第1章 引 言

1.1 本书的目标

本书可作为计算类专业本科生的入门课程教材。通过入门课程的学习,初学者应该对计算有一个系统的认识,并为后续课程的学习打下一个良好的基础。

对于初学者而言,计算机无疑是一个复杂的机器,但是实际上它只是一个由数量巨大的简单元件(晶体管)组成的集合。本书的目标就是向读者介绍计算系统,即如何由简单的元件组成计算机,并使其能执行用计算机语言编写的程序。

学习完本书后,读者可以使用一种高级计算机语言——C 语言编写比较复杂的程序,并能理解这些程序在计算机内部是如何执行的。

1.2 计算机与计算机系统

1. 计算机

前面提到的"计算机"指的是"现代计算机",全称为"通用电子数字计算机(General Purpose Electronic Digital Computer)"。世界上第一台通用电子数字计算机是 1946 年在美国宾夕法尼亚大学研制成功的 ENIAC(Electronic Numerical Integrator and Calculator,电子数字积分器和计算器)。

(1) 通用计算设备

"通用"是指计算机是一种通用计算设备,而不是一种专用设备。在现代计算机出现之前出现过加法器、乘法器等设备,这些设备只能执行专门的计算,如加法、乘法等。而计算机则既可以做加法,也可以做乘法,还可以实现排序或者其他计算。

通用计算设备的思想要归功于英国数学家阿兰·图灵。图灵在 1936 年发表了一篇题为"论可计算数及其在判定问题中的应用"的论文,给出了通用计算设备的数学描述。

基于通用计算设备思想设计出来的现代计算机具有以下两个特点。

① 所有的计算机(无论大小,快慢,昂贵还是便宜),如果给予足够的时间和足够容量的存储器,都可以做相同的计算。换句话说,所有的计算机都能做几乎完全相同的事情,只是在计算速度上有差别。

② 如果想做一种新的计算,不需要对计算机重新进行设计,只需要在计算机上安装合适的

软件,就可以达到目的。只要安装了合适的软件,利用计算机进行计算的领域可以说是非常广阔的,例如安装了拨打电话软件后,就可以利用计算机实现打电话的功能。

(2) 电子设备

1642 年,法国数学家帕斯卡发明了世界上第一台真正意义上的计算机。它利用齿轮传动原理制造而成,通过手摇方式操作运算,即手摇机械计算机。随着电子技术的发明与发展,电子元件逐渐演变成为计算机的主体,成为现代计算机硬件实现的物理基础。

计算机是非常复杂的电子设备,计算机执行的计算最终都是通过电子电路中的电流、电位等实现的。

(3) 数字设备

"数字"是现代计算机的一种基本特征,也是计算机通用性的一个重要基础。

在现代计算机出现之前,还出现过许多计算机器,大多属于模拟机。模拟机通过测量物理量(如距离或电压)得出计算结果。例如,计算尺通过读取两个标有对数的尺子之间的"距离"计算出乘法结果。模拟机的缺点是很难提高精度,例如,模拟手表通过时针、分针和秒针的移动,根据测量的角度来表示时间。如果要将其精度提高到 0.01 s,可以采用数字手表。因为数字手表使用数字表示时间,只要增加位数就可以提高精度。

在现代计算机里,所有信息都是采用数字化的形式表示的。无论是整数、小数、文字,还是图像、声音等,在计算机里都统一使用数字表示。

(4) 计算机

"计算机"意味着它是一种能够做计算的机器。

该机器的核心处理部件是CPU (Central Processing Unit,中央处理器)。处理器完成的最重要的工作就是执行指令,即执行加法、乘法等计算工作。

指令说明了计算机执行的一件明确定义的工作。计算机程序由一组指令组成,指令是计算机程序中规定的可执行的最小工作单位。也就是说,计算机要么执行指令所说明的工作,要么什么都不做。

在早期的机器中,处理器执行的程序并不是放在机器中的,程序通常表现为一叠打了孔的卡片。机器在工作时依靠一台读卡机读取卡片,再由处理器完成卡片上程序所要求的工作。采用这种方式,受读卡机工作速度的限制,计算速度较慢。

此后,美籍匈牙利科学家冯·诺依曼提出了"存储程序控制原理"思想,现代计算机就是按照其思想构建出来的。存储程序控制计算机的核心部件除 CPU 外,还有存储器(内存)。程序存储在存储器里,CPU 负责完成两项工作:指挥信息的处理和执行信息的实际处理。"指挥信息的处理",是指从存储器里读取下一条指令;而"执行信息的实际处理",则是执行该指令,即进行加法、乘法等计算工作。这两项工作循环进行,即读取指令,执行指令,读取指令,执行指令……

目前各类计算机的 CPU 都是采用半导体集成电路技术制造的。其基础材料为硅片,通过复杂的工艺,在只有指甲般大小的硅片上集成了数以亿计的晶体管,称之为"微处理器"。

2. 计算机系统

当人们提到“计算机”时,往往不单指处理器、存储器等部分,还包括外部设备,如输入命令的键盘,点击菜单的鼠标,显示计算机系统生成的信息的显示器,把信息输出为纸质品的打印机,保存信息的磁盘和 CD-ROM 等,此外还有用户希望执行的程序,如操作系统 UNIX 或 Windows,数据库系统 Oracle 或 DB2,应用程序 Microsoft Office 或 Oracle Open Office。

以上集合就构成了“计算机系统”。也就是说,计算机系统由硬件和软件两部分组成,硬件包括处理器、存储器和外部设备等,软件包括程序和文档。

硬件和软件是计算机系统的两个组成部分,在设计硬件或软件时,如果能够同时考虑两者的能力和限制,可使计算机系统达到最佳工作状态。

如果微处理器设计者理解了将要在其设计的微处理器上执行的程序的需求,就可以设计出更高效的微处理器。例如,Intel、Motorola 和其他主要的微处理器厂商认识到未来的很多应用都会将视频作为 E - mail、计算机游戏、电影的一部分,对于那些应用来说,高效执行非常重要。为此,Intel 定义了称为 MMX 的指令集,为其开发了特殊的硬件。Motorola 则定义了 AltiVec 指令集并开发出相应的硬件。

这一点对于软件设计者也是一样的。如果软件设计者理解了执行程序的硬件的能力和限制,也可以设计出性能更高的程序。例如,要编写一款有竞争力的游戏软件,软件设计者就需要了解处理器的并行化特征。

1.3 计 算 系 统

人类使用自然语言(即人类所讲的语言)来描述问题,而计算机则使用流动的电子解决问题。因此,必须将人类的自然语言转换成计算机能识别的指令,进而转换为能够影响电子流动的电压,才能使计算机完成复杂的任务。这种转换是一种有序的、系统的转换。

图 1.1 显示了在计算系统中使用电子解决问题的一个过程,可以将其表示为 7 个抽象层次(其中,程序层又包括两个抽象层次)。抽象层次是硬件和软件设计者在解决问题时使用的一种方法,每一层对它的上一层隐藏了自己的技术细节。

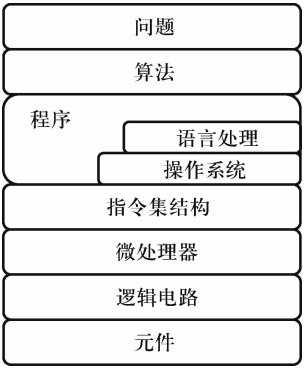

图 1.1 计算系统的抽象层次

本书的目标就是向读者解释这个系统,本章首先按照自顶向下的顺序进行简单的介绍。对于本章出现的术语和概念,并不需要读者完全理解,可以在读完本书后,再来阅读这些内容。

1. 问题

人类使用自然语言来描述那些希望计算机解决的问题。自然语言通俗易懂,但是不能直接作为计算机的指令,最重要的原因就是它具有“歧义性”:为了确定一句话的含义,听者通常需要根据说话人的发音、语调,语句的上下文来进行判断。例如,“羽毛球拍卖

完了。”可以理解为没有羽毛球拍了,被卖完了;也可以理解为没有羽毛球了,被拍卖完了。

如果把这种有歧义的自然语言作为指令提供给计算机,计算机是无法执行的,本质上,计算机仍是电子设备,它只能机械地执行明确的指令,如“Add A,B”是将两个数 A 和 B 相加。面对有歧义的自然语言,计算机将无所适从。

2. 算法

因此,第一步就是舍弃描述问题的自然语言中的歧义,将用自然语言描述的问题转换成一个无歧义的操作步骤,即算法。算法是一个逐步计算的过程,该过程一定能够结束,而且每个步骤都能够被明确描述,并能被计算机所执行。以上这 3 个特点可用专门的术语说明。

有限性(Finiteness):计算过程最终能够结束。

确定性(Definiteness):每个步骤都必须是明确的,不应存在歧义性。例如,“A 与一个数相加”就是“不确定”的,因为不知道 A 与哪一个数相加。

有效可计算性(Effective Computability):每个步骤都能被计算机执行。例如,“A 除以 0”就缺乏可计算性。

要解决一个问题通常可以采用多种算法,有的算法可能需要较少的计算时间;有的算法可能需要较少的存储空间。算法分析就是对一个算法需要多少计算时间和存储空间进行定量的分析。

3. 程序

第二步是使用一种程序设计语言把算法转换为程序。程序设计语言与自然语言不同,它不是在人类的交谈中演化形成的。相反地,程序设计语言是用于表达计算机指令的语言,不能存在歧义。

迄今为止,人类为满足各种需要而发明的程序设计语言可达千种之多,例如,FORTRAN 语言主要用于科学计算,COBOL 语言可用于进行商业数据处理,Prolog 语言可用于设计专家系统,LISP 语言可用于研究人工智能问题,Pascal 语言则适用于程序设计初学者学习如何编程,等等。C 语言是为了开发系统软件(如操作系统和编译器)而发明的,本书选择 C 语言进行程序设计。

程序设计语言可以分为高级语言与低级语言两个级别。高级语言和底层计算机有一定的距离,与执行程序的计算机无关,称之为“独立于机器”。上面提到的这些语言都是高级语言。低级语言则与执行程序的计算机紧密相关,基本上每种计算机都有自己的低级语言——机器语言和汇编语言。如图 1.1 所示,不需要进行语言处理的程序为机器语言程序,汇编语言程序及高级语言程序则需要经过语言处理,翻译为机器语言程序后才能执行;大部分程序,无论高级语言程序,还是低级语言程序,均需要操作系统的支持。

仍以“将两个数 A 和 B 相加”为例,C 语言可以表示为 A + B;而用某种机器的汇编语言表示,可以为“Add A,B”,其机器语言指令则为 0001001001000000。

4. 语言处理

对于使用高级语言和汇编语言编写的程序而言,如果要在计算机上执行,必须将其翻译成执

行程序作业的机器(目标机器)的指令,即机器语言。

因此,语言处理又可分为两层:高级语言处理和低级语言处理,如图 1.2 所示。高级语言处理是指将高级语言翻译为低级语言的过程;而低级语言处理是指将汇编语言翻译成机器语言的过程。这个翻译工作通常可以由翻译程序来完成。高级语言翻译程序称为编译器(又称编译程序)或解释器(又称解释程序),低级语言翻译程序称为汇编器(又称汇编程序)。

5. 操作系统

如何把编写的程序输入计算机中? 如何把计算机执行的结果输出给用户? 最初的操作系统包含的就是支持输入/输出(Input/Output,I/O)操作的设备管理例程。随着技术的发展,操作系统又扩展实现了文件管理、内存管理、进程管理等主要功能。

在计算机的发展过程中出现过许多操作系统,例如 DOS、Mac OS、Windows、UNIX、Linux、OS/2 等。

本书只介绍操作系统的 I/O 设备管理例程,在此又可分为两层:I/O 例程和系统调用,如图 1.3 所示。

图 1.2 语言处理层

图 1.3 操作系统层

6. 指令集结构

将高级语言程序翻译成某种低级的机器语言,其依据就是目标机器的指令集结构(Instruction Set Architecture,ISA)。

指令集结构是编写的程序和执行程序的底层计算机硬件之间的接口的完整定义。指令集结构指明了计算机能够执行的指令的集合,也就是说计算机能够执行的操作和每一步操作所需的数据。所需的数据称为"操作数",操作数在计算机中的表示方式称为"数据类型",而操作数位于存储器的什么地方则被称为"寻址模式"。除了规定操作数的数据类型和寻址模式外,指令集结构还规定了计算机存储单元的数量,以及每个存储单元存储信息的能力。

不同的指令集结构所规定的操作数、数据类型和寻址模式是不一样的。有些指令集结构有数百种操作,而有些则只有十来种。有些指令集结构只有一种数据类型,而有些却多达十几种。有的指令集结构只有一两种寻址模式,而有些却有 20 多种。1985 年 Intel 公司设计的 IA-32 指令集结构至今仍在广泛使用,它有 100 多种操作,10 多种数据类型和 20 多种寻址模式。1986 年发布的 MIPS 也是一种典型的指令集结构,与 IA-32 相比,它规定的操作、数据类型和寻址模式要少得多。

此外,其他的指令集结构还有 PowerPC(IBM 和 Motorola)、Alpha(COPMPAQ)、PA-RISC(HP)、SPARC(SUN 和 HAL)以及最新的 IA-64(Intel)等。

如果需要将某种高级语言(如 C 语言)翻译后在某种目标机器(如 IA-32)上执行,则必须使用相应的翻译程序。

7. 微处理器

接下来就是把指令集结构实现为微处理器。每一种指令集结构都可以采用多种微结构来实现,对于计算机设计者来说,每一种实现都是一次对微处理器的成本和性能的平衡。计算机设计就是平衡成本与性能的一种选择,使用更高(低)的成本,计算机就有更好(差)的性能表现。

以 IA - 32 指令集结构为例,从 1985 年 Intel 实现的 80386 微处理器,之后的 80486、80586 微处理器,到 1998 年推出的 Pentium(奔腾)微处理器,都是采用不同微结构对 IA - 32 指令集结构的实现。

MIPS 指令集结构由 Cisco、Nintendo、Sony 和 SGI 等公司生产,实现了不同的微处理器,用于 Sony、Nintendo 的游戏机,Cisco 的路由器和 SGI 超级计算机中。

8. 逻辑电路

下一步就是组成微处理器的每一个组件的逻辑电路实现。同样也有很多选择,因为设计者也需要考虑如何尽量平衡成本和性能。例如,对于组成微处理器的加法器的实现,选择超前进位逻辑电路比选择行波进位电路计算速度更快。

9. 元件

最终,每一种基本的逻辑电路都是由特定的物理元件实现的。例如,CMOS(Complementary Metal-Oxide-Semiconductor,互补金属氧化物半导体)逻辑电路采用金属氧化物半导体晶体管制造,双极型逻辑电路则采用双极型晶体管构成。

1.4 本书的结构

本书采用自顶向下和自底向上相结合的方式描述了计算系统的"逐层转换"过程,可分为 3 个部分。

第一部分(第 2 章 ~ 第 5 章)描述从问题到算法,从算法到 C 语言程序的转换。

这一部分只介绍一些程序设计语言最基本的主题:C 语言程序设计简介(第 2 章)、类型和变量(第 3 章)、结构化程序设计和控制结构(第 4 章)、测试和调试(第 5 章),使读者对程序设计有一个感性认识。要深入了解 C 语言相关理论,还必须从计算机的组织结构入手,即了解计算机是如何组成的。

第二部分(第 6 章 ~ 第 14 章)描述从晶体管到逻辑电路,从逻辑电路到处理器,从处理器到指令集结构,从指令集结构到程序的转换。

这一部分主要介绍计算机的组成。首先介绍数据在计算机中是如何表示以及如何运算的(第 6 章),在此基础上说明如何使用晶体管组成可以运算的组件(如加法器)、可以存储信息的组件等逻辑单元(第 7 章),之后即可以使用经典的冯·诺依曼模型(第 8 章)将这些组件组合成计算机(第 9 章)。第 9 章介绍的是一个 MIPS 指令集的简化版本 DLX,DLX 不是一个真实的计算机,但是其设计方法覆盖了现代真实机器实现的一般方法,而且由于其特性简单,容易被初学

者理解。

本书介绍的指令集基于美国加州大学伯克利分校计算机系 Patterson 教授和斯坦福大学计算机系 Hennessy 教授在《计算机系统结构:一种定量的方法(第二版)》一书中给出的 DLX 指令集,并根据需要对其进行了裁减和扩充。

在了解了 DLX 是如何工作之后,就可以在它上面进行编程了,首先使用它自身的语言——机器语言(第 10 章),然后是汇编语言——一种更易于理解的语言(第 11 章)。第 11 章还以 DLX 汇编语言和 C 语言为例介绍了语言处理方法,内容包括:如何将 DLX 汇编语言程序翻译为 DLX 机器语言,以及将 C 语言程序翻译为 DLX 汇编语言,为了完成 C 语言的翻译工作,引入一个非常重要的概念——栈。

为了处理信息进出计算机的问题,在第 12、13 章分别介绍 DLX 操作系统的 I/O 设备管理例程。第 12 章介绍 I/O 例程,第 13 章介绍两个机制:自陷(TRAP)机制,即系统调用,和一个比较复杂的机制——中断驱动的 I/O,以及 C 语言的 I/O。第 14 章介绍 DLX 的子例程。

第三部分(第 15 章 ~ 第 17 章)则是 C 语言的深入主题,包括函数(第 15 章)、指针和数组(第 16 章)以及递归(第 17 章)。在这一部分将 C 语言程序和底层的 DLX 联系起来,以便读者深入理解当在 C 语言程序中使用某一种结构时,在底层的计算机中是如何实现的。

小　　结

通过“逐层转换”,读者不仅可以用 C 语言程序解决问题,还能够理解 C 语言程序是如何通过电子的流动实现的。

在解决实际问题时,不必陷入“逐层转换”的细节之中。例如,当设计一个复杂的计算机应用程序时,如设计一个文字处理软件时,不需要考虑所有抽象层次的每一处细节,而应该集中考虑问题的本质,更多地关注从问题到算法、从算法到程序的转换。再如,在使用门电路设计逻辑电路时,也不必考虑每个门电路的内部结构。为了更好地解决问题,必须了解逐层转换的过程。

习　题　1

1.1　名词解释:计算、计算机、计算机系统和计算系统。

1.2　有两台计算机 A 和 B,A 有乘法指令,而 B 没有;两者都有加法和减法指令;在其他方面两者都相同。那么,在 A 和 B 中,哪台计算机可以解决更多的问题?

1.3　现代计算机为什么采用数字方式设计,而非模拟机?

1.4　现代计算机的核心部件有哪些?分别具有什么功能?

1.5　为什么自然语言不能直接作为计算机指令?举例说明。

1.6　给出如下问题的算法。

(1) 计算 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10

(2) 判定 2010—2500 年中的某一年是否为闰年

(3) 对一个大于或等于 3 的正整数,判断它是否为素数(质数)

1.7 当将计算机升级(如更换 CPU)后,原来的软件(如操作系统)还能够使用吗?

1.8 人们购买的软件通常是以什么语言编写的?是高级语言还是与目标机器 ISA 兼容的机器语言?

1.9 关于计算系统的每个抽象层次,分别举出两个以上的例子进行说明。

1.10 你对计算系统哪一部分比较熟悉?简要说明你对该部分的理解。

第 2 章　C 语言程序设计简介

第一章 1.3 节中介绍了计算系统的逐层转换,即把用自然语言描述的问题转换为算法,再把算法转换成用程序设计语言编写的程序……第 2 ~ 5 章将完成以上两个层次的转换,即将一个用自然语言描述的问题转换成用程序设计语言编写的程序。

本章将首先介绍高级程序设计语言的一些主要特征,并对如何将高级语言程序翻译成机器指令进行简单描述;然后通过一个简单的示例程序介绍 C 语言,给出一个典型的 C 程序所具有的一些重要特点。

2.1　高级程序设计语言

程序设计语言可以分为两个级别:高级语言与低级语言。其中,前者的用户友好程度比后者高,在前者中几乎所有的指令都类似于自然语言(如英语);同时,与后者相比,它还倾向于与特定计算机的指令集结构(ISA)无关。迄今为止,人类发明的高级语言数目达 1 000 种之多,目前应用比较广泛的有 Java、C ++ 、C、FORTRAN、Pascal 等。其中,C 语言与其他高级语言有些不同,它的一些功能与特定的 ISA 相关,具有一些低级语言的特征,因此,本书选择C 语言介绍程序设计的相关内容。

本书的目标是使程序设计初学者了解程序设计的基础知识,并且可以逐步写出复杂的程序,了解所编写的程序是如何在计算机内部执行的。基于该目标,本书将不会对 C 语言进行全面的介绍,仅介绍其基本要点。在第 2 ~ 5 章中将讨论高级语言程序设计的一些最基本的概念,如变量、类型、运算符和控制结构等,并依赖 C 语言来表达这些概念。

2.2　高级语言程序翻译技术

所有用高级语言编写的程序都需要被翻译成计算机能够理解的语言,即机器语言,才能够在计算机上执行。可以采用两种技术完成具体的翻译工作:解释和编译。使用解释技术,需要使用一个称为解释器的翻译程序读入高级语言程序,执行程序所指示的操作。高级语言程序不是被计算机直接执行的,而是被解释程序所执行。使用编译技术,则需要使用一个称为编译器的翻译程序完成将高级语言程序翻译为机器语言的工作。编译器的输出称为可执行映像,它可以直接在硬件上执行。而解释器和编译器本身都是运行在计算机系统上的程序。LISP、BASIC 和 PERL 等高级语言使用解释技术进行翻译,C、C ++ 和 FORTRAN 等则是典型的采用编译技术的

语言。

1. 解释

从解释技术角度看,一个高级语言程序就是一组提供给解释器程序的命令。由解释器读取命令,然后根据语义来执行。高级语言程序并不是直接被硬件执行的,它只是解释器的输入数据。解释器是一个执行程序的虚拟机,一次能够翻译高级语言程序的一段、一行、一条命令或一个子程序。

例如,解释器可以读入高级语言程序的一行代码,然后在底层硬件上执行该行。如果该行为:"计算 x 加 y 的和,把结果存入 z",解释器将通过执行计算机的 ISA 中的相应指令来完成计算。一旦当前行被处理完,解释器就会移到下一行继续处理,直到处理完整个高级语言程序。

2. 编译

使用编译技术,高级语言程序被翻译成可以在硬件上直接执行的机器代码。编译器必须在翻译之前将源程序当成大的单元(通常是整个源文件)来分析。一个程序只需要被编译一次就可以多次执行。编译器处理包括高级语言程序在内的文件后会生成一个可执行映像。编译器并不执行程序(虽然有些复杂的编译器为了优化性能而执行程序),它只是把高级语言翻译成计算机本身的机器语言。

3. 解释与编译的比较

解释与编译技术各有利弊。一般说来,使用解释技术,更容易进行开发和调试。解释器允许一次执行程序代码中的一段,这样,就方便程序员查看中间结果,并且在执行时可以修改代码,调试也更加容易。但是,使用解释将使程序执行起来花费更多的时间,因为有一个中间媒介——解释器在做执行工作。使用编译技术,可以产生更高效的代码,能够更加有效地使用内存,对代码进行优化,程序执行更快。因此,绝大多数商业化软件趋向于使用编译技术。

2.3 C 语言概述

C 程序设计语言是 Dennis Ritchie 于 1972 年在美国贝尔实验室开发出来的,目的是用来开发 UNIX 操作系统,也正是由于这个原因,使得这门语言具备了一些低级语言的功能。它让程序员可以直接对硬件进行操作,例如对内存地址进行操作、进行位运算等,同时,它还具备高级语言的表达能力和便利性。因此,直到今天 C 语言仍然被广泛应用,其应用领域也从系统软件开发扩展到各个领域。

C 程序设计语言在程序设计语言的进展中有其特殊地位。1954 年,第一个高级程序设计语言 FORTRAN 出现,此后,程序设计语言迅速发展,人们为了不同的目的而开发出来的语言种类已超过千种。例如,为了开发第一个 UNIX 操作系统,贝尔实验室的 Ken Thompson 设计出一种称为 B 语言的程序设计语言。C 语言就是在 B 语言的基础上发展而来的,它继承了 B 语言可以直接对硬件进行操作的特点,又在表达能力和便利性方面得到加强。之后,面向对象语言C++又在 C 语言的基础上发展起来,对 C 语言进行了扩充,并从 Simula 语言中吸取了类的概念,使得

C++语言成为当前面向对象程序设计的一种主流语言。而 Java 语言则吸取了 C++面向对象的概念,并进一步增强,成为当今最流行的面向对象程序设计语言。

综上所述,C 语言不仅具备低级语言功能,并且对当今主流的程序设计语言(如 C++和 Java)有着根本性的影响,因此,本书选择 C 语言来介绍计算系统。使用 C 语言,可以把计算机底层硬件与高级程序设计语言的基本概念清晰地联系在一起,同时,理解了本书中介绍的 C 语言知识,还有利于掌握 C++和 Java 语言。

和其他许多程序设计语言类似,C 语言也存在许多不同的版本。1989 年,美国国家标准学会(ANSI)提出了一个“明确的与机器无关的 C 语言的定义”将 C 语言进行了标准化,这个版本被为C89 。1999 年,又提出了一个新版本,在 C89 的基础上增加了一些新内容,称为C99 。本书中出现的大部分 C 语言示例和细节都是基于 ANSI C89 标准的,涉及 C99 标准时将予以说明。大多数C 编译器均支持 ANSI C,为了编译和验证本书中的样例代码,需要使用遵从 ANSI C 的 C 编译器。

2.4 第一个例子:Hello World

下面以一个简单的 C 语言程序示例开始介绍。图 2.1 为该例子的源代码(注意:每行前的行号并不是程序的一部分)。这是第一个例子,在显示器上输出“Hello World!”。这个例子虽然很简单,却反映出一个典型的 C 语言程序具有的一些重要特点。

```
1  /*
2   *
3   * 程序名称:Hello World,本书的第一个 C 程序
4   *
5   * 描述:这个程序在显示器上输出"Hello World!"
6   *
7   */
8
9  /* 下面一行是预处理指令 */
10  #include <stdio.h>
11
12  /* 函数   :main */
13  /* 描述   :在显示器上输出"Hello World!" */
14  int main( ){
15        printf("Hello World!");                    /* 在显示器上输出"Hello World!" */
16  }
```

图 2.1 在显示器上输出“Hello World!”的程序

在此,并不需要彻底理解每一行代码的意义。需要注意的只有如下 4 个方面的特征:main 函数、编程风格、预处理指令,以及输入/输出函数调用。

1. main 函数

main 函数从 14 行"int main( ){"开始,到 16 行结束"}"。这 3 行组成了名为 main 函数的函数定义。函数是 C 语言中一个十分重要的概念,将在第 15 章中详细介绍。在 C 语言中,main 函数起了一个很特殊的作用:程序从 main 函数开始执行。因此,每一个 C 语言程序都需要一个 main 函数。注意,在 ANSI C99 中,main 必须声明为返回一个整数值。也就是说,main 必须是 int 类型的,因而代码的第 14 行为"int main( ){"。而在 C89 中使用"main( ){"即可。

在这个例子中,main 函数的代码(即大括号之间的代码)仅包含一条语句(15 行),说明当函数执行时会发生的行为。对于所有的 C 语言程序,执行都从 main 开始,一条语句接着一条语句地处理,直到 main 中的最后一条语句被执行完。

执行 15 行的语句后将在显示器上显示:

```
Hello World!
```

注意,在例子中 15 行的源代码以分号结束。在 C 语言中,分号用来结束声明和语句,它的使用是为了使编译器能够将程序正确地分解为组件。

2. 编程风格

考虑如下所述的情况:程序员在编程时对某段代码的作用一定是很清楚的,但是过了两年,公司为了升级产品,需要修改程序,该程序员很可能不记得当时为什么写下某一行代码;或者该程序员已不在该公司工作了,修改程序的任务被指派给一个以前没看过这段代码的人。良好的编程风格有助于解决这个问题,即提高程序的可读性和可维护性。培养良好的编程风格,是为了使程序更易读,更加有利于团队合作。

(1) 格式

C 语言是一种自由格式的语言。也就是说,程序中单词之间和行之间的空格数量不会改变程序的意义。在遵守 C 语法规则的前提下,程序员可以以任意方式自由地构造程序。在示例程序的 main 函数中使用了一种缩进格式(15 行),这种格式使函数语句很容易被识别出来。在该示例中,还使用空行将不同的代码段分隔开来(如 11 行)。这些空行不是程序必需的,只是为了增强代码的可读性,将完成不同任务的代码隔开。

该示例还使用了一种关于大括号摆放位置的典型格式,它将开始的大括号放在一行的最后(14 行),而将结束大括号放在一行的第一位(16 行)。

只要有助于传达程序的意图,程序员就可以自由地定义自己的风格。本书中的 C 代码示例均采用典型的 C 语言编程风格。

(2) 注释

在示例程序中,以"/*"开始、以"*/"结束的部分称为注释。注释可以是单行的(如 9 行),也可以跨越多行(如 1 ~ 7 行)。注释是给程序员看的信息,对程序的执行没有任何影响。注释

的目的是为了使程序能够更好地被阅读者所理解,解释一行或一段代码非直观的表象。因此,做注释不是为了重申一个显而易见的表象,而是为了补充解释。

注释的另一个目的是通过在一行中合理地使用空格,使程序具有可读性,从而更易于理解。例如,注释可以用来将程序分割为一个个片段,增强程序的可读性。也就是说,为了计算某一个结果而组合在一起的代码行被放在连续的行中,而产生不同结果的程序片段则相互分隔开来。

恰当的代码注释是编程过程中非常重要的一部分。好的注释增强了代码的可读性,可以使那些对代码不熟悉的人理解得更快。因为编程任务通常由团队合作完成,代码需要在程序员之间共享或借用。为了使编程团体的工作效率更高,或是编写一些值得共享的代码,必须采用一种好的注释风格。

一种良好的注释风格要求:在每个源文件的开头说明代码的最后一次修改日期,以及由谁修改等信息;每个函数(见示例中的 main 函数)都应该有一个简洁的描述,说明这个函数完成的功能以及有关输入/输出的说明;而且,注释通常被点缀在代码中以解释不同代码片段的作用。但是过多的注释是有害的,因为它会使代码更难读,特别是那些用于说明代码中显而易见的信息的注释。

在不同的语言中,注释以不同的方式来表示。例如,C99 标准和 C++ 中的注释还可以以序列"//"开始,延伸至该行末尾。不管注释如何表示,其目的都是相同的:为程序员提供一种以人类语言来描述代码含义的方法。

总之,好的编程风格可提高程序的可读性,有利于团队合作。其他一些有益的 C 语言编程风格,将在后面的章节中结合示例进行介绍。

3. 预处理指令

顾名思义,预处理指令就是在程序被传入编译器之前,需要对其进行预处理的指令。由 C 预处理器完成预处理任务。C 预处理器从头至尾扫描源文件(图 2.1 所示的 C 语言程序就是一个源文件),寻找预处理指令,并根据预处理指令来执行。所有的预处理器指令都以字符"#"开头。

这个简单的例子包含了一个最常用的预处理指令:#include。

```
#include <X.h>
```

#include 指令指示预处理器将另一个文件 X.h(头文件)插入到源文件中。实际上,预处理后#include 指令本身被那个文件的内容所代替。关于 C 语言头文件更详细的内容,将在后续章节中逐步介绍。

例如,对于 C 语言,所有使用 C 的输入/输出(I/O)库函数的程序必须包含 I/O 库的头文件 stdio.h,该文件定义了 C 库中一些与 I/O 函数相关的信息,使用预处理器指令#include <stdio.h> 可以在编译开始之前将其插入该头文件中。

有两种不同形式的 #include 指令:

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

```
#include "myProgram.h"
```

第一种把文件名用尖括号(< >)括起来,指示预处理器可以在一个预定义的目录中找到该头文件。此目录通常由系统配置所决定,包含了许多与系统和库相关的头文件,如 stdio.h。

第二种使用双引号(" ")把文件名括起来,指示预处理器该头文件可以在和 C 源文件相同的目录中找到。这是程序员为了实现特殊目的而创建的头文件。

注意:预处理指令不是以分号结尾的,这是因为#include 是预处理器指令而非 C 语句,不以分号结尾。

4. 输入和输出

几乎所有的程序都需要执行某种形式的输入和输出,因此,在本章结束之前,还需要指出如何在 C 程序中实现输入和输出。在 C 语言中,输入和输出是通过库函数执行的。本节将从一个较高的层次来描述这些函数。关于I/O 库函数较低层次的细节知识,将在第 13 章中进行介绍。

(1) 输出

在示例程序中使用 C 库函数 printf 来执行输出(15 行)。

格式化输出函数printf 的作用是输出到标准的输出设备,典型的输出设备是显示器。

15 行使用 printf 输入文本信息“Hello World!”,对于仅需要输出一些文本信息的情况,只需把文本用双引号(" ")括起来即可。使用双引号(" ")括起来的内容称为“格式用字符串”。

例如:

```
printf("2 + 3 = 5.");
```

将打印如下文本到输出设备上:

```
2 + 3 = 5.
```

通常在很多场合下需要输出一些在程序中生成的数值。在 C 中,“格式用字符串”可以提供要打印的文本和如何在文本中输出数值的说明。要输出的数值列在“格式用字符串”后面。

例如:

```
printf("2 + 3 = %d.",5);
```

在此,“格式用字符串”中提供了要打印的文本“2 + 3 = .”和如何打印列在“格式用字符串”后面的数值 5 的格式说明。在这个例子中“格式用字符串”中包含了格式说明符%d,用于将“格式用字符串”后面的数值(此例中是 5)作为十进制数代替%d 嵌入到输出结果中。输出结果为

```
2 + 3 = 5.
```

下面的例子显示了 printf 的变化形式:

```
printf("2 + 3 = %d.",2 + 3);
```

在这个 printf 中,格式说明使得数值 5(2 + 3 的结果)被嵌入文本中,以十进制数的形式被打

印出来。

所有用于 printf 的格式说明均以百分号% 开头。在图 2.1 中只需要显示文本,不需要显示数值,因而不需要格式说明符。

下面显示了一个非常普遍但很有用的 printf 使用示例:

```
1  #include <stdio.h>
2
3  int main( ){
4      int x;                                  /*变量声明*/
5
6      scanf("%d",&x);                         /*输入一个整数*/
7      printf("2 + %d = %d.",x,2 + x);         /*输出加法结果*/
8      printf("2 - %d = %d.",x,2 - x);         /*输出减法结果*/
9  }
```

在这个例子中,main 函数的代码可以被分解成两部分。第一部分包含了函数中变量的声明(详见第 3 章)。此例中包含一个变量 x。变量,顾名思义,即其值可变的量。变量是高级程序设计语言提供的一个十分有用的特征,即在程序中用符号命名数值。

执行此例,需要用户从键盘输入一个整数,用户输入 3 并按回车键确认后,x 的值将为 3,随后的 printf 语句将根据 x 的值打印出加法和减法结果。

在第 7 行的 printf 中列在“格式用字符串”后面的有两项内容,分别为变量 x 和表达式 2 + x,它们是当程序执行的时候生成的数值。当执行这行代码时,显示的数值取决于 x 当前的值。因此当执行该行语句时,如果 x 等于 3,“格式用字符串”中的第一个% d 将被 3 代替,第二个% d 将被 2 + 3 的结果 5 代替,嵌入在文本中输出,输出结果如下:

```
2 + 3 = 5.
```

与第 7 行类似,在第 8 行的 printf 中列在“格式用字符串”后面的也有两项内容,分别为变量 x 和表达式 2 - x。因此当执行第 8 行语句时,如果 x 等于 3,“格式用字符串”中的第一个% d 将被 3 代替,第二个% d 将被 2 - 3 的结果 - 1 代替,嵌入在文本中输出,输出结果如下:

```
2 - 3 = -1.
```

一般说来,在一个 printf 语句中可以显示尽可能多的数值。格式说明符(例如,% d)的数目必须和“格式用字符串”之后的数值的个数相等。

在这例子中包含了两条 printf 语句,执行此示例后会看到它们会显示在同一行,而没有换行。也就是说,输出结果为:

```
2 + 3 = 5. 2 - 3 = -1.
```

如果要分行显示,就必须在“格式用字符串”里明确地把换行符放在需要的地方。换新行、

制表符和其他特殊的字符需要使用特殊的反斜线(\)。例如,为了打印一行新的字符(即换行),需要使用特殊的序列\n。将第7行的 printf 语句修改如下:

```
printf("2 + %d = %d. \n",x,2 + x);                    /* 输出加法运算结果 */
```

注意格式用字符串以打印新行字符 \n 结束,所以随后的 printf 将从一个新行开始。重新执行修改过的示例,由这些语句生成的输出结果如下:

```
2 + 3 = 5.
2 - 3 = -1.
```

(2) 输入

在介绍完输出函数后,现在转向输入函数。

格式化输入函数scanf 的作用是从标准输入设备输入,典型的输入设备是键盘。

它需要一个"格式用字符串"(类似于 printf 所需要的)和用来存储从键盘得到的数值的变量列表。scanf 函数从键盘读取输入,然后根据"格式用字符串"中的转换字符对输入进行转换,并将转换后的数值分配给变量列表。

在上面的例子中使用了 scanf 函数(6行),并使用格式说明符%d 来读入一个十进制数。格式说明符%d 通知 scanf 期待一系列数字的按键(即数字0~9),以回车键结束。这个序列被解释为一个十进制数,并转换为一个整数,最终被存储在 x 变量中。

注意,变量(例如,x)将被 scanf 函数改变,必须在其前面使用字符 &。在第16章中将讨论使用这个标记的原因。

前面的例子给出的是关于十进制整数输入、输出的格式说明符(%d),关于字符、小数等输入/输出的格式说明符将在后面的章节中结合示例进行介绍。

习 题 2

2.1 列举高级语言和低级语言的优缺点。

2.2 列举编译技术和解释技术的优缺点。

2.3 Java 语言的翻译、执行过程为:首先翻译成字节码文件,然后在 Java 虚拟机上执行。Java 采用的是编译技术还是解释技术?

2.4 高级语言的可移植性是指其代码是否可以在不同的目标机器(即不同的 ISA)上运行。C 语言和 Java 语言的可移植性如何?

2.5 对于如下算法:

(1) 从键盘获取 A

(2) X←A + 1

(3) Y←X + A

(4) Z←Y - A

(5) 输出 Z 到屏幕上

如果使用解释技术将其翻译为机器语言,至少需要执行多少次算术运算?如果使用编译技术,在将其翻译为机器语言之前,将优化这段代码,那么,至少需要执行多少次算术运算?

2.6 如下语句的输出分别是什么?

```
printf ("%d\n%d\n",12,12 +45);
printf ("%d,%d\n",12,12 +45);
printf ("%d %d\n",12,12 +45);
printf ("%d %d\n",12,12 +45);
printf ("%d.%d\n",12,12 +45);
```

第3章　类型和变量

本章将介绍高级程序设计语言的两个基本概念:类型和变量。变量保存了程序执行所需的数值,它有一个最重要的特征,就是所包含的信息的类型。C 语言中的变量是很简明的,它有 3 个最基本的类型:整数、字符以及浮点数类型。

而运算符则是对数值进行操作的语言机制。通过使用变量和运算符,能够方便地表达需要程序完成的工作。C 语言具有一个丰富的运算符集合,根据功能的不同可将其分为赋值、运算、按位运算、逻辑和关系比较几种类型。

本章最后将结合一个示例来说明如何使用变量和运算符来求解问题。

3.1　类型和变量

程序处理的对象是数值,例如,对用户输入的数进行处理,或者将一系列数字相加得到一个总和。

数值是非常重要的程序设计概念,高级语言使程序员能够很容易地管理数值。高级语言允许程序员使用符号引用数值,也就是用一个名字表示数值。在高级语言中,这些用符号命名的数值称为变量。

为了正确地记录程序中的变量,高级语言翻译器(例如 C 语言编译器)需要知道每个变量的几个属性:变量的符号名,变量包含的信息的类型,在程序中的何处可以访问该变量。在大多数语言中,包括 C 语言中,这些都是通过变量的声明提供的。

首先看一个例子,下面声明了一个存储整数值的 y 变量。

```
int y;
```

3.1.1　3 种基本数据类型

C 语言支持 3 种基本数据类型:整数、字符和浮点数。这些类型的变量可以通过使用类型标识符 int、char 和 double(双精度浮点数的缩写)创建。关于数据类型的概念,将在第 6 章中详细介绍。

1. int

int 类型标识符声明了一个有符号的(详见 6.2 节)整数变量。整数变量可以很方便地表示需要在程序中处理的现实世界中的数据,因而在程序中被大量使用。例如,想要表示选课学生的

人数,使用整数变量就是十分恰当的,即

```
int numberOfStudents;
```

再看一个声明表示年龄的变量的例子:

```
int age = 18;
```

注意,第二个声明与第一个有些不同,即声明的同时进行了初始化。在 C 语言中,对于任何一个变量都能够在声明中直接为其设置一个初值。在这个例子中,变量 age 将会有一个整数字面常量18 的初值。字面常量指的是源代码中无名称的数值。整数字面常量可以用一个十进制整数来表示,例如,18 和 -18 均为整数字面常量。对于 numberOfStudents 的初始值是什么,将在 3.3 节介绍。

2. char

char 类型标识符用于声明表示字符数据的变量(关于字符的详细介绍,见 6.7.1 节)。接下来有 4 个例子,第一个声明了一个名为 dot 的变量,第二个声明了 number,第三个声明了 nL,第四个声明了 grade。前三个声明还进行了初始化。变量 dot 将会有一个句点符号“.”的初值,变量 number 将会有一个字符字面常量“8”的初值,变量 nL 将会有一个字符字面常量“\n”的初值,即换新行。还要注意,这些字符字面常量都是被单引号界定的。在 C 语言中,被解释为字面常量的字符是被单引号界定的。

```
char dot = '.';
char number = '8';
char nL = '\n';
char grade;
```

3. double

double 类型标识符允许程序员声明浮点数类型(关于浮点数的详细介绍,见 6.7.2 节)的变量。使用浮点数能方便地处理有小数部分的很大或很小的数。

以下是关于 double 类型变量的两个例子:

```
double areaOfCircle;
double averageScore;
```

就像 int 和 char 一样,也可以在声明一个浮点数时对其进行初始化。在讨论如何初始化浮点数变量之前,先讨论一下如何在 C 语言中表示一个浮点数字面常量。浮点数字面常量可以用一个十进制小数来表示,可以采用指数形式,如下面的例子所示。指数用字符 e 或 E 来表示,可以是正数或负数,它表示 10 的幂次。注意指数必须是一个整数。

```
double pi = 3.14;
double twoHundredTen = 2.1E2;                    /* 这是 210.0 */
```

```
double threeThousand = 3E3;                    /* 这是 3000.0 */
double fiveTenths = 5E - 1;                    /* 这是 0.5 */
double minusFiveTenths = - 5E - 1;             /* 这是 - 0.5 */
```

C 语言中的另外一种浮点数类型标识符为 float,用于声明单精度的浮点数变量,而 double 声明的是双精度浮点数变量。精度与分配给小数部分的位数有关,double 较 float 而言,可以给小数部分分配更多的位数,确切地说,是不能比 float 少。在 C 语言中,具体的小数位数取决于编译器和指令集结构(详见 9.6 节)。

3.1.2 标识符

对于在程序内为变量命名(通常称为标识符),大多数高级语言有其固定的规则。C 语言允许构造由字母表中的字母、数字和下划线组成的标识符。并且,只有字母和下划线可以作为标识符的开头。而标识符的长度是由具体的 C 编译器规定的。值得注意的是,C 的标识符是大小写敏感的,即编译器把大小写字符作为完全不同的字符来考虑,例如,将 numberOfStudents 和 numberofStudents 作为不同的变量对待。

此外,在 C 语言中,有一些有特殊意义的关键字(又称为保留字,如表 3.1 所示),在用做标识符时会受到限制。例如,int 就是一个关键字,所以不能把 int 定义为变量名。这是因为将 int 定义为变量名会使编译器翻译时产生混乱,编译器无法确定某个 int 究竟指的是变量还是类型标识符。

表 3.1 C 语言的保留字列表

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

按照 C 语言命名惯例,以下划线开头的变量(例如,_index_)习惯上用于特殊的库代码。一般地,不使用全部大写的字母声明变量。另外,程序员希望直观地区分由多个单词组成的变量,在本书中,使用除第一个单词外,其他单词的首字母大写的形式(例如,numberOfStudents)。使用下划线区分多个单词也是一种好的命名风格(例如,number_of_students)。

为写出好的代码而给变量取一个有意义的名称是重要的。所选择的变量名应能反映其所表示的值的特性,以便程序员能够更容易识别该值是用来干什么的。例如,numberOfStudents 就是

用来记录学生人数的。

3.1.3 作用域

在 C 语言中,一个变量的声明传达了 3 条信息给编译器:变量的标识符,它的类型,以及它的作用域。其中前两项,即标识符和类型,C 编译器可以从变量声明中显式地获得。而第三项,作用域,编译器可以从代码中声明所在的位置推断出来。变量的作用域是指,在某段区域中,变量是“活跃”的,可以被访问。

在 C 语言中只有两种基本的变量作用域:全局变量和局部变量。

1. 局部变量

在 C 语言中,所有变量在使用之前必须先声明。实际上,一些变量必须在它们所在程序块的开头就声明,这些变量称为局部变量。在 C 语言中,一个程序块就是以大括号{开头和}结束的任意一段程序。所有的局部变量必须立刻跟在大括号{后面声明。C99 标准则允许在使用前声明,即声明不必位于块开头。

下面是一个包含了 3 个局部变量的 C 语言程序。整型变量 age、字符变量 dot 和双精度变量 pi 在包含 main 函数代码的程序块中被声明,这 3 个局部变量只对 main 函数可见,程序中包含的其他函数是无法访问这些变量的。正如代码所示,大部分局部变量都要在使用它们的程序块的开头被声明。

```
#include <stdio.h>

int main( ){
    int age = 18;
    char dot = '.';
    double pi = 3.14;

    printf ("age = %d,dot = %c,pi = %f. \n",age,dot,pi);
}
```

此例使用库函数 printf 输出 3 个局部变量的值。在格式用字符串中包含了 3 个格式说明符,依次为%d、%c 和%f,%d 使得列在格式用字符串后面的第一个数值被作为十进制数代替%d 嵌入在输出结果中,%c 使得列在格式用字符串后面的第二个数值被作为字符代替%c 嵌入在输出结果中,%f 则使得列在格式用字符串后面的第三个数值被作为浮点数代替%f 嵌入在输出结果中,输出形式为小数点后取 6 位小数。打印结果如下:

```
age = 18,dot = .,pi = 3.140000.
```

在不同的程序块中可以使用相同的名字来声明两个不同的变量,这一点有时是很有用的,只要共享同一名字的不同变量位于不同的程序块中即可。

2. 全局变量

对比只能在被声明的程序块中访问的局部变量,全局变量可以在整个程序中被访问。

下面的代码中既包含了一个全局变量,又包含了一个 main 函数的局部变量:

```
#include <stdio.h>
int globalVar = 1;                                    /* 这是全局变量 */

int main ( ){
    int localVar = 2;                                 /* 这是 main 函数的局部变量 */

    printf( "globalVar = %d localVar = %d\n" ,globalVar,localVar) ;
}
```

在某些情况下,全局变量极其有用,但是建议初学编程的人更多地使用局部变量而非全局变量。因为全局变量是公用的而且可以在代码中的任何地方进行修改,大量地使用全局变量会使代码更容易出现缺陷以致难以重新使用和修改。在本书中几乎所有的 C 语言代码示例都只使用局部变量。

再看一个稍微复杂一点的例子。图 3.1 所示的 C 语言程序与前面的程序很相似,唯一不同的是在 main 中也声明了一个名为 globalVar 的变量。执行这个程序后会发现,在 main 函数中,局部变量如果与全局变量同名,则在 main 函数中,该变量是局部变量,全局变量在 main 函数中失效。

```
#include <stdio.h>
int globalVar = 1;                                          /* 这是全局变量 */

int main( ){
    int localVar = 2;                                       /* 这是 main 函数的局部变量 */
    int globalVar = 3;                                      /* 这是 main 函数的局部变量 */

    printf( "globalVar = %d localVar = %d\n" ,globalVar,localVar) ;
                                                            /* 3,2 */
}
```

图 3.1　局部变量与全局变量同名的 C 语言程序

3.2　运　算　符

与其他高级语言一样,C 语言也具有一套丰富的运算符集合,包括算术运算符、逻辑运算符、比较运算符等类别,允许程序员自然、方便、简洁地表达运算。

1. 表达式和语句

使用运算符,把变量和字面常量结合起来就形成一个 C 表达式。例如,x + 1、x + y 和 y = x +

1 都是表达式的例子。

表达式后面加上分号,就构成一条语句。例如,“y = x + 1;”是一条语句。正如自然语言中的一个句子拥有一个完整的思想或动作一样,一条 C 语句表达了一个被计算机执行的完整的工作单元。与自然语言中用标点符号表示一个句子结束的方法一样,所有的 C 语句都以分号(或者大括号})结束。C 语言还有一个特点就是可以写出一条不执行任何操作的语句,但是在语法上可以认为是语句,即空语句。空语句就是一个简单的分号,不执行任何运算。

一条或者多条简单的语句组合起来就形成一个复合语句,即通过把这些简单的语句包围在大括号{ }内形成一个块。按照语法,复合语句等价于简单语句。在下一章将介绍复合语句的使用方法。

下面显示了一些简单语句、空语句和复合语句:

```
y = x + 1;                          /* 这是简单语句 */
i = i + 1;                          /* 这是简单语句 */
;                                   /* 这是空语句 */
{                                   /* 这是复合语句 */
    y = x + 1;
    i = i + 1;
}
```

2. 赋值运算符

C 语言的赋值运算符是等号 = 。使用赋值运算符,可以构造赋值表达式,例如 i = 3 和 x = y 都是赋值表达式。该运算符通过首先计算出等式右边的值,再将右边的数值赋值给左边的对象而发生作用。例如,在 C 语句

```
z = x + y;
```

中,变量 z 的值会被设置为表达式 x + y 的值。

注意,尽管数学中的等号与 C 语言中的赋值符号相同,但它们有不同的意义。在数学中使用 = 是要说明右边和左边的表达式是相等的,在 C 语言中使用 = 运算符,是让编译器生成将左边数值改变为右边数值的代码。换句话说,右边的数值被赋给左边的变量。下面的 C 语句表示整数变量 i 被增加了 1:

```
i = i + 1;
```

在 C 语言中,所有表达式都可以计算出一个特定类型的数值。在前面的例子中,表达式 i + 1 将表示整数 1 加到另一个整数上(变量 i),计算出的是整数数值,这个整数结果被赋给一个整数变量(变量 i)。

使用赋值表达式也可以计算出一个特定类型的数值,例如,由 i = 3 计算出的数值就是整数值 3。

但是如果构造了一个混合类型的表达式,例如 i = 3.1,i 的结果是什么(假设 i 为整数变量)?

在 C 语言中,一个变量的类型是保持不变的(即不能被改变),所以浮点数 3.1 会先被转换为整数,再赋值给 i。在 C 语言中,浮点数值通过省略小数部分而成为整数值。例如,当从浮点数转化为整数时,3.1 会被舍为 3,3.9 也会被舍为 3。因此,i = 3.1 的计算结果为整数值 3。

3. 算术运算符

加、减、乘、除算术运算与对应的符号和小学算术一样。例如,+ 代表加法,- 代表减法,* 代表乘法(为了避免与字母 X 混淆,与平时习惯使用的乘号不同),还有/代表除法。就像手工做算术运算一样,表达式也有一个计算顺序。乘法与除法先运算,之后是加法和减法。运算符的计算顺序称为优先级,将在后面做更详细的讨论。下面是使用算术运算符形成的几条 C 语句:

```
circumference = 2.0 * 3.14159 * radius;
area = 3.14159 * radius * radius;
sumOfScore = scoreOfExam1 + scoreOfExam2;
averageScore = sumOfScore/2.0;
y = a * x * x + b * x + c;
```

还有一个特殊的运算符:模运算符%(也称为整数求余运算符)。用 C 语言执行两个整数的除法运算,小数部分会被忽略,取整数部分作为运算结果。例如,表达式 10/3 的计算结果为 3。模运算符% 则可以用于计算余数。例如,10%3 的计算结果为 1。换句话说,(10/3) * 3 + (10% 3)等于 10。在下面的例子中,所有的变量都是整数:

```
quotient = x/y;                    /* 如果 x = 8, y = 5, quotient = 1 */
remainder = x% y;                  /* 如果 x = 8, y = 5, remainder = 3 */
```

乘法、除法和模运算符都比加法和减法优先级高。

对于算术运算符来说,如果构造一个混合类型的表达式,例如 i + 3.1,会发生什么呢(假设 i 为整数变量)?

一般地,在 C 语言中,像 i + 3.1 这样的混合类型表达式会将整数转换为浮点数,然后进行计算,结果为浮点数。

再把 i + 3.1 赋值给 i 呢? 即 i = i + 3.1,会怎么样? 根据前面的讨论,一个变量的类型不能改变,所以表达式 i + 3.1 的类型会被转换为变量 i 的类型,即 i + 3.1 的浮点计算结果将被转换成整数,再赋值给 i。

4. 运算符的优先级与结合性

在介绍其他 C 语言的运算符之前,先来回答一个重要的问题:作为下面语句的结果,x 中将保存什么值?

```
x = 1 + 2 * 3;
```

(1) 运算符的优先级

正如手工做算术一样,计算表达式是有顺序的,这个顺序称为优先级。例如,当进行算术运

算时,乘法和除法比加法和减法优先级更高。对于算术运算符,C 的优先级规则与小学算术相同。在上面的语句中,因为乘法运算符比加法运算符优先级高,而加法运算符的优先级比赋值运算符高,因此,先计算乘法,再计算加法,最后赋值,即 x 被赋值为 7。也就是说,表达式的计算与 x = (1 + (2 * 3)) 相同。

(2) 运算符的结合性

对于优先级相同的运算符,例如:

```
x = 1 + 2 - 3 + 4;
```

计算结果取决于首先计算哪个运算符,表达式 1 + 2 - 3 + 4 的值可能等于 4 或 -4。既然所有运算符的优先级都是相同的(在 C 语言中加法和减法有相同的优先级),则需要有一个明确的表达式计算规则。对于相同优先级的运算符,它们的结合性决定了被计算的顺序。对于加法和减法的情况,两者都是自左向右结合,因此 1 + 2 - 3 + 4 的计算与((1 + 2) - 3) + 4 一样,而非1 + (2 - (3 + 4))。

而赋值运算符的结合性则是自右向左。因此,j = i = 3 的计算与 j = (i = 3) 相同,即先计算 i = 3,结果为 3,再将 3 赋值给 j,最后,i 和 j 的值均为 3。

运算符优先级与结合性的完整列表如表 3.2 所示。

表 3.2　运算符优先级、结合性列表

优先级	结合性	运算符
1(最高)	自左至右	( )　[ ]　->　.
2	自右至左	*(指针)&(地址)-(一元)　~　!　sizeof　++　--　(type)
3	自左至右	*(乘法)/　%
4	自左至右	+(二元)-(二元)
5	自左至右	<<　>>
6	自左至右	<　>　<=　>=
7	自左至右	==　!=
8	自左至右	&
9	自左至右	^
10	自左至右	|
11	自左至右	&&
12	自左至右	||
13	自左至右	?:(条件表达式)
14	自右至左	=　+=　-=　*=等
15(最低)	自左至右	,

(3) 圆括号

圆括号不考虑计算规则而是直接地指明哪一个运算先于其他运算执行。与算术中一样,计算总是从最里面的一组圆括号开始。如果想让一个子表达式首先被执行,就可以用圆括号将该子表达式括起来。所以在下面的例子中,假定 a、b、c、d 和 e 都等于 5。语句

```
x = a * b + c * d/e;
```

等价于

```
x = ((a * b) + ((c * d)/e));
```

在这两个语句中,x 的结果都为 30。即在运用优先级规则之前,首先计算最里面的子表达式的值,再向外计算。

同样,假定 a、b、c、d 和 e 都等于 5,下面的表达式得出什么结果?

```
x = a * (b + c) * d/e;
```

对于大多数读代码的人来说,都不太可能记住 C 的全部优先级规则。因此,在构造表达时通常采用如下编程风格:在长的或复杂的表达式中,使用圆括号增强代码的可读性,即使没有它们代码仍可正常执行。

5. 关系运算符

C 语言中有几种比较两个数值之间关系的运算符。在第 4 章中将会看到,在 C 语言中经常使用这些运算符生成条件结构。

相等运算符 == 是 C 语言的关系运算符之一。这个运算符用于比较两个数值是否相等。如果它们相等,表达式的值就为 1,如果不相等就为 0。下面展示了两个例子:

```
x = (8 == 9);                    /* x 将等于 0 */
x = (a == b);                    /* 如果 a 和 b 相等,x 将等于 1 */
```

在第二个例子中,赋值运算符右边的是表达式 a == b,取决于 a 和 b 是否相等,其值为 1 或 0。(注意:圆括号不是必需的,因为 == 运算符的优先级高于 = 运算符。添加圆括号只是为了提高代码的可读性。)

与相等运算符含义相反的是不相等运算符 != ,如果两个操作数不相等,则表达式的值为 1。在下面的例子中描述了其他的关系运算符,用来测试大于、小于、相等等关系。在这些例子中,变量 x、a 和 b 是整数。变量 a 的值是 3,b 是 4。

```
x = a == b;                      /* 相等运算符,x 等于 0 */
x = a > b;                       /* 大于运算符,x 等于 0 */
x = a != b;                      /* 不相等运算符,x 等于 1 */
x = a <= b;                      /* 小于等于运算符,x 等于 1 */
```

关系运算符经常用于根据数值比较的结果改变程序流程。在第 4 章中将会详细地介绍 C

语言中的 if 语句。在此,如果变量 month 等于 1,将打印一条信息。

```
if( month == 1 )
    printf( "The month has 31 days. \n" );
```

6. 逻辑运算符

在介绍逻辑运算符之前,下面先说明 C 语言中的逻辑真和逻辑假的概念。C 语言采取了以一个非零的值(即一个除了零以外的值)表示逻辑真的方法。数值零是逻辑假。

C 语言支持 3 个逻辑运算符:&&、|| 和!。&& 运算符用于对两个操作数执行逻辑“与”运算,如果两个操作数都为逻辑真或非零,表达时计算结果为整数值 1(即逻辑真),否则结果为数值 0。例如,8&&9 的值为 1,而 8&&0 的值为 0。|| 运算符为 C 语言的逻辑“或”运算符。如果 x 或 y 中有一个非零,则表达式 x || y 的值为 1。例如,8 || 9 的值为 1,8 || 0 的值也为 1。逻辑“非”运算符! 用于计算出操作数的另一个逻辑状态,所以!x 仅当 x 为 0 时值为 1,否则为 0。

逻辑运算符的作用之一就是在程序内构造逻辑条件。例如,可以使用关系和逻辑运算符组合起来判断某个变量的值是否处于一个特定的范围内。为了检查表示年龄的变量 age 是否包含在 18 与 25 之间,可以使用如下表达式:

```
(18 <= age)&&(age <= 25)
```

或者用来测试表示月份的变量 month 是否是 4、6、9 或 11:

```
(month == 4) || (month == 6) || (month == 9) || (month == 11)
```

注意,上面两个例子中的圆括号对于表达式计算不是必需的,但是它们可以让代码的可读性更强。

这里有一些逻辑运算符的例子,与先前的例子一样,变量 x、a 和 b 是整数。变量 a 的值为 3,b 的值为 4。

```
x = a&&b;                    /* x = 1 */
x = a||b;                    /* x = 1 */
x = !a&&!b;                  /* x = 0 */
x = a||-2;                   /* x = 1 */
```

在这 4 个逻辑运算符中,逻辑“非”运算符拥有最高的优先级,其次是逻辑“与”,再次是逻辑“或”。

7. 自增/自减运算符

由于变量自增和自减运算是经常执行的,C 语言的设计者决定用专门的运算符来表示。++ 运算符表示将一个变量递增为比它大的数值,而 -- 运算符则表示递减。例如,表达式 i++ 表示将整数变量 i 的值加 1,表达式 i-- 表示将 i 的值减 1。注意这些运算符改变了变量本身的数值,也就是说,i++ 与 i = i + 1 运算是类似的。

++和--运算符可用在一个变量的任意一边。表达式++i和i++有些不同。如果i++是一个更大的表达式中的一部分,那么表达式i++的值就是i递增之前的值,而表达式++i的值则是递增之后的i的值。如果运算符++出现在变量之前,那么它就以前缀的形式被使用。如果它出现在变量之后,则以后缀的形式使用。前缀形式通常指的是先增或先减,而后缀则是后增或后减。例如:

```
i=1;
y=i++;
```

在此,整型变量i递增。然而,i的初值被赋给变量y(即表达式i++的值为i的原始值)。在这段代码执行之后,变量y的值为1,而i的值将为2。

类似地,下面的代码是将i递增:

```
i=1;
y=++i;
```

然而在这段代码中,表达式++i的值为递增后的值。在这种情况下,i和y的值都为2。

前缀和后缀形式之间的微妙差别并不影响对本书内容的理解。对于本书中用到这些运算符的几个例子,这些运算符的前缀和后缀形式是可以互换的。

8. 其他的C运算符

C语言还有一些特殊的运算符,可以作为C语言的标志。例如可以使用组合的运算符,以使表达式更简单。

(1) 特殊的赋值运算符

C语言允许将某些算术运算符和=组合成特殊的赋值运算符。例如,如果想要计算1与变量i的和,可以使用简化符号+=:

```
i+=1;
```

这行代码相当于:

```
i=i+1;
```

其他的赋值运算符还有-=、*=、/=、%=等。赋值运算符的优先级最低,并且每组都是从右向左结合的。

(2) 条件表达式

条件表达式是C语言的一个特征。条件表达式的符号是问号和冒号。下面是一个例子:

```
x=a? b:c;
```

这里变量x将会根据a的逻辑值得到b的值,或者c的值。如果a不是0,它将得到b的值;否则,它将得到c的值。

图 3.2 是一个使用条件表达式判断某个整数是否可以被 7 整除的程序。要判断的整数从键盘输入,如果该整数可以被 7 整除,则显示 T(代表真),否则显示 F(代表假)。

```
#include <stdio.h>

int main( ){
    char result;
    int input;

    printf("Input an integer:");
    scanf("%d",&input);

    result = (!(input%7))? 'T':'F';
    printf("The result is %c\n",result);
}
```

图 3.2 使用了条件表达式的 C 语言程序

9. 编程风格

一个真正有用的表达式有时会包含多个运算符,它把各种各样的运算符和操作数组合起来,形成一个复杂的表达式。下面显示了一个复杂的表达式:

```
x = a&&b - 4||5 + c/3;
```

为了计算出这条语句的值,需要检查运算符的计算顺序。表 3.2 中列出了所有的 C 语言运算符(包括还没有介绍的但在本书后面会介绍的运算符)以及它们的计算顺序。根据优先级规则,这条语句等价于下面的语句:

```
y = (a&&(b - 4))||(5 + (c/3));
```

再次强调良好的编程风格:在使用多个运算符的复杂表达式中,应使用圆括号增强代码的可读性。

3.3 附加主题

本节包括一些关于变量的附加主题,目的是比较完整地介绍 C 语言,但是这些内容对于理解本书的目标不是必需的。

1. 常量

常量是在程序执行中不会改变的数值。在 C 语言中,常见的常量有以下 3 种类型。

(1) 字面常量

字面常量是以字面形式出现在源代码中且没有名字的数值,在 3.1.1 节进行过描述,字符字

面常量值需要加单引号,如'A'代表了字符 A,浮点型字面常量值还可以使用指数表示法。

图 3.3 给出了一个计算给定半径的圆的面积的程序,这个例子对于区别 C 代码中常见的 3 种类型的常量数值很有帮助。例如,数值 3.14159 就是字面常量。

```
#include <stdio.h>

#define RADIUS  11.5

int main( ){
  const double pi = 3.14159;
  double area;

  /* 计算 */
  area = pi * RADIUS * RADIUS;                    /* 面积 = pi * r^2 */

  printf("Area of a circle with radius %f is %f cm^2\n",RADIUS,area);
}
```

图 3.3 一个计算圆的面积的 C 语言程序

(2) 使用预处理指令#define 创建的常量

关于预处理指令#include,在 2.4 节进行过描述。此例中的#define 是另一个常用的预处理指令。

```
#define X Y
```

#define 指令是一个简单而又有用的指令,能命令 C 预处理器用文本 Y 代替出现的与 X 匹配的任意文本。也就是说,宏 X 被 Y 替代。在图 3.3 所示的例子中,#define 使 RADIUS 被 11.5 替代,所以源文件中的下面两行

```
area = pi * RADIUS * RADIUS;
printf("Area of a circle with radius %f is %f cm^2\n",RADIUS,area);
```

被转换(在计算机内部,预处理器和编译器之间)为

```
area = pi * 11.5 * 11.5;
printf("Area of a circle with radius %f is %f cm^2\n",11.5,area);
```

通常,在一个程序内,使用#define 指令指定固定的数值。下面是几个例子:

```
#define NUMBER_OF_STUDENTS 25
#define LENGTH_OF_SIDE1 3.5
#define LENGTH_OF_SIDE2 4.5
```

```
#define COLOR_OF_TREE green
```

在上面的例子中,使用符号 NUMBER_OF_STUDENTS 表示学生人数。如果人数发生改变,只需简单地修改宏 NUMBER_OF_STUDENTS 的定义,然后由预处理器完成实际的替换工作,因而非常便利:如果学生人数在一个程序中被经常使用,只需要修改源代码中的一行,就能够改变所有代码中的学生人数。使用符号 LENGTH_OF_SIDE1 和 LENGTH_OF_SIDE2 表示三角形的两个边长。注意,最后一个例子与其他例子有稍许不同。在这个例子中,字符串 COLOUR_OF_TREE 被替代为另一个字符串 green。在通常采用的编程风格中,使用全部大写的字母作为宏的名称。

(3) 通过在变量类型标识符前加上限定词 const 声明的常量。

这种常量实际上就是在程序执行中数值不会被改变的变量。

在图 3.3 所示的这个例子中创建了一个初始化为 3.14159 的命名为 pi 的浮点型常量,使用限定词 const 将其声明为一个常量。

用 const 声明的常量和用#define 定义的符号值之间的差别如下:

声明的常量是那些传统上认为永远不变的常量。常量 pi 就是一个例子。物理常量中像光速、一周的天数等,都按照惯例使用声明的常量来表示。

在一个单独的程序执行过程中保持不变,但可能对于不同的用户或不同的调用代表不同的值,这样的数值使用#define 来定义。这些值可以被看做是程序的参数。例如,RADIUS 在图 3.3 里能够被改变为其他值,程序重新编译,然后重新执行。

总的来说,用 const 或#define 命名一个常量比在代码中使用字面常量更可取。比起没有名称的字面常量,命名可以表达更多的代码含义。

2. 存储类

在本章的前面曾经提到 C 的变量的 3 个基本属性:标识符、类型和作用域。关于变量,还有另外一个属性:存储类。一个变量的存储类指明了 C 编译器是怎样为它分配存储空间的,并且特别指明当包含这个变量的程序块执行完毕时该变量是否会失去它的值。C 语言中有两种存储类:静态的和自动的。静态变量在调用之间保留其中的值。自动变量在该块结束之后其值会丢失。在 C 语言中,全局变量是静态存储类,也就是说,全局变量的值直到程序结束都不会失去。局部变量在默认情况下是自动存储类。局部变量可以通过在声明中使用修饰符 static 声明为静态变量。例如,在某一函数中通过“static int localVar;”声明一个变量 localVar,函数执行结束后其值仍然会保留。如果在该程序执行过程中,该函数再次被执行,localVar 将仍然保留其先前的值。

在讨论了存储类之后,就可以回答前面曾提出的一个问题:如果一个变量未被初始化,其初值将会是什么? 在 C 语言中,在默认情况下,自动存储类型的变量以一个未知的值开始。也就是说,在 C 语言中,自动变量是未被初始化的。相反地,所有静态存储类型的变量则是在程序开始执行时被初始化为 0。

3.4 问题求解:长度单位换算

编写一个程序实现长度单位厘米到英尺和英寸的转换,要求从键盘输入需要被换算的长度,并将换算结果输出到显示器上。

编程采用自顶向下、逐步求精的方法,即从一个算法的粗略描述开始,然后把大步骤分解为多个小的步骤,直到可以很容易地写出 C 代码来。在第 4 章中将通过更多的示例详细介绍自顶向下、逐步求精方法。

最开始的一步(步骤 0),需要考虑如何表示程序需要处理的数据项(对于所有问题,都需要考虑)。可以从 3 种基本的数据类型:整型、字符型和浮点型中进行选择。对于这个问题,可以用浮点型或整型数表达内部的计算。本问题是将一个用厘米表示的长度转化为用英尺和英寸表示。在实际应用中,以一个人的身高为例,可以说 160 厘米,或者是 5 英尺 3 英寸,而不是 5.2 英尺0.6 英寸。因此,关于长度单位换算的数据类型最好选择整型(虽然存在舍入问题,但是对于这个问题则可以忽略)。

选择了数据表示方法后,就可以通过逐步求精来分解这个问题。为了求解这个问题,必须首先调查清楚要求实现什么,以及提供了哪些条件以便解决这个问题,这永远是一个求解问题的好思路。对于本问题,就是从键盘输入一个长度,根据输入进行换算,最后输出换算结果。

图 3.4 显示了对这个特定问题的分解。其中,步骤 1 是对这个问题最初进行的设计。它包含 3 个阶段:获取输入数据,计算,输出结果(对于大部分问题都适用)。在第一个阶段,向用户询问需要换算的长度(以厘米为单位)。在第二个阶段将进行必要的计算,然后在第三个阶段输出计算的结果。

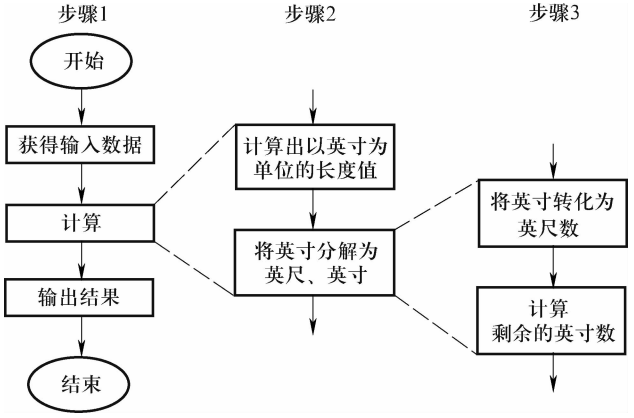

图 3.4 一个简单的长度单位换算问题的逐步求精

步骤 1 不够详细,不能直接转化成 C 代码,因此在步骤 2 中需要进行进一步的精练。其中,计算阶段可以被进一步精练为一个先用英寸为单位计算出总长度的子阶段——对给定输入数据

进行简单计算,和一个把用英寸表示的总长度转化为用英尺和英寸表示的子阶段。

步骤2中的第二个子阶段仍然不够细化,因而不能映射到C语言,需要在步骤3中进行再一次精练。在步骤3中把这个阶段又精练为两个子阶段。首先,在总的英寸数的基础上计算总的英尺数。计算英尺数以后,就得到了剩余的英寸数。至此,就可以直接映射到C代码了。

解决这个问题的完整C语言程序如图3.5所示。

```
#include <stdio.h>

int main( ){
    int length;                          /*需要转换的长度的厘米数*/
    int foot;                            /*长度的英尺数*/
    int inch;                            /*长度的英寸数*/

    /*获取输入:长度的厘米数*/
    printf("What is the length in cm to be transferred?");
    scanf("%d",&length);

    /*计算英寸数*/
    inch = length/2.54;

    /*计算英尺、英寸数*/
    foot = inch/12;                      /*商数是英尺*/
    inch = inch%12;                      /*余数是英寸*/

    /*输出结果*/
    printf("Length is %d foot %d inch \n",foot,inch);
}
```

图3.5　一个简单的长度单位换算的C语言程序

注意,语句“inch = length/2.54;”是一个混合类型的表达式,首先将length转换为浮点数,然后进行计算,再把“length/2.54”的浮点计算结果转换成整数,赋值给inch。

习　题　3

3.1　如下代码片段的输出是什么?

```
char number = '8';
char nL = '\n';
double twoHundredTen = 2.1E2;            /*这是210.0*/
```

```
double threeThousand = 3E3;                    /* 这是 3000.0 */
double fiveTenths = 5E - 1;                    /* 这是 0.5 */
double minusFiveTenths = -5E - 1;              /* 这是 -0.5 */


printf("%c%c\n",number,nL);
printf("%f,%f,%f,%f \n",twoHundredTen,threeThousand,fiveTenths,minusFiveTenths);
```

3.2 假设 a 和 b 都是整数,且 a 和 b 分别被赋值为 7 和 8,那么,下列表达式的值分别是多少? 并且,如果 a 和 b 的值发生变化,还需要指出 a 和 b 的新值。

(1) a = b

(2) a = b = 5

(3) a% b

(4) b% a

(5) a‖b

(6) a&&b

(7) !(a/b)

(8) ++a + b--

(9) a = b += 1

(10) a = (b++==8)? a:b

(11) a = (++b==8)? a:b

3.3 编写一个 C 语言程序:从键盘读入两个整数,将两个整数中较大的数输出。

3.4 假设现在设计一个新的计算机程序设计语言,包括运算符 +、-、* 和/,在下列不同的限定条件下,表达式 a + b - c * d/e 的计算结果分别是什么(使用圆括号表示运算顺序)?

(1) 优先级顺序为:* 和/优先级相同,+ 和 - 优先级相同,且 * 和/的优先级高于 + 和 -,结合性均为自左至右

(2) 优先级顺序为:* 和/优先级相同,+ 和 - 优先级相同,且 * 和/的优先级高于 + 和 -,结合性均为自右至左

(3) 优先级顺序为:* 和/优先级相同,+ 和 - 优先级相同,且 * 和/的优先级低于 + 和 -,结合性均为自左至右

(4) 优先级顺序为:* 和/优先级相同,+ 和 - 优先级相同,且 * 和/的优先级低于 + 和 -,结合性均为自右至左

(5) 优先级顺序为:+、-、*、/的优先级是依次降低的,即 + 优先级最高,/的优先级最低

(6) 优先级顺序为:+、-、*、/的优先级是依次升高的,即 + 优先级最低,/的优先级最高

(7) 4 个运算符的优先级相同,结合性均为自左至右

(8) 4 个运算符的优先级相同,结合性均为自右至左

3.5 如下所示的程序与图 3.1 类似,不同的是在 main 中加了一个子块,在子块中声明了一个新变量。该程序的输出是什么?

```
#include <stdio.h>
int globalVar = 1;
```

```
int main( ){
 int localVar = 1;

 localVar ++;
 globalVar ++;
 printf("Global% d Local% d\n",globalVar,localVar);
 {
    int localVar = 2;
    localVar ++;
    globalVar ++;
    printf("Global% d Local% d\n",globalVar,localVar);
 }
 localVar ++;
 globalVar ++;
 printf("Global% d Local% d\n",globalVar,localVar);
}
```

3.6　假设一个程序包含两个整型变量 a 和 b,值分别为 7 和 8。根据下列条件分别写出可以交换 a 和 b 的值的 C 语句,即执行这些语句后,a 和 b 的值分别是 8 和 7。

(1) 可以使用一个临时的变量,即第三个变量;

(2) 不使用临时变量(即只有 a 和 b 两个变量)。

3.7　编写一个 C 程序:将以秒计数的时间转换到小时、分钟和秒,要求从键盘输入需要换算的秒数,并将换算结果输出到显示器上。