示例代码 13-17

```
01  #include <stdio.h>
02  #include <errno.h>                                          /*包含头文件。*/
03
04  main()
05  {
06    char path[]="/root/a.txt";                                /*原文件。*/
07    char path1[]="/root/a11.txt";                             /*一个不存在的文件。*/
08    char newpath[]="/tmp/b.txt";                              /*一个不存在的文件。*/
09    extern int errno;                                         /*错误号。*/
10    if(rename(path,newpath)==0)                               /*移动文件。*/
11    {
12      printf("the file %s was moved to %s .\n",path,newpath); /*移动成功。*/
13    }
14    else
15    {
16      printf("can't move the file %s .\n",path);             /*显示移动失败。*/
17       printf("errno: %d\n",errno);                         /*显示错误号。*/
18       printf("ERR  : %s\n",strerror(errno));               /*显示错误信息。*/
19    }
20
21    if(rename(path1,newpath)==0)                              /*移动一个不存在的文件。*/
22    {
23      printf("the file %s was moved to %s .\n",path1,newpath); /*显示结果。*/
```

```
    }
    else
    {
      printf("can't move the file %s .\n",path1);      /*显示移动不成功。*/
       printf("errno: %d\n",errno);                    /*显示错误号。*/
       printf("ERR  : %s\n",strerror(errno));          /*显示错误信息。*/
    }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.15.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是移动一个已经存在的文件，显示文件移动成功。第二次是移动一个不存在的文件，显示文件移动失败。显示的错误信息是没有这个文件。

```
the file /root/a.txt was moved to /tmp/b.txt .
can't move the file /root/a11.txt .
errno: 2
ERR  : No such file or directory
```

在终端中输入下面的命令，查看/tmp 文件夹下有没有移动的文件。

```
ls /tmp/b.txt
```

显示的结果如下所示。表明/tmp 文件夹下面有程序中移动的文件。

```
/tmp/b.txt
```

【代码解析】

- 代码第 10~19 行调用 rename 函数移动一个存在的文件到新的目录下。
- 代码第 21~30 行调用 rename 函数移动一个不存在的文件到新的目录下，将导致系统出错。

13.8.3 文件复制实例

在 C 程序中，没有直接复制一个文件的函数。如果需要复制一个文件，可以分别打开原文件和目标文件。依次从原文件中读取一定长度的内容，然后写入到新文件中。下面的程序是使用这种方法进行文件复制的实例。

【范例 13-18】文件复制操作实现举例，示例代码如下所示。

示例代码 13-18

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>                                  /*包含头文件。*/

main()
{
   int fd,fd2,size,i=1;
   char path[]="/root/s.txt";                       /*要打开的文件。*/
   char newpath[]="/root/a2.txt";                   /*复制的目录文件。*/
   char buf[100];                                   /*定义一个字符串。*/
   extern int errno;                                /*定义一个错误号。*/

   fd=open(path,O_RDONLY);                          /*打开原文件。*/
```

```
17      fd2=open(newpath,O_WRONLY|O_CREAT);                    /*打开目标文件。*/
18      if(fd!=-1)
19      {
20        printf("opened file %s  .\n",path);                  /*原文件打开成功。*/
21      }
22      else
23      {
24        printf("cant't open file %s.\n",path);               /*原文件打开失败。*/
25         printf("errno: %d\n",errno);                        /*显示错误号。*/
26         printf("ERR  : %s\n",strerror(errno));              /*显示错误信息。*/
27      }
28      for(;i!=0;)
29      {
30          i=read(fd,buf,sizeof(buf));                       /*从原文件中读取内容。*/
31          printf("%d",i);                                   /*输出读取的字节个数。*/
32          printf("%s",buf);                                 /*输出内容。*/
33          if(i==-1)                                         /*到达末尾则结束。*/
34          {
35            break;                                          /*结束循环。*/
36          }
37          else
38          {
39            write(fd2,buf,sizeof(buf));                     /*将读取的内容写入到目标文件中。
*/
40        }
41      }
42      close(fd);                                            /*关闭文件。*/
43      close(fd2);                                           /*关闭文件。*/
44  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.15.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。显示文件已经复制。

```
opened file /root/s.txt  .
file was copied.
```

在终端中输入下面的命令，可以查看到已经复制的文件。

```
ls /root
```

【代码解析】

- 代码第 16 行和代码第 17 行分别调用 open 函数打开要复制的原文件和目标文件。
- 代码第 28~41 行使用 for 循环依次从原文件中读取固定长度的内容写入目标文件中，直到文件内容复制完毕。

13.9　综合练习

在程序中处理的数据，需要记录到文件中才能够长期保存。本章所讲的文章操作可以用来进行各种文件的读写操作。本节将讲述一个文件使用实例，设计一个电话号码管理程序。电话号码的内容使用文件读写函数记录到文件中。在本章的学习中，需要重点理解文件的读写操作。

13.9.1 程序功能分析

在设计一个程序之前，需要对程序的各种功能进行规划，安排到程序的各个模块和实现方式。本节将对这个程序的功能和实现方法进行大体分析。一个电话号码管理程序，需要对电话号码进行添加、删除保存、打开等操作。这个程序需要完成下面这些功能。

（1）进行程序以后，需要有一个选择菜单。
（2）需要添加电话号码，实现数据的输入。
（3）需要删除电话号码，实现数据的管理。
（4）需要进行数据列表，显示所有的电话号码信息。
（5）需要根据一个姓名进行查找，查出这个姓名的数据。
（6）使用文件操作，把信息保存到文件上。
（7）要有文件读取功能，从文件中读取以前保存的数据。

13.9.2 程序的函数

这个程序实现了复杂的功能，需要把这些功能写成不同的函数模块。这样可以把一个复杂的程序拆分成多个独立的简单模块，然后用主函数把这些程序组织到一起。这个程序需要编写下面这些模块。

- 菜单函数，完成菜单的显示和选择。

```
01  int menu();
```

- 显示学生函数，参数是一个结构体，显示结构体的信息。

```
01  void shownum(struct telnum t)
```

- 添加电话函数，完成一个电话号码的添加，返回一个结构体。

```
01  struct telnum addnum()
```

- 查找函数，用户输入一个姓名，查找到这个用户的电话。

```
01  void selectbyname()
```

- 删除电话号码函数。用户输入一个姓名，删除这个用户的电话号码。

```
01  void delenum()
```

- 保存信息功能，将所有电话号码保存到文件上。

```
01  void savetofile()
```

导入电话号码。从文件中读取以前的文件信息。

```
01  void loadfromfile()
```

主函数。完成各个函数的调用。

```
01  main()
```

13.9.3 包含文件

在这个程序中，需要进行字符串比较、字符串复制、错误处理、文件读写等操作。这些操作需要包含相应的头文件。程序的第一部分是包含需要使用的头文件，代码如下所示。

【范例 13-19】包含文件，示例代码如下所示。

示例代码 13-19

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

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>                                    /*包含相关的头文件。*/
```

13.9.4　数据的定义

程序中的电话号码和姓名之间存在着一对一的联系，需要定义一个结构体来体现出这种对应关系。电话号码和计数变量，是全局变量。程序中所有的函数需要访问这些数据。程序的这一部分需要定义一个结构体，定义程序的全局变量。

【范例 13-20】数据定义，示例代码如下所示。

示例代码 13-20

```
struct telnum
{
   char name[7];          /*姓名。*/
   char tel[13];          /*电话号码。*/
};        /*定义保存一个电话号码的结构体。*/
struct telnum num[100];    /*全局变量，一个结构体数组。*/
int i;    /*全局变量，当前记录的条数。*/
```

13.9.5　菜单函数

程序中有很多功能，需要把这些功能用一个菜单供用户选择。每一个菜单有一个编号，用户从键盘输入相对应的编号。函数对输入的编号进行判断，如果输入有效，则返回这个选项。如果输入的选项为 0，则退出这个程序。如果输入的选项无效，则显示错误，要求用户重新选择。这个函数的运行流程如图 13-1 所示。

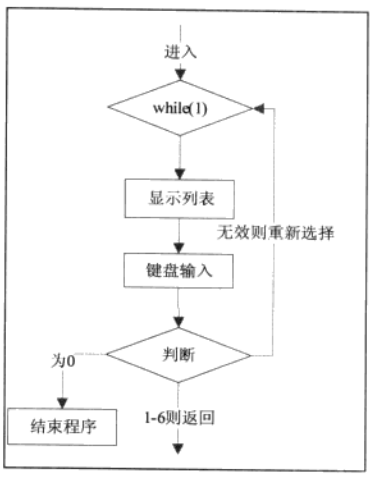


图 13-1　菜单选择的流程图

根据这个菜单选择的原理，编写的程序代码如下所示。

【范例 13-21】菜单函数，示例代码如下所示。

示例代码 13-21

```
int menu()
{
   int i;                                             /*选择的数字。*/
   i=0;                                               /*赋初值为 0。*/
   while(1)
   {
      printf("Please select a menu:\n");              /*提示输入。*/
      printf("      1: add a number.\n");             /*选项。*/
      printf("      2: all the number.\n");
      printf("      3: select a number by name.\n");
      printf("      4: delete a number .\n");
      printf("      5: save to file .\n");
      printf("      6: load numbers from file .\n");
      printf("      0: exit .\n");
      scanf("%d",&i);                                 /*输入信息。*/
      if(i==0)                                        /*如果为 0 则退出。*/
```

```
17        {
18          printf("Byebye.\n");
19          exit(1);                                    /*退出程序。*/
20        }
21        if(i<1 || i>6)                                /*不在正确的范围则显示错误。*/
22        {
23          printf("error.\n");
24          continue;                                   /*下次循环。*/
25        }
26        else
27        {
28          return(i);                                  /*正常范围则返回这个值。*/
29        }
30      }
31  }
```

13.9.6 显示电话信息函数

这个函数的作用是显示一条电话号码的信息。参数是一个电话信息结构体。函数访问这个结构体的成员变量，输出这些信息。完成数据的输出以后，自动结束程序，没有返回值。函数的代码如下所示。

【范例 13-22】显示电话号码函数，示例代码如下所示。

示例代码 13-22

```
01  void shownum(struct telnum t)
02  {
03      printf("Name   :%s\n",t.name);                  /*显示姓名。*/
04      printf("    tel:%s\n",t.tel);                   /*显示电话。*/
05  }
```

13.9.7 添加电话号码函数

这个函数的作用是提示用户输入电话和姓名信息。然后返回一个电话号码结构体。

注意：程序中使用 strcpy 函数将输入的字符串信息复制到结构体的字符串中。这个函数没有参数，返回值是一个结构体。

【范例 13-23】添加号码函数，示例代码如下所示。

示例代码 13-23

```
01  struct telnum addnum()
02  {
03      struct telnum numtmp;                           /*定义一个电话号码结构体变量。*/
04      char na[7],tel[13];                             /*定义两个数组。*/
05      printf("add a telephone number:\n");            /*提示信息。*/
06      printf("please input the name:\n");
07      scanf("%s",na);                                 /*输入姓名。*/
08      printf("please input the num:\n");              /*输入电话号码。*/
09      scanf("%s",tel);
10      strcpy(numtmp.name,na);                         /*变量复制到结构体的成员上。*/
11      strcpy(numtmp.tel,tel);
12      return(numtmp);                                 /*返回这个结构体。*/
13  }
```

13.9.8　按姓名查找函数

这个函数的作用是，提示用户输入一个姓名。函数用 for 循环语句，访问电话号码数组中的每一条数据。将每个电话号码与当前输入的电话号码做比较，如果相同则输出这条记录。程序中使用了一个计数器，如果没有输出任何记录则输出错误提示。

【范例 13-24】实例：按姓名查找函数，示例代码如下所示。

示例代码 13-24

```
void selectbyname()
{
    char na[20];                                    /*定义一个字符串。*/
    int n,j;                                        /*计数变量。*/
    n=0;
    printf("select a number by name:\n");           /*输入姓名。*/
    printf("please input a name :\n");
    scanf("%s",na);                                 /*输入。*/
    for(j=0;j<i;j++)                                /*循环访问结构体。*/
    {
        if(strcmp(num[j].name,na)==0)/*比较输入的变量与结构体中的姓名是否相同。*/
        {
            shownum(num[j]);                        /*相同则输出。*/
            n++;                                    /*计数加 1。*/
        }
    }
    if(n==0)
    {
        printf("no such a name");                   /*如果 n 为 0 就显示没有这一条记录。*/
    }
}
```

13.9.9　删除电话号码函数

这个函数的作用是，从电话号码数组中删除一条电话号码信息。实例的方法，是将用户输入的姓名与数组中的每一个姓名进行比较。如果这条记录与输入姓名相同，则将数组中这条记录后面的所有记录向前移动一个位置。这样便覆盖了需要删除的电话号码。同时，需要将电话号码总数的计数减 1。程序的代码如下所示。

【范例 13-25】实例：删除电话号码函数，示例代码如下所示。

示例代码 13-25

```
void delenum()
{
    char na[20];                            /*定义一个字符串。*/
    int j,n;                                /*定义计数变量。*/
    n=0;                                    /*计数变量赋初值为 0。*/
    printf("delete a num by name:\n");      /*提示。*/
    printf("please input a name :\n");      /*提示。*/
    scanf("%s",na);                         /*输入一个姓名。*/
    for(j=0;j<i;j++)                        /*循环访问结构体数组中的每一个姓名。*/
    {
        if(strcmp(num[j].name,na)==0)       /*比较输入的姓名与结构体数组中的每一个姓名。
*/
        {
            n++;                            /*相同则计数加 1。*/
```

```
15                  for(;j<i;j++)                    /*把后面的变量向前移动。*/
16              {
17                   num[j]=num[j+1];
18              }
19              i--;                                 /*总数减 1。*/
20                  break;                           /*结束循环。*/
21           }
22       }
23       if(n==0)                                    /*如果计数为 0 表示没有这个姓名。*/
24       {
25            printf("no such a name");
26       }
27   }
```

13.9.10 保存到文件函数

这个函数的作用，是将用户输入的所有电话号码，保存到一个文本文件中。函数没有参数和返回值。需要保存的内容保存的全局变量中，这个函数可以访问全局变量的内容。函数用“open”函数打开一个文件，如果没有文件，会自动创建这个文件。然后使用 for 循环，访问全局变量数组中的每一个结构体，将这些信息依次写入到文件中。函数的代码如下所示。

【范例 13-26】实例：将电话号码信息保存到文件函数，示例代码如下所示。

示例代码 13-26

```
01   void savetofile()
02   {
03      int j,fd;                                    /*定义变量。*/
04      char file[]="/root/tel.txt";                 /*定义文件名。*/
05      extern int errno;                            /*设置错误号。*/
06      fd=open(file,O_WRONLY|O_CREAT);              /*打开文件。*/
07      if(fd!=-1)                                   /*打开正常。*/
08      {
09        printf("opened file %s  .\n",file);        /*显示正常。*/
10      }
11      else                                         /*不能打开文件就显示错误。*/
12      {
13        printf("cant't open file %s.\n",file);
14         printf("errno: %d\n",errno);
15         printf("ERR  : %s\n",strerror(errno));
16      }
17      for(j=0;j<i;j++)                             /*循环访问结构体数组，写入变量。*/
18      {
19          printf(" %d %s\n",j,num[j].name);        /*显示姓名。*/
20        write(fd,num[j].name,7);                   /*保存姓名。*/
21        write(fd,num[j].tel,13);                   /*保存电话。*/
22      }
23      printf("saved.\n");                          /*输出信息。*/
24      close(fd);                                   /*关闭文件。*/
25   }
```

13.9.11 从文件导入信息函数

本函数的作用，是从以前保存的文件中读取电话号码和姓名信息，并将这些信息保存到全局变量的数组中。在导入信息时，依次读取若干个字符。如果读取的字符个数不为 0，则在结构体数组中添加一条记录。函数的代码如下所示。

【范例 13-27】实例：从文件导入电话号码信息函数，示例代码如下所示。

示例代码 13-27

```
void loadfromfile()
{
    int j=0,fd,t;                                   /*定义变量。*/
    i=0;
    char na[7];                                     /*定义姓名。*/
    char tel[13];                                   /*定义电话。*/
    char file[]="/root/tel.txt";                    /*定义文件名。*/
    extern int errno;                               /*设置错误号。*/
    fd=open(file,O_WRONLY|O_CREAT);                 /*打开文件。*/
    if(fd!=-1)                                      /*文件打开成功。*/
    {
      printf("opened file %s  .\n",file);
    }
    else                                            /*文件打开失败。*/
    {
      printf("cant't open file %s.\n",file);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
    }
    while((t=read(fd,na,7))!=0&&t!=-1)               /*读取文件。*/
    {
          strcpy(num[i].name,na);                   /*复制字符串到结构体中。*/
          read(fd,tel,13);  /*读取电话。*/
          strcpy(num[i].tel,tel);
                  /*复制字符串到结构体中。*/
          i++;      /*计数加 1。*/
     }
     close(fd);    /*关闭文件。*/
}
```

13.9.12　主函数

主函数的作用，是将上面编写的这些函数用一定的关系组织起来，统一完成一个程序功能。在程序开始时，进入一个 while 循环。在循环中反复调用 menu 函数来进行功能选择，然后根据返回的数值来调用相关的子函数。主函数的流程可用图 13-2 来表示。

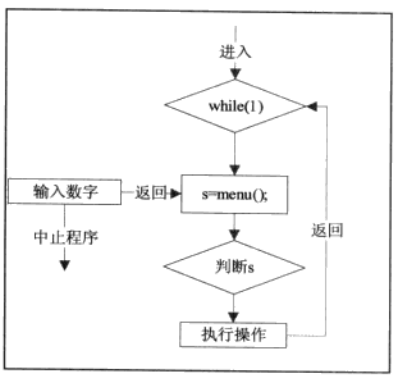


图 13-2　程序主函数的流程图

根据图中的程序运行流程，编写的主函数如下所示。

【范例 13-28】实例：主函数，示例代码如下所示。

示例代码 13-28

```
main()
{
   int s,j;                                          /*定义变量。*/
   printf("- - - - Telphone Notebook. - - - \n");     /*输出提示。*/

   while(1)                                          /*进入一个死循环。*/
   {
     s=menu();                                       /*显示菜单。*/
     if(s==1)                                /*分别判断 s 的值，调用不同的函数。*/
     {
```

```
            num[i]=addnum();                    /*添加一个电话号码。*/
            i++;                                /*总数加 1。*/
        }
        if(s==2)                                /*显示所有记录。*/
        {
            for(j=0;j<i;j++)                    /*循环。*/
            {
                shownum(num[j]);                /*显示所有记录，结构体作为参数。*/
                }
          }
          if(s==3)                              /*按姓名查找。*/
        {
            selectbyname();
          }
          if(s==4)                              /*删除一个记录。*/
        {
            delenum();
          }
        if(s==5)                                /*保存到文件。*/
        {
            savetofile();
          }
          if(s==6)                              /*从文件中导入。*/
        {
            loadfromfile();
          }
    }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 13.17.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序最开始显示一个选择菜单，要求用户选择一个功能。

```
- - - - Telphone Notebook. - - -
Please select a menu:
      1: add a number.
      2: all the number.
      3: select a number by name.
      4: delete a number .
      5: save to file .
      6: load numbers from file .
      0: exit .
```

用户输入“1”，然后按“Enter”键，提示用户输入姓名和电话。用同样的方法输入 3 个用户和电话。这时选择“2”，显示所有的电话，结果如下所示。

```
Name   :jim
   tel:65455676
Name   :Lily
   tel:65746367
Name   :Lucy
   tel:98347834
```

用户选择“4”，然后输入一个姓名，可以删除已经添加的姓名和电话。如果没有这个姓名，程序会显示没有这条记录。

用户选择“5”，程序会把输入的信息保存到文件“/root/tel.txt”中。这时输入下面的命令，查看保存的信息。

```
vim /root/tel.txt
```

vim 显示的文本信息如下所示。

```
jim65455676Lily65746367Lucy98347834
```

当程序再次启动时，输入“6”，从这个文件导入上次保存的信息。输入“0”可以退出这个程序。

13.10　小结

本章讲解了目录和文件的基本操作。其中需要掌握的知识包括目录的创建与删除、文件的创建与删除、文件的移动与复制、文件的锁定与权限、文件的缓冲区操作、文件的读写。其中文件的读写是本章的难点，需要做大量的编程练习。最后的综合实例讲解了文件在程序中的运用方法，难点是将程序处理的信息保存到文件。文件内容的具体操作将在下一节中讲到。

13.11　习题

一、填空题

1．所谓_____，指的是文件系统为了进行安全管理，需要在操作时进行用户身份认证。合法的用户可以进行操作。而没有权限的用户不能进行文件操作。

2．在进行文件操作时，可能会遇到权限不足、找不到文件等错误，这时需要在程序中设置错误捕捉语句并显示错误。错误捕捉和错误输出是应用_____和_____函数来实现的。

3．函数_____可以硬盘中建立一个目录，相当于 mkdir 命令。但与 mkdir 命令不同的是，这里的操作是用 C 语言的函数完成目录创建的。_____函数的作用是删除一个空目录。

4．所谓_____，是指在一个目录中建立一个空文件，可供其他程序进行写入操作。_____指将没用的文件从磁盘中删除。

5．_____指的是从磁盘中找到一个文件，返回一个整型的打开文件顺序编号。打开的文件处于可读可写状态。_____指的是释放打开的文件，使文件处于不可读写的状态。

二、选择题

1．下面关于文件与目录说法正确的是（　　）。

A．通常使用 open 函数打开文件，然后对文件进行读写操作，最后使用 close 函数关闭文件

B．所谓的文件锁定，指的是以共享的方式打开文件

C．使用 move 函数实现文件的移动操作

D．函数 delete 的作用是删除一个文件

2．下面关于文件锁定的说法不正确的是（　　）。

A．当多个应用程序同时访问同一个文件时，可能导致文件的内容不一致

B．例如一个文件中的数据是一个账户金额。用户打开这个文件读取数据，进行处理以后将写结果写入到文件。如果文件没有进行锁定时，可能发生错误

C．在访问一个文件时，可以用 flock 函数对文件进行锁定。防止其他用户同时访问这个文件发生数据不一致的情况

D．文件锁定函数 fcntl 与 flock 完全相同

三、简答题

1．简述 C 语言中文件读写操作的整个流程。

2．什么是文件的锁定，文件锁定有什么作用？简述文件锁定函数的用法。

四、编程题

1．编写程序实现文件复制功能（提示：可以分别打开原文件和目标文件。依次从原文件中读取一定长度的内容，然后写入到新文件中）。

2．设计一个电话号码管理程序。用户从终端中输入电话号码，电话号码的内容使用文件读写函数记录到文件中。

第 14 章　文件 I/O

文件 I/O 操作是程序设计时的一个重点，当程序完成算术或者逻辑运算后，通常都需要借助于文件 I/O 操作将运算结果保存到外围设备的文件中。本章详细介绍了与文件 I/O 操作有关的函数的应用，包括文件打开与关闭函数和文件的读写函数，具体内容如下：

- 文件的打开与关闭；
- 文件的读写操作。

14.1　文件的打开与关闭

打开文件是指在硬盘中找到这个文件，使这个文件处于被调用状态。进行文件读写之前需要进行文件打开操作。文件访问结束以后需要关闭文件。

14.1.1　文件打开函数 fopen

函数 fopen 的作用是打开一个文件，这个函数的使用方法如下所示。

```
01  FILE * fopen(char * path,char * mode);
```

在参数列表中，path 是需要打开的文件名字符串。mode 是文件打开形态的字符串，这个参数的可能内容如下所示。

- r：打开只读文件，该文件必须存在。
- r+：打开可读写的文件，该文件必须存在。
- w：打开只写文件，若文件存在则文件长度清为 0，即该文件内容全部删除。若文件不存在则建立该文件。
- w+：打开可读写文件，若文件存在则文件长度清为零，即该文件内容全部删除。若文件不存在则建立该文件。
- a：以追加的方式打开只写文件。若文件不存在，则会建立该文件。如果文件存在，写入的数据会被加到文件末尾，文件原来的内容保持不变。
- a+：以追加方式打开可读写的文件。若文件不存在，则会建立该文件。如果文件存在，写入的数据会被加到文件末尾。文件原来的内容保持不变。

上述的打开状态描述字符串都可以再加一个 b 字符，如 rb、w+b 或 ab+等组合。加入 b 字符用来告诉函数库打开的文件是二进制文件，而非纯文本文件。在使用这个文件时，需要包含下面的头文件。

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

如果文件被正常打开，会返回一个 FILE 类型的文件指针。打开失败则返回的内容为 NULL，可用 errno 来捕获所发生的错误。用 fopen 函数打开一个文件以后，一般会进行读写处理。如果文件打开发生错误，则后面的文件操作都会发生错误。所以需要对文件的打开状态进行判断并及时进行处理。下面是使用 fopen 函数打开文件的实例。

【范例 14-1】使用 fopen 函数打开一个文件，示例代码如下所示。

示例代码 14-1

```
#include <stdio.h>                                    /*包含头文件*/
#include <errno.h>

main()                                                /*主函数*/
{
   FILE * fp;
   extern int errno;
   char file[]="/root/a1.txt";

   fp=fopen(file,"r");
   if(fp==NULL)
   {
       printf("cant't open file %s.\n",file);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
       return;
     }
     else
     {
         printf("%s was opened.\n",file);
     }
   fclose(fp);

   fp=fopen("/root/a.sh","r");
   if(fp==NULL)
   {
       printf("cant't open file %s.\n",file);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
     }
     else
     {
         printf("file was opened.\n");
     }
   fclose(fp);

   fp=fopen("/root/a.sh","a+");
   if(fp==NULL)
   {
       printf("cant't open file %s.\n",file);
       printf("errno: %d\n",errno);
       printf("ERR  : %s\n",strerror(errno));
       return;
     }
     else
     {
         printf("file was created.\n");
     }
   fclose(fp);
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.1.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。第一次是打开一个已经存在的文件，可以正常打开。第二次是

以只读的方式打开一个不存在的文件，发生错误。第三次是用 a+的方式打开一个文件，没有这个文件时会自动新建这个文件。

```
/root/a1.txt  was opened.
cant't open file /root/a.sh.
errno: 2
ERR  : No such file or directory
/root/a.sh  was opened.
```

【代码解析】

- 代码第 10~22 行和第 24~35 行分别以只读方式打开文件，如果文件已存在则正常打开，否则将出错。在完成文件的操作后必须关闭文件。
- 代码第 37~49 行以 a+方式打开一个文件，如果文件不存在则自动创建此文件。

注意：在进行文件操作时，对于已经打开的文件，在完成操作后必须使用 fclose 函数关闭文件。

14.1.2　文件打开函数 fdopen

文件打开函数 fdopen 的作用，是将 open 函数打开文件返回的文件打开编号转换成文件指针返回。open 函数打开文件的方法如上一章所述，打开文件以后会返回一个整型的编号。fdopen 函数的使用方法如下所示。

```
01   FILE * fdopen(int fildes,const char *mode);
```

在参数列表中，fildes 是 open 函数打开文件以后返回的编号。mode 是一个字符串，用来表示文件打开的方式。字符串的内容如上一节中 fopen 函数中的 mode 参数。mode 的参数要与 open 函数打开文件的读写参数相同。如果文件打开成功，则返回这个文件的指针。否则返回一个空指针 NULL。函数可能返回下面的错误信息。可以用 errno 捕获文件打开时的错误。下面的实例使用这个函数打开一个文件。

【范例 14-2】使用 fdopen 函数打开一个文件，示例代码如下所示。

示例代码 14-2

```
01   #include <stdio.h>                                    /*包含头文件*/
02   #include <errno.h>
03   #include <unistd.h>
04   #include <sys/types.h>
05   #include <sys/stat.h>
06   #include <fcntl.h>
07
08   main()                                                /*主函数*/
09   {
10       int fd;                                           /*定义文件指针及文件操作必需的变量*/
11      char path[]="/root/txt1.txt";
12      FILE *fp;
13      extern int errno;
14
15     fd=open(path,O_WRONLY|O_CREAT,0766); /*打开文件*/
16     if(fd!=-1)
17     {
18       printf("opened file %s  .\n",path);
19     }
20     else
21     {
```

```
22        printf("cant't open file %s.\n",path);
23         printf("errno: %d\n",errno);
24         printf("ERR  : %s\n",strerror(errno));
25     }
26     fp =fdopen(fd,"r");              /*将 open 函数返回的文件编号转换成文件指针*/
27  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.2.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
opened file /root/txt1.txt
```

【代码解析】

- 代码第 15 行调用 open 函数打开文件。
- 代码第 26 行调用 fopen 函数将 open 函数的返回值转换为文件指针。

14.1.3 打开文件函数 freopen

函数 freopen 的作用，是将文件指针以前打开的文件关闭，然后重新打开一个文件。这个函数的使用方法如下所示。

```
01   FILE * freopen(char * path,char * mode,FILE * stream);
```

在参数列表中，path 是需要打开文件的字符串。stream 是原有的文件指针。mode 是文件的打开方式。调用这个函数时，会关闭 stream 指针所打开的文件，然后打开 path 所代表的文件指针，然后返回一个文件指针。如果文件打开成功，则返回打开文件的指针。否则返回 NULL。可以用 error 来捕获所发生的错误。下面的实例用 freopen 函数打开一个文件。

【范例 14-3】使用 freopen 函数重新打开一个文件，示例代码如下所示。

示例代码 14-3

```
01  #include <stdio.h>                                  /*包含头文件*/
02  #include <errno.h>
03
04  main()                                             /*主函数*/
05  {
06    FILE *fp;                                        /*定义文件指针及文件操作必需的变量*/
07    char path[]="/root/txt1.txt";
08    extern int errno;
09
10    fp=freopen("/root/txt1.txt","r",fp);             /*重新打开文件*/
11    if(fp==NULL)
12    {
13        printf("cant't open file %s.\n",path);
14        printf("errno: %d\n",errno);
15        printf("ERR  : %s\n",strerror(errno));
16        return;
17     }
18     else
19     {
20        printf("%s was opened.\n",path);
21      }
22  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.3.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
/root/txt1.txt was opened.
```

【代码解析】代码第 10~21 行首先调用 freopen 函数完成文件的打开操作，接下来使用 if 语句判断 freopen 函数的返回值是否为 NULL，如果返回值为 NULL 则表明文件打开失败，否则文件打开成功。

14.1.4　关闭文件函数 fclose

函数 fclose 的作用是关闭已经打开的文件指针。在打开和访问文件以后，需要及时地关闭打开的文件，以释放系统资源。这个函数的使用方法如下所示。

```
01  int fclose(FILE * stream);
```

参数 stream 是已经打开的文件指针。如果文件关闭成功，则返回整型的 0。如果失败，可用 errno 来捕获所发生的错误。下面的实例用这个函数关闭已经打开的文件。

【范例 14-4】使用 fclose 函数关闭已经打开的文件，示例代码如下所示。

示例代码 14-4

```
01  #include <stdio.h>                                  /*包含头文件*/
02  #include <errno.h>
03
04  main()                                              /*主函数*/
05  {
06    FILE * fp;                                        /*定义文件指针及文件操作必需的变量*/
07    extern int errno;
08    char file[]="/root/a1.txt";
09    fp=fopen(file,"r");                               /*以只读方式打开文件*/
10    if(fp==NULL)
11    {
12       printf("can't open file %s.\n",file);
13       printf("errno: %d\n",errno);
14       printf("ERR  : %s\n",strerror(errno));
15       return;
16     }
17     else
18     {
19         printf("%s was opened.\n",file);
20     }
21    if(fclose(fp)==0)                                 /*关闭文件*/
22    {
23       printf("file was closed.\n");
24     }
25     else
26     {
27       printf("can't close file .\n");
28       printf("errno: %d\n",errno);
29       printf("ERR  : %s\n",strerror(errno));
30     }
31  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.4.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表示打开文件以后又成功关闭文件。

```
/root/a1.txt was opened.
file was closed.
```

【代码解析】代码第 9 行完成文件的打开操作，代码第 21 行则调用 fclose 函数关闭已打开的文件。

提示：养成及时关闭不再使用的文件是一种良好的编程习惯，有利于提高程序的质量，节省系统资源。

14.2 文件的读写

所谓文件的读写，指的是向已经打开的文件中写入数据或者从文件中读取数据。本节将讲解文件读写操作，这些操作是通过文件读写函数实现的。

14.2.1 字符写入函数 putc 与 fputc

函数 putc 的作用是将一个字符写入到文本文件中。fputc 的作用与使用方法与 putc 的完全相同。这两个函数的使用方法如下所示。

```
01  int  putc(int c,FILE *stream);
02  int fputc(int c,FILE *stream);
```

在参数列表中，c 是需要写入的字符。stream 是已经打开的文件指针。如果写入成功，则返回这个字符。返回 EOF 表示写入失败。下面是这个函数的使用实例。

【范例 14-5】使用 putc 函数将字符写入文件，示例代码如下所示。

示例代码 14-5

```
01  #include <stdio.h>                              /*包含头文件*/
02  #include <errno.h>
03
04  main()                                          /*主函数*/
05  {
06     FILE * fp;                                   /*定义文件指针及文件操作必需的变量*/
07     extern int errno;
08     char file[]="/root/a1.txt";
09     char txt[5]="hello";
10     int i=0;
11     fp=fopen(file,"a+");                         /*打开文件*/
12     if(fp==NULL)
13     {
14        printf("cant't open file %s.\n",file);
15        printf("errno: %d\n",errno);
16        printf("ERR  : %s\n",strerror(errno));
17        return;
18      }
19      else
```

```
20        {
21            printf("%s was opened.\n",file);
22        }
23
24
25      for(i=0;i<5;i++)                          /*利用循环语句向文件中写入字符 */
26      {
27          putc(txt[i],fp);
28          printf("%c",txt[i]);
29        }
30        fclose(fp);                             /*关闭文件 */
31    }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.5.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示，表明已经向这个文件中写入字符。

```
/root/a1.txt was opened.
hello
```

输入下面的命令，查看文件/root/a1.txt 文件的内容。

```
vim /root/a1.txt
```

可以查看到文件/root/a1.txt 中有下面的文本内容。

```
hello
```

【代码解析】

- 代码第 11 行以 a+方式打开文件。
- 代码第 25~29 行则使用循环语句向已经打开的文件中写入 5 个字符。
- 代码第 30 行关闭由 fopen 函数打开的文件。

提示： 当需要向文件内写入数据时，打开文件时要以允许写入的模式打开，否则无法向文件中写入数据。

14.2.2　向文件中写入字符串函数 fputs

函数 fputs 的作用，是将一个字符串写入到文件中。这个函数的使用方法如下所示。

```
01   int fputs(char * s,FILE * stream);
```

在参数列表中，s 表示需要输入的字符串。stream 表示已经打开的文件指针。如果写入成功，则会返回实际写入字符的个数。写入不成功时会返回 EOF。下面是这个函数的使用实例。

【范例 14-6】使用 puts 函数将字符串写入文件，示例代码如下所示。

示例代码 14-6

```
01   #include <stdio.h>                           /*包含头文件*/
02   #include <errno.h>
03
04   main()                                       /*主函数*/
05   {
06      FILE * fp;                                /*定义文件指针及文件操作必需的变量*/
07      extern int errno;
08      char file[]="/root/a1.txt";
```

```
09      char *txt="hello";
10      int i;
11      fp=fopen(file,"a+");                              /*打开文件*/
12      if(fp==NULL)
13      {
14          printf("cant't open file %s.\n",file);
15          printf("errno: %d\n",errno);
16          printf("ERR  : %s\n",strerror(errno));
17          return;
18        }
19        else
20        {
21            printf("%s was opened.\n",file);
22        }
23
24        i=fputs(txt,fp);
25        {
26            printf("%d char was written.\n",i);
27        }
28        fclose(fp);                                     /*主函数*/
29    }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.6.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。程序打开了文件以后，向文件/root/a1.txt 中写入字符串。

```
/root/a1.txt was opened.
```

输入下面的命令，查看文件/root/a1.txt 文件的内容。

```
vim /root/a1.txt
```

可以查看到文件/root/a1.txt 中有下面的文本内容。

```
hello
```

【代码解析】

- 代码第 11 行以 a+方式打开文件。
- 代码第 24 行则使用 fputs 函数向已经打开的文件中写入字符串。
- 代码第 28 行关闭由 fopen 函数打开的文件。

14.2.3 数据写入函数 fwrite

函数 fwrite 的作用是将一条数据写入到已经打开的文件。写入的数据，可以是整型、字符、字符串、结构体等内容。这些数据需要存储在一段连续的内存上。这个函数的使用方法如下所示。

```
01   size_t fwrite(void * ptr,size_t size,size_t nmemb,FILE * stream);
```

在参数列表中，ptr 是需要写入数据的地址。size 表示每一条数据的字节数，nmemb 表示需要写入多少条数据。stream 表示已经打开的文件指针。如果写入成功，则返回实际写入的数据的条数。否则返回 EOF。下面是这个函数的使用实例。在程序中定义一个结构体数组，将已经赋值的结构体数组保存到已经打开的文件中。关于 fwirte 函数的代码示例请参看本章的综合练习。

14.3 综合练习

本节利用 fwrite 函数实现将 stu 类型的结构体数组保存到文件中。代码中首先定义结构体

stu，然后为结构体的成员赋值，最后调用 fwrite 函数将结构体数组保存到文本文件中。

【范例 14-7】使用 fwrite 函数将结构体数组写入文件中，示例代码如下所示。

示例代码 14-7

```
#include <stdio.h>                          /*包含头文件*/
#include <stdlib.h>
#include <errno.h>
struct stu                                  /*定义 stu 结构体类型*/

{
    char name[10];
    int age;
};

main()                                      /*主函数*/
{
  struct stu mystu[3];                      /*定义结构体数组*/
  FILE * fp;                                /*定义文件指针及文件操作必需的变量*/
  extern int errno;
  char file[]="/root/a1.txt";
  int i;

  strcpy(mystu[0].name,"Jim");              /*为结构体成员赋值*/
  mystu[0].age=14;
  strcpy(mystu[0].name,"Jam");
  mystu[1].age=14;
  strcpy(mystu[0].name,"Lily");
  mystu[2].age=19;

  fp=fopen(file,"a+");                      /*打开文件*/
  if(fp==NULL)
  {
     printf("cant't open file %s.\n",file);
     printf("errno: %d\n",errno);
     printf("ERR  : %s\n",strerror(errno));
     return;
    }
    else
    {
        printf("%s was opened.\n",file);
    }

   i=fwrite(mystu,sizeof(mystu),3,fp);      /*将数据写入文件*/
   printf("%d bit was written.\n",i);
   close(fp);                               /*关闭文件*/
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 14.7.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。表明这个程序向文件中写入了 48 个字节。

```
48 bit was written.
```

输入下面的命令，查看文件/root/a1.txt 文件的内容。

```
vim /root/a1.txt
```

14.4 小结

本章讲述了文件 I/O 操作，重点内容是文件打开关闭和读写操作。在使用文件打开函数时，需要注意打开方式和权限的设置。使用 putc、fputc、puts、fwite 等文件读写函数时，需要注意这些函数作用和不同参数的含义。在编程时，可以把程序的结果记录到文件中，实现程序数据的长期保存。

14.5 习题

一、填空题

1．_____是指在硬盘中找到这个文件，使这个文件处于被调用状态。进行文件读写之前需要进行此操作。文件访问结束以后需要关闭文件。

2．所谓_____，指的是向已经打开的文件中写入数据，或者从文件中读取数据。

3．文件打开函数_____的作用，是将 open 函数打开文件返回的文件打开编号转换成文件指针返回。open 函数打开文件的方法如上一章所述，打开文件以后会返回一个整型的编号。

4．函数_____的作用是将一条数据写入到已经打开的文件。写入的数据，可以是整型、字符、字符串、结构体等内容。这些数据需要存储在一段连续的内存上。

二、选择题

1．下面关于文件打开与关闭函数的说法中错误的是（　　）。

A．用 fopen 函数打开一个文件以后，一般会进行读写处理。如果文件打开发生错误，则后面的文件操作都会发生错误。所以需要对文件的打开状态进行判断并及时进行处理

B．文件打开函数 fdopen 的作用，是将 open 函数打开文件返回的文件打开编号转换成文件指针返回

C．函数 freopen 的作用，是将文件指针以前打开的文件关闭，然后重新打开一个文件

D．函数 fclose 的作用是关闭已经打开的文件指针。在打开和访问文件以后，不关闭打开的文件，对系统没有任何影响

2．下面关于文件的读写说法正确的是（　　）。

A．函数 putc 的作用是，将一个字符写入到文本文件中。fputc 的作用与使用方法与 putc 的作用完全相同

B．函数 fputs 的作用是将一个字符写入到文件中

C．函数 fwrite 的作用是将一条数据写入到已经打开的文件。写入的数据，可以是整型、字符、字符串、结构体等内容。这些数据不需要存储在一段连续的内存上

三、简答题

1．简述文件的打开与关闭常用的函数及其功能和用法。

2．简述文件的读写常用的函数及其功能和用法。

四、编程题

1．分别使用函数 fopen、fdopen 和 freopen 打开指定的磁盘文件，然后使用 fclose 函数关闭已打开的文件。

2．编程实现将学生基本信息写入文件（提示：学生基本信息保存在一个结构体数组中，将已经赋值的结构体数组保存到已经打开的文件中）。

第六篇　网络编程篇

第 15 章　网 络 编 程

当前计算机的工作模式已经从传统的单机模式过渡到了网络模式，基于网络的软件开发也越来越普及。本章详细介绍了网络编程的基本概念，以及在进行网络应用程序开发中所必备的基础知识和 C 语言中涉及网络编程的基本函数的用法。本章的主要内容如下：

- 网络编程的基本概念；
- 套接字；
- 域名与 IP 地址；
- 网络协议与网络服务；
- 网络地址的转换；
- 网络编程中的错误处理。

15.1　网络编程的基本概念

不同的程序进行网络通信时，是通过 IP 地址和套接字来访问一个主机的。在学习网络编程之前，需要理解一些关于网络的概念和术语。

15.1.1　IP 地址

IP 地址的作用是标识网络中的一台计算机，每一台计算机都有一个 IP 地址。在程序中通过 IP 地址来访问一台计算机。本节将讲述 IP 地址的一些知识。IP 地址是用来标识全球计算机地址的一种符号，相当于一个手机的号码。使用这个地址可以访问一台计算机。作为计算机的统一标识，IP 地址有以下特点。IP 地址具有统一的格式。IP 地址是 32 位长度的二进制数值，存储空间是 4 个字节。这 4 个字节的二进制字符值可以表示一台计算机。

IP 地址可以使用点分十进制数来表示。二进制的数值是不便于记忆的，可以把每个字节用一个整数来表示。例如 11000000 10101000 00000001 00000110 是一台计算机的 IP 地址。转换成点分十进制数后是 192.168.1.1。在同一个网络中，IP 地址是唯一的。因为需要根据 IP 地址来访问一台计算机，所以在可以访问的范围以内，每一台计算机的 IP 地址是唯一的。

在终端中输入下面的命令，可以查看自己的计算机的 IP 信息。

```
ifconfig
```

终端中显示的 IP 信息与网卡信息如下所示。

```
eth0      Link encap:Ethernet  HWaddr 00:0F:EA:45:4E:51
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::20f:eaff:fe45:4e51/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6332 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4484 errors:0 dropped:0 overruns:0 carrier:0
          collisions:56 txqueuelen:1000
          RX bytes:6710445 (6.3 MiB)  TX bytes:531858 (519.3 KiB)
          Interrupt:18
```

提示：IP 地址又称为逻辑地址，可以通过软件设置改变主机的 IP 地址；MAC 地址又称为物理地址，固化在网卡上，通常不能改变网卡的 MAC 地址。

15.1.2 端口

所谓端口，是计算机中为了标识计算机中访问网络的不同程序的编号。每一个程序在访问网络时，都会分配一个标识符。程序在访问网络或接受访问时，会用这个标识符表示这一网络数据属于这个程序。这里的端口并非网卡接线的端口，而是对不同程序的逻辑编号，并不是实际存在的。

端口号是一个 16 位的无符号整数，对应的十进制数取值范围是 0~65535。不同编号范围的端口有不同的作用。低于 256 的端口是系统保留端口号，主要用于系统进程通信。例如网站的 WWW 服务使用的是 80 号端口，FTP 服务使用的是 21 号端口。不在这一范围的端口号是自由端口号，在编程时可以调用这些端口号。

提示：端口号（Port）用于区分网络通信中主机上不同的网络应用程序。

15.1.3 域名

域名是用来代替 IP 地址以标识计算机的一种直观名称。例如百度网站的 IP 地址是 202.108.22.43，这个 IP 地址没有任何逻辑含义，不便于记忆。而 www.baidu.com 是一个便于记忆的名称，用于代替这个 IP 地址。在访问计算机时，可以用这个域名来代替 IP 地址。在使用 C 语言编程时，有时需要用域名来访问一个计算机。这些要将域名转换成相应的 IP 地址。在终端中，可以用 ping 命令来查看域名所对应的 IP 地址。例如可以输入下面的命令来查看百度的 IP 地址。

```
ping www.baidu.com
```

终端中显示的结果如下所示。表明百度的 IP 地址是 202.108.22.43。

```
PING www.a.shifen.com (202.108.22.43) 56(84) bytes of data.
64 bytes from xd-22-43-a8.bta.net.cn (202.108.22.43): icmp_seq=2 ttl=58 time=33.9 ms
```

15.1.4 TCP 与 UDP

TCP 与 UDP 是两种不同的网络传输方式。两个不同计算机中的程序，使用 IP 地址和端口，要使用一种约定的方法进行数据传输。TCP 与 UDP 就是网络中的两种数据传输约定，主要的区别是进行数据传输时是否进行连接。

- TCP：TCP 是一种面向连接的网络传输方式。这种方式可以理解为打电话。计算机 A 先呼叫计算机 B。计算机 B 接受连接后发出确认信息。计算机 A 收到确认信息以后发送信息。计算机 B 完成数据接收以后发送完毕信息。这时再关闭数据连接。所以 TCP 是面向连接的可靠的信息传输方式。这种方式是可靠的，缺点是传输过程复杂，需要占用较多的网络资源。
- UDP：UDP 是一种面向无连接的传输方式。可以简单理解成邮寄信件。将信件封装放入邮筒以后，不再参与邮件的传送过程。使用 UDP 传送信息时，不建立连接，直接把信息发送到网络上，由网络完成信息的传送。信息传递完成以后也不发送确认信息。这种传输方式是不可靠的，但是有很好的传输效率。对传输可靠性要求不高时，可以选择使用这种传输方式。

注意：如果使用 UDP 协议进行数据通信，数据传输的可靠性由软件开发人员编程实现。

15.2　套接字

套接字（Socket）的本义是插座，在网络中用来描述计算机中不同程序与其他计算机程序的通信方式。本节将讲述套接字的含义与数据类型。

15.2.1　什么是套接字

程序访问网络进行数据通信时，TCP 和 UDP 会遇到同时为多个应用程序同时并发进行通信的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序的进程和连接，需要使用应用程序与 TCP / IP 协议交互的套接字（Socket）的接口。

区分不同应用程序进程间的网络通信和连接，主要使用三个参数。通信的目的 IP 地址、使用的传输层协议（TCP 或 UDP）和使用的端口号。在编程时，就是使用为三个参数来构成一个套接字。这个套接字相当于一个接口，可以进行不同计算机程序的信息传输。

提示：套接字=传输层协议+端口号+IP 地址。

15.2.2　套接字相关的数据类型

C 程序进行套接字编程时，常会使用到 sockaddr 数据类型或 sockaddr_in 数据类型。这两种数据类型是系统中定义的结构体，用于保存套接字的信息。sockaddr 用来保存一个套接字，定义方式如下所示。

```
01  struct sockaddr
02  {
03     unsigned short int sa_family;
04     char sa_data[14];
05  };
```

在这个结构体中，成员的含义如下所示。

- sa_family：指定通信的地址类型。如果是 TCP/IP 通信，则该值为 AF_INET。
- sa_data：最多使用 14 个字符长度，用来保存 IP 地址和端口信息。

sockaddr_in 的功能与 soockaddr 相同，也是用来保存一个套接字的信息。不同的是将 IP 地址与端口分开为不同的成员。这个结构体的定义方法如下所示。

```
01  struct socketaddr_in
02  {
03     unsigned short int sin_family;
04     uint16_t sin_port;
05     struct in_addr sin_addr;
06     unsigned char sin_zero[8];
07  };
```

这个结构体的成员与作用如下所示。

- sin_family：与 sockaddr 结构体中的 sa_family 相同。
- sin_port：套接字使用的端口号。

- sin_addr：需要访问的 IP 地址。
- sin_zero：未使用的字段，填充为 0。

在这一结构体中，in_addr 也是一个结构体，定义方法如下所示。作用是保存一个 IP 地址。

```
01  struct in_addr
02  {
03    uint32_t s_addr;
04  };
```

提示：这两个套接字结构体的保存内容是相同的。可以用数据类型转换的方法来转换这两个类型的变量。

15.2.3 套接字类型

套接字类型指的是在网络通信中不同的数据传输方式。例如 UDP 和 TCP 就是两种不同的套接字类型。常用的套接字类型有下面三种。

- 流套接字（SOCK_STREAM）：流套接字使用了面向连接的可靠的数据通信方式，即 TCP（The Transmission Control Protocol）协议。这种服务可以实现无差错、无重复发送、并按顺序接收。
- 数据报套接字（Raw Sockets）：数据报套接字使用了面向无连接的数据传输方式，即 UDP（User Datagram Protocol）协议。这种协议在数据发送出去以后，即完成通信的任务。然后，完全依靠网络来完成数据传输。网络不能完全保证数据正确传输，也不能保证数据按顺序接收。这种套接字不能保证数据传输的可靠性，可能在通信中出现数据丢失的情况，需要在程序中做出相应的处理。
- 原始套接字（SOCK_RAW）：前面讲述的两种套接字是系统定义的，所有的信息都需要按照这种方式进行封装。原始套接字是没有经过处理的 IP 数据包，可以根据自己程序的要求进行封装。如果要访问其他的协议，需要使用原始套接字来构造相应的协议。

15.3 域名与 IP 地址

我们知道在网络编程时，域名是不能直接访问一个主机的，需要转换成相应的 IP 地址。有时在程序中需要将一个 IP 地址转换成一个域名。本节将讲解 C 程序中的 IP 地址与域名的转换问题。

提示：在 TCP/IP 网络中，通信双方的主机必须知道彼此的 IP 地址方可进行正常的通信，如果给出主机的域名，在开始正常的通信前必须把域名转换为 IP 地址。这个域名到 IP 地址的转换过程称为域名解析。

15.3.1 用域名取得主机的 IP 地址

域名是为了便于记忆，来代替 IP 地址访问网络的方法。在使用域名访问网络时，需要将这个域名转换成相对应的 IP 地址。用域名返回地址的函数是 gethostbyname。这个函数的使用方法如下所示。

```
01  struct hostent *gethostbyname(const char *name);
```

在参数列表中，name 是一个表示域名的字符串。函数会把这个域名转换成一个主机地址

结构体返回。结构体 hostent 的定义方法如下所示。

```
struct hostent
{
   char  *h_name;
   char **h_aliases;
   int    h_addrtype;
   int    h_length;
   char **h_addr_list;
}
```

这个结构体成员含义如下所示。

- h_name：正式的主机名称。
- h_aliases：这个主机的别名。
- h_addrtype：主机名的类型。
- h_length：地址的长度。
- addr_list：从域名服务器取得的主机的地址。

在解析域名时，可能没有这个域名或域名服务器发生错误。可能返回的错误信息如下所示。可以用 error 来捕获错误编号。

- HOST_NOT_FOUND：主机没有找到。
- NO_ADDRESS or NO_DATA：没有 IP 地址或没有数据。
- NO_RECOVERY：域名服务器发生错误。
- TRY_AGAIN：请稍候再重试。

下面的实例是解析域名 www.163.com 的 IP 地址，并且显示出相关信息。

【范例 15-1】使用 gethostbyname 函数取得一个域名的 IP 地址，示例代码如下所示。

示例代码 15-1

```
#include <stdio.h>
#include <sys/socket.h>
#include <netdb.h>                                          /*包含相关的头文件。*/

int  main(int argc,char *argv[])
{
   struct hostent *host;                                    /*定义 hostent 结构体指针。*/
   char hostname[]="www.163.com";                           /*域名。*/
   char hostname2[]="www.1d5r6f.com";                       /*一个不存在的域名。*/
   struct in_addr in;                                       /*结构体。*/
   struct sockaddr_in addr_in;
   extern int h_errno;                                      /*错误号。*/

   if((host=gethostbyname(hostname))!=NULL)                 /*取得主机地址。*/
   {
      memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4);      /*复制地址。*/
      in.s_addr=addr_in.sin_addr.s_addr;
      printf("Domain name: %s \n",hostname);                /*输出主机名。*/
      printf("IP length:   %d\n",host->h_length);           /*地址的长度。*/
      printf("Type:    %d\n",host->h_addrtype);             /*地址的种类。*/
      printf("IP        : %s \n",inet_ntoa(in));           /*IP 地址。*/
   }
   else                                                     /*出错的处理。*/
   {
       printf("Domain name: %s \n",hostname);
       printf("error: %d\n",h_errno);
        printf("%s\n",hstrerror(h_errno));
```

```
28      }
29
30      if((host=gethostbyname(hostname2))!=NULL)      /*解析一个不存在的地址。*/
31      {
32         memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4);
33         in.s_addr=addr_in.sin_addr.s_addr;
34         printf("Domain name: %s \n",hostname2);
35         printf("IP         : %s \n",inet_ntoa(in));
36         printf("IP length:   %d\n",host->h_length);
37         printf("Type:    %d\n",host->h_addrtype);
38       }
39       else                                          /*输出错误信息。*/
40       {
41           printf("Domain name: %s \n",hostname2);
42           printf("error: %d\n",h_errno);
43            printf("%s\n",hstrerror(h_errno));
44      }
45  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.1.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。程序解析了第一个域名的 IP，显示了 IP 地址和地址长度。第 2 个域名不存在，显示出错误信息。

```
Domain name: www.163.com
IP length:    4
Type:    2
IP         : 202.108.9.31
Domain name: www.1d5r6f.com
error: 1
Unknown host
```

【代码解析】代码第 14 行和第 30 行调用 gethostbyname 函数进行域名解析，前者给出的是正确的域名，可以输出域名的相关信息，后者给出的则是不存在的域名，将输出错误信息。

15.3.2 用 IP 地址返回域名

用一个 IP 地址可以查询到这个 IP 的域名，需要使用的函数是 gethostbyaddr。这个函数的使用方法如下所示。

```
01   struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
```

在参数列表中，addr 是一个保存了 IP 地址的字符串。len 是这个 IP 地址的长度。type 的值一般为 AF_INET。函数的返回值是 hostent 类型的指针。这一指针的定义和上一节的相同。如果转换失败，则返回 null 指针。下面的实例是用 gethostbyaddr 函数查找一个 IP 所对应的域名。其中使用的 IP 地址是央视国际的网站。

提示：把 IP 地址转换为域名的过程称为反向域名解析。

【范例 15-2】使用 gethostbyaddr 函数取得 IP 地址对应的域名，示例代码如下所示。

示例代码 15-2

```
01   #include <stdio.h>
02   #include <sys/socket.h>
```

```
03  #include <netdb.h>                                          /*包含相关的头文件。*/
04
05  int  main(int argc,char *argv[])
06  {
07     struct hostent *host;                                    /*主机名结构体。*/
08     char addr[]="202.108.249.216";                           /*定义 IP 地址。*/
09     struct in_addr in;                                       /*地址结构体。*/
10     struct sockaddr_in addr_in;
11     extern int h_errno;
12
13     if((host=gethostbyaddr(addr,sizeof(addr),AF_INET))!=(struct hostent *)NULL)
14     {
15        memcpy(&addr_in.sin_addr.s_addr,host->h_addr,4);     /*输出主机名的信息。*/
16        in.s_addr=addr_in.sin_addr.s_addr;
17        printf("Domain name: %s \n",host->h_name);
18        printf("IP length:   %d\n",host->h_length);
19        printf("Type:    %d\n",host->h_addrtype);
20        printf("IP         : %s \n",inet_ntoa(in));
21     }
22     else                                                     /*出错的处理。*/
23     {
24         printf("error: %d\n",h_errno);
25          printf("%s\n",hstrerror(h_errno));
26     }
27  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.2.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
Domain name: www.cctv.com
IP length:   4
Type:   2
IP         : 202.108.249.216
```

【代码解析】代码第 13~26 行通过调用 gethostbyaddr 函数完成 IP 地址到域名的转换，如果转换过程无误，则打印出给定 IP 地址的域名信息，否则打印错误信息。

15.4　网络协议

所谓网络协议，是指不同的计算机、不同的操作系统在进行网络通信时的统一约定。在进行网络编程时，需要遵守这些约定。本节将讲述在编程过程中如何取得协议信息。

15.4.1　由协议名取得协议数据

每一种协议，如 TCP、UDP 等，是协议的名称。在编程时，需要用这些协议名称取得这些协议的数据。函数 getprotobyname 的作用是按名称取得一个协议的数据。这个函数的使用方法如下所示。

```
01  struct protoent *getprotobyname(char *name);
```

参数 name 是一个协议名称字符串。返回值是一个 protoent 的结构体指针。结构体 protoent 的定义方法如下所示。

```
01  struct protoent
```

```
02  {
03     char  *p_name;
04     char **p_aliases;
05     int    p_proto;
06  }
```

这个结构体的成员含义如下所示。

- p_name：协议的名称。
- p_aliases：协议的别名。
- p_proto：协议的序号。

使用这个函数以前，需要在程序中包含下面的头文件。

```
01  #include <netdb.h>
```

下面的实例使用这个函数，取得几种常用协议的数据。

【范例 15-3】使用 getprotobyname 函数取得指定协议的信息，示例代码如下所示。

示例代码 15-3

```
01  #include <stdio.h>
02  #include <netdb.h>                                    /*包含相关的头文件。*/
03
04  int  main()
05  {
06     struct protoent *pro;                              /*定义一个表示协议的结构体。*/
07     pro=getprotobyname("icmp");                            /*取得 ICMP 协议的信息。*/
08     printf("protocol name : %s\n",pro->p_name);        /*名称。*/
09     printf("protocol number : %d\n",pro->p_proto);     /*协议号。*/
10     printf("protocol alias: %s\n",pro->p_aliases[0]);/*别名。*/
11     pro=getprotobyname("udp");                             /*取得 UDP 协议的信息。*/
12     printf("protocol name : %s\n",pro->p_name);
13     printf("protocol number : %d\n",pro->p_proto);
14     printf("protocol alias: %s\n",pro->p_aliases[0]);
15     pro=getprotobyname("tcp");                             /*取得 TCP 协议的信息。*/
16     printf("protocol name : %s\n",pro->p_name);
17     printf("protocol number : %d\n",pro->p_proto);
18     printf("protocol alias: %s\n",pro->p_aliases[0]);
19  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.3.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : icmp
protocol number : 1
protocol alias: ICMP
protocol name : udp
protocol number : 17
protocol alias: UDP
protocol name : tcp
protocol number : 6
protocol alias: TCP
```

【代码解析】

- 代码第 7 行调用 getprotobyname 函数获取 ICMP 协议的相关信息。
- 代码第 11 行调用 getprotobyname 函数获取 UDP 协议的相关信息。

- 代码第 15 行调用 getprotobyname 函数获取 TCP 协议的相关信息。

15.4.2　由协议编号取得协议信息

知道了一个协议的编号，可以用 getprotobynumber 函数来取得这个协议的信息。这个函数的使用方法如下所示。

```
01   struct protoent *getprotobynumber(int proto);
```

在参数列表中，proto 是一个协议的编号。函数会返回这个协议的结构体指针 protoent。结构体 protoent 的定义方法与上一节相同。在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01   #include <netdb.h>
```

下面的实例是使用 getprotobynumber 函数取得 0~4 号协议的信息。

【范例 15-4】使用 getprotobynumber 函数取得指定编号的协议信息，示例代码如下所示。

示例代码 15-4

```
01   #include <stdio.h>
02   #include <netdb.h>                                  /*包含相关的头文件。*/
03
04   int  main()
05   {
06      struct protoent *pro;                           /*定义一个表示协议的结构体指针。*/
07      int i;
08
09      for(i=0;i<5;i++)                                /*循环。*/
10      {
11      pro=getprotobynumber(i);                        /*取得协议信息。*/
12      printf("protocol name : %s\n",pro->p_name);             /*协议名。*/
13      printf("protocol number : %d\n",pro->p_proto);          /*协议号。*/
14      printf("protocol alias: %s\n",pro->p_aliases[0]);       /*别名。*/
15      }
16   }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.4.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : ip
protocol number : 0
protocol alias: IP
protocol name : icmp
protocol number : 1
protocol alias: ICMP
protocol name : igmp
protocol number : 2
protocol alias: IGMP
protocol name : ggp
protocol number : 3
protocol alias: GGP
protocol name : ipencap
protocol number : 4
protocol alias: IP-ENCAP
```

【代码解析】代码第 9~15 行使用 for 循环依次调用 getprotobynumber 函数获得对应编号的网络协议信息，并输出协议信息。

15.4.3 取得系统支持的所有协议

函数 getprotoent 的作用是取得系统中所有支持的协议。这个函数的使用方法如下所示。

```
01  struct protoent *getprotoent(void);
```

这个函数没有参数，每次调用时，依次返回一个系统支持的协议。返回的类型是 protoent 结构体指针。protoent 结构体指针如前面章节所述。到达末尾时会返回 NULL 指针。系统中支持协议的类型，是记录在/etc/protocols 文件中的。可以输入下面的命令查看这个文件中支持协议的内容。

```
vim /etc/protocols
```

下面的实例是使用 getprotoent 函数来查看系统支持协议的类型。在程序中，把返回的值是否为 NULL 作为循环的条件，然后输出这个协议的信息。

【范例 15-5】使用 getprotoent 函数取得系统支持的所有协议信息，示例代码如下所示。

示例代码 15-5

```
01  #include <stdio.h>
02  #include <netdb.h>                                  /*包含头文件。*/
03
04  int  main()
05  {
06     struct protoent *pro;                            /*定义一个表示协议的结构体。*/
07     while(pro=getprotoent())                         /*循环取得一个协议。*/
08     {printf("protocol name : %s  ",pro->p_name);
09     printf("protocol number : %d  ",pro->p_proto);
10     printf("protocol alias: %s\n",pro->p_aliases[0]);
11     }
12  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.5.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
protocol name : ip  protocol number : 0  protocol alias: IP
protocol name : hopopt  protocol number : 0  protocol alias: HOPOPT
protocol name : icmp  protocol number : 1  protocol alias: ICMP
protocol name : igmp  protocol number : 2  protocol alias: IGMP
protocol name : ggp  protocol number : 3  protocol alias: GGP
protocol name : ipencap  protocol number : 4  protocol alias: IP-ENCAP
protocol name : st  protocol number : 5  protocol alias: ST
protocol name : tcp  protocol number : 6  protocol alias: TCP
protocol name : cbt  protocol number : 7  protocol alias: CBT
protocol name : egp  protocol number : 8  protocol alias: EGP
protocol name : igp  protocol number : 9  protocol alias: IGP
protocol name : bbn-rcc  protocol number : 10  protocol alias: BBN-RCC-MON
protocol name : nvp  protocol number : 11  protocol alias: NVP-II
protocol name : pup  protocol number : 12  protocol alias: PUP
protocol name : argus  protocol number : 13  protocol alias: ARGUS
protocol name : emcon  protocol number : 14  protocol alias: EMCON
protocol name : xnet  protocol number : 15  protocol alias: XNET
```

```
protocol name : chaos  protocol number : 16  protocol alias: CHAOS
protocol name : udp  protocol number : 17  protocol alias: UDP
protocol name : mux  protocol number : 18  protocol alias: MUX
protocol name : dcn  protocol number : 19  protocol alias: DCN-MEAS
protocol name : hmp  protocol number : 20  protocol alias: HMP
```

【代码解析】

- 代码第 06 行定义用于保存协议信息的 protoent 结构体指针 pro。
- 代码第 07~11 行使用 while 循环语句依次调用 getprotoent 函数取得协议信息，并打印取得的协议信息。

15.5　网络服务

所谓网络服务，指的是网络上的计算机通过运行程序，为其他的计算机提供信息或运算的功能。例如打开一个网页，是访问了网站服务器上的服务，从服务器下载了网页文件。本节将讲解 C 程序中的网络服务类型。

15.5.1　取得系统支持的网络服务

函数 getservent 的作用是取得系统所支持的服务。这个函数的使用方法如下所示。

```
01  struct servent *getservent(void);
```

这个函数没有参数，返回值是一个 servent 类型的结构体指针。结构体 servent 的定义方法如下所示。

```
01  struct servent
02  {
03    char  *s_name;
04    char **s_aliases;
05    int    s_port;
06    char  *s_proto;
07  }
```

这个结构体的成员，含义如下所示。

- s_name：这个服务的名称。
- s_aliases：这个服务可能的别名。
- s_port：这个服务可能的端口。
- s_proto：这个服务使用的协议。

每次调用这个服务时，会返回一个系统支持的服务。如果已经到达最后一个结果，则会返回 NULL。可以输入下面的命令，查看系统文件列表中支持的服务。

```
vim /etc/services
```

下面的实例是使用 getservent 函数来返回系统中所有的服务。需要注意的是，输出的 IP 地址需要用 ntohs 函数进行转换。

【范例 15-6】使用 getservent 函数取得系统支持的网络服务，示例代码如下所示。

示例代码 15-6

```
01  #include <stdio.h>
02  #include <netdb.h>                          /*包含相关的头文件。*/
03
04  int  main()
05  {
```

```
06      struct servent *ser;                          /*定义一个表示服务的结构体指针。*/
07      while(ser=getservent())                       /*取得一个服务。*/
08      {
09        printf("name : %s  ",ser->s_name);                    /*服务名。*/
10        printf("port : %d  ",ntohs(ser->s_port));             /*服务端口。*/
11        printf("protocol:%s  ",ser->s_proto);                 /*协议。*/
12        printf("alias: %s\n",ser->s_aliases[0]);              /*别名。*/
13      }
14  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.6.c
```

输入下面的命令，运行这个程序。

```
./out
```

下面是程序运行结果的一部分。while 循环多次调用 getservent 函数返回了系统支持的服务。

```
name : tcpmux  port : 1  protocol:tcp  alias: (null)
name : tcpmux  port : 1  protocol:udp  alias: (null)
name : rje  port : 5  protocol:tcp  alias: (null)
name : rje  port : 5  protocol:udp  alias: (null)
name : echo  port : 7  protocol:tcp  alias: (null)
name : echo  port : 7  protocol:udp  alias: (null)
name : discard  port : 9  protocol:tcp  alias: sink
name : discard  port : 9  protocol:udp  alias: sink
name : systat  port : 11  protocol:tcp  alias: users
name : systat  port : 11  protocol:udp  alias: users
name : daytime  port : 13  protocol:tcp  alias: (null)
name : daytime  port : 13  protocol:udp  alias: (null)
name : qotd  port : 17  protocol:tcp  alias: quote
name : qotd  port : 17  protocol:udp  alias: quote
name : msp  port : 18  protocol:tcp  alias: (null)
name : msp  port : 18  protocol:udp  alias: (null)
name : chargen  port : 19  protocol:tcp  alias: ttytst
name : chargen  port : 19  protocol:udp  alias: ttytst
name : ftp-data  port : 20  protocol:tcp  alias: (null)
name : ftp-data  port : 20  protocol:udp  alias: (null)
name : ftp  port : 21  protocol:tcp  alias: (null)
```

【代码解析】

- 代码第 06 行定义用于保存系统支持的服务信息的 servent 结构体指针 ser。
- 代码第 07~13 行使用 while 循环语句依次调用 getservent 函数取得系统支持的服务信息，并打印取得的信息。

15.5.2 用名称取得系统所支持的服务

函数 getservbyname 可以用一个服务的名称来取得一个服务。这个函数的使用方式如下所示。

```
01  struct servent *getservbyname(char *name, char *proto);
```

在参数列表中，name 是一个服务的名称。proto 是某服务使用的协议。函数 getservent 可以查出参数所对应的系统服务返回一个 servent 指针。servent 的定义方法如下所示。如果没有查询到这个服务，会返回 NULL 指针。在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01  #include <netdb.h>
```

下面的实例是用 getservbyname 函数通过名称来取得一个系统服务。

【范例 15-7】使用 getservbyname 函数按名称取得一个服务的信息，示例代码如下所示。

示例代码 15-7

```
#include <stdio.h>
#include <netdb.h>                                          /*包含相关的头文件。*/

int  main()
{
    struct servent *ser;                                /*定义一个表示服务的结构体指针。*/

    if( ser=getservbyname("ftp","tcp"))                     /*获取 FTP 服务的信息。*/
    {
      printf("name : %s\n",ser->s_name);                    /*服务名称。*/
      printf("port : %d  ",ntohs(ser->s_port));             /*服务端口号。*/
      printf("protocol:%s\n",ser->s_proto);                 /*服务协议。*/
      printf("alias: %s\n",ser->s_aliases[0]);              /*别名。*/
    }
    else
    {
      printf("there is no such a service.\n");              /*错误处理。*/
    }

    if( ser=getservbyname("http","tcp"))                    /*取得 http 服务的信息。*/
    {
      printf("name : %s\n",ser->s_name);                    /*输出信息。*/
      printf("port : %d  ",ntohs(ser->s_port));             /*端口号。*/
      printf("protocol:%s\n",ser->s_proto);                 /*协议。*/
      printf("alias: %s\n",ser->s_aliases[0]);              /*别名。*/
    }
    else
    {
      printf("there is no such a service.\n");              /*出错处理。*/
    }

    if(ser=getservbyname("asdasd","tcp"))                   /*取得一个不存在的服务。*/
    {
      printf("name : %s\n",ser->s_name);                    /*输出服务的信息。*/
      printf("port : %d  ",ntohs(ser->s_port));
      printf("protocol:%s\n",ser->s_proto);
      printf("alias: %s\n",ser->s_aliases[0]);
    }
    else
    {
     printf("there is no such a service.\n");               /*出错处理。*/
    }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.7.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
name : ftp
port : 21  protocol:tcp
alias: (null)
name : http
port : 80  protocol:tcp
```

```
alias: www
there is no such a service.
```

【代码解析】

- 代码第 06 行定义 servent 结构体指针，用于保存获取到的系统服务信息。
- 代码第 20 行和代码第 32 行分别调用函数 getservbyname 基于名称获得系统服务信息。前者正常返回，而后者给定的服务名称不存在，所以将打印错误信息。

15.5.3 由端口取得服务名称

函数 getservbyport 可以从一个端口取得一个服务的信息。这个函数的使用方法如下所示。

```
01  struct servent *getservbyport(int port, char *proto);
```

在参数列表中，port 是一个端口的编号。需要注意的是，这个端口号需要用 htons 函数进行转换。proto 是一个表示协议的字符串。函数会返回这个端口服务的 servent 类型的指针。servent 的结构体定义方法如前面章节所示。下面的实例使用 getservbyport 函数将一个端口号转换成一个服务类型。

【范例 15-8】使用 getservbyport 按端口号取得一个服务的信息，示例代码如下所示。

示例代码 15-8

```
#include <stdio.h>
#include <netdb.h>                                  /*包含相关的头文件。*/

int  main()
{
    struct servent *ser;                            /*定义一个表示服务的结构体指针。*/

    if( ser=getservbyport(htons(23),"tcp"))  /*取得 23 号端口服务的信息。*/
    {
      printf("name : %s\n",ser->s_name);            /*输出服务的信息。*/
      printf("port : %d\n",ntohs(ser->s_port));
      printf("protocol:%s\n",ser->s_proto);
      printf("alias: %s\n",ser->s_aliases[0]);
    }
    else                                            /*没有这个服务时的处理。*/
    {
      printf("there is no such a service.\n");
    }

    if( ser=getservbyport(htons(80),"tcp"))         /*取得 80 号端口的服务。*/
    {
      printf("name : %s\n",ser->s_name);            /*名称。*/
      printf("port : %d\n",ntohs(ser->s_port));     /*端口号。*/
      printf("protocol:%s\n",ser->s_proto);         /*协议名称。*/
      printf("alias: %s\n",ser->s_aliases[0]);      /*别名。*/
    }
    else
    {
      printf("there is no such a service.\n");      /*出错处理。*/
    }

    if( ser=getservbyport(htons(61111),"tcp"))      /*取得一个未知端口的服务。*/
    {
      printf("name : %s\n",ser->s_name);            /*输出服务的信息。*/
      printf("port : %d\n",ntohs(ser->s_port));
      printf("protocol:%s\n",ser->s_proto);
```

```
37          printf("alias: %s\n",ser->s_aliases[0]);
38      }
39      else
40      {
41          printf("there is no such a service.\n");          /*出错处理。*/
42      }
43  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.8.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
name : telnet
port : 23
protocol:tcp
alias: (null)
name : http
port : 80
protocol:tcp
alias: www
there is no such a service.
```

【代码解析】

- 代码第 06 行定义 servent 结构体指针，用于保存获取到的系统服务信息。
- 代码第 08 行和代码第 20 行分别调用函数 getservbyport 基于端口号 23 和 80 获得系统服务信息。代码第 32 行给定的端口号不存在，所以将打印错误信息。

15.6　网络地址的转换

网络 IP 地址本是用 32 位二进制数来表示的。为了记忆方便，可以用点分十进制数来表示 IP 地址。同时，网络传输与计算机内部的字符存储的方式是不同的，需要用相关函数将端口号进行转换。

15.6.1　将网络地址转换成整型

函数 inet_addr 可以将一个网络 IP 地址转换成一个十进制长整型数。这个函数的使用方法如下所示。

```
01  long inet_addr(char *cp);
```

函数的参数 cp 是一个 IP 地址字符串。函数会将这个 IP 地址转换成为一个长整型数。使用这个函数之前，需要在程序中包含下面的头文件。

```
01  #include <sys/socket.h>
02  #include <netinet/in.h>
03  #include <arpa/inet.h>
```

下面的实例是使用这个函数将一个 IP 地址转换成一个长整型数进行输出。

【范例 15-9】使用 inet_addr 函数将网络 IP 地址转换成十进制长整型数，示例代码如下所示。

示例代码 15-9

```
01  #include <stdio.h>
02  #include <sys/socket.h>
```

```
03  #include <netinet/in.h>
04  #include <arpa/inet.h>                          /*包含相关的头文件。*/
05
06  int  main()
07  {
08       char cp[]="192.168.1.1";                   /*IP 地址。*/
09       printf("%ld\n",inet_addr(cp));             /*转换成长整型地址。*/
10  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.9.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
16885952
```

【代码解析】代码第 09 行调用一个 inet_add 函数将一个给定的网络 IP 地址转换为整数并输出。

15.6.2 将长整型数地址转换成网络地址

函数 inet_ntoa 可以将整型数地址转换成一个点分十进制数网络地址。这个函数的使用方法如下所示。

```
01  char *inet_ntoa(struct in_addr in);
```

函数的参数 in 是一个 in_addr 类型的结构体。这个结构体的定义方法如下所示。

```
01  struct in_addr
02  {
03    uint32_t s_addr;
04  };
```

结构体只有一个成员，s_addr 是一个长整型数，用来存储一个长整型数的 IP 地址。函数 inet_ntoa 会将这个 IP 地址转换成一个字符串返回。使用这个函数之前，需要在程序的最前面包含下面的头文件。

```
01  #include <sys/socket.h>
02  #include <netinet/in.h>
03  #include <arpa/inet.h>
```

提示： 在计算机内部经常使用长整型数表示 IP 地址，而对于用户则表示为点分十进制数格式更容易理解，在网络程序设计中经常使用 inet_ntoa 完成长整型数地址到点分十进制数记法的转换。

下面的程序是这个函数的实例，将上一节地址转换生成的长整型数转换成为一个 IP 地址。

【范例 15-10】 使用 inet_ntoa 函数将整数转换为网络 IP 地址，示例代码如下所示。

示例代码 15-10

```
01  #include <stdio.h>
02  #include <sys/socket.h>
03  #include <netinet/in.h>
04  #include <arpa/inet.h>                          /*包含相关的头文件。*/
05
06  int  main()
07  {
```

```
08      struct in_addr ip;                          /*定义一个地址结构体。*/
09      ip.s_addr=16885952;                         /*长整型的地址。*/
10      printf("%s\n",inet_ntoa(ip));               /*转换以后输出。*/
11  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.10.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
192.168.1.1
```

【代码解析】代码第 10 行调用 inet_ntoa 函数将长整型数地址转换为点分十进制数表示法。

15.6.3　主机字符顺序与网络字符顺序的转换

计算机中的字符与网络中的字符存储顺序是不同的。计算机中的整型数与网络中的整型数进行交换时，需要用相关的函数进行转换。如果计算机中的长整型地址转换成网络字符顺序的整型地址，使用 htonl 函数。这些函数如下所示。

- uint32_t htonl(uint32_t hostlong);
- uint16_t htons(uint16_t hostshort);
- uint32_t ntohl(uint32_t netlong);
- uint16_t ntohs(uint16_t netshort);

这些函数的参数都是需要转换的整型数，函数把这些整型数转换以后返回。使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01  #include <arpa/inet.h>
```

使用上述函数进行相关的数值转换的代码示例请参看综合练习。

15.7　错误处理

网络程序中，可能因为网络故障、域名无法解析、主机不响应等出现错误。在编程时，需要对这些错误进行处理。本节将讲述程序中的错误处理方法。

15.7.1　herror 函数显示错误

函数 herror 可以显示上一个网络函数发生的错误。这个函数的使用方法如下所示。

```
01  void herror(const char *s);
```

这个函数的参数是一个字符串。在调用这个函数时，会先输出这个字符串，然后直接输出错误信息。输出的错误信息是上一个网络相关的函数发生的错误。所以这个函数可以在程序中随意使用。使用这个函数时，需要在程序的最前面包含下面的头文件。

```
01  #include <netdb.h>
```

下面的实例使用 herror 函数输出网络程序中的错误。

【范例 15-11】使用 herror 函数输出网络程序中的错误，示例代码如下所示。

示例代码 15-11

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

```
02  #include <netdb.h>                              /*包含头文件。*/
03
04  int  main()
05  {
06      char err[]="err:";
07      herror(err);                                 /*输出错误。*/
08  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.11.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。因为程序中没有调用网络函数，程序输出的结果是没有错误。

```
err:Resolver Error 0 (no error)
```

【代码解析】代码第 07 行调用 herror 函数获取网络程序中的错误信息。

15.7.2 捕获错误编号

网络程序中，可以使用下面的语句来捕获发生错误的编号。

```
01  extern int h_errno;
```

捕获这个错误编号以后，可以用 hstrerror 函数来输出这个错误信息。这个函数的使用方法如下所示。

```
01  char *hstrerror(int err);
```

这个函数的参数是已经捕获的错误编程，函数会返回这个编号所对应的错误信息字符串。在使用这个函数时，需要在程序的最前面包含下面的头文件。

```
01  #include <netdb.h>
```

提示： 在程序设计时，对于最终用户通常都是给出错误提示字符串，经常需要使用 hstrerror 函数。

下面的实例使用这个函数输出 0 号到 5 号的网络错误编号。

【范例 15-12】 使用 hstrerror 函数输出 0~5 的网络错误编号，示例代码如下所示。

示例代码 15-12

```
01  #include <stdio.h>
02  #include <netdb.h>                                  /*包含相关的头文件。*/
03
04  int  main()
05  {
06      int i;
07      for(i=0;i<6;i++)
08      {
09        printf("%d : %s \n",i,hstrerror(i));          /*按序号输出一个错误。*/
10      }
11  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.12.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
0 : Resolver Error 0 (no error)
1 : Unknown host
2 : Host name lookup failure
3 : Unknown server error
4 : No address associated with name
5 : Unknown resolver error
```

【代码解析】代码第 07~10 行使用 for 循环语句打印错误号 0~5 的字符串含义，代码第 09 行调用 hstrerror 函数将错误编号转换为字符串。

15.8　综合练习

本节利用网络地址转换函数实现本地与网络数值转换的程序，代码中通过调用 htons 函数、htonl 函数、ntohl 函数和 ntohs 函数完成本地与网络数值的转换操作。

【范例 15-13】本地与网络数值转换，示例代码如下所示。

示例代码 15-13

```
01  #include <stdio.h>                                          /*包含头文件*/
02  #include <arpa/inet.h>
03
04  int  main()                                                 /*主函数*/
05  {
06     long local;
07     int port;
08     local =123456;
09     port=1024;
10     printf("net: %d\n",htonl(local));                        /*转换成网络字节顺序。*/
11     printf("net: %d\n",htons(port));
12     printf("local: %d\n",ntohl(htonl(local)));               /*轮换成本地字节顺序。*/
13     printf("local: %d\n",ntohs(htons(port)));
14  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 15.13.c
```

输入下面的命令，运行这个程序。

```
./out
```

程序的运行结果如下所示。

```
net: 1088553216
net: 4
local: 123456
local: 1024
```

15.9　小结

本章主要讲述了 IP 地址、域名、网络套接字、端口、服务等概念和相关的处理，其中套接字的理解和网络编程中常用的结构体是本章的重点。在编程中，需要常常用到服务、地址、主机等结构体，需要理解这些结构体和结构体成员的作用与使用方法。最后一节的错误处理是网络编程时通用的错误处理方法，用这些错误处理函数可以方便地进行程序跟踪和调试。

15.10 习题

一、填空题

1. IP 地址的作用是_____。
2. 所谓_____，是计算机中为了标识计算机中访问网络的不同程序的编号。
3. _____是用来代替 IP 地址来标识计算机的一种直观名称。
4. TCP 与 UDP 就是网络中的两种数据传输约定，主要的区别是_____。
5. 所谓_____，指的是网络上的计算机通过运行程序，为其他的计算机提供信息或运算的功能。例如打开一个网页，是访问了网站服务器上的服务，从服务器下载了网页文件。
6. 常用的套接字类型有_____、_____、_____。

二、选择题

1. 下面说法中正确的是（　　）。
   A. IP 地址的作用是标识网络中的一台计算机，每一台计算机都有一个 IP 地址
   B. 域名方便用户记忆，在不知道 IP 地址时可以使用域名代替。域名可以和 IP 地址没有任何关系
   C. 所谓端口，是计算机中为了标识计算机中访问网络的不同程序的编号。相同的端口可被多个网络程序共同使用
   D. TCP 与 UDP 都是传输层的网络协议，它们没有什么区别
2. 使用（　　）函数将一个端口号转换成一个服务类型。
   A. getservbyport　　B. getservent
   C. getprotobynumber　　D. getservbyport
3. 函数（　　）的作用是取得系统所支持的服务。
   A. getservbyport　　B. getservent
   C. getprotobynumber　　D. getservbyport
4. 由协议编号取得协议信息的函数是（　　）。
   A. getservbyport　　B. getservent
   C. getprotobynumber　　D. getservbyport

三、简答题

1. 简述 IP 地址与域名的概念及它们之间的关系。
2. 什么是套接字，套接字有哪些类型？
3. 什么是网络协议与网络服务？

四、编程题

1. 编写程序实现由协议名和协议编号获得协议数据。
2. 编写程序实现由名称取得系统所支持的服务。

第 16 章　无连接的套接字通信

本章首先介绍了有关套接字的基本概念，以及如何创建套接字、取得和设置套接字状态，然后介绍了基于 UDP 协议的无连接的套接字通信的工作流程，以及如何发送和接收信息。本章的主要内容如下：

- socket 套接字的建立、获取状态和状态设置；
- 无连接的套接字通信工作流程、信息发送与接收。

16.1　socket 套接字

所谓 socket 套接字，指的是在网络通信以前建立的通信接口。在进行网络连接以前，需要向系统注册申请一个新的 socket。然后使用这个 socket 进行网络连接。本章将讲解 socket 的建立与访问操作。

提示： 套接字=传输层协议+端口号+IP 地址。

16.1.1　建立 socket

在进行网络连接之前，需要用 socket 函数申请一个通信端口。这个函数的使用方法如下所示。

```
01  int socket(int domain, int type, int protocol);
```

在参数表中，domain 指定使用何种地址类型，可能的值是下面这些系统常量。

- PF_UNIX、PF_LOCAL、AF_UNIX、AF_LOCAL：这些是 UNIX 进程通信协议。
- PF_INET、AF_INET：Ipv4 网络协议。
- PF_INET6、AF_INET6：Ipv6 网络协议。
- PF_IPX、AF_IPX：IPX-Novell 协议。
- PF_NETLINK、AF_NETLINK：核心用户接口装置。
- PF_X25、AF_X25、ITU-T X.25：ISO-8208 协议。

type 参数的作用是设置通信的协议类型。可能的取值如下所示。

- SOCK_STREAM：提供面向连接的稳定数据传输，即 TCP 协议。
- OOB：在所有数据传送前必须使用 connect()来建立连线状态。
- SOCK_DGRAM：使用不连续不可靠的数据包连接。
- SOCK_SEQPACKET：提供连续可靠的数据包连接。

参数 protocol 指定 socket 所使用的传输协议编号。这一参数通常不具体设置，一般设置为 0 即可。在使用这个函数建立套接字以前，需要在程序的最前面包含下面的头文件。

```
01  #include <sys/types.h>
02  #include <sys/socket.h>
```

如果建立套接字成功，则返回这个套接字的编号。如果不成功，则返回-1。这个函数可能

发生的错误与含义如下所示。

- EPROTONOSUPPORT：参数 domain 指定的类型不支持参数 type 或 protocol 指定的协议。
- ENFILE：核心内存不足，无法建立新的 socket 结构。
- EMFILE：进程文件表溢出，无法再建立新的套接字。
- EACCESS ：权限不足，无法建立 type 或 protocol 指定的协议。
- ENOBUFS、ENOMEM：内存不足。
- EINVAL：参数不合法。

注意：只创建套接字无法进行正常的网络体系，要想和其他计算机通信，还必须借助于其他函数将建立的套接字与 IP 地址和端口号联系起来。

下面的实例使用 socket 函数建立一个套接字。

【范例 16-1】使用 socket 函数建立套接字，示例代码如下所示。

示例代码 16-1

```
#include<sys/types.h>
#include<sys/socket.h>
#include<stdio.h>                                          /*包含头文件。*/

int  main()
{
  int s;
  if((s = socket(AF_INET,SOCK_STREAM,0))<0)                /*建立一个套接字。*/
  {
    perror("connect");                                     /*输出错误信息。*/
    exit(1);                                               /*出错则退出程序。*/
  }
  else                                                     /*成功则输出相关信息。*/
  {
      printf("a socket was created.\n");
      printf("socket number:%d\n",s);
   }

  if((s = socket(AF_INET,123,0))<0)                        /*用错误的参数建立一个套
接字。*/
  {
    perror("connect");                                     /*输出错误信息。*/
    exit(1);
  }
  else                                                     /*建立成功的情况。*/
  {
      printf("a socket was created.\n");
      printf("socket number:%d\n",s);
    }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 16.14.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a socket was created.
```

```
socket number:3
connect: Invalid argument
```

【代码解析】

- 代码第 08 行使用 socket 函数建立一个套接字，此套接字使用 ipv4 地址和 TCP 协议。
- 代码第 19 行则通过传递给 socket 函数错误的参数演示了套接字创建出错的情况。

16.1.2　取得 socket 状态

函数 getsockopt 可以取得一个 socket 的参数。这个函数的使用方法如下所示。

```
01  int getsockopt(int s,int level,int optname,void* optval,socklen_t* optlen);
```

在参数列表中，s 是已经建立 socket 的编号。level 代表需要设置的网络层，一般设置成 SOL_SOCKET 来表示 socket 层。参数 optname 代码的是需要获取的选项，可以设置成下面这些值。

- SO_DEBUG：打开或关闭排错模式。
- SO_REUSEADDR：允许在 bind 函数中本地地址可重复使用。
- SO_TYPE：返回 socket 形态。
- SO_ERROR：返回 socket 已发生的错误原因。
- SO_DONTROUTE：送出的数据包不要利用路由设备来传输。
- SO_BROADCAST：使用广播方式传送。

参数 optval 是取得的某个参数的返回值指针，程序的返回值会保存在这个指针指向的变量中。optlen 是 optval 的内存长度。函数如果执行成功，则返回 0，反之返回-1。这个函数可能发生下面这些错误。

- EBADF：参数 s 不是合法的 socket 代码。
- ENOTSOCK：参数 s 为一个打开文件的编号，而不是一个 socket。
- ENOPROTOOPT：参数 optname 指定的选项不正确。
- EFAULT：参数 optval 指针指向的内存空间无法读取。

在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
```

下面的程序用 getsockopt 函数来读取一个 socket 的参数。

【范例 16-2】使用 getsockopt 函数取得套接字信息，示例代码如下所示。

示例代码 16-2

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
03  #include<stdio.h>                                    /*包含头文件。*/
04
05  int  main()
06  {
07    int s;                                             /*定义相关的变量。*/
08    int val,len ;
09    len= sizeof(int);                                  /*长度。*/
10
11    if((s = socket(AF_INET,SOCK_STREAM,0))<0)          /*建立一个套接字。*/
12    {
13      perror("connect");
14      exit(1);
15    }
16    else                                               /*建立成功。*/
17    {
```

```
18          printf("a socket was created.\n");
19          printf("socket number:%d\n",s);
20      }
21
22      getsockopt(s,SOL_SOCKET,SO_TYPE,&val,&len);   /*取得套接字的一个信息。*/
23      perror("socket:");                            /*输出错误。*/
24      printf("optval = %d\n",val);                  /*输出结果。*/
25      getsockopt(100,SOL_SOCKET,SO_TYPE,&val,&len); /*用错误的方法取得一个信息。*/
26      perror("socket:");
27  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 16.16.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a socket was created.
socket number:3
socket:: Success
optval = 1
socket:: Bad file descriptor
```

【代码解析】

- 代码第 11 行使用 socket 函数创建一个新的套接字，代码第 22 行使用 getsockopt 函数获取新建立的套接字信息。
- 代码第 25 行给 getsockopt 函数传递了错误的参数，代码第 26 行使用 perror 函数获得错误信息。

16.1.3 设置 socket 状态

函数 setsockopt 可以设置一个 socket 的状态，这个函数的使用方法如下所示。

```
01  int setsockopt(int s,int level,int optname,const void * optval,socklen_toptlen);
```

在参数列表中，s 是已经打开的 socket。参数 level 代表欲设置的网络层，通常设成 SOL_SOCKET 以存取 socket 层。参数 optname、optval、toptlen 的含义与 16.1.2 节中的这些参数作用相同。

如果函数设置 socket 成功则返回 0，若有错误则返回-1。这个函数可能发生下面列出的错误。可以用 errno 捕获已经发生的错误。

- EBADF：参数 s 不是合法的 socket 代码。
- ENOTSOCK：参数 s 为一个打开文件的编号，而不是一个 socket。
- ENOPROTOOPT：参数 optname 指定的选项不正确。
- EFAULT：参数 optval 指针指向的内存空间无法读取。

使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01  #include <sys/types.h>
02  #include <sys/socket.h>
```

下面的函数使用 setsockopt 函数来设置一个 socket 的状态。参数 optname 设置成 SO_TYPE，表示设置这个 socket 的状态，设置的值为 1。

【范例 16-3】使用 setsockopt 函数设置套接字状态，示例代码如下所示。

示例代码 16-3

```
01  #include<sys/types.h>
```

```
02  #include<sys/socket.h>
03  #include<stdio.h>                                    /*包含头文件。*/
04
05  int  main()
06  {
07    int s;                                            /*定义相关的变量。*/
08    int val=1,len,i ;
09    len= sizeof(int);
10
11    if((s = socket(AF_INET,SOCK_STREAM,0))<0)         /*建立一个套接字。*/
12    {
13      perror("connect");
14      exit(1);
15    }
16    else                                              /*套接字建立成功。*/
17    {
18        printf("a socket was created.\n");
19        printf("socket number:%d\n",s);
20     }
21
22     i=setsockopt(s,SOL_SOCKET,SO_TYPE,&val,len);     /*设置套接字的一个参数。*/
23     if("i==0")                                       /*判断结果情况。*/
24     {
25        printf("set socket ok.\n.");
26      }
27      else
28     {
29        printf("set socket error.\n.");
30     }
31
32     setsockopt (100,SOL_SOCKET,SO_TYPE,&val,&len);/*用错误的方法设置一个套接字。
33  */
34     perror("socket");                                /*输出错误。*/
35  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 16.16.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
a socket was created.
socket number:3
set socket ok.
socket: Bad file descriptor
```

【代码解析】

- 代码第 11 行使用 socket 函数创建一个新的套接字，代码第 22 行使用 setsockopt 函数设置新建立的套接字信息。
- 代码第 32 行给 setsockopt 函数传递了错误的参数，代码第 34 行使用 perror 函数获得错误信息。

16.2　无连接的套接字通信

所谓无连接的套接字通信，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信

时，两台计算机之间没有建立连接的过程。需要处理的内容只是把信息发送到另一台计算机。这个通信的方式比较简单。本节将讲述这种无连接的 UDP 通信。

提示：由于无连接的套接字通信基于 UDP 协议，所以无法保证数据传输的可靠性。

16.2.1 工作流程

无连接套接字的通信不需要建立起客户机与服务器之间的连接，因此在程序中没有建立连接的过程。进行通信之前，需要建立网络套接字。服务器需要绑定一个端口，在这个端口上监听接收到的信息。客户机需要设置远程 IP 和端口，需要传递的信息需要发送到这个 IP 和端口上。客户机和服务器的交互过程可用图 16-1 来表示。

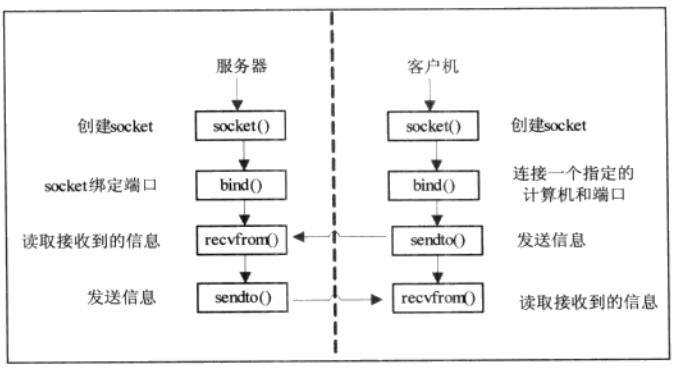


图 16-1 无连接的套接字通信

16.2.1 信息发送函数 sendto

函数 sendto 可以通过一个已经建立的套接字，将一段信息发送到另一个程序的套接字中。这个函数的使用方法如下所示。

```
01  int sendto ( int s ,  void * msg, int len, unsigned int flags,struct sockaddr
* to , int tolen ) ;
```

在参数列表中，s 是已经建立好的 socket。在使用 UDP 协议时，不需要进行计算机连接的操作。msg 是需要发送的字符串。len 是发送字符串的长度。参数 flags 一般设 0，其他可能的数值如下所示。

- MSG_OOB：传送的数据以 out-of-band 送出。
- MSG_DONTROUTE：取消路由表查询。
- MSG_DONTWAIT：设置为不可阻断传输。
- MSG_NOSIGNAL：此传输不愿被 SIGPIPE 信号中断。

参数 sockaddr 是一个表示套接字的结构体。这个结构体的定义如下所示。

这个结构体的定义方法如下所示。

```
01  struct socketaddr_in
02  {
03     unsigned short int sin_family;
04     uint16_t sin_port;
05     struct in_addr sin_addr;
06     unsigned char sin_zero[8];
```

```
07  };
```

这个结构体的成员与作用如下所示。

- sin_family：与 sockaddr 结构体中的 sa_family 相同。
- sin_port：套接字使用的端口号。
- sin_addr：需要访问的 IP 地址。
- sin_zero：未使用的字段，填充为 0。

在这一结构体中，in_addr 也是一个结构体，定义方法如下所示。作用是保存一个 IP 地址。

```
01  struct in_addr
02  {
03    uint32_t s_addr;
04  };
```

在程序中，需要设置这个结构体每一个成员的值。参数 tolen 是 sockaddr 结构体的长度，这一长度可用 sizeof 函数来取得。这个结构体会把 socket 用指定的 socket 传送给对方主机。结果传送成功，则返回传送字符的个数。传送失败则返回-1。传送失败时，错误原因保存在 errno 中。

提示：使用 sendto 函数发送信息之前，需要在程序中包含下面的头文件。

```
01 #include < sys/types.h >
02 #include < sys/socket.h >
```

16.2.2　信息接收函数 recvfrom

函数recvfrom可以从一个socket中接收其他主机发送来的信息。这个函数的使用方法如下所示。

```
01  int recvfrom(int s,void *buf,int len,unsigned int flags ,sockaddr *from ,int
*fromlen);
```

在参数列表中，s 是一个已经建立的网络套接字。buf 是接收到的信息保存到的内存地址。len 是可以保存信息的 buf 内存长度。flags 一般设为 0。其他可能的设置如下所示。

- MSG_OOB：接收以 out-of-band 送出的数据。
- MSG_PEEK：返回来的数据不从系统内删除，如果再调用 recv()，会返回相同的数据内容。
- MSG_WAITALL：强迫接收到 len 大小的数据后才能返回，除非有错误或信号产生。
- MSG_NOSIGNAL：此操作不愿被 SIGPIPE 信号中断。

参数 sockaddr 是 IP 地址与端口等信息。fromlen 是 sockaddr 的长度，这个值可以用 sizeof 函数来取得。这个函数调用成功，则返回接收到的字符数。失败则返回-1，错误原因存于 errno 中。使用 recvfrom 函数接收数据以前，需要在程序的最前面包含下面的头文件。

```
01  #include <sys/types.h>
02  #include <sys/socket.h>
```

16.3　综合练习

上一节讲述了无连接的套接字通信原理和相关函数。本节将使用这些函数编写无连接的 UDP 套接字信息传送程序。在信息传输时，需要有服务器端与客户端。服务器处于监听状态，接收到客户端发送的信息以后会返回一定的信息。客户端会主动向服务器发送信息。

16.3.1　无连接套接字通信客户端

本节将讲述无连接套接字通信的客户端。这个程序的主要内容是建立一个套接字，然后从键盘读取一个字符串。用 sendto 函数将这个字符串发送到服务器，然后接收服务器发送回的信

息。在实际编程时，客户机与服务器是在同一台计算机上的。可以使用 127.0.0.1 这个 IP 地址表示本地计算机。这样和访问远程计算机的效果是相同的。

【范例 16–4】编程实现 UDP 客户端程序，示例代码如下所示。

示例代码 16-4

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>                          /*包含相关的头文件。*/

#define REMOTEPORT 4567                          /*定义表示端口号的常量。*/
#define REMOTEIP "127.0.0.1"                     /*定义表示表 IP 的常量。*/

int  main(int argc,char *argv[])
{
   int s,len;                                    /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len;
   char msg[256];                                /*定义一个数组发送与接收数据。*/
   int i=0;

   if (( s= socket(AF_INET, SOCK_DGRAM, 0) )<0) /*建立一个 socket。*/
   {
      perror("error");                           /*输出错误。*/
      exit(1);
   }
   else
   {
    printf("socket created .\n");                /*输出提示信息。*/
    printf("socked id: %d \n",s);
    printf("remote ip: %s \n",REMOTEIP);
    printf("remote port: %d \n",REMOTEPORT);
   }

   len=sizeof(struct sockaddr_in);               /*长度。*/
   bzero(&addr,sizeof(addr));                    /*空间地址结构体所在的内存空间。*/
   addr.sin_family=AF_INET;                          /*填充地址与端口的信息。*/
   addr.sin_port=htons(REMOTEPORT);                  /*端口。*/
   addr.sin_addr.s_addr=inet_addr(REMOTEIP);

   while (1)                                         /*循环。*/
   {
      bzero(msg,sizeof(msg));                        /*清空 msg 所在的内存。*/
      len = read(STDIN_FILENO,msg,sizeof(msg)); /*接收信息。*/
      sendto(s,msg,len,0,&addr,addr_len);            /*发送信息。*/
      printf("\nInput message: %s \n",msg);          /*输出结果。*/
      len= recvfrom (s,msg,sizeof(msg),0,&addr,&addr_len);
                                                     /*这是接收到的信息。*/
      printf("%d :",i);                              /*输出计数。*/
      i++;                                           /*计数自加。*/
      printf("Received message: %s \n",msg);         /*这是服务器返回的信息。*/
   }
}
```

【运行结果】输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 16.17.c -o udpcli
```

程序的运行结果请参看前面章节。

16.3.2　无连接套接字通信服务器

本节将讲述无连接套接字通信的服务器。网络传输的功能需要多个计算机上的程序才能完成数据通信。上一节编写程序发送的信息，需要有一个服务器程序来接收和处理。

提示：创建网络服务器程序时需要调用 bind 函数将套接字与端口号和 IP 地址实现绑定。

在服务器程序中，主要是使用 recvfrom 函数从套接字中接收传入的信息。对信息进行处理以后，用 sendto 函数发回一段信息。服务器程序的代码如下所示。

【范例 16-5】编程实现 UDP 服务器端程序，示例代码如下所示。

示例代码 16-5

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>                          /*包含相关的头文件。*/

#define LOCALPORT 4567                           /*定义一个端口号。*/

int  main(int argc,char *argv[])
{
   int mysock,len;                               /*定义相关的变量。*/
   struct sockaddr_in addr;
   int i=0;
   char msg[256];                                /*定义一个数组，保存发送与接收的信息。
*/
   int addr_len;

    if (( mysock= socket(AF_INET, SOCK_DGRAM, 0) )<0)
                                                 /*建立一个连接。*/
   {
      perror("error");                           /*输出错误。*/
      exit(1);
   }
   else                                          /*socket 建立成功则提示信息。*/
   {
    printf("socket created .\n");
    printf("socked id: %d \n",mysock);
    }

   addr_len=sizeof(struct sockaddr_in);          /*长度。*/
   bzero(&addr,sizeof(addr));                    /*清空地址所在的内存。*/
   addr.sin_family=AF_INET;                      /*填充地址结构体。*/
   addr.sin_port=htons(LOCALPORT);
   addr.sin_addr.s_addr=htonl(INADDR_ANY);

   if(bind(mysock,&addr,sizeof(addr))<0)         /*在 socket 上面绑定端口号与 IP。
*/
```

```
    {
       perror("connect");                                  /*输出错误信息。*/
       exit(1);
    }
    else
    {
     printf("bind ok.\n");                                 /*绑定成功则输出信息。*/
     printf("local port :%d \n",LOCALPORT);
    }

    while (1)                                              /*进入一个循环。*/
    {
       bzero(msg,sizeof(msg));                             /*请空 msg 所在的内存。*/
       len= recvfrom (mysock,msg,sizeof(msg),0,&addr,&addr_len);
                                                           /*接收到信息。*/
       printf("%d :",i);                                   /*输出计数。*/
       i++;                                                /*计数自加。*/
       printf("message from : %s \n",inet_ntoa(addr.sin_addr));
                                                           /*输出 IP 地址。*/
       printf("  message length : %d \n",len);             /*输出长度信息。*/
       printf("  message : %s \n\n",msg);                  /*输出信息。*/
       sendto(mysock,msg,len,0,&addr,addr_len);            /*字符串返回给客户端*/
    }
}
```

【运行结果】输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 16.18.c -o udpser
```

程序的运行结果请参看前面章节。

16.3.3 测试 UDP 通信程序

编写出的网络通信程序，并不能单一测试一个功能。在进行测试时，需要使用客户端发送数据，查看服务器是否正常接收和处理了数据。然后再查看客户端是否接收到服务器发送回的数据。本章将对前面两节编写的程序进行测试。

① 输入下面的命令，打开 UDP 通信的服务器端。

```
./udpser
```

② 程序的运行结果如下所示。显示结果表明已经建立起一个 socket，监听的本地端口为 4567。

```
socket created .
socked id: 3
bind ok.
local port :4567
```

③ 打开另一个终端。在终端中输入下面的命令，打开客户端程序。

```
./udpcli
```

④ 程序显示的结果如下所示。表明已经建立了一个 socket，远程的 IP 是 127.0.0.1，实际上就是本地计算机。远程端口是 4567。

```
socket created .
socked id: 3
remote ip: 127.0.0.1
remote port: 4567
```

⑤ 输入下面的文本，向服务器发送一个信息。

```
hello
```

⑥ 程序再次显示这个字符串，然后将这个字符串发送到服务器。服务器接收这段信息后返回同样的信息，结果如下所示。

```
Input message: hello
0 :Received message: hello
```

⑦ 在服务器的终端中，显示的结果如下所示。

```
0 :message from : 127.0.0.1
  message length : 6
  message : hello
```

⑧ 再在客户端的终端中输入下面的信息。

```
good morning .
```

⑨ 客户端显示的结果如下所示。表明第二次接收到信息。

```
Input message: good morning .
1 :Received message: good morning .
```

⑩ 在服务端查看传递的信息，结果如下所示。

```
1 :message from : 127.0.0.1
  message length : 15
  message : good morning .
```

⑪ 上面的测试结果表明，这两个程序可以正常完成两个计算机之间的字符串传输。在本地计算机测试正常以后，在广域网中运行一般是正常的。这两个程序可以作为不同计算机之间的聊天工具。

注意：这个程序进入了 while 循环以后，就一直进行信息发送和接收的操作，不能结束循环。可以按“Ctrl”+“C”组合键关闭这两个程序。

16.4　小结

本章讲述了网络通信套接字和无连接的套接字通信。在学习时，需要重点理解套接字的含义和套接字的基本操作。无连接的套接字通信是一种简单的网络通信方式，编程的重点是信息的发送与接收。本章的实例，是无连接的套接字通信基本方式，可以实现两个计算机之间的文字信息传输，读者可以在这两个程序的基础上编写出文本模式下的聊天软件。

16.5　习题

一、填空题

1．所谓_____，指的是在网络通信以前建立的通信接口。进行网络连接以前，需要向系统注册申请一个新的_____。然后使用这个_____进行网络连接。

2．所谓_____，指的是使用 UDP 协议进行信息传输。使用这种协议进行通信时，两个计算机之间没有建立连接的过程。需要处理的内容只是把信息发送到另一台计算机。

3．无连接的套接字通信使用函数_____发送数据，使用函数_____接收函数。

二、选择题

1．取得 socket 状态使用函数（　　）。

A．getsockopt　　B．setsockopt

C．socket　　D．sendto

2．设置 socket 的状态使用函数（　　）。

A．getsockopt　　B．setsockopt

C．socket　　D．sendto

3．无连接的套接字通信中，收发信息的函数是（　　）。

A．send 和 recv　　B．sendto 和 recvfrom

C．socket 和 sendto　　D．sendto 和 socket

三、简答题

1．简述无连接的套接字通信原理及工作流程。

2．如何创建套接字，如何取得与设置 socket 状态信息？

四、编程题

编程实现无连接套接字通信的客户端与服务器端。实现客户的发送一行文件。立即显示在服务其上（在客户端建立一个套接字，然后从键盘读取一个字符串。用 sendto 函数将这个字符串发送到服务器，然后接收服务器发送回的信息）。

第 17 章　面向连接的套接字通信

网络应用程序中的通信大部分都是基于 TCP 协议的面向连接的套接字通信，本章首先介绍了面向连接的套接字通信工作流程，然后介绍了端口绑定、监听和连接有关的概念，最后介绍了基于连接的套接字通信中如何进行数据的发送与接收。本章的主要内容如下：

- 面向连接的套接字通信工作流程；
- 绑定端口；
- 监听与连接；
- 数据的发送与接收。

17.1　面向连接的套接字通信工作流程

为了实现服务器与客户机的通信，服务器和客户机都必须建立套接字。服务器与客户机的工作原理可以用下面的过程来描述。

（1）服务器先用 socket 函数来建立一个套接字，用这个套接字完成通信的监听。

（2）用 bind 函数来绑定一个端口号和 IP 地址。因为本地计算机可能有多个网址和 IP，每一个 IP 和端口有多个端口。需要指定一个 IP 和端口进行监听。

（3）服务器调用 listen 函数，使服务器的这个端口和 IP 处于监听状态，等待客户机的连接。

（4）客户机用 socket 函数建立一个套接字，设定远程 IP 和端口。

（5）客户机调用 connect 函数连接远程计算机指定的端口。

（6）服务器用 accept 函数来接收远程计算机的连接，建立起与客户机之间的通信。

（7）建立连接以后，客户机用 write 函数向 socket 中写入数据。也可以用 read 函数读取服务器发送来的数据。

（8）服务器用 read 函数读取客户机发送来的数据，也可以用 write 函数来发送数据。

（9）完成通信以后，用 close 函数关闭 socket 连接。

客户机与服务器建立面向连接的套接字进行通信，请求与响应过程可用图 17-1 来表示。

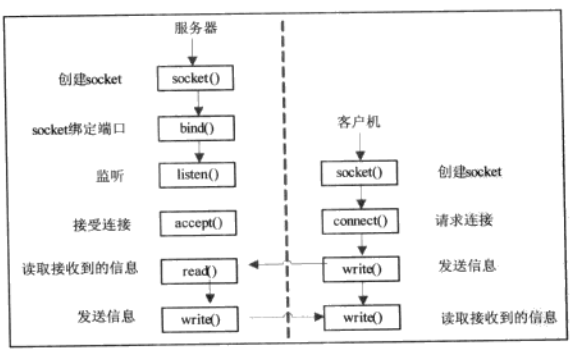


图 17-1　面向连接的套接字通信工作流程

17.2 绑定端口

绑定端口指的是将套接字与指定的端口相连。用 socket 函数建立起一个套接字以后，需要用 bind 函数在这个套接字上面绑定一个端口。本节将讲解套接字的绑定端口操作。

提示：只有套接字建立后才能够执行端口绑定操作。

17.2.1 绑定端口函数 bind

函数 bind 可以将一个端口绑定到一个已经建立的 socket 上，这个函数的使用方法如下所示。

```
01  int bind(int sockfd,struct sockaddr * my_addr,int addrlen);
```

参数列表中，sockfd 是已经建立的 socket 编号。sockaddr 是一个指向 sockaddr 结构体类型的指针。sockaddr 的定义方法如下所示。

```
01  struct sockaddr
02  {
03    unsigned short int sa_family;
04    char sa_data[14];
05  };
```

这个结构体的成员含义如下所示。

- sa_family：为调用 socket()时的 domain 参数，即 AF_xxxx 值。
- sa_data：最多使用 14 个字符长度，含有 IP 地址与端口的信息。

如果建立 socket 时使用的是 AF_INET 参数，则 socketaddr 结构体的定义方法如下所示。

```
01  struct socketaddr_in
02  {
03    unsigned short int sin_family;
04    uint16_t sin_port;
05    struct in_addr sin_addr;
06    unsigned char sin_zero[8];
07  };
```

结构体的成员 addr 也是一个结构体，定义方式如下所示。

```
01  struct in_addr
02  {
03    uint32_t s_addr;
04  };
```

参数 addrlen 是 my_addr 的长度，可以用 sizeof 函数来取得。函数可以把指定的 IP 与端口绑定到已经建立的 socket 上面。如果绑定成功，则返回 0。失败则返回-1。函数可能发生下面的错误，可以用 error 捕获发生的错误。

- EBADF：参数 sockfd 不是一个合法的 socket。
- EACCESS：权限不足。
- ENOTSOCK：参数 sockfd 是文件描述词，而不是 socket。

提示：在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01 #include<sys/types.h>
02 #include<sys/socket.h>
```

17.2.2　bind 函数绑定端口实例

下面的实例，是使用 bind 函数在一个打开的 socket 上面绑定 IP 与端口。绑定的端口是 5678，IP 为 INADDR_ANY，表示本地计算机的默认 IP 地址。

【范例 17-1】使用 bind 函数实现绑定端口，示例代码如下所示。

示例代码 17-1

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>                                    /*包含头文件。*/

#define PORT 5678                                     /*定义一个表示端口的常量。*/

main()
{
   int sockfd,newsockfd, fd;                          /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];              /*定义一个字符数组存储发送和接收到的信息。*/


   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else                                               /*建立成功。*/
   {
      printf("socket created .\n");
      printf("socked id: %d \n",sockfd);
   }

   bzero(&addr,sizeof(addr));                         /*清空表示地址的结构体变量。*/
   addr.sin_family =AF_INET;
   addr.sin_port = htons(PORT);                       /*设置 addr 的成员。*/
   addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if(bind(sockfd,&addr,sizeof(addr))<0)              /*bind 函数绑定端口。*/
   {
      perror("connect");
      exit(1);
   }
   else                                               /*失败的情况。*/
   {
      printf("connected.\n");
      printf("local port:%d\n",PORT)    ;
   }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.19.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
```

【代码解析】代码第 01~05 行引入必需的头文件。代码第 11 行建立一个套接字，代码第 34 行调用 bind 函数将已建立的套接字与地址进行绑定。

17.3 监听与连接

所谓监听，指的是 socket 的端口处于等待状态，如果有客户端有连接请求，这个端口会接受这个连接。连接指的是客户端向服务端发送一个通信申请，服务端会响应这个请求。本节将讲述 socket 的监听与连接操作。

17.3.1 等待监听函数 listen

服务器必须等待客户端的连接请，listen 函数用实于实现监听等待功能。这个函数的使用方法如下所示。

```
01   int listen(int s,int backlog);
```

在参数列表中，s 是已经建立的 socket。backlog 是能同时处理的最大连接请求。如果超过这个数目，客户端将会接收到 ECONNREFUSED 拒绝连接的错误。

注意：listen 并未真正接受连接，只是设置 socket 的状态为 listen 模式。真正接受客户端连接的是 accept 函数。通常情况下，listen 函数会在 socket、bind 函数之后调用，然后才会调用 accept 函数。

listen 函数只适用 SOCK_STREAM 或 SOCK_SEQPACKET 的 socket 类型。如果 socket 为 AF_INET，则参数 backlog 最大值可设至 128，即最多可以同时接收 128 个客户端的请求。如果调用成功，则函数的返回值为 0，失败返回-1。函数可能发生错误，可以用 errno 来捕获发生的错误。

提示：在使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
#include<sys/socket.h>
```

17.3.2 listen 函数使用实例

本节将讲述一个 listen 函数使用实例。在程序中，先建立一个 socket，然后用 bind 函数在这个 socket 上面绑定端与 IP。然后用 listen 函数设置这个 socket 进行监听。程序的代码如下所示。

【范例 17-2】使用 listen 函数监听指定端口，示例代码如下所示。

示例代码 17-2

```
01   #include<sys/types.h>
02   #include<sys/socket.h>
03   #include<netinet/in.h>
04   #include<arpa/inet.h>
05   #include<unistd.h>                                    /*包含头文件。*/
06
```

```
#define PORT 5678                                          /*定义一个表示端口号的常量。*/
#define MAX 10                                             /*最多的连接客户端数。*/

main()
{
   int sockfd,newsockfd,is_connected[MAX],fd;     /*定义相关的变量。*/
   struct sockaddr_in addr;
   int addr_len = sizeof(struct sockaddr_in);
   fd_set myreadfds;
   char msgbuffer[256];
   char msg[] ="This is the message from server.Connected.\n";

   if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket。*/
   {
      perror("socket");
      exit(1);
   }
   else                                                    /*socket 建立成功。*/
   {
      printf("socket created .\n");
      printf("socked id: %d \n",sockfd);
   }

   bzero(&addr,sizeof(addr));                  /*清空 addr 所在的内存。31     */
   addr.sin_family =AF_INET;                   /*设置 addr 的相关信息。33     */
   addr.sin_port = htons(PORT);
   addr.sin_addr.s_addr = htonl(INADDR_ANY);

   if(bind(sockfd,&addr,sizeof(addr))<0)       /*绑定端口。*/
   {
      perror("connect");
      exit(1);
   }
   else                                                    /*端口绑定成功。*/
   {
       printf("connected.\n");
       printf("local port:%d\n",PORT)    ;
   }

   if(listen(sockfd,3)<0)                                  /*监听一个端口。*/
   {
     perror("listen");                                     /*错误则输出提示。*/
     exit(1);
   }
   else                                                    /*输出结果。*/
   {
     printf("listenning......\n");
   }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.20.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
```

```
local port:5678
listenning......
```

【代码解析】本程序的前半部分与示例代码 17-1 相同，代码第 48 行调用 listen 函数使建立的套接字监听指定的端口。

17.3.3 接受连接函数 accept

服务器处于监听状态时，如果获得客户机的请求，会将这个请求放在等待队列中。当系统空闲时，将处理客户机的连接请求。接受连接请求的函数是 accept，这个函数的使用方法如下所示。

```
01  int accept(int s,struct sockaddr * addr,int * addrlen);
```

在参数列表中，s 是处于监听状态的 socket。addr 是一个 sockaddr 结构体类型的指针，系统会把远程主机的这些信息保存到这个结构体指针上。addrlen 是 sockaddr 的内存长度，可以用 sizeof 函数来取得。

注意：当 accept 函数接受一个连接时，会返回一个新的 socket 编号。以后的数据传输与读取就是通过这个新的 socket 编号来处理的。原来参数中的 socket 可以继续使用。接受连接以后，远程主机的地址和端口信息将会保存到 addr 所指的结构体内。如果处理失败，返回值为-1。

使用这个函数以前，需要在程序中包含下面的头文件。

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
```

17.3.4 accept 函数使用实例

本节将讲述一个 accept 函数使用实例。在程序中，先建立一个 socket。然后用 bind 函数在这个 socket 函数上面绑定一个端口。然后使用 listen 函数使这个端口处于监听状态。当有连接请求时，accpet 函数会产生一个新的 socket，然后输出提示信息。程序的代码如下所示。

【范例 17-3】使用 accept 函数接受连接，示例代码如下所示。

示例代码 17-3

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
03  #include<netinet/in.h>
04  #include<arpa/inet.h>
05  #include<unistd.h>                                    /*包含头文件。*/
06
07  #define PORT 5678                                     /*定义一个端口常量。*/
08
09  main()
10  {
11     int sockfd,newsockfd,fd;                           /*定义相关的变量。*/
12     struct sockaddr_in addr;
13     int addr_len = sizeof(struct sockaddr_in);
14     fd_set myreadfds;
15     char msgbuffer[256];                               /*定义一个字符串。*/
16
17     if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket。*/
18     {
19        perror("socket");
20        exit(1);
21     }
```

```
    else
    {
       printf("socket created .\n");                          /*socket 建立成功。*/
       printf("socked id: %d \n",sockfd);
    }

    bzero(&addr,sizeof(addr));                                /*设置 addr 的信息。*/
    addr.sin_family =AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if(bind(sockfd,&addr,sizeof(addr))<0)                     /*绑定端口号。*/
    {
       perror("connect");
       exit(1);
    }
    else
    {
        printf("connected.\n");
        printf("local port:%d\n",PORT)    ;
    }

    if(listen(sockfd,3)<0)                                    /*监听一个端口号。*/
    {
       perror("listen");
       exit(1);
    }
    else
    {
       printf("listenning......\n");
    }

    if((newsockfd = accept (sockfd,&addr,&addr_len))<0)       /*接受一个连接。*/
    {
      perror("accept");
    }
    else
    {
        printf("accepted a new connecttion.\n");
    }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.21.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
listenning......
```

结果表明，本地计算机的 5678 号端口处于监听状态。打开浏览器，在浏览器的地址栏中输入下面的网址，然后按“Enter”键。这样浏览器会请求连接本地计算机的 5678 号端口。

```
http://127.0.0.1:5678/
```

浏览器会显示无法打开这个网页。在终端中显示的结果如下所示。表明这个程序已经接受

了这个连接然后退出了程序。

```
accepted a new connecttion.
```

【代码解析】本程序的前半部分与示例代码 17-2 相同，代码第 54 行调用 accept 函数在建立的套接字上接受远程客户机的连接。

17.3.5 请求连接函数 connet

所谓请求连接，指的是客户机需要向服务器发送信息时，需要发送一个连接请求。connect 函数可以完成这项功能，这个函数的使用方法如下所示。

```
01   int connect (int sockfd,struct sockaddr * serv_addr,int addrlen);
```

在参数列表中，sockfd 是已经建立的 socket。serv_addr 是一个结构体指针，指向一个 sockaddr 结构体。这个结构体存储着远程服务器的 IP 与端口信息。addrlen 是 sockaddr 结构体的内存长度，可以用 sizeof 函数来获取。sockaddr 结构体的定义见前面章节中的 bind 函数所述。函数会将本地的 socket 连接到 serv_addr 所指定的服务器 IP 与端口。如果连接成功，返回值为 0，连接失败则返回-1。函数可能发生错误，可以用 error 来捕获发生的错误。

注意：如果需要使用这个函数，需要在程序的最前面包含下面的头文件。

```
01 #include<sys/types.h>
02 #include<sys/socket.h>
```

17.3.6 connet 函数使用实例

本节将讲解一个 connet 函数的实例，在程序中连接到远程服务器。在实例中连接的远程服务器是央视国际网站，域名是 www.cctv.com。可以在终端中输入下面的命令来取得这个域名的 IP 地址。

```
ping www.cctv.com
```

显示的结果如下所示。

```
PING www.cctv.com (202.108.249.216) 56(84) bytes of data.
```

所以央视国际的服务器 IP 地址是 202.108.249.216。网站服务的端口是 80。程序的代码如下所示。

【范例 17-4】使用 connect 函数连接远程服务器，示例代码如下所示。

示例代码 17-4

```
01  #include <stdio.h>
02  #include <netinet/in.h>
03  #include <arpa/inet.h>
04  #include <unistd.h>
05  #include <fcntl.h>
06  #include <sys/stat.h>
07  #include <sys/types.h>
08  #include <sys/socket.h>
09
10  #define PORT 80                                  /*定义一个端口号。*/
11  #define REMOTE_IP "202.108.249.216"              /*定义一个 IP 地址。*/
12
13  int  main(int argc,char *argv[])
14  {
15     int s ;                                       /*定义相关的变量。*/
16     struct sockaddr_in addr ;
```

```
17      char mybuffer[256];
18
19      if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )          /*建立一个 socket。*/
20      {
21        perror("socket");
22        exit(1);
23      }
24      else
25      {
26        printf("socket created .\n");
27        printf("socked id: %d \n",s);
28      }
29
30      bzero(&addr,sizeof(addr));                         /*设置 addr 的信息。*/
31      addr.sin_family =AF_INET;
32      addr.sin_port=htons(PORT);
33      addr.sin_addr.s_addr=inet_addr(REMOTE_IP);
34
35      if(connect(s,&addr,sizeof(addr))<0)                /*连接到远程服务器。*/
36      {
37        perror("connect");
38        exit(1);
39      }
40      else                                               /*输出信息。*/
41      {
42          printf("connected ok!\n");
43          printf("remote ip:%s\n",REMOTE_IP);
44          printf("remote port:%d\n",PORT);
45       }
46  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.22.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明程序已经正常连接到了远程服务器。

```
socket created .
socked id: 3
connected ok!
remote ip:202.108.249.216
remote port:80
```

【代码解析】代码第 36 行调用 connect 函数建立与远程服务器的连接。其他的代码与本章前面的示例代码功能相同。

17.4　数据的发送与接收

建立套接字并完成网络连接以后，可以把信息传送到远程主机上，这个过程就是信息的发送。远程主机发送来的信息，本地主机需要进行接收处理。本节将讲述这种面向连接的套接字信息发送与接收操作。

17.4.1　数据接收函数 recv

函数 recv 可以接收远程主机发送来的数据，并把这些数据保存到一个数组中。该函数的使用方法如下所示。

```
01   int recv(int s,void *buf,int len,unsigned int flags);
```

在参数列表中，s 是已经建立的 socket。buf 是一个指针，指向一个数组。接收到的数据会保存到这个数组上。len 是数组的长度，可以用 sizeof 函数来取得。flags 一般设置为 0，其他可能的赋值与含义如下所示。

- MSG_OOB：接收以 out-of-band 送出的数据。
- MSG_PEEK：返回来的数据并不会在系统内删除，如果再调用 recv 时会返回相同的数据内容。
- MSG_WAITALL：强迫接收到 len 大小的数据后才能返回，除非有错误或信号产生。
- MSG_NOSIGNAL：此操作被 SIGPIPE 信号中断。
- recv 函数如果接收到数据，会把这些数据保存在 buf 指针指向的内存中，然后返回接收到字符的个数。如果发生错误则会返回-1。函数可能发生下面这些错误，可以用 errno 来捕获错误。

提示：使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01 #include <sys/types.h>
02 #include <sys/socket.h>
```

17.4.2 recv 函数使用实例

本节将讲解一个 recv 函数使用实例。在程序中，连接到北京大学的 FTP 服务器，然后用 recv 函数取得 ftp 服务器返回的信息。北京大学的 FTP 服务器域名如下所示。

```
ftp.pkp.edu.cn
```

在终端中输入下面的命令，取得这个域名的 IP 地址。

```
ping ftp.pku.edu.cn
```

终端中显示的结果如下所示。

```
PING vineyard.pku.edu.cn (202.38.97.197) 56(84) bytes of data.
```

所以北京大学 FTP 服务器的 IP 地址是 202.38.97.197。FTP 服务的端口号是 21。

【范例 17-5】使用 recv 函数接收数据，示例代码如下所示。

示例代码 17-5

```
01  #include <stdio.h>
02  #include <netinet/in.h>
03  #include <arpa/inet.h>
04  #include <unistd.h>
05  #include <fcntl.h>
06  #include <sys/stat.h>
07  #include <sys/types.h>
08  #include <sys/socket.h>                          /*包含头文件。*/
09
10  #define PORT 21                                  /*定义一个端口号。*/
11  #define REMOTE_IP "202.38.97.197"                /*定义一个 IP 地址。*/
12
13  int  main(int argc,char *argv[])
14  {
15    int s ;                                        /*定义相关的变量。*/
16    struct sockaddr_in addr ;
17    char mybuffer[256];
18
19    if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )      /*建立一个 socket。*/
```

```
20      {
21        perror("socket");
22        exit(1);
23      }
24      else
25      {
26        printf("socket created .\n");                    /*socket 建立成功。*/
27        printf("socked id: %d \n",s);
28      }
29
30      bzero(&addr,sizeof(addr));                  /*清空 addr 所在的内存。
31    */
32      addr.sin_family =AF_INET;                          /*设置 addr 的成员。*/
33      addr.sin_port=htons(PORT);
34      addr.sin_addr.s_addr=inet_addr(REMOTE_IP);
35
36      if(connect(s,&addr,sizeof(addr))<0)                /*连接远程服务器。*/
37      {
38        perror("connect");
39        exit(1);
40      }
41      else
42      {
43          printf("connected ok!\n");                     /*连接成功。*/
44          printf("remote ip:%s\n",REMOTE_IP);
45          printf("remote port:%d\n",PORT);
46      }
47
48      recv(s ,mybuffer,sizeof(mybuffer),0);              /*接收信息。*/
49      printf("%s\n",mybuffer);                           /*输出接收到的信息。*/
50    }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.23.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明，程序已经正确连接到了北京大学的 FTP 服务器。服务器返回了一段欢迎信息。

```
socket created .
socked id: 3
connected ok!
remote ip:202.38.97.197
remote port:21
220 Welcome to VINEYARD FTP service.
```

【代码解析】代码第 48 行调用 recv 函数接收远程主机发送的信息，其他的代码与本章前面的示例代码类似。

17.4.3　信息发送函数 send

用 connect 函数连接到远程计算机以后，可以用 send 函数将信息发送到对方的计算机。这个函数的使用方法如下所示。

```
01   int send(int s,const void * msg,int len,unsigned int flags);
```

在参数列表中，s 是已经建立的 socket。msg 是需要发送数据的指针。len 是需要发送数据的长度。这个长度可以用 sizeof 函数来取得。参数 flags 一般设置为 0，可能的赋值与含义如下所示。

- MSG_OOB：传送的数据以 out-of-band 的方式送出。

- MSG_DONTROUTE：取消路由表查询。
- MSG_DONTWAIT：设置为不可阻断传输。
- MSG_NOSIGNAL：此传输不可被 SIGPIPE 信号中断。

如果数据成功，函数会返回已经传送的字符个数。否则会返回-1。函数可能发生错误，可以用 errno 来捕获函数的错误。

提示：使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01 #include<sys/types.h>
02 #include<sys/socket.h>
```

17.4.4 数据传输函数 write 与 read

在前面的章节讲述过，write 函数可以向文件中写入数据，read 函数可以从文件中读取数据。socket 建立连接以后，向这个 socket 中写入数据表示向远程主机传送数据。从 socket 中读取数据相当于接受远程主机传送过来的数据。这两个函数的使用方法如下所示。

```
01   ssize_t write(int fd, const void *buf, size_t count);
02   ssize_t read(int fd, void *buf, size_t count);
```

在参数列表中，fd 是已经建立的 socket。buf 是指向一段内存的指针，count 表示 buf 指向内存的长度。read 函数读取字节时，会把读取的内容保存到 buf 指向的内存中，然后返回读取到字节的个数。使用 write 函数传输数据时，会把 buf 指针指向的内存中的数据发送到 socket 连接的远程主机，然后返回实际发送字节的个数。

提示：使用这个函数以前，需要在程序的最前面包含下面的头文件。

```
01 #include <unistd.h>
```

17.4.5 read 函数接收数据实例

本节将讲解一个 read 函数读取数据的实例。在程序中，监听一个端口。如果有客户端连接这个端口则接受这个连接，然后用 read 函数读取远程主机发送的数据。输出这些数据以后结束这个程序。

【范例 17-6】使用 read 函数接收数据，示例代码如下所示。

示例代码 17-6

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
03  #include<netinet/in.h>
04  #include<arpa/inet.h>
05  #include<unistd.h>                                    /*包含头文件。*/
06
07  #define PORT 6677                                     /*定义一个端口号。*/
08
09  main()
10  {
11     int sockfd,newsockfd,fd;                           /*定义相关的变量。*/
12     struct sockaddr_in addr;
13     int addr_len = sizeof(struct sockaddr_in);
14     fd_set myreadfds;
15     char msgbuffer[256];
16     char msg[] ="This is the message from server.Connected.\n";
17
18     if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket。*/
```

```
    {
      perror("socket");
      exit(1);
    }
    else
    {
      printf("socket created .\n");                    /*socket 建立成功。*/
      printf("socked id: %d \n",sockfd);
    }

    bzero(&addr,sizeof(addr));                         /*清空 zero 所在的内存。*/
    addr.sin_family =AF_INET;
    addr.sin_port = htons(PORT);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if(bind(sockfd,&addr,sizeof(addr))<0)              /*绑定 IP 端口。*/
    {
      perror("connect");
      exit(1);
    }
    else
    {
       printf("connected.\n");
       printf("local port:%d\n",PORT)    ;
    }

    if(listen(sockfd,3)<0)                             /*监听一个端口号。*/
    {
      perror("listen");
      exit(1);
    }
    else
    {
      printf("listenning......\n");
    }

    if((newsockfd = accept (sockfd,&addr,&addr_len))<0)    /*接受一个连接。*/
    {
      perror("accept");
    }
    else                                                   /*输出结果。*/
    {
       printf("cnnect from %s\n",inet_ntoa(addr.sin_addr));
       if(read(newsockfd,msgbuffer,sizeof(msgbuffer))<=0)
                                                           /*接收信息。*/
       {
         perror("accept");
       }
       else
       {
          printf("message:\n%s \n",msgbuffer);             /*输出接收到的信息。*/
          }
      }
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 17.24.c
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如下所示。结果表明这个程序正在监听本地计算机的 6677 号端口。

```
socket created .
socked id: 3
connected.
local port:6677
listenning......
```

打开浏览器。在浏览器中输入下面的网址，然后按“Enter”键，使浏览器访问本地计算机的 6677 号端口。

```
http://127.0.0.1:6677/
```

浏览器显示无法打开网页。在终端中显示了下面的代码，这些代码是浏览器向本机的 6677 号端口请求打开网页的数据报。

```
cnnect from 127.0.0.1
message:
GET / HTTP/1.1
Host: 127.0.0.1:6677
User-Agent: Mozilla/5.0 (X11; U; Linux i686; zh-CN; rv:1.8.1.8) Gecko/20071030
Fedora/2.0.0.8-2.fc8 Firefox/2.0.0.8
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,ima
ge/png,*/*
```

【代码解析】本程序中应用到了 socket 函数，bind 函数，listen 函数和 accept 函数等完成数据通信的必要准备工作。然后使用 read 函数读取远程主机发送的信息，最后打印接收到的信息。本程序给出了最简单的网络通信的完整过程，需要仔细体会网络通信的整个流程。

17.5 综合练习

本节将讲述一个面向连接的套接字通信实例。面向连接的网络通信，包括客户端和服务器两个部分的程序。服务器实现监听功能，如果有客户端连接，则接受这个连接，并将用户发送来的信息发回。客户端实现请求连接的功能，可以连接到服务器，向服务器发送信息，并接收服务器发回的信息。

17.5.1 服务器程序

所谓服务器程序，指的是在网络通信时，这个程序始终处于等待状态，可以接受用户的连接请求，并且对用户发送的信息进行处理。本节的实例是面向连接的套接字通信服务器程序。程序的重点是端口的监听和接收发送的信息。

【范例 17-7】编程实现 TCP 服务器程序，示例代码如下所示。

示例代码 17-7

```
01  #include<sys/types.h>
02  #include<sys/socket.h>
03  #include<netinet/in.h>
04  #include<arpa/inet.h>
05  #include<unistd.h>                                  /*包含头文件。*/
06
07  #define PORT 5678                                   /*定义端口号。*/
08  #define MAX 10                                      /*最大连接数。*/
09
10  main()
11  {
12     int sockfd,newsockfd,is_connected[MAX],fd;       /*定义相关的变量。*/
13     struct sockaddr_in addr;
14     int addr_len = sizeof(struct sockaddr_in);
```

```
15    fd_set myreadfds;
16    char msgbuffer[256];
17    char msg[] ="This is the message from server.Connected.\n";
18
19    if ((sockfd = socket(AF_INET,SOCK_STREAM,0))<0)    /*建立一个 socket。*/
20    {
21      perror("socket");
22      exit(1);
23    }
24    else
25    {
26      printf("socket created .\n");                    /*socket 建立成功，输出提示。*/
27      printf("socked id: %d \n",sockfd);
28    }
29
30    bzero(&addr,sizeof(addr));                         /*清空 addr 所在的内存。*/
31    addr.sin_family =AF_INET;
32    addr.sin_port = htons(PORT);
33    addr.sin_addr.s_addr = htonl(INADDR_ANY);
34
35    if(bind(sockfd,&addr,sizeof(addr))<0)              /*绑定端口。*/
36    {
37      perror("connect");
38      exit(1);
39    }
40    else
41    {
42       printf("connected.\n");
43       printf("local port:%d\n",PORT)    ;
44    }
45
46    if(listen(sockfd,3)<0)                             /*开始监听。*/
47    {
48      perror("listen");
49      exit(1);
50    }
51    else
52    {
53      printf("listenning......\n");
54    }
55
56    for(fd=0;fd<MAX;fd++)
57    {
58      is_connected[fd]=0;                              /*设置所有的标记为 0。*/
59    }
60
61    while(1)                                       /*进入一个循环，处理所有的连接。*/
62    {
63      FD_ZERO(&myreadfds);                             /*清空一个标志。*/
64      FD_SET(sockfd,&myreadfds);                       /*设置标志。*/
65
66      for(fd=0;fd<MAX;fd++)
67      {
68        if(is_connected[fd])                           /*判断有没有连接。*/
69        {
70          FD_SET(fd,&myreadfds);                       /*设置标志。*/
71         }
72      }
73
74      if(!select(MAX,&myreadfds,NULL,NULL,NULL))     /*如果到达了最大的连接数则进入下次
75  循环。*/
```

```
76        {
77             continue;
78        }
79
80        for(fd=0;fd<MAX;fd++)                    /*进入一个循环。*/
81        {
82          if(FD_ISSET(fd,&myreadfds))            /*判断标志。*/
83          {
84            if(sockfd==fd)                       /*如果新建的 socket 与 fd 相同。*/
85            {
86              if((newsockfd = accept (sockfd,&addr,&addr_len))<0)
87                                                 /*接受一个连接，新建一个 socket。*/
88              {
89                perror("accept");
90               }
91              write(newsockfd,msg,sizeof(msg));          /*给客户端发送一段信息。*/
92              is_connected[newsockfd] =1;                /*设置标志。*/
93              printf("cnnect from %s\n",inet_ntoa(addr.sin_addr));
94                                                         /*输出客户端的 IP。*/
95            }
96            else
97            {
98              bzero(msgbuffer,sizeof(msgbuffer));        /*清空字符串。*/
99              if(read(fd,msgbuffer,sizeof(msgbuffer))<=0)
100                                                        /*读取结果。*/
101             {
102               printf("connect closed.\n");             /*输出连接关闭。*/
103               is_connected[fd]=0;                      /*设置标志。*/
104               close(fd);                               /*关闭一个 socket。*/
105              }
106              else
107              {
108                write(fd,msgbuffer,sizeof(msgbuffer));
109                                                        /*发送接收到的信息。*/
110                printf("message:%s \n",msgbuffer);
111                                                        /*输出接收到的信息。*/
112              }
113           }
114         }
115       }
116     }
117 }
```

【运行结果】输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 17.25.c -o tcpser
```

程序的运行结果请参看 17.5.3 节“实例程序测试”。

17.5.2 客户端程序

客户端指的是在网络通信时，主动向服务器发送连接请求，主动发送信息的程序。本节将进述面向连接的套接字通信的客户端程序。这个程序的内容是向服务器申请连接，并且发送用户输入的内容。

【范例 17-8】编程实现 TCP 客户端程序，示例代码如下所示。

示例代码 17-8

```
01  #include <stdio.h>
02  #include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>                                    /*包含头文件。*/

#define PORT 5678                                          /*定义远程端口。*/
#define REMOTE_IP "127.0.0.1"                              /*定义远程 IP。*/

int  main(int argc,char *argv[])
{
  int s ;                                                  /*定义相关变量。*/
  struct sockaddr_in addr ;
  char mybuffer[256];

  if( (s=socket(AF_INET,SOCK_STREAM,0))<0 )                /*新建一个 socket。*/
  {
    perror("socket");
    exit(1);
  }
  else
  {
    printf("socket created .\n");
    printf("socked id: %d \n",s);
  }

  bzero(&addr,sizeof(addr));                               /*设置 addr。*/
  addr.sin_family =AF_INET;
  addr.sin_port=htons(PORT);
  addr.sin_addr.s_addr=inet_addr(REMOTE_IP);

  if(connect(s,&addr,sizeof(addr))<0)                      /*连接远程服务器。*/
  {
    perror("connect");
    exit(1);
  }
  else
  {
      printf("connected ok!\n");
      printf("remote ip:%s\n",REMOTE_IP);
      printf("remote port:%d\n",PORT);
   }

  recv(s ,mybuffer,sizeof(mybuffer),0);     /*接收服务器发送的信息。
*/
  printf("%s\n",mybuffer);                                 /*输出信息。*/

  while(1)                                                 /*进入一个循环。*/
  {
    bzero(mybuffer,sizeof(mybuffer));                      /*清空 mybuffer 字符串。*/
    read(STDIN_FILENO,mybuffer,sizeof(mybuffer));          /*读取输入。*/

    if(send(s,mybuffer,sizeof(mybuffer),0)<0)              /*发送信息。*/
    {
      perror("send");                                      /*错误处理。*/
      exit(1);
    }
    else
    {
      bzero(mybuffer,sizeof(mybuffer));                    /*清空 mybuffer 内存。*/
      recv(s ,mybuffer,sizeof(mybuffer),0);                /*接收信息。*/
```

```
65          printf("received:%s\n",mybuffer);              /*输出信息。*/
66        }
67      }
68  }
```

【运行结果】输入下面的命令，编译这个程序。需要加-o 参数指定一个输出文件名。

```
gcc 17.26.c -o tcpcli
```

程序的运行结果请参看 17.5.3 节“实例程序测试”。

17.5.3 实例程序测试

本节将对面向连接套接字通信的两个实例程序进行测试。在运行这两个程序时，需要先打开服务程序，使相应的端口处于监听状态。然后打开客户端程序，请求连接服务器的端口。

（1）输入下面的命令，运行服务器程序。

```
./tcpser
```

（2）程序的运行结果如下所示。

```
socket created .
socked id: 3
connected.
local port:5678
listenning......
```

（3）结果显示，已经建立 socket，本地端口是 5678。程序正在这个端口上进行监听。

（4）这时打开另一个终端，在终端中输入下面的命令，打开客户端程序。

```
./tcpcli
```

（5）程序显示的结果如下所示。结果显示，程序已经建立了 socket 连接到了 127.0.0.1 这个 IP 上的 5678 号端口。服务器发送回一条信息。

```
socket created .
socked id: 3
connected ok!
remote ip:127.0.0.1
remote port:5678
This is the message from server.Connected.
```

（6）在服务器程序的终端中查看，显示的结果如下所示。

```
cnnect from 127.0.0.1
```

（7）结果表示服务器已经接受了客户端的连接请求。

（8）这时进入客户程序的终端，输入“hello”，然后按“Enter”键。显示的结果如下所示。

```
received:hello
```

（9）结果表明，程序把输入的字符发送到了服务器。服务器又返回了这个字符串。

（10）进入到服务器的终端，可以查看到客户端发送的信息如下所示。

```
message:hello
```

（11）这两个程序进入了 while 循环以后，会始终执行程序无法退出。可以按“Ctrl”+“C”组合键结束程序。

17.6 小结

本节讲述了面向连接的套接字网络通信，这种信息传输方式是网络中最常用、最重要的信息传输方式。本节的知识重点是端口监听、远程连接、接受连接、接收和发送信息四个操作。最后一节的程序实例演示了面向连接的信息传输过程，通过这个实例的学习，可以学习本节中

所讲述的套接字连接的综合操作。这个实例是网络传输的最基本的原理和形式，用这种方法可以实现不同计算机之间的文本、文件等类型数据的传输。

17.7　习题

一、填空题

1．_____指的是将套接字与指定的端口相连。用 socket 函数进行起一个套接字以后，需要用 bind 函数在这个套接字上面绑定一个端口。

2．所谓_____，指的是 socket 的端口处于等待状态，如果有客户端有连接请求，这个端口会接受这个连接。

3．函数_____可以将一个端口绑定到一个已经建立的 socket 上，_____函数用实于实现监听等待功能。

4．函数_____可以接收远程主机发送来的数据，并把这些数据保存到一个数组中。用 connect 函数连接到远程计算机以后，可以用_____函数将信息发送到对方的计算机。

二、选择题

1．accept 函数的功能是（　　）。

A．接受连接请求　　B．监听端口

C．发起到服务器的连接　　D．接收客户端发送的数据

2．下面关于 connect 函数的参数描述中错误的是（　　）。

A．sockfd 是已经建立的 socket

B．serv_addr 是一个结构体指针，指向一个 sockaddr 结构体。这个结构体存储着远程服务器的 IP 与端口信息

C．addrlen 是 sockaddr 结构体的内存长度，可以用 sizeof 函数来获取

D．buf 是一个指针，指向一个数组。接收到的数据会保存到这个数组上

3．下面关于面向连接的套接字编程中使用的函数的功能描述错误的是（　　）。

A．listen 函数的功能是等待监听

B．accept 函数的功能是接受连接请求

C．connect 函数的功能是客户机需要向服务器发送信息时，发送一个连接请求

D．send 函数的功能是用 connect 函数连接到远程计算机以后，用来将信息发送到对方的计算机

三、简答题

1．简述面向连接的套接字通信工作流程。

2．在进行面向连接的套接字编程中有哪些必须使用的函数？如何使用它们？

四、编程题

1．利用 read 函数编写读取读取客户端数据（提示：在程序中，首先监听一个端口，如果有客户端连接这个端口则接受这个连接，然后用 read 函数读取远程主机发送的数据。输出这些数据以后结束这个程序）。

2．编程实现面向连接的套接字通信服务器程序（提示：重点是端口的监听和接收发送的信息。需要调用的函数有：bind，listen，accept，recv，send 等）。

第七篇　图形界面开发篇

第 18 章　Linux 图形界面开发基础

本章介绍了 Linux 下的图形用户界面 X Window 系统的历史和体系结构，Linux 下的桌面环境 KDE 和 GNOME，以及如何搭建 Linux 下的图形界面程序开发环境。本章的主要内容如下：

- X Window 系统简介；
- Linux 桌面环境 KDE 与 GNOME；
- Linux 下的图形程序开发环境；
- Linux 图形界面开发常见问题。

18.1　X Window 系统简介

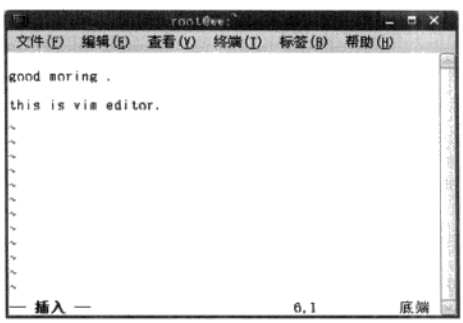
所谓 X Window，指的是 Linux 系统中独立于系统内核之外的用户界面系统。X Window 有时也被称为 X 或 X11。借助于 X Window，用户可以方便地同系统进行交互。系统可以展示给用户非常友好的界面效果。

18.1.1　X Window 的历史

Linux 系统中的图形桌面环境使用的是 X Window 系统。这种图形桌面最早来源于 1984 年在麻省理工学院（MIT）与 DEC 公司的研发的“X Window System”项目。这种技术的作用，是建立一个分散的、与硬件无关的、网络透明的图形环境。借助于 X Window，Linux 就不再是枯燥的图形界面，而是有着美观方便的图形界面的窗口程序。

现在所有的 Linux 系统发行版，都有着功能强大的 X Window 桌面。借助于 X Window，不需要记忆多少命令的情况下也可以方便地使用 Linux。如图 18-1 所示，是 vim 的工作界面。在 vim 中，需要输入各种命令才能完成文本的编辑操作。

如图 18-2 所示是 gedit 图形界面的文本编辑器。在这个软件中，有非常友好的界面，所有的操作都不需要输入命令，可以通过菜单、按钮等工具用鼠标的交互来完成操作。


root@we:~
文件(F) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
good moring .
this is vim editor.
~
~
~

18.1.2　X Window 体系结构

与 Windows 系统的桌面不同的是，X Window 是基于网络的图形系统。X 桌面下的任何程序都是 X 桌面的客户端。客户端的显示是以网络协议与服务器进行交互的。X Window 系统由服务器、用户端、网络协议三部分组成。

- X 服务器：X 服务器是控制显示器和输入设备的软件。服务器可以创建图形窗口，在图形界面中显示图形和文字，回应客户端程序的请求。Linux 系统启动以后，会启动 X11 服务器，提供给用户图形界面的显示。

提示：X 服务器控制着显示设备的工作，它包括所有显示设备的驱动程序。

- X 客户端：X 客户端是使用系统视窗功能的一些应用程序。在 X 下的应用程序称做 Client，原因是客户端是 X Server 的客户端程序，向 X Server 请求显示内容，显示 X Server 回应显示的内容。即使是在同一台计算机上登录 Linux 和客户端，本地计算机的客户端也登录到了 X Server 上面。多台计算机可以登录到同一个 X 服务器上。
- X 协议：X 协议是 X 客户程序和服务器进行通信的一套协定。X 协议支持网络，能在本地系统中和网络实现协议通信。X 协议支持的网络协议有 TCP/IP、DECnet 等。

X 服务器与 X 客户机、X 协议的关系，可用图 18-3 来表示。

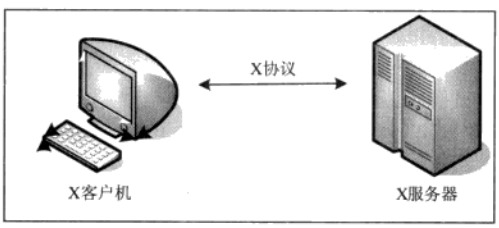


图 18-3　X 服务器与 X 客户端的关系

18.2　Linux 桌面环境

所谓 Linux 桌面环境，指的是 X 桌面的客户端。用户计算机通过 X 客户端与 X 服务器进行交互，显示出各种图形界面。最常用的两种 X 客户端是 KDE 和 GNOME。本节将对这两个桌面环境进行简单介绍。

18.2.1　KDE

所谓 KDE，指的是 K 桌面环境（K Desktop Environment），是一种运行于 UNIX及 Linux、FreeBSD等类 UNIX操作系统上面的自由图形工作环境。KDE 是一个综合的桌面环境，建立在 XFree86 和 QT 的基础上，提供了窗口管理器和许多实用工具。这些工具包括浏览器、文字处理软件、电子表格程序、演示文稿程序、游戏和大量附件工具。借助于这些工具，可以方便地使用 Linux 桌面环境。

许多 Linux 发行版默认都安装和使用 KDE 桌面环境。这种桌面环境有着与 Windows 非常相似的文件管理器和开始菜单、任务栏等工具。Gwenview、Kaffeine、Kate、Kopete、KOffice、Kontact 这些软件提供了功能强大的桌面应用功能。如图 18-4 所示，是一个 KDE 桌面的截图。

使用桌面上的工具可以方便地进行各种操作。

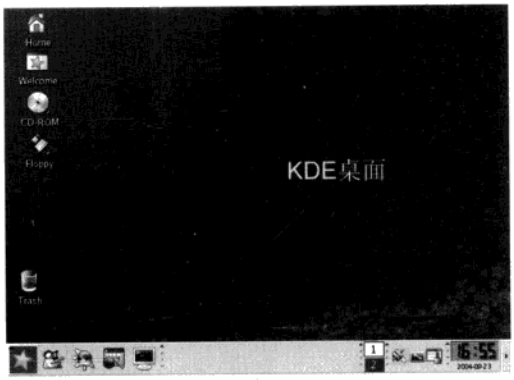


图 18-4　KDE 桌面截图

18.2.2　GNOME

GNOME 是不同于 KDE 的另一种 Linux 桌面环境，提供了一个功能强大的 Linux 用户桌面。这种桌面以实用性和界面友好著称，有下面这些特点。

- 友好性：设计和开发所有人都可以使用的环境，界面操作非常简单。只需要简单的鼠标交互即可完成大部分 Linux 操作。
- 国际化：桌面和程序支持很多种语言，对中文的支持非常好。

如图 18-5 所示，是 Red Hat Linux 9.0 的 GNOME 桌面截图。在 GNOME 桌面中，有着丰富的应用软件。这些应用软件可以替代 Windows 系统中的各种软件完成计算机的桌面操作。

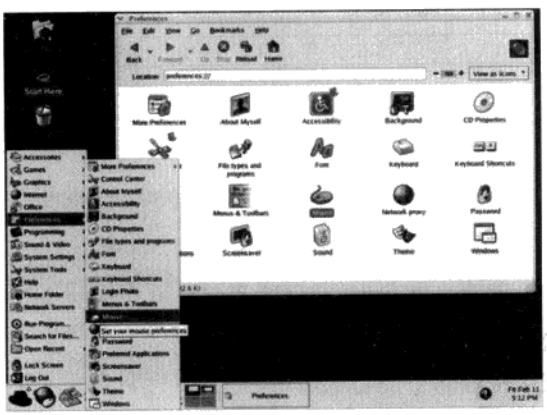


图 18-5　GNOME 桌面截图

GNOME 桌面系统使用 C 语言编程，对 C 语言的支持非常好。本书中使用的 Red Hat 编程使用的就是 GNOME 桌面。对其他的开发语言，如 C++、Java、Ruby、C#、Python、Perl 也提供了很好的支持。很多 Linux 软件项目都是在 GNOME 的环境下开发的。

提示：不同的 Linux 发行版采用的默认桌面系统可能不同，例如 Red Hat 同时支持 KDE 和 GNOME 桌面环境，但是把 GNOME 作为默认的桌面环境，用户在登录界面中可以选择使用任一种桌面环境。

18.3　Linux 下的图形程序开发环境

所谓图形程序开发环境，指的是开发环境和编译器中提供的图形界面库和函数的支持。Linux 系统下面常用的图形界面开发环境有 QT 和 GTK 两种。本书使用的是 GTK 图形库。本节将讲解 GTK 图形库的安装和使用。

18.3.1　GTK+简介

GTK（GIMP Toolkit）是一套跨多种平台的图形工具包，完全按照 LGPL 许可协议发布。最初的 GTK 是为 GIMP 写的，但目前已发展为一个功能强大、设计灵活的一个通用图形库。GTK 使用 GTK 作为图形界面开发环境，很多图形界面的程序使用 GTK 作为图形库。现在使用的 GTK 版本是 GTK+。GTK 是用 C 语言写的，对 C 语言有很好的支持。扩展库也可能支持 C++、Guile、Perl、Python、TOM、Ada95、Objective C、Free Pascal 等各种语言。

使用 GTK 库编写的图形界面程序，必须使用的 GTK 库才能编译。现在已经开发出非常成熟的 Windows 系统下的 GTK 支持环境。所有使用 GTK 库开发出的应用程序也可以在 Windows 系统下编译运行。

18.3.2　yum 安装 GTK

如果系统中没有安装 GTK 开发环境，需要自己安装 GTK 库。yum 是 Red Hat（Red hat Linux 9.0 中不支持 yum）或 Fedora 系统中的软件包管理工具。使用 yum 可以非常方便地安装 GTK 库，下面以 Fedora8 为例介绍使用 yum 安装 GTK，安装过程如下所示。

注意：Red Hat 9.0 默认情况下不支持 yum，所以推荐使用 Fedora 完成下面的操作。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入下面的命令，安装 gtk+。

```
yum install gtk+
```

③ yum 软件管理程序从网络的软件库中搜索需要安装的软件，然后显示出软件的信息。终端中显示的结果如下所示。

```
Setting up Install Process
Parsing package install arguments
Resolving Dependencies
--> Running transaction check
---> Package gtk+.i386 1:1.2.10-59.fc8 set to be updated
--> Processing Dependency: libgmodule-1.2.so.0 for package: gtk+
--> Processing Dependency: libglib-1.2.so.0 for package: gtk+
--> Running transaction check
---> Package glib.i386 1:1.2.10-28.fc8 set to be updated
--> Finished Dependency Resolution
Dependencies Resolved
================================================================================
 Package              Arch      Version         Repository       Size
================================================================================
Installing:
 gtk+                 i386      1:1.2.10-59.fc8  fedora          911 k
Installing for dependencies:
 glib                 i386      1:1.2.10-28.fc8  fedora          137 k
Transaction Summary
```

```
================================================================================
Install      2 Package(s)
Update       0 Package(s)
Remove       0 Package(s)
Total download size: 1.0 M
Is this ok [y/N]:
```

④ 显示的结果表明，安装 GTK 库需要下载两个文件，共 1.0MB 的内容。在提示是否安装后面输入“y”，然后按“Enter”键。这时系统会自动从服务器下载文件，然后完成 GTK 库的安装。

⑤ 安装 gtk2-devel 软件包，在终端中输入下面的命令。

```
yum install gtk2-devel
```

屏幕上的显示结果与安装方法和安装 gtk+的类似。

提示：使用 yum 可以解决软件包之间复杂的依赖关系，会自动安装依赖的软件包，还可以自动从互联网上下载所需要的软件包。

18.3.3 用软件包管理器安装 GTK

系统提供了图形界面的软件包管理器。借助于软件包管理器，可以方便地选择一个软件，对安装进行各种设置以后安装这个软件。本节将讲解使用软件包管理器安装 GTK 库，以 Fedora 为例介绍使用包管理器安装 GTK，操作步骤如下所示。

① 从主菜单中打开一个终端。

② 在终端输入下面命令，打开软件包管理器。

```
system-config-packages
```

③ 软件包管理器的界面如图 18-6 所示。

④ 在软件包管理器中，单击选择“搜索”标签。在文本框中输入需要查找的软件包“gtk+”。然后单击“搜索”按钮搜索这个软件包。

⑤ 如图 18-7 所示，在搜索的结果列表中，右键单击“gtk+”软件包，然后选择“选择”命令，选择安装这个软件包。

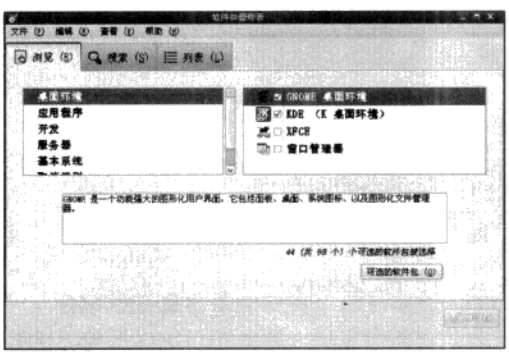


图 18-6 软件包管理器

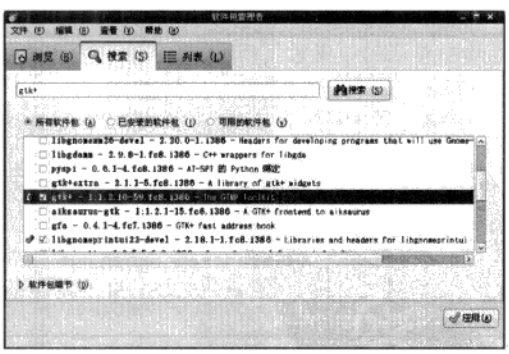


图 18-7 选择需要安装的软件

⑥ 单击“文件”|“应用”菜单命令。软件包管理器会自动从软件服务器下载软件，然后完成软件的安装。

⑦ 安装完毕以后，单击“文件”|“退出”命令，退出软件包管理器。

18.4　常见问题

在进行图形界面编程时，开发环境的配置非常重要。gtk 库并不是系统的标准库，有些 Linux 发行版中可能没有这个库。若需要编写图形界面的程序，需要正确安装这个库。

18.4.1　找不到 gtk.h 包含文件的错误

在上一节中编写的图形界面程序的第一行，有下面的包含语句包含系统中的 gtk.h 头文件。

```
01  #include <gtk/gtk.h>
```

包含文件的作用是定义程序中需要使用的界面编程相关的函数与变量。如果系统中没有安装 gtk 库，则程序在编译时会发生下面的错误。

```
gtk/gtk.h:没有那个文件和目录
```

如果发生这个错误，则表明系统中没有安装 gtk 库。这时需要用本章中前面章节中 gtk 库安装的方法安装 gtk 库。

提示：在 Fedora 中安装 gtk 库的命令是“ yum install gtk2-devel”。

18.4.2　pkg-config 参数有什么作用

在 18.4.2 节中，编译程序的命令如下所示。

```
gcc 18.1.c `pkg-config --cflags --libs gtk+-2.0`
```

在 gcc 的参数中，后面用反引号引用了这样的参数。

```
pkg-config --cflags --libs gtk+-2.0
```

反引号的作用是执行反引号内部的命令，然后返回执行的结果到 gcc 的编译参数中。这样 pkg-config 的运行结果就是编译程序的参数。pkg-config 的作用是警告系统中库的信息，这些信息包含各种库的版本、编译和连接需要的参数等。--cflags 表示库的编译环境是 C 语言。--libs gtk+-2.0 需要查找的库是 gtk+，版本是 2.0。从 pkg-config 的作用可知，要编译这个程序，需要可以正常运行 pkg-config 命令。如果系统中没有 pkg-config 软件，需要安装这个软件以后才能进行软件编译。

18.5　综合练习

本节将编写一个简单的图形界面程序。在程序中，需要使用 gtk 库。编译以后的程序在运行时，会打开一个窗口。图形界面程序的编写步骤如后面几个章节所述。

18.5.1　编写程序代码

用编写简单的 C 程序一样，图形界面程序的内容也是通过编写 C 程序代码来完成的。在程序中，需要调用 gtk 库中的图形函数，所以需要在程序中包含这个头文件。本节将使用图形界面下的 gedit 编写代码。

【范例 18-1】编写一个简单的图形界面程序，示例代码如下所示。

示例代码 18-1

```
01  #include <gtk/gtk.h>                          /*包含 gtk 头文件。*/
```

```
02
03  int main (int argc, char *argv[])
04  {
05    GtkWidget *window;                                    /*定义一个指向窗口的指针。*/
06    char title[]="Hello ,World.";                         /*定义标题。*/
07    gtk_set_locale ();
08    gtk_init (&argc, &argv);                              /*建立窗口。*/
09    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
10    gtk_window_set_title (GTK_WINDOW (window),title);     /*设置标题。*/
11    gtk_widget_show(window);                              /*显示窗口。*/
12    gtk_main ();                                          /*显示。*/
13    return 0;
14  }
```

① gedit 的工作界面如图 18-8 所示。

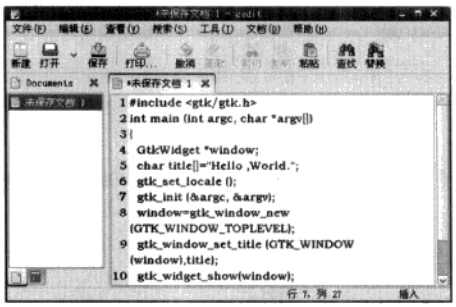


图 18-8　在图形界面下编写程序

② 单击“文件”|“另存为”菜单命令，保存这个文件。

③ 在“另存为”对话框中，“名称”文本框中输入需要保存的文件名“18.1.c”。单击左边目录列表中的目录“root”，选择把这个文件保存在用户的根目录中。

④ 单击“保存”按钮保存这个文件。

18.5.2　编译程序

编译含有图形界面的程序时，需要在编译命令加入图形库的参数。输入下面的命令编译这个程序。

```
gcc 18.1.c `pkg-config --cflags --libs gtk+-2.0`
```

这个命令正确地完成了程序的编译。

注意：命令中的引号是反引号，即数字键“1”左边的那个键。如果写成单引号，是不能编译的。

`pkg-config --cflags --libs gtk+-2.0`的作用是返回系统中的图形库参数。在终端中输入下面的命令。

```
pkg-config --cflags --libs gtk+-2.0
```

程序的运行结果如下所示。在编译命令中，这些结果会返回到编译命令中作为参数。在编译时会在程序中加入这些图形模块。

```
-I/usr/include/gtk-2.0
-I/usr/lib/gtk-2.0/include
-I/usr/include/atk-1.0
```

```
-I/usr/include/cairo
-I/usr/include/pango-1.0
-I/usr/include/glib-2.0
-I/usr/lib/glib-2.0/include
-lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0
-lpangocairo-1.0 -lpango-1.0 -lcairo -lgobject-2.0
-lgmodule-2.0 -ldl -lglib-2.0
```

18.5.3　运行程序

本节将运行前两节编写和编译的图形界面程序。在运行程序时，需要进入终端，从终端中输入命令启动这个程序。

① 单击“主菜单”|“系统工具”|“终端”命令，打开一个终端。

② 在终端中输入下面的命令，运行这个程序。

```
./a.out
```

注意：必须在 Xwindow 下的终端中输入上面的运行命令（不能在纯文本界面下输入运行命令），否则无法显示图形。

③ 程序的运行界面如图 18-9 所示。程序显示了一个界面，界面中没有内容。程序的标题是“Hello ,World.”。

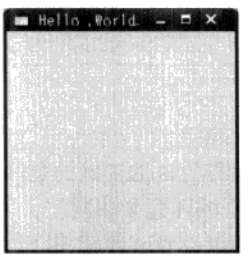


图 18-9　简单的图形界面程序

④ 在终端中按“Ctrl”+“C”组合键关闭这个窗口。

⑤ 也可以单击程序窗口右上角的“关闭”工具 来关闭这个程序。

18.6　小结

本节简要介绍了 Linux 系统的桌面环境和图形界面程序的编程步骤。Linux 系统有 GNOME 和 KDE 两种常用的桌面环境，桌面环境为 Linux 的使用带来了很大的便利。本章讲述了一个简单的图形界面程序的实例。图形界面的使用可以大大增强程序的交互功能。C 语言可以通过调用 GTK 库来编写各种复杂图形界面的程序。图形界面的编程中，需要注意程序编译的参数问题。

18.7　习题

一、填空题

1．所谓 Linux 桌面环境，指的是 X 桌面的_____。用户计算机通过 X 客户端与 X 服务

器进行交互，显示出各种图形界面。

2．Linux 下使用最多的两种 X 客户端是_____和_____。

3．_____作为一套跨多种平台的图形工具包，完全按照 LGPL 许可协议发布。最初的 GTK 是为 GIMP 写的，但目前已发展为一个功能强大、设计灵活的一个通用图形库。

4．所谓 X Window，指的是_____。X Window 有时也被称为 X 或 X11。

二、选择题

1．下面哪一项不是 X Window 系统的组成部分（　　）。

A．X 服务器

B．X 用户端

C．网络协议

D．X 驱动程序

2．下面关于 X Window 系统的说法中正确的是（　　）。

A．X 服务器控制着显示设备的工作，它包括所有显示设备的驱动程序

B．X 用户端控制着显示设备的工作，它包括所有显示设备的驱动程序

C．X Window 系统只能基于本地计算机工作，不支持 TCP/IP 协议

D．KDE 和 GNOME 中包含了所有的硬件驱动程序

3．Linux 中使用最广泛的桌面系统是（　　）。

A．Fvwm95 和 GNOME

B．KDE 和 GNOME

C．AfterStep 和 Fvwm95

D．AfterStep 和 KDE

4．关于 GTK 应用程序编译与运行的说法正确的是（　　）。

A．与基于文本界面的 C 程序编译完全相同

B．在编译时需要使用命令 pkg-config 传递相应的编译选项，否则将导致编译失败

C．GTK 应用程序必须在图形界面下编译

D．编译后 GTK 应用程序可以在纯文本界面下运行（不启动 X Window 系统）

三、简答题

1．简述 Linux 下的 X Window 体系结构的组成部分及其相应的功能。

2．Linux 下最常用的窗口管理系统是什么？简述它们各自的特点。

第 19 章　图形界面编程

本章介绍了图形界面应用程序开发中的最基本图形界面元素窗口、标签、按钮及文本框的基本概念和使用方法。本章的主要内容如下：

- 窗口；
- 标签；
- 按钮；
- 文本框。

19.1　窗口

所谓窗口，指的是一个应用程序的界面框架。程序的所有显示内容和交互都在这一个窗口中实现。在设置程序的界面时，需要先建立一个窗口。本节将讲述程序窗口的建立与设置。

19.1.1　最简单的窗口程序

本节将建立一个最简单的图形界面程序，用这个图形界面程序来讲述窗口的建立过程。程序的代码如下所示。

【范例 19-1】编写一个简单的图形界面程序，显示一个窗口，示例代码如下所示。

示例代码 19-1

```
01  #include <gtk/gtk.h>                                    /*包含头文件。*/
02
03  int main (int argc, char *argv[])
04  {
05    GtkWidget *window;                                    /*定义一个指向窗口的指针。*/
06    gtk_init (&argc, &argv);
07    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
08    gtk_widget_show(window);
09    gtk_main ();
10    return 0;
11  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.1.c `pkg-config --cflags --libs gtk+-2.0`
```

注意：命令行中输入的引号是反单引号，pkg-config 指令的输出信息被传递给编译器。

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行结果如图 19-1 所示。在这个界面中，没有设置标题，窗口的标题是这个程序的名称。

【代码解析】代码第 01 行引入必需的头文件，代码第 06 行调用函数 gtk_init 完成初始化工作，代码第 07 行创建一个新窗口，代码第 08 行显示建立的窗口，代码第 09 行为 gtk 程序的主循环。代码的详细解释请参考下一节。

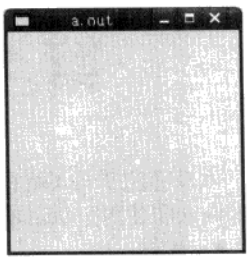


图 19-1　最简单的图形界面

19.1.2　窗口的构建过程

从上一节的程序可知，在 C 程序中建立一个窗口，编写的程序也是一个普通的 C 程序。只是在程序中调用了 gtk 库，使用了图形界面相关的数据类型。要建立一个图形界面的程序，需要在一个 C 程序中完成下面这些操作。

① 在 C 程序中，添加 gtk.h 头文件。代码如下所示。

```
01  #include <gtk/gtk.h>
```

② 在程序的最前面，需要声明一个窗口变量的指针，代码如下所示。

```
01  GtkWidget *window;
```

③ 需要用 gtk_init 函数建立一个 gtk 图形界面。主函数的参数可以带入这个函数中，作为新建窗口的参数。代码如下所示。

```
01  gtk_init (&argc, &argv);
```

④ 用 gtk_window_new 函数新建窗口，将返回的指针赋值已经定义的窗口指针。代码如下所示。

```
01  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

⑤ 用 gtk_widget_show 设置这个窗口的显示状态，代码如下所示。

```
01  gtk_widget_show(window);
```

⑥ 绘制图形界面。经过上面这些步骤设置图形界面以后，这个界面并没有显示。需要用下面的代码来绘制界面，这时窗口才会显示。

```
01  gtk_main ();
```

注意：在编写 GTK 程序时，必须包含头文件 gtk.h，否则将导致编译失败。

19.1.3　新建窗口函数 gtk_window_new

gtk_window_new 函数可以建立一个 GTK 窗口。这个函数的使用方法如下所示。

```
01  GtkWidget* gtk_window_new (GtkWindowType type);
```

在参数列表中，type 是一个表示窗口状态的常量，可能有下面这两种取值。

- GTK_WINDOW_TOPLEVEL：表示这个窗口是正常的窗口。窗口可以最小化，最小化以后，在窗口管理器中可以看到这个窗口的按钮。窗口管理器就是 Windows 系统的任务栏。
- GTK_WINDOW_POPUP：表示这个窗口是一个弹出式窗口，不可以最小化。但这个窗口是一个独立运行的程序，并不是一个对话框。

返回值是一个 GtkWidget 类型的指针。图形界面的所有元素都会返回这个指针。GtkWidget 结构体的定义方式如下所示。

```
01  typedef struct
02  {
```

```
    GtkStyle *style;
      /*元件的风格。*/
    GtkRequisition requisition;
      /*元件的位置。*/
    GtkAllocation allocation;
      /*元件允许使用的空间。*/
    GdkWindow *window;
      /*元件所在的窗口或父窗口。*/
    GtkWidget *parent;
      /*元件的父窗口。*/
  }
  GtkWidget;
```

创建一个窗口以后，这个窗口并不会马上显示出来。需要调用 gtk_widget_show 函数来显示这个窗口。这个函数的使用方法如下所示。

```
void  gtk_widget_show  (GtkWidget *widget);
```

参数 widget 是一个 GtkWidget 类型的结构体指针，表示需要显示这一个元件。在程序中，新建的元件都需要调用这个函数进行显示。

完成窗口的构建以后，需要调用 gtk_main 函数来实现绘图和显示。窗口显示以后，会一直停留在这个窗口，等待用户交互事件的发生。

注意：完成窗口的创建和设置其显示属性后，窗口不会立即显示，调用 gtk_main 函数后窗口才会显示。

19.1.4　设置窗口的标题

函数可以设置一个窗口的标题，这个函数的使用方法如下所示。

```
void gtk_window_set_title (GtkWindow *window, gchar *title);
```

在参数列表中，window 是表示一个窗口的指针。title 是需要设置的标题。下面的实例使用这个函数设置窗口的标题。

【范例 19-2】使用 gtk_window_set_title 函数设置窗口的标题，示例代码如下所示。

示例代码 19-2

```
#include <gtk/gtk.h>

int main (int argc, char *argv[])
{
  GtkWidget *window;
  char title[]="My first Window.";
  gtk_init (&argc, &argv);
  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window),title); /*这个函数的作用是设置标题。*/
  gtk_widget_show(window);
  gtk_main ();
  return 0;
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.2.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-2 所示。窗口的标题已经改变。

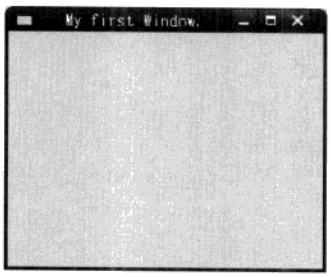


图 19-2　设置窗口的标题

【代码解析】本程序与示例代码 19-1 的不同地方是，在代码第 09 行调用 gtk_window_set_title 函数设置窗口的标题。

19.1.5　设置窗口的大小与位置

窗口的大小指的是窗口的宽度和高度。可以用 gtk_widget_set_usize 函数来设置一个窗口的大小。窗口的位置指的是窗口的左上顶点到屏幕左边和顶边的边距。可以用 widget_set_uposition 函数来设置窗口的位置。这两个函数的使用方法如下所示。

```
01  void gtk_widget_set_usize(GtkWidget *widget,gint width,gint height);
02  void gtk_widget_set_uposition (GtkWidget *widget,gint x,gint y);
```

在参数列表中，widget 表示一个窗口的指针。width 表示需要设置窗口的宽度。height 表示这个窗口的高度。*x* 表示窗口的左边距，也就是窗口左上顶点的 *x* 坐标。*y* 表示窗口的顶边距，也就是窗口的 *y* 坐标。下面的实例用这两个函数对窗口的大小与位置进行设置。使用这两个函数时，需要注意函数参数的含义。

【范例 19-3】使用 gtk_widget_set_usize 函数与 gtk_widget_set_uposition 函数设置窗口大小与位置，示例代码如下所示。

示例代码 19-3

```
01  #include <gtk/gtk.h>
02
03  int main (int argc, char *argv[])
04  {
05    GtkWidget *window;
06    char title[]="set size , set position.";
07    gtk_init (&argc, &argv);
08    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
09    gtk_window_set_title (GTK_WINDOW (window),title);
10    gtk_widget_set_usize(GTK_WINDOW (window),400,200);      /*这个函数的作用是设
置窗口的大
11  小。*/
12    gtk_widget_set_uposition (GTK_WINDOW (window),200,200);/*设置窗口的位置。*/
13    gtk_widget_show(window);
14    gtk_main ();
15    return 0;
16  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.3.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-3 所示。

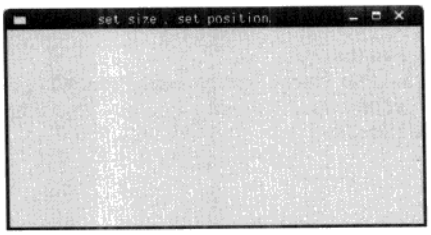


图 19-3　设置窗口的大小与位置

【代码解析】本程序与示例代码 19-2 的不同地方是，在代码第 10 行和代码第 12 行调用 gtk_widget_set_usize 函数和 gtk_widget_set_uposition 函数设置窗口的大小与显示位置。

19.2　标签

所谓标签，指的是程序中的一个文本。这个文本可以显示一定的信息，但是用户不能改变和输入标签所显示的文本内容。程序中的提示信息或显示内容都是通过标签来实现的。

19.2.1　建立一个标签

在窗口中使用标签以前，需要创建一个标签。函数 gtk_label_new 可以新建一个标签，这个函数的使用方法如下所示。

```
01  GtkWidget* gtk_label_new(gchar *str);
```

参数列表中，str 是标签中需要显示的内容。函数会建立一个标签，然后返回一个 GtkWidget 类型的指针。如果标签创建失败，会返回一个 NULL 指针。建立标签以后，需要用 gtk_widget_show 函数来显示这个标签。

19.2.2　将标签添加到窗口中

在 gtk 窗口中，除了 window 窗口以外，其他的任何元件都必须放置在一个容器中。上一节所建立的标签，并不能直接显示，而需要放在一个窗口元件中。gtk_container_add 函数的作用，是把一个元件放置在另一个元件窗口中。这个函数的使用方法如下所示。

```
01  void gtk_container_add (GtkContainer *container, GtkWidget *widget);
```

参数列表中，container 是一个父级容器指针。widget 是需要放置元件的指针。下面的实例将一个标签放置到一个窗口中，然后显示这个窗口和标签。

【范例 19-4】标签的使用举例，示例代码如下所示。

示例代码 19-4

```
01  #include <gtk/gtk.h>
02
03  int main (int argc, char *argv[])
04  {
05    GtkWidget *window;
06    GtkWidget *label;                          /*定义一个指针标签的指针。*/
07    char title[]="label example.";
08    char txt[]="hello,Linux .";
```

```
09    gtk_init (&argc, &argv);
10    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
11    gtk_window_set_title (GTK_WINDOW (window),title);
12    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
13
14    label=gtk_label_new(txt);                          /*新建一个标签。*/
15    gtk_container_add(GTK_CONTAINER(window),label);    /*标签添加到窗口上。*/
16    gtk_widget_show(window);
17    gtk_widget_show(label);                            /*显示标签。*/
18    gtk_main ();
19    return 0;
20  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.4.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-4 所示。在窗口中，用一个标签显示了一个文本。

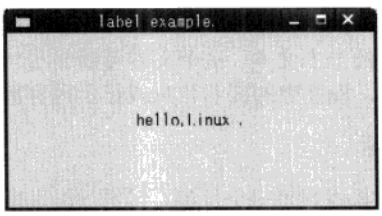


图 19-4　窗口中的标签

【代码解析】代码第 14 行创建一个标签实例，代码第 15 行将新创建的标签添加到窗口中。

注意：因为没有将元件加入到容器中，所以被加到窗口中的元件将占满整个窗口。

19.2.3　设置与获取标签的文本

在程序中，可以用 gtk_label_get_text 函数来获取一个标签的文本，或者用 gtk_label_set_text 函数来设置一个标签的文本。这两个函数的使用方法如下所示。

```
01  const gchar* gtk_label_get_text(GtkLabel *label);
02  void gtk_label_set_text(GtkLabel *label, gchar *str);
```

在参数列表中，label 是表示一个标签的指针。gtk_label_set_text 函数参数中的 str 是需要设置标签的文本。gtk_label_get_text 函数会取得标签的文本然后返回一个字符串。

下面的实例，是在上一节程序的基础上，用 gtk_label_get_text 函数取得标签的文本。然后在返回的文本后面连接一个字符串，用 gtk_label_set_text 函数设置这个标签的文本。程序需要使用字符串的复制与连接函数。

【范例 19-5】使用 gtk_label_get_text 函数和 gtk_label_set_text 函数获取与设置标签文本，示例代码如下所示。

示例代码 19-5

```
01  #include <gtk/gtk.h>
02  #include <stdlib.h>
03
```

```
04  int main (int argc, char *argv[])
05  {
06    GtkWidget *window;
07    GtkWidget *label;
08    char title[]="label example.";
09    char txt[]="hello,Linux .";
10    char txt2[50];
11    gtk_init (&argc, &argv);
12
13    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);              /*建立和处理窗口。*/
14    gtk_window_set_title (GTK_WINDOW (window),title);
15    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
16
17    label=gtk_label_new(txt);                                /*新建一个标签。*/
18    gtk_container_add(GTK_CONTAINER(window),label);          /*添加标签。*/
19    strcpy(txt2,gtk_label_get_text(GTK_LABEL(label)));       /*取得标签的文本。*/
20    strcat(txt2,"\nthis is the append string.");             /*添加字符串。*/
21    gtk_label_set_text(GTK_LABEL(label),txt2);               /*设置标签。*/
22
23    gtk_widget_show(window);
24    gtk_widget_show(label);
25    gtk_main ();
26    return 0;
27  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.5.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-5 所示。在添加文本的最前面，用了一个“\n”换行符，则这一个标签的内容显示为两行。

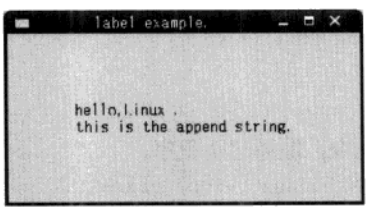


图 19-5　获取与设置标签中的文本

【代码解析】代码第 19~21 行先取得标签的文本字符串，然后和另一个字符串进行连接，最后将连接后的字符串设置为标签的文本。

注意：在这个程序中，需要注意 strcpy 和 strcat 两个函数的使用方法。在参数列表中，第一个参数是目录字符串，第二个参数是需要操作的字符串。

19.3　按钮

窗口程序中的很多操作都是通过窗口程序的单击来实现的。本节将讲解如何在 GTK 界面中添加按钮元件，主要内容包括按钮的创建、设置等操作。

19.3.1 添加一个按钮

函数 gtk_button_new_with_label 可以新建一个带有标签的按钮。这个函数的使用方法如下所示。

```
01  GtkWidget*  gtk_button_new_with_label(gchar *label);
```

函数的参数是一个字符串，这个字符串会显示在按钮上。函数会返回一个 GtkWidget 类型的指针。建立这个按钮以后，并不会直接显示，而需要调用 gtk_widget_show 函数来显示这个按钮。下面的实例在窗口中添加一个按钮。

【范例 19-6】使用 gtk_button_new_with_label 函数新建一个按钮，示例代码如下所示。

示例代码 19-6

```
01  #include <gtk/gtk.h>
02  #include <time.h>
03  #include <stdlib.h>                                    /*包含头文件。*/
04
05  int main (int argc, char *argv[])
06  {
07    GtkWidget *window;
08    GtkWidget *button;
09    char title[]="button example";
10
11    gtk_init (&argc, &argv);
12    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);          /*建立和处理窗口。*/
13    gtk_window_set_title (GTK_WINDOW (window),title);
14    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
15
16    button=gtk_button_new_with_label("NEW BUTTON");      /*建立一个按钮。*/
17    gtk_container_add(GTK_CONTAINER(window),button);     /*将按钮添加到窗口上。*/
18
19    gtk_widget_show(window);
20    gtk_widget_show(button);
21    gtk_main ();
22    return 0;
23  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.6.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-6 所示。在窗口中，因为没有容器容纳这个按钮，所以按钮会充满整个窗口。单击这个按钮时，按钮的颜色会发生改变，这时发生了一个单击事件。

图 19-6　窗口的按钮

【代码解析】代码第 16~17 行创建按钮实例并将按钮添加到窗口中。

19.3.2　取得与设置按钮的标签

按钮的标签指的是按钮上的文字。函数 gtk_button_get_label 可以取得一个按钮的标签，函数 gtk_button_set_label 可以设置一个按钮的标签。这两个函数的使用方法如下所示。

```
01  const gchar* gtk_button_get_label (GtkButton *button);
02  void  gtk_button_set_label (GtkButton *button, const gchar *label);
```

在参数列表中，button 是表示一个按钮的指针。函数 gtk_button_get_label 会取得这个按钮的标签作为一个字符串返回。函数 gtk_button_set_label 会把参数 label 的字符串设置成按钮的标签。下面的实例是这两个函数的使用实例。先用 gtk_button_get_label 函数取得一个按钮的标签。然后在返回的字符串后面添加一个字符串，再把这个字符串设置成按钮的标签。程序的代码如下所示。

【范例 19-7】使用 gtk_button_get_label 函数和 gtk_button_set_label 函数获取与设置按钮的标签，示例代码如下所示。

示例代码 19-7

```
#include <gtk/gtk.h>
#include <stdlib.h>                                          /*包含头文件。*/

int main (int argc, char *argv[])
{
  GtkWidget *window;                                         /*定义相关的变量。*/
  GtkWidget *button;
  char title[]="button example.";
  char txt[]="button .";
  char txt2[50];

  gtk_init (&argc, &argv);                                   /*建立与处理窗口。*/
  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window),title);
  gtk_widget_set_usize(GTK_WINDOW (window),300,150);

  button=gtk_button_new_with_label(txt);                     /*建立一个按钮。*/
  gtk_container_add(GTK_CONTAINER(window),button);/*按钮添加到窗口上。*/
  strcpy(txt2,gtk_button_get_label(GTK_BUTTON(button)));
                                                             /*取得按钮的文本。*/
  strcat(txt2,"\n append string.");                          /*添加一个字符串。*/
  gtk_button_set_label(GTK_BUTTON(button),txt2);             /*设置按钮的文本。*/

  gtk_widget_show(window);
  gtk_widget_show(button);
  gtk_main ();
  return 0;
}
```

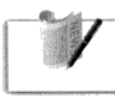
【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.7.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-7 所示。



提示：在 gtk 窗口中，按钮的标签可以是多行文字。

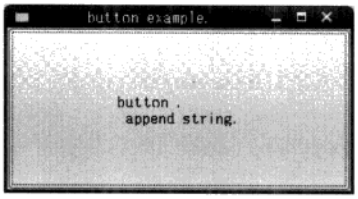


图 19-7　获取与设置按钮的标签

【代码解析】代码第 17~22 行创建新按钮并将其添加到窗口中，然后又取得按钮的标签字符串并与另一字符串连接，最后将连接后的字符串设置为按钮的标签字符串。

19.4　文本框

文本框指的是程序中的输入区域。用户可以在这个区域中用键盘输入内容。界面程序中的各种输入都是通过文本框来完成的。本节将讲解 gtk 窗口中文本框的使用。

19.4.1　建立文本框

在窗口中使用文本框以前需要建立文本框，gtk_entry_new 函数可以完成建立文本框的操作。这个函数的使用方法如下所示。

```
01  GtkWidget*  gtk_entry_new (void);
```

从函数的使用方法可知，这个函数没有参数，会返回一个 GtkWidget 类型的指针。如果文本框创建不成功，会返回一个 NULL 指针。

另一个建立文本框的函数如下所示。在这个函数中，有一个参数，用来表示这个文本框最多可以输入的字符。如果已经输入这些数目的字符，就不能再向文本框中输入内容。

```
01  GtkWidget* gtk_entry_new_with_max_length  (gint max);
```

下面的实例使用这个函数在窗口中建立一个文本框。

【范例 19-8】文本框使用举例示例代码如下所示。

示例代码 19-8

```
01  #include <gtk/gtk.h>
02  #include <stdlib.h>                                  /*包含头文件。*/
03
04  int main (int argc, char *argv[])
05  {
06    GtkWidget *window;                                 /*定义相关的变量。*/
07    GtkWidget *txt;
08    char title[]="entry example";
09
10    gtk_init (&argc, &argv);                           /*建立与处理窗口。*/
11    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
12    gtk_window_set_title (GTK_WINDOW (window),title);
13    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
14
15    txt=gtk_entry_new();                               /*建立一个文本框。*/
16    gtk_container_add(GTK_ENTRY(window),txt);          /*将文本框添加到窗口上。*/
17
18    gtk_widget_show(window);
19    gtk_widget_show(txt);
```

```
20    gtk_main ();
21    return 0;
22  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.8.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

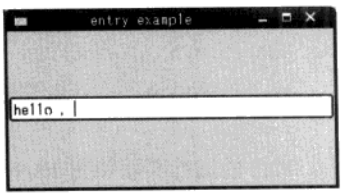


图 19-8　窗口中的文本框

程序的运行界面如图 19-8 所示。在窗口的文本框中可以输入字符。在这个文本框中，系统提供了复制、粘贴、全选等系统功能。

【代码解析】代码第 16~17 行创建文本框实例并将文本框添加到窗口中。

19.4.2　获取与设置文本框数据

文本框中输入数据以后，需要取得这些数据进行处理。新建文本框时可以设置文本框的初始内容。函数 gtk_entry_get_text 与函数 entry_set_text 可以分别完成这两个功能。这两个函数的使用方法如下所示。

```
01  const gchar* gtk_entry_get_text (GtkEntry *entry);
02  void gtk_entry_set_text (GtkEntry *entry, const gchar *text);
```

在参数列表中，entry 是一个表示文本框的指针。函数 gtk_entry_get_text 会返回用户在文本框中输入的字符串。函数 gtk_entry_set_text 中，text 是需要设置到文本框中的字符串，这个函数没有返回值。下面的实例，是在建立一个按钮以后设置这个文本框的初始内容。然后取得文本框的文本添加一个字符串以后设置到这个文本框中。程序的代码如下所示。

【范例 19-9】使用 gtk_entry_get_text 函数和 gtk_entry_set_text 函数获取与设置文本框数据，示例代码如下所示。

示例代码 19-9

```
01  #include <gtk/gtk.h>
02  #include <time.h>
03  #include <stdlib.h>                                    /*包含头文件。*/
04
05  int main (int argc, char *argv[])
06  {
07    GtkWidget *window;                                   /*定义相关的变量。*/
08    GtkWidget *txt;
09    char title[]="entry example";
10    char txt2[50];
11
12    gtk_init (&argc, &argv);                             /*建立与处理窗口。*/
13    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
14    gtk_window_set_title (GTK_WINDOW (window),title);
15    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
16
17    txt=gtk_entry_new();                                 /*建立文本框。*/
18    gtk_container_add(GTK_ENTRY(window),txt);            /*窗口上添加文本框。*/
19    gtk_entry_set_text(GTK_ENTRY(txt),"first text .");   /*文本框设置文本。*/
20    strcpy(txt2,gtk_entry_get_text(GTK_ENTRY(txt)));     /*取得文本框的文本。*/
21    strcat(txt2,"  append string.");                     /*添加一段文本。*/
22    gtk_entry_set_text(GTK_ENTRY(txt),txt2);             /*设置文本。*/
```

```
23
24    gtk_widget_show(window);
25    gtk_widget_show(txt);
26    gtk_main ();
27    return 0;
28  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.9.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-9 所示。

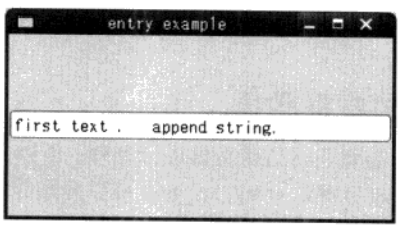


图 19-9　取得与设置文本框的值

【代码解析】代码第 17~22 行首先创建文本框实例并将其添加到窗口中，然后设置取得文本框显示的字符串值并与另一字符串连接，最后将连接后的字符串设置为文本框的值。

19.5　综合练习

本节将讲解一个标签应用实例，在一个窗口中用一个标签显示当前的时间。前面的章节中已经讲述过时间的处理方法，可以把当前的时间转换成一个字符串保存在字符串变量中。然后用 gtk_label_set_text 函数将字符串的内容设置为标签的文本。程序的代码如下所示。

【范例 19-10】使用标签显示当前系统时间，示例代码如下所示。

示例代码 19-10

```
01  #include <gtk/gtk.h>
02  #include <time.h>
03  #include <stdlib.h>                                    /*包含头文件。*/
04
05  int main (int argc, char *argv[])
06  {
07    GtkWidget *window;                                   /*定义相关的变量。*/
08    GtkWidget *label;
09    char title[]="label example:time";
10    char txt[50]="time:\n";
11    char time_txt[50];
12    time_t timep;
13
14    gtk_init (&argc, &argv);                             /*建立和处理窗口。*/
15    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
16    gtk_window_set_title (GTK_WINDOW (window),title);
17    gtk_widget_set_usize(GTK_WINDOW (window),300,150);
18
19    label=gtk_label_new("");                             /*新建标签。*/
```

```
20    gtk_container_add(GTK_CONTAINER(window),label);          /*标签添加到窗口上。*/
21
22    time (&timep);                                           /*处理时间。*/
23    strcpy(time_txt,asctime(gmtime(&timep)));
24    strcat(txt,time_txt);
25    gtk_label_set_text(GTK_LABEL(label),txt);       /*将标签的文本设置为时间。*/
26
27    gtk_widget_show(window);
28    gtk_widget_show(label);
29    gtk_main ();
30    return 0;
31  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 19.10.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 19-10 所示。

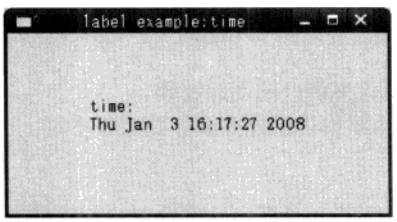


图 19-10　标签显示当前的时间

注意：在这个程序中，需要注意时间函数的使用和字符串的处理方法。asctime 函数返回的是一个字符串，需要用 strcpy 函数将这个字符串复制到另一个字符串中进行处理。

19.6　小结

本节讲述了 GTK 图形界面编程时基本原理、窗口、按钮、标签、文本框等图形元件的简单操作。通过这些知识的学习，可以对 Linux 界面编程有一个大致的了解，可以用这些元件编写出简单的 Linux 界面程序。窗口等元件的使用是本章的重点，需要理解元件建立和过程、元件简单设置等操作。

19.7　习题

一、填空题

1．在 GTK 程序设计时，新建窗口函数是_____。

2．设置一个窗口的标题的函数是_____。用_____函数来设置一个窗口的大小。

3．所谓_____，指的是程序中的一个文本。这个文本可以显示一定的信息，但是用户不能改变和输入标签的内容。

4. _____指的是程序中的输入区域。用户可以在这个区域中用键盘输入内容。界面程序中的各种输入都是通过它来完成的。

二、选择题

1. 关于 GTK 元件中能够接收用户输入信息的元件是（　　）。

A. 按钮

B. 标签

C. 文本框

D. 窗口

2. 在进行 GTK 程序设计是，设置图形界面以后，界面并不会显示。需要调用函数（　　）来绘制界面，这时窗口才会显示。

A. gtk_window_new

B. gtk_main ()

C. window_set_title

D. gtk_widget_set_usize

三、简答题

1. 简述使用 GTK 程序设计中窗口的创建过程。

2. 简述标签、按钮和文本框如何创建并添加到窗口中。

四、编程题

1. 编写 GTK 应用程序，创建一个窗口并设置窗口的标题为“new window”，窗口大小为宽 200，高 200。

2. 编写 GTK 应用程序，在一个窗口上分别放置标签、按钮、和文本框。

3. 编写 GTK 应用程序，在一个窗口上放置一个标签，在标签上显示当前的系统时间。

第 20 章　界 面 布 局

本章介绍了图形界面应用程序中如何利用表格、垂直框和水平窗格进行图形界面元素的布局，以达到特定的显示效果。还介绍了图形化的界面设计工具 Glade 的使用。本章的主要内容如下：

- 表格布局；
- 垂直框布局；
- 水平窗格布局；
- Glade 设计程序界面。

20.1　表格

所谓表格，是指用横竖布局的线和格子将一个窗口划分成多个区域。每一个区域可以放置不同的元件。如果一个元件中可放置其他的元件，这个元件就被称为容器。GTK 的容器都是二进制的，就是每一个容器中只能放置一个元件。如果需要在一个窗口中放置多个元件，则需要用表格、窗格等有很多单元格的容器。

20.1.1　表格的建立

在窗口中使用表格以前，需要用 gtk_table_new 函数新建一个表格。这个函数的使用方法如下所示。

```
01  GtkWidget* gtk_table_new (guint rows,guint columns,gboolean homogeneous);
```

在参数列表中，rows 表示表格的行数，columns 表示表格的列数。需要注意的是，这里的行数和列数是从行开始算的。homogeneous 是一个布尔值，如果设置为 TRUE，则每一个单元格的大小相同。所有单元格的高度与宽度和表格中最大的一个元件的宽度和高度相同。如果设置为 FALSE，则表格的单元格大小会根据单元格中的元件自动调整。

提示：表格的作用只是将表格划分成不同的区域，并不能显示出这个表格。添加一个表格后，需要用 gtk_container_add 函数将这个表格添加到窗口中。并且需要用 gtk_widget_show 函数显示这个表格。

【范例 20-1】使用 gtk_table_new 函数创建表格并将其添加到窗口中，示例代码如下所示。

示例代码 20-1

```
01  #include <gtk/gtk.h>                          /*包含头文件*/
02  int main (int argc, char *argv[])             /*主函数*/
03  {
04    GtkWidget *window;                          /*定义窗口及所需元件的指针变量*/
05    GtkWidget *txt;
06    GtkWidget *table;
```

```
07    char title[]="table example";
08    gtk_init (&argc, &argv);
09    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);          /*创建新窗口*/
10    gtk_window_set_title (GTK_WINDOW (window),title);    /*设置窗口标题*/
11    gtk_widget_set_usize(GTK_WINDOW (window),300,150);   /*设置窗口大小*/
12    table=gtk_table_new(2,2,TRUE);                       /*创建表格*/
13
14    gtk_container_add(GTK_ENTRY(window),table);          /*将表格添加到窗口中*/
15    gtk_widget_show(window);                             /*设置窗口与表格的可见性*/
16    gtk_widget_show(table);
17
18    gtk_main ();                                         /*显示窗口*/
19    return 0;
20  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.1.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

在运行的程序中，只显示了一个窗口，如图 20-1 所示。表格是一个容纳其他元件的容器，并不会实际显示。

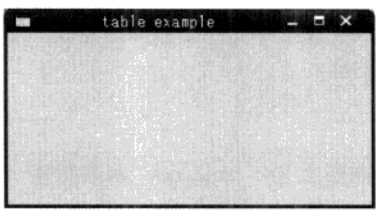


图 20-1　窗口中的表格

【代码解析】代码第 12 行创建了一个两行两列的表格，代码第 14 行将新创建的表格添加到窗口中。

提示：由于在表格的单元格中还没有添加任何元件，所以在窗口中看不到表格的内容。

20.1.2　表格中添加元件

建立表格以后，需要将元件添加到这个表格中。函数 gtk_table_attach 的作用是将一个元件添加到表格，并且设置在表格中的位置和填充的选项。这个函数的使用方法如下所示。

```
01  void gtk_table_attach (GtkTable *table,          GtkWidget *child,
02                         guint left_attach,        guint right_attach,
03                         guint top_attach,         guint bottom_attach,
04                         GtkAttachOptions xoptions, GtkAttachOptions yoptions,
05                         guint xpadding,           guint ypadding);
```

这个函数的参数比较复杂，参数的含义和作用如下所示。

- table：容器表格的指针。
- child：需要添加的元件的指针。
- left_attach、right_attach：元件的左边是表格的第几条边，右边是表格的第几条边。需

要注意的是，这里的边数是从 0 开始算的。

- top_attach、right_attach：上边是表格的第几条边、下边是表格的第几条边。
- xoptions、yoptions：元件在表格中的水平方向、垂直方向的对齐方式。取值的类型是 GtkAttachOptions。
- xpadding、ypadding：元件与边框水平方向或垂直方向的边距。

注意：在向表格中添加元件时，必须注意函数 gtk_table_attach 中的单元格位置的表示方法。

20.1.3　表格使用实例

本节将讲述一个表格实例。在程序的窗口中，添加一个两行两例的表格。表格的第一行第一列中添加一个按钮。第一行第二列中添加一个标签。第二行第一列中添加一个文本框。在表格中，需要注意添加元件的行列参数。程序的代码如下所示。

【范例 20-2】使用 gtk_table_attach 函数向表格中添加元件，示例代码如下所示。

示例代码 20-2

```
#include <gtk/gtk.h>                                          /*包含头文件*/
int main (int argc, char *argv[])                             /*主函数*/
{
  GtkWidget *window;                                          /*定义窗口及所需元件的指针变量*/
  GtkWidget *txt;
  GtkWidget *table;
  GtkWidget *button;
  GtkWidget *label;
  char title[]="table example";
  gtk_init (&argc, &argv);

  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);                 /*创建新窗口*/
  gtk_window_set_title (GTK_WINDOW (window),title);           /*设置窗口标题*/
  gtk_widget_set_usize(GTK_WINDOW (window),350,150);          /*设置窗口大小*/
  table=gtk_table_new(2,2,FALSE);                             /*创建表格*/
  button=gtk_button_new_with_label("OK");                     /*创建所需元件*/
  label=gtk_label_new("this is a label");
  txt=gtk_entry_new();

  gtk_container_add(GTK_ENTRY(window),table);                 /*将表格添加到窗口中*/
  gtk_table_attach (GTK_TABLE (table), button, 0, 1, 0, 1,    /*将元件添加到表格单元格中*/
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0,0);
  gtk_table_attach (GTK_TABLE (table), label, 1, 2, 0, 1,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0,0);
  gtk_table_attach (GTK_TABLE (table), txt, 0, 1, 1, 2,
                    (GtkAttachOptions) (GTK_FILL),
                    (GtkAttachOptions) (0), 0,0);

  gtk_widget_show(window);                                    /*设置窗口、表格、元件的可见性*/
  gtk_widget_show(table);
  gtk_widget_show(txt);
  gtk_widget_show(button);
  gtk_widget_show(label);
```

```
36
37    gtk_main ();
38    return 0;
39  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.2.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-2 所示。在这个界面中，三个元件已经按照设置的行和列排列在表格中。

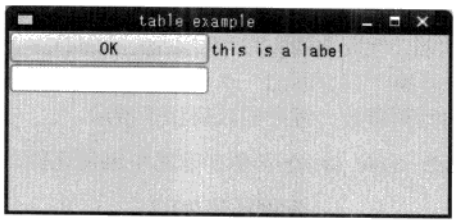


图 20-2　表格控制元件的布局

【代码解析】代码第 21~29 行向表格相应的单元格中添加按钮、文本框和文本字符串。其他代码与前面的示例相似。

20.1.4　单元格的合并

所谓单元格合并，指的是一个元件占据一个表格中同行或同列的多个单元格。表格的合并没有专门函数来实现。而是元件设置在表格的位置时，设置这个元件在表格中跨多个单元格的边界。例如，表格中的一个元件，设置左右边线分别为 0 和 1，设置上下边线分别为 1 和 2。则这个元件在表格中就是第 1 列，占据第 1 行和第 2 行。下面的实例将上一节中的右边元件跨越两个单元格。

【范例 20-3】在表格中实现单元格的合并效果，示例代码如下所示。

示例代码 20-3

```
01  #include <gtk/gtk.h>                                          /*包含头文件*/
02  int main (int argc, char *argv[])                             /*主函数*/
03  {
04    GtkWidget *window;                                  /*定义窗口及所需元件的指针变量*/
05    GtkWidget *txt;
06    GtkWidget *table;
07    GtkWidget *button;
08    GtkWidget *label;
09    char title[]="table example";
10    gtk_init (&argc, &argv);
11
12    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);             /*创建新窗口*/
13    gtk_window_set_title (GTK_WINDOW (window),title);       /*设置窗口标题*/
14    gtk_widget_set_usize(GTK_WINDOW (window),350,150);      /*设置窗口大小*/
15    table=gtk_table_new(2,2,FALSE);                         /*创建表格*/
16    button=gtk_button_new_with_label("OK");                 /*创建所需元件*/
17    label=gtk_label_new("this is a label");
18    txt=gtk_entry_new();
```

```
19
20    gtk_container_add(GTK_ENTRY(window),table);    /*将表格添加到窗口中*/
21    gtk_table_attach (GTK_TABLE (table), button, 0, 1, 0, 1,    /*将元件添加到表
格的指定单元格中*/
22                      (GtkAttachOptions) (GTK_FILL),
23                      (GtkAttachOptions) (0), 0,0);
24    gtk_table_attach (GTK_TABLE (table), label, 1, 2, 0,2,
25                      (GtkAttachOptions) (GTK_FILL),
26                      (GtkAttachOptions) (0), 0,0);
27    gtk_table_attach (GTK_TABLE (table), txt, 0, 1, 1, 2,
28                      (GtkAttachOptions) (GTK_FILL),
29                      (GtkAttachOptions) (0), 0,0);
30
31    gtk_widget_show(window);                          /*设置窗口、表格和元件的可见性*/
32    gtk_widget_show(table);
33    gtk_widget_show(txt);
34    gtk_widget_show(button);
35    gtk_widget_show(label);
36
37    gtk_main ();
38    return 0;
39  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.3.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-3 所示。在这个程序中，右边的元件在垂直方向占据了两个单元格。

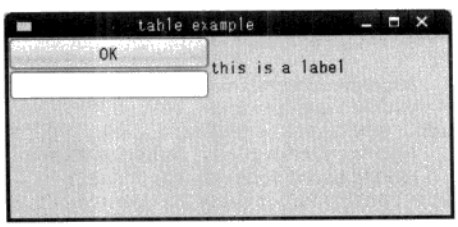


图 20-3　单元格的合并

【代码解析】代码第 24~26 行在元件设置在表格中的位置时，设置这个元件在表格中跨多个单元格的边界，以达到单元格的合并的功能。其他代码与示例代码 20-2 相同。

注意：表格的合并没有专门函数来实现。而是元件设置在表格的位置时，设置这个元件在表格中跨多个单元格的边界。

20.1.5　表格中添加表格

在复杂的界面中，使用一个表格并不能完成布局网页中所有的元件。这时需要在表格的单元格中添加表格。表格也是一个普通元件，可以把一个表格添加到另一个表格的单元格中。这样，通过表格的嵌套，就可以实现复杂的布局方式。下面的程序是一个表格嵌套的实例。外层表格是一个两行一列的表格，在下面的表格中添加一个两列一行的表格，然后在这些单元格中添加元件。程序的代码如下所示。

【范例 20-4】用表格嵌套实现复杂的窗口布局，示例代码如下所示。

示例代码 20-4

```
#include <gtk/gtk.h>                                    /*包含头文件*/
#include <time.h>
#include <stdlib.h>
int main (int argc, char *argv[])                       /*主函数*/
{
  GtkWidget *window;                                    /*定义窗口及所需元件的指针变量*/
  GtkWidget *txt;
  GtkWidget *table;
  GtkWidget *table2;
  GtkWidget *button;
  GtkWidget *label;
  char title[]="table example";
  gtk_init (&argc, &argv);

  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);           /*创建新窗口*/
  gtk_window_set_title (GTK_WINDOW (window),title);     /*设置窗口标题*/
  gtk_widget_set_usize(GTK_WINDOW (window),350,150);    /*设置窗口大小*/
  table=gtk_table_new(2,1,FALSE);                       /*创建表格*/
  table2=gtk_table_new(1,2,FALSE);
  button=gtk_button_new_with_label("  OK  ");           /*创建元件*/
  label=gtk_label_new(" this  is  a  label ");
  txt=gtk_entry_new();

  gtk_container_add(GTK_ENTRY(window),table);           /*将表格添加到窗口中*/
  gtk_table_attach (GTK_TABLE (table), table2, 0, 1, 1, 2,    /*将第二个表格添
加到第一个表格中*/
                    (GtkAttachOptions) (GTK_EXPAND),
                    (GtkAttachOptions) (0), 0,0);
  gtk_table_attach (GTK_TABLE (table), txt, 0, 1, 0, 1,  /*向表格中添加普通元
件*/
                    (GtkAttachOptions) (GTK_EXPAND),
                    (GtkAttachOptions) (0), 0,0);
  gtk_table_attach (GTK_TABLE (table2), label, 0, 1, 0,1,
                    (GtkAttachOptions) (GTK_EXPAND),
                    (GtkAttachOptions) (0), 0,0);
  gtk_table_attach (GTK_TABLE (table2), button, 1, 2, 0,1,
                    (GtkAttachOptions) (GTK_EXPAND),
                    (GtkAttachOptions) (0), 0,0);

  gtk_widget_show(window);                  /*设置窗口、表格和元件的可见性*/
  gtk_widget_show(table);
  gtk_widget_show(table2);
  gtk_widget_show(txt);
  gtk_widget_show(button);
  gtk_widget_show(label);

  gtk_main ();
  return 0;
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.4.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-4 所示。在这个界面中，用两个表格实现了不规则的元件布局。

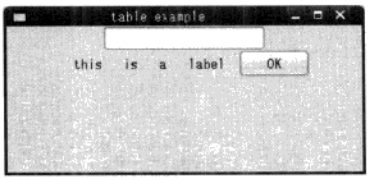


图 20-4　表格的嵌套

【代码解析】代码第 31~33 行将第二个表格添加到第一个表格的单元格中实现表格的嵌套，其他的代码与前面的示例代码相似。

提示：可以把一个表格添加到另一个表格的单元格中实现表格的嵌套，达到复杂的布局效果。

20.2　垂直框

所谓垂直框，指的是一个容器，其他的元件可以在这个容器的垂直方向上依次排列。

提示：垂直框可以看成是只有一列的表格，但是操作比表格简单，放置元件时不需要考虑元件的位置。

20.2.1　添加垂直框

窗口中需要使用垂直框时，需要用 gtk_vbox_new 函数添加一个垂直框，函数的使用方法如下所示。

```
01  GtkWidget* gtk_vbox_new (gboolean homogeneous, gint spacing);
```

参数列表中，homogeneous 是一个布尔值，表示元件在垂直方向是不是平均分配行高。spacing 表示每一行元件之间的间距。建立一个垂直框以后，需要用 gtk_container_add 函数将这个表格添加到窗口中。并且需要用 gtk_widget_show 函数显示这个表格。下面的实例在窗口中添加一个垂直框。

【范例 20-5】使用 gtk_vbox_new 函数创建垂直框并将其添加到窗口中，示例代码如下所示。

示例代码 20-5

```
01  #include <gtk/gtk.h>                                    /*包含头文件*/
02  int main (int argc, char *argv[])                       /*主函数*/
03  {
04    GtkWidget *window;                                    /*定义窗口及所需元件的指针变量*/
05    GtkWidget *vbox1;
06    char title[]="hpand example";
07    gtk_init (&argc, &argv);
08    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);    /*创建新窗口*/
09    gtk_window_set_title (GTK_WINDOW (window),title);  /*设置窗口标题*/
10    gtk_widget_set_usize(GTK_WINDOW (window),350,150); /*设置窗口大小*/
11
12    vbox1 = gtk_vbox_new (FALSE, 0);                      /*创建垂直框*/
13    gtk_widget_show (vbox1);
```

```
14    gtk_container_add (GTK_CONTAINER (window), vbox1);  /*将表格添加到窗口中*/
15
16    gtk_widget_show(window);
17    gtk_main ();
18    return 0;
19  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.5.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-5 所示。在这个界面中，垂直框中没有元件，所以无法看到垂直框的效果。

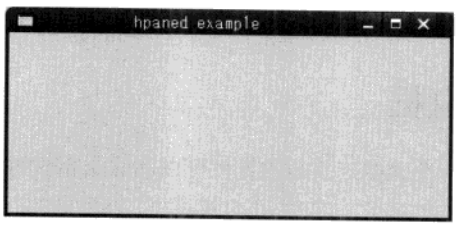


图 20-5　垂直框

【代码解析】本例向窗口中添加了一个垂直框（代码第 12~13 行），没有添加任何显示元件。

20.2.2　垂直框中添加元件

函数 gtk_box_pack_start 可以向已经建立的垂直框中加入一个元件。这个函数的使用方法如下所示。

```
01  void  gtk_box_pack_start (GtkBox *box,
02                            GtkWidget *child,
03                            gboolean expand,
04                            gboolean fill,
05                            guint padding);
```

在参数列表中，box 是一个垂直框容器指针。child 是需要添加的元件指针。expand 是一个布尔值，如果设置为 TRUE，表示为这个元件分配新的位置。fill 设置元件是不是填充垂直框的所有区域。padding 表示元件之间的间距。可以在一个垂直框中添加多个元件，不同的元件以添加的次序从上向下排列。

【范例 20-6】使用 gtk_box_pack_start 函数向垂直框中添加元件，示例代码如下所示。

示例代码 20-6

```
01  #include <gtk/gtk.h>                          /*包含头文件*/
02  int main (int argc, char *argv[])             /*主函数*/
03  {
04    GtkWidget *window;                          /*定义窗口及所需元件的指针变量*/
05    GtkWidget *vbox1;
06    GtkWidget *button;
07    GtkWidget *label;
08    GtkWidget *entry;
09    char title[]="vbox example";
10    gtk_init (&argc, &argv);
11
```

```
12    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);          /*创建新窗口*/
13    gtk_window_set_title (GTK_WINDOW (window),title);    /*设置窗口标题*/
14    gtk_widget_set_usize(GTK_WINDOW (window),350,150);   /*设置窗口大小*/
15    vbox1 = gtk_vbox_new (FALSE, 0);                     /*创建垂直框*/
16    gtk_widget_show (vbox1);                             /*创建垂直框与所需元件*/
17    entry=gtk_entry_new();
18    label=gtk_label_new("this is a label");
19    button=gtk_button_new_with_label(" OK ");
20    gtk_widget_show (button);
21    gtk_widget_show (label);
22    gtk_widget_show (entry);
23
24    gtk_container_add (GTK_CONTAINER (window), vbox1); /*将垂直框添加到窗口中*/
25    gtk_box_pack_start (GTK_BOX (vbox1),label, TRUE, FALSE, 0);/*将元件添加到垂
直框中*/
26    gtk_box_pack_start (GTK_BOX (vbox1),button, TRUE, FALSE, 0);
27    gtk_box_pack_start (GTK_BOX (vbox1),entry, TRUE, FALSE, 0);
28
29    gtk_widget_show(window);
30    gtk_main ();
31    return 0;
32  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.6.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-6 所示。在这个界面中，在垂直框中添加了三个元件。

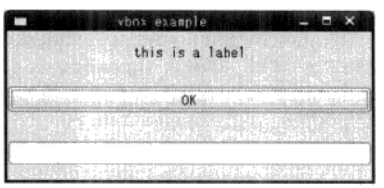


图 20-6　垂直框

【代码解析】代码第 24 行将创建的垂直框添加到窗口中，代码第 25~27 行调用 gtk_box_pack_start 函数向垂直框中添加元件。

20.3　水平窗格

窗格指的是一种界面布局方式，可以把一个界面划分成水平或垂直两个区域。窗格有水平窗格和垂直窗格两种。本章只讲垂直窗格的内容。

提示：拖动两个区域中间的分界线，可以改变两个窗格的大小。

20.3.1　新建一个窗格

如果需要在窗体中使用一个水平窗格，需要用 hpaned 函数新建一个水平窗格。这个函数的使用方法如下所示。

```
01  GtkWidget*  gtk_hpaned_new  (void);
```

从上面的使用方法可知，这个函数没有参数。返回值是一个表示窗格的指针。建立窗格以后，并不会直接显示。而需要用 gtk_container_add 函数将这个窗格添加到窗口中。并且需要用 gtk_widget_show 函数显示这个窗格。可以用 gtk_paned_set_position 设置窗格第一个区域的大小。这个函数的使用方法如下所示。

```
01  void gtk_paned_set_position (GtkPaned *paned,  gint position);
```

在参数列表中，paned 是一个表示窗格的指针。position 是一个整型数，表示这个窗格的分隔线到边界的位置。下面的实例是在一个窗口中添加一个窗格。

【范例 20-7】使用 gtk_hpaned_new 函数创建窗格并将其添加到窗口中，示例代码如下所示。

示例代码 20-7

```
01  #include <gtk/gtk.h>                                          /*包含头文件*/
02  int main (int argc, char *argv[])                             /*主函数*/
03  {
04    GtkWidget *window;                                  /*定义窗口及水平窗格的指针变量*/
05    GtkWidget *hpaned1;
06    char title[]="hpaned example";
07
08    gtk_init (&argc, &argv);
09    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);           /*创建新窗口*/
10    gtk_window_set_title (GTK_WINDOW (window),title);     /*设置窗口标题*/
11    gtk_widget_set_usize(GTK_WINDOW (window),350,150);    /*设置窗口大小*/
12
13    hpaned1 = gtk_hpaned_new ();                          /*创建水平窗格*/
14    gtk_widget_show (hpaned1);
15    gtk_container_add (GTK_CONTAINER (window), hpaned1);  /*将水平窗格添加到窗口中*/
16    gtk_paned_set_position (GTK_PANED (hpaned1), 62);      /*设置水平窗格的位置*/
17
18    gtk_widget_show(window);
19    gtk_main ();
20    return 0;
21  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.7.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-7 所示。

注意：因为窗格中没有内容，所以看不到这个窗格的显示效果。

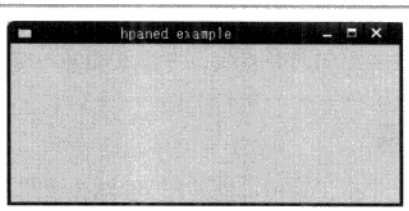


图 20-7　窗格

【代码解析】代码第 13~16 行完成水平窗格的创建，并把新建的水平窗格添加到窗口中，然后设置了水平窗格的配置。

20.3.2　窗格中添加元件

建立一个窗格以后，可以把元件添加到窗格中。把元件添加到窗格的函数是 gtk_paned_pack1 和 gtk_paned_pack2。这两个函数的使用方法如下所示。

```
01  void gtk_paned_pack1 (GtkPaned *paned,  GtkWidget *child,
02                        gboolean resize,   gboolean shrink);
03  void gtk_paned_pack2 (GtkPaned *paned,  GtkWidget *child,
04                         gboolean resize,  gboolean shrink);
```

这两个函数的使用方法是相同的。gtk_paned_pack1 的作用是把元件添加到窗格的第一个区域中。gtk_paned_pack2 的作用是把元件添加到窗格的第二个区域中。paned 表示作为容器的窗格。child 表示需要装入的元件。resize 是一个布尔值，如果设置为 TRUE，表示当窗格的大小改变时，装入的元件随之改变大小。shrink 设置为 TRUE 时，如果窗格比这个元件小，元件会自动改变大小。

【范例 20-8】使用 gtk_paned_pack1 函数和 gtk_paned_pack2 函数向窗格中添加元件，示例代码如下所示。

示例代码 20-8

```
01  #include <gtk/gtk.h>                                  /*包含头文件*/
02  int main (int argc, char *argv[])                     /*主函数*/
03  {
04    GtkWidget *window;                                  /*定义窗口及所需元件的指针变量*/
05    GtkWidget *label;
06    GtkWidget *entry;
07    GtkWidget *hpaned1;
08    char title[]="hpaned example";
09
10    gtk_init (&argc, &argv);
11    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);          /*创建新窗口*/
12    gtk_window_set_title (GTK_WINDOW (window),title);    /*设置窗口标题*/
13    gtk_widget_set_usize(GTK_WINDOW (window),350,150); /*设置窗口大小*/
14
15    hpaned1 = gtk_hpaned_new ();                        /*创建水平窗格、元件等*/
16    gtk_widget_show (hpaned1);
17    entry=gtk_entry_new();
18    gtk_widget_show (entry);
19    label=gtk_label_new("this is a label");
20    gtk_widget_show (label);
21
22    gtk_paned_pack1 (GTK_PANED (hpaned1), entry, TRUE, TRUE);/*将元件添加到水平窗
格中*/
23    gtk_paned_pack2 (GTK_PANED (hpaned1), label, TRUE, TRUE);
24    gtk_container_add (GTK_CONTAINER (window), hpaned1); /*将水平窗格添加到窗口中*/
25    gtk_paned_set_position (GTK_PANED (hpaned1), 62);  /*设置水平窗格的位置*/
26
27    gtk_widget_show(window);
28    gtk_main ();
29    return 0;
30  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.8.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-8 所示，文本框的标签被放置到窗格的左右两个区域。当拖动窗格的分隔线时，文本框和标签会随之改变大小，如图 20-9 所示。

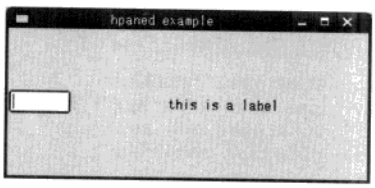


图 20-8　窗口中的窗格

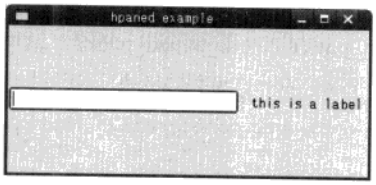


图 20-9　拖动间隔线

【代码解析】代码第 22~23 行调用 gtk_paned_pack1 函数向水平窗格中添加了两个显示元件。其他代码与示例代码 20-7 相似。

提示：当拖动水平窗格中的分隔线时，对应的元件的大小会随之发生改变，

20.4　Glade

在 C 程序中，用 GTK 虽然可以设计出各种界面。但在设计程序界面时，需要考虑到各种元件的关系，需要用复杂的代码来描述各种元件的属性。本节将要介绍的 Glade 很好地解决了程序中的界面设计问题，用各种元件工具即可完成界面的设计。

20.4.1　Glade 简介

Glade 是一个功能强大的 GTK 图形界面产生器。也就是说，Glade 是一个界面化的程序设计工具，和 Windows 系统的 VB、VC++类似，可以用各种功能设计出程序的界面。

注意：Glade 并不是一个程序开发平台。设计出界面以后，Glade 可以自动生成程序的界面部分代码，自动生成工程编译文件。这时需要用其他的编译器编写程序的事件代码。

使用 Glade 可以生成 C、C++、Ada95 等语言的界面代码。

20.4.2　使用 Glade 设计程序界面

Glade 是一个功能强大的 GTK 界面设计工具，使用 Glade 可以方便地设计出各种 GTK 界面。本节将讲述 Glade 设计一个 C 程序界面的过程。

注意：Glade 并非集成化的开发环境，仅能使用 Glade 生成程序的界面，剩余的工作需要借助其他工具完成。

1. 启动 Glade

Red Hat 9.0 Linux 或 Fedora 等 Linux 版本都默认安装了 Glade。下面基于 Fedora8 讲解 Glade 的使用。启动 Glade 软件的步骤如下。单击“主菜单”|“编程”|“Glade Interface Designer”

菜单命令，即可打开 Glade。Glade 的工作界面如图 20-10 所示。

2．打开 Glade 的工作面板

Glade 有工具栏、属性编译器、构建树等多个工作面板。使用 Glade 以前需要打开这些工作面板。

（1）打开调色板。这里所说的调色板指的是 Glade 的常用工具栏。在“Glade”的工作界面中，单击“查看”|“显示调色板”菜单命令，即可打开主工具栏。主工具栏如图 20-11 所示。

（2）显示属性编辑器。属性编译器是 Glade 中对程序中的各种元件进行属性设置的工具。单击“查看”|“显示属性编辑器”菜单命令，即可打开属性编辑器，如图 20-12 所示。

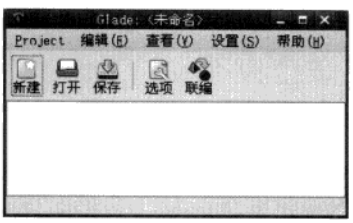


图 20-10　Glade 的启动界面

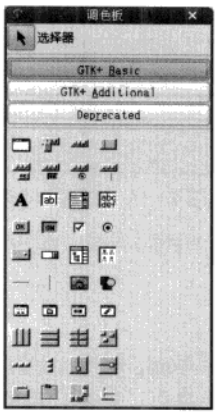


图 20-11　Glade 的主工具栏

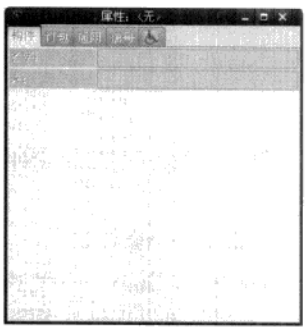


图 20-12　打开属性编辑器

（3）显示构建树。构建树指的是 Glade 中对各种元件进行管理和树状列表的工具。单击“查看”|“显示构建树”菜单命令，即可显示构建树工具栏，如图 20-13 所示。

（4）显示剪贴板。剪贴板的作用，是将用户复制的元件或文本进行列表，用户可以从列表中选择需要粘贴的内容。单击“查看”|“显示剪贴板”菜单命令，可以打开剪贴板工具，如图 20-14 所示。

图 20-13　Glade 的构建树工具

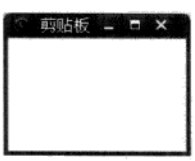


图 20-14　Glade 的剪贴板

20.4.3　Glade 设计窗口

所谓窗口，就是一个程序的工作界面。程序中的所有元件需要放置在一个窗口中。新建一

个项目以后，可以在项目中添加多个窗口。建立窗口以后，需要对窗口的名称、标题、大小、图标等内容进行设置。

1. 新建一个工程

窗口是界面程序中各种元件的载体。在一个 GTK 工程中，首先需要建立一个窗口才能建立其他的元件。窗口的建立步骤如下所示。

所谓工程，指的是一个 C 程序项目，所设计的界面程序需要放置在这个工程中。设计程序界面以前，需要新建一个工程。

① 单击“Project”|“新建”菜单命令，新建工程。显示的“新项目”对话框如图 20-15 所示。

图 20-15　新建一个项目

② “新项目”对话框的作用是项目的类型。单击“New GTK+ Project”按钮，新建 GTK 项目。

2. 新建一个窗口

所谓窗口，就是一个程序的工作界面。程序中的所有元件需要放置在一个窗口中。新建一个项目以后，可以在项目中添加多个窗口。新建和设置窗口的操作如下所示。

① 在主工具栏中，单击“窗口”工具，即可新建一个窗口。新建的窗口如图 20-16 所示。在窗口的任意位置单击可以选择这个窗口。

② 新建窗口以后，可以在 Glade 的主界面中查看到这个窗口，如图 20-17 所示。

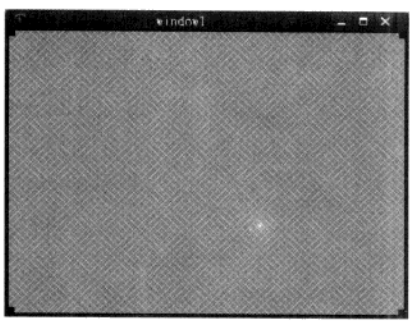


图 20-16　新建一个窗口

图 20-17　Glade 主界面中的窗口

3. 设置窗口的标题和名称

窗口有名称、标题、大小、位置等常用的属性。新建一个窗口以后，所有的设置都是默认的。可以在属性工具栏中设置这个窗口的属性。本节将讲述窗口的属性设置。

① 单击选择已经新建的窗口。

② 设置窗口的名称。窗口的名称指的是在编程时，调用这个窗口的指针的名称。如图 20-18 所示，在属性对话框的“名字”文本框中输入这个窗口的名称“mywindow”。

③ 设置窗口的标题。窗口的标题指的是显示在窗口最上面的提示。如图 10-18 所示，在

“标题”文本框中输入窗口的标题“First Window”。

④ 设置标题以后的窗口如图 20-19 所示，可见窗口的标题已经改变。

图 20-18　设置窗口的名称

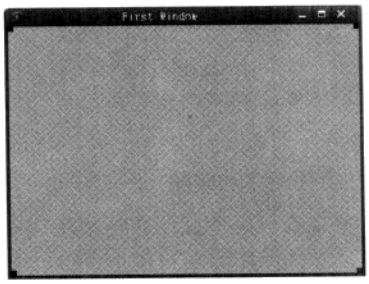


图 20-19　设置标题以后的窗口

4．设置窗口的大小

窗口的大小指的是窗口的宽度和高度。可以在属性工具栏中设置一个窗口的大小。

① 如图 20-20 所示，单击选择“默认宽度”和“默认高度”后面的复选框，然后分别输入宽度为“300”，高度为“200”。这个大小是程序启动时这个窗口的大小。

② 设置用户是否可以调整窗口大小。在窗口对话框中，单击“Resizable”按钮，可以更改这个设置，如图 20-20 所示。如果设置为“否”，则这个窗口没有最大化按钮，用户不能用拖动边框的方式改变窗口的大小。

5．设置程序的图标

程序的图标，指的是程序编译以后，程序文件的图标。程序运行时，标题栏上会显示这个图标。下面将讲述程序图标的设置。

① 在属性对话框中，单击“图标”后面的选择按钮，显示图 20-21 所示的“选择文件”对话框。

② 如图 20-21 所示，在“文件选择”对话框中，双击需要打开的文件，然后双击选择一个图标文件。然后单击“确定”按钮。

图 20-20　设置窗口的宽度和高度

图 20-21　选择一个图标文件

提示：图标文件的扩展名是.ico。

③ 如图 20-22 所示，窗口的左上角显示了已经设置的图标。

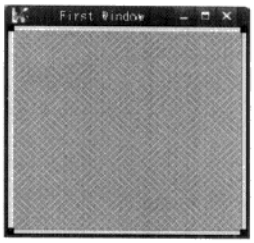


图 20-22　窗口改变图标

20.4.4　添加表格

在窗口中，是通过表格来控制元件的布局的。建立窗口以后，需要在窗体中添加表格来容纳其他元件。本节将讲述窗口中表格的操作。

1．窗体中添加表格

如果要在窗体中使用表格来实现元件的布局，首先需要添加一个表格。添加表格的步骤如下所示。

① 单击选择需要添加表格的窗口。

② 在主工具栏中，单击选择“表格”工具，然后在窗口上单击。显示的“新建表”对话框如图 20-23 所示。

③ 在“新建表”对话框中，在“行数”和“列数”文本框中分别输入“3”，表示新建一个 3 行 3 列的表格。然后单击“确定”按钮。

④ 添加表格以后的窗口如图 20-24 所示。

2．插入行和列

建立表格以后，如果行数和列数不符合要求，可以在表格中插入行和列。也可以删除表格的行和列。本节将讲述表格的行列操作。

① 在表格中，单击选择一个单元格。右键单击这个单元格，然后在快捷菜单中单击“table2”|“在以下对象之后插入行”命令，即可在这个单元格的后面插入一行。

② 插入一列。右键单击一个单元格，然后在快捷菜单中单击“table2”|“在以下对象之后插入列”命令，即可在这个单元格的后面添加一列。

③ 在表格中添加了行和列的窗口如图 20-25 所示。

图 20-23　“新建表”对话框

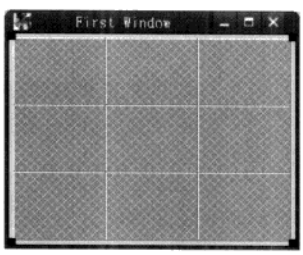


图 20-24　插入表格的窗口

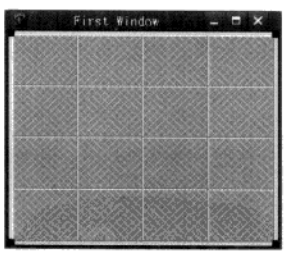


图 20-25　在表格中添加行和列

④ 用同样的方法，可以在表格中删除行和列。

提示：表格是强大的元件布局工具，巧妙地应用表格可以实现复杂的界面布局效果。

20.4.5　添加标签

标签是程序中用来提示静态信息的文本。用户不能在标签中输入内容。用 Glade 的标签工

具可以方便地在窗口中添加标签。本节将讲述窗口中的标签操作。

1．表格中添加标签

要在窗口中使用标签，首先要用标签工具在程序中添加一个标签。本节将讲解 Glade 中的标签操作。

① 标签需要放置在一个窗口或表格中。表格可以容纳各种元件，本小节将在表格的一个单元格中添加一个标签。

② 在主工具栏中单击选择“标签”工具。然后在窗口表格中需要添加标签的单元格中单击，即可添加一个标签。窗口表格中的标签如图 20-26 所示。标签中的默认文本是“label1”。

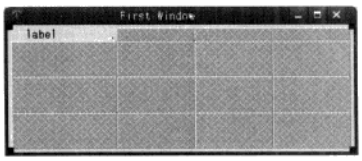


图 20-26　窗口表格中的标签

2．设置标签的名称和内容

标签的名称指的是在编程时调用这个标签的指针名称。标签的内容是显示在程序中标签上的文字。可以在标签的属性对话框中设置标签的名称和内容。

① 单击选择这个标签。这时这个标签就处于选中状态。这时可以在属性对话框中对标签的属性进行设置。

② 在“属性”对话框中，“名字”文本框中输入这个标签的名称“mylabel”，如图 20-27 所示。

③ 如图 20-27 所示，在“标签”文本框中，输入标签的内容“this is a label .”。这些文本将会显示在程序的标签中。

④ 窗口中，设置了名称与内容的标签，如图 20-28 所示。

图 20-27　设置标签的名称与内容

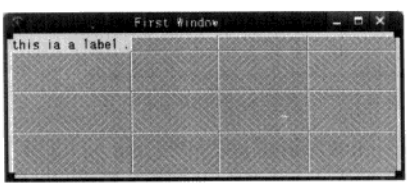


图 20-28　标签的内容

3．设置标签的填充边界

从上一小节图中的标签可知，标签的四周是没有边界的。可以在标签的“属性”对话框中设置标签的填充边界。

① 单击选择需要设置的标签。

② 在“属性”对话框中，向下拖动滚动条。在“X 填充”文本框中，输入“20”。在“Y 填充”文本框中输入“30”。这一操作如图 20-29 所示。

③ 设置边界以后的标签，如图 20-30 所示。

图 20-29　设置标签的边界

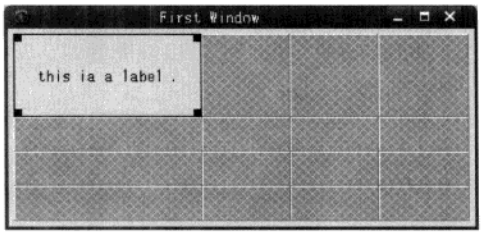


图 20-30　设置填充边界以后的标签

4．设置标签的倾斜角度

标签的倾斜角度指的是标签中的文字可以按一定的角度旋转。可以在标签的“属性”对话框中设置一个标签的倾斜角度。

① 单击选择一个标签，“属性”对话框可以对这个标签进行设置。

② 在标签的“属性”对话框中，“Angle”文本框中输入“45”，如图 20-31 所示。这个设置表示将标签旋转 45 度。

③ 设置 45 度倾斜的标签如图 20-32 所示。

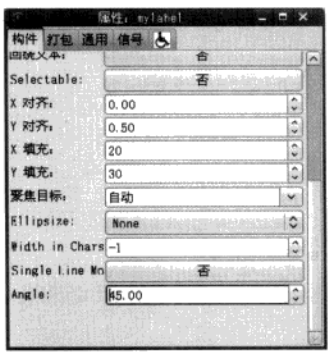


图 20-31　设置标签的倾斜角度

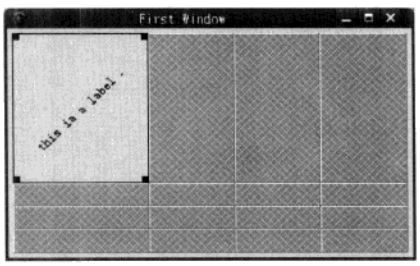


图 20-32　设置倾斜角度以后的标签

④ 在“Angle”文本框中输入“270”，可以将文字垂直排列显示，效果如图 20-33 所示。

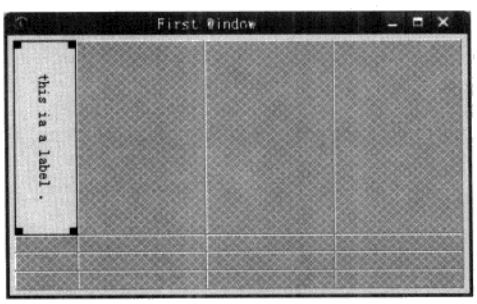


图 20-33　设置垂直排列的标签

20.4.6　添加文本框

文本框是程序中用户可以输入文字的区域。用 Glade 的“文本框”工具可以方便地在程序中添加一个文本框。本节将讲述窗口中的文本框操作。

1．表格中添加文本框

窗口中的文本框可以实现数据的输入功能。本节将讲述如何在表格中添加一个文本框。

① 文本框需要放置在一个窗口或表格中。表格可以容纳各种元件，本小节将在表格的一个单元格中添加一个文本框。

② 在主工具栏中单击选择“文档框”工具。然后在窗口表格中需要添加文本框的单元格中单击，即可添加一个文本框。窗口表格中添加文本框以后的效果如图 20-34 所示。

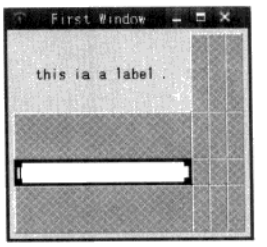


图 20-34　窗口表格中添加一个文本框

2．设置文本框

本节将讲述文本框的设置操作。在窗口插入的文本框，需要设置名称、最大长度、初始字符等内容。

① 在窗口中单击选择需要设置的文本框。

② 设置文本框的名称。文本框的名称指的是在编程时程序中调用这个文本框的指针。如图 20-35 所示，在“属性”对话框的“名字”文本框中输入这个文本框的名称“mytext”。

③ 设置文本框的最大字符数。最大字符数指的是用户在这个文本框中最多可以输入的字符数。如图 22-26 所示，在“最大长度”文本框中输入“10”，表示这个文本框中最多可以输入 10 个字符。

④ 设置初始文本。初始文本指的是程序启动时这个文本框中已有的文本。如图 22-26 所示，在“文本”文本框中输入“hello”。

⑤ 经过这些属性设置的文本框如图 20-36 所示。

图 20-35　设置文本框的属性

图 20-36　设置属性以后的文本框

3．将文本框设置成密码文本框

密码文本框是一种特殊的文本框，用户在文本框中输入内容时，这些字符会以星号或其他字符显示。下面将讲述如何把一个文本设置成密码文本框。

① 单击“文本可见”后面的按钮，这时按钮上面的文字“是”会变成否，如图 20-37 所示。

② 如图 22-28 所示，在“Invisible Char”文本框中输入“X”，表示文本框中所输入的内容会以“X”字符显示。需要注意的是，文本框中的字符依然是输入的字符，只是不显示这些字符。

③ 设置以后的密码文本框如图 20-38 所示。

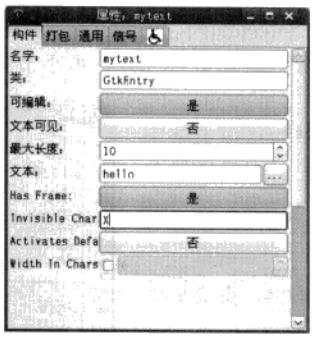


图 20-37 设置密码文本框

图 20-38 密码文本框

20.4.7 添加按钮

按钮是图形界面中，实现用户激活一个事件的工具。Glade 中可以用按钮工具完成按钮的添加与设置。本节将讲述界面中的按钮操作。

1. 添加按钮

窗口中的按钮实现用户单击功能。下面的步骤可以在窗口中表格的一个单元格里添加一个按钮。

① 按钮需要放置在一个窗口或表格中。表格可以容纳各种元件，本小节将在表格的一个单元格中添加一个按钮。

② 在主工具栏中单击选择“按钮”工具。然后在窗口表格中需要添加按钮的单元格中单击，即可添加一个按钮。窗口表格中添加按钮以后的效果如图 20-39 所示。

图 20-39 在窗口中添加一个按钮

2. 按钮的属性设置

按钮需要设置名称、标签、边界宽度等属性。本节将讲述这些属性的设置。

① 单击选择需要设置属性的按钮。

② 设置按钮的名字。按钮的名字指的是程序中用来调用这个按钮的指针。如图 20-40 所示，在“属性”对话框的“名字”文本框中输入这个按钮的名字“mybutton”。

③ 设置边界宽度。边界宽度指的是按钮的边缘与表格的分隔线之间的距离。如图 20-40 所示，在“边界宽度”文本框中输入“4”。

④ 设置按钮的标签。按钮标签指的是按钮上显示的文本。如图 22-31 所示，在“标签”文本框中输入“OK”。

⑤ 设置属性以后的按钮如图 20-41 所示。

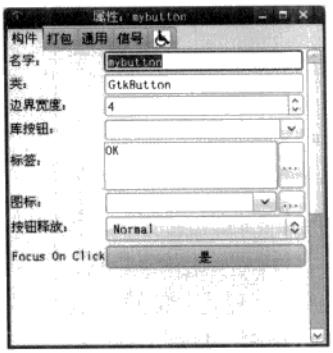


图 20-40　设置按钮的属性

图 20-41　设置属性以后的按钮

20.4.8　添加信号与事件

使用 Glade 可以方便地对窗体中的元件设置各种信号。设置信号以后，Glade 会自动生成和调用事件函数。关于信号与事件的详细内容可以参考下一章。本节将讲述 Glade 中的事件操作。

1. 在 Glade 添加信号

Glade 中添加的信号可以自动生成事件和信号函数。下面以一个按钮的信号添加为例讲解信号添加的过程。

① 单击选择需要设置信号的元件。在本节中，需要对上一节添加的按钮添加一个单击事件，需要单击选择窗口中的按钮。

② 单击“属性”对话框的“信号”标签。显示的“信号”标签如图 20-42 所示。

③ 添加一个信号。如图 20-41 所示，单击“信号”文本框后面的浏览按钮，显示的“选择信号”对话框如图 20-43 所示。

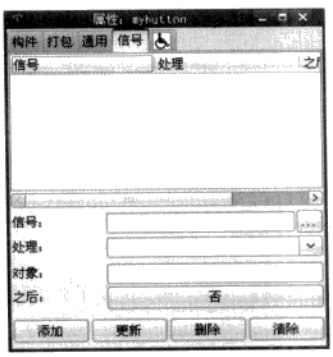


图 20-42　“信号”标签

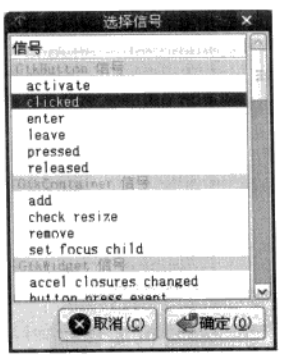


图 20-43　“选择信号”对话框

④ 在“选择信号”对话框中，双击“clicked”事件，Glade 会对按钮添加一个单击事件。这时“信号”标签中显示的 clicked 事件如图 20-44 所示。

⑤ 在“信号”标签单击“添加”按钮，将这个事件添加到“信号”列表中。“信号”列表中显示已经添加的信号如图 20-45 所示。

⑥ 单击“更新”按钮，将这个信号应用到按钮上。

图 20-44　添加的 clicked 事件

图 20-45　列表中的事件

2. 信号函数的编写

Glade 只能生成信号事件与函数，但不能实现事件函数的编写。事件的功能需要用其他的文本编辑器编写。下面将讲述如何用 gedit 工具编写 Glade 生成的事件函数。

① 单击“主菜单”|“附件”|“文本编辑器”工具，打开 gedit。

② 在 gedit 中，单击“文件”|“打开”菜单命令，打开 Glade 工程的信号函数文件。文件的目录是“/root/aa/src/callbacks.c”。

③ 这个文件的内容如下所示。其中，on_mybutton_clicked 函数是按钮的事件函数。

```
01  #ifdef HAVE_CONFIG_H
02  #  include <config.h>
03  #endif
04  #include <gtk/gtk.h>
05  #include "callbacks.h"
06  #include "interface.h"
07  #include "support.h"
08  void  on_mybutton_clicked (GtkButton *button,
09                     gpointer  user_data)
10  {
11  }
```

④ 在大括号中输入下面的代码，表示关闭这个窗口。则在程序中，用户单击这个按钮时会关闭程序的窗口。

```
gtk_main_quit();
```

⑤ 单击“文件”|“保存”菜单命令保存这个文件。

20.4.9　Glade 实例

本章前面已经详细介绍了 Glade 的基本用法，Glade 工程已经自动生成了程序编译的安装文件。本节将讲述如何使用这些自动生成的安装文件编译与安装 Glade 设计的程序。

① 打开一个终端，在终端中输入下面的命令进入到工程目录。

```
cd /root/aa
```

② 输入下面的命令，查看这个目录下的工程文件。

```
ls
```

③ 显示的结果如下所示，在这个工程中，自动生成了系统安装与配置文件。

```
aclocal.m4      autom4te.cache  config.sub    Makefile      README
aa.glade        ChangeLog       configure     Makefile.am   src
aa.glade.bak    config.guess    configure.in  Makefile.in   stamp-h1
```

```
aa.gladep       config.h       COPYING      missing       stamp-h.in
aa.gladep.bak   config.h.in    depcomp      mkinstalldirs
AUTHORS         config.log     INSTALL      NEWS
autogen.sh      config.status  install-sh   po
```

④ autogen.sh 是工程是自动编译文件。输入下面的命令，对这个文件添加可执行权限。

```
chmod +x autogen.sh
```

提示：通常，由 Glade 生成的工程中文件 autogen.sh 已经具有了可执行权限，当此文件不具有可执行权限时才需要执行上面的指令。

⑤ 输入下面的命令，对编译的文件进行预处理。

```
./autogen.sh
```

⑥ 程序会自动对这个工程进行配置和处理，显示了复杂的处理过程。最后一行显示了如下所示的提示。

```
Now type `make' to compile.
```

⑦ 提示的含义是输入“make”命令编译这个程序。这时在终端中输入下面的命令。

```
make
```

⑧ 输入下面的命令，可以运行这个程序。编译以后的程序在 src 目录下面。

```
src/aa
```

⑨ 输入下面的程序，可以把这个程序安装到计算机中。

```
make install
```

20.5　综合练习

本节将利用上面所讲述的表格布局实现一个用户登录窗口，使用表格进行窗口中元件的布局。

【范例 20-9】使用表格布局实现用户登录窗口，示例代码如下所示。

示例代码 20-9

```
#include <gtk/gtk.h>                                    /*包含需要的头文件*/
#include <time.h>
#include <stdlib.h>
int main (int argc, char *argv[])                       /*主函数*/
{
  GtkWidget *window;                                    /*定义窗口及所需元件的指针变量*/
  GtkWidget *txt;
  GtkWidget *txt2;
  GtkWidget *table;
  GtkWidget *button;
  GtkWidget *label;
  GtkWidget *label2;
  char title[]="user login window";
  gtk_init (&argc, &argv);

  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);           /*创建新窗口*/
  gtk_window_set_title (GTK_WINDOW (window),title);     /*设置窗口标题*/
  gtk_widget_set_usize(GTK_WINDOW (window),350,100);    /*设置窗口大小*/
  table=gtk_table_new(3,2,FALSE);                       /*创建表格*/
  button=gtk_button_new_with_label("  login  ");        /*创建需要的元件*/
  label=gtk_label_new(" username: ");
  label2=gtk_label_new(" password: ");
```

```
    txt=gtk_entry_new();
    txt2=gtk_entry_new();

    gtk_container_add(GTK_ENTRY(window),table);          /*将表格添加到窗口中*/

    gtk_table_attach (GTK_TABLE (table), label, 0, 1, 0, 1,      /*向表格的指定单
元格中添加元件*/
                      (GtkAttachOptions) (GTK_EXPAND),
                      (GtkAttachOptions) (0), 0,0);
    gtk_table_attach (GTK_TABLE (table), txt, 1, 2, 0, 1,
                      (GtkAttachOptions) (GTK_EXPAND),
                      (GtkAttachOptions) (0), 0,0);
    gtk_table_attach (GTK_TABLE (table), label2, 0, 1, 1, 2,
                      (GtkAttachOptions) (GTK_EXPAND),
                      (GtkAttachOptions) (0), 0,0);
    gtk_table_attach (GTK_TABLE (table), txt2, 1, 2, 1, 2,
                      (GtkAttachOptions) (GTK_EXPAND),
                      (GtkAttachOptions) (0), 0,0);
    gtk_table_attach (GTK_TABLE (table), button, 1, 2, 2, 3,
                     (GtkAttachOptions) (GTK_EXPAND),
                     (GtkAttachOptions) (0), 0,0);


    gtk_widget_show(window);                                  /*设置窗口及元件的可见性*/
    gtk_widget_show(table);
    gtk_widget_show(txt);
    gtk_widget_show(txt2);
    gtk_widget_show(button);
    gtk_widget_show(label);
    gtk_widget_show(label2);


    gtk_main ();
    return 0;
}
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 20.9.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 20-46 所示。

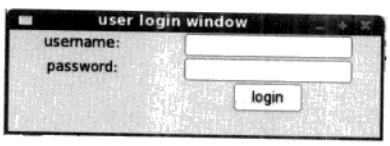


图 20-46　用户登录窗口

20.6　小结

本章主要讲述了 GTK 图形界面程序中元件的布局方法。表格元件是非常重要的布局元件，可以在表格的单元格中插入不同的元件，从而实现元件的布局和排列。本章的难点是如何将一个元件添加到表格的一个单元格中，如何控制单元格的行列。通过对表格布局的学习，可以在一个窗口中设计出复杂的界面和程序功能。本章还讲述了使用 Glade 自动生成程序界面的方法。

Glade 提供了各种图形界面工具，可以自动生成各种程序图形界面，自动生成 C 语言的图形界面代码。在本章的学习中，需要注意图形工具的使用方法。

20.7　习题

一、填空题

1．所谓_____，是指用横竖布局的线和格子将一个窗口划分成多个区域。每一个区域可以放置不同的元件。如果一个元件中可放置其他的元件，这个元件就被称为_____。

2．所谓_____，指的是一个容器，其他的元件可以在这个容器的垂直方向上依次排列。

3．_____指的是一种界面布局方式，可以把一个界面划分成水平或垂直的两个区域。

4．在 Linux 下有一个界面化的程序设计工具称为_____，它和 Windows 系统的 VB、VC++类似，可以用各种功能设计出程序的界面。

5．Glade 有_____、_____、_____等多个工作面板。使用 Glade 以前需要打开这些工作面板。

6．_____是界面程序中各种元件的载体。在一个 GTK 工程中，首先需要建立一个_____才能建立其他的元件。

7．在窗口中，是通过_____来控制元件的布局的。建立窗口以后，需要在窗体中添加它来容纳其他元件。

二、选择题

1．下面关于 Glade 的叙述正确的是（　　）。

A．Glade 是一个功能强大的 GTK 图形界面产生器。同时也是一个功能强大的编程平台

B．Glade 中的元件不需要将元件添加到窗口上就可以直接生成并显示

C．Glade 是一个功能强大的 GTK 界面设计工具，使用 Glade 可以方便地设计出各种 GTK 界面。但是 Glade 并非集成化的开发环境，仅能使用 Glade 生成程序的界面，剩余的工作需要借助其他工具完成

D．Glade 中的标签中的文字是可以编辑的

2．在 Glade 中控制元件的布局的元件是（　　）。

A．窗口　　B．表格

C．按钮　　D．标签

3．在 Glade 中关于信号和事件的叙述中正确的是（　　）。

A．Glade 中添加的信号可以自动生成事件和信号函数

B．Glade 不但能生成信号事件与函数，而且能实现事件函数的编写

C．设置信号以后，Glade 不会自动生成和调用事件函数

D．在 Glade 中，向元件中添加信号必须手动修改代码，无法通过图形界面直接完成信号的添加

4．在 Glade 的元件中文本内容可编辑的元件是（　　）。

A．文本框　　B．表格

C．按钮　　D．标签

三、简答题

1．简述常用的窗口元件布局方式及各自的特点。

2．简述 Glade 工具的特点。

3．简述使用 Glade 进行界面开发的整个过程。

四、编程题

1．编程实现在窗口中添加一个 2 行 2 列的表格，在表格单元格中添加标签、文本框、按钮等元件。

2．用表格实现一个用户登录窗口，界面如图 20-45 所示。

3．使用 Glade 创建一个窗口，然后向窗口中添加一个 3 行 4 列的表格。在表格的第 1 行第 1 列的单元格中添加一个文本框。在第 1 行第 2 列的单元格中添加一个标签。在第 1 行第 3 列的单元格中添加一个按钮。

4．使用 Glade 创建一个窗口，在窗口中添加一个按钮，在按钮上添加一个单击事件，并且编写代码实现单击按钮时退出程序。编译并运行此程序。

第 21 章　信号与事件

图形界面应用程序中具有各种各样的图形界面元素，这些界面元素之间通过信号和事件机制协调工作。本章将详细介绍信号和事件的相关概念和编程细节，具体内容如下：

- 信号；
- GTK 事件。

21.1　信号

当用户对窗口上的一个元件进行操作时，这个元件将会向系统发出一个信息。例如，当用户单击一个按钮时，按钮会向系统发出一个单击事件的信息。系统如果需要处理这个信息，需要设置一个信号处理函数。

注意：默认情况下，元件没有事件处理功能，即当元件发生某个事件时，不能完成任何动作。必须在程序中为元件添加事件处理函数。

21.1.1　添加信号

信号的作用是对一个元件添加一个用户交互功能。g_signal_connect 函数可以把一个信号处理函数添加到一个元件上。这个函数的使用方法如下所示。

```
01  #define g_signal_connect(instance, detailed_signal, c_handler, data);
```

在参数列表中，instance 是一个元件的指针。detailed_signal 表示这个信号的类型。c_handler 表示这个信号需要执行的函数。data 表示执行函数时的参数。下面的实例对一个按钮添加一个事件。当单击这个按钮时，会关闭当前的窗口。

【范例 21-1】使用 g_signal_connect 函数为元件添加信号处理函数，示例代码如下所示。

示例代码 21-1

```
01  #include <gtk/gtk.h>                                          /*包含头文件。*/
02
03  void on_clicked(GtkWidget *widget, gpointer data)             /*事件管理函数。*/
04  {
05     gtk_main_quit();                                           /*关闭窗口。*/
06  }
07
08  int main(int argc, char **argv)
09  {
10    GtkWidget *window;                                         /*定义相关变量。*/
11    GtkWidget *button;
12    char title[]="button .close .";
13
14    gtk_init(&argc, &argv);                                    /*建立和处理窗口。*/
15    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
16    gtk_window_set_title (GTK_WINDOW (window),title);
17    gtk_window_set_default_size (GTK_WINDOW (window), 250, 150);
18
19    button=gtk_button_new_with_label("Close the window."); /*建立一按钮。*/
20    g_signal_connect(G_OBJECT(button),  "clicked",    G_CALLBACK(on_clicked),
NULL);
                                                          /*为按钮添加事件。*/
21
22    gtk_container_add(GTK_CONTAINER(window), button);  /*在窗口上添加按钮。*/
23
24    gtk_widget_show_all(window);
25    gtk_main ();
26    return 0;
27  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 21.1.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

注意：必须在 X window 下的终端中输入运行命令（不能在纯文本界面下输入运行命令），否则无法显示图形。

程序的运行界面如图 21-1 所示。当单击这个按钮时，这个窗口会自动关闭。

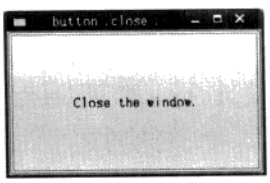


图 21-1　按钮关闭一个窗口

【代码解析】

- 代码第 03~06 行定义了事件处理函数，当点击按钮时将触发并执行此函数。
- 代码第 19 行创建了一个按钮，代码第 20 行为新建的按钮添加事件，代码第 22 行将按钮添加到窗口上。

21.1.2　事件函数的参数

在上一节的实例中，当用户单击按钮时，系统会调用一个函数。系统默认为这个函数提供两个参数。本节将讲解这两个参数的含义和使用方法。这个函数的声明如下所示。

```
01  void on_clicked(GtkWidget *widget, gpointer data)
```

从函数的声明可知，这个函数没有返回值，有两个参数。这两个参数的使用方法如下所示。

- widget：表示当前发出信号的元件的指针。在程序中，可以用这个指针来表示发出信号的元件。
- data：data 是 g_signal_connect 函数调用这个函数时的最后一个参数。这个参数可以实现窗口与被调用函数之间的数据交互。

下面是事件处理函数的实例，程序的代码如下所示。

提示：码中 g_signal_connect 函数调用事件处理函数时，把窗口的指针作为函数的参数。在处理函数中，用窗口的指针来改变窗口的标题。

【范例 21-2】事件函数 on_clicked 的参数应用举例，示例代码如下所示。

示例代码 21-2

```
01  #include <gtk/gtk.h>                                    /*包含头文件。*/
```

```
02
03  void on_clicked(GtkWidget *widget, gpointer data)    /*信号处理函数。*/
04  {
05      char txt[]="this is the new title .";            /*新标题。*/
06      gtk_window_set_title (GTK_WINDOW (data),txt);    /*设置标题。*/
07  }
08
09  int main(int argc, char **argv)
10  {
11    GtkWidget *window;                                 /*定义相关的变量。*/
12    GtkWidget *button;
13    char title[]="set title";
14
15    gtk_init(&argc, &argv);                            /*建立和处理窗口。*/
16    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
17    gtk_window_set_title (GTK_WINDOW (window),title);
18    gtk_window_set_default_size (GTK_WINDOW (window), 250, 150);
19
20    button=gtk_button_new_with_label("set title .");   /*建立按钮。*/
21    g_signal_connect(G_OBJECT(button),  "clicked",     G_CALLBACK(on_clicked),
window);
22                                                       /*为按钮添加事件。*/
23    gtk_container_add(GTK_CONTAINER(window), button);  /*窗口上添加按钮。*/
24
25    gtk_widget_show_all(window);
26    gtk_main ();
27    return 0;
28  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 21.2.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 21-2 所示。当单击这个按钮时，会改变当前窗口的标题，如图 21-3 所示。

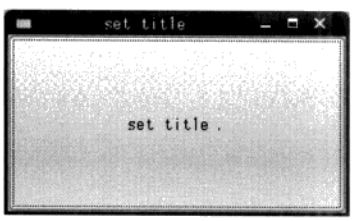


图 21-2　窗口中的按钮

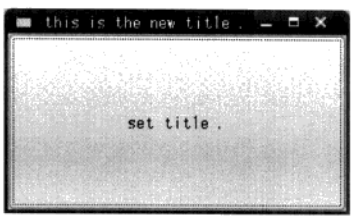


图 21-3　单击按钮改变窗口的标题

【代码解析】代码第 03~06 行定义了事件处理函数，当单击按钮时将触发并执行此函数。在函数内部调用 gtk_window_set_title 函数设置窗口的标题。其他代码与示例代码 21-1 相同。

21.1.3　在函数中使用全局变量

从上一节的鼠标单击事件和处理函数可知，事件函数可以访问 g_signal_connect 函数的一个参数。然后可以用这个参数来访问其他的元件。没有传递过来的变量，事件处理函数是不能访问的。为了解决这个问题，可以把程序中需要供其他函数访问的变量定义为全局变量。这样程序中所有的函数都可以访问这个变量。定义全局变量的方法，是在程序的最前面程序的外部

定义这个变量。

注意：当用户单击按钮时，可以访问全局变量，并且可以根据全局变量指针访问程序中的其他元件。

【范例 21-3】在事件函数 on_clicked 中使用全局变量，示例代码如下所示。

示例代码 21-3

```
01  #include <gtk/gtk.h>
02  #include <stdlib.h>                                    /*包含头文件。*/
03
04  int i ;                                               /*定义全局变量。*/
05  GtkWidget *window;
06  GtkWidget *button;
07
08  void on_clicked(GtkWidget *widget, gpointer data)     /*信号处理函数。*/
09  {
10      char a[20],b[20];
11      i++;                                              /*计数自加。*/
12      gcvt((float)i,3,a);                               /*i 转换成字符串。*/
13      strcat(a, " times");                              /*添加字符串。*/
14      strcpy(b,"you have clicked ");                    /*复制字符串。*/
15      strcat(b,a);                                      /*添加字符串。*/
16      gtk_window_set_title (GTK_WINDOW (window),b);     /*设置标题。*/
17  }
18
19  int main(int argc, char **argv)
20  {
21    char title[]="button";
22    gtk_init(&argc, &argv);
23    window=gtk_window_new(GTK_WINDOW_TOPLEVEL);         /*建立和处理窗口。*/
24    gtk_window_set_title (GTK_WINDOW (window),title);   /*设置标题。*/
25    gtk_window_set_default_size (GTK_WINDOW (window), 250, 150);
26
27    button=gtk_button_new_with_label(" click ");        /*添加按钮。*/
28    g_signal_connect(G_OBJECT(button),  "clicked",     G_CALLBACK(on_clicked),
window);
29                                                        /*对按钮添加事件。*/
30    gtk_container_add(GTK_CONTAINER(window), button);   /*按钮添加到窗口上。*/
31
32    gtk_widget_show_all(window);
33    gtk_main ();
34    return 0;
35  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 21.3.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 21-4 所示。

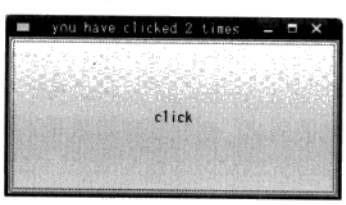


图 21-4　标题显示的计数

【代码解析】代码第 04 行定义了一个全局变量 i，代码第 08~17 行定义了事件处理函数，当点击按钮时将触发并执行此函数。在函数内部调用访问了全局变量 i，并设置窗口的标题。其他代码与示例代码 21-1 相同。

21.2　GTK 事件

事件指的是用户与图形界面程序的交互。窗口程序中的每一个元件，都有不同的方式接受用户的交互发出一个信号。例如鼠标点击、鼠标移动、键盘交互等操作都可以产生事件。一个按钮所有的事件类型与含义如下所示。

- activate：激活的时候发生。
- clicked：单击以后发生。
- enter：鼠标指针进入这个按钮以后发生。
- leave：鼠标离开按钮以后发生。
- pressed：鼠标按下以后发生。
- released：鼠标释放以后发生。

在 g_signal_connect 函数中，使用这些事件的不同名称来区分事件。例如下面的代码，表示在鼠标离开按钮以后发生的动作。

```
01  g_signal_connect(G_OBJECT(button),   "leave",      G_CALLBACK(on_clicked),
window);
```

21.3　实例：加法计算器

本节讲解 GTK 程序实例，在一个界面中用两个文本框输入数据，实现加法运算。在这个程序中，需要注意界面的实现方法和信号事件的处理。

21.3.1　程序功能的分析

要在程序中实现加法运算，首先要设计出程序的界面。需要新建一个窗口，窗口中添加一个四行一列的表格。表格中分别将入两个文本框、一个按钮、一个标签。两个文本框用于数据的输入，按钮用于实现用户单击。标签用以输出计算以后的结果。

提示：进行计算时，需要用 gtk_entry_get_text 函数取得文本框中的数据。取得的数据是字符串，需要把这个字符串转换成浮点型数字进行计算。然后把结果转换成字符串，再用 gtk_label_set_text 函数设置标签的文本。

21.3.2　程序实现

根据上一节的分析，编写的程序代码如下所示。在程序中，窗口、文本框、按钮、标签等元件的指针定义成全局变量。这样就可以在函数内部访问这些全局变量。

【范例 21-4】编程实现加法计算器，示例代码如下所示。

示例代码 21-4

```
01  #include <gtk/gtk.h>
02  #include <stdio.h>
03  #include <stdlib.h>                              /*包含头文件。*/
04
05  GtkWidget *window;                               /*定义全局变量。*/
06  GtkWidget *txt1;
07  GtkWidget *txt2;
```

```
GtkWidget *table;
GtkWidget *button;
GtkWidget *label;

void on_clicked(GtkWidget *widget, gpointer data)        /*定义按钮事件。*/
{
    char a1[10];                                          /*定义函数的变量。*/
    char a2[10];
    char a3[10];
    float a,b,c;

    strcpy(a1,gtk_entry_get_text(GTK_ENTRY(txt1)));       /*取得输入的字符串。*/
    strcpy(a2,gtk_entry_get_text(GTK_ENTRY(txt2)));

    a=atof(a1);                                          /*字符串转换成浮点型。*/
    b=atof(a2);
    c=a+b;                                               /*两个数做加法。*/

    gcvt(c,7,a3);                                        /*浮点型转换成字符串。*/
    gtk_label_set_text(GTK_LABEL(label),a3);             /*设置标签的文本。*/
}
int main(int argc, char **argv)
{
  char title[]="my window";
  gtk_init(&argc, &argv);

  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);/*建立与处理窗口。*/
  gtk_window_set_title (GTK_WINDOW (window),title);
  gtk_window_set_default_size (GTK_WINDOW (window), 250, 300);
  g_signal_connect(G_OBJECT(window), "delete_event",  G_CALLBACK(gtk_main_
quit), NULL);

  txt1 = gtk_entry_new ();
  gtk_widget_show (txt1);
  txt2 = gtk_entry_new ();
  gtk_widget_show (txt2);
  button=gtk_button_new_with_label("OK");
  gtk_widget_show (button);
  label=gtk_label_new(" Result. ");
  gtk_widget_show (label);              /*建立两个文本框、一个按钮、一个标签。*/

  table = gtk_table_new (4, 1, FALSE);                    /*建立一个表格。*/
  gtk_widget_show (table);
  gtk_container_add (GTK_CONTAINER (window), table);

  gtk_table_attach (GTK_TABLE (table), txt1,0, 1,0, 1,    /*表格中添加文本框。*/
                (GtkAttachOptions) (GTK_FILL),
                (GtkAttachOptions) (0), 11, 11);

  gtk_table_attach (GTK_TABLE (table), txt2,0, 1,1, 2,    /*表格中添加文本框。*/
                (GtkAttachOptions) (GTK_FILL),
                (GtkAttachOptions) (0), 11,11);
  gtk_table_attach (GTK_TABLE (table), button,0, 1,2,3,   /*表格中添加按钮。*/
                (GtkAttachOptions) (GTK_FILL),
                (GtkAttachOptions) (0), 11,11);

  gtk_table_attach (GTK_TABLE (table), label,0, 1,3,4,    /*表格中添加标签。*/
                (GtkAttachOptions) (GTK_FILL),
                (GtkAttachOptions) (0), 11,11);

  g_signal_connect(G_OBJECT(button), "clicked",  G_CALLBACK(on_clicked),NULL);
```

```
68                                                    /*按钮添加事件。*/
69
70      gtk_widget_show_all(window);
71      gtk_main ();
72      return 0;
73  }
```

【运行结果】输入下面的命令，编译这个程序。

```
gcc 21.4.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 21-5 所示。在程序中输入两个数字，然后单击“OK”按钮。程序会把这两个数相加，结果显示在标签中，如图 21-6 所示。

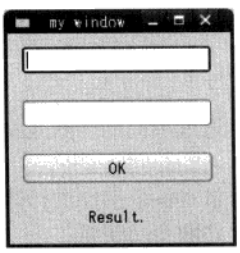


图 21-5　程序的界面

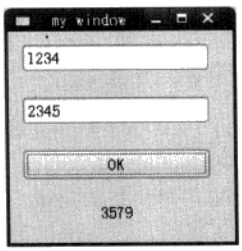


图 21-6　程序实现两个数字的相加

【代码解析】

- 代码第 05~10 行定义程序所需的窗口及元件的指针变量，这些变量都是全局变量。
- 代码第 12~28 行定义事件函数，当按下按钮时触发此函数的代码。
- 在主函数中通过表格进行窗口的布局，并将创建好的元件添加到窗口中。最后为按钮添加事件函数。

21.4　综合练习

本节将讲述一个计算器程序实例。在这个实例中，用 GTK 的窗口、按钮、文本框、表格等元件建立一个计算器界面。用户鼠标单击计算器上的按钮时，可以实现数据的加减乘除运算。

21.4.1　程序的功能分析

计算器程序主要包括界面、用户输入、数据运算三个方面的内容。本节将分析计算器程序的功能与要求。在编写复杂程序以前，需要对这个程序的功能进行认真分析，将复杂的功能拆解成的棋坛模块。程序的界面需要用下面这些方法来实现。

- 需要有一个窗口，用来容纳所有的元件。
- 需要有一个文本框，用来接收和显示用户的数据输入。
- 需要有 0~9 一共 10 个按钮，用户可以单击这些按钮实现数据的输入。
- 需要有+、-、*、/4 个按钮，实现 4 种运算的输入。
- 需要有一个复位按钮，用来清除用户的输入。
- 需要有一个等于号按钮，完成数据的计算。

- 需要有一个小数点按钮，实现小数点的输入。

程序的主要输入是单击按钮输入数据，这个交互需要实现下面的功能。

- 用户单击数字按钮时，将数字添加到文本框中。
- 用户单击小数点按钮时，如果没有小数点，则在文本框中添加一个小数点。如果已经有小数点，则不添加。
- 用户单击运算符号按钮时，需要记录数据的运算方法。
- 用户单击等于号按钮时，需要完成数据的计算。

数据处理是程序运算功能的最终实现模块。数据运算模块需要完成下面的功能。

- 用户单击运算符号时，需要有一个变量来存储运算的方法。
- 用户完成一个数字的输入时，需要有一个变量来存储这个输入的数据。
- 取得的数据是一个字符串，需要将这个字符串轮换成浮点型数。
- 数据运算的结果是浮点型变量，需要转换成字符串以后设置成文本框的文本。

21.4.2 程序的功能模块

上一节分析了这个程序需要完成的功能，这些功能是通过不同的函数来实现的。这个程序的函数和功能模块如下所示。

（1）头文件与全局变量的定义。程序需要在不同函数里对各个元件进行访问，这些访问是通过全局变量来实现的。在程序的最开始部分需要定义程序的全局变量。

（2）用户单击输入数字函数。用户单击数字按钮时，需要将这个按钮上的数字添加到文本框的后面。这个函数定义方法如下所示。

```
01  void input(GtkWidget *widget, gpointer data)
```

（3）加法运算函数。在这个程序中，加法和等于号按钮是同一个按钮，用户单击这个按钮时，需要完成加法运算或等于号运算。

```
01  void jia(GtkWidget *widget, gpointer data)
```

（4）减法运算函数，完成减法运算的功能。

```
01  void jian(GtkWidget *widget, gpointer data)
```

（5）乘法运算函数，完成乘法运算的功能。

```
01  void cheng(GtkWidget *widget, gpointer data)
```

（6）除法运算函数，实现除法运算的功能。

```
01  void chu(GtkWidget *widget, gpointer data)
```

（7）小数点输入函数，当用户单击小数点按钮时，这个函数对事件进行处理。

```
01  void dot(GtkWidget *widget, gpointer data)
```

（8）清除数据函数，这个函数可以清除所有已经输入的数据，程序回到初始状态。

```
01  void clear(GtkWidget *widget, gpointer data)
```

（9）界面构建函数，这个函数实现程序用户界面的构建。

```
01  void interface()
```

（10）添加事件函数，作用是添加元件的交互事件。

```
01  void addsignal()
```

（11）主函数，作用是完成所有模块和功能的调用。

```
01  main
```

21.4.3　包含头文件与定义全局变量

在程序的最前面，需要包含程序需要的头文件。GTK 图形界面需要包含 gtk.h 头文件。数据处理需要包含 stdlib.h 头文件。

提示：为了在函数中调用不同的元件和变量，需要将元件指针和某些变量定义成全局变量。

【范例 21-5】包含必需的头文件与定义全局变量，示例代码如下所示。

示例代码 21-5

```
#include <stdlib.h>
#include <gtk/gtk.h>

float a,b ;                                  /*定义两个参与运算的变量。*/
int hasdot;                                  /*是否有小数点。*/
int method;                                  /*运算方法。*/
GtkWidget *window;                           /*这一部分是定义元件。*/
GtkWidget *table1;                           /*表格。*/
GtkWidget *entry1;                           /*文本框。*/
GtkWidget *table2;
GtkWidget *button1;                          /*按钮。*/
GtkWidget *button2;
GtkWidget *button3;
GtkWidget *button4;
GtkWidget *button5;
GtkWidget *button6;
GtkWidget *button7;
GtkWidget *button8;
GtkWidget *button9;
GtkWidget *button10;
GtkWidget *button11;
GtkWidget *button12;
GtkWidget *button13;
GtkWidget *button14;
GtkWidget *button15;
GtkWidget *button16;
```

21.4.4　单击按钮输入函数

用户单击一个数据按钮，需要将这个按钮上的文本附加到文本框的最后。可以用全局变量指针来访问文本框和按钮。gtk_entry_append_text 函数可以在一个文本框原有的文本上附加文本。

【范例 21-6】定义单击按钮输入函数，示例代码如下所示。

示例代码 21-6

```
void input(GtkWidget *widget, gpointer data)
{
    gtk_entry_append_text(GTK_ENTRY(entry1),gtk_button_get_label(widget));
}
```

21.4.5　加法运算函数

这个加法运算函数同时实现等于号的功能。如果用户已经输入了其他的运行符号，则需要在这个函数中完成加、减、乘、除四种运算。

注意：如果是第一次输入运算符，则需要对已经输入的数据进行存储，第二次输入运算符则表示进行计算。

【范例 21-7】定义加法运算函数，示例代码如下所示。

示例代码 21-7

```
void jia(GtkWidget *widget, gpointer data)
{
    char num[20];
    strcpy(num,gtk_entry_get_text(GTK_ENTRY(entry1)));   /*取得文本框的内容。*/
    if(a==0)                              /*如果没有第一个数，则存储为第一个
数。*/
    {
      gtk_entry_set_text(GTK_ENTRY(entry1),"");          /*清空文本框。*/
      a=atof(num);                                       /*转换成浮点型。*/
    }
    else
    {
       b=atof(num);                                    /*如果 a 已经有值，则进行运算。*/
       switch(method)
       {
           case  0 :
             a=a+b; break ;
           case 1:
             a=a-b; break ;
           case  2 :
             a=a*b; break ;
           case  3 :
             a=a/b; break ;
        }
    gcvt(a,5,num);                                     /*运算结果转换成字符串。*/
    b=0;
    method=0;
    gtk_entry_set_text(GTK_ENTRY(entry1),num);  /*显示结果。*/
    }
    hasdot=0;                                          /*表示已经没有小数点。*/
}
```

21.4.6 减法运算函数

这个函数的作用是实现减法的运算。

注意：在函数中，需要注意处理数据输入情况和数据类型转换的问题，需要将文本框中的字符串转换成浮点型变量。

【范例 21-8】定义减法运算函数，示例代码如下所示。

示例代码 21-8

```
void jian(GtkWidget *widget, gpointer data)
{
    method=1;                                          /*运算方法。*/
    char num[20];
    strcpy(num,gtk_entry_get_text(GTK_ENTRY(entry1)));
    if(a==0)                                           /*没有数据输入的情况。*/
    {
      gtk_entry_set_text(GTK_ENTRY(entry1),"");
```

```
09          a=atof(num);                                    /*字符串转换成浮点型。*/
10        }
11        else
12        {
13            b=atof(num);                                  /*如果 a 有数据则存储到 b。*/
14        }
15        hasdot=0;                                         /*设置没有小数点。*/
16    }
```

21.4.7　乘法运算函数

这个函数的作用是乘法运算的功能。程序的主要处理是取得文本框中输入的内容，然后判断结果 a 与 b 变量的情况，判断存储到哪一个变量。

【范例 21-9】定义乘法运算函数，示例代码如下所示。

示例代码 21-9

```
01  void cheng(GtkWidget *widget, gpointer data)
02  {
03     method=2;                                            /*设置运算方法。*/
04     char num[20];
05      strcpy(num,gtk_entry_get_text(GTK_ENTRY(entry1)));
06      if(a==0)
07      {
08        gtk_entry_set_text(GTK_ENTRY(entry1),"");
09        a=atof(num);
10      }
11      else
12      {
13          b=atof(num);
14      }
15      hasdot=0;
16  }
```

21.4.8　除法运算函数

这个函数的作用是进行除法运算，处理方法与上面两种运算的处理方法是相同的。

【范例 21-10】定义除法运算函数，示例代码如下所示。

示例代码 21-10

```
01  void chu(GtkWidget *widget, gpointer data)
02  {
03     method=3;                                            /*运算方法。*/
04     char num[20];
05      strcpy(num,gtk_entry_get_text(GTK_ENTRY(entry1)));   /*取得文本框中输入的
内容。*/
06      if(a==0)
07      {
08        gtk_entry_set_text(GTK_ENTRY(entry1),"");
09        a=atof(num);
10      }
11      else
12      {
13          b=atof(num);
14      }
15      hasdot=0;
16  }
```

21.4.9 输入小数点函数

这个函数的作用是完成小数点的输入。因为在一个数字中不可以有两个小数点，所以要判断数据中是不是已经有小数点以后才能添加小数点。程序中已经定义了一个变量 hasdot 用来表示有无小数点，只需要对这个变量的值进行判断即可。

【范例 21-11】定义输入小数点函数，示例代码如下所示。

示例代码 21-11

```
void dot(GtkWidget *widget, gpointer data)
{
    if(hasdot==0)                                    /*没有小数点则添加一个小数点。*/
    {
      gtk_entry_append_text(GTK_ENTRY(entry1),gtk_button_get_label(widget));
      hasdot=1;                                      /*表示已经有一个小数点。*/
    }
}
```

21.4.10 清除数据函数

清除数据函数的作用是清除文本框中已经输入的数据，完成程序的初始化，在函数中需要完成下面这些操作。

- 设置文本框的内容为空字符串。
- 设置已经没有小数点。
- 设置运算方法为加法。
- 设置两个运算变量为 0。

程序的代码如下所示。

【范例 21-12】定义清除数据函数，示例代码如下所示。

示例代码 21-12

```
void clear(GtkWidget *widget, gpointer data)
{
    gtk_entry_set_text(GTK_ENTRY(entry1),"");
    hasdot=0;
    a=0;
    b=0;
    method=0;
}
```

21.4.11 界面构建函数

界面构建指的是建立程序中需要的各种元件，对各种元件进行排列，生成一个用户图形界面。这个程序的主要内容是生成窗口，在窗口中用表格容纳窗口中的各种元件。

【范例 21-13】界面构建函数，示例代码如下所示。

示例代码 21-13

```
void interface()
{
  char title[]="mywin";                              /*建立和处理窗口。*/
  window=gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window),title);

```

```
07    table1 = gtk_table_new (2, 1, FALSE);                    /*建立一个表格。*/
08    gtk_widget_show (table1);
09    gtk_container_add (GTK_CONTAINER (window), table1);
10
11    entry1 = gtk_entry_new ();                               /*建立文本框。*/
12    gtk_widget_show (entry1);
13    gtk_table_attach (GTK_TABLE (table1), entry1, 0, 1, 0, 1,
14                  (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
15                  (GtkAttachOptions) (0), 0, 0);
16
17    table2 = gtk_table_new (4, 4, FALSE);                    /*建立一个表格。*/
18    gtk_widget_show (table2);
19    gtk_table_attach (GTK_TABLE (table1), table2, 0, 1, 1, 2,
20                  (GtkAttachOptions) (GTK_FILL),
21                  (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0, 0);
22
23   button1 = gtk_button_new_with_mnemonic ("1");             /*添加和处理数字按钮。*/
24    gtk_widget_show (button1);
25    gtk_table_attach (GTK_TABLE (table2), button1, 0, 1, 0, 1,
26                  (GtkAttachOptions) (GTK_FILL),
27                  (GtkAttachOptions) (0), 0, 0);
28    gtk_widget_set_size_request (button1, 50, 30);
29
30    button2 = gtk_button_new_with_mnemonic ("2");             /*添加和处理数字按钮。*/
31    gtk_widget_show (button2);
32    gtk_table_attach (GTK_TABLE (table2), button2, 1, 2, 0, 1,
33                  (GtkAttachOptions) (GTK_FILL),
34                  (GtkAttachOptions) (0), 0, 0);
35    gtk_widget_set_size_request (button2, 50, 30);
36
37
38    button3 = gtk_button_new_with_mnemonic ("3");             /*添加和处理数字按钮。*/
39    gtk_widget_show (button3);
40    gtk_table_attach (GTK_TABLE (table2), button3, 2, 3, 0, 1,
41                  (GtkAttachOptions) (GTK_FILL),
42                  (GtkAttachOptions) (0), 0, 0);
43    gtk_widget_set_size_request (button3, 50, 30);
44
45    button4 = gtk_button_new_with_mnemonic ("4");             /*添加和处理数字按钮。*/
46    gtk_widget_show (button4);
47    gtk_table_attach (GTK_TABLE (table2), button4, 0, 1, 1, 2,
48                  (GtkAttachOptions) (GTK_FILL),
49                  (GtkAttachOptions) (0), 0, 0);
50
51    button5 = gtk_button_new_with_mnemonic ("5");             /*添加和处理数字按钮。*/
52    gtk_widget_show (button5);
53    gtk_table_attach (GTK_TABLE (table2), button5, 1, 2, 1, 2,
54                  (GtkAttachOptions) (GTK_FILL),
55                  (GtkAttachOptions) (0), 0, 0);
56
57    button6 = gtk_button_new_with_mnemonic ("6");             /*添加和处理数字按钮。*/
58    gtk_widget_show (button6);
59    gtk_table_attach (GTK_TABLE (table2), button6, 2, 3, 1, 2,
60                  (GtkAttachOptions) (GTK_FILL),
61                  (GtkAttachOptions) (0), 0, 0);
62
63    button7 = gtk_button_new_with_mnemonic ("7");             /*添加和处理数字按钮。*/
64    gtk_widget_show (button7);
65    gtk_table_attach (GTK_TABLE (table2), button7, 0, 1, 2, 3,
66                  (GtkAttachOptions) (GTK_FILL),
67                  (GtkAttachOptions) (0), 0, 0);
68
```

```
    button8 = gtk_button_new_with_mnemonic ("8");       /*添加和处理数字按钮。*/
    gtk_widget_show (button8);
    gtk_table_attach (GTK_TABLE (table2), button8, 1, 2, 2, 3,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button9 = gtk_button_new_with_mnemonic ("9");       /*添加和处理数字按钮。*/
    gtk_widget_show (button9);
    gtk_table_attach (GTK_TABLE (table2), button9, 2, 3, 2, 3,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button10 = gtk_button_new_with_mnemonic ("0");      /*添加和处理数字按钮。*/
    gtk_widget_show (button10);
    gtk_table_attach (GTK_TABLE (table2), button10, 1, 2, 3, 4,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button11 = gtk_button_new_with_mnemonic (".");      /*添加和处理小数点按钮。*/
    gtk_widget_show (button11);
    gtk_table_attach (GTK_TABLE (table2), button11, 0, 1, 3, 4,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button12 = gtk_button_new_with_mnemonic ("+ =");    /*添加和处理加法按钮。*/
    gtk_widget_show (button12);
    gtk_table_attach (GTK_TABLE (table2), button12, 2, 3, 3, 4,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button13 = gtk_button_new_with_mnemonic ("-");      /*添加和处理减法按钮。*/
    gtk_widget_show (button13);
    gtk_table_attach (GTK_TABLE (table2), button13, 3, 4, 0, 1,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);
    gtk_widget_set_size_request (button13, 50, -1);

    button14 = gtk_button_new_with_mnemonic ("X");      /*添加和处理乘法按钮。*/
    gtk_widget_show (button14);
    gtk_table_attach (GTK_TABLE (table2), button14, 3, 4, 1, 2,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button15 = gtk_button_new_with_mnemonic ("/");      /*添加和处理除法按钮。*/
    gtk_widget_show (button15);
    gtk_table_attach (GTK_TABLE (table2), button15, 3, 4, 2, 3,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);

    button16 = gtk_button_new_with_mnemonic ("CE");     /*添加和处理复位按钮。*/
    gtk_widget_show (button16);
    gtk_table_attach (GTK_TABLE (table2), button16, 3, 4, 3, 4,
                      (GtkAttachOptions) (GTK_FILL),
                      (GtkAttachOptions) (0), 0, 0);
}
```

21.4.12 添加事件函数

窗口的按钮添加信号与事件才能完成程序的交互。本节将对上一节中添加的按钮添加事件函数，完成程序的交互功能。

【范例 21-14】为元件添加事件函数，示例代码如下所示。

示例代码 21-14

```
01  void addsignal()
02  {
03                                                    /*下面的 10 个按钮实现数字的输入。*/
04    g_signal_connect(G_OBJECT(button1), "clicked",  G_CALLBACK(input),NULL);
05    g_signal_connect(G_OBJECT(button2), "clicked",  G_CALLBACK(input),NULL);
06    g_signal_connect(G_OBJECT(button3), "clicked",  G_CALLBACK(input),NULL);
07    g_signal_connect(G_OBJECT(button4), "clicked",  G_CALLBACK(input),NULL);
08    g_signal_connect(G_OBJECT(button5), "clicked",  G_CALLBACK(input),NULL);
09    g_signal_connect(G_OBJECT(button6), "clicked",  G_CALLBACK(input),NULL);
10    g_signal_connect(G_OBJECT(button7), "clicked",  G_CALLBACK(input),NULL);
11    g_signal_connect(G_OBJECT(button8), "clicked",  G_CALLBACK(input),NULL);
12    g_signal_connect(G_OBJECT(button9), "clicked",  G_CALLBACK(input),NULL);
13    g_signal_connect(G_OBJECT(button10), "clicked",  G_CALLBACK(input),NULL);
14                                                    /*下面的按钮实现小数点的输入。*/
15    g_signal_connect(G_OBJECT(button11), "clicked",  G_CALLBACK(dot),NULL);
16                                                    /*下面的按钮实现加减乘除的运算。*/
17    g_signal_connect(G_OBJECT(button12), "clicked",  G_CALLBACK(jia),NULL);
18    g_signal_connect(G_OBJECT(button13), "clicked",  G_CALLBACK(jian),NULL);
19    g_signal_connect(G_OBJECT(button14), "clicked",  G_CALLBACK(cheng),NULL);
20    g_signal_connect(G_OBJECT(button15), "clicked",  G_CALLBACK(chu),NULL);
21                                                    /*下面的按钮实现复位功能。*/
22    g_signal_connect(G_OBJECT(button16), "clicked",  G_CALLBACK(clear),NULL);
23    g_signal_connect(G_OBJECT(window), "delete_event",  gtk_main_quit, NULL);
24  }
```

21.4.13　主函数

主函数可以实现上面所有功能的调用，使程序可以编译运行。在主程序中，需要对某些变量进行初始赋值，然后依次调用函数完成构建界面、添加事件的功能。主函数的代码如下所示。

【范例 21-15】主函数，示例代码如下所示。

示例代码 21-15

```
01  int main (int argc, char *argv[])
02  {
03    a=0;
04    b=0;
05    hasdot=0;
06    gtk_set_locale ();
07    gtk_init (&argc, &argv);                    /*建立窗口。*/
08    method=0;                                   /*运算方法。*/
09    interface();                                /*建立界面。*/
10    addsignal();                                /*添加事件。*/
11    gtk_widget_show(window);
12    gtk_main ();
13    return 0;
14  }
```

21.4.14　程序的编译和运行

前面多个章节的程序是不同的模块。在 C 程序中，不同函数和模块的内容是互相独立互不影响的。将前面几节的程序写在一个文件中，即可完成这个程序的编写。本节对这些程序进行编译和运行。输入下面的命令，编译这个程序。

```
gcc 21.5.c `pkg-config --cflags --libs gtk+-2.0`
```

输入下面的命令，运行这个程序。

```
./a.out
```

程序的运行界面如图 21-7 所示。程序的界面是模拟实际界面的计算器。用户用鼠标单击不同的按钮，可以实现数据的输入和运算。

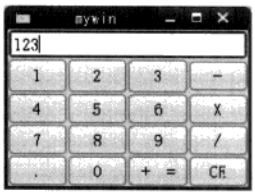


图 21-7　计算器程序

21.5　小结

本章讲述了 GTK 元件的信号与事件。在图形界面程序中，程序的功能是通过信号和事件来完成的。在本章的学习中，需要理解信号与事件的原理，信号与事件的处理方法。最后一节的计算器实例，是信号与事件的综合应用，读者通过这一实例可以学习和理解程序中的事件处理过程。

21.6　习题

一、填空题

1．_____的作用是对一个元件添加一个用户交互功能。

2．当要把一个信号处理函数添加到一个元件上时应该调用的函数时_____。

3．没有传递过来的变量，事件处理函数是不能访问的。为了解决这个问题，可以把程序中需要供其他函数访问的变量作为_____。

4．当用户单击按钮，系统会调用一个函数，系统默认为个这函数提供_____个参数。

二、选择题

1．一个按钮不支持的事件类型有（　　）。

A．activate：激活的时候发生

B．clicked：单击以后发生

C．enter：鼠标指针进入这个按钮以后发生

D．closed：关闭后发生

2．可以把一个信号处理函数添加到一个元件上的函数是（　　）。

A．g_signal_connect　　B．g_signal_connection

C．g_signal_connected　　D．g_signal_connecting

三、简答题

1．简述 g_signal_connect 函数的参数所代表的含义。

2．简述为什么在事件处理函数中要使用全局变量。

3．在编写事件处理函数时，系统默认为这个函数提供两个参数，简述这两个参数的含义及用法。

四、编程题

1．编写程序实现一个简单的加法计算器。在一个界面中用两个文本框输入数据，实现加法运算。

2．编写一个计算器程序程序。用 GTK 的窗口、按钮、文本框、表格等元件建立一个计算器界面。用户鼠标单击计算器上的按钮时，实现数据的加减乘除运算。