



清华大学出版社
北 京

内 容 简 介

本书是一本与众不同的 C 语言学习读物，是一本化繁为简，把“抽象”问题“具体”化，把复杂问题简单化的书。在本书中，避免出现云山雾罩、晦涩难懂的讲解，代之以轻松活泼、由浅入深的剖析，这必将使每一位阅读本书的读者少走弯路，快速上手，从而建立学习 C 程序设计的信心。

本书 15 章，分为 5 篇，从实用出发，由遇到的问题引出解决问题的方法来系统讲述 C 语言的各个特性及程序设计的基本方法。本书内容主要包括常量、变量、程序结构、数组、字符串、指针、结构体、共同体、枚举类型、函数、局部变量和全局变量、预处理命令和文件等一些非常重要的知识。通过阅读本书，读者可以在较短的时间内理解 C 程序设计的各个重要概念和知识点，为进一步学习打好基础。

本书配带 1 张 DVD 光盘，收录了本书重点内容的教学视频和涉及的源代码，光盘中还赠送了大量超值的 C 语言进阶视频。

本书最适合没有基础的 C 语言入门新手阅读；对于有一定基础的读者，可通过本书进一步理解 C 语言的各个重要知识点和概念；对于大、中专院校的学生和培训班的学员，本书也不失为一本好教材。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目 (CIP) 数据

C 语言入门很简单 / 马磊等编著. —北京：清华大学出版社，2012.6
ISBN 978-7-302-28102-3

I. ①C… II. ①马… III. ①C 语言—程序设计—基本知识 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2012) 第 030491 号

责任编辑：夏兆彦

封面设计：欧振旭

责任校对：徐俊伟

责任印制：何 芊

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京鑫丰华彩印有限公司

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185mm×260mm 印 张：21.5 字 数：540 千字
(附光盘 1 张)

版 次：2012 年 6 月第 1 版 印 次：2012 年 6 月第 1 次印刷

印 数：1~5000

定 价：49.00 元

产品编号：045285-01

前言

C 语言是一门历史悠久、博大精深的程序设计语言。它对计算机技术的发展起到了极其重要的促进作用，而且这种促进作用一直在持续并将继续持续下去。它从产生之时就肩负了很多重要使命，开发操作系统、开发编译器、开发驱动程序……它可深可浅，浅到你可以用几周的时间掌握它的基本概念和功能，深到几乎可以解决计算机中的大部分问题。

C 语言几乎是每一个致力于程序设计人员的必学语言。但从学习之初，它往往给读者以神秘而艰难的感觉。下面给出 C 语言入门新手的一些典型感受。

- ❑ 术语太难理解。C 语言对于没有基础的人来说比较抽象，因为一些专业术语对于初学者来说不好理解，更别说写程序了。
- ❑ 看不到界面。C 语言的编写是没有界面的，导致初学者很难理解写出来的程序是什么样，如何才能看出效果。
- ❑ 写的程序很长。对于实际开发人员，C 语言的使用概率比较小，要想实现某个效果，其程序很长，导致不容易完成。

但实际上，C 语言并非想象的那么难。它的很多优点让它一直保持着魅力而在程序语言中永葆青春。总结起来，主要体现在以下几个方面。

- ❑ C 语言是基础语言，容易理解，对初学者没有太大的限制。
- ❑ 它很灵活，一件事往往可以通过多种方式来实现。
- ❑ C 语言虽然没有界面，但是 C 程序语句看起来很直观，容易理解。
- ❑ C 语言没有那么多的库函数，没有“对象”与“类”之说，实现起来很方便。
- ❑ C 语言执行效率高，更多地执行了计算机底层的程序设计工作。
- ❑ 掌握了 C 语言，再学习其他程序设计语言往往比较容易。

本书即将展现的是一个简单的 C 语言，让 C 语言入门新手能在较短的时间内快速掌握 C 程序设计的基本思维和基础知识。本书和其他 C 语言图书的讲解方式有所不同。本书讲解时从实际出发，对 C 语言中的很多概念用生活中的例子进行类比。语言上力求幽默直白、轻松活泼，避免云山雾罩、晦涩难懂。讲解形式上图文并茂，由浅入深，抽丝剥茧。通过阅读本书，读者会少走很多弯路，会感觉到 C 语言其实没有想象的那么难。

本书特色

1. 语言幽默直白，轻松活泼，通俗易懂

本书避免使用那些艰涩难懂的术语云山雾罩地分析问题，代之以轻松活泼、幽默直白的语言讲解书中的每一个知识点。笔者力争让 C 语言的学习变得像看故事会一样通俗易懂。

2. 实例丰富，实用性强，并注重原理的讲解

本书结合大量生活中的实例，对 C 语言中的基本概念和知识做了深入浅出的讲解，并给出了大量生动形象的图示对程序的原理进行讲解，以加深读者的理解。

3. 图示丰富，容易理解

本书针对 C 语言中一些较难理解的概念，提供了大量的图示进行介绍，让读者以更加形象、直观的方式来理解所讲解的知识，从而达到更好的学习效果。

4. 举一反三

授之以鱼不如授之以渔。本书讲解时注重由此及彼，启发读者的思维，让读者通过已经掌握的知识进一步延伸到更深、更宽、更广的领域，从而达到举一反三的作用。

5. 习题丰富

本书每章后面都提供了有针对性的典型练习题，并给出了必要分析和实现的关键代码，以便于读者巩固和提高。

本书内容及体系结构

第1篇 一切从基础开始（第1~2章）

本篇简单讲述了计算机语言的相关基础知识，帮助大家计算机语言有个感性的认识，进而讲述了 C 语言的相关背景，并重点讲述了 C 语言的开发工具和学习经验。

第2篇 简单程序的构建（第3~4章）

本篇主要讲述了 C 语言的相关基本概念，作为学习 C 语言最基本的储备。所谓万丈高楼平地起，本篇知识掌握得好坏会直接影响后面章节的学习。

第3篇 复杂数据的表示（第5~10章）

本篇讲述了 C 语言中一些比较复杂的知识点，也可以称之为高级知识。这些看着稍微复杂的知识也正是 C 语言的核心，能否使用 C 语言进行游刃有余的开发，就看对这篇内容的掌握和理解程度了。

第4篇 复杂功能的实现（第11~13章）

本篇主要讲述了在使用 C 语言进行实际开发时需要使用的知识点，掌握了本篇内容，就可以自己开始进行实际的编程开发了。

第5篇 C语言的高级内容（第14~15章）

本篇是对前面所有知识点的总结，主要讲述了如何使用 C 语言进行文件操作。

本书读者对象

- ❑ 没有基础的 C 语言入门新手;
- ❑ 刚入职的初、中级程序员;
- ❑ 大、中专院校的学生;
- ❑ 相关培训学校的学员;
- ❑ C 语言开发爱好者。

本书作者

本书由马磊主笔编写。其他参与编写的人员有陈世琼、陈欣、陈智敏、董加强、范礼、郭秋滢、郝红英、蒋春蕾、黎华、刘建准、刘霄、刘亚军、刘仲义、柳刚、罗永峰、马奎林、马味、欧阳昉、蒲军。

阅读本书的过程中,如果有疑问或发现本书有任何纰漏,可与笔者联系。联系邮箱:
xd_malei@163.com。

编著者

目 录

第 1 篇 一切从基础开始

第 1 章 概述 (🎥 教学视频: 21 分钟)	2
1.1 C 语言简介	2
1.1.1 C 语言的位置	2
1.1.2 C 语言的优缺点	3
1.1.3 C 语言适合什么开发	4
1.2 C 语言的开发环境	4
1.2.1 编辑器、编译器和链接器	4
1.2.2 集成开发环境	6
1.3 Visual Studio 使用简介	8
1.3.1 Visual Studio 版本	8
1.3.2 Visual Studio 的安装	9
1.3.3 新建项目	9
1.3.4 编写代码	13
1.3.5 编译链接	15
1.3.6 运行可执行程序	16
1.4 如何学好 C 语言	17
1.5 小结	17
1.6 习题	17
第 2 章 开始 C 语言之旅 (🎥 教学视频: 22 分钟)	19
2.1 为什么要写代码	19
2.1.1 为什么要写程序	19
2.1.2 从本书开始学编程	20
2.1.3 从一个现实的例子开始	20
2.2 编程的核心——数据	21
2.2.1 数据从哪里来	21
2.2.2 数据的表示	23
2.2.3 数据类型面面观——精度和范围	23
2.2.4 C 语言基本数据类型	23
2.2.5 数据的变与不变——变量、常量	25
2.3 使用变量和常量	26
2.3.1 变量的使用	26
2.3.2 命名的方式	27

2.3.3 关键字	28
2.3.4 常量的使用	29
2.4 小结	30
2.5 习题	30

第2篇 简单程序的构建

第3章 简单数学运算 (教学视频: 44 分钟)	34
3.1 什么是赋值	34
3.1.1 赋值的作用——把数据存起来	34
3.1.2 赋值运算的形式	35
3.1.3 赋值表达式	35
3.1.4 机动灵活的赋值——scanf()	35
3.1.5 看看我们的劳动成果——printf()	36
3.1.6 赋值的重要性	37
3.2 开始赋值——整型赋值	38
3.2.1 整数在计算机中的表示——二进制	39
3.2.2 更先进的表示方法——八进制和十六进制	42
3.2.3 进制之间的转换——以二进制为桥梁	44
3.2.4 给整型赋值	45
3.3 浮点型赋值	48
3.3.1 小数在计算机中的表示	48
3.3.2 给浮点型赋值	52
3.4 字符型赋值	54
3.4.1 字符在计算机中的表示——ASCII	54
3.4.2 给字符赋值	55
3.5 类型转换	58
3.5.1 什么是类型转换	58
3.5.2 类型转换的利弊	59
3.5.3 隐式类型转换和显式类型转换	59
3.5.4 赋值中的类型转换	61
3.6 基本数学运算	64
3.6.1 数学运算和数学表达式	64
3.6.2 商与余数	67
3.6.3 位运算	68
3.6.4 优先级的奥秘	73
3.6.5 数学运算中的类型转换	77
3.7 复合赋值运算	79
3.7.1 复合赋值运算	79
3.7.2 自增自减运算——特殊的复合赋值	81
3.7.3 自增自减运算的使用	82
3.8 小结	83
3.9 习题	84

第4章 程序结构 (教学视频: 45 分钟)	88
4.1 语句和语句块	88
4.1.1 简单语句	88
4.1.2 语句块	89
4.2 变量的作用域	90
4.2.1 局部变量的声明定义位置规则	90
4.2.2 局部变量的作用域规则	91
4.2.3 嵌套语句块的同名变量作用域规则	91
4.3 最常见的语句执行结构——顺序结构	93
4.4 判断结构	94
4.4.1 判断的基础——逻辑真假	94
4.4.2 基础的判断——关系运算	95
4.4.3 复杂的判断——逻辑运算	97
4.5 if 判断结构	101
4.5.1 基本 if 结构	101
4.5.2 if-else 结构	102
4.5.3 另类的条件判断——?运算符的使用	104
4.5.4 if-else if-else 结构	106
4.5.5 嵌套的 if 结构	109
4.6 switch 判断结构	112
4.6.1 switch 基本结构	112
4.6.2 果断结束——break 的使用	114
4.7 循环结构	116
4.7.1 while 循环结构	116
4.7.2 for 循环结构	119
4.7.3 goto 语句	123
4.7.4 循环嵌套	124
4.7.5 break 和 continue	126
4.8 真正的程序——三种结构的揉和	129
4.9 小结	131
4.10 习题	131

第3篇 复杂数据的表示

第5章 数组 (教学视频: 39 分钟)	144
5.1 数组简介	144
5.1.1 数组的用途	144
5.1.2 数组变量的定义	145
5.2 数组变量初始化和赋值	146
5.2.1 数组的初始化	146
5.2.2 数组的下标	147
5.2.3 给数组赋值	147
5.2.4 数组元素的引用	148

5.3 二维数组	149
5.3.1 数组的维	150
5.3.2 二维数组表示和含义	150
5.3.3 二维数组的初始化	151
5.3.4 二维数组的赋值	152
5.3.5 二维数组的引用	154
5.4 多维数组	155
5.5 小结	157
5.6 习题	158
第 6 章 字符数组——字符串 (教学视频: 31 分钟)	161
6.1 字符数组	161
6.1.1 字符数组的表示	161
6.1.2 字符数组的初始化	161
6.1.3 字符数组的赋值和引用	162
6.2 字符串	164
6.2.1 字符串的 C 语言表示	164
6.2.2 使用字符串为字符数组初始化	164
6.2.3 字符串的保存形式	165
6.3 字符串的输入/输出——scanf 和 printf 字符串	166
6.3.1 输入/输出字符串的 C 语言表示	166
6.3.2 scanf() 函数对字符串的特殊处理	168
6.4 小结	169
6.5 习题	170
第 7 章 指针 (教学视频: 65 分钟)	172
7.1 地址的概念	172
7.1.1 地址的含义	172
7.1.2 为什么要用地址	173
7.1.3 地址的表示与取址运算	174
7.2 指针和指针变量	175
7.2.1 指针的含义和用途	175
7.2.2 指针类型	176
7.2.3 指针变量的定义和使用	177
7.2.4 void 指针	177
7.3 指针运算	179
7.3.1 取指针元素	179
7.3.2 指针的自增自减	181
7.3.3 指针的类型转换	183
7.4 数组和指针	186
7.4.1 数组名也是指针	186
7.4.2 数组名是指针常量	187
7.4.3 使用数组名访问数组元素	187
7.4.4 三种访问数组元素的方法	189
7.4.5 数组指针和指针数组	190

7.5 多重指针和多维数组	192
7.5.1 多重指针	192
7.5.2 取多重指针元素运算	194
7.5.3 多维数组名和各维元素	195
7.5.4 使用指针访问多维数组	197
7.6 字符串和指针	199
7.6.1 字符指针	199
7.6.2 字符指针和字符串	200
7.6.3 scanf()、printf()函数和字符指针	201
7.7 小结	203
7.8 习题	203
第8章 结构体 (教学视频: 35 分钟)	205
8.1 结构体的含义	205
8.2 结构体类型的表示	206
8.2.1 结构体类型的一般格式	206
8.2.2 结构体的成员变量	207
8.2.3 复杂的结构体	208
8.3 结构体变量	209
8.3.1 结构体变量的声明定义	209
8.3.2 结构体变量初始化	210
8.3.3 取结构体成员运算	211
8.4 结构体数组	213
8.5 结构体指针	215
8.5.1 一重结构体指针	215
8.5.2 使用结构体指针取结构体数据	216
8.5.3 结构体指针例子	217
8.6 回到问题	218
8.7 小结	218
8.8 习题	219
第9章 共同体类型 (教学视频: 36 分钟)	222
9.1 共同体的含义与表示	222
9.1.1 共同体的用途	222
9.1.2 共同体的表示	222
9.1.3 复杂的共同体	224
9.2 共同体变量	225
9.2.1 共同体变量	225
9.2.2 共同体成员变量的相互覆盖	225
9.2.3 使用共同体变量	226
9.3 共同体数组	228
9.4 共同体的指针	231
9.4.1 一重共同体指针类型	231
9.4.2 共同体指针变量	231
9.4.3 完整的例子	232

9.5 小结	233
9.6 习题	233
第 10 章 枚举类型 (教学视频: 35 分钟)	235
10.1 枚举类型的含义与表示	235
10.1.1 枚举类型的含义	235
10.1.2 枚举类型的表示	236
10.2 枚举常量和枚举变量	236
10.2.1 枚举常量	237
10.2.2 枚举变量的定义	239
10.2.3 枚举变量的使用	240
10.3 枚举数组和枚举指针	241
10.3.1 枚举数组	241
10.3.2 枚举指针	241
10.3.3 用枚举指针来访问枚举数组	242
10.4 typedef 类型定义符	243
10.5 小结	245
10.6 习题	245

第 4 篇 复杂功能的实现

第 11 章 函数 (教学视频: 50 分钟)	248
11.1 函数的意义	248
11.2 函数的形式	249
11.2.1 函数的一般形式	249
11.2.2 函数的参数列表	250
11.2.3 函数的返回值类型	251
11.3 函数的声明和定义形式	251
11.3.1 函数的声明	251
11.3.2 函数的定义形式	252
11.3.3 函数的形参	253
11.3.4 return 返回值语句	254
11.4 自己动手写一个函数——加法函数	255
11.4.1 确定加法函数的样子	255
11.4.2 实现加法函数体	256
11.4.3 完整的加法函数定义	256
11.5 函数调用	257
11.5.1 函数的调用作用	257
11.5.2 函数的调用表达式	258
11.5.3 函数的实参	258
11.5.4 简单函数的调用	259
11.6 复杂参数	260
11.6.1 数组参数	260
11.6.2 指针参数	262

11.6.3 结构体、共同体和枚举参数	265
11.7 小结	268
11.8 习题	268
第 12 章 特殊的函数——main()函数 (教学视频: 32 分钟)	272
12.1 main()函数的作用	272
12.2 main()函数的声明定义	273
12.2.1 main()函数的声明形式	273
12.2.2 main()函数的参数	273
12.2.3 main()函数的返回值	275
12.3 小结	276
12.4 习题	276
第 13 章 局部变量和全局变量 (教学视频: 39 分钟)	277
13.1 变量的作用域和生命周期	277
13.2 函数内的局部变量	278
13.2.1 局部变量的作用域	278
13.2.2 局部变量的生命周期	278
13.2.3 局部变量的覆盖作用	280
13.3 函数外的全局变量	282
13.3.1 全局变量的作用域	282
13.3.2 全局变量的生命周期	283
13.3.3 局部变量对全局变量的覆盖作用	284
13.4 变量修饰符	286
13.4.1 使用修饰符改变变量的作用域和生命周期	286
13.4.2 C 语言中常用变量修饰符的作用	286
13.5 小结	289
13.6 习题	289

第 5 篇 C 语言的高级内容

第 14 章 预处理命令、文件包含 (教学视频: 48 分钟)	292
14.1 预处理命令的作用	292
14.1.1 程序预处理	292
14.1.2 预处理命令	293
14.1.3 C 语言的几类预处理命令	294
14.2 C 语言中的宏定义	294
14.2.1 C 语言的宏定义形式	294
14.2.2 不带参的宏定义	295
14.2.3 带参的宏定义	296
14.3 预编译控制	298
14.3.1 C 语言预编译控制	298
14.3.2 三种预编译控制组合形式	299
14.3.3 一个简单的例子	300

14.4 文件包含	301
14.4.1 头文件和源文件的文件名	301
14.4.2 头文件和源文件的内容	302
14.5 include 包含头文件	303
14.5.1 自定义头文件和系统头文件	303
14.5.2 文件包含的两种形式	303
14.5.3 完整的 circle 例子	304
14.5.4 C 语言中的标准头文件	306
14.6 小结	308
14.7 习题	309
第 15 章 文件操作 (教学视频: 47 分钟)	313
15.1 文件	313
15.1.1 重新认识文件	313
15.1.2 计算机眼里的文件	314
15.1.3 开发人员能对文件干些什么	315
15.2 文件的打开与关闭	315
15.2.1 文件指针	315
15.2.2 文件打开函数	316
15.2.3 文件关闭函数	318
15.3 文件读写	318
15.3.1 读写一个字符	318
15.3.2 读写一个字符串	319
15.3.3 读写一个数据块	320
15.4 文件的其他操作	322
15.4.1 随机读写文件	322
15.4.2 出错检验	323
15.5 小结	325
15.6 习题	325
附录 A ASCII 码表	327

第 1 篇 一切从基础开始

▶▶ 第 1 章 概述

▶▶ 第 2 章 开始 C 语言之旅



第 1 章 概 述

作为本书的开始，我们不涉及 C 语言的语法和使用细节，先来看看这门语言的相关背景，以及其在计算机程序中举足轻重的地位。然后，为给后面的学习作准备，来看看如何建立 C 语言的开发环境。最后，抛砖引玉，说说笔者自己的 C 语言学习经验。

1.1 C 语言简介

本节先来看看 C 语言的相关背景介绍。在众多的计算机语言之中，它到底处于一个什么样的位置？相对于其他计算机语言而言，它有什么优势和弱点？另外，C 语言适合做些什么样的软件开发？带着这些问题开始本节的学习。

1.1.1 C 语言的位置

图 1.1 是计算机语言发展过程的一个简单的示意图，只展示了一些主流的计算机语言的出现时间和顺序，不是很全，但是足以显示 C 语言与其他语言的关系了。

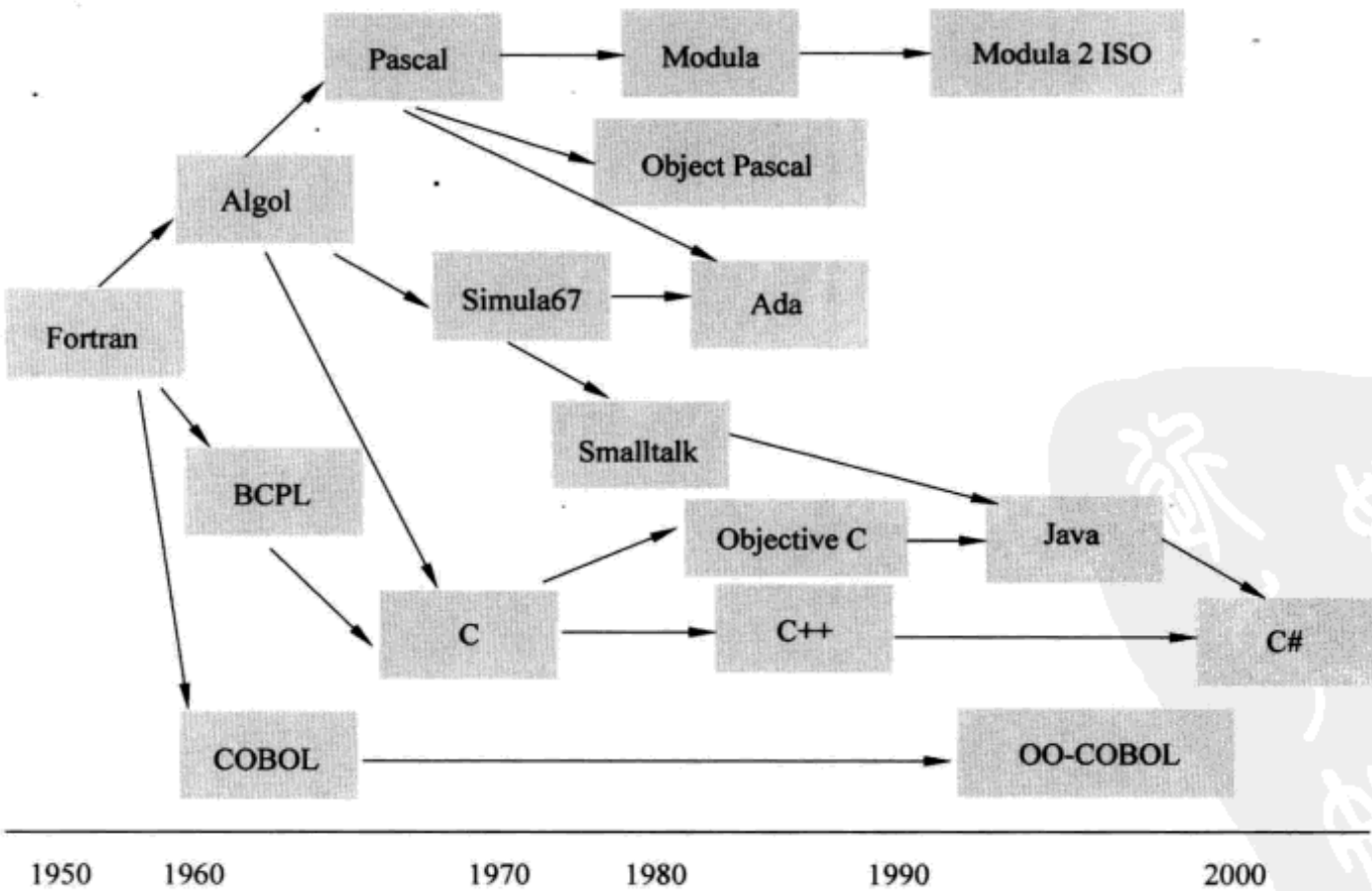


图 1.1 计算机语言发展史（简）

从图 1.1 中可以看出，所有的高级语言都起于 Fortran，之后就有不同的分支了。C 语言也是其中的一个分支，主要起源于 BCPL 语言（Basic Combined Programming Language），是对 BCPL 语言的一种简化。C 语言之所以被称为 C 语言，是因为 BCPL 语言被称为 B 语言，C 语言不能再取 BCPL 的首字母了，因此，取 BCPL 的第二个字母 C，C 语言因此得名。

C 语言诞生于 20 世纪 70 年代，之后不断完善、标准化。目前流行的 C 语言编译系统大多数是以 ANSI C（美国国家标准协会（ANSI）对 C 语言发布的标准）为基础进行开发的。但不同版本的 C 编译系统所实现的语言功能和语法规则略有差别。现在比较通用的是 1990 年，国际标准化组织 ISO（International Organization for Standards）接受的 1989 年的 ANSI C 标准，并以它作为 ISO C90 标准——ISO/IEC9899:1990。

1.1.2 C 语言的优缺点

任何一种计算机语言都有它的优势和不足，C 语言也是如此。不要以为 C 语言都那么老了，是不是没用了，也不要以为 C 语言是万能钥匙，什么问题都能解决好。

1. C 语言的优势

C 语言的优点细数起来，估计手指加脚趾都不够用，但是这些优点可以总结为一点，那就是“灵活”。C 语言的灵活主要体现在同一件事情可以通过好几种方式实现，并不会定死什么问题非得用什么方式来解决。C 语言之所以灵活，是因为它有下面这些属性，保证了它天生就是一种灵活的语言。

- ❑ 结构丰富多变：C 语言提供了三种基本的程序设计结构，通过这三种程序设计结构，就能够完成所有的计算机逻辑。而且每种结构中，又有略有差别的不同的形式，你可以选择自己喜欢的任意形式来完成需要的功能。
- ❑ 提供了多种基本运算：C 语言提供了 30 多种运算符号，分为 15 个运算优先级，不仅可以完成基本的数学运算，还可以完成类似于计算机底层操作的位运算。有的运算符号在不同情况下的含义和使用方式也是不同的，而且可以通过强制结合来改变运算符的优先级。
- ❑ 丰富的数据类型：数据类型决定了一种语言可以用来操作什么样的数据。C 语言的数据类型真可谓丰富，从基本的几种数据类型，到复杂数据类型，再到可以自定义的数据类型。从某种程度上来说，C 语言可以用来操作任何类型的数据。
- ❑ 程序设计自由，语法限制不大：C 语言代码在书写的时候，从书写格式到代码组织限制都不是很大，所以写代码的风格可谓百花齐放，什么样的都有。只要符合最基本的要求，无论你怎么写，都是对的。只不过代码风格不好，不利于阅读和理解。

对于上面提到的 C 语言的各种优点，在没有接触 C 语言之前，可能大家还体会不到。不要紧，相信大家学完这本书之后，回过头来看看这一部分内容，一定会有所感悟的。

2. C 语言的不足

一个事物，往往最强的地方也就是它最弱的地方。C 语言的弱点也正是由于它的“灵活”造成的。因为太灵活了，怎么样都行，对计算机的控制太过自由，稍不留意就会出现错误！所以，能灵活运用 C 语言，也是一种能力啊。

1.1.3 C 语言适合什么开发

由于 C 语言本身的灵活性，导致它可以很好地处理复杂和具有差异的环境。因此，它至少适合进行下面三类软件的开发。

1. 多平台通用软件

所谓多平台通用软件，就是指这个软件可以在很多系统上使用。例如，如果想让你写的同一个程序很好地运行在 DOS、Windows 98、Windows XP、Windows CE、Linux、UNIX 等多个操作系统上，C 语言确实是个不错的选择。知道 Apache 吗？它是一个开源 Web 服务器工程，其中的 HTTP 服务器可以运行于很多平台，Windows、Linux 和 UNIX 都是可以的，这个服务器就是用 C 语言开发的。

2. 操作系统

由于 C 语言可以很自然地与汇编语言结合，又比汇编语言好用，能够很灵活地控制计算机硬件，因此很适合开发操作系统。Windows 很老的版本都是用 C 语言写的，之后改用 C++了，不过 C++是兼容 C 语言的。Linux 和 UNIX 系列的操作系统内核几乎都是用 C 语言写的，而且很多运行在板子上的嵌入式操作系统基本都是用 C 语言结合汇编写的。如果你真想做个操作系统，C 语言绝对可以胜任。

3. 复杂运算软件

之所以说 C 语言适合进行复杂计算软件的开发，是因为：（1）复杂计算软件本身很单纯，只需要计算机进行计算就可以了；（2）C 语言本身有丰富的运算功能，完全可以实现复杂计算功能；（3）因为 C 语言是一种接近底层语言的高级语言，所以它写出来的程序在计算机上的运行效率很高。所以，要想做一个需要复杂而高效计算功能的软件，选择 C 语言绝对不会令你失望的。如果你经常做科学计算，用过 MATLAB 软件，它其中一部分也是用 C 语言写的。

1.2 C 语言的开发环境

“工欲谋其事，必先利其器”。要想学好 C 语言，选择一个好的 C 语言开发环境是很有必要的，而且是首要任务。在本节中，来看看 C 语言的开发过程，教大家学习 Windows 平台上主流的 C 语言开发工具 Visual Studio 的基本使用方法。

1.2.1 编辑器、编译器和链接器

编辑器、编译器和链接器是使用 C 语言进行开发所需要的三个最基本的工具，而且也是最小的工具集，缺一不可。

从图 1.2 所示的一个可以运行的程序的生成过程，就可以看出编辑器、编译器和链接

器在这条可执行程序“生成链”上所处的位置了。从键盘一句一句敲出代码到一个可以在操作系统上运行的程序，必须经过这三个工具的处理才可以。接下来逐个看一看这三个程序的作用。

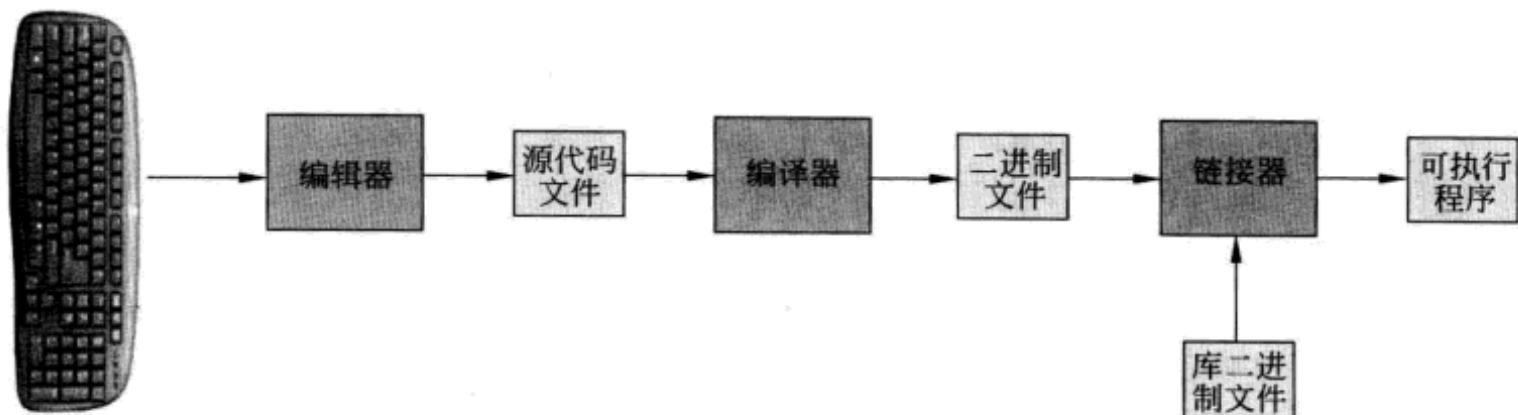


图 1.2 可执行程序的生成过程

1. 编辑器

编辑器的作用就是将我们在键盘上的敲击转换成写有代码的文件，这个文件被称为程序源文件，或者直接叫做源文件。

这个编辑器在形式上很像是一个记事本程序，可以在上面添加、删除、插入字母或者文字，并且可以将这些字母和文字保存到一个文件中。其实，记事本是可以作为 C 语言编辑器的，我们在记事本上编辑的字母和文字就是将要学习的 C 语言代码。这些代码保存到一个文件中，供下次编辑或者直接交给编译器去处理，这个文件就是源文件。

不过，记事本可以算是最简单和最不专业的 C 语言编辑器了！一般而言，专业的 C 语言编辑器都有其他方便 C 语言代码编辑的功能。例如，最基本的语法高亮功能，所谓语法高亮，就是程序中不同用途的字母或者文字会以不同的颜色显示出来，让你一看就知道正在写的代码是做什么用的。越高级、越专业的 C 语言编辑器，越能最大程度地方便你写 C 语言代码。

其实，只要能将键盘的输入转换成源文件的就可以称为编辑器。一般的编辑器也是可以编写 C 语言代码的，只不过，比起专业的 C 语言编辑器会显得难用和不舒服一点。

2. 编译器

编译器的作用是将保存着我们所写代码的源文件，转换成一种称为目标文件的二进制文件。源文件是我们能看懂的，是给我们用的。而二进制文件是计算机能看懂的文件，是给计算机用的。计算机根据二进制文件中的内容决定该做什么事情，不该做什么事情。

在这个转换过程中，编译器先对源文件中的内容进行扫描，根据 C 语言的语法要求，逐个检查源文件中出现的每一个字母或者文字。

如果这些文字符合 C 语言的语法要求，那么它就能根据这些字母和文字的含义将其转换成计算机可以识别的二进制代码，并将其按照一定的格式保存在二进制文件中。如果某些地方的字母或者文字不符合 C 语言的语法要求，那么编译器将报告所有不符合的地方，不再生成二进制文件，只有改正所有不符合语法要求的地方，让编译器重新对改正的源文件进行转化，才可以生成二进制文件。

编译器对源文件的转换过程在计算机中有个专业的名字，叫“编译”。编译器也因此得名，表示编译的工具。

3. 链接器

一般所写的程序最终是要运行在某个操作系统上的。因此，即使是一个很简单的程序也需要操作系统来处理很多事情，才能使程序正常运行。操作系统往往会提供一些被称为开发库的二进制文件，编译器产生的目标二进制文件只有和这些库二进制文件结合才能生成一个可执行程序，才能使我们写的程序正常地运行于某个操作系统之上。

另外，有的时候我们可能会开发一些专业的或者功能很复杂的软件，这类软件要从头做，往往很麻烦。这个时候，就得看看其他公司或者业界有没有提供此类功能实现，可以买过来使用。往往买过来的也是一堆库二进制文件，只有把这些库二进制文件和编译器产生的目标二进制文件结合起来才能产生需要的可执行程序。

链接器所做的工作就是将所有的二进制文件链接起来融合成一个可执行程序，不管这些二进制文件是目标二进制文件还是库二进制文件。链接器将二进制文件融合的这一过程，在计算机中也有一个专业的名字——“链接”，链接器也因此得名，表示链接的工具。

1.2.2 集成开发环境

按照前面介绍的可执行程序的生成过程，要想用 C 语言写一个可以真正运行于某个操作系统之上的程序，至少需要编辑器、编译器和链接器三个开发工具。集成开发工具也是一个工具，不过它的功能更强大，因为它集成（包含）了编辑器、编译器、链接器和其他用于开发的工具。

1. 直观印象

先来直观地感受一下，一个真正的集成开发环境是什么样的。图 1.3 所示就是一个 Windows 操作系统上的集成开发环境，Windows 上的其他集成开发环境，以及其他操作系统上的集成开发环境基本都长这个样子。



图 1.3 集成开发环境

集成开发环境也是一个程序，它是一个用于开发的程序。图 1.3 显示的集成开发环境

就是一个标准的 Windows 应用程序，有点像我们经常使用的 Office Word。

一个集成开发环境基本都是以窗口的形式展现在我们面前的，如图 1.3 所示。在这个窗口中一般会包含三个部分：菜单栏、工具栏和子窗口。图 1.3 中最上边的“文件”、“编辑”等一行就是菜单栏，菜单栏之下的一行图标组成了工具栏，工具栏之下被分成一块一块的就是子窗口了。

基本上所有的集成开发环境都是由这三部分组成的，只是组成的方式不同而已。例如，菜单栏中的内容、工具栏中的图标的位置和数目、子窗口的大小和数目等会有所不同。大家先过一下眼就好，后面会讲到如何使用这个集成开发环境。

2. 集成开发环境中的编辑器

集成开发环境是用来更好地开发程序的，它一定会包含生成一个可执行程序所必需的三个基本工具。这里先来看看第一个工具——编辑器在集成开发环境中的什么地方！

集成开发环境中的子窗口，往往有一个或者多个会用来完成编辑器的编辑、显示功能。菜单栏和工具栏中的某些菜单和按钮会用来完成编辑器的文件新建、保存等功能。

图 1.4 所示就是图 1.3 所示的集成开发环境中编辑、显示代码的编辑器。这个编辑器算得上一个高级的或者说专业的 C 语言编辑器了。因为至少从图 1.4 中可以看出，所写的代码中有不同颜色的单词，表明这个编辑器可以用颜色区分我们所写的代码中的不同部分，方便代码的阅读和编辑。其实它还有很多更强大的功能呢，这里就不一一赘述了，留待读者自己发掘吧！

这个集成开发环境中哪些菜单和工具是用在编辑器中的，在后边讲这个工具使用的时候，再详细讲解。这里大家知道集成开发环境中有一个子窗口是用于编辑代码的就可以了，它是编辑器的核心，也是我们以后会经常使用的。

3. 集成开发环境中的编译器和链接器

找到了集成开发环境中的编辑器以后，再来看看集成开发环境中的编译器和链接器吧！

集成开发环境中有一个子窗口是和编译链接有关系的，它可以将编译器和链接器在编译链接过程中的报告全部显示出来。这个窗口一般被称为输出窗口，因为编译链接的所有报告都是通过它输出的。图 1.3 所示集成开发环境的输出窗口如图 1.5 所示。



图 1.4 代码编辑、显示窗口

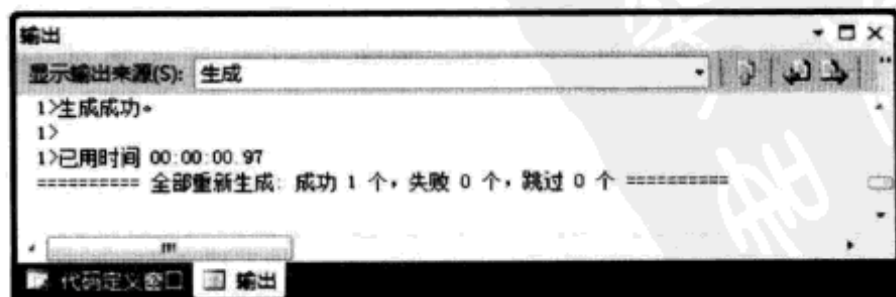


图 1.5 输出窗口

这个输出窗口只是一个“前台”而已，真正为我们干事的还是“编译器”和“链接器”这样的工具，或者说是程序。开发工作中，它们可谓是大功臣，但是它们只会默默无闻地干活，很少有人关注它们是谁，长什么样。

现在，就让大家拜会一下图 1.3 所示的集成开发环境中使用的编译器和链接器，如图 1.6 所示。其中，画圈的两个就是编译器和链接器，“cl.exe”是编译器，“cl”表示“compiler”中的两个字母，表示编译；“link.exe”就是链接器，“link”就是链接的意思。

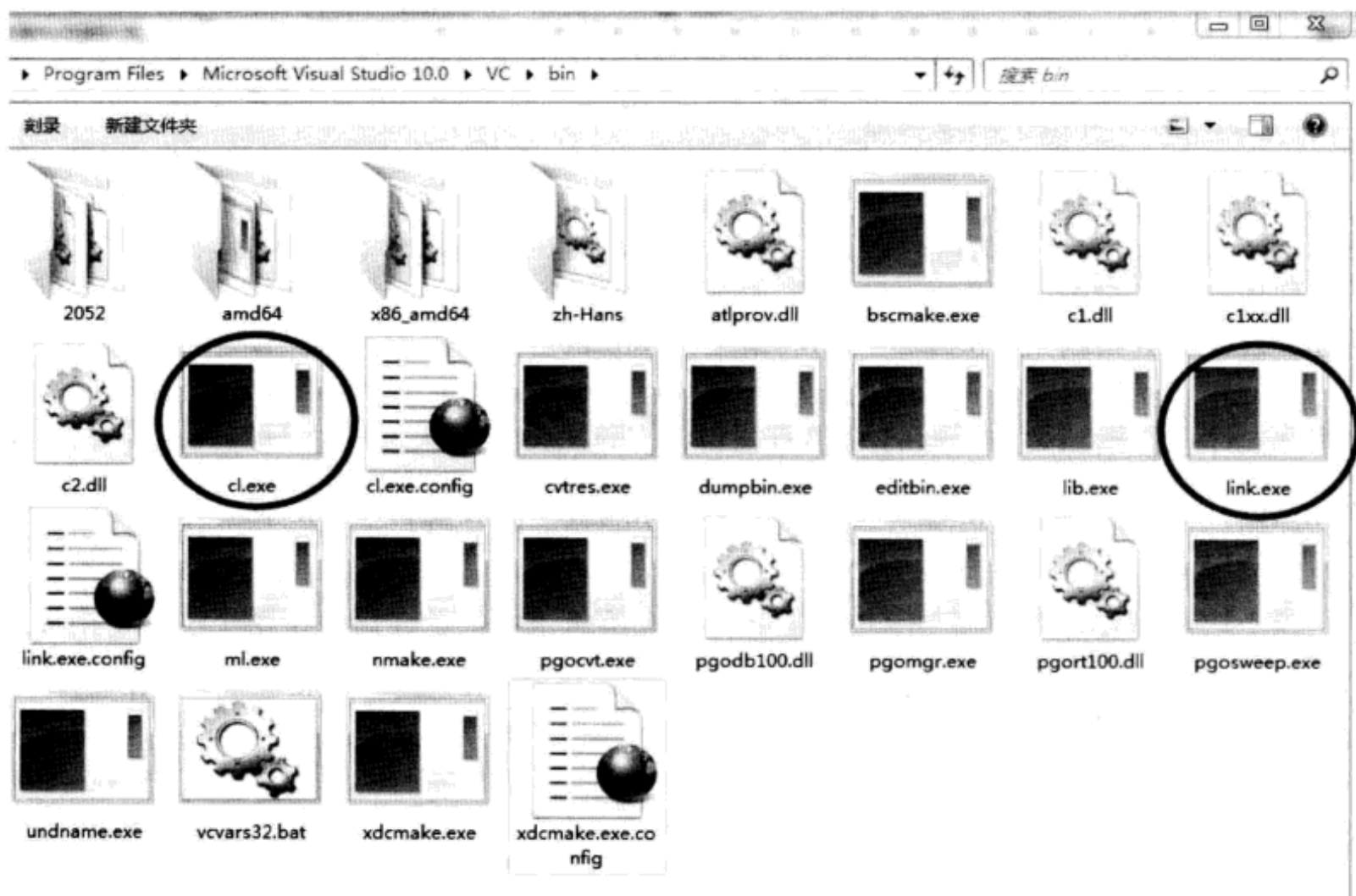


图 1.6 编辑器和链接器

对于如何在集成开发环境中使用这两个工具，在后边讲解如何使用集成开发环境的时候，会详细讲到，这里大家先认识一下这两个工具和输出窗口就可以了！

1.3 Visual Studio 使用简介

C 语言开发的集成开发环境众多，它们之间大同小异，可以根据自己的喜好和不同的平台进行选择。这里，选择 Windows 操作系统上较常使用的 Visual Studio 来介绍一个集成开发环境的简单使用方法。如果你喜欢其他集成开发环境，可以照猫画虎，和 Visual Studio 是类似的。

1.3.1 Visual Studio 版本

Visual Studio 的全称为 Microsoft Visual Studio，意思是“微软可视化工作室”，简称

为 VS。它包含 VB、VC、VF、Delphi、控件、数据库 ODBC 等开发工具，其中 VC 就是用来进行 C 和 C++ 开发的，表示 Visual C/C++。

VS 是由微软公司开发的，用于开发 Windows 程序的工具，要追溯它的版本信息，那就久远了。不过主要被使用的版本有 VS 6.0、VS 2003、VS 2005、VS 2008、VS 2010。每一个版本中都会包含一个 VC 开发组件或工具，用于 C 和 C++ 的开发，我们在学习 C 语言的时候，使用的就是 VC 开发工具。

对于众多版本的 VS，你可以任选一个进行安装使用，它们之间的差别不是很大。不过，还是建议尽量选择较新的版本，因为更新的版本对于标准的支持往往比较好，而且功能会更强大。

1.3.2 Visual Studio 的安装

选定一个 VS 版本之后，就可以进行安装了！每个 VS 版本的安装过程都是一样的，插入安装光盘，待光盘运行以后，就会弹出安装界面。如果没有弹出安装界面，你可以直接打开光盘文件夹，双击“autorun.exe”或者“setup.exe”文件就会弹出安装界面，如图 1.7 所示是 VS 2010 的安装界面。



图 1.7 VS 2010 安装界面

选择安装 Microsoft Visual Studio 2010，就可以进入安装过程了，剩下的安装过程就是一路“下一步”，中间需要输入序列号和选择安装路径，也就是选择安装的地方。

1.3.3 新建项目

要使用 Visual Studio 制作出一个能完成一定功能的软件，首先需要新建一个解决方案，一个解决方案中还可能包含一个或者多个项目。每个项目是用来组织脚本文件和代码文件并生成一个可执行程序或者库二进制文件的基本单位。所以，一个解决方案最终会产生一个或者多个可执行程序或者库二进制文件。使用这些可执行程序 and 库二进制文件就可以构成实际可以运行的程序了。

好了，这里大家只要知道，要使用 VS 制作一个实际可以运行的程序，必须有一个解决方案。这个解决方案包含一个或者多个项目，每个项目最终会产生一个可执行程序或者库二进制文件。

无论是要新建解决方案还是新建项目，都得先进入图 1.8 所示的界面。可能不同 VS 版本的这个界面会有所不同，但是打开方式都是一样的：找到“Microsoft Visual Studio 2010”启动程序（开始|所有程序|Microsoft Visual Studio 2010|Microsoft Visual Studio 2010），单击“Microsoft Visual Studio 2010”图标启动“VS”开发环境，依次单击图 1.3 所示左上角的“文件”|“新建”|“项目”命令，就会出现图 1.8 所示的 VS 2010 “新建项目”对话框。



图 1.8 VS 2010 “新建项目”对话框

使用 Visual Studio 新建解决方案和项目的方式有很多种，下面介绍两种常用的方式。

1. 先建解决方案再建项目

在图 1.8 所示对话框最左边一列“已安装的模板”中选择“其他项目类型”，再选择“Visual Studio 解决方案”，窗口就会变成图 1.9 所示了。最后填写解决方案“名称”和所放“位置”，就可以新建一个解决方案了。

(1) 新建完解决方案以后，就可以往解决方案中添加项目了，添加的方式如图 1.10 所示。在 Visual Studio 的“解决方案资源管理器”子窗口中，单击右键就会弹出图 1.10 所示的对话框。单击“添加”|“新建项目”命令，如果你有已经建好的项目，还可以选择“现有项目”。选择了“新建项目”以后，就会弹出如图 1.8 所示的“新建项目”对话框，这时就可以新建一个项目了。



图 1.9 新建解决方案

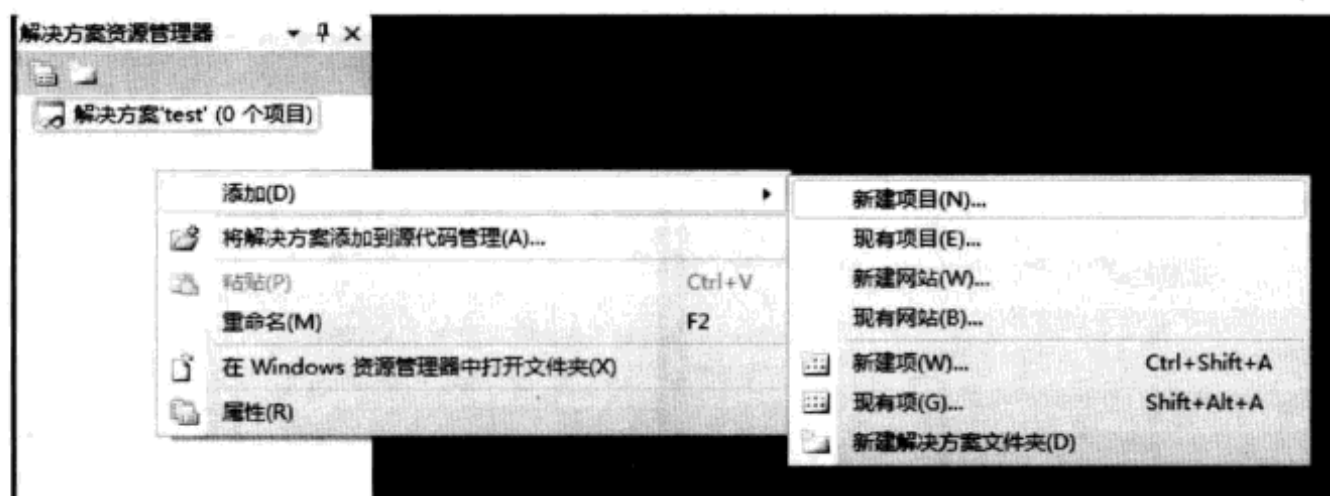


图 1.10 添加解决方案

(2) 弹出如图 1.8 所示的“新建项目”对话框之后，在最左边一列“已安装的模板”中选择“Visual C++”，再选择“Win32”，对话框就会变成图 1.11 所示了。选择“Win32 控制台项目”选项，最后填写项目“名称”和所放“位置”，就可以新建一个名为“test”的 Win32 控制台项目了。

(3) 之后，会弹出一系列的 Win32 应用程序向导，如图 1.12 所示。它会带领我们填完一个新建的项目的所有选项，基本都是直接打钩，然后单击“下一步”按钮，最后单击“完成”按钮。

(4) 完成项目新建之后，Visual Studio 的“解决方案资源管理器”子窗口会变成如图 1.13 所示的样子，表示我们已经完成了解决方案和项目的新建工作。



图 1.11 “新建项目”对话框

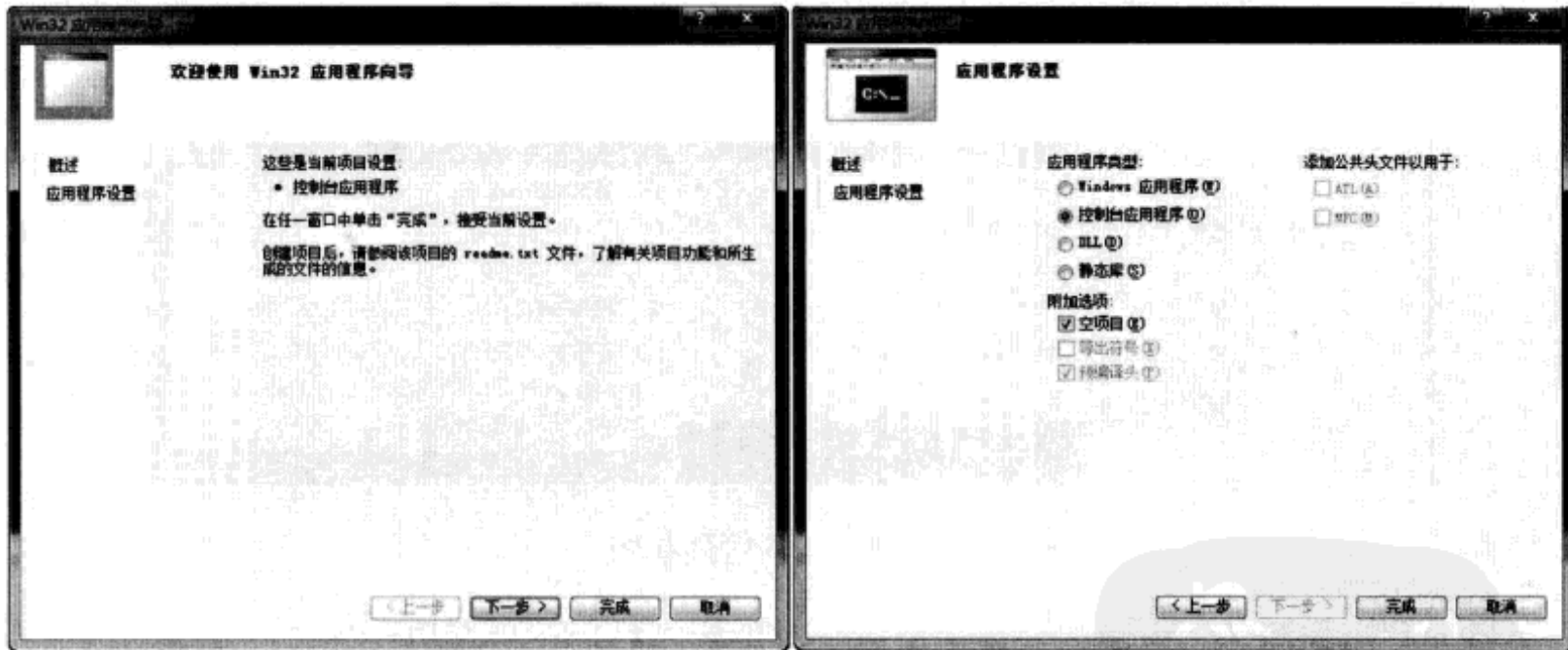


图 1.12 项目新建向导

2. 解决方案和项目一起建

其实，可以一次性完成解决方案和项目新建的工作。打开 VS 开发环境，单击“文件”|“新建”|“项目”命令，弹出图 1.8 所示的“新建项目”对话框时，直接新建项目就可以了。不过，在图 1.14 所示的对话框最后填写名称和路径的时候得再加上一个解决方案名称，VS 就会自动同时新

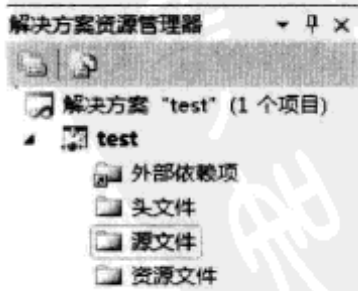


图 1.13 完成解决方案和项目新建

建解决方案和项目的名称填写,如图 1.14 所示。对于初学者,建议使用这种方式进行新建。剩下的新建项目过程和图 1.11、图 1.12 所示是一样的。

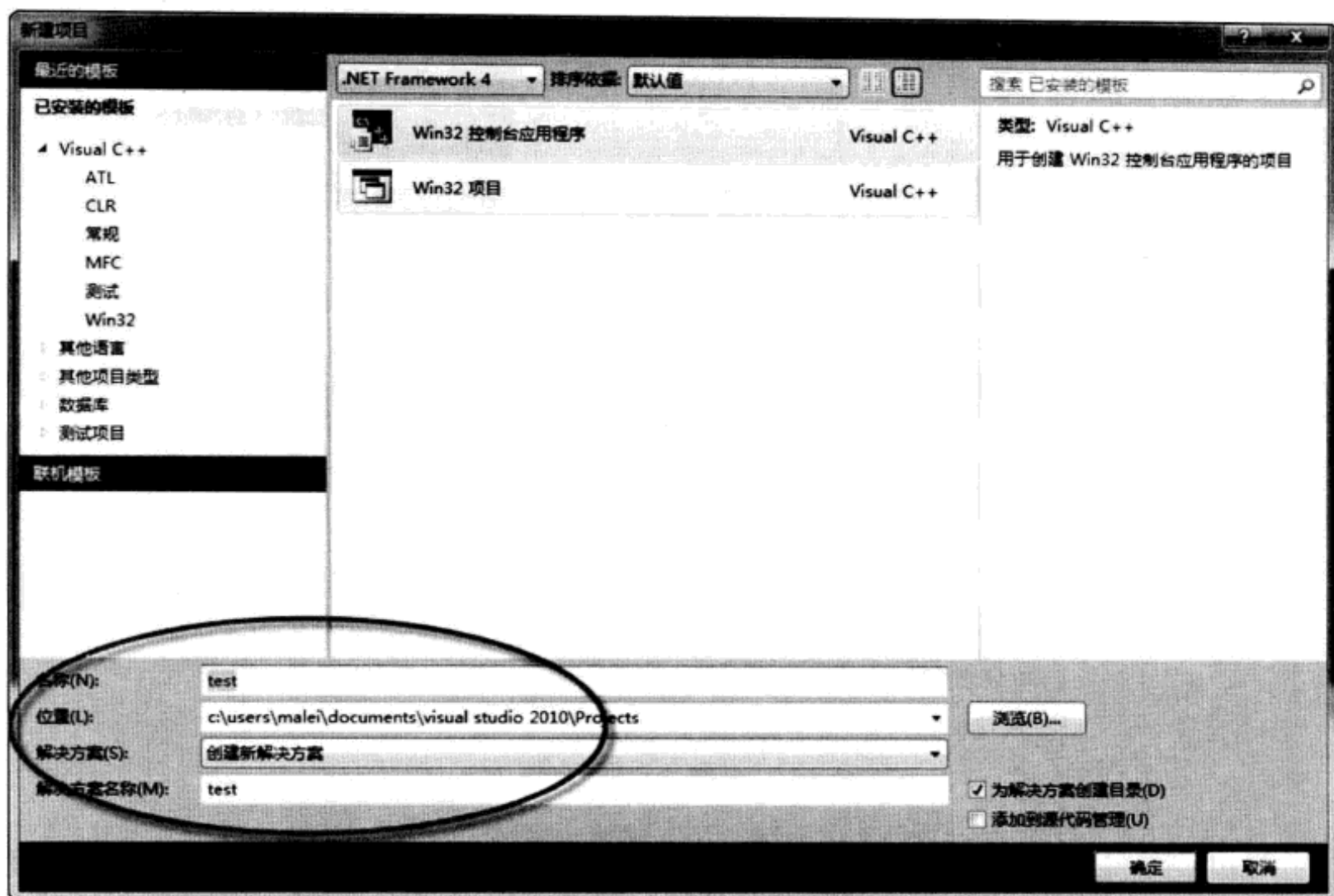


图 1.14 解决方案和项目一起建

这里说明一下, Win32 控制台应用程序运行出来的结果是一个如图 1.15 所示的窗口。它可以接受键盘输入的字母、数字和符号,也可以输出程序运行产生的字母、数字和符号。本书所有的代码都将在“Win32 控制台应用程序”项目中运行。另外,在图 1.12 所示对话框的“附加选项”中选择空项目,不需要 VS 给我们创建任何代码文件。

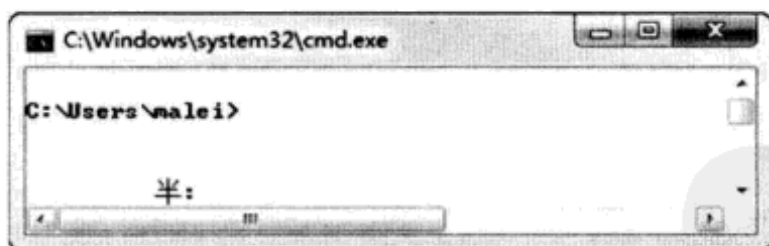


图 1.15 控制台程序窗口

1.3.4 编写代码

新建好项目以后,就可以编写代码了。在编写代码之前,得先新建一个用来保存代码的源文件,并将其添加到相应的项目中。一般最简单直接的方式是在对应的项目中单击右键,在弹出的快捷菜单中选择“添加”|“新建项”命令,如图 1.16 所示。

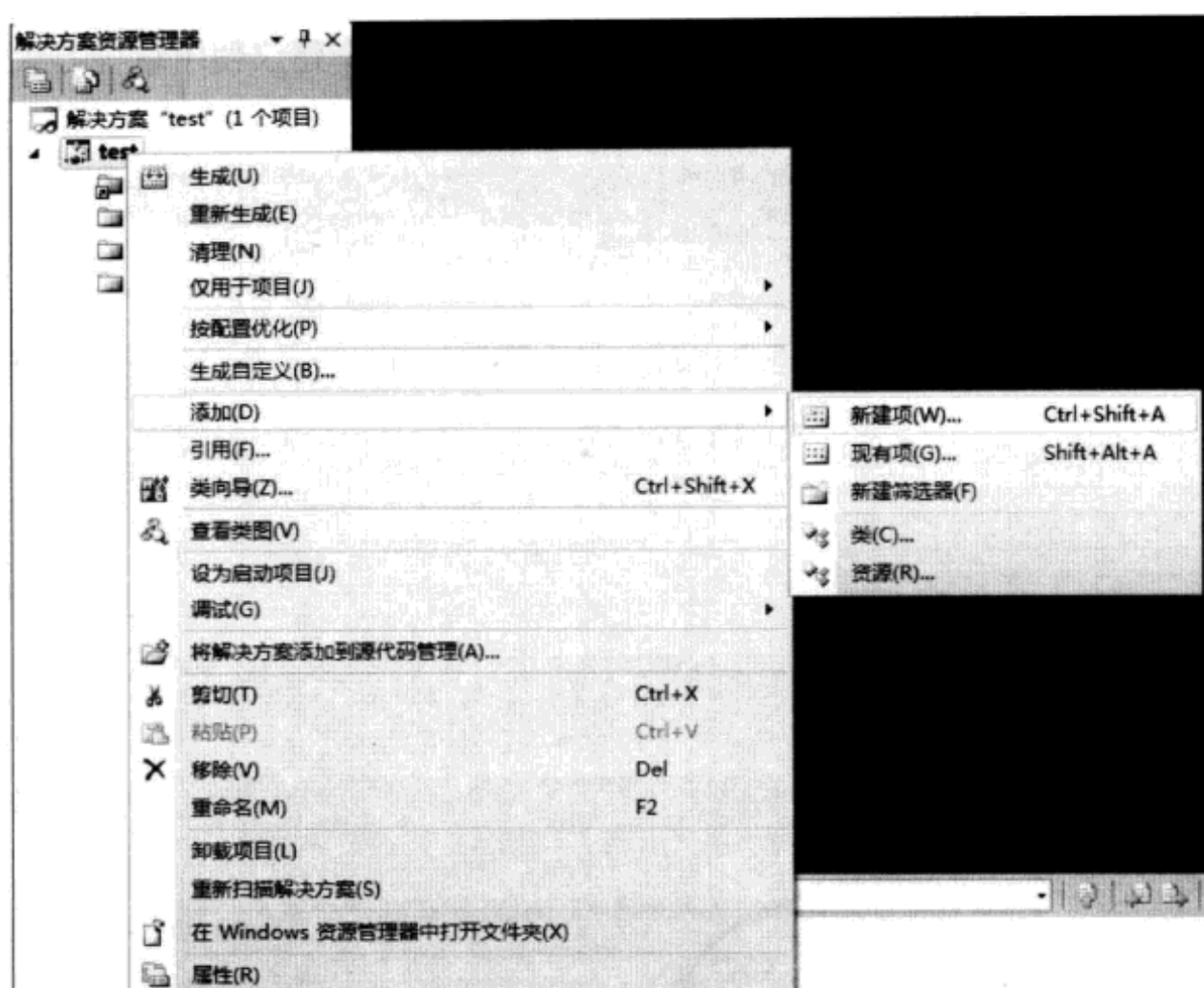


图 1.16 往项目中添加源文件

单击“添加”|“新建项”命令以后，会弹出如图 1.17 所示的“添加新项”对话框。选择“Visual C++”下的“代码”，然后选择“C++文件”，最后在“名称”栏中填写文件名“test.c”。C 语言源文件必须以“.c”作为后缀。最后选择“test.c”源文件放置位置。这里之所以要选择“C++文件”选项，是因为 VS 对 C 和 C++文件的处理差异不是很大，没必要分别列出，只要在填写文件“名称”时，以“.c”作为文件名后缀就可以了。



图 1.17 添加新项

选择并填写完添加源文件的所有选项和空格以后，单击“添加”按钮，新建的源文件

就被添加到对应的项目了。VS 2010 开发环境的“解决方案资源管理器”子窗口和“代码编辑”子窗口就会变成图 1.18 所示的样子了。在图 1.18 中，我们已经在源文件“test.c”中编辑好了一段简单的 C 语言代码。



图 1.18 添加完源文件并且编辑代码

1.3.5 编译链接

代码编辑好之后，就可以把代码源文件交给编译器进行编译了，编译完之后再交给链接器生成一个可以执行的程序。在 VS 的菜单栏中有一个菜单“生成”选项，就是用来让编译器和链接器完成编译链接工作的，如图 1.19 所示。

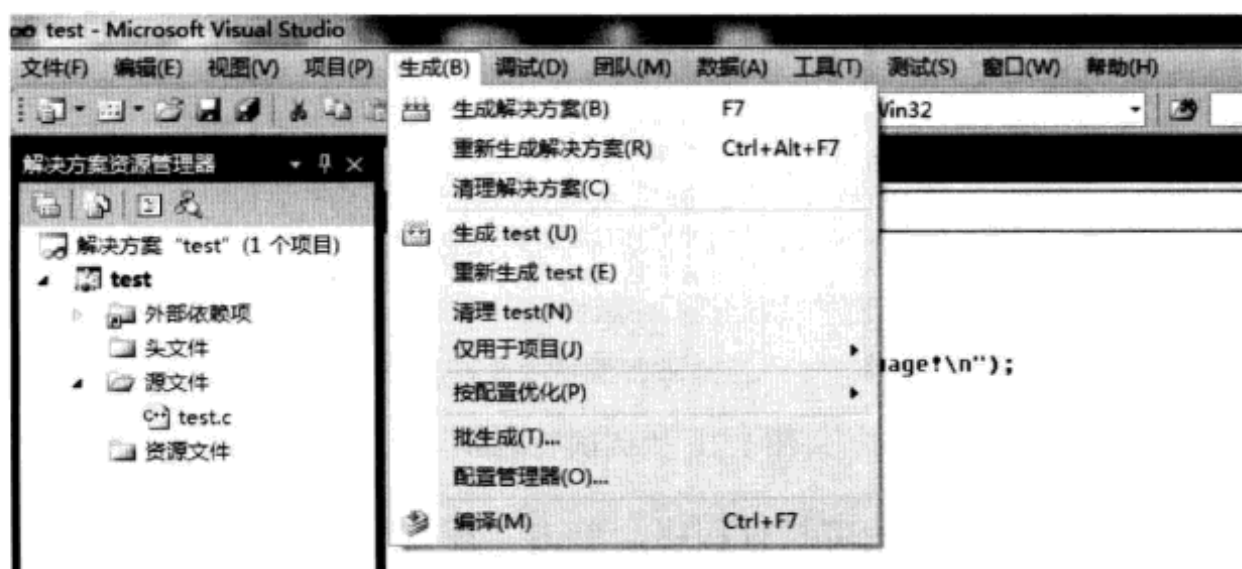


图 1.19 编译链接

我们较常使用的有“生成解决方案”、“生成 XX”和“编译”，XX 表示项目名称，此处为“test”。“生成解决方案”就是编译链接之后，生成该解决方案下的所有项目，其所对应的可执行文件或者库二进制文件；“生成 test”就是只对 test 项目进行编译链接，生成可执行程序；“编译”就是只对代码进行编译而不链接。

在这里，我们的解决方案中只有一个项目，所以“生成解决方案”和“生成 test”是一样的，就直接生成解决方案吧，快捷键为 Ctrl+Alt+F7。编译链接输出的报告在“输出”子窗口中，如图 1.20 所示。此报告显示，所写代码正确无误，成功进行编译链接，生成的可执行程序的名字为“test.exe”，保存在目录“C:\Users\malei\documents\visual studio 2010\Projects\test\Debug”下。

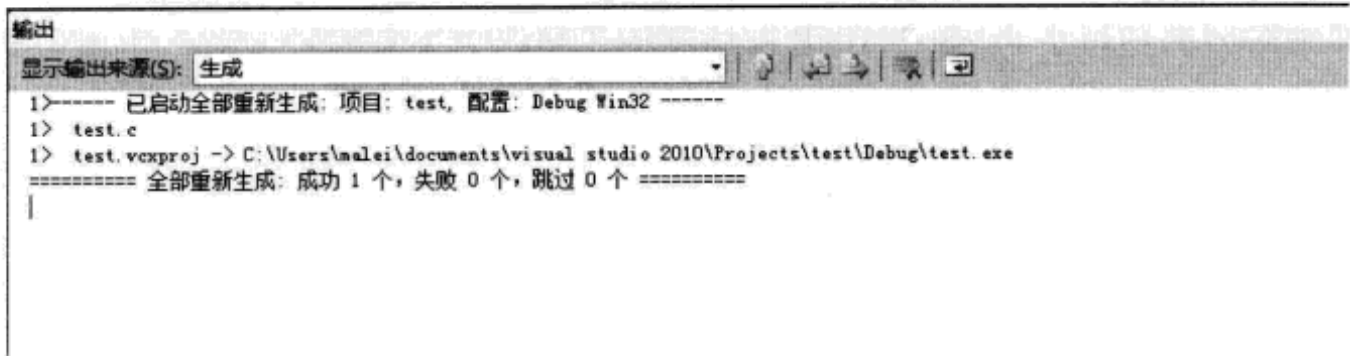


图 1.20 编译链接报告

1.3.6 运行可执行程序

可执行程序生成好以后，就可以看看程序的运行结果了，这也是我们写代码的最终目的。VS 有专门的菜单可以帮助你直接运行生成的程序，即菜单栏中的“调试”菜单，如图 1.21 所示。

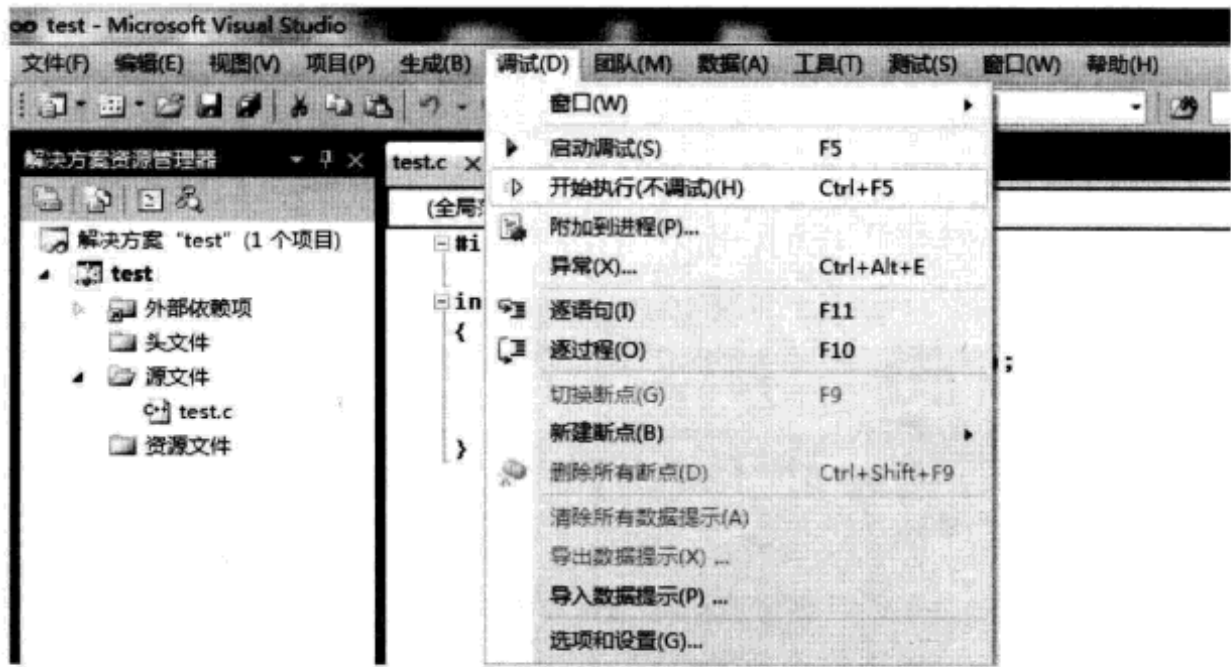


图 1.21 “调试”菜单

如果只需要运行一下生成的可执行程序，单击“开始执行（不调试）”菜单项就可以了，刚才生成的“test.exe”可执行程序的运行结果如图 1.22 所示。程序只是简单地在控制台窗口中输出了一行英文单词“C Programming Language!”。

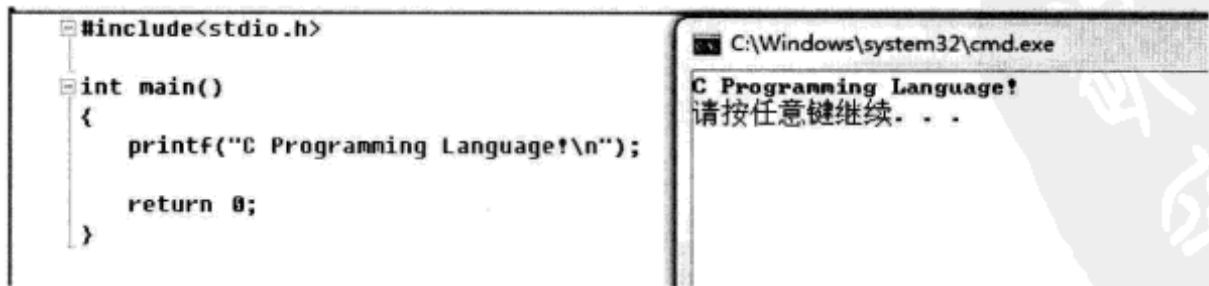


图 1.22 运行输出

好，到现在为止，就可以简单地使用 VS 了。虽然有很多的功能菜单和工具没有仔细讲解，但是，上面讲的知识足够你把 C 语言代码变成一个可执行程序了。要实现更多更复杂的功能，可以等需要的时候，再参阅 VS 的开发文档。

如果使用其他版本的 VS 或者其他的集成开发环境，参照本节的内容，找到对应的几个点，在你的集成开发环境上照猫画虎就可以了。这些点分别是：

- (1) 如何创建解决方案（有的集成开发环境没有解决方案这个概念，可以略去此步）；
- (2) 如何创建项目（集成开发环境一般都有项目的概念）；
- (3) 如何新建源文件，并将其添加到对应的项目中；
- (4) 如何编译链接，生成可执行程序；
- (5) 如何运行可执行程序。

只要找到这 5 个方面，就可以简单使用一个集成开发环境了，更复杂的功能就参考开发文档吧！

1.4 如何学好 C 语言

对于如何学好 C 语言，每个人都会有自己的看法和方式，所以并不苛求非得如何如何，只要你觉得能帮助你提高就可以了。这里，我只将自己的一些看法写下来，只给大家提供一个借鉴。从我自己的学习经历来说，要学好 C 语言，只要能够真的做到三点就好，“多看、多写、多想”。哇！好空洞啊，还是来解释一下吧！

- “多看”，看什么呢？看别人写的代码。当你没有任何开发经验时，唯一能做的就是看别人是怎么做的，只要能用自己已有的 C 语言知识解释通别人写的代码是什么意思，就算有长进了。如果死活看不懂，那么就得继续巩固自己的 C 语言储备了。
- “多写”，指的是自己多动手写代码。“纸上得来终觉浅”，看别人的代码终归是别人的，只有使用 C 语言自己写代码解决了实际问题，得来的知识才是自己的。而且，在写代码的时候，往往还会遇到其他问题，解决好这些问题也是一种收获啊！
- “多想”，是贯穿于“多看”和“多写”之中的，不要只停留在表面现象，要多问几个问题：“为什么是这样的？”、“这样可不可以？”、“我感觉他是错的！？”等等。只有通过思考，才不会仅仅停留在肤浅的表面

好了，以上三点，称之为“三多”吧！觉得有用的话，不妨试一试，贵在行动，“纸上得来终觉浅”！

1.5 小 结

本章是 C 语言的概述，主要讲了一些题外话。不过这对之后的 C 语言学习来说，还是有一定用途的，希望大家学习后面章节的时候，多回过头来看一看本章内容，一定会有所收获的。从下一章开始，我们就真正进入 C 语言的世界了，准备好了吗？

1.6 习 题

【题目 1】 没有集成开发环境可以进行 C 语言开发吗？

【分析】 当然可以了，要想把 C 源代码变成可执行程序，只需要三个工具：编辑器、编译器和链接器。只要你找齐了这三个工具就可以开工了，不介意的话，记事本+cl.exe+link.exe 就可以进行 C 语言开发了，只不过麻烦了点，还得知道 cl.exe 和 link.exe 的具体用法！

【题目 2】 集成开发环境对于开发者来说，是好还是坏？

【分析】 这个问题的答案就不能那么绝对了，好坏得看环境，还得看对谁而言。集成开发环境最大的好处就是使用方便，可以极大地提高开发的速度；集成开发环境最大的坏处就是它隐藏了很多很重要的细节，而且很多时候正是由于对这些细节的模糊，导致很难发现程序中的错误！所以，不要只依赖于集成开发环境的方便，当你很熟练地使用一个集成开发环境的时候，探索一下最基础最核心的工具——编辑器、编译器和链接器，你一定会有更大发现的。



第2章 开始 C 语言之旅

从本章开始就算真正进入了 C 语言的世界了！不过，不要着急动手写代码，欲速则不达，应该先从最基本的 C 语言概念开始打好基本功。本章将先从与数据有关的 C 语言基本概念开始介绍。

2.1 为什么要写代码

计算机语言是用来编写代码的，C 语言也不例外。那么，再往深处想一想，为什么要写代码呢？难道仅仅是因为好玩吗？不是的，如果仅仅是因为好玩，玩完之后就没意思了，也不会有那么多丰富多彩的软件了。

2.1.1 为什么要写程序

“工具都是给懒人发明的”。铁器时代之前，人们都是用手或者未经雕琢的木头来挖掘食物或者攻击猎物。人们发现这样很费力，然后就把石头进行打磨做出了石头工具。懒人可以用石头工具偷懒了。后来的青铜器时代、铁器时代也是如此。到了计算机时代，最伟大的发明就是计算机了。它可是很多懒人的福音啊！

呵呵，开个玩笑。不一定用计算机的都是懒人，不然我也是懒人一个了。不过计算机确实从各个方面给人们的生活带来很大的方便。

1. 计算机很给力

买东西可以用淘宝，足不出户，就可以货比三家，挑出理想的宝贝；再也不用为一封家书而苦苦等待好几个星期了，E-mail 瞬间可以传递你的问候，或者收到家人的祝福；记录和保存文档再也不必费力地用草稿纸去手写，电子文档传输方便而且易于保管……诸如此类，计算机确实是个很给力的工具。

2. 程序让计算机更给力

计算机是个又笨又努力的家伙，空有一身蛮力，但是不知道该干什么。它需要程序告诉它，什么时候该做什么，什么时候不该做什么。

只有写出好的程序才可以发挥计算机最大的潜力，操作系统（如 Windows）就是一个很实用的程序。但是，只有操作系统还不够，操作系统只是为各种优秀的软件提供一个公共的平台。可以想象一下，要是 Windows 上没有装任何软件，要它有啥用？练习开机和关机吗？所以，类似于 Word、QQ、浏览器、音乐播放器这些多姿多彩的应用程序配合操作

系统使得计算机更加给力。

2.1.2 从本书开始学编程

有这样一类人，他们或许已经具备普通计算机的使用能力，或者连普通计算机的使用能力都不具备，如不清楚 QQ 为何物。但是，他们立志做一个软件开发者，编写一个 Office。如果你是这类人，那么本书非常适合你。

本书将会告诉读者，开发者如何用语言将自己的意思告诉计算机，来完成想要完成的目的。在这里，使用 C 语言来阐述我们的观点，同时给出大量的 C 语言编程例子，来满足大家强烈的动手欲望。

我们所要达到的目标就是使初学者不仅仅学会 C 语言的语法规则，还能够知道计算机语言是什么，以后遇到新的计算机语言时能够很快搞定。读者学习完本书，能说出“C 语言也就那样!”、“计算机程序设计语言也不过如此!”的豪言壮语，我们的目的就达到了。

2.1.3 从一个现实的例子开始

再远大的理想，也需要一步步来走。这里先从第一个例子开始。例如，我们工作一个月，老板要给发薪水了。在合同中规定好了，每个月 3000 块钱，按照国家的法律，每个人要交 5% 的个人所得税。那么，我们辛辛苦苦一个月到底能够拿到多少血汗钱呢？

1. 普通人的做法

聪明人脑瓜子一转，很简单，3000 乘以 5% 是 150，3000 去掉 150 就是 2850 了。反应稍微慢点的人可能会拿出一支笔，一张纸，写出如下的计算式子：

$$\begin{array}{r} 3000 - 3000 \times 5\% = ? \\ 3000 \\ \times 0.05 \\ \hline 150.00 \\ \\ 3000 \\ - 150 \\ \hline 2850 \end{array}$$

所以， $3000 - 3000 \times 5\% = 2850$ 。

2. 程序员的做法

那么程序员是如何做的呢？咣、咣、咣……敲几行代码，然后按几个按钮，弹出了一个黑色的窗口。窗口上面写着 2850。如果你只惊叹地喊出一句：“哇，好牛啊！”，如图 2.1 所示，然后不去思考，不去查资料，不去学习这是如何做到的，恭喜你，你没有做一个软件开发者或者说程序员的潜质。呵呵，开个玩笑，我是希望大家都要有求知欲。

那我们求知一回，程序员咣、咣、咣……敲了什么代码？代码的意思是什么？按下的那几个按钮干了什么？弹出的黑色窗口是什么？窗口上为啥会显示 2850？大家一下变成“十万个为什么”了。而本书主要就是解决这些问题的！

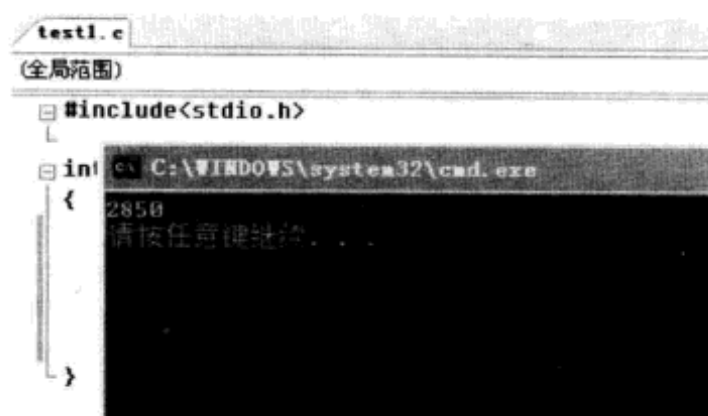


图 2.1 代码和结果

3. 如何解决“十万个为什么”

敲的代码是什么？代码的意思是什么？黑色的窗口上为啥会显示 2850？……这些都是编程语言的问题了，将在本书中占大量篇幅。程序员按的那几个按钮是什么，干了什么？弹出的黑色窗口是什么？……这些都是开发环境的问题了。这也是本书的一个重点，通过实例来满足大家的动手欲望。

带着这些问题，开始我们的 C 语言之旅！相信“旅游”回来时，会很好地回答这些问题，并且觉得这些问题好傻瓜啊！但是记住，这些问题现在并不傻瓜，而且还很有意义！

2.2 编程的核心——数据

计算机有个最核心的工作，那就是运算，对什么进行运算呢？对数据进行运算！计算机程序设计语言是用来指导计算机进行运算的。因此，使用计算机语言进行编程的核心，就是操作数据。

2.2.1 数据从哪里来

再来看看前面举的那个发工资的例子：工资 3000、税 5%，税后工资是多少钱？闭上眼睛，你还能记得起这句话中什么样的信息，或许最应该记住的就是 3000 和 5%了。要计算税后工资，恐怕没有这两个数字是不行的吧？这两个数字正是这句话的关键数据。接下来看看什么是数据。

数据是对客观事物的符号表示，是用于表示客观事物的未经加工的原始素材。它可以是图形符号、数字、字母等。在 C 语言中，简单地说，数据就是表示一定有用信息的事物元素的符号或者信号。分解开来说吧，首先，数据是事物元素；其次，数据必须包含有用信息；最后，数据应该是个符号。

知道了要成为数据必须满足的三个条件之后，就一步一步地来看看在前面举的那个发

工资的例子中，哪些是数据，哪些不是数据？为什么是，为什么不是？

1. 数据是事物元素

关于发工资的例子中都有哪些事物元素，图 2.2 更直观地表示了出来。

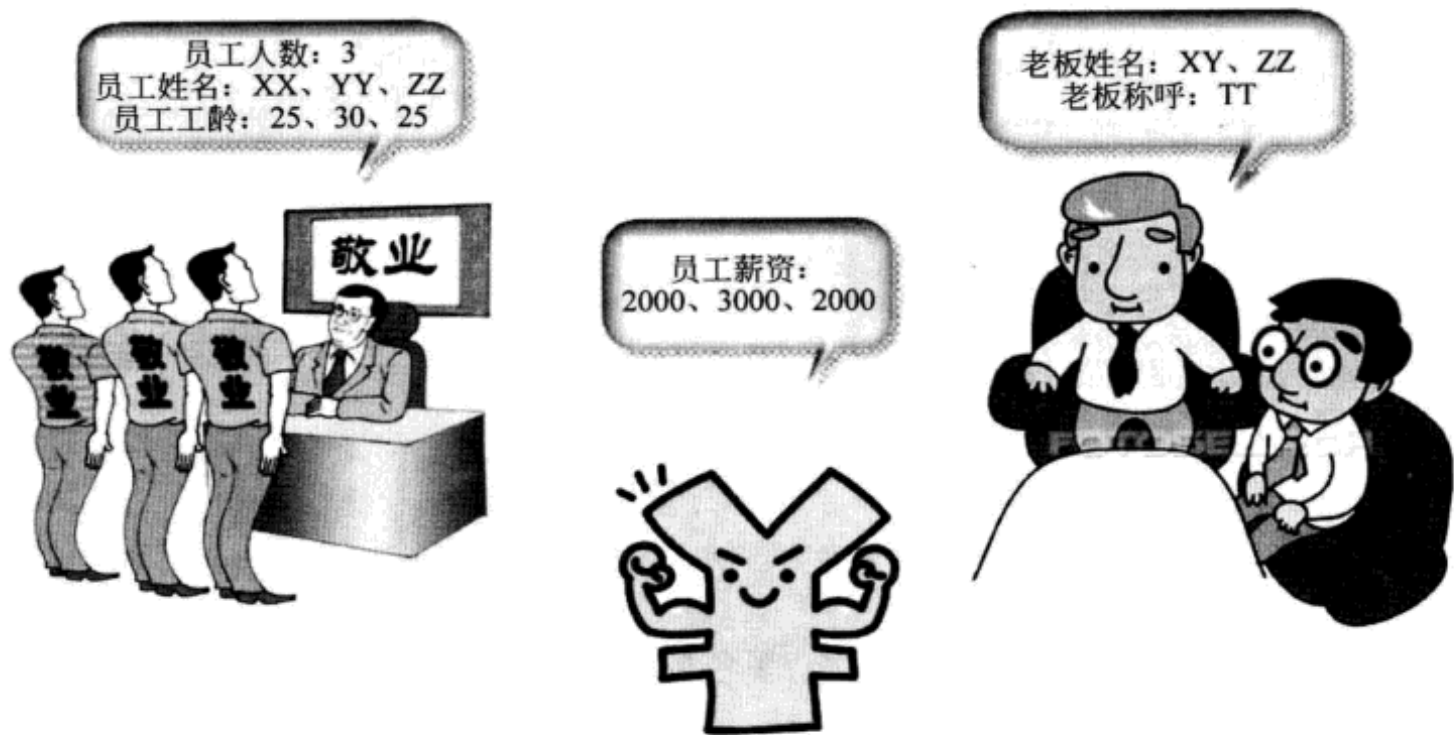


图 2.2 发工资中的事物元素

2. 数据是包含有用信息的

那么到底什么是有用信息呢？有没有用要看对什么而言。例如，一块橡皮，对于写错字的人可谓是雪中送炭，但是对于不写字的人，简直都没有石头有用。信息的有没有用是相对于要描述的事物而言的。例如，要计算税后工资，那么 3000 的基本工资和 5% 的税率就是有用的信息。发工资的老板是谁，就不是有用信息了。再例如，要完成公司财务报表的打印，那么发工资的财务经理的名字及发工资的时间就是有用信息，财务报表要填写这些信息。对于这两种场景的有用信息的比较，如表 2.1 所示。

表 2.1 有用信息

计算税后工资	报表打印
工资: 3000	工资: 3000
税率: 5%	税率: 5%
税后工资: 2850	税后工资: 2850
	发工资时间: 2010-1-12

3. 数据应该是个符号

那么到底什么是符号？符号就是具有一定意义的数字、字母、汉字、图片、声音等。例如，数字符号 1、字母符号'A'、汉字符号“我”都是符号。

以发工资为例，里面包含的数据如图 2.3 所示，有 3000、5% 和 2850，分别是税前工资、税率和税后工资，都是数字符号。

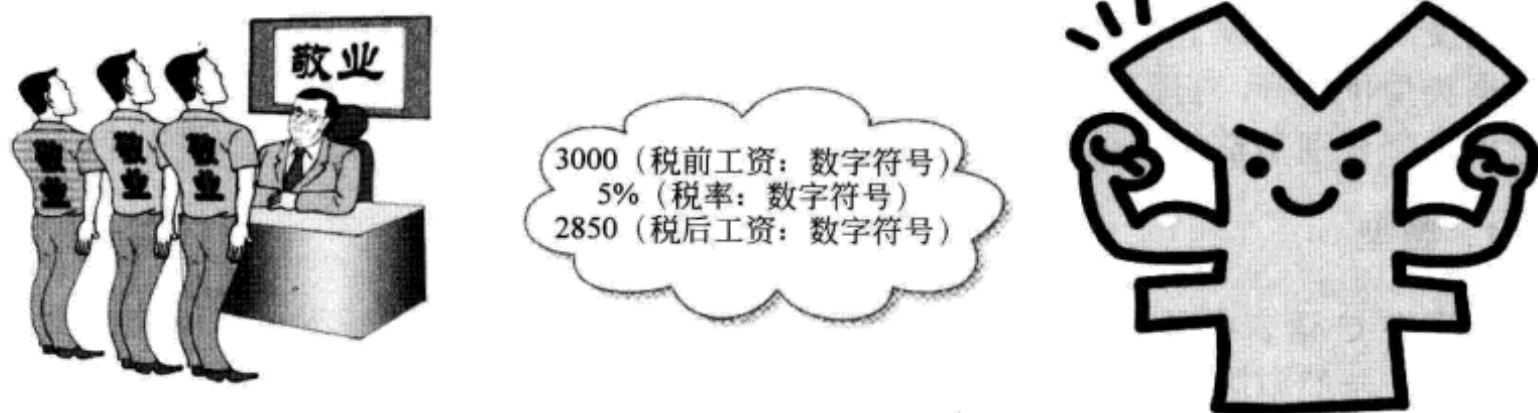


图 2.3 发工资中的数据

2.2.2 数据的表示

在前面描述计算税后工资的例子中，要告诉计算机：工资是 3000 元、税率 5%，计算它的税后工资。该怎么告诉计算机呢？而且，要确定告诉计算机工资是 3000 而不是 2999.99，税率是 5% 而不是 5‰，这就涉及数据在 C 语言中的表示了。

为了使数据能在 C 语言中容易表示，将数据分为了好多类！每一类数据被叫做属于同一种数据类型。对于每一种运算，不同的数据类型计算出来的结果都是不一样的。这就好比同样的化学纤维做出来的衣服类型不同，使用就不同。男士的西服让女士穿着就显得别扭，女士的裙子要是被男士穿着就闹出尴尬了，更有甚者，总不能把裤子穿在头上吧！所以，得分清楚数据的数据类型，然后才能正确地使用它们。

2.2.3 数据类型面面观——精度和范围

在了解 C 语言数据类型之前，先了解两个概念——精度和范围，这是讨论数据类型的重点。

1. 精度

精度是用来说明可以表示的数据的最小粒度，也就是可以表示的精细程度。讲个笑话，一个人过收费站，过磅整车重 4.29 吨。然后司机上了一趟厕所，又过收费站，过磅，整车重 4.20 吨。一趟厕所下来，减少了 0.09 吨，共 90 公斤。电子秤只能称到 0.1 吨，所以司机不上厕所对于称重没有影响，因为电子秤的精度不够。这个故事告诉我们不必憋尿来增加车重，没必要，呵呵……

2. 范围

范围是用来说明可以表示的数据的最大尺度，也就是可以表示的广度。不同数据表示要使用不同的广度，不然就会出现用游标卡尺去量万里长城的笑话。游标卡尺虽然精度很细，但是范围远远不够，估计把长城量完至少也得好几年吧，还不如用卫星直接去测呢，瞬间搞定！

2.2.4 C 语言基本数据类型

精度和范围（也就是广度）之间得有个权衡，很多时候有精度没有广度，或者是有广

度没精度。就像游标卡尺和卫星测量一样，需要我们适当选择。在 C 语言中，有三种基本的数据类型供选择，它们有着不同的精度和广度，可以根据自己的需要选择合适的。这三种数据类型分别是整型、浮点型、字符型，它们可谓是 C 语言数据的三大变形金刚。

1. 整型

整型在 C 语言中是用来表示整数的，主要用符号 `int` 表示。为了满足不同的需要，整型按照可表示的范围分为以下几类：整型（或者叫一般整型）、短整型、长整型。短整型用来表示较小范围的数，长整型用来表示较大范围的数。这就好像你要装水得有瓢或有水桶或有缸一样，都是用来装水的，但是分别用于不同的用途。对于不同的整型，在 C 语言中使用如下不同的符号表示。

- ❑ 整型：使用 `int` 关键字表示；
- ❑ 短整型：使用 `short int` 关键字表示，简写为 `short`；
- ❑ 长整型：使用 `long int` 关键字表示，简写为 `long`，可以省略 `int` 不写。

整数分为正整数、负整数和零。有的时候需要表示整数既可以是正的也可以是负的，有的时候只需要表示正的整数。就像我们表示人民币的时候用的都是正数或者零元，表示温度的时候既可以是负一度，也可以是正一度。

为了满足上面的需求，C 语言中的整型又可以分为有符号的和无符号的。有符号整型可以表示正整数、零和负整数，有符号使用 `signed` 符号表示。无符号整型只可以表示正整数和零，无符号使用 `unsigned` 符号表示。

因此，三种整型分别和有符号、无符号组合，因此整型总共有 6 种。

- ❑ 有符号整型：简称整型，用 `signed int` 关键字表示，简写为 `int`；
- ❑ 有符号短整型：简称短整型，用 `signed short int` 关键字表示，简写为 `short int` 或者 `short`；
- ❑ 有符号长整型：简称长整型，用 `signed long int` 关键字表示，简写为 `long int` 或者 `long`；
- ❑ 无符号整型：用 `unsigned int` 关键字表示；
- ❑ 无符号短整型：用 `unsigned short int` 关键字表示，简写为 `unsigned short`；
- ❑ 无符号长整型：用 `unsigned long int` 关键字表示，简写为 `unsigned long`。

列表来表示每一种整型的范围和适合表示的数，如表 2.2 所示。

表 2.2 6 种简单数据类型

类 型 名	表 示 符	精 度	表 示 范 围	适合表示的数字
有符号整型	<code>int</code>	1	-2^{31} 到 $2^{31}-1$	大整数
有符号短整型	<code>short</code>	1	-2^{15} 到 $2^{15}-1$	小整数
有符号长整型	<code>long</code>	1	-2^{31} 到 $2^{31}-1$	大整数
无符号整型	<code>unsigned int</code>	1	0 到 $2^{32}-1$	大正整数
无符号短整型	<code>unsigned short</code>	1	0 到 $2^{16}-1$	小正整数
无符号长整型	<code>unsigned long</code>	1	0 到 $2^{32}-1$	大正整数

从表 2.2 中可以看出长整型和整型是一样的，其实对于不同的系统，这个范围是不一样的，长整型会比整型的范围大，但是现在一般系统都把它们定义为一样的。

2. 浮点型

浮点型在C语言中主要用来表示小数，分为单精度浮点类型和双精度浮点类型。它们的区别仅仅是表示的精度不一样，双精度的表示精度会更细点，并且表示的范围也会更大。单精度浮点类型使用 `float` 关键字表示；双精度浮点类型使用 `double` 表示，简写为 `double`。浮点型没有有符号和无符号的区别，都是有符号的。下面列表加以比较，如表 2.3 所示。

表 2.3 浮点类型

类 型 名	表 示 符	精 度	表 示 范 围	适合表示的数字
单精度浮点型	<code>float</code>	3.4×10^{38}	3.4×10^{38} 到 3.4×10^{-38}	精度要求不高的小数
双精度浮点型	<code>double</code>	1.7×10^{308}	1.7×10^{308} 到 1.7×10^{-308}	精度要求高的小数

3. 字符型

字符型在C语言中表示各种字符，如英文字母、标点、数字之类。它只是一种符号表示，就像我们用 ‘A’，一撇、一捺、一横来表示大写英文字母 A；用 ‘1’，一个竖线来表示数字 1；用 ‘，’，类似于豆芽的东西表示语文中的逗号分隔符。

所以，对照上面几种数据类型的表示方法，可以使用整数来表示 3000 元的工资，用浮点型来表示税率 5%。

2.2.5 数据的变与不变——变量、常量

一旦了解数据后，就会发现身边到处都是数据。万事万物都可以用数据表示。就像黑客帝国酷哥尼奥看待自己所在的虚拟世界一样：神马都是数据。我们老祖宗更超前：“万变不离其宗”、“道生一、一生二、二生三、三生万物”。万物已经变成“一、二、三”了。我们没有那么高深，只能看到数据是变与不变的。

例如，当需要计算圆的面积时，事先，圆的半径是不知道的。半径是变化的，计算出来的圆的面积也是随着圆的半径而不断变化的。然而，计算过程中有一个量是一直不变的，那就是圆周率 3.1415926（为了便于大家阅读，这里省略圆周率的后面无数位。有兴趣的读者，可以自己手工计算）。在C语言中把计算中变化的这些量叫做“变量”，把这些不变的量叫做“常量”。

1. 如何区分变量和常量

为了区分常量和变量，再来看看那个发工资的例子：工资是 3000、税率是 5%，问税后可以拿到多少薪水？在计算过程中，工资是不变的（3000），是一个常量。税率 5%也是不变的，也是一个常量。计算结果虽然还不知道，但是可以肯定它也是不变的，也是一个常量。

有人会问，在这个例子中就看不到变量了吗？我们把这个例子改改，或者说把计算的问题改改，就会出现变量了。

假如，你是公司的会计，老板要发工资了，要你计算出该给每个人发多少薪水。这时，

你得考虑，每个人的税前工资都是不一样的，按照国家规定，处于每个工资段的税率也是不一样的。这样，要计算每个员工的税后工资时，税前工资、税率就都是变化的了，如图 2.4 所示。

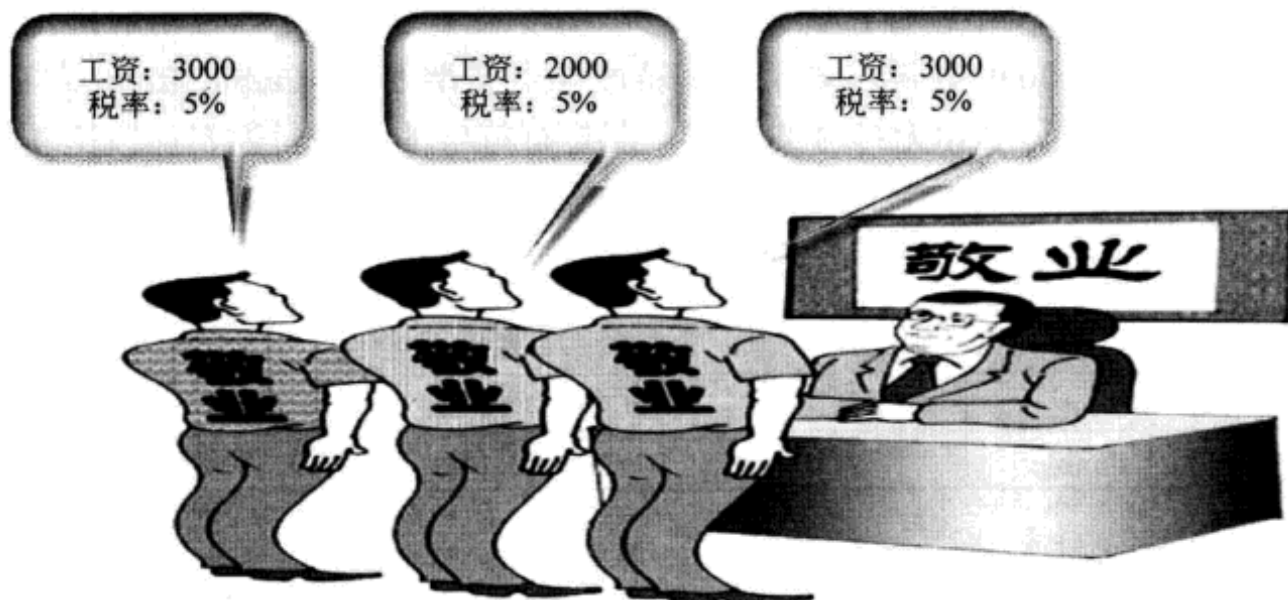


图 2.4 常量和变量示例

在图 2.4 中可以看到有三个员工，他们的工资分别是 3000、2000 和 3000，税率为 5%。我们可以看出税前工资是变化的，要给每个人发工资，就得用一个变量来表示税前工资。税率不变，我们就用常量表示吧！

这个时候，就出现变量了。同样的计算方法，对于不同的工资和不同的税率，结果也是不同的。所以在用 C 语言编程时，就得用变量来处理。

2. 回到C语言

从上面的描述可以看出这样一条准则：要进行 C 语言编程，先得确定出哪些数据要作为变量处理，哪些数据要作为常量处理。判断的依据就是在用 C 语言处理时，这个数据是一直变化的还是一直不变的，这就是区分变量和常量的准则。

2.3 使用变量和常量

在数据的世界中，数据被分为一成不变的常量和一直变化的变量。这样划分以后，计算机操作数据也变得方便多了，只要计算机能操作变量和常量，那么它就可以操作任何数据。本章就来看看在 C 语言中如何使用变量和常量！

2.3.1 变量的使用

在 C 语言中，要使用变量，就得先对变量进行声明和定义。变量的声明是说明变量的类型，而变量定义是创建要求的变量。通俗地说，声明变量就是告诉计算机“我要的是浮点型的变量，记住，一定是浮点型的！”；变量定义就是告诉计算机“我要的是浮点型的变量，现在就要！”，如图 2.5 所示。两者差别在于，声明变量的时候，只是要求，计算

机并没有创建，而只有定义的时候才会创建变量。



图 2.5 变量声明、定义示意

在 C 语言中，变量的声明和定义是一体的。其语法形式如下：

变量类型 变量名；

其中，变量类型是要求计算机给的变量的类型；变量名是我们给问计算机要的变量取一个名字，以后就可以用这个名字，来“召唤”计算机给我们的变量了。

有人要问了，就像你说的，我要计算圆的面积，半径事先不知道，按照变量来，我也事先确定了半径是小数。我需要一个浮点型的变量来表示半径，在 C 语言中，浮点型的变量该怎么表示呢？

要计算圆的面积，定义的半径可以表示为：

```
float radius;
```

其中，float 是变量的类型名称，radius 是变量的名字。这样表示做了两件事情，即半径变量的声明和定义。即告诉计算机，“给我一个浮点型的变量，记住一定要浮点型的”，又告诉计算机，“给我一个浮点型的变量，现在就给我”！

2.3.2 命名的方式

在上一节进行变量的定义和声明的时候，使用了两个概念：类型名和变量名，其实它们都是 C 语言中的标识符。标识符就是用来表示 C 语言中的各种东西的符号，如用来表示类型、变量等。每一个标识符，是类型名也罢，是变量名也罢，都是一个名字。名字都要遵循一定的命名方式。命名方式涉及两方面的内容：构成元素和构成方式。

构成元素就是 C 语言标识符的组成符号，构成方式就是 C 语言标识符的命名规则。

1. 构成元素——符号

一门语言是由不同的符号组成的。例如，汉语是由不同的汉字和标点符号构成的。同

样，C 语言也是一门语言，组成它的符号有三类：命名符号、分隔计算符号、数值符号。这样表述简洁，如表 2.4 所示。

表 2.4 用于命名的符号

命 名 符 号															
大写字母	A	B	C	D	E	F	G	H	I	J	W	X	Y	Z
小写字母	a	b	c	d	e	f	g	h	i	j	w	x	y	z
阿拉伯数字	0	1	2	3	4	5	6	7	8	9					
分隔计算符号															
分隔符	()	{	}							;			
计算符	+	-	*	/	%	->	>	<	==	!	<=	>=		
数 值 符 号															
阿拉伯数字	0	1	2	3	4	5	6	7	8	9					
指数表示符	E	e													

- 命名符号主要用于组成标识符，如英文字母和阿拉伯数字。给孩子取名字的时候，姓名都是汉字，而且姓得是百家姓。这里的汉字就类似我们的命名符号。
- 分隔计算符号是用于 C 语言分块和计算的。例如，大括号主要用于分块，小括号主要用于数学计算等。
- 数值符号在 C 语言中表示数值。

本节只关注第一类用于命名标识符的符号，其他两类暂且不用关注。

2. 构成方式

就像我们的姓名一样，C 语言的标识符构成也有规范标准。在 C 语言中，标识符由首部和其他部分构成，如图 2.6 所示。首部就相当于我们的姓，其他部分就相当于我们的名。



图 2.6 标识符组成

- 首部（姓）就是变量名的第一个组成元素。C 语言要求首部只能使用大写或者小写的英文字母和下划线。
- 其他（名）是变量名中除去首部以外的部分。C 语言对此没有要求，只要是大小写的英文字母、数字和下划线就可以。

为了理解标识符各个部分的差异，举个例子。在给孩子取名字的时候，姓名必须是汉字，姓还必须是百家姓里的一个。标识符也是如此，首部和其他部分都是命名符号，首部还要求一定不能是数字。

按照 C 语言对标识符名称的规范，像我们之前见到的，浮点数据类型“float”和变量名“radius”都是合格的标识符。

2.3.3 关键字

关键字就是 C 语言系统自己保留的标识符。就像我们生活中的“警察”、“医生”之

类的词语，一旦有人冒用，就会受到严厉的惩罚。C语言中的关键字如表 2.2 所示。

表 2.5 关键字

auto	break	case	char	const	continue	default
do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short
signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while			

这 32 个标识符，大家先过一过眼，后面遇到的时候，会详细地讲解每个关键字的意义和用途，在此就不做赘述了。

⚠注意：C语言规定，用户自定义的标识符不能和关键字相同。

像前面提到的浮点类型的类型名 float 就是关键字，它们都不能再被用做其他用途了。像变量名 radius 就是一个用户自定义标识符，它既可以在我们的例子里标识圆的半径，下次再计算关于球体的体积时也可以用来标识球体的半径。

2.3.4 常量的使用

常量用来表示各种不变的东西。在 C 语言中，提供了很多常量的表示方法，下面先来学习其中最简单的两种。

1. 直接表示

这种方法最简单，直接用数字、字母来表示就可以了。

像发工资的那个例子，税前工资 3000，税率为 5%，要求计算税后应该拿多少薪水。其中数字常量有工资 3000 和税率 5%，我们就可以直接使用整型常量 3000 和浮点型常量 0.05 来表示。C 语言不能直接使用百分数，没有这样的分数类型，所以用浮点型 0.05 来表示 5%。

2. const变量表示

const 变量也是一个变量，只不过是一个很特殊的变量。它的特殊之处在于不用来表示变量，而只用来表示常量。

const 变量就是告诉计算机，“我要一个变量，而且我往变量里放了东西以后就不准拿出来再放其他东西了，这个变量被我这个东西永久性独占！”是不是太霸道了？！因此，const 变量拥有了常量的一切特性，用来表示常量，人们也不把它叫变量了，干脆直接叫常量。

C 语言中 const 常量使用 const 关键字+变量定义的形式，具体如下：

```
const 变量类型名 变量名=变量要赋的值;
```

工资 3000 和税率 5%的 const 常量表示形式如下所示：

```
const int salary = 3000;
const float tax_rate = 0.5;
```

通过这两种方式，就可以告诉计算机在程序中保留一个一直不变的常量了。

2.4 小 结

本章中主要讲解了计算机中的几个概念：数据、变量和常量，以及进行计算机编程的作用。本章的重点就是理解这几个概念的含义，难点是如何进行变量的命名、定义，还有就是常量的使用形式。在下一章中将继续深入，介绍 C 语言中的简单数学运算，看看这些运算是如何使用变量和常量来操作数据的。

2.5 习 题

【题目 1】“地球”是不是一个数据？

【分析】要成为一个数据，必须满足三个条件：首先，数据是事物元素；其次，数据必须包含有用信息；最后，数据应该是个符号。“地球”要想成为一个数据，就得满足这三个条件，在描述银河系的时候，“地球”算得上是一个事物元素。地球是银河系的一员，具有一定有用的信息。我们可以在计算机中使用汉字符号“地球”来表示。所以，对于描述银河系这件事情来说，“地球”是一个数据。但是，对于其他事物，地球就不见得是数据了！

【题目 2】从精度上来说，浮点型的精度比整型精确，从表示范围上来说，浮点型表示的范围比整型表示的范围大。那么，为什么还需要整型呢？

【分析】“尺有所长，寸有所短”，使用雕刻刀去森林里砍木材，不是不可以，但是总觉得很别扭，有的整数浮点型是表示不出来的！每种数据类型都有优缺点，有的时候可以互相代替，但是有的时候就非得某种类型不可了。至于这些“特殊时候”，深入理解了各种数据类型之后，自然就会发现的。

【题目 3】假设要在程序中表示一个班学生的身高信息，你是打算使用常量还是变量呢？

【分析】判断该使用常量还是该使用变量的依据是在计算过程中，要表示的数据是一直变化的还是一成不变的。要表示一个班学生的身高信息，从整个班级来看，这是一个变化的数据，得使用变量来表示。从具体某个学生来看，这个数据是一成不变的，得使用常量来表示！所以，要使用变量还是要使用常量，需要看你针对的是什么具体的事物。就像身高这个数据，如果你想一下把全班学生的身高在程序中都表示了，就使用变量吧，如果你只是表示某个学生的身高，常量就够了！

【题目 4】在 C 语言中，能不能使用 5%、1/4、6‰这样的数字表示作为常量呢？

【分析】C 语言有三种基本数据类型：整型、浮点型和字符型。整型是用来表示整数的，浮点型是用来表示小数的，字符型是用来表示字母或者符号的。5%、1/4、6‰准确来说都是分数，虽然分数可以化成小数，但是它们毕竟不是小数，所以不可以使用浮点型来表示，更不用说用整型和字符型了。所以，既然 5%、1/4、6‰不属于任何类型，那么，它们就不可以作为常量出现在 C 语言的代码中。

【题目 5】声明定义一个变量，表示 6 月 8 号，街道上的人数。

【分析】 街道上的人数是一个整数，而且数目估计会比较大，因为这天天气不错，风和日丽！另外，人数总不可能是负的吧？因此，可以使用 `unsigned long int` 类型来表示人数，这个类型可以表示的整数确实大得可以，绝对超过 64 亿了，不会全世界的人都集中在这条街吧？！

C 语言中，要给变量命名，得用英文字母和数字，6 月 8 号的人数，翻译一下，有这些单词和数字 6、8、month、day、person 和 number。另外，C 语言规定数字不能作为变量名的首部，我们可以写出这样的名字 `person_number_6_month_8_day`，这个变量名是符合 C 语言要求的，就是太长了，再压缩一下，会比较好看点，`persons_6_8`。

【核心代码】

```
unsigned long int persons_6_8;
```

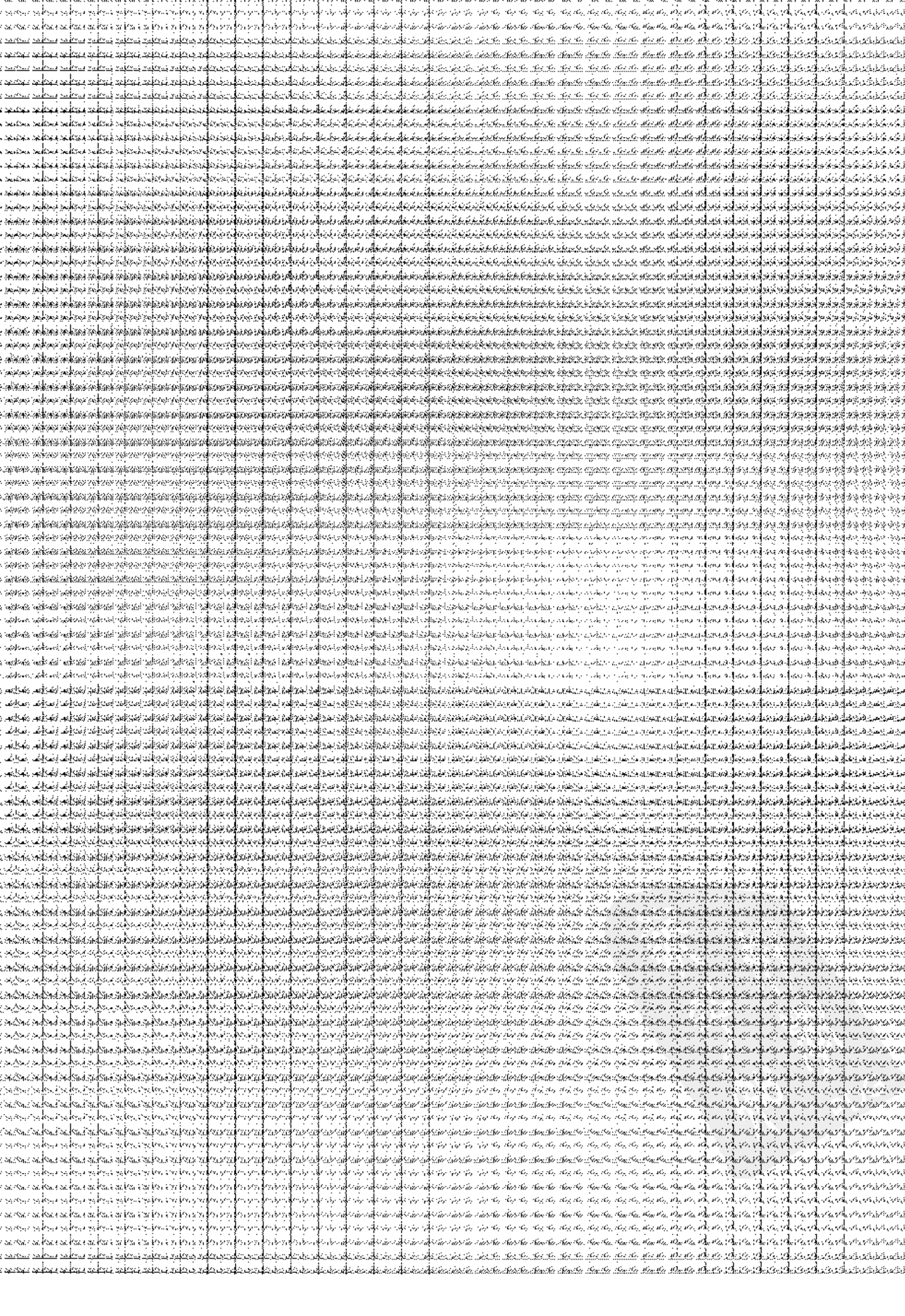
【题目 6】 要定义一个类型为 `char` 型，名字为案例的变量，该怎么给它取名字呢？

【分析】 要给变量取名字，就得遵循 C 语言变量命名规范！首先组成必须是字母、数字和下划线，案例的单词是 `case`，完全符合 C 语言命名规范，而且首部不是数字，是不是就可以使用它来给这个变量命名了？不行，`case` 是 C 语言中的一个关键字，C 语言规定关键字是不能作为变量名的！那么该怎么办呢？很简单，只要和关键字 `case` 不一样就可以了，如 `Case`、`CASE`、`case_`、`_case`、`_case_`、`case1` 都是可以的。

【核心代码】

```
char Case;
```





第2篇 简单程序的构建

▶▶ 第3章 简单数学运算

▶▶ 第4章 程序结构



第 3 章 简单数学运算

上一章已经学习了数据在计算机中的分类（数据类型）及表示（常量、变量）。有了数据，我们才踏出编写 C 语言程序的第一步，原材料只有经过加工以后，才能得到想要的结果。就像粮食蔬菜经过加工以后才可以做成香喷喷的饭菜，数据就是粮食，数学运算就是炉灶，计算得到的结果就是饭菜了。本章我们来讲解 C 语言中的“炉灶”！

3.1 什么是赋值

在正式进入 C 语言的计算世界之前，先看看所有的计算都会用到的一种基础的数学运算——赋值运算。

3.1.1 赋值的作用——把数据存起来

我们知道，C 语言中的变量用来表示在计算过程中一直变化的数据。没错，是这样的！但是，你有没有想过这些变化的数据在每个瞬间是什么？该怎么表示？每个瞬间是一个个可以用常量来表示的数据。只不过，这些数据变化的速度很快，如流星般在每个瞬间一闪而过，如图 3.1 所示，并不像我们提到的常量那样在计算过程中如日月般永恒存在。



图 3.1 一闪而过的数据

在计算圆的面积的时候，圆的半径一会儿是 3，一会儿是 4，唰唰唰地变个不停，一

闪过，就像黑客帝国里不断刷新的屏幕（图 3.1）一样。

计算机程序是要使用这些一闪而过的数据来计算所需的结果的，就像要使用 3、4、1 这些数据来计算出各自对应的圆的面积，不能让它就这样一闪而过了！那么该怎么办呢？找个东西存放起来！计算机中，所找的东西就是我们之前提到的变量，存放起来的过程就是赋值运算。

由此可见，赋值运算的作用就是把一闪而过的数据交给变量保存起来，以备他用。

3.1.2 赋值运算的形式

赋值运算就是给变量赋值的运算。要给变量赋值，就得有一个变量和一个值，然后用赋值运算符把它们连接起来，这样就构成了赋值运算的形式。C 语言中使用等号“=”作为赋值运算符，赋值运算的具体形式如下所示：

变量名 = 值

例如，先声明一个整型的半径变量，然后给它赋值为 1。在 C 语言中的表示就是下面这样：

```
int radius;  
radius = 1;
```

在 C 语言中规定：必须先声明、定义变量，然后才能给它赋值。例如，要想给圆的半径变量赋值，就得先定义圆的半径变量，取 int 类型：int radius，然后赋值为 1。这就像你给孩子取了名字以后，才可以喊他过来啊！

有的时候，为了简单，在变量声明定义以后就直接对它进行赋值。这样的操作在 C 语言中叫做变量初始化，因为是第一次赋值，所以叫“初始”，具体形式是：

变量类型 变量名 = 值

上面给半径赋值的 C 语言表示，也可以使用下面的初始化形式来代替：

```
int radius = 1;
```

3.1.3 赋值表达式

在 C 语言中，由运算符连接变量、常量等组成的式子叫做表达式。可以这样理解，表达式就是为了表达一定的意思而列的式子。

像上面讲到的变量的赋值和初始化的式子，就是为了表示赋值而列的式子，因而称之为赋值表达式。这样别人问你形如：

```
int radius = 1
```

的东西叫做什么的时候，你就不必手足无措了，它有一个专门的名字叫做赋值表达式。

3.1.4 机动灵活的赋值——scanf()

前面在给半径进行赋值的时候，我们预先知道这次要计算的是半径值为 1 的圆的面积，

因此，直接给半径 `radius` 赋值为 1。要是事先不知道要计算的圆的半径，还要计算圆的面积该怎么办呢？有人会说“这都行”？！这个真的行。

可以使用一个有用的函数 `scanf()`。先不管该函数是什么（我们会在后面告诉你的），现在只要你会用就行。`scanf()`函数的使用格式是这样的：

```
scanf("%变量类型表示", &变量名);
```

在这里，只要关注两个点：“变量类型表示”和“变量名”。变量类型表示就是用来表示变量数据类型的一个字母，变量名就是进行变量声明定义的时候给变量取的名字。例如，要使用 `scanf()`函数给整型变量 `radius` 赋值，使用的整型类型字母为“d”，完整的可运行的 C 语言表示如下所示：

```
#include<stdio.h>

int main()
{
    int radius;
    scanf("%d",&radius);
    return 0;
}
```

其中，

- ❑ `int radius;` 是声明定义一个整型的变量 `radius`，不赘述。
- ❑ `scanf("%d",&radius);` 是使用 `scanf()`函数给变量 `radius` 赋一个值，这个值来自终端（图 3.2 右边所示窗口）。遵照上面列出的 `scanf()`函数使用格式，`%d` 表示要赋值的变量的类型，使用字母 `d` 表示整型，`radius` 是要赋值的变量名，在 `scanf()`函数之前已经对其进行了定义。
- ❑ 对于除了变量声明定义和 `scanf()`函数以外的其他 C 语言代码，先不用关心，等到相应的部分时再详细讲解。

使用上面的代码，每次从终端（图 3.2 所示）输入一个数字，按回车键，这个数字就会被赋值到 `radius` 变量里，连等号“=”都省了，方便吧？而且还不必每次都提前知道半径的值，第一次输入 1，第二次可以输入 2。

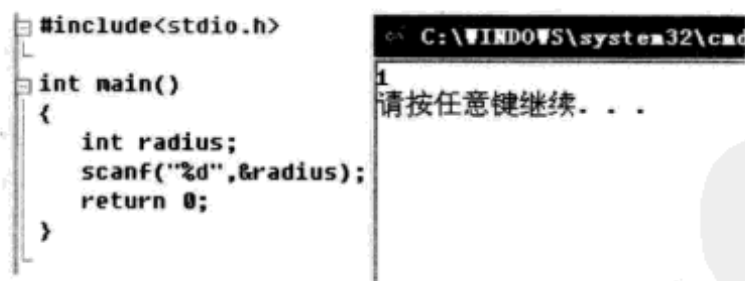


图 3.2 `scanf()`函数的使用

3.1.5 看看我们的劳动成果——`printf()`

我们使用 `scanf()`函数给一个变量按照自己临时的需要从终端赋值，但是怎么看到已经赋的值呢？你说赋值了就赋值了，我不信！哈哈……那就用另一个很有用的函数 `printf()`来检验检验！

printf()函数和 scanf()函数刚好有点相反。scanf()函数是从终端把一个输入的值赋值给一个变量，printf()函数是把一个变量的值输出到终端给我们看。所以，可以用 printf()函数来检验给变量赋的值。printf()函数的使用格式如下所示：

```
printf("%变量类型表示", 变量名);
```

和 scanf()类似，其中，“变量类型表示”的含义是要输出的变量数据类型的一个表示字母，“变量名”就是要输出的变量的变量名，最终的输出结果会显示在终端上。现在，来检验给半径变量 radius 赋的值是不是 1。完整的 C 语言具体实现如下所示：

```
#include<stdio.h>

int main()
{
    int radius;
    scanf("%d",&radius);
    printf("%d",radius);
    return 0;
}
```

代码运行的结果如图 3.3 所示。在终端上，第一行的“1”是输入的“1”，输入完“1”，按回车键，scanf()函数就把输入的“1”读入变量 radius 里了。之后，printf()函数把变量 radius 保存的“1”输出到终端上，那就是我们看到的第二行的“1”。

第二行的“1”后面还出现了一句“请按任意键继续...”，这是程序运行完的一个输出标志，表示程序已经成功运行完毕，你可以按任意键关闭这个终端。我们现在可以不用关心这个。

从上面的分析可以看出 printf()成功地验证了，可以使用 scanf()给变量赋一个任意从终端输入的值。以后需要验证的时候，都可以使用 printf()函数把要验证的东西输出到终端来仔细分析。

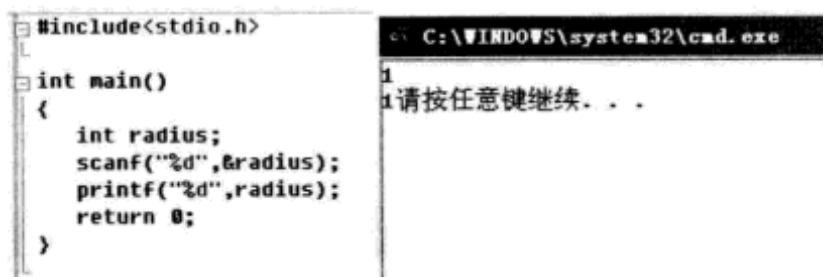


图 3.3 printf()函数的使用

3.1.6 赋值的重要性

赋值运算是将数据的值暂时保存在变量里，那么，保存到变量里到底有什么用呢？不能只是为了保存而保存，为了赋值而赋值。其实，赋值也是有很大用途的，它的重要性主要体现在两个方面：（1）为了方便；（2）为了避免出错。

1. 为了方便

举个例子，要计算这样的一个式子： $3.14 \times 5 \times 10^3 / 2.414 + 7.2 / 0.5 + 3.3333 + 8$ 。这个式子貌似比较复杂啊，估计直接去心算会有点困难，所以通常都是在草稿纸上一步一步地计算。

$$5 \times 10^3 = 5000$$

$$3.14 \times 5000 = 15700$$

$$15700 / 2.414 \approx 6503.728$$

.....

通过这样一步一步的计算,就可以得到这个数学式子的最终结果。对于像 5000、15700、6503.728、..., 这样的临时结果,我们在草稿纸上可以直接写出来以备后面使用。但是,在程序里如何“写出来以备后面使用呢”?那就得使用变量赋值暂时存下来了。所以说,赋值可以方便你对于临时结果的使用。

2. 为了避免出错

可以想象一下这样一种使用变量的情况:定义了一个变量,没有给它赋值,然后就使用了这个变量,会出现什么情况呢?用 printf() 函数验证一下这种情况。来看看这样一段代码:

```
#include<stdio.h>

int main()
{
    int radius;
    printf("%d",radius);
    return 0;
}
```

和前面的那段代码很相似,就是少了给已经定义的变量 radius 赋值,直接就用 printf() 函数输出了。输出结果如图 3.4 所示。程序没有输出,而是直接运行错误,这是一个很严重的问题。所以,在使用变量的时候,一定要给变量进行赋值,不然,小则导致一些意想不到的错误,大则导致程序崩溃,就像上面的例子那样。

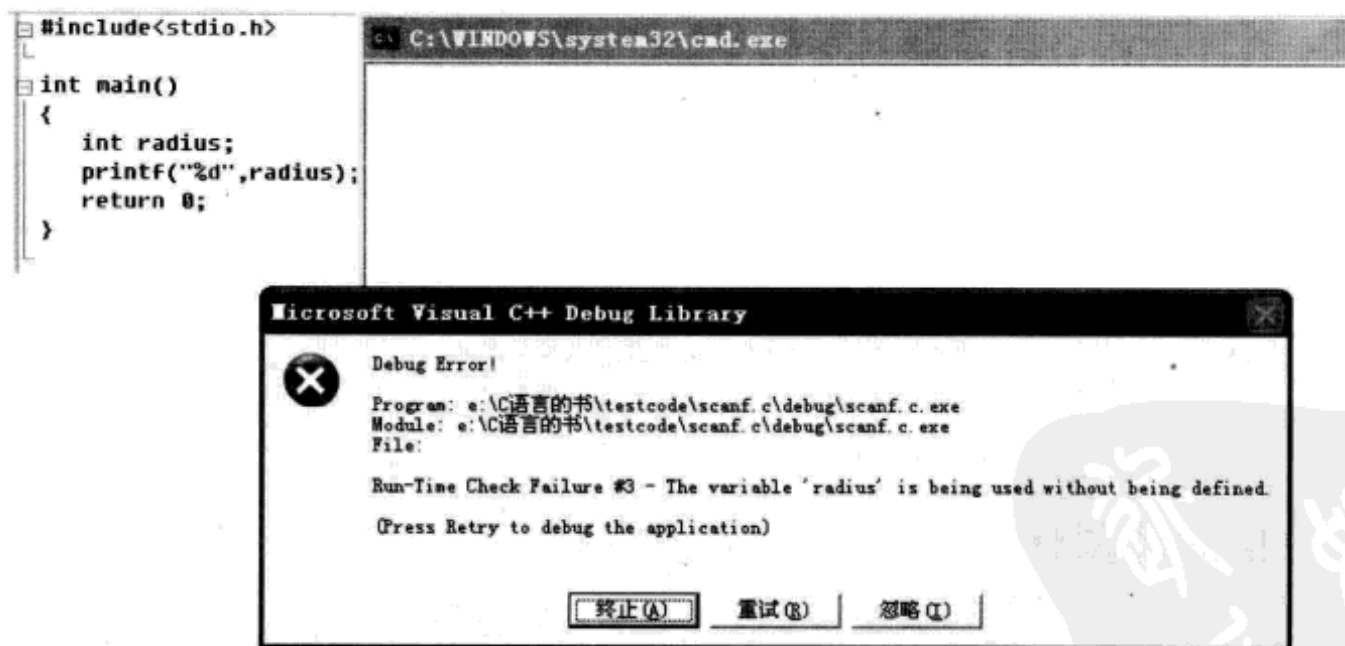


图 3.4 未赋值的变量

3.2 开始赋值——整型赋值

赋值在 C 语言中是很重要的、必不可少的一部分。接下来从最简单的开始,先来看看

C语言中是如何给整型变量赋值的，以及给整型变量赋值时需要注意些什么。

3.2.1 整数在计算机中的表示——二进制

要想深入了解计算机中的数据，就得先来好好了解一下计算机中的二进制表示。因为整个计算机的世界是一个二进制的世界。

1. 正整数的计算机表示

在生活中，使用 0、1、2、3、4、…、9 的各种组合来表示整数。像这样的逢十进一的数，被称为十进制数。但是，计算机由于电路设计的原因，只能表示两个状态 0 和 1，这就形成了二进制的计算机世界。与十进制不同的是二进制是逢二进一的，表 3.1 给出了一些简单的十进制数和二进制数的对应关系。

表 3.1 一些简单数字的二进制表示

十 进 制 数	二 进 制 数	十 进 制 数	二 进 制 数
2	10	5	101
3	11
4	100		

在表 3.1 中， $2 = 1+1$ ，逢 2 进 1，十进制的 2 就可以表示成二进制的 10。依此类推，十进制的 $3=2+1$ ，2 表示成二进制为 10，再加上 1 就是 11 了，因此十进制的 3 表示成二进制就是 11 了。十进制的 4 表示成二进制就是 100，十进制的 5 表示成二进制就是 101，诸如此类就形成了简单二进制数和十进制数之间的对应关系。二进制数表示中的每个 0 或者 1 都被叫做 1 位，101 有三位，分别是 1、0、1。

2. 负数的表示（原码表示）

上面的二进制表示的数都是正数，那么负数怎么表示呢？例如，在计算机中表示 0-1 的结果，该怎么办？可能你会很自然地想到 $0-1=-1$ ，用一个东西表示符号，然后再用一个东西表示-1 的绝对值就可以表示-1。你很聪明，确实有这样一种表示法，叫做原码表示，如图 3.5 所示。使用最前面的一位来表示符号，0 表示正整数，1 表示负整数，剩下的部分用来表示绝对值。

以-3 为例，来讲解一下原码表示。因为计算机中用到的 int 类型是用 32 位来表示整数的，为了统一我们也用 32 位来表示-3，表示的结果中总共将会有 32 个数字，分别是 0 或者 1。-3 是负数，最开始那一位为 1，-3 的绝对值 3 的二进制表示为 11。那么-3 的原码表示就是 10 0000000000 0000000000 0000000011。

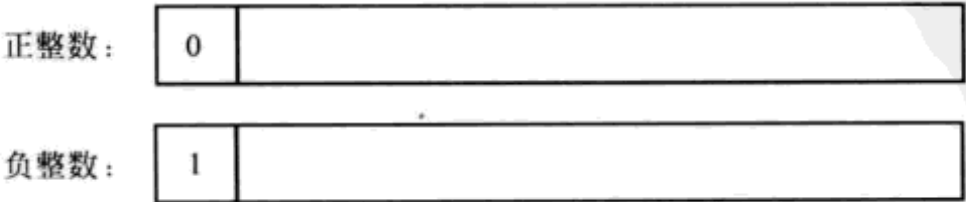


图 3.5 原码符号位

3. C语言整数的表示——补码表示

在C语言（或者说计算机）中，为了能把加减法都统一成一种运算，采用补码来表示一个整数。补码表示的规定如下。

(a) 对于正数，补码表示和原码表示是一样的。

(b) 对于负数，补码表示是绝对值的原码表示的每一位取反，然后给最终的结果再加1。

以-3为例来进行说明吧。

第一步：-3是一个负数，按照(b)规定来进行；

第二步：-3的绝对值为3，它按照32位原码表示形式如下：

$$|-3| = 00\ 0000000000\ 0000000000\ 0000000011$$

第三步：给|-3|的每一位取反，结果如下所示：

$$|-3|\text{取反} = 11\ 1111111111\ 1111111111\ 1111111100$$

第四步：给|-3|取反的结果再加1：

$$-3\text{的补码表示} = |-3|\text{取反} + 1 = 11\ 1111111111\ 1111111111\ 1111111101$$

通过四步的计算，我们最终得到了-3的补码表示。

4. 补码表示的好处——加减法的统一

以3加-3来看看计算机为何使用二进制的补码表示整数。我们都知道 $3 + (-3) = 0$ ，换成二进制是不是也是这个结果呢？看看吧，假设使用32位补码来分别表示3和-3。

$$3 = 00\ 0000000000\ 0000000000\ 0000000011$$

$$-3 = 11\ 1111111111\ 1111111111\ 1111111101$$

我们对每一位进行二进制的加法运算，逢二进一，最高位的进位不作为最终结果，最后得到的结果为0，和我们料想的一样。但是注意，我们是使用加法的规则来计算减法的，这就是补码的妙处所在。

5. 二进制和十进制的含义

在十进制和二进制之间进行转换的时候，只要记住一条就好：十进制是逢十进一的，二进制是逢二进一的。也就是说二进制从最低位开始，每一位上的数字代表的是 2^N ， N 代表的是所在的位数，最低位 N 为0。十进制每一位数字代表的是 10^N ， N 代表的是所在的位数，最低位 N 为0。

假设，有一个二进制数为 $X_N X_{N-1} X_{N-2} \cdots X_3 X_2 X_1 X_0$ ，那么 X_N 代表的就是 $X_N * 2^N$ ， X_{N-1} 代表的就是 $X_{N-1} * 2^{N-1}$ ，依此类推， X_0 代表的就是 $X_0 * 2^0$ 也就是 X_0 。在这里 X_N 、 X_{N-1} 、 \cdots 、 X_0 的取值只能是0或者1，图3.6展示了二进制每一位上的数对应的数值含义。

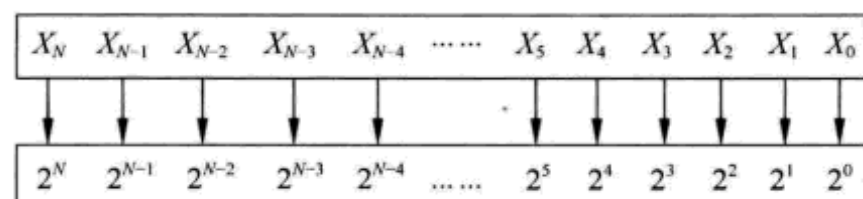


图 3.6 二进制位

与此类似, 给定一个十进制数 $X_N X_{N-1} X_{N-2} \cdots X_3 X_2 X_1 X_0$, X_N 代表的就是 $X_N * 10^N$, X_{N-1} 代表的就是 $X_{N-1} * 10^{N-1}$, 依此类推, X_0 代表的就是 $X_0 * 10^0$ 也就是 X_0 。在这里 X_N 、 X_{N-1} 、 \cdots 、 X_0 的取值只能是 $0 \sim 9$, 图 3.7 展示了十进制每一位上的数字含义。

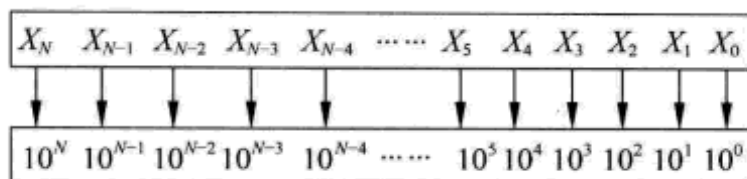


图 3.7 十进制位

6. 二进制转十进制

二进制转换成十进制的时候, 只要把二进制每一位上的数字与每一位代表的数值相乘, 最后把所有结果都加起来就可以了。以 01101011 为例来进行说明, 数学计算过程如下所示:

$$\begin{aligned}
 01101011 &= 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
 &= 0 \times 128 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\
 &= 64 + 32 + 8 + 2 + 1 \\
 &= 107
 \end{aligned}$$

图 3.8 更清楚地说明了二进制转十进制的计算过程。01101011 对应的式子表示为: $X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$ 。

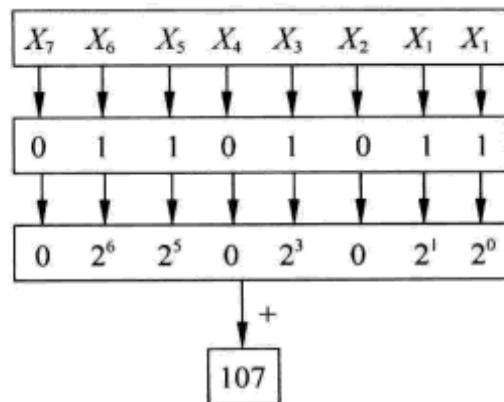


图 3.8 二进制转十进制

7. 十进制转二进制

十进制转二进制的时候, 是二进制转十进制的逆过程。相当于我们去解这样的一个方程:

$$X_N \times 2^N + X_{N-1} \times 2^{N-1} + X_{N-2} \times 2^{N-2} + \cdots + X_3 \times 2^3 + X_2 \times 2^2 + X_1 \times 2 + X_0 = Y$$

其中, $X_N, X_{N-1}, X_{N-2}, \cdots, X_3, X_2, X_1, X_0$ 每一个只能取值为 0 或者 1, Y 是十进制数, 求解 $X_N, X_{N-1}, X_{N-2}, \cdots, X_3, X_2, X_1, X_0$ 。

对于这样一个 n 元一次方程, 可以采用除法来求解, 步骤如下。

- (1) 对于 $2^N \leq Y < 2^{N+1}$, 先确定 N 的值。
- (2) n 取值依次从 N 到 0, 用 Y 除以 2^n , 商为 Z , 余数为 R , $X_n = Z$, $Y = R$; ($n = N, \cdots, 0$)。
- (3) 组合每一个 X , 就得到最终的二进制结果。

以上步骤看着很复杂, 其实操作起来很简单, 只有比较大小和减法两种数学计算。以十进制的 55 为例。

- (1) $2^5 < 55 < 2^6$, $N=5$;
- (2) 对于 $n=5$: Y 除以 2^5 , 商 Z 为 1, 余数 R 为 23, 则 $X_5 = Z = 1$, $Y = R = 23$;
 对于 $n=4$: Y 除以 2^4 , 商 Z 为 1, 余数 R 为 7, 则 $X_4 = Z = 1$, $Y = R = 7$;
 对于 $n=3$: Y 除以 2^3 , 商 Z 为 0, 余数 R 为 7, 则 $X_3 = Z = 0$, $Y = R = 7$;
 对于 $n=2$: Y 除以 2^2 , 商 Z 为 1, 余数 R 为 3, 则 $X_2 = Z = 1$, $Y = R = 3$;
 对于 $n=1$: Y 除以 2^1 , 商 Z 为 1, 余数 R 为 1, 则 $X_1 = Z = 1$, $Y = R = 1$;

对于 $n=0$: Y 除以 2^0 , 商 Z 为 1, 余数 R 为 7, 则 $X_0=Z=1, Y=R=0$

(3) 组合所有的余数, 也就是所有的 X , 就会得到 55 的二进制表示为 110111。

在图 3.9 中可以清楚地看到, 通过除法运算就可以很轻易地把十进制转换成二进制, 只要取每次除法的商组合起来就行。

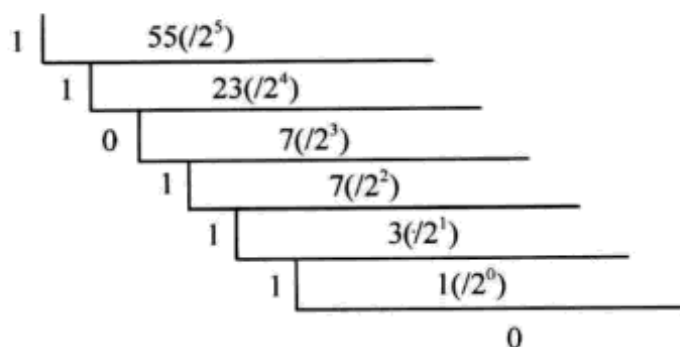


图 3.9 十进制转二进制

3.2.2 更先进的表示方法——八进制和十六进制

计算机的二进制存储, 是为了让计算机存储方便。在进行赋值或者其他运算的时候, 如果也是用二进制去表示一个数, 那肯定会累死的! 所以, 在写计算机程序的时候, 采用的是人容易识别的八进制、十进制、十六进制表示, 下面来看看这几种进制之间的微妙的关系。

1. 八进制的含义

与十进制类似, 对于一个八进制数 $X_N X_{N-1} X_{N-2} \cdots X_3 X_2 X_1 X_0$, X_N 代表的就是 $X_N \times 8^N$, X_{N-1} 代表的就是 $X_{N-1} \times 8^{N-1}$, 依此类推, X_0 代表的就是 $X_0 \times 8^0$ 也就是 X_0 。在这里 $X_N, X_{N-1}, \cdots, X_0$ 的取值只能是 0~7, 图 3.10 就表示了这个意思。

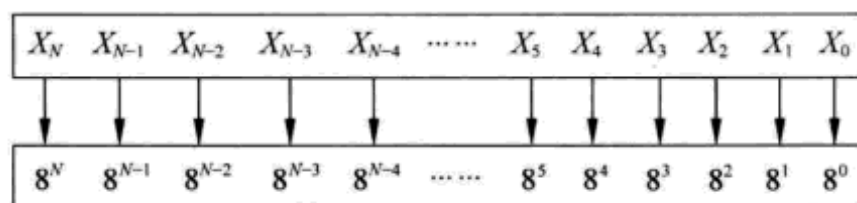


图 3.10 八进制的表示

2. 八进制转十进制

八进制转换成十进制和二进制转换成十进制基本是一样的, 也是把八进制每一位上的数字与每一位代表的数值相乘, 最后把所有结果都加起来就可以了。以 606066 为例来说明, 数学计算过程如下所示:

$$\begin{aligned}
 606066 &= 6 \times 8^5 + 0 \times 8^4 + 6 \times 8^3 + 0 \times 8^2 + 6 \times 8^1 + 6 \times 8^0 \\
 &= 6 \times 32768 + 0 \times 4096 + 6 \times 512 + 0 \times 64 + 6 \times 8 + 6 \times 1 \\
 &= 196608 + 3072 + 48 + 6 \\
 &= 199734
 \end{aligned}$$

大家可以对照着二进制转十进制的图示 (图 3.8), 在草稿纸上实现这个计算过程。

3. 十进制转八进制

十进制转八进制和十进制转二进制类似, 也是相当于解这样的一个方程:

$$X_N \times 8^N + X_{N-1} \times 8^{N-1} + X_{N-2} \times 8^{N-2} + \cdots + X_3 \times 8^3 + X_2 \times 8^2 + X_1 \times 8 + X_0 = Y$$

其中, $X_N, X_{N-1}, X_{N-2}, \cdots, X_3, X_2, X_1, X_0$ 每一个只能取值为 0~7, Y 是十进制数, 求

解 $X_N, X_{N-1}, X_{N-2}, \dots, X_3, X_2, X_1, X_0$ 。

解这个方程的具体步骤如下所示:

(1) 对于 $8^N \leq Y < 8^{N+1}$, 先确定 N 的值。

(2) n 的取值依次从 N 到 0, 用 Y 除以 8^n , 商为 Z , 余数为 R , $X_n = Z$, $Y = R$; ($n = N, \dots, 0$)。

(3) 组合每一个 X , 就得到最终的八进制结果。

用 199734 来演示, 首先确定 $N=5$, 按照这个规则的具体实现, 如图 3.11 所示。

然后组合所有的除数, 也就是方程中的 X , 就得到了 199734 的八进制表示 606066。

4. 十六进制的含义

与其他进制类似, 对于一个十六进制数 $X_N X_{N-1} X_{N-2} \dots X_3 X_2 X_1 X_0$, X_N 代表的就是 $X_N \times 16^N$, X_{N-1} 代表的就是 $X_{N-1} \times 16^{N-1}$, 依此类推, X_0 代表的就是 $X_0 \times 16^0$ 也就是 X_0 。十六进制中分别使用 A~F 表示 10~15。在这里 X_N, X_{N-1}, \dots, X_0 的取值只能是 0~F (即 0~9 和 A~F), 图 3.12 就表示了这个意思。

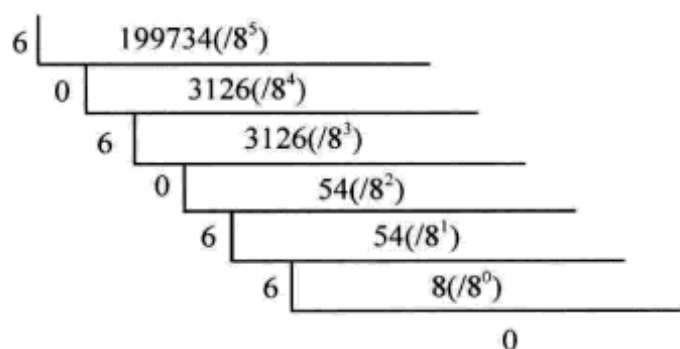


图 3.11 十进制转八进制

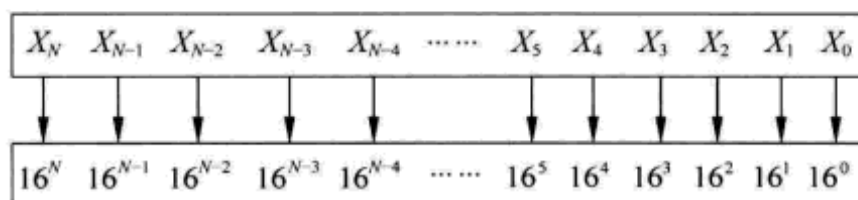


图 3.12 十六进制格式

5. 十六进制转十进制

十六进制转十进制和其他进制转十进制基本是一样的, 也是把十六进制每一位上的数字与每一位代表的数值相乘, 最后把所有结果都加起来就可以了。以 12F0EC 为例来说明。数学计算过程如下所示:

$$\begin{aligned}
 12F0EC &= 1 \times 16^5 + 2 \times 16^4 + 15 \times 16^3 + 0 \times 16^2 + 14 \times 16^1 + 12 \times 16^0 \\
 &= 1 \times 1048576 + 2 \times 65536 + 15 \times 4096 + 0 \times 256 + 14 \times 16 + 12 \times 1 \\
 &= 1048576 + 131072 + 61440 + 224 + 12 \\
 &= 1241324
 \end{aligned}$$

和其他进制转换成十进制的方法是一样的, 只是每一位上对应的数值不一样而已。

6. 十进制转十六进制

十进制转十六进制和十进制转二进制类似, 也是相当于解这样的方程:

$$X_N \times 16^N + X_{N-1} \times 16^{N-1} + X_{N-2} \times 16^{N-2} + \dots + X_3 \times 16^3 + X_2 \times 16^2 + X_1 \times 16 + X_0 = Y$$

其中, $X_N, X_{N-1}, X_{N-2}, \dots, X_3, X_2, X_1, X_0$ 每一个只能取值为 0~F, Y 是十六进制数, 求解 $X_N, X_{N-1}, X_{N-2}, \dots, X_3, X_2, X_1, X_0$ 。

解这个方程的具体步骤如下所示:

(1) 对于 $16^N \leq Y < 16^{N+1}$, 先确定 N 的值。

(2) n 的取值依次从 N 到 0, 用 Y 除以 16^n , 商为 Z , 余数为 R , $X_n = Z$, $Y = R$; ($n = N, \dots, 0$)。

(3) 组合每一个 X , 就得到最终的十六进制结果。

我们用 1241324 来演示, 首先确定 $N=5$, 按照这个规则的具体实现, 如图 3.13 所示。组合所有的商, 也就是公式中的 X , 就得到了 1241324 的十六进制表示为 12F0EC。

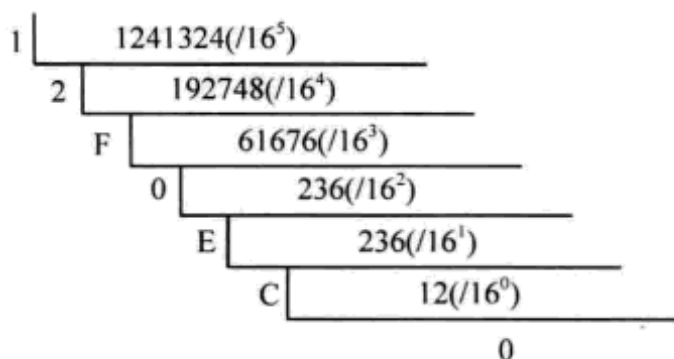


图 3.13 十进制转十六进制

3.2.3 进制之间的转换——以二进制为桥梁

在上面的几个小节里, 可以看到通过除法可以很好地把十进制转换到其他进制。但是, 总用除法还是会觉得有点麻烦, 有没有更简单的方法来实现各种进制之间的转换呢? 有, 那就是把要转换的进制先转换成二进制, 然后再从二进制转换到其他的进制, 以二进制为桥梁。尤其对于八进制、十六进制这种 2 的倍数的进制会显得尤为简单!

1. 八进制和二进制之间的转换

先来看看二进制转换成八进制吧, 很简单, 只要从右到左, 每三位一个分隔组合, 就可以得到相应二进制数的八进制表示。例如, 以 33 (十进制) = 100001 为例, 从右到左, 每三位一个分隔, 然后组合就可以得到 100001 的八进制表示, 如图 3.14 所示。

二进制转八进制简单吧! 看完了二进制转八进制, 有人也许会问, 那么八进制转二进制该怎么办呢? 其实, 也很简单, 只要把二进制转八进制的方法倒过来就可以了。对于八进制数的每一位, 变换成三位的二进制, 然后再组合起来就可以了。因为八进制每一位上的数, 只能是 0~7, 转换成二进制是很简单的。我们再来看看, 八进制的 41 是如何简单地转换成二进制的, 如图 3.15 所示。

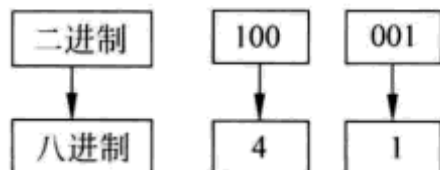


图 3.14 二进制转八进制

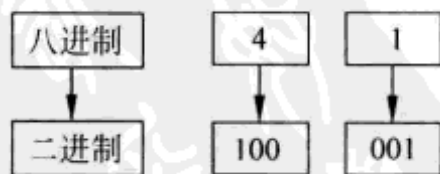


图 3.15 八进制转二进制

2. 十六进制和二进制之间的转换

和八进制类似, 十六进制和二进制之间的转换也是很简单的, 在二进制转换成十六进制的时候, 每 4 位进行组合, 十六进制转换成二进制时, 每个十六进制位上的数对应 4 位二进制数。还是以 33 为例, 33 的二进制表示为 100001, 它和十六进制的转换关系, 如图 3.16 所示。二进制数 100001 的十六进制表示为 21。

3. 以二进制为桥梁实现各种进制之间的转换

在计算机中,最常使用的是二进制、八进制、十进制和十六进制。从本节可以看出二进制、八进制和十六进制之间的转换是很简单的,十进制和二进制的转换也不是很麻烦,所以可以

通过二进制这个桥梁来实现各种进制之间的转换。所以,计算机很“喜欢”二进制,它可以方便地提供我们喜欢的八进制、十进制和十六进制之间的转换。

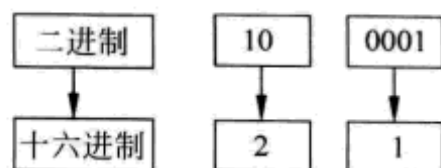


图 3.16 二进制转十六进制

3.2.4 给整型赋值

看完了 C 语言中整型数据的各种进制的表示,就可以使用各种进制的整型数值表示来给整型变量赋值了。

1. 整型赋值

根据前面讲到的给变量赋值的形式,可以给出整型变量赋值的一般形式:

整型变量名 = 整型数值

其中,整型变量名就是声明定义的变量名称,整型数值既可以是一个常量,也可以是一个已经有暂存值的变量。如果有一个名为 `radius` 的整型变量,想给它赋值为 3,就可以用这样的表示:

```
radius = 3;
```

按照 C 语言的要求,变量要先进行声明定义,然后才可以赋值。即按照以下的步骤进行:

- (1) 声明定义一个变量。
- (2) 给声明的变量赋值。

例如,先声明定义一个整型变量,并取名为 `radius`,用来表示半径,C 语言代码如下所示:

```
int radius;
```

然后使用等号赋值运算符,按照上面所讲的格式,赋值为 3,其 C 语言代码为:

```
radius = 3
```

有的时候,为了简单,也可以用如下初始化的形式来一次性完成声明定义和赋值,这样的表示被称为变量初始化,其格式如下所示。

整数类型名 整数变量名 = 整型数值

如果定义一个 `radius` 的整型变量,然后将它初始化为 3,就可以使用下面的 C 语言表示:

```
int radius =3;
```

进行整数赋值的时候,整型数值可以是任何进制的整数表示,一般不用二进制,表示一个数要写好长一串 0 或者 1。十进制、八进制和十六进制是在整型赋值的时候最常使用

的整数表示形式，接下来将一一介绍这几种进制的赋值。

2. 按照十进制赋值

按照十进制给整型变量赋值是在编写代码的时候最常使用的方式，因为十进制是我们最熟悉的，用起来也方便。在 C 语言中，按照十进制进行赋值的时候，按照前一小节所讲的赋值运算的格式来就可以了。声明一个整型变量，然后再用等号运算符给这个整型变量赋一个十进制表示的数。

```
#include<stdio.h>

int main()
{
    int radius = 33;
    printf("%d", radius);
    return 0;
}
```

上面这段代码初始化了一个 radius 整型变量，给其赋值为 33，然后使用 printf() 函数进行输出。图 3.17 是程序按十进制输出的结果。

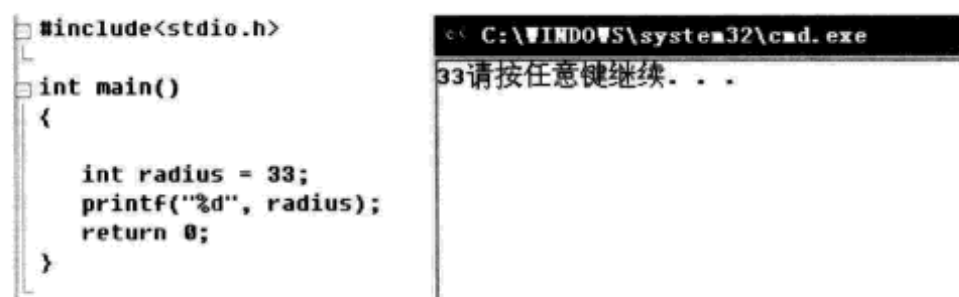


图 3.17 十进制赋值

3. 按照八进制赋值

C 语言规定，八进制表示一个整型常量的时候必须以 0 开始。因为从来没有人以十进制表示的时候在前面多写个 0，所以 C 语言用 0 开始作为八进制的标志，以与十进制进行区分。并且八进制数每个数位上的数只能是 0~7。

可以使用之前介绍的 printf() 函数来验证一下 33 的八进制表示是不是正确的。要把一个数按照八进制表示的结果输出，printf() 函数中的“%变量类型表示”使用的字母是 o，也就是 %o，具体代码如下所示。

```
#include<stdio.h>
int main()
{
    int radius1;
    int radius2;
    radius1 = 33;
    radius2 = 041;

    printf("%o\n", radius1);
    printf("%d\n", radius2);
    return 0;
}
```

在代码中，先定义了两个整型变量 radius1 和 radius2，然后按照十进制表示给 radius1

赋值为 33，按照八进制表示给 radius2 赋值为 41。对于 radius1 按照八进制输出，计算机会自动进行转换，料想结果应该是 41。对于 radius2 按照十进制输出，计算机自动转换以后输出应该是 33，结果如图 3.18 所示。对于代码中出现的“\n”，先不用过多关心，这里只要知道它是为了输出好看用的就可以了。

4. 按照十六进制赋值

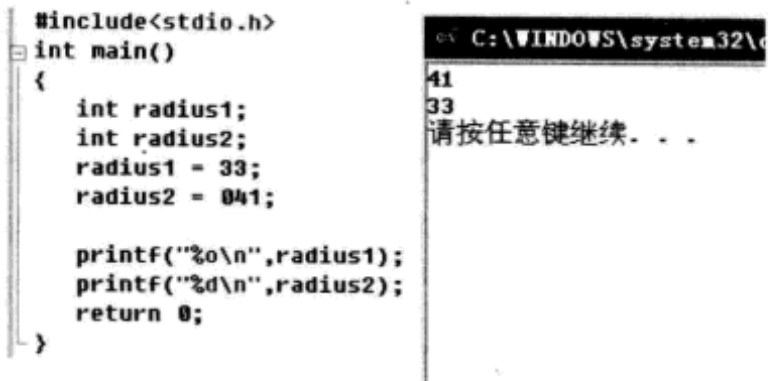
C 语言规定，十六进制表示一个整型常量的时候必须以 0x 或者 0X 开始，以与其他进制进行区分。并且每个数位上的数字必须是 0~F（0~9、A、B、C、D、E、F）或者 0~f（0~9、a、b、c、d、e、f）。

下面用和八进制表示那段代码类似的代码验证一下十六进制表示。printf() 函数中的“%变量类型表示”使用的字母是 x 或者 X，也就是 %x 或 %X，具体代码如下所示。

```
#include<stdio.h>
int main()
{
    int radius1;
    int radius2;
    int radius3;
    radius1 = 33;
    radius2 = 0x21;
    radius3 = 0X21;

    printf("%x\n",radius1);
    printf("%X\n",radius1);
    printf("%d\n",radius2);
    printf("%d\n",radius3);
    return 0;
}
```

在代码中，定义了三个整型变量，分别是 radius1、radius2 和 radius3，然后对于 radius1 按照十进制赋值为 33，对于 radius2 和 radius3 按照十六进制的两种赋值方式（0x 和 0X）分别赋值为 21。对 radius1 按照十六进制的两种表示分别进行输出，料想结果应该为 21。对于 radius2 和 radius3 按照十进制分别输出，期望得到的结果是 33，如图 3.19 所示。

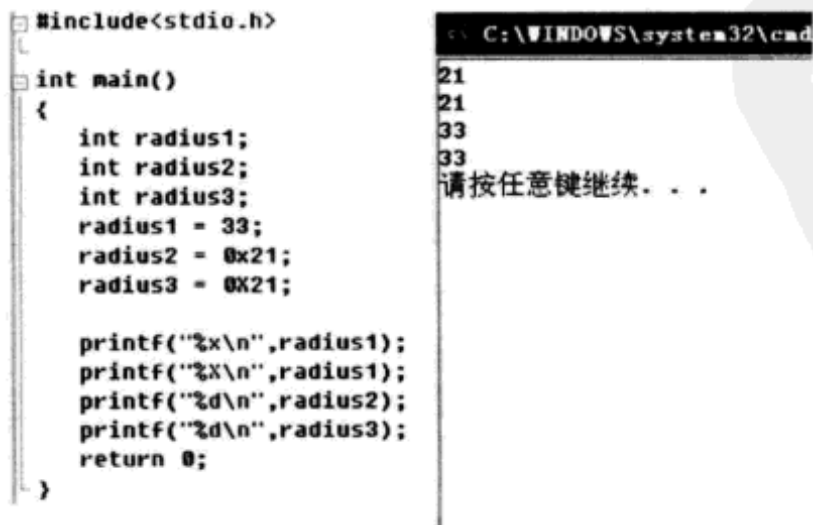


```
#include<stdio.h>
int main()
{
    int radius1;
    int radius2;
    radius1 = 33;
    radius2 = 041;

    printf("%o\n",radius1);
    printf("%d\n",radius2);
    return 0;
}
```

C:\WINDOWS\system32\cmd
41
33
请按任意键继续...

图 3.18 八进制输出



```
#include<stdio.h>
int main()
{
    int radius1;
    int radius2;
    int radius3;
    radius1 = 33;
    radius2 = 0x21;
    radius3 = 0X21;

    printf("%x\n",radius1);
    printf("%X\n",radius1);
    printf("%d\n",radius2);
    printf("%d\n",radius3);
    return 0;
}
```

C:\WINDOWS\system32\cmd
21
21
33
33
请按任意键继续...

图 3.19 十六进制输出

3.3 浮点型赋值

接下来看看稍微有点复杂的浮点型的赋值。与整型赋值相比，浮点型赋值确实有点不同或者有点复杂，但相信读者一定可以学会。

3.3.1 小数在计算机中的表示

要想真正了解 C 语言中的浮点型数值的赋值，就得从最本质上来了解计算机中的浮点型数值到底是什么样的？也就是在二进制的世界里，如何表示浮点型数据。

1. 小数的二进制表示

前面学习了整数的二进制表示。现在看看常见的十进制的小数，对应的二进制表示是什么样的。图 3.20 显示了十进制小数和二进制小数每一位对应的数值，图的上半部分是十进制小数每位对应的数值，下半部分是二进制小数每一位对应的数值。

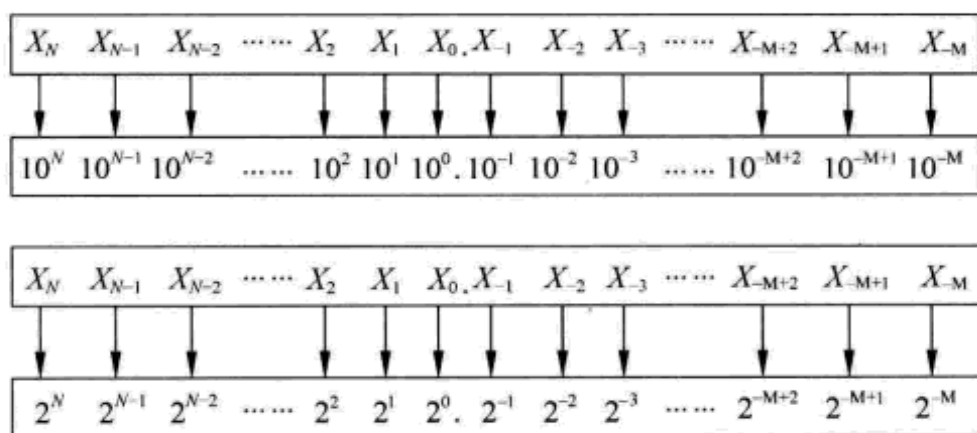


图 3.20 十进制和二进制小数的表示

从图 3.20 中可以看出，小数的十进制和二进制每一位上的数值的含义和整数的每一位上的含义是一样的，只不过小数点后边的数对应的数值是负指数。

要想把十进制小数转换成二进制小数，可以使用类似于十进制整数转换成二进制整数的方法。相当于去解这样一个方程：

$$X_N \times 2^N + X_{N-1} \times 2^{N-1} + X_{N-2} \times 2^{N-2} + \dots + X_2 \times 2^2 + X_1 \times 2 + X_0 + X_{-1} \times 2^{-1} + X_{-2} \times 2^{-2} + \dots + X_{-M+1} \times 2^{-M+1} + X_{-M} \times 2^{-M} = Y$$

其中， $X_N, X_{N-1}, X_{N-2}, \dots, X_3, X_2, X_1, X_0, \dots, X_{-M+1}, X_{-M}$ 每一个只能取值为 0 或者 1， Y 是十进制数，求解 $X_N, X_{N-1}, X_{N-2}, \dots, X_3, X_2, X_1, X_0, \dots, X_{-M+1}, X_{-M}$ 。

对于这样一个 n 元一次方程，可以采用除法来求解，步骤如下。

(1) 对于 $2^N \leq Y < 2^{N+1}$ ，先确定 N 的值，再确定 M 的值。 M 表示的是精确范围， M 越大越精确。

(2) 依次从 N 到 $-M$ ，用 Y 除以 2^n ，商为 Z ，余数为 R ， $X_n = Z$ ， $Y = R$ ；($n = N, \dots, 0, -1, -2, \dots, -M$)。

(3) 组合每一个 X ，就得到最终的二进制结果。

以上步骤看着很复杂，其实操作起来很简单，只有比较大小和减法两种数学计算。以

十进制的 0.12 为例。

(1) $2^{-4} < 0.12 < 2^{-3}$, $N=-4$; 假设要精确到 $M=10$ 。

(2) 对于 $n=-4$: Y 除以 2^{-4} , 商 Z 为 1, 余数 R 为 0.0575, 则 $X_5=Z=1$, $Y=R=0.0575$;

对于 $n=-5$: Y 除以 2^{-5} , 商 Z 为 1, 余数 R 为 0.02625, 则 $X_4=Z=1$, $Y=R=0.02625$;

对于 $n=-6$: Y 除以 2^{-6} , 商 Z 为 1, 余数 R 为 0.010625, 则 $X_3=Z=1$, $Y=R=0.010625$;

对于 $n=-7$: Y 除以 2^{-7} , 商 Z 为 1, 余数 R 为 0.0028125, 则 $X_2=Z=1$, $Y=R=0.0028125$;

对于 $n=-8$: Y 除以 2^{-8} , 商 Z 为 0, 余数 R 为 0.0028125, 则 $X_1=Z=1$, $Y=R=0.0028125$;

对于 $n=-9$: Y 除以 2^{-9} , 商 Z 为 1, 余数 R 为 0.000859375, 则 $X_0=Z=1$, $Y=R=0.000859375$;

对于 $n=-10$: Y 除以 2^{-10} , 商 Z 为 0, 余数 R 为 0.000859375, 则 $X_0=Z=1$, $Y=R=0.000859375$ 。

(3) 组合所有的商数, 也就是所有的 X , 就会得到 0.12 的二进制表示为 0.0001111010, 最后的余数 $R=0.000859375$ 。这也就是 M 取 10 时的精度, 或者说是误差, 如果需要更大的精度, M 就取更大的值。

在图 3.21 中可以清楚地看到, 通过除法运算就可以很轻易地把十进制小数转换成二进制小数, 只要取每次除法的商组合起来就行。

2. 浮点型的数学表示

把一个小数赋值为一个浮点类型的变量时, 计算机内部就会按照一定的规则把这个小数表示出来。计算机是按照什么样的一种规则去表示这个小数的呢? 那就先分析一下小数吧! 任何一个小数都可以写成以下这种格式:

$$N = M \times B^E \quad (\text{公式 3.1})$$

其中, N 是要表示的数, M 被称为尾数, B 被称为基数, E 被称为指数。以 0.0003 为例来说明一下, $0.0003=3 \times 10^{-4}$, 对应到公式 3.1, 也就是 $N=0.0003$, 尾数 $M=3$, 基数 $B=10$, 指数 $E=-4$ 。

3. 浮点型的计算机表示

在计算机中表示浮点数的时候, 基数 B 不采用 10, 而是采用 2。因此, 公式 3.1 就成为以下格式:

$$N = M \times B^E \quad (\text{公式 3.2})$$

从公式 3.2 可以看出: 只要能表示出 M 和 E 这两个数值, 就可以表示出任何一个具体的小数了。那么计算机到底是如何表示 M 和 E 这两个数值的呢?

还是先看一个例子吧。例如, $0.12=0.24 \times 2^{-1}=0.48 \times 2^{-2}=0.96 \times 2^{-3}=\dots$ 。可以看出, 当把一个小数变换成公式 3.2 的形式的时候, E 是 2 的指数, 它是一个整数。可以通过前面整数的计算机表示来类似地表示 E 这个数。从 0.12 的例子中, 还可以看出尾数 M 往往都是小数, 而且根据 E 的变化而变化。那该怎么表示这个小数 M 呢?

可以想办法把这个小数以整数来表示, 就可以使用前面学到的知识了。在计算机中采用的方式是固定小数 M 的小数点, 小数点固定了, 就可以把这个小数当成整数来对待了。

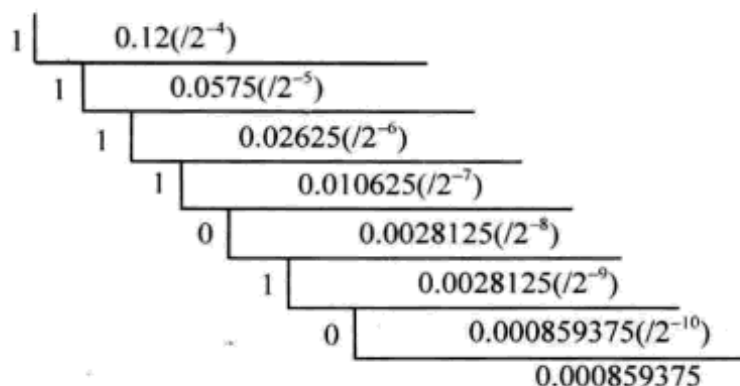


图 3.21 十进制小数转二进制小数

计算机中使用的是二进制表示, M 的小数点被固定在第一个有效位二进制位的前面, 如图 3.22 所示。 X_0 表示的是尾数的符号, 那么, 尾数的第一个有效二进制位就是 X_{-1} , 小数点就位于 X_0 和 X_{-1} 这两个二进制之间。小数的这种表示方式被称为规格化表示。

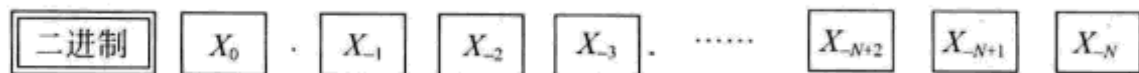


图 3.22 尾数的计算机表示格式

还是以 0.12 为例来说明一下。使用图 3.21 所示的方法, 并取 $M=23$, 可以得到 0.12 的二进制小数表示为 0.00011 11010 11100 00101 00。按照规格化的要求, 小数点必须位于第一有效位的前面, 也就是第一个 1 的前面, 因此, 0.12 的规格化表示中, 尾数 $M=0.11110 10111 00001 0100$, 最前面的 0 即 X_0 , 表示符号为正, 接下来的 1 即 X_{-1} 。

尾数确定了, 指数 E 也就确定了, 0.1111 01011 10000 10100 约等于把 0.12=0.00011 11010 11100 00101 00 的小数点向右移动了 3 位, 即 $0.12 = N = 0.1111 01011 10000 10100 \times 2^{-3}$, 所以指数 $E=-3$ 。

4. 小数表示的更简单方法

当按照规格化表示小数的时候, 就可以把尾数 M 当作整数来看了。使用类似于整数的补码表示, 若尾数大于零, 为正数, 则规格化数应该是 01XXXX 的形式。若尾数小于 0, 为小数, 按照补码表示, 会对绝对值的有效位进行取反加一 (求补运算), 则规格化表示应为 10XXXX 的形式。其中, 最左面的 0 表示尾数为正, 最左面的 1 表示尾数为负。

通过上面的介绍可以知道, 小数点永远位于最高一位有效二进制位的前面, 二进制的有效位永远是 1。所以, 尾数的绝对值永远是形如 0.1XXX 的样子, 转换成十进制, 即 M 的绝对值的范围永远为 $1/2 \leq M < 1$ 。

有了上面的观察, 就不必使用上面讲解的那么麻烦的方法来把一个小数表示为浮点型, 一个更简单的方法如下所示。

(1) 确定尾数 M 的值, 使其 $1/2 \leq M < 1$ 。

(2) 把 M 表示成形如 01XXXX 或者 10XXXX 的形式。前者是 M 为正的情况, 后者是 M 为负的情况。

(3) 根据已确定的尾数 M , 确定指数 E , 使要表示的小数 $N = M \times 2^E$ 。

(4) 按照补码表示表示 E 。

现在把 0.12 表示为计算机可识别的浮点类型的过程按照上面讲的步骤进行归纳。

(1) 确定 M 的范围, $0.12 = 0.96 \times 2^{-3}$, $1/2 \leq 0.96 < 1$, $M=0.96$ 。

(2) 把 $M=0.96$, 表示成计算机二进制形式。按照十进制小数转换成二进制小数的方法, 0.96 的二进制表示为 0.1111 01011 10000 10100。其中, 最前面的 0 表示的是正数。

(3) $0.12 = 0.96 \times 2^{-3}$, 可以确定 $E=-3$ 。

(4) E 的补码表示为 00000011, 假设使用 8 个二进制位来表示指数 E 。

5. C语言浮点型表示

在 C 语言中, 小数是按照浮点类型来表示的。浮点型分为 float 浮点型和 double float

浮点型。两种浮点型在计算机中的表示都是按照尾数 M 和指数 E 来表示的，只是对于不同的浮点型， M 和 E 所占的二进制位不同而已。

(1) 对于 float 浮点类型来说，使用了 32 个二进制位来表示，按照 0 到 31 来标号，如图 3.23 所示。其中，符号位相当于图 3.20 中的 X_0 ，尾数相当于 $X_{-1}X_{-2}X_{-3}\cdots X_{-N+2}X_{-N+1}X_{-N}$ 。

31	30-23	22-0
符号	指数	尾数

图 3.23 float 浮点型表示分段

- 符号位：第 31 位，0 表示正数，1 表示负数；
- 指数位：第 30~23 位，用来表示指数，它的取值范围为-128~127；
- 尾数位：第 22~0 位，用来表示尾数，就是上面所说的规格化数。

随便取一个数，就 2.42 吧！它的符号为正，那么符号位就是 0。然后把 2.42 变换成规格化小数，也就是把 2.42 除以 2 的整数次方，得到一个数 M ，使 $1/2 \leq M < 1$ 。通过观察可以看出， $2.42/2^2 = 0.605$ ，也就是说 $0.605 \times 2^2 = 2.42$ 。根据公式 3.2，尾数 $M=0.605$ ，指数 $E=2$ 。所以，2.42 的实际 float 型的 32 位存储如图 3.24 所示。

31	30-23	22-0
0	2	.605

图 3.24 2.42 的 float 浮点表示

(2) 对于 double float，计算机是使用 64 个二进制位来表示的，按照 0 到 63 来标号，如图 3.25 所示。它和 float 一样，符号位相当于图 3.20 中的 X_0 ，尾数相当于 $X_{-1}X_{-2}X_{-3}\cdots X_{-N+2}X_{-N+1}X_{-N}$ 。

63	62-52	51-0
符号	指数	尾数

图 3.25 double 浮点型表示分段

- 符号位：第 63 位，0 表示正数，1 表示负数；
- 指数位：第 62~52 位，用来表示指数，它的取值范围为-1024~1023；
- 尾数位：第 51~0 位，用来表示尾数，是上面所说的规格化数。

2.42 的 double 表示和 float 表示是基本一样的，尾数 $M=0.605$ ，指数 $E=2$ ，只是 M 和 E 占用的二进制位不一样而已。2.42 的实际 64 位存储，如图 3.26 所示。

63	62-52	51-0
0	2	.605

图 3.26 2.42 的实际 64 位存储

3.3.2 给浮点型赋值

深入了解了浮点型数据的计算机表示以后，只要掌握了上面的知识，浮点数就真的都成为浮云了。接下来看看 C 语言中是如何把一堆 0 和 1 表示的小数值赋值给一个浮点型变量的。

1. 浮点型赋值

根据前面讲到的给变量赋值的形式，可以给出浮点型变量赋值的一般形式：

浮点型变量名 = 浮点型数值

其中，浮点型变量名就是声明定义的变量名称，浮点型数值既可以是一个常量，也可以是一个已经有暂存值的变量。如果有一个名为 `radius` 的浮点型变量，想给它赋值为 0.45，就可以用这样的表示形式：

```
radius = 0.45;
```

按照 C 语言的要求，变量要先进行声明定义，然后才可以赋值，即按照以下的步骤进行：

- (1) 声明定义一个变量。
- (2) 给声明的变量赋值。

例如，先声明定义一个浮点型变量，并取名为 `radius`，用来表示小数半径。C 语言代码如下所示：

```
float radius;
```

然后使用等号赋值运算符，按照上面所讲的格式赋值为 0.45。其 C 语言代码为：

```
radius = 0.45;
```

有的时候，为了简单，也可以用如下的初始化形式来一次性完成声明定义和赋值，这样的表示形式被称为变量初始化。

浮点类型名 浮点型变量名 = 浮点型数值

例如，定义一个名为 `radius` 的浮点型变量，然后将它初始化为 0.45。那么，就可以使用下面的 C 语言表示形式了：

```
float radius = 0.45;
```

进行浮点型赋值的时候，既可以给 `float` 浮点型变量赋小数，也可以给 `double` 浮点型变量赋小数。

2. float浮点型的赋值

对 `float` 浮点型变量进行赋值和对整型赋值是一样的。先声明定义一个 `float` 变量，然后使用上一小节的等号赋值运算符格式给其赋值一个小数就可以了，或者也可以使用 `scanf()` 函数为其赋值。在使用 `scanf()` 函数对 `float` 变量进行赋值的时候，使用的数据类型表示字母为 `f`，也就是“`%f`”。

举一个具体的例子。先声明一个 float 型半径变量 radius，然后使用 scanf() 函数给其赋值为 0.5，最后使用 printf() 函数来验证赋值是不是正确的。具体代码如下所示：

```
#include<stdio.h>
int main()
{
    float radius;
    scanf("%f",&radius);
    printf("%f", radius);
    return 0;
}
```

printf() 函数按照 float 浮点型进行输出的时候，使用的数据类型表示字母为 f，也就是“%f”，程序的输出如图 3.27 所示。第一行输入的是 0.5，它通过 scanf() 函数赋值给 radius 浮点型变量。第二行是 printf() 函数，输出的结果是 0.500000，后面多了几个 0，这是为了表示一定的精度而输出的。

3. double 浮点型的赋值

为 double 浮点型变量赋值和为 float 浮点型变量赋值是一样的。先声明定义，然后使用等号赋值运算或者 scanf() 函数对其进行赋值。使用 scanf() 函数为 double

```
#include<stdio.h>
int main()
{
    float radius;
    scanf("%f",&radius);
    printf("%f", radius);
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
0.5
0.500000请按任意键继续...
```

图 3.27 float 赋值

浮点型变量赋值的时候，使用的数据类型表示字母为 lf，也就是“%lf”。

接下来的例子就是先声明一个 double 浮点型的半径变量，然后使用 scanf() 给它赋值为 0.5，最后使用 printf() 函数予以验证，代码如下所示：

```
#include<stdio.h>
int main()
{
    double radius;
    scanf("%lf",&radius);
    printf("%lf", radius);
    return 0;
}
```

printf() 函数按照 double 浮点型进行输出的时候，使用的数据类型表示字母为 lf，也就是“%lf”。程序的输出如图 3.28 所示。第一行是输入的 0.5，它通过 scanf() 函数赋值给 radius 浮点型变量，第二行是 printf() 函数，输出的结果是 0.500000。

```
#include<stdio.h>
int main()
{
    double radius;
    scanf("%lf",&radius);
    printf("%lf", radius);
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
0.5
0.500000请按任意键继续...
```

图 3.28 double 赋值

3.4 字符型赋值

字符型的计算机存储和整型、浮点型相比，又是另一种模式。整型和浮点型都是先有数字的含义，之后以一定规则的二进制（编码）进行表示，而字符型是先有二进制表示，然后再赋予这些二进制表示一定的意义。

3.4.1 字符在计算机中的表示——ASCII

计算机中字符最常用的编码就是 ASCII 编码，适用于所有拉丁文字字母的编码。ASCII 的意思是美国标准信息交换码（American Standard Code for Information Interchange），它已被国际标准化组织（ISO）定为国际标准，称为 ISO 646 标准。本书附录中给出了一张 ASCII 表，其中包含所用的 ASCII 编码，以及其所代表的字符，本节就让我们具体看看所谓的 ASCII 编码吧！

1. 字符在计算机中的表示

与其他两种数据类型相比，字符型的计算机表示简单很多。在计算机中，把常用的字符整理出 256 个。然后，给这 256 个字符依次编号为 0~255。这样，计算机通过这些编号直接表示这些字符。这类似于每个学生都有一个学号，一个学号就可以代表一个具体的学生。计算机中是使用八位的二进制来给字符编号的，而 255 刚好可以表示为最大的八位二进制数 11111111。

例如，以 ASCII 表中的字母 a 为例，a 对应的十进制数表示为 97，97 使用八位表示的二进制数为 01100001，如图 3.29 所示。

0	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---

图 3.29 字符 a 的编码表示

2. 字符的 C 语言表示

在计算机中是使用八位的二进制数来表示一个字符的，那是为了计算机方便。我们进行编程的时候，为了方便，使用一些常见的并且易于表示的符号来表示 C 语言中的字符。在 C 语言中对于表示字符的符号有以下要求。

- 字符得用单引号包含起来。
- 字符只能是一个字母、数字、标点符号或者其他，而不能是像两个字母、数字等这样的符号。

例如，‘a’、‘B’、‘1’、‘0’、‘+’等都是合法的 C 语言字符表示。像 a、B、1、0、+等都不是合法的字符表示，“a”、“B”、“1”、“0”、“+”等也不是合法的字符表示，‘aa’、‘BB’、‘11’、‘00’等也不是合法的字符表示。

3. 转义字符

在 C 语言中，有一些字符是不能简单地通过类似 a、b、c、+、-、*这样的符号来表示

的，如回车换行、退格，而这些都是在进行显示的时候必需的字符。C 语言是通过转义字符来表示这些特殊的字符的。表 3.2 列出了 C 语言中支持的一些主要的转义字符，以及相应的含义和八位二进制表示对应的十进制表示。

表 3.2 C语言常用的转义字符

转义字符	含 义	十进制表示
\a	响铃 (DEL)	7
\b	退格 (BS)	8
\f	换页 (FF)	12
\n	换行 (FF)	10
\r	回车 (CR)	13
\t	水平制表 (HT)	9
\v	垂直制表 (VT)	11
\“	双引号	92
\\	问号字符	63
\’	单引号	39
\"	双引号	34
\0	空字符	0
\ddd	三位八进制数对应的字符	三位八进制数
\xhh	两位十六进制数对应的字符	两位十六进制数

从表 3.2 中可以看出，所有的转义字符都是以 ‘\’（反斜杠）开始的。另外，在 C 语言中转义字符主要有两类表示方法：一类是用特定符号表示，另一类是用 ASCII 码数值表示。

对于第一类表示，对应到表 3.2 中，就是除最后两个以外的表示方式。例如，

- ❑ ‘\a’ 表示的是响铃，也就是这个字符会使计算机发出响铃的声音；
- ❑ ‘\n’ 表示的是换行，这个字符可以使输出的时候从新的行开始显示字符；
- ❑ ‘\’ 表示的是单引号，因为单引号用来在表示字符的时候括住字符，所以要想表示单引号，就得使用转义字符来表示。

诸如此类，转义字符都是用来表示一定特殊的含义的，对于表中的其他字符，遇到的时候再做详细的讲解，此处就不再赘述了。

对于第二类表示，主要就是表 3.2 中最后两个：‘\ddd’ 和 ‘\xhh’。

- ❑ ‘\ddd’ 代表的是使用三位的八进制数 ddd 来表示一个字符，这个八进制数就是每个字符的八位二进制表示对应的八进制表示。例如，‘\141’ 代表的就是字符 ‘a’，因为 141 的二进制表示就是 01100001，就是字符 ‘a’ 的表示。
- ❑ ‘\xhh’ 代表的是使用两位十六进制数 hh 来表示一个字符。例如，‘\x61’ 代表的就是字符 ‘a’，因为十六进制的 61 对应二进制表示也是 (01100001)。

3.4.2 给字符赋值

有了字符的二进制表示和字符的 C 语言表示的知识以后，对字符的理解就算是上了一个台阶了。接下来，看一看 C 语言中如何给一个字符变量赋值。

1. 字符型赋值

根据前面讲到的给变量赋值的形式，可以给出字符型变量赋值的一般形式：

字符型变量名 = 字符表示

其中，字符变量名就是进行变量声明定义的时候给变量取的名字，字符表示是上面介绍的单引号括住的字符表示，也可以是一个已经赋了值的字符型变量。既可以是类似于‘a’的符号表示的字符，又可以是转义字符表示的字符。举个例子，有一个名为 character 的字符型变量，想给它赋值为 ‘a’，表示如下：

```
character = 'a';
```

按照 C 语言的要求，变量可以按照以下的步骤进行：

- (1) 声明定义一个变量。
- (2) 给声明的变量赋值。

例如，先声明定义一个字符型变量，并取名为 character，用来保存一个字符，C 语言代码如下所示：

```
char character;
```

然后使用等号赋值运算符，按照上面的所讲的格式赋值为 ‘a’，其 C 语言代码为：

```
character = 'a';
```

有的时候，为了简单，也可以用如下初始化的形式来一次性完成声明定义和赋值，这样的表示被称为变量初始化。

字符类型名 字符型变量名 = 字符表示

若定义一个 character 的字符型变量，然后将它初始化为 ‘a’，那么 C 语言的表示形式如下：

```
char character = 'a';
```

进行字符型赋值的时候，既可以使用类似于 ‘a’、‘b’ 这样一般的字符表示来给字符变量赋值，也可以使用转义字符来给字符变量赋值。

2. 使用一般字符表示给字符变量赋值

一般字符表示就是类似于 ‘a’ 的使用单引号括住简单符号的表示方法，也就是不使用 ‘\’ 的非转义字符表示。举个 C 语言中使用一般字符表示给字符变量赋值的例子。在这个例子中：

- (1) 声明一个字符型的变量 name;
- (2) 使用 scanf() 函数给其赋值为 ‘a’;
- (3) 用 printf() 函数来验证赋值是不是正确的;
- (4) 使用等号赋值运算给变量 name 赋值为 ‘b’;
- (5) 使用 printf() 函数进行验证。

具体代码如下所示：

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



```

{
    char name;
    scanf("%c",&name);
    printf("%c",name);

    name = 'b';
    printf("%c",name);
    return 0;
}

```

printf()函数按照字符型进行输出和 scanf()函数按照字符型进行输入的时候,使用的数据类型表示字母为 c,也就是“%c”,程序的输出如图 3.30 所示。第一行是输入的字符‘a’,它通过 scanf()函数赋值给 name 字符型变量。第二行是 printf()函数输出的结果‘a’和‘b’,‘a’是 scanf()函数赋值的结果,‘b’是等号赋值运算赋值的结果。

```

#include<stdio.h>

int main()
{
    char name;
    scanf("%c",&name);
    printf("%c",name);

    name = 'b';
    printf("%c",name);
    return 0;
}

```

C:\WINDOWS\system32\cmd.exe
a
ab请按任意键继续. . .

图 3.30 一般字符型赋值

3. 使用转义字符给字符变量赋值

使用转义字符给字符变量赋值,就是使用‘\’表示的转义字符给字符变量赋值。这种赋值可以有两类,一类就是使用转义字符的第一类表示方法(特定符号表示)给字符变量赋值,另一类就是使用转义字符的第二类表示方法(ASCII 码数值表示)给字符变量赋值。

按照字符型变量赋值的形式,先声明定义一个字符型变量 line_end,然后按照表 3.2 所示,使用不同的转义字符形式给其赋值为换行转义字符,并对每一种赋值形式分别使用 printf()函数进行验证。具体的代码如下所示:

```

#include<stdio.h>

int main()
{
    char line_end;
    line_end = '\n';
    printf("%c",line_end);

    line_end = '\012';
    printf("%c",line_end);

    line_end = '\x0a';
    printf("%c",line_end);
    return 0;
}

```

程序的输出如图 3.31 所示,按照不同的转义字符赋值,所得到的结果都是一样的,都输出了一个空行,最后的结果就是三个空行。

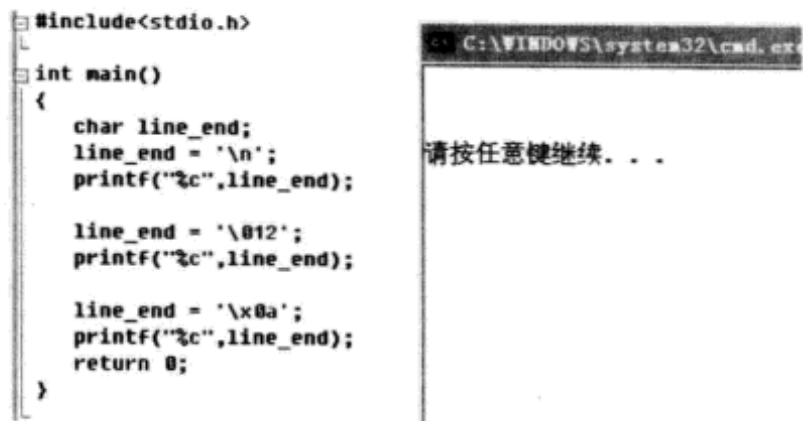


图 3.31 转义字符型赋值

3.5 类型转换

在数学运算中，对于同一种类型数据的运算，之前见到很多了，前面见到的赋值运算基本上都是。但是，对于不同类型数据的运算，至今还没有接触到。那么先从类型转换开始，它使不同类型的数据能够使用同一种运算规则，是掌握各种类型数据运算的基础。

3.5.1 什么是类型转换

在对不同类型的数据进行运算的时候，为了能够使用统一的运算规则，会把一些数据的类型进行变换，这样的变换就是类型转换。

例如，水和西瓜是不同类型的东西。现在有一个杯子，要用这个杯子盛水和西瓜。“水”和“西瓜”就类似于进行运算的数据，“盛放到杯子中”就类似于要进行的运算。水是很容易盛放到杯子中的，西瓜就有点难了，为了能使用同样的一种规则——盛放到杯子中，我们得对西瓜做一种类型转换，使用榨汁机把它从固体变成液体，图 3.32 为类型转换的示意图。

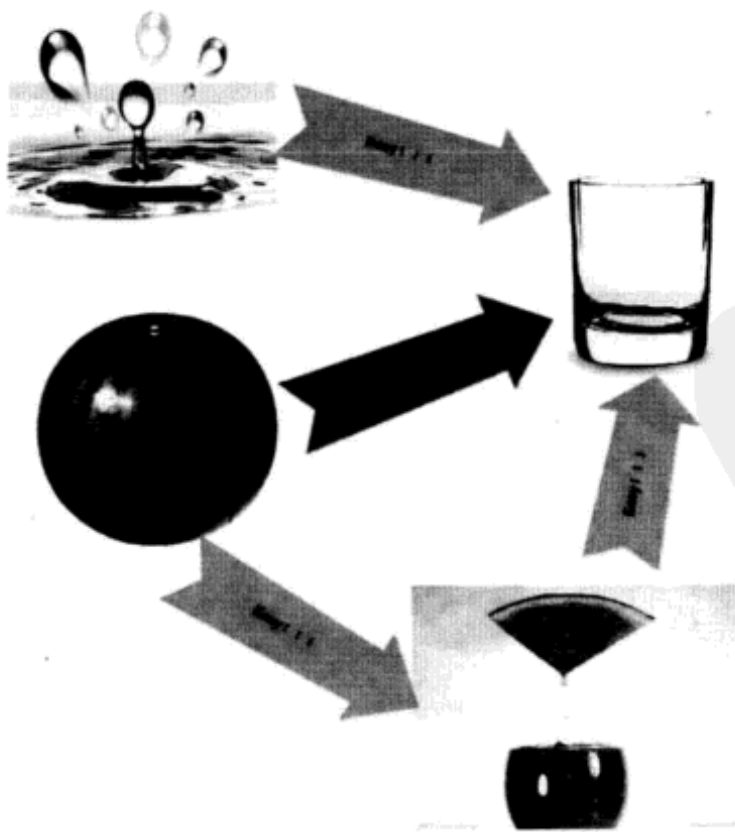


图 3.32 类型转换示意图

在计算机运算的时候，也要像把西瓜榨成汁一样，对某些数据的类型进行转换，来达到不同的运算目的。

3.5.2 类型转换的利弊

类型转换有利有弊，就像把西瓜榨成汁可以享受西瓜汁带给我们的甘甜，但是，同时难免在榨汁过程中损失一部分营养或者果汁。

对于计算机而言，类型转换带来的一个好处就是可以使用同样一种运算规则来实现不同类型数据的运算，使运算更方便。另一个好处就是，当不是很确定使用什么数据类型来表示数据的时候，可以使用比较通用的数据类型，具体运算的时候只要进行类型转换就可以。例如，要计算圆的面积，事先不知道该用整型还是该用浮点型，那就用浮点型吧，因为运算的时候可以进行类型转换。

类型转换在计算机运算的时候，也是存在一些弊漏的，主要是会损失精度。例如，要把浮点型的数转换成整型的数，就有可能损失精度，因为浮点型转换为整型时，浮点型表示的小数的小数部分会被丢弃。知道了类型转换的利和弊，在使用类型转换的时候加以注意，就可以在数学运算中很好地利用这一技术。

3.5.3 隐式类型转换和显式类型转换

在 C 语言中，类型转换有隐式类型转换和显式类型转换之分。隐式类型转换就是在运算中不知不觉就进行类型转换了，是由计算机自己做的；显示类型转换就是在运算的过程中，人为地把某些数据的类型进行了转换，是我们写程序的时候自己做的。

1. 隐式类型转换

隐式类型转换在进行类型不一致的数据运算的时候都会出现，而且转换的规则和转换的结果都是计算机事先规定好的。例如，声明定义一个整型变量 `radius`，然后给它赋值为一个小数 2.56。这个时候不用我们干涉，计算机就会把小数 2.56 先转换成整数然后赋值给整数变量。像这样的隐式类型转换是为了计算机处理方便，对于整型变量赋值都可以统一按照整型数据来处理。写代码的时候要尽量注意或者避免隐式的赋值，因为很有可能出现精度的损失。例如，下面的代码：

```
#include<stdio.h>

int main()
{
    int radius = 2.56;
    printf("%d\n",radius);
    return 0;
}
```

程序输出的结果是 2，如图 3.33 所示。因为按照 C 语言中小数转换成整数的规则是使用舍尾法，2.56 舍尾以后就是 2 了。

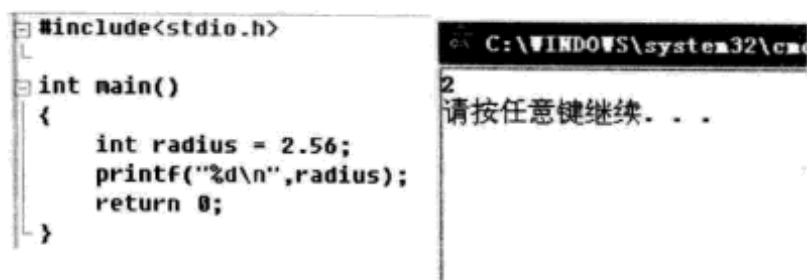


图 3.33 隐式类型转换

2. 显式类型转换

对于显式类型转换，主要用在进行不同类型的数据运算时，隐式类型转换规则不能满足我们的要求，或者我们特意强调某些地方发生了类型转换。显式类型转换的格式是：

(数据类型) 数据

或者

(数据类型) (数据)

上述格式中的“数据”可以是数值，也可以是变量和常量。例如，把浮点型表示的小数 2.56 显式地转换成整型，使用的代码如下：

(int)2.56

或者

(int) (2.56)

例如，声明定义了一个浮点型的变量 `radius` 用来表示圆的半径。假设半径很大，而且测量的小数部分都是不准确的，其实使用整数部分就可以了。所以，可以把半径显式地转换成整数，然后再赋值给 `radius` 变量。以 18299.21 这个半径为例来写演示代码，如下所示：

```
#include<stdio.h>

int main()
{
    float radius;
    radius = (int)18299.21;

    printf("%f\n",radius);
}
```

程序中，先把 18299.21 显式转换成整型，转换出来的结果为 18299，因为使用的是舍尾法，然后再把整型数值 18299 赋值给浮点型变量 `radius`。由于类型不一致，所以出现了隐式的类型转换，把 18299 转换成了小数 18299.0。最后使用 `printf()` 函数把 `radius` 这个浮点型的变量按照浮点型输出，其结果如图 3.34 所示，为 18299.000000。

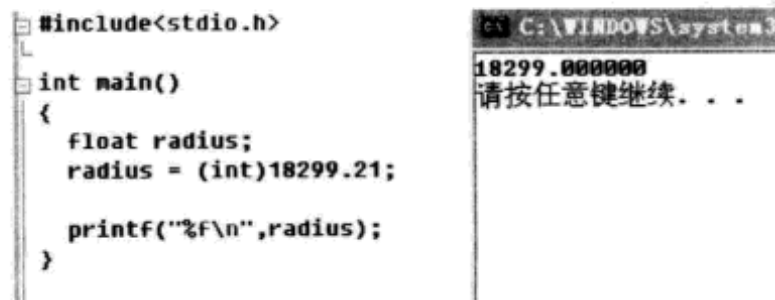


图 3.34 显式类型转换

3.5.4 赋值中的类型转换

如果将一种类型的数据赋值给另一种类型的变量，就会出现类型转换。即赋值表达式中，等号前面的变量和后面的数值类型不一致的时候，就会在赋值运算中出现隐式类型转换。

在 C 语言中，常见的类型转换有四种：浮点型赋值给整型、整型赋值给浮点型、字符型赋值给整型、整型赋值给字符型。这四种类型转换都是按照一定的规则进行的，接下来一一进行讲解。

1. 浮点型赋值给整型

浮点型赋值给整型，在 C 语言中使用的规则是：先将浮点型使用舍尾法转换成整型，即丢弃浮点型的小数部分，然后将转换后的整型赋值给整型变量。按照这个规则，浮点型赋值给整型的时候，会出现精度损失，因为小数部分都舍弃了，写代码的时候要多加注意。

例如，声明一个整型变量 `radius`，要求计算半径为 2.1 的圆的面积。那么，就得把浮点型变量 2.1 赋值给整型变量 `radius`，也就是浮点型赋值给整型。按照规则，先把浮点型的常量 2.1 进行舍尾，转换成整型常量 2，然后将整型常量 2 赋值给整型变量 `radius`。使用下面的代码予以说明：

```
#include<stdio.h>

int main()
{
    int radius;

    radius = 2.1;
    printf("%d\n",radius);

    return 0;
}
```

程序的运行结果如图 3.35 所示。第一行的输出是使用隐式类型转换来实现赋值运算的输出结果，按照浮点型赋值给整型的规则，结果为 2。

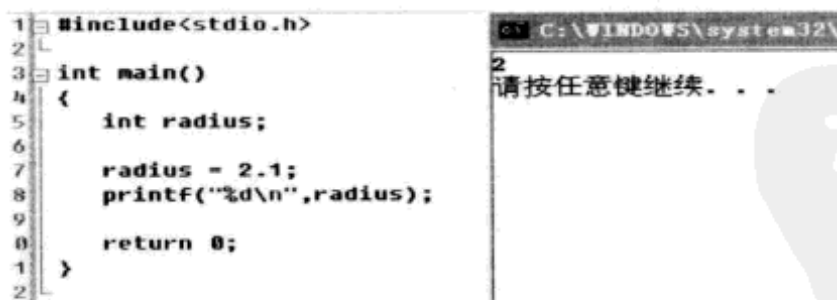


图 3.35 浮点型赋值给整型

2. 整型赋值给浮点型

C 语言中，整型赋值给浮点型的时候，类型转换的规则是：先将整型转换成浮点型，然后将转换后的浮点型赋值给浮点型变量。按照这个规则，整型赋值给浮点型的时候，实际就是给整型的小数部分补零，因此不会出现精度损失的问题。

例如，声明一个浮点型的变量 `radius`，用来表示圆的半径。在一次计算的时候，圆的半径是 3，为了暂存这个半径，要把 3 这个整型常量赋值给 `radius` 这个浮点型变量，这将出现整型赋值给浮点型的类型转换。按照转换规则，整型 3 将会被转换成浮点型的 3.0，然后赋值给浮点型变量 `radius`。下面的代码可以说明这一点：

```
#include<stdio.h>

int main()
{
    float radius;

    radius = 3;
    printf("%f\n",radius);
    return 0;
}
```

如我们所料，程序的结果输出为 3.000000，是一个浮点型变量，如图 3.36 所示。

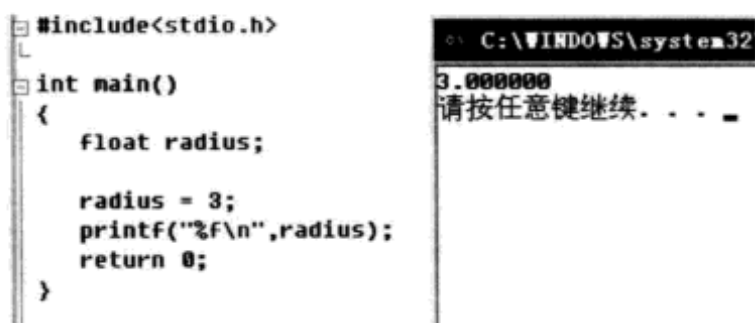


图 3.36 整型赋值浮点型

3. 字符型赋值给整型

C 语言中字符型赋值给整型使用的规则是：将字符型的 ASCII 码值直接赋值给整型变量。按照这个规则，只要查看 ASCII 表，找到对应字符的 ASCII 值，就知道类型转换的结果了。

举个例子，声明定义一个整型变量 `number`，然后把一个字符型常量 'a' 赋值给这个整型变量，这样的赋值将会出现字符型赋值给整型的类型转换。按照转换规则，'a' 的 ASCII 码值的十进制表示为 97，将 'a' 赋值给整型变量 `number`，`number` 的值将会变为 97。可以用以下的代码予以验证：

```
#include<stdio.h>

int main()
{
    int number;
    number = 'a';
    printf("%d\n",number);

    return 0;
}
```

程序的输出如图 3.37 所示，`number` 赋值 'a' 以后，`number` 的值变成了 'a' 的 ASCII 码值。

4. 整型赋值给字符型

整型赋值给字符型在 C 语言中的规则是：将整型值在 ASCII 码表中对应的字符赋值给

字符型变量。按照这个规则转换时要注意，整型可以表示的范围很大，int 整型表示的范围是 -2^{31} 到 $2^{31}-1$ ，但是字符型的 ASCII 码值的范围是 0 到 255。如果将 ASCII 码表中没有的 ASCII 值对应的整型数值赋值给字符型，就会出现乱码，乱码就是输出的时候，输出不认识的字符。

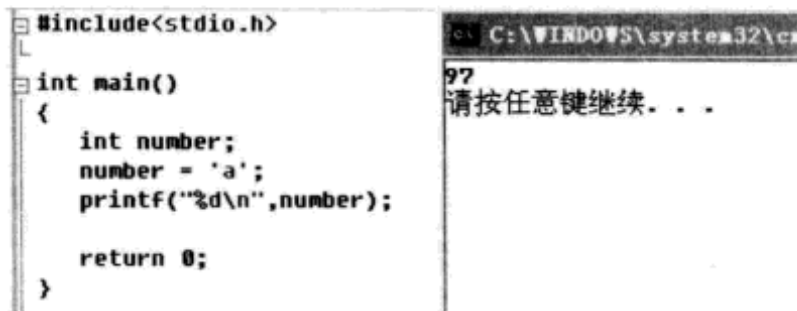


图 3.37 字符型赋值给整型

对于这个规则的验证，举这样一个例子，声明定义一个字符型变量 name，然后给它赋值 97，也就是 'a' 所对应的 ASCII 码，使用 printf() 函数输出，来验证正常的情况。接着给 name 赋值 300，在 ASCII 码表中没有对应的字符，最后使用 printf() 函数输出验证。代码如下：

```

#include<stdio.h>

int main()
{
    char name;
    name = 97;
    printf("%c\n",name);

    name =300;
    printf("%c\n",name);
    return 0;
}

```

程序的输出如图 3.38 所示。第一行的输出 'a' 是给字符变量 name 赋值 97 的结果，第二行的输出为 ','，ASCII 码表中没对 300 做规定，但是却很诡异地输出了 ','。后面再对这一诡异的现象予以解释，先记住给字符变量赋值 ASCII 码表中没有的数值，会出现想不到的结果。

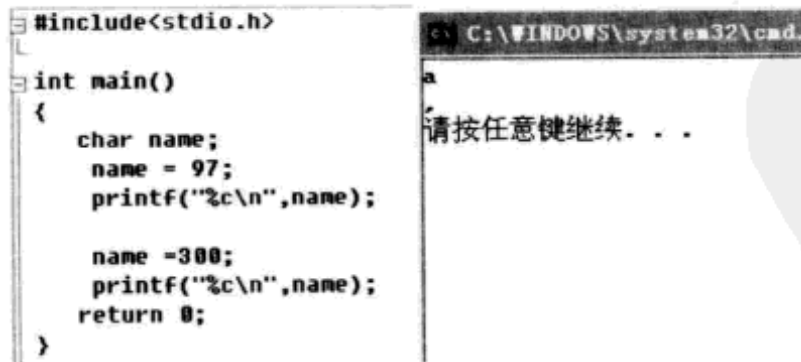


图 3.38 整型赋值字符型

现在只讲了这 4 种最常用的类型转换，其实只要出现类型不一样的赋值，都是会出现赋值的类型转换的。例如，浮点型赋值给字符型，字符型赋值给浮点型；double 浮点型赋值给 float 浮点型，float 浮点型赋值给 double 浮点型；int 整型赋值给 short 整型，short 整

型赋值给 int 整型；long 整型赋值给 short 整型，short 整型赋值给 long 整型等。

5. 其他赋值运算的类型转换

对于如此之多的赋值中出现的类型转换，该怎么去记住它们之间的类型转换规则呢？只要记住以下两条就可以。

- ❑ 所有的赋值类型转换，都是把赋值表达式中等号右边的数值转换成等号左边变量的类型，然后再进行赋值。
- ❑ 在对等号右边的数值类型进行转换的时候，先将其表示成计算机的二进制形式，然后将二进制位赋值到等号左边变量的相应二进制位中。

例如，按照这个规则把整型常量 97 赋值到字符型变量 name 中。整型常量 97 的 C 语言二进制表示形式是 00 00000 00000 00000 00000 00011 00001。把 97 赋值给字符变量 name，就是将 97 的二进制表示的最后 8 位复制到 name 变量的 8 位二进制表示中，如图 3.39 所示。

在图 3.39 中，整型二进制表示中多余的部分将会被丢弃。这也可以解释前面给字符变量赋值 ASCII 没有规定的数值 300 时，却输出了一个字符 ‘,’，其实就是因为二进制复制的时候丢弃了整型的一些部分。300 的二进制表示为 100101100，将其赋值给字符型的时候，只赋值后 8 个二进制位，也就是 00101100，对应的十进制数是 44，ASCII 码 44 对应的字符就是 ‘,’。复制过程如图 3.40 所示。

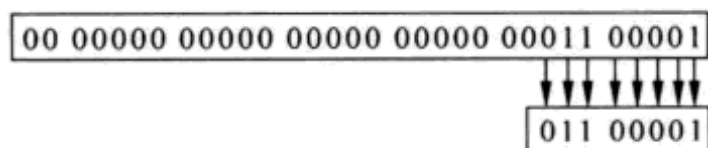


图 3.39 赋值类型转换图示

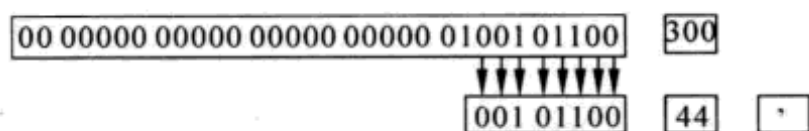


图 3.40 300 变换成逗号

3.6 基本数学运算

在 C 语言中，除了用于暂存数据的赋值运算以外，还有很多用于其他目的的数学运算。本节中就着重介绍一下 C 语言中的基本数学运算表达方式，以及使用的时候需要注意的地方。学习完本节，就可以写一些简单的用于计算的 C 语言程序了。

3.6.1 数学运算和数学表达式

C 语言中的基本数学运算主要有以下两类。

- ❑ 算术运算：主要有加、减、乘、除和求余；
- ❑ 位运算：有左移、右移、与、或、取反、异或。

算术运算主要是对常用的八进制数、十进制数和十六进制数的操作。这些运算在上学的时候，经常用到。位运算主要是对不常见的二进制数的操作，这些操作对于实现一些二进制数的运算是很方便的，由于它们是直接操作二进制数的，因此效率往往比算术运算高得多。

1. 数学运算的符号

C 语言中，是使用一些符号来表示这些运算的，就像之前讲的使用 ‘=’ 来表示赋值运算。对于基本数学运算，每种运算对应的符号，也被称为操作符，如表 3.3 所示。

表 3.3 基本数学运算符号

运算名称	操作符	表示含义
加法	+	计算两个数的和
减法	-	计算两个数的差
乘法	*	计算两个数相乘得到的积
除法	/	计算两个数相除得到的商
求余	%	计算两个数相除得到的余数
左移	<<	把一个数的二进制算术左移，最高位丢弃，最低位补零
右移	>>	把一个数的二进制算术右移，最低位丢弃，最高位补原先的最高位
与操作	&	把两个数的二进制表示的对应位求与，只要有一个 0 时为 0，两个 1 时为 1
或操作		把两个数的二进制表示的对应位求或，只要有一个 1 时为 1，两个 0 时为 0
取反操作	~	把一个数的二进制表示的每一位上的数取反，1 变 0，0 变 1
异或操作	^	把两个数的二进制表示的对应位求异或，相同为 0，相异为 1

大家对于加减乘除，可能比较眼熟，但是对于其他运算就不见得熟悉了，或者根本没有见过。没关系，只要大家先记住这些符号，讲到具体的例子的时候就会更深入地理解其含义了。

2. 数学运算表达式

在上面讲到的数学运算中，按照操作数主要有两种：一种是只有一个操作数的运算，被称为一元操作；另一种是只有两个操作数的运算，被称为二元操作。像取反操作就是一元操作，‘~’ 被称为一元操作符；加减乘除、左移、右移等都是二元操作符。对应于两种操作的 C 语言表示，分别为一元操作表达式和二元操作表达式。

(1) 一元操作表达式

对于一元操作符，它的数学运算表达式的形式如下所示：

一元操作符 操作数

“一元操作符”就是类似于取反 ‘~’，这种只有一个操作数的运算的符号表示，它的“操作数”既可以是变量，也可以是常量。例如，对常量 3 进行取反操作，C 语言表示如下所示：

~3 ;

(2) 二元操作表达式

二元操作的数学运算表达式如下所示：

操作数 1 二元运算符 操作数 2

“二元运算符”就是类似于 ‘+’ 这样的两个数的操作的符号表示，“操作数 1”和“操

作数 2”是要进行数学运算的两个数，既可以是变量也可以是常量。一般二元操作的两个操作数分别位于操作符的两侧，就像所列的二元操作表达式那样。

如果要表示 1.2 加 0.5，就可以使用加法二元操作表达式：

```
1.2 + 0.5
```

其中，“1.2”是操作数 1，是一个浮点型的常量；“+”是加法二元操作符；“0.5”是操作数 2，是一个浮点型的常量。

例如，对于一个整型变量 `num`，乘以 10 这个整型常量，求最终的计算结果。可以这样做：

- (1) 声明定义一个整型变量 `num`；
- (2) 声明定义一个整型变量 `result`；
- (3) 给 `num` 赋值为 3；
- (4) 使用乘法二元表达式的形式进行计算；
- (5) 把计算的结果赋值保存到 `result` 变量；
- (6) 使用 `printf()` 函数输出计算结果进行验证。

具体的 C 语言的实现如下所示：

```
#include<stdio.h>

int main()
{
    int num;
    int result;

    num = 3;
    result = num*10;
    printf("%d\n",result);
    return 0;
}
```

其中，`num*10` 就是乘法二元操作表达式，程序的运行结果如图 3.41 所示，输出结果为 30，和我们想象的一样。

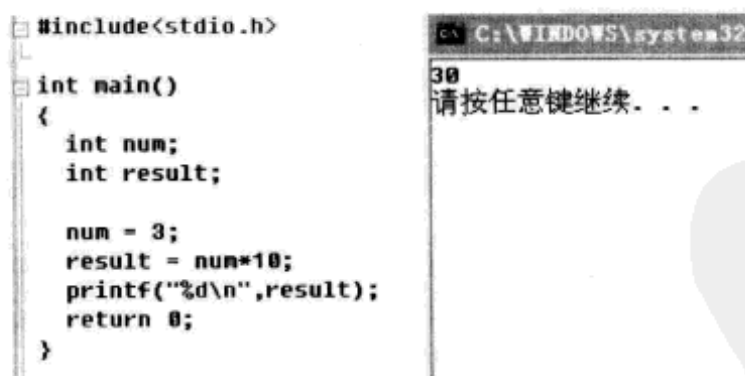


图 3.41 乘法操作表达式示例

3. 表达式的值

C 语言中，每一个表达式都是有一个值的，而且这个值是按照常量来处理的。对于数学运算表达式，它的值就是数学运算的结果。例如，上面的表达式 `num*10` 的值就是它的计算结果 30，是一个常量。因此我们可以按照赋值表达式的要求把它赋值给一个整型变量 `result`。

3.6.2 商与余数

对于算术运算中简单的加减乘除含义和运算规则，我们已经再熟悉不过了，这里就不再多讲了。在此只着重讲讲除法运算中的商和余数在 C 语言中的表示。

对于除法运算，按理来说，对小数和整数都是适合的，既有小数除法也有整数除法。1/2 可以，1.2/2 也是可以的。在 C 语言中规定只要除数不为零，都是可以使用除法进行运算的。为了方便讨论商和余数，现在只针对整数除法。

大家都知道，一个整数除以一个整数的时候，如 31 除以 2，按照图 3.42 所示按照步骤去计算，最后相减得到的结果不为 0，叫做不能整除，15 称为商，1 称为余数。

$$\begin{array}{r} 15 \\ 2 \overline{) 31} \\ \underline{2} \\ 11 \\ \underline{10} \\ 1 \end{array}$$

图 3.42 商和余数

在 C 语言中对两个整型数据使用除法运算，得到的结果就是这两个数相除的商；对这两个数使用求余运算，得到的就是这两个数相除的余数。除法运算和求余运算都是二元运算，按照二元运算表达式的形式，除法运算的表达式形式为：

被除数 / 除数

除法运算表达式的值，就是被除数除以除数得到的商。如果要计算 31/2 的商，就可以使用上面除法运算的表达式，具体形式如下：

31 / 2

其中，“31”就是被除数，是一个整型常量；“/”是除法运算符；“2”是除数，是一个整型常量；整个表达式“31/2”的结果是一个整型常量，其值为 31 除以 2 所得到的商。

求余运算的表达式如下：

被除数 % 除数

求余运算表达式的值，就是被除数除以除数所得的余数。还是以 31/2 作为例子，计算 31/2 所得到的余数。使用上面求余运算的表达式，具体形式如下：

31 % 2

其中，“31”就是被除数，是一个整型常量；“%”是求余运算符；“2”是除数，是一个整型常量；整个表达式“31%2”的结果是一个整型常量，其值为 31 除以 2 所得到的余数。

让我们来看看，要想求出 31 除以 2 得到的商和余数，在 C 语言中是如何实现的。首先，声明定义两个整型变量 quotient 和 remainder，分别用来表示商和余数。然后把 31 除以 2 的结果赋值给 quotient，把 31 对 2 求余的结果赋值给 remainder。最后使用 printf() 函数输出给予验证。程序如下：

```
#include<stdio.h>

int main()
{
    int quotient;
    int remainder;
```

```
quotient = 31/2;
remainder = 31%2;
printf("%d\n",quotient);
printf("%d\n",remainder);
}
```

程序输出的商为 15，余数为 1，如图 3.43 所示。

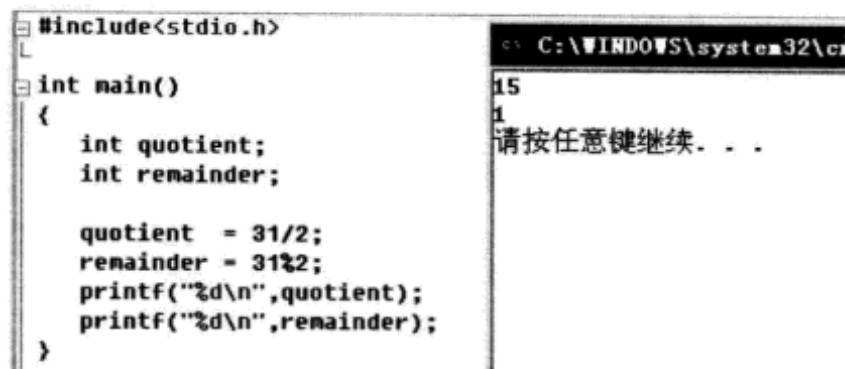


图 3.43 C 语言中计算商和余数

3.6.3 位运算

由于位运算直接对二进制表示进行操作，因此其效率远远高于对其他进制表示的算术运算。掌握好位运算可以大大提升程序的性能。C 语言提供的位运算主要有 6 种：左移运算、右移运算、与操作运算、或操作运算、异或操作运算、按位取反运算。

表 3.4 列出了这 6 种运算符和运算规则，下面将按照表中的运算符和规则，详细地介绍这 6 种位运算。

表 3.4 C 语言中的位运算

名 称	运算符	运 算 规 则
左移运算	<<	把一个数的二进制整数左移，最高位丢弃，最低位补零
右移运算	>>	把一个数的二进制整数右移，最低位丢弃，最高位补原先的最高位
与操作运算	&	把两个数的二进制表示的对应位求与，只要有一个 0 时为 0，两个 1 时为 1
或操作运算		把两个数的二进制表示的对应位求或，只要有一个 1 时为 1，两个 0 时为 0
异或操作运算	^	把两个数的二进制表示的对应位求异或，相同为 0，相异为 1
按位取反运算	~	把一个数的二进制表示的每一位上的数取反，1 变 0，0 变 1

从表 3.4 中可以看出，按照运算规则，可以把这 6 种位运算分为三类。

- 移位运算：包括左移运算和右移运算。它们的运算规则比较相近，都是移位，然后补位。
- 二元位运算：包括与操作运算、或操作运算和异或操作运算。它们都是对两个数按照对应位进行运算的，可以归为一类。
- 取反运算：包括按位取反运算。之所以将其单独列为一类，是因为它是对一个数自身进行运算。

1. 移位运算

由于移位运算是一个二元运算，因此按照二元运算的表达式的形式如下所示：

被移位的数 移位运算符 移动的位数

其中，“被移位的数”就是要进行移位的操作数，可以是整型常量，也可以是整型变量；“移位运算符”是“<<”或者“>>”；“移动的位数”表示要对“被移位的数”移动几位。

例如，对3左移一位，按照左移运算表达式的格式要求，其形式如下：

3 << 1

对6右移一位，按照右移运算表达式的格式要求，其形式如下：

6 >> 1

下面分别来看看这两个移位运算的规则是如何实现的。

(1) 左移运算

假设使用32位的补码表示3（C语言就是这样表示的），如图3.44所示。按照左移运算规则：把3的二进制表示左移一位后丢掉最高位，并在最低位补0所得的结果，按照补码表示的二进制数对应的十进制表示为6，也就是3左移一位后得到的结果为6。

(2) 右移运算

与左移运算类似，假设使用32位的补码表示6，如图3.45所示。按照右移运算规则：把6的二进制表示右移一位后丢掉最低位，并在最高位补原来的最高位所得的结果，按照补码表示的二进制数对应的十进制表示为3，也就是6右移一位后得到的结果为3。

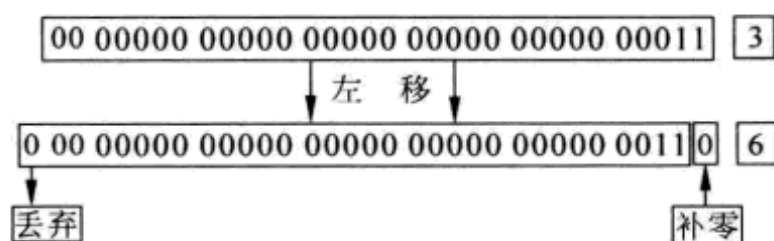


图 3.44 左移运算

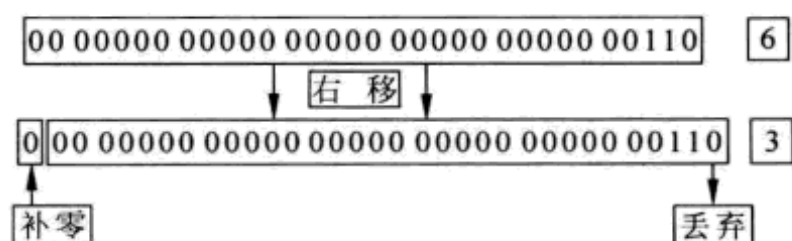


图 3.45 右移运算

接下来，用C语言的代码进行左移运算表示，并来验证一下移位运算的运算规则。

- 声明一个整型变量 result；
- 给 result 赋值 3<<1，由于表达式 3<<1 也是一个常量值，所以可以这样赋值；
- 使用 printf()函数进行输出验证左移结果；
- 给 result 赋值 6>>1；
- 使用 printf()函数进行输出验证右移结果。

代码如下所示：

```
#include<stdio.h>

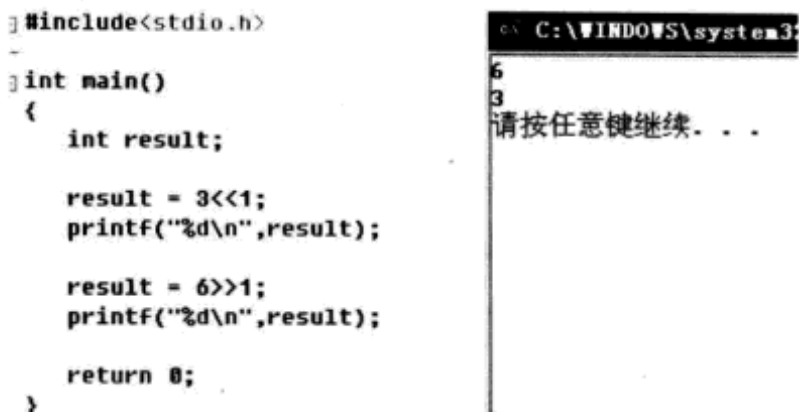
int main()
{
    int result;

    result = 3<<1;
    printf("%d\n",result);

    result = 6>>1;
    printf("%d\n",result);
}
```

```
return 0;
}
```

程序的输出结果为 6 和 3，如图 3.46 所示。



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

    result = 3<<1;
    printf(\"%d\\n\",result);

    result = 6>>1;
    printf(\"%d\\n\",result);

    return 0;
}
```

C:\WINDOWS\system32
6
3
请按任意键继续...

图 3.46 移位运算验证

2. 二元位运算

由于二元位运算也是一个二元运算，因此按照二元位运算的表达式的形式，其表达式的形式如下所示：

操作数 1 位运算符 操作数 2

其中，“操作数 1”就是要进行二元位运算的第一个操作数，可以是整型常量，也可以是整型变量；“位运算符”就是“&”、“|”或者“^”中的任意一个；“操作数 2”就是要进行二元位运算的第二个操作数，可以是整型常量，也可以是整型变量；

例如，要对 15 和 8 进行与运算，其形式如下：

15 & 8

要对 15 和 8 进行或运算，其形式如下：

15 | 8

要对 15 和 8 进行异或运算，其形式如下：

15 ^ 8

下面分别来看看这三个位运算的规则是如何实现的。

(1) 与运算

假设使用 32 位的补码表示 15 和 8，如图 3.47 所示。按照与运算规则：把 15 和 8 对应位进行与运算得到的结果，按照补码表示的二进制数对应的十进制表示为 8，也就是 15 与 8 进行与运算得到的结果为 8。

(2) 或运算

假设使用 32 位的补码表示 15 和 8，如图 3.48 所示。按照或运算规则：把 15 和 8 对应位进行或运算得到的结果，按照补码表示的二进制数对应的十进制表示为 15，也就是 15 和 8 进行或运算得到的结果为 15。

(3) 异或运算

假设使用 32 位的补码表示 15 和 8，如图 3.49 所示。按照异或运算规则：把 15 和 8 对应位进行异或运算得到的结果，按照补码表示的二进制数对应的十进制表示为 7，也就是 15 和 8 进行异或运算得到的结果为 7。

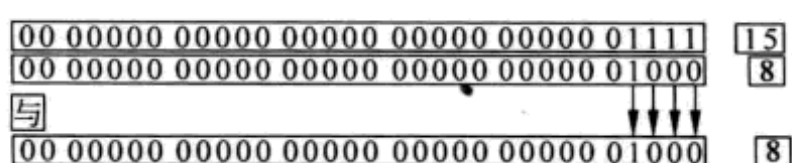


图 3.47 与操作运算

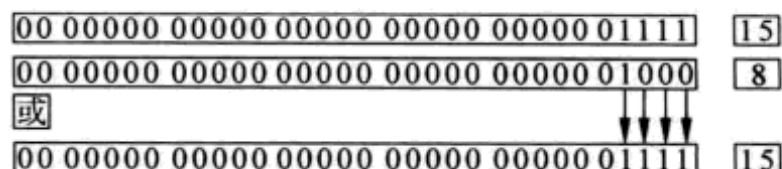


图 3.48 或操作运算

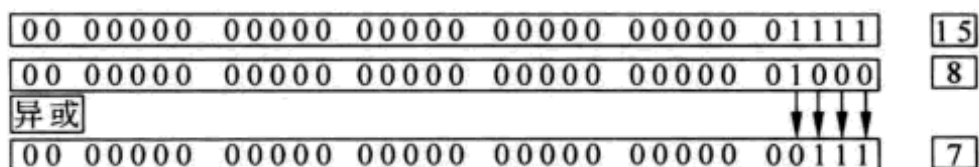


图 3.49 异或操作运算

接下来，用 C 语言的代码进行与运算表示，并来验证一下这三个二元位运算的运算规则。按照以下步骤进行程序编写：

- 声明一个整型变量 `result`;
- 给 `result` 赋值 `15&8`;
- 使用 `printf()` 函数进行输出验证与运算结果;
- 给 `result` 赋值 `15|8`;
- 使用 `printf()` 函数进行输出验证或运算结果;
- 给 `result` 赋值 `15^8`;
- 使用 `printf()` 函数进行输出验证异或运算结果。

代码如下所示：

```
#include<stdio.h>

int main()
{
    int result;

    result =15&8;
    printf("%d\n",result);
    result =15|8;
    printf("%d\n",result);
    result =15^8;
    printf("%d\n",result);

    return 0;
}
```

程序的输出结果为 8、15 和 7，如图 3.50 所示。

```
#include<stdio.h>

int main()
{
    int result;

    result =15&8;
    printf("%d\n",result);
    result =15|8;
    printf("%d\n",result);
    result =15^8;
    printf("%d\n",result);

    return 0;
}
```

```
C:\WINDOWS\system32\
8
15
7
请按任意键继续...
```

图 3.50 二元位运算验证

3. 取反运算

与其他的位运算不同的是，取反运算是一个一元运算。因而，取反运算的表达式如下：

~取反操作数

这个表达式中，“~”表示的是取反操作符，“取反操作数”就是要进行取反操作的数，可以是整型常量，也可以是整型变量。例如，对 15 进行取反，按照取反运算表达式的格式要求，其形式如下：

~15

对 15 进行取反运算，得到的结果就是-16。假设使用 32 位的补码表示 15，如图 3.51 所示。按照或运算规则：把 15 的每一位取反所得的结果，按照补码表的二进制数对应的十进制表示为-16（对于结果为什么是-16，请回过头复习十进制和二进制的转换），也就是对 15 进行取反操作后得到的结果为-16。

接下来，用 C 语言的代码进行取反运算表示，并来验证一下取反运算的运算规则。要对 15 进行取反，首先声明定义一个整型变量 result，然后把~15 这个整型常量赋值给 result，最后使用 printf()函数进行输出验证结果。代码如下所示：

```
#include<stdio.h>

int main()
{
    int result;
    result = ~15;

    printf("%d\n",result);
}
```

程序的输出结果为-16，如图 3.52 所示。

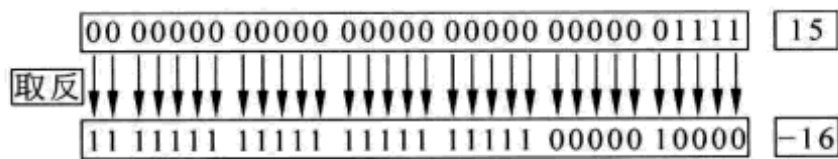


图 3.51 取反运算

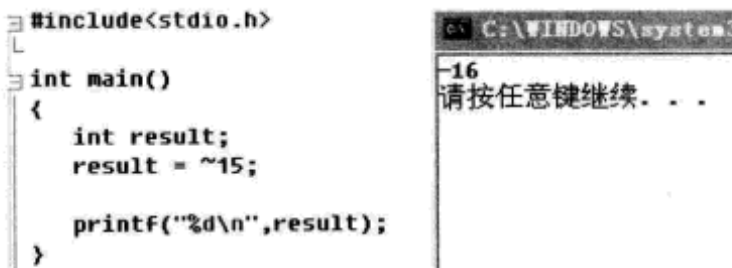


图 3.52 取反运算验证

4. 负数的位运算

位运算主要是针对整数的，也就是针对 C 语言中的整型数据的。前面看到的主要是正整数的位运算，接下来，专门介绍一下负整数的位运算，它比正整数的位运算稍微麻烦点，涉及的补码运算比较多。

对于负数的各个位运算规则和前面讲的都是一样的，这里只举例子加以说明。看一下 -3 左移和-2 取反的例子吧，分别代表二元运算和一元运算。

如图 3.53 所示，-3 左移一位得到的结果为-6，图 3.53 中的二进制都是负数的补码表示。大家可以参见十进制和二进制的转换来复习负数的补码表示。

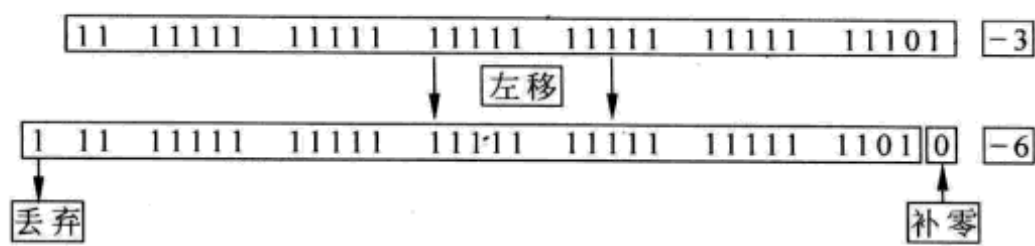


图 3.53 负数左移

如图 3.54 所示，-2 按位取反所得到的结果为 1，通过这个例子和上面图 3.51 的例子，大家可以看出一个正数取反的结果为负数，一个负数取反得到的结果为正数。

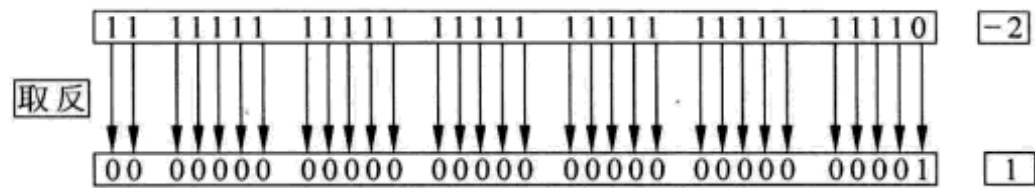


图 3.54 负数取反

3.6.4 优先级的奥秘

在基本数学运算中，还有一个东西需要注意，那就是运算符的优先级。大家知道，在数学中，先计算乘法和除法，后计算加法和减法，这个问题被称为运算符的优先级问题。当加法和减法同时出现的时候，按照从左到右的顺序，哪个在前先计算哪个，这个问题被称为运算符的结合性问题。在 C 语言中的数学运算也有类似的运算优先级和结合性的问题，本节就讲讲基本数学运算的运算优先级问题。

1. 复合表达式

像之前讲解的一元运算表达式和二元运算表达式，都是简单表达式。例如，1+5，~15，15|8 等都是简单表达式。由于简单表达式本身都有一个值，其值为表达式运算的结果，因此，我们可以把一个表达式当作另一个表达式的操作数来使用，这样组成的表达式被称为复合表达式。

现在最基础的基本数学运算的表达式有以下两种：

- (1) 一元操作符 操作数。
- (2) 操作数 1 二元运算符 操作数 2。

把这两个简单的表达式组合在一起，就可以得到各种形式的复合表达式。

例如，4+5，2*2 都是一个简单表达式。用 2*2 替换 4+5 中的操作数 4 就可以得到一个复合表达式 2*2+5；如果再往下替换，用 10/2 替换 5 就可以得到更复杂的复合表达式 2*2+10/2。

2. 自然结合

一个复合表达式中包含着各种简单表达式。如果要计算复合表达式的结果，就得先知道应该先算哪个简单表达式，后算哪个简单表达式。这就是基本数学运算的优先级和结合性问题。按照数学运算的优先级来确定先算哪个简单表达式，后算哪个简单表达式，这样的结合方式被称为自然结合。

- C 语言规定的运算的优先级如表 3.5 所示。其中，
- 优先级表示的就是这种运算在所有运算中的优先级。例如，加法运算排第 4，除法运算排第 3，那么除法运算的优先级就高于加法运算的优先级。
 - 结合方向表示的是对于同一优先级的运算，按照结合方向来进行运算就可以。除 ‘/’、乘 ‘*’ 和求余 ‘%’ 运算的优先级都是 3，遇到这几个运算在一起，按照结合方向从左到右运算就可以了。有的时候也会出现从右到左的结合，例如负号运算符 ‘-’，-3 就是把 3 和 ‘-’ 从右向左结合的。

表 3.5 C语言中的运算优先级

优先级	运算符	名称或含义	表达式形式	结合方向	说 明
1	[]	数组下标	数组名[常量]	左到右	一元运算符
	()	圆括号	(表达式)/函数名(形参表)		一元运算符
	.	成员选择(对象)	对象.成员名		二元运算符
	->	成员选择(指针)	对象指针->成员名		二元运算符
2	-	负号运算符	-操作数	右到左	一元运算符
	(类型)	强制类型转换	(数据类型)操作数		一元运算符
	++	自增运算符	++变量名/变量名++		一元运算符
	--	自减运算符	--变量名/变量名--		一元运算符
	*	取值运算符	*指针变量		一元运算符
	&	取地址运算符	&变量名		一元运算符
	!	逻辑非运算符	!操作数		一元运算符
	~	按位取反运算符	~操作数		一元运算符
	sizeof	长度运算符	sizeof(操作数)		
3	/	除	操作数/操作数	左到右	二元运算符
	*	乘	操作数*操作数		二元运算符
	%	余数(取模)	整型操作数/整型操作数		二元运算符
4	+	加	操作数+操作数	左到右	二元运算符
	-	减	操作数-操作数		二元运算符
5	<<	左移	操作数<<操作数	左到右	二元运算符
	>>	右移	操作数>>操作数		二元运算符
6	>	大于	操作数>操作数	左到右	二元运算符
	>=	大于等于	操作数>=操作数		二元运算符
	<	小于	操作数<操作数		二元运算符
	<=	小于等于	操作数<=操作数		二元运算符
7	==	等于	操作数==操作数	左到右	二元运算符
	!=	不等于	操作数!=操作数		二元运算符
8	&	按位与	操作数&操作数	左到右	二元运算符
9	^	按位异或	操作数^操作数	左到右	二元运算符
10		按位或	操作数 操作数	左到右	二元运算符
11	&&	逻辑与	操作数&&操作数	左到右	二元运算符
12		逻辑或	操作数 操作数	左到右	二元运算符

续表

优先级	运算符	名称或含义	表达式形式	结合方向	说明
13	?:	条件运算符	操作数 1? 操作数 2: 操作数 3	右到左	三元运算符
14	=	赋值运算符	变量=操作数	右到左	二元运算符
	/=	除后赋值	变量/=操作数		二元运算符
	=	乘后赋值	变量=操作数		二元运算符
	%=	取模后赋值	变量%=操作数		二元运算符
	+=	加后赋值	变量+=操作数		二元运算符
	-=	减后赋值	变量-=操作数		二元运算符
	<<=	左移后赋值	变量<<=操作数		二元运算符
	>>=	右移后赋值	变量>>=操作数		二元运算符
	&=	按位与后赋值	变量&=操作数		二元运算符
	^=	按位异或后赋值	变量^=操作数		二元运算符
	=	按位或后赋值	变量 =操作数		二元运算符
15	,	逗号运算符	操作数,操作数,...	左到右	从左向右顺序运算

现在只关心我们见过的基本数学运算，对于其他的运算见到时再做详细讲解。之前见过的基本数学运算的优先级如表 3.6 所示。

表 3.6 基本数学运算符优先级

优先级	运算符	名称或含义	表达式形式	结合方向	说 明
2	~	按位取反运算符	~操作数	从右向左	一元运算符
3	/	除	操作数/操作数	左到右	二元运算符
	*	乘	操作数*操作数		二元运算符
	%	余数（取模）	整型操作数/整型操作数		二元运算符
4	+	加	操作数+操作数	左到右	二元运算符
	-	减	操作数-操作数		二元运算符
5	<<	左移	操作数<<操作数	左到右	二元运算符
	>>	右移	操作数>>操作数		二元运算符
8	&	按位与	操作数&操作数	左到右	二元运算符
9	^	按位异或	操作数^操作数	左到右	二元运算符
10		按位或	操作数 操作数	左到右	二元运算符

从表 3.6 中可以看出，在介绍的基本数学运算中，按位取反运算的优先级最高，接下来，依次为：乘除余、加减、左右移、与、异或、或。所以，在计算基本数学运算表达式时，只要按照这个顺序去组合简单表达式，就可以一步一步计算出复合表达式的值了。

例如一个比较复杂的表达式：10/~2+16%4*3&8-2+6|8，看上去貌似很复杂，我们一步一步进行解析就可以得出正确的结果。具体步骤如图 3.55 所示，图中的标号①②③④…表示的是具体的计算顺序。

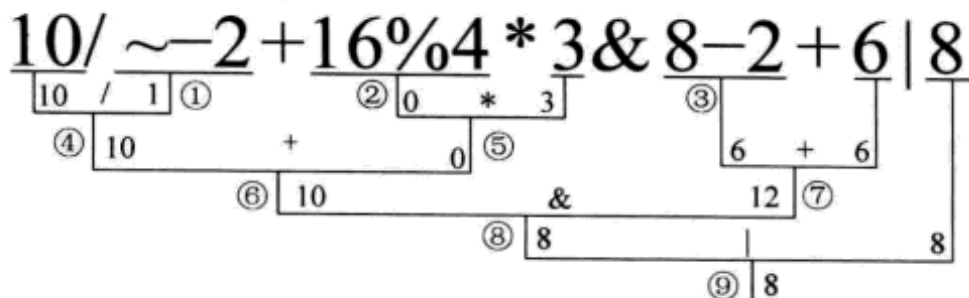


图 3.55 复合表达式自然结合

可以总结图 3.55 的计算过程为以下 4 步。

- (1) 复合表达式中所有的基本数学运算符：/、~、+、%、*、&、-、+、|;
- (2) 按照各种运算的优先级依次结合组成简单表达式;
- (3) 依次计算每个简单表达式，并用结果的值取代表达式;
- (4) 计算最终结果为 8。

接下来，用 C 来验证一下这个复合表达式的值是不是 8。首先声明定义一个整型变量 `result`，然后把 `10/~-2+16%4*3&8-2+6|8` 复合表达式常量赋值给 `result`，最后使用 `printf()` 函数进行输出验证结果。代码如下所示：

```
#include<stdio.h>

int main()
{
    int result;
    result =10/~-2+16%4*3&8-2+6|8;

    printf("%d\n",result);
}
```

程序的输出结果为 8，如图 3.56 所示。

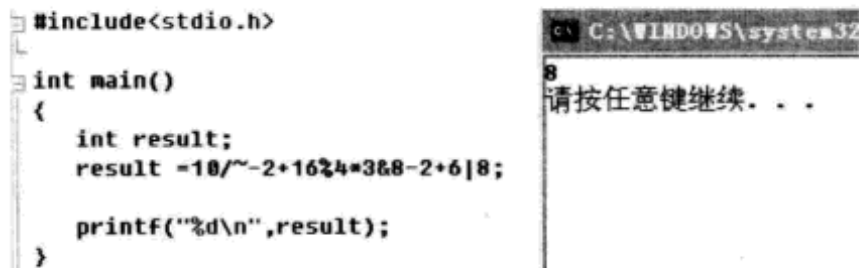


图 3.56 复合表达式自然结合验证

3. 改变优先级——强制结合

有的时候，在进行运算的时候，需要先计算一些优先级比较低的运算。这个时候，就需要优先级的改变了，在 C 语言中，使用小括号“（）”进行强制结合来提高低优先级的运算的优先级。

例如，在复合表达式 `10/~-2+16%4*3&8-2+6|8` 中，要先计算 `3&8`，也就是提高与运算‘&’的优先级，就可以使用小括号将其括住。即 `10/~-2+16%4*(3&8)-2+6|8`，这样计算结果就会发生变化。按照上一节的对复合表达式的解析步骤，可以计算出这个表达式的值。具体步骤如图 3.57 所示，图中的标号①②③④…表示的是具体的计算顺序。

可以总结图 3.57 的计算过程为以下 4 步。

- (1) 复合表达式中所有的基本数学运算符：/、~、+、%、*、&、-、+、|;

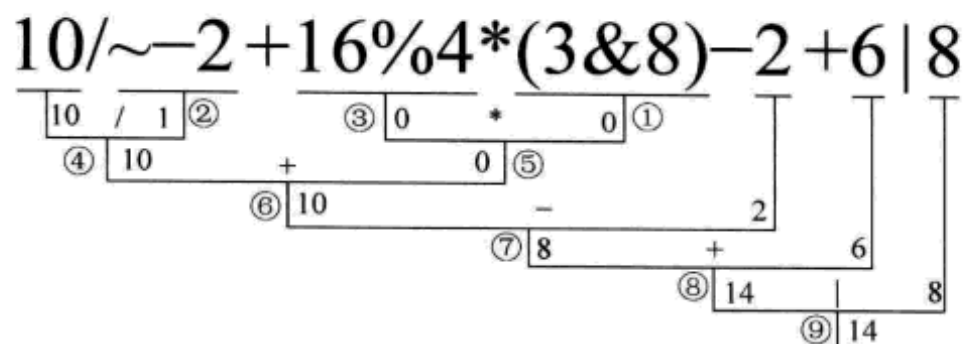


图 3.57 强制结合改变运算优先级

- (2) 按照各种运算的优先级依次结合组成简单表达式，先结合小括号中的表达式；
- (3) 依次计算每个简单表达式，并用结果的值取代表表达式；
- (4) 计算最终结果为 14。

接下来，用 C 语言来验证一下这个添加了强制结合的复合表达式的值是不是 14。首先声明定义一个整型变量 `result`，然后把 `10/~-2+16%4*(3&8)-2+6|8` 复合表达式常量赋值给 `result`，最后使用 `printf()` 函数进行输出验证结果。代码如下所示：

```
#include<stdio.h>

int main()
{
    int result;
    result =10/~-2+16%4*(3&8)-2+6|8;

    printf("%d\n",result);
}
```

程序的输出结果为 14，如图 3.58 所示。

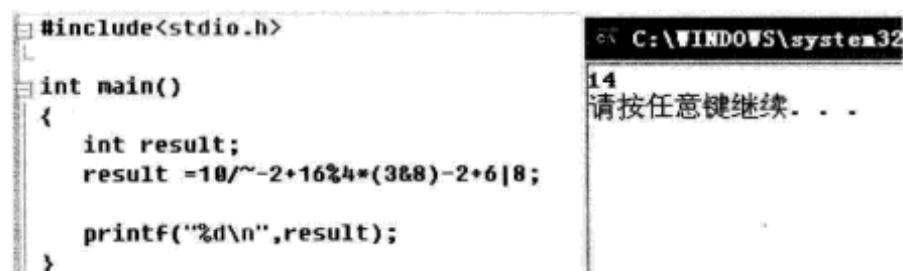


图 3.58 复合表达式强制结合验证

3.6.5 数学运算中的类型转换

凡是出现类型不一致的运算的时候都会有类型转换的存在。之前所讲的二元基本数学运算的表达形式中，对于“操作数 1”和“操作数 2”，只要求其为常量或者变量，但是，并未对其类型进行限制，只要“操作数 1”和“操作数 2”的类型不一致就会出现数学运算中的类型转换。

数学运算中的类型转换也分为隐式类型转换和显式类型转换。

1. 隐式类型转换

隐式类型转换和赋值运算表达式类似，二元基本数学运算中的隐式类型转换也是自动发生的，只要是两个操作数的类型不一致就会自动发生类型转换，如图 3.59 所示。

隐式类型转换的转换规则为：根据两个操作数的类型，把其中的一个按照图 3.59 箭头所示的方向转换成另一个的类型。例如， $1.2 + 1$ ， 1.2 在 C 语言中是一个浮点型常量，其类型为 `double`， 1 在 C 语言中是一个整型常量，其类型为 `int`。按照转换规则，先将 1 转换成 `double` 类型（对于整型转浮点型，参见 3.5.4 节中介绍的类型转换），然后再进行加法运算，结果为 2.2 。

可以用 C 语言程序来验证这个转换结果。首先声明定义一个 `double` 浮点型变量 `result`，然后把 $1.2+1$ 赋值给 `result`，最后使用 `printf()` 函数进行输出验证结果。代码如下所示：

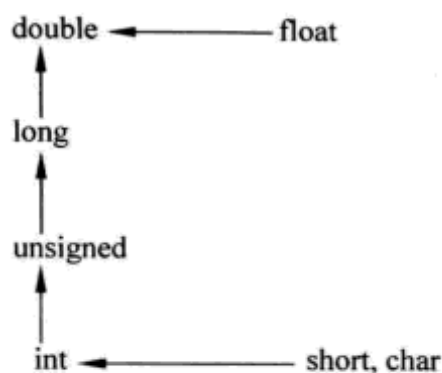


图 3.59 类型转换规则

```
#include<stdio.h>

int main()
{
    double result;
    result =1.2+1;

    printf("%lf\n",result);
}
```

程序的输出结果为 2.200000 ，如图 3.60 所示。

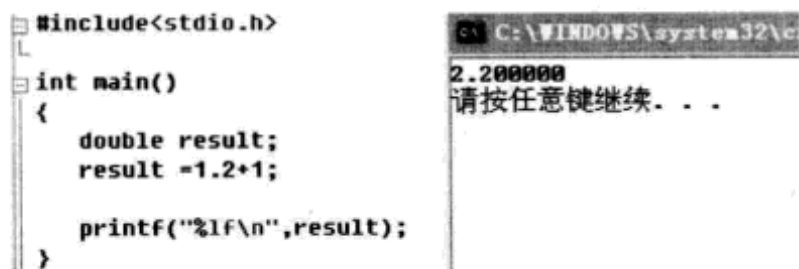


图 3.60 隐式类型转换验证

2. 显式类型转换

基本数学运算中，显式类型转换和赋值运算中的显示类型转换格式是一样的，如下所示：

(数据类型) 数据

或者

(数据类型) (数据)

上述格式中的“数据”可以是数值，也可以是变量和常量。例如，把浮点型表示的小数 2.56 显式地转换成整型，代码如下：

(int)2.56

或者

(int) (2.56)

以 $1.2+1$ 这个加法数学运算作为例子来说明显式类型转换的使用方法。 1.2 在 C 语言中是一个浮点型常量，其类型为 `double`， 1 在 C 语言中是一个整型常量，其类型为 `int`。我们

不希望按照隐式转换规则那样将 1 转换成 double，而是要将 1.2 转换成 int，然后再进行整型 int 之间的加法运算。C 语言代码如下所示，首先声明定义一个 int 整型变量 result，然后把 (int)1.2+1 赋值给 result。

```
#include<stdio.h>

int main()
{
    int result;
    result = (int)1.2+1;

    printf("%d\n",result);
}
```

程序的输出结果为 2，如图 3.61 所示。按照 3.5.4 节中所述浮点型转换成整型的规则，浮点型的 1.2 会被转换成整型的 1，然后两个整型的 1 相加所得的结果就是 2，跟程序的输出是一样的。

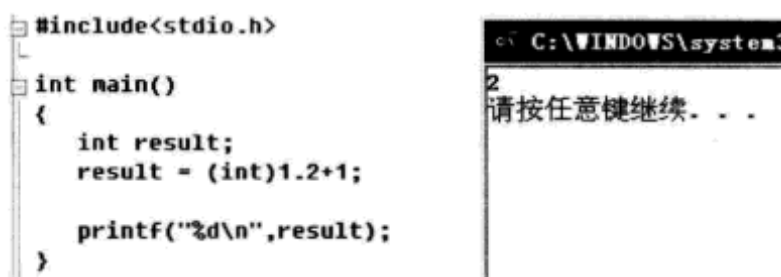


图 3.61 显式类型转换验证

3.7 复合赋值运算

C 语言还提供一种特殊的运算符——复合赋值运算符。它不仅可以用来进行基本算术运算，还可以把计算结果赋值给一个变量。

3.7.1 复合赋值运算

复合赋值运算是一般数学运算和赋值运算的结合。本节中主要讲解一般的复合赋值运算的含义、形式及使用方法。

1. 复合赋值运算的含义和形式

复合赋值运算是结合了简单数学运算和赋值运算的简单运算。其实，写代码的时候完全可以不使用复合赋值运算，使用一般的简单数学运算和赋值运算也能完成任务。既然 C 语言提供了这种运算来简化我们的工作，就有必要介绍给大家学习。

一般的复合赋值运算是一个二元运算，也就是它的操作数有两个。其表达式格式如下所示：

变量 操作符 操作数

与其他的二元运算不同的是，复合赋值运算的第一个操作数必须是变量。第二个操作

数既可以是变量也可以是常量。“操作符”就是表示复合赋值运算的符号。

复合赋值表达式完成以下两步工作：

- (1) 将“变量”和“操作数”进行数学运算；
- (2) 将 (1) 的运算结果赋值给变量。

2. 复合赋值运算符

接下来看看 C 语言中所有的复合赋值运算。表 3.7 列出了 C 语言中的复合赋值运算符，以及它们的优先级、含义、表达式形式和结合方向。

表 3.7 复合赋值运算符

优 先 级	运 算 符	名称或含义	表达式形式	结 合 方 向	说 明
14	/=	除后赋值	变量/=操作数	从右到左	二元运算符
	=	乘后赋值	变量=操作数		二元运算符
	%=	取模后赋值	变量%=操作数		二元运算符
	+=	加后赋值	变量+=操作数		二元运算符
	-=	减后赋值	变量-=操作数		二元运算符
	<<=	左移后赋值	变量<<=操作数		二元运算符
	>>=	右移后赋值	变量>>=操作数		二元运算符
	&=	按位与后赋值	变量&=操作数		二元运算符
	^=	按位异或后赋值	变量^=操作数		二元运算符
	=	按位或后赋值	变量 =操作数		二元运算符

从表 3.7 中可以看出复合赋值运算的优先级为 14，是很低的。也就是说，一般是当其他运算都完成以后才进行复合赋值运算的。

下面用表 3.7 中的“除后赋值”运算来说明复合赋值运算的使用方法。表 3.7 中其他的复合赋值运算都是一样的。“除后赋值”运算复合运算的表达式形式如下：

变量 /= 除数

其中的“变量”，在开始计算的时候，保存的是被除数的值，最后复合赋值运算之后保存的值为除法运算的商。“/=”为除后赋值复合运算符。“除数”就是进行除法运算的除数，既可以是变量，也可以是常量，还可以是一个运算的表达式，因为表达式也是一个常量值。

例如，要使用复合赋值运算把 10 除以 2 的结果赋值给一个整型变量 result，可以使用以下的表示：

```
int result;
result = 10;
result /= 2;
```

第一、二个表达式是我们熟悉的赋值表达式，先定义了一个整型变量 result，然后把整型常量 10 赋值给整型变量 result。完成了这两步以后，result 的值就变成 10 了。

接下来看看第三个表达式，这就是所要介绍的复合赋值表达式。其中，result 就是“变量”，“/=”是除后赋值操作符，2 是“操作数”。按照复合赋值表达式的运算步骤，这个表达式完成了下面两件事。

(1) 将“变量” result 和“操作数” 2 相除, result (其值已经为 10 了) 除以 2, 结果为 5。

(2) 将 (1) 的结果, 也就是 5 赋值给“变量” result。

通过这两步的计算, 复合表达式“result /= 2”就完成了基本数学运算(除法运算)和赋值运算的结合。另外, 复合赋值表达式也是一个表达式, 凡是表达式都是有一个值的, 复合赋值表达式的值就是运算完成以后表达式中“变量”的值。

3.7.2 自增自减运算——特殊的复合赋值

在上一小节中介绍了一般的复合赋值运算, 这些赋值运算都是二元运算。本小节将介绍两个很特殊的复合赋值运算, 它们已经将复合赋值运算简化到了极点, 有点不像复合赋值运算了。这两种运算分别是自增运算和自减运算。

之所以说自增自减运算特殊, 而且很简化, 是因为它是一元运算。一个一元运算要完成基本的数学运算(自增自减运算完成的基本数学运算为加法和减法运算), 还要完成赋值运算, 这是何等的简化!

虽然自增自减运算是一元运算, 但是它与一般的一元运算表达式的形式有所不同, 自增和自减运算的表达式形式分别有两种: 左自增和右自增, 左自减和右自减。

□ 左自增运算的表达式形式如下:

```
++变量
```

□ 右自增运算的表达式形式如下:

```
变量++
```

□ 左自减运算的表达式形式如下:

```
--变量
```

□ 右自减运算的表达式形式如下:

```
变量--
```

这 4 种自增自减运算表达式都是要完成两件事情, 只是在和其他运算结合时, 完成的时间不一样而已。这两件事情分别是:

(1) 将“变量”的值加或者减 1;

(2) 将 (1) 的结果赋值给“变量”。

下面举一个 C 语言中很常见的简单例子。例如, 有一个整型变量 i, 我们每次要对 i 加 1, 这个时候就可以使用自增运算了, 具体形式如下:

```
i++;
```

执行完这个表达式以后, i 的值就会被自动加 1。因为右自增就是先读取 i 的值, 对该值加 1, 然后把所得的结果赋值给 i。

作为对上面内容的一个总结, 表 3.8 列出了自增自减运算符的表达式形式、优先级及结合性。从表 3.8 中可以看出自增自减运算的优先级是很高的, 为 2, 而且自增和自减运算分别都有两种表达式形式。

表 3.8 自增自减运算符

优 先 级	运 算 符	名称或含义	表达式形式	结 合 方 向	说 明
2	++	自增运算符	++变量名/变量名++	从右到左	一元运算符
	--	自减运算符	--变量名/变量名--		一元运算符

3.7.3 自增自减运算的使用

由于自增自减运算有四种形式的表达式，而且它们都是完成加减 1 和赋值两件事，只有在具体的使用中，和不同的运算结合起来才会显示出各自的特点，因此，接下来讲解自增自减的使用，以更加深入地理解其不同形式。

下面以左自增和右自增为例，左自减、右自减与左自增、右自增与之类似的，只是将“加”变成“减”而已。

1. 纯粹的自增运算

纯粹的自增运算，也就是自增不与其他运算结合，只是一个简单表达式的形式。这个时候，左自增和右自增运算的效果是相同的，都是先对“变量”加 1，然后将结果赋值回给“变量”。

例如，有一个整型变量 i，要对其进行纯粹的自增运算，使其值加 1，无论使用左自增或右自增都是可以的，也就是说：

```
i++;  
  
和  
  
++i
```

可以使用下面的这段程序验证一下：

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

首先声明定义一个整型变量 i，然后为其赋值 0，再对其进行纯粹的右自增运算，最后使用 printf()函数将 i 的值输出。与此类似，再将变量 i 的值重新赋值为 0，然后对其使用纯粹的左自增运算，最后使用 printf()函数将 i 的值输出。如果纯粹的左自增和右自增运算的效果是相同的，那么程序的输出结果应该是相同的。程序的输出都是 1，如图 3.62 所示。

2. 自增运算与其他运算的结合

当自增运算和其他运算结合组成复合表达式的时候，左自增和右自增运算就有差异了。

- 对于左自增运算，是先将“变量”的值加1，然后将结果赋值给“变量”，最后再把“变量”和其他运算结合进行运算。

```
#include<stdio.h>
int main()
{
    int i;
    i=0;
    i++;
    printf("%d\n",i);

    i=0;
    ++i;
    printf("%d\n",i);
    return 0;
}
```

图 3.62 纯粹的自增运算

- 对于右自增运算，是先把“变量”和其他运算结合进行运算，然后将“变量”值加1，最后将结果赋值给“变量”。

上面这两个运算的区别可以通过图 3.63 表示出来。

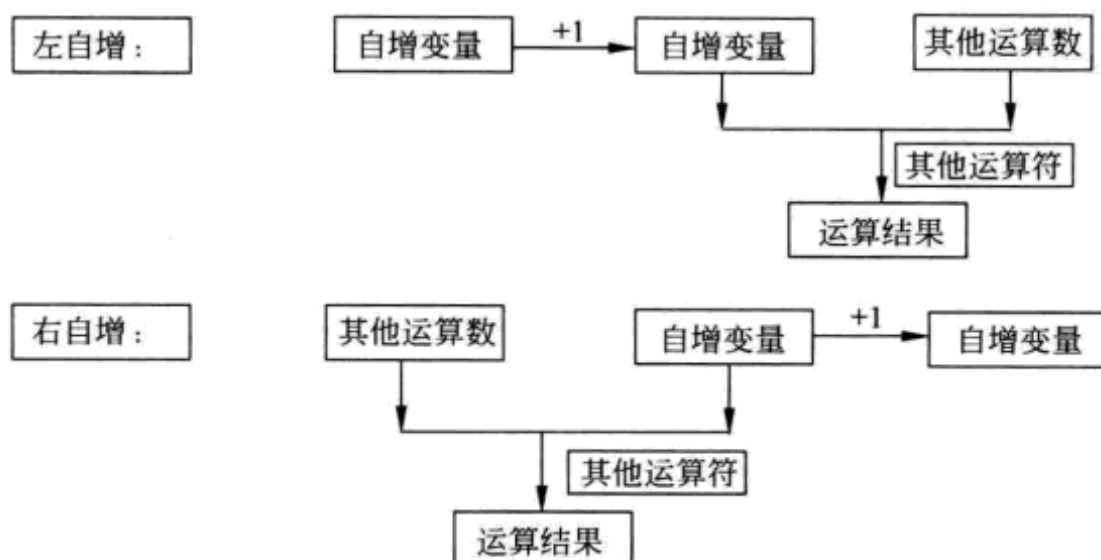


图 3.63 自增和右自增运算规则

假设我们有一个“自增变量” i ，开始的时候，其值为0，分别使用左自增和右自增运算对其加1，然后，将其和加法运算结合，可以按照图 3.63 所示，画出图 3.64 所示的计算过程图，冒号后面的数字为变量当前的值，从中可以看出得到上面的输出的原因。

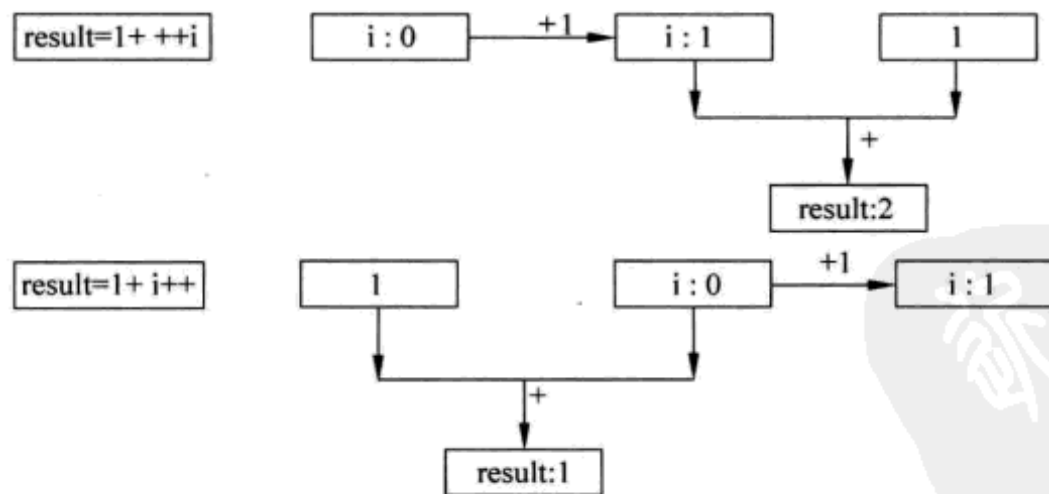


图 3.64 自增和加法运算结合解释

3.8 小 结

本章的内容比较庞大，主要是因为 C 语言中的数学运算不仅繁多而且灵活！不过大家

不要摸不着头脑，只要提纲挈领就可以了，抓住重点，细节的东西日后用到再查阅也不迟。

在本章中主要讲解了 C 语言的三类简单数学运算：赋值运算、基本数学运算和复合赋值运算，这些也是这一章的重点。另外，在讲解这三类简单数学运算的时候，穿插了数值的进制、类型转换和运算的优先级等知识点，这些可以算是本章的难点，掌握它们得需要些时间来思考和实践！本章学习的都是 C 语言中关于运算的知识，下一章将换个角度，来看一看 C 语言中关于逻辑控制的知识——程序结构。

3.9 习 题

【题目 1】 已经定义了一个整型变量 `num`，现在需要将整型数值 3 保存到其中，说一说常用的有哪些办法？

【分析】 要往变量中保存数据，在 C 语言中使用的是赋值运算。通常至少有三种方式可以完成赋值运算：（1）初始化；（2）一般赋值；（3）使用 `scanf()` 函数。现在要往整型变量 `num` 中保存数值 3，就可以使用这三种方式。

【核心代码】

```
int num = 3;

int num ;
num = 3;

int num ;
scanf("%d",&num); //从终端输入 3
```

【题目 2】 将十进制的 55 转换成二进制、八进制和十六进制表示各为多少？

【分析】 十进制转换成二进制比较容易，以二进制为桥梁，将二进制转换成八进制和十六进制，远比直接将十进制转换成八进制和十六进制容易得多。所以，可以按照图 3.65 所示的步骤很容易地将十进制的 55 转换成其他三种进制的表示。

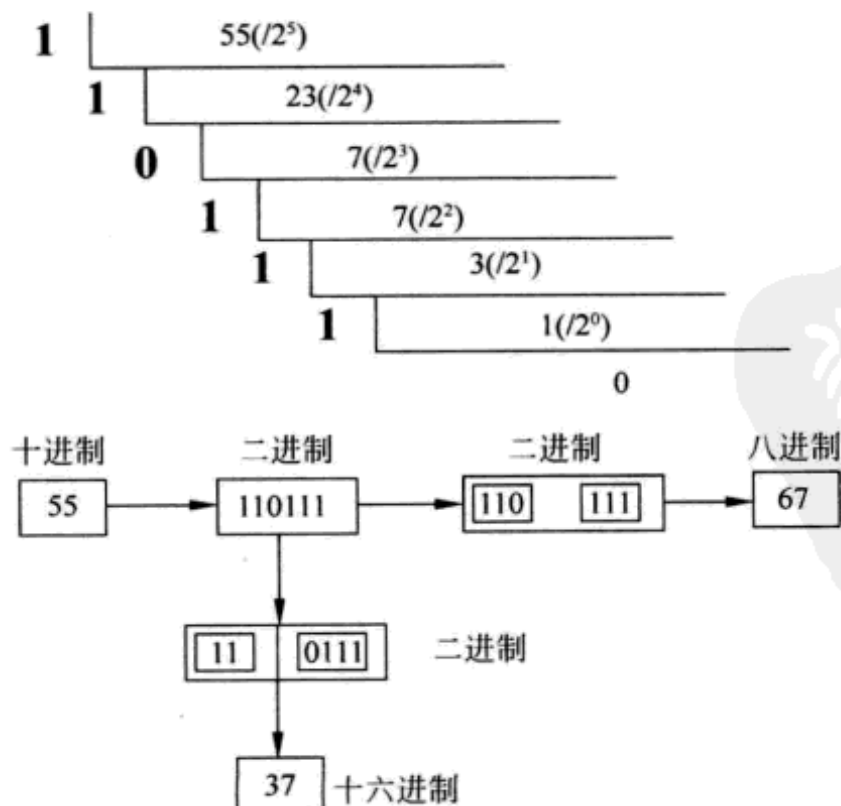


图 3.65 55 的各种进制转换

【题目 3】 写个程序验证一下，float 浮点型能不能表示 0.1999999，如果不可以，分析一下原因。

【分析】 要验证一个 float 可不可以表示 0.1999999，可以将 0.1999999 保存在一个 float 变量中，再使用 printf() 函数将其中保存的数值按照浮点型进行输出，如果输出是 0.1999999，则说明可以表示，否则说明不可以表示。要想解释为什么不能表示，找支笔，找张纸，将 0.1999999 表示成计算机可以识别的 float 规格化二进制表示，你就可以发现奥秘了！

【核心代码】

```
float a = 0.1999999;
printf("%f", a);
```

【题目 4】 如果要给字符变量 a 赋值换行符号，可以有哪些方法？

【分析】 换行符是一个特殊的字符，一般的字母和符号无法表示，因此只能使用转义字符表示了！对于转义字符，可以使用 ‘\n’ 的一般转义字符，也可以使用十进制、八进制和十六进制的 ASCII 码表示的转义字符表示 10、‘\012’ 和 ‘\x0A’。

【核心代码】

```
char a;
a = '\n';
```

```
char a;
a = 10;
```

```
char a;
a = '\012';
```

```
char a;
a = '\x0A';
```

【题目 5】 写个程序看一看，把浮点型数值 0.1 赋值给字符型变量 a 之后，a 将会是什么样的一个字符？

【分析】 要将浮点型数值赋值给字符型变量，由于是不同类型数值之间的运算，将会发生隐式类型转换，转换的规则是按照计算机中二进制表示直接赋值。可以通过这个题目看一看 C 语言中计算机表示差别最大的两种数据类型之间的类型转换有多么诡异！

【核心代码】

```
char a;
a = 0.1;
printf("%c", a);
```

【题目 6】 用基本数学运算计算出 2 的 30 次方的值，看看这个值有多少亿。

【分析】 如果你只对加减乘除很熟悉，把 2 乘 30 次就可以计算出 2 的 30 次方的值了，这确实是一个可行的方法，但是没有发挥计算机的潜力。C 语言中提供了左移运算，而且左移规则是对二进制左移的！所以，(1<<30) 就是 2 的 30 次方，比 30 个 2 相乘简单多了吧！

【核心代码】

```
int num;
num = (1<<30);
printf("%d", num);
```


【题目 7】 假设我们已经给整型变量 `i` 赋值 0 了，然后分别直接输出 `i++` 和 `++i` 的值，并解释结果为什么是这样的。

【分析】 右自加和左自加的最大区别在于自增和赋值的顺序，是先自增还是先赋值，而且这个区别主要体现在自增运算与其他运算结合的时候！直接输出 `i++` 和 `++i` 就是右自增和左自增与 `printf()` 函数结合的时候了，左右自增是有差别的。

【核心代码】

```
int i = 0;
printf("%d\n", i++);
printf("%d\n", ++i);
```

【题目 8】 有趣的验证：随便输入一个大于 1000 的奇数，你会发现一个有趣的现象，这个数的平方减一所得到的差值，永远是 8 的倍数。写段程序验证一下这个有趣的数学现象。

【分析】 我们可以使用 `scanf()` 函数将输入的奇数保存在一个整型变量中。使用基本乘法运算来计算这个数的平方，使用赋值运算将这个数的平方减一所得的结果保存在另一个变量中。使用求余运算计算所得的结果是不是 8 的倍数，如果求余的结果为 0，就是 8 的倍数，否则，就不是 8 的倍数。

【核心代码】

```
int num;
int result;
int mod_8;

scanf("%d", &num);
result = num*num-1;
mod_8 = result%8;
printf("%d\n", mod_8);
```

【题目 9】 写一段程序分别验证一下 `int`、`short`、`long` 这几种常见整型数据的取值范围。

【分析】 对于这几种不同的整型类型，它们的取值分为如表 3.9 所示。对于每一种整型数据类型，对它的边界及边界之外的两个数进行验证就可以确定它们的取值范围了。

例如，要验证 `short` 类型，就可以使用 $-2^{15}-1$ 、 $-2^{15}+1$ 、 -2^{15} 、 $2^{15}-1$ 、 $2^{15}+1$ 、 2^{15} 这 6 个数进行验证。我们将这些数保存在对应的变量中，然后使用 `printf()` 函数输出。如果输出正常，证明这个数是在该类型的取值范围中的，否则不在取值范围中。对于 2^{15} 可以使用移位运算来得到，即 $2^{15} = 1 \ll 15$ 。如果你觉得有必要，也可以使用同样的方式验证一下 `unsigned int`、`unsigned short`、`unsigned long`。

表 3.9 三种简单数据类型

类 型 名	表 示 符	表 示 范 围
整型	<code>int</code>	-2^{31} 到 $2^{31}-1$
短整型	<code>short</code>	-2^{15} 到 $2^{15}-1$
长整型	<code>long</code>	-2^{31} 到 $2^{31}-1$

【核心代码】

```
int          num_i;
short       num_s;
```



```
long          num_l;

num_i = -(1<<31);
printf("num_i = %d\n", num_i);
num_i = -(1<<31)-1;
printf("num_i = %d\n", num_i);
num_i = -(1<<31)+1;
printf("num_i = %d\n", num_i);

num_i = (1<<31);
printf("num_i = %d\n", num_i);
num_i = (1<<31)-1;
printf("num_i = %d\n", num_i);
num_i = (1<<31)+1;
printf("num_i = %d\n", num_i);

num_s = -(1<<15);
printf("num_s = %d\n", num_s);
num_s = -(1<<15)-1;
printf("num_s = %d\n", num_s);
num_s = -(1<<15)+1;
printf("num_s = %d\n", num_s);

num_s = (1<<15);
printf("num_s = %d\n", num_s);
num_s = (1<<15)-1;
printf("num_s = %d\n", num_s);
num_s = (1<<15)+1;
printf("num_s = %d\n", num_s);

num_l = -(1<<31);
printf("num_l = %d\n", num_l);
num_l = -(1<<31)-1;
printf("num_l = %d\n", num_l);
num_l = -(1<<31)+1;
printf("num_l = %d\n", num_l);

num_l = (1<<31);
printf("num_l = %d\n", num_l);
num_l = (1<<31)-1;
printf("num_l = %d\n", num_l);
num_l = (1<<31)+1;
printf("num_l = %d\n", num_l);
```



第4章 程序结构

在计算机中，程序是被一句一句执行的。C 语言中安排了三种结构来组织这样一句一句的程序，分别为顺序结构、分支结构和循环结构。使用这三种结构，可以完成任何复杂的程序，这三种结构也是写复杂 C 语言程序的基础。

4.1 语句和语句块

语句是 C 语言中各种结构的基本组成单位，也是学习写复杂程序的基础。本节就来介绍 C 语言中的简单语句和语句块。

4.1.1 简单语句

在 C 语言中，语句就是能够独立完成一个简单功能的基本构成单位。程序是由一条条的语句组成和执行的，每一条语句完成一个基本的功能。最基础的不可再分的 C 语言语句被称为简单语句。

之前，已经见过好多 C 语言的简单语句了。例如，“`radius = 3;`”就是一条简单语句。C 语言中常见的简单语句有 4 种：表达式语句、空语句、控制语句和函数调用语句。先来介绍两种简单的语句：表达式语句和空语句。其他的语句形式和作用将在讲解完相应的知识点之后再作介绍。

1. 表达式语句

表达式语句就是在表达式后面加分号构成的语句，形式如下：

表达式 ;

像这样的表示，就构成了一条表达式语句，可以在程序中完成一定的功能。例如，下面就是曾见过的表达式语句，只是当时并没有给出具体的名字而已。

```
radius = 1;  
10/5;  
~-2;
```

该程序中，第一个语句就是一个赋值表达式语句，它完成的功能就是把 1 赋值给变量 `radius`。第二个语句是一个除法运算表达式语句，它完成 10 除以 5 的运算。第三个语句是按位取反表达式语句，它计算 -2 按位取反的结果。

这些我们所熟悉的赋值运算、除法运算、按位取反运算所组成的赋值语句、除法运算语句、按位取反语句，都属于表达式语句。

2. 空语句

空语句在 C 语言中很少用到，它的作用就是什么都不干，让计算机空转，它的主要目的就和我们军训的时候原地踏步是一样的，是为了暂时等待其他的東西先完成。

空语句的形式为一个简单的“;”，其他什么都没有：

```
;
```

一个单独的分号就是告诉计算机，先等一等，在此暂留一步，然后再往下执行其他的语句。

4.1.2 语句块

语句块有时也被称为复合语句。之所以叫它复合语句，是因为它是由其他简单语句组成的，看着像一块语句，所以也叫语句块。

1. 语句块的形式

C 语言中的语句块是由一对大括号和其中包含的若干条语句组成的，形式如下所示：

```
{
    语句 1;
    语句 2;
    ...
    语句 n;
}
```

若要写一个包含几个表达式语句和空语句的语句块，就可以用下面的形式：

```
{
    radius = 1;
    ;
    10/5;
    ~~2;
}
```

这是一个语句块的例子，其中包含四个已经了解的语句，依次为赋值表达式语句、空语句、除法运算表达式语句和按位取反表达式语句。

2. 语句块的嵌套

C 语言中的语句块中是可以包含语句块的，叫做语句块嵌套。语句块中的语句块被当作一个简单的语句来处理。例如，下面是一个语句块嵌套的包含表达式语句和空语句的例子。

```
{
    radius = 1;
    ;
    {
        10/5;
        ~~2;
    }
}
```

这个程序中的语句块稍微有点复杂，语句“10/5;”和“~2;”这两个语句被大括号括起来组成了一个语句块，这个语句块又和“radius = 1;”、“;”一起被另一个大括号括了起来，组成了一个更复杂的语句块。

4.2 变量的作用域

在C语言中，变量的作用域就是可以使用这个变量的范围。每个变量都是有作用域的，如果变量在作用域之外，我们就不能使用它了，否则就会出错。这就好比，你的孩子在家的時候，你可以喊“Baby”，他会马上跑过来，但是你的孩子去外婆家了，你再喊“Baby”，就没人答应了。

在C语言中，常见的变量都是在语句块里声明定义的，这些变量被称为局部变量。局部变量在C语言中的使用要遵循下面三个规则。

4.2.1 局部变量的声明定义位置规则

C语言规定，变量的声明定义必须位于该语句块中的所有语句之前。也就是说一个语句块的最合理的形式为：最前面全是局部变量的声明定义，后边全是语句，就像下面这样：

```
{
    变量声明定义 1;
    变量声明定义 2;
    ...
    变量声明定义 n;

    语句（或语句块） 1;
    语句（或语句块） 2;
    ...
    语句（或语句块） n;
}
```

在上面的格式中，变量的声明定义的次序可以无论先后。但是，变量的声明定义绝对不能位于语句之后，否则开发工具将会告诉你编译出错。

例如，下面的这个程序就违反了这个规则，导致编译出错。

```
#include<stdio.h>

int main()
{
    {
        int radius;
        radius =1;

        int result;
        result = 3*radius;
    }
    return 0;
}
```

在程序进行编译的时候出现编译错误信息，如图4.1所示。这是由于在语句块中，变

量 `result` 的声明定义位于赋值语句之后导致的。只要将 `result` 整型变量的声明定义放到赋值语句 `radius=1` 之前，就满足上面的规则了，也就不会出现这样的错误了，不信，可以试一试！

```
1>正在编译...
1>t.c
1>g:\c语言的书\c语言的书\test\t.c(9) : error C2143: syntax error : missing ';' before 'type'
1>g:\c语言的书\c语言的书\test\t.c(10) : error C2065: 'result' : undeclared identifier
1>生成日志保存在 "file:///g:/c语言的书/c语言的书/test/Debug/BuildLog.htm"
1>test - 2 个错误, 0 个警告
===== 全部重新生成: 0 已成功, 1 已失败, 0 已跳过 =====
```

图 4.1 变量声明定义位于语句中的错误

4.2.2 局部变量的作用域规则

正确地在语句块中声明定义局部变量以后，来看看关于局部变量的作用域规则：局部变量的作用域为它所在的语句块，也就是局部变量只有在它声明定义的语句块中是可以使用的，出了这个语句块范围，再使用这个变量将会出错。

从语句块的构成来看，局部变量的作用域就是它所在的一对大括号之间，出了这个大括号再使用这个变量就会出现错误。例如，下面的代码就犯了这个错误。

```
#include<stdio.h>

int main()
{
    {
        int radius;
        int result;

        radius =1;
    }
    result = 3*radius;
    return 0;
}
```

在程序中，在一个语句块中声明定义了两个变量。但是，在语句块外面的一个赋值运算语句“`result=3*radius;`”中使用了这两个变量，结果导致了错误，如图 4.2 所示。“`radius`”和“`result`”是未声明的标识符。

```
1>正在编译...
1>t.c
1>g:\c语言的书\c语言的书\test\t.c(11) : error C2065: 'result' : undeclared identifier
1>g:\c语言的书\c语言的书\test\t.c(11) : error C2065: 'radius' : undeclared identifier
1>生成日志保存在 "file:///g:/c语言的书/c语言的书/test/Debug/BuildLog.htm"
1>test - 2 个错误, 0 个警告
===== 全部重新生成: 0 已成功, 1 已失败, 0 已跳过 =====
```

图 4.2 语句块外使用变量导致的错误

4.2.3 嵌套语句块的同名变量作用域规则

我们知道一个局部变量的作用域为它所在的语句块之中。对于一个嵌套的语句块，如果外面的语句块和里面的语句块都有一个同样名字的局部变量，那该怎么办？就像下面所示：


```

{
    A 变量声明定义;
    {
        A 变量声明定义;
        A 的操作语句;
    }
}

```

嵌套语句块内外都有 A 的声明定义，那么里面 A 的操作语句到底是对哪个 A 进行操作的呢？

C 语言使用的规则为：就近原则。也就是说代码“A 的操作语句”，使用的是里面的语句块的变量 A。这就好像，家里有两个孩子，一个在客厅，一个在卧室，你在卧室喊了声“Baby”，肯定是卧室的孩子会向你跑来。

可以使用下面的程序来验证一下这个规则：

```

#include<stdio.h>

int main()
{
    {
        int i;
        i = 1;
        {
            int i;
            i = 10;

            i--;
            printf("%d\n",i);
        }
        i++;
        printf("%d\n",i);
    }
    return 0;
}

```

程序中，有一个嵌套的语句块。内外的语句块中都声明定义了一个整型变量 i，外面的 i 赋值为 1，并对其进行自增运算，里面的 i 赋值为 10，并对其进行自减运算，然后分别使用 printf() 函数输出结果。程序的结果为 9 和 2，如图 4.3 所示，符合上面所述的“就近原则”。

```

#include<stdio.h>

int main()
{
    {
        int i;
        i = 1;
        {
            int i;
            i = 10;

            i--;
            printf("%d\n",i);
        }
        i++;
        printf("%d\n",i);
    }
    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
9
2
请按任意键继续. . .

```

图 4.3 就近原则验证

4.3 最常见的语句执行结构——顺序结构

在 C 语言中,语句的组织是可以按照不同的方式进行的,对于每一种方式组织的语句,执行顺序都是不同的。顺序结构是最简单的组织方式,它是按照程序中 C 语言语句的顺序一句一句地执行的。

1. 顺序结构的一般形式

顺序结构在 C 语言程序中的形式如下所示,一条语句接着一语句往下写:

```
语句 1;  
语句 2 ;  
...  
语句 n-1;  
语句 n;
```

假设,程序中有 n 条语句,并且是按照顺序结构组织的,那么,程序就会从第一条语句开始一直顺序地执行到第 n 条语句,如图 4.4 所示。

2. 顺序结构的例子

看看实际的顺序结构的例子吧,下面的程序有好几条语句,它们都是按照顺序结构组织的。

```
#include<stdio.h>  
  
int main()  
{  
    int i = 0;  
    printf("%d\n",i);  
    i++;  
    printf("%d\n",i);  
  
    i = 5;  
    printf("%d\n",i);  
    i++;  
    printf("%d\n",i);  
  
    return 0;  
}
```

程序中写了两个赋值语句和两个 `i++` 自增表达式语句。它们是按照顺序结构的方式组织的,一条在另一条之后,程序的输出如图 4.5 所示。从程序输出中可以看出,顺序结构组织的语句确实是一条一条地执行的。

程序先输出了一个 0,表示初始化赋值语句已经执行。接下来程序输出为 1,表示 `i` 自增运算已经完成。再接着程序输出为 5,表示 `i` 赋值为 5 语句执行成功。最后程序输出为 6,表示 `i` 再一次自增成功。这正是顺序结构程序执行的顺序。

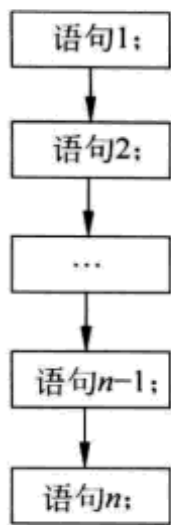


图 4.4 顺序结构图示

```
#include<stdio.h>
int main()
{
    int i = 0;
    printf("%d\n",i);
    i++;
    printf("%d\n",i);

    i = 5;
    printf("%d\n",i);
    i++;
    printf("%d\n",i);

    return 0;
}
```

图 4.5 顺序结构程序验证

4.4 判断结构

顺序结构是按照语句的先后顺序来依次执行的，简单有效。但是，当我们需要根据不同的条件来有选择地执行某些语句时，顺序结构就不能满足要求了。这个时候就需要判断结构出马了！

4.4.1 判断的基础——逻辑真假

根据条件有选择地执行某些语句，是判断语句的核心。所以，先来看看这里所谓的“条件”。在计算机中，判断的条件也被称为逻辑条件，计算机就是根据这个条件来进行逻辑判断的——决定下一步该做什么。计算机的逻辑判断很简单，它只认识两个条件：真和假。在计算机中，真和假也被称为逻辑真和逻辑假。

1. 逻辑真假

逻辑真和逻辑假，只是两个对立的逻辑概念，不是真就是假，就像我们所说的对错一样。但是，计算机不像现实世界那样复杂，有的事情从不同方面去想可能是对的也可能是错的。在计算机的世界中，非真即假，绝没有中间态。

我们习惯用“1”表示真，用“0”表示假。这里的“1”和“0”只是一个符号而已，表示对立的两个方面。在 C 语言中对于所有的逻辑条件，非“1”即“0”，非“0”即“1”，只有这两个状态或者逻辑条件。

2. 简单数据类型的逻辑真假

在 C 语言中，所有的数据都是可以作为逻辑条件来进行逻辑判断的。我们知道每一种数据类型都可以表示很多个值，但是逻辑条件只有两个，“1”和“0”，那么到底哪些值对应的是“1”，哪些值对应的是“0”呢？

在 C 语言中，有这样一个规定：数值为 0 的数据表示的是逻辑条件的“0”（假），数值不是 0 的值表示的是逻辑条件的“1”（真）。下面，通过表格的形式列出整型、浮点型、字符型表示的所有数值，哪些代表真，哪些代表假，如表 4.1 所示。

表 4.1 各种数据的真假关系

数据类型	真 ("1")	假 ("0")	说 明
整型	-32768 到 32767 中除了 0 的任意整数	0	此处以 int 为例，其他整型类似。0 为假，其他为真
浮点型	不为 0 的小数	0.0	对于 float 和 double 浮点型都适用
字符型	ASCII 码不为 0 的字符	NULL	ASCII 表规定的字符

通过表 4.1，可以总结出这样一个规律：对于不同数据类型的数据，只要其二进制表示中所有的位为 0，这个数据表示的就是假。否则，只要有一位不是 0，它表示的就是真。

3. 表达式的逻辑真假

在 C 语言中，除了各种数值可以作为逻辑判断的条件以外，表达式也可以作为逻辑判断的条件。前面讲解了两种表达式：赋值表达式和基本数学运算表达式，它们都可以作为逻辑判断的条件。至于它们如何来表示真假，是根据表达式自身的值和类型确定的。两种表达式的值和类型有下面的规则。

- ❑ 赋值表达式，赋值表达式的值为赋值运算所赋的值，其类型为赋值表达式等号左边变量的类型。
- ❑ 基本数学运算表达式，基本数学运算表达式的值为数学运算计算完成之后的值，其类型为数学运算经过隐式和显式类型转换最终得到的类型。

确定了表达式的值和类型以后，就可以通过表 4.1 来确定表达式代表的到底是真还是假了。

4.4.2 基础的判断——关系运算

我们知道数学运算表达式可以作为逻辑判断条件，赋值运算表达式可以，基本数学运算表达式也可以。这节看看另外一种数学运算——关系运算，它的表达式也可以作为逻辑判断的条件，而且专门作为逻辑判断的条件。

1. 关系运算

关系运算，顾名思义，就是判断两个东西的关系的。如果关系正确，那么运算结果为真，如果关系不正确，那么运算结果为假。由此可见，关系运算的结果就是真和假，因此它专门作为逻辑判断的条件。

表 4.2 列出了 C 语言支持的所有的逻辑运算，以及它们的优先级、表达式和结合性。

表 4.2 C语言中的关系运算

优先级	关系运算符	名称或含义	表达式形式	结 合 方 向	说 明
6	>	大于	操作数>操作数	左到右	二元运算符
	>=	大于等于	操作数>=操作数		二元运算符
	<	小于	操作数<操作数		二元运算符
	<=	小于等于	操作数<=操作数		二元运算符
7	==	等于	操作数==操作数	左到右	二元运算符
	!=	不等于	操作数!= 操作数		二元运算符

从表 4.2 中可以看出，关系运算的表达式形式如下所示：

操作数 1 关系运算符 操作数 2

关系运算表达式中的“操作数 1”和“操作数 2”，就是要进行关系判断的两个东西。它们既可以是常量，也可是变量，还可以是前面见过的表达式。“关系运算符”就是表 4.2 所示的 6 种关系运算的运算符号。整个表达式的结果根据实际的“操作数 1”和“操作数 2”而定，若“操作数 1”和“操作数 2”符合“关系运算符”表示的关系，则表达式的结果为真，否则为假。

像下面的关系运算表达式的值就为真：

1 >= 0 ; 2 < 3 ; 7 == 7 ; 8 != 7

因为它们的两个操作数的值符合关系运算符所表示的关系。反之，下面的例子中，表达式的值为假：

1 <= 0 ; 2 > 3 ; 7 != 7 ; 8 == 7

2. 真和假的值

我们知道了关系运算表达式的值为真和假，那么到底真和假表示的值是什么呢？可以使用 printf() 函数来看一下：

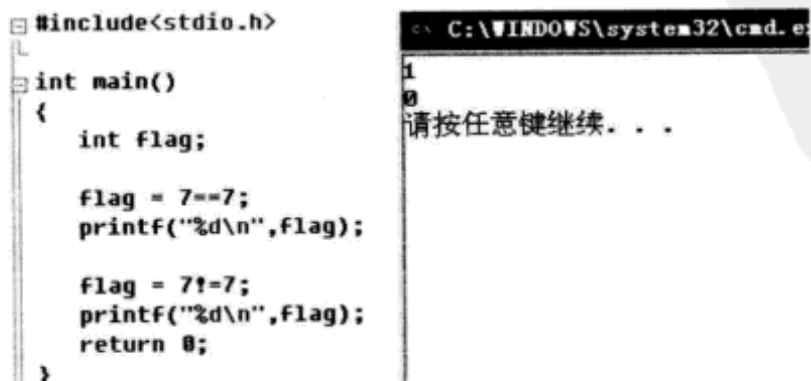
```
#include<stdio.h>

int main()
{
    int flag;

    flag = 7==7;
    printf("%d\n",flag);

    flag = 7!=7;
    printf("%d\n",flag);
    return 0;
}
```

在程序中，声明定义了一个整型变量 flag，然后分别把一个值为真的表达式和一个值为假的表达式赋值给 flag，最后分别使用 printf() 按照整型输出。程序的结果如图 4.6 所示，“真”的值为 1，“假”的值为 0。



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

    flag = 7==7;
    printf("%d\n",flag);

    flag = 7!=7;
    printf("%d\n",flag);
    return 0;
}
```

C:\WINDOWS\system32\cmd.e
1
0
请按任意键继续...

图 4.6 真和假的值

3. 逻辑真假和数值的相互转换

前面通过各种数据类型的值和表达式的逻辑真假，了解了数值和逻辑真假的对应关系。本节又通过关系运算表达式知道了逻辑真假的值。接下来总结一下，逻辑真假和数值之间的相互对应和转换关系。

以整型为例，其他类型的数据参见表 4.1。在图 4.7 中，展示了整型值与逻辑真假的对应关系。其中逻辑的“真”和“假”到整型的数值的箭头，表示的是逻辑真假到整型值的转换关系。整型整个表示范围的值到逻辑“真”“假”的箭头，表示的是整型数值到逻辑真假的转换关系。

大家看了图 4.7 是不是觉得有点奇怪？-32738 是“真”，为什么“真”不是-32768 呢？其实，主要的原因就是逻辑真假只有两个状态，但是整型数值却有 2 的 32 次方个值。从 -32768 到“真”，这是确定的，一一对应，在程序中使用的时候是不会出错的，因为计算机知道-32768 肯定是“真”，不可能是其他值！但是，反过来，如果把“真”的值设为除 0 以外所有的整型数值，就有问题了。因为这是一个一对多的关系，程序无法确定关系表达式的值为 1，还是-32768，也就无法继续执行了。所以 C 语言规定“真”的值，只能为 1。

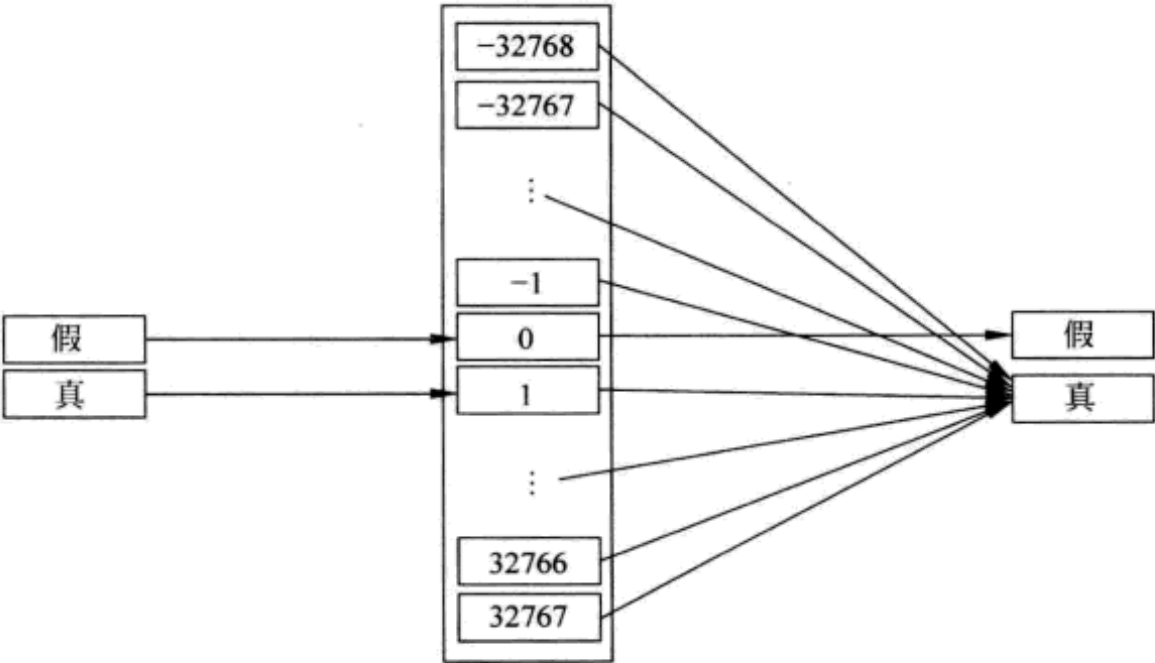


图 4.7 整型值与逻辑真假的相互转换

4.4.3 复杂的判断——逻辑运算

前面已经介绍过，关系运算的结果为逻辑的真和假，可以很好地作为逻辑判断的条件。但是，关系运算有一个缺点：只能判断一个条件是不是满足。现实生活中，往往不是根据一个条件进行判断，而是需要多个条件进行判断。例如，早上早起，带上钱，才可能去餐厅吃到早餐，这两个条件缺一不可。在 C 语言中，也是如此。有的时候，需要用到几个逻辑条件满足才能进行判断，这就需要用到本节将要讲到的逻辑运算了。

1. 逻辑运算

和关系运算类似，逻辑运算也是用来判断某些东西或者操作数是不是满足某种逻辑关

系的。如果满足，则逻辑运算的结果为真，否则，逻辑运算的结果为假。

与关系运算不同的是关系运算一般一次只能进行一个判断，而逻辑运算可以同时进行一个或者几个判断。一般进行判断的时候会有这样一些情况：“不要…”，“同时都要”，“只要一个就行”。

例如，小孩子不撒谎，妈妈就给糖吃。这里的“不撒谎”就是一个条件。再例如，当你大学毕业的时候，只有拿到毕业证和学位证，才算真正完成了学业，这里同时拿到毕业证和学位证就是完成学业的条件。另外，再例如，老板发话了，今天干完活或者等到晚上9点才能下班，这里的“干完活”，“等到9点”只要一个条件满足就可以回家了。

这三种判断条件的组合囊括了所有的条件组合的基本形式。C语言也正好提供了这三种条件组合对应的逻辑运算：非、与、或运算。分别对应上面介绍的“不要…”，“同时都要”，“只要一个就行”的判断条件的组合。

表4.3列出了C语言中的三种逻辑运算，包括它们的优先级、运算符、表达式形式、结合方向等。使用表4.3中的信息，就可以写出一般的逻辑运算的表达式了。

表 4.3 逻辑运算

优先级	运算符	名称或含义	表达式形式	结 合 方 向	说 明
2	!	逻辑非运算	!操作数	右到左	一元运算符
11	&&	逻辑与运算	操作数&&操作数	左到右	二元运算符
12		逻辑或运算	操作数 操作数	左到右	二元运算符

表4.3中，逻辑非运算是一个一元运算，逻辑与和逻辑或都是二元运算，可以将这三种运算归为两类：一元运算和二元运算，如下所示：

!操作数

和

操作数1 与或操作符 操作数2

对于两种运算表达式，其中的操作数既可以是常量、变量，也可以是表达式（赋值表达式、基本数学运算表达式，关系运算表达式，甚至是本节所讲的逻辑运算表达式都是可以的）。对于与或操作符，就是与运算的操作符“&&”和或运算的操作符“||”中的一个了。

例如，!5、8&&1、0||0、(0+1)&&3、(0>1)&&3、(0||0)&&3等都是合法的逻辑运算表达式。

2. 逻辑运算的值

表4.3所述的三种逻辑运算的结果都是逻辑的“真”或“假”，比起关系运算，逻辑运算的规则稍微有点复杂。接下来，就来看看这三种运算的规则。

- 把逻辑运算的“操作数”（可以是常量、变量或表达式）按照表4.4节所讲的各种对应关系，变换成逻辑的“真”和“假”。
- 按照表4.4所示的规则，得出逻辑运算表达式的结果：“真”或“假”。表4.4中“操作数的逻辑真假”一行中的ABCD为操作数的逻辑真假值的不同组合，“结果

的逻辑真假”中的 ABCD 表示的是对应于“操作数的逻辑真假”列的不同情况时，逻辑运算表达式的逻辑真假。

- ❑ 结果为“真”的逻辑运算表达式的值为 1，结果为“假”的逻辑运算表达式的结果为 0。

表 4.4 逻辑运算的结果

运 算 名 称	表 达 式	操作数的逻辑真假	结果的逻辑真假
逻辑非运算	!操作数	A:真 B:假	A:假 B:真
逻辑与运算	操作数 1 && 操作数 2	A:真真 B:真假 C:假真 D:假假	A:真 B:假 C:假 D:假
逻辑或运算	操作数 1 操作数 2	A:真真 B:真假 C:假真 D:假假	A:真 B:真 C:真 D:假

来看一个简单的例子：!5。按照上面的规则，

(1) 整型常量 5 的逻辑真假值为：“真”。

(2) !5 是一个逻辑非运算表达式，根据表 4.4 所示，对应的是 A 情况。所以，逻辑运算表达式!5 的逻辑真假值为：假。

(3) 结果为假的逻辑表达式的值为 0，所以逻辑非表达式!5 的值为 0。

下面的这段程序可以验证这个结果，程序中声明一个整型变量 test，然后给其赋值为!5，最后使用 printf()函数输出运算结果。

```
#include<stdio.h>

int main()
{
    int test;

    test = !5;
    printf("%d\n",test);

    return 0;
}
```

程序的输出结果为 0，如图 4.8 所示。

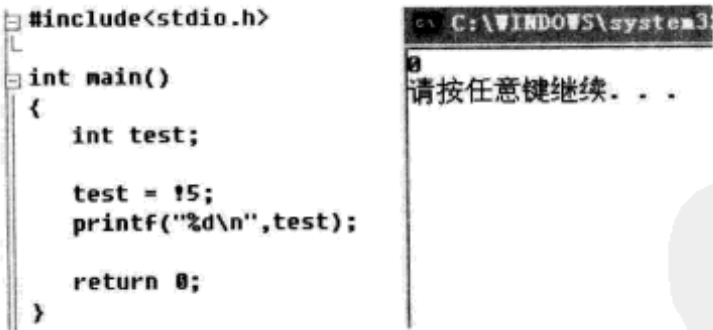


图 4.8 逻辑非运算结果

3. 逻辑运算的优先级

和其他运算一样，逻辑运算也可以和其他运算一起组成更为复杂的复合运算。当进行包含逻辑运算的复合运算时，需要注意各种运算的优先级和结合性。接下来看看逻辑运算的优先级和结合性，以及其应用。

表 4.5 列出了现在所见到的所有运算的优先级和结合性。根据表 4.5 所示信息，就可以进行包含各种运算的复杂运算了。

表 4.5 各种运算优先级和结合性

运算	含 义	优先级	结合方向	运算	含 义	优先级	结合方向
-	负号	2	右到左	()	强制类型转换	2	右到左
++	自增	2	右到左	--	自减	2	右到左
!	逻辑非	2	右到左	~	按位取反	2	右到左
/	除	3	左到右	*	乘	3	左到右
%	求余	3	左到右	+	加	4	左到右
-	减	4	左到右	<<	左移	5	左到右
>>	右移	5	左到右	>	大于	6	左到右
>=	大于等于	6	左到右	<	小于	6	左到右
==	等于	7	左到右	!=	不等于	7	左到右
&	按位与	8	左到右	^	按位异或	9	左到右
	按位或	10	左到右	&&	逻辑与	11	左到右
	逻辑或	12	左到右	=	赋值	14	右到左
/=	除后赋值	14	右到左	*=	乘后赋值	14	右到左
%=	求余后赋值	14	右到左	+=	加后赋值	14	右到左
-=	减后赋值	14	右到左	<<=	左移后赋值	14	右到左
>>=	右移后赋值	14	右到左	&=	按位与后赋值	14	右到左
^=	按位异或后赋值	14	右到左	=	按位或后赋值	14	右到左

这里只关注本节所讲的三个逻辑运算：逻辑非（2）、逻辑与（11）、逻辑或（12）。由于这几个运算优先级的跨度比较大，所以使用的时候一定要注意。

还是看一个例子吧。`!5+2&&3>=1||4`，这个例子包含了基本数学运算、关系运算和本节讲的逻辑运算，所以有必要好好分析一下。从高优先级到低优先级，依次计算过程如图 4.9 所示，图中①②③④…为计算步骤。从图 4.9 中可以看出，关系运算和逻辑运算的结果要么为 0，要么为 1，这也证明了真和假的值为 1 和 0。

可以使用下面这段程序来验证一下，声明一个整型变量 `test`，然后给其赋值为 `!5+2&&3>=1||4` 表达式，最后使用 `printf()` 输出 `test` 的值。

```
#include<stdio.h>

int main()
{
    int test;

    test = !5+2&&3>=1||4;
    printf("%d\n",test);
    return 0;
}
```

程序的输出结果为 1，如图 4.10 所示。

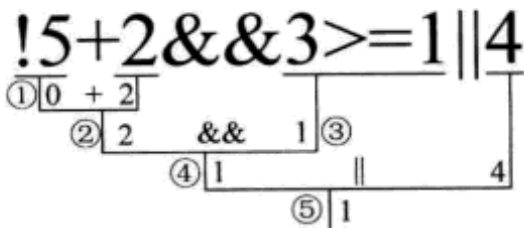


图 4.9 逻辑运算优先级

```
#include<stdio.h>

int main()
{
    int test;

    test = !5+2&&3>=1||4;
    printf("%d\n",test);
    return 0;
}
```

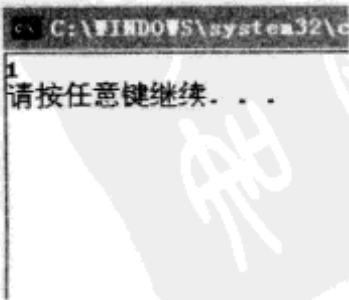


图 4.10 逻辑运算优先级程序结果

4.5 if 判断结构

前面所讲的“真”、“假”、“关系运算”和“逻辑运算”都是判断的条件。判断结构就是根据这些判断条件来有选择地决定哪些语句是要执行的，哪些语句是不需要执行的。本节就来看看使用 if 关键字来组织的判断结构。

4.5.1 基本 if 结构

基本的 if 结构是判断结构中最简单的一种。它根据条件的真假来判断之后的语句到底是执行还是不执行，条件为真就执行，为假就不执行。图 4.11 表示 if 结构的逻辑判断和语句执行的关系，该图被称为流程图。其中，“条件”就是之前所讲的具有真假值的常量、变量、赋值表达式、基本数学运算表达式、关系运算表达式和逻辑运算表达式。当条件的值为“真”的时候，程序执行语句，否则，程序不去执行语句。

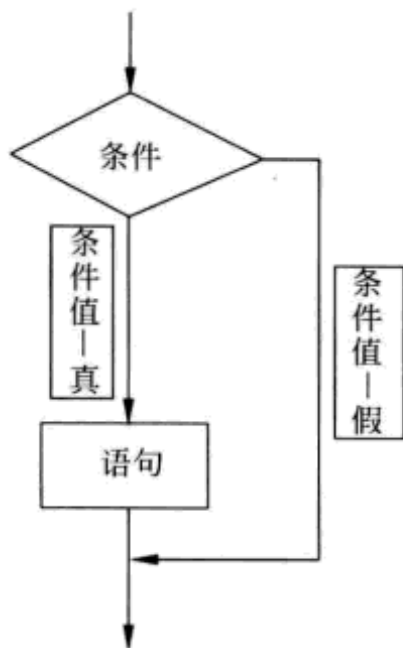


图 4.11 if 语句的判断逻辑

说明：图 4.11 中，菱形是专门用来表示逻辑判断的条件的，矩形表示可执行的语句，箭头表示程序执行的流程，也就是先后顺序。箭头旁边的标识表示执行箭头所示的流程时，条件要满足的值。

if 结构的 C 语言表示如下所示：

```
if(条件)
{
    语句;
}
```

程序中的条件对应的就是图 4.11 中的“条件”，大括号中的语句就是图 4.11 中所示的“语句”。大括号中的语句可以不止一条，只要在大括号中，当条件的值为真的时候都会被执行。

举个有意思的程序来说明 if 结构的使用方法。例如，小孩子撒谎，妈妈就不给糖吃。这个程序和以前见到的很不一样吧，但是也是可以用 C 语言实现的。

用一个整型变量 `is_lie` 来表示判断的条件，另外有一个整型变量 `candy` 表示糖果的个数。开始的时候糖果的个数有 10 个，是妈妈给孩子准备的。但是，妈妈给孩子糖果的时候先看孩子今天的表现——有没有撒谎。如果撒谎了，也就是 `is_lie` 的值对应的是逻辑的真，不给孩子糖吃，把 `candy` 的值变为 0。如果没撒谎，也就是 `is_lie` 的值对应的是逻辑的假，给孩子 10 个糖吃，不改变 `candy` 的值。

假设，今天孩子撒谎了，妈妈没给糖吃。就可以使用下面这段程序来表示：

```
#include<stdio.h>

int main()
{
```



```
int is_lie = 1;
int candy = 10;

if(is_lie)
{
    candy = 0;
}
printf("the number of candy is : %d\n",candy);

return 0;
}
```

程序中，给整型变量 `is_lie` 赋值为 1，表示孩子说谎了。因为 1 对应的是逻辑的真，其实，也可以有任何除 0 以外的值，参见表 4.1。另外，将整型变量 `candy` 初始化为 10，表示开始的时候有 10 个糖。在 `if` 结构中，按照图 4.11 所示，条件 `is_lie` 为真，那么执行 `candy=0` 语句，不给孩子糖吃。最后用 `printf()` 函数输出今天孩子可以拿到几个糖。程序的输出结果为 0，如图 4.12 所示，表示孩子没吃到糖。

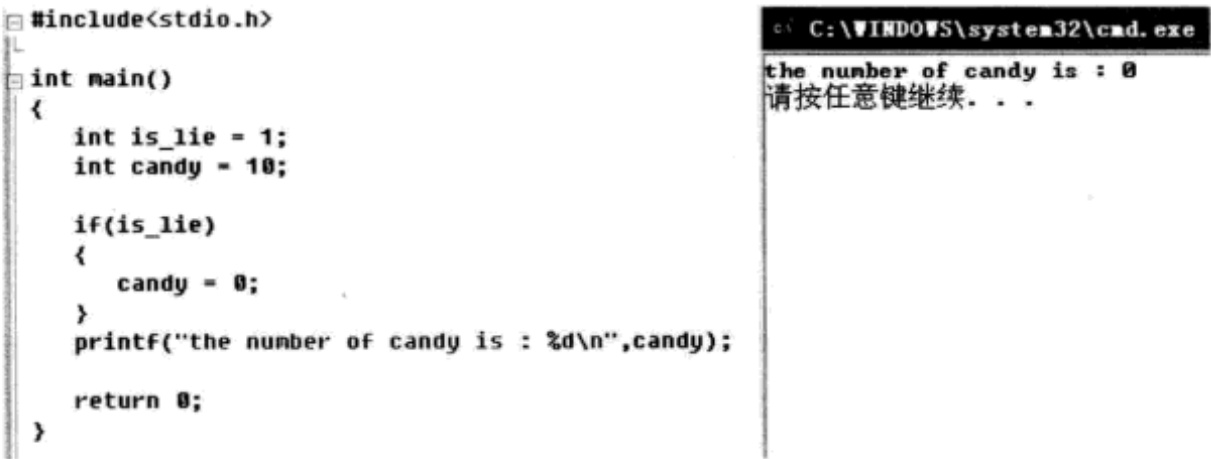


图 4.12 if 结构使用

4.5.2 if-else 结构

`if-else` 结构比 `if` 结构功能稍微强大一些。它可以在语句之间有选择地执行。条件满足则执行一些语句，条件不满足则执行另外一些语句，如图 4.13 所示。

与 `if` 结构类似，图 4.13 中的条件可以是任何具有真假值的表示：常量、变量、赋值表达式、基本数学运算表达式、关系运算表达式和逻辑运算表达式。当条件的值为“真”的时候，程序选择执行语句 1，否则，程序放弃执行语句 1 而去执行语句 2。

`if-else` 结构的 C 语言表示如下所示：

```
if(条件)
{
    语句 1;
}
else
{
    语句 2;
}
```

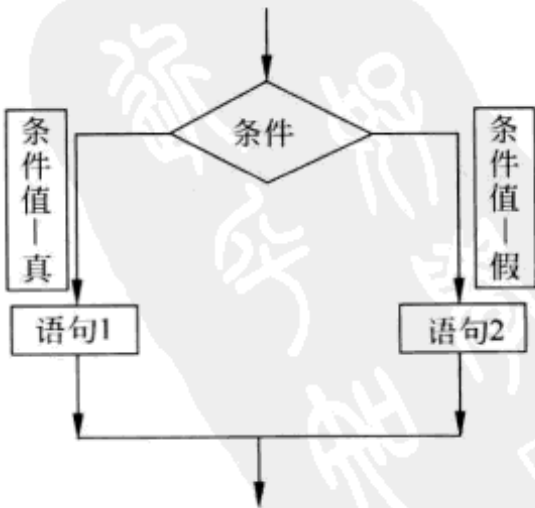


图 4.13 if-else 结构执行流程图

程序中的条件对应的就是图 4.13 中的“条件”，大括

号中的语句就是图 4.13 中所示的“语句 1”和“语句 2”。大括号中的“语句 1”和“语句 2”可以不止一条，当相应的条件满足的时候，同一个大括号中的语句都会被执行。

还是以“小孩子撒谎，妈妈就不给糖吃”为例来说明 C 语言中的 if-else 语句的使用方法。这次，妈妈赏罚分明，当孩子没撒谎时，多给一倍的糖吃；要是撒谎了，不是不给糖吃，而是只给一半糖吃。

还是用一个整型变量 `is_lie` 来表示判断的条件，用一个整型变量 `candy` 表示糖果的个数，开始的时候糖果的个数有 10 个。妈妈还是看孩子的表现来发给糖果吃。如果今天孩子撒谎了，也就是 `is_lie` 的值对应的是逻辑的真，则糖果数减半；如果没有撒谎，也就是 `is_lie` 的值对应的是逻辑的假，则糖果数加倍。

假设，今天孩子没撒谎，妈妈加倍奖励。就可以使用下面这段程序来表示：

```
#include<stdio.h>

int main()
{
    int is_lie = 0;
    int candy = 10;

    if(is_lie)
    {
        candy /= 2;
    }
    else
    {
        candy *= 2;
    }
    printf("the number of candy is : %d\n",candy);

    return 0;
}
```

程序中，将 `is_lie` 的值初始化为 0，对应逻辑假，表示孩子没撒谎。在 if-else 结构中，如果 `is_lie` 的值为真，我们使用除后赋值复合运算给糖果数 `candy` 减半。如果 `is_lie` 的值为假，我们使用乘后赋值复合运算给糖果数 `candy` 加倍。最后使用 `printf()` 函数输出孩子可以得到的糖果数。可以看到程序的输出糖果数为 20，如图 4.14 所示，不撒谎真好！

```
#include<stdio.h>

int main()
{
    int is_lie = 0;
    int candy = 10;

    if(is_lie)
    {
        candy /= 2;
    }
    else
    {
        candy *= 2;
    }
    printf("the number of candy is : %d\n",candy);

    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
the number of candy is : 20
请按任意键继续...
```

图 4.14 if-else 结构的使用方法

4.5.3 另类的条件判断——?运算符的使用

看完了 if 判断结构和 if-else 判断结构，再来看一个很强大的运算符——? 运算符，也被称为条件判断运算符。之所以称为条件判断运算符，是因为它可以完成 if-else 判断结构的条件判断功能。由条件判断符组成的运算称为条件判断运算，条件判断运算形成的表达式称为条件判断表达式，条件判断表达式构成的语句称为条件判断语句。

1. 条件判断运算的含义和表达式

本节就来看看条件判断语句，它与 if-else 判断结构有着异曲同工之妙，而且形式上比 if-else 更简洁。条件判断运算的主要功能就是完成类似于 if-else 判断结构的逻辑判断，与 if-else 不同的是它是一种新的运算。这个运算的形式和之前见到的都不同，它是一个三元运算，也是 C 语言中唯一一个三元运算。其形式如下所示：

条件 ? 操作数 1 : 操作数 2

表达式中的“条件”就是具有真假值的常量、变量、表达式等。像前面讲的关系运算表达式、逻辑运算表达式都可以作为“条件”。“操作数 1”和“操作数 2”可以是常量、变量和表达式，只要具有值就可以。

2. 条件判断运算的执行逻辑

既然条件判断运算完成了类似于 if-else 的运算，那么它到底是如何进行判断的，又是如何进行有条件地选择执行语句的呢？下面就让我们来看看判断运算的执行逻辑吧。

和 if-else 结构类似，条件运算的流程图如图 4.15 所示。图 4.15 中唯一和 if-else 不同的就是把 if-else 流程图中的“语句 1”和“语句 2”变成了条件判断表达式中的“操作数 1”和“操作数 2”。当写了一条条件判断运算语句之后，程序就会按照图 4.15 所示进行运算了。其执行顺序可以总结为如下两步。

(1) 首先判断“条件”所对应的值是“真”还是“假”。如果“条件”为真，就进行“操作数 1”的运算，否则进行“操作数 2”的运算。

(2) 当操作数为常量或者变量时，就不需要运算了。操作数为表达式时，就进行表达式所表示的运算，直到计算出表达式的最终结果数值。

例如，以下面的判断运算语句来说吧：

`1 ? 3.14*2 : radius/3;`

“条件”为 1，其逻辑值为真，那么就进行“操作数 1”($3.14*2$) 的运算。 $3.14*2$ 是一个表达式，那么计算出表达式的值，完成乘法运算。

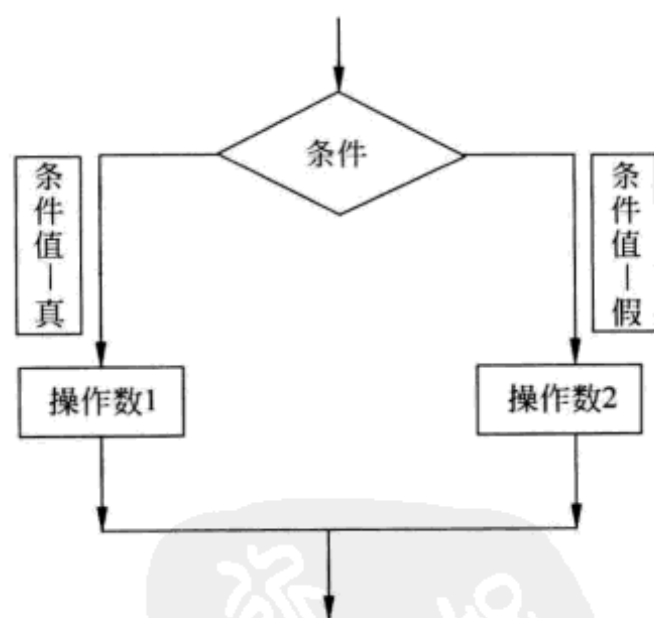


图 4.15 条件判断运算执行逻辑

3. 条件判断表达式的值

条件判断表达式的值与上面讲到的条件判断执行逻辑有关，其计算规则如下。

- 如果“条件”的值为“真”，条件判断表达式的值为“操作数 1”的值；
- 如果“条件”的值为“假”，条件判断表达式的值为“操作数 2”的值。

还是使用上面提到的表达式来分析一下，条件判断表达式的值：

```
1 ? 3.14*2 : radius/3;
```

这个条件表达式中，条件的逻辑真假值为“真”，那么条件表达式的值就为“操作数 1”的值，即 $3.14*2=6.28$ ，一个浮点型的常量。

可以使用下面这段程序来验证一下这个表达式的值是不是正确：

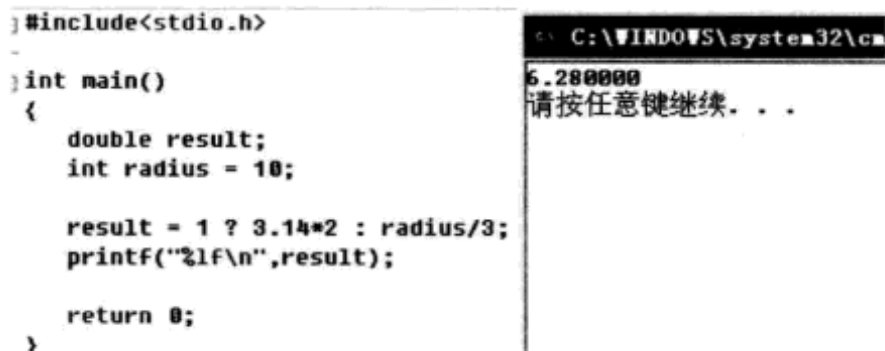
```
#include<stdio.h>

int main()
{
    double result;
    int radius = 10;

    result = 1 ? 3.14*2 : radius/3;
    printf("%lf\n",result);

    return 0;
}
```

程序的输出结果为 6.280000，如图 4.16 所示。从这个结果也可以验证图 4.15 的条件判断执行逻辑，当“条件”的值为“真”时，执行“操作数 1”的运算。



```
#include<stdio.h>

int main()
{
    double result;
    int radius = 10;

    result = 1 ? 3.14*2 : radius/3;
    printf("%lf\n",result);

    return 0;
}
```

C:\WINDOWS\system32\cmd
6.280000
请按任意键继续...

图 4.16 条件判断程序验证

4. 条件判断运算的优先级

当条件判断运算和其他的运算组合成更为复杂的表达式的时候，就要考虑各个运算的优先级和结合性了，这里就来看看条件判断的优先级，如表 4.6 所示。

表 4.6 条件判断运算

优先级	运算符	名称或含义	表达式形式	结合方向	说 明
13	?:	条件判断运算符	操作数1? 操作数2: 操作数3	右到左	三元运算符

表 4.6 列出了条件判断运算的相关性质，其中主要看看其优先级和结合性。条件判断运算的优先级为 13，比赋值运算、复合赋值运算和逗号运算高。所以，在由条件判断和其

他运算组成的复杂运算中，一般都是后计算条件判断语句的。

例如，看这个条件判断运算表达式： $3!=4 ? 1+1\&\&5>2 : 2*2$ 。这个例子中包含基本数学运算、关系运算、逻辑运算和条件判断运算。我们来看看它的运算顺序是什么。

图 4.17 为条件判断表达式的运算过程，其中①②③④为计算步骤，小括号中的数字表示运算优先级。条件判断运算的运算顺序和其他的运算稍微有点不同，它是具有逻辑判断的有选择运算。图 4.17 中，先判断“条件”真假，此处“条件”为“真”，就选择运算“操作数 1”所表示的运算，而不去执行“操作数 2”所表示的运算。

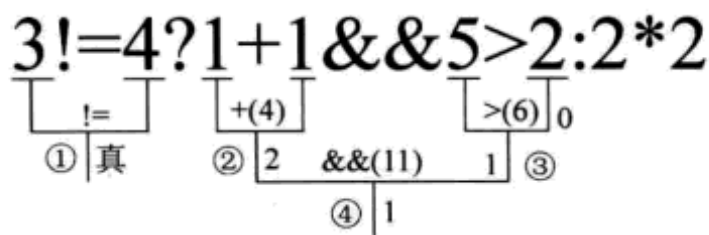


图 4.17 条件判断优先级

根据前面所讲的“条件表达式的值”，可以知道表达式 $3!=4 ? 1+1\&\&5>2 : 2*2$ 的值为 $1+1\&\&5>2$ 的值，其值为 1。可以使用下面的程序检查一下：

```
#include<stdio.h>

int main()
{
    int result;

    result = 3!=4 ? 1+1&&5>2 : 2*2;
    printf("%d\n",result);

    return 0;
}
```

程序的输出结果为 1，如图 4.18 所示。

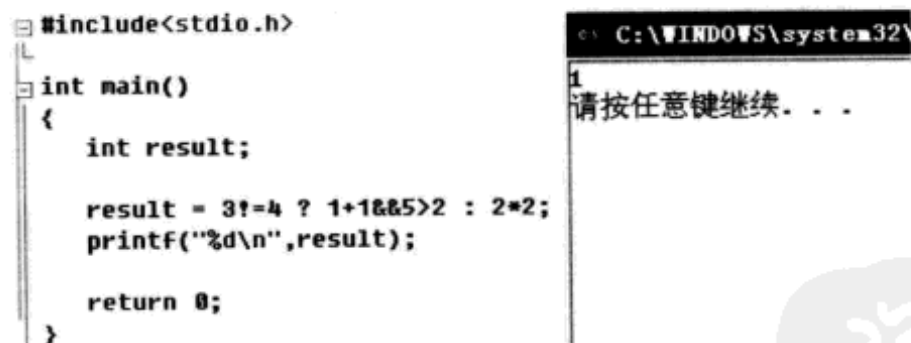


图 4.18 条件判断表达式运算优先级验证

4.5.4 if-else if-else 结构

从前面讲的（图 4.11 和图 4.13）内容可以了解到 if 结构是一个分支的判断结构，if-else 是两个分支的判断结构。接下来看一个更加强大的判断结构，它可以有多个分支，即 if-else if-else 结构，如图 4.19 所示。

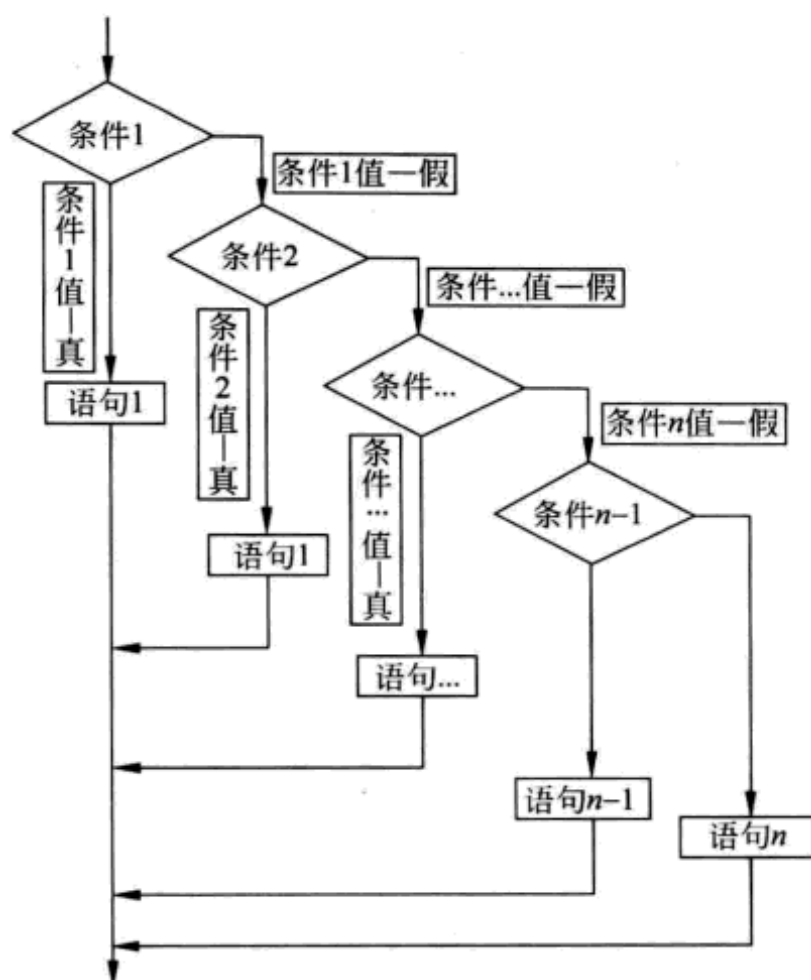


图 4.19 if-else if-else 流程图

图 4.19 就是 if-else if-else 的流程图，明显比 if 结构和 if-else 结构更为复杂一些。但是，其基本构成都是一样的，都是由“条件”和“语句”组成的。“条件”就是前面讲到的具有逻辑真假值的变量、常量和表达式。“语句”就是要执行的程序语句，不过图 4.19 中，每个分支中只写了一个语句，其实可以有任意多条。

if-else if-else 结构的 C 语言表示如下所示：

```

if(条件 1)
{
    语句 1;
}
else if(条件 2)
{
    语句 2;
}
else if(条件...)
{
    语句...;
}
else if(条件 n-1)
{
    语句 n-1;
}
else
{
    语句 n;
}
  
```

还是以“小孩撒谎，妈妈不给糖吃”为例。这次，使用如下规则给孩子发糖吃。

- 小孩今天没说谎，给发的糖的数目加倍；
- 小孩今天说了一个谎，给发的糖的数目减半；
- 小孩今天说了超过一个谎，不给发糖。

这里所述的例子稍微有点复杂，初次接触有点费劲。如果是以后遇到类似的或者更复杂的例子，可以按照下面的方法一步步分析并写出合适的程序。

(1) 分析例子中有多少种情况需要分别处理。

(2) 根据不同的情况，确定需要哪些判断条件。

(3) 根据(1)中的情况和(2)中的条件画出类似于4.19所示的流程图。如果你脑子里很清楚要写的程序的执行流程是什么样的，这一步可以省略。

(4) 根据流程图写代码，用常量、变量或者表达式合理地表示“条件”，用语句表示不同情况对应的处理方法。

(5) 按照 if-else if-else 结构的 C 语言表示，结合(4)写出完整的程序。

通过这5步，我相信只要大家认真分析，再复杂的例子也不在话下。现在让我们结合这5个步骤，重现审视一下上面所讲的例子。

(1) 通过分析可知例子中有三个情况：“小孩没说谎”、“小孩说了一个谎”和“小孩说了超过一个谎”。

(2) 只需要判断两个条件：“小孩没说谎”和“小孩说了一个谎”就可以把三种情况完全分开。

(3) 结合上面三种情况和两个条件，画出如图4.20所示的流程图。

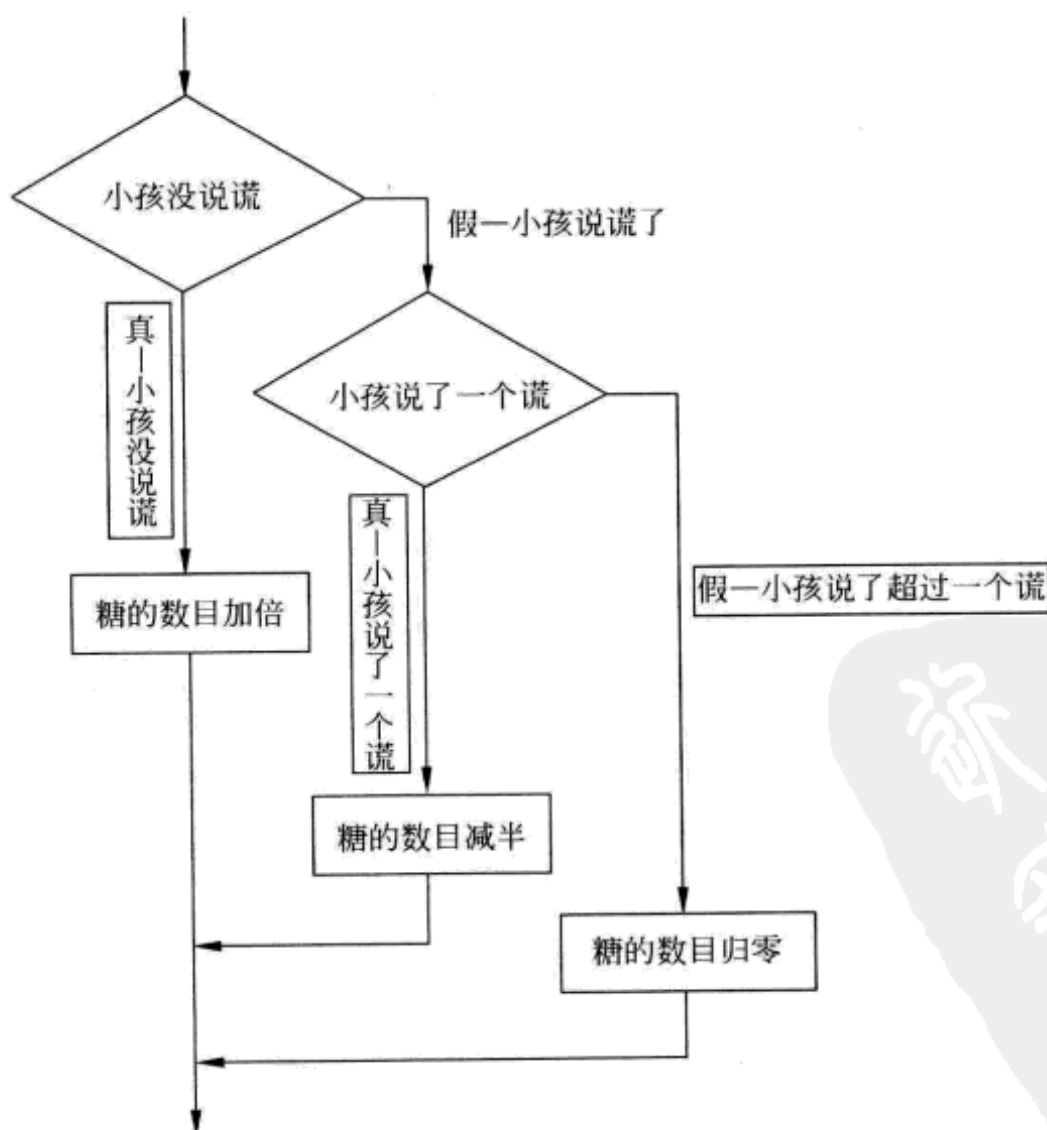


图 4.20 小孩分糖流程图

图 4.20 中有两个判断条件，分别为：“小孩没说谎”和“小孩说了一个谎”；根据这两个判断条件的真假，可以有三种情况，也就是图 4.20 中所示的三个分支：“小孩没说谎”、“小孩说了一个谎”和“小孩说了超过一个谎”，这三个分支正好对应的是例子中描述的三个要求。

(4) 整型变量 `num_lie` 表示判断条件中说谎的数目，并以此作为判断的条件。另外，用语句“`candy*=2;`”、“`candy/=2;`”和“`candy=0;`”来表示糖果的数目加倍、减半和置零三种情况所对应的处理办法。

(5) 结合 if-else if-else 判断结构的 C 语言表示和图 4.20，就可以用 if-else if-else 判断结构写成 C 语言程序了。声明定义两个整型变量 `num_lie` 和 `candy`。`num_lie` 表示小孩今天说谎数，`candy` 表示要发给小孩的糖的个数。假设，起初有 10 个糖果，今天小孩说了 3 个谎，被妈妈发现了，那么今天就没糖吃了。程序可以写成下面的样子：

```
#include<stdio.h>

int main()
{
    int num_lie;           //整型变量 num_lie，用来表示说谎次数
    int candy;             //整型变量 candy，用来表示小孩可以得到的糖果数目

    num_lie = 3;           //给变量 num_lie 赋值为 3，表示今天小孩说了 3 个谎
    candy = 10;            //给变量 candy 赋值为 10，表示默认糖果数为 10
    if(num_lie == 0)
    {
        candy *= 2;        //乘后赋值复合运算，candy 的值加倍——糖果的数目加倍
    }
    else if(num_lie == 1)
    {
        candy /= 2;        //除后赋值复合运算，candy 的值减半——糖果的数目减半
    }
    else
    {
        candy = 0;         //candy 变量赋值为 0——糖果的数目置零
    }

    printf("the number of candy is : %d\n",candy);
}
```

在这个程序中出现了之前没有见过的东西，那就是符号“//”和其后的文字。在 C 语言中，这些东西被称为注释，是写代码的人给自己和别人看，帮助自己和别人理解程序的，不会对程序产生任何影响，你可以当它们不存在。程序的输出结果为 0，如图 4.21 所示。小孩今天说谎数超过一个，因此没糖吃了。

4.5.5 嵌套的 if 结构

前面学习了三种判断结构：if 结构、if-else 结构和 if-else if-else 结构，分别用于单分支、双分支和多分支的判断结构。本节看一看嵌套 if 结构，它是将这三种分支结构揉和起来的一种结构。

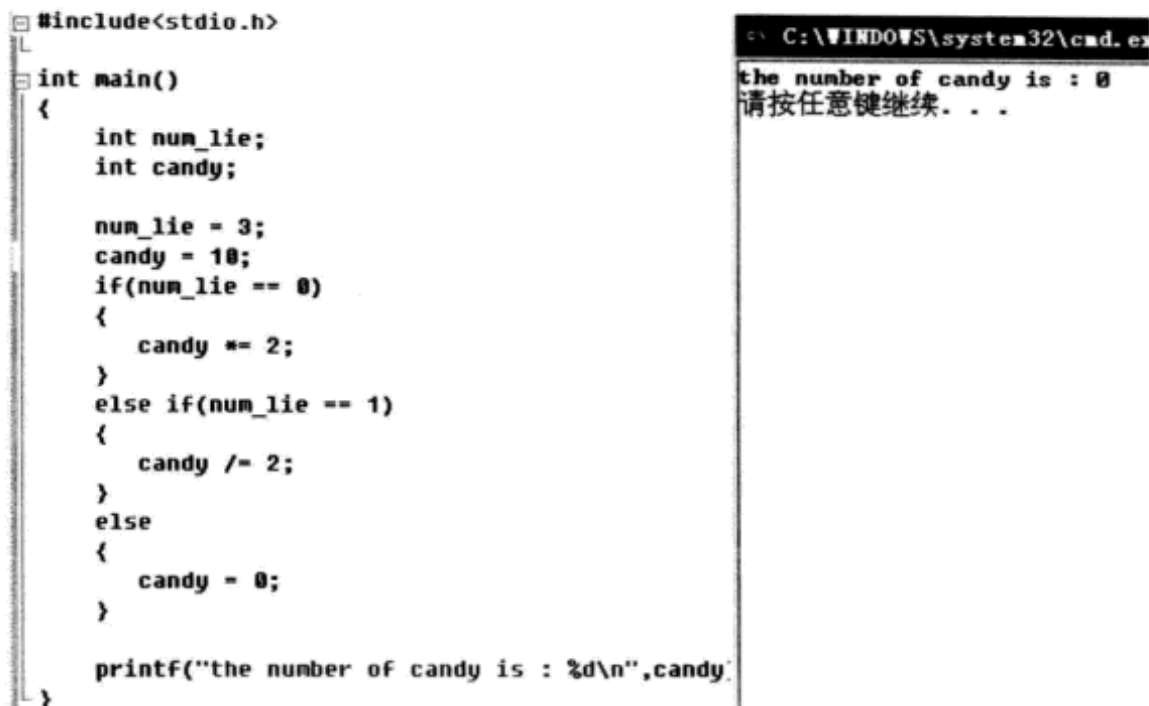


图 4.21 if-else if-else 结构程序验证

1. 嵌套结构的形式

前面讲的三种判断结构中，每种结构都包含被选择执行的语句。不过，之前看到的语句都是按照顺序结构组织的顺序结构语句。如果将判断结构中语句的组织方式从顺序结构变成判断结构，就形成了嵌套 if 结构。

下面来看一个简单的嵌套 if 结构的程序结构组织方式：

```

if(条件)
{
    if(条件 1)
    {
        语句 1;
    }
    else
    {
        语句 2;
    }
}

```

这个结构形式就是一个 if 结构中嵌套 if-else 的结构。其实，嵌套的 if 结构的形式有成千上万种，只要你愿意去写，可以写出任意多个。

现在已经学习了三种判断结构了，每一种判断结构中都包含语句。如果把其中一个或者几个语句变成一个判断结构，就产生了一种新的嵌套的 if 结构了。如果在嵌套的 if 结构中再做类似的替换处理，就又产生了一种新的嵌套的 if 结构，如此往复，就可以形成任意多的嵌套的 if 结构了。

2. if和else的配对——避免混乱

在 C 语言的三种 if 判断结构中，如果分支中的语句只有一条，可以省略包含这个语句的大括号。例如，下面的三个结构是等价的。

(1) 标准的 if 结构：

```
int i = 0;
```

```
if(1>2)
{
    i++;
}
```

(2) 去掉大括号的 if 结构:

```
int i = 0;
if(1>2)
    i++;
```

(3) 判断条件和执行语句写在同一行的 if 结构:

```
int i = 0;
if(1>2) i++;
```

既然三种判断结构中的大括号有的时候是可以省略的,那么在嵌套的 if 结构中出现很多的 if 和 else 时,如何确定哪个 if 和哪个 else 组成 if-else 结构?哪个 if 是单独组成 if 结构的?哪些 if、哪些 else if 和哪些 else 是组成 if-else if-else 结构的?这就是下面将要讲到的 if 和 else 的配对问题了。

还是坚持这样一个原则:就近原则。if、else if、else 这些关键字就近组合,形成三种判断结构中的一种。先看是否形成 if-else if-else 结构,如果不行,然后看是否形成 if-else 结构,如果不行,最后看是不是形成 if 结构。

来看看下面的例子:

```
int i = 0;
int flag = 8;
if(flag!= 1)
if(flag == 0)
i=2;
else
i=3;
```

这个例子和下面这个标准的条件判断结构是等价的:

```
int i = 0;
int flag = 8;
if(flag!= 1)
{
    if(flag == 0)
    {
        i=2;
    }
    else
    {
        i=3;
    }
}
```

其中,

```
if(flag == 0)
i=2;
else
i=3;
```

是被当作 if 结构的语句,而这个 if-else 结构又是一个省略了大括号的 if-else 结构。可

以使用 `printf()` 函数来分别输出这两段程序运行结束后 `i` 的值, 结果如图 4.22 所示。



图 4.22 嵌套 if 结构中的 if-else 配对

建议大家在写代码的时候, 按照标准的判断结构来写, 以便于自己和别人更容易理解代码的含义。

4.6 switch 判断结构

除了三种 `if` 判断结构以外, C 语言还提供了一种 `switch` 判断结构。它类似于 `if-else if-else` 结构, 是一种多分支的判断结构, 不过它也可以实现另外两种 `if` 判断结构。

4.6.1 switch 基本结构

`switch` 判断结构是 C 语言提供的另一种多分支的判断结构。与 `if-else if-else` 结构相比, `switch` 的代码结构比较简洁直观。

1. switch 结构的格式

`switch` 判断结构的 C 语言形式如下所示:

```
switch(表达式)
{
    case 常量表达式 1      : 语句 1;
    case 常量表达式 2      : 语句 2;
    ...
    case 常量表达式 n-1    : 语句 n-1;
    case 常量表达式 n      : 语句 n;
    default                 : 语句 n+1;
}
```

在这个结构中, `switch` 关键字后面的“表达式”可以是一个常量、变量和一个表达式。

case 关键字后面的“常量表达式”必须是常量或者常量组成的表达式。

2. switch结构的执行逻辑

switch 判断结构的流程图如图 4.23 所示。从图 4.23 中可以看出,switch 判断结构是根据 switch 关键字后面的“表达式”的值和 case 关键字后面的“常量表达式”的值是否相等作为判断条件来选择语句执行的。

3. switch结构使用

根据 switch 结构的 C 语言格式,可以写出下面的程序。声明一个变量,赋值为 2,然后使用 switch 判断结构,对 flag 为 1、2、3 和其他这四种情况进行判断,在每种情况中使用 printf()函数输出对应的情况。

```
#include<stdio.h>

int main()
{
    int flag = 2;

    switch(flag)
    {
        case 1 : printf("1\n");
        case 2 : printf("2\n");
        case 3 : printf("3\n");
        default: printf("default\n");
    }

    return 0;
}
```

程序的输出结果为 2、3 和 default,如图 4.24 所示。按照 switch 判断结构的流程图,显然,当 flag 为 2 的时候,2、3 和其他(default)这几个对应的情况都是要输出的。

```
#include<stdio.h>

int main()
{
    int flag = 2;

    switch(flag)
    {
        case 1 : printf("1\n");
        case 2 : printf("2\n");
        case 3 : printf("3\n");
        default: printf("default\n");
    }

    return 0;
}
```

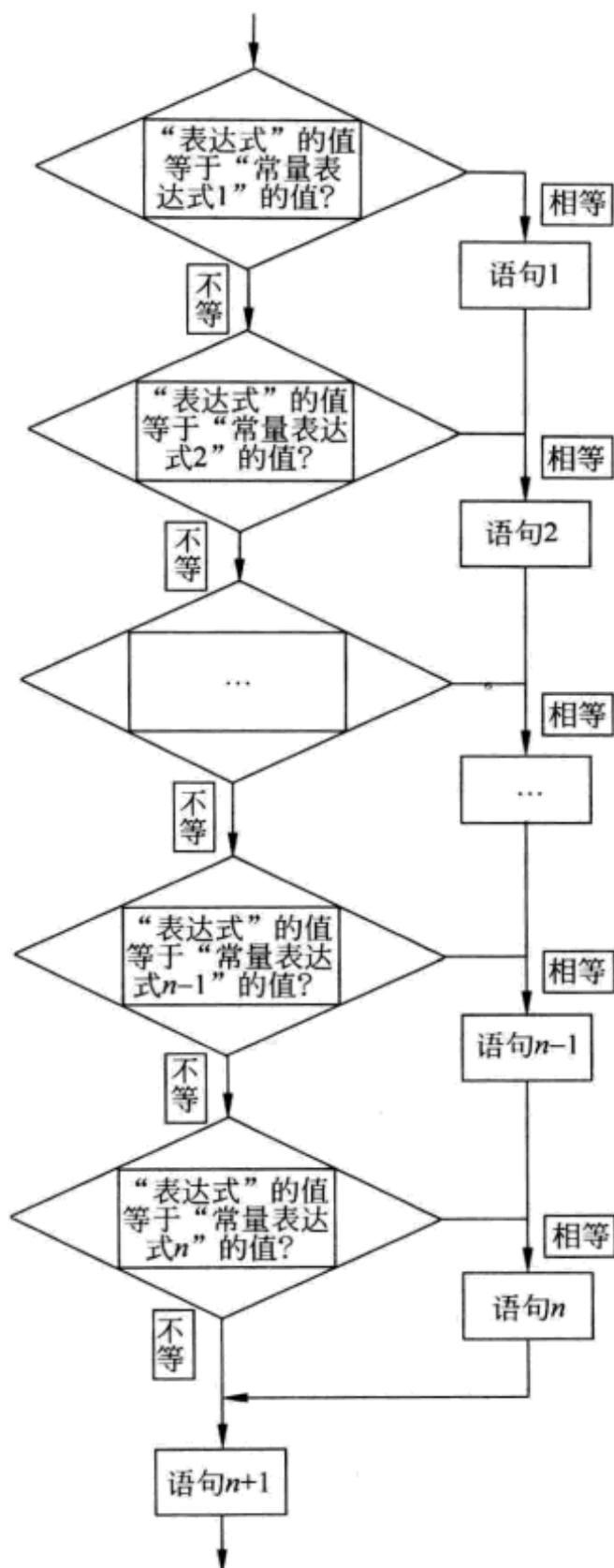


图 4.23 switch 判断结构流程图

```
C:\WINDOWS\system32
2
3
default
请按任意键继续...
```

图 4.24 switch 判断结构程序

4.6.2 果断结束——break 的使用

细心观察，可以从 switch 结构的流程图看出，其中的条件判断只是选择从哪个语句开始执行，并一直执行到底。这样 switch 结构和 if-else if-else 结构完全是两种不同的执行顺序，怎么能说 switch 可以实现类似于 if-else if-else 的多分支判断结构呢？

C 语言提供了一个关键字 break，可以使 switch 结构很容易变成类似于 if-else if-else 判断的多分支结构。break 关键字可以使 switch 结构中一直执行到底的语句在任意的地方停止下来，果断结束正常的执行。

1. 添加了break的switch判断结构格式

添加了 break 的 switch 判断结构的 C 语言形式如下所示：

```
switch(表达式)
{
    case 常量表达式 1      : 语句 1;break;
    case 常量表达式 2      : 语句 2;break;
    ...
    case 常量表达式 n-1    : 语句 n-1;break;
    case 常量表达式 n      : 语句 n;break;
    default                 : 语句 n+1;break;
}
```

可以选择在任意的地方添加或者不添加 break。上面的程序在每个 case 分支后都添加 break 以后，这个结构就可以实现类似于 if-else if-else 判断的多分支结构了。

2. 添加了break的switch判断结构执行逻辑

上面的每个 case 分支都添加了 break 的 switch 判断结构，其执行的流程图如图 4.25 所示。这次是不是和 if-else if-else 判断结构的流程图很像？只不过把 if-else if-else 判断结构的流程图翻转了一下而已。

3. 添加了break的switch结构

当我们在 switch 判断结构的 case 分支后面添加 break 的时候，会终止其后的其他 case 分支的执行，以此来达到类似于 if-else if-else 的多分支判断结构的效果。

作为和前面的不添加 break 的 switch 结构的对比，可以写出下面的程序：

```
#include<stdio.h>

int main()
{
    int flag = 2;

    switch(flag)
    {
        case 1 :
            printf("1\n");
            break;
        case 2 :
```

```
printf("2\n");  
break;  
case 3 :  
    printf("3\n");  
    break;  
default:  
    printf("default\n");  
    break;  
}  
  
return 0;  
}
```

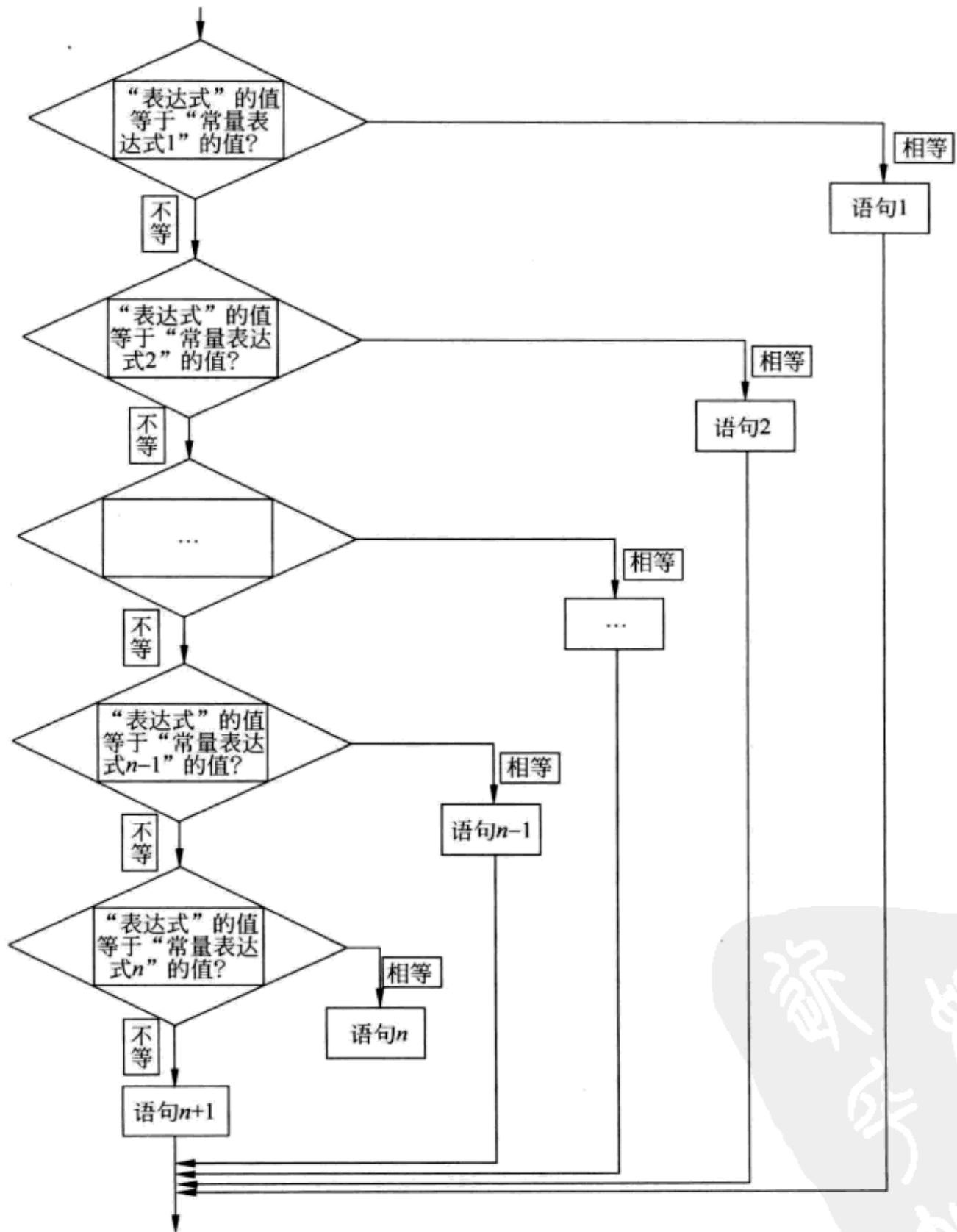


图 4.25 添加 break 的 switch 判断结构流程图

程序的输出结果为 2，如图 4.26 所示。可见，添加 break 确实达到了我们想要的目的。



```
#include<stdio.h>

int main()
{
    int flag = 2;

    switch(flag)
    {
        case 1 :
            printf("1\n");
            break;
        case 2 :
            printf("2\n");
            break;
        case 3 :
            printf("3\n");
            break;
        default:
            printf("default\n");
            break;
    }

    return 0;
}
```

C:\WINDOWS\system32
2
请按任意键继续. . .

图 4.26 添加了 break 的 switch 结构程序

4.7 循环结构

和顺序结构、判断结构类似，循环结构也是一种组织程序的方式。不过，这种方式 and 前面两种都不一样，它是按照条件决定要不要继续执行同样的语句。这也是很重要的，而且是写程序时经常用到的一种结构，可以说没有它程序很难写或者说没法写。

4.7.1 while 循环结构

while 循环结构是循环结构中比较简单的一种。循环结构是根据条件确定某些语句要不要继续重复执行。作为循环结构的一种，while 循环结构也是完成这样的事情的，那么它是怎么来完成的呢？先来看看 while 循环结构的两种 C 语言表示：“while 循环”和“do-while”循环。这两种结构类似，但是稍微有点区别。

1. while 循环

while 循环的程序结构用 C 语言表示如下：

```
while(条件)
{
    语句 1;
    语句 2;
    ...
    语句 n;
}
```

while 循环结构中的“条件”和 if 判断结构中的条件是一样的，可以是具有逻辑真假值的任意常量、变量和表达式。大括号中的语句，就是将要重复循环执行的语句，可以有多个也可以有一个，只要在大括号中，就有机会被重复循环执行。

当大括号中只有一条语句的时候，大括号可以省略，和 if 判断结构是一样的。省略大括号的 while 循环可以是下面两种：

```
while(条件) 语句;
```

和

```
while(条件)
    语句;
```

根据 C 语言的规定，对于这种 while 循环，其组织的程序的执行逻辑流程图如图 4.27 所示。

从图 4.27 中就可以知道 while 结构是如何根据条件来判断是不是重复执行某些语句了。当“条件”为真的时候，就选择继续执行大括号中的语句，当条件为假的时候，就不再执行大括号中的语句了。

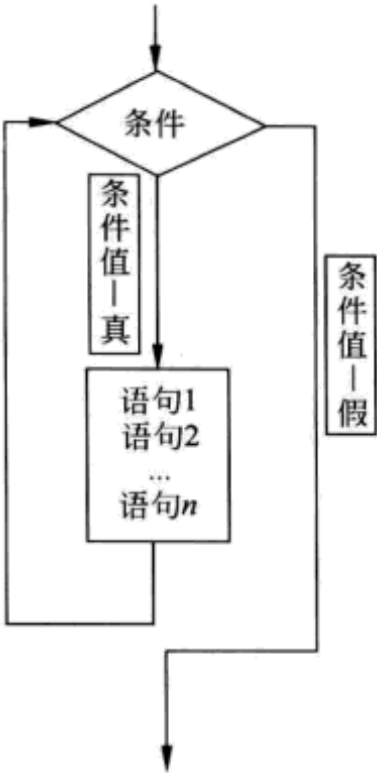


图 4.27 while 循环流程图

2. do-while 循环

do-while 循环完成和 while 循环类似的功能，但是其 C 语言表示略微有点区别。do-while 循环的 C 语言表示如下所示：

```
do
{
    语句 1;
    语句 2;
    ...
    语句 n;
}
while(条件);
```

do-while 循环结构中的“条件”和“语句”与 while 循环结构中的“条件”和“语句”的含义是一样的。当大括号中只有一条语句的时候，do-while 结构也可以简化成以下两种形式：

```
do 语句
while(条件);
```

和

```
do
    语句
while(条件);
```

do-while 循环结构与 while 循环结构差别比较大的就是它的执行逻辑，如图 4.28 所示为 do-while 的执行流程图。

为了理解两种 while 循环结构，来看这样一个例子：让电脑说 5 次 “you are so beautiful! ”。这样的例子就可以使用本节所讲的循

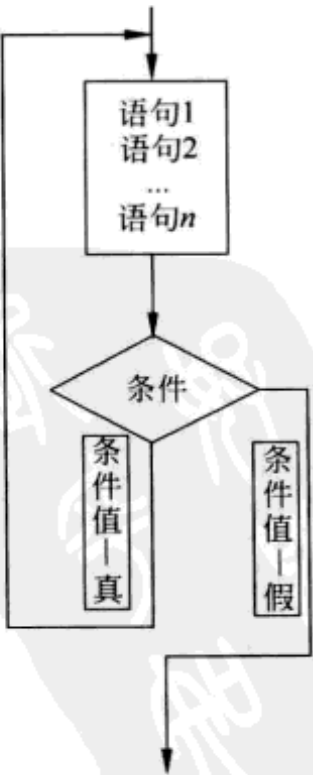


图 4.28 do-while 流程图

环结构，因为循环结构就是根据条件重复循环地做某些事。在这里使用循环结构让电脑重复循环地说：“you are so beautiful!”。

使用 while 循环的程序如下所示：

```
#include<stdio.h>

int main()
{
    int num = 0;

    while(num < 5)
    {
        printf("you are so beautiful!\n");
        num++;
    }

    return 0;
}
```

在这个程序中使用了 while 循环，不断地执行 printf()函数和 num 自增语句。决定是否继续执行 printf()函数和 num 自增语句的条件是 num<5。如果 num<5 为“真”就继续执行，否则就不执行。程序先判断 num<5 是否为真，然后再决定是否继续执行输出和自增。由于每进行一次输出 num 的值就会加 1，所以输出 5 以后，num 的值就会变成 5。这时 num<5 这个条件就会为“假”，不再执行输出和自增语句了。

程序的输出结果如图 4.29 所示，输出了 5 行“you are so beautiful!”，程序命令电脑说了 5 次“you are so beautiful!”。



图 4.29 while 循环程序

同样也可以使用 do-while 循环写出这个功能的程序，程序如下：

```
#include<stdio.h>

int main()
{
    int num = 0;

    do
```

```

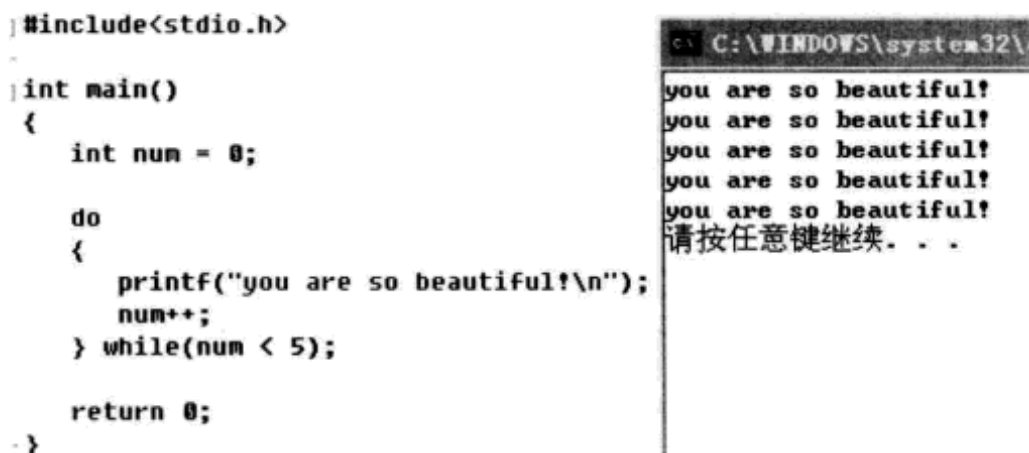
{
    printf("you are so beautiful!\n");
    num++;
} while(num < 5);

return 0;
}

```

按照 do-while 的执行流程，这个程序和上面的那个 while 程序有些不同。程序先执行输出和自增，然后判断 num 的值是不是小于 5，如果小于 5 则继续执行。由于每输出一次，num 自增 1，因此输出 5 次后，条件 num<5 才为假。所以，也是只输出了 5 个 “you are so beautiful!”。

程序的输出结果和 while 循环一样，如图 4.30 所示。



```

#include<stdio.h>

int main()
{
    int num = 0;

    do
    {
        printf("you are so beautiful!\n");
        num++;
    } while(num < 5);

    return 0;
}

```

C:\WINDOWS\system32\
you are so beautiful!
you are so beautiful!
you are so beautiful!
you are so beautiful!
you are so beautiful!
请按任意键继续. . .

图 4.30 do-while 循环程序

4.7.2 for 循环结构

for 循环结构是 C 语言提供的另一种循环结构。它和 while 循环结构一样，根据条件来选择某些语句是否继续重复循环执行。

1. for 循环结构的表示和意义

比起 while 循环结构，for 循环结构稍微有些复杂，因为它揉和了 while 循环以外的其他东西。到底揉和了什么？先来看看 for 循环结构的 C 语言表示吧。

```

for(表达式 1; 条件; 表达式 2)
{
    语句 1;
    语句 2;
    ...
    语句 n;
}

```

在这个 C 语言表示中有 4 个东西：“表达式 1”、“条件”、“表达式 2”和“语句”。其

中，“条件”和“语句”跟 while 循环结构中的“条件”和“语句”是一样的，条件是具有真假值的常量、变量或表达式，语句就是程序中执行的语句。另外，多出来的两个东西都是表达式，它可以是前面介绍的任何表达式。那么这两个表达式到底是干什么的呢？来看看 for 循环的执行流程吧！

上面所述的 for 循环是按照下面这样的步骤一步一步执行的。

- (1) 进行表达式 1 的运算；
- (2) 判断条件是否为真，如果为“真”，则进行第 (3) 步。否则结束 for 循环，执行 for 循环结构后面的语句；
- (3) 执行大括号中的所有语句；
- (4) 进行表达式 2 的运算，并转入第 (2) 步。

可以画出类似于 while 循环的流程图来表示 for 循环结构的执行流程，如图 4.31 所示。从图 4.31 中可以看出，for 循环结构和 while 循环结构的最大的不同，就是它在 while 循环结构的执行流程的前面和后面分别加入了一个表达式运算的执行。

正是由于 for 循环结构融合了两个表达式到循环结构里，才使得循环结构中一些东西的实现变得紧凑、直观。到底使什么东西变得直观和紧凑了呢？来看看下面这段 for 循环结构的实现，还是让计算机说 5 遍 “you are so beautiful!”。

```
#include<stdio.h>

int main()
{
    int num;

    for(num=0; num<5; num++)
    {
        printf("you are so beautiful!\n");
    }

    return 0;
}
```

对比前面讲的 while 循环的例子可以发现：将给 num 赋初值作为 for 循环结构的“表达式 1”了；将 num 自增作为 for 循环结构的“表达式 2”了。

这么做是有一定道理的，因为根据如图 4.31 所示的 for 循环流程图，for 循环结构中的“表达式 1”在进入 for 循环之前执行一次，因此很适合做一些初始化操作。例如，这里给 num 赋值为 0；for 循环中的“表达式 2”是在每次执行完要执行的语句之后执行的。因此，很适合写一些每次 for 循环最后都执行的语句，例如这里的给 num 自增。

在 C 语言中，for 循环中的“表达式 1”和“表达式 2”都是可以不写的，不写就什么都不执行了。这样的 for 循环结构就和 while 结构是一样的了。例如，上面的 for 循环的例子可以写成类似于 while 循环结构的样子。

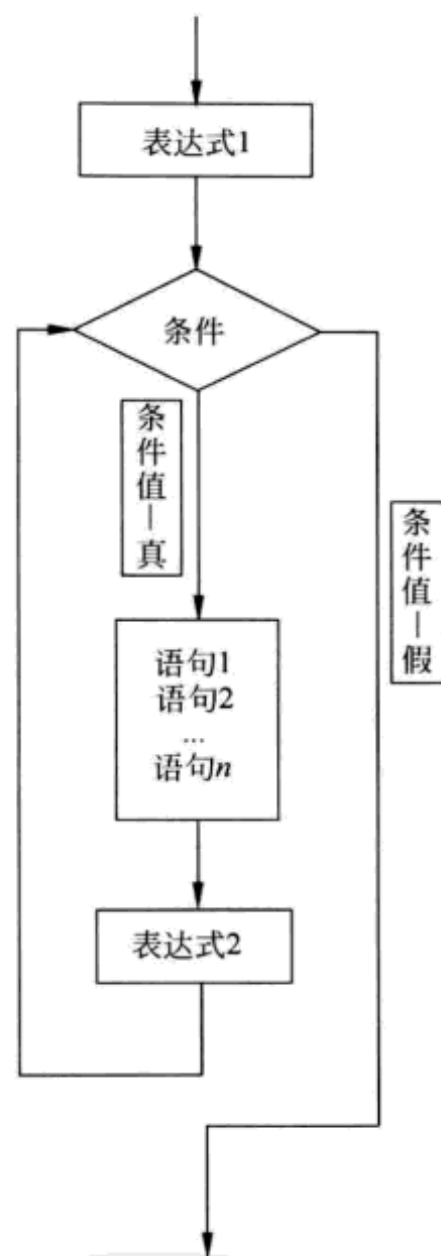


图 4.31 for 循环流程图

```
#include<stdio.h>

int main()
{
    int num = 0;

    for(; num<5;)
    {
        printf("you are so beautiful!\n");
        num++;
    }

    return 0;
}
```

两个 for 循环结构的程序的输出和前面 while 循环结构的输出是一样的，都是输出 5 遍 “you are so beautiful!”，如图 4.32 所示。



图 4.32 for 循环结构程序

2. 逗号表达式

可以看到 for 循环结构中有两个表达式：“表达式 1”和“表达式 2”。凡是表达式都可以成为 for 循环结构中的两个表达式之一。之前介绍的表达式都是完成一个运算的表达式，如果能让 for 循环中的表达式同时完成好几个运算，该怎么办呢？这就需要逗号运算了。逗号是用来完成好几个运算的运算，逗号运算所形成的表达式叫做逗号表达式。

先来看看逗号表达式的 C 语言表示形式吧：

表达式 1, 表达式 2, 表达式 3, ..., 表达式 n

可以看出逗号表达式是由 n 个表达式组成的，之间由逗号隔开。其中，每个表达式可以是任何一种表达式，如赋值表达式、基本数学运算表达式、关系运算表达式、逻辑运算表达式等，都可以作为逗号表达式其中的一个。

逗号表达式的执行逻辑如图 4.33 所示。逗号表达式和顺序结构类似，是按照表达式的

顺序一个一个执行的。与顺序结构不同的是逗号表达式是一个表达式，而顺序结构是一种语句组织结构，或者说语句组织方式。

每一个表达式都是有一个值的，那逗号表达式的值是什么呢？它比其他表达式值的计算方法都简单。C 语言规定，逗号表达式的值就是组成逗号表达式的最后一个表达式的值，也就是图 4.33 中表达式 n 的值。

接下来举个例子来说明一下，逗号语句如何作为 for 循环结构中的表达式，以及逗号语句值的验证。

```
#include<stdio.h>

int main()
{
    int num;
    int count;
    int comma_result = (5/2, 1^0, 5>0, 1+1);
                        //逗号表达式的值赋值给整型变量

    printf("comma_result = %d\n\n", comma_result);

    for(num = 0, count = 1; num<5 ; num++,count++)
        //逗号表达式作为 for 循环结构中的表达式
    {
        printf("[%d] you are so beautiful!\n", count);
    }

    return 0;
}
```



图 4.33 逗号表达式的执行逻辑

在这个程序中有两处用到了逗号表达式，如程序中注释的地方所示。

- ❑ 我们把逗号表达式：“5/2, 1^0, 5>0, 1+1”赋值给整型变量 `comma_result`，来验证逗号表达式的值是不是组成逗号表达式的最后一个。由于赋值运算的优先级高于逗号表达式，所以用小括号强制改变了逗号表达式的结合性和优先级。
- ❑ 在 for 循环结构中，也使用了逗号语句。“表达式 1”使用逗号表达式分别给两个变量赋值，“表达式 2”使用逗号表达式给两个变量自增 1。

程序还是完成让计算机说 5 次“you are so beautiful!”。不过这次用变量 `count` 统计了一下说的次数，看计算机有没有偷懒，少说了一次。程序的输出如图 4.34 所示。逗号表达式的值就是组成逗号表达式的最后一个表达式的值，计算机也没有少说一次“you are so beautiful!”，程序验证成功。

```

#include<stdio.h>

int main()
{
    int num;
    int count;
    int comma_result = (5/2, 1^0, 5>0, 1+1);

    printf("comma_result = %d\n\n", comma_result);

    for(num = 0, count = 1; num<5 ; num++,count++)
    {
        printf("[%d] you are so beautiful!\n", count);
    }

    return 0;
}
```

C:\WINDOWS\system32\cmd.

comma_result = 2

[1] you are so beautiful!

[2] you are so beautiful!

[3] you are so beautiful!

[4] you are so beautiful!

[5] you are so beautiful!

请按任意键继续...

图 4.34 逗号表达式程序验证

4.7.3 goto 语句

goto 语句是 C 语言中能力比较大的语句，它可以让程序去任何地方执行。类似于马路上开着警笛的警车，可以越过任意的红灯，不必遵守交通规则。由于 goto 语句不守 C 语言的规则，所以很多人建议不要用 goto 语句了。但是，就像没有警车也是不行的，只要不乱用 goto 语句就好，不必一棒子打死。

要用好 goto 语句，先得看看 C 语言中的 goto 语句的格式，然后再看它的含义。C 语言中的 goto 语句的使用格式如下：

```
...
label:
    语句;
...
goto label;
...
```

在这个结构中“label”被称为标识符，它可以是任何由字母、数字和下划线组成的标识符。goto 是一个关键字，不可以改，“goto label;”就是一个 goto 语句。goto 语句既可以位于 label 标签之前，也可以位于 label 标签之后，上面只表示了一种情况，另一种情况如下所示：

```
...
goto label;
...
label:
    语句;
...
```

goto 语句在 C 语言中使用的时候，一般都是这两种。前一种叫向前 goto，后一种叫向后 goto。接下来看看 goto 语句是如何不守 C 语言的规矩的，也就是它的执行流程，如图 4.35 所示。

图 4.35 是一个向前 goto 的流程图。从图 4.35 中可以看出，正常的程序执行流程被 goto 语句打乱，转去某个地方执行。

还是以让电脑说 5 次“you are so beautiful!”为例，看看 goto 语句的威力。这次，我们只准电脑说 5 次，说多了不行。那么，可以用 goto 语句写这样的例子，让电脑说 5 次就转去其他地方执行。

```
#include<stdio.h>

int main()
{
    int num = 0;
    while(num < 10)
    {
        if(num == 5)
        {
```

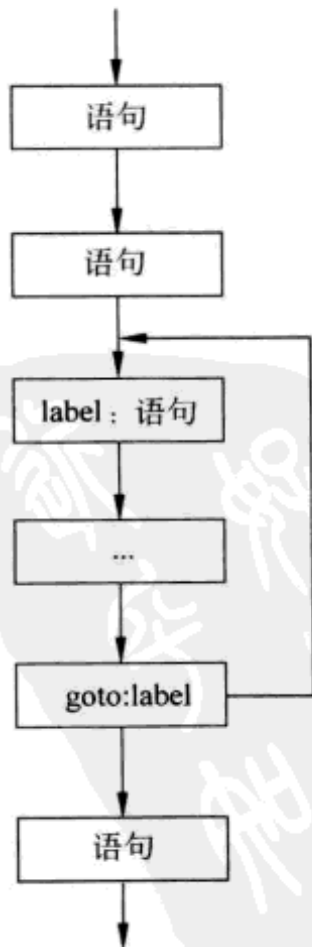


图 4.35 goto 语句的执行流程

```

        goto enough;
    }
    printf("you are so beautiful!\n");
    num++;
}

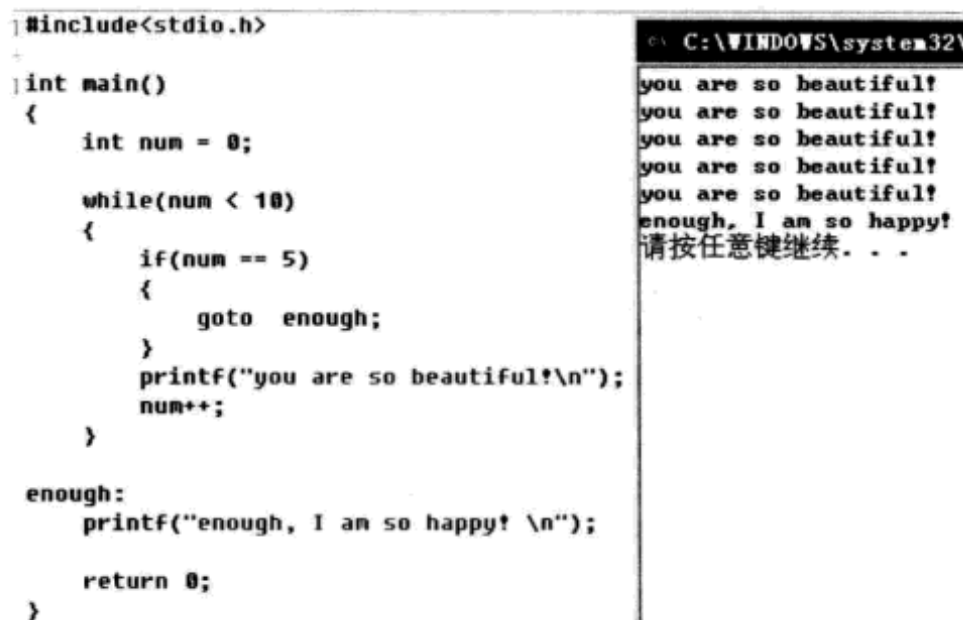
enough:
    printf("enough, I am so happy! \n");

    return 0;
}

```

在程序中，修改 while 循环结构的条件为“num < 10”，想让电脑说 10 次“you are so beautiful!”，来献殷勤！但是，只让电脑说 5 次。因此，在 while 循环中使用 if 结构一直检查着不让电脑多说。说够 5 次就直接转向 enough 标签后面的 printf() 函数，绕过正常的 while 循环规则。

程序的输出如图 4.36 所示，电脑马屁确实没有拍成，只说了 5 次“you are so beautiful!”。



```

#include<stdio.h>

int main()
{
    int num = 0;

    while(num < 10)
    {
        if(num == 5)
        {
            goto enough;
        }
        printf("you are so beautiful!\n");
        num++;
    }

    enough:
        printf("enough, I am so happy! \n");

    return 0;
}

```

C:\WINDOWS\system32\cmd.exe
you are so beautiful!
you are so beautiful!
you are so beautiful!
you are so beautiful!
you are so beautiful!
enough, I am so happy!
请按任意键继续. . .

图 4.36 goto 语句程序

4.7.4 循环嵌套

循环结构的作用就是重复执行某些语句。但是，有的时候单个循环结构是不够的，需要循环嵌套才能完成想要的功能。打一个比方：日复一日，年复一年，同样都是时间，但是年中有日。类似地，循环嵌套就是循环结构中包含循环结构，简单地理解，就是把循环结构中的某些语句再按照想要的条件让它循环执行。

前面已经介绍了三种循环结构的 C 语言表示：while 循环、do-while 循环和 for 循环。在循环嵌套的时候，可以使用上面三种结构的任意组合进行嵌套。当然嵌套的循环结构之中，依然还是可以进行循环嵌套的，不过这就更为复杂了。

接下来，看一种 while 循环嵌套 while 循环的 C 语言表示，其他的循环嵌套与之类似。

```

while(条件1)
{
    语句;
    ...
}

```



```

while(条件 2)
{
    语句;
    ...
}

```

这个 while 循环嵌套结构和前面的 while 循环结构类似，只是将其中的某些语句换成一个循环结构而已。

图 4.37 就是上面所示的 while 循环嵌套的流程图。其中，外层 while 循环的两个省略号之间的语句被换成了一个里层的 while 循环，构成一个简单的 while 循环嵌套。

接下来举一个例子，来说明循环嵌套语句的作用与 C 语言实现。我们小的时候经常背诵乘法口诀表：“一一得一，一二得二，…，九九八十一”。现在写程序让计算机背诵乘法口诀表，就可以使用嵌套循环结构，因为乘法口诀表中的被乘数和乘数两个因子在不断地变化和重复，两个循环的嵌套就可以实现两个因子的重复和变化。

先来看看让计算机背诵乘法口诀表的程序的实现吧。

```

#include<stdio.h>

int main()
{
    int multiplicand = 1 ; //被乘数
    int multiplier;        //乘数
    int product;           //乘积

    while(multiplicand <= 9)
        //被乘数要小于等于 9
    {
        multiplier = 1;
        //每一次乘数都从 1 开始

        while(multiplier <= multiplicand)
            //乘数小于等于被乘数，这样可以
        {
            product = multiplicand*multiplier;
            printf("%dx%d=%d ",multiplicand,multiplier,product);
            //输出乘法口诀
            multiplier++; //乘数加 1
        }

        multiplicand++; //被乘数加 1
        printf("\n");
        //换一行输出，另一个被乘数的乘法运算
    }
}

```

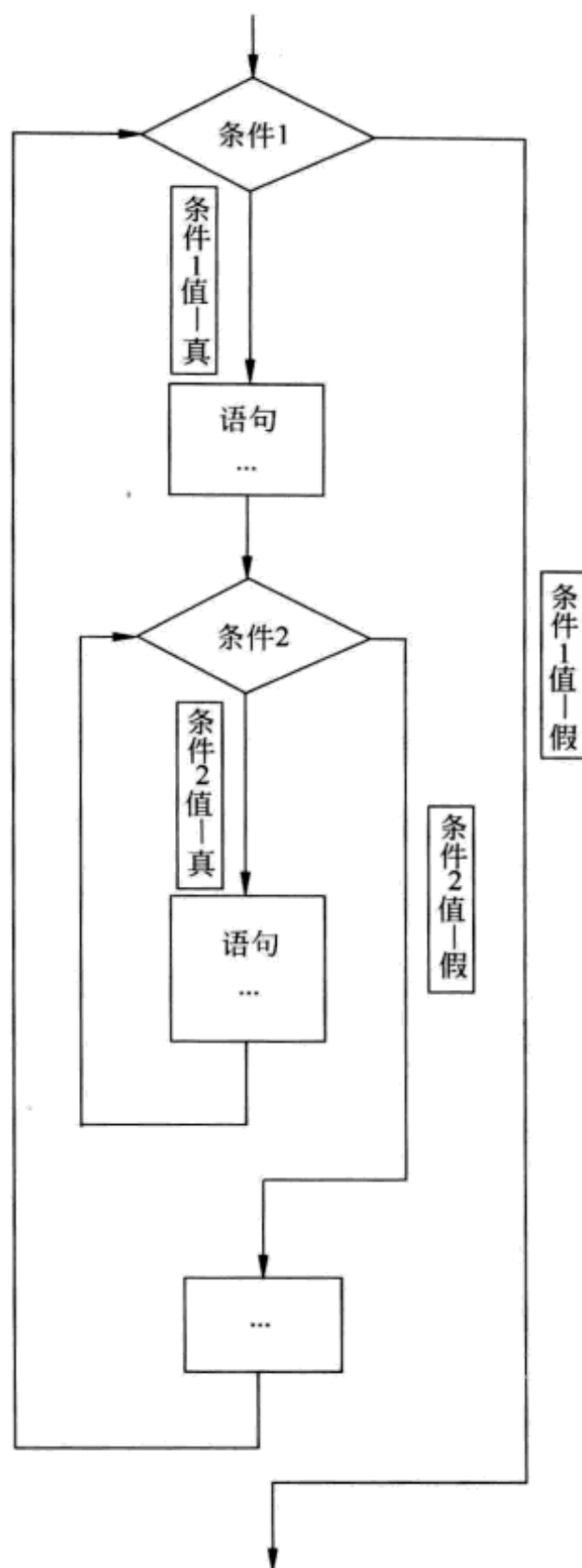


图 4.37 while 循环嵌套流程图


```

    return 0;
}

```

这个程序相对于我们之前所见到的程序稍微有点复杂，下面一点点进行分析。

- ❑ 看清楚结构：这是一个由两个 `while` 循环组成的嵌套循环结构。
- ❑ 看外层循环：主要完成内层循环和被乘数加 1。循环一次执行一次内层循环和被乘数加 1，总共执行 9 次，也就是被乘数最大到 9。
- ❑ 看内层循环：主要完成乘法口诀表达式输出和乘数加 1。循环一次输出一个乘法口诀表达式，直到输出被乘数个表达式，这样就可以只输出类似 3×2 ， 2×3 这样的表达式中的一个，避免重复。

程序的输出如图 4.38 所示，看见了吧，久违的九九乘法表啊。

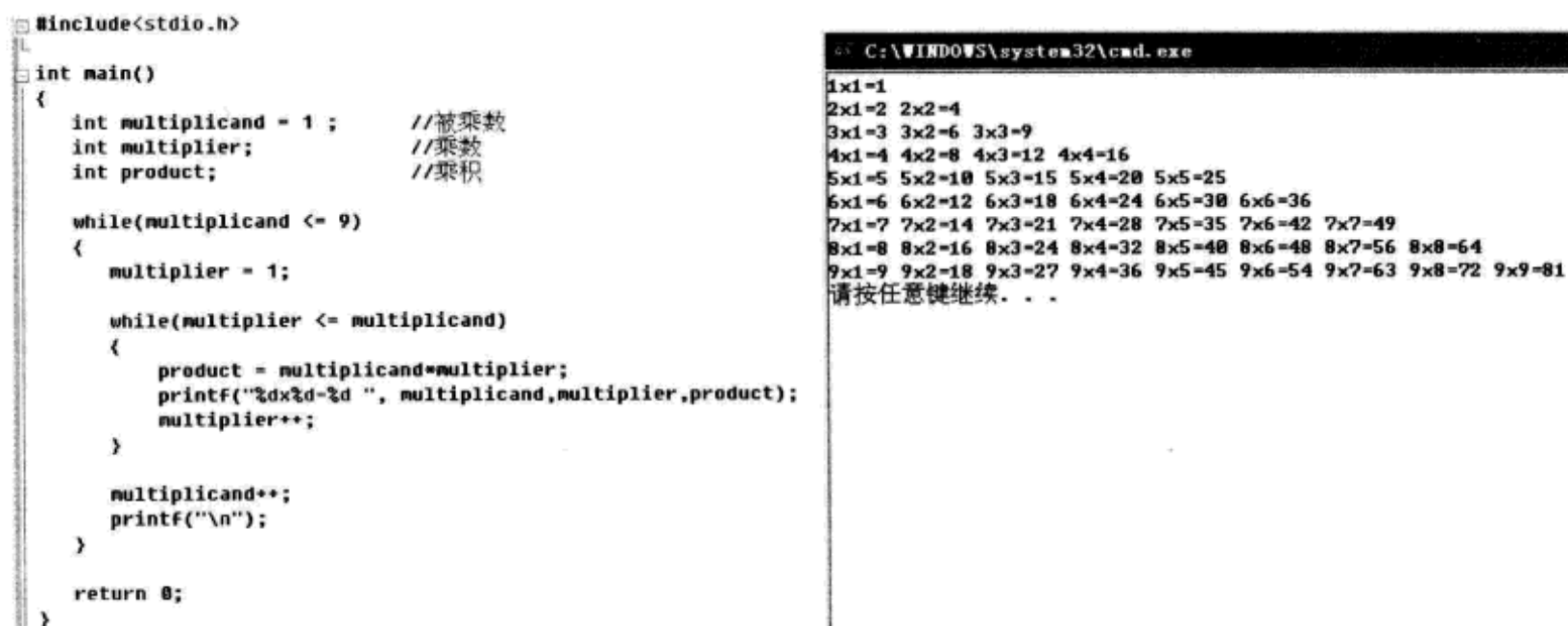


图 4.38 乘法口诀表程序验证

4.7.5 break 和 continue

循环结构可以使某些语句重复循环地执行。但是，有的时候出现意外情况，需要中途停止循环，该怎么办呢？有人会说可使用 `goto` 语句跳出循环结构，当然这是可以的。但是，`goto` 语句太过神通广大了，C 语言提供了另外两个关键字专门用来停止循环结构的重复循环：`break` 和 `continue`。

`break` 和 `continue` 都可以用在循环结构中停止正常的循环结构循环。它们在 C 语言中的使用如下所示：

```

while(条件)
{
    语句;
    ...
    break;
    语句;
}

```

这里只举了 `break` 关键字在 `while` 循环结构中的使用。`break` 在 `do-while` 和 `for` 另外两种循环结构中，以及 `continue` 在三种循环结构中的使用都是类似的：作为循环结构中的一

条语句被执行。

既然 `break` 和 `continue` 都是用来停止循环结构的重复循环的,那么它们有什么区别呢?先来看看 `break` 和 `continue` 在 `while` 循环中的流程图吧,如图 4.39 所示。

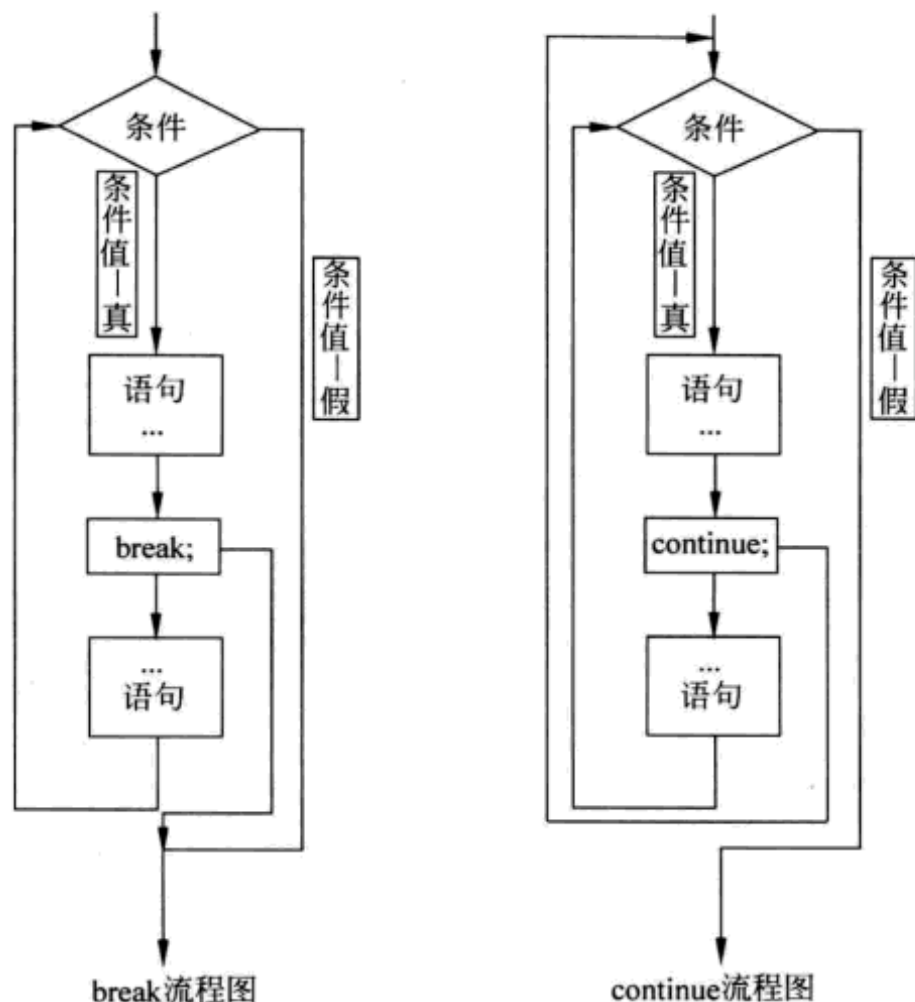


图 4.39 `break` 和 `continue` 流程图

图 4.39 所示就是 `break` 和 `continue` 在 `while` 循环结构中的流程图。对于 `do-while` 及 `for` 循环结构, `break` 和 `continue` 起的作用都是一样的。通过图 4.39 中 `break` 和 `continue` 的对比,可以发现它们之间的异同点。

- ❑ `break` 和 `continue` 的出现都可以使其后的语句停止执行,都使程序的执行流程转向其他的地方。
- ❑ `continue` 会使程序的执行流程转向条件判断,然后继续进入 `while` 循环体,只是暂时不执行 `continue` 语句之后的语句;而 `break` 会使程序的执行流程离开 `while` 循环体,执行 `while` 循环之后的语句。

下面写段程序来看看 `break` 和 `continue` 的区别。

```

#include<stdio.h>

int main()
{
    int num;

    //验证 break 的作用
    printf("test break:\n");
    num = 0;
    while(num < 6)
    {
        num++;
    }
}

```

```

        if(num == 3)
        {
            break;                                //使用 break 语句跳出循环结构
        }
        printf("num : %d\n", num);
    }

    //验证 continue 的作用
    printf("test continue:\n");
    num = 0;
    while(num < 6)
    {
        num++;
        if(num == 3)
        {
            continue;                            //使用 continue 语句跳出本次循环
        }
        printf("num : %d\n", num);
    }

    return 0;
}

```

在这个程序中出现了两段类似的 while 循环代码，都是输出变量 num 的值。唯一不同的就是在 num 等于 3 的时候，第一个 while 循环结构使用了 break 语句，第二个 while 循环结构使用了 continue 语句。

图 4.40 为程序的输出，“test break:” 之下的数字显示的是第一个 while 循环结构的输出。当 num 等于 3 的时候，break 语句使程序直接退出了 while 循环结构。所以，3 及 3 之后的数字都没有输出来。“test continue:” 之下的数字是第二个 while 循环的输出，我们看到其中缺少 3。这是因为当 num 等于 3 的时候，continue 语句使程序的执行流程转向条件判断，所以之后的 printf() 输出函数没有被执行。



```

#include<stdio.h>
int main()
{
    int num;

    //test break
    printf("test break:\n");
    num = 0;
    while(num < 6)
    {
        num++;
        if(num == 3)
        {
            break;
        }
        printf("num : %d\n", num);
    }

    //test continue
    printf("test continue:\n");
    num = 0;
    while(num < 6)
    {
        num++;
        if(num == 3)
        {
            continue;
        }
        printf("num : %d\n", num);
    }

    return 0;
}

```

Output (C:\WINDOWS\system32\cmd.exe):

```

test break:
num : 1
num : 2
test continue:
num : 1
num : 2
num : 4
num : 5
num : 6
请按任意键继续. . .

```

图 4.40 break 和 continue 程序验证

4.8 真正的程序——三种结构的揉和

前面已经介绍了 C 语言中的三种程序结构：顺序结构，判断结构和循环结构，它们各自有自己的使用格式和适用范围。在实际的程序中往往是各种程序结构相互嵌套、相互揉和，来完成丰富多彩的软件功能。

接下来看看这三种程序结构在真正的程序中到底都起到了哪些作用。下面看一个实际有用的例子。比如说，考试成绩出来了，班主任要你对本次成绩进行统计，要求如下。

- 统计不同等级的分数的人数，四个等级的分数分别为：60 以下、60~79、80~89，90~100，分别为差、中、良、优。
- 计算全班的平均分数。

你可以用笔在草稿纸上计算，也可以用计算器计算。如果掌握了我们之前讲的 C 语言的知识，完全可以写一段代码很快完成这个任务。

```
#include<stdio.h>

int main()
{
    int score;                //记录每一个学生的成绩
    int total_score;          //记录总成绩
    int total_students;       //全班的总人数
    int num;                  //记录已统计的人数
    double average_score;     //记录平均成绩

    int num_score_60;         //记录 60 以下的人数
    int num_score_60_79;      //记录 60~79 的人数
    int num_score_80_89;      //记录 80~89 的人数
    int num_score_90_100;     //记录 90~100 的人数

    total_score = 0;
    num_score_60 = 0;
    num_score_60_79 = 0;
    num_score_80_89 = 0;
    num_score_90_100 = 0;

    total_students = 50;      //假设全班有 50 个学生
    num = 0;

    while(num < total_students) //循环条件，检查已统计的人数是不是为总人数
    {
        scanf("%d", &score);
        //使用 scanf 函数从终端读入一个学生的成绩，然后保存在变量 score 中

        total_score += score; //统计总成绩
        if(score >= 90)        //统计 90~100 的人数
        {
            num_score_90_100++;
        }
        else if(score <=89 && score >= 80) //统计 80~89 的人数
        {
```



```

        num_score_80_89++;
    }
    else if(score <=79 && score >= 60)    //统计 60~79 的人数
    {
        num_score_60_79++;
    }
    else
    {
        num_score_60++;    //统计 60 以下的人数
    }

    num++;    //增加已统计的人数
}
average_score = (double)total_score/total_students;

printf("\n 统计结果:\n");
printf("优:%d\n",num_score_90_100);
printf("良:%d\n",num_score_80_89);
printf("中:%d\n",num_score_60_79);
printf("差:%d\n",num_score_60);
printf("平均分:%lf\n",average_score);
printf("\n");

return 0;
}

```

这是一段可以完成实际工作的程序，在这个程序中用到了许多之前学习到的知识，依次为变量命名、变量初始化、scanf()函数给变量赋值、自增运算、顺序结构、while 循环结构、if-else 分支结构、while 循环结构中揉和 if-else 结构、强制类型转换、printf()函数输出等。

其中与本章有关的是几种程序结构的使用。从这个程序中也可以看出，一个真实的程序往往不是单纯的一种程序结构就能完事，需要好几种结构的揉和。至于该如何去揉和，取决于要完成的任务和用程序解决问题的方法。

接下来看看这段程序的输出，然后分析分析为什么会有这样的输出，也就是程序的执行流程。程序的输出如图 4.41 所示，由于程序过长，只给出程序的输出窗口和部分代码的截图。



图 4.41 统计成绩程序

在输出窗口中先输入了 50 个学生的成绩，之间使用逗号隔开，然后按回车键，统计的结果就出来了，优 14 人、良 9 人、中 22 人、差 5 人，平均成绩 76.78 分，看来这次成绩不是很好啊！

下面来分析分析这段程序的输出，程序的真正执行是从 while 循环开始的，具体的执行步骤如下。

- ❑ 当已统计的人数 num 小于总人数 total_students 时，程序就从终端读入一个学生的分数，并把这个分数赋值给变量 score。
- ❑ 使用加后赋值运算把输入的学生成绩加到总成绩变量 total_score 上。
- ❑ 使用 if-else 分支结构判断该学生成绩属于哪个等级，并将相应等级的人数加 1。
- ❑ 最后，将统计的人数 num 加 1。

当上面的 4 个步骤执行 50 次的时候，50 个人的成绩也就处理完了，这个时候 num 的值变为 50，条件 num < total_students 不满足，不会进入 while 了。程序接下来就执行 while 循环之后的语句，使用总成绩 total_score 除以总人数 total_students 统计平均成绩，为了保证精度，这里使用了强制类型转换。最后输出所有的统计结果，如图 4.41 所示。

4.9 小 结

本章主要讲解了 C 语言中的三种程序结构：顺序结构、分支结构和循环结构，作为铺垫，还讲解了语句的概念及变量的作用域的相关知识。本章的重点就是三种程序结构的含义及具体的使用方式，难点是灵活使用三种结构的揉和写出真正有意义的程序。在下一章中，将介绍处理大量数据的数组数据类型。

4.10 习 题

【题目 1】 在一个语句块中定义两个相同名字的整型变量 num。

【分析】 按照 C 语言的规定，同一语句块中是不可以定义两个同样名字的变量的，这样会引起重名！要想避开重名的约束，可以使用语句块的嵌套，这样就可以在外部语句块中定义一个变量，在内部语句块中定义另一个同样名字的变量！这在 C 语言中是可以的，不过使用的时候得注意一下变量的作用域，仔细想想你正在使用的是哪个变量。

【核心代码】

```
{
    int num;
    ...
    {
        int num;
        ...
    }
    ...
}
```

【题目 2】 写程序验证一下哪些字符表示的是逻辑假，哪些字符表示的是逻辑真。

【分析】 C语言中的字符常用的有128个，其ASCII码值分别从0~127，我们可以给一个字符变量按照ASCII码的方式赋值，然后，将这个字符变量作为if-else判断结构的条件，判断为真的时候，输出真及相应的字符和ASCII码值；判断为假的时候，输出假及相应的字符和ASCII码值。

【核心代码】

```
char c;
for(c=0; c<127; c++)
{
    if(c)
    {
        printf("true:%c:%d\n",c,c);
    }
    else
    {
        printf("false:%c:%d\n",c,c);
    }
}

c = 127;
if(c)
{
    printf("true:%c:%d\n",c,c);
}
else
{
    printf("false:%c:%d\n",c,c);
}
```

【题目3】 使用尽量多的方式实现对60以下，60~79，80~100数字的分类统计。

【分析】 像这样具有分支的要求，就得使用判断结构了。这个题目中总共要求分成三部分：60以下，60~79，80~100，一般的思路是使用if-else if-else结构，其实，任何一种if结构都是可以实现的！这就是C语言的强大之处，不信可以试试。

【核心代码】

```
int num; //num 保存要进行分类的数据

int i;
int count_90 = 0;
int count_70 = 0;
int count_60 = 0;

for(i=0; i<100; i++) //假设有100个数要进行统计
{
    scanf("%d ",num); //获取num的值
    if(num<80)
    {
        count_70++;
        if(num<60)
        {
            count_60++;
        }
    }
}

count_70 = count_70 - count_60;
count_90 = 100 - count_70 - count_60;
```



```

int num;                                //num 保存要进行分类的数据

int i;
int count_90 = 0;
int count_70 = 0;
int count_60 = 0;

for(i=0; i<100; i++)                    //假设有 100 个数要进行统计
{
    scanf("%d ", num);                  //获取 num 的值
    if(num>80)
    {
        count_90++;
    }
    if(num>60 && num<80)
    {
        count_70++;
    }
    if(num<60)
    {
        count_60++;
    }
}

```

```

int num;                                //num 保存要进行分类的数据

int i;
int count_90 = 0;
int count_70 = 0;
int count_60 = 0;
for(i=0; i<100; i++)                    //假设有 100 个数要进行统计
{
    scanf("%d ", num);                  //获取 num 的值
    if(num>80)
    {
        count_90++;
    }
    else
    {
        if(num>60)
        {
            count_70++;
        }
        else
        {
            count_60++;
        }
    }
}

```

```

int num;                                //num 保存要进行分类的数据

int i;
int count_90 = 0;
int count_70 = 0;
int count_60 = 0;

for(i=0; i<100; i++)                    //假设有 100 个数要进行统计

```



```

{
    scanf("%d ", num);           //获取 num 的值
    if(num>80)
    {
        count_90++;
    }
    else if(num > 60)
    {
        count_70++;
    }
    else
    {
        count_60++;
    }
}

```

【题目 4】 写一个程序让计算机永无止境地说“I Love You!”,直到程序挂掉或者机器挂掉。

【分析】 计算机中可以重复干一件事情的是循环结构,当然 goto 语句也是可以的。循环结构中决定是否继续执行同样的程序语句得根据条件是否为真而定,为真则继续执行,为假则停止执行。所以,只要让循环结构的条件永远为真就可以一直不断地说“I Love You!”,最简单的方式就是在判断条件的地方直接写个 1 就行了。下面 4 种方式都可以让计算机永无止境地说“I Love You!”

【核心代码】

```

do
{
    printf("I Love You!\n");
}while(1);

```

```

while(1)
{
    printf("I Love You!\n");
}

```

```

for( ;1 ; )
{
    printf("I Love You!\n");
}

```

```

Label:
    printf("I Love You!\n");
goto Label;

```

【题目 5】 求最大约数: 给你一个数 1111155555, 写程序求出它的约数中最大的三位数。

【分析】 根据约数的定义: 对于一个整数 N , 除去 1 和它自身以外, 凡是能够整除 N 的数, 就是 N 的约数。要求一个数 N 的约数, 最简单的方式就是用 2 到 $N-1$ 之间的所有的数去除 N , 除得尽就是 N 的约数, 除不尽就不是 N 的约数。

本题中, 要求求出 1111155555 的约数中最大的三位数, 那么就不必从 2 开始一直除到 1111155555, 直接从 100~999 去找就可以了。还有个技巧, 要求的是最大的三位数, 所以从 999 开始除, 依次减小, 直到遇到第一个能除尽的数就是最大的三位数, 剩下的就不用管了, 反正不是最大的。

【核心代码】

```

long i;
int j;
printf("please input number: ");
scanf("%ld",&i);
for(j= 999; j>=100; j--)
{
    if(i%j == 0)
    {
        printf("The max factor with 3 digits in %ld is : %d.\n",i,j);
        break;
    }
}

```

【题目 6】 借书方案：假设现在有 5 本书，要借给三个小朋友，若每人每次只能借一本，那么有多少种不同的借法？写程序列出所有的借书方案。

【分析】 这道题从本质上来讲，是求出 5 的所有的组合情况。如果列出 5 的所有的排列组合情况，在数学中，就是一个一个地将第一本书借给 a 朋友，第二本书借给 b 小朋友，第三本书借给 c 小朋友；第一本书借给 a 朋友，第三本书借给 b 小朋友，第四本书借给 c 小朋友；第一本书借给 a 朋友，第四本书借给 b 小朋友，第五本书借给 c 小朋友；第一本书借给 a 朋友，第五本书借给 b 小朋友，第六本书借给 c 小朋友；第一本书借给 a 朋友，第二本书借给 c 小朋友，第三本书借给 b 小朋友等，这样一一列举，保证每一种情况不一样就可以。

在 C 语言中也是这样去一个一个地试的，只要保证每一种情况不一样就可以了。要想检查有没有列出所用的情况，计算一下 5 的组合就可以了，5 的组合等于 $5 \times 4 \times 3 = 60$ 。所有 60 种书籍分法如表 4.7 所示。

表 4.7 5 本书的分法

编号	A	B	C	编号	A	B	C	编号	A	B	C	编号	A	B	C
1	1	2	3	16	2	3	1	31	3	4	1	46	4	5	1
2	1	2	4	17	2	3	4	32	3	4	2	47	4	5	2
3	1	2	5	18	2	3	5	33	3	4	5	47	4	5	3
4	1	3	2	19	2	4	1	34	3	5	1	48	5	1	2
5	1	3	4	20	2	4	3	35	3	5	2	50	5	1	3
6	1	3	5	21	2	4	5	36	3	5	4	51	5	1	4
7	1	4	2	22	2	5	1	37	4	1	2	52	5	2	1
8	1	4	3	23	2	5	3	38	4	1	3	53	5	2	3
9	1	4	5	24	2	5	4	39	4	1	5	54	5	2	4
10	1	5	2	25	3	1	2	40	4	2	1	55	5	3	1
11	1	5	3	26	3	1	4	41	4	2	3	56	5	3	2
12	1	5	4	27	3	1	5	42	4	2	5	57	5	3	4
13	2	1	3	28	3	2	1	43	4	3	1	58	5	4	1
14	2	1	4	29	3	2	4	44	4	3	2	59	5	4	2
15	2	1	5	30	3	2	5	45	4	3	5	60	5	4	3

【核心代码】

```
//使用 a, b, c 表示三个小朋友分别借到第几本书
```

```

int a;
int b;
int c;

//count 统计有几种借书方案
int count = 0;
printf("distribute books to 3 readers\n");
for( a= 1; a<=5; a++)      //a 小朋友从 1 到 5 开始一个一个地试
{
    for(b=1; b<=5; b++)      //b 小朋友从 1 到 5 开始一个一个地试
    {
        //c 小朋友从 1 到 5 开始一个一个地试并且 a 没有借到 b 已经借到的书
        for(c = 1; a != b && c <= 5 ; c ++ )
        {
            //c 没有借到 a 已经借到的书也没有借到 b 已经借到的书
            if(c !=a && c != b)
            {
                printf("%3d:%d, %d, %d\n",++count,a,b,c);
            }
        }
    }
}

```

【题目 7】 有限 5 位数：个位数为 6 且能被 3 整除的 5 位数共有多少个？

【分析】 要解决这个问题，首先必须有一个 5 位数，而且其个位数为 6，怎么得到这个数呢？可以将一个任意 4 位数（1000 到 9999）乘以 10，然后加上 6 就得到了这样一个 5 位数，将这个 5 位数对 3 求余就知道能否被 3 整除了，构造这个 5 位数的过程如图 4.42 所示。

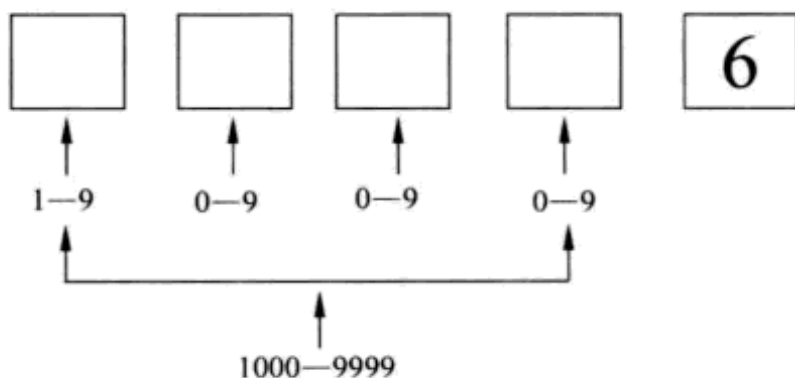


图 4.42 以 6 结尾的 5 位数

【核心代码】

```

long int i;
int count = 0;
for(i=1000; i<9999;i++)
{
    if(!((i*10+6)%3)) count++;
}
printf("total number is %d", count);

```

【题目 8】 捕鱼和分鱼：A、B、C、D、E 5 个人在某夜合伙捕鱼，到第二天凌晨都疲惫不堪，于是各自找地方睡觉。日上三杆，A 第一个醒来，他将鱼分为 5 份，把多余的一条鱼扔掉，拿走自己的一份。B 第二个醒来，也将鱼分为 5 份，把多余的一条扔掉，拿走自己的一份。C、D、E 依次醒来，也按照同样的方法拿鱼。问他们合伙至少捕了多少条鱼？

【分析】 从5个人的分鱼过程中，可以总结出这样的规律，每次将鱼分为5份，总是会多出一条鱼。这个规律可以用这样的数学公式表示，假设分鱼之前鱼的总数为 X ，那么， $X-1$ 是可以被5整除的，分完之后，余下的鱼是 $4*(X-1)/5$ 。只要我们不断地尝试，每次令 $X=4*(X-1)/5$ ，当 X 可以被5整除的时候， X 就是最小的捕鱼数目了！由于 $X-1$ 必须可以被5整除，所以从6开始尝试。这个分鱼过程满足图4.43所示的式子，分鱼之前为 $X1$ ，分鱼之后为 $X2$ 。

$$\frac{X1 - 1}{5} * 4 = X2$$

图4.43 分鱼问题

【核心代码】

```
int n, i, x, flag = 1;
for(n = 6; flag; n++)
{
    for(x=n, i=1 &&flag; i<=5; i++)
    {
        if( (x-1)%5 == 0) x= 4*(x-1)/5;
        else flag = 0;
    }
    if(flag) break;
    else flag = 1;
}
```

【题目9】 反序数：设 N 是一个4位数，它的9倍恰好是其反序数，写程序求出 N 的具体值是多少。

【分析】 反序数的概念：反序数就是将整数的数字倒过来所形成的数。例如，1234的反序数就是4321。使用 $i*10^3 + j*10^2 + k*10 + l = t$ 表示任意的一个4位数的形式，则这个数的千位数 $i = t/1000$ 、百位数 $j = t/100\%10$ 、十位数 $k = t/10\%10$ 、个位数 $l = t\%10$ ；那么 t 的反序数 $m = l*10^3 + k*10^2 + j*10 + i = t\%10*10^3 + t/100\%10*10^2 + t/100\%10*10 + t/1000$ 。

只要求出 $t*9$ 等于 $t\%10*10^3 + t/100\%10*10^2 + t/100\%10*10 + t/1000$ 中的 t ，就找到所要求的数字了。

【核心代码】

```
int t;
for(t=1002; t<1111; t++)
{
    if(t%10*1000 + t/10%10*100 + t/100%10*10 + t/1000 == t*9)
        printf("the number is : %d\n", t);
}
```

【题目10】 百钱百鸡问题：中国古代有个数学家张丘建在他的《算经》中提出了一个著名的“百钱百鸡问题”：鸡翁一，值钱五，鸡母一，值钱三，鸡雏三，值钱一，百鸡百钱，问翁、母、雏各几何？

【分析】 假设100钱可以买到公鸡 x 个，母鸡 y 个，小鸡 z 个。根据题目描述 x 、 y 、 z

必须满足下面的方程组：

$$\begin{cases} 5x+3y+z/3=100 \\ x+y+z=100 \end{cases}$$

这是一个三元一次方程组，只要求出这个方程组的解就可以知道 100 钱可以买到的公鸡、母鸡和小鸡的个数分别是多少了。从实际出发，我们知道鸡个数都是整数，不可能出现半只鸡之类的事情，另外 100 钱最多可以买到 20 只公鸡、最多可以买到 33 只母鸡。通过这些实际条件可以方便我们编程。

【核心代码】

```
int x, y, z, j=0;
printf("possible plans : \n");
for(x=0; x<=20; x++)
{
    for(y=0; y<=33; y++)
    {
        z = 100 - x - y;
        if(z%3 == 0 && 5*x + 3*y + z/3 == 100)
        {
            printf("%2d: cock=%2d hen = %2d, chicken = %2d\n", ++j, x, y, z);
        }
    }
}
```

【题目 11】 有 30 个人，其中有男人，女人和小孩，在一家饭馆吃饭共花了 50 先令，每个男人花 3 先令，每个女人花 2 先令。每个小孩花 1 先令。问男人，女人和小孩各有多少人？

【分析】 此题有点类似于“百鸡百钱”问题，假设有男人 x 个，女人 y 个，孩子 z 个。则可以使用下面的三元一次方程组表示这个问题：

$$\begin{cases} 3x+2y+z=50 \\ x+y+z=30 \end{cases}$$

只要求出这个方程组的解，就可以知道这次聚餐有多少个男人、女人和小孩了。根据具体情况，可以知道人的个数都是整数，并且这次聚餐中男人最多不超过 10 个，女人最多不超过 15 个，小孩最多不超过 30 个。通过这些实际条件，可以方便我们编程，确定到底要对哪些数进行验证。

【核心代码】

```
int men;
int women;
int childs;

for(men=1; men<=10; men++)
{
    for(women=1; women<=15; women++)
    {
        for(childs=1; childs<=30; childs++)
        {
            if(men+women+childs==30 && men*3+women*2+childs*1==50)
            {
```

```

        printf("men = %d、women = %d、childs = %d\n", men, women, childs);
    }
}
}

```

【题目 12】 公约数和公倍数：编写程序，求任意两个正整数的最大公约数（GCD）和最小公倍数（LCM）。

【分析】 如果直接使用求最大公约数和最小公倍数的一般数学方法，很难实现！在计算机中有一种专门求最大公约数和最小公倍数的方法——辗转相除法。这个方法是基于下面这样的思想。

设两数为 a 、 b ($b < a$)，求最大公约数 (a, b) 的步骤如下：用 b 除 a ，得 $a = b * q + r_1$ ($0 \leq r_1$)。若 $r_1 = 0$ ，则 $(a, b) = b$ ；若 $r_1 \neq 0$ ，则再用 r_1 除 b ，得 $b = r_1 * q + r_2$ ($0 \leq r_2$)。若 $r_2 = 0$ ，则 $(a, b) = r_1$ ，若 $r_2 \neq 0$ ，则继续用 r_2 除 r_1 ……如此下去，直到能整除为止。其最后一个非零余数即为 (a, b) 。

有了最大公约数以后，直接使用公式 $a * b$ 除以最大公约数就可以得到 a 和 b 的最小公倍数了。

【核心代码】

```

int a, b, num1, num2, temp;
printf(" please input the number : ");
scanf("%d%d", &num1, &num2);
if(num1 > num2)
{
    //进行数据交换，确保被除数大于除数
    temp = num1;
    num1 = num2;
    num2 = temp;
}

a = num1;
b = num2;
while(b != 0)    //辗转相除法
{
    temp = a % b;
    a = b;
    b = temp;
}

printf("The GCD of %d and %d is %d \n", num1, num2, a);
printf("The LCM of them is %d \n", num1 * num2 / a);

```

【题目 13】 有 4 位同学中的一位做了好事没留名，表扬信来了之后，校长问这 4 位中谁做的好事。

A 说：不是我。

B 说：是 C。

C 说：是 D。

D 说：C 胡说。

已知 3 个人说的是真话，1 个人说的是假话。现在要根据这些信息，找出做了好事的人。

【分析】 将 4 人用 1、2、3、4 编号，分别列举各种情况来解决问题。变量 x 表示做好事者的编号序号，则 x 从 1 到 4，4 个人所说的话分别写成：

A 说: $x \neq 1$

B 说: $x = 3$

C 说: $x = 4$

D 说: $x \neq 4$

当这 4 个逻辑式的值相加等于 3 时, 也就是 3 个人说的话是真的, 即可得到答案。

【核心代码】

```
int x;
for (x = 1; x <= 4; ++x)
{
    if ((x != 1) + (x == 3) + (x == 4) + (x != 4) == 3)
    {
        printf("the good people is %c\n.", (char)(64 + x));
        //字母 a 的 ASCII 码值为 64
    }
}
```

【题目 14】 数字之和: 求 100~1000 之间有多少个数字之和为 5 的整数。

【分析】 要算出数字之和为 5 的整数, 最常规的方法就是逐个判断从 100 到 1000 之间的每个数是否符合条件。可以使用一个 for 语句从 100 至 1000 来判断, 注意合理使用 / 和 % 运算符来取每位数字, 然后相加进行判断。也可以用 3 个 for 语句来表示百位、十位、个位数, 然后判断和是否为 5。

由于要求这个数各个位上的和为 5, 所以 100 到 103 这几个数就不考虑了, 因为这几个数根本不可能所有位上的数加起来得到 5, 另外, 500 以上也不用考虑了。所以, for 循环从 104 开始, 到 501 就结束了, 不必从 100 到 1000。

【核心代码】

```
int i;
for(i=104; i<501; i++)
{
    if((i%10)+(i/100)+((i/10)%10)==5) //依次取个位、百位、十位判断
    {
        printf("%d\n", i);
    }
}
```

【题目 15】 一个自然数被 8 除余 1, 所得的商被 8 除也余 1, 再将第二次的商被 8 除后余 7, 最后得到一个商为 a 。又知这个自然数被 17 除余 4, 所得的商被 17 除余 15, 最后得到一个商是 a 的 2 倍。求这个自然数。

【分析】 根据题意, 可设最后的商为 i (i 从 0 开始取值), 可以列出下面的相等关系式: $((i*8+7)*8+1)*8+1=((2*i*17)+15)*17+4$, 再用试探法求出商 i 的值。

【核心代码】

```
int men;
int i;
for(i=0;; i++) //试探商的值
{
    if(((i*8+7)*8+1)*8+1==((2*i*17)+15)*17+4)
    {
        //逆推判断所取得的当前 i 值是否满足关系式
        //若满足则输出结果
    }
}
```



```
        printf("The required number is: %d\n", (34*i+15)*17+4);
        break;           //找到结果之后就退出循环
    }
}
```

【题目 16】 5 个学生 A, B, C, D, E 参加某一项比赛。甲, 乙两人在猜比赛的结果。甲猜的名次顺序为 A, B, C, D, E, 结果没有猜中任何一个学生的名次, 也没猜中任何一对相邻名次 (所谓相邻名次, 是指其中一对选手在名次上邻接, 如 1 和 2 或 2 和 3 等)。乙猜的名次顺序为 D, A, E, C, B, 结果猜中了两个学生的名次, 并猜对了两对学生名次是相邻的。求最终的排名顺序。

【分析】 使用穷举法 (列出所有的可能), 然后依次判断题目所给的条件, 对于甲是否猜中的条件, 直接进行判断即可, 对于乙是否猜中的条件, 可以利用等式的值为 1 来判断。

【核心代码】

```
int a,b,c,d,e;
for(a='A';a<'F';a++)           //对第一名进行穷举
for(b='A';b<'F';b++)           //对第二名进行穷举
if(a!=b)                         //保证不出现一人得多个名次
for(c='A';c<'F';c++)           //对第三名进行穷举
if((a!=c)&&(b!=c))              //保证不出现一人得多个名次
for(d='A';d<'F';d++)           //对第四名进行穷举
if((a!=d)&&(b!=d)&&(c!=d))      //保证不出现一人得多个名次
for(e='A';e<'F';e++)           //对第五名进行穷举
if((a!=e)&&(b!=e)&&(c!=e)&&(d!=e)) //保证不出现一人得多个名次
if((a!='A')&&(b!='B')&&(c!='C')&&(d!='D')&&(e!='E'))
//判断甲的条件,即无一个名次是猜对的
if((a=='D')+(b=='A')+(c=='E')+(d=='C')+(e=='B')==2)
//判断乙的条件,即猜中两人,利用相等返回 1,不相等返回 0 实现
printf("%c,%c,%c,%c,%c\n",a,b,c,d,e);
```

【题目 17】 企业发放的奖金根据利润提成。利润低于或等于 10 万元时, 奖金可提 10%; 利润高于 10 万元, 低于 20 万元时, 低于 10 万元的部分按 10%提成, 高于 10 万元的部分, 可提成 7.5%; 利润在 20 万到 40 万之间时, 高于 20 万元的部分, 可提成 5%; 利润在 40 万到 60 万之间时高于 40 万元的部分, 可提成 3%; 利润在 60 万到 100 万之间时, 高于 60 万元的部分, 可提成 1.5%, 利润高于 100 万元时, 超过 100 万元的部分按 1%提成。从键盘输入当月利润 I, 求应发放奖金总数。

【分析】 发放奖金是根据利润的不同层次来进行处理的, 对于一个利润, 首先得判断它最高符合哪个层次, 然后再对之后的不同层次的利润进行不同的处理, 最后再计算出所有层次的总利润。所有层次的利润计算表如表 4.8 所示。

表 4.8 利润计算表

利润范围 (万元)	百 分 率	奖 金
$X \leq 10$	10%	$0.1 * X$
$10 < X \leq 20$	7.5%	$0.075 * (X - 10) + 10 * 0.1$
$20 < X \leq 40$	5%	$0.05 * (X - 20) + 10 * 0.075 + 10 * 0.1$
$40 < X \leq 60$	3%	$0.03 * (X - 40) + 20 * 0.05 + 10 * 0.075 + 10 * 0.1$
$60 < X \leq 100$	1.5%	$0.015 * (X - 60) + 20 * 0.03 + 20 * 0.05 + 10 * 0.075 + 10 * 0.1$
$X > 100$	1%	$0.001 * (X - 100) + 40 * 0.015 + 20 * 0.03 + 20 * 0.05 + 10 * 0.075 + 10 * 0.1$

【核心代码】

```
int level;
float a;
float sum=0;
printf("the number:");
scanf("%f",&a);

while(a>0){
    //判断 a 的等级
    level=a>100?6:(a>60?5:(a>40?4:(a>20?3:(a>10?2:1)));
    //对每个等级分别进行处理
    switch(level){
        case 1:
            sum=sum+a*0.1;
            a-=10;
            break;
        case 2:
            sum=sum+(a-10)*0.75;
            a=10;
            break;
        case 3:
            sum=sum+(a-20)*0.05;
            a=20;
            break;
        case 4:
            sum=sum+(a-40)*0.03;
            a=40;
            break;
        case 5:
            sum=sum+(a-60)*0.15;
            a=60;
            break;
        case 6:
            sum=sum+(a-100)*0.01;
            a=100;
            break;
    }//switch
};//while
printf("amount of money is: %.2f\n\n",sum);
```

第3篇 复杂数据的表示

- ▶▶ 第5章 数组
- ▶▶ 第6章 字符数组——字符串
- ▶▶ 第7章 指针
- ▶▶ 第8章 结构体
- ▶▶ 第9章 共同体类型
- ▶▶ 第10章 枚举类型



第 5 章 数 组

在程序设计中，有的时候需要保存很多同样类型的数据。如果都声明定义成简单类型的数据，会显得非常麻烦，而且当数据量大到一定程度的时候，仅仅依靠使用简单类型是根本没法实现的。C 语言提供了一种更简单、更方便的数据类型——数组，它可以用来保存多个同样数据类型的数据。

5.1 数 组 简 介

作为 C 语言中的一种高级数据类型，数组有比整型、浮点型和字符型这些简单数据类型更强大的功用，同时也与这些基本数据类型有密不可分的关系。本章就来剖析一下数组这种高级数据类型。

5.1.1 数组的用途

在介绍数组的用途之前，先来看一个例子。例如，要计算一家人的平均年龄。假设你的家庭成员有父亲、母亲、弟弟、妹妹和你。用程序实现的时候，可以简单地定义如下几个变量来保存你的家人的年龄。

```
int father;  
int mother;  
int brother;  
int sister;  
int myself;
```

好，完全可以这样实现：定义好变量并赋值以后，相加，然后除以家庭成员的个数，就可以完成任务了。但是，要是让你去计算你们村的所有人的平均年龄，还可以这样做吗？要是计算的是全市、全省，乃至全国人口的平均年龄呢？难道要定义 13 亿个变量吗？如果不嫌麻烦可以试试！

当需要保存很多同样类型的数据的时候，数组就是最好的选择。它生来就是干这种事情的，不用白不用。

可以打个比方，小学生每天上学需要从家里把书带到学校去。起初，只需要几本书就够了，可以直接夹在咯吱窝带去学校，后来发现要带的书越来越多，这个时候就需要一个书包了。

这里“书”就相当于程序中的“数据”，用“咯吱窝”夹着带书，就相当于程序中用“简单数据类型”保存数据。书多了，也就类似于要保存的数据多了，“咯吱窝”也就是简

单数据类型不能满足需要了。这个时候，书包，也就是数组就出现了。图 5.1 展示了带书和保存数据之间的关系，体现了数组的作用。

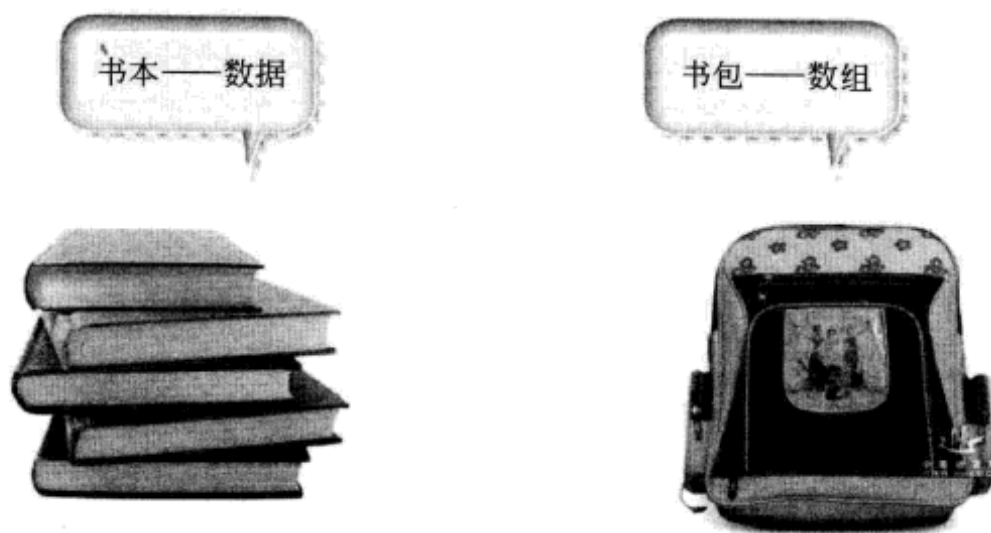


图 5.1 数组作用图示

说了那么多，大家只需要记住一点就行：数组就是用来保存相同类型数据的。如果要保存不同类型的数据，就要用到另外一个知识点——结构体了，这里先不介绍，以后会详细讲解的。

知道了数组是用来保存同样类型数据的一种数据类型，接下来看看三个问题。

- 数组类型用 C 语言是如何表示的？
- 如何把同样类型的数据保存到数组类型的变量中去？
- 如何使用保存到数组变量中的数据呢？

带着这三个问题，开始我们的数组之旅吧！

5.1.2 数组变量的定义

数组也是一种数据类型，也可以像基本数据类型那样定义数组变量。当然，与基本数据类型不同的是数组没有常量的说法，因为不能直接写出类似于基本数据类型常量 3、3.4、'a' 这样的数组常量。

现在看看 C 语言中的数组变量是如何声明定义的，其声明定义如下：

数据类型 数组名[表达式]

在这个变量声明定义表示中，“数据类型”就是一种数据类型的关键字，如 int、float、char 等。“数组名”和之前所讲的变量名一样，只要遵循标识符的命名规范就行。“[]”（中括号）是 C 语言中数组表示和运算的固有符号，不能改变。中括号中的“表达式”，可以是一个常量或者常量表达式，其值必须固定，不能使用值不固定的变量或者变量表达式。

例如，下面就是一些合法的 C 语言数组的声明定义：

```
int    radius[20];
float  result[100];
char   name[5];
```


看完了数组声明定义的表示以后，下面来看看数组的声明定义中，各个部分都是做什么的。

- 数据类型：用来表示数组中可以保存的数据的类型。因为在数组声明定义的时候，这个类型已经固定了，所以数组只能保存同样类型的数据。
- 数组名：类似于之前所讲的变量名。当要使用数组中的元素时，需要这个变量名。
- 表达式：这个表达式的值表示定义的数组中将来最多可以保存多少个数据。所以，它必须是一个有固定值的常量或者常量表达式。

例如，声明一个保存半径的整型数据的数组：

```
int    radius[20];
```

它的作用就是声明一个叫做 `radius` 的整型数组变量。这个数组只能用来保存整型数据，而且最多只能保存 20 个整型数据。

5.2 数组变量初始化和赋值

在知道了如何声明定义一个数组变量以后，接下来的事情就是如何使用这个数组来保存数据了。这就好比你买了一个新书包，接下来的事情就是考虑如何把书装进去了，到底是竖着放，还是横着放？随你的便！

C 语言中给数组中放数据可没有横着放和竖着放一说。但是，它有两种方式可以完成保存数据到数组的工作，分别是数组初始化和数组赋值。

5.2.1 数组的初始化

先来看看数组变量的初始化。数组变量初始化的 C 语言表示如下：

```
数据类型 数组名[表达式] = {值, 值, 值, ..., 值};
```

例如，要对一个可以保存 5 个整型数据的整型数组变量 `radius` 进行初始化的时候，可以使用下面的 C 语言表示：

```
int radius[5] = {0, 1, 2, 3, 4};
```

在进行数组初始化的时候，数组声明定义形式中的“表达式”可以省略。也就是说，上面的数组变量初始化的 C 语言表示可以简化成下面的样子了：

```
数据类型 数组名[ ] = {值, 值, 值, ..., 值};
```

之所以可以这样用，是因为进行数组初始化的时候，往数组中保存的数据的个数就是上面大括号中的值的个数，已经确定了。数组声明定义形式中的“表达式”就可以省略不写了。

```
int radius[ ] = {0, 1, 2, 3, 4};
```

这样的数组初始化在 C 语言中也是合法的。

⚠注意：数组的这两种初始化的形式只能在数组声明定义的时候和数组的声明定义一起使用，在其他地方使用都是不对的。

5.2.2 数组的下标

在C语言中，要给数组进行赋值的时候，不能像给数组初始化那样使用一个大括号一次性给数组中的元素全都赋值，而应该给数组中的每个元素一个一个地赋值。要给数组中的某一个元素赋值，必须先找这个元素，用什么找呢？用数组的下标。

所以，在给数组进行赋值之前，先来看一个新的概念——数组的下标。C语言中找数组中的某一个元素的表示方法如下：

数组名[下标]

这个表示方法中，“数组名”就是已经声明定义的数组的数组名称。“[]”（中括号）和数组声明定义中的中括号意义不同，这里的中括号是一个取数组元素的运算符。“下标”表示要取的数组元素在数组中的位置，它可以是一个整型常量、整型变量或者是一个具有整型值的表达式。“数组名[下标]”这个表示被叫做取数组元素运算表达式。

数组是按照图 5.2 所示的线性顺序的结构来保存数据的。图 5.2 中每个类似于格子的小方框就是用来存放数据的地方，每个数据就是数组中的元素，小方框下边的数字 0、1、2、…、 $n-1$ 、 n 就是数组的下标，上面取数组元素表达式中的“下标”表示的就是图 5.2 中的下标 0、1、2、…、 $n-1$ 、 n 。

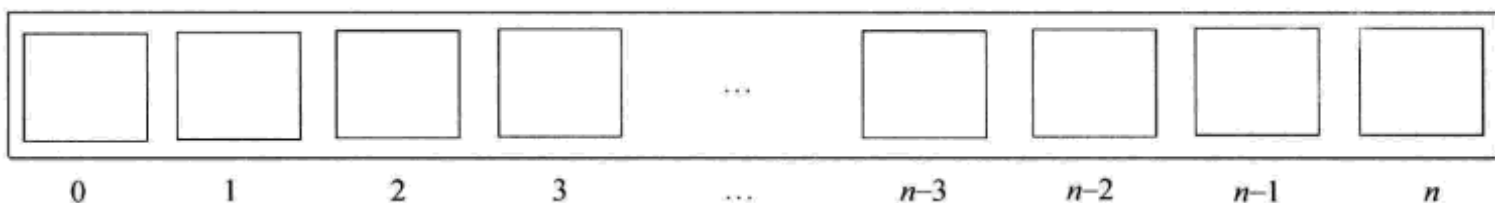


图 5.2 数组结构图

⚠注意：C语言中数组的下标是从 0 而不是从 1 开始的。下标为 0 的数组元素其实就是数组中的第一个元素。

5.2.3 给数组赋值

取数组元素表达式，就相当于打开图 5.2 中某一个小方格的盖子。当使用下面所示的赋值表达式的时候就是往打开盖子的格子中放入所赋的值。

数组名[下标] = 数值;

这是一个赋值表达式，等号之前部分是我们讲的取数组元素运算，相当于一个变量。等号之后部分是一个具有数值的常量、变量或者表达式。

例如，想往可以放 5 个数据的整型数组中的第 2 个位置（小方格）处中放一个数字 3，就可以使用上面所讲的数组赋值表达式：

```
int radius[5];
radius[1] = 3;
```

先声明定义一个长度为 5 的整型数组 `radius`，之后使用数组赋值表达式给数组中位置为 2 的地方存储一个整数 3。由于数组的下标是从 0 开始的，所以下标 1 表示的就是数组中位置为 2 的地方。当用上面的 C 语言赋值完成以后，数组 `radius` 就变成图 5.3 所示的样子了。

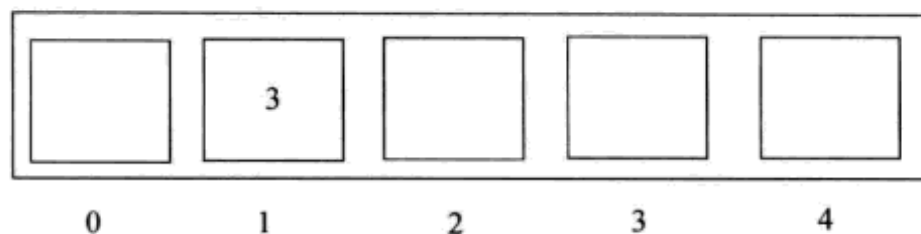


图 5.3 数组赋值

其实，数组初始化也是一种赋值，只不过是在数组刚声明定义的时候就用大括号一次性给所有的“格子”中都放入了数据。下面的初始化表示：

```
int radius[5] = {0, 1, 2, 3, 4};
```

和数组赋值语句：

```
int radius[5];
radius[0] = 0;
radius[1] = 1;
radius[2] = 2;
radius[3] = 3;
radius[4] = 4;
```

是等价的，完成的事情是一样的，都是往数组 `radius` 的所有格子中依次放入 0、1、2、3、4，其效果如图 5.4 所示。

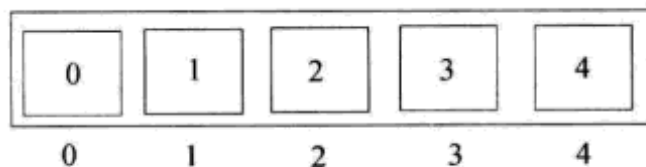


图 5.4 数组初始化和赋值等价

5.2.4 数组元素的引用

无论是数组使用初始化还是使用赋值的方式保存数据，目的都是为了以后方便地去使用这些数据。那么该如何从数组中拿出想要的数据呢？这就是数组元素的引用。

数组元素的引用是使用取数组元素运算：

数组名[下标]

这个运算相当于打开图 5.4 中小方格的盖子。

例如：

```
radius[1]
```

就是打开数组 `radius` 的第 2 个小格子的盖子。在打开小格子的盖子之后，就可以使用方格中的数据了。可以使用这个数据进行赋值运算、基本数学运算、关系运算、逻辑运算、输出等各种操作。当然，使用完数组中的数据还得将其放回去，以备下次使用，就像你上完课还得把书装进书包带回家一样。

可以使用下面的例子对数组中的元素求和，每加一个数，就将和的结果输出。

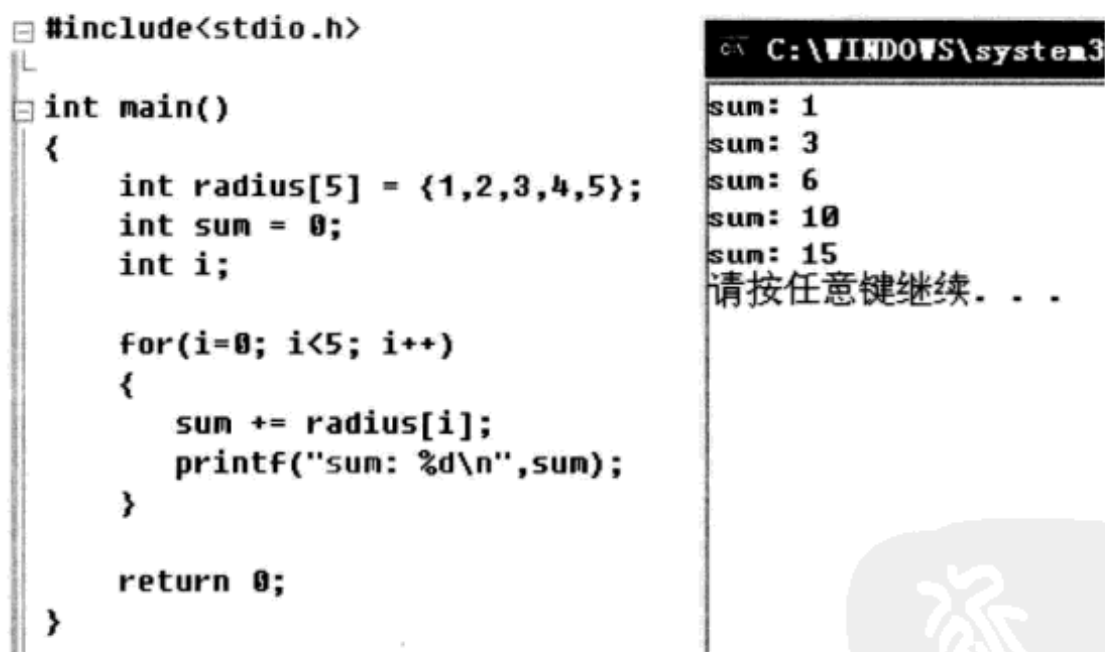
```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int sum = 0;
    int i;

    for(i=0; i<5; i++)
    {
        sum += radius[i];           //把下标为 i 的数组元素加到 sum 上
        printf("sum: %d\n",sum);
    }

    return 0;
}
```

程序中使用了一个 `for` 循环。依次从下标 0 到 4 引用数组中的元素，然后使用加后赋值运算将每个数组元素和 `sum` 相加并赋值给 `sum`，然后输出 `sum` 的值。程序的输出结果如图 5.5 所示。



```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int sum = 0;
    int i;

    for(i=0; i<5; i++)
    {
        sum += radius[i];
        printf("sum: %d\n",sum);
    }

    return 0;
}
```

```
C:\WINDOWS\system32
sum: 1
sum: 3
sum: 6
sum: 10
sum: 15
请按任意键继续...
```

图 5.5 数组元素引用求和

5.3 二维数组

我们已经知道，数组是用来存储同一种类型的数据的。之前看到的都是用数组存储简单数据类型的数据。其实，数组还可以用来存储数组类型的数据，就是说数组中的元素本身就是一个数组。

5.3.1 数组的维

数组中的元素既可以是简单数据类型的数据，也可以是数组类型的数据。这样数组元素就有了数组嵌套的关系，数组元素可以是数组，数组元素中的数组的元素也可以是数组……依此类推就形成了不同嵌套深度的数组。这就好像我们送礼物的时候，小盒子套中盒子，中盒子再套大盒子一样：4 个装有糖果的小盒子放到一个中号盒子中，然后 5 个中号的盒子，放到一个大号的盒子中，最后把大盒子装到礼包里送给亲朋好友。

这样的嵌套深度，称之为数组的“维”。对于元素只是简单数据类型的数组，称为一维数组。在上一节中介绍的，以及像下面这样的保存简单数据类型数据的数组都是一维数组。

```
int    radius[20];
float  result[100];
char   name[5];
```

数组中的嵌套深度加一其维数就加一。如果一维数组中保存的元素不是简单数据类型数据，而是一维数组数据时，其嵌套深度相当于加了一，称之为二维数组。类似地，如果二维数组的一维数组元素中的元素保存的不是简单数据类型，而是一维数组数据，就形成了三维数组。依此类推，可以形成四维数组、五维数组、……、 n 维数组。

5.3.2 二维数组表示和含义

在 C 语言中，一维数组和二维数组是较常用的数组类型。前面介绍的都是一维数组，本小节详细介绍二维数组。

按照数组维数的定义，二维数组就是用来保存一维数组数据的数组。其中的一维数组数据又是用来保存简单数据类型的数据的数组。我们已经知道一维数组的 C 语言表示，二维数组在 C 语言中又是如何表示的呢？其表示如下：

```
数据类型 数组名 [表达式 1] [表达式 2]
```

其中，“数据类型”、“数组名”、“表达式 1”和“表达式 2”的含义和一维数组的 C 语言表示的含义是相同的。“数据类型”就是一种数据类型的关键字，如 `int`、`float`、`char` 等。“数组名”是数组变量标识符。中括号中的“表达式”是一个具有常量值的表达式。

二维数组表示和一维数组表示唯一不同的就是，其表达式中有两个“[]”和两个“表达式”。这正是二维数组的特点，两个中括号和两个表达式分别表示二维数组的两个维。第一个中括号中的表达式表示的是二维数组中最多可以保存多少个一维数组元素，第二个中括号中的表达式表示二维数组中保存的每个一维数组最多可以保存多少个基本数据类型的数据。

例如，要保存 m 个一维数组元素，并且这 m 个一维数组中都可以保存 n 个整型数据。给这个二维数组取名为 `Matrix`，这里的 m 和 n 都是常量或者常量表达式。可以用下面的方法表示：

```
int Matrix[m][n];
```

进行这样的声明定义之后, 计算机就会按照图 5.6 所示的样子来保存数据了。在图 5.6 中纵向的大方格用来保存的是 m 个一维数组, 大方格中的横向的小方格用来保存每个一维数组中的 n 个整型数据。

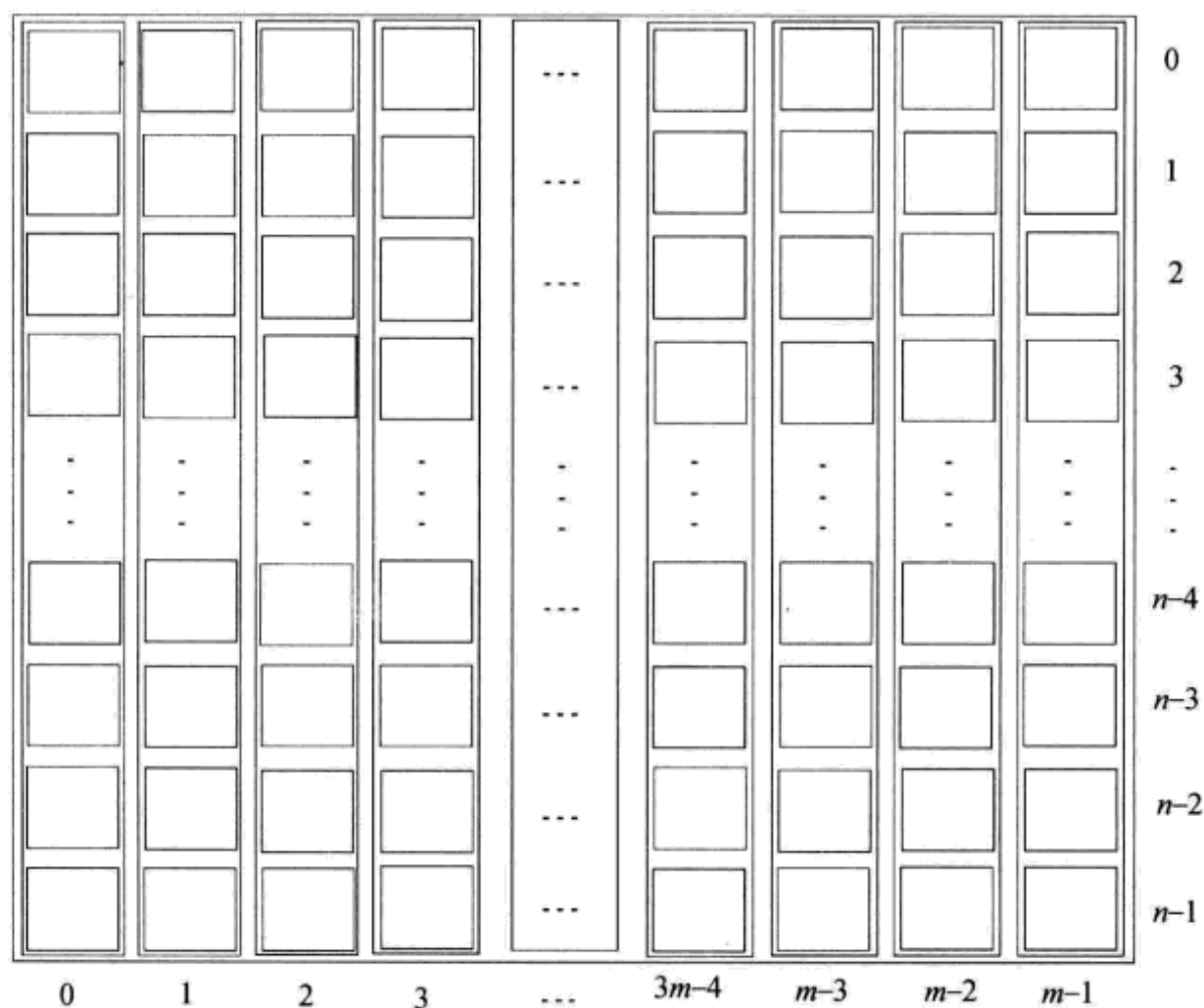


图 5.6 二维数组

5.3.3 二维数组的初始化

类似于一维数组, 要想往二维数组中保存数据, 也是初始化和赋值两个方法。二维数组初始化的 C 语言表示, 如下所示:

```
数据类型 数组名[表达式 1][表达式 2] = {数值, 数值, 数值, 数值, ..., 数值, 数值, 数值};
```

和

```
数据类型 数组名[表达式 1][表达式 2] = { {数值, ..., 数值}, {数值, ..., 数值}, ..., {数值, ..., 数值} };
```

两个表达式是等效的, 只不过第二个比较直观, 因为外面大括号中的每个内置括号表示的就是二维数组中的每个一维数组元素。

另外, 两个表达式中的“数值”的个数最多为“表达式 1”乘以“表达式 2”个。如果数值的个数不够, 就相当于数组中某些地方没有保存数据, 如果太多, 编译器就会报错。可以写一个简单的二维数组的例子, 如下所示:

```
#include<stdio.h>

int main()
{
    int Matrix[3][2] = {1,2,3,4,5,6,7,8};
}
```

其中，二维数组 Matrix 可以保存 $3 \times 2 = 6$ 个整型数值。但是，我们给其初始化了 8 个整型数值，当进行编译的时候就会出现图 5.7 所示的错误。

```
1>----- 已启动全部重新生成: 项目: test, 配置: Debug Win32 -----
1>正在删除项目“test”(配置“Debug|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>g:\c语言的书\c语言的书\test\t.c(5) : error C2078: too many initializers|
1>生成日志保存在“file://g:\C语言的书\C语言的书\test\Debug\BuildLog.htm”
```

图 5.7 数组初始化太多值

类似于一维数组，在对二维数组进行初始化的时候，初始化表示中的“表达式”的值可以省略不写，交给计算机根据所初始化的值，自己去判断数组可以保存多少个元素。但是，只能省略“表达式 1”的值，不然计算机无法判断图 5.6 中的 m 和 n 的值，也就不知道你到底声明定义的是一个什么样的二维数组。

例如，下面就是一个正确的省略“表达式 1”值的程序例子。

```
#include<stdio.h>

int main()
{
    int Matrix[][2] = {1,2,3,4,5,6,7,8};
}
```

经过这样的赋值以后，系统就知道数组 Matrix 中可以存 4 个一维数组，每个一维数组中有两个值，就像图 5.8 所示的那样。

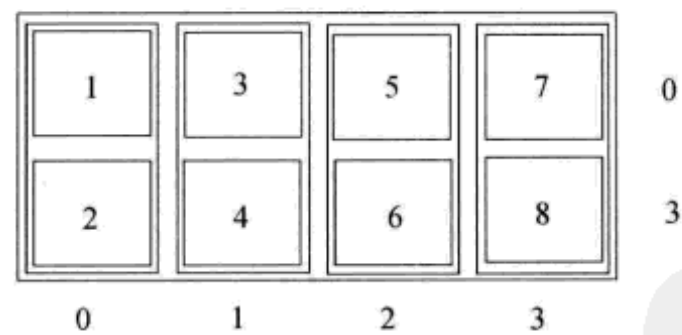


图 5.8 Matrix 数组初始化之后的结果

5.3.4 二维数组的赋值

当声明定义了一个二维数组之后，这个数组的样子如图 5.6 所示。总共有 $m \times n$ 个格子，可以使用初始化给每个格子中放入数据，也可以使用赋值给某个格子中放入数据。依然可以使用取数组元素表达式，来打开二维数组中的某个格子的盖子，然后使用赋值语句给这个打开盖子的格子放入数据元素。

二维数组的取数组元素运算表达式如下所示：

```
数组名[下标 1][下标 2]
```

在这个表达式中，“下标 1”表示的是二维数组中存储的某个一维数组的标号，如图 5.6 中下方的 $0 \sim m$ 。“下标 2”表示的是一维数组中某个格子的标号，如图 5.6 中右边的 $0 \sim n$ 。

按照二维数组的取数组元素运算表达式的表示形式，C 语言给二维数组赋值的表示形式如下所示：

数组名[下标 1][下标 2] = 数值；

这个赋值语句就把“数值”放到了由“下标 1”和“下标 2”所确定的数组中的一个格子中了。

例如，定义声明了一个二维数组 Matrix。它可以保存 4 个一维数组元素，每个一维数组元素中可以保存 2 个整型的数据。C 语言表示如下所示：

```
int Matrix[4][2] ;
```

在完成这样的声明定义后，计算机就给出了如图 5.9 所示的 8 个格子。与一维数组相同，二维数组的两个下标都是从 0 开始的。

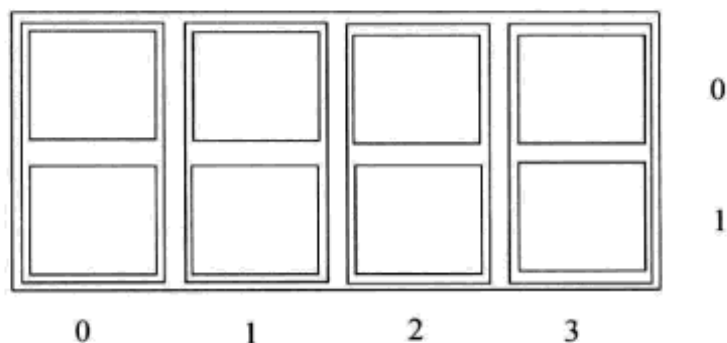


图 5.9 Matrix 二维数组

有了声明定义的 Matrix 数组以后，就可以使用赋值语句给图 5.9 中的格子放入数据了。可以使用下面的代码给 Matrix 矩阵中每个元素赋值。

```
#include<stdio.h>

int main()
{
    int Matrix[4][2];
    int i,j;
    int value;

    value = 1;

    for(i=0; i<4; i++)                //使 i ("下标 1") 不断循环加 1
    {
        for(j=0; j<2; j++)            //使 j ("下标 2") 不断循环加 1
        {
            Matrix[i][j] = value++;
            //用 i 作为"下标 1"，用 j 作为"下标 2"，给数组 ij 确定的格子中放入值
        }
    }

    return 0;
}
```


在这个程序中使用了两个 for 循环的嵌套。逐个打开数组 `Matrix` 中每一个格子的盖子，然后使用二维数组赋值语句，往数组 `Matrix` 中放入 `value` 值，每放入一次将 `value` 的值加 1。程序执行完以后，数组 `Matrix` 的样子如图 5.10 所示。

1	3	5	7	0
2	4	6	8	1
0	1	2	3	

图 5.10 赋值后的 `Matrix` 二维数组

5.3.5 二维数组的引用

二维数的引用和一维数组类似，也是使用取数组元素运算表达式：

数组名[下标 1][下标 2]

例如：

`Matrix[1][1]`

就相当于打开了数组 `Matrix` 中第 2 个一维数组元素中的第 2 个格子。打开之后，就可以使用这个格子中所保存的数据进行各种运算或者操作了。用完之后，格子中的数据还会被放回去，以备下次再用。

可以使用下面的程序对二维数组的初始化、赋值及引用的各方面规则予以展示。

```
#include<stdio.h>

int main()
{
    //二维数组 Matrix 初始化
    int Matrix[4][2] = {1,2,3,4,5,6,7,8};
    int i,j;

    //Matrix 数组引用，输出每个格子中的数值
    for(i=0; i<4; i++)
    {
        for(j=0; j<2; j++)
        {
            printf("%d ",Matrix[i][j]);
        }
        printf("\n");
    }

    //给 Matrix 数组中的每个元素赋值为 0
    for(i=0; i<4; i++)
    {
        for(j=0; j<2; j++)
```

```

    {
        Matrix[i][j] = 0;
    }
}

//Matrix 数组引用, 输出每个格子中的数值
for(i=0; i<4; i++)
{
    for(j=0; j<2; j++)
    {
        printf("%d ", Matrix[i][j]);
    }
    printf("\n");
}

return 0;
}

```

这个程序中每个“//”注释之下的代码完成注释所描述的功能, 依次为给数组 Matrix 初始化->Matrix 引用输出->Matrix 赋值->Matrix 引用输出。Matrix 的引用和赋值都是使用两个 for 循环组成的嵌套循环打开二维数组中的每个格子。

程序的输出如图 5.11 所示。由于程序的开始使用了二维数组的初始化给 Matrix 中的元素依次初始化为 1~8, 所以图 5.11 中先输出 1~8, 之后程序使用了二维数组赋值将 Matrix 中的元素都赋值为 0, 接下来输出 8 个 0, 如图 5.11 所示。

```

//二维数组Matrix初始化
int Matrix[4][2] = {1,2,3,4,5,6,7,8};
int i,j;

//Matrix数组引用, 输出每个格子中的数值
for(i=0; i<4; i++)
{
    for(j=0; j<2; j++)
    {
        printf("%d ", Matrix[i][j]);
    }
    printf("\n");
}

//给Matrix数组中的每个元素赋值为0
for(i=0; i<4; i++)
{
    for(j=0; j<2; j++)
    {
        Matrix[i][j] = 0;
    }
}

//Matrix数组引用, 输出每个格子中的数值
for(i=0; i<4; i++)
{
    for(j=0; j<2; j++)
    {
        printf("%d ", Matrix[i][j]);
    }
    printf("\n");
}

```

```

C:\WINDOWS\system32\cmd
1 2
3 4
5 6
7 8
0 0
0 0
0 0
0 0
请按任意键继续. . .
搜狗拼音 半:

```

图 5.11 二维数组程序验证

5.4 多维数组

当数组的维数超过 2 的时候, 就称这样的数组为多维数组。多维数组的 C 语言表示形

式、取数组元素运算、初始化、赋值和引用等规则和一维二维数组都是类似的，唯一不同的就是它的维数变多了。

对于多维数组，C 语言中使用得比较少，一维、二维对于基本的表示已经够用了。所以这里不再过于详细地讲解多维数组。只给出一个四维数组的例子，其中包含四维数组的表示形式、初始化、取数组元素运算、赋值、引用等方面。相信大家参照这个例子和一维、二维数组的对应知识点，就可以得出其他多维数组的使用方式了。

下面就是使用四维数组 **Matrix** 进行操作的一个例子，展示了多维数组声明定义、初始化、取四维数组运算表示、赋值和引用等各个方面的 C 语言表示。

```
#include<stdio.h>

int main()
{
    //二维数组 Matrix 初始化
    int Matrix[1][2][3][3] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
    17,18};
    int i,j,k,l;

    //Matrix 数组引用，输出每个格子中的数值
    for(i=0; i<1; i++)
    {
        for(j=0; j<2; j++)
        {
            for(k=0; k<3; k++)
            {
                for(l=0; l<3; l++)
                {
                    printf("%d ",Matrix[i][j][k][l]);
                }
                printf("\n");
            }
        }
    }

    //给 Matrix 数组中的每个元素赋值为 0
    for(i=0; i<1; i++)
    {
        for(j=0; j<2; j++)
        {
            for(k=0; k<3; k++)
            {
                for(l=0; l<3; l++)
                {
                    Matrix[i][j][k][l] = 0;
                }
            }
        }
    }

    //Matrix 数组引用，输出每个格子中的数值
    for(i=0; i<1; i++)
    {
        for(j=0; j<2; j++)
        {
            for(k=0; k<3; k++)
            {
```



```

        for(l=0; l<3; l++)
        {
            printf("%d ",Matrix[i][j][k][l]);
        }
        printf("\n");
    }
}

return 0;
}

```

这个例子和本章中“二维数组的引用”一部分的例子实现的功能是一样的，结构也类似，唯一不同的就是将数组 `Matrix` 从二维变到四维了，展示了一个四维数组的定义声明、初始化、取四维数组运算表示、赋值和引用的各个方面。对于其他多维数组，方法类似，大家照猫画虎就可以了。

程序的输出如图 5.12 所示。图 5.12 中的输出也和图 5.11 所示的“二维数组的引用”类似。图 5.12 中，由于四维数组初始化，输出了 1~18，之后由于四维数组赋值把每个元素的值变成 0，因此输出了 18 个 0。

```

//二维数组Matrix初始化
int Matrix[1][2][3][3] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
int i,j,k,l;

//Matrix数组引用，输出每个格子中的数值
for(i=0; i<1; i++)
{
    for(j=0; j<2; j++)
    {
        for(k=0; k<3; k++)
        {
            for(l=0; l<3; l++)
            {
                printf("%d ",Matrix[i][j][k][l]);
            }
            printf("\n");
        }
    }
}

//给Matrix数组中的每个元素赋值为0
for(i=0; i<1; i++)
{
    for(j=0; j<2; j++)
    {
        for(k=0; k<3; k++)
        {
            for(l=0; l<3; l++)
            {
                Matrix[i][j][k][l] = 0;
            }
        }
    }
}

```

```

C:\WINDOWS\system32\cmd
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
0 0 0
请按任意键继续. . .
搜狗拼音 半:

```

图 5.12 多维数组程序验证

5.5 小 结

本章介绍了数组类型的概念，也是我们接触的第一种高级数据类型。本章的重点是数组的相关概念和操作，难点是各维数组的使用。下一章将介绍另一种特殊的数组——字符数组。在 C 语言中这种数组有个专门的称呼，叫字符串。

5.6 习 题

【题目 1】 一般情况下，数组的下标都是整数。如果使用一个小数或者字符作为数组的下标，会发生什么情况呢？写个程序试验一下！

【分析】 在给数组元素赋值和引用数组元素的时候，都要用到数组下标，而且这个下标一般情况下都是整型数据。如果使用浮点型，而且使用的编译器对于语言检测不是很严格，会将浮点型自动转换成整型；如果你使用的编译器对于语言检测比较严格会直接报错。如果使用字符型，由于字符型的 ASCII 码是整型数字，所以一般会使用 ASCII 的整型数字作为下标使用。可以写段代码检验一下你使用的开发环境的 C 语言编译器到底是如何处理的。

【核心代码】

```
int nums[10] = {0,1,2,3,4,5,6,7,8,9};

printf("%d\n",nums[2.14]);
printf("%d\n",nums['\0']);
```

【题目 2】 写一段程序使用一维数组来代替一个 `int matrix[2][3]` 的二维数组。

【分析】 二维数组的作用就是用来保存一维数组，`matrix[2][3]` 就是用来保存两个一维 `int` 型数组的，每个一维 `int` 型数组可以保存 3 个 `int` 型的数据。要想使用一维数组代替 `matrix[2][3]`，可以将 `matrix[2][3]` 保存的两个一维数组拿出来，分别进行声明定义，不必保存在 `matrix` 这个二维数组中。

【核心代码】

```
int matrix_2_1[3];
int matrix_2_2[3];
```

【题目 3】 计算一下今天是今年的第几天。

【分析】 要计算今天是今年的第几天，先得知道今年是闰年还是平年，如果是平年，二月只有 28 天，闰年就有 29 天了。判断是平年还是闰年以后，就可以使用一个二维数组来保存每个月的天数，方便计算。

【核心代码】

```
int year;
int month;
int day;

int lp;
int i;
int days = 0;
int day_tab[12][13] = { {0,31,28,31,30,31,30,31,31,30,31,30,31},
                        {0,31,29,31,30,31,30,31,31,30,31,30,31}};

...

//判断是否是闰年
lp = year%4 == 0 && year%100 != 0 || year%400 == 0;
for(i= 1; i < month; i++)
```

```
{
    days += day_tab[lp][i];
}
days += day;
...
```

【题目 4】 魔术游戏：魔术师拿出一副牌中的 13 张黑桃，然后告诉观众“我不看牌，只用数数就可以知道每张牌是什么”，他数 1，翻开这张牌是黑桃 A，他将黑桃 A 放在座位上。然后按顺序从上到下数手上的余牌，第二次数 1、2，将第一张牌放在这迭牌的下面，将二张牌翻开，是黑桃 2，将黑桃 2 放在桌子上。第三次数 1、2、3，将前两张牌放在这迭牌的下面，翻开第三张牌是黑桃 3，将黑桃 3 放在桌子上。这样依次找出了 13 张牌。问魔术师手中的牌原始的次序是怎样的？

【分析】 这个问题，其实就是还原牌序，我们可以使用逆向思维：在桌子上方将 13 个空盒子排成一圈，从 1 开始按顺序编号，将黑桃 A 放在 1 号盒子中，从下一个空盒子开始对空的盒子计数，当数到第二个空盒子时，将黑桃 2 放在空盒子中，然后再从下一个空盒子开始对空盒子计数，按顺序放入 3、4、5…，直到放入全部 13 张牌。注意在计数时要跳过非空的盒子，只对空盒子计数。最后盒子中牌的顺序，就是魔术师手中原来牌的顺序，整个过程如图 5.13 所示。

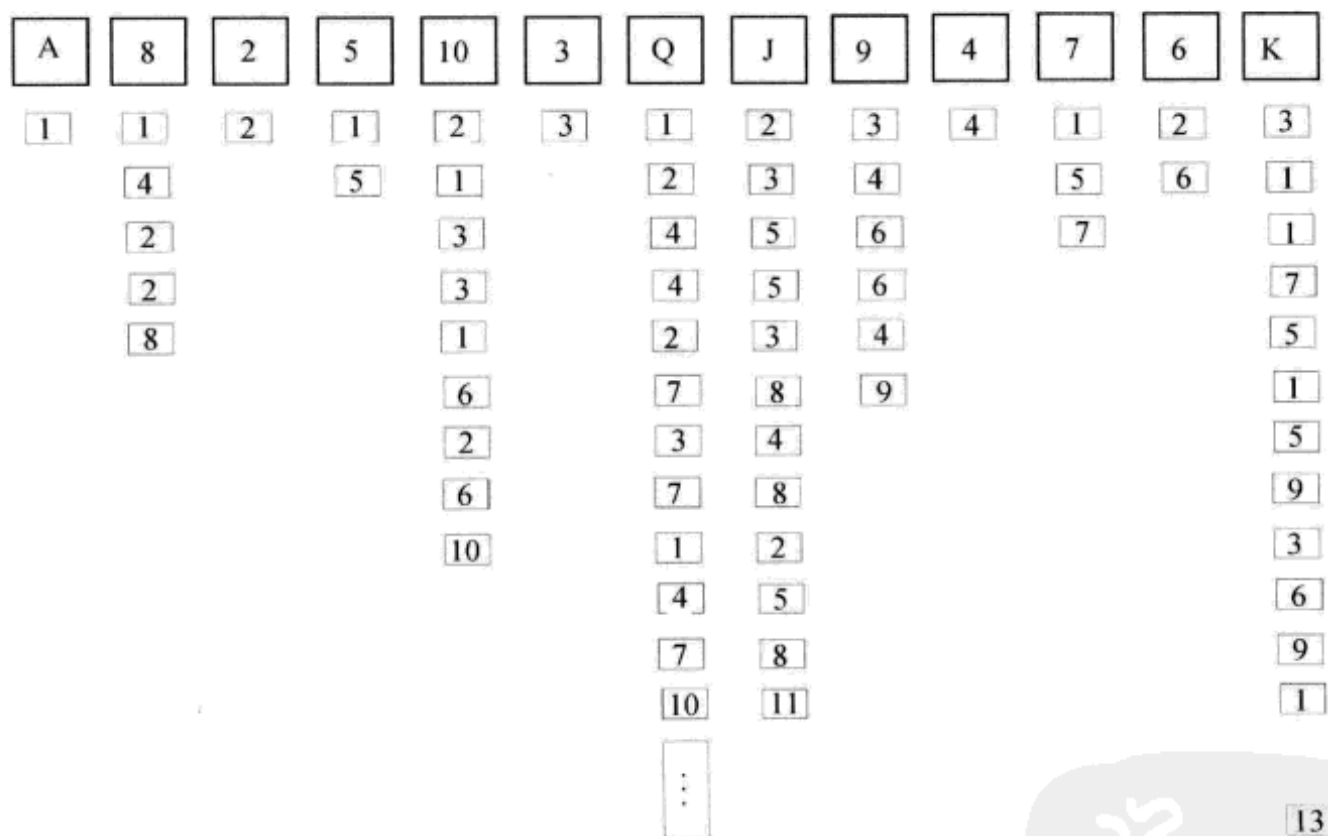


图 5.13 魔术牌排序

【核心代码】

```
int a[14]={0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int i, n, j =1;
for(i=1; i<13; i++)
{
    n =1;
    do
    {
        if(j >13) j=1;          //从头开始
```

```
        if(a[j]) j++;           //跳过已经有牌的位置
        else
        {
            if(n==i) a[j] = i;   //将牌放在合适的位置
            j++;
            n++;
        }
    } while(n<=i);
}

for(i=1; i<=13; i++)
    printf("%d",a[i]);
```

第 6 章 字符数组——字符串

如果数组中保存的数据是字符类型的，就是字符数组。因为字符数组经常与输入和显示有关，所以 C 语言对其进行了特殊的处理和表示。字符串是专门用来表示一串字符的。本章我们就来看看 C 语言是如何表示和处理字符数组和字符串的。

6.1 字符数组

字符数组，顾名思义就是存储字符数据的数组。它可以是一维字符数组、二维字符数组，甚至多维字符数组。在这一点上它与其他类型的数组都是一样的，唯一特殊的只是字符数组存储的是字符数据。

6.1.1 字符数组的表示

在 C 语言中，最常使用的也是一维和二维字符数组，对于多维字符数组很少使用。这里只给出一维、二维字符数组的 C 语言表示，如下所示：

```
char 一维字符数组名[表达式]
char 二维字符数组名[表达式 1][表达式 2]
```

上面是一维和二维字符数组的表示，与前面介绍的一维和二维数组唯一不同的，就是一维、二维字符数组的数据类型为 `char`。数组名还是表示数组变量的命名，只要遵循标识符的命名规范就行。表达式还是表示可以存储的字符的个数。

例如，可以用下面的一维字符数组 `name` 保存一个名字，用二维字符数组 `names` 保存 100 个名字。假设一个名字的表示不会超过 10 个字符。

```
char name[10];
char names[100][10];
```

6.1.2 字符数组的初始化

用字符数组保存字符的方法也有两个：初始化和赋值，下面先来看看字符数组的初始化。字符数组初始化的 C 语言表示和一般数组初始化的 C 语言表示是类似的，具体形式如下所示：

```
char 一维字符数组名[表达式] = {字符, 字符, 字符, ..., 字符};
```


这是一维字符数组的初始化，对于二维字符的初始化，有两种形式。

```
char 二维字符数组名[表达式1][表达式2] = {字符, 字符, 字符, 字符, ..., 字符, 字符, 字符};
```

和

```
char 二维字符数组名[表达式1][表达式2] = { {字符, ..., 字符}, {字符, ..., 字符}, ..., {字符, ..., 字符}};
```

第一种形式要靠系统来区分哪个字符该放到二维字符数组的哪个地方。相比之下，第二种形式比较直观，它可以清楚地表示出二维字符数组的构成。

有了字符数组的初始化表示之后，就可以在之前定义的一维字符数组 `name` 和二维字符数组 `names` 中保存名字了。

```
char name[10] = {'x', 'i', 'a', 'o', ' ', 'm', 'i', 'n', 'g'};
char names[100][10] = { {'x', 'i', 'a', 'o', ' ', 'h', 'u', 'a'}, {'d', 'a', ' ', 'h', 'u'}, {'h', 'u', 'a', 'n', ' ', 'h', 'u', 'a', 'n'}};
```

显然，在这两个字符数组的初始化中，并没有用完所有的小方格。一维字符数组 `name` 中并没有 10 个字符，二维字符数组 `names` 中也没有 100 个名字。在 C 语言中这些没有用到的小方格会空置着，没有填满不会出错，但是不能超量。

图 6.1 就是初始化完成之后 `name` 字符数组的样子。下标为 4 的地方是一个空格，下标为 9 的地方也就是没有被使用的格子，有的编译器会将没有使用的格子都放入一个字符 `null` (ASCII 值为 0)，但是，有的编译器不会，建议大家不要从没有数据的格子里面拿数据，否则有时会出错，当然有时不会出错，但不能保证。

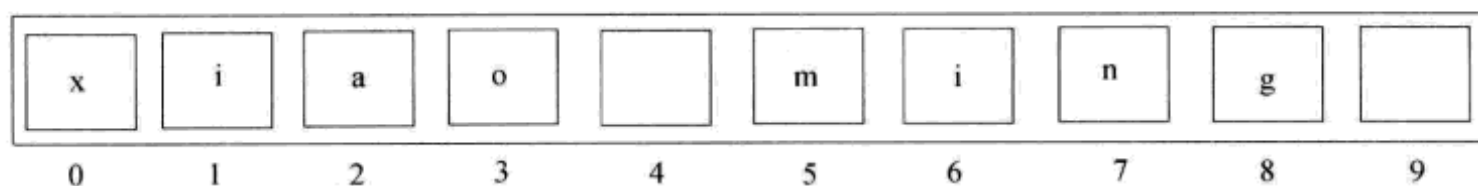


图 6.1 `name` 字符数组

图 6.2 是经过初始化的 `names` 二维字符数组。在这个二维数组的保存形式中，只用到了 100 个保存一维字符数组的中型格子中的前 3 列，对于这 3 个中型格子也只是用到了其中的一部分。从图 6.2 中看出，对二维字符数组进行初始化的时候，计算机是按照每一个内部大括号对应一个中型的格子来进行保存的。

如果初始化的时候不加上面的大括号，计算机就会按照每个字符一个小格子来进行保存。从 0 号中格子开始的 0 号小格子开始进行保存。当 0 号中格子中的小格子都用完以后，再从 1 号中格子的 0 号小格子开始继续保存，直到保存完所有初始化的字符。

如果希望二维字符数组保存字符的时候，是以中格子为单位进行保存的，就要把每个中型格子中要保存的字符用大括号括起来。

6.1.3 字符数组的赋值和引用

字符数组的赋值是在字符数组中保存数据的操作。字符数组的引用是从已经保存了数据的字符数组中取出数据的操作。由于它们都用到取数组元素运算，所以放到一起来讲。

				...					9
		n		...					8
a		a		...					7
u		u		...					6
h		h		...					5
	u			...					4
o	h	n		...					3
a		a		...					2
i	a	u		...					1
x		h		...					0
0	1	2	3		96	97	98	99	

图 6.2 names 字符数组

字符数组的取数组元素运算的 C 语言表示如下所示，它和一般数组的取数组元素运算是类似的。

```
一维字符数组名[下标]
二维字符数组名[下标 1][下标 2]
```

例如，要给前面的 name 字符数组的第 1 个元素赋值，或者引用 name 字符数组中的第 1 个元素。就可以使用字符数组的取数组元素运算，其形式如下所示：

```
name[0];
```

C 语言中，数组的下标是从 0 开始的，所以要取第 1 个元素，下标当然就是 0 了。要给 name 中的第 1 个小格子赋值为字符'x'，就可以使用数组元素赋值操作，其形式如下：

```
name[0] = 'x';
```

要引用字符数组 name 中的第 1 个元素进行输出的时候，直接用字符数组的取数组元素运算就行，可以这样表示：

```
printf("%c", name[0]);
```

下面的例子中，先对 name 的第 1 个元素赋值，然后输出第 1 个元素的值：

```
#include<stdio.h>

int main()
{
    char name[10];
```

```

    name[0] = 'x';

    printf("%c\n",name[0]);
    printf("%c\n",name[1]);

    return 0;
}

```

在程序中，只给 `name` 的第 1 个元素赋值为 'x'。但是，输出的时候不仅引用了 `name` 数组的第 1 个元素，而且还引用了没有赋值的第 2 个元素。对没有赋值的数组元素进行引用时，各个编译器的处理是不一样的，目前所用的编译器中就直接出错了，程序输出和出错提示如图 6.3 所示。建议大家不要对没有赋值的数组元素进行引用。

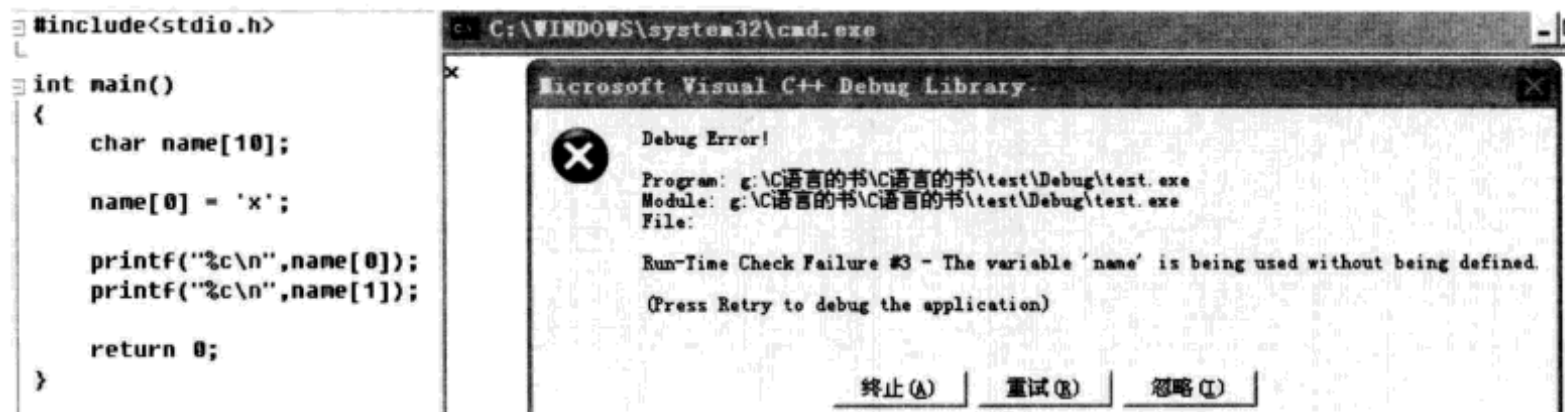


图 6.3 字符数组的赋值和引用

6.2 字符串

字符串是用来表示一串字符的一种数据类型。由于 C 语言没有专门用于表示字符串的类型，所以就借用字符数组来表示了，刚好字符数组也能够承担此大任。

6.2.1 字符串的 C 语言表示

在 C 语言中，字符串常量的表示是用一对双引号，中间括住一连串的字符，其形式如下所示：

"字符字符字符..."

例如，下面这些都是合法的 C 语言表示的字符串常量：

```

"xiao hua"
"da hu"
"huan huan"

```

6.2.2 使用字符串为字符数组初始化

在 C 语言中，由于字符串是按照字符数组的形式进行表示和保存的，因此，可以直接用字符串对字符数组进行初始化。

例如，对二维字符数组 `names` 的初始化：

```
char names[100][10] = { {'x', 'i', 'a', 'o', ' ', 'h', 'u', 'a'}, {'d', 'a', ' ', 'h', 'u'}, {'h', 'u', 'a', 'n', ' ', 'h', 'u', 'a', 'n'}};
```

改为：

```
char names[100][10] = { {"xiao hua"}, {"da hu"}, {"huan huan"}};
```

或者也可以直接省略内部的大括号。C 语言编译器会把每个字符串保存到对应的中型格子中。

```
char names[100][10] = { "xiao hua", "da hu", "huan huan"};
```

6.2.3 字符串的保存形式

字符串在 C 语言中是作为字符数组进行保存的。但是它与一般的字符数组还是有一点点区别的，C 语言中的字符串是以字符 `'\0'`（ASCII 码值为 0）为结尾的。当写一对双引号括住字符串时，不用直接写这个 `'\0'`，C 语言的编译器会自动加上。这个 `'\0'` 字符正是 C 语言字符串的标志。

例如，将字符数组 `name[10]`，初始化为字符串 `"xiao ming"`。

```
char name[10] = "xiao ming";
```

进行了这样的初始化以后，`name` 数组的保存形式就会变成如图 6.4 所示的样子，后面多了一个字符 `'\0'`。

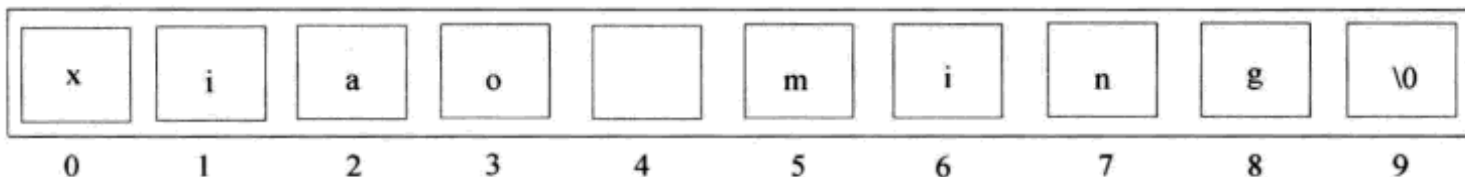


图 6.4 字符串初始化的字符数组

可以用下面的程序看看分别经过字符初始化和字符串初始化之后，字符数组的每个元素的 ASCII 值到底是什么样的。

```
#include<stdio.h>

int main()
{
    char name1[10] = {'x', 'i', 'a', 'o', ' ', 'm', 'i', 'n', 'g'};
    char name2[10] = "xiao ming";

    int i;
    int ascii_value;

    for(i=0; i<10; i++)
    {
        ascii_value = (int)name1[i];
        //使用强制转换将字符型变成整型，这个整型值就是字符的 ASCII 码值
        printf("name1[%d]:%d\n", i, ascii_value);
    }
}
```



```

        ascii_value = (int)name2[i];
        //使用强制转换将字符型变成整型，这个整型值就是字符的 ASCII 码值
        printf("name2[%d]:%d\n",i,ascii_value);
    }

    return 0;
}

```

在这个程序中，使用字符赋值和字符串赋值两种方式分别对字符数组 `name1` 和 `name2` 进行初始化。然后使用 `for` 循环输出 `name1` 和 `name2` 中每个元素的 ASCII 码值。为了得到每个字符的 ASCII 码值，使用了强制转换，将字符型转换成了整型数值。

程序的输出如图 6.5 所示。从输出结果中可以看出，`name2[9]` 的值为 0，也就是字符 `'\0'`，符合 C 语言对字符串的要求。但是有人会问，`name1[9]` 的值为什么也是 0 啊？这是因为初始化时？没有用到的格子中到底是什么？根据编译器而定，目前使用的编译器会往里面放一个字符 `'\0'`。

```

#include<stdio.h>

int main()
{
    char name1[10] = {'x', 'i', 'a', 'o', ' ', 'm', 'i', 'n', 'g'};
    char name2[10] = "xiao ming";

    int i;
    int ascii_value;

    for(i=0; i<10; i++)
    {
        ascii_value = (int)name1[i];
        printf("name1[%d]:%d\n",i,ascii_value);

        ascii_value = (int)name2[i];
        printf("name2[%d]:%d\n",i,ascii_value);
    }

    return 0;
}

```

```

C:\WINDOWS\system32
name1[0]:120
name2[0]:120
name1[1]:105
name2[1]:105
name1[2]:97
name2[2]:97
name1[3]:111
name2[3]:111
name1[4]:32
name2[4]:32
name1[5]:109
name2[5]:109
name1[6]:105
name2[6]:105
name1[7]:110
name2[7]:110
name1[8]:103
name2[8]:103
name1[9]:0
name2[9]:0
请按任意键继续...

```

图 6.5 字符数组的两种初始化方式

6.3 字符串的输入/输出——scanf 和 printf 字符串

像整型、浮点型和字符型一样，C 语言中的字符串可以直接使用 `scanf()` 函数输入，同时也可以使用 `printf()` 函数进行输出。

6.3.1 输入/输出字符串的 C 语言表示

接下来看看如何使用 `scanf()` 函数直接输入字符串，保存到字符数组中，以及如何使用 `printf()` 函数输出保存在字符数组中的字符串。使用 `scanf()` 函数输入字符串，并保存到字符数组的 C 语言表示如下所示，是将一般的 `scanf()` 函数赋值表示：

```
scanf("%变量类型表示",&变量名);
```

特殊化成:

```
scanf("%s", 字符数组名);
```

这个 `scanf()` 函数的使用形式和以前使用 `scanf()` 函数给其他类型的变量赋值的形式是一样的。只是“变量类型表示”用小写字母's'表示,“&变量名”直接写成了字符数组名。

用 `printf()` 函数输出一个字符串,也是将一般的 `printf()` 函数的 C 语言表示特殊化成如下形式:

```
printf("%s", 字符数组名);
```

好了,有了 `scanf()` 和 `printf()` 输入/输出字符串的 C 语言表示,就可以很容易地给一个字符数组赋值,也就可以很容易地输出一个字符串了。不必一个一个字符地输入然后赋值到字符数组的每个元素格子中,也不必一个一个地从字符数组的每个格子中拿出字符然后输出。

例如,使用 `scanf()` 函数给一个字符数组 `name[10]` 赋值为“XiaoMing”,然后使用 `printf()` 函数将字符数组 `name` 中的内容输出,就可以使用下面的程序。

```
#include<stdio.h>

int main()
{
    char name[10];

    scanf("%s", name);
    printf("%s\n",name);

    return 0;
}
```

这个程序简单地按照 `scanf()` 和 `printf()` 输入/输出字符串的格式,对字符数组赋值,然后输出所赋的值。程序的输出如图 6.6 所示。第一行的字符串“XiaoMing”是我们从键盘输入的,它将被 `scanf()` 函数保存到字符数组 `name` 中,第二行的“XiaoMing”是 `printf()` 函数从字符数组 `name` 中输出的。

如果不是按照上面的方式(“XiaoMing”两个词是连起来的)进行输入,而是将两个词分开,中间用一个空格隔开,即“Xiao Ming”,结果会如何呢?如图 6.7 所示,当将两个词分开输入的时候,程序只输出了第一个词“Xiao”,并没有输出第二个词“Ming”,这是为什么呢?

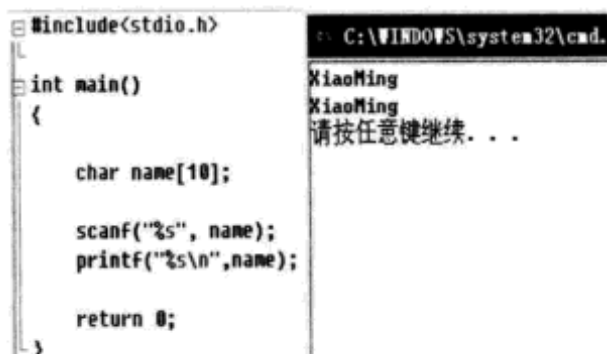


图 6.6 输入/输出字符串

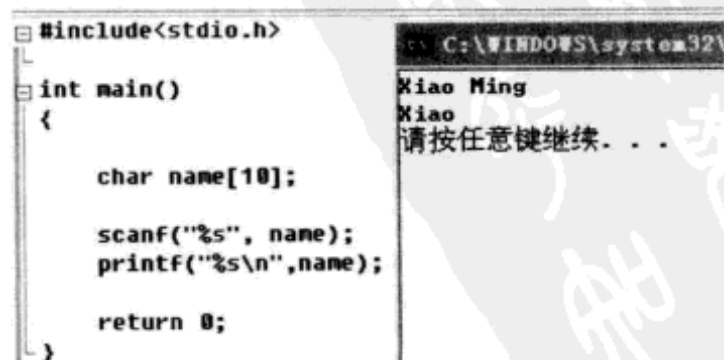


图 6.7 分开两个词的输入/输出

6.3.2 scanf()函数对字符串的特殊处理

之所以在图 6.7 中会出现输入的字符串只输出了一半的现象，是因为 scanf()函数对输出的字符串做了特殊的处理。具体来说就是 scanf()函数是按照以下几步执行的。

(1) 把输入的字符串中的字符一个一个地保存到字符数组中，直到遇见空格或者回车。

(2) 当 scanf()函数遇到空格或者回车以后，scanf()就退出了。

按照上面的步骤，来看看图 6.7，为什么会有那样的输出。从开始输入字符串“Xiao Ming”说起，当输入完字符串后，按下回车，就触发了 scanf()函数按照上面的两个步骤执行了。

(1) scanf()函数一个一个地读入字符并保存到字符数组 name 中，直到遇到空格，此时 name 中的内容为“Xiao”。

(2) scanf()函数退出以后，就开始 printf()函数的执行了；printf()函数输出 name 的内容，所以屏幕上显示只有输入字符串的一半“Xiao”。

有人会问，那么输入的字符串的另一半“Ming”跑哪去了？其实，它没有跑到哪里去，留在那儿等着下一个 scanf()函数来读呢！可是在程序中，没有 scanf()函数来读，程序输出完 name 就关闭了。所以，“Ming”在程序关闭的时候被丢掉了。

要想不丢掉输入的另一半字符串，就必须再使用一次 scanf()函数把“Ming”读回来，具体的程序如下所示：

```
#include<stdio.h>

int main()
{
    char name[10];
    int i;

    for(i=0; i<2; i++)
    {
        scanf("%s", name);
        printf("[%d] %s\n", i, name);
    }

    return 0;
}
```

在这个程序中，使用了 for 循环把 scanf()和 printf()函数分别执行了两次。这样，输入的字符串就被 scanf()函数分两次进行读取了，读取之后，printf()函数紧跟着就把读取的内容输出，程序执行结果如图 6.8 所示。

之所以用 scanf()函数对字符数组赋值的时候，会出现意想不到的结果，是因为 scanf()函数看到的字符串和输入的字符串是不一样的。scanf()函数是按照空格和回车来区分字符串的，中间有一个空格的字符串会被认为是两个字符串。要想让 scanf()函数能按照目的来读取字符串，就得遵循按照空格或者回车分割字符串的规矩。

```

#include<stdio.h>
int main()
{
    char name[10];
    int i;

    for(i=0; i<2; i++)
    {
        scanf("%s", name);
        printf("[%d] %s\n", i, name);
    }

    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
Xiao Ming
[0] Xiao
[1] Ming
请按任意键继续. . .

```

图 6.8 scanf()函数分两次读取

即使输入的字符串再复杂，scanf()函数也只认空格和回车。下面就是输入一段毫无规律的字符串，然后交给scanf()函数来处理例子：

```

#include<stdio.h>

int main()
{
    char name[10];
    int i;

    for(i=0; i<10; i++)
    {
        scanf("%s", name);
        printf("%d name = %s\n", i, name);
    }

    return 0;
}

```

程序的输入和输出如图 6.9 所示。随便输入了由不同类型字符组成的字符串，然后按回车键结束。程序确实按照scanf()处理字符串的规则，将输入的字符串以空格为间隔分成了 5 个字符串。所以，printf()一下输出了 5 个。同时输出了正在执行的循环次数，可知每执行一次循环，scanf()函数就进行一次输入，printf()函数进行一次输出。

```

#include<stdio.h>
int main()
{
    char name[10];
    int i;

    for(i=0; i<10; i++)
    {
        scanf("%s", name);
        printf("%d name = %s\n", i, name);
    }

    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
xiao 12!,? :''*#6 123ly)< (>~';
0 name = xiao
1 name = 12!,?
2 name = :''*#6
3 name = 123ly)<
4 name = (>~';

```

图 6.9 scanf()函数对输入字符串的特殊处理

6.4 小 结

本章详细地讲述了一种特殊的数组——字符数组，在 C 语言中称之为字符串。本章的

重点是字符数组和字符串的概念和表示,难点是字符数组和字符串之间的关系及使用方法。下一章将介绍另一种特殊的高级数据类型——指针,它和数组之间有许多共同之处。

6.5 习 题

【题目 1】 写一段程序,给字符数组 `char str[10]` 中的 10 个元素都赋值为非 `'\0'` 的字符,并输出这个字符数组所表示的字符串,看看会有什么样的现象,并解释这个现象。

【分析】 要想给字符数组中的元素赋值,可以使用两种方式:按照数组的方式赋值、使用字符串常量赋值。如果想简单一些,就使用字符串给字符数组赋值。要想输出字符串的值,可以使用 `printf()` 函数,变量类型表示使用 `"%s"`。C 语言规定字符串的结尾必须是 `'\0'`,但是我们的 `str` 字符数组中保存的字符串没有以 `'\0'` 结尾,直接输出,会有什么效果呢?写段代码试一下!

【核心代码】

```
char str[10] = "1234567890";
printf("%s\n",str);
```

【题目 2】 写程序将阿拉伯数字翻译成罗马数字,也就是说给你一个阿拉伯数字数组,写出与它对应的罗马数字表示,并且符合罗马数字组合规则。

【分析】 罗马数字和阿拉伯数字的关键数字的对应关系如下所示,其他数字都是由这些数字表示按照一定的规则组合起来的。

1	2	3	4	5	6	7	8	9
I	II	III	IV	V	VI	VII	VIII	IX
10	20	30	40	50	60	70	80	90
X	XX	XXX	XL	L	LX	LXX	LXXX	XCC
100	200	300	400	500	600	700	800	900
C	CC	CCC	CD	D	DC	DCC	DCCC	CM

我们在写代码的时候,可以将这个对应关系保存在一个数组中,在用的时候直接取出来就可以了。对于罗马数字我们采用 C 语言的字符串表示。

【核心代码】

```
char *a[][10] = {"", "I", "II", "IV", "V", "VI", "VII", "VIII", "IX",
    "", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XCC",
    "", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "DM"};
int n,t,i,m;
printf("please enter number:");
scanf("%d",&n);
printf("%d = ", n);
for(m=0,i=1000;m<3;m++,i/=10)
{
    t = (n%i)/(i/10);
    printf("%s",a[2-m][t]);
}
printf("\n");
```

【题目 3】 派遣任务:某次会议要求从 A、B、C、D、E、F6 个人中挑选若干个,但

是有以下限制条件。

- (1) A 和 B 两人中至少去一人。
- (2) A 和 D 不能一起去。
- (3) A、E 和 F 三人中要派两人去。
- (4) B 和 C 都去或都不去。
- (5) C 和 D 两人中去一个。
- (6) D 不去，则 E 不去。

问应当让哪些人去参加会议？

【分析】 从题目来看，这是一道关于逻辑推理的题目。通常我们使用数学的方式就是直接根据条件进行逻辑推导，直到出现正确的结果。不过计算机程序设计，不会采取这种方式，因为直接的逻辑推理不是计算机擅长的事情，那么计算机该怎么做呢？计算机直接列举出所有的情况，然后挨个判断是不是满足所有的限制条件，满足了就是正确的解答，不满足就不是正确的解答。在进行判断之前，先得将所有的限制条件转换为计算机可以判断的逻辑表达式，如下所示：

```
a + b > 1
a + d != 2
a + e + f == 2
b + c == 0 或 b + c == 2
c + d == 1
d + e == 0 或 d == 1
```

其中每个变量的取值只可以是 0 或者 1，0 表示不去，1 表示去。

【核心代码】

```
int a, b, c, d, e, f;
for(a = 1; a >= 0 ; a--)
    for(b = 1; b >= 0 ; b--)
        for(c = 1; c >= 0 ; c--)
            for(d = 1; d >= 0 ; d--)
                for(e = 1; e >= 0 ; e--)
                    for(f = 1; f >= 0 ; f--)
                        if( a+b >= 1 && a+d != 2 && a+e+f == 2 && (b+c==0 || b+c ==2)
&& c+d ==1 && (d+e == 0 || d==1))
{
    printf("A will %s be assigned! \n", a? "" : "not");
    printf("B will %s be assigned! \n", b? "" : "not");
    printf("C will %s be assigned! \n", c? "" : "not");
    printf("D will %s be assigned! \n", d? "" : "not");
    printf("E will %s be assigned! \n", e? "" : "not");
    printf("F will %s be assigned! \n", f? "" : "not");
}
```

第 7 章 指 针

不管读者有没有用过计算机，只要没有系统地学过计算机语言，指针都可能是一个陌生的概念。不仅如此，指针还是 C 语言难点中的难点，希望大家能足够重视。毕竟，指针是“C 语言的灵魂”！

7.1 地址的概念

要学指针，必须先弄清楚一个概念，那就是地址。比起指针，地址可能更容易理解些，地址和指针是同一个东西，只是用在不同的地方而已。

在讲解数组的时候，提到了数组的下标和取数组元素的运算，明白了地址的概念，就会更清楚这些数组上的操作的真正意义了。先来看一下图 7.1，这是一个数组的图示，其中 1, 2, ..., $n-2$, $n-1$ 都是数组的下标。

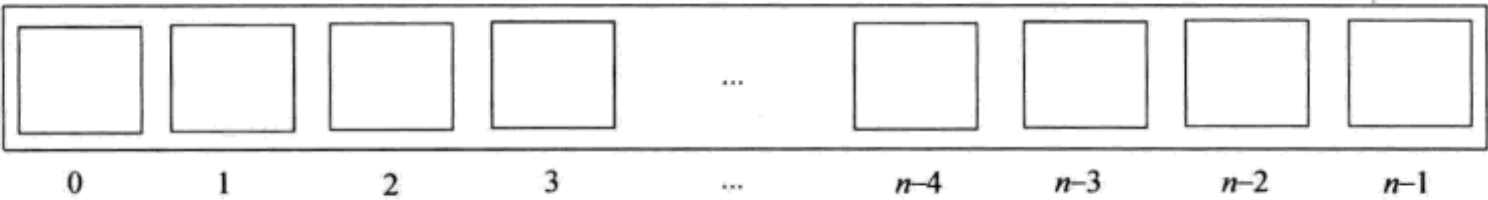


图 7.1 数组下标

7.1.1 地址的含义

在 C 语言程序中，可以在取数组元素的时候，使用这些下标拿到数组中的元素。当然，也可以在给数组赋值的时候使用这些下标，来打开数组中的某个格子。写程序的时候，我们看到的是数组的一个个下标，那么计算机看到的是什么呢？计算机看到的是一个一个的地址！

这就好比家里的“门牌号”和“通信地址”。我们看到数组中的“下标”就相当于家里的“门牌号”，而计算机看到的“地址”，就相当于名片上的“通信地址”。

例如，家里面的门牌号为“888”，通信地址为“XXX 省 XXX 市 XXXX 区 XXXX 街道 888 号”。写个程序，有个数组 `home[1000]`，现在用的元素在下标为“888”处。我们看到的是数组运算的下标“888”，计算机看到的就是 888 处元素的内存地址了，也就是 888 处元素在内存中的位置。可以简单地理解内存地址就是内存中的某个位置，就像图 7.2 所示的样子。

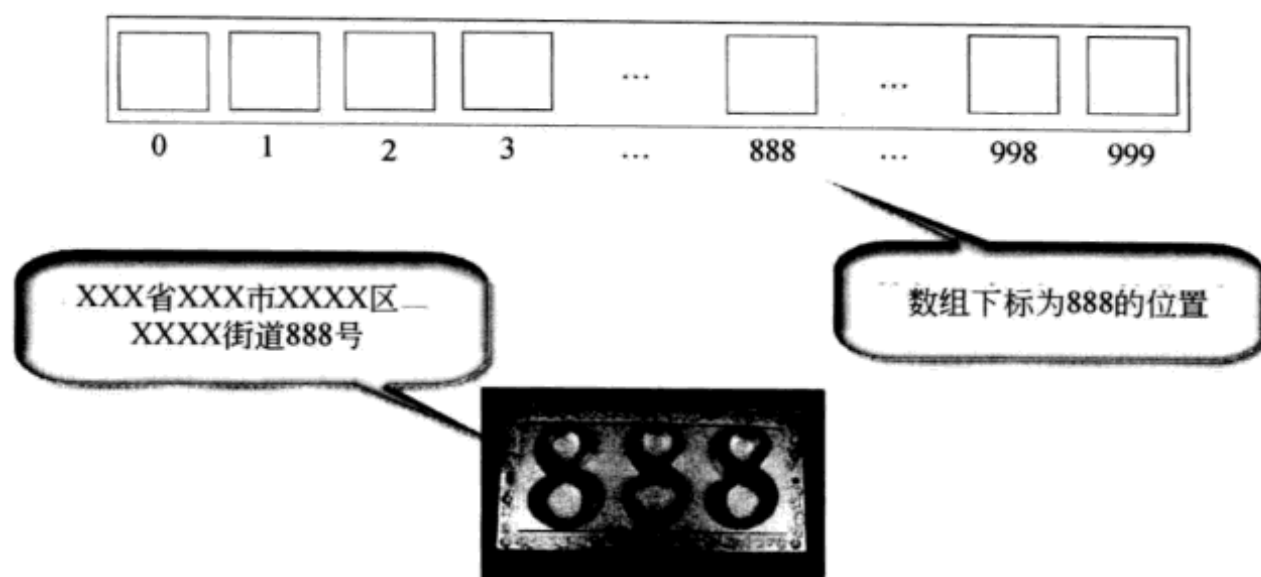


图 7.2 下标和门牌

7.1.2 为什么要用地址

为什么 C 语言中既有下标又有地址，有一个不行吗？这个问题就有点类似于，有门牌号不就行了，要通信地址干什么？当然这是不行的，门牌号只对具体的街道或者村子有用。当谈起某个村子的时候，只用门牌号就可以了。但是当谈起某个省甚至全国，只用门牌号就不行了。

类似地，在对数组进行操作的时候，我们明确使用的是哪个数组，所以用数组的下标就可以了。但是，对于计算机程序来说，一个程序中可能有好几个数组，甚至成千上万的数组，只用下标是不够的。因此，计算机使用了类似于现实生活中的“通信地址”的“地址”来操作数组中的元素。这就像图 7.3 所示的，不同的数组可能都有下标为 888 的位置，不同地方的门牌号可能都是 888。

简而言之，计算机中的地址和数组的下标的功能是类似的，都是寻找具体的数据元素的，只是它们的适用范围不一样而已！

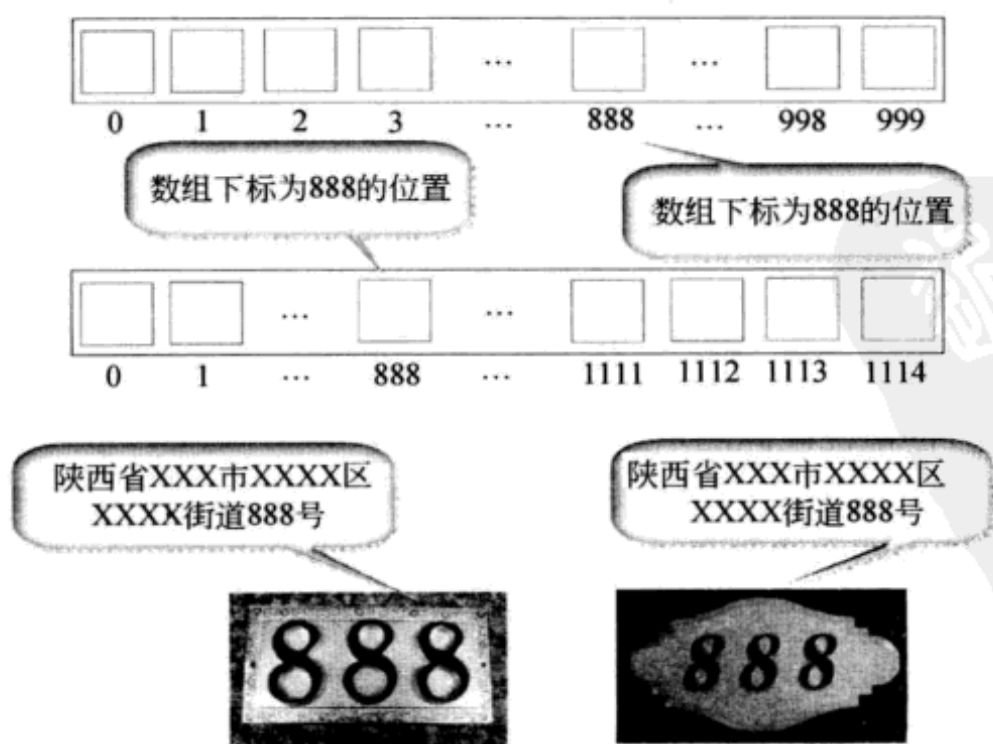


图 7.3 多个相同的下标和门牌

7.1.3 地址的表示与取址运算

在现实世界中，通信地址是使用类似于“XXX省XXX市XXXX区XXXX街道XXX号”的形式来表示的，计算机程序中的地址是怎么样的？用什么表示？在看计算机中的地址是什么样的之前，先来看一下C语言中的一种运算——取址运算。这种运算可以直接计算出C语言程序中任何一个变量的地址，当然包括数组中元素的位置了。

C语言中的取址运算的形式如下所示：

`&变量名`

在取址运算表达式中，“&”是取址运算的运算符，“变量名”就是程序中声明定义的变量。使用取址运算语句就可以知道一个变量在计算机程序中的内存地址了。例如，下面的程序先定义一个整型变量 `radius`，然后使用取址运算语句计算出 `radius` 变量在计算机中的内存地址。

```
int radius;  
&radius;
```

在C语言中任何表达式都是有一个值的，那么取址运算表达式的值是什么呢？这个容易回答，就是变量在计算机中的内存地址了！那么内存地址又是什么呢？呵呵，不好回答了吧？

其实，变量的内存地址就是一个数字，具体来说就是一个非负的整数。计算机给内存中的每个位置都编了一个号，变量在计算机中的内存地址就是变量所在内存位置的编号，是一个非负的整数。可以用无符号整数来保存一个变量的地址。

例如，下面的程序就把变量 `radius` 的地址输出来给大家看看。

```
#include<stdio.h>  
  
int main()  
{  
    int radius;  
    unsigned int address;  
  
    address = &radius;  
    printf("%x\n",address);  
  
    return 0;  
}
```

程序中，定义了一个无符号整型变量，用来保存 `radius` 变量所在的内存地址。使用取址运算把 `radius` 的内存地址计算出来，然后赋值给 `address` 变量；最后使用 `printf()` 函数输出 `address` 变量中保存的值，也就是 `radius` 的内存地址。由于程序中的内存地址一般较大，所以使用“%x”，按照十六进制表示进行输出，程序的输出如图7.4所示。图7.4中的十六进制数 `12ff7c` 就是变量 `radius` 的内存地址。

当然，也可以用类似的方式计算出数组元素的内存地址。例如，要计算出 `home[1000]` 中下标为 888 位置处的内存地址，就可以使用下面的程序。

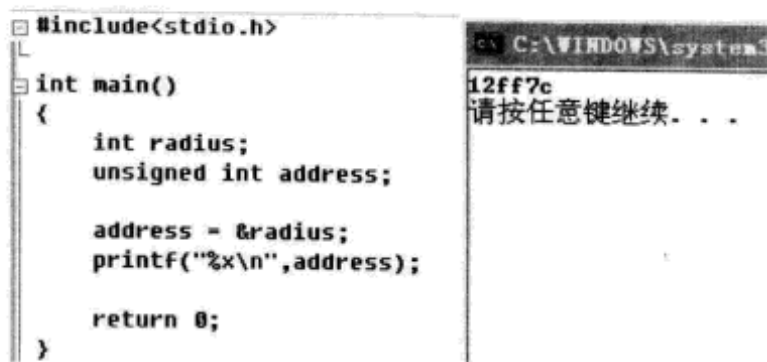


图 7.4 变量的地址

```
#include<stdio.h>

int main()
{
    int home[1000];
    unsigned int address;

    address = &home[888];
    printf("index=%d :: addr=%x\n",888,address);

    return 0;
}
```

程序的输出结果如图 7.5 所示，输出 home 下标为 888 处元素的内存地址为 12fdc0。

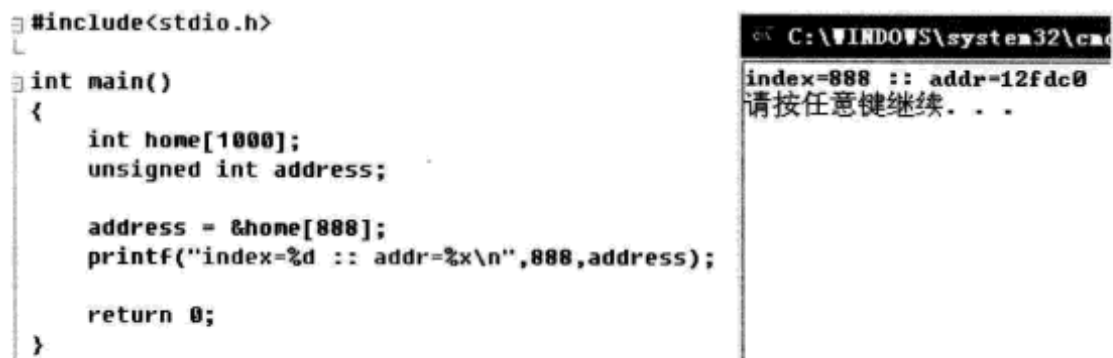


图 7.5 计算数组元素地址

7.2 指针和指针变量

有了地址的概念，接下来学习指针就容易得多了！因为指针的概念就是源于地址的，只不过在程序中经常用到的是指针的概念。现在就让我们看看什么是指针吧！

7.2.1 指针的含义和用途

前面已经了解了地址的概念了，地址就是数据元素在内存中的位置表示。那么指针又是什么呢？指针其实和地址是一个东西，指针即地址，地址即指针。比起“地址”，在程序中，指针能更加直观地表示指向某个位置这个意思！

指针，顾名思义，类似于指南针上的那个指针，只不过指南针上的指针是用来指示方向的，C 语言中的指针是用来指向某个内存地址的，如图 7.6 所示。

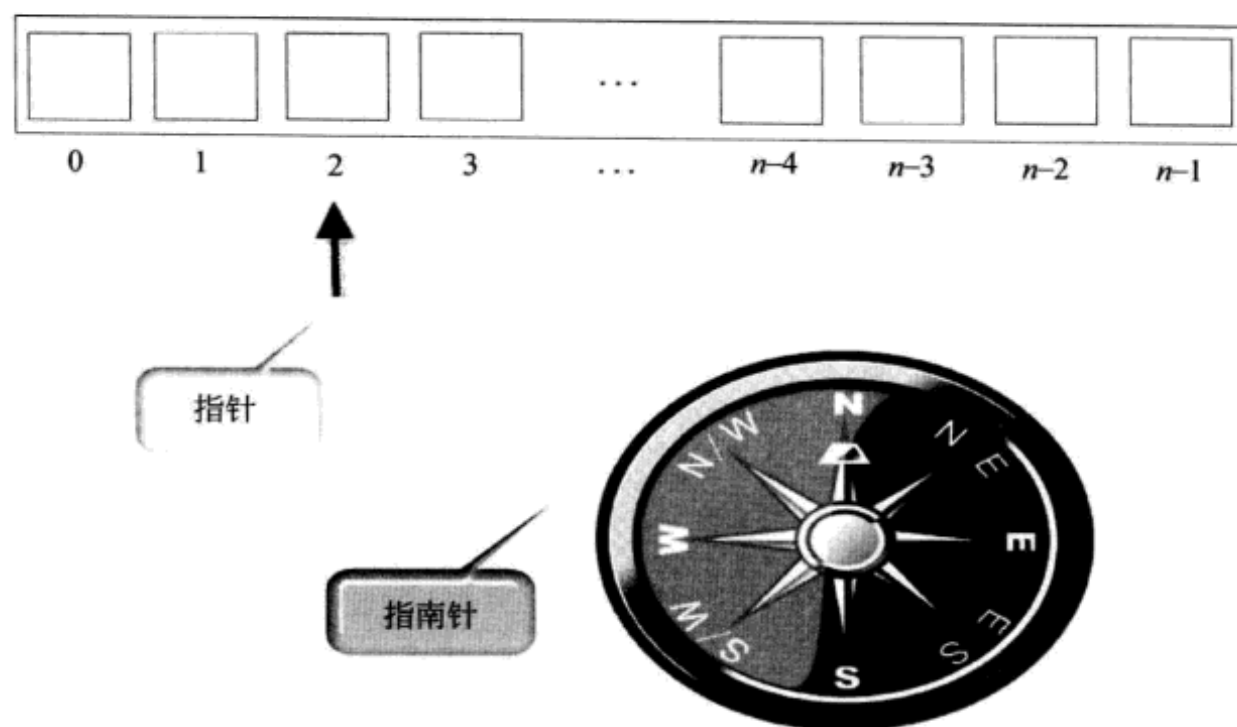


图 7.6 指针和指南针

地址表示一个位置，指针指向这个地址表示的位置，本质上它们是一个东西。只不过在谈到计算机内存的时候，用到地址的概念会多点，谈到程序的时候，用到指针的概念会多点。

7.2.2 指针类型

既然指针是程序语言中的概念，那么程序语言就应该有表示和保存指针的东西了。C 语言用指针类型来表示指针，用指针变量来保存指针，因而指针变量也被叫做指针类型变量。

指针类型在 C 语言中是一种类似于数组的高级数据类型，专门用来表示指针。换成计算机中的概念，就是专门用来表示计算机的内存地址的。那么 C 语言中的指针类型是如何表示的呢？之所以说指针类型是一种高级数据类型，是因为它的表示使用了其他数据类型，如 `int`、`char`、`float` 等。这在 C 语言指针类型的表示中可见一斑，其表示如下所示：

数据类型 *

这里的“数据类型”就是一种数据类型的关键字，当然可以是基本数据类型，也可以是其他的数据类型，例如，将在后面讲到的结构体数据类型等。“*”是指针类型的标志，类似于数组类型中的中括号“[]”。

既然指针是表示数据的内存地址的，那么为什么指针类型前面还要一个数据类型呢？难道不同数据类型数据的内存地址是不一样的？不同数据类型的数据的内存地址是一样的。但是，指针类型的表示中之所以要有一个数据类型，是因为 C 语言是一种有类型的语言。C 语言要求指针也是有类型的，不同类型的指针只能表示不同类型数据的内存地址。这样既安全又方便一些指针操作。

例如，需要一个专门用于表示整型数据地址的指针，就可以使用下面的指针类型：

```
int *
```

7.2.3 指针变量的定义和使用

有了指针类型，就可以定义指针变量了，用指针变量来保存指针的值，也就是数据在内存中的地址。C语言中指针变量的定义类似于基本数据类型变量的定义，其形式如下：

指针类型 指针变量；

“指针类型”就是上面讲到的“int*”、“float*”、“char*”等的指针类型表示。变量名就是符合C语言标识符的任何数字、字母和下划线的组合。

例如，要定义一个整型指针类型的指针变量 address，可以使用下面的表示方法：

```
int *address;
```

有了指针变量，就可以把用取址运算得到的地址保存到指针变量中了。例如，使用下面的程序把变量 radius 的内存地址保存到整型指针变量 address 中。

```
#include<stdio.h>

int main()
{
    int radius;
    int * address;

    address = &radius;
    printf("%x\n",address);

    return 0;
}
```

由于C语言不可以直接输出指针类型的数据，因此上面的程序中，把指针变量中的数据按照整型数据的样子输出，这样就可以看到 radius 变量的内存地址了。程序的输出如图 7.7 所示，radius 变量的内存地址为十六进制数 12ff7c。



图 7.7 指针变量

7.2.4 void 指针

我们知道C语言中的指针是有类型的，不同类型的指针只能表示不同类型数据的内存地址。这样的话，要求太严格了，有的时候，人们在使用指针的时候事先真的不知道指针到底是什么类型的。所以，随着C语言的不断发展和完善，出现了一种特殊的指针类型——void 指针类型。

void 指针也被称为空指针或者无类型指针。使用这种指针的时候,事先可以知道要表示什么类型数据的内存地址。有了 void 指针,就可以用 void 指针变量随便地保存任何数据类型的内存地址了。void 指针变量的定义形式如下:

```
void * 变量名;
```

例如,定义一个空指针变量 address,就可以使用下面的 C 语言表示。

```
void* address;
```

当然,也可以使用空指针变量来保存任意类型数据的内存地址,就像下面这段程序中,依次定义了三个类型的变量,然后使用 void 指针保存其内存位置。

```
#include<stdio.h>

int main()
{
    int    num;
    float  radius;
    char   name;

    void *address;

    address = &num;
    printf("num    addr = %x\n",address);
    address = &radius;
    printf("radius addr = %x\n",address);
    address = &name;
    printf("name   addr = %x\n",address);

    return 0;
}
```

这个程序和前面的程序功能一样,都是使用 address 保存变量的内存地址,然后使用 printf()函数输出。在输出的时候是按照十六进制的整型进行输出的,程序的输出如图 7.8 所示,分别输出了三个不同类型数据的内存地址。

```
#include<stdio.h>

int main()
{
    int    num;
    float  radius;
    char   name;

    void *address;

    address = &num;
    printf("num    addr = %x\n",address);
    address = &radius;
    printf("radius addr = %x\n",address);
    address = &name;
    printf("name   addr = %x\n",address);

    return 0;
}
```

```
C:\WINDOWS\system32\cmd
num    addr = 12ff78
radius addr = 12ff7c
name   addr = 12ff77
请按任意键继续...
```

图 7.8 void 指针

7.3 指针运算

指针也是一种计算机数据，就像整数、小数、字符一样。指针也是有专门的数据类型来表示的，那就是指针类型。像基本数据类型的加减乘除运算一样，对于指针类型，C语言有专门针对指针的运算。本节中就专门介绍一下有关指针的运算。

7.3.1 取指针元素

既然指针表示的是计算机数据在内存中的位置，那么我们如果用指针去直接访问，或者获取计算机中的数据岂不是会很方便？当然是了，指针的最直接的意义就在于此。因而，C语言提供了一种运算来用指针获取计算机中的数据——取指针元素运算。

取指针元素运算，顾名思义，就是使用指针取得指针所指的内存地址处的数据。它的C语言表示如下所示：

*指针变量

在这个表示中，“*”表示的就是取指针运算符，和基本运算乘法的运算符“*”模样是一样的。但是，含义不同，操作数个数和类型也不同：乘法运算的操作数是两个，而取指针元素运算的操作数是一个；乘法运算的操作数是整型或者浮点型，而取指针元素运算的操作数是指针类型。其中的“指针变量”就是声明定义过的指针变量名。

例如，定义了一个整型变量 `radius`，并用一个整型指针变量 `addr` 保存了 `radius` 的内存地址。之后就可以使用取指针元素运算得到 `radius` 变量中的保存的内容了，完整的程序如下：

```
#include<stdio.h>

int main()
{
    int radius = 3;
    int *addr;
    int value;

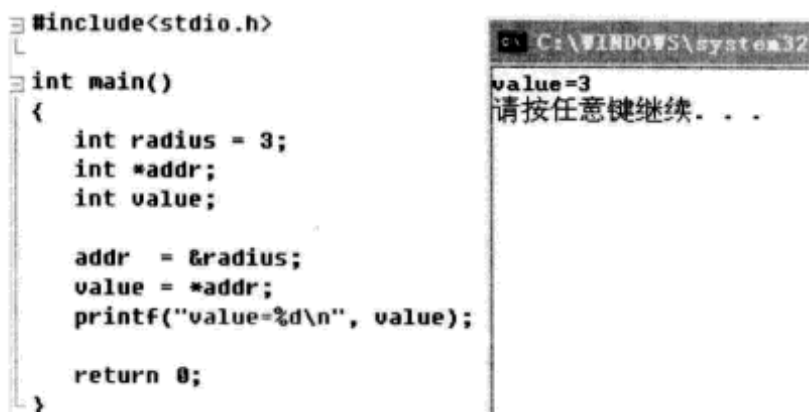
    addr = &radius;
    //对 radius 进行取址运算，radius 整型变量的地址将会被保存到 addr 指针变量中
    value = *addr;
    printf("value=%d\n", value);

    return 0;
}
```

程序中用取址运算把 `radius` 的地址保存在整型指针变量 `addr` 中，然后又使用取指针元素运算把指针 `addr` 所指地址处的值拿出来赋值给整型变量 `value`，最后程序使用 `printf()` 函数输出了 `value` 的值。程序的输出如图 7.9 所示，结果为 3。证明取指针元素运算确实取出了 `radius` 变量的值。

上面的例子是一个正常的使用指针进行取址运算的例子，所以执行正常，并得到了想

要的结果。如果不正常地使用指针会怎么样呢？例如，第一种情况：使用了一个没有赋值的指针变量来进行取指针元素运算，也就是不知道要取内存中什么地址的数据。第二种情况：使用指针变量去取没有保存数据的变量中的数据，也就是从一个空的箱子中取东西。现在对上面的例子稍作修改，就可以看到在这两种不正常的情况下，会发生什么情况。



```
#include<stdio.h>

int main()
{
    int radius = 3;
    int *addr;
    int value;

    addr = &radius;
    value = *addr;
    printf("value=%d\n", value);

    return 0;
}
```

Output: value=3
请按任意键继续...

图 7.9 取指针元素运算

```
#include<stdio.h>

int main()
{
    int radius;
    int *addr;
    int value;

    value = *addr;
    printf("addr=%x value=%d\n", addr, value);

    addr = &radius;
    value = *addr;
    printf("addr=%x value=%d\n", addr, value);

    return 0;
}
```

在这个程序中，第一个 printf() 之前，并没有给整型指针变量赋值，然后就用取址指针运算取出了 addr 表示的地址处的数据赋值给 value 了，符合第一种情况。然后，输出了 addr 表示的地址和 value 的值。之后，用取址运算得到了变量 radius 的内存地址保存在 addr 整型指针变量中，接着使用取指针元素取出 addr 表示的指针处的数据，也就是 radius 中的数据。但是，radius 并没有赋值啊，这就符合第二种情况了，最后还是输出 addr 表示的地址和 value 的值！

程序输出如图 7.10 所示，第一种情况下，程序输出了地址 781c37e4 处的数据 3883024。由于我们并没有给指针变量 addr 赋值，所以 addr 的值是随机的，本次程序运行是 781c37e4，下次就不知道是什么了！在第二种情况下，程序的 radius 的地址 12ff7c 处的值 2015115236。由于我们并没有在 radius 中保存数据，所以其中的数据也是随机的，本次程序运行的时候是 2015115236，下次也不知道是什么了！

所以，大家在使用指针变量进行取指针元素运算时，应该倍加小心，至少应该清楚取的是什么地址的数据，这个地址处有没有想要的数。不然，轻则取指针元素运算不会达到你想要的结果，重则导致程序崩溃，甚至会导致系统崩溃！所以说指针是 C 语言的灵

魂，意义就在于此。它能很方便地取想要的数，但是用不好就会导致致命的破坏！

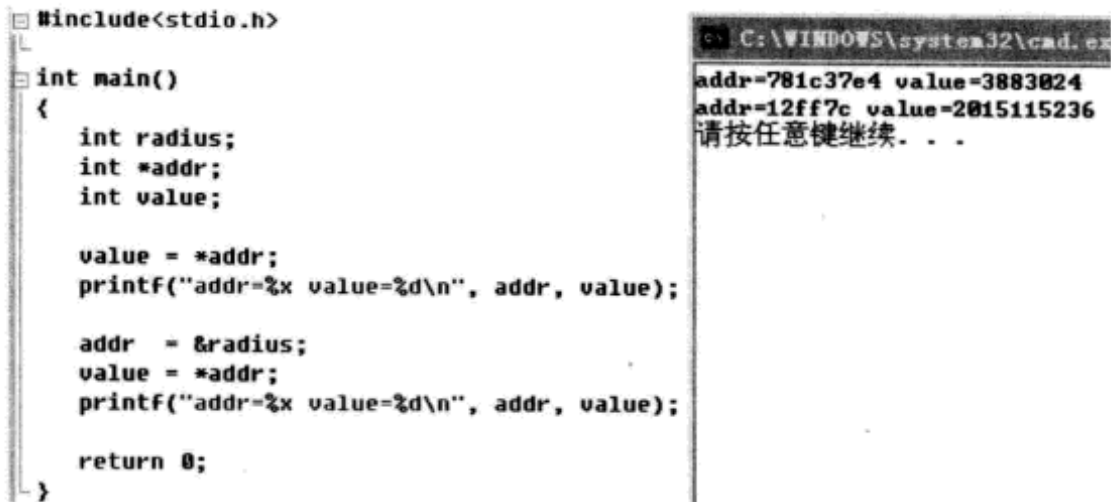


图 7.10 不正常地使用指针

7.3.2 指针的自增自减

和整型数据变量类似，指针也是有自增自减运算的，都是增减“1”。但是其含义却有很大的区别。主要原因就是指针类型不是一般的类型，指针也不是一般的数据，随便自增自减会导致意想不到的现象。所以，C 语言对指针的自增自减运算有严格的规定。

C 语言中指针变量的自增自减运算的表示形式和整型变量的自增自减运算形式是一样的。如下所示前两行就是指针的自增运算的表示形式，后两行是指针的自减运算的表示形式。

```

指针变量++;
++指针变量;
指针变量--;
--指针变量;

```

指针变量的自增自减运算也分为右自增、左自增、右自减、左自减，上面依次是这几种运算的表示形式。对于左自增、右自增、左自减、右自减，其中左右的含义与整型变量自增自减中的左右是一样的。左：先自增（减）、再赋值、最后进行其他运算，右：先自增（减）、再进行其他运算、最后赋值。

指针变量自增自减与整型变量自增自减唯一不同的就是增减的数值，虽然都是增减“1”，但是这个“1”是不一样的。整型变量增减的“1”就是数值的 1，指针变量增减的“1”根据指针变量的类型不同而不同，对于指针变量，这个增减的“1”表示的是指针所指数据在内存中所占的字节数。对于整型指针变量自增就是加 4，自减就是减 4，因为 1 个整型数据在内存中占 4 个字节。

从下面的程序中，可以看出整型变量自增和指针类型自增的区别。

```

#include<stdio.h>

int main()
{
    int radius;
    int *addr;

```



```

radius=0;
addr=&radius;
printf("radius=%d addr=%d\n",radius,addr);

radius++;
addr++;
printf("radius=%d addr=%d\n",radius,addr);

return 0;
}

```

在程序中，开始的时候给整型变量 `radius` 赋值为 0，给指针变量 `addr` 赋值为 `radius` 的地址，然后输出 `radius` 和 `addr` 的值。接着，对 `radius` 和 `addr` 分别进行自增运算，最后还是输出 `radius` 和 `addr` 的值。程序的输出如图 7.11 所示。通过两次输出结果的比较，可见整型变量的自增却实是加了一个 1，但是整型指针变量的自增却加了 1 个整型数据所占的字节数 4。

```

#include<stdio.h>

int main()
{
    int radius;
    int *addr;

    radius=0;
    addr=&radius;
    printf("radius=%d addr=%d\n",radius,addr);

    radius++;
    addr++;
    printf("radius=%d addr=%d\n",radius,addr);

    return 0;
}

```

```

C:\WINDOWS\system32
radius=0 addr=1245052
radius=1 addr=1245056
请按任意键继续. . .

```

图 7.11 指针变量自增

有人会问，指针自增自减时，增减的都是指针所指位置数据所占的内存字节数，那么 `void` 指针自增自减的时候，增减的到底是多少呢？`void` 指针指向的数据所占的内存字节数，到底是多少呢？对于这个问题，答案很简单：“不知道”！因为 C 语言规定，`void` 指针是不能进行自增自减运算的。

例如，下面的程序对 `void` 指针进行自增运算，编译时就会出现错误。

```

#include<stdio.h>

int main()
{
    int radius;
    void *addr;

    addr = &radius;
    addr++;

    return 0;
}

```

这个程序在编译的时候会出现如图 7.12 所示的错误显示：`void` 指针不知道尺寸大小！这也可以回答上面的两个问题了。`void` 指针自增自减的时候，增减多少？不知道。`void` 指针指向的数据所占的内存字节数到底是多少，也不知道。

```

1>----- 已启动全部重新生成: 项目: test, 配置: Release Win32 -----
1>正在删除项目“test”(配置“Release|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>\t.c(9) : error C2036: 'void *' : unknown size
1>生成日志保存在“file:///g:/C语言的书/C语言的书/test/Release/BuildLog.htm”
1>test - 1 个错误, 0 个警告
===== 全部重新生成: 0 已成功, 1 已失败, 0 已跳过 =====

```

图 7.12 void 指针自增错误

7.3.3 指针的类型转换

指针作为一种数据类型,也可以像基本数据类型一样,进行强制类型转换和隐式类型转换。指针类型的转换规则和基本数据类型转换规则基本一致。

1. 指针的强制类型转换

可能有人知道了 void 指针不能自增自减,又有问题要问了! void 指针不能自增自减,那么如果数据类型不确定,使用了 void 指针,之后又要对指针进行自增自减运算,岂不是没办法了?其实,办法还是有的,那就是指针类型的强制转换,把无类型的 void 指针强制转换成其他有类型的指针。

指针类型的强制转换和之前的基本类型之间的强制转换的意义和形式都是类似的。无论是哪种强制类型转换都是为了使用转换之后的类型的某些性质,在这里就是为了使 void 指针能够使用其他类型指针的自增自减运算的性质。

理论上来说, void 指针是可以强制转换成其他任何数据类型的,包括基本数据类型。但是,实际上使用的时候,基本上都是将 void 指针强制转换成其他的指针类型。void 指针强制转换成其他类型指针的 C 语言表示如下所示:

```
(其他非 void 指针类型)void 指针变量名;
```

其中,小括号里面的“其他非 void 指针类型”表示 int*、char*、float*等。“void”指针变量名,就是声明定义的 void 指针变量的名字。下面这段程序将 void 指针转换成其他类型指针,然后再进行自增自减运算。

```

#include<stdio.h>

int main()
{
    int        num;
    float      radius;
    char       name;

    void *addr;

    addr = &num;
    printf("int [%x] ->",addr);
    addr = ((int*)addr)++;
    printf("[%x]\n",addr);

    addr = &radius;
    printf("float [%x] ->",addr);
    addr = ((float*)addr)++;
    printf("[%x]\n",addr);
}

```

```

    addr = &name;
    printf("char [%x] ->", addr);
    addr = ((char*)addr)++;
    printf("[%x]\n", addr);

    return 0;
}

```

程序中，先定义了3个不同类型的变量：num、radius、name。然后，定义了一个 void 指针变量 addr。接着，对每个变量依次进行下面的操作：取变量地址保存到 addr 中、输出地址 addr 的值、把 void 指针变量 addr 强制转换成其他指针类型并自增、输出 addr 的值。

程序的输出如图 7.13 所示，显示了把 void 指针强制转换成 int、float 和 char 这三种类型指针之后，进行自增运算的地址值的变化。从图 7.13 中可以看出，整型数据在内存中占 4 个字节，因为 void 指针按照整型指针强制转换之后，自增，地址从 12ff78 变为 12ff7c 了。另外，也可以看出 float 类型数据在内存中也占 4 个字节，char 类型数据在内存中占 1 个字节。



```

#include<stdio.h>

int main()
{
    int        num;
    float      radius;
    char       name;

    void *addr;

    addr = &num;
    printf("int [%x] ->", addr);
    addr = ((int*)addr)++;
    printf("[%x]\n", addr);

    addr = &radius;
    printf("float [%x] ->", addr);
    addr = ((float*)addr)++;
    printf("[%x]\n", addr);

    addr = &name;
    printf("char [%x] ->", addr);
    addr = ((char*)addr)++;
    printf("[%x]\n", addr);

    return 0;
}

```

C:\WINDOWS\system32\cmd.
int [12ff78] ->[12ff7c]
float [12ff7c] ->[12ff80]
char [12ff77] ->[12ff78]
请按任意键继续. . .
搜狗拼音 半:

图 7.13 void 指针强制转换之后自增

2. 指针的隐式类型转化

在上面写的代码中，多处使用了类似下面的代码：

```
printf("[%x]\n", addr);
```

把指针类型的数据按照整型的十六进制进行输出了。整型和指针类型是两种不同的类型，为什么就可以把指针类型数据按照整型数据输出呢？

这是因为在输出之前，发生了隐式的类型转换，计算机自动地根据输出格式“%x”将指针类型数据隐式地转换成了整型的数值，然后将这个数字输出。从图 7.14 所示的编译警告也可以看得出来，警告显示，输出格式和 void* 类型有冲突。

```

1>----- 已启动全部重新生成: 项目: test, 配置: Release Win32 -----
1>正在删除项目“test”(配置“Release|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>.\\t.c(12) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>.\\t.c(14) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>.\\t.c(17) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>.\\t.c(19) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>.\\t.c(22) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>.\\t.c(24) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'void *'
1>正在链接...
1>正在生成代码
1>已完成代码的生成
1>正在嵌入清单...
1>生成日志保存在“file:///c:/VC语言的书/VC语言的书/test/Release/BuildLog.htm”
1>test - 0 个错误, 6 个警告
===== 全部重新生成: 1 已成功, 0 已失败, 0 已跳过 =====

```

图 7.14 隐式类型转换的编译警告

注意：警告在 C 语言中一般不会对程序产生很大的影响，就是告诉你程序中发生了非常规的操作。

指针类型的隐式转换和之前讲的基本数据类型在赋值的时候发生隐式的类型转换是一样的。指针类型在赋值给其他类型变量的时候也会发生隐式的类型转换。像下面的程序，将指针赋值给整型就发生了隐式的类型转换。

```

#include<stdio.h>

int main()
{
    int radius;
    int addr;
    int* p_addr;

    p_addr = &radius;
    addr = p_addr;
    printf("addr = %x\n",addr);

    return 0;
}

```

这个程序在编译的时候就会发生如图 7.15 所示左边的警告，警告显示类型不一致。程序的输出如图 7.15 右边的样子，输出了 radius 变量在内存中的地址，也就是 addr 按照整型十六进制显示的值。

```

1>----- 已启动全部重新生成: 项目: test, 配置: Release Win32 -----
1>正在删除项目“test”(配置“Release|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>.\\t.c(10) : warning C4047: '=' : 'int' differs in levels of indirection from 'int *'
1>正在链接...
1>正在生成代码
1>已完成代码的生成
1>正在嵌入清单...
1>生成日志保存在“file:///c:/VC语言的书/VC语言的书/test/Release/BuildLog.htm”
1>test - 0 个错误, 1 个警告
===== 全部重新生成: 1 已成功, 0 已失败, 0 已跳过 =====

```

```

#include<stdio.h>

int main()
{
    int radius;
    int addr;
    int* p_addr;

    p_addr = &radius;
    addr = p_addr;
    printf("addr = %x\n",addr);

    return 0;
}

```

C:\WINDOWS\system32\cmd.exe
 addr = 12ff7c
 请按任意键继续...

图 7.15 指针赋值的隐式类型转换

7.4 数组和指针

数组的下标和指针（也就是内存地址）有着类似的作用，都是寻找某个位置的数据，只不过适用的范围不同，指针适用的范围要比数组的下标广得多！既然数组的下标和指针这样类似，那么数组和指针又有怎样的渊源呢？

7.4.1 数组名也是指针

C 语言的数组中的数组名其实也是一个指针，即数组中第一个元素的地址，很惊讶吧？有了指针的概念以后，其实也不难理解，我们可以先使用下面的程序验证一下。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *p_addr;
    int element;

    p_addr = radius;
    printf("radius = %x\n", p_addr);
    element = *radius;
    printf("element = %d\n", element);

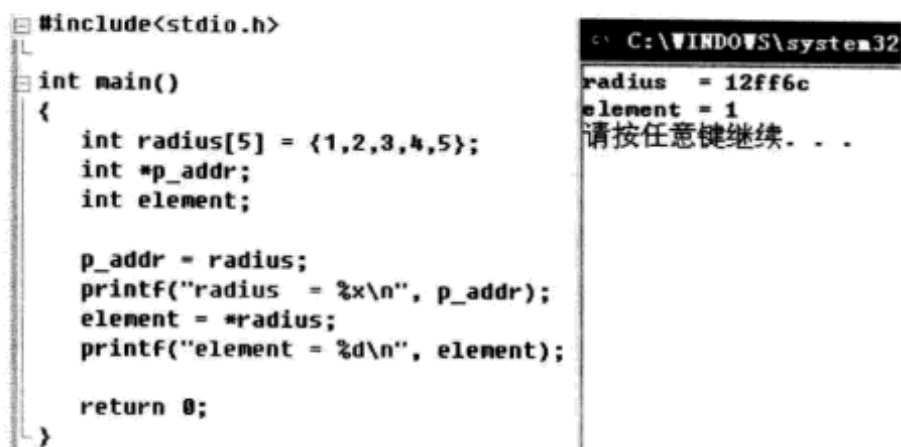
    return 0;
}
```

在编译程序的时候，编译输出如图 7.16 所示，警告只显示 printf() 函数把一个地址按照整型输出会有类型冲突。代码 “p_addr = radius” 将数组名赋值给一个指针变量，显示没有问题，证明数组名就是一个指针，赋值给一个指针变量不会有问題。

```
1>----- 已启动全部重新生成: 项目: test, 配置: Release Win32 -----
1>正在删除项目“test”(配置“Release|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>\t.c(10) : warning C4313: 'printf' : '%x' in format string conflicts with argument 1 of type 'int *'
1>正在链接...
1>正在生成代码
1>已完成代码的生成
1>正在嵌入清单...
1>生成日志保存在“file:///g:/C语言的书/C语言的书/test/Release/BuildLog.htm”
1>test - 0 个错误, 1 个警告
===== 全部重新生成: 1 已成功, 0 已失败, 0 已跳过 =====
```

图 7.16 数组名赋值给指针编译警告

因为 radius 是一个整型数组，所以数组名 radius 就是一个整型指针。我们将数组名 radius 赋值给整型指针变量 p_addr，然后输出 p_addr 的值。接着对指针 radius 进行取指针元素运算，将 radius 所指内存地址处的值赋值给整型变量 element，之后输出 element 的值。程序的输出如图 7.17 所示，数组名所表示的地址为 12ff6c，数组名所指向的地址确实是数组的一个元素所在的内存位置，因为 element 的值为 1，所以就是数组的第一个元素的值。



```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *p_addr;
    int element;

    p_addr = radius;
    printf("radius = %x\n", p_addr);
    element = *radius;
    printf("element = %d\n", element);

    return 0;
}
```

```
C:\WINDOWS\system32
radius = 12ff6c
element = 1
请按任意键继续. . .
```

图 7.17 数组名赋值给指针输出

7.4.2 数组名是指针常量

我们已经知道，数组名是一个指针，那么它到底是一个指针变量还是一个指针常量呢？如果它是指针变量，就可以对它进行赋值了。如果不是，那么它只能参与其他有关常量的运算。所以，使用数组名这个指针之前，得先弄清楚，它是一个变量还是一个常量。

可以用下面的程序验证一下，数组名是指针变量还是指针常量。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int num;

    radius = &num;

    return 0;
}
```

在程序中，使用取址运算得到整型变量 `num` 的地址，赋值给数组名 `radius`。如果 `radius` 是一个指针变量，这段程序将不会有任何问题，否则，程序将会出错。这段程序在编译的时候，输出如图 7.18 所示。编译出错了，错误显示赋值运算的左边应该是变量。由此可见数组名是一个指针常量，不可以作为一个变量使用。

```
1>----- 已启动全部重新生成: 项目: test, 配置: Release Win32 -----
1>正在删除项目“test”(配置“Release|Win32”)的中间文件和输出文件
1>正在编译...
1>t.c
1>\t.c(8) : error C2106: '=' : left operand must be l-value
1>生成日志保存在“file:///c:/c语言的书/c语言的书/test/Release/BuildLog.htm”
1>text - 1 个错误, 0 个警告
===== 全部重新生成: 0 已成功, 1 已失败, 0 已跳过 =====
```

图 7.18 赋值给数组名

7.4.3 使用数组名访问数组元素

既然数组名是一个指针，那么就可以使用数组名来访问数组了。由于数组中的元素在内存中是一个挨着一个放的，并且数组名指向的是数组中的第一个元素，因此，可以用数组名加上一个数来访问数组中的元素。

又有一个问题出现了，为了访问一个数组元素，到底该把数组名加多少呢？为了回答这个问题，先来看 C 语言的规定：给数组名加减 1，相当于给数组名对应的指针加减一个所指元素所占的字节数的值。

这个规定和指针变量自增自减类似，但是不能给数组名自增自减，因为前面已经说了，数组名是一个指针常量，不能进行自增自减运算！为了用数组名访问数组元素，只能使用适合常量的加法运算了。下面的程序使用数组名来依次访问数组中的每个元素，效果和使用下标取数组元素运算是一样的。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *addr;

    addr = radius + 0;
    printf("radius[0] = %d\n", *addr);

    addr = radius + 1;
    printf("radius[1] = %d\n", *addr);

    addr = radius + 2;
    printf("radius[2] = %d\n", *addr);

    addr = radius + 3;
    printf("radius[3] = %d\n", *addr);

    addr = radius + 4;
    printf("radius[4] = %d\n", *addr);

    return 0;
}
```

在程序中，每次给数组名加一个值来得到相应位置元素的地址，并将这个地址赋值给一个指针变量。然后再用 `printf()` 函数使用取指针元素运算将数组中元素的值取出来输出。程序的输出如图 7.19 所示，和我们期望的一样，程序输出了数组中相应位置处元素的值。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *addr;

    addr = radius + 0;
    printf("radius[0] = %d\n", *addr);

    addr = radius + 1;
    printf("radius[1] = %d\n", *addr);

    addr = radius + 2;
    printf("radius[2] = %d\n", *addr);

    addr = radius + 3;
    printf("radius[3] = %d\n", *addr);

    addr = radius + 4;
    printf("radius[4] = %d\n", *addr);

    return 0;
}
```

```
C:\WINDOWS\system32
radius[0] = 1
radius[1] = 2
radius[2] = 3
radius[3] = 4
radius[4] = 5
请按任意键继续...
```

图 7.19 使用数组名访问数组元素的输出结果

7.4.4 三种访问数组元素的方法

数组的下标和指针都可以用来寻找某个位置的数据。用指针去访问数组中的元素会不会更加方便呢？回答这个问题之前，先来看看访问数组元素的几个方法吧！看完之后，你再好好比较一下！

1. 使用数组下标

关于使用数组下标访问数组元素之前讲解过，在这里我们再回顾一下，另外可以与指针进行对比。下面的程序使用数组的下标访问数组 `radius` 中的每个元素，然后将其使用 `printf()` 函数输出。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int i;
    int element;

    for(i=0;i<5;i++)
    {
        element = radius[i];
        printf("radius[%d] = %d\n",i,element);
    }
    return 0;
}
```

在这个程序中，使用 `for` 循环结构依次取出 `radius` 数组中的每个元素，并赋值给整型变量 `element`，然后将数组元素位置和 `element` 的值都输出。

2. 使用数组名

使用数组名访问数组元素的方法，我们也不陌生，因为上一节刚讲过。数组名是指向数组第一个元素的指针常量，可以使用这个指针常量来访问数组元素。下面的例子就使用 `radius` 这个数组名访问数组中的每个元素，然后将其使用 `printf()` 函数输出。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int element;
    int i;

    for(i=0;i<5;i++)
    {
        element = *(radius+i); //使用取指针元素运算，取出 radius+i 地址处的数据
        printf("radius[%d] = %d\n",i,element);
    }
    return 0;
}
```


这个程序和前面用数组下标访问数组元素的例子类似，只是在这个例子中使用的是数组名，这个指针常量进行加法运算，然后用取指针元素得到数组中相应位置的数据。

3. 使用指针变量

因为指针变量比数组名这个指针常量使用起来更加灵活，所以使用指针变量来访问数组中的元素这个方法在C语言中经常用到。下面的程序使用指针变量实现与前面两种方法类似的功能，访问数组 `radius` 中的每个元素，然后使用 `printf()` 函数输出其值。

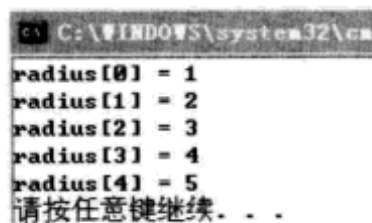
```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int element;
    int i;
    int* addr;

    addr = radius;
    for(i=0;i<5;i++)
    {
        element = *addr;
        printf("radius[%d] = %d\n",i,element);
        addr++;           //指针变量自增，得到数组下一个元素的内存地址
    }
    return 0;
}
```

这段程序中使用了指针变量，开始时指针变量 `addr` 保存数组名，也就是数组元素的首地址，然后使用自增来依次得到数组中的每个元素的地址，通过取指针元素运算得到数组中元素的值。

上面的三种方法实现的功能是一样的，三段程序的输出也是一样的，如图 7.20 所示，都输出了数组中的每个元素及其在数组中的位置。



```
C:\WINDOWS\system32\cmd
radius[0] = 1
radius[1] = 2
radius[2] = 3
radius[3] = 4
radius[4] = 5
请按任意键继续...
```

图 7.20 三种访问数组元素方法的结果

使用数组的下标、数组名和指针变量都可以访问数组中的元素，但是哪个方法更好、更灵活呢？其实，我觉得都差不多，关键在于个人喜好。不过，大多数人都喜欢使用下标和指针变量的方法来访问数组元素。

7.4.5 数组指针和指针数组

“数组指针”，按照名字拆开来看就是“数组”的“指针”。“指针”其实就是地址，所以“数组指针”就是数组的地址。那么数组的地址又是什么呢？回答这个问题之前，先

来回答这样的一个问题，如果你需要一个数组，你最想知道的是什么？答案是：“数组名”！有了数组名，无论是使用下标、数组名、还是指针变量，都可以得到数组中的元素。其实，数组的地址就是数组的第一个元素的地址，也叫首地址，也就是我们说的数组名。

“指针数组”，按照名字拆开来看就是“指针”的“数组”，也就是数组中的元素的数据类型是指针类型。这个好理解，把基本数据类型的数组中的元素换为保存指针（地址）的指针类型就可以了。

那么，指针数组到底长什么样？其实也没什么特别的，就是把一般数据类型的表示换为指针类型表示就可以了，如下所示：

```
指针类型 数组名[表达式];
```

这个表达式是一个一维指针数组的声明定义形式，“指针类型”就是类似于 `int*`、`float*`、`char*` 甚至 `void*` 这样的指针类型表示。“数组名”就是给定义的数组所取的名字，也就是数组的首地址，或者说数组指针。“表达式”就是数组的大小，也就是数组中最多可以放的元素的数目。

例如，

```
int *addrs[5];
```

就是一个指针数组，里面可以放整型指针的值。

下面这段程序使用指针数组来保存一系列地址，然后使用取指针元素运算取出每个地址中保存的数组的值。顺使用这一段程序给大家看看指针数组的作用。

```
#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *addrs[5];
    int element;
    int i;

    for(i=0;i<5;i++)
    {
        addrs[i] = &radius[i];
    }

    for(i=0;i<5;i++)
    {
        element = *addrs[i];
        printf("addr[%d] = %x element = %d\n",i,addrs[i],element);
    }

    return 0;
}
```

在这段程序中，使用指针数组 `addrs` 保存了数组 `radius` 中每一个元素的位置。然后又使用取指针元素运算取出 `addrs` 中保存的地址处的元素的值，并赋值给整型变量 `element`。最后，输出地址和对应地址处的数值。程序的输出如图 7.21 所示，如我们所料，指针数组确实保存了数组 `radius` 中每个元素的地址值。

```

#include<stdio.h>

int main()
{
    int radius[5] = {1,2,3,4,5};
    int *addrs[5];
    int element;
    int i;

    for(i=0;i<5;i++)
    {
        addrs[i] = &radius[i];
    }

    for(i=0;i<5;i++)
    {
        element = *addrs[i];
        printf("addr[%d] = %x  element = %d\n",i,addrs[i],element);
    }

    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
addr[0] = 12ff58  element = 1
addr[1] = 12ff5c  element = 2
addr[2] = 12ff60  element = 3
addr[3] = 12ff64  element = 4
addr[4] = 12ff68  element = 5
请按任意键继续. . .

```

图 7.21 指针数组

7.5 多重指针和多维数组

一维数组可以用指针来访问，多维数组能用指针来访问吗？多维数组的数组名又是什么呢？多维数组中每一维存的数据到底是什么呢？带着这些问题，展开本节的讨论！

7.5.1 多重指针

之前见到的都是一重指针，现在来看看多重指针。多重指针？什么是多重指针？这也许是最多人听到这个概念的时候的第一反应！下面就先从“什么是多重指针？”这个问题开始。

我们知道指针变量是用来存数据的地址的，而且每一种指针都是有类型的。`int*`指针是用来存整型数据的地址的，`float*`指针是用来存浮点型数据地址的，`char*`指针是用来存字符型数据地址的等。

有一个问题，指针也是一种数据，是用来表示内存地址的数据，有没有指针变量来保存指针的地址呢？也就是有没有一种指针类型用来保存指针的指针呢，就像 `int*` 这种指针类型用来保存 `int` 数据的指针一样？答案是肯定的。C 语言使用多重指针来保存指针数据的地址，这也就是多重指针的含义与作用。说着还是有点抽象，来看个图吧，如图 7.22 所示。

在图 7.22 中，横向最长的长方形代表的是整个计算机中的内存。这个长方形中的小长方形中保存着数据，数据的上面是用来保存这个数据的变量的名字，数据的下面是数据所在的内存地址。

图 7.22 中有两个常见的变量，一个变量是 `int radius`，是用来保存整型数据的，`radius` 现在保存的是整数 3，`radius` 这个整型变量所在的内存地址为 1732c。另一个变量是 `int* addr`，是用来保存地址的，`addr` 保存的是 `radius` 的地址 1732c，`addr` 这个指针变量现在所在的内存地址为 17330。

如果要保存指针变量 `addr` 的地址，就可以像图 7.22 中那样，在内存地址 25300 处保存 `addr` 的地址 17330。这个时候，内存地址 25300 处的数据就是一个多重指针了，准确地

说是一个二重指针，因为它保存的是指针变量 `addr` 的地址。如果要保存二重指针的地址，如图 7.22 中的 25300，需要一个三重指针。依此类推就知道各重指针的含义了。



图 7.22 多重指针

知道了 C 语言的多重指针是什么意思，再来看看 C 语言中是如何表示多重指针类型的。其形式如下：

指针类型 *

这个表示中，“指针类型”可以是一个一重指针类型，如 `int*`、`float*`、`char*` 等。当“指针类型”是一个一重指针的时候，上面的表示就是一个二重指针类型，即 `int**`、`float**`、`char**` 等。当“指针类型”是一个二重指针的时候，上面的表示就是一个三重指针类型，即 `int***`、`float***`、`char***` 等。依此类推可以得到其他多重指针类型。不过在 C 语言中最长使用的就是一重和二重指针，三重和更高重的指针很少用。

好，有了多重指针类型的表示之后，多重指针变量的 C 语言表示就好说多了，其形式如下：

多重指针类型 多重指针变量；

这个表示中，“指针类型”既可以是类似于 `int*` 的一重指针类型，也可以是类似于 `int**` 的二重指针类型，甚至可以是更高重的指针类型。“指针变量名”就是符合 C 语言命名规范的标识符。

通过下面一段代码，就可以看出多重指针变量的表示和用途。

```
#include<stdio.h>

int main()
{
    int radius;
    int*  addr;
    int** p_addr;

    radius = 3;
    addr   = &radius;
    p_addr = &addr;

    printf("[%x]=%d\n",&radius,radius);
    printf("[%x]=%x\n",&addr,addr);
    printf("[%x]=%x\n",&p_addr,p_addr);
    return 0;
}
```

在这个程序中，依次定义了一个整型变量 `radius`、一个整型指针变量 `addr`、一个二重指针变量 `p_addr`。然后，在整型变量 `radius` 中保存整型常量 3，在整型指针 `addr` 保存 `radius` 的地址，在二重指针变量 `p_addr` 中保存 `addr` 的地址。最后，分别输出各个地址和地址中保存的数值。程序的输出如图 7.23 所示。


```

#include<stdio.h>

int main()
{
    int radius;
    int* addr;
    int** p_addr;

    radius = 3;
    addr = &radius;
    p_addr = &addr;

    printf("[%x]=%d\n",&radius,radius);
    printf("[%x]=%x\n",&addr,addr);
    printf("[%x]=%x\n",&p_addr,p_addr);
    return 0;
}

```

```

C:\WINDOWS\system32
[12ff78]=3
[12ff74]=12ff78
[12ff7c]=12ff74
请按任意键继续. . .

```

图 7.23 多重指针程序验证

从上面的程序输出可以看出,地址 12ff78(radius)中保存的是数值 3,地址 12ff74(addr)中保存的是 radius 的地址 12ff78,地址 12ff7c(p_addr)中保存的是 addr 的地址 12ff74。可以使用图 7.24 来表示这个关系。



图 7.24 多重指针关系

7.5.2 取多重指针元素运算

有了多重指针,如何使用多重指针来访问保存的数据呢?这和一重指针是类似的,可以使用取多重指针元素运算进行。在 C 语言中,取二重指针元素的表示形式如下所示:

****二重指针变量;**

这个表示形式中的两个星号,表示的是进行了两次取指针元素运算。第一次得到的是一个一重指针,第二次得到的就是一重指针所指的元素。这就像图 7.24 中的两个箭头一样,一个箭头就相当于一次取指针元素。对于更多重的取指针元素运算,要根据具体情况来确定需要多少个星号。

下面的例子,就是通过使用取二重指针元素运算得到需要的数据的。

```

#include<stdio.h>

int main()
{
    int radius;
    int* addr;
    int** p_addr;
    int element;

    radius = 3;
    addr = &radius;
    p_addr = &addr;

```

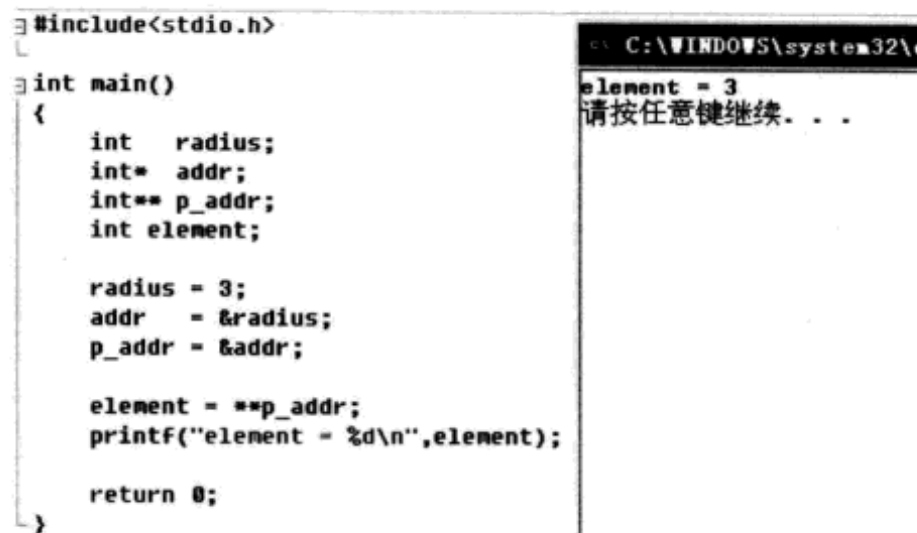
```

    element = **p_addr;
    printf("element = %d\n",element);

    return 0;
}

```

程序的输出如图 7.25 所示，如我们所料，正确地输出了 radius 的值 3。



```

#include<stdio.h>

int main()
{
    int radius;
    int* addr;
    int** p_addr;
    int element;

    radius = 3;
    addr = &radius;
    p_addr = &addr;

    element = **p_addr;
    printf("element = %d\n",element);

    return 0;
}

```

C:\WINDOWS\system32\cmd.exe
 element = 3
 请按任意键继续. . .

图 7.25 取多重指针元素

7.5.3 多维数组名和各维元素

多维数组的数组名就是一个多重指针，除了第一维保存的是基本数据外，其他的维数中保存的元素都是指针。

下面举个例子来看看多维数组名及多维数组各维的元素到底是什么，它们和指针之间到底有什么关系。下面是一段全面剖析数组的例子，使用 printf() 函数输出了数组的各方各面。

```

#include<stdio.h>

int main()
{
    int num[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int i,j;

    printf("[%x]:num=%x\n",num,*num);
    for(i=0;i<3;i++)
    {
        printf("[%x]:num[%d]=%d\n",num[i],i,*num[i]);
        for(j=0;j<4;j++)
        {
            printf("[%x]:num[%d][%d]=%d\n",&num[i][j],i,j,num[i][j]);
        }
    }

    return 0;
}

```

这段程序是不是看着有点复杂？那是因为使用了取址、取指针元素加上多维数组的各种元素，看懂了这段程序，你就“除却巫山不是云了”，所以不要怕难！跟着我一点一点

地分析吧！先来看一下程序的输出，其实输出挺简单的，如图 7.26 所示。其中，每一行中括号中的数字是一个内存地址，冒号和等号之间的部分是这块内存地址所对应的变量，等号后面就是这块内存中保存的数据。从数组名开始，到数组第二维，再到数组的第一维，依次展示了数组的各个方面。

图 7.26 中的输出可以用更形象的方式表示出来，如图 7.27 所示。在图 7.27 中灰底小长方形中的数字是内存地址，也就是指针数值。小正方形中的数值是数组中存的元素。白底小长方形中的名字是数组名或者它的各维元素。

```

#include<stdio.h>
int main()
{
    int num[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int i,j;

    printf("[%x]:num=%x\n",num,num);
    for(i=0;i<3;i++)
    {
        printf("[%x]:num[%d]=%d\n",
            for(j=0;j<4;j++)
            {
                printf("[%x]:num[%d][%d]=%d\n",
                    num[i],j,num[i][j]);
            }
        }
    }

    return 0;
}

```

```

C:\WINDOWS\system32\cmd.exe
[12ff50]->[12ff50]:num=12ff50
[12ff50]->[12ff50]:num[0]=0
[12ff50]:num[0][0]=0
[12ff54]:num[0][1]=1
[12ff58]:num[0][2]=2
[12ff5c]:num[0][3]=3
[12ff60]->[12ff60]:num[1]=4
[12ff60]:num[1][0]=4
[12ff64]:num[1][1]=5
[12ff68]:num[1][2]=6
[12ff6c]:num[1][3]=7
[12ff70]->[12ff70]:num[2]=8
[12ff70]:num[2][0]=8
[12ff74]:num[2][1]=9
[12ff78]:num[2][2]=10
[12ff7c]:num[2][3]=11
请按任意键继续. . .

```

图 7.26 多维数组的各维数据

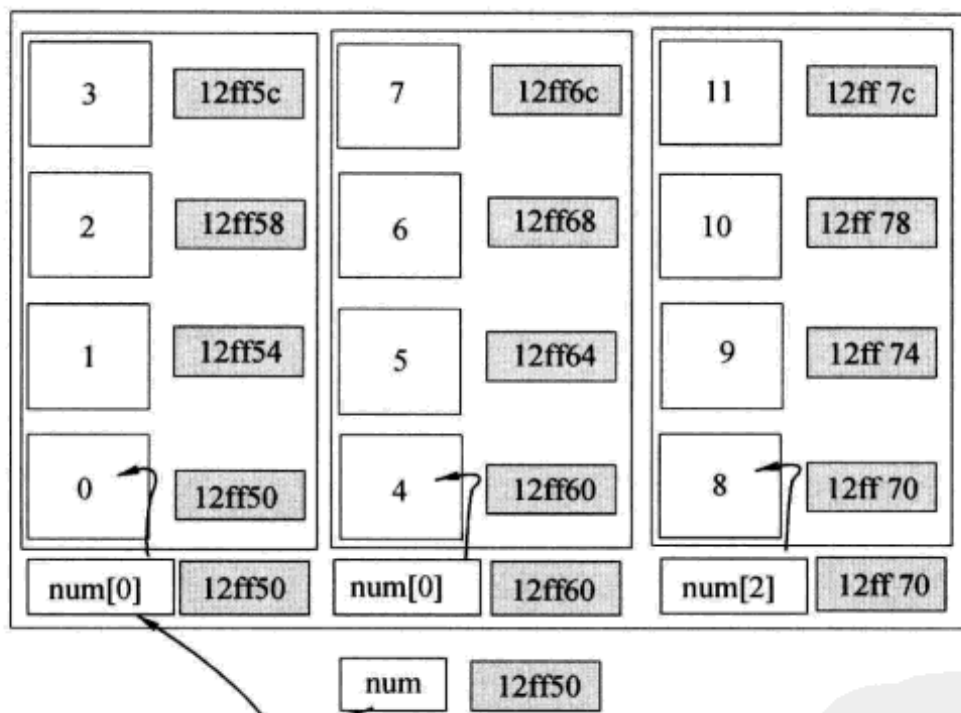


图 7.27 num 数组和指针的关系

这里告诉大家一个 C 语言的规定：多维数组最后一维的元素是数组中保存的数据，剩下其他各维和多维数组名都是指针。

有了这个规定，我们就可以很好地解释图 7.27 了。在图 7.27 中，num 是一个指针，它的值为 12ff50，它指向的是 num[0]，也就是它的下一维的第一个元素。num[0]、num[1] 和 num[2] 都是一个一重指针，它们的值分别为 12ff50、12ff60 和 12ff70，分别指向它们下一维的第一个元素，这里就是数组中保存的数据，依次为：0、4、8。

好了，现在可以总结一下多维数组名和各维元素与指针之间的关系了！很简单，大家只要记住一句话就可以：多维数组中，最后一维保存的是数据，其他各维都是指针。

7.5.4 使用指针访问多维数组

现在已经知道了多维数组名和除最后一维外的其他维元素都是指针。那么，就可以使用指针来访问多维数组中的元素了。具体方法和使用一维数组名访问一维数组元素类似，有三个方法：下标、数组名或各维元素、指针变量。对于使用下标访问多维数组元素就不再赘述了，具体可以查阅第5章“多维数组”一节。现在来讲讲其他两种方法。

因为多维数组名和除最后一维外的其他维元素都是指针，而且是指针常量，所以可以用指针常量的方式来访问数组中的每个元素。先从简单的开始，先来看如何使用数组非最后一维的元素来访问数组中的数据，如下面的例子所示。

```
#include<stdio.h>

int main()
{
    int num[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int i,j;
    int element;

    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            element = *(num[i]+j);
            printf("%d\n",element);
        }
    }

    return 0;
}
```

在这个程序中最关键的一句代码就是 `element = *(num[i]+j);` 将数组 `num` 中的第一维的每个元素 `num[0]`、`num[1]` 和 `num[2]` 加上一个数值，然后再进行取指针元素运算。因为 `num[0]`、`num[1]` 和 `num[2]` 都是一重指针，所以取指针元素运算可以正确进行。

接下来，再来看看使用数组名来访问数组中的数据的例子。

```
#include<stdio.h>

int main()
{
    int num[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int i,j;
    int element;

    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            element = (*(num+i)+j);
            printf("%d\n",element);
        }
    }

    return 0;
}
```


}

在这个程序中最关键的一句代码就是 `element = (*(num+i)+j);`。“`*(num+i)`”得到的将是 `num[0]`、`num[1]`和 `num[2]`，接下来就和前面的例子是一样的了。

最后，再来看看如何使用多重指针来访问数组中的各个元素。多维数组指针变量和其他的指针变量的定义是不同的，下面是一个二维数组指针的定义。

数据类型 (*数组指针名) [表达式];

其中，“数据类型”就是类似于 `int`、`float`、`char` 等这样的数据类型标识符。“数组指针名”就是要定义的多维数组指针变量，只要遵循 C 语言标识符的规定就可以。“表达式”是二维数组中最后一维的长度。例如，`num[3][4]`中的 4 就是 `num` 数组中的最后一维的长度。另外记住，小括号必须得有，不然定义出来就不是一个指针变量，而是 5.6.3 节所讲的指针数组了，数组名是一个常量，而不是一个变量。

C 语言中的数组指针之所以要这样定义，是因为数组各维是有大小的，定义的指针必须知道数组中各维的大小，这样可以在自增自减的时候，方便确定到底增加减少多少。要定义一个更高维的数组指针，就必须知道除最高维以外的其他各维的大小。例如，要定义 `int num[2][3][3][5]`的多维数组指针，形式就是 `int *num[3][3][5]`。

下面的程序使用多维数组指针变量来访问数组中的每个元素。

```
#include<stdio.h>

int main()
{
    int num[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    int i,j;
    int element;
    int (*p_addr)[4];

    p_addr = num;

    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            element = (*(p_addr+j));
            printf("%d\n",element);
        }
        p_addr++;
    }

    return 0;
}
```

这个程序中的关键还是“`element = (*(p_addr+j));`”和“`p_addr++;`”这两句。“`*p_addr`”相当于 `num[0]`、`num[1]`和 `num[2]`，“`p_addr++`”是将 `num[i]`中的 `i` 不断增加 1。以上这三个程序虽然形式不同，但是功能是一样的，输出当然也是一样的，如图 7.28 所示，依次输出了二维数组 `num` 中的所有的元素。

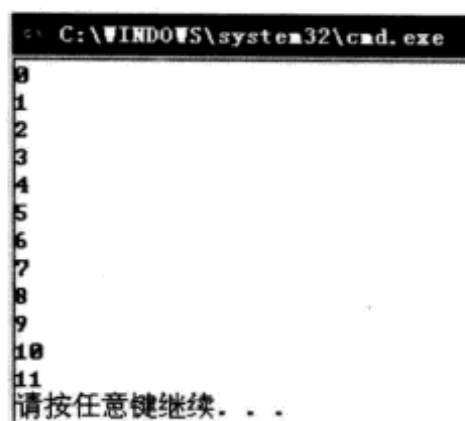


图 7.28 使用指针访问多维数组

7.6 字符串和指针

在 C 语言中，字符串是一种特殊的数组，因而所有针对数组的属性和操作，对于字符串也是适用的。像前面介绍的数组和指针的关系，在字符串和指针之间也存在类似的关系。本节就来讲讲字符串与指针之间的关系。

7.6.1 字符指针

通过前面的学习，我们知道，在 C 语言中指针也是有类型的。不同类型的指针表示的是不同类型的数据在内存中的存储地址。字符数据也是类似的，要表示它在内存中的存储地址，就需要用字符指针。下面先来看看字符指针类型在 C 语言中是如何表示的吧，它的形式如下所示：

```
char*
```

这个表示形式是将一般的指针类型表示中的“数据类型”特例化成为字符型 `char` 了。

有了字符指针类型以后，就可以定义字符指针变量了，其形式和一般的指针变量的定义形式类似，如下所示：

```
char* 字符指针名;
```

例如，要定义一个名为 `p_str` 的字符指针变量，可以使用下面的表示形式。

```
char* p_str;
```

下面的程序使用字符指针变量 `p_str` 来保存字符 `character` 的内存地址，并使用取指针运算取出 `p_str` 所指内存中的字符，再使用 `printf()` 函数输出这个字符。

```
#include <stdio.h>

int main()
{
    char character;
    char value;
    char *p_str;
```

```

character = 'c';
p_str     = &character;
value     = *p_str;

printf("[%x] = %c\n", p_str, value);

return 0;
}

```

这个程序中分别使用了取地址运算和取指针元素运算得到了 `character` 字符变量的地址和这个地址中保存的数据。程序的输出如图 7.29 所示, `character` 的内存地址为 `12ff2b`, 这个内存地址中的数组为字符 `'c'`。程序的输出和我们想象的一样。

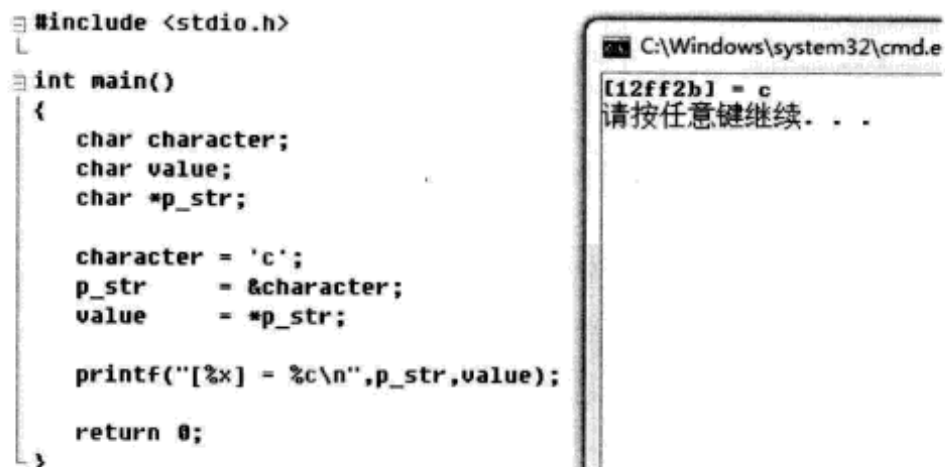


图 7.29 字符指针

7.6.2 字符指针和字符串

C 语言是使用字符数组来表示和存储字符串的。我们已经知道了, 字符指针是可以保存字符数据的地址的。通过指针和数组的关系, 可以知道数组名就是数组首个元素的地址。

通过这三点信息可以知道字符指针是可以用来访问操作字符串的。如果还没想清楚, 听我慢慢道来。因为 C 语言是用字符数组来表示和存储字符串的, 所以可以暂且把字符串和字符数组等价起来。又因为数组名就是数组的首个元素的地址, 那么字符数组名也就是字符数组中首个字符的地址了。再因为字符指针可以用来保存字符的地址, 因而可以用字符指针保存字符数组的首个字符的地址, 也就是字符数组名。这样就可以使用字符指针访问操作字符串了。

图 7.30 表示了上面的逻辑关系, 简单地说就是字符串等价于字符数组。字符指针又可以保存字符数组的首个元素的地址, 进而访问字符数组的每个元素。因此字符指针也是可以访问字符串的每个元素的。

下面的这段程序使用字符指针访问字符串中每个元素, 具体的访问方法遵循图 7.30 所示的逻辑。

```

#include <stdio.h>

int main()
{
    char name[10] = "xiao ming";
    char *p_str;

```

```

char value;
int i;

p_str = name;
for(i=0; i<10; i++,p_str++)    //使用了逗号表达式
{
    value = *p_str;
    printf("[%x] = '%c' =%d\n",p_str,value,value);
}

return 0;
}

```

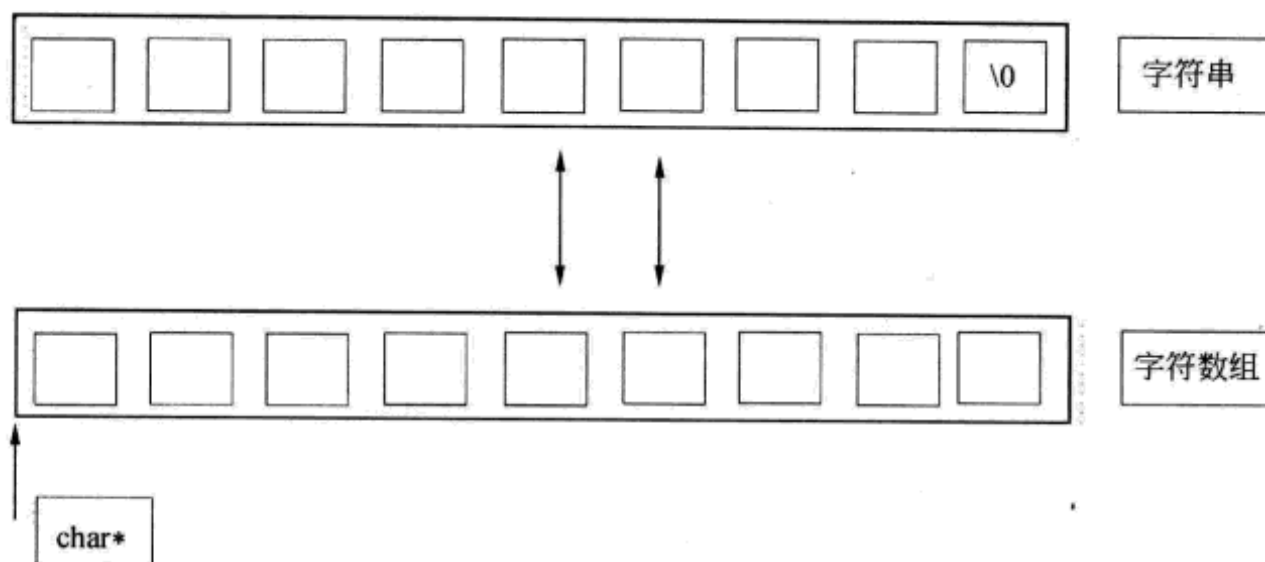


图 7.30 字符指针访问字符串逻辑示意图

在这段程序中，按照 C 语言的要求使用字符数组来保存字符串，然后把字符数组的数组名 `name`，也就是字符数组的首个元素的地址赋值给字符指针变量 `p_str`，之后使用 `for` 循环和字符指针变量自增和取指针元素运算来访问字符串中的每个元素，并将其输出。

程序的输出如图 7.31 所示，中括号中的是字符串中的每个字符的内存地址，两个等号之间的是字符的输出形式，第二个等号之后的输出是对应字符的 ASCII 码值。

```

#include <stdio.h>

int main()
{
    char name[10] = "xiao ming";
    char *p_str;
    char value;
    int i;

    p_str = name;
    for(i=0; i<10; i++,p_str++)    //使用了逗号表达式
    {
        value = *p_str;
        printf("[%x] = '%c' =%d\n",p_str,value,value);
    }

    return 0;
}

```

```

C:\Windows\system32\cmd.e
[12ff1c] = 'x' =120
[12ff1d] = 'i' =105
[12ff1e] = 'a' =97
[12ff1f] = 'o' =111
[12ff20] = ' ' =32
[12ff21] = 'm' =109
[12ff22] = 'i' =105
[12ff23] = 'n' =110
[12ff24] = 'g' =103
[12ff25] = ' ' =0
请按任意键继续. . .

```

图 7.31 使用字符指针访问字符串元素程序

7.6.3 scanf()、printf()函数和字符指针

在使用 `scanf()`和 `printf()`函数的时候，可以使用“`%s`”来直接输入和输出字符串，这是

为什么呢？主要是因为 `scanf()` 和 `printf()` 函数在输入和输出的时候需要的是字符指针！

```
scanf("%s", 字符指针);
```

以上这句表示会将输入的字符串挨个保存到“字符指针”开始的内存地址处。由于字符数组名就是字符数组的第一个元素的地址，因此可以在 `scanf()` 函数的“字符指针”处填写一个字符数组名。

```
printf("%s", 字符指针);
```

同样，`printf()` 函数输出字符串的时候，需要的也是字符指针，字符数组名代表的就是字符数组第一个元素的地址，也就是指针，所以可以用到 `printf()` 函数中进行输出。既然可以使用字符指针，或者说字符指针常量来访问字符串中的元素，那么 C 语言能不能使用字符指针变量直接进行 `scanf()` 输入和 `printf()` 输出呢？

先来看看一个以前没见过的、直接给字符指针变量赋值为字符串常量的表示形式。但是，这只能在初始化的时候使用。

```
char* 字符指针变量 = "字符串";
```

例如，可以使用下面的程序给字符指针 `name` 赋值为字符串“xiao ming”。赋值完成之后，`name` 中保存的是字符串中第一个字符的地址。

```
char* name = "xiao ming";
```

有了字符串的第一个字符的地址就可以使用 `printf()` 函数输出整个字符串了。下面这段程序显示了 `scanf()` 函数和 `printf()` 函数是如何使用字符指针变量的。

```
#include <stdio.h>

int main()
{
    char name[10];
    char *p_name;
    char *p_name1= "xiaoming";

    p_name = name;
    scanf("%s", p_name);

    printf("p_name[%x] = %s\n", p_name, p_name);
    printf("p_name1[%x] = %s\n", p_name1, p_name1);

    return 0;
}
```

这个程序在 `scanf()` 函数中使用了字符指针变量 `p_name`。因为 `p_name` 指向的是字符数组的地址，所以第一个 `printf()` 函数输出了 `p_name` 保存的地址和其中的字符串，第二个 `printf()` 输出的是 `p_name1` 中保存的地址和这个地址中的字符串。

程序的输出如图 7.32 所示。第一行为我们的输入，它将通过 `scanf()` 函数保存到 `p_name` 所在的内存地址。后面的两行输出分别是 `p_name` 和 `p_name1` 对应的字符串的相关信息。

```

#include <stdio.h>

int main()
{
    char name[10];
    char *p_name;
    char *p_name1= "xiaoming";

    p_name = name;
    scanf("%s",p_name);

    printf("p_name[%x] = %s\n",p_name,p_name);
    printf("p_name1[%x] = %s\n",p_name1,p_name1);

    return 0;
}

```

```

C:\Windows\system32\cmd.exe
xiaoming
p_name[12ff1c] = xiaoming
p_name1[415670] = xiaoming
请按任意键继续. . .

```

图 7.32 scanf()和 printf()函数对字符指针的操作

7.7 小 结

在本章中我们学习了 C 语言中的指针，它与数组有很多共同的地方，从本章中也可以看得出来。本章的重点是指针的相关概念和操作，难点是数组和指针的关系，以及它们之间的互相转换。接下来的几章，将介绍几个类似于数组的数据类型——结构体、共同体和枚举，到时候还会接触到指针。下一章我们先从结构体开始吧！

7.8 习 题

【题目 1】 一般情况下，都是使用取地址运算来获取变量的地址的，主要原因是在写程序的时候，不知道计算机将这个变量放到哪个内存的哪个地址。但是，有的时候，我们很确定就是需要哪个内存地址上的数据，该怎么办？例如，我们就是需要内存 0x1111 处的数据。

【分析】 要取得内存中的数据，得使用取指针元素运算，而取指针元素运算需要将地址保存在一个指针变量中，然后才能取这个指针变量中保存地址处的数据。我们需要内存 0x1111 处的数据，可以直接将 0x1111 这个地址赋值给指针变量，再进行取指针元素运算。

【核心代码】

```

int element;
int *addr;

addr = 0x1111;
element = *addr;

```

【题目 2】 使用指针的自增自减运算，可以很方便地得到下一个或者上一个元素的值。但是，有的时候需要得到上 n 个或者下 n 个元素的值（ n 大于 1）该怎么办？写一个程序得到一个元素的下 5 个元素的值。

【分析】 要想得到一个元素的下 5 个元素的值，有两种方法：第一种，指针变量自增

运算进行 5 次；第二种，直接给指针变量增加 5。显然，第二种方法更加方便快捷，因为它可以一次搞定！

【核心代码】

```
char str[10] = "1234567890";  
char *pstr = str;  
  
pstr = pstr+4;  
printf("%c\n",*pstr);
```

【题目 3】 既然数组能实现的功能用指针变量也是可以实现的，为什么还要保留数组这种数据类型呢？何不用指针代之？

【分析】 数组是一种高级的数据结构，之所以说它高级，是因为它规定了数据的存放格式，是按照矩阵的方式来存储的。而指针并不会规定数据是按照怎么样的格式进行存储的，它只是表示数据在哪，不表示数据是怎么样的！数组规定数据格式，指针表示数据在哪，连接数组和指针的桥梁就是数组的数组名，所以指针变量可以实现数组的功能。但是，试想一下，如果没有数组，你只用指针来实现“维”的概念，会有多麻烦！因此，为了直观和操作简单，还是需要数组的。

【题目 4】 说说字符数组、字符指针、字符指针变量和字符串的区别和联系。

【分析】 要想区分这几个字符概念，就得从它们是干什么的出发。字符数组是一个数组，是用来保存多个字符的一种数组类型；字符指针是一个指针，表示一个字符数据所在的内存地址。字符指针变量是一个指针变量，用来保存字符指针，也就是字符数据所在的内存地址。字符串是一个数据类型，用来表示一串字符，C 语言中的字符串要求必须以'\0'作为结束。它们之间主要的联系，就是字符串是用字符数组存储的，字符数组的数组名是一个字符指针，可以通过字符变量来保存这个指针。有这条联系线路，我们就可以通过字符指针变量，实现字符数组和字符串之间的互相操作了。

第 8 章 结 构 体

与数组类型类似，结构体类型也是一种高级的组合数据类型。说它和数组类型类似，是因为它也可以保存多个数据，甚至可以保存不同类型的多个数据。说它是一种高级的组合类型，是因为它是由其他数据类型组成的。

8.1 结构体的含义

数组是用来保存多个同一种类型数据的高级类型，有人会想，要是需要保存多个不同类型数据，该怎么办呢？像下面的形式吗？

```
int    num;  
char  ch;  
float radius;  
...
```

如果在程序中，只有简单的几个不同类型的数据要存储，上面的方法是完全可取的。但是，如果数据量非常多而且杂，上面的方法就有点不合适了。例如，新生开学，需要记录学生的信息。要记录学生甲的姓名、体重、身高、年龄、性别、入学成绩等，学生乙也有同样的信息要记录，学生丙丁也是如此。每个学生的信息，都需要一堆变量来表示；而且每个学生都有姓名、年龄、身高等重要的信息（信息冗余），定义成变量，很有可能会出现同样名字的变量。C 语言不允许同一位置出现相同名字的变量（变量重名）。

为了解决“信息冗余”和“变量重名”这两个定义变量时出现的问题，就可以使用 C 语言提供的另外一个高级数据类型——结构体。C 语言中的结构体类似于数组，数组就像书包，里面只装书这一种类型的东西。而结构体更像是旅行箱，里面可以放衣服、书籍、钱包、甚至是书包……如图 8.1 所示。

在图 8.1 中，书包（类似于数组）只用来装书这种类型的数据，而旅行箱（类型于结构体）就可以用来装衣服、钱包、书和书包等各种类型的数据。好，通过上面的讲解，大家只要知道一点就好：结构体可以用来保存不同类型的数据。以后你写程序的时候，当有这样的需求时只要能想到结构体，上面的讲解就算没有白费。

讲了那么多，还是没有说明结构体是如何解决：“数据量过大（信息冗余）导致记忆混乱”和“命名重复”这两个问题的。这里先留个悬念，等到学完后面的内容，你自然就会理解到结构体是如何解决这两个问题的，还会看到结构体的强大威力！

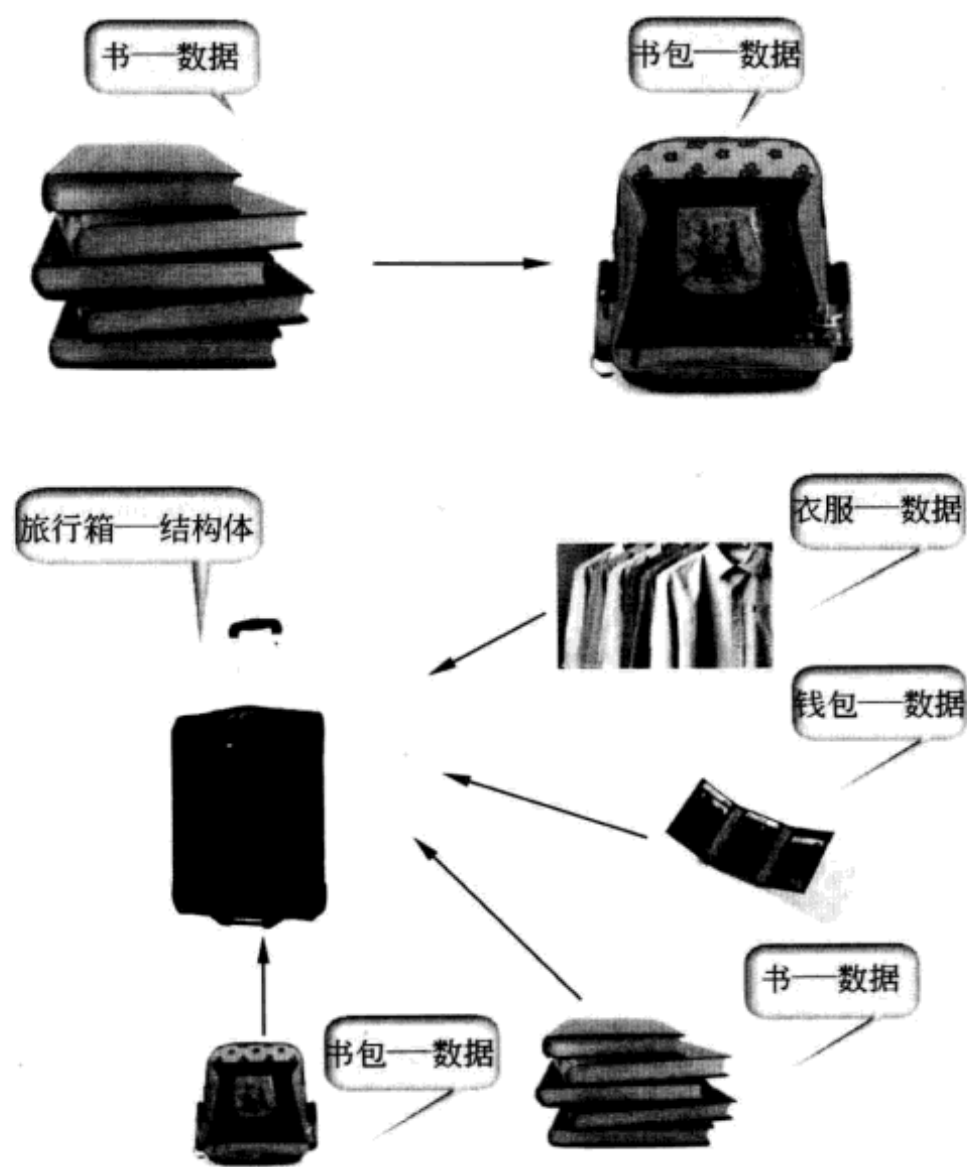


图 8.1 结构体含义示意图

8.2 结构体类型的表示

我们已经知道，结构体可以用来保存多个不同类型的数据。有人会问，那么它是如何保存多个不同类型的数据的呢？不要着急，先来看看 C 语言中的结构体到底长什么样，有了直观的印象，再来看它是如何使用的！

8.2.1 结构体类型的一般格式

```
struct 结构体类型名
{
    数据类型 变量 1;
    数据类型 变量 2;
    .....
    数据类型 变量 n-1;
    数据类型 变量 n;
};
```

上面就是结构体类型的定义，是不是和之前见到的数据类型很不一样？主要是因为结构体是一种自定义的高级类型，各个方面都需要自己来定义，而前面见到的都是系统定义好的数据类型。

上面的结构体定义表示中，“struct”是一个类型关键字，表示将要定义的是一个结构体类型。“结构体类型名”是自己给这个结构体类型取的名字，类似于之前见到的 int、char 等关键字，int、char 是系统规定的，而这里的类型名是我们自己定义的。类型名的定义只要遵循 C 语言标识符的要求就可以。接下来就是一对大括号，大括号之后有一个分号。这是 C 语言规定好的，必须得这样写，不然就会出错。大括号之间是很多变量的声明定义，都以分号作为结束，这些变量被称为结构体的成员变量。

例如，要定义一个 student 的结构体类型，可以使用下面的 C 语言表示。

```
struct student
{
    char name[10];    //姓名
    float weight;     //体重
    int years_old;    //年龄
    char sex;         //性别
    int score;        //入学成绩
};
```

这样，就定义了一个名为 student 的结构体类型，可以像使用 int 一样来使用 student 保存多个不同类型的数据了。

8.2.2 结构体的成员变量

在结构体的定义中，最重要的就是成员变量，结构体就是靠这些变量来保存不同类型数据。结构体中有多少个成员变量，就可以用它保存多少个不同类型的数据。当然，反过来，需要保存多少个不同类型的数据，就得定义多少个成员变量！

也许学生不止有姓名、体重、年龄、性别和入学成绩这几个信息要保存，可能还会有身份证号、家庭成员、爱好等信息要保存。这个时候，就得定义符合自己需要的 student 结构体类型，只要更改成员变量就可以。能够根据自己的需要来定义保存什么信息，是结构体最大的特色！

定义了一种结构体类型，就相当于告诉计算机按照某种格式去保存信息。这个格式就是结构体中的成员变量。就拿之前定义的 student 结构体来说吧，计算机按照如图 8.2 所示的格式来保存数据。在图 8.2 中，student 结构体会告诉计算机按照这样的格式来保存数据：先保存有 10 个字符的字符型数组，然后保存一个 float 类型的数据，再保存一个 int 类型的数据，然后保存一个字符型数据，最后保存一个 int 类型的数据。

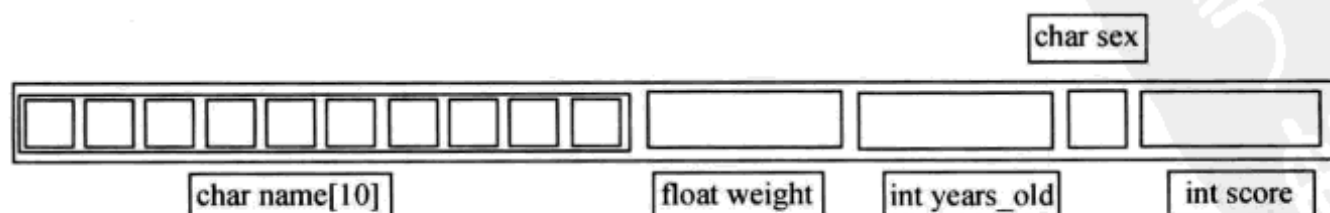


图 8.2 student 结构体类型保存格式

结构体类型较其他类型的优点也就在于它可以规定计算机去按照怎么样的格式来保存数据。想要保存学生有关信息，可以定义一个 `student` 结构体。想要保存电脑有关信息，可以定义一个 `computer` 结构体。想要保存小狗有关信息，可以定义一个 `dog` 结构体。想要保存汽车有关信息，就可以定义一个 `car` 结构体等。

想要保存什么就定义什么样的结构体，结构体中要保存什么样的具体信息要靠结构体中的成员变量。例如，要定义一个计算机结构体，用来保存显示器尺寸、CPU 型号、内存大小、显存大小、主板品牌等信息，就可以在 `computer` 这样的结构体中，定义 `monitors_size`、`cpu_type`、`memory_size`、`graph_mem_size`、`mainboard_brand` 这些成员变量。`computer` 结构体的定义可以如下所示：

```
struct computer
{
    int      monitors_size;
    char     cpu_type[20];
    int      memory_size;
    int      graph_mem_size;
    char*    mainboard_brand;
};
```

有了 `computer` 这个结构体类型及其中成员变量的规定，就可以保存计算机的相关信息了。当然，也可以修改其中的成员变量定义来满足你的需要，保存计算机的其他信息。

8.2.3 复杂的结构体

在前面定义的 `student` 和 `computer` 结构体中可以看到，结构体的成员变量可以是数组。例如，`student` 中的 `name[10]`，`computer` 中的 `cpu_type[20]`。另外，还可以看到 `computer` 中还有一个 `mainboard_brand` 的字符指针变量。

其实，结构体中不仅可以有数组和指针这样的高级数据类型的成员变量，甚至，还可以有结构体成员变量。这种类型被称为结构体类型的嵌套，或者说是复杂结构体类型。

例如，要定义一个学生的结构体类型，用来保存学生的姓名、性别、年龄、入学成绩、家庭成员等信息。分析一下，姓名可以用字符数组来保存，性别可以用字符变量来保存，年龄可以用整型变量来保存，入学成绩也可以用整型变量来保存，家庭成员呢？貌似没有一种数据类型可以保存家庭成员这个信息？其实可以用一个家庭成员结构体来表示。

```
struct home_member
{
    char name[10];           //成员名
    char addr[100];         //住址
    int  years_old;         //年龄
    char work[20];          //工作职位
};
```

在这个家庭成员的结构体中可以保存：家庭成员名、住址、年龄和工作职位这些信息。当要定义一个包含家庭成员信息的学生结构体的时候，就可以在其中定义一个家庭成员结构体变量。

```
struct student
{
```

```

char name[10];           //姓名
float weight;            //体重
int years_old;           //年龄
char sex;                //性别
int score;               //入学成绩
struct home_member father; //父亲
struct home_member mother; //母亲
struct home_member brother; //弟弟
};

```

这个结构体是一个复杂的结构体类型，它包含了三个结构体变量作为结构体成员变量。对于结构体变量的定义，后面再讲。大家只要知道，在结构体类型定义的时候，可以用结构体变量作为成员变量。

8.3 结构体变量


定义了结构体类型以后，大家就可以像使用 int、char、float 这些基本数据类型一样来定义结构体变量了。

8.3.1 结构体变量的声明定义

结构体变量的声明定义的含义及表示形式，与基本数据类型变量的声明定义是一样的。结构体变量的含义或者说作用就是保存一堆有组织结构的数据。表示形式如下所示：

```
结构体类型名 结构体变量名;
```

在这个表示形式中，“结构体类型名”就是定义的结构体类型的名字，如“struct student”、“struct computer”等。需要注意的是这里的“结构体类型名”要加“struct”关键字。“结构体变量名”就是给定义的结构体变量取的名字，只要遵循 C 语言的标识符命名规范就可以。

 **注意：**“结构体类型名”按照严格的要求是要加“struct”关键字的。但是，这与编译器有关，有的编译器要求必须加上，有的编译器可以省略。建议写代码的时候加上，加上是不会有错的。

例如，要定义两个 student 类型的结构体变量，名字为 boy、girl。就可以使用下面的 C 语言表示：

```

struct student
{
    char name[10];           //姓名
    float weight;            //体重
    int years_old;           //年龄
    char sex;                //性别
    int score;               //入学成绩
};

```



```
struct student boy, girl;
```

如果嫌上面的定义麻烦，还可以在定义结构体类型的时候，顺便定义结构体变量，形式如下所示：

```
struct student
{
    char name[10];           //姓名
    float weight;           //体重
    int years_old;          //年龄
    char sex;               //性别
    int score;              //入学成绩
} boy, girl;
```

是不是比上面的形式稍微简化了一点？不过还是建议使用上面的比较麻烦的形式，因为看着比较清楚直观，类型是类型，变量是变量，眉目分明。

8.3.2 结构体变量初始化

有了结构体变量，接下来就可以使用它来保存一堆有结构的不同类型的数据了。那么到底怎么样保存呢？方法有两个：初始化和成员赋值。

先看第一个，结构体变量的初始化，和其他类型变量的初始化是一样的，就是在变量定义的时候向变量中保存初始值。结构体变量的初始化形式如下，有点类似于数组变量的初始化：

```
结构体类型名 结构体变量名 =
{
    成员变量 1 的值,
    成员变量 2 的值,
    .....
    成员变量 n-1 的值,
    成员变量 n 的值
};
```

初始化的形式，就是在结构体变量声明定义的时候，使用等号“=”和大括号“{}”向结构体成员变量中保存需要保存的值。大括号中的值的顺序，要和结构体中定义的成员变量的顺序一样，否则，就会赋错值。每个值之间使用逗号隔开，最后一个值之后没有逗号。另外，等号、大括号、各个值、分号可以写在一行，也可以分开在不同的行写，这一点没有严格的要求，只要你看舒服就好。

如果是给之前定义的 `student` 结构体变量 `boy` 赋值，就可以使用下面的 C 语言表示。

```
struct student
{
    char name[10];          //姓名
    float weight;           //体重
    int years_old;          //年龄
    char sex;               //性别
    int score;              //入学成绩
};

struct student boy =
```

```
{
    "xiaoming",
    120.0,
    18,
    'M',
    90
};
```

8.3.3 取结构体成员运算

向结构体变量中保存数据的另一个方法就是结构体成员赋值。介绍结构体成员赋值之前，先来看看取结构体成员运算。只有从结构体变量中取出结构体成员变量以后，才可以给相应的结构体成员赋值。

取结构成员运算是一个二元运算，它的 C 语言表示形式如下所示：

结构体变量. 结构体成员变量；

在这个运算表达式中，有三部分：“结构体变量”、“.”、“结构体成员变量”。“结构体变量”和“结构体成员变量”，大家已经很清楚了吧。就像上面的 boy 就是一个结构体变量，name 就是一个成员变量。“.”是取结构体成员运算符，得注意一下，以前没见过！

取结构体成员变量运算得到的是结构体变量中的成员变量，是一个变量。有了这个变量就可以用它来保存我们想要保存的数据了。这有点像大盒子中有不同大小的小盒子。声明定义一个结构体变量，相当于问计算机要了一个包含小盒子的大盒子。要想往这样的一个小盒子中放东西，必须首先打开大盒子，然后打开小盒子，最后才能把东西放到小盒子里。取结构体成员变量运算就相当于打开大盒子中的小盒子。

打开了小盒子，就可以往小盒子中放东西了。可以使用一般的赋值运算往小盒子中放东西，其形式如下：

结构体变量. 结构体成员变量 = 值；

把数据保存到了结构体变量中以后，接下来的事情就是在要使用的时候把它拿出来，做我们想要做的事。那么该怎么拿出来呢？还是使用取结构体成员运算，然后从成员变量中取出对应的数据，这和使用一般的变量是一样的。

还是看看下面这段程序吧，它综合了上面的从结构体类型定义、结构体变量定义、结构体变量赋值，到使用结构体变量中的数据。

```
#include<stdio.h>

int main()
{
    //结构体类型定义
    struct student
    {
        char    name[10];        //姓名
        float    weight;        //体重
        int    years_old;        //年龄
        char    sex;            //性别
    }
```

```

    int    score;           //入学成绩
};

//结构体变量定义, 并进行初始化
struct student boy =
{
    "xiaoming",
    120.0,
    18,
    'M',
    90
};

printf("name      : %s\n",boy.name);
printf("weight    : %f\n",boy.weight);
printf("years_old : %d\n",boy.years_old);
printf("sex       : %c\n",boy.sex);
printf("score     : %d\n\n",boy.score);

//成员赋值
boy.name[0] = 'h';
boy.name[1] = 'u';
boy.name[2] = 'a';
boy.name[3] = ' ';
boy.name[4] = 'h';
boy.name[5] = 'u';
boy.name[6] = 'a';
boy.name[7] = '\0';
boy.weight = 110.0;
boy.years_old = 16;
boy.sex = 'M';
boy.score = 100;

printf("name      : %s\n",boy.name);
printf("weight    : %f\n",boy.weight);
printf("years_old : %d\n",boy.years_old);
printf("sex       : %c\n",boy.sex);
printf("score     : %d\n",boy.score);

return 0;
}

```

在这段程序中, 先定义了一个 `student` 结构体类型, 然后定义了一个 `student` 结构体的变量 `boy` 并对 `boy` 进行了初始化。接着, 使用取结构体成员变量运算得到 `boy` 中保存的数据, 并使用 `printf()` 函数进行输出。然后, 使用给结构体成员变量赋值的方法向 `boy` 变量中又保存了另外一些数据。最后, 使用取结构体成员变量运算得到 `boy` 中保存的数据, 并使用 `printf()` 函数进行输出。

由于 `student` 结构体类型的第一个成员变量是一个字符数组类型, 给字符数组赋值的方法有两个。第一个是初始化的时候使用大括号或者字符串, 这里不是初始化, 所以这个方法不可取。第二个方法就是使用取数组元素运算, 挨个给数组中的元素赋值, 就像上面那样!

程序的输出如图 8.3 所示。从中也可以看出, 程序正确输出, 初始化和结构体成员赋

值确实可以用来向结构体变量中保存数据。

```
printf("sex      : %c\n",boy.sex);
printf("score    : %d\n\n",boy.score);

//成员赋值
boy.name[0] = 'h';
boy.name[1] = 'u';
boy.name[2] = 'a';
boy.name[3] = ' ';
boy.name[4] = 'h';
boy.name[5] = 'u';
boy.name[6] = 'a';
boy.name[7] = '\0';
boy.weight = 110.0;
boy.years_old = 16;
boy.sex = 'M';
boy.score = 100;
```

```
C:\WINDOWS\system32\cmd
name      : xiaoming
weight    : 120.000000
years_old : 18
sex       : M
score     : 90

name      : hua hua
weight    : 110.000000
years_old : 16
sex       : M
score     : 100
请按任意键继续...
```

图 8.3 结构体变量

8.4 结构体数组

结构体是一种用来保存多个不同类型数据的高级数据类型，数组是用来保存多个同一种数据的高级数据类型。那么，能不能把它们揉和起来，使用数组来保存多个同样类型的结构体数据呢？在 C 语言里，这是可以的，这样的数据类型被称为结构体数组。

C 语言中，结构体数组的定义形式和一般的数组的定义形式是一样的，也有一维数组、二维数组和多维数组。这里只讨论一维结构体数组，对于其他维的结构体数组，大家可以参考数组一章对比学习。一维结构体数组的定义形式如下所示：

结构体类型名 数组名[表达式];

这个表达式中，“结构体类型名”就是定义的结构体类型的名字，如“struct student”、“struct computer”等。“数组名”就是定义的数组的名字，只要遵循 C 语言标识符命名规范就可以。“表达式”是一个常量或者常量表达式，表示的是这个数组中可以保存多少个结构体数据。

例如，定义了一个结构体 struct student，其成员变量如图 8.4 的上部分所示，有 name、weight、years_old、sex 和 score 几个成员变量。如果再定义一个 struct student 的数组，其保存的样子就像图 8.4 下部分的大矩形框所示，可以保存很多 struct student 类型的结构体数据，具体是几个就得看你定义的数组中的“表达式”是多少了。

知道了结构体数组是如何定义的，接下来就可以往结构体数组中保存数据，或者从结构体数组中取出保存的数据了。要取出结构体数组中的数据，涉及两个运算：取数组元素运算和取结构体成员运算，这两个运算在数组一章和本章前面讲述过了。

看一个实际中使用的结构体数组的例子。例如，一个班有 50 个学生，在开学的时候，要登记每个学生的信息，如姓名、体重、年龄、性别和入学成绩。要保存一个学生的信息，可以使用 student 结构体，但是要保存 50 个学生的信息，该怎么办呢？声明定义 50 个 student 的变量，当然可以，只是太麻烦了，可以定义一个 student 结构体数组。

下面的程序就可以保存 50 个学生的入学信息。在保存的时候使用 scanf() 函数来给结

构体成员变量赋值，使用 printf()函数输出每个学生的信息。

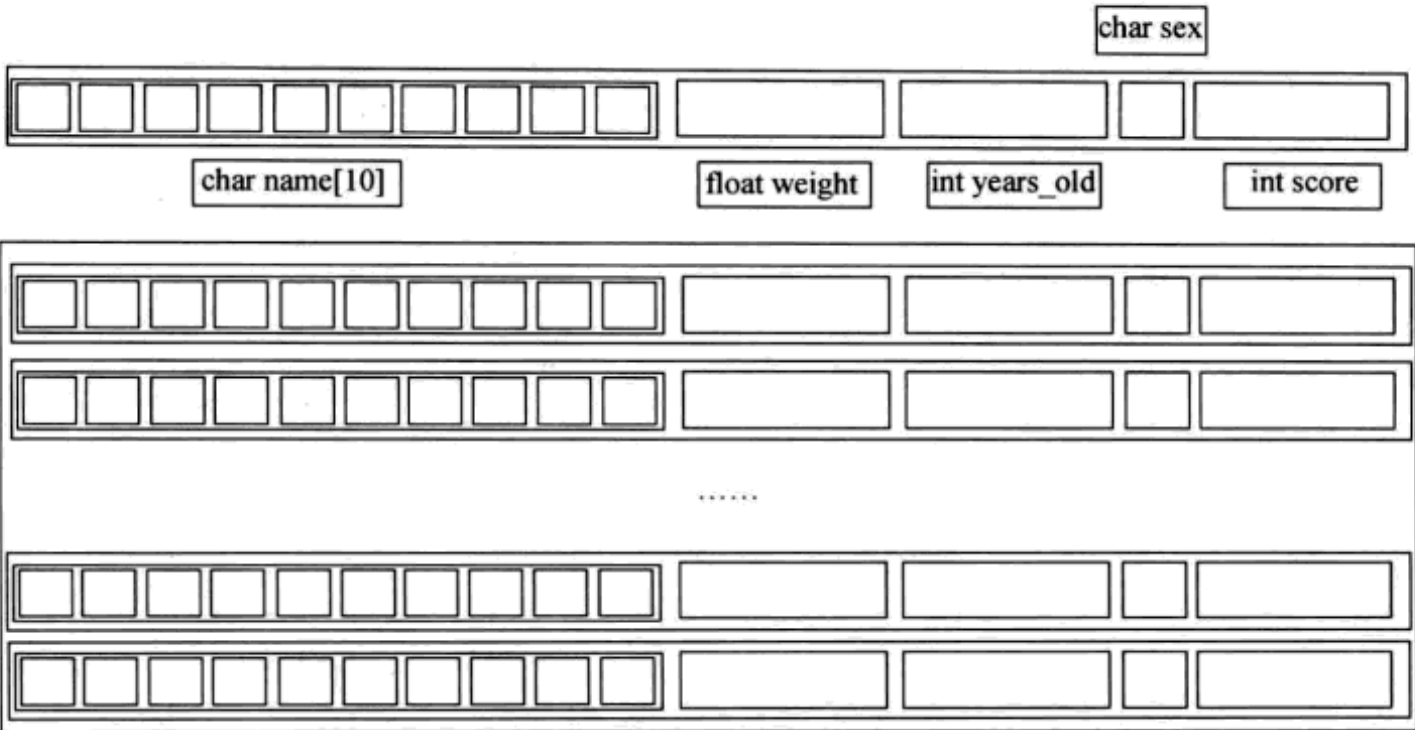


图 8.4 结构体数组

```
#include<stdio.h>

int main()
{
    //结构体类型定义
    struct student
    {
        char    name[10];        //姓名
        float    weight;        //体重
        int    years_old;        //年龄
        char    sex[5];        //性别
        int    score;        //入学成绩
    };

    struct student    students[50];
    int    i;

    for(i=0; i<2; i++)
    {
        printf("name    = ");
        scanf("%s",students[i].name);
        printf("weight    = ");
        scanf("%f",&students[i].weight);
        printf("years_old = ");
        scanf("%d",&students[i].years_old);
        printf("sex    = ");
        scanf("%s",students[i].sex);
        printf("score    = ");
        scanf("%d",&students[i].score);
    }

    printf("\n\nstudet information:\n");
    for(i=0; i<2; i++)
    {
        printf("name:    %s\n",students[i].name);
```

```

printf("weight:  %f\n",students[i].weight);
printf("years_old:%d\n",students[i].years_old);
printf("sex:      %s\n",students[i].sex);
printf("score:    %d\n",students[i].score);
}

return 0;
}

```

在这个程序中，先定义了一个 struct student 的结构类型，然后使用这个结构体类型定义了一个结构体数组 students，它可以保存 50 个 student 结构体变量。接着，使用 for 循环和 scanf() 函数给数组中的结构成员变量赋值，注意，为了避免太多的输入，这里只输入了两个同学的信息。如果要输入 50 个学生的信息，将程序中 for 循环的 2 改为 50 就可以了。最后，使用 for 循环和 printf() 函数输出了两个学生的信息。如果保存了 50 个学生的信息，并且要将其全部输出，将第二个 for 循环中的 2 改为 50 就可以了。

程序的输出如图 8.5 所示。前面一部分，等号和之前的单词是为了提示我们要输出什么而进行的输出，等号之后是使用 scanf() 进行赋值的时候输入的信息。后面一部分，就是程序输出的两个学生的信息了，可见结构体数组确实可以保存多个结构体数据。

```

//结构体类型定义
struct student
{
    char    name[10];    //姓名
    float   weight;      //体重
    int     years_old;   //年龄
    char    sex[5];      //性别
    int     score;       //入学成绩
};

struct student students[50];
int i;

for(i=0; i<2; i++)
{
    printf("name      = ");
    scanf("%s",students[i].name);
    printf("weight    = ");
    scanf("%f",&students[i].weight);
    printf("years_old = ");
    scanf("%d",&students[i].years_old);
    printf("sex       = ");
    scanf("%s",students[i].sex);
    printf("score     = ");
    scanf("%d",&students[i].score);
}

```

```

C:\WINDOWS\system32\cmd.exe
name      = xiaoming
weight    = 120.0
years_old = 18
sex       = male
score     = 90
name      = hauhua
weight    = 100.0
years_old = 18
sex       = female
score     = 90

studet information:
name:      xiaoming
weight:    120.000000
years_old: 18
sex:       male
score:     90
name:      hauhua
weight:    100.000000
years_old: 18
sex:       female
score:     90
请按任意键继续. . .
搜狗拼音 半:

```

图 8.5 结构体数组程序

8.5 结构体指针

指针用来表示数据在内存中的位置，也就是地址。结构数据也是数据，它在内存中的位置也可以用指针变量来保存。本节就来看一看 C 语言中的结构体指针。

8.5.1 一重结构体指针

结构体指针变量的声明定义和一般指针变量的声明定义是一样的，也存在一重结构体

指针和多重结构体指针。在这里只讲述一下一重结构体指针，对于多重结构体指针，大家可以参见“指针”一章类比着学习。一重结构体指针的 C 语言表示形式如下所示：

结构体类型 * 结构体指针名；

在这个表示形式中，“结构体类型”就是已经定义的结构体类型的名字，如 `struct student` 等。“*”是指针定义的表示符号。“结构体指针名”就是定义的结构体指针变量的名字，只要遵循 C 语言标识符命名规范就可以了。

例如，要定义一个 `struct student` 结构体类型的指针，可以使用下面的 C 语言表示形式：

```
struct student * p_student;
```

有了结构体指针变量，就可以用它来保存结构数据的地址了，要得到一个结构体变量中数据的内存地址，可以使用取址运算。例如，要将 `struct student` 结构体类型变量 `boy` 中数据的地址保存到 `p_student` 结构体指针变量中，就可以使用下面的程序。

```
struct student boy;
struct student * p_student;

p_student = &boy;
```

这和将一般变量中数据的地址保存到指针变量中没有多大的差别。

8.5.2 使用结构体指针取结构体数据

一般取指针所指地址中保存的数据的方法就是使用取指针元素运算，也可以使用这个方法取出结构指针所指结构体数据。但是，光取出结构体数据还不够，还需要取出结构体中成员变量的值，这才是我们所需要的。例如，如果要取出 `p_student` 所指结构体数据中结构体成员 `name` 的值，就可使用下面的表示方法：

```
(*p_student) . name
```

这其实是取指针元素运算和取结构体成员运算的结合，`*p_student` 是将 `p_student` 结构体指针所指的结构体数据取出来，`.name` 是将结构体中成员变量取出来，结合这两种运算就可以取出结构体指针所指结构体数据中的成员变量了。由于取结构体成员运算的优先级高于取指针元素运算的优先级，所以得在取指针元素运算上加个小括号，改变其优先级，让它先进行运算。

是不是觉得这样的取结构体数据成员的方法很麻烦啊？C 语言提供了一个简单地使用指针取得结构体成员变量的运算，其形式如下：

结构体指针->结构体成员变量

这个表示形式中，“结构体指针”就是已经定义的结构体指针变量，像上面的 `p_student`。“->”是取结构体指针所指结构体成员变量运算符。“结构体成员变量”就是指指针所指结构体类型中定义的结构体成员变量。例如，要取 `p_student` 所指结构体中成员变量 `name` 的值，就可以使用下面的表示形式：

```
p_student ->name
```


这个表示形式和

```
(*p_student) . name
```

所达到的效果是一样的，只不过就是将取指针元素运算符‘*’和取结构体成员运算符‘.’换成了取结构体指针所指结构体成员变量运算符‘->’了，这也是结构体指针和其他一般指针的最大的区别。

8.5.3 结构体指针例子

下面的程序就是结构体指针在 C 语言中的使用，包括定义、赋值及使用结构体指针，取出结构体中元素的方法。

```
#include<stdio.h>

int main()
{
    //结构体类型定义
    struct student
    {
        char    name[10];        //姓名
        float    weight;        //体重
        int    years_old;        //年龄
        char    sex[5];        //性别
        int    score;        //入学成绩
    };

    //结构体变量定义，并进行初始化
    struct student boy =
    {
        "xiaoming",
        120.0,
        18,
        'M',
        90
    };

    //定义结构体指针变量
    struct student *p_student;

    p_student = &boy;
    printf("name : %s\n", (*p_student).name);
    printf("name : %s\n", p_student->name);

    return 0;
}
```

在这个程序中，先定义了一个结构体类型 `student`，然后定义了一个 `student` 结构体变量 `boy`，并对其进行初始化。接着，定义了一个 `student` 结构体指针 `p_student`，并使用取变量地址运算将 `boy` 的内存地址赋值给 `p_student`。最后，使用两种取结构体指针所指结构体数据的成员变量的方法取出 `boy` 中 `name` 的值，并使用 `printf()` 函数将其输出。

程序的输出如图 8.6 所示，从输出的结果可以看出，两种取结构体指针所指结构体数

据的成员变量的方法效果确实是一样的。

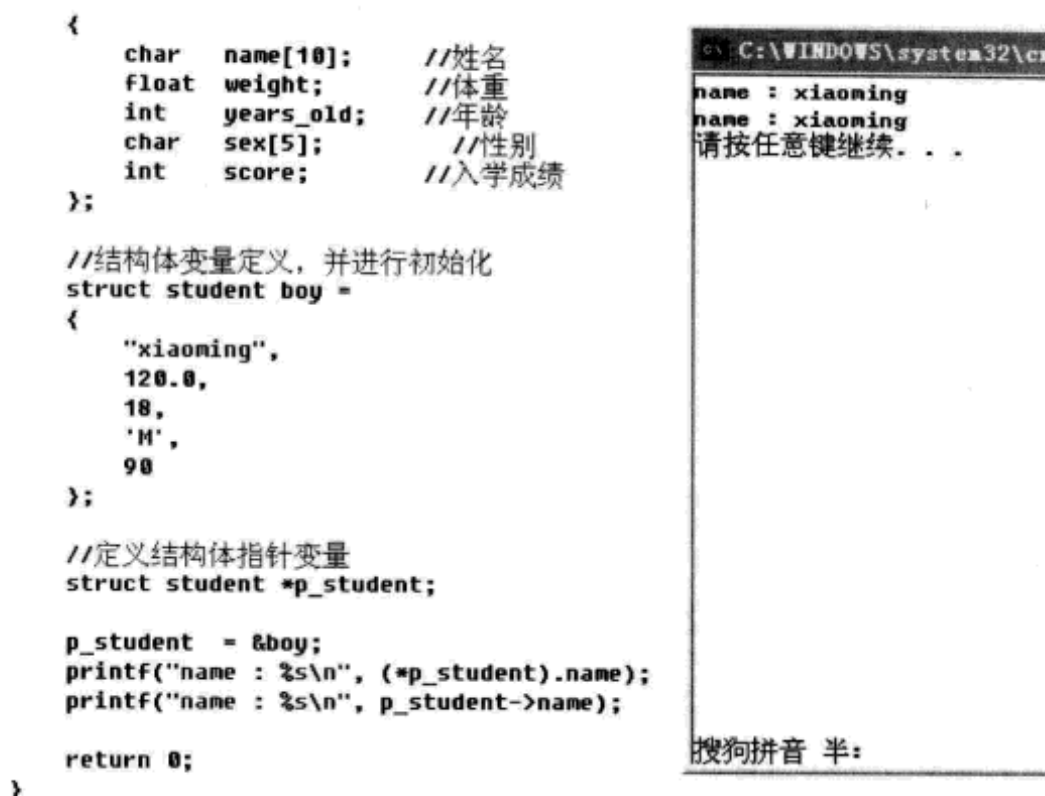


图 8.6 结构体指针

8.6 回到问题

在本章开始的时候, 我们说结构体可以解决: “数据量过大导致记忆混乱” 和 “命名重复” 这两个问题。本章内容到现在已经基本讲完了, 不知道大家体会到了结构体是如何解决这两个问题的没? 如果没体会到, 现在可以跟着我来一起想一想!

结构体之所以能够解决“数据量过大导致记忆混乱”, 是因为每个结构体都有一个名字。正因为这个名字, 结构体中的成员变量就相当于被划分进了一个小的组织, 记忆的时候只需要记住或者知道组织的名字就可以, 至于组织中成员的名字, 没有用到的时候不必操心, 这样就可以避免记忆混乱的现象。

对于结构体能够解决重名问题, 是因为每一个结构体定义都是独立的, 互不干扰。这就像不同的组织一样, A 组织中有个张三, B 组织中也有一个张三, 找张三的时候是不会有问题的。因为我们可以先找组织, 然后再找张三。如果没有组织, 或者说没有结构体, 找张三的时候就会有问题, 因为名字是重复的, 无法区分。

看了结构体这两个用途, 是不是觉得结构体威力很大啊? 其实, 这只是结构体比较直观的作用而已, 如果深究还会发现结构体的很多妙处, 不妨试试!

8.7 小 结

在本章中, 介绍了第一种自定义高级数据类型——结构体类型。本章的重点是理解结构体类型的含义和使用方法, 难点是结构体类型和数组及指针的结合使用。下一章将介绍

第二种自定义高级数据类型——共同体类型。

8.8 习 题

【题目 1】 定义一个结构体类型，使其中出现名字相同的成员变量。

【分析】 按照 C 语言的规定，结构体类型中是不可以出现名字相同的成员变量的。那么，这个题目是不是就无解了？其实，可以通过结构体嵌套来让结构体中出现两个名字一样的结构体成员变量，这样就符合 C 语言要求了！

【核心代码】

```
struct type_out
{
    int a;
    ...
    struct type_in
    {
        int a;
    }b;
    ...
};
```

【题目 2】 写一段程序给题目 1 中的两个成员变量 a 分别赋值为 1 和 2。假设，已经定义了一个 struct type_out 的结构体变量 elem。

【分析】 要想给结构体的成员变量赋值，先得使用取结构体成员变量运算，取得结构体的成员变量，然后再使用赋值运算给其赋值。为了得到 struct type_out 中的第一个 a 成员变量，直接使用取结构体成员变量运算就可以了。为了得到 struct type_out 中的第二个 a 成员变量，必须先得到 b 成员变量，然后才可以得到 a 成员变量。

【核心代码】

```
struct type_out elem;

type_elem.a = 1;
type_elem.b.a = 2;
```

【题目 3】 我们知道结构体中可以包含结构体类型的成员变量，从而组成嵌套结构体类型。有一个问题，一个结构体类型中能不能包含本结构体类型的成员变量呢？写一段程序试一下！

【分析】 结构体成员变量中包含本结构体类型的成员变量，这个成员变量中，又会包含同样类型的一个成员变量，如此往复，将会无限地包含下去。这就好比一个盒子是用来装同样大小的盒子的，你能想象这样一个盒子是什么样的吗？先不管它装不装得下，你能想象一下它到底有几个盒子吗？打开一个，又是一个，打开一个，又是一个，如此往复，有无穷多个！所以，C 语言不允许出现这样的结构体定义，因为编译器和我们一样不知道它到底有几个嵌套。

【核心代码】

```
struct type_name
{
    struct type_name elem;
```

```
...
};
```

【题目 4】 对于题目 3，有的时候确实是不可避免的，如果确实需要在结构体中定义一个本类型的成员变量该怎么办？

【分析】 C 语言中不允许在结构体定义的时候，成员变量中出现本结构体类型的成员变量。但是，可以出现本结构体类型的指针变量，我们可以使用这个指针变量来保存本类型数据所在的地址，这样就可以解决题目 3 所遇到的尴尬！

【核心代码】

```
struct type_name
{
    struct type_name* p_elem;
    ...
};
```

【题目 5】 打鱼还是晒网：中国有句俗语叫“三天打鱼两天晒网”，从 2000 年 1 月 1 号开始起，某人“三天打鱼两天晒网”，问这个人在以后的某天是在“打鱼”还是在“晒网”。

【分析】 要想计算出这个人在 2000 年 1 月 1 日之后的某一天是在“打鱼”还是在“晒网”，必须先算出，这一天距离 2000 年 1 月 1 日几天，再用这个天数对 5 求余，如果结果是 1、2 或 3，则这个人在“打鱼”，否则，在“晒网”。在写代码的时候，可以使用结构体表示年月日，然后计算出从 2000 年 1 月 1 日到现在有多少天，就解决了大部分的问题了。

【核心代码】

```
struct date
{
    int year;
    int month;
    int day;
};

int main()
{
    struct date today ;
    struct date temp ;
    int yeardays , year, day;
    int i, lp;
    //表示平年和闰年的每个月的天数
    int day_tab[12][13] = { {0,31,28,31,30,31,30,31,31,30,31,30,31},{0,31,29,31,30,31,30,31,31,30,31,30,31}};

    int yearday;

    printf("Enter year/month/day:");
    scanf("%d%d%d", &today.year, &today.month, &today.day);

    temp.month =12;
    temp.day =31;
    yearday = 0;

    //计算 2000 年 1 月 1 日到这一天总共有多少天，保存在 yearday 中
    for(yeardays = 0 , year =2000; year < today.year; year++)
    {
        temp.year =year;
```

```
lp = temp.year%4 == 0 && temp.year%100 != 0 || temp.year%400 == 0;
for(i= 1; i <temp.month; i++)
{
    yearday += day_tab[lp][i];
}

//计算这一年的这一天距 2000 年 1 月 1 日有多少天
for(i= 1; i <today.month; i++)
{
    yearday += day_tab[lp][i];
}
yearday += today.day;

//判断打鱼还是晒网
day = yeardays %5;
if(day>0 && day <4) printf("fishing!\n");
else printf("no fishing!\n");

return 0;
}
```


第 9 章 共同体类型

结构体的功能已经够强大了，可以一次保存多个不同类型的数据。可是尺有所短，寸有所长，结构体也是有弱点的。如果一次只保存一种数据，但是每次保存数据的类型都不一样，用结构体就有点浪费了，因为结构体会把所有要保存的数据都定义一个成员变量。C 语言提供了另外一种高级的自定义类型——共同体，它可以解决这个问题。

9.1 共同体的含义与表示

共同体是类似于结构体的一种高级的自定义数据类型，它可以弥补结构体每次保存不同种数据太麻烦的不足。

9.1.1 共同体的用途

我们知道，结构体有点像旅行箱或者橱柜，里面有各种不一样的格子，每次可以保存多个不同种类的数据。与结构体相比，共同体一次只能保存一种类型的数据，但是每次保存数据的类型可以不一样。就像一个小抽屉一样，可以放任何东西，但是却只能放得下一种类型的东西。

图 9.1 表示了结构体和共同体在用途上的差异，所谓结构体，就是将所有的体（数据）结合在一起组成一个结构体，就像橱柜将所有大小不一的柜子和格子组合在一起。而所谓的共同体，就是将所有的体（数据）共同存在一起，就像小抽屉一样，本身是可以保存任何种类的东西的，但是每次只能保存一种类型的东西，放了书就放不了布娃娃，放了布娃娃就放不了书。

好了，说了那么多，大家只要知道本节讲的共同体，是一种高级的自定义数据类型，这种数据类型一次只能保存一种类型的数据，但是，每次保存的数据类型可以不同，有点像小抽屉一样。

9.1.2 共同体的表示

说共同体是一种自定义的高级数据类型，是因为它类似于结构体，保存怎样的数据需要使用者自己来定义。在 C 语言中，共同体类型的定义形式如下所示：

```
union 共同体类型名
{
    数据类型 变量 1;
```

```

数据类型 变量 2;
.....
数据类型 变量 n-1;
数据类型 变量 n;
};

```

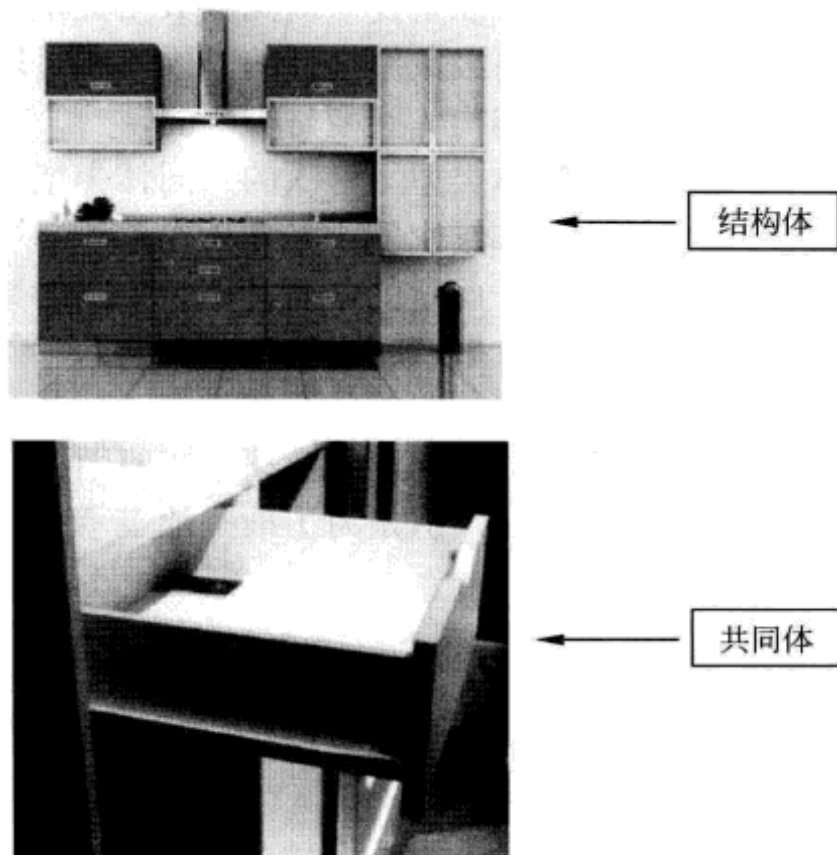


图 9.1 结构体和共同体

共同体类型的定义和结构体类型的定义很类似，只要将结构体类型定义中的 `struct` 改为 `union` 就可以了，其他部分的含义和结构体一样。“`union`”是共同体类型定义的关键字，这是 C 语言规定的，不能更改，否则就会出错。一对大括号中包含了很多的变量定义，这些变量被称为成员变量，就是靠它们来保存数据的。大括号之后必须以“`;`”结束，这也是 C 语言的规定，否则就出错了。如果你非要问为什么要以分号结束，那是因为 C 语言中的语句都是以“`;`”结尾的，类型定义也可以看做是一个语句。

有人会说，这个定义和结构体没有什么区别啊？确实没有什么区别，唯一的区别就是关键字“`union`”，也正是这个关键字，使 C 语言的编译器按照共同体的作用和含义来组织其中的成员变量，使其能达到共存一体的效果。具体怎么共存一体，后边讲结构体变量的时候将会深入讲解，现在大家只要知道如何定义一个共同体类型就可以了！

例如，要用一种数据类型来表示圆的半径，但是事先并不能确定圆的半径到底是整数还是小数。这个时候就可以定义一个共同体类型 `radius`。它既可以保存整型数据，也可以保存浮点型数据，但是每次只能选一种来保存。`radius` 共同体的类型定义如下所示：

```

union radius
{
    int    i_radius;
    float  f_radius;
};

```

这样的—个共同类型，既可以保存整数，又可以保存小数。

9.1.3 复杂的共同体

从共同体的类型定义可以看出，其中的成员变量可以是任何类型的，既可以是简单的基本数据类型，如 `int`、`float`、`char`；也可以是复杂的高级数据类型，如数组，指针，结构体，甚至共同体。共同体中的变量包含高级数据类型的时候，就可以认为这个共同体类型是一个复杂的共同体类型。这和复杂的结构体类型是类似的。

例如，还是要定义一个 `radius` 共同体类型，这次的共同体类型有点复杂。我们知道半径在一维空间中是由其长度确定的，在二维空间中是由一个二维点 (x, y) 确定的，在一个三维空间中是由一个三维点 (x, y, z) 确定的。而且每一维上的数据既可以是整数也可以是小数。要表示这样一个复杂的半径数据，就需要一个复杂的共同体类型了。

我们可以先分析分析，如何定义这个复杂的共同体类型。

首先，最需要表示的就是坐标系上的每一维的数据。按照上面的描述，可以是整数也可以是小数，可以只用浮点型类型来表示，但是为了准确地表示上面的描述，还是使用下面的共同体来表示吧：

```
union data_dim
{
    int    i_data;
    float  f_data;
};
```

其次，要表示的是一维空间、二维空间及三维空间上的半径信息。一维空间上的半径信息可以直接用每一维上数据的类型（共同体 `union data_dim`）来表示，二维空间上的数据就需要定义一个包含 `x`、`y` 两个成员变量的结构体了，而 `x`、`y` 本身又使用每一维上数据的类型（共同体 `union data_dim`）来表示：

```
struct radius_2d
{
    union data_dim x;
    union data_dim y;
};
```

要表示三维空间上的半径信息，也需要一个类似于 `struct radius_2d` 的结构体，三维半径信息结构体只是比二维半径结构体多了一个维的数据 `z` 而已，形式如下：

```
struct radius_3d
{
    union data_dim x;
    union data_dim y;
    union data_dim z;
};
```

最后，要定义一个可以表示一维半径信息、二维半径信息和三维半径信息的共同体。有了前面的定义，这个共同体就好定义多了。

```
union radius_info
{
    union data_dim    data_1d;
    struct radius_2d  data_2d;
    struct radius_3d  data_3d;
};
```

在这个复杂共同体 `radius_info` 的定义中，使用了共同体变量的定义，这将在后面的部分进行详细讲解。大家只要知道结构体中可以嵌套共同体，共同体中也可以嵌套结构体，从而组成更为复杂的自定义数据类型就可以了。

9.2 共同体变量

知道了如何定义一个共同体类型以后，就可以使用共同体类型来定义共同体变量了！顺便来看看，C 语言中的共同体是如何实现可以保存不同种类的数据，但是每次只能保存一种的。

9.2.1 共同体变量

在 C 语言中，要想定义一个共同体变量是很简单的，把自己定义的共同类型当作一种已经知道的数据类型就可以了，就像 `int`、`char`、`float` 等。然后，按照定义一般数据类型变量的形式来定义一个共同体变量。具体形式如下所示：

共同体类型名 共同体变量；

在这个共同体变量定义的表示中，“共同体类型名”就是已经定义的共同体类型，就像前面见到的 `union radius` 就是一个共同体类型名。“共同体变量”就是要定义的共同体变量的名字，只要遵循 C 语言标识符命名规范就可以。特别强调一点，在“共同体类型名”中的 `union` 关键字最好不要少，不然就会出错，这和结构变量定义中的要求是一样的。

例如，现在要定义一个 `union radius` 共同体的变量 `r`，可以使用下面的 C 语言表示形式。

```
union radius r;
```

假设在定义 `r` 之前，已经定义了 `union radius` 这个共同体类型。它包含了一个整型成员变量 `i_radius` 和浮点型成员变量 `f_radius`。那么，既可以用共同体变量 `r` 来保存整型数据，又可以用 `r` 来保存浮点型数据。

当然，也可以使用类似于结构体变量的定义方法，在定义共同体类型的时候，顺便定义共同体变量。例如，在定义 `radius` 共同体类型的时候，也可以定义 `radius` 共同体变量 `r`。就像下面这样：

```
union radius
{
    int    i_radius;
    float  f_radius;
} r;
```

不过，还是推荐不要在定义共同体的时候定义共同体变量，这样看着比较混乱。而且在定义共同体类型的时候，不一定已经确定需要什么样的共同体变量，需要多少个共同体变量。

9.2.2 共同体成员变量的相互覆盖

有人会问，既然共同体和结构体一样，都有好多成员变量，那么，为什么结构体可以

同时保存好几个不同种类的数据，而共同体每次却只能保存一个类型的数据？其他的成员变量跑哪去了？要回答这个问题，就得了解共同体成员变量的相互覆盖关系！

例如，定义了一个 `union radius` 共同体类型，其中包含两个成员变量，整型的 `i_radius` 和浮点型的 `f_radius`。计算机看到定义了这样的一个共同体之后，先看看每个成员需要用多大的空间来保存数据，然后再按照需要最大空间的成员变量来给这个共同体变量分配空间。

这就好比，高速路要修桥洞，保证各种高度的车辆都可以通过。如果按照结构体的方法来修桥洞，就得给每一种车辆修一个桥洞，显然这是不可取的，太麻烦，也太浪费了。如果按照共同体的方法来修桥洞，只要修一个高度比所有类型车辆都高的桥洞就行了，但是所有的车辆必须公用一个桥洞的空间，同时只能有一辆车从桥洞中通过。

计算机中保存共同体也是类似的，所有的成员变量都重叠到一个空间里，计算机确保这个空间可以容纳任何一个成员变量。

回到 `union radius` 的例子中来，在计算机中，整型数据是占 4 个字节，浮点型也占 4 个字节。根据前面讲的共同体分配方法，计算机会给 `union radius` 变量分配 4 个字节的空間，足以容纳得下一个整型数据或者一个浮点型数据。

就像图 9.2 那样，计算机只给 `union radius` 类型的变量 4 个字节的空間，这 4 个字节的空間既可以保存 `i_radius` 整型变量的数据，也可以保存 `f_radius` 浮点型的数据，但是只能选其中的一个。

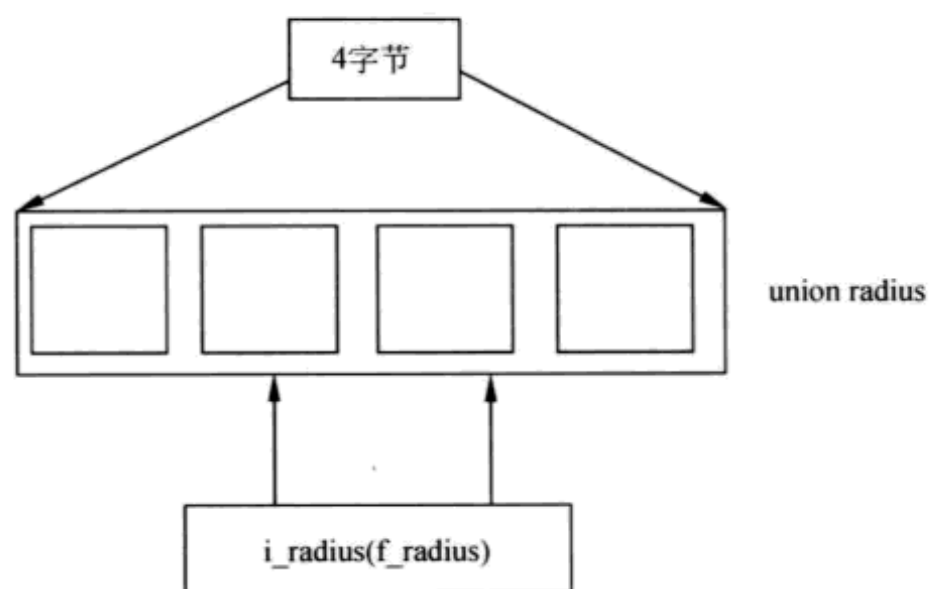


图 9.2 共同体保存格式

9.2.3 使用共同体变量

了解了共同体变量的表示方法，以及计算机是如何处理共同体变量的以后，就可以使用共同体变量来保存数据了，也可以把共同体变量中保存的数据取出来了。

往共同体变量中保存数据，和往结构体变量中保存数据是类似的。这个赋值运算既可以进行初始化，又可以给成员变量赋值。共同体变量初始化必须是在变量定义的时候，而且共同体变量的初始化和结构体有些不同，其形式如下：

```
共同体类型名 共同体变量 = {数值};
```

在共同体初始化的时候，大括号中只能有一个数值，因为共同体只能保存一个数值！例如，要对 `union radius` 变量 `r` 进行初始化，可以使用下面的形式：

```
union radius r = {2};
```

在这个初始化形式中，按照要求，大括号中只有一个整型常数 2，所以不会出现问题。如果不按照规定，给 `r` 初始化两个数值，如下所示：

```
union radius r = {2, 1.0};
```

就会出现图 9.3 所示的编译错误，错误显示共同体初始化值太多。

```
1>----- 已启动生成: 项目: test, 配置: Debug Win32 -----
1> test.c
1>e:\c语言的书\project\test\test\test.c(34): error C2078: 初始值设定项太多
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====
```

图 9.3 共同体初始化太多值

再来看一看如何给共同体变量赋值。这就得用到取共同体成员变量运算了，其形式和结构体变量的取结构体成员变量运算是一样的，如下所示：

共同体变量.成员变量名

这个运算也分为 3 个部分：“共同体变量”、“.”和“成员变量名”。它们的含义和结构体取成员变量运算是一样的。例如，要取 `union radius` 共同体变量 `r` 的 `i_radius` 成员变量，可以用下面的表示方法。

```
r.i_radius;
```

取到共同体的成员变量以后，就可以直接使用一般的赋值运算给共同体变量的成员变量赋值了。要想使用保存到共同体中的数值，还是使用取共同体成员变量运算，取出成员变量就相当于拿到了共同体中相应的数据，接着就可以使用这个值做其他运算了。

下面是一个使用共同体变量的完整的例子，包含了定义、赋值到取值的所有环节。

```
#include <stdio.h>

int main()
{
    union radius
    {
        int    i_radius;
        float  f_radius;
    };

    union radius r = {1};

    printf("i_radius = %d\n", r.i_radius);

    r.i_radius = 3;
    printf("i_radius = %d\n", r.i_radius);

    r.f_radius = 2.1;
    printf("f_radius = %f\n", r.f_radius);
    printf("i_radius = %d\n", r.i_radius);
}
```

```

    return 0;
}

```

在这个程序中，先定义了一个共同体类型：union radius。然后定义了一个 union radius 的变量 r 并对其进行了初始化，往里面保存了一个整型常量 1。接着，使用取共同体成员运算取出了整型成员变量 i_radius，整型值就保存在其中，并使用 printf() 输出了这个整型值。接着，又使用取共同体成员变量运算，对共同体进行赋值，并将其中的值输出。大家注意一下，在给 r 的两个成员变量都赋值以后，使用 printf() 函数输出了 r 的每个成员变量的值。

程序的输出如图 9.4 所示，第一行的输出是对 r 初始化为 1 的结果，第二行的输出是对 r 的 i_radius 成员变量赋值为 3 的结果，第三行的输出是对 r 的 f_radius 成员变量赋值为 2.1 的结果。

```

#include <stdio.h>

int main()
{
    union radius
    {
        int    i_radius;
        float  f_radius;
    };

    union radius r = {1};

    printf("i_radius = %d\n", r.i_radius);

    r.i_radius = 3;
    printf("i_radius = %d\n", r.i_radius);

    r.f_radius = 2.1;
    printf("f_radius = %f\n", r.f_radius);
    printf("i_radius = %d\n", r.i_radius);

    return 0;
}

```

```

C:\Windows\system32\cmd.exe
i_radius = 1
i_radius = 3
f_radius = 2.100000
i_radius = 1074161254
请按任意键继续. . .

```

图 9.4 共同体变量程序

重点解释一下第四行的输出，按照程序的期望是输出 r 的 i_radius 成员变量的值，而第四行却输出了一个很大的整数。这是共同体成员变量相互覆盖的结果。当给 r 的 f_radius 成员变量赋值为 2.1 的时候，覆盖了前面给 r 的 i_radius 赋的值 3。把 2.1 强制按照整型数据进行解析和输出就得到了图 9.4 所示的结果：107 416 154。

9.3 共同体数组

数组是保存多个同一种类型数据的高级数据类型。它不仅可以保存多个基本数据类型的数据，如整型数组、字符数据和浮点型数组，也可以保存多个高级数据类型数据，如结构体数组。当然，数组也可以保存多个共同体数据，即下面将要讲到的共同体数组。

理论上讲，共同体数组也是存在一维共同体数组、二维共同体数组和多维共同体数组的。二维和多维共同体数组在实际中很少用到，如果用到可以参见数组一节类比着学习，性质和用法都是类似的。这里着重讲一下一维共同体数组。

C语言中要定义一个一维共同体数组，可以使用下面的表示形式：

```
共同体类型名 数组名[表达式];
```

在这个表示形式中，“共同体类型名”就是已经定义了的共同体类型名，如前面定义的 `union radius`。“数组名”就是要定义的一维共同体数组名。“表达式”是一个常量或者常量表达式，表示数组的容量，也就是数组中可以保存多少个共同体数据。

例如，要定义一个可以保存 10 个 `union radius` 共同体数据，名为 `r` 的数组，就可以使用下面给出的形式。

```
union radius r[10];
```

有了共同体数组之后，就要想办法来使用它，向它里面保存数组，在需要的时候，把共同体数组中保存的数据取出来使用。

往共同体数组中保存数据的方法有两个：初始化和赋值。初始化就是在定义数组的时候，使用类似于数组初始化的方法，往共同体数组中保存一定的数据。赋值就是使用取数组元素运算，把数组中保存的共同体取出来，然后再使用取共同体成员变量运算取出共同体变量的某个成员变量。最后使用一般的赋值运算往共同体成员变量中保存数据就可以了。下面分别是共同体初始化和赋值的 C 语言表示：

初始化：

```
共同体类型名 数组名[表达式] = {数值, 数值, ……., 数值, 数值};
```

赋值：

```
共同体数组名[下标].成员变量 = 数值;
```

要想把保存在共同体数组中的数据取出来，只有一个办法，那就是先取出共同体数组中某个位置的元素，然后使用取共同体成员变量运算取出某个成员变量，最后就可以利用这个成员变量来得到数组中保存的共同体数据了。具体表示方法如下所示：

```
共同体数组名[下标].成员变量
```

有了上面的知识，看下面的例子。它包含了共同体数组的定义、初始化、赋值及使用保存在共同体数组中的数据。

```
#include <stdio.h>

int main()
{
    union radius
    {
        int i_radius;
        float f_radius;
    };

    union radius r[5] = {1, 2.0, 3, 4.0, 5};
    int i;

    for(i=0; i<5; i++)
    {
        if(i%2 == 0)
            printf("r[%d].i_radius = %d\n", i, r[i].i_radius);
```



```

        else
            printf("r[%d].f_radius = %f\n", i, r[i].f_radius);
    }

    printf("\n");

    for(i=0; i<5; i++)
    {
        if(i%2 == 0)
            r[i].i_radius = i;
        else
            r[i].f_radius = (float)i;
    }

    for(i=0; i<5; i++)
    {
        if(i%2 == 0)
            printf("r[%d].i_radius = %d\n", i, r[i].i_radius);
        else
            printf("r[%d].f_radius = %f\n", i, r[i].f_radius);
    }

    return 0;
}

```

在这个程序中，首先定义了 union radius 共同体。然后定义了一个共同体数组 r，其容量为 5，并给它交错着初始化赋值整数和小数，接着输出数组中的数值。再接着使用赋值的方式给共同体数值中的元素赋值，也是交错着赋值整数和小数，最后将数组中的数值输出。

程序的输出如图 9.5 所示。前 5 行是数组初始化输出的结构体，可以看到整数正常输出了。但是小数，也就是浮点型数值没有正常地输出。这也是大家需要注意的，在对共同体数组进行初始化的时候，由于没有指定到底是对哪个成员变量进行初始化，所以编译器会把所有的数值都按照第一个数值所对应的类型进行处理。因此，所有的小数都按照整数进行了处理，导致结果都为 0。

由此可见，大家在对共同体数组进行初始化的时候，最好不要初始化成不同类型的数据。程序输出的后 5 行是对共同体数组进行赋值的结果，可以看到这个输出是正确的，所以大家如果想往共同体数组中保存不同类型的数值，还是使用赋值的方式吧。

```

};

union radius r[5] = {1, 2.0, 3, 4.0, 5};
int i;

for(i=0; i<5; i++)
{
    if(i%2 == 0)
        printf("r[%d].i_radius = %d\n", i,
    else
        printf("r[%d].f_radius = %f\n", i,
}

printf("\n");

for(i=0; i<5; i++)
{
    if(i%2 == 0)
        r[i].i_radius = i;
    else
        r[i].f_radius = (float)i;
}

```

```

C:\Windows\system32\cmd.exe
r[0].i_radius = 1
r[1].f_radius = 0.000000
r[2].i_radius = 3
r[3].f_radius = 0.000000
r[4].i_radius = 5

r[0].i_radius = 0
r[1].f_radius = 1.000000
r[2].i_radius = 2
r[3].f_radius = 3.000000
r[4].i_radius = 4
请按任意键继续. . .

```

图 9.5 共同体数组程序

9.4 共同体的指针

指针是数据在内存中的地址，指针变量是用来保存这一地址的。共同体数据也是数据，它在内存中也是有地址的，也可以使用共同体指针进行表示，同时也可以使用共同体指针变量来保存这一地址。本节就来看看共同体指针。

9.4.1 一重共同体指针类型

和其他类型数据的指针一样，共同体指针也分为一重共同体指针、二重共同体指针和多重共同体指针。这里只介绍一重共同体指针，二重和多重共同体指针很少用到，这里就不再赘述了，具体可以类比地从“指针”一章进行学习。

在C语言中，一重共同体指针类型的表示如下所示：

共同体类型名 *

在这个指针类型的定义表示中，“共同体类型名”就是已经定义好的共同体类型，如 `union radius`。“*”是指针类型的特殊表示符号，是C语言规定的，没有什么可以讨论的。

例如，要定义一个 `union radius` 指针类型就可以使用下面的表示方法：

`union radius *`

9.4.2 共同体指针变量

看完了如何定义一个共同体指针类型，现在就可以使用共同体指针类型来定义指针变量了。其形式就是将指针变量定义的一般形式中的“指针类型”换成共同体指针类型：

共同体指针类型 * 共同体指针变量；

按照这个表示，要定义一个 `union radius` 的指针变量，就太简单了，具体表示如下所示：

`union radius* p_r;`

有了这个共同体指针变量，就可以使用它来保存共同体数据在内存中的地址了。要得到共同体变量在内存中的地址，需要用到取值运算`&`。下面就是给共同体指针变量赋值为共同体变量的地址的一般形式：

共同体指针变量 = `&`共同体变量；

例如，已经定义了一个 `union radius` 共同体变量 `r`，并且定义了一个 `union radius` 共同体指针变量 `p_r`。现在需要把 `r` 的地址保存到变量 `p_r` 中，需要用下面的C语言语句。

`p_r = &r;`

往 `p_r` 共同体指针变量中保存了地址之后，该如何使用 `p_r` 得到 `r` 中保存的共同体数据呢？又该如何使用这个指针往共同体中保存数据呢？类似于结构体指针，可以使用取指

针运算和取共同体成员变量运算的结合来得到共同体中保存的数据，同样也可以往共同体中保存数据。其具体形式如下所示：

`(*共同体指针变量).成员变量`

在这个表示形式中，“(*共同体指针变量)”就是取指针元素运算。它可以得到指针所指地址中的共同体，加括号是因为取成员变量运算的优先级高于取地址运算的优先级。“成员变量”就是取共同体成员变量运算，它得到的就是共同体中的成员变量，有了这个成员变量，就可以往其中保存数值了，也就是所需要的共同体数据。

例如，要取 `p_r` 中保存的共体型的整型数据，就可以使用下面的表示形式：

`(*p_r).i_radius`

在 C 语言中，一次写两种运算，并且还要加括号，太麻烦了。所以，提供了一种使用指针取共同体成员变量的运算，类似于使用结构体指针取结构体成员变量的运算，形式如下：

`共同体指针变量->成员变量`

按照这个运算表示形式，还可以使用下面的表示形式来取 `p_r` 中保存的共体型的整型变量。

`p_r -> i_radius`

9.4.3 完整的例子

前面已经基本讲完了共同体指针的各方面知识，现在来看一个完整的例子，看看共同体指针在程序中到底是如何使用的。

```
#include <stdio.h>

int main()
{
    union radius
    {
        int    i_radius;
        float  f_radius;
    };

    union radius r;
    union radius* p_r;

    r.i_radius = 10;
    p_r = &r;

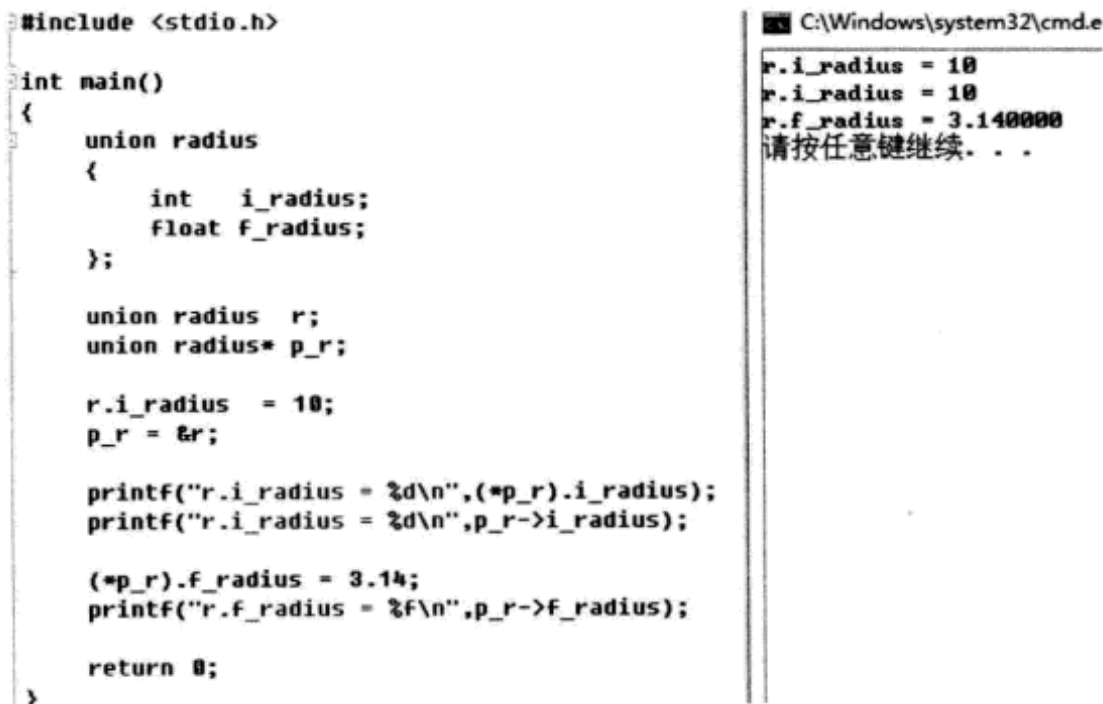
    printf("r.i_radius = %d\n", (*p_r).i_radius);
    printf("r.i_radius = %d\n", p_r->i_radius);

    (*p_r).f_radius = 3.14;
    printf("r.f_radius = %f\n", p_r->f_radius);

    return 0;
}
```

在这个程序中，先定义了一个 union radius 的共同体类型。然后定义了一个 union radius 的变量 r 和 union radius 的指针变量 p_r，并向 r 中保存一个整数 10，向 p_r 中保存 r 的地址。接着，使用两种方式取得了 p_r 所指结构体中的数据，并使用 printf() 函数进行输出。最后，使用一个方式向 p_r 所指共同体 r 中保存了一个小数 3.14，使用另一种方式得到保存的小数，并将其输出。

程序的输出如图 9.6 所示，从输出可以看出共同体指针变量的两种使用方式确实可得到共同体的成员变量，而且它们是等价的。



```
#include <stdio.h>

int main()
{
    union radius
    {
        int    i_radius;
        float  f_radius;
    };

    union radius r;
    union radius* p_r;

    r.i_radius = 10;
    p_r = &r;

    printf("r.i_radius = %d\n", (*p_r).i_radius);
    printf("r.i_radius = %d\n", p_r->i_radius);

    (*p_r).f_radius = 3.14;
    printf("r.f_radius = %f\n", p_r->f_radius);

    return 0;
}
```

```
C:\Windows\system32\cmd.e
r.i_radius = 10
r.i_radius = 10
r.f_radius = 3.140000
请按任意键继续...
```

图 9.6 共同体指针程序

9.5 小 结

本章介绍了 C 语言中第二种自定义高级数据类型——共同体类型。本章的重点是理解共同体类型的含义和使用方法，以及它与结构体的区别，难点是共同体类型和数组及指针的结合使用。下一章将介绍最后一种自定义数据类型——枚举类型。

9.6 习 题

【题目 1】 定义一个结构体类型，其中包含共同体类型的成员变量；定义一个共同体类型，其中包含结构体类型的成员变量。

【分析】 C 语言中的结构体和共同体的成员变量，不仅可以是基本数据类型的，也可以是复杂数据类型的，所以，结构体中有共同体类型的成员变量，共同体中有结构体类型的成员变量都是可以的。

【核心代码】

```
//方法 1：结构体中包含共同体类型的成员变量
struct type_s_name
```



```
{
    union type_u_name
    {
        ...
    } elem;
    ...
};

//方法 2: 结构体中包含共同体类型的成员变量
union type_u_name
{
    ...
};
struct type_s_name
{
    union type_u_name elem;
    ...
};
```

```
//方法 1: 共同体中包含结构体成员变量
union type_u_name
{
    struct type_s_name
    {
        ...
    } elem;
    ...
};
```

```
//方法 2: 共同体中包含结构体成员变量
struct type_s_name
{
    ...
};
union type_u_name
{
    struct type_s_name elem;
    ...
};
```



第 10 章 枚举类型

前面已经学习了结构体和共同体，它们都可以用来表示和存储不同类型的数据，只是表示和存储的方式不同而已。C 语言还提供了另外一种数据类型——枚举类型，它也是一种由用户自定义的数据类型，不过它与结构体和共同体有着较大的区别。本章就来讲一下枚举类型，看看它和结构体、共同体有什么区别。

10.1 枚举类型的含义与表示

大家有没有想过，在实际生活中有这样一类数据，它们只会取到某些特定的值。例如，月份只能是一月、二月、三月、……、十一月、十二月中的一个；星期，只能是星期一、星期二、……、星期六、星期日中的一个。性别，只能是男和女中的一个；诸如此类，数不胜数；这样的数据该怎么表示呢？

10.1.1 枚举类型的含义

要表示上面所述的一类数据，用整型、字符串是可以的，用浮点型、数组、结构体、共同体其实也能行。不过，使用这些类型表示要么显得太麻烦，要么显得太别扭。C 语言提供了一种专门的数据类型来表示这样一类只能取特定值的数据，那就是枚举类型。

枚举，顾名思义，就是列举的意思，将所有需要表示的数据一个一个地列举出来！要想只表示月份，就将月份的 12 个月全部列举出来。要只表示星期几，就将 7 天全部列举出来。要想只表示性别，就将两种性别列举出来。

图 10.1 就展示了星期的枚举，图中的圆圈就相当于要表示的枚举类型，它里面枚举了一个星期的 7 天。当定义了这样的一个枚举类型以后，这种类型就只能表示一个星期的 7 天了。

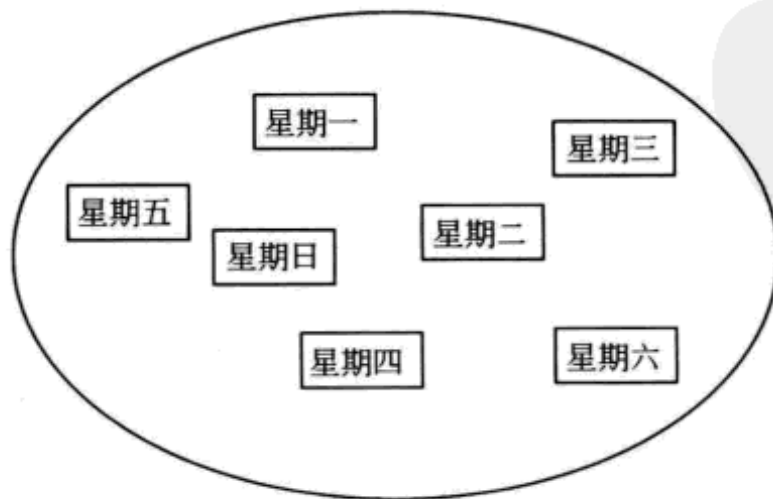


图 10.1 枚举星期

10.1.2 枚举类型的表示

知道了枚举类型是用来表示有限的几个数据的类型以后，接下来看看 C 语言是如何表示枚举类型的，即枚举类型的定义。

枚举类型的 C 语言定义形式如下所示：

```
enum 枚举类型名
{
    枚举值 1 名,
    枚举值 2 名,
    .....
    枚举值 n-1 名,
    枚举值 n 名
};
```

从枚举类型定义的表示形式可以看出，枚举类型类似于结构体和共同体，是一种用户自定义类型。说它是用户自定义类型，是因为这种类型中有什么是用户自己定义的。来看看这个定义表示吧，“enum”是枚举类型关键字，类似于 struct 和 union。大括号中是要枚举的数据的符号表示，都是一个个标识符，由字母、数字和下划线组成，遵循 C 语言标识符要求。大括号和其后的分号是 C 语言规定的，不能少也不能写错。

例如，要定义一个 week 枚举类型，就可以使用上面的格式，定义出下面的形式：

```
enum week
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};
```

C 语言中不限制这些枚举符号列表非得放在不同的行上，也可以把它们全都放在一行上，只要自己觉得好看就行。

```
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

从枚举类型的定义可以看出枚举类型与结构体、共同体的两个不同之处了。

- ❑ 枚举类型没有成员变量。这跟枚举类型的作用有关，它只需要列出能够表示的数据就可以了。从定义中可以看出，枚举类型定义中是使用标识符来列出所表示的数据的。
- ❑ 枚举结构中所列的数据表示是使用逗号隔开的，有点像数组初始化里的数值，而结构体和共同体成员变量是使用分号隔开的。

10.2 枚举常量和枚举变量

除了在枚举类型定义的时候，可以看到枚举类型与结构体、共同体的两点不同，它们

之间还有一个最大的区别，那就是：枚举是一种基本数据类型，类似于整型、浮点型和字符型，而结构体和共同体是一种高级数据类型。

之所以说枚举是一种基本数据类型，是因为它不是由其他数据类型组合而成的，而是自成一体，是一种独立的数据类型。既然枚举是一种基本数据类型，那么，就有枚举常量和枚举变量。枚举常量用来直接表示枚举数据，而枚举变量用来保存枚举常量。

10.2.1 枚举常量

C 语言中的枚举常量，就是在定义枚举类型的时候，出现在大括号中的所有枚举值名。例如，定义 `week` 枚举类型中的 `Friday`，就是一个枚举常量，和整数中的 1 是同一个概念，都是基本数据类型的常量表示。

我们知道，在 C 语言中，赋值运算的左边必须是一个变量，如果给一个常量赋值就会出现错误。可以用这个规则试试 `Friday` 是不是常量。

```
#include <stdio.h>

int main()
{
    enum week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

    Friday = 5;

    return 0;
}
```

例如，在这个程序中，给枚举常量 `Friday` 赋值为 5，在编译的时候，就会出现如图 10.2 所示的错误。错误显示等号的左边必须是一个左值，也就是一个变量，由此可见 `Friday` 确实是一个常量。

```
1>----- 已启动生成: 项目: test, 配置: Debug Win32 -----
1> test.c
1>e:\c语言的书\project\test\test\test.c(7): error C2106: "=": 左操作数必须为左值
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====
```

图 10.2 枚举常量赋值

在计算机中每一个常量都会被计算机按照一定的形式表示和存储。例如，整型常量是按照补码的形式表示和存储的，浮点型常量是按照规格化小数的形式表示和存储的。字符型常量是按照 ASCII 的形式表示和存储的。既然枚举是一种基本数据类型，枚举常量又是怎样被计算机表示和存储的呢？

在回答这个问题之前，先来看看之前定义的 `week` 枚举类型中的常量的值都是多少。可以使用下面的程序：

```
#include <stdio.h>

int main()
{
    enum week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```



```

printf("Monday    = %d\n",Monday);
printf("Tuesday   = %d\n",Tuesday);
printf("Wednesday = %d\n",Wednesday);
printf("Thursday  = %d\n",Thursday);
printf("Friday    = %d\n",Friday);
printf("Saturday  = %d\n",Saturday);
printf("Sunday    = %d\n",Sunday);

return 0;
}

```

在程序中,把所有的枚举常量按照整型数值进行输出,如图 10.3 所示。从程序的输出来看,枚举类型定义中,第一个枚举常量标识符表示的是 0,其余的依次加 1。

```

#include <stdio.h>

int main()
{
    enum week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

    printf("Monday    = %d\n",Monday);
    printf("Tuesday   = %d\n",Tuesday);
    printf("Wednesday = %d\n",Wednesday);
    printf("Thursday  = %d\n",Thursday);
    printf("Friday    = %d\n",Friday);
    printf("Saturday  = %d\n",Saturday);
    printf("Sunday    = %d\n",Sunday);

    return 0;
}

```

C:\Windows\system32\cmd.exe

```

Monday    = 0
Tuesday   = 1
Wednesday = 2
Thursday  = 3
Friday    = 4
Saturday  = 5
Sunday    = 6
请按任意键继续. . .

```

图 10.3 枚举常量的值

其实,枚举常量的值是可以由我们自己指定的,指定的方式如下所示:

```
enum 枚举类型{ 枚举值 1 名=数值, 枚举值 2 名=数值, 枚举值 3 名, ..., 枚举值 n-1 名, 枚举值 n 名};
```

从这个表示形式中可以看出,要想给枚举常量指定数值,就得在枚举类型定义的时候,使用等号给枚举值名指定数值。

可以给任意的枚举常量指定数值,但是没有指定值的枚举常量的值又是多少?先来看一段程序,然后从这段程序的输出来总结一下枚举常量指定数值的规律。

```

#include <stdio.h>

int main()
{
    enum week {Monday = 1, Tuesday, Wednesday = 4, Thursday, Friday, Saturday, Sunday};
    enum weekday {Mon, Tue = 4, Wed, Thu, Fri, Sat, Sun};

    printf("Monday    = %d\n",Monday);
    printf("Tuesday   = %d\n",Tuesday);
    printf("Wednesday = %d\n",Wednesday);
    printf("Thursday  = %d\n",Thursday);
    printf("Friday    = %d\n",Friday);
    printf("Saturday  = %d\n",Saturday);
    printf("Sunday    = %d\n\n",Sunday);

    printf("Mon      = %d\n",Mon);
    printf("Tue      = %d\n",Tue);
    printf("Wed      = %d\n",Wed);
    printf("Thu      = %d\n",Thu);
}

```

```

printf("Fri    = %d\n", Fri);
printf("Sat    = %d\n", Sat);
printf("Sun    = %d\n", Sun);

return 0;
}

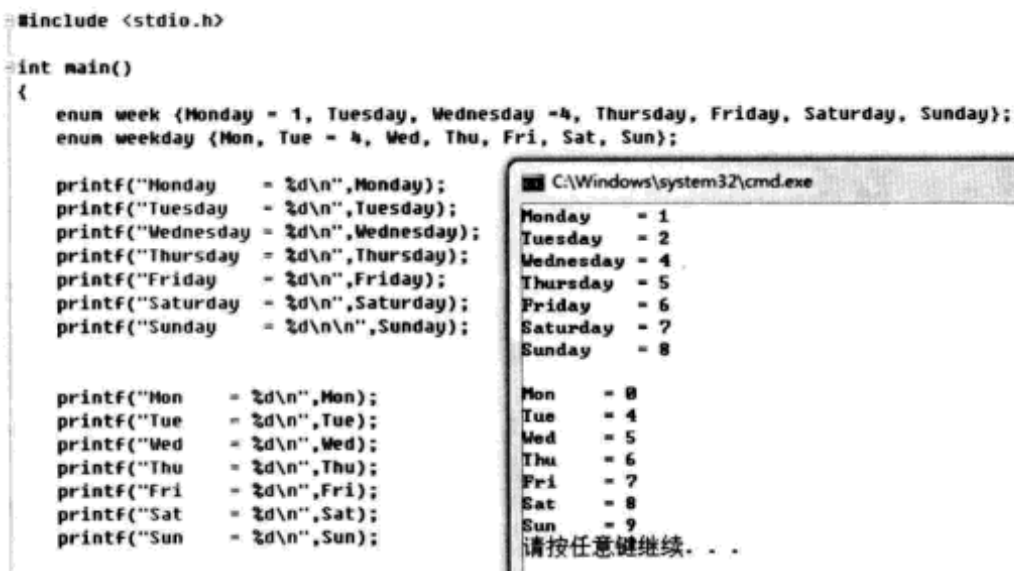
```

在这段程序中，定义了两个枚举类型 `week` 和 `weekday`，并分别给其中的某些枚举常量指定了数值，然后将所有的枚举常量按照整型数值进行输出。

程序的输出如图 10.4 所示。我们给 `Monday` 指定数值为 1，给 `Wednesday` 指定数值为 4，给 `Tuesday` 指定数值为 2，所指定的值都按照我们的期望输出了。大家有没有看出没有指定数值的枚举常量遵照的什么样的规则呢？这个规则其实很简单，具体如下。

- ❑ 没有指定数值的枚举常量的值是紧随指定数值的枚举常量的值，依次递增的。例如，给 `Monday` 指定了数值 1，其后的 `Tuesday` 的值就随其递增为 2。给 `Wednesday` 指定数值为 4，其后的枚举常量的值就依次是 5、6、7 和 8。
- ❑ 如果第一个枚举常量的值没有指定，那么它的值就为 0。例如，程序中的 `Mon`，其余的紧随其后的枚举常量的值依次递增。

有了这两条规则，就可以根据已知的枚举常量的数值，确定没有指定数值的枚举常量的值了。



```

#include <stdio.h>

int main()
{
    enum week {Monday = 1, Tuesday, Wednesday = 4, Thursday, Friday, Saturday, Sunday};
    enum weekday {Mon, Tue = 4, Wed, Thu, Fri, Sat, Sun};

    printf("Monday    = %d\n", Monday);
    printf("Tuesday    = %d\n", Tuesday);
    printf("Wednesday   = %d\n", Wednesday);
    printf("Thursday    = %d\n", Thursday);
    printf("Friday      = %d\n", Friday);
    printf("Saturday    = %d\n", Saturday);
    printf("Sunday      = %d\n", Sunday);

    printf("Mon       = %d\n", Mon);
    printf("Tue       = %d\n", Tue);
    printf("Wed       = %d\n", Wed);
    printf("Thu       = %d\n", Thu);
    printf("Fri       = %d\n", Fri);
    printf("Sat       = %d\n", Sat);
    printf("Sun       = %d\n", Sun);
}

```

C:\Windows\system32\cmd.exe

```

Monday    = 1
Tuesday    = 2
Wednesday   = 4
Thursday    = 5
Friday      = 6
Saturday    = 7
Sunday      = 8

Mon       = 0
Tue       = 4
Wed       = 5
Thu       = 6
Fri       = 7
Sat       = 8
Sun       = 9
请按任意键继续. . .

```

图 10.4 枚举常量指定值

10.2.2 枚举变量的定义

相对于结构体变量和共同体变量而言，枚举变量的定义要简单得多。因为枚举是一种基本数据类型，就像之前介绍的整型 `int` 一样。与整型 `int` 唯一不同的就是，枚举类型是自定义的，而 `int` 类型是系统规定的。

C 语言中，枚举变量的定义形式如下所示：

枚举类型名 枚举变量名；

这个表示形式中，“枚举类型名”就是已经定义的枚举类型。例如，`enum week` 和结构体、共同体是类似的。“枚举变量名”就是定义的枚举变量，只要遵循 C 语言标识符命名规范就可以。

遵照这个定义形式，如果想要定义一个 `enum week` 枚举类型的变量 `day`，就可以使用下面的 C 语言语句。

```
enum week day;
```

10.2.3 枚举变量的使用

有了枚举类型变量以后，就可以使用它来保存枚举数据，并且可以在需要的时候把保存的枚举数据拿出来使用。枚举变量的使用要比结构体变量和共同体变量简单得多。因为枚举类型是一种基本数据类型，没有成员变量的概念，因而没有取成员变量运算。

使用枚举变量，简单得就像使用整型 `int` 变量一样，使用等号赋值运算给其赋值，直接使用已经赋值的变量来进行其他的操作。先来看一段程序，看看 C 语言程序中是如何使用枚举变量的。

```
#include <stdio.h>

int main()
{
    enum week {Monday = 1, Tuesday, Wednesday = 4, Thursday, Friday, Saturday, Sunday};
    enum weekday {Mon, Tue = 4, Wed, Thu, Fri, Sat, Sun};

    enum week    day1;
    enum weekday day2;

    day1 = Thursday;
    printf("Thursday = %d\n", day1);

    day2 = Fri;
    printf("Thursday = %d\n", day2);

    return 0;
}
```

在这个程序中，定义了两种枚举类型 `enum week` 和 `enum weekday`，并且分别定义了两个对应的枚举变量 `day1` 和 `day2`，然后为其赋值枚举常量，最后将枚举变量中的数值按照整型进行输出。程序的输出如图 10.5 所示，如我们所期望的，程序正常输出了枚举变量中保存的数值。

```
#include <stdio.h>

int main()
{
    enum week {Monday = 1, Tuesday, Wednesday = 4, Thursday, Friday, Saturday, Sunday};
    enum weekday {Mon, Tue = 4, Wed, Thu, Fri, Sat, Sun};

    enum week    day1;
    enum weekday day2;

    day1 = Thursday;
    printf("Thursday = %d\n", day1);

    day2 = Fri;
    printf("Thursday = %d\n", day2);

    return 0;
}
```

C:\Windows\system32\cmd.exe
Thursday = 5
Thursday = 7
请按任意键继续. . .

图 10.5 枚举变量的使用

10.3 枚举数组和枚举指针

枚举是一种数据类型，所以它也是有数组和指针的。枚举数组是用来保存多个枚举数据的，而枚举指针是用来保存枚举数据所在内存的地址的。不过，相对于结构体和共同体而言，枚举数组和指针要简单得多，因为枚举是一种基本数据类型，你可以按照 `int` 类型的使用方法来使用它。

10.3.1 枚举数组

枚举数组和其他类型的数组作用一样，都是用来保存多个某种类型的数据，枚举数组用来保存多个枚举数据。当然，也可以按照一维枚举数组、二维枚举数组甚至多维枚举数组的形式来保存多个枚举数据。不过，这里只介绍一维枚举数组，对于其他维枚举数组，大家可以参照数组一章进行学习，性质和使用方法都是类似的。在 C 语言中，一维枚举数组的定义形式如下所示：

```
枚举类型名 枚举数组名[表达式];
```

在这个表示形式中，“枚举类型名”就是已经定义了的枚举类型，如 `enum week`。“枚举数组名”是我们将要定义的枚举数组。“表达式”是一个常量或者是一个常量表达式，表示要定义的枚举数组中可以保存多少个枚举数据。例如，要定义一个可以保存 5 个 `week` 枚举数据的枚举数组，可以使用下面的 C 语言表示了。

```
enum week days[5];
```

有了枚举数组之后，就可以往里面保存枚举数据了，保存的方法有两个：初始化和赋值。对枚举数组进行初始化，有点类似于整型数组的初始化，形式如下：

```
枚举类型名 枚举数组名[表达式] = {枚举常量, 枚举常量, ..., 枚举常量, 枚举常量};
```

例如，要将枚举数组 `days` 初始化，就可以使用下面的 C 语言语句：

```
enum week days[5] = {Wednesday, Friday, Friday, Sunday, Monday};
```

要使用赋值的方式给枚举数组中保存数据，也很简单，只要使用取数组元素运算，然后使用简单的赋值运算把要保存的数值保存到取出的数组元素中即可。例如，给枚举数组 `days` 的第二个元素赋值为 `Friday`，就可以使用下面的 C 语言语句。

```
days[1] = Friday;
```

要使用保存在枚举数组中的数据也很简单，还是使用取数组元素运算，取出数组中保存的枚举数据，然后就可以使用这一数据进行其他运算和操作了。

10.3.2 枚举指针

枚举指针是用来保存枚举数据在内存中的地址的。有了这一地址，就可以使用地址进

行枚举数据的访问和操作了。在 C 语言中，枚举指针类型的定义如下所示：

枚举类型名 *

在这个表示中，“枚举类型名”就是之前定义的枚举类型，如 `enum week`。“*”是指针类型表示符号，是由 C 语言规定的。要保存枚举数据的地址，首先得定义一个枚举指针变量，然后使用取值运算取得枚举数据的地址，最后使用赋值运算将取得的枚举数据的地址赋值给定义的枚举指针变量。

先来看第一步，定义一个枚举指针变量，其 C 语言表示为：

枚举指针类型 枚举指针变量名；

在这个表示中，“枚举指针类型”就是定义的枚举指针类型，如 `enum week *`。“枚举指针变量名”就是要定义的枚举指针变量的名字，其命名只要遵循 C 语言对标识符的要求就可以。要定义一个 `enum week` 枚举类型的指针变量，C 语言表示如下所示：

```
enum week * p_day;
```

再来看第二步，取得枚举数据的内存地址。这就需要用到取地址运算了。例如，定义了一个 `enum week` 变量 `day`，要确定它的地址，可以使用下面的 C 语言表示。

```
&day
```

最后来看第三步，很简单，只要一个等号赋值运算表达式就可以搞定：

```
enum week * p_day;
enum week day;
p_day = &day;
```

知道如何将一个枚举数据的地址保存到枚举指针变量中以后，下面看看如何使用枚举指针取出内存中的枚举数据。也很简单，只要一个取指针元素运算就可以：

***枚举指针变量**

其中“*”是取指针元素运算运算符；“枚举指针变量”就是已经保存了枚举数据地址的指针变量。例如，要取出 `p_day` 中的枚举数据，就可以使用下面的 C 语言表示：

```
*p_day
```

10.3.3 用枚举指针来访问枚举数组

本小节是对上面两个小节的总结和融合。枚举数组用来保存多个枚举数据，而枚举指针用来表示枚举数据在内存中的位置。用下面的程序看看如何用枚举指针访问枚举数组元素。

```
#include <stdio.h>

int main()
{
    enum week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

    enum week days[5] = {Wednesday, Friday, Friday, Sunday, Monday };
    enum week *p_day;
```

```

int i;

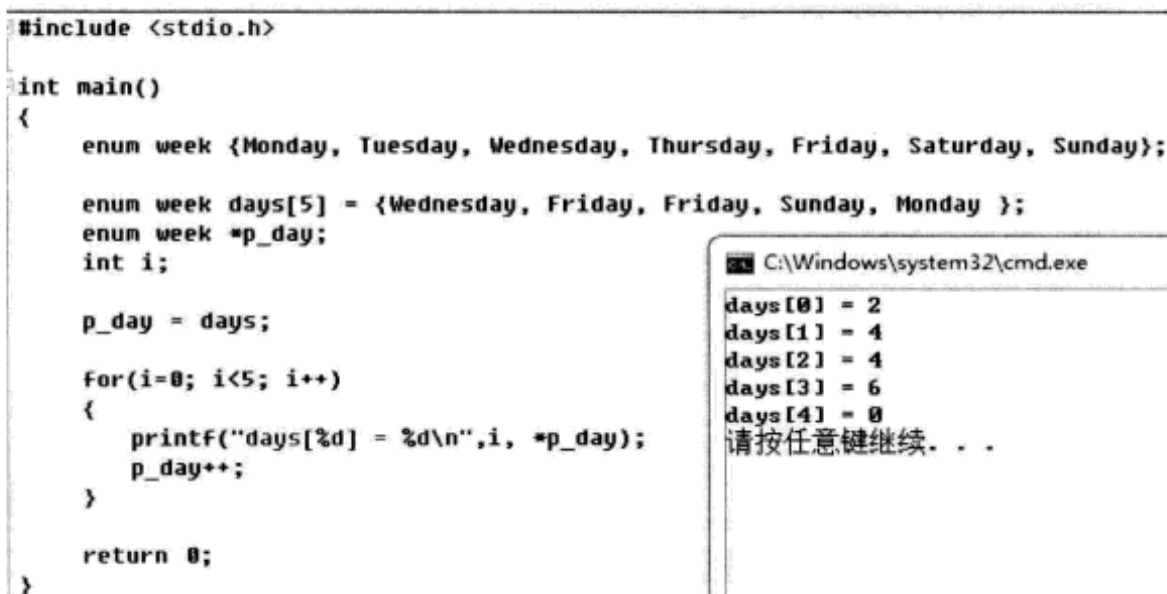
p_day = days;

for(i=0; i<5; i++)
{
    printf("days[%d] = %d\n", i, *p_day);
    p_day++;
}

return 0;
}

```

在这个程序中，先定义了一个枚举类型 `enum week`，然后定义了一个 `enum week` 的枚举数组 `days`，并对其进行初始化。接着又定义了一个 `enum week` 的枚举指针，并将 `days` 数组的首地址，也就是数组名赋值给 `p_day`。最后，使用 `for` 循环、`p_day` 指针变量和 `printf()` 函数输出了枚举数组 `days` 中的所有的值。程序的输出如图 10.6 所示，程序正确地输出了 `days` 枚举数组中所有元素的值。



```

#include <stdio.h>

int main()
{
    enum week {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

    enum week days[5] = {Wednesday, Friday, Friday, Sunday, Monday };
    enum week *p_day;
    int i;

    p_day = days;

    for(i=0; i<5; i++)
    {
        printf("days[%d] = %d\n", i, *p_day);
        p_day++;
    }

    return 0;
}

```

C:\Windows\system32\cmd.exe

```

days[0] = 2
days[1] = 4
days[2] = 4
days[3] = 6
days[4] = 0
请按任意键继续. . .

```

图 10.6 使用枚举指针访问枚举数组元素

10.4 typedef 类型定义符

C 语言是一种很灵活的语言，它的灵活体现在很多方面。C 语言在数据类型方面的灵活性，主要体现在任何已有的数据类型名（无论是系统规定的还是自己定义的）都是可以重新定义的。

要想重新定义一个已有的类型名，需要用到 C 语言的一个关键字 `typedef`，这个关键字被称为类型重定义关键字。它可以给一个已有的类型重新取个名字，无论这个类型是系统规定的，像 `int`、`float` 和 `char`，还是由用户自己定义的，如 `struct student`、`union radius` 和 `enum week`。

C 语言中，类型重定义表示形式如下所示：

```
typedef 已有类型名 重定义类型名;
```

在这个表示形式中，“`typedef`”是类型重定义关键字。“已有类型名”就是已经有的

类型名，它可以是系统规定的，如 `int`，也可以是用户自己定义的，如 `struct student`。“重定义类型名”就是我们想要给已有类型名重新取的名字，可以是任何没有使用的 C 语言标识符，得遵循 C 语言标识符命名规范。“;”是必须有的，因为类型定义是条语句。

例如，给系统规定的 `int` 类型名，重新取个名字 `my_int`，就可以使用下面的 C 语言语句。

```
typedef int my_int;
```

有了这个类型重定义以后，就既可以使用 `my_int` 类型名来定义整型变量，又可以使用 `int` 类型名来定义整型变量了，如下面的程序所示：

```
#include <stdio.h>

typedef int my_int;

int main()
{
    int    radius1;
    my_int radius2;

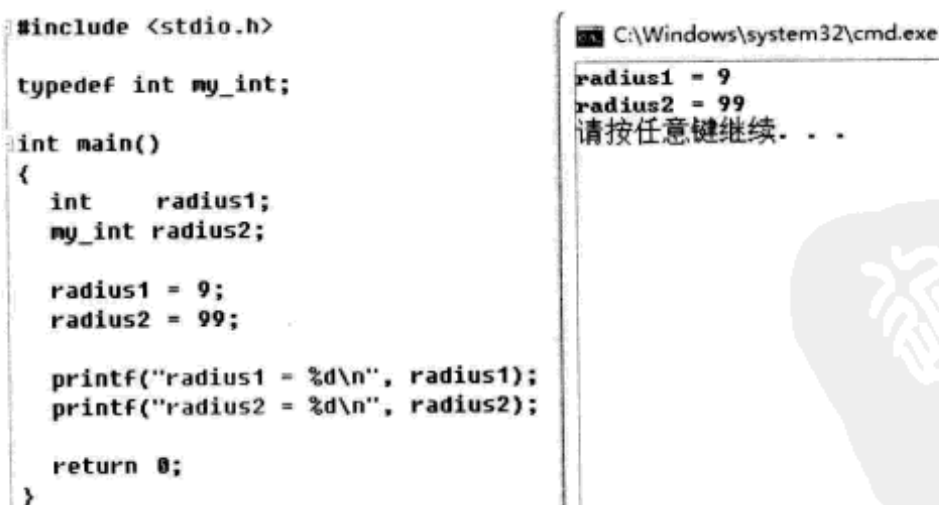
    radius1 = 9;
    radius2 = 99;

    printf("radius1 = %d\n", radius1);
    printf("radius2 = %d\n", radius2);

    return 0;
}
```

在这段程序开始的时候，使用 `typedef` 把 `int` 类型重新取了个名字 `my_int`。之后在程序中，使用 `int` 和 `my_int` 类型名分别定义了两个整型变量 `radius1` 和 `radius2`。接着，给 `radius1` 赋值为 9，给 `radius2` 赋值为 99。最后，使用 `printf()` 函数将 `radius1` 和 `radius2` 的值分别输出。

程序的输出如图 10.7 所示，程序正常输出了两个整型变量中保存的值，可见 `typedef` 确实可以重新命名已有的类型名，另外重新命名的类型名和原有的类型名在程序中都可以用来定义变量。



```
#include <stdio.h>

typedef int my_int;

int main()
{
    int    radius1;
    my_int radius2;

    radius1 = 9;
    radius2 = 99;

    printf("radius1 = %d\n", radius1);
    printf("radius2 = %d\n", radius2);

    return 0;
}
```

```
C:\Windows\system32\cmd.exe
radius1 = 9
radius2 = 99
请按任意键继续. . .
```

图 10.7 `int` 类型重定义

现在我们已经知道了，`typedef` 可以对已有的类型名重新取个名字。但是，已有的数据类型名用得好好地，为什么要对已有的数据类型名进行重命名呢？主要有以下两个原因。

□ 丰富和具体化已有的数据类型。这主要体现在两个方面。（1）系统规定的数据类型

型就那么几种，我们定义的数据类型也不够多。要想使用更多类型名，要么重新定义类型，要么重新给已有类型取个名字。如果现有的数据类型可以重新使用，何不干脆取个名字呢！（2）系统定义的类型。例如，`int`除了表示定义一个整型数据，再没有任何意义。要想使用一个 `fruit` 类型来表示水果，它也是整数，就可以将 `int` 类型重新命名为 `fruit`，将 `int` 类型的含义具体化。

- 简化类型名表示。我们知道，在定义结构体、共同体和枚举类型的时候，都得加上 `struct`、`union` 和 `enum` 关键字。如果嫌麻烦就可以使用 `typedef` 给已经定义的结构体、共同体和枚举类型重新取个名字，省掉 `struct`、`union` 和 `enum` 关键字。例如，可以将 `struct student` 重新取名为 `student`。C 语言表示如下所示：

```
typedef struct student student;
```

有了这个类型重定义，就可以使用 `student` 代替 `struct student` 定义结构体变量了。

10.5 小 结

到这一章为止，我们介绍完了 C 语言中的三种自定义数据类型——结构体、共同体和枚举类型，它们之间既有相似之处又有区别，对比着学习会得到更好的收获。本章的重点是理解枚举类型的含义和使用方法，以及它与结构体和共同体的区别，难点是枚举类型和数组及指针的结合使用。下一章将讲解 C 语言中很重要的一个知识点——函数，它是写大型程序时必不可少的方法。

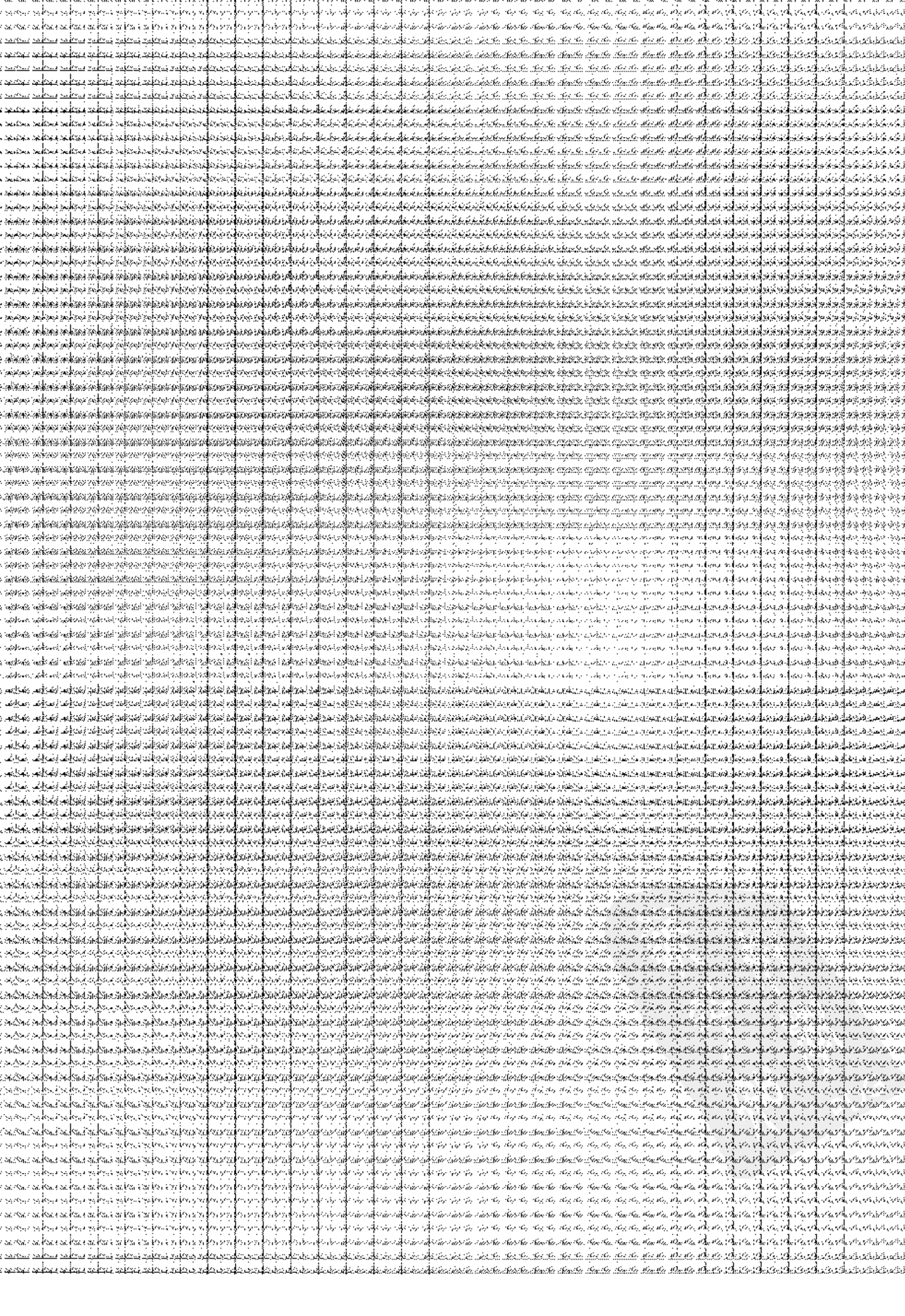
10.6 习 题

【题目 1】 定义一个枚举类型，用来表示三基色：红、绿、蓝，并给这个枚举类型重新取一个名字 `RGB`。

【分析】 定义一个枚举类型和定义结构体、共同体有点不同，因为枚举类型没有成员变量，但是，枚举类型是有枚举常量的，只要我们能够表示枚举常量就可以了。给一种类型重新取名，需要用到 `typedef` 类型定义符，这个很简单，按照类型重定义的 C 语言表示来做就可以了。

【核心代码】

```
enum colour_base {Red, Green, Blue};  
typedef enum colour_base RGB;
```

第4篇 复杂功能的实现

▶▶ 第11章 函数

▶▶ 第12章 特殊的函数——main()函数

▶▶ 第13章 局部变量和全局变量



第 11 章 函 数

我们知道 C 语言程序是由一条条的语句组成的。所以，若想写一个 C 语言程序，可以写一条一条的语句，并按照顺序结构、分支结构和循环结构体中的一种或者多种对语句进行组织。要写简单的程序这样是没有问题的，因为语句数很少。如果写大型的程序这样的方式就显得太凌乱了，甚至无法编写程序。要解决这个问题，可以将大型程序分成一个个小块来编写，再将这些小块串起来组成大型程序，就不会凌乱了。函数就是将大型程序分块和组块的方法。

11.1 函数的意义

在讲函数的含义之前，先来看一个实际生活中的例子，通过这个例子可以更容易地理解函数在 C 语言程序中的意义和作用。

假如你是个雕刻艺术家，有人认为写代码是一种艺术，写代码的也是艺术家，反正都一样，都是进行一种创造性活动。艺术家在进行创作的时候，面对的最基本的问题就是以怎样的方式加工原材料，画家如何使用原材料笔墨纸砚，作家如何使用原材料文字，作曲家如何使用音符等，都是在创作之初最先面临的问题。

当然，雕刻家也会面临这样的问题。例如，给你一块玉，刚出土的，希望你把它加工成一个首饰。这个时候，你就得考虑一下该如何加工这块玉呢？是直接在这块玉上雕琢呢，还是把这块玉石分成一块一块再进行加工呢？诸如此类，一系列问题就出现了，这些问题已经不再涉及雕刻细节，而是对艺术作品的整体把握。

对于这块玉的处理方法基本有两种：一种是一体化处理，一种是分块化处理。一体化处理是可以将这块玉加工成一个玉手镯；分块化处理是可以将这块玉加工成一条玉手链。如图 11.1 所示，手镯和手链就是对玉石原材料进行加工的两种截然不同的方式。

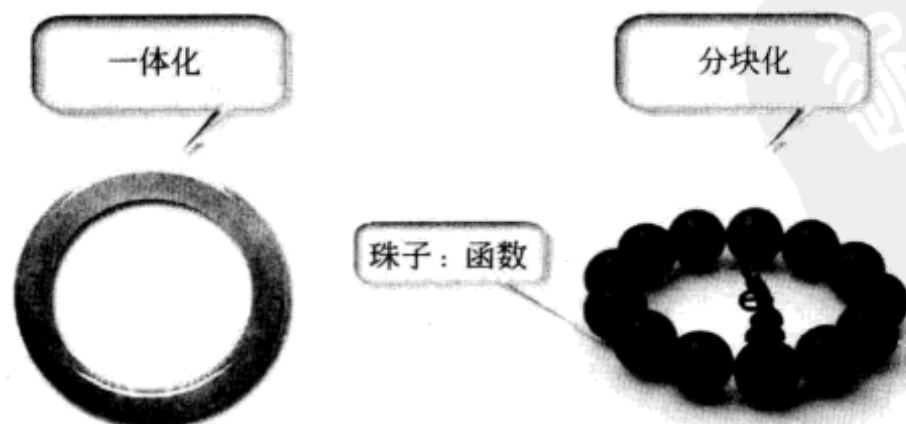


图 11.1 对玉石的处理方式

其实，程序员面临的问题和雕刻家是一样的。当我们掌握了 C 语言的各个原材料——常量、变量、语句、顺序结构、分支结构和循环结构体等之后，接下来的事情就是把这些原材料加工成程序，来达到操作计算机的目的。其实，这个时候已经不涉及 C 语言的细节了，纯属整体把握问题！

如果无从下手，就要向雕刻家学习学习。因为问题是类似的。雕刻家的方法有两种：一体化和分块化，我们也可以使用这两种方法。如果按照一体化的方式，可以直接使用 C 语言原材料（常量、变量、语句等）来一个一个堆积成我们需要的程序。如果按照分块化的方式，就先使用 C 语言原材料加工成类似于图 11.1 中的珠子的中间块，然后将这些珠子（中间块）串起来，组成需要的程序。

在写程序的时候，这两种方式都是可行的，但是对于不同规模的程序，两种方式的好坏程度是不一样的。对于完成简单功能的小型程序，使用一体化的方式会更好，因为小型程序本来只使用简单的几行语句就可以完成，如果硬要将它划分成一个小块，就有点画蛇添足了。对于复杂功能的大型程序，使用分块化的方式会更好，因为对于有成千上万行甚至百万行语句的程序，如果将这些语句都写到一块，即使机器不崩溃，你自己写着都会崩溃。

就像把玉石做成一个一个的玉珠子一样，可以把大型的程序分成一个小块。在 C 语言中，这样的小块被称为函数。函数在大型程序中起着两个很重要的作用。

- ❑ 将复杂的问题进行分解。往往程序越大，完成的功能就会越复杂，要解决的问题也就越复杂。对于复杂的问题可以将其分解，类似地，对于大型的程序也可以将其分解为一个个简单的函数。打磨一个玉珠子总比打磨一个手镯容易得多吧！
- ❑ 分解的小块可以重复使用。这就像我们把玉石打磨成一个个小玉珠子，今天可以把玉珠子串成玉手链，明天可以把玉珠子串成玉项链，后天还可以把玉珠子镶嵌到发卡上，诸如此类有很多用法。试想，如果你把玉石雕琢成手镯，能有这么多的用途吗？

11.2 函数的形式

函数就像玉链子中的珠子，起到 C 语言半成品的作用。我们知道玉珠子是一个圆圆的球形的东西，中间有一个孔，用来把珠子串起来组成一个整体的手链。C 语言中的函数长什么样？它有没有孔？用什么把函数串成一个有用的程序呢？带着这些问题，来学习本节。

11.2.1 函数的一般形式

C 语言中的函数主要由两部分组成，分别是：函数外特性部分和函数内特性部分。其具体形式如下所示：

```
返回值类型 函数名(参数列表)
{
    函数体
}
```


这是一个完整的 C 语言函数的样子，以大括号为界。大括号之前的部分是函数的外部特性，也就是函数的样子。大括号中的部分就是函数的内部特性，主要由 C 语言原材料组成，用来完成一定的功能。

先来看一看函数的外特性部分，主要有三个元素：返回值类型、函数名和参数列表。

- ❑ “函数名”就是给这个函数取的名字，就像珠子是圆的，函数的样子主要体现在函数名上，函数名只要遵循 C 语言命名规范就好，也是一个 C 语言标识符。
- ❑ “参数列表”在函数中可有可无，看自己需要来定。它主要是一些变量的定义，如 `int a`。如果有多个变量定义，各个变量定义之间用逗号隔开。
- ❑ 函数的“返回值类型”是一个数据类型表示符。它主要用于表示函数运行完成后将给出一个什么类型的数据。

对于函数的内特性部分，将在本章后面的部分详细讲到。现在大家只要知道它是由 C 语言原材料（常量、变量、语句、顺序结构、分支结构和循环结构等）组成的，是为了完成一定的功能就可以了。

例如，要完成一个加法功能的函数，就可以使用上面的函数形式定义如下的函数：

```
int add(int a, int b)
{
    完成加法运算的语句
}
```

在这个定义中，`add` 就是我们所说的“函数名”，在这里表示的是这个函数完成的是加法功能。`int a` 和 `int b` 就是函数的“参数列表”，这里表示的是加法运算需要的两个加数。最前面的 `int` 就是函数的“返回值类型”，这里表示函数将会得到一个 `int` 类型的值，其实就是表示加法运算得到的结果是整型数值。

11.2.2 函数的参数列表

函数的参数列表的作用就是为函数提供需要的数据，这些数据保存在函数的参数列表中。像之前定义的加法函数中，两个整型参数的作用就是用来保存加法运算所需要的两个加数。

更形象点，函数的参数列表相当于玉珠子中间孔的一端。不过函数的“孔”有前后端之分，玉珠子的孔没有前后端之分。函数的参数列表相当于是函数孔的前端，串起所有函数的“针”就是先从这个孔进入的。

图 11.2 所示就是形象的类比，串玉珠子的线从珠子孔的一端进入，串函数的“线”从函数的参数列表进入，并且把函数所需要的数据从参数列表带进去。

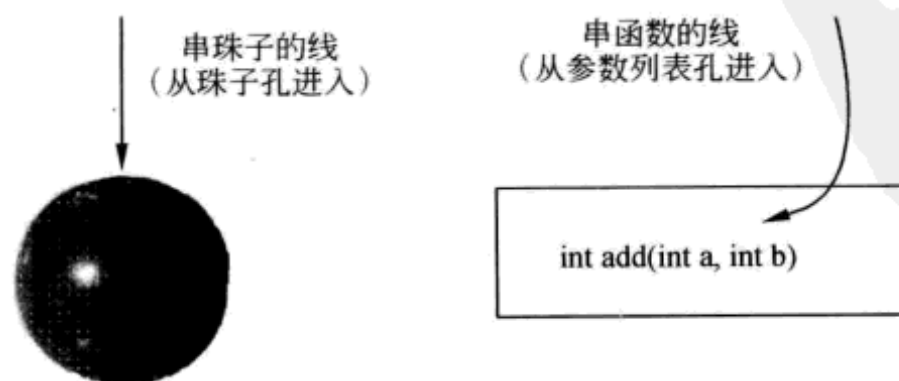


图 11.2 函数的参数列表——入口

11.2.3 函数的返回值类型

用线串珠子，从珠子一端的孔进入，从珠子另外一端的孔出来。要想把函数“串”起来，从函数的参数列表进，从什么地方出呢？从函数的返回值出。函数的“返回值类型”表示的就是函数的返回值的数据类型，如整型、字符型、浮点型、指针类型、结构体类型、共同体类型和枚举类型。

函数的返回值就是函数完成功能后交出来的数据，当然这是可选的，可以交也可以不交。对于怎样从函数中返回一个值，在后边讲函数体的时候会详细讲解。这里，大家只要知道“函数的返回值”就是函数的“出口”，“函数的返回值类型”代表的是函数返回值的数据类型就可以了。

图 11.3 是玉珠子的出口和函数的出口，串珠子的线从珠子孔的另一端出来，串函数的“线”从函数的返回值出来。

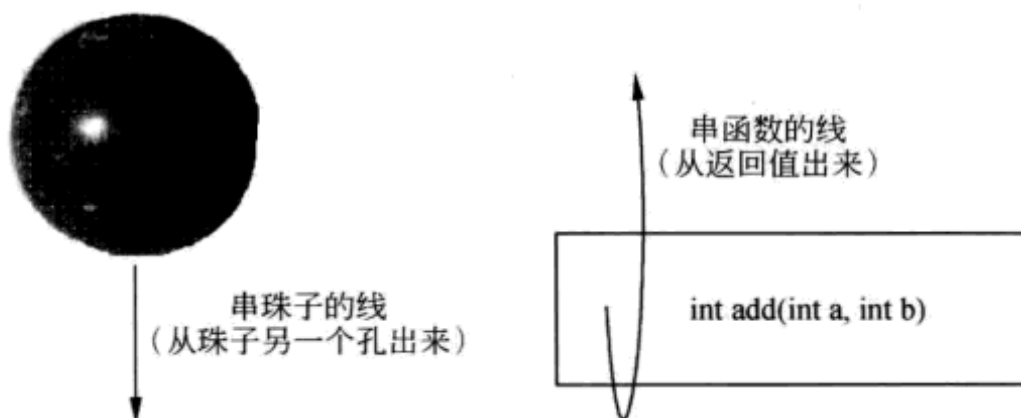


图 11.3 函数的返回值——出口

11.3 函数的声明和定义形式

在讲变量的时候，我们知道 C 语言中的变量有声明还有定义。声明是告诉计算机需要什么类型的变量，定义是计算机分配给你一个什么类型的变量。函数也是类似的，也有声明和定义，函数的声明告诉计算机函数长什么样，函数的定义告诉计算机函数是怎样实现我们需要的功能的。

11.3.1 函数的声明

在变量一节，我们知道了变量的声明和定义是在一起的。一个变量声明定义语句，就完成了变量的声明和定义。函数的声明是不是与之类似，可以和函数定义一起完成呢？

答案是肯定的，在完成函数定义的时候，可以把函数的声明也一次性完成！在告诉计算机我们的函数在如何做事的时候，顺便把函数长啥样也让计算机看到，这是顺理成章的事，难不成让函数蒙着面干活？不过，计算机中确实有蒙着面干活的函数，这不在我们讲解的范畴之内。

C 语言中，函数声明和定义在一起的形式如下所示：

```
返回值类型 函数名(参数列表)
{
    函数体
}
```

这个表示形式同时完成了函数的声明和定义，被称为函数声明定义表示，和前面讲的函数的一般形式是一样的！大括号之前的部分完成函数的声明，也就是告诉计算机函数长什么样，入口是什么样的，出口又是怎样的。大括号和它之中的部分完成函数的定义，也就是告诉计算机该如何完成我们所要的功能。

不过，C 语言还提供了另外一种专门用来声明函数的形式，只告诉计算机函数长什么样，暂时不告诉函数是如何实现想要的功能的。其形式如下所示：

```
返回值类型 函数名(参数列表);
```

这是一个函数声明语句，它和前面的函数声明定义形式的前半部分是一样的，只不过多了一个分号作为语句结束的符号。有了这个函数声明就可以告诉计算机，有长相是这样的函数可以使用。至于这个函数是如何实现想要的功能的，暂时先不用关心。

例如，要告诉计算机，有一个加法函数，它的长相为：两个整型参数作为加数，函数名为 `add`，将会返回一个 `int` 类型的值，就可以使用下面的函数声明语句：

```
int add( int a, int b);
```

11.3.2 函数的定义形式

上一小节介绍了函数声明的两种表示形式：（1）声明和定义一起；（2）单独声明。有了这两种函数声明形式，就可以告诉计算机函数长什么样了。光告诉计算机函数长什么样还不够，还得告诉计算机如何实现我们需要的功能，这就是函数的定义。

函数的声明可以用两种形式，但是函数的定义就只有一种，其形式如下所示：

```
返回值类型 函数名(参数列表)
{
    函数体
}
```

函数的定义形式就是上面这一种，声明和定义在一起。之所以在定义的时候还不忘声明，告诉计算机函数长啥样，是因为如果没有声明部分，谁知道你定义的是哪个函数啊？

这里主要关注函数的定义，也就是上面的函数定义形式中大括号之中的部分，这部分主要是告诉计算机该如何完成函数要完成的功能。C 语言中，函数定义部分细分起来可以由三部分组成：声明部分、功能实现部分和返回值部分。按照上面所说，将函数的函数体展开就成了下面的样子。

```
返回值类型 函数名(参数列表)
{
    声明部分

    实现部分
}
```

返回值语句

}

函数体的声明部分，就是函数中所需要的所有的声明，主要有变量声明、类型声明等；实现部分就是实现函数功能的语句及组织这些语句的结构（顺序结构、分支结构和循环结构）；返回值语句用来返回得到的数据。

对于这几部分，C 语言有一些限制：声明部分必须位于另外两部分之前，在实现部分之中，之后都可以出现一个或者多个返回值语句。例如，下面的函数定义就会出现错误：

```
void fun()
{
    int i;
    i++;

    int j;
    j++;
}
```

这个程序中，为了简单省去了函数的参数列表和返回值，要省去参数列表，只要不在小括号中写任何东西就可以，要省去返回值，只要把返回值类型改为 void 就可以了。这个程序出错的主要原因就是，声明语句 int j 位于执行语句 i++, j++之中了。

出现的错误如图 11.4 所示，错误显示程序的第 8 行，也就是“int j;”这一行缺少分号，第 9 行，也就是“j++”这一行 j 没有声明。当然，这个错误说得并不是很准确，但是确实反映了错误是由于 j 定义位置不对引起的。

```
1>----- 已启动生成: 项目: test, 配置: Debug Win32 -----
1> test.c
1>e:\c语言的书\project\test\test\test.c(8): error C2143: 语法错误 : 缺少“;” (在“类型”的前面)
1>e:\c语言的书\project\test\test\test.c(9): error C2065: “j”: 未声明的标识符
===== 生成: 成功 0 个, 失败 1 个, 最新 0 个, 跳过 0 个 =====
```

图 11.4 函数定义错误

将程序改为下面的样子，符合 C 语言中对函数定义的要求，就不会有图 11.4 所示的错误了。

```
void fun()
{
    int i;
    int j;

    i++;
    j++;
}
```

11.3.3 函数的形参

在上面的例子中，为了简单省略了函数的参数列表，这里专门讲一下函数的参数列表。函数定义中的参数列表，在 C 语言中被称为形式参数，或者简称形参。之所以称之为形式参数，是因为在定义函数时，参数列表中变量的值事先是不知道的，只能知道这些变量的

类型及含义，也就是这些参数的形式。

形参实质上就是一些变量，对于这些变量，我们已经知道了它们的类型和含义，这些就足够完成我们需要的功能了。这就好比我们在干活的时候，师傅给了几个盒子，说盒子中放的是 XXX，需要的时候来取就可以了。这里师傅给的盒子就类似于函数中的形式参数，我们知道参数中放的是什么，这样在需要的时候就可以使用了。

图 11.5 中，上面的每个形参变量就相当于一个盒子，里面将会放函数中需要的数据，至于这个数据是多少，暂时还不知道。当函数体中需要数据的时候，就可以直接打开盒子的盖子，然后从中取得数据。

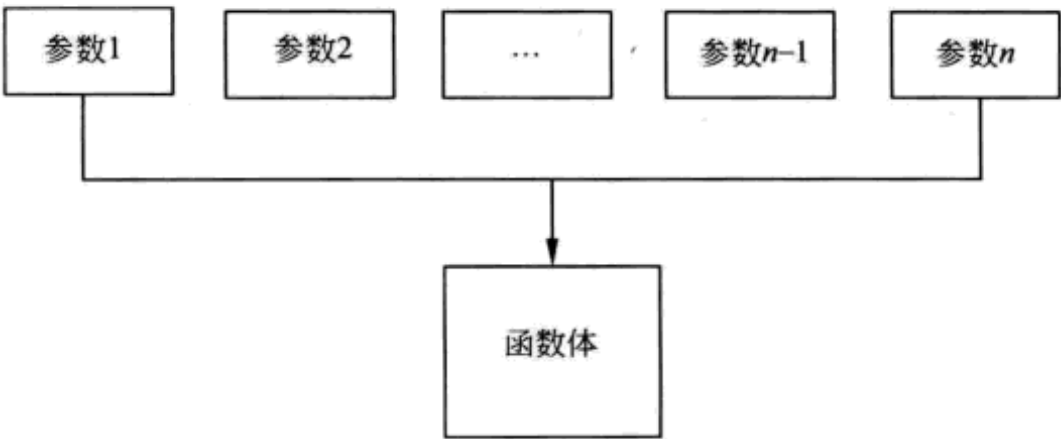


图 11.5 形式参数

例如，若规定了一个加法运算，其参数列表为 `int a`、`int b`，代表的是两个加数，而且加数的类型为整型，具体数据是什么，还不知道。在实现加法的时候，直接从这两个变量中取出数据加起来就可以，至于取的是多少暂时就不用管了。

另外，有个问题需要注意一下，函数的形参都是变量，变量是有使用范围的，也就是变量的作用域。函数形参的使用范围仅限于函数体，也就是大括号之中，出了大括号就不能使用这些形参变量了。

11.3.4 return 返回值语句

在函数定义的时候，往往要把计算的结果或者有关信息报告给计算机，这个时候就需要返回值语句返回一些数据了。C 语言中的返回值语句如下所示，使用了一个关键字 `return`，因此返回值语句也被称为 `return` 语句。

```
return 表达式;
```

这个语句中，“`return`”是一个关键字，不容更改，也不可缺少。“表达式”可以是一个常量、常量表达式、变量或者变量表达式。不过无论是什么，其中的表达式值类型要和函数的返回值类型一样，不然就会发生隐式的类型转换。

例如，要返回两个数的和，可以有以下三种方式。

- ❑ 返回常量表达式。假设要计算的是 5 和 4 的和，就可以直接返回 `5+4`。

```
return 5+4;
```

- ❑ 返回变量。假设返回的还是 `5+4` 的和，可以先把 `5+4` 计算的结果保存到一个变量中，然后返回这个变量保存的值。

```
int result;

result = 5+4;
return result;
```

- 返回变量表达式。假设返回的还是 5+4 的和，可先将 5 和 4 分别保存到两个变量 `int a` 和 `int b` 中，然后返回变量表达式 `a+b` 也是可以的。

```
int a, b;

a = 5;
b = 4;
return a+b;
```

可以随便使用以上任何一种形式返回想要的数，不过还是要注意一点。`return` 语句执行以后，它后面的语句将不会再执行，所以要慎重使用 `return` 语句，在该结束的地方返回，特别是对于分支和循环语句中的 `return` 语句，更要慎重。

例如，下面的程序片段中，当 `i` 等于 5 的时候，函数就会返回，不会输出 5 和 5 之后 `i` 的值。

```
for(i=0; i<10; i++)
{
    if(i == 5) return i;
    printf("i = %d\n", i);
}
```

11.4 自己动手写一个函数——加法函数

知道了在 C 语言中如何声明定义一个函数之后，下面来动手实践一下吧，声明定义一下之前讲到的加法函数。通过下面的几个环节就可以完成加法函数的定义。

11.4.1 确定加法函数的样子

要确定加法函数长什么样，就得确定三样东西：函数名，入口——形参，出口——返回值类型。这三个部分也正是函数声明所需要的。

- 函数名就是要给加法函数取一个名字，`add`，`my_add`，`add_function`，`Add` 等都是可以的。只要遵循 C 语言标识符命名规范就可以，这里暂且选择 `add` 吧。
- 函数的形参，也就是要提供给函数的数据该放的地方。要进行加法运算，需要的数据是两个进行加法运算的加数。暂且假定函数只对整数进行加法运算，那么就可以用两个整型变量 `int a` 和 `int b` 来保存两个加数。当然也可以选择其他变量来保存这两个数据，数组、指针、结构体等都是可以的。
- 加法函数的返回值是什么类型的？我们知道，加法函数是要对两个整数进行加法运算，两个整数相加，所得的结果也是一个整数，所以就定义返回值的类型为整型 `int`。

确定了加法函数的函数名、形参、返回值类型这三部分之后，就可以按照函数声明定

义的形式来确定加法函数到底长什么样了，如下所示：

```
int add(int a, int b)
```

参数列表位于小括号中，每个参数变量之间用逗号隔开，最后一个参数变量之后不需要逗号。

11.4.2 实现加法函数体

确定了加法函数的样子之后，接下来要告诉计算机，函数该如何完成需要的加法功能，也就是实现函数定义的函数体。C 语言中的函数体有三个部分：声明部分、实现部分和返回值语句。

只要完成了加法函数的这三个部分，就算实现加法函数了。来一个一个地实现吧！

- ❑ 函数的声明部分。对于加法函数，先来考虑要不要使用变量来保存加法运算的中间和最终结果，如果需要就声明定义一些变量来保存这些结果。这里还是用一个整型变量 `result` 来保存加法运算的最终结果吧。
- ❑ 函数的实现部分。由于加法函数的声明中已经提供了两个整型的形参变量 `int a` 和 `int b`，因此可以直接从这两个变量中取得两个加数。然后，再使用 C 语言中的加法运算表达式语句完成两个加数的相加。最后，将加法表达式直接赋值给 `result` 来保存加法运算的最终结果。具体表示如下所示：

```
result = a+b;
```

- ❑ 函数的返回值语句。对于加法函数，需要返回的是加法运算的最终结果，而这个结果是保存在 `result` 中的，所以直接使用 `return` 返回值语句返回 `result` 就可以了。具体表示如下所示：

```
return result;
```

11.4.3 完整的加法函数定义

有了加法函数的样子（声明）和加法函数的函数体（定义），现在就可以完成整个加法函数的声明和定义了，将前面讲的组合起来就行。下面就是需要的加法函数的声明定义，这里声明和定义是在一起的。

```
//函数的样子
int add(int a, int b)
{
    //函数的声明部分
    int result;

    //函数的实现部分
    result = a + b;

    //函数的返回值语句
    return result;
}
```

上面就是一个完整的符合 C 语言要求的标准的加法函数，有了这个函数的声明定义，就相当于玉石已经成功打造一个 C 语言中的“玉珠子”了。对于其他的“玉珠子”也可以使用类似的方法进行打造。

11.5 函数调用

当声明定义好一个函数以后，就可以使用这个函数来完成需要的功能了。如果要完成的功能比较复杂，可能需要多个函数，这个时候需要把多个函数串起来，该怎么“串”呢？需要用到本节讲到的知识——函数调用。

11.5.1 函数的调用作用

在做手链的时候，先将玉石打磨成一个一个小玉珠子，然后用针和线将这些小玉珠子串起来。我们写一个复杂的或者大型的程序时也是类似的，当把程序划分成一个个可以重复使用的小功能以后，就相当于设计出了不同大小的“珠子”。

设计好了之后就是打磨了。我们使用 C 语言中提供的原材料声明定义一个函数，就相当于打磨好了一个珠子。与打磨珠子不同的是，C 语言中的同一个函数只能定义一次，而对于玉珠子，无论是不是大小一样，需要几个就得打磨几个。不过，C 语言中的函数声明可以有多个。

打磨好了之后就可以把函数串起来了，串起珠子需要的是针和线，串起函数需要的是什么呢？在 C 语言中，使用函数调用来把函数串起来。当然，有多个函数时使用函数调用串起函数，有一个函数时也是使用函数调用串起函数的，只有一个珠子的手链应该见过吧？

图 11.6 就是函数调用和针线的一个形象的类比，串起玉珠子使用的是针和线，串起函数需要的是 C 语言中一个很重要的概念——函数调用，这也是函数调用在 C 语言中起到的唯一作用。

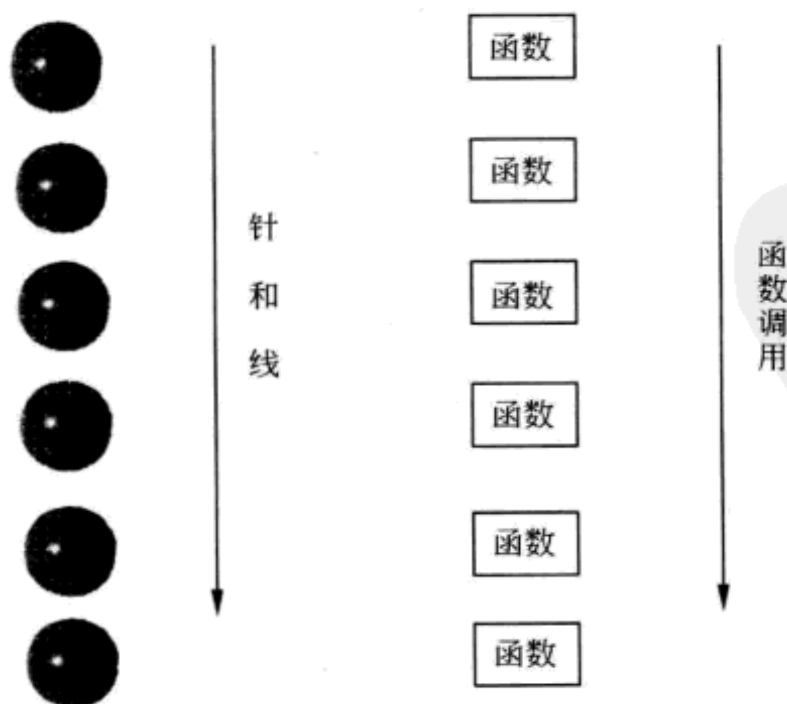


图 11.6 函数调用的作用

11.5.2 函数的调用表达式

我们已经知道了，函数调用的作用是把 C 语言中的函数串起来组成一个完整的程序。有了这个理论基础，下面来看一看 C 语言中的函数到底是怎么调用的，也就是 C 语言中的函数调用到底是什么样子的。

C 语言中的函数调用是通过表达式来实现的，可以类比之前讲到的数学运算表达式进行学习，其实它们有着相似的特点：都有一定的格式，都是要完成一个功能，都可以加一个分号（;）形成一个可以独立执行的语句。C 语言中的函数调用表达式的形式如下所示：

```
函数名(数据 1, 数据 2, 数据 3, ..., 数据 n-1, 数据 n)
```

这个表达式中，“函数名”就是声明定义函数时给函数所取的名字，是一个 C 语言标识符。一对小括号是 C 语言规定的必不可少的符号，就像加法运算表达式中的“+”号一样。小括号之中的 n 个数据，可以是常量、变量、甚至是表达式，只要有值就行。这些值将会被放到函数声明的参数列表中对应的参数变量里，给函数定义使用，所以函数调用中的数据个数应该和函数声明中的参数变量个数是相等的。其实，这个表达式很像函数的声明，只是少了函数返回值，参数列表变成了数据列表。

例如，要使用加法函数完成 4 加 5，就可以使用下面的函数调用表达式。

```
add(4, 5)
```

在这里，`add` 是已经定义的加法函数的名字，4 和 5 是调用加法函数所需要的数据列表，它们会被分别保存在整型变量 `a` 和 `b` 中，供加法函数的定义使用。

11.5.3 函数的实参

函数调用表达式中的数据列表，在 C 语言中有专门的术语，被称为实际参数，简称实参。这些参数就是我们提供给函数的实际数据，供函数完成具体的功能。

这些实参数据会在函数调用的时候，传递给函数声明定义中的形参变量，最后经由形参变量传递给函数体，完成我们所需要的功能。例如，上面 `add(4,5)` 加法函数调用表达式将会完成图 11.7 所示的数据传输过程。

在图 11.7 中，`add(4,5)` 函数调用语句在执行的时候，先将实参数据 5 传递给形参变量 `b`，也就是将整型常量 5 保存到整型变量 `b` 中，同样将实参数据 4 传递给形参变量 `a`。具体先传 5 还是先传 4，是可以设置的，一般情况是先传 5。之后，`a` 和 `b` 这两个整型参数会参与加法运算表达式的运算过程，将得到的结果保存在 `result` 整型变量中。最后，`return` 语句会将 `result` 的值返回，退出函数的执行，就相当于“针线”从玉珠子（加法函数）中穿了出来。

加法函数中的 `return` 语句将 `result` 的值返回了，返回到哪去了？我们怎么得到这个返回的结果呢？回答这个问题之前，先来回顾 C 语言中这样一个规定：所有的表达式都是有一个值的。函数调用也是一个表达式，它的值是什么呢？也许你猜到了，函数调用表达式的值就是 `return` 语句的返回值！也就是说表达式 `add(4,5)` 的值，就是计算完成之后 `result` 整型变量中保存的值，也就是 9。

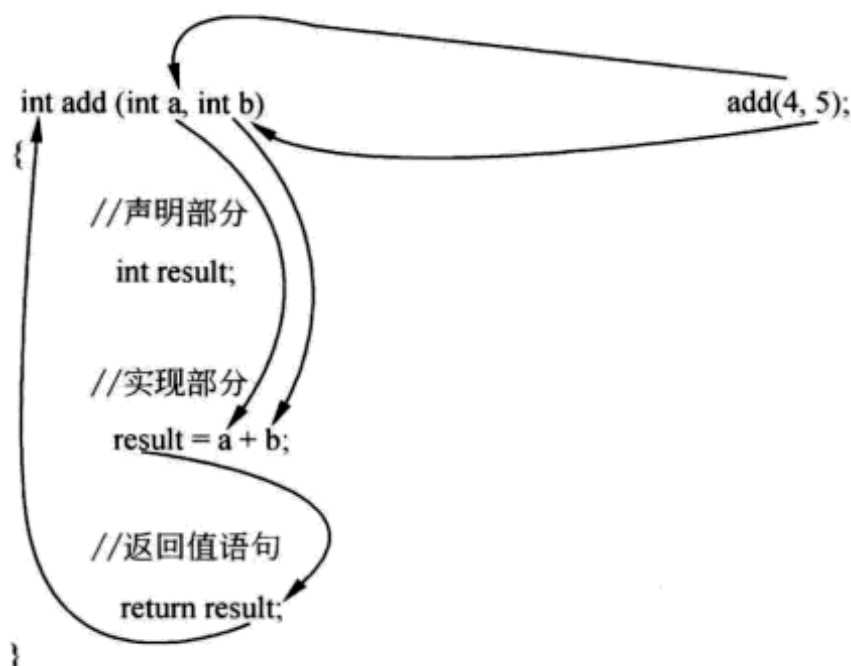


图 11.7 加法函数的参数传递过程

11.5.4 简单函数的调用

到现在为止已经知道了如何声明定义一个函数，以及如何使用一个已经声明定义的函数，就相当于我们已经知道如何将玉石打磨成一颗颗的玉珠子，又知道如何将这些玉珠子串起来，接下来的事情就是该串出一条想要的玉手链了。

来看一个很简单的例子，使用加法函数完成 $4+5+6$ 。先来分析一下这个要求： $4+5+6$ 是三个数值相加，加法函数是两个数相加。所以，一个加法函数不能完成这个要求，得需要两个加法函数才能完成。一个加法函数完成 4 和 5 的相加，另一个函数完成 4 和 5 相加的结果与 6 的相加。现在需要将这两个形状一样的函数（珠子）串起来才能完成要求。具体怎么串，且看下面的程序。

```

#include <stdio.h>

//函数声明
int add(int a, int b);

int main()
{
    int temp;

    temp = add(4, 5);
    printf("4+5=%d\n", temp);

    temp = add(temp, 6);
    printf("4+5+6=%d\n", temp);

    return 0;
}

//函数定义
int add(int a, int b)
{
    //函数的声明部分
    int result;

```

```
//函数的实现部分
result = a + b;

//函数的返回值语句
return result;
}
```

在这个程序中，先对 add 加法函数进行了声明，然后在最后对 add()函数进行了定义。之所以要将这里对 add()函数的声明和定义分开，是因为 add()函数的定义在调用之后，调用的时候不知道 add 为何物，所以在调用之前对函数 add()进行了声明，告诉计算机 add 是一个函数，可以安全使用。

这里对 add()函数进行了两次调用，第一次对 4 和 5 进行相加，将相加的结果保存到 temp 整型变量之中。然后再对 temp 和 6 进行相加，将计算的结果保存在 temp 整型变量中，第二次保存会覆盖第一次保存的结果。同时将计算的结果进行输出，如图 11.8 所示，如我们所料，将两个加法函数串起来正好完成了 4+5+6 的运算。



图 11.8 函数调用程序验证

11.6 复杂参数

在声明定义函数的时候，形参列表都是一些变量的定义。既然只要是变量就可以，那么就不局限变量的类型了，可以是简单类型的变量，也可以是复杂类型的变量，如结构体、共同体和枚举变量，甚至数组和指针变量都是可以的。暂且称这些复杂变量的函数参数为复杂参数。

11.6.1 数组参数

使用简单数据类型的参数来给函数传递数据就已经足够了，那么，为什么还需要数组来作为参数呢？可以设想这样一个情景，假设要算出两个学生的平均成绩，使用前面定义的有两个参数的加法函数就可以了。如果要计算 100 个学生的成绩，需要设计一个拥有 100

个整型参数的加法函数吗？假设算 1000 个、10000 个、一百万个，甚至更多学生的成绩，即使计算机能承受这么多的参数，你愿意写吗？所以，干脆使用数组作为参数吧！

1. 数组参数的函数声明定义

使用数组形参来声明定义函数和定义声明函数的一般形式是一样的。把需要使用数组作为形参的变量改为数组变量就可以了，具体形式如下：

返回值类型 函数名(数据类型 数组名[表达式])

这个函数声明形式和前面讲到的函数声明形式唯一不同的是，将其中的形参列表确定为一个一维数组变量了，当然也可以是更高维的数组变量。当然需要多个数组作为参数也是可以的，其形式如下：

返回值类型 函数名(数据类型 数组 1 名[表达式], 数据类型 数组名 2[表达式], ..., 数据类型 数组 n 名[表达式])

只要将形参列表确定为你需要的多个数组，之间使用逗号隔开就可以了。C 语言中规定，当函数的参数是数组的时候，数组中最后一维的“表达式”可以省略，也就是上面的两个表示形式可以简化为以下的形式。

返回值类型 函数名(数据类型 数组名[])

和

返回值类型 函数名(数据类型 数组 1 名[], 数据类型 数组名 2[], ..., 数据类型 数组 n 名[])

假设现在要声明定义一个计算 20 个整数之和的函数，就可以使用下面两种函数声明定义形式中的任何一种：

```
int add( int nums[20] )
```

和

```
int add( int nums[ ] )
```

2. 数组参数的函数调用

当调用这些以数组作为参数的函数时，要遵循一般的函数调用的形式和规则。唯一特殊的是函数调用时，传入的数据是已经保存了数据的数组的数组名，也就是数组首地址，是一个指针值。其形式如下：

函数名(数组名);

例如，要使用上面计算 20 个整型数据之和的函数，来计算保存在 `int scores[20]` 中的 20 个学生的成绩的总和时，就可以使用下面的函数调用。

```
add(scores);
```

3. 计算 20 人的平均成绩

下面是使用数组作为参数计算 20 个整型数据之和的函数，以及使用这个函数计算 20

个学生的平均成绩的完整程序。

```
#include <stdio.h>

//函数声明
int add(int nums[ ]);

int main()
{
    int scores[20] = { 80, 90, 85, 91, 88, 70, 87, 88, 92, 98,
                      77, 80, 100, 90, 68, 88, 76, 91, 86, 93};

    double average;
    average = add(scores)/20.0;

    printf("the average score of 20 students is : %lf\n", average);

    return 0;
}

//函数定义
int add(int nums[20])
{
    int i;
    int sum = 0;

    for(i=0; i<20; i++)
    {
        sum += nums[i];
    }

    return sum;
}
```

在这个程序中，使用数组参数中没有“表达式”的形式进行加法函数的声明，在函数定义中使用有“表达式”的加法函数表示。所以，当函数的参数是数组的时候，有没有表示数组大小的“表达式”都是可以的。C语言根本不关心这个数组到底有多大，到底数组有多大得你自己清楚。如果传入的数组和你在函数定义中使用的数组的大小不一致就会出现错误，甚至导致程序崩溃。

add()函数定义中，使用了一个for循环结构对数组中的20个整型数据进行求和，并将所求的和保存到整型变量sum中，最后将sum的值返回。

我们使用add()函数计算20个学生的平均成绩时，将已经保存的20个学生的成绩的整型数组名scores传入add()函数的调用。然后，将add()函数调用表达式除以20.0，之所以使用20.0作为除数，是因为整型数据除以整型数据会导致舍尾，丧失精度，使用浮点数作为除数会导致隐式的类型转换，从而得到一个更精确的平均成绩。最后，输出保存在双精度浮点型变量average中的平均成绩。

程序的输出如图11.9所示，本次考试成绩为85.9，挺不错的嘛！

11.6.2 指针参数

在用数组作为函数的参数的时候，传入的是数组名，也就是数组的首地址。那么，为

什么不直接使用指针变量作为函数的参数呢？当然是可以的，当需要给函数传入一大块数据的时候，直接传入这块数据的首地址确实是一个不错的选择！

```
#include <stdio.h>

//函数声明
int add(int nums[]);

int main()
{
    int scores[20] = { 88, 98, 85, 91, 88, 78, 87, 88, 92, 98,
                      77, 80, 100, 90, 68, 88, 76, 91, 86, 93};

    double average;
    average = add(scores)/20.0;

    printf("the average score of 20 students is : %1f\n", average);

    return 0;
}

//函数定义
int add(int nums[20])
{
    int i;
    int sum = 0;

    for(i=0; i<20; i++)
    {
        sum += nums[i];
    }

    return sum;
}
```

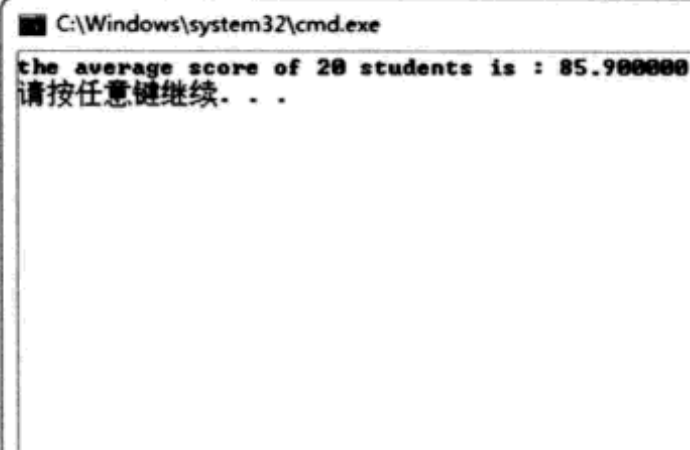


图 11.9 使用数组作为函数的参数

1. 指针参数函数的声明定义

在声明定义函数的时候，如果使用指针作为参数，其形式和使用基本类型作为参数是一样的，只需要将参数列表中需要使用指针类型的参数确定为指针变量就可以了。例如，声明定义需要一个指针参数的函数，使用下面的形式。

返回值类型 函数名 (数据类型* 指针变量)

这里的指针变量是一个一重指针变量，当然，可以是任意重的指针变量。这里简单地以一重指针变量为例，对其他重的指针变量，参见多重指针变量的定义。

例如，要计算 20 个整型数据之和，就可以定义一个使用整型指针变量作为参数的 add() 函数。其形式如下所示：

```
int add(int * p_num)
```

2. 指针参数函数的调用

要使用以指针作为参数的函数时，直接调用这个函数就可以了。调用的时候可以传入指针数值或者已经保存了指针数值的指针变量。其形式如下所示：

函数名 (指针);

这个函数调用语句形式中，指针就是传入的参数，可以是指针数值（数组名、取值运算得到的指针值等），也可以是已经保存了地址的指针变量。如果传入的是没有保存指针

数值的指针变量，很有可能会导致错误甚至程序崩溃，所以传入的是什么都自己清楚。

例如，有一个函数名为 `fun()`，需要一个整型指针作为参数，就可以使用下面任意一种方式去调用这个函数。

❑ 传入数组名这个数组首地址：

```
int num[10];
fun(num);
```

❑ 使用取址运算得到变量地址，将这个地址传入函数 `fun()`：

```
int num;
fun(&num);
```

❑ 传入已经保存了地址的指针变量：

```
int num;
int* p_num;

p_num = &num;
fun(p_num);
```

3. 重写 `add()` 函数计算平均成绩

既然指针也可以作为函数的参数传给函数使用，那么，就可以使用传入指针的方式来重写 `add()` 函数，从而完成对 20 个学生平均成绩的计算。完整的程序如下所示：

```
#include <stdio.h>

//函数声明
int add(int* p_num);

int main()
{
    int scores[20] = { 80, 90, 85, 91, 88, 70, 87, 88, 92, 98,
                      77, 80, 100, 90, 68, 88, 76, 91, 86, 93};

    double average;
    average = add(scores)/20.0;

    printf("the average score of 20 students is : %lf\n", average);

    return 0;
}

//函数定义
int add(int* p_num)
{
    int i;
    int sum = 0;

    for(i=0; i<20; i++)
    {
        sum += *p_num++;
    }

    return sum;
}
```


和前面使用数组作为 add() 函数参数的那个程序相比, 改动的地方除了将 add() 函数的参数改为指针, 剩下的就是在 add() 函数定义中, 使用指针来访问成绩数组中的元素, 对它们进行相加。在取成绩数组元素时, 使用指针变量自加获得每个数组元素的地址, 然后使用取指针元素运算得到该地址处的成绩, 最后, 使用 for 循环对所有成绩进行求和。

程序的输出如图 11.10 所示, 和上面使用数组作为参数的程序输出是一样的, 20 个人的平均成绩还是 85.9。



图 11.10 使用指针作为函数参数

11.6.3 结构体、共同体和枚举参数

有的时候, 仅仅以数组或指针作为函数的参数是远远不够的。因为数组和基本类型指针都有缺点。数组只能保存同样类型的数据, 而基本类型指针只能访问一个地址中的数据或者多个连续地址中的数据。要是定义函数的时候, 需要不同数据类型的数据, 或者需要的数据在不连续的地址中, 该怎么办呢? 这个时候, 还得结构体、共同体和枚举类型出马, 把这三种类型或者它们的指针作为函数的参数都是可以解决上面的问题的!

结构体、共同体和枚举作为参数的 C 语言表示形式和使用方式都是类似的, 不同的只是这几种数据类型的含义, 暂且只以结构体为例。

1. 结构体参数的函数声明定义

使用结构体类型作为函数声明定义的参数的 C 语言表示形式并没有什么特殊之处, 只要将参数列表中需要结构体类型的参数, 确定为结构体变量或者结构体指针变量就可以了。这样, 进行函数定义的时候, 就可以使用结构体数据了。

C 语言中, 使用结构体作为函数参数的声明定义形式如下所示。假设函数只有一个结构体参数。

返回值类型 函数名 (结构体类型 结构体变量)

这个函数声明定义中, 参数列表是一个结构体变量的定义。其中, “结构体类型”是一个已经定义的结构体类型, 如 struct student, 其中的 struct 不可少。或者是使用经 typedef

重新定义的类型名，也可以使用结构体指针作为参数来用地址传给函数结构体数据，其形式如下所示。

返回值类型 函数名 (结构体类型* 结构体变量)

这里定义的是一个一重结构体指针，如果需要也可以定义一个多重结构体指针。

例如，已经定义了一个结构体类型 `struct student`，想把一个 `struct student` 类型的数据传入函数 `fun()` 中完成想要的功能，就可以使用下面的函数声明定义，假设不需要返回值。

```
void fun(struct student boy)
```

或者使用结构体指针也可以。

```
void fun(struct student* boy)
```

2. 结构体参数的函数调用

调用使用结构体作为参数的函数也是很简单的，只要按照一般的函数调用形式，将结构体数据（一般都是已经保存数据的结构体变量）传入就可以了。

函数名 (结构体数据)

例如，要调用上面声明定义的 `fun()` 函数，可以使用下面的 C 语言表示。

```
struct student boy = {数据 1, 数据 2, ..., 数据 n-1, 数据 n};
fun(boy);
```

3. 功能更强大的add()函数

有了结构体可以作为函数的参数的理论基础以后，就可以写一个功能更强大的 `add()` 求和函数了，这个函数无论计算多少个整型数据的和都是可以的。为了做到可以计算任意多个整型数据之和，先得定义一个结构体 `scores`，如下所示：

```
struct stu_scores
{
    int* p_num;
    int len;
};
```

这个结构体中的 `p_num` 指针成员变量用来保存所有整型数据的首地址，`len` 整型成员变量用来保存学生的个数。如果用这个结构体作为 `add()` 函数的参数，就可以计算任意多个整型数据的和，不仅仅限于 20 个。完整的程序如下所示：

```
#include <stdio.h>

struct stu_scores
{
    int* p_num;
    int len;
};

//函数声明
int add(struct stu_scores students);

int main()
```

```

{
    int scores[20] = { 80, 90, 85, 91, 88, 70, 87, 88, 92, 98,
                      77, 80, 100, 90, 68, 88, 76, 91, 86, 93};

    double average;
    struct stu_scores students;

    students.p_num = scores;
    students.len = 20;

    average = add(students)/20.0;

    printf("the average score of 20 students is : %lf\n", average);

    return 0;
}

//函数定义
int add(struct stu_scores students)
{
    int i;
    int sum = 0;

    for(i=0; i<students.len; i++)
    {
        sum += *students.p_num++;
    }

    return sum;
}

```

这一次，先定义了一个 struct stu_scores 结构体类型，之所以将结构体类型定义放到最前面，因为它会在很多地方用到，所以得在它之前让别人知道 struct stu_scores 为何物。定义完结构体 struct stu_scores 之后，重新声明定义 add() 函数，这次使用的参数是 struct stu_scores 结构体。在定义 add() 函数的时候，使用结构体变量 students 的 len 成员变量作为 for 循环的循环次数，从而实现计算任意个整数的和。

在调用 add() 函数的时候，我们使用一个已经保存了数据的 struct stu_scores 类型变量作为实参。程序的输出如图 11.11 所示，跟前面的一样，平均成绩还是 85.9。

```

//函数声明
int add(struct stu_scores students);

int main()
{
    int scores[20] = { 80, 90, 85, 91, 88, 70, 87, 88, 92, 98,
                      77, 80, 100, 90, 68, 88, 76, 91, 86, 93};

    double average;
    struct stu_scores students;

    students.p_num = scores;
    students.len = 20;

    average = add(students)/20.0;

    printf("the average score of 20 students is : %lf\n", average);

    return 0;
}

//函数定义
int add(struct stu_scores students)
{
    int i;
    int sum = 0;

    for(i=0; i<students.len; i++)
    {
        sum += *students.p_num++;
    }

    return sum;
}

```

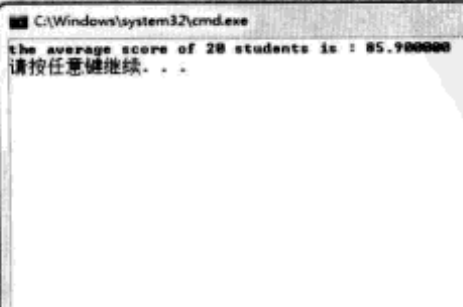


图 11.11 使用结构体作为函数参数

11.7 小 结

本章中学习了 C 语言中一个很重要的东西——函数，它是完成大型功能的基础。本章的重点是函数的声明、定义和调用的概念和使用，难点是如何定义一个函数，函数调用中各种参数的使用，以及数据在函数中从参数到返回值是如何传递的。在下一章中将学习 C 语言中一个特殊的函数——main()函数，C 语言程序都是从这个函数开始执行的。

11.8 习 题

【题目 1】 不使用函数，可不可以写 C 语言代码完成需要的功能？

【分析】 函数只是对 C 语言基本元素的一种组织方式，是一种程序设计方法，它将程序分为一块一块的，并提供了函数调用来将这一块块的代码连接起来，从而组织成一个完整的程序。当我们遇到一个很复杂的问题时，一般很难一下子理解透彻，往往采用的方式就是各个击破，将一个一个小问题解决，最终就可以完全解决整个问题。这个思想就是我们使用函数的初衷完全可以在程序中不使用函数，只要你能一下子理解透彻问题，能很轻易地写一段，或者很长一段代码实现要解决的问题！

【题目 2】 我们知道，函数只能返回一个值，如果希望函数返回值多于一个，该怎么办？假设需要返回一个整型变量 a 中的内容和字符型变量 b 中的内容，写一段程序解决这个问题。

【分析】 在 C 语言中，要想返回多个数值，往往可以有两个思路：（1）将多个值组合成一个值，然后返回；（2）使用函数的参数返回数值。

对于第一种方式，怎么将多个值组合成一个值呢？别忘了，C 语言中有高级数据类型的，它们是可以放置多个值的，可以使用高级数据类型来将多个值组合成一个值。也就是在函数的返回值类型中使用高级数据类型，一般都是返回一个结构体，因为它足够灵活。

对于第二种方式，怎么使用函数的参数返回多个值呢？函数的参数中是可以传入指针的，也就是我们可以告诉函数，在内存中有一块地方可以使用，让函数把要放的东西放里面就可以了。可以在函数的参数中传入几个指针，然后将函数要返回的多个值放到这几个指针所指的内存地址就可以了。

回到问题，方法一：定义一个可以放置整型数据和字符型数据的结构体类型，作为函数的返回值；方法二：在函数的参数中传入两个指针，一个用于放置整型数据，另一个用于放置字符型数据。

【核心代码】

```
//方法一：返回结构体数据
struct type_name
{
    int    num;
    char  c;
};
```



```

struct type_name fun()
{
    struct type_name data;
    int a;
    char b;
    ...
    data.num = a;
    data.c = b;

    return data;
}

```

```

//方法二：使用参数返回数据
void fun(int* p_num, int* p_c)
{
    int a;
    char b;
    ...
    *p_num = a;
    *p_c = b;
}

```

【题目 3】 如果在一个函数中调用自己，会出现什么情况？写段代码试验一下，试着解释出现这种结果是为什么！

【分析】 在一个函数中调用自己，会出现什么情况？这得从函数调用是干什么的开始分析。函数调用是将需要的函数连接起来，这是函数调用宏观上的功能。从更细微的角度来说，函数调用的作用就是从函数的第一条语句开始一直执行到最后一条，或者直到遇到 `return` 语句。

接下来就可以分析在一个函数中调用自己会发生什么情况了。当定义了一个自己调用自己的函数后，这个函数的调用语句执行时，会从这个函数的第一条语句开始执行，直到遇到这个函数中自己调用自己的那条语句，使执行逻辑又回到这个函数的第一条语句再开始执行，如此往复。执行的结果有点像一个永无止境的循环，遇到这个函数调用，就从头开始！

【核心代码】

```

void fun()
{
    printf("test fun call!\n");
    fun();
}

int main()
{
    ...
    fun();
    ...
}

```

【题目 4】 对于题目 3 中所述的函数调用方式，在 C 语言中有个专门的术语，叫做函数的递归调用，而且经常会用到。从题目 3 中，我们知道，这种递归调用方式会引起永无止境的循环执行。现在的问题就是写一段程序避免这种永无止境的循环执行，会有点难，but have a try！

【分析】 要解决这个问题，首先得清楚函数调用结束的条件：要想使一个函数终止执行，或者让函数执行完其中所有的语句，或者执行到 `return` 语句。对于递归调用的函数，让函数执行完所有的语句是不可能的，它老回退到开头，所以得使用 `return` 语句，该怎么使用呢？这要看你的目的了，不过一般是在函数执行到一定条件的时候，使用 `return` 语句，否则继续调用这个函数。可以使用 `if-else` 程序结构来实现。

例如，写一段程序，当函数传入的参数为 10 的时候，就结束函数的执行。否则，将函数传入的参数自增，再将其作为函数的实参进行函数递归调用，这样递归下去直到函数的参数为 10 的时候，函数自然会终止退出。

【核心代码】

```
int fun(int a)
{
    printf("a = %d\n",a);
    if(a == 10)
    {
        return 0;
    }
    else
    {
        a++;
        fun(a);
    }
    return 1;
}
```

【题目 5】 杨辉三角：数学中由个有数字组成的三角形，被称为杨辉三角形，它的样子如图 11.12 所示，写程序在屏幕上显示这个由数字组成的三角形。

```

      1
    1 1
  1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
```

图 11.12 杨辉三角形

【分析】 从这个三角形的形式上，可以发现这样一个规则，每个数都是由其肩上的两个数的和构成的，如果某个肩上没有数字，就补 1。可以用下面的公式来表示这个三角形上的数的形成规则。

$$c(x,y)=\begin{cases} 1 & x=1\text{或}x=N+1 \\ c(x-1,y-1)+c(x-1,y) & \text{其他} \end{cases}$$

要写代码来生成这个三角形，关键就是实现上面这个公式中的 $c(x,y)$ ，可以使用函数

的定义来实现这个公式中的 $c(x,y)$ 。可以看出这个函数实现的时候，会自己用到自己，也就是会涉及递归函数调用，这一点需要注意一下。

【核心代码】

```
#include<stdio.h>

//函数声明
int c(int x, int y);

int main()
{
    int i,j,n=13;

    printf("N="); //表示这个三角形总共几层
    while(n>12)
        scanf("%d", &n);
    for(i=0;i<n;i++)
    {
        for(j=0; j<12-i;j++) printf(" ");
        for(j=1; j<i+2;j++) printf("%6d", c(i,j));
        printf("\n");
    }

    return 0;

    int c(int x, int y) //定义三角形上的数值的计算函数
    {
        int z ;
        if ( (y==1) || (y== x+1)) return 1; //保证递归调用不会无止境

        z = c(x-1,y-1) + c(x-1,y); //函数的递归调用
        return z;
    }
}
```

第 12 章 特殊的函数——main()函数

在制作玉手链的时候，我们使用针线将所需要的珠子串起来，然后将线的两端打成结就可以完成玉手链的制作了，最后将其戴在手上。在写一个大型程序的时候，使用函数调用把所需要的函数串起来，但是如何将串起来的函数收尾“打成结”供计算机使用呢？C 语言提供了一个 main()函数来完成这件事情。本节就来看一看 C 语言中的 main()函数！

12.1 main()函数的作用

C 语言提供了一个很特殊的函数——main()函数。之所以说它特殊，是因为它的作用是“统领”其他自定义函数的，其他函数都必须在它的控制下才能使用。英文单词 main 的意思为“主要的”，所以，main()函数也被称为主函数。

当需要用 C 语言完成一个复杂的功能时，先将这个复杂的功能划分成许多小的功能。然后使用函数声明定义实现这些小的功能，最后使用函数调用，将这些函数“串”起来，来完成复杂的功能。在“串”函数的时候，就会出现如下两个问题。（1）谁来调用第一个函数，也就是这一串函数的头交给谁？（2）谁来接函数的尾，也就是这一串函数的尾交给谁？

交给 main()函数！main()函数会抓起这一串函数的头，拉起这一串函数的尾，然后将其打个结交给计算机，这样你写的程序就会被计算机识别。main()函数就是你写的程序与计算机融合的一个函数，只要将你的函数调用放到 main()函数的定义中就可以实现这一融合。当然其中肯定还是会穿插非函数调用的语句用来实现一些简单的补充功能的。

这一融合过程可以用图 12.1 所示的示例来说明，图 12.1 中 main()函数定义的函数体中就是一连串的函数调用。其中，也可以包含其他的语句来完成一些简单的补充功能。当然，实现功能比较简单的时候，也可以不需要定义函数。这个时候 main()函数除了一些简单的 C 语言语句和程序结构之外，没有任何的函数调用语句，我们之前写的程序例子大多都是这种情况。

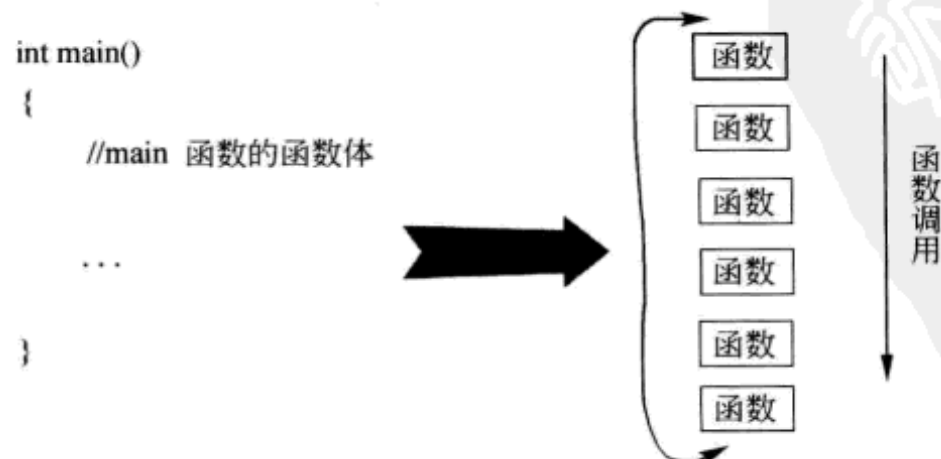


图 12.1 main()函数的作用示例

12.2 main()函数的声明定义

main()函数是 C 语言规定的函数，对于它的形式是有要求的，主要体现在 main()函数的样子上，也就是 main()函数的函数名、参数和返回值类型上。

12.2.1 main()函数的声明形式

C 语言中的 main()函数的声明可以有好几种形式，分别满足不同的 C 语言标准要求。最常使用的主要有以下 4 种。

□ 无返回值无参数：

```
void main()
```

□ 无返回值有参数：

```
void main(int argc, char* argv[])
```

□ 有返回值无参数：

```
int main()
```

□ 有返回值有参数：

```
int main(int argc, char* argv[])
```

这 4 种形式的区别主要在于是不是有返回值和参数上。如果有返回值，返回值类型必须是 int。如果无返回值，返回值类型是 void。如果有参数，参数必须是一个 int 类型的变量和一个一维数组，其中保存的是字符指针，变量名要求一个是 argc，另一个是 argv。当然，也可以使用其他变量名，不过最好按照要求来，使用 argc 和 argv 这两个变量名，以免特殊情况。如果没有参数，就什么都不写了。

之前写的程序都是遵循第三种形式的，有返回值没有参数。对于这 4 种形式，最新的标准要求为最后两个，可以没有参数，但必须有返回值。现在所有的编译器基本上都支持最后两种 main()函数声明形式，以免出现问题，建议大家最好使用有返回值的 main()函数形式。

12.2.2 main()函数的参数

之前使用的都是没有参数的 main()函数形式，接下来主要看看带参数的 main()函数形式，以及它的两个参数的意义和使用。

1. argc参数

int argc 是 main()函数的第一个参数。它是一个整型变量，这个变量是系统在运行程序的时候使用的。它代表的是在用命令行运行程序时，输入的命令中包含的字符串的个数。

例如，图 12.2 中在终端输入的命令：“test.exe xiaoming lihua”，这个命令的作用是运行“E:\c 语言的书\project\test\debug”目录下的程序 test.exe，并且输入两个字符串“xiaoming”和“lihua”。这个命令输入完，按回车键，计算机就会给 argc 传入一个整数 3，为什么是 3 呢？这是因为输入的命令按照空格进行分隔，有三个字符串：“test.exe”、“xiaoming”和“lihua”。

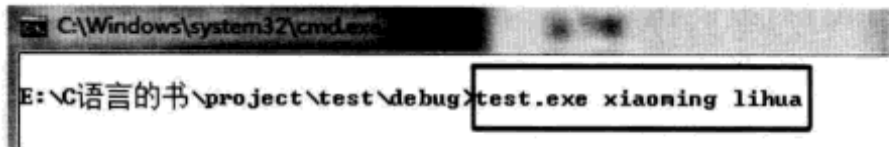


图 12.2 命令行运行程序

总而言之，argc 这个参数是用来保存运行程序时，输入的命令以空格分割的字符串的个数。这个数字是系统运行程序的时候传给 main()函数的，我们不需要管，也管不了。但是，可以肯定这个数字一定是正确地表示命令中以空格分割的字符串的个数。

2. argv参数

系统在运行程序的时候，光传入命令中有几个字符串是根本不够的，还得告诉程序，命令中的字符串都是什么，不然程序就没法使用用户输入的命令！计算机系统就是使用 argv 这个一维数组来告诉程序，用户输入的命令中的字符串都是什么。

argv 是一个一维数组，其中保存的是命令中的每个字符串在内存中的地址。通过这个地址就可以知道用户输入命令中的字符串。对数组进行遍历，就可以一次访问到第二个、第三个、…、第 argc 个字符串了。例如，图 12.3 就展示了图 12.2 中输入命令以后，argv 数组中保存的地址，以及这些地址对应内存中保存的字符串内容。

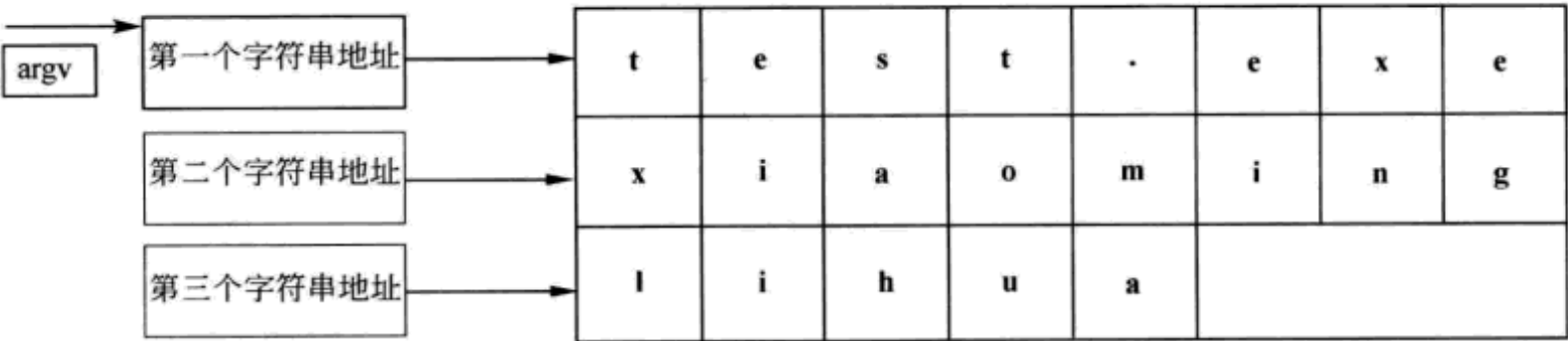


图 12.3 argv 参数的内容

可以使用带参数的 main()函数，写一个 test.exe 程序，在这个程序中只输出 argc 的值和 argv 保存的地址处的字符串。完整的程序如下所示：

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;

    printf("argc = %d\n",argc);

    for(i=0; i<argc; i++)
    {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
}
```

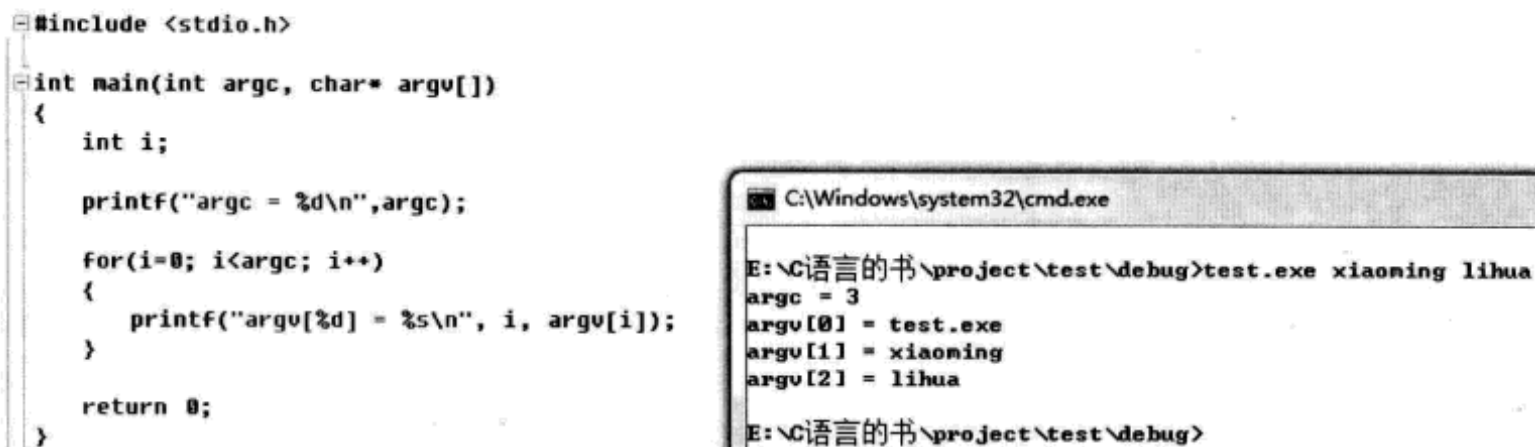
```

    }

    return 0;
}

```

这个程序很简单，就是简单的一些输出。先输出参数 `argc` 的值，之后使用 `for` 循环输出 `argv` 数组中每个字符串地址中保存的字符串的内容。程序的输出如图 12.4 所示，程序正确输出了我们运行程序时输入命令中的字符串。



```

#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;

    printf("argc = %d\n",argc);

    for(i=0; i<argc; i++)
    {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}

```

```

C:\Windows\system32\cmd.exe

E:\C语言的书\project\test\debug>test.exe xiaoming lihua
argc = 3
argv[0] = test.exe
argv[1] = xiaoming
argv[2] = lihua
E:\C语言的书\project\test\debug>

```

图 12.4 使用带参 `main()` 函数实现 `test` 程序

12.2.3 `main()` 函数的返回值

按照 C 语言中 `main()` 函数的 4 种形式，`main()` 函数的返回值类型要么为空，要么为 `int` 整型。对于返回值为空的 `main()` 函数，一般不推荐，如果非要使用这种形式，可以不用管 `main()` 函数的返回值。如果使用以 `int` 类型作为返回值的 `main()` 函数形式，一般 `main()` 函数能够正常完成需要的功能则返回值 0，否则返回其他值。

通过上面的介绍，推荐的 `main()` 函数的完整使用形式有如下两种。

❑ 不带参数形式，其中参数 `void` 可以省略不写。

```

int main(void)
{
    ...

    return 0;
}

```

❑ 带参数形式。

```

int main( int argc, char *argv[])
{
    ...

    return 0;
}

```

写程序的时候，就按照这两种方式来写 `main()` 函数。如果在程序运行的时候传入数据，就使用第一种形式，否则使用第二种形式。`main()` 函数的函数体中实现对其他函数的调用，通过函数调用及其他语句来完成我们想要计算机完成的功能。具体使用方式就像上一节函

数调用中的例子一样！

12.3 小 结

本章中学习了 C 语言中一个重要的函数——`main()`函数，我们写的 C 语言代码都是从这个函数开始执行的。本章的重点是 `main()`函数的声明定义、参数和返回值的形式，难点是带参 `main()`函数的参数的含义和使用。下一章将详细地介绍一下位于函数内外和其他不同位置定义的变量的作用域和生命周期。

12.4 习 题

【题目 1】 我们写程序的时候，可不可以不需要 `main()`函数，直接写自己的函数？

【分析】 大家要知道 `main()`函数真正是做什么用的。先来回顾一下一个可执行应用程序的生成过程，使用编辑器编写代码，使用编译器对每个源文件进行编译，使用链接器生成可执行程序。`main()`函数会在哪一步真正起作用呢？是在链接器生成可执行程序的时候真正起作用的，操作系统会从 `main()`函数开始的地方执行我们所写的代码，这是 C 语言规定的。

如果我们写的代码不需要运行在操作系统上，而是直接交给硬件的，或者程序从什么地方开始执行，自己可以随便设置，就不必使用 `main()`函数了。换句话说，如果不是生成运行在某个操作系统上的可执行程序，就不必使用 `main()`函数。但是，我们做的都是运行在某个操作系统上的可执行程序，所以一般情况下 `main()`函数还是少不了的！

【题目 2】 写一段程序使用 `main()`函数的参数将你的名字的拼音传入程序，然后使用 `printf()`函数进行输出。

【分析】 要想将数据传给 `main()`函数，就得使用带参 `main()`函数的声明形式。带参 `main()`函数的声明形式有两种，一种是有返回值的，一种是没有返回值的。这里无论是使用返回值的 `main()`函数形式，还是无返回值的 `main()`函数形式，都是可以的。简单起见，我们使用没有返回值的 `main()`函数形式。`main()`函数的参数有两个，一个表示参数的字符串的个数，另一个是传入的字符串的指针。为了解决问题，只传入一个字符串，所以 `argc` 将等于 2，`argv[1]`将保存你的名字拼音所在内存的地址。

【核心代码】

```
void main(int argc, char* argv[])
{
    printf("my name is %s\n", argv[1]);
}
```

第 13 章 局部变量和全局变量

在讲语句块的时候，我们已经接触了变量的作用域，不过当时是局部变量。在本章中，将扩大范围，深入看看 C 语言中的局部变量和全局变量，看看这两类不同的变量到底有什么样的性质和作用。

13.1 变量的作用域和生命周期

变量的作用域就是指变量在程序中多大的范围内是有效的、可以使用的。例如，语句块中定义的变量的使用范围就是语句块的一对大括号之间，出了这个大括号就不能使用语句块中定义的变量了。变量的生命周期就是变量能“活”多久，也就是变量在程序运行后多久时间内是可以使用的。

如果说变量的作用域是一个空间的范围，那么变量的生命周期就是一个时间上的范围！程序中的空间体现在我们写的代码中，也就是在代码中的范围，就像语句块中定义的变量的作用空间就是一对大括号之间的范围。程序的时间体现在程序运行时，也就是程序中一些语句被执行的一段时间，就像语句块中定义的变量的作用时间就是这个语句块中所有语句被一条条执行的那段时间。

图 13.1 所示的坐标系中的灰色区域就是语句块中定义的变量的使用范围。在空间轴上，也就是变量的作用范围，语句块变量是从左大括号到与其对应的另一个右大括号。在时间轴上，也就是变量的生命周期，语句块变量的作用范围是从语句块中的第一条语句开始到最后一条语句结束。

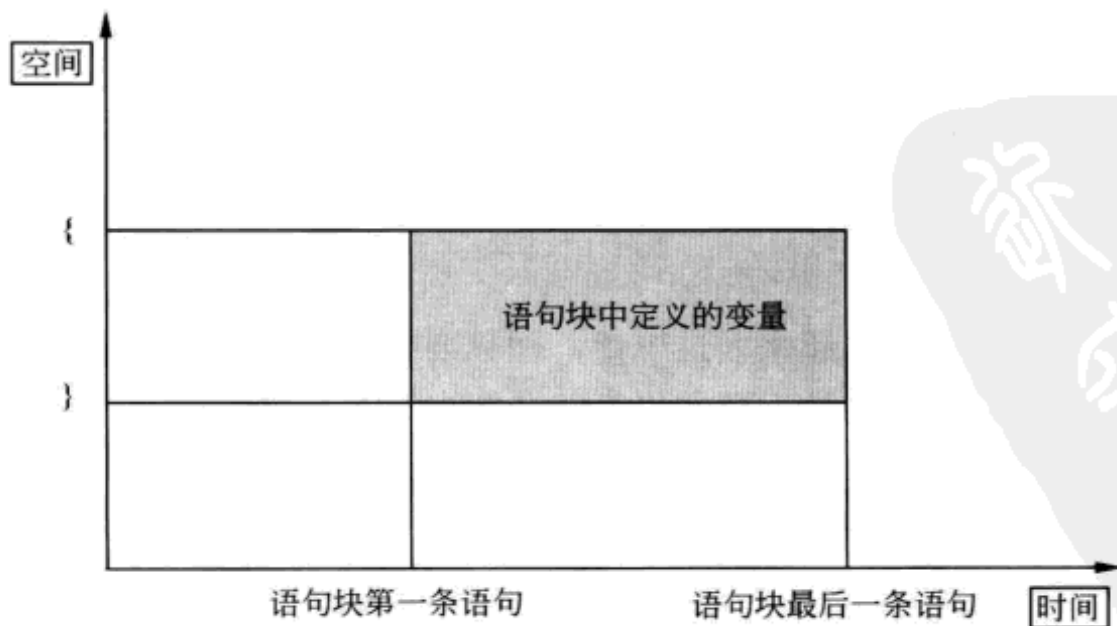


图 13.1 语句块中变量的作用域和生命周期

C 语言中，所有的变量的使用范围都可以用作用域和生命周期来表示，也就是说所有变量的使用范围都可以画成图 13.1 中的方格，在这个方格中，就可以使用相应的变量了。当然，在这一个方格中不可以定义同名的变量。

另外，C 语言中的变量按照是不是在函数体中声明定义来分有局部变量和全局变量两种。在函数体中声明定义的变量被称为局部变量，在函数之外声明定义的变量被称为全局变量。由于语句块也是在函数之中的（无论是在 `main()` 函数还是其他函数中），所以也将其归到局部变量中。函数的形参变量也被归为局部变量。

13.2 函数内的局部变量

C 语言中的局部变量主要是函数之中声明定义的变量，同时也包括语句块和函数形参列表中声明定义的变量。对于这些变量的使用范围，在 C 语言中都是有相同的规则的。接下来就来看看这些规则。

13.2.1 局部变量的作用域

对于局部变量的作用域，大家只要记住这样一句话就可以了：“局部变量的作用域是在其定义的那对大括号之内”。对于函数的形参变量，这句话得稍微变通一下，函数的形参变量的作用域是在函数体之中，也就是包含函数体的那对大括号之中。

下面的这段代码就出现了三种局部变量的定义：整型变量 `a` 是一个函数形参列表中定义的变量，它的作用域是 `fun()` 函数的函数体，也就是 `fun()` 函数后函数定义的那对大括号之中。整型变量 `b` 是一个函数体中定义的变量，它的作用域也是 `fun()` 函数的函数体，即包含它定义的那对大括号之中。整型变量 `c` 是一个语句块中的变量，它的作用域范围是包含它定义的那对大括号之中。

```
int fun(int a)
{
    int b;
    ...
    {
        int c;
    }
    ...
    return 0;
}
```

如果感觉上面的规则记忆起来太过麻烦，你只要知道局部变量的作用域是在一对大括号中就可以，具体是哪对大括号，稍加思索就会明白了。

13.2.2 局部变量的生命周期

局部变量的生命周期，也就是程序在执行的时候，局部变量“存活”的那个时间段，也与包含变量声明定义的那对大括号有关！局部变量的生命周期是从大括号中的第一条语

句开始到最后一条语句结束有效，当然函数形参变量的生命周期就不是包含其声明定义的那对大括号了，而是其后包含函数体的那对大括号。

例如，下面的一段程序，从对函数 `test1()` 和 `test2()` 的每次调用，就可以知道局部变量的生命周期是多久。

```
#include <stdio.h>

void test1(int a)
{
    int b = 1;

    b++;
    printf("a = %d\n",a);
    printf("b = %d\n",b++);
    {
        int c = 1;
        c++;
        printf("b = %d\n",c);
    }
    printf("\n");
}

void test2(int a)
{
    int b = 2;

    b++;
    printf("a = %d\n",a);
    printf("b = %d\n",b++);
    {
        int c = 2;
        c++;
        printf("b = %d\n",c);
    }
    printf("\n");
}

int main()
{
    printf("test1\n");
    test1(1);
    test1(2);
    test1(3);

    printf("test2\n");
    test2(4);
    test2(5);
    test2(6);

    return 0;
}
```

在这段程序中，定义了两个函数 `test1()` 和 `test2()`，这两个函数极其相似，都有一个形参变量 `a`，一个函数内变量 `b` 和一个语句块变量 `c`。唯一不同的是将 `test1()` 中的变量 `b` 和 `c` 初始化为 1，将 `test2()` 中的变量 `b` 和 `c` 初始化为 2。最后在 `main()` 函数中对这两个函数分别调用 3 次，每次传入的参数都不相同。

程序的输出如图 13.2 所示，对于 `test1()` 函数，每次传入的实参数值会被保存在整型变

量 *a* 中，所以每次输出的 *a* 的值都是不一样的。对于 *b* 和 *c*，初始化的时候为 1，对其进行自加运算以后其值变为 2，每次调用 `test1()` 函数输出都为 2。所以，可见每一次函数调用结束之后，变量 *b* 和 *c* 都会随之“灭亡”，当重新调用 `test1()` 函数的时候，就会重新“生成”一个 *b* 和 *c*。定义一个和 `test1()` 函数类似的 `test2()` 函数的目的是为了进行类比，证明局部变量的使用范围确实只在对应的大括号之中，不同大括号之中出现同名变量是不会相互影响的！

```

void test1(int a)
{
    int b = 1;

    b++;
    printf("a = %d\n",a);
    printf("b = %d\n",b++);
    {
        int c = 1;
        c++;
        printf("b = %d\n",c);
    }
    printf("\n");
}

void test2(int a)
{
    int b = 2;

    b++;
    printf("a = %d\n",a);
    printf("b = %d\n",b++);
    {
        int c = 2;
        c++;
        printf("b = %d\n",c);
    }
    printf("\n");
}

int main()
{
    printf("test1\n");
    test1(1);
    test1(2);
    test1(3);
}

```

```

C:\Windows\system32\cmd.exe
test1
a = 1
b = 2
b = 2
a = 2
b = 2
b = 2
a = 3
b = 2
b = 2

test2
a = 4
b = 3
b = 3
a = 5
b = 3
b = 3
a = 6
b = 3
b = 3

请按任意键继续. . .

```

图 13.2 局部变量的生命周期

13.2.3 局部变量的覆盖作用

原则上，C 语言中的每个由大括号包含的局部变量使用范围中是不可以有同名的变量的。但是，对于使用范围嵌套的情况就不一样了，在嵌套的使用范围中允许出现同名的变量。对于这些同名的变量，它们的作用域和生命周期又是什么呢？

对于嵌套使用范围中出现的同名局域变量，它们的作用域和生命周期也是遵循一般局部变量作用域和生命周期的规则的。即作用域为一对大括号之内，生命周期为大括号中第一条语句到最后一条语句。不过，对于嵌套使用范围中的同名变量，得再加一个覆盖规则：使用范围小的变量在其作用域中会覆盖使用范围大的变量。

这个覆盖规则在图 13.3 中明显地表示了出来，图 13.3 中用两个嵌套的矩形框表示两个同名变量的嵌套使用范围。其中，灰色区域是大语句块中的变量的使用范围，它遵循一般变量的作用域和生命周期，大括号包含的范围是其作用域，大括号中的语句是其生命周

期。另外，黄色的区域是小语句块的使用范围，它也是遵循一般变量的作用域和生命周期的。

唯一特殊的就是在黄色区域中只有小语句块中定义的变量是有效的，大语句块中定义的同名变量在此将会失去作用，变得无效。这也正是所谓的局部变量的覆盖作用！

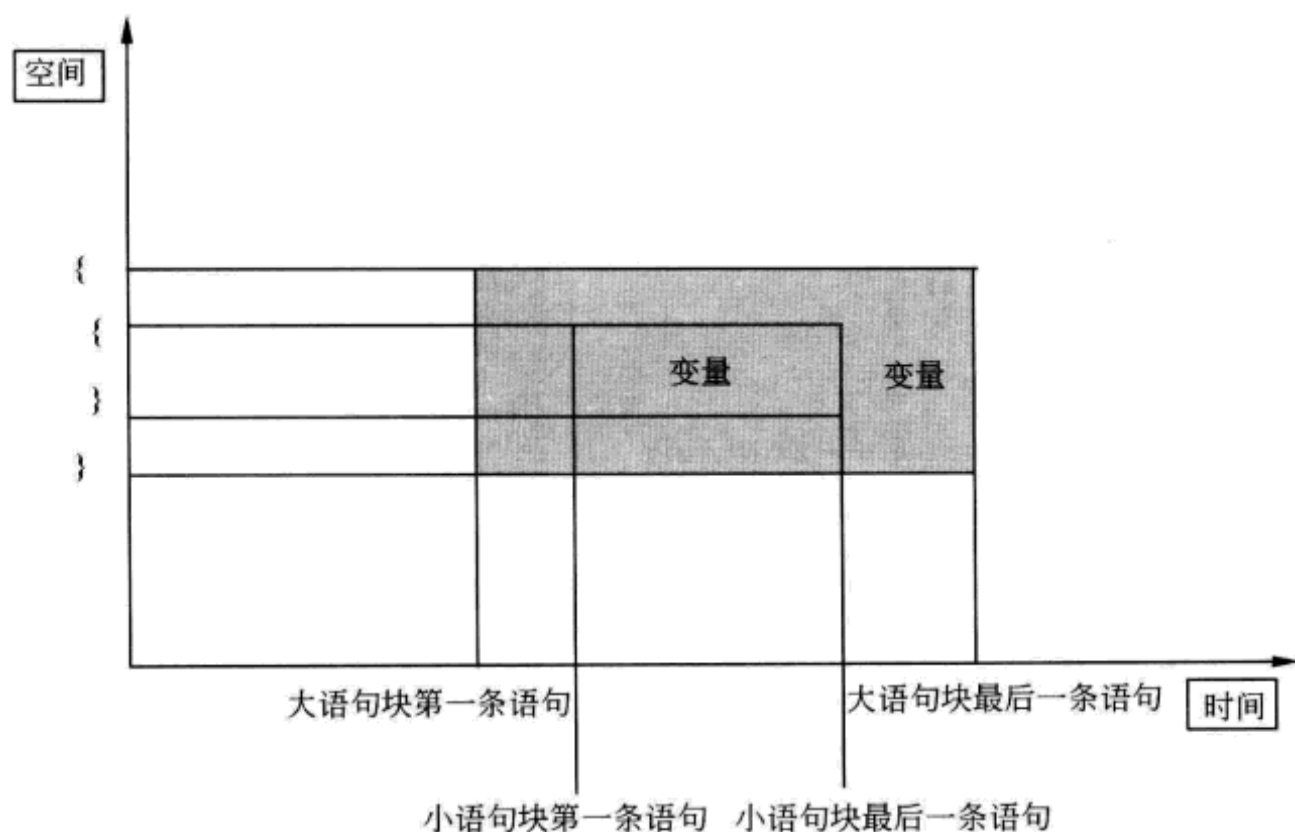


图 13.3 嵌套作用域中的局部变量的覆盖

接下来看一个例子，从这个例子中就可以看出嵌套使用范围中的局部变量是有覆盖作用的。

```
#include <stdio.h>

void test(int a)
{
    int b;

    b = a;
    printf("outside b = %d\n", b);
    {
        int b = 0;
        printf("inside b = %d\n", b);
    }
    printf("outside b = %d\n\n", b);
}

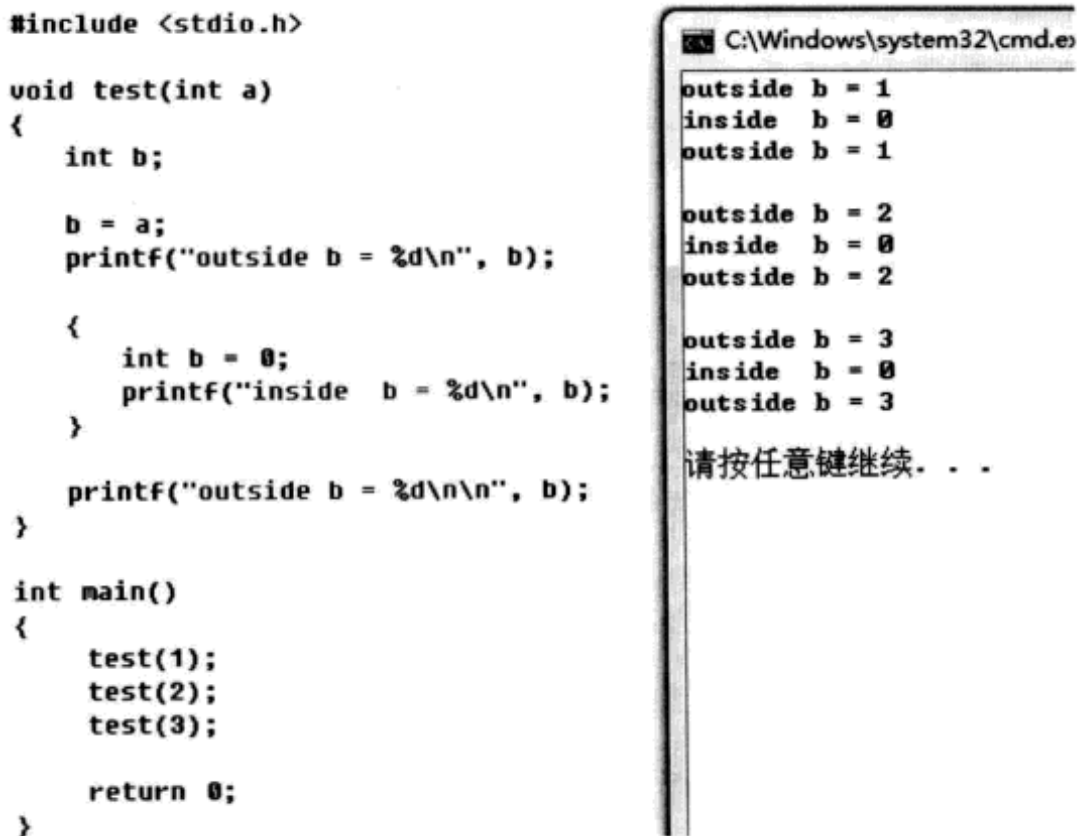
int main()
{
    test(1);
    test(2);
    test(3);

    return 0;
}
```

在这个程序中，定义了一个函数 `test()`。其中，有三个整型变量，`a` 是函数形参变量，第一个 `b` 是函数体变量，第二个 `b` 是 `test()` 函数中的语句块变量。函数的形参变量和函数体

中的变量的使用范围是一样的，所以不能重名，也不存在所谓的嵌套。但是，函数体中的变量和函数体中语句块中的变量就可以重名，就像这个例子中的 `b`，这两个整型变量中就存在局部变量的覆盖作用。这一点从程序的输出就可以看出程序的输出如图 13.4 所示。

在这个程序中，在调用函数 `test()` 时，每次传入不同的参数来改变函数体第一个变量 `b` 的值。对语句块中第二个 `b` 的值不做变动，语句块的前后，分别输出变量 `b` 的值。在图 13.4 所示的输出中，语句块外部的 `b` 每次都改变，但是语句块内部的变量 `b` 一直未变，可见局部变量 `b` 确实存在变量覆盖作用。



```

#include <stdio.h>

void test(int a)
{
    int b;

    b = a;
    printf("outside b = %d\n", b);

    {
        int b = 0;
        printf("inside b = %d\n", b);
    }

    printf("outside b = %d\n\n", b);
}

int main()
{
    test(1);
    test(2);
    test(3);

    return 0;
}

```

```

C:\Windows\system32\cmd.e
outside b = 1
inside b = 0
outside b = 1

outside b = 2
inside b = 0
outside b = 2

outside b = 3
inside b = 0
outside b = 3

请按任意键继续. . .

```

图 13.4 局部变量的覆盖作用

13.3 函数外的全局变量

在 C 语言中，函数的外部也是可以定义变量的，这些变量被称为全局变量。对于这些变量，我们之前没见过，也没有使用过，现在就来看看这些变量使用时应该注意的地方，也就是它们的作用域和生命周期。

13.3.1 全局变量的作用域

在了解全局变量的作用之前，先来简单介绍一下 C 语言中的源文件，在后面的章节中还会详细地讲到的，这里只做简单了解。之前写的代码被称为源代码，C 语言的源代码会被放到一个个的文件中，这些文件被称为源文件。

例如，上一节验证局部变量的源程序就放在一个名为“`test.c`”的源文件中，如图 13.5 所示。图 13.5 中左上角红圈中的“`test.c`”就是所写代码将会被保存到的源文件的名称。

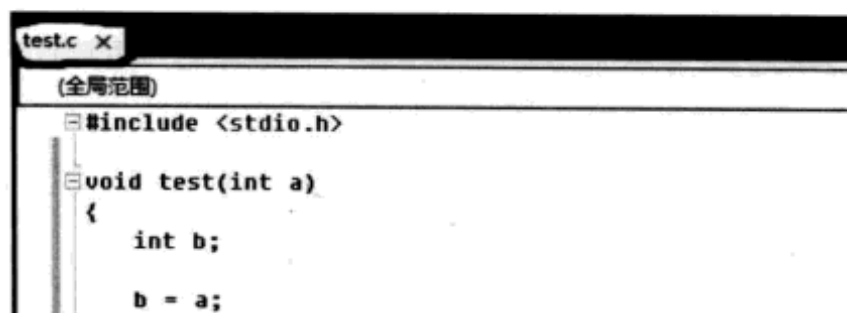


图 13.5 源文件

有人可能会感到困惑，介绍全局变量的作用域与 C 语言中的源文件有什么关系？它们之间确实有关系！C 语言中的全局变量的作用域就是声明定义它的那个源文件，在这个源文件中的任何地方，任何函数，任何函数中的语句块中，都可以使用声明定义在这个源文件中的全局变量。

例如，下面这个简略的程序片段中，在函数 `test()`、函数 `fun()` 和函数 `main()` 的任何地方可以使用全局变量 `a`。

```

int a;

void test ()
{
    //可以使用变量 a
    {
        //可以使用变量 a
    }
}

int fun()
{
    //可以使用变量 a
}

int main()
{
    //可以使用变量 a
    return 0;
}
  
```

13.3.2 全局变量的生命周期

说起全局变量的生命周期，那就长了，根本没有一个大括号可以局限它。全局变量的生命周期是从程序执行开始一直到程序执行结束，也就是贯穿整个程序执行的始末，够长吧！当然，它也不是永生不灭的，程序开始时它生，程序结束时它灭。

通过下面的一个程序来看看全局变量是不是像说的那样可以“活”得很久。

```

#include <stdio.h>

int a = 0;

void test()
{
    a++;
}
  
```

```

    printf("test: a = %d\n",a);
}

int main()
{
    a++;
    printf("main: a = %d\n",a);
    test();
    test();
    a++;
    printf("main: a = %d\n",a);

    return 0;
}

```

在这个程序中，定义了一个函数 `test()`，实现的功能是对 `a` 进行自增，然后在 `main()` 函数中对 `a` 进行自增并连续两次调用函数 `test()`。最后再对 `a` 进行自增，并且在每次自增以后输出 `a` 的值。

使用命令行将这个程序运行三次，输出如图 13.6 所示。可以看出在每次运行的时候，`a` 的值都是不断增大的，这也说明 `a` 在程序运行的时候都是存在的。但是，重新运行程序的时候，`a` 的值又重新从 1 开始，说明全局变量 `a` 是生于程序运行开始，终于程序运行结束的！



```

C:\Windows\system32\cmd.exe

E:\C语言的书\project\test\debug>test.exe
main: a = 1
test: a = 2
test: a = 3
main: a = 4

E:\C语言的书\project\test\debug>test.exe
main: a = 1
test: a = 2
test: a = 3
main: a = 4

E:\C语言的书\project\test\debug>test.exe
main: a = 1
test: a = 2
test: a = 3
main: a = 4

```

图 13.6 全局变量的生命周期

13.3.3 局部变量对全局变量的覆盖作用

我们知道对于局部变量，不同使用范围内可以定义同名的变量。局部变量和全局变量也不是同一个使用范围，那么是不是可以定义同名的变量？如果可以，那么这些同名的变量的使用范围又有怎样的规则呢？

对于第一个问题，答案是肯定的。如果已经定义了一个全局变量，还是可以定义一个和它名字一模一样的局部变量的。对于第二个问题，变量使用范围有嵌套的全局变量和局部变量，还是遵循和局部变量覆盖作用类似的规则，局部变量会覆盖全局变量。

下面的程序就是对这两个问题的答案的很好的验证：

```

#include <stdio.h>

int a = 0;

```

```

void test1()
{
    a++;
    printf("test1: a = %d\n",a);
}

void test2()
{
    int a = 0;
    printf("test2: a = %d\n",a);
}

int main()
{
    a++;
    printf("main: a = %d\n",a);
    test1();
    test2();
    test1();
    test2();
    a++;
    printf("main: a = %d\n",a);

    return 0;
}

```

在这个程序中, 定义了两个函数 test1() 和 test2(), test1() 实现对全局变量 a 的自增运算, 并将其值输出。在 test2() 中又声明定义了一个函数体中的整型局部变量 a, 将其初始化为 0, 并将其值输出。最后, 在 main() 函数中只实现了 a 的两次自增和对函数 test1() 和 test2() 的两次调用。

程序的输出如图 13.7 所示。由于函数 test1() 和 main() 函数中没有定义局部变量 a, 所以对应它们的输出都是持续地给全局变量 a 自增。而函数 test2() 就不同了, 由于在 test2() 中定义了一个局部变量 a, 发生了局部变量对全局变量的覆盖作用, 所以对应函数 test1() 的输出就只是局部变量产生时初始化的值 0。

```

#include <stdio.h>

int a = 0;

void test1()
{
    a++;
    printf("test1: a = %d\n",a);
}

void test2()
{
    int a = 0;
    printf("test2: a = %d\n",a);
}

int main()
{
    a++;
    printf("main: a = %d\n",a);
    test1();
    test2();
    test1();
    test2();
    a++;
    printf("main: a = %d\n",a);

    return 0;
}

```



```

C:\Windows\system32\cmd.exe
main: a = 1
test1: a = 2
test2: a = 0
test1: a = 3
test2: a = 0
main: a = 4
请按任意键继续. . .

```

图 13.7 局部变量对全局变量的覆盖作用

13.4 变量修饰符

通过前面的介绍，我们知道了 C 语言中的变量按照是否在函数体中分为局部变量和全局变量。这两种变量的作用域和生命周期都遵循不同的规则。C 语言是一种灵活的语言，所以，很多规则都不是一成不变的，就像我们之前讲的全局变量和局部变量的作用域和生命周期，也是可以通过添加变量修饰符来改变的。本节来看看这些可以改变变量作用域和生命周期的变量修饰符。

13.4.1 使用修饰符改变变量的作用域和生命周期

C 语言中有一些特殊的关键字，这些关键字可以被放在变量的声明定义之前，用来改变变量的作用域和生命周期，称为变量修饰符。是不是感觉这些变量修饰符就像小说里的神丹妙药一样，可以让变量“腾云驾雾”扩大作用范围，“延年益寿”增长生命周期？其实，确实是这样的，C 语言中的变量修饰符没有副作用，只会扩大变量的使用范围。

C 语言中的变量修饰的使用形式如下所示：

变量修饰符 数据类型 变量名；

这个形式就是在变量声明定义之前加了一个变量修饰符，其余部分和声明定义一个变量是一样的。“数据类型”就是 C 语言中的一种数据类型表示，“变量名”就是为变量取的名字，遵循 C 语言命名规范。

这里主要来看一看 C 语言中的“变量修饰符”，它们和 `int`、`char`、`float` 等一样都是 C 语言规定的关键字，不容更改！C 语言中，用做变量修饰符的关键字有以下 7 个：

`auto` `restrict` `extern` `volatile` `const` `static` `register`

它们中的每一个都可以单独作为变量修饰符加在变量声明定义之前来改变变量的作用域和生命周期，也可以在意义不冲突的情况下使用其中几个的组合作为变量的修饰符。

例如，下面就是变量声明定义之前加变量修饰符的例子。其中就包含变量修饰符关键字的单独使用和组合使用。

```
auto int result;
const static char p;
restrict float * point_f;
```

这三个变量声明定义中，第一个和最后一个分别单独使用关键字 `auto` 和 `restrict` 作为变量修饰符，而中间的字符变量 `p` 则使用关键字 `const` 和 `static` 的组合作为变量修饰符。

13.4.2 C 语言中常用变量修饰符的作用

通过前面的介绍可以知道，C 语言中有 7 个可以用做变量修饰符的关键字，它们可以单独使用或者组合起来作为变量修饰符来改变变量的作用域和生命周期。下面就来看看几个常用的变量修饰符的常见用途，看看它们是如何改变变量的作用域和生命周期的。对于

那些很少使用而且作用很特殊的关键字，等到需要的时候可以查阅相关文档或标准，只要记住它们都是改变变量的使用范围或者约束变量访问的就可以了。

1. 默认的auto

C 语言中，在变量声明定义的时候，如果前面没有加任何变量修饰符，默认相当于加了一个 auto。所以，auto 并没有改变前面所讲的变量作用域和生命周期，加与不加都是一样的，一般为了写代码方便干脆就直接省略了。像下面两个整型变量 a 的声明定义是等价的，作用一样！

```
int a;
auto int a;
```

2. extern扩大全局变量的作用域

从前面的内容我们已经知道，全局变量的生命周期是程序运行的整个始末，已经够长了，再加也加不到哪去了。唯一不足的就是全局变量的作用域被限制在声明定义它的那个源文件了。如果全局变量在任何源文件中都可以使用，那就完美了！

extern 关键字就是来完成这件完美的事情的，它的作用就是扩大全局变量的作用域。如果在一个源文件中使用在另一个源文件中声明定义的全局变量，就在这个源文件中使用下面的形式重新声明一下这个全局变量。

```
extern 数据类型 变量名;
```

记住，这个形式不是重新定义一个全局变量，而是声明一下一个已经定义的全局变量。这跟在两个源文件中定义两个同名的全局变量是不一样的，是将一个源文件中定义的全局变量的作用域扩大到另一源文件中。这也是我们至今为止，见到的唯一一个变量的声明和定义分开的情况，一般变量的声明定义都是一起的。

在图 13.8 中展示了一个整型变量 a 定义在源文件 test2.c 中，并将其初始化为 2，使用 extern 将其使用范围扩大到源文件 test1.c 中，将其自增以后输出。程序的输出为 3，可见 extern 确实把 test2.c 源文件中的整型变量 a 扩大到 test1.c 源文件中了。

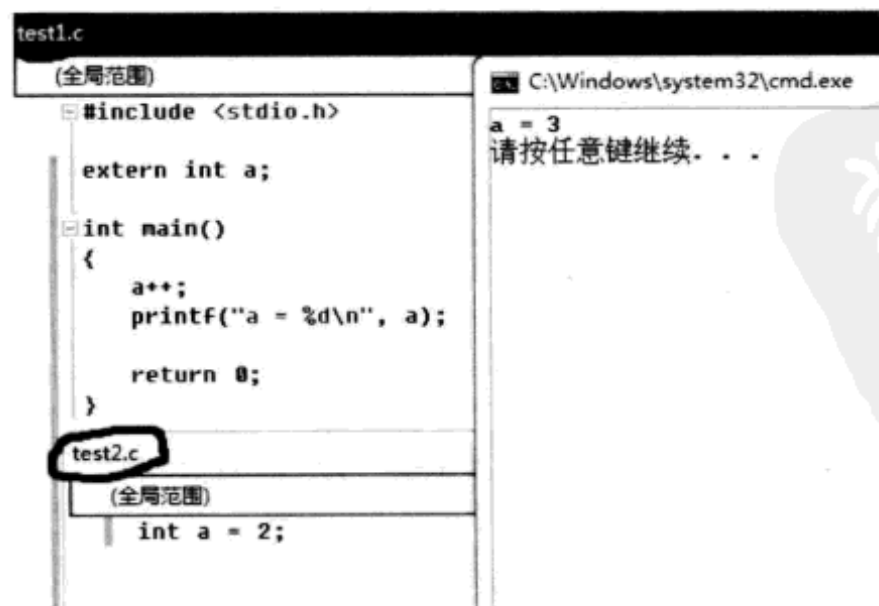


图 13.8 extern 变量修饰符

3. static增加局部变量的生命周期

我们知道，局部变量的作用域是在一对大括号中，出了这对大括号就不能使用局部变量了。按理说，这个作用域还有扩展的余地，但是C语言中没有对其进行扩展的关键字，原因是再扩就成了全局变量了，那样何不直接定义一个全局变量得了！

按照前面的规则，局部变量的生命周期是从一个大括号开始的第一条语句一直到最后一条语句。出了这个大括号中的语句，局部变量就会消失，下次进入这个大括号中的语句时，又会重新产生。C语言提供了一个变量修饰符 `static`，可以将局部变量的生命周期扩展到程序执行的整个过程，和全局变量的生命周期是一样的，但是就是没有全局变量的作用域大。

下面来看一个例子，看看 `static` 关键字对局部变量的这种影响。

```
#include <stdio.h>

void test()
{
    static int a = 0;

    a++;
    printf("a = %d\n", a);
}

int main()
{
    test();
    test();
    test();

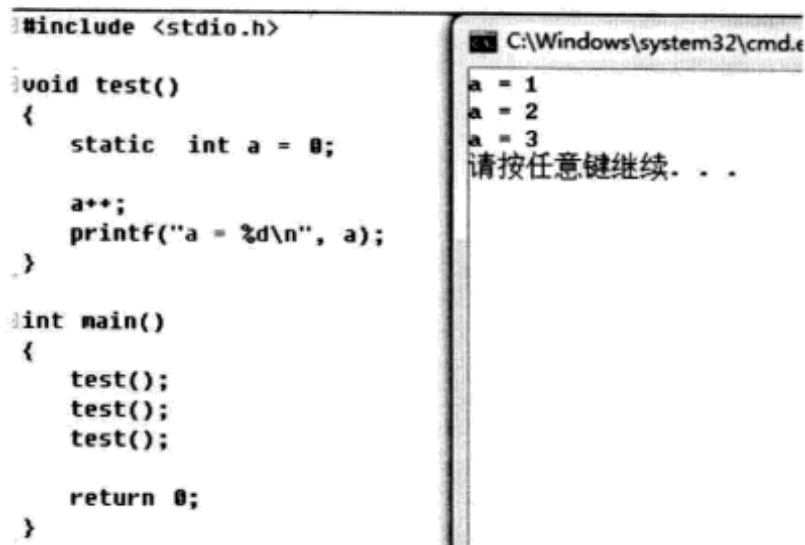
    return 0;
}
```

在这个程序中，定义了一个函数 `test()`，在其中声明定义了一个带有 `static` 变量修饰符的整型变量 `a` 并将其赋值为 0，然后将 `a` 自增，最后将其输出。在 `main()` 函数中，我们对 `test()` 函数进行了三次调用，按照一般的规则程序的输出应该是三个 1，但是实际上程序的输出如图 13.9 所示。

在图 13.9 中，输出了连续增长的数字 1、2、3，这是因为 `static` 变量修饰符将整型变量 `a` 的生命周期增加到整个程序的执行始末了。另外，还需注意一点，`static` 修饰的变量只会被初始化一次，这也保证了输出是连续的 1、2、3，而不是每次都初始化产生的三个 1。

4. register让变量访问变得更快

我们知道，程序中的变量是在内存中保存数据的，访问内存中的数据已经很快了，但是对于一些需要更快地访问数据的应用，内存还是显得有点慢。为了解决这个问题，可以将数据放到寄存器（`register`）中来访问，因为寄存器的访问速度比内存快多了。C语言中提供了另外一个变量修饰符 `register`，被它修饰的变量会用寄存器来存储数据，这样就可实现更快地访问某些数据的目的了。



```

#include <stdio.h>

void test()
{
    static int a = 0;

    a++;
    printf("a = %d\n", a);
}

int main()
{
    test();
    test();
    test();

    return 0;
}

```

C:\Windows\system32\cmd.e
a = 1
a = 2
a = 3
请按任意键继续. . .

图 13.9 static 变量修饰符

13.5 小 结

本章中讲述了程序中不同位置定义的变量的作用域和生命周期，从时间和空间的角度来看待和使用我们定义的变量。本章中的重点是变量的作用域和生命周期的概念，难点是不同的变量修饰符对变量作用域和生命周期的影响。下一章将学习一个新的 C 语言知识点——预处理命令，它可以在程序进行编译之前对其进行一些“预处理”。

13.6 习 题

【题目 1】 不在变量作用范围和生命周期之内使用该变量，会有什么结果？写段程序试验一下！

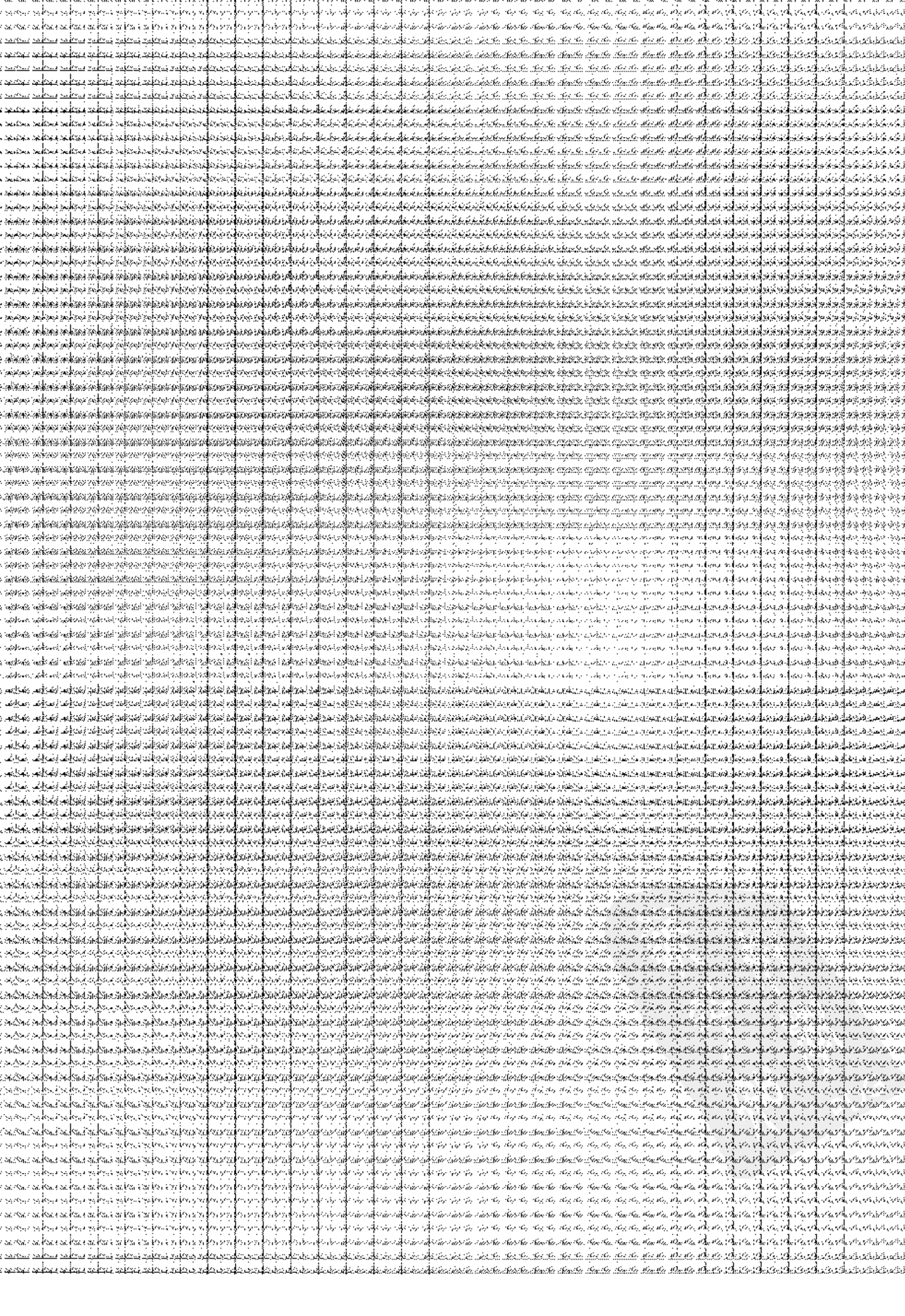
【分析】 一般不在变量的作用范围和生命周期之内使用变量，会很容易检查出来，编译器直接告诉你没有定义这个变量。

【核心代码】

```

void fun()
{
    {
        int a;
        ...
    }
    printf("%d\n",a);
}

```

第 5 篇 C 语言的高级内容

▶▶ 第 14 章 预处理命令、文件包含

▶▶ 第 15 章 文件操作



第 14 章 预处理命令、文件包含

到现在为止，C 语言中的主要内容已经基本介绍完了，读者完全可以凭借前面的知识写出满足一般需要的程序了。不过，为了能够更好、更灵活地使用 C 语言，本章将 C 语言中最后一部分很重要的知识介绍一下，内容主要是关于如何灵活组织工程和如何使用系统功能与资源的。

14.1 预处理命令的作用

C 语言是一种很灵活的语言，很多方面都可以由编程人员控制，预处理命令就是 C 语言提供的一类很强大的命令，它可以控制编译器在进行编译链接之前，对写的代码进行一些其他的处理。所谓预处理就是在编译之前对程序的处理，所以预处理命令也叫做预编译命令。本节就来看看 C 语言中功能强大的预处理命令。

C 语言的编译器主要完成两个作用：（1）如果我们写的程序中有预处理命令，就根据预处理命令对程序做一些前期的处理，否则什么都不做；（2）对经过预处理过的程序进行编译，生成计算机可以识别的二进制文件。

这两个作用可以通过图 14.1，一个简化了的程序生成过程看出来。实际的过程要比这复杂得多，不过这个图足以说明问题了。

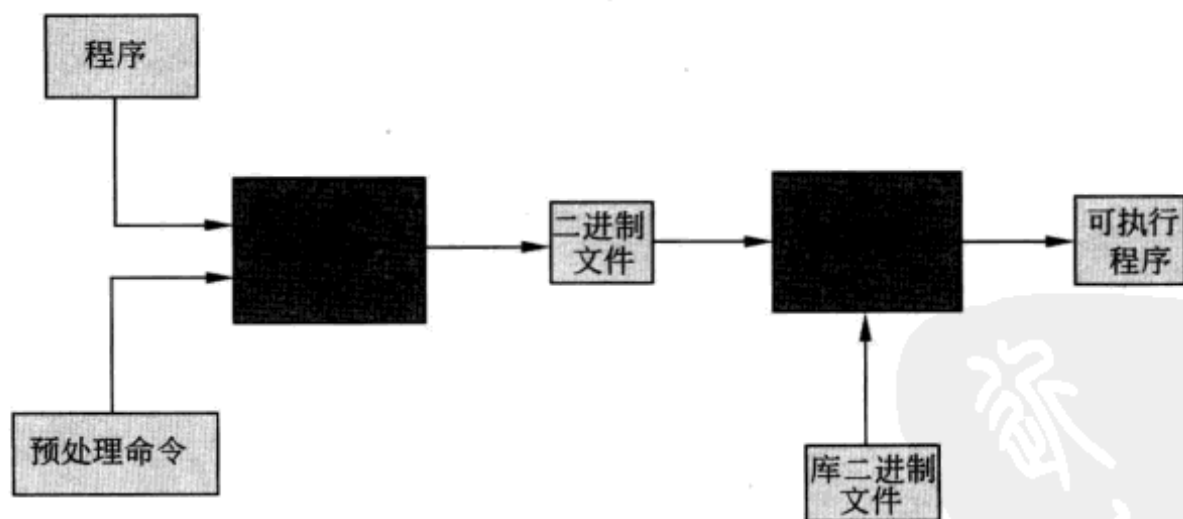


图 14.1 可执行文件生成过程简单示例

14.1.1 程序预处理

我们知道编译器的第一个作用，就是在将程序编译成二进制文件之前，根据本次编译需求和环境约束对程序做一些前期的处理，这个前期的处理被称为程序的预处理。

那么，为什么要对 C 语言的程序进行预处理呢？难道我们写的程序不能满足需要，还需要一些其他处理？既然还需要其他处理，为什么不在编写程序的时候处理好呢？

先来回答第一个问题，为什么要对 C 语言程序进行预处理呢？回答了这个问题，其他问题也就明了了。对程序预处理的主要原因有三个：（1）修改代码；（2）确定程序中没能确定的东西；（3）重复使用某些代码。

- 我们写的代码，进入编译器之后，如果不经预处理，里面的关键字、标识符、语句等就都定死了，再也无法修改了。如果在这个时候，还想对程序再做一些修改，就必须进行预处理。否则，就得在程序还没有进入编译器之前，自己手动修改代码。
- 我们写程序的时候，其实很多东西都是事先无法确定的。例如，写的程序将会运行在哪个操作系统上。就拿 Windows 来说吧，Windows 也是有 98、XP、Vista、Win7 的，这些系统也是有差别的。可以在编译程序的时候，告诉编译器译出来的程序要运行在什么系统上，然后让编译器对我们的程序进行修改。像这样的不确定因素还有很多，基本上都是通过对程序进行预处理来消除的。
- C 语言的预处理提供了一些机制可以让你的程序使用一些代码，这样就可以免去很多重复劳动。

以上三点是 C 语言预处理命令的主要用途和功能，如果去探索还会有很多细微的用途。这些用途归结为一点，为了解决不同问题，需要对程序做不同程度的修改，这样的工作既枯燥又庞大，我们可以通过预处理将这些工作交给编译器替我们完成。其实，在写程序的时候可以完全不使用预处理，不过，我怕你会忙不过来！

14.1.2 预处理命令

C 语言预处理命令是用来指导编译器对程序进行预处理的。这些命令有的是通用的，是由 C 语言标准规定的，有的是特殊的，只能在某些编译器中使用。从图 14.1 中，也可以看出预处理命令的作用，它会随着我们写的程序一起交给编译器进行处理，最后得到计算机可以识别的二进制文件。

那么它到底是如何指导的呢？需不需要像输入命令那样在编译的时候手动输入给编译器呢？其实，预处理命令没有你想象得那么复杂，它是在我们写程序的时候随程序一起写到源文件中的！

C 语言预处理命令的形式如下所示：

```
#预处理关键字 其他
```

这个表示形式中，“#”符号是预处理命令特有的符号。C 语言中凡是出现“#”号的地方一定是预处理命令，编译器通过“#”号来将我们写的程序和预处理命令区分开。“预处理关键字”是一些指导编译器的特殊命令关键字。“其他”根据要处理的命令不同而不同，没有统一的规则。

14.1.3 C 语言的几类预处理命令

C 语言提供了三类预处理命令，指导编译器来完成对程序的不同层次的修改，它们分别是：宏定义、预编译控制和头文件包含。

14.2 C 语言中的宏定义

C 语言中的宏定义是预处理命令的一种，也是最简单、最直接地对程序进行修改的一种预处理命令。它是让编译器在程序预处理的时候对符号、文字、标识符等进行直接的替换。对程序中的某些符号、文字、标识符等进行直接替换之后，我们的程序就会变成另外的样子了。

这样的替换有的时候是很有用的，大家在用软件的时候经常会有英文版和中文版，很多人都喜欢用中文版！其实，这两种版本程序一般就是在编译的时候，通过宏定义直接替换得到的。在需要英文版的时候就将有些文本替换成英文，在需要中文版的时候将同样位置的文本替换成中文就可以了。

14.2.1 C 语言的宏定义形式

C 语言中的宏定义是通过对程序中的某些部分实现直接替换来达到对程序修改的目的。那么它是如何实现的呢？答案很简单，是通过预处理命令来实现的！

C 语言中的宏定义命令是一个标准规定的命令，也就是说只要是个 C 语言编译器，都是支持宏定义命令的！C 语言宏定义的命令是使用关键字 `define` 表示的。结合 C 语言预处理命令的一般形式，可以推测出 C 语言宏定义的一般形式，如下所示：

```
#define 其他
```

C 语言宏定义的一般形式确实是这样的，以“#”开头，接着是预处理命令关键字 `define`，最后是一些所需的东西。

这个一般形式确实没错，但它还是太简单了，C 语言中的宏定义形式要比它稍微复杂一些。复杂的地方就是 C 语言规定了“其他”部分的形式。你也可以先猜测一下，其他部分应该是什么样子？“其他”部分主要是完成宏定义的替换工作的，“替换”是将一个东西换成另一个东西。所以，“其他”就是规定替换和被替换这两个东西的形式的。

因而一个完整的宏定义的形式如下所示：

```
#define 被替换的内容 替换为的内容
```

其中，“被替换的内容”就是将要被替换的 C 语言符号、文本和标识符等，这些内容将会被替换成“替换为的内容”中的 C 语言符号、文本和标识符等。只要有上面的宏定义，程序中所有出现“被替换的内容”的地方都会在编译器预处理的时候被替换，所以还是得悠着点用，不要把不需要替换的地方也替换了。另外，这个形式中，“被替换的内容”是

在“替换为的内容”的后面，不能放反，放反了就会替换反。

例如，要写一个程序，实现的功能是在窗口中写一个水果的名字，但是具体写什么事先还不知道，得根据具体的环境才能知道。直接在程序中写个“Fruit”就行，在具体的应用中，我们再将“Fruit”替换成具体的水果名。

例如这次要写个苹果，就直接定义“Fruit”为“Apple”就可以了。

```
#define Fruit Apple
```

下次需要写的是橘子，把“Fruit”定义为“Orange”就可以了。

```
#define Fruit Orange
```

14.2.2 不带参的宏定义

C 语言中的宏定义主要有两种：不带参的宏定义和带参的宏定义。不带参的宏定义是最简单的一种，直接实现文字符号的替换，不会再有任何的改变了，之前见到的都是不带参的宏定义。它的形式如下所示：

```
#define 被替换的符号文字 替换符号文字
```

不带参的宏定义将程序中的所有东西都看成是符号和文字，它直接根据预处理命令 `define` 对符号文字进行替换就可以了，简单直接！

例如，从下面的程序中就可以看出，宏定义的替换是只认符号文字的。

```
#include <stdio.h>

#define num_a num_b

int main()
{
    int num_a = 1;

    num_b = 2;
    printf("num_a = %d\n", num_a);

    return 0;
}
```

如果没有宏定义，这个程序是很难解释通的。程序中没有定义 `num_b`，但是却给它赋值了。程序的输出也是很诡异的，如图 14.2 所示，`num_a` 的值明明是 1，为什么输出就成 2 了？

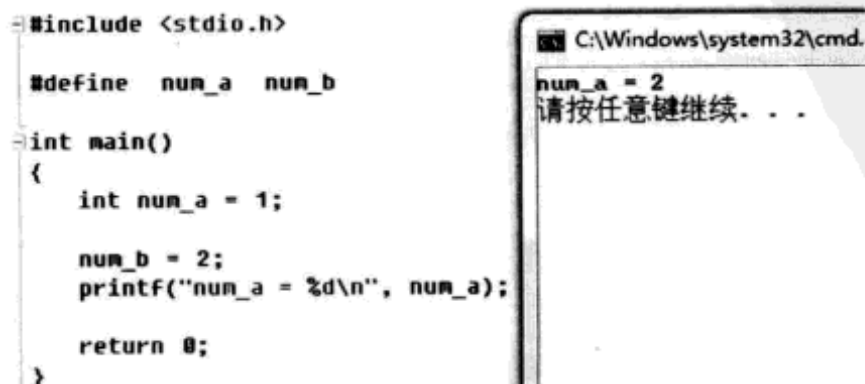


图 14.2 无参宏定义

原因只有一个，我们使用预处理命令 `define` 将标识符 `num_a` 替换成 `num_b` 了，如果手动对上面的程序做一下预处理，就会变成下面的样子。程序中除了 `printf()` 函数中双引号之中的 `num_a` 还存在，其他的 `num_a` 都变成 `num_b` 了。这个变化之后的程序就可以解释上面的疑惑了，其实，`num_b` 已经定义了，输出的也正是 `num_b` 的值，`num_a` 早都已经消失了，所以输出为 2！

有人会问，所有的 `num_a` 都被替换了，为什么唯独 `printf()` 函数中双引号之中的 `num_a` 没变呢？其实，这是 C 语言的一个规定，字符串中的符号不能被宏定义替换，`printf()` 函数中双引号之中的部分是一个字符串，所以没有被替换。

```
#include <stdio.h>

int main()
{
    int num_b = 1;

    num_b = 2;
    printf("num_a = %d\n", num_b);

    return 0;
}
```

14.2.3 带参的宏定义

由于不带参宏定义过于严格而且死板，所以 C 语言提供了另一种宏定义，带参宏定义。之所以叫它带参宏定义，是因为它在形式上和函数很相似。到底是怎么个像法？来看看带参宏定义的 C 语言形式就知道了。

```
#define 宏名(符号 1, 符号 2, ..., 符号 n) 替换内容
```

带参宏定义对被替换内容的形式做了更具体的要求，有点像函数，不过参数列表中不是变量定义而是一串符号列表。在这个表示中“宏名”和函数名有点像，只要遵循 C 语言标识符命名规范就可以了。括号中的所有符号也要遵循 C 语言标识符命名规范，之间使用逗号隔开。“替换内容”是包含小括号中的“符号”的一些 C 语言表达式、语句等。

上面的带参宏定义形式所完成的工作，就是将程序中出现“宏名(…)”的地方，替换成“替换内容”，并用小括号中的东西，按照宏定义中符号表中的顺序，依次来替换“替换内容”中相应的符号。

例如，我们定义了这样一个宏：

```
#define add(a, b) a+b
```

在程序中，如果出现了 `add(…)`，就会被替换成 `a+b`，并且 `a` 和 `b` 也会被具体的内容替换。如果程序中有 `add(1,2)`，将会被替换成表达式 `1+2`；如果程序中有 `add(3.14, 5)`，将会被替换成 `3.14+5`；诸如此类。

有人也许要问，带参宏定义和函数简直太像了，函数调用也是这么用的，它们之间到底有什么区别呢？它们的区别主要有如下三点。

- ❑ 处理时间不同：宏定义是在预处理阶段进行的。而函数定义是在编译阶段实现的，函数调用是在函数运行时完成的。

- ❑ 处理方法不同：宏定义是直接替换的，尽管稍微有点复杂，但是本质上还是在进行符号替换。而函数调用是有一整套的参数传递和返回值过程的，不是简单的符号替换。
- ❑ 符号类型不一样：宏定义是无数据类型的，仅仅只是符号。而函数中的符号都是变量，是有类型的，如果类型不一致还会出现警告甚至错误。

另外，有一点需要注意一下，带参宏定义中的“替换内容”可以是由很多语句、表达式等组成的一大块东西，很有可能一行写不完，而 C 语言中的宏定义只认与宏定义在一行的“替换内容”！如果想让宏定义多认几行内容，就在每一行后面写一个反斜杠“\”，最后一行不需要。反斜杠的作用就是告诉编译器预处理命令还没完，后面还有呢！

例如，下面的表示就是 C 语言可以识别的带参宏定义，对于不带参的宏定义也可以使用这一规则。

```
#define add(a, b, c, result)  result = a + b ; \
                             result += c;
```

C 语言是很灵活的，一个功能可以使用很多种方式来完成。之前介绍的使用函数完成三个数相加的功能，也可以使用上面定义的 add 宏来完成。下面就是完整的程序。

```
#include <stdio.h>

#define add(a, b, c, result)  result = a + b ; \
                             result += c;

int main()
{
    int sum;

    add(1,2,3, sum);
    printf("sum = %d\n",sum);

    return 0;
}
```

在编译器预处理阶段，这个程序就会变成下面的样子。

```
#include <stdio.h>

int main()
{
    int sum;

    sum = 1 + 2;
    sum += 3;
    printf("sum = %d\n",sum);

    return 0;
}
```

程序中的 add(1,2,3,sum)被预处理命令替换成了两条 C 语言语句，这两条语句完成 1,2,3 相加，并将结果保存到 sum 中，所以程序能正确地完成三个数相加并保存结果的功能。程序的输出如图 14.3 所示，跟我们预想的一样，计算结果为 6。


```
#include <stdio.h>

#define add(a, b, c, result)    result = a + b ; \
                                result += c;

int main()
{
    int sum;

    add(1,2,3, sum);
    printf("sum = %d\n",sum);

    return 0;
}
```

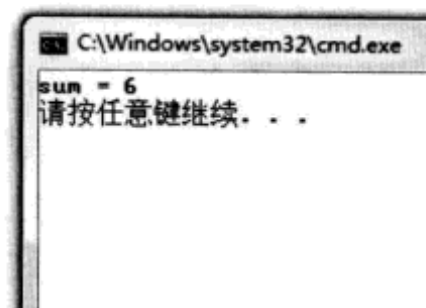


图 14.3 带参宏定义

14.3 预编译控制

预编译控制是 C 语言提供的另一类预处理命令，它对程序的修改作用远比宏定义大得多。它可以根据条件决定哪一片代码需要，哪一片代码不需要，是对代码一片一片进行处理的，需要的被留下，不需要的直接删掉。

14.3.1 C 语言预编译控制

预编译控制也是一种 C 语言预处理命令，它和前面讲的 if 选择结构有点类似，也是根据条件的真假来做一些动作。if 选择结构是根据条件的真假，来决定哪些语句需要执行哪些语句不需要执行；预编译控制则是根据条件的真假，在预处理阶段决定哪片程序应该保留哪片程序应该删除。

在形式上预编译控制和 if 选择结构也很相似，只是它们一个是在预处理阶段完成的，一个是在程序执行的时候完成的。C 语言预处理命令的一般形式如下所示：

#预编译控制关键字 其他

这个形式和一般的预处理命令的形式一样，只是将预处理命令确定为“预编译控制关键字”了。C 语言就是根据“预编译控制关键字”的含义和“其他”的内容来确定哪一片代码可以保留，哪一片代码应该删除的。

C 语言中的预编译控制关键字有以下 6 个：

if ifdef ifndef elif else endif

(1) if 命令，跟 if 选择结构中的 if 类似，其后跟的“其他”部分是一个具有真假值的表达式。如果这个表达式为真，那么之后的语句块保留，否则，之后的语句块删除。

(2) ifdef 命令，其后跟的“其他”部分是一个标识符。如果这个标识符是 define 命令中的“被替换内容”，那么，之后跟的语句块保留，否则，之后的语句块删除。

(3) ifndef 命令和 ifdef 命令刚好相反。其后跟的标识符如果不是 define 命令中的“被替换内容”，那么，之后的语句块保留，否则，之后的语句块删除。

(4) elif 命令和 if 选择结构中的 else if 类似，其后跟的“其他”部分是一个具有真假

值的表达式。如果这个表达式为真，那么之后的语句块保留，否则，之后的语句块删除。

(5) `else` 命令和 `if` 结构中的 `else` 类似，其后不需要跟任何东西，如果之前的语句块都被删除了，那么它之后的语句块就保留下来，否则就被删除。

(6) `endif` 命令只是一个预编译控制结束标志，相当于 `if` 选择结构中的最后一个右大括号，其后不需要跟任何东西。

通过这几个预编译控制命令的组合，就可以实现决定哪片代码需要，哪片代码不需要的功能了。其中，`if`、`ifdef` 和 `ifndef` 总是作为命令组合的开始，有点像 `if` 选择结构中的 `if` 关键字和左大括号。`elif` 作为命令组合中的中间预处理命令，有点像 `if` 选择结构中的 `else if`。`else` 作为组合命令中的最后一个中间命令，有点像 `if` 选择结构中的 `else` 关键字。`endif` 总是作为组合命令中的最后一个命令，有点像 `if` 选择结构中的右大括号。

例如，下面就是一个符合要求的 C 预编译控制的组合：

```
#ifdef 其他
代码片 1
#elif 其他
代码片 2
#else
代码片 3
#endif
```

在这个组合中，根据命令之后“其他”中的内容，预编译的时候就可以确定到底是保留“代码片 1”、“代码片 2”还是“代码片 3”。当然，无论如何最终只有一个代码片被保存下来。

这个结构和 `if-else if-else` 结构很相似，不过要清楚，预编译控制和选择结构不是一回事，只是形式上相似而已，预编译控制是在预编译阶段选择如何保留代码片的，而选择结构是在程序执行的时候选择如何执行语句的。

14.3.2 三种预编译控制组合形式

对于各种预编译控制命令，如果你觉得不好组合，可以回想一下前面学习的 `if` 控制结构的各种形式，预编译控制和选择结构控制是类似的。你可以先从它们开始，以后逐步变化。

1. 单分支控制

```
#if 表达式
语句片
#endif
```

在这个组合形式中，只进行了单个分支的预编译控制。如果判断成功，语句片就被保留，如果判断不成功，语句片就被删除。

2. 双分支控制

```
#if 表达式
    语句片 1
#else
    语句片 2
#endif
```

这里通过命令 `if`、`else` 和 `endif` 的组合，实现了一个双重分支的预编译控制。如果 `if` 命令判断成功，就保留“语句片 1”，否则，就保留“语句片 2”。

3. 多分支控制

```
#if 表达式 1
    语句片 1
#elif 表达式 2
    语句片 2
...
#elif 表达式 n
    语句片 n
#else
    语句片 n+1
#endif
```

在这个结构中，中间的省略号省略了多个 `elif` 命令的判断和之后的语句片。在结构组合中，使用了 `if`、`elif`、`else` 和 `endif` 实现了多分支的预编译控制，从上到下依次判断表达式，直到遇到一个成功的判断，就保留其后的语句块，其余的都删除。

在这三种组合中 `if` 命令都可以被 `ifdef` 和 `ifndef` 命令代替，从而形成一个新的判断结构，不过组合形式还是上面三种。

14.3.3 一个简单的例子

接下来看一个简单的例子，看看预编译控制是如何保留和删除代码片的。

```
#include <stdio.h>

int main()
{
    #ifdef PRINT
        printf("print ifdef\n");
    #else
        printf("print else\n");
    #endif

    return 0;
}
```

在这个程序中，我们使用双分支预编译控制来决定将会删除哪个 `printf()` 输出语句，使用的是 `ifdef` 预编译控制起始命令。如果标识符在 `define` 命令定义（出现在“被替换内容”中）就输出“`print ifdef`”，否则就输出“`print else`”。

我们对上面的程序稍作修改（添加和不添加 `define` 预处理命令），结果截然不同，如图 14.4 所示。左边的程序中使用了 `define` 宏定义，定义了 `PRINT`，虽然定义它为 `空`，但是只要定义了就好，所以第一个 `printf()` 语句被保留，输出为“`print ifdef`”；右边的程序没有定义 `PRINT`，所以第二个 `printf()` 语句被保留，输出为“`print else`”。

我们使用的开发环境比较好，在图 14.4 中显示的代码中，凡是在预编译的时候会被删除的部分都会变成灰色。左边程序中第二个 `printf()` 语句变灰，会被删除。右边程序中第一个 `printf()` 语句变灰，会被删除。所以最终的输出就是图 14.4 中所示了。



图 14.4 预编译控制

14.4 文件包含

除了宏定义和预编译控制以外，C 语言还提供了另外一种预处理命令——文件包含。这个预处理命令可以告诉编译器在预处理的时候，将一个文件的内容全部复制到另一个文件中，是一种对程序修改幅度较大的预处理命令。本节就让我们来看看这种特殊的预处理命令。

14.4.1 头文件和源文件的文件名

对于公共代码和非公共代码的放置文件，C 语言是有一些讲究的。这里所谓的公共代码就是在我们所写的程序中将会经常使用的代码，非公共代码就是在程序中会很少出现，或者说只出现一次的代码。这就好比公园的锻炼器材和你自己买的跑步机一样，公园里的锻炼器材是大家都可以使用的，当然要放在公园这样的公共场所。你买的跑步机是你的私有财产，当然放在你家了。

在 C 语言中，放置公共代码的文件，被称为头文件，这类文件是以后缀“`.h`”结尾的，`h` 代表的就是“`header`”（头的意思）。放置非公共代码的文件，被称为源文件，这类文件是以后缀“`.c`”结尾的，`c` 代表的是 C 语言的意思。有的时候根据文件的后缀，我们也称头文件为“点 `h` 文件”，称源文件为“点 `c` 文件”。

对于头文件和源文件后缀之前的文件名，C 语言没有规定，只要是你用的系统可以识别的文件名就可以。如果你无法判断你所取的文件名是不是系统可以识别的，直接新建一个，新建成功则表示系统可以识别，否则就是系统不识别的。

对于头文件和源文件的后缀，“h”和“c”规定使用小写字母，不过有的地方也可以写成大写的“H”和“C”。尤其是 Windows 系统的文件不区分大小写，“test.h”和“test.H”被认为是同一个文件，会以大写的“H”作为头文件后缀，以大写的“C”作为源文件的后缀。虽然如此，但是使用大写的“H”和“C”并非通用规则，还是建议使用小写的“h”和“c”。

就像下面的头文件和源文件命令都是正确的：

```
test.h aac_dec.c flag-123.h FFGGG.h Tree.c
```

14.4.2 头文件和源文件的内容

我们经常将一些常用的函数声明、宏定义、公共结构体类型、共同体类型和枚举类型、全局变量等公共代码都放在头文件中。而将一些非公共的宏定义、结构体类型定义、共同体类型和枚举类型，以及函数定义都放在源文件中。

例如，要定义一个有关圆数学计算的程序，就可以将公共的代码和非公代码分开放在不同的文件中，来实现部分代码的重复使用。先新建一个名为 circle.h 的头文件，其中可以放置如下代码：

```
struct point
{
    int x;
    int y;
};

#define PI 3.14

//计算圆的周长
double Circumference(int radius);
//计算圆的面积
double Area(int radius);
//在屏幕上画一个圆
void Paint(struct point center, int radius);
```

在这个头文件中，定义了一个公共的结构体类型 point 来存放点信息，定义了一个经常使用的宏 PI，它将会被圆周率 3.14 替换。声明了三个函数，告诉使用这个头文件的程序，程序中是有这三个函数的，不过不在此处定义。这三个函数分别用来完成圆的周长和面积的计算，以及在屏幕上画圆的功能。如果需要这些代码中的一个或者多个，直接在需要的地方使用文件包含预处理命令包含这个头文件就可以了。

下面来看看源文件中非公有代码。先新建一个 circle.c 源文件，在其中放置如下代码：

```
//计算圆的周长的函数定义
double Circumference(int radius)
{
    return 2*PI*radius;
}

//计算圆的面积的定义
double Area(int radius)
{
    return PI*radius*radius;
```

```

}

//在屏幕上画一个圆的定义
void Paint(struct point center, int radius)
{
    //点亮屏幕中以(center.x , center.y)为原点, 以 radius 为半径的一片区域
    .....
}

```

在这个源文件中, 完成了三个函数的定义, 实现了所需的功能。对于在屏幕上画圆, 由于要使用操作系统提供的功能, 比较麻烦, 这里暂且省略。有人可能会问, 为什么要将函数的定义放到源文件中呢? 它们难道就不能公用吗? 确实不能公用! 因为 C 语言规定, 同一个函数只能定义一次, 如果你将函数的定义放在头文件中, 如果不加限制, 将来在预编译的时候被复制到多个源文件中, 就会造成同一个函数多次定义的错误。函数的声明没有这个限制, 所以, 一般都将函数的声明放在头文件中, 允许多处复制, 而将函数的定义放在源文件中, 避免函数的多处定义错误。

14.5 include 包含头文件

知道了如何分配不同的代码到头文件和源文件之后, 来看一看 C 语言中的文件包含预处理命令, 看看如何指导编译器将头文件的内容复制到源文件中。

14.5.1 自定义头文件和系统头文件

在看 C 语言的文件包含预处理命令之前, 先来看看 C 语言中的两类头文件: 自定义头文件和系统头文件。

- ❑ 自定义头文件, 就是在写代码的时候, 自己建的头文件。使用它的主要目的是为了更方便, 将一些公共的代码放在头文件里, 以便之后编译器复制其中的内容。
- ❑ 系统头文件, 是操作系统或其他公司机构为开发人员提供的头文件。使用它的主要目的是给开发人员提供一个接口, 以便其使用操作系统或者其他公司机构所提供的功能。

像之前为了完成有关圆的计算, 自己实现的头文件“circle.h”就是一个自定义头文件。之前在写代码的时候, 都会在代码的头部出现一个头文件“stdio.h”, 它就是一个系统头文件。经常使用的函数 printf() 和 scanf() 就是在这个系统头文件中声明的, 因为这两个函数实现是操作系统完成的, 我们只使用它们提供的功能。

14.5.2 文件包含的两种形式

知道了 C 语言中的两类头文件以后, 下面来看看 C 语言中的两种头文件包含形式, 这两种形式与 C 语言中的两类头文件有关。C 语言中的文件包含预处理命令关键字为 include, 文件包含预处理命令的形式有如下两种:

```
#include <头文件>
```

和

```
#include "头文件"
```

从形式上看，这两种形式的不同之处在于，头文件是在尖括号中还是在双引号中。形式不同，含义当然也不同。在作用上，两种形式的不同之处主要在于从什么地方（路径）开始找要被复制的头文件。

对于第一种，使用尖括号的文件包含预处理命令，它是从系统规定和用户设定的包含路径查找头文件的，找到了就进行复制，没找到就直接报错。

对于第二种，使用双引号的文件包含预处理命令，它是从当前的源代码文件放置的路径开始查找。如果找到了就包含，如果没有找到就去系统规定和用户设定的包含路径查找，如果还找不到的话才报错。

由于第一种形式是从系统规定和用户设定的包含路径查找头文件的，而这些路径中经常放的是系统头文件，因此一般系统头文件都使用第一种形式进行文件包含。第二种形式是先从源代码文件放置的当前路径开始查找的，而这个路径中一般放的都是自定义头文件，因此自定义头文件的包含一般都使用第二种形式。

根据上面的约束，我们之前见到的 `stdio.h` 头文件包含就使用第一种形式，而 `circle.h` 就可以使用第二种形式，具体如下所示：

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

14.5.3 完整的 circle 例子

有了上面这些关于 C 语言文件包含的知识以后，就可以使用文件包含来完成一个完整的程序了，还是使用那个圆数学计算的例子。为了简单，只保留计算圆周长和面积两个函数。新建三个文件：`circle.h`、`circle.c` 和 `main.c`，前两个文件的作用和前面讲的是一样的，完成圆的有关计算，`main.c` 的作用是完成 `main()` 函数定义。

三个文件的内容如下所示。

□ `circle.h` 头文件：

```
#define PI 3.14

//计算圆的周长
double Circumference(int radius);
//计算圆的面积
double Area(int radius);
```

以上是圆周率宏 `PI` 的定义，以及两个函数的声明。

□ `circle.c` 源文件：

```
#include "circle.h"

//计算圆的周长的函数定义
double Circumference(int radius)
{
    return 2*PI*radius;
```



```

}

//计算圆的面积的定义
double Area(int radius)
{
    return PI*radius*radius;
}

```

以上是两个函数的定义。此处多了一个自定义头文件包含预处理命令，是因为 PI 宏是定义在 circle.h 头文件中的。如果不包含这个头文件，程序就不认识标识符 PI 是什么。

□ main.c 源文件：

```

#include<stdio.h>
#include"circle.h"

int main()
{
    double result;

    result = Circumference(3);
    printf("the circumference is %lf\n",result);

    result = Area(3);
    printf("the area is %lf\n",result);

    return 0;
}

```

定义 main() 函数的 main.c 文件中，有两个头文件包含预处理命令，一个是包含系统头文件 stdio.h 的，一个是包含自定义头文件 circle.h 的。之所以要包含 stdio.h 头文件，是因为 printf() 函数在其中声明，如果不包含，则程序不认识 printf() 是什么。之所以要包含 circle.h 头文件，是因为计算圆周长的函数 Circumference() 和计算圆面积的函数 Area() 都在其中声明，如果不包含，程序就不认识它们是什么！

整个程序的运行结果如图 14.5 所示，正确计算出了半径为 3 的圆的周长和面积。



图 14.5 头文件包含

从某种程序上来说，头文件包含不仅仅提供了代码复制的功能，它也是一种很好的代码组织方式。通过这种方式，可以将不同类型和不同功能的代码放在不同的文件中，每个文件取一个合适的名字，各司其职。这样，无论是查阅还是修改代码都会变得井井有条。就像上面有关圆计算的例子，其实完全只用写一个 main.c 源文件，将所用的东西都放入其

中。不过那样会显得非常凌乱，代码量再大点，你可能就受不了了，太乱了。

14.5.4 C 语言中的标准头文件

前面讲到头文件的种类时，讲到了系统头文件，它是操作系统或者其他公司提供的头文件，用来使用操作系统或者是其他公司提供的一些功能。像之前见到的头文件 `stdio.h` 就是一个系统头文件，它是由操作系统提供的，主要用来完成输入 / 输出有关的功能。

在这里，“`stdio`”代表的是“`standard input and output`”（标准输入 / 输出），为什么要冠名为“标准”呢？是因为这个文件是 C 语言规范规定的头文件，所有遵循这个规范的 C 语言开发环境都得提供这个头文件。正因如此，这一类由 C 语言规范规定的头文件被称为 C 语言标准头文件。除了“`stdio.h`”这个头文件以外，C 语言还规定了其他的标准头文件，用来完成不同的功能。接下来就来看看 C 语言中提供的这些标准头文件吧。

表 14.1 列出了 C 语言最新标准所规定的标准头文件的名称其内容和实现功能简要描述。这些标准头文件基本囊括了写一个 C 语言程序所需要的基本功能，有了这些头文件，很多麻烦的事就都交给操作系统去完成了。我们只要使用文件包含预处理指令包含需要的头文件，使用其中提供的数据类型、宏和函数就可以了，至于头文件中提供的函数是如何实现的，就不用关心了，反正挺麻烦的。

表 14.1 C语言标准头文件

头文件名	内容和实现功能
<assert.h>	程序诊断：定义了 <code>assert</code> ， <code>static_assert</code> 宏和 <code>NDEBUG</code> 宏
<ctype.h>	字符操作：声明了字符类别和相关的函数
<complex.h>	复数运算：定义了复数运算的宏和函数声明
<errno.h>	错误标示：定义了有关出错的一些宏
<fenv.h>	浮点环境：提供了访问浮点环境的类型和宏，用于观察和设置浮点运算行为
<float.h>	浮点类型：定义了一些宏来扩展标准的浮点类型模型的范围和参数
<inttypes.h>	整型格式转化：包含<stdint.h>头文件并对其做了扩展
<iso646.h>	可选拼写：定义了位操作的 11 个宏，如 <code>#define</code> and <code>&&...</code>
<limits.h>	整型的尺寸：定义了一些宏来扩展标准整型模型的范围和参数
<locale.h>	本地化：声明了一些宏和函数，来获得和设置本地化信息，如本地时间格式，货币符号...
<math.h>	数学运算：定义了有关数学运算的宏，结构体和函数
<setjmp.h>	非局部跳转：通过了宏，类型和函数来绕过一般的函数调用和返回规则
<signal.h>	信号操作：提供了操作在程序运行时出现的信号的函数，宏和类型
<stdarg.h>	可变参数：提供了处理可变参数的宏，类型和函数
<stdbool.h>	Bool 类型和值：定义了 4 个宏： <code>bool</code> ， <code>_Bool</code> ， <code>true</code> 和 <code>false</code>
<stddef.h>	公共定义：定义了类似于 <code>NULL</code> 的宏和 <code>size_t</code> 的数据类型
<stdint.h>	整型：定义了整型的宽度和相应的宏的集合
<stdio.h>	输入输出：提供了输入输出的宏，类型和函数
<stdlib.h>	实用工具：提供了一般的通用工具和与之相关的宏，类型
<string.h>	字符串操作：提供了宏，类型和字符数组的处理函数
<tgmath.h>	通用类型数学计算：包含<math.h>和<complex.h>，以及一些通用类型的宏
<time.h>	日期和时间：提供了操作时间的宏，类型和函数
<wchar.h>	多字节和宽字节工具：提供了处理扩展多字节和宽字节的宏，类型和函数
<wctype.h>	宽字节字符的类型：类似于<ctype.h>，提供了宽字节类型有关的宏，类型和函数

先大概看看表 14.1 中这些标准头文件实现了什么功能，对于每个头文件具体提供了什么数据类型、宏和函数，大家需要的时候查阅相关的开发文档就可以知道了。另外，有一点得强调一下，由于这个表里列的标准头文件是最新标准所规定的标准头文件，因此，并不是现在所有的开发工具都支持这些头文件的，可能有的会没有。我现在所见的，除了处理复数的头文件“`complex.h`”外，其他头文件基本每个开发工具都支持。

下面来看初学者较常使用的几个头文件：`math.h`、`stdio.h`、`stdlib.h` 和 `string.h`。

1. `math.h`头文件

这个头文件提供了有关数学运算的宏、类型和函数。例如，要计算一个数的平方，就可以直接使用 `pow()` 函数；要计算一个数以 2 为底的对数，就可以使用函数 `log()`；要计算一个数的 `sin` 值，就可以使用函数 `sin()` 等。诸如此类数学运算的函数，系统都已经帮你实现了，只要使用文件包含预处理指令包含 `math.h` 这个标准头文件就可以了。

2. `stdio.h`头文件

这个头文件主要提供了有关输入/输出的宏、类型和函数。所谓输入/输出，就是从—个地方把数据读到另一个地方。这里的“地方”是计算机中的不同设备，如内存、显示器、磁盘等。而这里的入和出都是相对于内存而言的，如果是从—个地方读入内存的就是输入，是从内存写入其他地方的就是输出。例如，之前使用的 `scanf()` 输入函数，就是把数据从键盘读入内存的；`printf()` 函数就是从内存把数据输出到显示器的终端窗口的。这个头文件中还提供了一些其他的输入/输出函数，如果需要，可以查阅相关的开发文档。

3. `stdlib.h`头文件

这个头文件提供的内容比较杂，它主要提供一些辅助功能的函数，或者称之为工具。例如，将一个字符串转换成整型数字可以使用函数 `atoi()`；产生一个随机数就可以使用 `rand()` 函数；问计算机要一块内存的使用权可以使用 `malloc()` 函数等。诸如此类的函数就是在 `stdlib.h` 这个头文件中声明的。

4. `string.h`头文件

这个头文件主要提供字符串操作的宏、类型和函数。例如，要将两个字符串拼接起来，可以使用 `strcat()` 函数；对两个字符串进行比较，可以使用函数 `strcmp()`；将一个字符串复制到另一个地方，可以使用函数 `strcpy()` 等。诸如此类的字符串操作函数都是声明在 `string.h` 头文件中的。

对于每个头文件中的每个函数的具体使用，也就是函数需要传入的参数、函数完成的功能及函数的返回值是什么意思，如果不清楚，还是建议查阅相关的开发文档。下面举一个例子，来使用一下 C 语言标准头文件中的几个典型的函数。

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

int main()
{
```

```

double num = 2.6;
double result;
int str_num;

char* str = "1000";
char buf[20];

//cos() function int math.h
result = cos(num);
printf("cos(%lf) = %lf\n", num, result);

//atoi() function in stdlib.h
str_num = atoi(str);
printf("string : %s to int : %d\n", str, str_num);

//strcpy function in string.h
strcpy(buf, str);
printf("str [%s] is copied to buf [%s]\n", str, buf);

return 0;
}

```

在这个程序中，除了 `main()` 函数之外，没有定义自己的函数，使用的都是系统提供的函数，为了使用这些函数，使用了文件包含预处理命令包含了所需的头文件。分别使用 `cos()` 函数计算 2.6 的 `cos` 值，使用 `atoi()` 函数将字符串“1000”转换成整型数值 1000，使用 `strcpy()` 函数将字符串“1000”复制到字符数组 `buf` 中，并分别使用 `printf()` 函数输出相应信息到终端，程序的输出如图 14.6 所示。正确完成了相应的功能，说明我们正确地使用了这些系统提供的函数，否则程序就会出错。

```

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

int main()
{
    double num = 2.6;
    double result;
    int str_num;

    char* str = "1000";
    char buf[20];

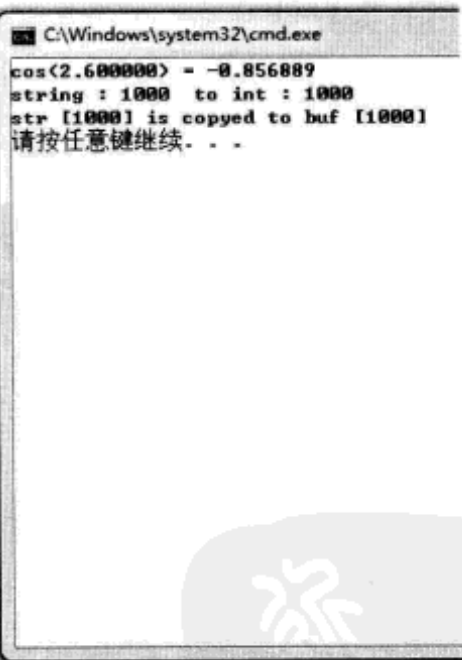
    //cos() function int math.h
    result = cos(num);
    printf("cos(%lf) = %lf\n", num, result);

    //atoi() function in stdlib.h
    str_num = atoi(str);
    printf("string : %s to int : %d\n", str, str_num);

    //strcpy function in string.h
    strcpy(buf, str);
    printf("str [%s] is copied to buf [%s]\n", str, buf);

    return 0;
}

```



```

C:\Windows\system32\cmd.exe
cos(2.600000) = -0.856889
string : 1000 to int : 1000
str [1000] is copied to buf [1000]
请按任意键继续. . .

```

图 14.6 标准头文件的使用

14.6 小 结

在本章中主要讲述了 C 语言中的预处理命令的使用。预处理命令主要有宏定义、预编译控制和头文件包含三种，因此本章的重点就是这三种预处理命令的形式和含义。本章的

难点就是三种预处理命令的使用，比较麻烦的就是带参宏定义、预编译控制和自己定义头文件。在下一章中，作为对标准头文件的使用，我们主要讲述C语言中的另一个重要知识点——文件。

14.7 习 题

【题目1】 不使用预处理命令，可以写C语言代码吗？

【分析】 预处理命令所起的作用，是在编译之前，对我们所写的代码进行局部或者全局的修改，以适应不同的环境，或者使用系统提供的功能。如果既不需要使用不同的环境，也不需要系统提供的功能，完全可以不使用预处理命令。预处理命令只是简化和方便我们写代码的C语言工具而已，用不用完全由你自己决定！

【题目2】 使用预处理命令将你所写代码中的符号“LOVE”都替换成字符串“I Love You So Much! ”，并输出这一字符串！

【分析】 C语言中，用于简单的符号替换的预处理命令是不带参宏定义，关键字是define，参考define宏定义的表示形式，就可以写出需要的预处理命令代码了！

【核心代码】

```
#include<stdio.h>

#define LOVE "I Love You So Much!"

int main()
{
    ...
    printf(LOVE);
    return 0;
}
```

【题目3】 写一段代码，如果程序中定义了BELIEVE宏，将符号“LOVE”使用字符串“I Love You So Much!”替换。如果没定义BELIEVE宏的话，将符号“LOVE”使用字符串“I Love You So Much, Silently!”替换，并输出符号“LOVE”所代表的字符串！

【分析】 C语言中，要根据情况来判断该将符号具体替换成什么，就需要预编译控制宏了，有点类似于判断结构。参照预编译控制宏的使用方式，就可以完成这个任务了！

【核心代码】

```
#include<stdio.h>

#ifdef BELIEVE
#define LOVE "I Love You So Much!"
#else
#define LOVE "I Love You So Much, Silently!"
#endif

int main()
{
    ...
    printf(LOVE);
    return 0;
}
```


【题目 4】 写一段程序，问计算机要 4 个字节的内存空间，并往这个空间存入整型数值 10，然后，将这个空间中的数据输出。

【分析】 向计算机申请内存空间需要操作系统提供的函数 `malloc()`，这个函数的声明是包含在标准头文件 `stdlib.h` 中的。函数的声明形式为 `void * malloc (size_t size)`，`size_t` 是一种表示尺寸的数据类型，以字节为单位，你可以把它当作无符号整型来使用，`malloc()` 函数的返回值是一个 `void` 指针，表示的是计算机给你的内存空间的地址。

C 语言中，一般问计算机要的内存空间在使用完之后，都是要还回去的，需要使用 `free()` 函数来归还，这个函数的声明也是包含在标准头文件 `stdlib.h` 中的。`free()` 函数的声明形式为 `void free (void * ptr)`，参数是要归还的内存空间的地址，也就是 `malloc()` 函数返回的地址。在使用 `malloc()` 和 `free()` 这两个函数之前得使用文件包含预处理命令将 `stdlib.h` 这个头文件包含到需要使用这两个函数的源文件中，否则编译器不知道 `malloc()` 和 `free()` 这两个函数到底是什么！

【核心代码】

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

int main()
{
    int *pdata;
    ...

    pdata = (int *)malloc(4);
    *pdata = 10;
    printf("%d\n",*pdata);
    free(pdata);
    ...

    return 0;
}
```

【题目 5】 绘制余弦曲线：在屏幕上使用 “*” 号显示出 $0\sim 360^\circ$ 的余弦函数 $\cos(x)$ 的曲线，不要使用数组。图 14.7 中，我们绘制画出了一个简单的 \cos 曲线图形，横坐标是弧度值 $0\sim 6.3$ ，相当于角度值的 $0\sim 360^\circ$ ；纵坐标为横坐标对应的 \cos 函数值，从 -1 到 1。

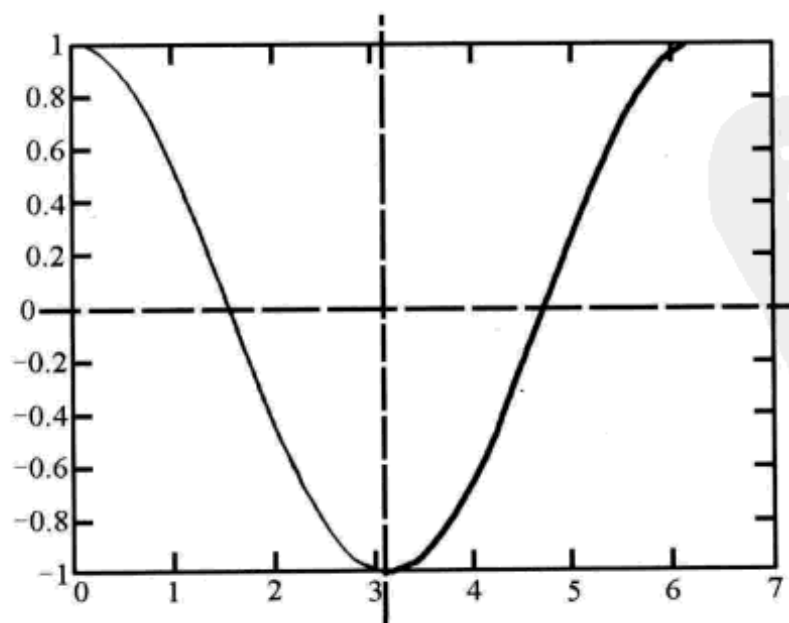


图 14.7 \cos 曲线

【分析】 要绘制余弦曲线，就是在屏幕上确定一个一个的余弦函数的坐标点。我们使用屏幕的竖直方向为 Y 轴，水平方向为 X 轴；对于 $y=\cos(x)$ ， x 在 $0\sim 360^\circ$ 之间，那么， y 则在 -1 到 1 之间，假设屏幕上每一行的宽度为 0.1 ，也就是每一行的 y 值间隔为 0.1 。

假设每一行的一个字符的宽度为 0.1 弧度，那么 $0\sim 360^\circ$ 约是 $0\sim 6.3$ ，也就是 x 轴上最多有 63 个字符。根据余弦函数的对称性，我们可以先画出对称轴左边的点，再画出对称轴右边的点，就可以画出一个完整的余弦函数了。写代码的时候，使用了 `acos()` 函数计算对应 y 处的 x 的值，这个函数声明在 `<math.h>` 头文件中，使用的时候要记住使用 `include` 预处理指令包含这个头文件。

【核心代码】

```
double y;
int i, x;
for(y = 1; y>=-1; y-=0.1)
{
    x = acos(y)*10;
    for(i=1; i<x; i++) printf(" ");
    printf("*"); //画对称轴左边的余弦点
    for(; i<63-x; i++) printf(" ");
    printf("*\n"); //画对称轴右边的余弦点
}
```

【题目 6】 绘制正弦曲线：在屏幕上使用 “*” 号显示出 $0\sim 360^\circ$ 的正弦函数 $\sin(x)$ 的曲线，不要使用数组。在图 14.8 中，绘制画出了一个简单的 \sin 曲线图形，横坐标是弧度值 $0\sim 6.3$ 相当于角度值的 $0\sim 360^\circ$ ；纵坐标为横坐标对应的 \sin 函数值，从 -1 到 1 。

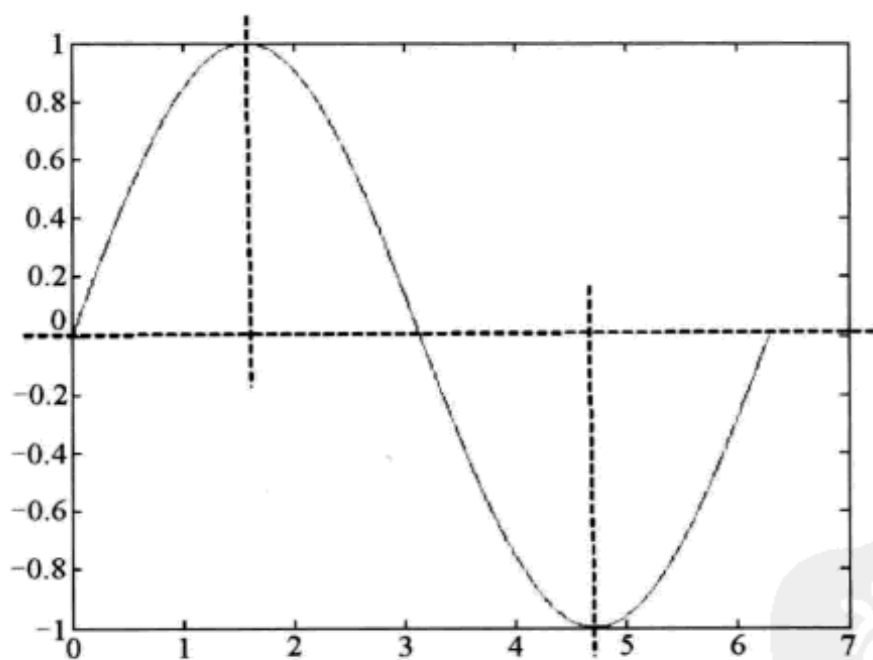


图 14.8 \sin 曲线

【分析】 绘制正弦曲线和绘制余弦曲线是有点区别的，因为正弦曲线在 $0\sim 360^\circ$ 上，不是关于 Y 轴对称的，因此不能分两边对称来画了。不过，我们可以参考绘制余弦曲线的方法，稍作修改来完成正弦曲线的绘制。对于 $y=\sin(x)$ ， x 在 $0\sim 180^\circ$ 之间时， y 是在 0 到 1 之间， x 在 $180\sim 360^\circ$ 之间时， y 则是在 0 到 -1 之间。假设屏幕上每一行的宽度为 0.1 ，也就是每一行的 y 值间隔为 0.1 。

假设，每一行的一个字符的宽度为 0.1 弧度，那么 $0\sim 360^\circ$ 约是 $0\sim 6.3$ ，也就是 x 轴上最多有 63 个字符。 y 在 0 到 1 之间时，正弦函数在这个范围是关于 y 轴对称的，我们可

以先画出对称轴左边的点，再画出对称轴右边的点。同样，对于 y 在 0 到 -1 之间的时候，也是一样的。

写代码的时候，使用了 `asin()` 函数计算对应 y 处的 x 的值，这个函数声明在 `<math.h>` 头文件中，使用的时候要记住使用 `include` 预处理指令包含这个头文件。

【核心代码】

```
double y;
int i, x;
for(y = 1; y >= 0; y -= 0.1)
{
    x = asin(y)*10;
    for(i=1; i<x ; i++) printf(" ");
    printf("*");           //画-对称轴左边的正弦点
    for(; i<31-x; i++) printf(" ");
    printf("*\n");         //画-对称轴右边的正弦点
}

for(y = 0 ; y >= -1; y -= 0.1)
{
    x = asin(y)*10;
    for(i=1; i<32-x ; i++) printf(" ");
    printf("*");           //画-对称轴左边的正弦点
    for(; i<63+x; i++) printf(" ");
    printf("*\n");         //画-对称轴右边的正弦点
}
```

【题目 7】 编程检查输入表达式中的圆括号是否配对。例如表达式

((a+(b*9))(a+B))

其中的小括号是不是配对的？写段代码来实现这样的检查。

【分析】 可以使用一个变量做记号，记录当前左括号的个数，每遇到一个右括号，左括号的个数减一。对输入的表达式挨个字符进行处理，遇到括号时对做记号的变量的值进行改动，中间标记变量如果出现负值，表明目前右括号多于左括号，不配对。处理完后如果变量值为 0，则表明是配对的，大于 0 表示左括号比右括号多，括号不配对，小于 0 表示左括号比右括号少。

在进行输入的时候，我们使用一个特殊的系统函数 `getchar()`。这个函数在标准头文件 `<stdio.h>` 中有声明，使用的时候必须使用 `include` 预处理指令来包含这个头文件。

【核心代码】

```
char ch;
int m=0; /*m 用来记录左括号个数*/
while(ch=getchar(), ch!='\n')           //当读入的字符不为换行符时，继续处理
{
    if(ch=='(') m++;                     //读入左括号 m 加一
    if(ch==')') m--;                     //读入右括号 m 减一
    if(m<0) break;                       //m 出现则表示目前的右括号比左括号多，不配对，直接跳出
} //读入非括号字符时不予处理
if(m==0) printf("'('=='')'\n");         //m==0 即括号是配对的
else if(m>0) printf("'('>')'\n");       //左括号多于右括号
else printf("'('>'')'\n");              //右括号多于左括号
```


第 15 章 文件操作

在计算机中，通常都是将数据保存在文件中的。需要处理的时候，使用特定的工具将其打开，对此进行处理。处理完之后，如果有数据要保存，就将其保存在打开的文件中，或者另存为其他文件，如果没有数据要保存，直接关闭就可以了。不过，之前我们操作文件都是使用的特定计算机软件，像 Word、记事本、画图板、播放器等。现在学习 C 语言了，来看看用 C 语言是如何操作文件的，也就是我们使用的那些软件是如何做出来的。

15.1 文 件

在计算机中，文件是一个很重要的概念，虽然使用计算机的时候经常遇到，但是我们看到的文件和计算机看到的文件未必是一样的。接下来就来重新认识一下我们所熟知的文件。

15.1.1 重新认识文件

感性一点来说，一旦打开计算机，在每个磁盘、每个文件夹中，所看到的都是文件，如经常使用的以后缀“.txt”结束的记事本文件，以后缀“.doc”结束的 Word 文件，以后缀“.exe”结束的可执行程序文件等。你打开电脑随便一扫，到处都是文件，各种文件。

理性一点来看，文件就是存放在磁盘上的一堆数据，而且这堆数据是以一组 0 和 1 的形式存在于磁盘上的。为了便于查找显示这一堆堆的数据，给每一堆数据都取了个名字，这个名字被称为文件名。我们在新建文件的时候，给要建的文件取个名字，然后把一堆数据放到磁盘，计算机操作系统就负责把这个名字和这堆数据管理起来，保证使用这个名字可以访问到你需要的数据。这一堆数据就是我们所说的文件。

按文件的后缀可以将文件分为不同的类型，如 TXT 文件、DOC 文件、JPG 文件、MP3 文件、MP4 文件、EXE 文件，这是一种感性的分类，看见什么后缀就是什么样的文件，结果导致文件类型太多了，你自己都可以发明一种文件。在有的操作系统上是不认文件后缀的，只认文件里的数据，只要文件里面的数据符合要求就可以正确使用，根本不管文件的后缀到底是什么。例如，你完全可以把一个可执行文件以后缀“.mp3”结束，然后把它交给计算机操作系统，照样可以运行。

在文件命名的时候之所以使用不同的后缀，是为了把文件 and 对应打开文件的程序关联起来，让你仅仅使用鼠标双击一下就可以使用正确的程序打开文件，而不用关心该使用什么样的程序打开怎样的文件。其实，文件后缀是操作系统为了使用户更好地使用文件所提

供的一种方式，计算机操作系统对文件操作的时候是不认文件后缀的，它只管文件中的数据。因此，会出现以“.mp3”为后缀的文件照样可以运行正常的情况！

15.1.2 计算机眼里的文件

说了这么多终于可以转入正题了，前面的讲解主要是想让大家从一个一般的文件使用者视角转换到计算机的视角来看待文件，而这个计算机的视角正是程序的视角，也就是一个程序开发人员的视角。从计算机的角度来看，每一个文件都是一堆的0和1，再加一个文件名。把这一堆的0和1按照一定的格式解析出来就是有用的信息，解析格式正确就可以得到正确的信息，解析格式不正确得到的就是不正确的信息，我们经常见到的文件乱码或者文件无法播放基本原因都是解析格式不正确！

如果一个文件能够按照ASCII码或者其他基本的编码将其中的0和1解析成我们可以认识的字符或者符号，这类文件就被称为无格式文件，因为它只有简单的字符编码格式。如果一个文件只使用ASCII码或者其他基本的编码无法解析成我们可以识别的信息，就称之为有格式文件，因为它在最基本的字符编码格式中添加了自己的格式。

像常见的TXT文件就是无格式文件，DOC文件和EXE文件都是有格式文件。判断一个文件是不是无格式文件，最简单的方法就是使用记事本将其打开，如果不显示乱码就是无格式文件，如果显示乱码就是有格式文件。记事本就是一个将文件保存的0和1按照最简单的字符编码的格式解析出来展示给大家的软件，并且可以将大家写入的字符按照基本字符编码变成0和1保存到磁盘。

在图15.1中，展示了一个有格式文件test.doc和一个无格式文件test.txt。这幅图中总共共有三部分，最上面的部分分别是用Word打开的test.doc文件和使用winhex软件打开的test.doc文件。winhex是显示一个文件在磁盘上的0和1数据的十六进制表示，可以使用它看到文件到底存了什么，也就是文件的本质。中间的部分分别是用记事本和winhex打开的test.txt文件。最下面的部分是使用记事本打开的test.doc文件，无法解析，所以显示乱码。

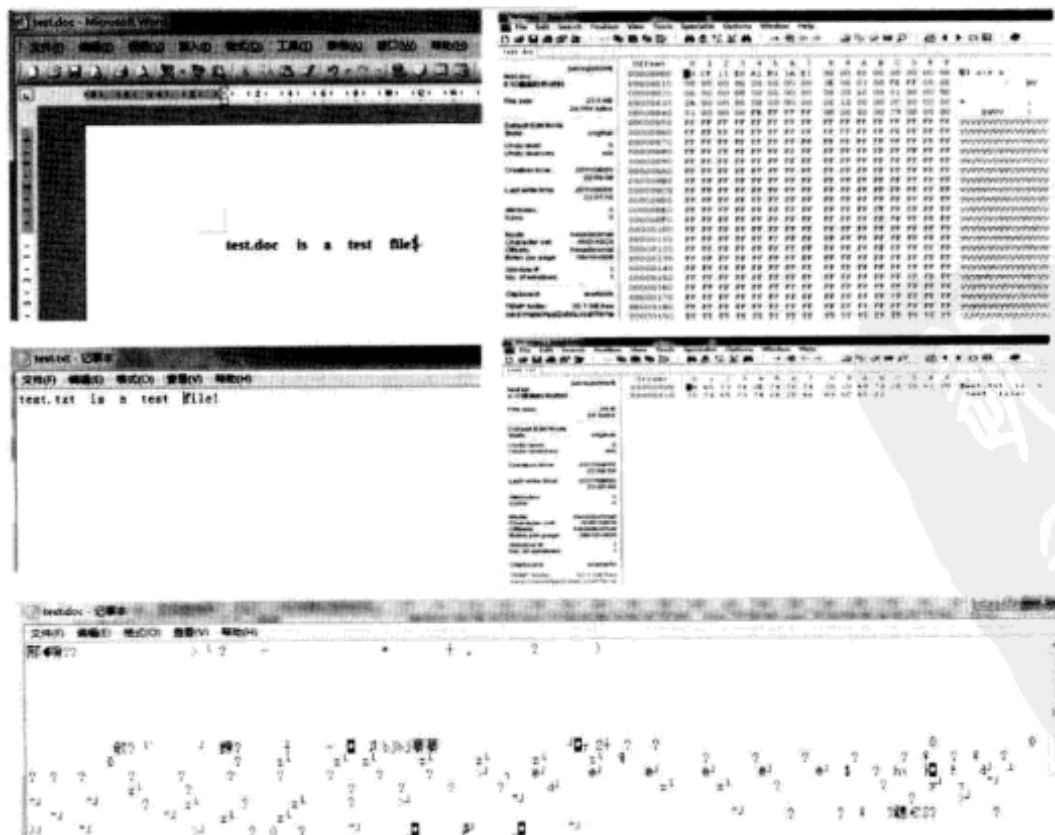


图 15.1 文件

15.1.3 开发人员能对文件干什么

好了，对文件有了由浅到深、由表及里的了解之后，下面来看看作为一个开发人员，我们能对文件做些什么，也就是能对文件做些什么操作。在开发人员的眼里，或者说在 C 语言的眼里，文件就是一堆 0 和 1，外加一个文件名。

因此，开发人员能对文件做的最基本的事情就是从磁盘将这一堆 0 和 1 读入程序中，至于如何解析这些 0 和 1，就得看你的应用了。每种文件的格式都是不同的，要解析需要专业的知识。当然也可以把程序中的数据，按照一定的格式转换成 0 和 1 存入磁盘里。这里的读出和写入被称为文件的读写操作。在读写之前我们需要计算机帮我们做一些准备工作，在计算机中称这些准备工作为文件打开；使用完文件之后，我们还需要计算机帮我们做一些善后工作，这些善后工作被称为文件关闭。除了文件的打开、关闭、读写之外，计算机还提供了一些其他的文件操作功能。

计算机提供的这三类文件操作，是可以使用 C 语言实现的，具体的实现留待后面详细讲解。

15.2 文件的打开与关闭

在对文件进行处理之前，得先打开文件；处理完文件之后，还得关闭文件。就像我们对 Word 文档进行编辑之前，得先使用 Office Word 软件打开要编辑的文档，编辑完之后，还要关闭编辑的软件一样。双击 Word 文件就可以打开 Word 文档，单击 Office Word 右上角上的叉号“×”就可以关闭编辑完成的 Word 文档。在使用 Office Word 软件操作 Word 文档时，可以使用简单的双击和单击“×”来打开和关闭文件。在 C 语言中又是怎么做的呢？带着这个问题我们来学习这一节！

15.2.1 文件指针

在看使用 C 语言如何打开关闭一个文件之前，先来想一个问题。我们在使用软件打开文件的时候需要指定文件名，也就是哪个目录下的哪个文件，因为使用文件路径和文件名就可以确定文件了。类似地，程序中是使用什么来确定文件的？还是文件路径和文件名吗？不是了，C 语言使用一种新类型的数据——文件指针，来标识和确定文件。

文件指针类型是用来表示一个指向文件信息的指针，就像整型指针是指向整型数据所在的地址的，文件指针是用来指向一种新类型数据所在内存的地址的，这种类型就是文件类型。文件类型是 C 语言自定义的一种类型，使用这种类型来保存文件的相关信息，有了这些信息，计算机就可以很方便地确定一个文件了。不过，一般不需要知道这些信息的具体内容，只要能告诉计算机这些信息在什么地方放就可以了。所以，只要知道保存需要的文件类型数据的内存地址就可以了，而这个内存地址正是文件指针。

在 C 语言中，文件指针是保存在文件指针类型变量中的。定义一个文件指针变量的 C

语言表示形式如下所示：

```
FILE * 文件指针变量名;
```

在这个表示形式中，FILE 是 C 语言中的文件类型名，是 C 语言自定义的，必须全是大写字母，不容更改。“*”是指针类型表示符号。“文件指针变量名”是一个 C 语言标识符，只要遵循 C 语言标识符命名规范就可以了。

例如，定义一个文件指针变量 pfile 来保存一个文件有关的信息所在的内存地址，就可以使用下面的变量定义表示。

```
FILE *pfile;
```

15.2.2 文件打开函数

我们知道文件指针是用来保存文件信息所在的内存地址的，那么这个地址是怎么得到的？怎么知道它保存在内存的什么地方呢？我们还知道，C 语言中的文件打开操作是计算机为文件操作所做的前期处理，这些前期的处理又干了什么呢？这两个问题，现在就可以统一于 C 语言中的文件打开函数了！

C 语言中是使用文件打开函数来完成文件打开任务的，而文件打开函数所实现的功能就是根据文件路径和文件名来建立计算机所需要的文件信息。这里的文件路径和文件名是我们使用的，而这里的文件信息是给计算机使用的，文件打开函数会将这些信息保存在内存中，然后把内存的地址告诉我们。之后，在让计算机对这个文件进行操作的时候，只要告诉计算机这个地址就可以了，这个地址所在的地方有计算机需要的信息。

这就好比武侠小说里面讲的，在某某墓穴里有本古书，有了这本古书就可以重新召集某些武艺高强的侠客，让他们完全替你效劳。这里的“文件名”就相当于“古书名”，“文件所在地址”就相当于“古墓所在的地方”，“计算机所在的信息”就相当于“古书里的内容”。不需要关心计算机需要的文件信息就如同寻找古书的人不需要关心古书的内容一样。我们使用文件打开函数得到文件信息的地址，就像武侠小说里所使用的藏宝图一样，使用它就可以知道古墓所在的地方……

你可以尽情遐想，不过类比到这里大家应该知道文件打开函数是做什么的了。简单地说就是使用它由文件名得到文件指针。现在就来看一看这个神秘的文件打开函数长什么样，该怎么使用。

```
FILE *fopen( const char *fname, const char *mode );
```

这就是 C 语言中的文件打开函数的声明。fopen 为函数名，f 表示文件的意思，open 为打开的意思。函数有两个字符指针类型的参数，参数变量之前使用了变量修饰符 const，表示字符指针所指的地址中的内容不能修改。函数的返回值是一个文件指针类型，这个指针所指的地方就保存着文件信息。

接下来讲解一下文件打开函数的字符指针参数。

1. 参数fname

fname 是一个字符指针变量，用来指向一个字符串的地址，而这个字符串正是我们经

常见到的文件路径和文件名，表示要打开的文件在计算机磁盘的什么地方放着。可以用两种方式来表示这个字符串：绝对路径和相对路径。

(1) 绝对路径就是从文件根目录算起的路径，如在 Windows 上，有一个文件存放在 D 盘下，那么它的绝对路径就从 D 盘的根目录算起，例如：

```
"D:\\Program Files\\Microsoft Office\\OFFICE11\\test.doc"
```

在这里之所以使用双斜杠是因为“\\”斜杠在 C 语言中有特殊用途，是特殊字符的转义字符表示。所以，第一个“\\”表示是转义符号，第二个“\\”才表示真正的“\\”。在 C 语言中，“\\”相当于字符‘\’。

(2) 相对路径是从程序运行的地方算起的路径。例如，我们写的程序在 D:\Program Files\Microsoft Office\OFFICE11 的地方运行，就可以直接使用相对路径，例如：

```
"test.doc"
```

文件打开函数中的参数 `fname`，使用这两种路径的字符串表示都是可以的。

2. 参数mode

`mode` 也是一个字符指针变量，也是用来指向一个字符串的，用来表示以怎样一种方式（模式）打开文件，表 15.1 列出了 C 语言中所有的文件打开方式及其含义，主要差别在于是以二进制形式打开，是以读、写还是追加（尾部写）形式打开。你需要哪一种，就直接将 `Mode` 一列中的字符串作为 `mode` 参数的实参就可以了。

表 15.1 文件打开方式

Mode	含 义
r	打开一个文件只用来读数据
w	打开一个文件只用来写数据，如果文件不存在就创建这个文件
a	打开一个文件，从其结尾开始写数据
rb	以二进制的形式打开一个文件，只用来读二进制数据
wb	以二进制的形式打开一个文件，只用来写二进制数据，如果文件不存在就创建这个文件
ab	以二进制的形式打开一个文件，从其结尾开始写二进制数据
r+	打开一个文件，既可以读数据也可以写数据
w+	打开一个文件，既可以读数据也可以写数据，如果文件不存在就创建这个文件
a+	打开一个文件，既可以读数据也可以写数据，写数据的时候从文件尾部开始写
rb+	以二进制的形式打开一个文件，既可以读数据也可以写数据
wb+	以二进制的形式打开一个文件，既可以读数据也可以写数据，如果文件不存在就创建这个文件
ab+	以二进制的形式打开一个文件，既可以读数据也可以写数据，写数据的时候从文件尾部开始写

知道了文件打开函数 `fopen()` 函数的形参列表的含义及返回值的作用以后，就可以使用这个函数来打开文件了。例如，要打开的文件路径为 `D:\Program Files\Microsoft Office\OFFICE11\test.doc`，而且程序正好在路径 `D:\Program Files\Microsoft Office\OFFICE11` 下，就可以使用相对路径表示的字符串了。如果仅仅是想读 `test.doc` 文件中的数据，就可以用只读方式打开文件使用模式字符串 `r`。打开文件以后，计算机交给我们一个文件指针，表示文件信息所在的内存地址，我们得使用一个文件指针变量妥善保管。具体的 C 语言表示如下所示：


```
FILE *pfile;  
pfile = fopen("test.doc", "r");
```

15.2.3 文件关闭函数

在对文件操作完之后，得告诉计算机做一些善后工作，这些工作主要是清除当初问计算机所要的文件信息。就像前面讲的武侠小说中的例子，故事的结尾往往是古书被销毁，以免再危害武林。

C语言中，是使用一个函数来完成文件关闭的善后工作的，即 `fclose()` 函数，其声明如下所示：

```
int fclose( FILE *stream );
```

文件关闭函数要比文件打开函数简单得多，参数是一个文件指针变量，而这个指针变量正是文件打开的时候，计算机交给我们保存打开文件的信息的内存地址，我们就是使用这个地址让计算机完成关闭操作的。函数的返回值是一个整型数值，C语言规定，如果关闭文件成功，则返回值为 0，否则返回 EOF，EOF 是 C 语言中的宏定义，也是一个整型数值，至于是多少就不需要关心了。

好，有了文件关闭函数 `fclose()`，就可以使用这个函数来关闭我们之前打开的 `test.doc` 文件了，具体的 C 语言表示如下所示：

```
FILE *pfile;  
pfile = fopen("test.doc", "r");  
...  
//其他操作  
...  
fclose(pfile);
```

15.3 文件读写

计算机中使用文件的主要目的是为了将数据保存在磁盘上，这样数据就可以被保存很长的时间，只要需要，随时都可以读出来。从磁盘上把数据读出来的操作被称为文件读操作；把数据写到磁盘上的操作被称为文件写操作。C语言提供了几对函数分别实现对文件的不同层次的读写，接下来就来看看这些函数。

15.3.1 读写一个字符

先来看一对最简单的文件读写函数，这一对函数只能从文件中读一个字符或者将一个字符写入文件，除此之外再不能操作更多的文件数据了。

1. `fgetc()` 函数

`fgetc()` 函数是 C 语言提供的用来从一个打开的文件读出一个字符的函数。第一次调用这个函数的时候，它从文件的开头读一个字符，之后的调用就是读出紧跟着的下一个字符。

fgetc()函数的声明形式如下所示:

```
int fgetc( FILE *stream );
```

fgetc 是函数名, f 表示 file, get 表示获取, c 表示 character, 意思就是从文件中获取一个字符。函数只有一个参数, 是一个文件指针, 表示的就是需要从这个文件指针所确定的文件中获取一个字符。函数的返回值是一个整型数据, 如果读操作正确执行, 这个整型数据正是读出的字符, 只不过是读出字符的 ASCII 码值而已, 可以使用类型转换将其转换成字符型数据, 如果读操作出现错误或者读到文件尾部无数据可读了, 这个整数值就为宏 EOF 所定义的整数。

使用 fgetc()函数的时候, 只能从以读模式或者是读写模式打开的文件中获得数据, 否则就会出错。

2. fputc()函数

fputc()函数是 C 语言提供的往文件中写一个字符的函数。每次调用 fputc()都会将要写的字符写到文件的最后面。fputc()函数的声明形式如下所示:

```
int fputc( int ch, FILE *stream );
```

fputc 是函数名, put 是放置的意思, 其他两字母的含义和 fgetc 是一样的, 函数名的意思就是将字符放置到文件。这个函数有两个参数, 一个是整型变量 ch, 表示的是要往文件中写的字符的 ASCII 码值, 另一个是一个文件指针变量, 表示的就是要往这个文件指针所确定的文件中写一个字符。函数的返回值是一个整型数值, 如果写入成功, 则返回值为写入的字符的 ASCII 码值, 如果出错, 则返回值为 EOF 宏所定义的整型数值。

另外, fputc()函数只能针对以写模式、读写模式或者追加模式打开的文件, 针对读模式打开的文件进行写操作就会出错。

15.3.2 读写一个字符串

C 语言提供的另一对进行文件读写操作的函数有比 fgetc()和 fputc()更强的功能, 它们可以一次从文件中读或者写一个字符串。

1. fgets()函数

fgets()函数可以实现从一个打开的文件中读一个字符串, 每次调用都从之前已读的字符串之后开始读一个字符串。fgets()函数的声明形式如下所示:

```
char *fgets( char *str, int num, FILE *stream );
```

fgets 为函数名, s 表示的是 string, 其他字母的含义与 fgetc()函数是一样的, 整个函数名的意思就是从文件中获取一个字符串。这个函数有三个参数, 第一个参数是一个字符指针变量, 它表示的是读出的字符将会被放置在这个字符指针所指位置开始的一片内存里, 所以它往往是一个字符数组名。第二个参数表示的是将会读出多少个字符, 最多 num-1 个, 第 num 个字符是给字符串结束符'\0'留着的。第三个参数是一个文件指针变量, 用它来确定要从哪个文件中读一个字符串。函数的返回值是一个字符指针, 如果函数读数据成功, 返

回值为存放读出字符串所在的内存地址，否则，返回值为由 NULL 宏定义的指针值。

和 fgetc()函数一样，fgets()函数只能从以读模式或者是读写模式打开的文件中获得数据，否则就会出错。

2. fputs()函数

fputs()函数是用来将一个字符串写入文件中的函数，每次调用都将字符串写入文件的末尾。fputs()函数的声明如下所示：

```
int fputs( const char *str, FILE *stream );
```

“fputs”为函数名，每个字母的含义可以通过前面几个文件读写函数名看得出来，表示的是往文件中放置一个字符串。这个函数有两个参数，第一个参数是一个字符指针，表示的是要放置的字符串所在内存的地址，第二个参数是一个文件指针变量，表示的是要往这个文件指针所确定的文件中写一个字符串。

和 fputc()函数一样，fputs ()函数只能针对以写模式、读写模式或者追加模式打开的文件，针对读模式打开的文件进行写操作就会出错。

15.3.3 读写一个数据块

C 语言中提供的另一个最灵活、也是 C 语言中最常使用的文件读写函数，是一次从一个打开的文件中读写一个数据块。

1. fread()函数

fread()函数是从一个打开的文件中读取一块数据，每次调用都读出紧跟已读块之后的一个数据块。这个函数的声明如下所示：

```
int fread( void *buffer, size_t size, size_t num, FILE *stream );
```

fread 为函数名，f 表示的是 file，read 是读的意思，整个函数名表示的是读文件的意思。这个函数的参数有点多，总共有 4 个，第一个参数是一个 void 指针变量，表示的是读出的数据将会被放到这个指针所指的地方。第二个参数是一个 size_t 类型的变量，是一个尺寸大小，表示要读出的数据块是以 size 个字节为单位的，你可以把这个变量当作整型变量来使用。第三个参数，还是一个 size_t 类型的变量，表示的是要读出的数据块的大小为 num 个单位，因此整个块的大小为 size*num 个字节。最后一个参数是一个文件指针变量，表示的是要从这个文件指针所确定的文件中读数据。

函数的返回值是一个整型数据，表示的是真正读出的数据块的大小，如果这个数值小于 size*num，表示出错或者已经读到文件尾部了，这个时候可以使用 feof()和 ferror()函数来检查错误。当然，fread()函数和 fgetc()、fgets()函数一样，只能从以读模式或者是读写模式打开的文件中获得数据，否则就会出错。

2. fwrite()函数

fwrite()函数实现将一个数据块写入一个已经打开的文件，每次都把数据写入文件的尾

部，函数的声明形式如下所示：

```
int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

`fwrite` 为函数名，`f` 表示的是 `file`，`write` 是写的意思，整个函数名表示的是写文件的意思。这个函数的参数列表和 `fread()` 函数是一样的，其含义也是一样的，`buffer` 表示的是写内存什么地方数据，`size` 是单位，`count` 是单位数，`stream` 是文件指针。函数的返回值的含义也和 `fread` 一样，表示的是实际写入的数据，出错就用 `feof()` 和 `ferror()` 函数检验错误。

有一点需要注意，`fwrite()` 函数和 `fputc()`、`fputs()` 函数只能针对以写模式、读写模式或者追加模式打开的文件，针对读模式打开的文件进行写操作就会出错。接下来看一个完整的例子，来使用一下这些 C 语言提供的文件读写操作的函数。

```
#include<stdio.h>

int main()
{
    FILE *pfile;
    char* p_buffer = "fwrite test!";
    char r_buffer[50];
    char s_buffer[50];
    char c;

    pfile = fopen("test.txt","w");

    fputc('c',pfile);
    fputs("fputs test!",pfile);
    fwrite(p_buffer,1,13,pfile);

    fclose(pfile);

    pfile = fopen("test.txt","r");
    c = fgetc(pfile);
    printf("fgetc : %c\n", c);

    fgets(s_buffer,12,pfile);
    printf("fgets : %s\n", s_buffer);

    fread(r_buffer,1, 13,pfile);
    printf("fread : %s\n", r_buffer);

    fclose(pfile);

    return 0;
}
```

文件操作函数都是声明在标准头文件 `stdio.h` 中的，所以程序一开始得先使用 `include` 文件包含预处理命令包含这个头文件。在程序中分别使用 `fputc()`、`fputs()` 和 `fwrite()` 函数往文件 `test.txt` 文件中写数据，之后又使用 `fgetc()`、`fgets()` 和 `fread()` 函数从这个文件中读数据。在程序中使用两次打开和关闭文件操作，是因为调用了写函数以后，数据还没有真正写到文件中，只有在调用 `fclose()` 函数以后数据才被写入文件，所以先关闭文件将数据写入，之后才能读出数据，否则什么都读不出来。

程序的输出和 `test.txt` 文件的内容如图 15.2 所示，这些函数使用正确，输出如我们所料。



图 15.2 文件读写程序实例

15.4 文件的其他操作

除了文件打开、关闭、读写操作以外，C 语言还提供了另外几个函数用来完成其他的功能，辅助我们更好地读写文件。本节就来看看这几个函数。

15.4.1 随机读写文件

之前讲文件读写函数的时候，提到所有的文件读函数都是从上上次已读数据之后开始读的，文件写函数都是将数据写到文件的尾部的。C 语言是一门很灵活的语言，这个规定也是可以修改的，只不过需要一些特殊的文件操作函数而已。

1. fseek()函数——从文件任意地方开始读写

fseek()函数是一个功能强大的函数，它可以设置文件读写的位置，从而实现想读写哪里就读写哪里，这种随意读写文件的操作被称为文件的随机读写。这个函数的声明如下所示：

```
int fseek( FILE *stream, long offset, int origin );
```

函数名为 fseek，seek 是搜索、寻找的意思，整个函数名的意思是文件寻找，也就是在文件中寻找某个位置。函数有三个参数，第一个参数是一个文件指针变量，也就是在这个文件指针所确定的文件中设置读写位置的；第二个参数是一个 long 整型变量，表示的是一

个偏移大小数字；第三个参数是一个整型变量，表示的是从什么地方开始偏移，这个起始位置+偏移大小，就是文件中确定的一个位置了，文件读写就从这个位置开始，这个整型变量数值为表 15.2 中的三个宏所定义的数字。函数的返回值是一个整型数值，0 表示设置成功，非零表示失败。

表 15.2 文件偏移起始位置

宏 名	含 义	数值
SEEK_SET	从文件头开始	0
SEEK_CUR	从文件当前位置开始	1
SEEK_END	从文件尾开始	2

例如，想要设置文件读写位置是从文件尾部算起，倒退 4 个字节的地方，就可以使用下面的 C 语言函数调用语句。假设 pfile 是确定已经打开的文件的文件指针。

```
fseek(pfile, 4, SEEK_END );
```

2. rewind()函数——回到文件头

fseek()函数虽然功能强大，但是有的时候需要更简洁的方式，rewind()函数可以把文件读写位置设置到文件开头，比 fseek()函数好用多了。其声明形式如下：

```
void rewind( FILE *stream );
```

函数名为 rewind，re 表示的是重新的意思，wind 表示的是缠绕的意思，整个函数名的意思是重新缠绕，也就是绕到文件的开头，从文件开头开始读写的意思。函数只有一个参数，是一个文件指针变量，表示要设置的是这个文件指针所确定的文件的读写位置。函数没有返回值。从这个函数的声明可以看出，调用它很简单。

其实，这个函数可以使用 fseek()代替，但是因为 rewind()函数使用简单，所以依然保留。下面的两个函数调用语句是等价的。显然，第二个简单嘛！

```
fseek(fpile,0,SEEK_SET);  
rewind(fpile);
```

15.4.2 出错检验

有时在进行文件操作的时候会出现错误，或者读到了文件的末尾。所以，在对文件进行操作的时候就得判断是不是出现错误了，是不是已经到达文件尾了。这时候就得使用 C 语言提供的文件读写出错检验函数了。

1. ferror()函数检验是否出错

这个函数用来检验在上一个文件操作函数调用时，是不是有操作错误出现。函数的声明形式如下所示：

```
int ferror( FILE *stream );
```

函数名为 ferror，error 是出错的意思，整个函数的意思是文件出错，表示的就是检验文件操作时是不是有错误。函数的参数是一个文件指针变量，表示的就是检验这个函数指

针所确定的文件，在操作的时候是不是出现了错误。函数的返回值是一个整型数值，0 表示没错，非零表示有错。

2. feof()函数检验是否到达文件尾

这个函数用来检验在上一个文件操作函数调用时，是不是已经到达了文件的尾部。函数的声明形式如下所示：

```
int feof( FILE *stream );
```

函数名为 feof，e 表示的是 end 结尾，of 表示“的”的意思，整个文件名表示的是“end of file”文件的结尾，用来检验文件操作时是不是已经到达了文件的尾部。函数的参数是一个文件指针变量，表示的就是检验这个函数指针所确定的文件，在操作的时候是不是到达了文件尾部。函数的返回值是一个整型数值，0 表示没到达结尾，非零表示到达结尾。

接下来看一个例子，看看如何使用这几个特殊的文件操作函数。

```
#include<stdio.h>

int main()
{
    FILE *pfile;
    int i;
    char c;

    pfile = fopen("test.txt", "r");

    for(i=0; i<20; i++)
    {
        if(!feof(pfile))
        {
            c = fgetc(pfile);
            printf("%c",c);
        }
        else
        {
            rewind(pfile);
            c = fgetc(pfile);
            printf("\n%c",c);
        }
    }
    fclose(pfile);
    return 0;
}
```

在这个程序中，每次从文件中读取一个字符，然后将其输出。使用 feof()函数检验是不是已经读到文件的尾部，如果到达了尾部就使用 rewind()让读取函数从文件的开头重新读，总共读取了 20 次。

test.txt 文件的内容和程序的输出如图 15.3 所示，每次到达文件尾部以后，就从一个新行开始输出，所以图中显示了两行文件中的内容，第二行没有输出完文件的内容是因为我们只读 20 次。



图 15.3 其他文件操作

15.5 小 结

本章是本书的最后一章，主要讲解了 C 语言在实际中的一个应用：对文件进行操作。本章的重点是各种文件操作及与之对应的函数的形式和作用。本章的难点是文件的含义，以及不同人对文件的不同视角，还有就是文件操作函数的使用。

15.6 习 题

【题目 1】 写段代码，让计算机生成一个 1GB 的空文件，留待以后往其中某些位置保存需要的数据。

【分析】 先来看一个概念，计算机中的空文件。从用户角度来看就是文件的内容为空，从计算机的角度来看就是文件中保存的数据全是二进制的 0。在计算机中，有的时候，为了某些特定的目的会生成一些空文件，这些文件的大小不一，有的时候会很小，几个字节，有的时候上 MB，上 GB。要产生一个 1GB 大小的保存的全是二进制的 0 的文件，你可以想到什么方法呢？

可以使用文件写函数 `fputc()`，往文件中写字符'\0'，写 $1024 \times 1024 \times 1024$ 次。可以使用 `fputs()` 函数，写一个只有'\0'的字符串，不停地写，直到写够 1G 个'\0'。使用 `fwrite()` 函数，写一个只包含 0 的数据块，不停地写，直到写够 1GB 个字节的 0 等，诸如此类的办法都是可以的。但是，还有更简单的方法，那就是使用 `fseek()` 函数，将要写的数据定位到文件中 $1024 \times 1024 \times 1024$ 字节处，然后往里写一个字符'\0'，计算机就会自动生成文件大小为 1GB 的空文件。

【核心代码】

```
FILE* file;  
  
file = fopen("test.txt", "w");  
fseek(file, 1024*1024*1024, SEEK_SET);  
fputc('\0', file);  
fclose(file);
```

【题目2】 如果只打开文件，而不关闭文件，会有什么影响？

【分析】 文件关闭的作用，是为了清除当初在进行文件打开的时候所保存的文件信息。计算机所能保存的打开文件的信息数目是有限的，如果在程序中只打开文件，而不关闭已经不需要的文件，就会造成可以保存文件信息的数目被消耗完，到最后你会发现你再也无法打开文件了。所以，最好在不使用已打开的文件的时候，关闭这个文件，保证计算机总是能够在需要的时候，可以打开文件。

附录 A ASCII 码表

表 A.1 列出了 ASCII 字符集。每一个字符有它的十进制值、十六进制值、终端显示结果、ASCII 助记名和 ASCII 控制字符含义。

表A.1 ASCII码表

十进制值	十六进制值	终端显示	ASCII 助记名	含 义
0	00	^@	NUL	空字符（Null）
1	01	^A	SOH	标题开始
2	02	^B	STX	文本的开始
3	03	^C	ETX	文本的结束
4	04	^D	EOT	传输的结束
5	05	^E	ENQ	请求
6	06	^F	ACK	确认回应
7	07	^G	BEL	响铃
8	08	^H	BS	后退
9	09	^I	HT	水平定位符号
10	0A	^J	LF	换行
11	0B	^K	VT	垂直定位符号
12	0C	^L	FF	换页键
13	0D	^M	CR	回车
14	0E	^N	SO	取消变换（Shift out）
15	0F	^O	SI	启用变换（Shift in）
16	10	^P	DLE	跳出数据通信
17	11	^Q	DC1	设备控制 1（XON 启用软件速度控制）
18	12	^R	DC2	设备控制 2
19	13	^S	DC3	设备控制 3（XOFF 停用软件速度控制）
20	14	^T	DC4	设备控制 4
21	15	^U	NAK	确认失败回应
22	16	^V	SYN	同步用暂停
23	17	^W	ETB	传输块结束
24	18	^X	CAN	取消
25	19	^Y	EM	连接介质中断
26	1A	^Z	SUB	替换
27	1B	^[ESC	跳出
28	1C	^*	FS	文件分隔符
29	1D	^]	GS	组群分隔符
30	1E	^^	RS	记录分隔符
31	1F	^_	US	单元分隔符

续表

十进制值	十六进制值	终端显示	ASCII 助记名	含 义
32	20	(Space)	Space	空格
33	21	!	!	感叹号
34	22	"	"	双引号
35	23	#	#	#号
36	24	\$	\$	\$号
37	25	%	%	百分号
38	26	&	&	&号
39	27	'	'	单引号
40	28	((左小括号
41	29))	右小括号
42	2A	*	*	星号
43	2B	+	+	加号
44	2C	,	,	逗号
45	2D	-	-	减号
46	2E	.	.	点号
47	2F	/	/	正斜杠
48	30	0	0	数字 0
49	31	1	1	数字 1
50	32	2	2	数字 2
51	33	3	3	数字 3
52	34	4	4	数字 4
53	35	5	5	数字 5
54	36	6	6	数字 6
55	37	7	7	数字 7
56	38	8	8	数字 8
57	39	9	9	数字 9
58	3A	:	:	冒号
59	3B	;	;	分号
60	3C	<	<	小于号
61	3D	=	=	等号
62	3E	>	>	大于号
63	3F	?	?	问号
64	40	@	@	@号
65	41	A	A	大写字母 A
66	42	B	B	大写字母 B
67	43	C	C	大写字母 C
68	44	D	D	大写字母 D
69	45	E	E	大写字母 E
70	46	F	F	大写字母 F
71	47	G	G	大写字母 G
72	48	H	H	大写字母 H
73	49	I	I	大写字母 I

续表

十进制值	十六进制值	终端显示	ASCII 助记名	含 义
74	4A	J	J	大写字母 J
75	4B	K	K	大写字母 K
76	4C	L	L	大写字母 L
77	4D	M	M	大写字母 M
78	4E	N	N	大写字母 N
79	4F	O	O	大写字母 O
80	50	P	P	大写字母 P
81	51	Q	Q	大写字母 Q
82	52	R	R	大写字母 R
83	53	S	S	大写字母 S
84	54	T	T	大写字母 T
85	55	U	U	大写字母 U
86	56	V	V	大写字母 V
87	57	W	W	大写字母 W
88	58	X	X	大写字母 X
89	59	Y	Y	大写字母 Y
90	5A	Z	Z	大写字母 Z
91	5B	[[左中括号
92	5C	\	\	反斜杠
93	5D]]	右中括号
94	5E	^	^	^号
95	5F	_	_	下划线
96	60	`	`	`号
97	61	a	a	小写字母 a
98	62	b	b	小写字母 b
99	63	c	c	小写字母 c
100	64	d	d	小写字母 d
101	65	e	e	小写字母 e
102	66	f	f	小写字母 f
103	67	g	g	小写字母 g
104	68	h	h	小写字母 h
105	69	i	i	小写字母 i
106	6A	j	j	小写字母 j
107	6B	k	k	小写字母 k
108	6C	l	l	小写字母 l
109	6D	m	m	小写字母 m
110	6E	n	n	小写字母 n
111	6F	o	o	小写字母 o
112	70	p	p	小写字母 p
113	71	q	q	小写字母 q
114	72	r	r	小写字母 r
115	73	s	s	小写字母 s

续表

十进制值	十六进制值	终端显示	ASCII 助记名	含 义
116	74	t	t	小写字母 t
117	75	u	u	小写字母 u
118	76	v	v	小写字母 v
119	77	w	w	小写字母 w
120	78	x	x	小写字母 x
121	79	y	y	小写字母 y
122	7A	z	z	小写字母 z
123	7B	{	{	左大括号
124	7C			竖线符
125	7D	}	}	右大括号
126	7E	~	~	波浪线符
127	7F		DEL	删除 (Delete)

⚠注意：其中十进制 ASCII 码值为 0~31 和 127 的字符为控制字符（非显示字符），不同终端和设备上的显示可能有所不同。十进制 ASCII 码值为 96 的字符不是单引号，而是反引号，它一般位于键盘左上角 ESC 按键下方，和“~”位于同一个按键。