C 语言小白变怪兽

——当你决定阅读本教程时，你已然超越了 90%的程序员

作者：严长生　　完整版地址：http://c.biancheng.net/c/

关注微信公众号「编程帮」，和作者一起学习编程

「关于教程」

《C 语言小白变怪兽》发布于 C 语言中文网，由站长亲自执笔，将多年的编程经验灌输其中，典型的实践派。

八年的编程功力，加上四年的精雕细琢，使得这部教程独具匠心，不仅通俗易懂，而且深入你心。这看似平凡的背后，是默默的坚持以及超乎常人的付出，只要你稍加留意，就会处处见精妙。

《C 语言小白变怪兽》分为基础部分和高级部分：

- 初级部分重点讲解语法知识，培养编程思维；
- 高级部分还会讲解内存、多文件编程（模块化开发）、字符编码、调试技巧、缓冲区（缓存）、复杂指针（花样指针）、数据存储格式、职业规划等进阶技能。

阅读初级部分能够能轻松入门 C 语言，学会手敲代码，建立大脑的思维模式；阅读高级部分能够醍醐灌顶，颠覆三观， 以后在大神面前也可以吹牛逼。

《C 语言小白变怪兽》经历过 5 次大改版，每次都推翻重来。随着编程功力的精进，每次翻阅前文都顿感文笔拙略，技艺不佳，未能向读者传达足够的信息，遂推倒重来。编写教程是一个寂寞的过程，需要一个安静的环境苦心孤诣，字斟句酌，深夜是最好的选择。不规律的作息导致我生物钟错乱，经常失眠，甚是苦恼。

读者怎么说

《C 语言小白变怪兽》已经累计了几百万人次的阅读量，几乎每天都有读者加 QQ 好友给我们好评，夸赞教程通俗易懂，一针见血，颠覆了以前对 C 语言的认知，是初学者的 “启蒙读物” 。下面的链接收集了一些具有代表性的评价：http://c.biancheng.net/cpp/about/dig/

除了给教程点赞，还有读者给我们发红包，请我们吃鸡腿。

也有读者给我们写了长长的感谢信，我都一一回复，不敢怠慢。

刚刚离职的时候还有一位盲人读者，愿意赞助 3000 元支持我们更新教程，感动到扑街。

「关于作者」

严长生，《C 语言小白变怪兽》作者，C 语言中文网站长，毕业于西安电子科技大学，曾就职于去哪儿网，从事网站开发工作。写作是我持之以恒的爱好，八年的程序员生涯，我和团队编写了十余套教程，包括 C 语言小白变怪兽、C++教程、数据结构教程、Python 教程、Golang 教程、Linux 教程、Shell 脚本编程、Socket 编程、Windows 编程入门、GCC 简明教程等。

大学四年

吊儿郎当读书，认认真真编程。

就读于西安电子科技大学期间，我主修电子信息工程，细分方向是天线和微波，也就是研究通信技术。

大一加入我校文学社，获得诗歌征文比赛二等奖，后来担任文学社副主席。

大二开始逃课，潜心研究编程，所以成绩一直不好，经常挂科，被同学和老师鄙视，然而我不 care。

也正是大二，我创办了「陕西专升本网」，获得人生第一桶金，毕业后转让他人，现已关站。

大三担任我校网络协会副主席，并举办技术讲座，逐渐变得忙碌起来。

大三这年还创办了「C 语言中文网」，致力于做精品教程，帮助对编程感兴趣的读者。

大四上半年拿到去哪儿网 Offer，算是大学期间所有努力的最好见证。

大四这年还获得风险投资做在线教育，但「too young, too simple」，没折腾起来。

工作和离职

一个人要想有所造诣，必须得有执念。

毕业后加入去哪儿网，从事网站开发工作。由于工作压力较大，业余时间少，C 语言中文网基本处于停滞状态。

最终我决定辞去工作，全职编写教程。离职后感觉整个人都变得轻松和自由了，没有了任何束缚。

离职后的日子相当清苦，当时的教程都是免费的，网站只有零星的广告收入，不足我一个人的饭费，靠着积蓄苦撑了一年。

现在我依然坚守在第一线，除了编写教程，还进行 VIP 会员的一对一答疑。

「目录」

第 01 章　编程基础 1
1.1 通俗地理解什么是编程语言 1
1.2 C 语言究竟是一门怎样的语言？ 4
1.3 C 语言是菜鸟和大神的分水岭 7
1.4 学编程难吗？多久能入门？ 8
1.5 英语和数学不好，能学编程吗？ 10
1.6 初中毕业能学会编程吗？ 11
1.7 C 语言和 C++到底有什么关系？ 11
1.8 学了 C 语言到底能做什么，能从事什么工作？ 11
1.9 进制详解：二进制、八进制和十六进制 12
1.10 不同进制之间的转换 14
1.11 数据在内存中的存储（二进制形式存储） 21
1.12 载入内存，让程序运行起来 22
1.13 ASCII 编码，将英文存储到计算机 24
1.14 GB2312 编码和 GBK 编码，将中文存储到计算机 30
1.15 Unicode 字符集，将全世界的文字存储到计算机 30
1.16 程序员的薪水和发展方向大全 31
1.17 不要这样学习 C 语言，这是一个坑！ 31
1.18 明白了这点才能学好编程，否则参加什么培训班都没用 31
第 02 章 C 语言初探 31
2.1 第一个 C 语言程序 32
2.2 选择正确的输入法，严格区分中英文 33
2.3 什么是源文件？ 35
2.4 什么是编译和链接（通俗易懂，深入本质） 35
2.5 主流 C 语言编译器有哪些？ 37
2.6 什么是 IDE（集成开发环境）？ 39
2.7 什么是工程/项目（Project）？ 39
2.8 哪款 C 语言编译器（IDE）适合初学者？ 40
2.9 如何在手机上编写 C 语言代码？ 43
2.10 C 语言的三套标准：C89、C99 和 C11 43

2.11 C 语言为什么有那么多编译器？.....43
2.12 程序安装是怎么回事？.....43
2.13 制作安装包，让用户安装程序.....43
2.14 C 语言程序的错误和警告.....43
2.15 分析第一个 C 语言程序.....46
2.16 C 语言代码中的空白符.....49
2.17 彩色版的 C 语言，让文字更漂亮.....50
2.18 一个真正带界面的 C 语言程序.....51
第 03 章 变量和数据类型.....51
3.1 大话 C 语言变量和数据类型.....51
3.2 在屏幕上输出各种类型的数据.....54
3.3 C 语言中的整数（short,int,long）.....59
3.4 C 语言中的二进制数、八进制数和十六进制数.....63
3.5 C 语言中的正负数及其输出.....66
3.6 整数在内存中是如何存储的，为什么它堪称天才般的设计.....70
3.7 整数的取值范围以及数值溢出.....70
3.8 C 语言中的小数（float,double）.....70
3.9 小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）.....74
3.10 在 C 语言中使用英文字符.....74
3.11 在 C 语言中使用中文字符.....77
3.12 C 语言到底使用什么编码？谁说 C 语言使用 ASCII 码，真是荒谬！.....78
3.13 C 语言转义字符.....78
3.14 C 语言标识符、关键字、注释、表达式和语句.....80
3.15 C 语言加减乘除运算.....82
3.16 C 语言自增(++)和自减(--)运算符.....86
3.17 变量的定义位置以及初始值.....88
3.18 C 语言运算符的优先级和结合性.....88
3.19 C 语言数据类型转换（自动转换+强制转换）.....90
第 04 章 C 语言输入输出.....93
4.1 C 语言数据输出大汇总以及轻量进阶.....93
4.2 C 语言在屏幕的任意位置输出字符，开发小游戏的第一步.....101
4.3 使用 scanf 读取从键盘输入的数据（含输入格式汇总表）.....101
4.4 C 语言输入字符和字符串（所有函数大汇总）.....108
4.5 进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症.....112

4.6 结合 C 语言缓冲区谈 scanf 函数，那些奇怪的行为其实都有章可循 112
4.7 C 语言清空（刷新）缓冲区，从根本上消除那些奇怪的行为 112
4.8 C 语言 scanf 的高级用法，原来 scanf 还有这么多新技能 112
4.9 C 语言模拟密码输入（显示星号） 113
4.10 C 语言非阻塞式键盘监听，用户不输入数据程序也能继续执行 113
第 05 章 循环结构和选择结构 113
5.1 C 语言 if else 语句详解 114
5.2 C 语言关系运算符详解 118
5.3 C 语言逻辑运算符详解 120
5.4 C 语言 switch case 语句详解 123
5.5 C 语言条件运算符（? :）详解 127
5.6 C 语言 while 循环和 do while 循环详解 128
5.7 C 语言 for 循环（for 语句）详解 131
5.8 C 语言跳出循环（break 和 continue） 134
5.9 C 语言循环嵌套 136
5.10 对选择结构和循环结构的总结 139
5.11 谈编程思维的培养，初学者如何实现自我突破（非常重要） 140
5.12 写一个内存泄露的例子，让计算机内存爆满 140
第 06 章 C 语言数组 140
6.1 什么是数组？ 141
6.2 C 语言二维数组 145
6.3【实例】判断数组中是否包含某个元素 149
6.4 C 语言字符数组和字符串 151
6.5 C 语言字符串的输入和输出 154
6.6 C 语言字符串处理函数 157
6.7 C 语言数组是静态的，不能插入或删除元素 159
6.8 C 语言数组的越界和溢出 159
6.9 C 语言变长数组：使用变量指明数组的长度 159
6.10 C 语言对数组元素进行排序（冒泡排序法） 159
6.11 对 C 语言数组的总结 162
第 07 章 C 语言函数 163
7.1 什么是函数？ 164
7.2 C 语言函数定义（C 语言自定义函数） 167
7.3 函数的形参和实参（非常详细） 171

74. 函数返回值（return 关键字）精讲 173
7.5 函数调用详解（从中发现程序运行的秘密） 176
7.6 函数声明以及函数原型 178
7.7 全局变量和局部变量（带实例讲解） 181
7.8 C 语言变量的作用域（加深对全局变量和局部变量的理解） 184
7.9 C 语言块级变量（在代码块内部定义的变量） 188
7.10 C 语言递归函数（递归调用）详解[带实例演示] 192
7.11 C 语言中间递归函数（比较复杂的一种递归） 195
7.12 C 语言多层递归函数（最烧脑的一种递归） 195
7.13 递归函数的致命缺陷：巨大的时间开销和内存开销（附带优化方案） 196
7.14 忽略语法细节，从整体上理解函数 196
第 08 章 C 语言预处理命令 197
8.1 C 语言预处理命令是什么？ 198
8.2 C 语言#include 的用法（文件包含命令） 200
8.3 C 语言宏定义（#define 的用法） 202
8.4 C 语言带参数的宏定义 205
8.5 C 语言带参宏定义和函数的区别 209
8.6 宏参数的字符串化和宏参数的连接 210
8.7 C 语言中几个预定义宏 210
8.8 C 语言条件编译 210
8.9 #error 命令，阻止程序编译 214
8.10 C 语言预处理命令总结 215
第 09 章 C 语言指针（精讲版） 216
9.1 1 分钟彻底理解指针的概念 216
9.2 C 指针变量的定义和使用（精华） 218
9.3 指针变量的运算（加法、减法和比较运算） 222
9.4 C 语言数组指针（指向数组的指针） 224
9.5 C 语言字符串指针（指向字符串的指针） 228
9.6 C 语言数组灵活多变的访问形式 231
9.7 指针变量作为函数参数 231
9.8 指针作为函数返回值 235
9.9 二级指针（指向指针的指针） 237
9.10 空指针 NULL 以及 void 指针 238
9.11 数组和指针绝不等价，数组是另外一种类型 238

9.12 数组到底在什么时候会转换为指针 238
9.13 C 语言指针数组（数组每个元素都是指针） 239
9.14 一道题目玩转指针数组和二级指针 240
9.15 二维数组指针（指向二维数组的指针） 240
9.16 函数指针（指向函数的指针） 243
9.17 只需一招，彻底攻克 C 语言指针，再复杂的指针都不怕 244
9.18 main()函数的高级用法：接收用户输入的数据 244
9.19 对 C 语言指针的总结 244
第 10 章 C 语言结构体 245
10.1 什么是结构体？ 246
10.2 结构体数组（带实例演示） 248
10.3 结构体指针（指向结构体的指针） 250
10.4 C 语言枚举类型（enum 关键字） 254
10.5 C 语言共用体（union 关键字） 257
10.6 大端小端以及判别方式 261
10.7 C 语言位域（位段） 261
10.8 C 语言位运算详解 265
10.9 使用位运算对数据或文件内容进行加密 270
第 11 章 C 语言重要知识点补充 270
11.1 C 语言 typedef 的用法 270
11.2 C 语言 const 的用法 273
11.3 C 语言随机数：rand()和 srand()函数 276
第 12 章 C 语言文件操作 279
12.1 C 语言中的文件是什么？ 279
12.2 C 语言打开文件：fopen()函数的用法 280
12.3 文本文件和二进制文件到底有什么区别？ 283
12.4 以字符形式读写文件 284
12.5 以字符串的形式读写文件 287
12.6 以数据块的形式读写文件 289
12.7 格式化读写文件 292
12.8 随机读写文件 294
12.9 C 语言实现文件复制功能(包括文本文件和二进制文件) 296
12.10 FILE 结构体以及缓冲区深入探讨 296
12.11 C 语言获取文件大小（长度） 296

12.12 C 语言插入、删除、更改文件内容 297
第 13 章 C 语言调试教程（非常详细） 297
13.1 调试的概念以及调试器的选择 297
13.2 设置断点，开始调试 297
13.3 查看和修改变量的值 297
13.4 单步调试（逐语句调试和逐过程调试） 298
13.5 即时窗口的使用 298
13.6 查看、修改运行时的内存 298
13.7 有条件断点的设置 298
13.8 assert 断言函数 298
13.9 调试信息的输出 298
13.10 VS 调试的总结以及技巧 299
第 14 章 C 语言内存精讲，让你彻底明白 C 语言的运行机制！ 299
14.1 一个程序在计算机中到底是如何运行的？ 299
14.2 虚拟内存到底是什么？为什么我们在 C 语言中看到的地址是假的？ 300
14.3 虚拟地址空间以及编译模式 300
14.4 C 语言内存对齐，提高寻址效率 300
14.5 内存分页机制，完成虚拟地址的映射 300
14.6 分页机制究竟是如何实现的？ 300
14.7 MMU 部件以及对内存权限的控制 301
14.8 Linux 下 C 语言程序的内存布局（内存模型） 301
14.9 Windows 下 C 语言程序的内存布局（内存模型） 301
14.10 用户模式和内核模式 301
14.11 栈（Stack）是什么？栈溢出又是怎么回事？ 301
14.12 一个函数在栈上到底是怎样的？ 301
14.13 函数调用惯例(Calling Convention) 302
14.14 用一个实例来深入剖析函数进栈出栈的过程 302
14.15 栈溢出攻击的原理是什么？ 302
14.16 C 语言动态内存分配 302
14.17 malloc 函数背后的实现原理——内存池 302
14.18 C 语言野指针以及非法内存操作 303
14.19 C 语言内存泄露（内存丢失） 303
14.20 C 语言变量的存储类别和生存期 303
第 15 章 C 语言头文件的编写（多文件编程） 303

15.1 从 extern 关键字开始谈 C 语言多文件编程 303
15.2 那些被编译器隐藏了的过程 304
15.3 目标文件和可执行文件里面都有什么？ 304
15.4 到底什么是链接，它起到了什么作用？ 304
15.5 符号——链接的粘合剂 304
15.6 强符号和弱符号 304
15.7 强引用和弱引用 304
15.8 C 语言模块化编程中的头文件 305
15.9 C 语言标准库以及标准头文件 305
15.10 细说 C 语言头文件的路径 305
15.11 防止 C 语言头文件被重复包含 305
15.12 C 语言 static 变量和函数 305
15.13 一个比较规范的 C 语言多文件编程的例子 306
第 16 章 C 语言项目实战（带源码和解析） 306
16.1 贪吃蛇游戏（彩色版）【带源码和解析】 306
16.2 2048 小游戏【带源码和解析】 308
16.3 推箱子小游戏（彩色版）【带源码和解析】 310
16.4 扫雷游戏【带源码和解析】 312
16.5 学生信息管理系统（文件版）【带源码和解析】 315
16.6 学生信息管理系统（数据结构版）【带源码和解析】 318
16.7 学生信息管理系统（密码版）【带源码和解析】 319

第 01 章　编程基础

本章是正式进入 C 语言学习的一道「开胃小菜」，并没有涉及具体的语法，目的是让读者对编程的基本知识有所了解，并且告诉读者如何少走弯路。

大家在阅读本章教程的时候请放松心情，不用死记硬背，理解即可。

本章目录：

1.通俗地理解什么是编程语言
2.C 语言究竟是一门怎样的语言？
3.C 语言是菜鸟和大神的分水岭
4.学编程难吗？多久能入门？
5.英语和数学不好，能学编程吗？
6.初中毕业能学会编程吗？
7.C 语言和 C++到底有什么关系？
8.学了 C 语言到底能做什么，能从事什么工作？
9.进制详解：二进制、八进制和十六进制
10.不同进制之间的转换
11.数据在内存中的存储（二进制形式存储）
12.载入内存，让程序运行起来
13.ASCII 编码，将英文存储到计算机
14.GB2312 编码和 GBK 编码，将中文存储到计算机
15.Unicode 字符集，将全世界的文字存储到计算机
16.程序员的薪水和发展方向大全
17.不要这样学习 C 语言，这是一个坑！
18.明白了这点才能学好编程，否则参加什么培训班都没用

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

1.1 通俗地理解什么是编程语言

学习编程语言之前，首先要搞清楚「编程语言」这个概念。

很小的时候，父母就教我们开口说话，也教我们如何理解别人讲话的意思。经过长时间的熏陶和自我学习，我们竟然在不知不觉中学会了说话，同时也能听懂其他小朋友说话的意思了，我们开始向父母要零花钱买零食和玩具、被欺负了向父母倾诉……

我们说的是汉语，是“中国语言”，只要把我们的需求告诉父母，父母就会满足，我们用“中国语言”来控制父母，让父母做我们喜欢的事情。

“中国语言”有固定的格式，每个汉字代表的意思不同，我们必须正确的表达，父母才能理解我们的意思。例如让父母给我们 10 元零花钱，我们会说“妈妈给我 10 块钱吧，我要买小汽车”。如果我们说“10 元给我汽车小零花钱

妈妈”，或者“妈妈给我 10 亿人民币，我要买 F-22”，妈妈就会觉得奇怪，听不懂我们的意思，或者理解错误，责备我们。

我们通过有固定格式和固定词汇的“语言”来控制他人，让他人为我们做事情。语言有很多种，包括汉语、英语、法语、韩语等，虽然他们的词汇和格式都不一样，但是可以达到同样的目的，我们可以选择任意一种语言去控制他人。

同样，我们也可以通过”语言“来控制计算机，让计算机为我们做事情，这样的语言就叫做编程语言（Programming Language）。

编程语言也有固定的格式和词汇，我们必须经过学习才会使用，才能控制计算机。

编程语言有很多种，常用的有 C 语言、C++、Java、C#、Python、PHP、JavaScript、Go 语言、Objective-C、Swift、汇编语言等，每种语言都有自己擅长的方面，例如：

编程语言	主要用途
C/C++	C++ 是在 C 语言的基础上发展起来的，C++ 包含了 C 语言的所有内容，C 语言是 C++的一个部分，它们往往混合在一起使用，所以统称为 C/C++。C/C++主要用于 PC 软件开发、Linux 开发、游戏开发、单片机和嵌入式系统。 C 语言和 C++有着千丝万缕、剪也剪不断的关联，我们将在《C 语言和 C++到底有什么关系》一节中详细探讨。
Java	Java 是一门通用型的语言，可以用于网站后台开发、Android 开发、PC 软件开发，近年来又涉足了大数据领域（归功于 Hadoop 框架的流行）。
C#	C# 是微软开发的用来对抗 Java 的一门语言，实现机制和 Java 类似，不过 C# 显然失败了，目前主要用于 Windows 平台的软件开发，以及少量的网站后台开发。
Python	Python 也是一门通用型的语言，主要用于系统运维、网站后台开发、数据分析、人工智能、云计算等领域，近年来势头强劲，增长非常快。
PHP	PHP 是一门专用型的语言，主要用来开发网站后台程序。
JavaScript	JavaScript 最初只能用于网站前端开发，而且是前端开发的唯一语言，没有可替代性。近年来由于 Node.js 的流行，JavaScript 在网站后台开发中也占有了一席之地，并且在迅速增长。
Go 语言	Go 语言是 2009 年由 Google 发布的一款编程语言，成长非常迅速，在国内外已经有大量的应用。Go 语言主要用于服务器端的编程，对 C/C++、Java 都形成了不小的挑战。
Objective-C Swift	Objective-C 和 Swift 都只能用于苹果产品的开发，包括 Mac、MacBook、iPhone、iPad、iWatch 等。
汇编语言	汇编语言是计算机发展初期的一门语言，它的执行效率非常高，但是开发效率非常低，所以在常见的应用程序开发中不会使用汇编语言，只有在对效率和实时性要求极高的关键模块才会考虑汇编语言，例如操作系统内核、驱动、仪器仪表、工业控制等。

可以将不同的编程语言比喻成各国语言，为了表达同一个意思，可能使用不同的语句。例如，表达“世界你好”的意思：

- 汉语：世界你好；
- 英语：Hello World
- 法语：Bonjour tout le monde

在编程语言中，同样的操作也可能使用不同的语句。例如，在屏幕上显示“C 语言中文网”：

- C 语言：puts("C 语言中文网");
- PHP：echo "C 语言中文网";
- Java：System.out.println("C 语言中文网");

编程语言类似于人类语言，由直观的词汇组成，我们很容易就能理解它的意思，例如在 C 语言中，我们使用 puts 这个词让计算机在屏幕上显示出文字；puts 是 output string（输出字符串）的缩写。

使用 puts 在屏幕上显示“C 语言中文网”：

```
puts("C 语言中文网");
```

我们把要显示的内容放在("和")之间，并且在最后要有;。你必须要这样写，这是固定的格式。

总结：编程语言是用来控制计算机的一系列指令（Instruction），它有固定的格式和词汇（不同编程语言的格式和词汇不一样），必须遵守，否则就会出错，达不到我们的目的。

C 语言（C Language）是编程语言的一种，学习 C 语言，主要是学习它的格式和词汇。下面是一个 C 语言的完整例子，它会让计算机在屏幕上显示“C 语言中文网”。

> 这个例子主要演示 C 语言的一些固有格式和词汇，看不懂的读者不必深究，也不必问为什么是这样，后续我们会逐步给大家讲解。

```
#include <stdio.h>
int main(){
    puts("C 语言中文网");
    return 0;
}
```

这些具有特定含义的词汇、语句，按照特定的格式组织在一起，就构成了源代码（Source Code），也称源码或代码（Code）。

那么，C 语言肯定规定了源代码中每个词汇、语句的含义，也规定了它们该如何组织在一起，这就是语法（Syntax）。它与我们学习英语时所说的“语法”类似，都规定了如何将特定的词汇和句子组织成能听懂的语言。

编写源代码的过程就叫做编程（Program）。从事编程工作的人叫程序员（Programmer）。程序员也很幽默，喜欢自嘲，经常说自己的工作辛苦，地位低，像农民一样，所以称自己是”码农“，就是写代码的农民。也有人自嘲称是”程序猿“。

1.2 C 语言究竟是一门怎样的语言？

对于大部分程序员，C 语言是学习编程的第一门语言，很少有不了解 C 的程序员。

C 语言除了能让你了解编程的相关概念，带你走进编程的大门，还能让你明白程序的运行原理，比如，计算机的各个部件是如何交互的，程序在内存中是一种怎样的状态，操作系统和用户程序之间有着怎样的“爱恨情仇”，这些底层知识决定了你的发展高度，也决定了你的职业生涯。

如果你希望成为出类拔萃的人才，而不仅仅是码农，这么这些知识就是不可逾越的。也只有学习 C 语言，才能更好地了解它们。有了足够的基础，以后学习其他语言，会触类旁通，很快上手，7 天了解一门新语言不是神话。

C 语言概念少，词汇少，包含了基本的编程元素，后来的很多语言（C++、Java 等）都参考了 C 语言，说 C 语言是现代编程语言的开山鼻祖毫不夸张，它改变了编程世界。

正是由于 C 语言的简单，对初学者来说，学习成本小，时间短，结合本教程，能够快速掌握编程技术。

在世界编程语言排行榜中，C 语言、Java 和 C++ 霸占了前三名，拥有绝对优势，如下表所示：

--	2015 年 01 月榜单		2015 年 06 月榜单		2018 年 01 月榜单	
排名	语言	占有率	语言	占有率	语言	占有率
1	C	16.703%	Java	17.822%	Java	14.215%
2	Java	15.528%	C	16.788%	C	11.037%
3	Objective-C	6.953%	C++	7.756%	C++	5.603%
4	C++	6.705%	C#	5.056%	Python	4.678%
5	C#	5.045%	Objective-C	4.339%	C#	3.754%
6	PHP	3.784%	Python	3.999%	JavaScript	3.465%
7	JavaScript	3.274%	Visual Basic .NET	3.168%	Visual Basic .NET	3.261%
8	Python	2.613%	PHP	2.868%	R	2.549%
9	Perl	2.256%	JavaScript	2.295%	PHP	2.532%
10	PL/SQL	2.014%	Delphi/Object Pascal	1.869%	Perl	2.419%

2017 年，由于小型软件设备的蓬勃发展以及汽车行业底层软件的增加，C 语言还拿下了「年度编程语言」的桂冠，成为 2017 年全球增长最快的编程语言。下表列出了最近 10 年的“年度编程语言”：

年份	优胜者
2017	C

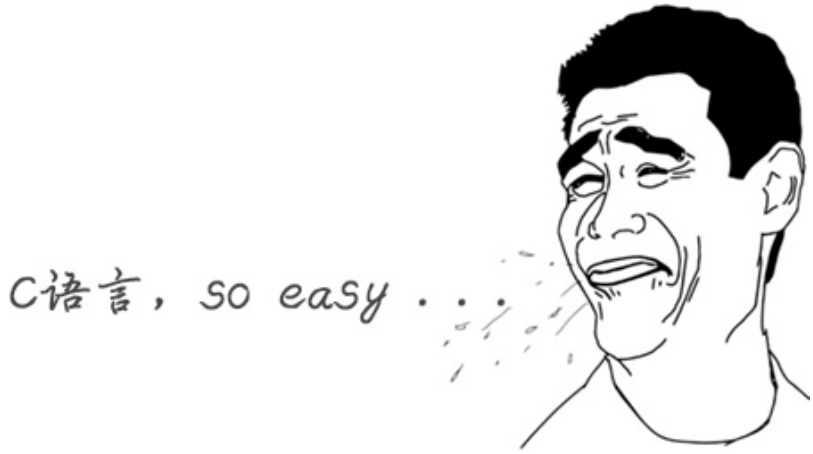
2016	Go
2015	Java
2014	JavaScript
2013	Transact-SQL
2012	Objective-C
2011	Objective-C
2010	Python
2009	Go
2008	C

C 语言诞生于 20 世纪 70 年代，年龄比我们都要大，我们将在《C 语言的三套标准：C89、C99 和 C11》一节中讲解更多关于 C 语言的历史。

当然，C 语言也不是没有缺点，毕竟是 70 后老人，有点落后时代，开发效率较低，后来人们又在 C 语言的基础上增加了面向对象的机制，形成了一门新的语言，称为 C++，我们将在《C 语言和 C++到底有什么关系》中讲解。

C 语言难吗？

和 Java、C++、Python、C#、JavaScript 等高级编程语言相比，C 语言涉及到的编程概念少，附带的标准库小，所以整体比较简洁，容易学习，非常适合初学者入门。



编程语言的发展大概经历了以下几个阶段：

汇编语言 --> 面向过程编程 --> 面向对象编程

- 汇编语言是编程语言的拓荒年代，它非常底层，直接和计算机硬件打交道，开发效率低，学习成本高；
- C 语言是面向过程的编程语言，已经脱离了计算机硬件，可以设计中等规模的程序了；
- Java、C++、Python、C#、PHP 等是面向对象的编程语言，它们在面向过程的基础上又增加了很多概念。

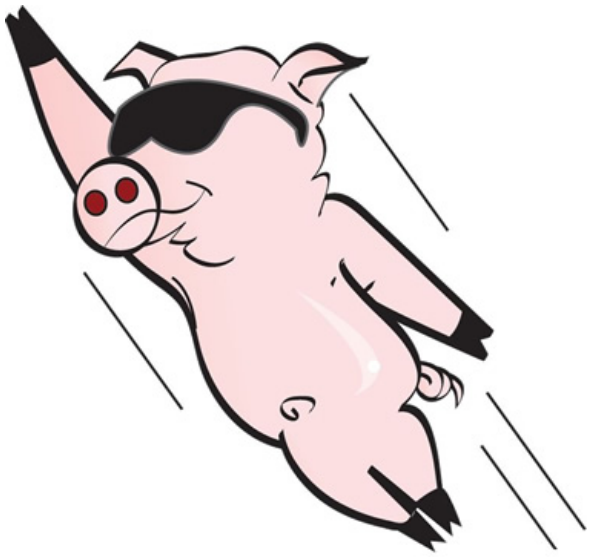
C 语言出现的时候，已经度过了编程语言的拓荒年代，具备了现代编程语言的特性，但是这个时候还没有出现“软件危机”，人们没有动力去开发更加高级的语言，所以也没有太复杂的编程思想。

也就是说，C 语言虽然是现代编程语言，但是它涉及到的概念少，词汇少，思想也简单。C 语言学习成本小，初学者能够在短时间内掌握编程技能，非常适合入门。

C 语言是计算机产业的核心语言

也许是机缘巧合，C 语言出现后不久，计算机产业开始爆发，计算机硬件越来越小型化，越来越便宜，逐渐进入政府机构，进入普通家庭，C 语言成了编程的主力军，获得了前所未有的成功，操作系统、常用软件、硬件驱动、底层组件、核心算法、数据库、小游戏等都使用 C 语言开发。

雷军说过，站在风口上，猪都能飞起来；C 语言就是那头猪，不管它好不好，反正它飞起来了。



C 语言在计算机产业大爆发阶段被万人膜拜，无疑会成为整个软件产业的基础，拥有核心地位。

软件行业的很多细分学科都是基于 C 语言的，学习数据结构、算法、操作系统、编译原理等都离不开 C 语言，所以大学将 C 语言作为一门公共课程，计算机相关专业的同学都要学习。

C 语言被誉为“上帝语言”，它不但奠定了软件产业的基础，还创造了很多其它语言，例如：

- PHP、Python 等都是用 C 语言开发出来的，虽然平时做项目的时候看不到 C 语言的影子，但是如果想深入学习 PHP 和 Python，那就要有 C 语言基础了。
- C++ 和 Objective-C 干脆在 C 语言的基础上直接进行扩展，增加一些新功能后变成了新的语言，所以学习 C++ 和 Objective-C 之前也要先学习 C 语言。

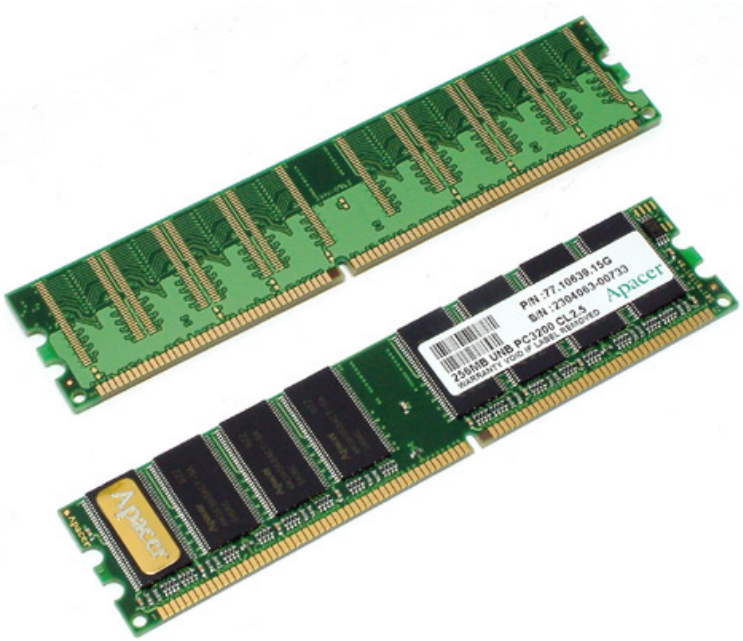
C 语言是有史以来最为重要的编程语言：要进入编程行业高手级别必学 C 语言，要挣大钱必学 C 语言，要做黑客、红客必学 C 语言，要面试名企、外企、高薪职位必学 C 语言。

1.3 C 语言是菜鸟和大神的分水岭

作为一门古老的编程语言，C 语言已经坚挺了好几十年了，初学者从 C 语言入门，大学将 C 语言视为基础课程。不管别人如何抨击，如何唱衰，C 语言就是屹立不倒；Java、C#、Python、PHP、Perl 等都有替代方案，它们都可以倒下，唯独 C 语言不行。

程序是在内存中运行的（我们将在《载入内存，让程序运行起来》一节中详细说明），一名合格的程序员必须了解内存，学习 C 语言是了解内存布局的最简单、最直接、最有效的途径，C 语言简直是为内存而生的，它比任何一门编程语言都贴近内存。

所谓内存，就是我们常说的内存条，就是下图这个玩意，相信你肯定见过。



所有的程序都在拼尽全力节省内存，都在不遗余力提高内存使用效率，计算机的整个发展过程都在围绕内存打转，不断地优化内存布局，以保证可以同时运行多个程序。

不了解内存，就学不会进程和线程，就没有资格玩中大型项目，没有资格开发底层组件，没有资格架构一个系统，命中注定你就是一个菜鸟，成不了什么气候。

以上这点我有深刻的体会！工作期间我曾专注于网站开发，虽然能够设计出界面漂亮、体验良好的网页，但是对内存泄漏、多线程、共享内存等底层概念一窍不通，感觉和周围同事的差距很大，这让我非常郁闷，不知道如何突破。我曾多次尝试学习内存和线程，也找了很多资料，但是无论如何都啃不懂，到头来还是一头雾水。

离职后我全职运营 C 语言中文网，于是决定再次系统、深入、全面地学习 C 语言，并结合 C 语言去了解一些内存知识，这个时候我才发现，原来 C 语言就是为内存而生的，C 语言的设计和内存的布局是严密贴合的，我因为学习 C 语言而吃透了内存，了解了计算机内存是如何分布和组织的。

> C 语言无时无刻不在谈内存，内存简直就是如影随形，你不得不去研究它。

至关重要的一点是，我能够把内存和具体的编程知识以及程序的运行过程结合起来，真正做到了学以致用，让概念落地，而不是空谈，这才是最难得的。

另外一个惊喜是，攻克内存后我竟然也能够理解进程和线程了，原来进程和线程也是围绕内存打转的，从一定程度上讲，它们的存在也是为了更加高效地利用内存。

从 C 语言到内存，从内存到进程和线程，环环相扣：不学 C 语言就吃不透内存，不学内存就吃不透进程和线程。

我感觉自己瞬间升华了，达到了一个新的高度，之前的很多谜团都解开了，和大神交流也没有障碍了。

「内存 + 进程 + 线程」这几个最基本的计算机概念是菜鸟和大神的分水岭，也只有学习 C 语言才能透彻地理解它们。Java、C#、PHP、Python、JavaScript 程序员工作几年后会遇到瓶颈，有很多人会回来学习 C 语言，重拾底层概念，让自己再次突破。

1.4 学编程难吗？多久能入门？

这篇文章主要是解答初学者的疑惑，没有信心的读者看了会吃一颗定心丸，浮躁的读者看了会被泼一盆冷水。

学编程难吗？

编程是一门技术，我也不知道它难不难，我只知道，只要你想学，肯定能学会。每个人的逻辑思维能力不同，兴趣点不同，总有一部分人觉得容易，一部分人觉得吃力。

在我看来，技术就是一层窗户纸，是有道理可以遵循的，最起码要比搞抽象的艺术容易很多。

但是，隔行如隔山，学好编程也不是一朝一夕的事，想“吃快餐”的读者可以退出编程界了，浮躁的人搞不了技术。

在技术领域，编程的入门门槛很低，互联网的资料很多，只要你有一台计算机，一根网线，具备初中学历，就可以学习，投资在 5000RMB 左右。

不管是技术还是非技术，要想有所造诣，都必须潜心钻研，没有几年功夫不会鹤立鸡群。所以请先问问你自己，你想学编程吗，你喜欢吗，如果你觉得自己对编程很感兴趣，想了解软件或网站是怎么做的，那么就不要再问这个问

题了，尽管去学就好了。

多久能学会编程？

这是一个没有答案的问题。每个人投入的时间、学习效率和基础都不一样。如果你每天都拿出大把的时间来学习，那么两三个月就可以学会 C/C++，不到半年时间就可以编写出一些软件。

但是有一点可以肯定，几个月从小白成长为大神是绝对不可能的。要想出类拔萃，没有几年功夫是不行的。学习编程不是看几本书就能搞定的，需要你不断的练习，编写代码，积累零散的知识点，代码量跟你的编程水平直接相关，没有几万行代码，没有拿得出手的作品，怎能称得上“大神”。

每个人程序员都是这样过来的，开始都是一头雾水，连输出九九乘法表都很吃力，只有通过不断练习才能熟悉，这是一个强化思维方式的过程。

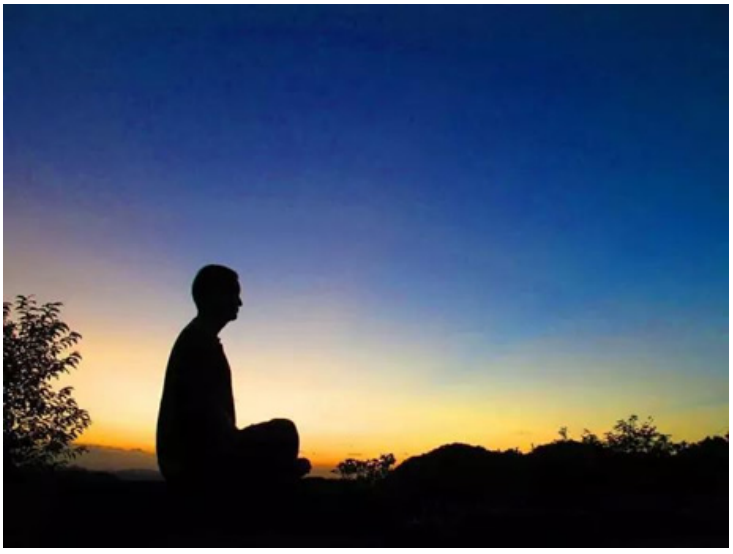
知识点可以在短时间内了解，但是思维方式和编程经验需要不断实践才能强化，这就是为什么很多初学者已经了解了 C 语言的基本概念，但是仍然不会编写代码的原因。

程序员被戏称为”码农“，意思是写代码的农民，要想成为一个合格的农民，必须要脚踏实地辛苦耕耘。

也不要压力太大，一切编程语言都是纸老虎，一层窗户纸，只要开窍了，就容易了。

“浸泡”理论

这是我自己独创的一个理论，意思是说：一个人要想在某一方面有所成就，就必须有半年以上的时间，每天花 10 个小时“浸泡”在这件事情上，最终一定会有所收获。



很多领域都是「一年打基础，两年见成效，三年有突破」，但是很多人在不到一年的时间里就放弃了，总觉得这个行业太难，不太适合自己。

轻言放弃是很可怕的，你要知道，第一次放弃只是浪费了时间，第二次放弃会打击你的信心，第三次放弃会摧毁你的意志，你就再也没有尝试的勇气了，“蹉跎人生”就是这么来的。

你也不要羡慕那些富二代官二代，你以为人生就是一次百米短跑，你赢了就是赢了，其实人生是一场接力赛，你的父辈祖辈都得赢，那些富二代官二代从好几十年以前就开始积累了。

所以，沉下一颗心来，从现在开始积累吧，有执念的人最可怕。

1.5 英语和数学不好，能学编程吗？

很多初学者认为，编程语言是由英文组成的，而且会涉及很多算法，自己的英语和数学功底不好，到底能不能学会编程呢？

英语基础不好可以学会编程吗？

首先，学习编程需要你有英语基础；但是，要求并不高，初中水平完全可以胜任。

编程语言起源于美国，是由英文构成的，其中包括几十个英文的关键字以及几百个英文的函数，除非需要对文本进行处理，否则一般不会出现中文。但是，它们都是孤立的单词，不构成任何语句，不涉及任何语法。

几十个关键字不多，用得多了自然会记住，相信大家也不会担心。下面是 C 语言中的 32 个关键字：

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

几百个函数就没人能记住了（包括我），也不用记住，查询文档即可，每种编程语言都会提供配套的文档。常用到的函数也就几十个，记住它们就足够应付日常开发了，生僻的函数查询文档即可。

此外，我推荐大家安装有道词典，它的划词取词功能非常棒，选中一个单词或者句子能够及时翻译，这对大家记忆和理解代码非常有帮助。

对于英文资料

如果你希望达到很高的造诣，希望被人称为“大神”，那么肯定要阅读英文的技术资料（不是所有资料都被翻译成了中文），初中水平就有点吃力了。

不过，长期阅读英文会提高你的英文水平，只要你坚持一段时间，即使只有初中水平，我相信借助有道词典也会提高很快。

数学基础不好可以学编程吗？

谈到数学，那真是多虑了，它根本不构成障碍，会加减乘除就能学编程。

编程语言确实涉及到很多算法，有一些还需要高等数学知识，但是，这些算法都已经被封装好了，你直接拿来用就可以，根本不用你重复造轮子。

另外，这些算法都是在很深的底层为我们默默的工作，初级程序员根本不会涉及到算法，即使是别人已经封装好的算法，一般也没有机会使用，所以，你就别瞎操心了。我学编程八年了，至今都没有设计过什么算法，也没有使用过别人的算法。

1.6 初中毕业能学会编程吗？

首先，初中毕业能学会编程，但是，一般达不到太高的造诣。

编程是知识密集型的行业，需要很强的学习能力。初中就毕业了，肯定学习不好。大家的智商都差不多，成绩不好一般都是学习能力差。什么是学习能力呢？这包括专注能力、理解能力、自律能力等。

专注能力

有很多人不能专注于一件事情，容易走神，人虽然在，心已经飞了，根本钻研不进去。

理解能力

也可以说是逻辑思维能力。

同一道题目，有些人一看就知道思路，就知道如何切入；也有些人绞尽脑汁都想不到方案，不知道从哪里下手。

同一个现象，有些人觉得就应该这样，这是理所当然的，就像公理一样，不需要理由；也有些人觉得很费解，为什么是这样呢，理由是什么呢？

自律能力

学习是一件枯燥的事情，有些人能坚持下来，有些人就熬不住。

我也不知道为什么人的学习能力有差异，难道是与生俱来的？有没有心理学家给科普一下，让我涨涨姿势。

拥有良好的学习能力是一件幸事，你将终生受益，这个社会越来越奖励知识分子。

1.7 C 语言和 C++到底有什么关系？

1.8 学了 C 语言到底能做什么，能从事什么工作？

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

1.9 进制详解：二进制、八进制和十六进制

我们平时使用的数字都是由 0~9 共十个数字组成的，例如 1、9、10、297、952 等，一个数字最多能表示九，如果要表示十、十一、二十九、一百等，就需要多个数字组合起来。

例如表示 5+8 的结果，一个数字不够，只能”进位“，用 13 来表示；这时”进一位“相当于十，”进两位“相当于二十。

因为逢十进一（满十进一），也因为只有 0~9 共十个数字，所以叫做十进制（Decimalism）。十进制是在人类社会发展过程中自然形成的，它符合人们的思维习惯，例如人类有十根手指，也有十根脚趾。

进制也就是进位制。进行加法运算时逢 X 进一（满 X 进一），进行减法运算时借一当 X，这就是 X 进制，这种进制也就包含 X 个数字，基数为 X。十进制有 0~9 共 10 个数字，基数为 10，在加减法运算中，逢十进一，借一当十。

二进制

我们不妨将思维拓展一下，既然可以用 0~9 共十个数字来表示数值，那么也可以用 0、1 两个数字来表示数值，这就是二进制（Binary）。例如，数字 0、1、10、111、100、1000001 都是有效的二进制。

在计算机内部，数据都是以二进制的形式存储的，二进制是学习编程必须掌握的基础。本节我们先讲解二进制的概念，下节讲解数据在内存中的存储，让大家学以致用。

二进制加减法和十进制加减法的思想是类似的：
对于十进制，进行加法运算时逢十进一，进行减法运算时借一当十；
对于二进制，进行加法运算时逢二进一，进行减法运算时借一当二。

下面两张示意图详细演示了二进制加减法的运算过程。

1) 二进制加法：1+0=1、1+1=10、11+10=101、111+111=1110

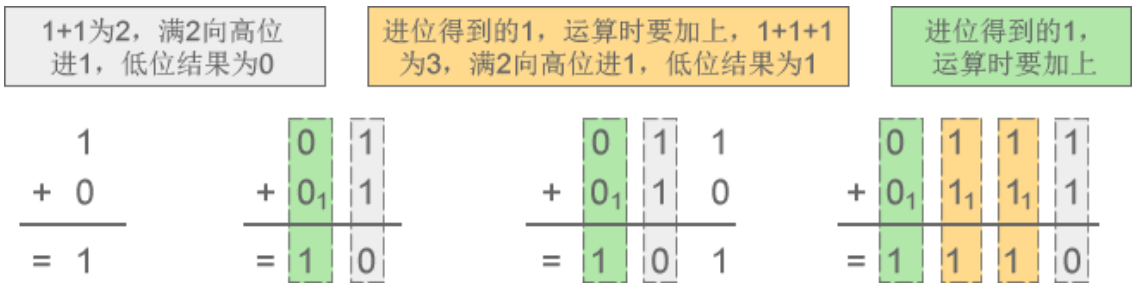


图 1：二进制加法示意图

2) 二进制减法：1-0=1、10-1=1、101-11=10、1100-111=101

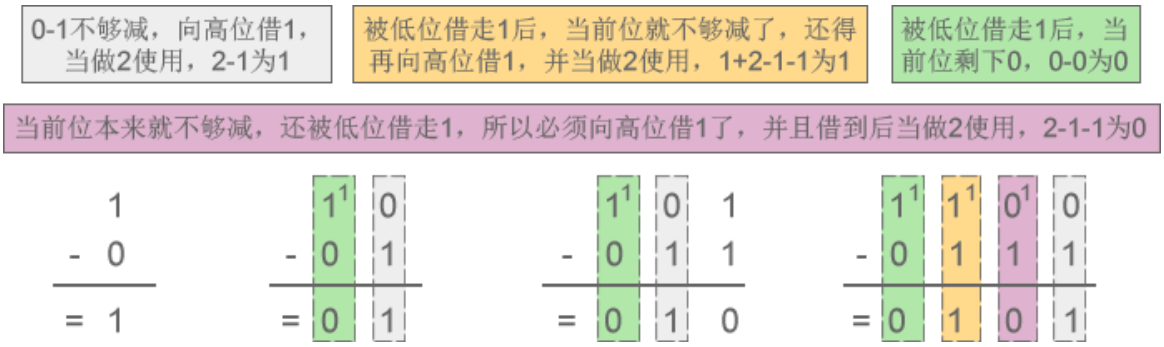


图 2：二进制减法示意图

八进制

除了二进制，C 语言还会使用到八进制。

八进制有 0~7 共 8 个数字，基数为 8，加法运算时逢八进一，减法运算时借一当八。例如，数字 0、1、5、7、14、733、67001、25430 都是有效的八进制。

下面两张图详细演示了八进制加减法的运算过程。

1) 八进制加法：3+4=7、5+6=13、75+42=137、2427+567=3216

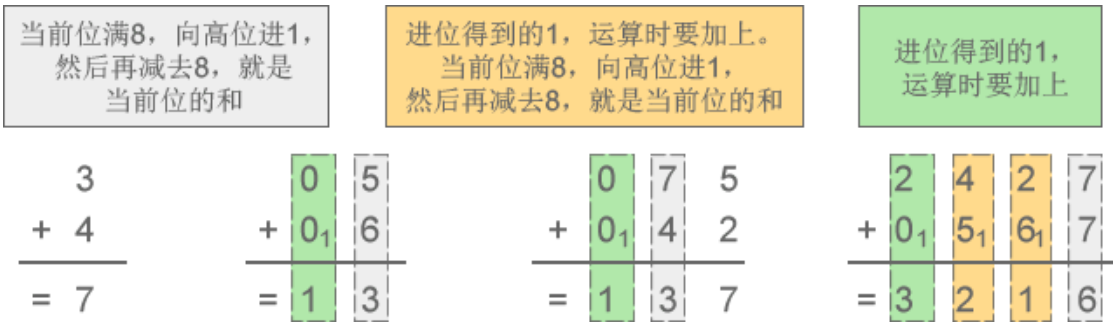


图 3：八进制加法示意图

2) 八进制减法：6-4=2、52-27=23、307-141=146、7430-1451=5757

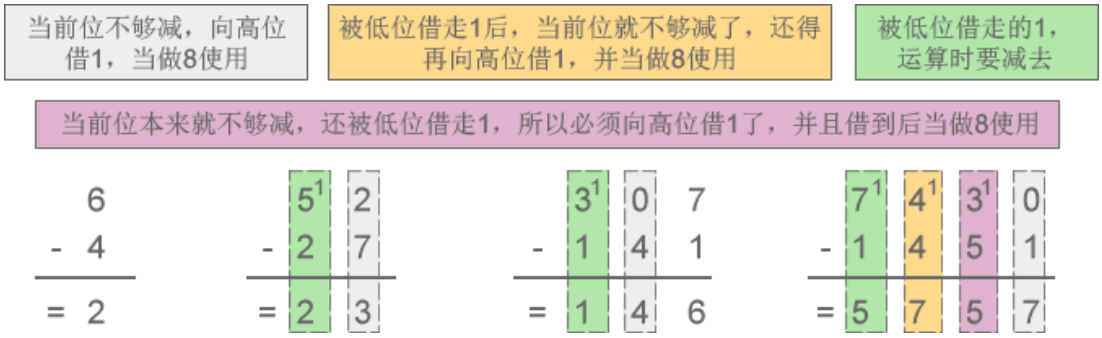


图 4：八进制减法示意图

十六进制

除了二进制和八进制，十六进制也经常使用，甚至比八进制还要频繁。

十六进制中，用 A 来表示 10，B 表示 11，C 表示 12，D 表示 13，E 表示 14，F 表示 15，因此有 0~F 共 16 个数字，基数为 16，加法运算时逢 16 进 1，减法运算时借 1 当 16。例如，数字 0、1、6、9、A、D、F、419、EA32、80A3、BC00 都是有效的十六进制。

> 注意，十六进制中的字母不区分大小写，ABCDEF 也可以写作 abcdef。

下面两张图详细演示了十六进制加减法的运算过程。

1) 十六进制加法：6+7=D、18+BA=D2、595+792=D27、2F87+F8A=3F11

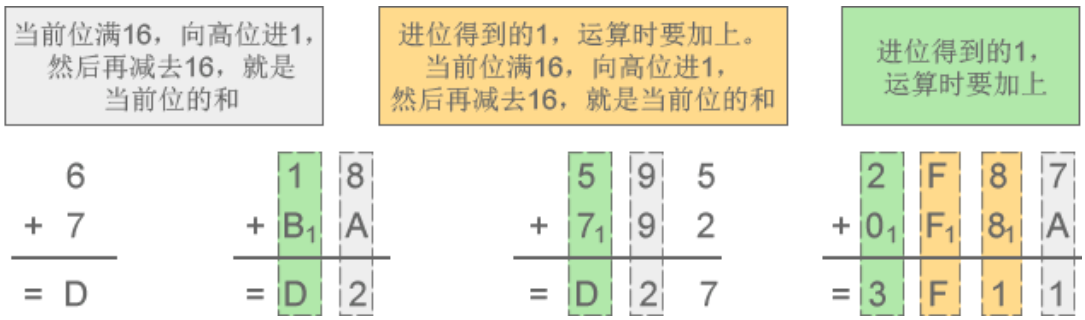


图 5：十六进制加法示意图

2) 十六进制减法：D-3=A、52-2F=23、E07-141=CC6、7CA0-1CB1=5FEF

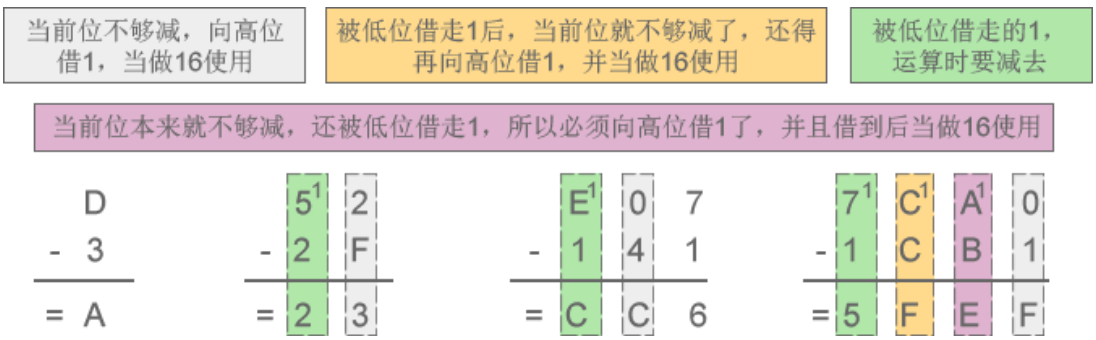


图 6：十六进制减法示意图

1.10 不同进制之间的转换

> 对于基础薄弱的读者，本节的内容可能略显晦涩和枯燥，如果你觉得吃力，可以暂时跳过，基本不会影响后续章节的学习，等用到的时候再来阅读。

上节我们对二进制、八进制和十六进制进行了说明，本节重点讲解不同进制之间的转换，这在编程中经常会用到，尤其是 C 语言。

将二进制、八进制、十六进制转换为十进制

二进制、八进制和十六进制向十进制转换都非常容易，就是“按权相加”。所谓“权”，也即“位权”。

假设当前数字是 N 进制，那么：

- 对于整数部分，从右往左看，第 i 位的位权等于 N^{i-1}
- 对于小数部分，恰好相反，要从左往右看，第 j 位的位权为 N^{-j}。

更加通俗的理解是，假设一个多位数（由多个数字组成的数）某位上的数字是 1，那么它所表示的数值大小就是该位的位权。

1) 整数部分

例如，将八进制数字 53627 转换成十进制：

$$53627 = 5\times8^4 + 3\times8^3 + 6\times8^2 + 2\times8^1 + 7\times8^0 = 22423\text{（十进制）}$$

从右往左看，第 1 位的位权为 8^0=1，第 2 位的位权为 8^1=8，第 3 位的位权为 8^2=64，第 4 位的位权为 8^3=512，第 5 位的位权为 8^4=4096 …… 第 n 位的位权就为 8^{n-1}。将各个位的数字乘以位权，然后再相加，就得到了十进制形式。

> 注意，这里我们需要以十进制形式来表示位权。

再如，将十六进制数字 9FA8C 转换成十进制：

$$9FA8C = 9\times16^4 + 15\times16^3 + 10\times16^2 + 8\times16^1 + 12\times16^0 = 653964\text{（十进制）}$$

从右往左看，第 1 位的位权为 16^0=1，第 2 位的位权为 16^1=16，第 3 位的位权为 16^2=256，第 4 位的位权为 16^3=4096，第 5 位的位权为 16^4=65536 …… 第 n 位的位权就为 16^{n-1}。将各个位的数字乘以位权，然后再相加，就得到了十进制形式。

将二进制数字转换成十进制也是类似的道理：

$$11010 = 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 26\text{（十进制）}$$

从右往左看，第 1 位的位权为 2^0=1，第 2 位的位权为 2^1=2，第 3 位的位权为 2^2=4，第 4 位的位权为 2^3=8，第 5 位的位权为 2^4=16 …… 第 n 位的位权就为 2^{n-1}。将各个位的数字乘以位权，然后再相加，就得到了十进制形式。

2) 小数部分

例如，将八进制数字 423.5176 转换成十进制：

$$423.5176 = 4\times8^2 + 2\times8^1 + 3\times8^0 + 5\times8^{-1} + 1\times8^{-2} + 7\times8^{-3} + 6\times8^{-4} = 275.65576171875\text{（十进制）}$$

小数部分和整数部分相反，要从左往右看，第 1 位的位权为 8^{-1}=1/8，第 2 位的位权为 8^{-2}=1/64，第 3 位的位权为 8^{-3}=1/512，第 4 位的位权为 8^{-4}=1/4096 …… 第 m 位的位权就为 8^{-m}。

再如，将二进制数字 1010.1101 转换成十进制：

1010.1101 = $1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 + 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}$ = 10.8125（十进制）

小数部分和整数部分相反，要从左往右看，第 1 位的位权为 2^{-1}=1/2，第 2 位的位权为 2^{-2}=1/4，第 3 位的位权为 2^{-3}=1/8，第 4 位的位权为 2^{-4}=1/16 …… 第 m 位的位权就为 2^{-m}。

更多转换成十进制的例子：

- 二进制：1001 = $1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$ = 8 + 0 + 0 + 1 = 9（十进制）
- 二进制：101.1001 = $1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 0\times2^{-3} + 1\times2^{-4}$ = 4 + 0 + 1 + 0.5 + 0 + 0 + 0.0625 = 5.5625（十进制）
- 八进制：302 = $3\times8^2 + 0\times8^1 + 2\times8^0$ = 192 + 0 + 2 = 194（十进制）
- 八进制：302.46 = $3\times8^2 + 0\times8^1 + 2\times8^0 + 4\times8^{-1} + 6\times8^{-2}$ = 192 + 0 + 2 + 0.5 + 0.09375= 194.59375（十进制）
- 十六进制：EA7 = $14\times16^2 + 10\times16^1 + 7\times16^0$ = 3751（十进制）

将十进制转换为二进制、八进制、十六进制

将十进制转换为其它进制时比较复杂，整数部分和小数部分的算法不一样，下面我们分别讲解。

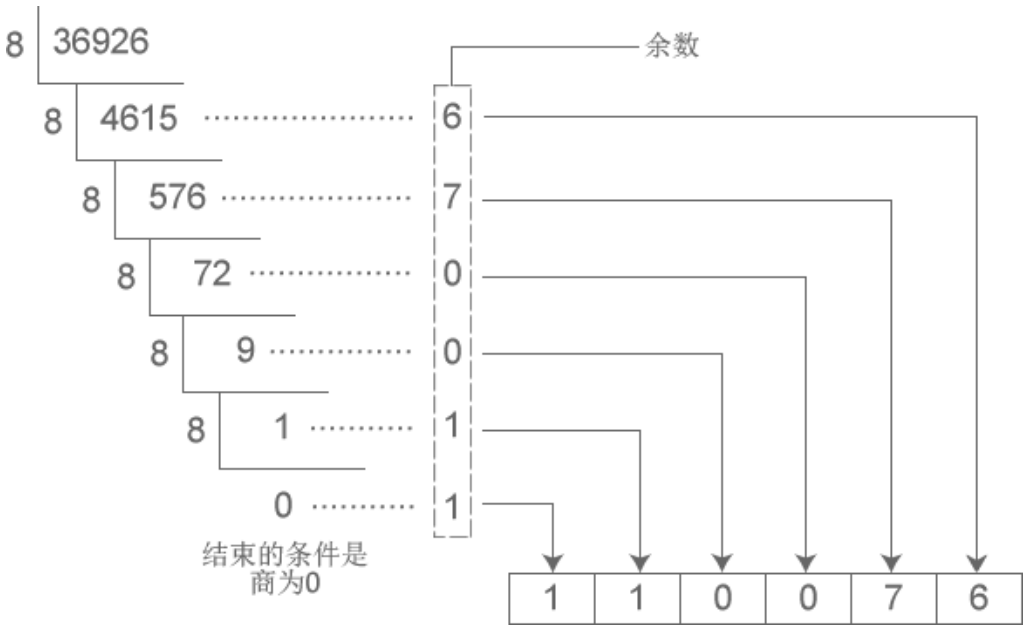
1) 整数部分

十进制整数转换为 N 进制整数采用“除 N 取余，逆序排列”法。具体做法是：

- 将 N 作为除数，用十进制整数除以 N，可以得到一个商和余数；
- 保留余数，用商继续除以 N，又得到一个新的商和余数；
- 仍然保留余数，用商继续除以 N，还会得到一个新的商和余数；
- ……
- 如此反复进行，每次都保留余数，用商接着除以 N，直到商为 0 时为止。

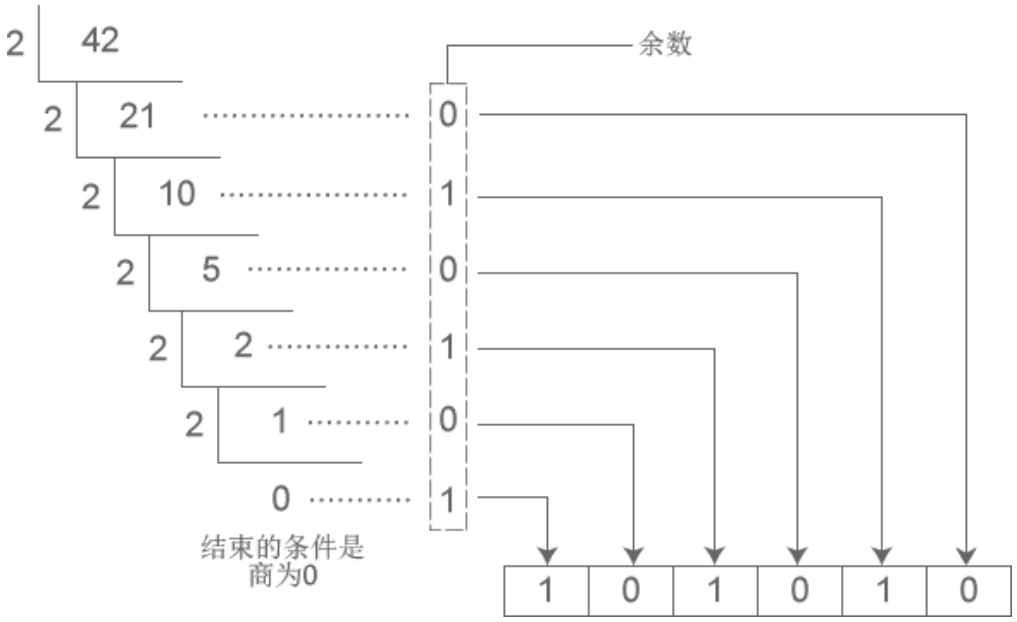
把先得到的余数作为 N 进制数的低位数字，后得到的余数作为 N 进制数的高位数字，依次排列起来，就得到了 N 进制数字。

下图演示了将十进制数字 36926 转换成八进制的过程：



从图中得知，十进制数字 36926 转换成八进制的结果为 110076。

下图演示了将十进制数字 42 转换成二进制的过程：



从图中得知，十进制数字 42 转换成二进制的结果为 101010。

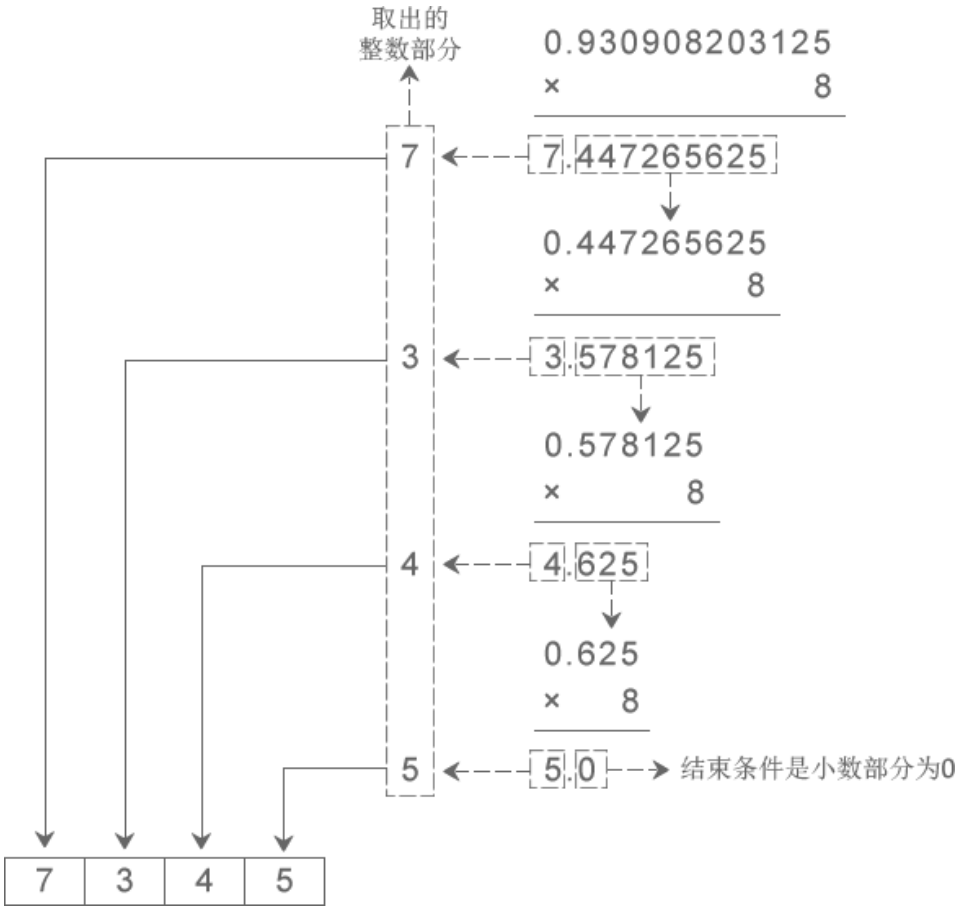
2) 小数部分

十进制小数转换成 N 进制小数采用“乘 N 取整，顺序排列”法。具体做法是：

- 用 N 乘以十进制小数，可以得到一个积，这个积包含了整数部分和小数部分；
- 将积的整数部分取出，再用 N 乘以余下的小数部分，又得到一个新的积；
- 再将积的整数部分取出，继续用 N 乘以余下的小数部分；
- ……
- 如此反复进行，每次都取出整数部分，用 N 接着乘以小数部分，直到积中的小数部分为 0，或者达到所要求的精度为止。

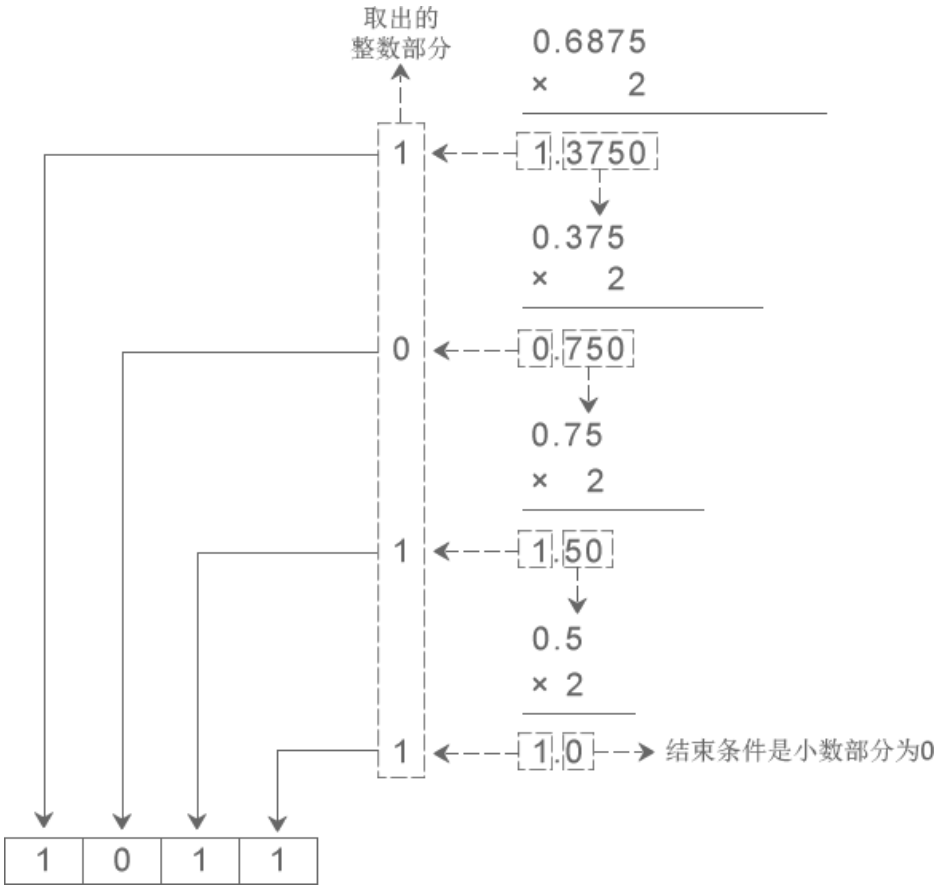
把取出的整数部分按顺序排列起来，先取出的整数作为 N 进制小数的高位数字，后取出的整数作为低位数字，这样就得到了 N 进制小数。

下图演示了将十进制小数 0.930908203125 转换成八进制小数的过程：



从图中得知，十进制小数 0.930908203125 转换成八进制小数的结果为 0.7345。

下图演示了将十进制小数 0.6875 转换成二进制小数的过程：



从图中得知，十进制小数 0.6875 转换成二进制小数的结果为 0.1011。

如果一个数字既包含了整数部分又包含了小数部分，那么将整数部分和小数部分开，分别按照上面的方法完成转换，然后再合并在一起即可。例如：

- 十进制数字 36926.930908203125 转换成八进制的结果为 110076.7345；
- 十进制数字 42.6875 转换成二进制的结果为 101010.1011。

下表列出了前 17 个十进制整数与二进制、八进制、十六进制的对应关系：

十进制	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
二进制	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000
八进制	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17	20
十六进制	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10

注意，十进制小数转换成其他进制小数时，结果有可能是一个无限位的小数。请看下面的例子：

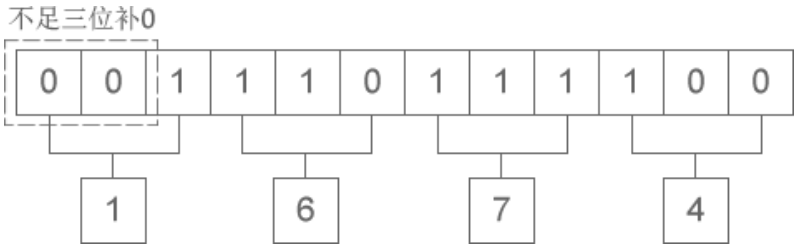
- 十进制 0.51 对应的二进制为 0.10000010100011110101110000101000111101011...，是一个循环小数；
- 十进制 0.72 对应的二进制为 0.10111000010100011110101110000101000111110...，是一个循环小数；
- 十进制 0.625 对应的二进制为 0.101，是一个有限小数。

二进制和八进制、十六进制的转换

其实，任何进制之间的转换都可以使用上面讲到的方法，只不过有时比较麻烦，所以一般针对不同的进制采取不同的方法。将二进制转换为八进制和十六进制时就有非常简洁的方法，反之亦然。

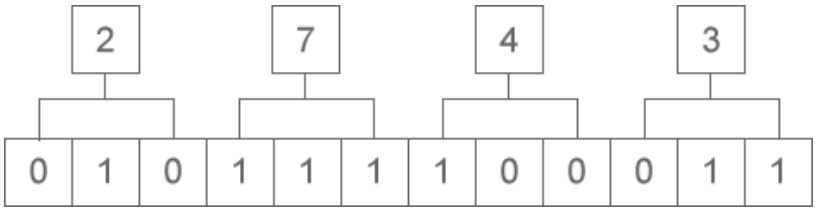
1) 二进制整数和八进制整数之间的转换

二进制整数转换为八进制整数时，每三位二进制数字转换为一位八进制数字，运算的顺序是从低位向高位依次进行，高位不足三位用零补齐。下图演示了如何将二进制整数 1110111100 转换为八进制：



从图中可以看出，二进制整数 1110111100 转换为八进制的结果为 1674。

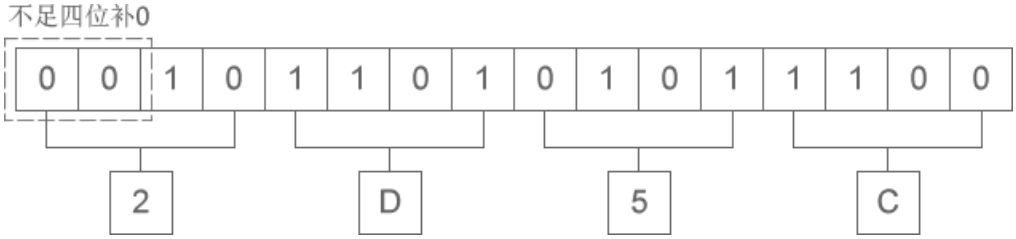
八进制整数转换为二进制整数时，思路是相反的，每一位八进制数字转换为三位二进制数字，运算的顺序也是从低位向高位依次进行。下图演示了如何将八进制整数 2743 转换为二进制：



从图中可以看出，八进制整数 2743 转换为二进制的结果为 10111100011。

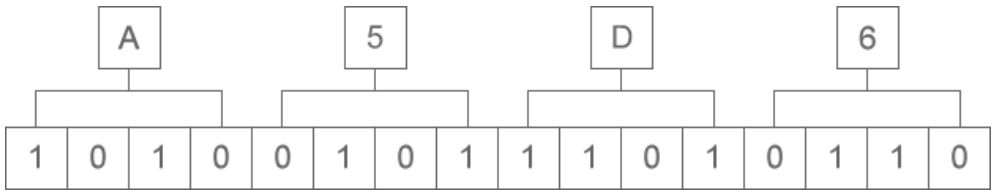
2) 二进制整数和十六进制整数之间的转换

二进制整数转换为十六进制整数时，每四位二进制数字转换为一位十六进制数字，运算的顺序是从低位向高位依次进行，高位不足四位用零补齐。下图演示了如何将二进制整数 10 1101 0101 1100 转换为十六进制：



从图中可以看出，二进制整数 10 1101 0101 1100 转换为十六进制的结果为 2D5C。

十六进制整数转换为二进制整数时，思路是相反的，每一位十六进制数字转换为四位二进制数字，运算的顺序也是从低位向高位依次进行。下图演示了如何将十六进制整数 A5D6 转换为二进制：



从图中可以看出，十六进制整数 A5D6 转换为二进制的结果为 1010 0101 1101 0110。

在 C 语言编程中，二进制、八进制、十六进制之间几乎不会涉及小数的转换，所以这里我们只讲整数的转换，大家学以致用足以。另外，八进制和十六进制之间也极少直接转换，这里我们也不再讲解了。

总结

本节前面两部分讲到的转换方法是通用的，任何进制之间的转换都可以采用，只是有时比较麻烦而已。二进制和八进制、十六进制之间的转换有非常简洁的方法，所以没有采用前面的方法。

1.11 数据在内存中的存储（二进制形式存储）

计算机要处理的信息是多种多样的，如数字、文字、符号、图形、音频、视频等，这些信息在人们的眼里是不同的。但对于计算机来说，它们在内存中都是一样的，都是以二进制的形式来表示。

要想学习编程，就必须了解二进制，它是计算机处理数据的基础。

内存条是一个非常精密的部件，包含了上亿个电子元器件，它们很小，达到了纳米级别。这些元器件，实际上就是电路；电路的电压会变化，要么是 0V，要么是 5V，只有这两种电压。5V 是通电，用 1 来表示，0V 是断电，用 0 来表示。所以，一个元器件有 2 种状态，0 或者 1。

我们通过电路来控制这些元器件的通断电，会得到很多 0、1 的组合。例如，8 个元器件有 2^8=256 种不同的组合，

16 个元器件有 2^{16}=65536 种不同的组合。虽然一个元器件只能表示 2 个数值，但是多个结合起来就可以表示很多数值了。

我们可以给每一种组合赋予特定的含义，例如，可以分别用 1101000、00011100、11111111、00000000、01010101、10101010 来表示 C、语、言、中、文、网 这几个字，那么结合起来 1101000 00011100 11111111 00000000 01010101 10101010 就表示”C 语言中文网 “。

一般情况下我们不一个一个的使用元器件，而是将 8 个元器件看做一个单位，即使表示很小的数，例如 1，也需要 8 个，也就是 00000001。

1 个元器件称为 1 比特（Bit）或 1 位，8 个元器件称为 1 字节（Byte），那么 16 个元器件就是 2Byte，32 个就是 4Byte，以此类推：

- 8×1024 个元器件就是 1024Byte，简写为 1KB；
- 8×1024×1024 个元器件就是 1024KB，简写为 1MB；
- 8×1024×1024×1024 个元器件就是 1024MB，简写为 1GB。

现在，你知道 1GB 的内存有多少个元器件了吧。我们通常所说的文件大小是多少 KB、多少 MB，就是这个意思。

单位换算：

- 1Byte = 8 Bit

- 1KB = 1024Byte = 2^{10}Byte
- 1MB = 1024KB = 2^{20}Byte
- 1GB = 1024MB = 2^{30}Byte
- 1TB = 1024GB = 2^{40}Byte
- 1PB = 1024TB = 2^{50}Byte
- 1EB = 1024PB = 2^{60}Byte

我们平时使用计算机时，通常只会设计到 KB、MB、GB、TB 这几个单位，PB 和 EB 这两个高级单位一般在大数据处理过程中才会用到。

你看，在内存中没有 abc 这样的字符，也没有 gif、jpg 这样的图片，只有 0 和 1 两个数字，计算机也只认识 0 和 1。所以，计算机使用二进制，而不是我们熟悉的十进制，写入内存中的数据，都会被转换成 0 和 1 的组合。

我们将在《C 语言调试》中的《查看、修改运行时的内存》一节教大家如何操作 C 语言程序的内存。

程序员的幽默

为了加深印象，最后给大家看个笑话。

程序员 A：“哥们儿，最近手头紧，借点钱？”

程序员 B：“成啊，要多少？”

程序员 A：“一千行不？”

程序员 B：“咱俩谁跟谁！给你凑个整，1024，拿去吧。”

你看懂这个笑话了吗？请选出正确答案。

A) 因为他同情程序员 A，多给他 24 块
B) 这个程序员不会数数，可能是太穷饿晕了
C) 这个程序员故意的，因为他独裁的老婆规定 1024 是整数
D) 就像 100 是 10 的整数次方一样，1024 是 2 的整数次方，对于程序员就是整数

1.12 载入内存，让程序运行起来

如果你的电脑上安装了 QQ，你希望和好友聊天，会双击 QQ 图标，打开 QQ 软件，输入账号和密码，然后登录就可以了。

那么，QQ 是怎么运行起来的呢？

首先，有一点你要明确，你安装的 QQ 软件是保存在硬盘中的。

双击 QQ 图标，操作系统就会知道你要运行这个软件，它会在硬盘中找到你安装的 QQ 软件，将数据（安装的软件本质上就是很多数据的集合）复制到内存。对！就是复制到内存！QQ 不是在硬盘中运行的，而是在内存中运行的。

为什么呢？因为内存的读写速度比硬盘快很多。

对于读写速度，内存 > 固态硬盘 > 机械硬盘。机械硬盘是靠电机带动盘片转动来读写数据的，而内存条通过电路来读写数据，电机的转速肯定没有电的传输速度（几乎是光速）快。虽然固态硬盘也是通过电路来读写数据，但是因为与内存的控制方式不一样，速度也不及内存。

所以，不管是运行 QQ 还是编辑 Word 文档，都是先将硬盘上的数据复制到内存，才能让 CPU 来处理，这个过程就叫作载入内存（Load into Memory）。完成这个过程需要一个特殊的程序（软件），这个程序就叫做加载器（Loader）。

CPU 直接与内存打交道，它会读取内存中的数据进行处理，并将结果保存到内存。如果需要保存到硬盘，才会将内存中的数据复制到硬盘。

例如，打开 Word 文档，输入一些文字，虽然我们看到的不一样了，但是硬盘中的文档没有改变，新增的文字暂时保存到了内存，Ctrl+S 才会保存到硬盘。因为内存断电后会丢失数据，所以如果你编辑完 Word 文档忘记保存就关机了，那么你将永远无法找回这些内容。

虚拟内存

如果我们运行的程序较多，占用的空间就会超过内存（内存条）容量。例如计算机的内存容量为 2G，却运行着 10 个程序，这 10 个程序共占用 3G 的空间，也就意味着需要从硬盘复制 3G 的数据到内存，这显然是不可能的。

操作系统（Operating System，简称 OS）为我们解决了这个问题：当程序运行需要的空间大于内存容量时，会将内存中暂时不用的数据再写回硬盘；需要这些数据时再从硬盘中读取，并将另外一部分不用的数据写入硬盘。这样，硬盘中就会有一部分空间用来存放内存中暂时不用的数据。这一部分空间就叫做虚拟内存（Virtual Memory）。

3G - 2G = 1G，上面的情况需要在硬盘上分配 1G 的虚拟内存。

硬盘的读写速度比内存慢很多，反复交换数据会消耗很多时间，所以如果你的内存太小，会严重影响计算机的运行速度，甚至会出现”卡死“现象，即使 CPU 强劲，也不会有大的改观。如果经济条件允许，建议将内存升级为 4G，在 win7、win8、win10 下运行软件就会比较流畅了。

关于内存的更多知识，大家可以阅读《C 语言内存精讲》，我敢保证你将会顿悟。

总结：CPU 直接从内存中读取数据，处理完成后将结果再写入内存。

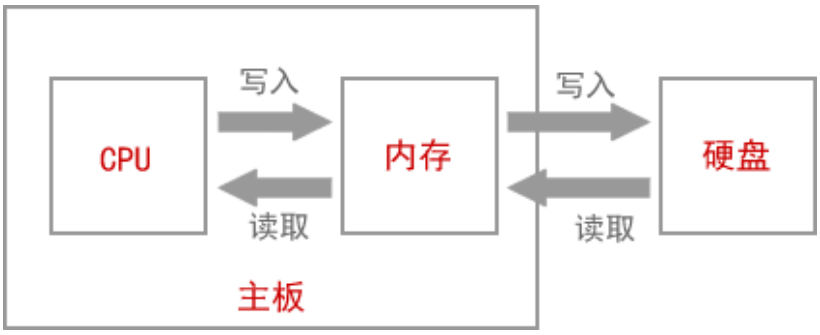


图 1：CPU、内存、硬盘和主板的关系

1.13 ASCII 编码，将英文存储到计算机

前面我们已经讲到，计算机是以二进制的形式来存储数据的，它只认识 0 和 1 两个数字，我们在屏幕上看到的文字，在存储之前都被转换成了二进制（0 和 1 序列），在显示时也要根据二进制找到对应的字符。

可想而知，特定的文字必然对应着固定的二进制，否则在转换时将发生混乱。那么，怎样将文字与二进制对应起来呢？这就需要有一套规范，计算机公司和软件开发者都必须遵守，这样的一套规范就称为字符集（Character Set）或者字符编码（Character Encoding）。

> 严格来说，字符集和字符编码不是一个概念，字符集定义了文字和二进制的对应关系，为字符分配了唯一的编号，而字符编码规定了如何将文字的编号存储到计算机中。我们暂时先不讨论这些细节，姑且认为它们是一个概念，本节中我也混用了这两个概念，未做区分。

字符集为每个字符分配一个唯一的编号，类似于学生的学号，通过编号就能够找到对应的字符。

可以将字符集理解成一个很大的表格，它列出了所有字符和二进制的对应关系，计算机显示文字或者存储文字，就是一个查表的过程。

在计算机逐步发展的过程中，先后出现了几十种甚至上百种字符集，有些还在使用，有些已经淹没在了历史的长河中，本节我们要讲解的是一种专门针对英文的字符集——ASCII 编码。

拉丁字母（开胃小菜）

在正式介绍 ASCII 编码之前，我们先来说说什么是拉丁字母。估计也有不少读者和我一样，对于拉丁字母、英文字母和汉语拼音中的字母的关系不是很清楚。

拉丁字母也叫罗马字母，它源自希腊字母，是当今世界上使用最广的字母系统。基本的拉丁字母就是我们经常见到的 ABCD 等 26 个英文字母。

> 拉丁字母、阿拉伯字母、斯拉夫字母（西里尔字母）被称为世界三大字母体系。

拉丁字母原先是欧洲人使用的，后来由于欧洲殖民主义，导致这套字母体系在全球范围内开始流行，美洲、非洲、澳洲、亚洲都没有逃过西方文化的影响。中国也是，我们现在使用的拼音其实就是拉丁字母，是不折不扣的舶来品。

后来，很多国家对 26 个基本的拉丁字母进行了扩展，以适应本地的语言文化。最常见的扩展方式就是加上变音符

号，例如汉语拼音中的ü，就是在 u 的基础上加上两个小点演化而来；再如，áà就是在 a 的上面标上音调。

总起来说：

- 基本拉丁字母就是 26 个英文字母；
- 扩展拉丁字母就是在基本的 26 个英文字母的基础上添加变音符号、横线、斜线等演化而来，每个国家都不一样。

ASCII 编码

计算机是美国人发明的，他们首先要考虑的问题是，如何将二进制和英文字母（也就是拉丁文）对应起来。

当时，各个厂家或者公司都有自己的做法，编码规则并不统一，这给不同计算机之间的数据交换带来不小的麻烦。但是相对来说，能够得到普遍认可的有 IBM 发明的 EBCDIC 和此处要谈的 ASCII。

我们先说 ASCII。ASCII 是“American Standard Code for Information Interchange”的缩写，翻译过来是“美国信息交换标准代码”。看这个名字就知道，这套编码是美国人给自己设计的，他们并没有考虑欧洲那些扩展的拉丁字母，也没有考虑韩语和日语，我大中华几万个汉字更是不可能被重视。

但这也无可厚非，美国人自己发明的计算机，当然要先解决自己的问题

ASCII 的标准版本于 1967 年第一次发布，最后一次更新则是在 1986 年，迄今为止共收录了 128 个字符，包含了基本的拉丁字母（英文字母）、阿拉伯数字（也就是 1234567890）、标点符号（,.!等）、特殊符号（@#$%^&等）以及一些具有控制功能的字符（往往不会显示出来）。

在 ASCII 编码中，大写字母、小写字母和阿拉伯数字都是连续分布的（见下表），这给程序设计带来了很大的方便。例如要判断一个字符是否是大写字母，就可以判断该字符的 ASCII 编码值是否在 65~90 的范围内。

EBCDIC 编码正好相反，它的英文字母不是连续排列的，中间出现了多次断续，给编程带来了一些困难。现在连 IBM 自己也不使用 EBCDIC 了，转而使用更加优秀的 ASCII。

ASCII 编码已经成了计算机的通用标准，没有人再使用 EBCDIC 编码了，它已经消失在历史的长河中了。

ASCII 编码一览表

标准 ASCII 编码共收录了 128 个字符，其中包含了 33 个控制字符（具有某些特殊功能但是无法显示的字符）和 95 个可显示字符。

ASCII 编码一览表（淡黄色背景为控制字符，白色背景为可显示字符）

二进制	十进制	十六进制	字符/缩写	解释
00000000	0	00	NUL (NULL)	空字符
00000001	1	01	SOH (Start Of Headling)	标题开始
00000010	2	02	STX (Start Of Text)	正文开始

00000011	3	03	ETX (End Of Text)	正文结束
00000100	4	04	EOT (End Of Transmission)	传输结束
00000101	5	05	ENQ (Enquiry)	请求
00000110	6	06	ACK (Acknowledge)	回应/响应/收到通知
00000111	7	07	BEL (Bell)	响铃
00001000	8	08	BS (Backspace)	退格
00001001	9	09	HT (Horizontal Tab)	水平制表符
00001010	10	0A	LF/NL(Line Feed/New Line)	换行键
00001011	11	0B	VT (Vertical Tab)	垂直制表符
00001100	12	0C	FF/NP (Form Feed/New Page)	换页键
00001101	13	0D	CR (Carriage Return)	回车键
00001110	14	0E	SO (Shift Out)	不用切换
00001111	15	0F	SI (Shift In)	启用切换
00010000	16	10	DLE (Data Link Escape)	数据链路转义
00010001	17	11	DC1/XON (Device Control 1/Transmission On)	设备控制 1/传输开始
00010010	18	12	DC2 (Device Control 2)	设备控制 2
00010011	19	13	DC3/XOFF (Device Control 3/Transmission Off)	设备控制 3/传输中断
00010100	20	14	DC4 (Device Control 4)	设备控制 4
00010101	21	15	NAK (Negative Acknowledge)	无响应/非正常响应/拒绝接收
00010110	22	16	SYN (Synchronous Idle)	同步空闲
00010111	23	17	ETB (End of Transmission Block)	传输块结束/块传输终止
00011000	24	18	CAN (Cancel)	取消
00011001	25	19	EM (End of Medium)	已到介质末端/介质存储已满/介质中断
00011010	26	1A	SUB (Substitute)	替补/替换
00011011	27	1B	ESC (Escape)	逃离/取消
00011100	28	1C	FS (File Separator)	文件分割符
00011101	29	1D	GS (Group Separator)	组分隔符/分组符
00011110	30	1E	RS (Record Separator)	记录分离符

00011111	31	1F	US (Unit Separator)	单元分隔符
00100000	32	20	(Space)	空格
00100001	33	21	!	
00100010	34	22	"	
00100011	35	23	#	
00100100	36	24	$	
00100101	37	25	%	
00100110	38	26	&	
00100111	39	27	'	
00101000	40	28	(	
00101001	41	29	)	
00101010	42	2A	*	
00101011	43	2B	+	
00101100	44	2C	,	
00101101	45	2D	-	
00101110	46	2E	.	
00101111	47	2F	/	
00110000	48	30	0	
00110001	49	31	1	
00110010	50	32	2	
00110011	51	33	3	
00110100	52	34	4	
00110101	53	35	5	
00110110	54	36	6	
00110111	55	37	7	
00111000	56	38	8	
00111001	57	39	9	
00111010	58	3A	:	
00111011	59	3B	;	

00111100	60	3C	<	
00111101	61	3D	=	
00111110	62	3E	>	
00111111	63	3F	?	
01000000	64	40	@	
01000001	65	41	A	
01000010	66	42	B	
01000011	67	43	C	
01000100	68	44	D	
01000101	69	45	E	
01000110	70	46	F	
01000111	71	47	G	
01001000	72	48	H	
01001001	73	49	I	
01001010	74	4A	J	
01001011	75	4B	K	
01001100	76	4C	L	
01001101	77	4D	M	
01001110	78	4E	N	
01001111	79	4F	O	
01010000	80	50	P	
01010001	81	51	Q	
01010010	82	52	R	
01010011	83	53	S	
01010100	84	54	T	
01010101	85	55	U	
01010110	86	56	V	
01010111	87	57	W	
01011000	88	58	X	

01011001	89	59	Y	
01011010	90	5A	Z	
01011011	91	5B	[	
01011100	92	5C	\	
01011101	93	5D	]	
01011110	94	5E	^	
01011111	95	5F	_	
01100000	96	60	`	
01100001	97	61	a	
01100010	98	62	b	
01100011	99	63	c	
01100100	100	64	d	
01100101	101	65	e	
01100110	102	66	f	
01100111	103	67	g	
01101000	104	68	h	
01101001	105	69	i	
01101010	106	6A	j	
01101011	107	6B	k	
01101100	108	6C	l	
01101101	109	6D	m	
01101110	110	6E	n	
01101111	111	6F	o	
01110000	112	70	p	
01110001	113	71	q	
01110010	114	72	r	
01110011	115	73	s	
01110100	116	74	t	
01110101	117	75	u	

01110110	118	76	v	
01110111	119	77	w	
01111000	120	78	x	
01111001	121	79	y	
01111010	122	7A	z	
01111011	123	7B	{	
01111100	124	7C	|	
01111101	125	7D	}	
01111110	126	7E	~	
01111111	127	7F	DEL (Delete)	删除

上表列出的是标准的 ASCII 编码，它共收录了 128 个字符，用一个字节中较低的 7 个比特位（Bit）足以表示（27 = 128），所以还会空闲下一个比特位，它就被浪费了。

如果您还想了解每个控制字符的含义，请转到：完整的 ASCII 码对照表以及各个字符的解释

ASCII 编码和 C 语言

稍微有点 C 语言基本功的读者可能认为 C 语言使用的就是 ASCII 编码，字符在存储时会转换成对应的 ASCII 码值，在读取时也是根据 ASCII 码找到对应的字符。这句话是错误的，严格来说，你可能被大学老师和 C 语言教材给误导了。

C 语言有时候使用 ASCII 编码，有时候却不是，而是使用后面两节中即将讲到的 GBK 编码和 Unicode 字符集，我们将在《C 语言到底使用什么编码？谁说 C 语言使用 ASCII 码，真是荒谬！》一节中展开讲解。

1.14 GB2312 编码和 GBK 编码，将中文存储到计算机

1.15 Unicode 字符集，将全世界的文字存储到计算机

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

1.16 程序员的薪水和发展方向大全

1.17 不要这样学习 C 语言，这是一个坑！

1.18 明白了这点才能学好编程，否则参加什么培训班都没用

第 02 章 C 语言初探

本章主要讲解 C 语言编程环境的搭建，让大家能够编写并运行 C 语言代码，其中，编译器是重点讲解内容。

本章还对一段简单的 C 语言代码进行了分析，让大家明白了 C 语言程序的基本结构。

本章目录：

1.第一个 C 语言程序
2.选择正确的输入法，严格区分中英文
3.什么是源文件？
4.什么是编译和链接（通俗易懂，深入本质）
5.主流 C 语言编译器有哪些？
6.什么是 IDE（集成开发环境）？
7.什么是工程/项目（Project）？
8.哪款 C 语言编译器（IDE）适合初学者？
9.如何在手机上编写 C 语言代码？
10.C 语言的三套标准：C89、C99 和 C11
11.C 语言为什么有那么多编译器？

12.程序安装是怎么回事？
13.制作安装包，让用户安装程序
14.C 语言程序的错误和警告
15.分析第一个 C 语言程序
16.C 语言代码中的空白符
17.彩色版的 C 语言，让文字更漂亮
18.一个真正带界面的 C 语言程序

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

2.1 第一个 C 语言程序

我们有两种方式从计算机获得信息：一是看屏幕上的文字、图片、视频等，二是听从喇叭发出来的声音。让喇叭发出声音目前还比较麻烦，我们先来看看如何在屏幕上显示一些文字吧。

在屏幕上显示文字非常简单，只需要一个语句，例如，下面的代码会让屏幕显示出“C 语言中文网”：

```
puts("C 语言中文网");
```

这里有一个生疏的词汇 puts，用来让计算机在屏幕上显示文字。

更加专业的称呼：
- "在屏幕上显示文字"叫做输出（Output）；
- 每个文字都是一个字符（Character）；
- 多个字符组合起来，就是一个字符序列，叫做字符串（String）。

puts 是 output string 的缩写，意思是”输出字符串“。

在 C 语言中，字符串需要用双引号" "包围起来，C 语言中文网什么也不是，计算机不认识它，"C 语言中文网"才是字符串。

puts 在输出字符串的时候，需要将字符串放在()内。

在汉语和英语中，分别使用。和.表示一句话的结束，而在 C 语言中，使用;表示一个语句的结束。puts("C 语言中文网")表达了完整的意思，是一个完整的语句，需要在最后加上;，表示当前语句结束了。

总结起来，上面的语句可以分为三个部分：
- puts()命令计算机输出字符串；
- "C 语言中文网"是要输出的内容；
- ;表示语句结束。

C 语言程序的整体框架

puts 可以在显示器上输出内容，但是仅有 puts 是不够的，程序不能运行，还需要添加其他代码，构成一个完整的框架。完整的程序如下：

```
#include <stdio.h>
int main()
{
    puts("C 语言中文网");
    return 0;
}
```

第 1~3 行、第 5~6 行是固定的，所有 C 语言源代码都必须有这几行。你暂时不需要理解它们是什么意思，反正有这个就是了，以后会慢慢讲解。

但是请记住，今后我们写的所有类似 puts 这样的语句，都必须放在{}之间才有效。

上面的代码，看起来枯燥无趣，不好区分各个语句，我们不妨来给它们加上颜色和行号，如下所示：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      puts("C语言中文网");
5.      return 0;
6.  }
```

> 颜色和行号是笔者自己加上去的，主要是为了让大家阅读方便，明显地区分各个语句，C 语言本身没有对这些作要求，你可以随意设置各个字符的颜色，也可以没有颜色。

这就是我们的第一个 C 语言程序，它非常简单，带领我们进入了 C 语言的大门。

2.2 选择正确的输入法，严格区分中英文

计算机起源于美国，C 语言、C++、Java、JavaScript 等很多流行的编程语言都是美国人发明的，所以在编写代码的时候必须使用**英文半角输入法**，尤其是标点符号，初学者一定要引起注意。

例如，上节我们使用 puts 语句在显示器上输出内容：

```
puts("C 语言中文网");
```

这里的括号、双引号、分号都必须是英文符号，而且是半角的。下图演示了如何将搜狗输入法切换到英文半角状态：

图 1：搜狗输入法

一些相似的中英文标点符号：

- 中文分号；和英文分号;；
- 中文逗号，和英文逗号,；
- 中文冒号：和英文冒号:；
- 中文括号（）和英文括号()；

- 中文问号？和英文问号?；
- 中文单引号’　‘和英文单引号' '；
- 中文双引号 “ ” 和英文双引号" "。

初学者请务必注意标点符号的问题，它们在视觉上的差别很小，一旦将英文符号写成中文符号就会导致错误，而且往往不容易发现。我在给 VIP 会员进行一对一答疑的过程中，经常会遇到类似的错误，有些同学甚至会在这里跌倒好几次。

全角和半角输入法的区别

全角和半角的区别主要在于除汉字以外的其它字符，比如标点符号、英文字母、阿拉伯数字等，全角字符和半角字符所占用的位置的大小不同。

在计算机屏幕上，一个汉字要占两个英文字符的位置，人们把一个英文字符所占的位置称为“半角”，相对地把一个汉字所占的位置称为“全角”。

标点符号、英文字母、阿拉伯数字等这些字符不同于汉字，在半角状态它们被作为英文字符处理，而在全角状态作为中文字符处理，请看下面的例子。

半角输入：

```
C 语言中文网！Hello C,I like!
```

全角输入：

```
Ｃ语言中文网！Ｈｅｌｌｏ　Ｃ，Ｉ　ｌｉｋｅ！
```

另外最重要的一点是：“相同”字符在全角和半角状态下对应的编码值（例如 Unicode 编码、GBK 编码等）不一样，所以它们是不同的字符。

图：搜狗输入法半角和全角

我们知道，在编程时要使用英文半角输入法。为了加强练习，出个选择题，请大家判断下面哪一种描述是正确的：
A) 编程的时候不用在意中英文符号的区别。
B) 在源代码的任何地方都不能出现中文汉字、字符等。
C) 感叹号没有中文和英文的区别。
D) 编程时，使用的英文引号，也有左引号和右引号的区别。
E) 中文和英文模式下的制表符（键盘 tab 键）输入效果一致。

答案：E 选项正确。

2.3 什么是源文件？

在开发软件的过程中，我们需要将编写好的代码（Code）保存到一个文件中，这样代码才不会丢失，才能够被编译器找到，才能最终变成可执行文件。这种用来保存代码的文件就叫做源文件（Source File）。

> 我们将在《编译和链接》一节中讲解编译器的概念。

每种编程语言的源文件都有特定的后缀，以方便被编译器识别，被程序员理解。源文件后缀大都根据编程语言本身的名字来命名，例如：

- C 语言源文件的后缀是.c；
- C++语言（C Plus Plus）源文件的后缀是.cpp；
- Java 源文件的后缀是.java；
- Python 源文件的后缀是.py；
- JavaScript 源文件后置是.js。

源文件其实就是纯文本文件，它的内部并没有特殊格式，能证明这一结论的典型例子是：在 Windows 下用记事本程序新建一个文本文档，并命名为 demo.txt，输入一段 C 语言代码并保存，然后将该文件强制重命名为 demo.c（后缀从.txt 变成了.c），发现编译器依然能够正确识别其中的 C 语言代码，并顺利生成可执行文件。

源文件的后缀仅仅是为了表明该文件中保存的是某种语言的代码（例如.c 文件中保存的是 C 语言代码），这样程序员更加容易区分，编译器也更加容易识别，它并不会导致该文件的内部格式发生改变。

C++ 是站在 C 语言的肩膀上发展期来的，是在 C 语言的基础上进行的扩展，C++ 包含了 C 语言的全部内容（请猛击《C 语言和 C++到底有什么关系》一文了解更多），将 C 语言代码放在.cpp 文件中不会有错，很多初学者都是这么做的，很多大学老师也是这么教的。但是，我还是强烈建议将 C 语言代码放在.c 文件中，这样能够更加严格地遵循 C 语言的语法，也能够更加清晰地了解 C 语言和 C++的区别。

2.4 什么是编译和链接（通俗易懂，深入本质）

我们平时所说的程序，是指双击后就可以直接运行的程序，这样的程序被称为可执行程序（Executable Program）。在 Windows 下，可执行程序的后缀有.exe 和.com（其中.exe 比较常见）；在类 UNIX 系统（Linux、Mac OS 等）下，可执行程序没有特定的后缀，系统根据文件的头部信息来判断是否是可执行程序。

可执行程序的内部是一系列计算机指令和数据的集合，它们都是二进制形式的，CPU 可以直接识别，毫无障碍；但是对于程序员，它们非常晦涩，难以记忆和使用。

例如，在屏幕上输出“VIP 会员”，C 语言的写法为：

```
puts("VIP 会员");
```

二进制的写法为：

```
指令部分                                  数据部分
--------  ----------------------------------------------------------
10110000  10101011  00110001  00011100  00110011  00111110  01010011  11110000
--------  --------  --------  --------  ----------------    ----------------
输出指令      V         I         P           会                  员
```

你感受一下，直接使用二进制是不是想撞墙，是不是受到一吨重的伤害？

> 在计算机发展的初期，程序员就是使用这样的二进制指令来编写程序的，那个拓荒的年代还没有编程语言。

直接使用二进制指令编程对程序员来说简直是噩梦，尤其是当程序比较大的时候，不但编写麻烦，需要频繁查询指令手册，而且除错会异常苦恼，要直接面对一堆二进制数据，让人眼花缭乱。另外，用二进制指令编程步骤繁琐，要考虑各种边界情况和底层问题，开发效率十分低下。

这就倒逼程序员开发出了编程语言，提高自己的生产力，例如汇编、C 语言、C++、Java、Python、Go 语言等，都是在逐步提高开发效率。至此，编程终于不再是只有极客能做的事情了，不了解计算机的读者经过一定的训练也可以编写出有模有样的程序。

编译（Compile）

C 语言代码由固定的词汇按照固定的格式组织起来，简单直观，程序员容易识别和理解，但是对于 CPU，C 语言代码就是天书，根本不认识，CPU 只认识几百个二进制形式的指令。这就需要一个工具，将 C 语言代码转换成 CPU 能够识别的二进制指令，也就是将代码加工成 .exe 程序的格式 这个工具是一个特殊的软件，叫做**编译器（Compiler）**。

编译器能够识别代码中的词汇、句子以及各种特定的格式，并将他们转换成计算机能够识别的二进制形式，这个过程称为**编译（Compile）**。

编译也可以理解为“翻译”，类似于将中文翻译成英文、将英文翻译成象形文字，它是一个复杂的过程，大致包括词法分析、语法分析、语义分析、性能优化、生成可执行文件五个步骤，期间涉及到复杂的算法和硬件架构。对于学计算机或者软件的大学生，“编译原理”是一门专业课程，有兴趣的读者请自行阅读《编译原理》一书，这里我们不再展开讲解。

> 注意：不了解编译原理并不影响我们学习 C 语言，我也不建议初学者去钻研编译原理，贪多嚼不烂，不要把自己绕进去。

C 语言的编译器有很多种，不同的平台下有不同的编译器，例如：

- Windows 下常用的是微软开发的 Visual C++，它被集成在 Visual Studio 中，一般不单独使用；
- Linux 下常用的是 GUN 组织开发的 GCC，很多 Linux 发行版都自带 GCC；
- Mac 下常用的是 LLVM/Clang，它被集成在 Xcode 中（Xcode 以前集成的是 GCC，后来由于 GCC 的不配合才改为 LLVM/Clang，LLVM/Clang 的性能比 GCC 更加强大）。

你的代码语法正确与否，编译器说了才算，我们学习 C 语言，从某种意义上说就是学习如何使用编译器。

编译器可以 100% 保证你的代码从语法上讲是正确的，因为哪怕有一点小小的错误，编译也不能通过，编译器会告诉你哪里错了，便于你的更改。

链接（Link）

C 语言代码经过编译以后，并没有生成最终的可执行文件（.exe 文件），而是生成了一种叫做目标文件（Object File）的中间文件（或者说临时文件）。目标文件也是二进制形式的，它和可执行文件的格式是一样的。对于 Visual C++，目标文件的后缀是.obj；对于 GCC，目标文件的后缀是.o。

目标文件经过链接（Link）以后才能变成可执行文件。既然目标文件和可执行文件的格式是一样的，为什么还要再链接一次呢，直接作为可执行文件不行吗？

不行的！因为编译只是将我们自己写的代码变成了二进制形式，它还需要和系统组件（比如标准库、动态链接库等）结合起来，这些组件都是程序运行所必须的。

链接（Link）其实就是一个“打包”的过程，它将所有二进制形式的目标文件和系统组件组合成一个可执行文件。完成链接的过程也需要一个特殊的软件，叫做**链接器（Linker）**。

随着我们学习的深入，我们编写的代码越来越多，最终需要将它们分散到多个源文件中，编译器每次只能编译一个源文件，生成一个目标文件，这个时候，链接器除了将目标文件和系统组件组合起来，还需要将编译器生成的多个目标文件组合起来。

再次强调，编译是针对一个源文件的，有多少个源文件就需要编译多少次，就会生成多少个目标文件。

总结

不管我们编写的代码有多么简单，都必须经过「编译 --> 链接」的过程才能生成可执行文件：

- 编译就是将我们编写的源代码“翻译”成计算机可以识别的二进制格式，它们以目标文件的形式存在；
- 链接就是一个“打包”的过程，它将所有的目标文件以及系统组件组合成一个可执行文件。

如果不是特别强调，一般情况下我们所说的“编译器”实际上也包括了链接器，比如，你使用了哪种编译器？去哪里下载 C 语言编译器？我的编译器为什么报错了呢？

2.5 主流 C 语言编译器有哪些？

在上节《C 语言编译和链接》中我们已经讲解了 C 语言编译器的概念，由于 C 语言的历史比较久，而且早期没有规范，整个计算机产业也都处于拓荒的年代，所以就涌现了很多款 C 语言编译器，它们各有特点，适用于不同的平台，本节就来给大家科普一下。

我们分两部分介绍 C 语言的编译器，分别是桌面操作系统和嵌入式操作系统。

桌面操作系统

对于当前主流桌面操作系统而言，可使用 Visual C++、GCC 以及 LLVM Clang 这三大编译器。

Visual C++（简称 MSVC）是由微软开发的，只能用于 Windows 操作系统；GCC 和 LLVM Clang 除了可用于 Windows 操作系统之外，主要用于 Unix/Linux 操作系统。

像现在很多版本的 Linux 都默认使用 GCC 作为 C 语言编译器，而像 FreeBSD、macOS 等系统默认使用 LLVM Clang 编译器。由于当前 LLVM 项目主要在 Apple 的主推下发展的，所以在 macOS 中，Clang 编译器又被称为 Apple LLVM 编译器。

MSVC 编译器主要用于 Windows 操作系统平台下的应用程序开发，它不开源。用户可以使用 Visual Studio Community 版本来免费使用它，但是如果要把通过 Visual Studio Community 工具生成出来的应用进行商用，那么就得好好阅读一下微软的许可证和说明书了。

而使用 GCC 与 Clang 编译器构建出来的应用一般没有任何限制，程序员可以将应用程序随意发布和进行商用。

MSVC 编译器对 C99 标准的支持就十分有限，加之它压根不支持任何 C11 标准，所以本教程中设计 C11 的代码例子不会针对 MSVC 进行描述。所幸的是，Visual Studio Community 2017 加入了对 Clang 编译器的支持，官方称之为——Clang with Microsoft CodeGen，当前版本基于的是 Clang 3.8。

也就是说，应用于 Visual Studio 集成开发环境中的 Clang 编译器前端可支持 Clang 编译器的所有语法特性，而后端生成的代码则与 MSVC 效果一样，包括像 long 整数类型在 64 位编译模式下长度仍然为 4 个字节，所以各位使用的时候也需要注意。

为了方便描述，本教程后面涉及 Visual Studio 集成开发环境下的 Clang 编译器简称为 VS-Clang 编译器。

嵌入式系统

而在嵌入式系统方面，可用的 C 语言编译器就非常丰富了，比如：

- 用于 Keil 公司 51 系列单片机的 Keil C51 编译器；
- 当前大红大紫的 Arduino 板搭载的开发套件，可用针对 AVR 微控制器的 AVR GCC 编译器；
- ARM 自己出的 ADS（ARM Development Suite）、RVDS（RealView Development Suite）和当前最新的 DS-5 Studio；
- DSP 设计商 TI（Texas Instruments）的 CCS（Code Composer Studio）；
- DSP 设计商 ADI（Analog Devices，Inc.）的 Visual DSP++ 编译器，等等。

通常，用于嵌入式系统开发的编译工具链都没有免费版本，而且一般需要通过国内代理进行购买。所以，这对于个人开发者或者嵌入式系统爱好者而言是一道不低的门槛。

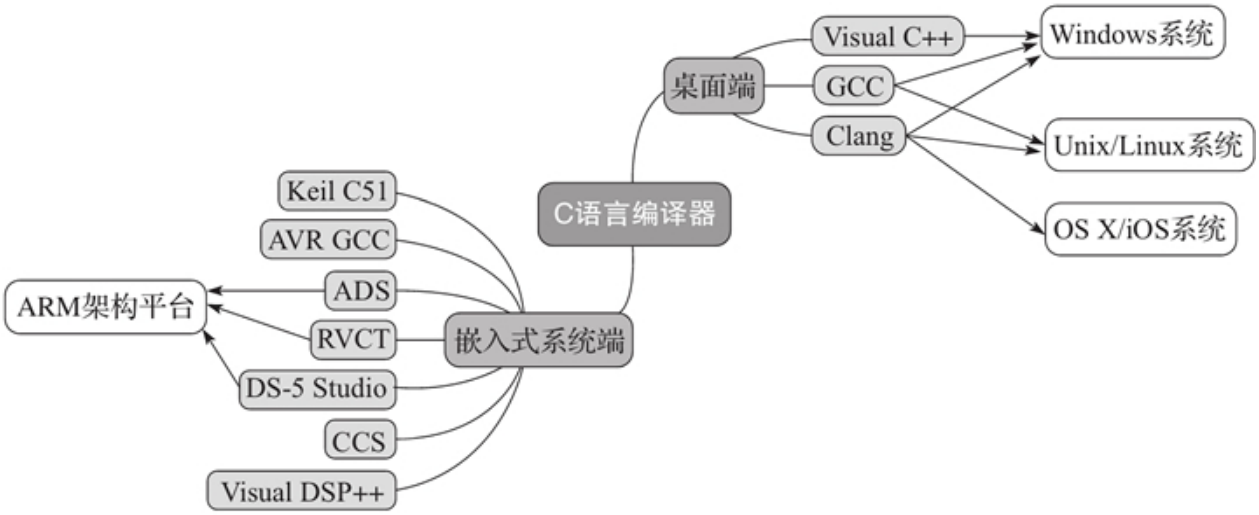
不过 Arduino 的开发套件是可免费下载使用的，并且用它做开发板连接调试也十分简单。Arduino 所采用的 C 编译器是基于 GCC 的。

还有像树莓派（Raspberry Pi）这种迷你电脑可以直接使用 GCC 和 Clang 编译器。此外，还有像 nVidia 公司推出的 Jetson TK 系列开发板也可直接使用 GCC 和 Clang 编译器。树莓派与 Jetson TK 都默认安装了 Linux 操作系统。

在嵌入式领域，一般比较低端的单片机，比如 8 位的 MCU 所对应的 C 编译器可能只支持 C90 标准，有些甚至连 C90 标准的很多特性都不支持。因为它们一方面内存小，ROM 的容量也小；另一方面，本身处理器机能就十分有限，有些甚至无法支持函数指针，因为处理器本身不包含通过寄存器做间接过程调用的指令。

而像 32 位处理器或 DSP，一般都至少能支持 C99 标准，它们本身的性能也十分强大。而像 ARM 出的 RVDS 编译器甚至可用 GNU 语法扩展。

下图展示了上述 C 语言编译器的分类。



2.6 什么是 IDE（集成开发环境）？

实际开发中，除了编译器是必须的工具，我们往往还需要很多其他辅助软件，例如：

- 编辑器：用来编写代码，并且给代码着色，以方便阅读；
- 代码提示器：输入部分代码，即可提示全部代码，加速代码的编写过程；
- 调试器：观察程序的每一个运行步骤，发现程序的逻辑错误；
- 项目管理工具：对程序涉及到的所有资源进行管理，包括源文件、图片、视频、第三方库等；
- 漂亮的界面：各种按钮、面板、菜单、窗口等控件整齐排布，操作更方便。

这些工具通常被打包在一起，统一发布和安装，例如 Visual Studio、Dev C++、Xcode、Visual C++ 6.0、C-Free、Code::Blocks 等，它们统称为集成开发环境（IDE，Integrated Development Environment）。

集成开发环境就是一系列开发工具的组合套装。这就好比台式机，一个台式机的核心部件是主机，有了主机就能独立工作了，但是我们在购买台式机时，往往还要附带上显示器、键盘、鼠标、U 盘、摄像头等外围设备，因为只有主机太不方便了，必须有外设才能玩的爽。

集成开发环境也是这个道理，只有编译器不方便，所以还要增加其他的辅助工具。在实际开发中，我一般也是使用集成开发环境，而不是单独地使用编译器。

通俗的称呼

有时候为了称呼方便，或者初学者没有严格区分概念，也会将 C 语言集成开发环境称作“C 语言编译器”或者“C 语言编程软件”。这里大家不要认为是一种错误，就把它当做“乡间俗语”吧。

2.7 什么是工程/项目（Project）？

一个真正的程序（也可以说软件）往往包含多项功能，每一项功能都需要几十行甚至几千行、几万行的代码来实现，如果我们将这些代码都放到一个源文件中，那将会让人崩溃，不但源文件打开速度极慢，代码的编写和维护也将变得非常困难。

在实际开发中，程序员都是将这些代码分门别类地放到多个源文件中。除了这些成千上万行的代码，一个程序往往

还要包含图片、视频、音频、控件、库（也可以说框架）等其它资源，它们也都是一个一个地文件。

为了有效地管理这些种类繁杂、数目众多的文件，我们有理由把它们都放到一个目录（文件夹）下，并且这个目录下只存放与当前程序有关的资源。实际上 IDE 也是这么做的，它会为每一个程序都创建一个专门的目录，将用到的所有文件都集中到这个目录下，并对它们进行便捷的管理，比如重命名、删除文件、编辑文件等。

这个为当前程序配备的专用文件夹，在 IDE 中也有一个专门的称呼，叫做“Project”，翻译过来就是“工程”或者“项目”。在 Visual C++ 6.0 下，这叫做一个“工程”，而在 Visual Studio 下，这又叫做一个“项目”，它们只是单词“Project”的不同翻译而已，实际上是一个概念。

工程类型/项目类型

“程序”是一个比较宽泛的称呼，它可以细分为很多种类，例如：

- 有的程序不带界面，完全是“黑屏”的，只能输入一些字符或者命令，称为控制台程序（Console Application），例如 Windows 下的 cmd.exe，Linux 或 Mac OS 下的终端（Terminal）。
- 有的程序带界面，看起来很漂亮，能够使用鼠标点击，称为 GUI 程序（Graphical User Interface Program），例如 QQ、迅雷、Chrome 等。
- 有的程序不单独出现，而是作为其它程序的一个组成部分，普通用户很难接触到它们，例如静态库、动态库等。

不同的程序对应不同的工程类型（项目类型），使用 IDE 时必须选择正确的工程类型才能创建出我们想要的程序。换句话说，IDE 包含了多种工程类型，不同的工程类型会创建出不同的程序。

不同的工程类型本质上是对 IDE 中各个参数的不同设置；我们也可以创建一个空白的工程类型，然后自己去设置各种参数（不过一般不这样做）。

控制台程序对应的工程类型为“Win32 控制台程序（Win32 Console Application）”，GUI 程序对应的工程类型为“Win32 程序（Win32 Application）”。

控制台程序是 DOS 时代的产物了，它没有复杂的功能，没有漂亮的界面，只能看到一些文字，虽然枯燥无趣，也不实用，但是它非常简单，不受界面的干扰，所以适合入门，我强烈建议初学者从控制台程序学起。等大家对编程掌握的比较熟练了，能编写上百行的代码了，再慢慢过渡到 GUI 程序。

2.8 哪款 C 语言编译器（IDE）适合初学者？

C 语言的集成开发环境有很多种，尤其是 Windows 下，多如牛毛，初学者往往不知道该如何选择，本节我们就针对 Windows、Linux 和 Mac OS 三大平台进行讲解。

Windows 下如何选择 IDE？

Windows 下的 IDE 多如牛毛，常见的有以下几种。

1) Visual Studio

Windows 下首先推荐大家使用微软开发的 Visual Studio（简称 VS），它是 Windows 下的标准 IDE，实际开发中大家也都在使用。

为了适应最新的 Windows 操作系统，微软每隔一段时间（一般是一两年）就会对 VS 进行升级。VS 的不同版本以发布年份命名，例如 VS2010 是微软于 2010 年发布的，VS2017 是微软于 2017 年发布的。

不过 VS 有点庞大，安装包有 2~3G，下载不方便，而且会安装很多暂时用不到的工具，安装时间在半个小时左右。

对于初学者，我推荐使用 VS2015。最好不用使用 VS2017，有点坑初学者。

2) Dev C++

如果你讨厌 VS 的复杂性，那么可以使用 Dev C++。Dev C++ 是一款免费开源的 C/C++ IDE，内嵌 GCC 编译器（Linux GCC 编译器的 Windows 移植版），是 NOI、NOIP 等比赛的指定工具。Dev C++ 的优点是体积小（只有几十兆）、安装卸载方便、学习成本低，缺点是调试功能弱。

> NOI 是 National Olympiad in Informatics 的缩写，译为“全国青少年信息学奥林匹克竞赛”；NOIP 是 National Olympiad in informatics in Provinces 的缩写，译为“全国青少年信息学奥林匹克联赛”。NOI、NOIP 都是奥林匹克竞赛的一种，参加者多为高中生，获奖者将被保送到名牌大学或者得到高考加分资格。

3) Visual C++ 6.0

Visual C++ 6.0（简称 VC 6.0）是微软开发的一款经典的 IDE，很多高校都以 VC 6.0 为教学工具来讲解 C 和 C++。但 VC 6.0 是 1998 年的产品，很古老了，在 Win7、Win8、Win10 下会有各种各样的兼容性问题，甚至根本不能运行，所以不推荐使用。

VC 6.0 早就该扔进垃圾桶了，可是依然有很多大学把它作为教学工具，并且选用的教材也以 VC 6.0 为基础来讲解 C 语言和 C++，可见教学体制的极端落后，课程体系的更新远远跟不上技术的进步。

4) Code::Blocks

Code::Blocks 是一款开源、跨平台、免费的 C/C++ IDE，它和 Dev C++ 非常类似，小巧灵活，易于安装和卸载，不过它的界面要比 Dev C++ 复杂一些，不如 Dev C++ 来得清爽。

5) Turbo C

Turbo C 是一款古老的、DOS 年代的 C 语言开发工具，程序员只能使用键盘来操作 Turbo C，不能使用鼠标，所以非常不方便。但是 Turbo C 集成了一套图形库，可以在控制台程序中画图，看起来非常炫酷，所以至今仍然有人在使用。

6) C-Free

C-Free 是一款国产的 Windows 下的 C/C++ IDE，最新版本是 5.0，整个软件才 14M，非常轻巧，安装也简单，界面也比 Dev C++ 漂亮。C-Free 的缺点也是调试功能弱。可惜的是，C-Free 已经多年不更新了，组件都老了，只能在 XP、Win7 下运行，在 Win8、Win10 下可能会存在兼容性问题。

下面我们给出了各种 IDE（含不同版本）的下载地址、安装方法以及使用教程，并以红色字体附带了建议。

- **VS2015 [力荐]**
  - VS2015 下载地址和安装教程（图解）
  - 使用 VS2015 编写 C 语言程序
- **VS2017**

- VS2017 下载地址和安装教程（图解）
- 使用 VS2017 编写 C 语言程序

➢ **VS2010 [荐]**
- VS2010 下载地址和安装教程（图解）
- 使用 VS2010 编写 C 语言程序

➢ **Dev C++ [荐]**
- Dev C++下载地址和安装教程（图解）
- 使用 Dev C++编写 C 语言程序

➢ **VC6.0 [不建议]**
- VC6.0（VC++6.0）下载地址和安装教程（图解）
- 使用 VC6.0（VC++6.0）编写 C 语言程序

➢ **Code::Blocks**
- Code::Blocks 下载地址和安装教程（图解）
- Code::Blocks 汉化教程（附带汉化包）
- 使用 Code::Blocks 编写 C 语言程序

➢ **Turbo C [不建议]**
- Turbo C 2.0 下载地址和安装教程（图解）
- 使用 Turbo C 2.0 编写 C 语言程序

➢ **C-Free [不建议]**
- C-Free 5.0 下载地址和激活教程（图解）
- 使用 C-Free 编写 C 语言程序

为什么不建议初学者使用最新的 VS2017？

VS2017 对初学者并不友好，或者说有点坑初学者，例如：

➢ 新创建的 C 语言工程里面默认会附带多个源文件，初学者往往不知道如何使用它们，还得一个一个删除，非常麻烦。

➢ 按下 Ctrl+F5 组合键运行程序，程序不能自动暂停，每次都得在最后添加暂停代码，这是最致命的。

VS2015 和 VS 2010 就没有上述问题，所以推荐使用；又考虑到 VS2010 可能不兼容最新的 Win10，所以推荐使用 VS2015。

Linux 下如何选择 IDE？

Linux 下可以不使用 IDE，只使用 GCC 编译器和一个文本编辑器（例如 Gedit）即可，这样对初学者理解 C 语言程序的生成过程非常有帮助，请参考：Linux GCC 简明教程（使用 GCC 编写 C 语言程序）

当然，如果你希望使用 IDE，那么可以选择 CodeLite、Code::Blocks、Anjuta、Eclipse、NetBeans 等。

Mac OS 下如何选择 IDE？

Mac OS 下推荐使用 Apple 官方开发的 Xcode，在 APP Store 即可下载，具体请参见：Xcode 简明教程（使用 Xcode 编写 C 语言程序）

另外，Visual Studio 也推出了 Mac 版本，已经习惯了 Visual Studio 的用户可以高兴一把了。

2.9 如何在手机上编写 C 语言代码？

2.10 C 语言的三套标准：C89、C99 和 C11

2.11 C 语言为什么有那么多编译器？

2.12 程序安装是怎么回事？

2.13 制作安装包，让用户安装程序

2.14 C 语言程序的错误和警告

一段 C 语言代码，在编译、链接和运行的各个阶段都可能会出现问题。编译器只能检查编译和链接阶段出现的问题，而可执行程序已经脱离了编译器，运行阶段出现问题编译器是无能为力的。

如果我们编写的代码正确，运行时会提示没有错误（Error）和警告（Warning），如下图所示：

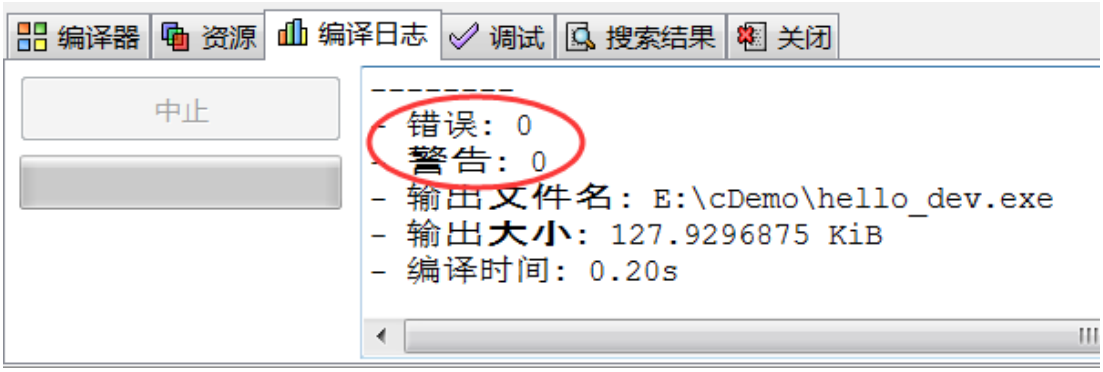


图 1：Dev C++ 的提示

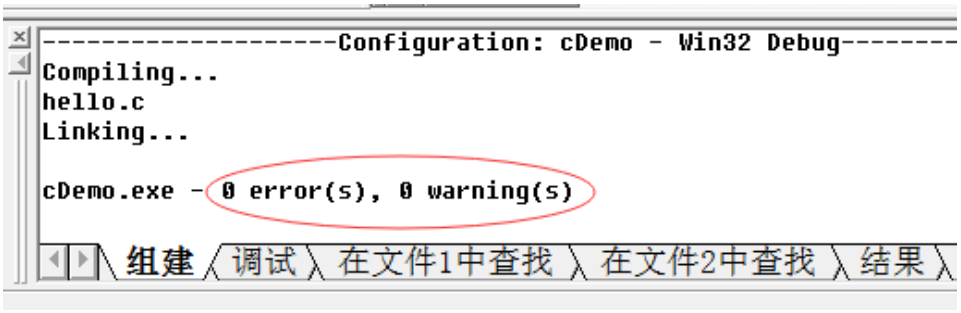


图 2：VC 6.0 的提示

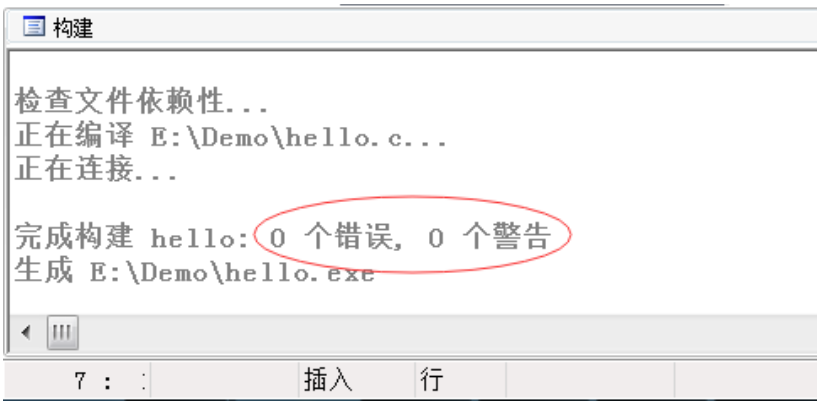


图 3：C-Free 5.0 的提示

> 对于 VS、GCC、Xcode 等，如果代码没有错误，它们只会显示“生成成功”，不会显示“0 个错误，0 个警告”，只有代码真的出错了，它们才会显示具体的错误信息。

错误（Error）表示程序不正确，不能正常编译、链接或运行，必须要纠正。

警告（Warning）表示可能会发生错误（实际上未发生）或者代码不规范，但是程序能够正常运行，有的警告可以忽略，有的要引起注意。

错误和警告可能发生在编译、链接、运行的任何时候。

例如，puts("C 语言中文网")最后忘记写分号;，就会出现错误，如下图所示：

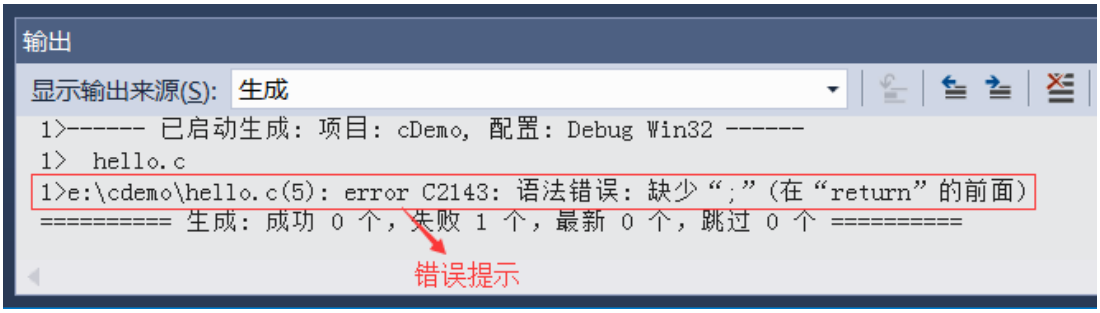


图 4：VS2015 的错误提示

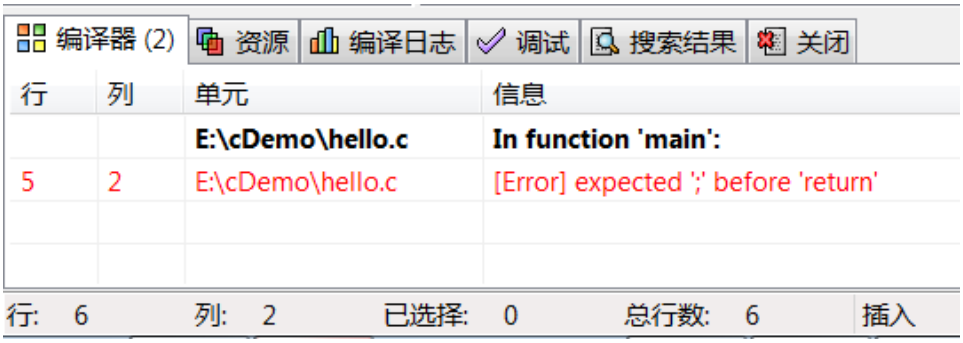


图 5：Dev C++ 的错误提示

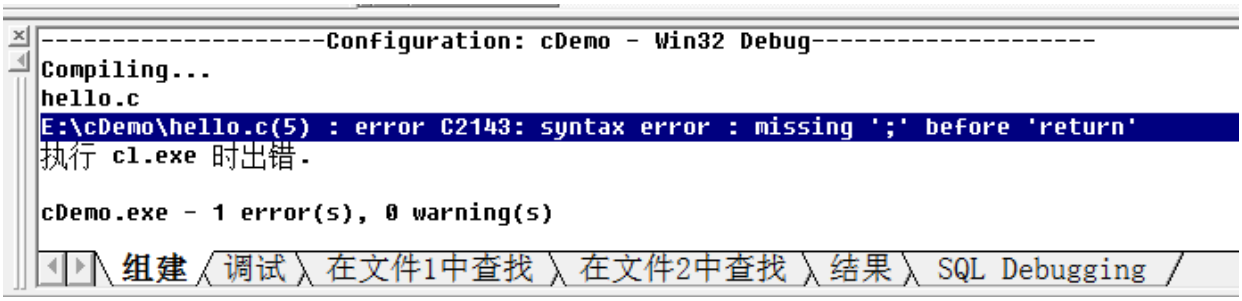


图 6：VC 6.0 的错误提示

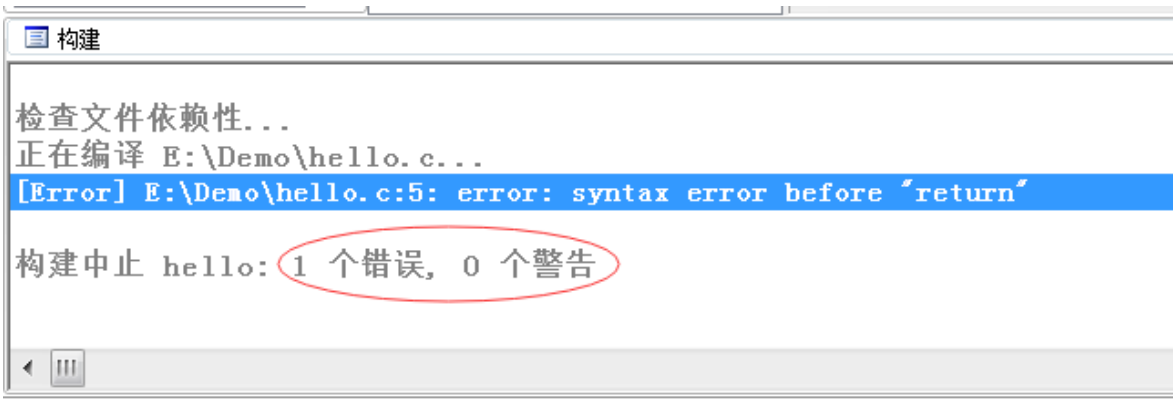


图 7：C-Free 5.0 的错误提示

可以看出，C-Free 的错误提示信息比较少，不方便程序员纠错。VC 和 VS 的错误信息类似，只是中英文的差别。

下图分析了 VC 6.0 的错误信息：

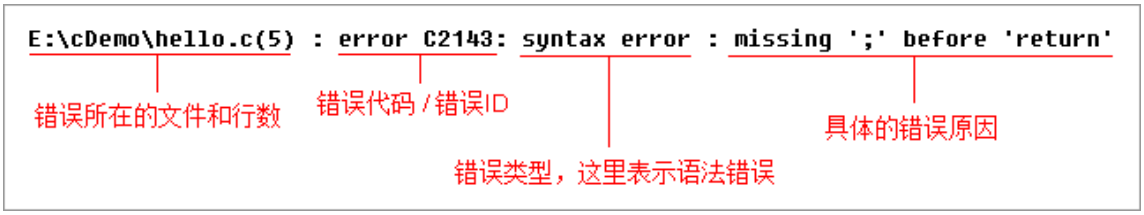


图 8：错误信息说明

翻译：源文件 E:\cDemo\hello.c 第 5 行发生了语法错误，错误代码是 C2143，原因是 'return' 前面丢失了 ';'。

我敢保证，你写的代码肯定会发生错误，一定要有分析错误的能力，这是一个合格的程序员必备的技能。

2.15 分析第一个 C 语言程序

前面我们给出了一段最简单的 C 语言代码，并演示了如何在不同的平台下进行编译，这节我们来分析一下这段代码，让读者有个整体的认识。代码如下：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      puts("C语言中文网");
5.      return 0;
6.  }
```

函数的概念

先来看第 4 行代码，这行代码会在显示器上输出“C 语言中文网”。前面我们已经讲过，puts 后面要带( )，字符串也要放在( )中。

在 C 语言中，有的语句使用时不能带括号，有的语句必须带括号。带括号的称为函数（Function）。

C 语言提供了很多功能，例如输入输出、获得日期时间、文件操作等，我们只需要一句简单的代码就能够使用。但是这些功能的底层都比较复杂，通常是软件和硬件的结合，还要要考虑很多细节和边界，如果将这些功能都交给程序员去完成，那将极大增加程序员的学习成本，降低编程效率。

好在 C 语言的开发者们为我们做了一件好事，他们已经编写了大量代码，将常见的基本功能都完成了，我们可以直接拿来使用。但是现在问题来了，那么多代码，如何从中找到自己需要的呢？一股脑将所有代码都拿来显然是非常不明智的。

这些代码，早已被分门别类地放在了不同的文件中，并且每一段代码都有唯一的名字。使用代码时，只要在对应的名字后面加上( )就可以。这样的一段代码能够独立地完成某个功能，一次编写完成后可以重复使用，被称为函数（Function）。读者可以认为，函数就是一段可以重复使用的代码。

函数的一个明显特征就是使用时必须带括号( )，必要的话，括号中还可以包含待处理的数据。例如 puts("C 语言中文网")就使用了一段具有输出功能的代码，这段代码的名字是 puts，"C 语言中文网" 是要交给这段代码处理的数据。使用函数在编程中有专业的称呼，叫做函数调用（Function Call）。

如果函数需要处理多个数据，那么它们之间使用逗号,分隔，例如：

```
pow(10, 2);
```

该函数用来求 10 的 2 次方。

> 需要注意的是，C 语言中的函数和数学中的函数不是同一个概念，不要拿两者对比。函数的英文名称是 Function，它还有“功能”的意思。大陆将 Function 翻译为“函数”，而台湾地区翻译为“函式”，读者要注意区分。

自定义函数和 main 函数

C 语言自带的函数称为库函数（Library Function）。库（Library）是编程中的一个基本概念，可以简单地认为它是一些列函数的集合，在磁盘上往往是一个文件夹。C 语言自带的库称为标准库（Standard Library），其他公司或个人开发的库称为第三方库（Third-Party Library）。

> 关于库的概念，我们已在《不要这样学习 C 语言，这是一个坑》中进行了详细介绍。

除了库函数，我们还可以编写自己的函数，拓展程序的功能。自己编写的函数称为自定义函数。自定义函数和库函数在编写和使用方式上完全相同，只是由不同的机构来编写。

示例中第 2~6 行代码就是我们自己编写的一个函数。main 是函数的名字，( ) 表明这是函数定义，{ } 之间的代码是函数要实现的功能。

函数可以接收待处理的数据，同样可以将处理结果告诉我们；使用 return 可以告知处理结果。示例中第 5 行代码表明，main 函数的处理结果是整数 0。return 可以翻译为“返回”，所以函数的处理结果被称为返回值（Return Value）。

第 2 行代码中，int 是 integer 的简写，意为“整数”。它告诉我们，函数的返回值是整数。

需要注意的是，示例中的自定义函数必须命名为 main。C 语言规定，一个程序必须有且只有一个 main 函数。main 被称为主函数，是程序的入口函数，程序运行时从 main 函数开始，直到 main 函数结束（遇到 return 或者执行到函数末尾时，函数才结束）。

也就是说，没有 main 函数程序将不知道从哪里开始执行，运行时会报错。

综上所述：第 2~6 行代码定义了主函数 main，它的返回值是整数 0，程序将从这里开始执行。main 函数的返回值在程序运行结束时由系统接收。

> 关于自定义函数的更多内容，我们将在《C 语言函数》一章中详细讲解，这里不再展开讨论。

有的教材中将 main 函数写作：

```
void main()
{
```

```
    // Some Code...
}
```

这在 VC6.0 下能够通过编译，但在 C-Free、GCC 中却会报错，因为这不是标准的 main 函数的写法，大家不要被误导，最好按照示例中的格式来写。

头文件的概念

还有最后一个问题，示例中第 1 行的#include <stdio.h>是什么意思呢？

C 语言开发者们编写了很多常用函数，并分门别类的放在了不同的文件，这些文件就称为头文件（header file）。每个头文件中都包含了若干个功能类似的函数，调用某个函数时，要引入对应的头文件，否则编译器找不到函数。

> 实际上，头文件往往只包含函数的说明，也就是告诉我们函数怎么用，而函数本身保存在其他文件中，在链接时才会找到。对于初学者，可以暂时理解为头文件中包含了若干函数。

引入头文件使用#include 命令，并将文件名放在< >中，#include 和 < > 之间可以有空格，也可以没有。

头文件以.h 为后缀，而 C 语言代码文件以.c 为后缀，它们都是文本文件，没有本质上的区别，#include 命令的作用也仅仅是将头文件中的文本复制到当前文件，然后和当前文件一起编译。你可以尝试将头文件中的内容复制到当前文件，那样也可以不引入头文件。

.h 中代码的语法规则和.c 中是一样的，你也可以#include <xxx.c>，这是完全正确的。不过实际开发中没有人会这样做，这样看起来非常不专业，也不规范。

较早的 C 语言标准库包含了 15 个头文件，stdio.h 和 stdlib.h 是最常用的两个：

- stdio 是 standard input output 的缩写，stdio.h 被称为“标准输入输出文件”，包含的函数大都和输入输出有关，puts() 就是其中之一。
- stdlib 是 standard library 的缩写，stdlib.h 被称为“标准库文件”，包含的函数比较杂乱，多是一些通用工具型函数，system() 就是其中之一。

最后的总结

初学编程，有很多基本概念需要了解，本节就涉及到很多，建议大家把上面的内容多读几遍，必将有所收获。

本节开头的示例是一个 C 语言程序的基本结构，我们不妨整理一下思路，从整体上再分析一遍：

1) 第 1 行引入头文件 stdio.h，这是编程中最常用的一个头文件。头文件不是必须要引入的，我们用到了 puts 函数，所以才引入 stdio.h。例如下面的代码完全正确：

```
1.  int main()
2.  {
3.      return 0;
4.  }
```

我们没有调用任何函数，所以不必引入头文件。

2) 第 2 行开始定义主函数 main。main 是程序的入口函数，一个 C 程序必须有 main 函数，而且只能有一个。

3) 第 4 行调用 puts 函数向显示器输出字符串。

4) 第 5 行是 main 函数的返回值。程序运行正确一般返回 0。

2.16 C 语言代码中的空白符

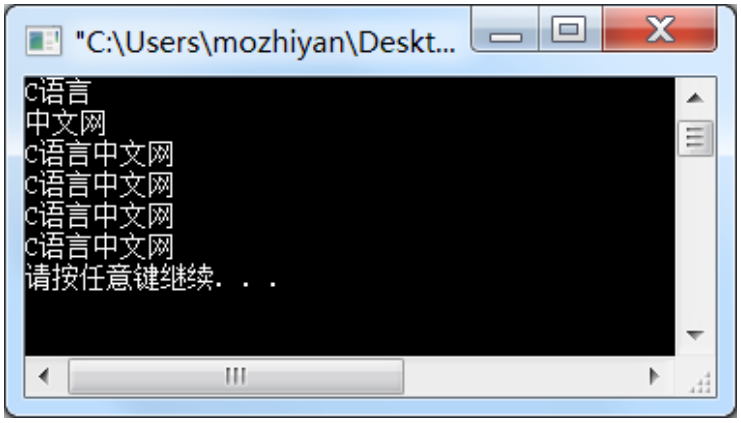
空格、制表符、换行符等统称为空白符（space character），它们只用来占位，并没有实际的内容，也显示不出具体的字符。

> 制表符分为水平制表符和垂直制表符，它们的 ASCII 编码值分别是 9 和 11。
>
> - 垂直制表符在现代计算机中基本不再使用了，也没法在键盘上直接输入，它已经被换行符取代了。
> - 水平制表符相当于四个空格，对于大部分编辑器，按下 Tab 键默认就是输入一个水平制表符；如果你进行了个性化设置，按下 Tab 键也可能会输入四个或者两个空格。

对于编译器，有的空白符会被忽略，有的却不能。请看下面几种 puts 的写法：

```
#include<stdio.h>
int main()
{
	puts("C 语言");
	puts("中文网");

	puts
	("C 语言中文网");

	puts
	(
	"C 语言中文网"
	)
	;

	puts	("C 语言中文网");

	puts	(	"C 语言中文网"	)	;

	return 0;
}
```

运行结果：

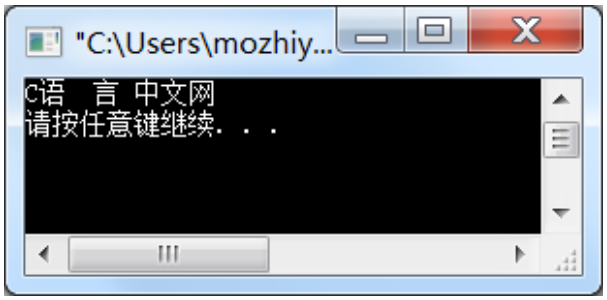


看到输出结果，说明代码没有错误，以上几种 puts 的用法是正确的。puts 和()之间、" "和()之间可以有任意的空白符，它们会被编译器忽略，编译器不认为它们是代码的一部分，它们的存在只是在编辑器中呈现一定的格式，让程序员阅读方便。

需要注意的是，由" "包围起来的字符串中的空白符不会被忽略，它们会被原样输出到控制台上；并且字符串中间不能换行，否则会产生编译错误。请看下面的代码：

```
#include<stdio.h>
int main()
{
    puts("C 语   言  中文网");
    puts("C 语言
    中文网");
    return 0;
}
```

第 5~6 行代码是错误的，字符串必须在一行内结束，不能换行。把这两行代码删除，运行结果为：



程序员要善于利用空白符：缩进（制表符）和换行可以让代码结构更加清晰，空格可以让代码看起来不那么拥挤。专业的程序员同样追求专业的代码格式，大家在以后的学习中可以慢慢体会。

2.17 彩色版的 C 语言，让文字更漂亮

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能

2.18 一个真正带界面的 C 语言程序

第 03 章 变量和数据类型

本章也是 C 语言的基础知识，主要讲解变量、数据类型以及运算符，这其中涉及到了数据的存储格式以及不同进制。

本章目录：

1. 大话 C 语言变量和数据类型
2. 在屏幕上输出各种类型的数据
3. C 语言中的整数（short,int,long）
4. C 语言中的二进制数、八进制数和十六进制数
5. C 语言中的正负数及其输出
6. 整数在内存中是如何存储的，为什么它堪称天才般的设计
7. C 语言整数的取值范围以及数值溢出
8. C 语言中的小数（float,double）
9. 小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）
10. 在 C 语言中使用英文字符
11. 在 C 语言中使用中文字符
12. C 语言到底使用什么编码？谁说 C 语言使用 ASCII 码，真是荒谬！
13. C 语言转义字符
14. C 语言标识符、关键字、注释、表达式和语句
15. C 语言加减乘除运算
16. C 语言自增(++)和自减(--)运算符
17. C 语言变量的定义位置以及初始值
18. C 语言运算符的优先级和结合性
19. C 语言数据类型转换（自动类型转换+强制类型转换）

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

3.1 大话 C 语言变量和数据类型

在《数据在内存中的存储》一节中讲到：

- 计算机要处理的数据（诸如数字、文字、符号、图形、音频、视频等）是以二进制的形式存放在内存中的；

➢ 我们将 8 个比特（Bit）称为一个字节（Byte），并将字节作为最小的可操作单元。

我们不妨先从最简单的整数说起，看看它是如何放到内存中去的。

变量（Variable）

现实生活中我们会找一个小箱子来存放物品，一来显得不那么凌乱，二来方便以后找到。计算机也是这个道理，我们需要先在内存中找一块区域，规定用它来存放整数，并起一个好记的名字，方便以后查找。这块区域就是“小箱子”，我们可以把整数放进去了。

C 语言中这样在内存中找一块区域：

```
int a;
```

int 又是一个新单词，它是 Integer 的简写，意思是整数。a 是我们给这块区域起的名字；当然也可以叫其他名字，例如 abc、mn123 等。

这个语句的意思是：在内存中找一块区域，命名为 a，用它来存放整数。

> 注意 int 和 a 之间是有空格的，它们是两个词。也注意最后的分号，int a 表达了完整的意思，是一个语句，要用分号来结束。

不过 int a;仅仅是在内存中找了一块可以保存整数的区域，那么如何将 123、100、999 这样的数字放进去呢？

C 语言中这样向内存中放整数：

```
a=123;
```

=是一个新符号，它在数学中叫“等于号”，例如 1+2=3，但在 C 语言中，这个过程叫做赋值（Assign）。赋值是指把数据放到内存的过程。

把上面的两个语句连起来：

```
int a;
a=123;
```

就把 123 放到了一块叫做 a 的内存区域。你也可以写成一个语句：

```
int a=123;
```

a 中的整数不是一成不变的，只要我们需要，随时可以更改。更改的方式就是再次赋值，例如：

```
int a=123;
a=1000;
a=9999;
```

第二次赋值，会把第一次的数据覆盖（擦除）掉，也就是说，a 中最后的值是 9999，123、1000 已经不存在了，再也找不回来了。

因为 a 的值可以改变，所以我们给它起了一个形象的名字，叫做变量（Variable）。

int a;创造了一个变量 a，我们把这个过程叫做变量定义。a=123;把 123 交给了变量 a，我们把这个过程叫做给变量赋值；又因为是第一次赋值，也称变量的初始化，或者赋初值。

你可以先定义变量，再初始化，例如：

```
int abc;
abc=999;
```

也可以在定义的同时进行初始化，例如：

```
int abc=999;
```

这两种方式是等价的。

数据类型（Data Type）

数据是放在内存中的，变量是给这块内存起的名字，有了变量就可以找到并使用这份数据。但问题是，该如何使用呢？

我们知道，诸如数字、文字、符号、图形、音频、视频等数据都是以二进制形式存储在内存中的，它们并没有本质上的区别，那么，00010000 该理解为数字 16 呢，还是图像中某个像素的颜色呢，还是要发出某个声音呢？如果没有特别指明，我们并不知道。

也就是说，内存中的数据有多种解释方式，使用之前必须要确定；上面的 int a;就表明，这份数据是整数，不能理解为像素、声音等。int 有一个专业的称呼，叫做数据类型（Data Type）。

顾名思义，数据类型用来说明数据的类型，确定了数据的解释方式，让计算机和程序员不会产生歧义。在 C 语言中，有多种数据类型，例如：

说　明	字符型	短整型	整型	长整型	单精度浮点型	双精度浮点型	无类型
数据类型	char	short	int	long	float	double	void

这些是最基本的数据类型，是 C 语言自带的，如果我们需要，还可以通过它们组成更加复杂的数据类型，后面我们会一一讲解。

连续定义多个变量

为了让程序的书写更加简洁，C 语言支持多个变量的连续定义，例如：

```
int a, b, c;
float m = 10.9, n = 20.56;
char p, q = '@';
```

连续定义的多个变量以逗号,分隔，并且要拥有相同的数据类型；变量可以初始化，也可以不初始化。

数据的长度（Length）

所谓数据长度（Length），是指数据占用多少个字节。占用的字节越多，能存储的数据就越多，对于数字来说，值就会更大，反之能存储的数据就有限。

多个数据在内存中是连续存储的，彼此之间没有明显的界限，如果不明确指明数据的长度，计算机就不知道何时存取结束。例如我们保存了一个整数 1000，它占用 4 个字节的内存，而读取时却认为它占用 3 个字节或 5 个字节，这显然是不正确的。

所以，在定义变量时还要指明数据的长度。而这恰恰是数据类型的另外一个作用。数据类型除了指明数据的解释方式，还指明了数据的长度。因为在 C 语言中，每一种数据类型所占用的字节数都是固定的，知道了数据类型，也就知道了数据的长度。

在 32 位环境中，各种数据类型的长度一般如下：

说　明	字符型	短整型	整型	长整型	单精度浮点型	双精度浮点型
数据类型	char	short	int	long	float	double
长　度	1	2	4	4	4	8

C 语言有多少种数据类型，每种数据类型长度是多少、该如何使用，这是每一位 C 程序员都必须要掌握的，后续我们会一一讲解。

最后的总结

数据是放在内存中的，在内存中存取数据要明确三件事情：数据存储在哪里、数据的长度以及数据的处理方式。

变量名不仅仅是为数据起了一个好记的名字，还告诉我们数据存储在哪里，使用数据时，只要提供变量名即可；而数据类型则指明了数据的长度和处理方式。所以诸如 `int n;`、`char c;`、`float money;`这样的形式就确定了数据在内存中的所有要素。

C 语言提供的多种数据类型让程序更加灵活和高效，同时也增加了学习成本。而有些编程语言，例如 PHP、JavaScript 等，在定义变量时不需要指明数据类型，编译器会根据赋值情况自动推演出数据类型，更加智能。

除了 C 语言，Java、C++、C#等在定义变量时也必须指明数据类型，这样的编程语言称为强类型语言。而 PHP、JavaScript 等在定义变量时不必指明数据类型，编译系统会自动推演，这样的编程语言称为弱类型语言。

强类型语言一旦确定了数据类型，就不能再赋给其他类型的数据，除非对数据类型进行转换。弱类型语言没有这种限制，一个变量，可以先赋给一个整数，然后再赋给一个字符串。

最后需要说明的是：数据类型只在定义变量时指明，而且必须指明；使用变量时无需再指明，因为此时的数据类型已经确定了。

3.2 在屏幕上输出各种类型的数据

在《第一个 C 语言程序》一节中，我们使用 puts 来输出字符串。puts 是 output string 的缩写，只能用来输出字

符串，不能输出整数、小数、字符等，我们需要用另外一个函数，那就是 printf。

printf 比 puts 更加强大，不仅可以输出字符串，还可以输出整数、小数、单个字符等，并且输出格式也可以自己定义，例如：

- 以十进制、八进制、十六进制形式输出；
- 要求输出的数字占 n 个字符的位置；
- 控制小数的位数。

printf 是 print format 的缩写，意思是“格式化打印”。这里所谓的“打印”就是在屏幕上显示内容，与“输出”的含义相同，所以我们一般称 printf 是用来格式化输出的。

先来看一个简单的例子：

```
printf("C 语言中文网");
```

这个语句可以在屏幕上显示“C 语言中文网”，与 puts("C 语言中文网");的效果类似。

输出变量 abc 的值：

```
int abc=999;
printf("%d", abc);
```

这里就比较有趣了。先来看%d，d 是 decimal 的缩写，意思是十进制数，%d 表示以十进制整数的形式输出。输出什么呢？输出变量 abc 的值。%d 与 abc 是对应的，也就是说，会用 abc 的值来替换 %d。

再来看个复杂点的：

```
int abc=999;
printf("The value of abc is %d !", abc);
```

会在屏幕上显示：
The value of abc is 999 !

你看，字符串 "The value of abc is %d !" 中的 %d 被替换成了 abc 的值，其他字符没有改变。这说明 %d 比较特殊，不会原样输出，会被替换成对应的变量的值。

再来看：

```
int a=100;
int b=200;
int c=300;
printf("a=%d, b=%d, c=%d", a, b, c);
```

会在屏幕上显示：
a=100, b=200, c=300

再次证明了 %d 与后面的变量是一一对应的，第一个 %d 对应第一个变量，第二个 %d 对应第二个变量……

%d 称为格式控制符，它指明了以何种形式输出数据。格式控制符均以%开头，后跟其他字符。%d 表示以十进制形式输出一个整数。除了 %d，printf 支持更多的格式控制，例如：

- %c：输出一个字符。c 是 character 的简写。
- %s：输出一个字符串。s 是 string 的简写。
- %f：输出一个小数。f 是 float 的简写。

除了这些，printf 支持更加复杂和优美的输出格式，考虑到读者的基础暂时不够，我们将在《C 语言数据输出大汇总以及轻量进阶》一节中展开讲解。

我们把代码补充完整，体验一下：

```
#include <stdio.h>
int main()
{
    int n = 100;
    char c = '@';  //字符用单引号包围，字符串用双引号包围
    float money = 93.96;
    printf("n=%d, c=%c, money=%f\n", n, c, money);

    return 0;
}
```

输出结果：

n=100, c=@, money=93.959999

要点提示：

1) \n 是一个整体，组合在一起表示一个换行字符。换行符是 ASCII 编码中的一个控制字符，无法在键盘上直接输入，只能用这种特殊的方法表示，被称为转义字符，我们将在《C 语言转义字符》一节中有具体讲解，请大家暂时先记住\n 的含义。

> 所谓换行，就是让文本从下一行的开头输出，相当于在编辑 Word 或者 TXT 文档时按下回车键。

puts 输出完成后会自动换行，而 printf 不会，要自己添加换行符，这是 puts 和 printf 在输出字符串时的一个区别。

2) //后面的为注释。注释用来说明代码是什么意思，起到提示的作用，可以帮助我们理解代码。注释虽然也是代码的一部分，但是它并不会给程序带来任何影响，编译器在编译阶段会忽略注释的内容，或者说删除注释的内容。我们将在《C 语言标识符、关键字和注释》一节中详细讲解。

3) money 的输出值并不是 93.96，而是一个非常接近的值，这与小数本身的存储机制有关，这种机制导致很多小数不能被精确地表示，即使像 93.96 这种简单的小数也不行。我们将在《小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）》一节详细介绍。

我们也可以不用变量，将数据直接输出：

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

```
4.      float money = 93.96;
5.      printf("n=%d, c=%c, money=%f\n", 100, '@', money);
6.
7.      return 0;
8.  }
```

输出结果与上面相同。

在以后的编程中，我们会经常使用 printf，说它是 C 语言中使用频率最高的一个函数一点也不为过，每个 C 语言程序员都应该掌握 printf 的用法，这是最基本的技能。

不过 printf 的用法比较灵活，也比较复杂，初学者知识储备不足，不能一下子掌握，目前大家只需要掌握最基本的用法，以后随着编程知识的学习，我们会逐步介绍更加高级的用法，最终让大家完全掌握 printf。

【脑筋急转弯】%ds 输出什么

%d 输出整数，%s 输出字符串，那么 %ds 输出什么呢？

我们不妨先来看一个例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a = 1234;
5.      printf("a=%ds\n", a);
6.
7.      return 0;
8.  }
```

运行结果：

a=1234s

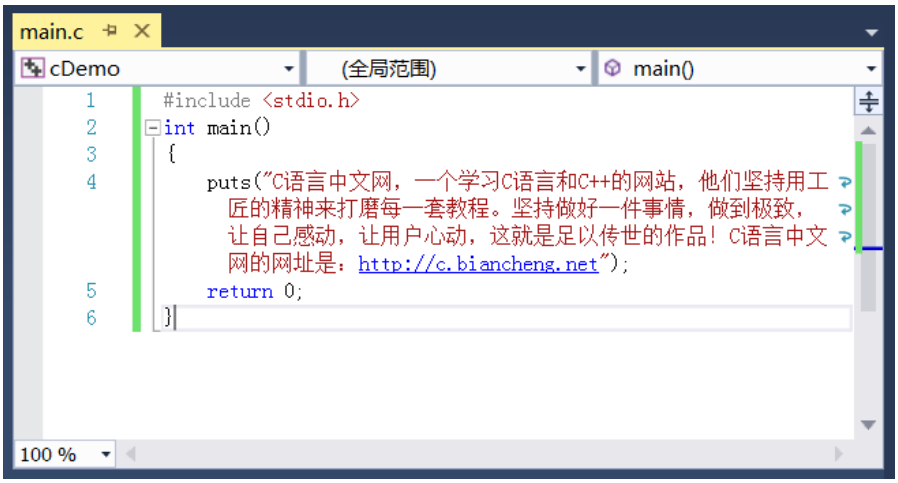
从输出结果可以发现，`%d` 被替换成了变量 a 的值，而 `s` 没有变，原样输出了。这是因为，`%d` 才是格式控制符，`%ds` 在一起没有意义，`s` 仅仅是跟在`%d` 后面的一个普通字符，所以会原样输出。

【拓展】如何在字符串中书写长文本

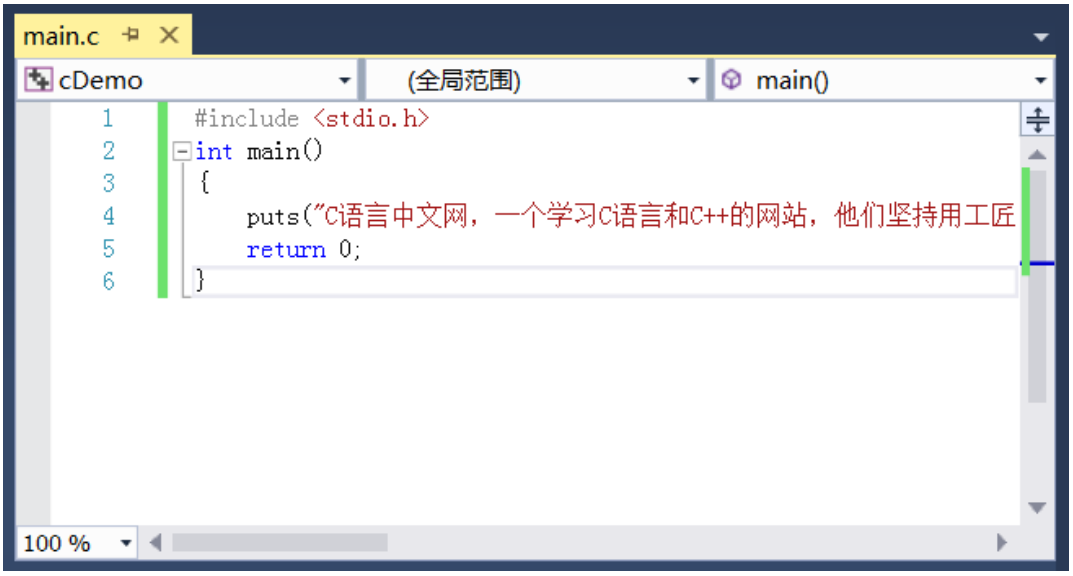
假设现在我们要输出一段比较长的文本，它的内容为：

C 语言中文网，一个学习 C 语言和 C++的网站，他们坚持用工匠的精神来打磨每一套教程。坚持做好一件事情，做到极致，让自己感动，让用户心动，这就是足以传世的作品！C 语言中文网的网址是：http://c.biancheng.net

如果将这段文本放在一个字符串中，会显得比较臃肿，格式也不好看，就像下面这样：



超出编辑窗口宽度的文本换行



超出编辑窗口宽度的文本隐藏

当文本超出编辑窗口的宽度时，可以选择将文本换行，也可以选择将文本隐藏（可以在编辑器里面自行设置），但是不管哪种形式，在一个字符串里书写长文本总是不太美观。

当然，你可以多写几个 puts 函数，就像下面这样：

我不否认这种写法也比较美观，但是这里我要讲的是另外一种写法：

```
#include <stdio.h>
int main()
{
    puts(
        "C语言中文网，一个学习C语言和C++的网站，他们坚持用工匠的精神来打磨每一套教程。"
        "坚持做好一件事情，做到极致，让自己感动，让用户心动，这就是足以传世的作品！"
        "C语言中文网的网址是：http://c.biancheng.net"
        );
    return 0;
}
```

在 puts 函数中，可以将一个较长的字符串分割成几个较短的字符串，这样会使得长文本的格式更加整齐。

注意，这只是形式上的分割，编译器在编译阶段会将它们合并为一个字符串，它们放在一块连续的内存中。

多个字符串并不一定非得换行，也可以将它们写在一行中，例如：

```
#include <stdio.h>
int main()
{
    puts("C语言中文网！" "C语言和C++！" "http://c.biancheng.net");
    return 0;
}
```

本节讲到的 puts、printf，以及后面要讲到的 fprintf、fputs 等与字符串输出有关的函数，都支持这种写法。

3.3 C 语言中的整数（short,int,long）

整数是编程中常用的一种数据，C 语言通常使用 int 来定义整数（int 是 integer 的简写），这在《大话 C 语言变量和数据类型》中已经进行了详细讲解。

在现代操作系统中，int **一般**占用 4 个字节（Byte）的内存，共计 32 位（Bit）。如果不考虑正负数，当所有的位都为 1 时它的值最大，为 2^{32}-1 = 4,294,967,295 ≈ 43 亿，这是一个很大的数，实际开发中很少用到，而诸如 1、99、12098 等较小的数使用频率反而较高。

使用 4 个字节保存较小的整数绰绰有余，会空闲出两三个字节来，这些字节就白白浪费掉了，不能再被其他数据使用。现在个人电脑的内存都比较大了，配置低的也有 2G，浪费一些内存不会带来明显的损失；而在 C 语言被发明的早期，或者在单片机和嵌入式系统中，内存都是非常稀缺的资源，所有的程序都在尽力节省内存。

反过来说，43 亿虽然已经很大，但要表示全球人口数量还是不够，必须要让整数占用更多的内存，才能表示更大的值，比如占用 6 个字节或者 8 个字节。

让整数占用更少的内存可以在 int 前边加 **short**，让整数占用更多的内存可以在 int 前边加 **long**，例如：

```
short int a = 10;
short int b, c = 99;
long int m = 102023;
long int n, p = 562131;
```

这样 a、b、c 只占用 2 个字节的内存，而 m、n、p **可能**会占用 8 个字节的内存。

也可以将 int 省略，只写 short 和 long，如下所示：

```
short a = 10;
short b, c = 99;
long m = 102023;
long n, p = 562131;
```

这样的写法更加简洁，实际开发中常用。

int 是基本的整数类型，short 和 long 是在 int 的基础上进行的扩展，short 可以节省内存，long 可以容纳更大的值。

short、int、long 是 C 语言中常见的整数类型，其中 int 称为整型，short 称为短整型，long 称为长整型。

整型的长度

细心的读者可能会发现，上面我们在描述 short、int、long 类型的长度时，只对 short 使用肯定的说法，而对 int、long 使用了“一般”或者“可能”等不确定的说法。这种描述的言外之意是，只有 short 的长度是确定的，是两个字节，而 int 和 long 的长度无法确定，在不同的环境下有不同的表现。

> 一种数据类型占用的字节数，称为该数据类型的长度。例如，short 占用 2 个字节的内存，那么它的长度就是 2。

实际情况也确实如此，C 语言并没有严格规定 short、int、long 的长度，只做了宽泛的限制：

- short 至少占用 2 个字节。
- int 建议为一个机器字长。32 位环境下机器字长为 4 字节，64 位环境下机器字长为 8 字节。
- short 的长度不能大于 int，long 的长度不能小于 int。

总结起来，它们的长度（所占字节数）关系为：

```
2 ≤ short ≤ int ≤ long
```

这就意味着，short 并不一定真的”短“，long 也并不一定真的”长“，它们有可能和 int 占用相同的字节数。

在 16 位环境下，short 的长度为 2 个字节，int 也为 2 个字节，long 为 4 个字节。16 位环境多用于单片机和低级嵌入式系统，在 PC 和服务器上已经见不到了。

对于 32 位的 Windows、Linux 和 Mac OS，short 的长度为 2 个字节，int 为 4 个字节，long 也为 4 个字节。PC 和服务器上的 32 位系统占有率也在慢慢下降，嵌入式系统使用 32 位越来越多。

在 64 位环境下，不同的操作系统会有不同的结果，如下所示：

操作系统	short	int	long
Win64（64 位 Windows）	2	4	4
类 Unix 系统（包括 Unix、Linux、Mac OS、BSD、Solaris 等）	2	4	8

目前我们使用较多的 PC 系统为 Win XP、Win 7、Win 8、Win 10、Mac OS、Linux，在这些系统中，short 和 int 的长度都是固定的，分别为 2 和 4，大家可以放心使用，只有 long 的长度在 Win64 和类 Unix 系统下会有所不同，使用时要注意移植性。

sizeof 操作符

获取某个数据类型的长度可以使用 sizeof 操作符，如下所示：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      short a = 10;
5.      int b = 100;
6.
7.      int short_length = sizeof a;
8.      int int_length = sizeof(b);
9.      int long_length = sizeof(long);
10.     int char_length = sizeof(char);
11.
12.     printf("short=%d, int=%d, long=%d, char=%d\n", short_length, int_length, long_length, char_length);
13.
14.     return 0;
15. }
```

在 32 位环境以及 Win64 环境下的运行结果为：

```
short=2, int=4, long=4, char=1
```

在 64 位 Linux 和 Mac OS 下的运行结果为：

```
short=2, int=4, long=8, char=1
```

sizeof 用来获取某个数据类型或变量所占用的字节数，如果后面跟的是变量名称，那么可以省略()，如果跟的是数据类型，就必须带上()。

需要注意的是，sizeof 是 C 语言中的操作符，不是函数，所以可以不带()，后面会详细讲解。

不同整型的输出

使用不同的格式控制符可以输出不同类型的整数，它们分别是：

- %hd 用来输出 short int 类型，hd 是 short decimal 的简写；
- %d 用来输出 int 类型，d 是 decimal 的简写；
- %ld 用来输出 long int 类型，ld 是 long decimal 的简写。

下面的例子演示了不同整型的输出：

```
#include <stdio.h>
int main()
{
    short a = 10;
    int b = 100;
    long c = 9437;

    printf("a=%hd, b=%d, c=%ld\n", a, b, c);
    return 0;
}
```

运行结果：

a=10, b=100, c=9437

在编写代码的过程中，我建议将格式控制符和数据类型严格对应起来，养成良好的编程习惯。当然，如果你不严格对应，一般也不会导致错误，例如，很多初学者都使用%d 输出所有的整数类型，请看下面的例子：

```
#include <stdio.h>
int main()
{
    short a = 10;
    int b = 100;
    long c = 9437;

    printf("a=%d, b=%d, c=%d\n", a, b, c);
    return 0;
}
```

运行结果仍然是：

a=10, b=100, c=9437

当使用%d 输出 short，或者使用%ld 输出 short、int 时，不管值有多大，都不会发生错误，因为格式控制符足够容纳这些值。

当使用%hd 输出 int、long，或者使用%d 输出 long 时，如果要输出的值比较小（就像上面的情况），一般也不会发生错误，如果要输出的值比较大，就很有可能发生错误，例如：

```
#include <stdio.h>
int main()
{
    int m = 306587;
    long n = 28166459852;
    printf("m=%hd, n=%hd\n", m, n);
    printf("n=%d\n", n);

    return 0;
}
```

在 64 位 Linux 和 Mac OS 下（long 的长度为 8）的运行结果为：

```
m=-21093, n=4556
n=-1898311220
```

输出结果完全是错误的，这是因为%hd 容纳不下 m 和 n 的值，%d 也容纳不下 n 的值。

读者需要注意，当格式控制符和数据类型不匹配时，编译器会给出警告，提示程序员可能会存在风险。

> 编译器的警告是分等级的，不同程度的风险被划分成了不同的警告等级，而使用%d 输出 short 和 long 类型的风险较低，如果你的编译器设置只对较高风险的操作发出警告，那么此处你就看不到警告信息。

3.4 C 语言中的二进制数、八进制数和十六进制数

C 语言中的整数除了可以使用十进制，还可以使用二进制、八进制和十六进制。

二进制数、八进制数和十六进制数的表示

一个数字默认就是十进制的，表示一个十进制数字不需要任何特殊的格式。但是，表示一个二进制、八进制或者十六进制数字就不一样了，为了和十进制数字区分开来，必须采用某种特殊的写法，具体来说，就是在数字前面加上特定的字符，也就是加前缀。

1) 二进制

二进制由 0 和 1 两个数字组成，使用时必须以 0b 或 0B（不区分大小写）开头，例如：

```
//合法的二进制
int a = 0b101;  //换算成十进制为 5
int b = -0b110010;  //换算成十进制为 -50
int c = 0B100001;  //换算成十进制为 33

//非法的二进制
```

```
int m = 101010;  //无前缀 0B，相当于十进制
int n = 0B410;  //4不是有效的二进制数字
```

读者请注意，标准的 C 语言并不支持上面的二进制写法，只是有些编译器自己进行了扩展，才支持二进制数字。换句话说，并不是所有的编译器都支持二进制数字，只有一部分编译器支持，并且跟编译器的版本有关系。

下面是实际测试的结果：

- Visual C++ 6.0 不支持。
- Visual Studio 2015 支持，但是 Visual Studio 2010 不支持；可以认为，高版本的 Visual Studio 支持二进制数字，低版本的 Visual Studio 不支持。
- GCC 4.8.2 支持，但是 GCC 3.4.5 不支持；可以认为，高版本的 GCC 支持二进制数字，低版本的 GCC 不支持。
- LLVM/Clang 支持（内嵌于 Mac OS 下的 Xcode 中）。

2) 八进制

八进制由 0~7 八个数字组成，使用时必须以 0 开头（注意是数字 0，不是字母 o），例如：

```
//合法的八进制数
int a = 015;  //换算成十进制为 13
int b = -0101;  //换算成十进制为 -65
int c = 0177777;  //换算成十进制为 65535

//非法的八进制
int m = 256;  //无前缀 0，相当于十进制
int n = 03A2;  //A不是有效的八进制数字
```

3) 十六进制

十六进制由数字 0~9、字母 A~F 或 a~f（不区分大小写）组成，使用时必须以 0x 或 0X（不区分大小写）开头，例如：

```
//合法的十六进制
int a = 0X2A;  //换算成十进制为 42
int b = -0XA0;  //换算成十进制为 -160
int c = 0xffff;  //换算成十进制为 65535

//非法的十六进制
int m = 5A;  //没有前缀 0X，是一个无效数字
int n = 0X3H;  //H不是有效的十六进制数字
```

4) 十进制

十进制由 0~9 十个数字组成，没有任何前缀，和我们平时的书写格式一样，不再赘述。

二进制数、八进制数和十六进制数的输出

C 语言中常用的整数有 short、int 和 long 三种类型，通过 printf 函数，可以将它们以八进制、十进制和十六进制的形式输出。上节我们讲解了如何以十进制的形式输出，这节我们重点讲解如何以八进制和十六进制的形式输出，下表列出了不同类型的整数、以不同进制的形式输出时对应的格式控制符：

	short	int	long
八进制	%ho	%o	%lo
十进制	%hd	%d	%ld
十六进制	%hx 或者 %hX	%x 或者 %X	%lx 或者 %lX

十六进制数字的表示用到了英文字母，有大小写之分，要在格式控制符中体现出来：

- %hx、%x 和 %lx 中的 x 小写，表明以小写字母的形式输出十六进制数；
- %hX、%X 和 %lX 中的 X 大写，表明以大写字母的形式输出十六进制数。

八进制数字和十进制数字不区分大小写，所以格式控制符都用小写形式。如果你比较叛逆，想使用大写形式，那么行为是未定义的，请你慎重：

- 有些编译器支持大写形式，只不过行为和小写形式一样；
- 有些编译器不支持大写形式，可能会报错，也可能会导致奇怪的输出。

注意，虽然部分编译器支持二进制数字的表示，但是却不能使用 printf 函数输出二进制，这一点比较遗憾。当然，通过转换函数可以将其它进制数字转换成二进制数字，并以字符串的形式存储，然后在 printf 函数中使用%s 输出即可。考虑到读者的基础还不够，这里就先不讲这种方法了。

【实例】以不同进制的形式输出整数：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      short a = 0b1010110;  //二进制数字
5.      int b = 02713;  //八进制数字
6.      long c = 0X1DAB83;  //十六进制数字
7.
8.      printf("a=%ho, b=%o, c=%lo\n", a, b, c);  //以八进制形似输出
9.      printf("a=%hd, b=%d, c=%ld\n", a, b, c);  //以十进制形式输出
10.     printf("a=%hx, b=%x, c=%lx\n", a, b, c);  //以十六进制形式输出（字母小写）
11.     printf("a=%hX, b=%X, c=%lX\n", a, b, c);  //以十六进制形式输出（字母大写）
12.
13.     return 0;
14. }
```

运行结果：

```
a=126, b=2713, c=7325603
a=86, b=1483, c=1944451
a=56, b=5cb, c=1dab83
a=56, b=5CB, c=1DAB83
```

从这个例子可以发现，一个数字不管以何种进制来表示，都能够以任意进制的形式输出。数字在内存中始终以二进制的形式存储，其它进制的数字在存储前都必须转换为二进制形式；同理，一个数字在输出时要进行逆向的转换，也就是从二进制转换为其他进制。

输出时加上前缀

请读者注意观察上面的例子，会发现有一点不完美，如果只看输出结果：

- 对于八进制数字，它没法和十进制、十六进制区分，因为八进制、十进制和十六进制都包含 0~7 这几个数字。
- 对于十进制数字，它没法和十六进制区分，因为十六进制也包含 0~9 这几个数字。如果十进制数字中还不包含 8 和 9，那么也不能和八进制区分了。
- 对于十六进制数字，如果没有包含 a~f 或者 A~F，那么就无法和十进制区分，如果还不包含 8 和 9，那么也不能和八进制区分了。

区分不同进制数字的一个简单办法就是，在输出时带上特定的前缀。在格式控制符中加上#即可输出前缀，例如 %#x、%#o、%#lX、%#ho 等，请看下面的代码：

```
#include <stdio.h>
int main()
{
    short a = 0b1010110;  //二进制数字
    int b = 02713;  //八进制数字
    long c = 0X1DAB83;  //十六进制数字

    printf("a=%#ho, b=%#o, c=%#lo\n", a, b, c);  //以八进制形似输出
    printf("a=%hd, b=%d, c=%ld\n", a, b, c);  //以十进制形式输出
    printf("a=%#hx, b=%#x, c=%#lx\n", a, b, c);  //以十六进制形式输出（字母小写）
    printf("a=%#hX, b=%#X, c=%#lX\n", a, b, c);  //以十六进制形式输出（字母大写）

    return 0;
}
```

运行结果：

a=0126, b=02713, c=07325603
a=86, b=1483, c=1944451
a=0x56, b=0x5cb, c=0x1dab83
a=0X56, b=0X5CB, c=0X1DAB83

十进制数字没有前缀，所以不用加#。如果你加上了，那么它的行为是未定义的，有的编译器支持十进制加#，只不过输出结果和没有加#一样，有的编译器不支持加#，可能会报错，也可能会导致奇怪的输出；但是，大部分编译器都能正常输出，不至于当成一种错误。

3.5 C 语言中的正负数及其输出

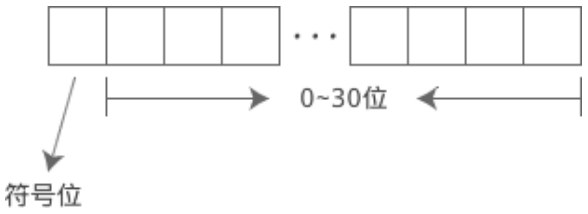
在数学中，数字有正负之分。在 C 语言中也是一样，short、int、long 都可以带上正负号，例如：

```
//负数
short a1 = -10;
short a2 = -0x2dc9;  //十六进制
//正数
int b1 = +10;
int b2 = +0174;  //八进制
```

```
int b3 = 22910;
//负数和正数相加
long c = (-9) + (+12);
```

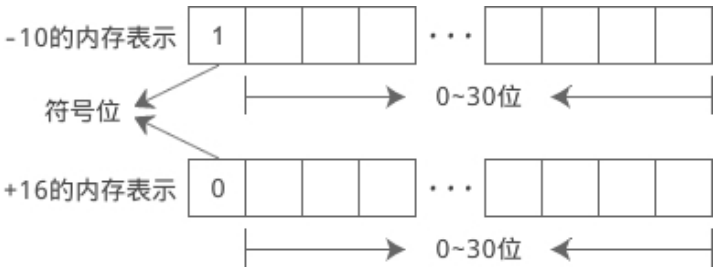
如果不带正负号，默认就是正数。

符号也是数字的一部分，也要在内存中体现出来。符号只有正负两种情况，用 1 位（Bit）就足以表示；C 语言规定，把内存的最高位作为符号位。以 int 为例，它占用 32 位的内存，0~30 位表示数值，31 位表示正负号。如下图所示：



> 在编程语言中，计数往往是从 0 开始，例如字符串 "abc123"，我们称第 0 个字符是 a，第 1 个字符是 b，第 5 个字符是 3。这和我们平时从 1 开始计数的习惯不一样，大家要慢慢适应，培养编程思维。

C 语言规定，在符号位中，用 0 表示正数，用 1 表示负数。例如 int 类型的 -10 和 +16 在内存中的表示如下：



short、int 和 long 类型默认都是带符号位的，符号位以外的内存才是数值位。如果只考虑正数，那么各种类型能表示的数值范围（取值范围）就比原来小了一半。

但是在很多情况下，我们非常确定某个数字只能是正数，比如班级学生的人数、字符串的长度、内存地址等，这个时候符号位就是多余的了，就不如删掉符号位，把所有的位都用来存储数值，这样能表示的数值范围更大（大一倍）。

C 语言允许我们这样做，如果不希望设置符号位，可以在数据类型前面加上 **unsigned** 关键字，例如：

```
unsigned short a = 12;
unsigned int b = 1002;
unsigned long c = 9892320;
```

这样，short、int、long 中就没有符号位了，所有的位都用来表示数值，正数的取值范围更大了。这也意味着，使用了 unsigned 后只能表示正数，不能再表示负数了。

如果将一个数字分为符号和数值两部分，那么不加 unsigned 的数字称为有符号数，能表示正数和负数，加了 unsigned 的数字称为无符号数，只能表示正数。

请读者注意一个小细节，如果是 unsigned int 类型，那么可以省略 int ，只写 unsigned，例如：

```
unsigned n = 100;
```

它等价于：

```
unsigned int n = 100;
```

无符号数的输出

无符号数可以以八进制、十进制和十六进制的形式输出，它们对应的格式控制符分别为：

	unsigned short	unsigned int	unsigned long
八进制	%ho	%o	%lo
十进制	%hu	%u	%lu
十六进制	%hx 或者 %hX	%x 或者 %X	%lx 或者 %lX

上节我们也讲到了不同进制形式的输出，但是上节我们还没有讲到正负数，所以也没有关心这一点，只是“笼统”地介绍了一遍。现在本节已经讲到了正负数，那我们就再深入地说一下。

严格来说，格式控制符和整数的符号是紧密相关的，具体就是：

- %d 以十进制形式输出有符号数；
- %u 以十进制形式输出无符号数；
- %o 以八进制形式输出无符号数；
- %x 以十六进制形式输出无符号数。

那么，如何以八进制和十六进制形式输出有符号数呢？很遗憾，printf 并不支持，也没有对应的格式控制符。在实际开发中，也基本没有“输出负的八进制数或者十六进制数”这样的需求，我想可能正是因为这一点，printf 才没有提供对应的格式控制符。

下表全面地总结了不同类型的整数，以不同进制的形式输出时对应的格式控制符（--表示没有对应的格式控制符）。

	short	int	long	unsigned short	unsigned int	unsigned long
八进制	--	--	--	%ho	%o	%lo
十进制	%hd	%d	%ld	%hu	%u	%lu
十六进制	--	--	--	%hx 或者 %hX	%x 或者 %X	%lx 或者 %lX

有读者可能会问，上节我们也使用 %o 和 %x 来输出有符号数了，为什么没有发生错误呢？这是因为：

- 当以有符号数的形式输出时，printf 会读取数字所占用的内存，并把最高位作为符号位，把剩下的内存作为数值位；
- 当以无符号数的形式输出时，printf 也会读取数字所占用的内存，并把所有的内存都作为数值位对待。

对于一个有符号的正数，它的符号位是 0，当按照无符号数的形式读取时，符号位就变成了数值位，但是该位恰好是 0 而不是 1，所以对数值不会产生影响，这就好比在一个数字前面加 0，有多少个 0 都不会影响数字的值。

如果对一个有符号的负数使用 %o 或者 %x 输出，那么结果就会大相径庭，读者可以亲试。

可以说，“有符号正数的最高位是 0”这个巧合才使得 %o 和 %x 输出有符号数时不会出错。

再次强调，不管是以 %o、%u、%x 输出有符号数，还是以 %d 输出无符号数，编译器都不会报错，只是对内存的解释不同了。%o、%d、%u、%x 这些格式控制符不会关心数字在定义时到底是有符号的还是无符号的：

- 你让我输出无符号数，那我在读取内存时就不区分符号位和数值位了，我会把所有的内存都看做数值位；
- 你让我输出有符号数，那我在读取内存时会把最高位作为符号位，把剩下的内存作为数值位。

说得再直接一些，我管你在定义时是有符号数还是无符号数呢，我只关心内存，有符号数也可以按照无符号数输出，无符号数也可以按照有符号数输出，至于输出结果对不对，那我就不管了，你自己承担风险。

下面的代码进行了全面的演示：

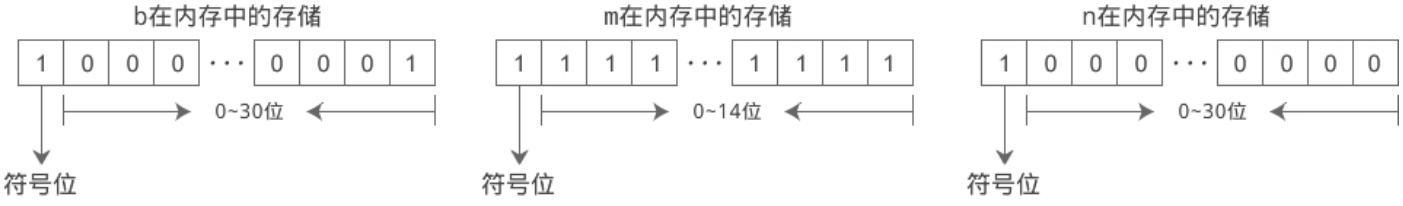
```
#include <stdio.h>
int main()
{
    short a = 0100;  //八进制
    int b = -0x1;  //十六进制
    long c = 720;  //十进制

    unsigned short m = 0xffff;  //十六进制
    unsigned int n = 0x80000000;  //十六进制
    unsigned long p = 100;  //十进制

                                //以无符号的形式输出有符号数
    printf("a=%#ho, b=%#x, c=%ld\n", a, b, c);
    //以有符号数的形式输出无符号类型（只能以十进制形式输出）
    printf("m=%hd, n=%d, p=%ld\n", m, n, p);

    return 0;
}
```

运行结果：

a=0100, b=0xffffffff, c=720

m=-1, n=-2147483648, p=100

对于绝大多数初学者来说，b、m、n 的输出结果看起来非常奇怪，甚至不能理解。按照一般的推理，b、m、n 这三个整数在内存中的存储形式分别是：



当以 %x 输出 b 时，结果应该是 0x80000001；当以 %hd、%d 输出 m、n 时，结果应该分别是 -7fff、-0。但是实际的输出结果和我们推理的结果却大相径庭，这是为什么呢？

> 注意，-7fff 是十六进制形式。%d 本来应该输出十进制，这里只是为了看起来方便，才改为十六进制。

其实这跟整数在内存中的存储形式以及读取方式有关。b 是一个有符号的负数，它在内存中并不是像上图演示的那样存储，而是要经过一定的转换才能写入内存；m、n 的内存虽然没有错误，但是当以 %d 输出时，并不是原样输出，而是有一个逆向的转换过程（和存储时的转换过程恰好相反）。

也就是说，整数在写入内存之前可能会发生转换，在读取时也可能会发生转换，而我们没有考虑这种转换，所以才会导致推理错误。那么，整数在写入内存前，以及在读取时究竟发生了怎样的转换呢？为什么会发生这种转换呢？我们将在《整数在内存中是如何存储的，为什么它堪称天才般的设计》一节中揭开谜底。

3.6 整数在内存中是如何存储的，为什么它堪称天才般的设计

3.7 整数的取值范围以及数值溢出

3.8 C 语言中的小数（float,double）

小数分为整数部分和小数部分，它们由点号`.`分隔，例如 0.0、75.0、4.023、0.27、-937.198 -0.27 等都是合法的小数，这是最常见的小数形式，我们将它称为十进制形式。

此外，小数也可以采用指数形式，例如 7.25×10^{2}、0.0368×10^{5}、100.22×10^{-2}、-27.36×10^{-3} 等。任何小数都可以用指数形式来表示。

C 语言同时支持以上两种形式的小数。但是在书写时，C 语言中的指数形式和数学中的指数形式有所差异。

C 语言中小数的指数形式为：

```
aEn 或 aen
```

a 为尾数部分，是一个十进制数；n 为指数部分，是一个十进制整数；`E`或`e`是固定的字符，用于分割尾数部分和指数部分。整个表达式等价于 $a\times10^{n}$。

指数形式的小数举例：

- 2.1E5 = 2.1×10^{5}，其中 2.1 是尾数，5 是指数。
- 3.7E-2 = 3.7×10^{-2}，其中 3.7 是尾数，-2 是指数。
- 0.5E7 = 0.5×10^{7}，其中 0.5 是尾数，7 是指数。

C 语言中常用的小数有两种类型，分别是 float 或 double；float 称为单精度浮点型，double 称为双精度浮点型。

不像整数，小数没有那么多幺蛾子，小数的长度是固定的，float 始终占用 4 个字节，double 始终占用 8 个字节。

小数的输出

小数也可以使用 printf 函数输出，包括十进制形式和指数形式，它们对应的格式控制符分别是：

- %f 以十进制形式输出 float 类型；
- %lf 以十进制形式输出 double 类型；
- %e 以指数形式输出 float 类型，输出结果中的 e 小写；
- %E 以指数形式输出 float 类型，输出结果中的 E 大写；
- %le 以指数形式输出 double 类型，输出结果中的 e 小写；
- %lE 以指数形式输出 double 类型，输出结果中的 E 大写。

下面的代码演示了小数的表示以及输出：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main()
4.  {
5.      float a = 0.302;
6.      float b = 128.101;
7.      double c = 123;
8.      float d = 112.64E3;
9.      double e = 0.7623e-2;
10.     float f = 1.23002398;
11.     printf("a=%e \nb=%f \nc=%lf \nd=%lE \ne=%lf \nf=%f\n", a, b, c, d, e, f);
12.
13.     return 0;
14. }
```

运行结果：

```
a=3.020000e-01
b=128.100998
c=123.000000
d=1.126400E+05
e=0.007623
f=1.230024
```

对代码的说明：

1) %f 和 %lf 默认保留六位小数，不足六位以 0 补齐，超过六位按四舍五入截断。

2) 将整数赋值给 float 变量时会变成小数。

3) 以指数形式输出小数时，输出结果为科学计数法；也就是说，尾数部分的取值为：0 ≤ 尾数 < 10。

4) b 的输出结果让人费解，才三位小数，为什么不能精确输出，而是输出一个近似值呢？这和小数在内存中的存储形式有关，很多简单的小数压根不能精确存储，所以也就不能精确输出，我们将在下节《小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）》中详细讲解。

另外，小数还有一种更加智能的输出方式，就是使用%g。%g 会对比小数的十进制形式和指数形式，以最短的方式来输出小数，让输出结果更加简练。所谓最短，就是输出结果占用最少的字符。

%g 使用示例：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main()
4.  {
5.      float a = 0.00001;
6.      float b = 30000000;
7.      float c = 12.84;
8.      float d = 1.229338455;
9.      printf("a=%g \nb=%g \nc=%g \nd=%g\n", a, b, c, d);
10.
11.     return 0;
12. }
```

运行结果：

a=1e-05
b=3e+07
c=12.84
d=1.22934

对各个小数的分析：

- a 的十进制形式是 0.00001，占用七个字符的位置，a 的指数形式是 1e-05，占用五个字符的位置，指数形式较短，所以以指数的形式输出。
- b 的十进制形式是 30000000，占用八个字符的位置，b 的指数形式是 3e+07，占用五个字符的位置，指数形式较短，所以以指数的形式输出。
- c 的十进制形式是 12.84，占用五个字符的位置，c 的指数形式是 1.284e+01，占用九个字符的位置，十进制形式较短，所以以十进制的形式输出。
- d 的十进制形式是 1.22934，占用七个字符的位置，d 的指数形式是 1.22934e+00，占用十一个字符的位置，十进制形式较短，所以以十进制的形式输出。

读者需要注意的两点是：

- %g 默认最多保留六位有效数字，包括整数部分和小数部分；%f 和 %e 默认保留六位小数，只包括小数部分。

- %g 不会在最后强加 0 来凑够有效数字的位数，而 %f 和 %e 会在最后强加 0 来凑够小数部分的位数。

总之，%g 要以最短的方式来输出小数，并且小数部分表现很自然，不会强加零，比 %f 和 %e 更有弹性，这在大部分情况下是符合用户习惯的。

除了 %g，还有 %lg、%G、%lG：
- %g 和 %lg 分别用来输出 float 类型和 double 类型，并且当以指数形式输出时，e 小写。
- %G 和 %lG 也分别用来输出 float 类型和 double 类型，只是当以指数形式输出时，E 大写。

数字的后缀

一个数字，是有默认类型的：对于整数，默认是 int 类型；对于小数，默认是 double 类型。

请看下面的例子：

```
1.  long a = 100;
2.  int b = 294;
3.
4.  float x = 52.55;
5.  double y = 18.6;
```

100 和 294 这两个数字默认都是 int 类型的，将 100 赋值给 a，必须先从 int 类型转换为 long 类型，而将 294 赋值给 b 就不用转换了。

52.55 和 18.6 这两个数字默认都是 double 类型的，将 52.55 赋值给 x，必须先从 double 类型转换为 float 类型，而将 18.6 赋值给 y 就不用转换了。

如果不想让数字使用默认的类型，那么可以给数字加上后缀，手动指明类型：
- 在整数后面紧跟 l 或者 L（不区分大小写）表明该数字是 long 类型；
- 在小数后面紧跟 f 或者 F（不区分大小写）表明该数字是 float 类型。

请看下面的代码：

```
1.  long a = 100l;
2.  int b = 294;
3.  short c = 32L;
4.
5.  float x = 52.55f;
6.  double y = 18.6F;
7.  float z = 0.02;
```

加上后缀，虽然数字的类型变了，但这并不意味着该数字只能赋值给指定的类型，它仍然能够赋值给其他的类型，只要进行了一下类型转换就可以了。

对于初学者，很少会用到数字的后缀，加不加往往没有什么区别，也不影响实际编程，但是既然学了 C 语言，还是要知道这个知识点的，万一看到别人的代码这么用了，而你却不明白怎么回事，那就尴尬了。

> 关于数据类型的转换，我们将在《C 语言数据类型转换》一节中深入探讨。

小数和整数相互赋值

在 C 语言中，整数和小数之间可以相互赋值：

- 将一个整数赋值给小数类型，在小数点后面加 0 就可以，加几个都无所谓。
- 将一个小数赋值给整数类型，就得把小数部分丢掉，只能取整数部分，这会改变数字本来的值。注意是直接丢掉小数部分，而不是按照四舍五入取近似值。

请看下面的代码：

```
1.  #include <stdio.h>
2.  int main() {
3.      float f = 251;
4.      int w = 19.427;
5.      int x = 92.78;
6.      int y = 0.52;
7.      int z = -87.27;
8.
9.      printf("f = %f, w = %d, x = %d, y = %d, z = %d\n", f, w, x, y, z);
10.
11.     return 0;
12. }
```

运行结果：

f = 251.000000, w = 19, x = 92, y = 0, z = -87

由于将小数赋值给整数类型时会“失真”，所以编译器一般会给出警告，让大家引起注意。

3.9 小数在内存中是如何存储的，揭秘诺贝尔奖级别的设计（长篇神文）

3.10 在 C 语言中使用英文字符

前面我们多次提到了字符串，字符串是多个字符的集合，它们由" "包围，例如"http://c.biancheng.net"、"C 语言中文网"。字符串中的字符在内存中按照次序、紧挨着排列，整个字符串占用一块连续的内存。

当然，字符串也可以只包含一个字符，例如"A"、"6"；不过为了操作方便，我们一般使用专门的字符类型来处理。

初学者经常用到的字符类型是 char，它的长度是 1，只能容纳 ASCII 码表中的字符，也就是英文字符。

要想处理汉语、日语、韩语等英文之外的字符，就得使用其他的字符类型，char 是做不到的，我们将在下节《在 C 语言中使用中文字符》中详细讲解。

字符的表示

字符类型由单引号''包围，字符串由双引号""包围。

下面的例子演示了如何给 char 类型的变量赋值：

```
1.  //正确的写法
2.  char a = '1';
3.  char b = '$';
4.  char c = 'X';
5.  char d = ' ';  // 空格也是一个字符
6.
7.  //错误的写法
8.  char x = '中';  //char 类型不能包含 ASCII 编码之外的字符
9.  char y = 'A';  //A 是一个全角字符
10. char z = "t";  //字符类型应该由单引号包围
```

说明：在字符集中，全角字符和半角字符对应的编号（或者说编码值）不同，是两个字符；ASCII 编码只定义了半角字符，没有定义全角字符。

字符的输出

输出 char 类型的字符有两种方法，分别是：

- 使用专门的字符输出函数 putchar；
- 使用通用的格式化输出函数 printf，char 对应的格式控制符是%c。

请看下面的演示：

```
1.  #include <stdio.h>
2.  int main() {
3.      char a = '1';
4.      char b = '$';
5.      char c = 'X';
6.      char d = ' ';
7.
8.      //使用 putchar 输出
9.      putchar(a); putchar(d);
10.     putchar(b); putchar(d);
11.     putchar(c); putchar('\n');
12.     //使用 printf 输出
13.     printf("%c %c %c\n", a, b, c);
14.
15.     return 0;
```

```
16.  }
```

运行结果：

```
1 $ X
1 $ X
```

putchar 函数每次只能输出一个字符，输出多个字符需要调用多次。

字符与整数

我们知道，计算机在存储字符时并不是真的要存储字符实体，而是存储该字符在字符集中的编号（也可以叫编码值）。对于 char 类型来说，它实际上存储的就是字符的 ASCII 码。

无论在哪个字符集中，字符编号都是一个整数；从这个角度考虑，字符类型和整数类型本质上没有什么区别。

我们可以给字符类型赋值一个整数，或者以整数的形式输出字符类型。反过来，也可以给整数类型赋值一个字符，或者以字符的形式输出整数类型。

请看下面的例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      char a = 'E';
5.      char b = 70;
6.      int c = 71;
7.      int d = 'H';
8.
9.      printf("a: %c, %d\n", a, a);
10.     printf("b: %c, %d\n", b, b);
11.     printf("c: %c, %d\n", c, c);
12.     printf("d: %c, %d\n", d, d);
13.
14.     return 0;
15. }
```

输出结果：

```
a: E, 69
b: F, 70
c: G, 71
d: H, 72
```

在 ASCII 码表中，字符 'E'、'F'、'G'、'H' 对应的编号分别是 69、70、71、72。

a、b、c、d 实际上存储的都是整数：

- 当给 a、d 赋值一个字符时，字符会先转换成 ASCII 码再存储；
- 当给 b、c 赋值一个整数时，不需要任何转换，直接存储就可以；

- 当以 %c 输出 a、b、c、d 时，会根据 ASCII 码表将整数转换成对应的字符；
- 当以 %d 输出 a、b、c、d 时，不需要任何转换，直接输出就可以。

可以说，是 ASCII 码表将英文字符和整数关联了起来。

再谈字符串

前面我们讲到了字符串的概念，也讲到了字符串的输出，但是还没有讲如何用变量存储一个字符串。其实在 C 语言中没有专门的字符串类型，我们只能使用数组或者指针来间接地存储字符串。

在这里讲字符串很矛盾，虽然我们暂时还没有学到数组和指针，无法从原理上深入分析，但是字符串是常用的，又不得不说一下。所以本节我不会讲解太多，大家只需要死记硬背下面的两种表示形式即可：

```
1.  char str1[] = "http://c.biancheng.net";
2.  char *str2 = "C语言中文网";
```

str1 和 str2 是字符串的名字，后边的[ ]和前边的*是固定的写法。初学者暂时可以认为这两种存储方式是等价的，它们都可以通过专用的 puts 函数和通用的 printf 函数输出。

完整的字符串演示：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      char web_url[] = "http://c.biancheng.net";
5.      char *web_name = "C语言中文网";
6.
7.      puts(web_url);
8.      puts(web_name);
9.      printf("%s\n%s\n", web_url, web_name);
10.
11.     return 0;
12. }
```

3.11 在 C 语言中使用中文字符

3.12 C 语言到底使用什么编码？谁说 C 语言使用 ASCII 码，真是荒谬！

3.13 C 语言转义字符

字符集（Character Set）为每个字符分配了唯一的编号，我们不妨将它称为编码值。在 C 语言中，一个字符除了可以用它的实体（也就是真正的字符）表示，还可以用编码值表示。这种使用编码值来间接地表示字符的方式称为转义字符（Escape Character）。

转义字符以\或者\x开头，以\开头表示后跟八进制形式的编码值，以\x开头表示后跟十六进制形式的编码值。对于转义字符来说，只能使用八进制或者十六进制。

字符 1、2、3、a、b、c 对应的 ASCII 码的八进制形式分别是 61、62、63、141、142、143，十六进制形式分别是 31、32、33、61、62、63。下面的例子演示了转义字符的用法：

```
1.  char a = '\61';  //字符1
2.  char b = '\141';  //字符a
3.  char c = '\x31';  //字符1
4.  char d = '\x61';  //字符a
5.  char *str1 = "\x31\x32\x33\x61\x62\x63";  //字符串"123abc"
6.  char *str2 = "\61\62\63\141\142\143";  //字符串"123abc"
7.  char *str3 = "The string is: \61\62\63\x61\x62\x63"  //混用八进制和十六进制形式
```

转义字符既可以用于单个字符，也可以用于字符串，并且一个字符串中可以同时使用八进制形式和十六进制形式。

一个完整的例子：

```
1.  #include <stdio.h>
2.  int main() {
3.      puts("\x68\164\164\x70://c.biancheng.\x6e\145\x74");
4.      return 0;
5.  }
```

运行结果：

http://c.biancheng.net

转义字符的初衷是用于 ASCII 编码，所以它的取值范围有限：

- 八进制形式的转义字符最多后跟三个数字，也即\ddd，最大取值是\177；
- 十六进制形式的转义字符最多后跟两个数字，也即\xdd，最大取值是\7f。

超出范围的转义字符的行为是未定义的，有的编译器会将编码值直接输出，有的编译器会报错。

对于 ASCII 编码，0~31（十进制）范围内的字符为控制字符，它们都是看不见的，不能在显示器上显示，甚至无法从键盘输入，只能用转义字符的形式来表示。不过，直接使用 ASCII 码记忆不方便，也不容易理解，所以，针对常用的控制字符，C 语言又定义了简写方式，完整的列表如下：

转义字符	意义	ASCII 码值（十进制）
\a	响铃(BEL)	007
\b	退格(BS) ，将当前位置移到前一列	008
\f	换页(FF)，将当前位置移到下页开头	012
\n	换行(LF) ，将当前位置移到下一行开头	010
\r	回车(CR) ，将当前位置移到本行开头	013
\t	水平制表(HT)	009
\v	垂直制表(VT)	011
\'	单引号	039
\"	双引号	034
\\	反斜杠	092

\n 和\t 是最常用的两个转义字符：

- \n 用来换行，让文本从下一行的开头输出，前面的章节中已经多次使用；
- \t 用来占位，一般相当于四个空格，或者 tab 键的功能。

单引号、双引号、反斜杠是特殊的字符，不能直接表示：

- 单引号是字符类型的开头和结尾，要使用\'表示，也即'\'' ；
- 双引号是字符串的开头和结尾，要使用\"表示，也即"abc\"123" ；
- 反斜杠是转义字符的开头，要使用\\表示，也即'\\'，或者"abc\\123"。

转义字符示例：

```
1.  #include <stdio.h>
2.  int main() {
3.      puts("C\tC++\tJava\n\"C\" first appeared!");
4.      return 0;
5.  }
```

运行结果：

C	C++	Java

"C" first appeared!

3.14 C 语言标识符、关键字、注释、表达式和语句

这一节主要讲解 C 语言中的几个基本概念。

标识符

定义变量时，我们使用了诸如 a、abc、mn123 这样的名字，它们都是程序员自己起的，一般能够表达出变量的作用，这叫做标识符（Identifier）。

标识符就是程序员自己起的名字，除了变量名，后面还会讲到函数名、宏名、结构体名等，它们都是标识符。不过，名字也不能随便起，要遵守规范；C 语言规定，标识符只能由字母（A~Z, a~z）、数字（0~9）和下划线（_）组成，并且第一个字符必须是字母或下划线，不能是数字。

以下是合法的标识符：
a, x, x3, BOOK_1, sum5

以下是非法的标识符：
- 3s　不能以数字开头
- s*T　出现非法字符*
- -3x　不能以减号（-）开头
- bowy-1　出现非法字符减号（-）

在使用标识符时还必须注意以下几点：
- C 语言虽然不限制标识符的长度，但是它受到不同编译器的限制，同时也受到操作系统的限制。例如在某个编译器中规定标识符前 128 位有效，当两个标识符前 128 位相同时，则被认为是同一个标识符。
- 在标识符中，大小写是有区别的，例如 BOOK 和 book 是两个不同的标识符。
- 标识符虽然可由程序员随意定义，但标识符是用于标识某个量的符号，因此，命名应尽量有相应的意义，以便于阅读和理解，作到“顾名思义”。

关键字

关键字（Keywords）是由 C 语言规定的具有特定意义的字符串，通常也称为保留字，例如 int、char、long、float、unsigned 等。我们定义的标识符不能与关键字相同，否则会出现错误。

> 你也可以将关键字理解为具有特殊含义的标识符，它们已经被系统使用，我们不能再使用了。

标准 C 语言中一共规定了 32 个关键字，大家可以参考 C 语言关键字及其解释[共 32 个]，后续我们会一一讲解。

注释

注释（Comments）可以出现在代码中的任何位置，用来向用户提示或解释代码的含义。程序编译时，会忽略注释，不做任何处理，就好像它不存在一样。

C 语言支持单行注释和多行注释：
- 单行注释以//开头，直到本行末尾（不能换行）；

- 多行注释以/*开头，以*/结尾，注释内容可以有一行或多行。

一个使用注释的例子：

```
1.  /*
2.    Powered by: c.biancheng.net
3.    Author: 严长生
4.    Date: 2017-10-25
5.  */
6.  #include <stdio.h>
7.  int main()
8.  {
9.      /* puts 会在末尾自动添加换行符 */
10.     puts("http://c.biancheng.net");
11.     printf("C语言中文网\n");  //printf要手动添加换行符
12.     return 0;
13. }
```

运行结果：
http://c.biancheng.net
C 语言中文网

在调试程序的过程中可以将暂时将不使用的语句注释掉，使编译器跳过不作处理，待调试结束后再去掉注释。

需要注意的是，多行注释不能嵌套使用。例如下面的注释是错误的：

```
/*C 语言/*中文*/网*/
```

而下面的注释是正确的：

```
/*C 语言中文网*/  /*c.biancheng.net*/
```

表达式（Expression）和语句（Statement）

其实前面我们已经多次提到了「表达式」和「语句」这两个概念，相信读者在耳濡目染之中也已经略知一二了，本节我们不妨再重点介绍一下。

表达式（Expression）和语句（Statement）的概念在 C 语言中并没有明确的定义：

- 表达式可以看做一个计算的公式，往往由数据、变量、运算符等组成，例如 3*4+5、a=c=d 等，表达式的结果必定是一个值；
- 语句的范围更加广泛，不一定是计算，不一定有值，可以是某个操作、某个函数、选择结构、循环等。

赶紧划重点：

- 表达式必须有一个执行结果，这个结果必须是一个值，例如 3*4+5 的结果 17，a=c=d=10 的结果是 10，printf("hello")的结果是 5（printf 的返回值是成功打印的字符的个数）。
- 以分号;结束的往往称为语句，而不是表达式，例如 3*4+5;、a=c=d;等。

3.15 C 语言加减乘除运算

加减乘除是常见的数学运算，C 语言当然支持，不过，C 语言中的运算符号与数学中的略有不同，请见下表。

	加法	减法	乘法	除法	求余数（取余）
数学	+	-	×	÷	无
C 语言	+	-	*	/	%

C 语言中的加号、减号与数学中的一样，乘号、除号不同；另外 C 语言还多了一个求余数的运算符，就是 %。

下面的代码演示了如何在 C 语言中进行加减乘除运算：

```
#include <stdio.h>
int main()
{
    int a = 12;
    int b = 100;
    float c = 8.5;

    int m = a + b;
    float n = b * c;
    double p = a / c;
    int q = b % a;

    printf("m=%d, n=%f, p=%lf, q=%d\n", m, n, p, q);

    return 0;
}
```

输出结果：

m=112, n=850.000000, p=1.411765, q=4

你也可以让数字直接参与运算：

```
#include <stdio.h>
int main()
{
    int a = 12;
    int b = 100;
    float c = 8.9;

    int m = a - b;  // 变量参与运算
    int n = a + 239;  // 有变量也有数字
    double p = 12.7 * 34.3;  // 数字直接参与运算

```

```
12.     printf("m=%d, n=%d, p=%lf\n", m, n, p);
13.     printf("m*2=%d, 6/3=%d, m*n=%ld\n", m * 2, 6 / 3, m*n);
14.
15.     return 0;
16. }
```

输出结果：
m=-88, n=251, p=435.610000
m*2=-176, 6/3=2, m*n=-22088

对除法的说明

C 语言中的除法运算有点奇怪，不同类型的除数和被除数会导致不同类型的运算结果：

- 当除数和被除数都是整数时，运算结果也是整数；如果不能整除，那么就直接丢掉小数部分，只保留整数部分，这跟将小数赋值给整数类型是一个道理。
- 一旦除数和被除数中有一个是小数，那么运算结果也是小数，并且是 double 类型的小数。

请看下面的代码：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a = 100;
5.      int b = 12;
6.      float c = 12.0;
7.
8.      double p = a / b;
9.      double q = a / c;
10.
11.     printf("p=%lf, q=%lf\n", p, q);
12.
13.     return 0;
14. }
```

运行结果：
p=8.000000, q=8.333333

a 和 b 都是整数，a / b 的结果也是整数，所以赋值给 p 变量的也是一个整数，这个整数就是 8。

另外需要注意的一点是除数不能为 0，因为任何一个数字除以 0 都没有意义。

然而，编译器对这个错误一般无能为力，很多情况下，编译器在编译阶段根本无法计算出除数的值，不能进行有效预测，“除数为 0”这个错误只能等到程序运行后才能发现，而程序一旦在运行阶段出现任何错误，只能有一个结果，那就是崩溃，并被操作系统终止运行。

请看下面的代码：

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

```
2.  int main()
3.  {
4.      int a, b;
5.      scanf("%d %d", &a, &b);  //从控制台读取数据并分别赋值给a和b
6.      printf("result=%d\n", a / b);
7.
8.      return 0;
9.  }
```

这段代码用到了一个新的函数，就是 scanf。scanf 和 printf 的功能相反，printf 用来输出数据，scanf 用来读取数据。此处，scanf 会从控制台读取两个整数，并分别赋值给 a 和 b。关于 scanf 的具体用法，我们将在《C 语言 scanf：读取从键盘输入的数据（含输入格式汇总表）》一节中详细讲解，这里大家只要知道它的作用就可以了，不必求甚解。

程序开头定义了两个 int 类型的变量 a 和 b，程序运行后，从控制台读取用户输入的整数，并分别赋值给 a 和 b，这个时候才能知道 a 和 b 的具体值，才能知道除数 b 是不是 0。像这种情况，b 的值在程序运行期间会改变，跟用户输入的数据有关，编译器根本无法预测，所以就没法及时发现“除数为 0”这个错误。

对取余运算的说明

取余，也就是求余数，使用的运算符是 %。C 语言中的取余运算只能针对整数，也就是说，% 的两边都必须是整数，不能出现小数，否则编译器会报错。

另外，余数可以是正数也可以是负数，由 % 左边的整数决定：

- 如果 % 左边是正数，那么余数也是正数；
- 如果 % 左边是负数，那么余数也是负数。

请看下面的例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      printf(
5.          "100%%12=%d \n100%%-12=%d \n-100%%12=%d \n-100%%-12=%d \n",
6.          100 % 12, 100 % -12, -100 % 12, -100 % -12
7.          );
8.      return 0;
9.  }
```

运行结果：

```
100%12=4
100%-12=4
-100%12=-4
-100%-12=-4
```

在 printf 中，% 是格式控制符的开头，是一个特殊的字符，不能直接输出；要想输出 %，必须在它的前面再加一个 %，

这个时候 % 就变成了普通的字符，而不是用来表示格式控制符了。

加减乘除运算的简写

有时候我们希望对一个变量进行某种运算，然后再把运算结果赋值给变量本身，请看下面的例子：

```
#include <stdio.h>
int main()
{
    int a = 12;
    int b = 10;

    printf("a=%d\n", a);

    a = a + 8;
    printf("a=%d\n", a);

    a = a * b;
    printf("a=%d\n", a);

    return 0;
}
```

输出结果：
a=12
a=20
a=200

a = a + 8 相当于用原来 a 的值（也即 12）加上 8，再把运算结果（也即 20）赋值给 a，此时 a 的值就变成了 20。

a = a * b 相当于用原来 a 的值（也即 20）乘以 b 的值（也即 10），再把运算结果（也即 200）赋值给 a，此时 a 的值就变成了 200。

以上的操作，可以理解为对变量本身进行某种运算。

在 C 语言中，对变量本身进行运算可以有简写形式。假设用 # 来表示某种运算符，那么

```
a = a # b
```

可以简写为：

```
a #= b
```

表示 +、-、*、/、% 中的任何一种运算符。

上例中 a = a + 8 可以简写为 a += 8，a = a * b 可以简写为 a *= b。

下面的简写形式也是正确的：

```
int a = 10, b = 20;
```

```
2.  a += 10;  //相当于 a = a + 10;
3.  a *= (b - 10);  //相当于 a = a * (b-10);
4.  a -= (a + 20);  //相当于 a = a - (a+20);
```

注意：a #= b 仅是一种简写形式，不会影响程序的执行效率。

3.16 C 语言自增(++)和自减(--)运算符

一个整数类型的变量自身加 1 可以这样写：

```
a = a + 1;
```

或者

```
a += 1;
```

不过，C 语言还支持另外一种更加简洁的写法，就是：

```
a++;
```

或者

```
++a;
```

这种写法叫做自加或自增，意思很明确，就是每次自身加 1。

相应的，也有 a--和--a，它们叫做自减，表示自身减 1。

++和--分别称为自增运算符和自减运算符，它们在循环结构（后续章节会讲解）中使用很频繁。

自增和自减的示例：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a = 10, b = 20;
5.      printf("a=%d, b=%d\n", a, b);
6.      ++a;
7.      --b;
8.      printf("a=%d, b=%d\n", a, b);
9.      a++;
10.     b--;
11.     printf("a=%d, b=%d\n", a, b);
12.
13.     return 0;
14. }
```

运行结果：

```
a=10, b=20
a=11, b=19
a=12, b=18
```

自增自减完成后，会用新值替换旧值，将新值保存在当前变量中。

自增自减的结果必须得有变量来接收，所以自增自减只能针对变量，不能针对数字，例如 10++就是错误的。

需要重点说明的是，++ 在变量前面和后面是有区别的：
- ++ 在前面叫做前自增（例如 ++a）。前自增先进行自增运算，再进行其他操作。
- ++ 在后面叫做后自增（例如 a++）。后自增先进行其他操作，再进行自增运算。

自减（--）也一样，有前自减和后自减之分。

下面的例子能更好地说明前自增（前自减）和后自增（后自减）的区别：

```
#include <stdio.h>
int main()
{
    int a = 10, b = 20, c = 30, d = 40;
    int a1 = ++a, b1 = b++, c1 = --c, d1 = d--;

    printf("a=%d, a1=%d\n", a, a1);
    printf("b=%d, b1=%d\n", b, b1);
    printf("c=%d, c1=%d\n", c, c1);
    printf("d=%d, d1=%d\n", d, d1);

    return 0;
}
```

输出结果：
a=11, a1=11
b=21, b1=20
c=29, c1=29
d=39, d1=40

a、b、c、d 的输出结果相信大家没有疑问，下面重点分析 a1、b1、c1、d1：
1) 对于 a1=++a，先执行 ++a，结果为 11，再将 11 赋值给 a1，所以 a1 的最终值为 11。而 a 经过自增，最终的值也为 11。

2) 对于 b1=b++，b 的值并不会立马加 1，而是先把 b 原来的值交给 b1，然后再加 1。b 原来的值为 20，所以 b1 的值也就为 20。而 b 经过自增，最终值为 21。

3) 对于 c1=--c，先执行 --c，结果为 29，再将 29 赋值给 c1，所以 c1 的最终值为 29。而 c 经过自减，最终的值也为 29。

4) 对于 d1=d--，d 的值并不会立马减 1，而是先把 d 原来的值交给 d1，然后再减 1。d 原来的值为 40，所以 d1 的值也就为 40。而 d 经过自减，最终值为 39。

可以看出：a1=++a;会先进行自增操作，再进行赋值操作；而 b1=b++;会先进行赋值操作，再进行自增操作。c1=-

-c;和 d1=d--;也是如此。

为了强化记忆，我们再来看一个自增自减的综合示例：

```
#include <stdio.h>
int main()
{
    int a = 12, b = 1;
    int c = a - (b--);  // ①
    int d = (++a) - (--b);  // ②

    printf("c=%d, d=%d\n", c, d);

    return 0;
}
```

输出结果：
c=11, d=14

我们来分析一下：
1) 执行语句①时，因为是后自减，会先进行 a-b 运算，结果是 11，然后 b 再自减，就变成了 0；最后再将 a-b 的结果（也就是 11）交给 c，所以 c 的值是 11。

2) 执行语句②之前，b 的值已经变成 0。对于 d=(++a)-(--b)，a 会先自增，变成 13，然后 b 再自减，变成 -1，最后再计算 13-(-1)，结果是 14，交给 d，所以 d 最终是 14。

3.17 变量的定义位置以及初始值

3.18 C 语言运算符的优先级和结合性

本节我们从一个例子入手讲解，请看下面的代码：

```
#include <stdio.h>
int main() {
    int a = 16, b = 4, c = 2;
    int d = a + b * c;
    int e = a / b * c;
    printf("d=%d, e=%d\n", d, e);

    return 0;
```

```
9.    }
```

运行结果：
d=24, e=8

1) 对于表达式 a + b * c，如果按照数学规则推导，应该先计算乘法，再计算加法；b * c 的结果为 8，a + 8 的结果为 24，所以 d 最终的值也是 24。从运行结果可以看出，我们的推论得到了证实，C 语言也是先计算乘法再计算加法，和数学中的规则一样。

先计算乘法后计算加法，说明乘法运算符的优先级比加法运算符的优先级高。所谓优先级，就是当多个运算符出现在同一个表达式中时，先执行哪个运算符。

C 语言有几十种运算符，被分成十几个级别，有的运算符优先级不同，有的运算符优先级相同，我们在《C 语言运算符的优先级和结合性一览表》中给出了详细的说明，大家可以点击链接自行查阅。

一下子记住所有运算符的优先级并不容易，还好 C 语言中大部分运算符的优先级和数学中是一样的，大家在以后的编程过程中也会逐渐熟悉起来。如果实在搞不清，可以加括号，就像下面这样：

```
int d = a + (b * c);
```

括号的优先级是最高的，括号中的表达式会优先执行，这样各个运算符的执行顺序就一目了然了。

2) 对于表达式 a / b * c，查看了《C 语言运算符的优先级和结合性一览表》的读者会发现，除法和乘法的优先级是相同的，这个时候到底该先执行哪一个呢？

按照数学规则应该从左到右，先计算除法，在计算乘法；a / b 的结果是 4，4 * c 的结果是 8，所以 e 最终的值也是 8。这个推论也从运行结果中得到了证实，C 语言的规则和数学的规则是一样的。

当乘法和除法的优先级相同时，编译器很明显知道先执行除法，再执行乘法，这是根据运算符的结合性来判定的。所谓结合性，就是当一个表达式中出现多个优先级相同的运算符时，先执行哪个运算符 :先执行左边的叫左结合性，先执行右边的叫右结合性。

/和*的优先级相同，又都具有左结合性，所以先执行左边的除法，再执行右边的乘法。

3) 像 +、-、*、/ 这样的运算符，它的两边都有要计算的数据，每份这样的数据都称作一个操作数，一个运算符需要 n 个操作数就称为 n 目运算符。例如：

- +、-、*、/、= 是双目运算符；
- ++、-- 是单目运算符；
- ?: 是三目运算符（这是 C 语言里唯一的一个三目元算符，后续我们将会讲解）。

总结

当一个表达式中出现多个运算符时，C 语言会先比较各个运算符的优先级，按照优先级从高到低的顺序依次执行；当遇到优先级相同的运算符时，再根据结合性决定先执行哪个运算符：如果是左结合性就先执行左边的运算符，如果是右结合性就先执行右边的运算符。

C 语言的运算符众多，每个运算符都具有优先级和结合性，还拥有若干个操作数，为了方便记忆和对比，我们在《C

语言运算符的优先级和结合性一览表》中将它们全部列了出来。对于没有学到的运算符，大家不必深究，一带而过即可，等学到时再来回顾。

3.19 C 语言数据类型转换（自动转换+强制转换）

数据类型转换就是将数据（变量、数值、表达式的结果等）从一种类型转换为另一种类型。

自动类型转换

自动类型转换就是编译器默默地、隐式地、偷偷地进行的数据类型转换，这种转换不需要程序员干预，会自动发生。

1) 将一种类型的数据赋值给另外一种类型的变量时就会发生自动类型转换，例如：

```
float f = 100;
```

100 是 int 类型的数据，需要先转换为 float 类型才能赋值给变量 f。再如：

```
int n = f;
```

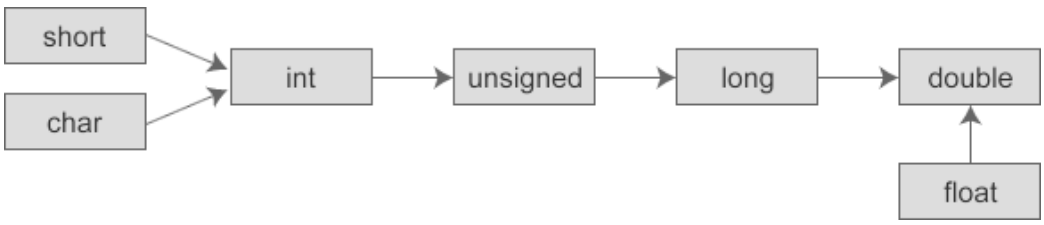
f 是 float 类型的数据，需要先转换为 int 类型才能赋值给变量 n。

在赋值运算中，赋值号两边的数据类型不同时，需要把右边表达式的类型转换为左边变量的类型，这可能会导致数据失真，或者精度降低；所以说，自动类型转换并不一定是安全的。对于不安全的类型转换，编译器一般会给出警告。

2) 在不同类型的混合运算中，编译器也会自动地转换数据类型，将参与运算的所有数据先转换为同一种类型，然后再进行计算。转换的规则如下：

- 转换按数据长度增加的方向进行，以保证数值不失真，或者精度不降低。例如，int 和 long 参与运算时，先把 int 类型的数据转成 long 类型后再进行运算。
- 所有的浮点运算都是以双精度进行的，即使运算中只有 float 类型，也要先转换为 double 类型，才能进行运算。
- char 和 short 参与运算时，必须先转换成 int 类型。

下图对这种转换规则进行了更加形象地描述：



> unsigned 也即 unsigned int，此时可以省略 int，只写 unsigned。

自动类型转换示例：

```
1.  #include<stdio.h>
2.  int main() {
3.      float PI = 3.14159;
```

```
4.      int s1, r = 5;
5.      double s2;
6.      s1 = r * r * PI;
7.      s2 = r * r * PI;
8.      printf("s1=%d, s2=%f\n", s1, s2);
9.
10.     return 0;
11. }
```

运行结果：
s1=78, s2=78.539749

在计算表达式 r*r*PI 时，r 和 PI 都被转换成 double 类型，表达式的结果也是 double 类型。但由于 s1 为整型，所以赋值运算的结果仍为整型，舍去了小数部分，导致数据失真。

强制类型转换

自动类型转换是编译器根据代码的上下文环境自行判断的结果，有时候并不是那么“智能”，不能满足所有的需求。如果需要，程序员也可以自己在代码中明确地提出要进行类型转换，这称为强制类型转换。

自动类型转换是编译器默默地、隐式地进行的一种类型转换，不需要在代码中体现出来；强制类型转换是程序员明确提出的、需要通过特定格式的代码来指明的一种类型转换。换句话说，自动类型转换不需要程序员干预，强制类型转换必须有程序员干预。

强制类型转换的格式为：

```
(type_name) expression
```

type_name 为新类型名称，expression 为表达式。例如：

```
1.  (float)a;  //将变量 a 转换为 float 类型
2.  (int)(x + y);  //把表达式 x+y 的结果转换为 int 整型
3.  (float)100;  //将数值 100（默认为int类型）转换为 float 类型
```

下面是一个需要强制类型转换的经典例子：

```
1.  #include <stdio.h>
2.  int main() {
3.      int sum = 103;  //总数
4.      int count = 7;  //数目
5.      double average;  //平均数
6.      average = (double)sum / count;
7.      printf("Average is %lf!\n", average);
8.
9.      return 0;
10. }
```

运行结果：
Average is 14.714286!

sum 和 count 都是 int 类型，如果不进行干预，那么 sum / count 的运算结果也是 int 类型，小数部分将被丢弃；虽然是 average 是 double 类型，可以接收小数部分，但是心有余力不足，小数部分提前就被“阉割”了，它只能接收到整数部分，这就导致除法运算的结果严重失真。

既然 average 是 double 类型，为何不充分利用，尽量提高运算结果的精度呢？为了达到这个目标，我们只要将 sum 或者 count 其中之一转换为 double 类型即可。上面的代码中，我们将 sum 强制转换为 double 类型，这样 sum / count 的结果也将变成 double 类型，就可以保留小数部分了，average 接收到的值也会更加精确。

在这段代码中，有两点需要注意：

- 对于除法运算，如果除数和被除数都是整数，那么运算结果也是整数，小数部分将被直接丢弃；如果除数和被除数其中有一个是小数，那么运算结果也是小数。这一点已在《C 语言加减乘除运算》中进行了详细说明。
- ( )的优先级高于/，对于表达式(double) sum / count，会先执行(double) sum，将 sum 转换为 double 类型，然后再进行除法运算，这样运算结果也是 double 类型，能够保留小数部分。注意不要写作(double) (sum / count)，这样写运算结果将是 3.000000，仍然不能保留小数部分。

类型转换只是临时性的

无论是自动类型转换还是强制类型转换，都只是为了本次运算而进行的临时性转换，转换的结果也会保存到临时的内存空间，不会改变数据本来的类型或者值。请看下面的例子：

```
1.  #include <stdio.h>
2.  int main() {
3.      double total = 400.8;  //总价
4.      int count = 5;  //数目
5.      double unit;  //单价
6.      int total_int = (int)total;
7.      unit = total / count;
8.      printf("total=%lf, total_int=%d, unit=%lf\n", total, total_int, unit);
9.
10.     return 0;
11. }
```

运行结果：

total=400.800000, total_int=400, unit=80.160000

注意看第 6 行代码，total 变量被转换成了 int 类型才赋值给 total_int 变量，而这种转换并未影响 total 变量本身的类型和值。如果 total 的值变了，那么 total 的输出结果将变为 400.000000；如果 total 的类型变了，那么 unit 的输出结果将变为 80.000000。

自动类型转换 VS 强制类型转换

在 C 语言中，有些类型既可以自动转换，也可以强制转换，例如 int 到 double，float 到 int 等；而有些类型只能强制转换，不能自动转换，例如以后将要学到的 void * 到 int *，int 到 char * 等。

可以自动转换的类型一定能够强制转换，但是，需要强制转换的类型不一定能够自动转换。现在我们学到的数据类型，既可以自动转换，又可以强制转换，以后我们还会学到一些只能强制转换而不能自动转换的类型。

可以自动进行的类型转换一般风险较低，不会对程序带来严重的后果，例如，int 到 double 没有什么缺点，float 到 int 顶多是数值失真。只能强制进行的类型转换一般风险较高，或者行为匪夷所思，例如，char * 到 int * 就是很奇怪的一种转换，这会导致取得的值也很奇怪，再如，int 到 char * 就是风险极高的一种转换，一般会导致程序崩溃。

使用强制类型转换时，程序员自己要意识到潜在的风险。

第 04 章 C 语言输入输出

输入输出（Input and Output, IO）是用户和程序“交流”的过程。在控制台程序中，输出一般是指将数据（包括数字、字符等）显示在屏幕上，输入一般是指获取用户在键盘上输入的数据。

本章将给出常用的 C 语言输入输出函数，并对输入输出的底层知识——缓冲区（缓存）——进行深入讲解，帮助大家破解输入输出过程中的疑难杂症。

本章目录：

1. C 语言数据输出大汇总以及轻量进阶
2. 在屏幕的任意位置输出字符，开发小游戏的第一步
3. 使用 scanf 读取从键盘输入的数据（含输入格式汇总表）
4. C 语言输入字符和字符串（所有函数大汇总）
5. 进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂
6. 结合 C 语言缓冲区谈 scanf 函数，那些奇怪的行为其实都有章可循
7. C 语言清空（刷新）缓冲区，从根本上消除那些奇怪的行为
8. C 语言 scanf 的高级用法，原来 scanf 还有这么多新技能
9. C 语言模拟密码输入（显示星号）
10. C 语言非阻塞式键盘监听，用户不输入数据程序也能继续执行

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

4.1 C 语言数据输出大汇总以及轻量进阶

在 C 语言中，有三个函数可以用来在显示器上输出数据，它们分别是：

- puts()：只能输出字符串，并且输出结束后会自动换行，在《第一个 C 语言程序》中已经进行了介绍。
- putchar()：只能输出单个字符，在《在 C 语言中使用英文字符》中已经进行了介绍。
- printf()：可以输出各种类型的数据，在前面的很多章节中都进行了介绍。

printf() 是最灵活、最复杂、最常用的输出函数，完全可以替代 puts() 和 putchar()，大家一定要掌握。前面的章节中我们已经介绍了 printf() 的基本用法，本节将重点介绍 printf() 的高级用法。

> 对于初学者，这一节的内容可能有些繁杂，如果你希望加快学习进度，尽早写出有趣的代码，也可以跳过这节，后面遇到不懂的 printf() 用法再来回顾。

首先汇总一下前面学到的格式控制符：

格式控制符	说明
%c	输出一个单一的字符
%hd、%d、%ld	以十进制、有符号的形式输出 short、int、long 类型的整数
%hu、%u、%lu	以十进制、无符号的形式输出 short、int、long 类型的整数
%ho、%o、%lo	以八进制、不带前缀、无符号的形式输出 short、int、long 类型的整数
%#ho、%#o、%#lo	以八进制、带前缀、无符号的形式输出 short、int、long 类型的整数
%hx、%x、%lx %hX、%X、%lX	以十六进制、不带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字也小写；如果 X 大写，那么输出的十六进制数字也大写。
%#hx、%#x、%#lx %#hX、%#X、%#lX	以十六进制、带前缀、无符号的形式输出 short、int、long 类型的整数。如果 x 小写，那么输出的十六进制数字和前缀都小写；如果 X 大写，那么输出的十六进制数字和前缀都大写。
%f、%lf	以十进制的形式输出 float、double 类型的小数
%e、%le %E、%lE	以指数的形式输出 float、double 类型的小数。如果 e 小写，那么输出结果中的 e 也小写；如果 E 大写，那么输出结果中的 E 也大写。
%g、%lg %G、%lG	以十进制和指数中较短的形式输出 float、double 类型的小数，并且小数部分的最后不会添加多余的 0。如果 g 小写，那么当以指数形式输出时 e 也小写；如果 G 大写，那么当以指数形式输出时 E 也大写。
%s	输出一个字符串

printf() 的高级用法

通过前面的学习，相信你已经熟悉了 printf() 的基本用法，但是这还不足以把它发挥到极致，printf() 可以有更加炫酷、更加个性、更加整齐的输出形式。

假如现在老师要我们输出一个 4×4 的整数矩阵，为了增强阅读性，数字要对齐，怎么办呢？我们显然可以这样做：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a1 = 20, a2 = 345, a3 = 700, a4 = 22;
5.      int b1 = 56720, b2 = 9999, b3 = 20098, b4 = 2;
6.      int c1 = 233, c2 = 205, c3 = 1, c4 = 6666;
7.      int d1 = 34, d2 = 0, d3 = 23, d4 = 23006783;
8.
9.      printf("%d      %d      %d      %d\n", a1, a2, a3, a4);
10.     printf("%d      %d      %d      %d\n", b1, b2, b3, b4);
11.     printf("%d      %d      %d      %d\n", c1, c2, c3, c4);
12.     printf("%d      %d      %d      %d\n", d1, d2, d3, d4);
13.
```

```
14.     return 0;
15. }
```

运行结果：

```
20      345     700     22
56720   9999    20098   2
233     205     1       6666
34      0       23      23006783
```

> 矩阵一般在大学的《高等数学》中会讲到，m×n 的数字矩阵可以理解为把 m×n 个数字摆放成 m 行 n 列的样子。

看，这是多么地自虐，要敲那么多空格，还要严格控制空格数，否则输出就会错位。更加恶心的是，如果数字的位数变了，空格的数目也要跟着变。例如，当 a1 的值是 20 时，它后面要敲八个空格；当 a1 的值是 1000 时，它后面就要敲六个空格。每次修改整数的值，都要考虑修改空格的数目，逼死强迫症。

类似的需求随处可见，整齐的格式会更加美观，让人觉得生动有趣。其实，我们大可不必像上面一样，printf() 可以更好的控制输出格式。更改上面的代码：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a1 = 20, a2 = 345, a3 = 700, a4 = 22;
5.      int b1 = 56720, b2 = 9999, b3 = 20098, b4 = 2;
6.      int c1 = 233, c2 = 205, c3 = 1, c4 = 6666;
7.      int d1 = 34, d2 = 0, d3 = 23, d4 = 23006783;
8.
9.      printf("%-9d %-9d %-9d %-9d\n", a1, a2, a3, a4);
10.     printf("%-9d %-9d %-9d %-9d\n", b1, b2, b3, b4);
11.     printf("%-9d %-9d %-9d %-9d\n", c1, c2, c3, c4);
12.     printf("%-9d %-9d %-9d %-9d\n", d1, d2, d3, d4);
13.
14.     return 0;
15. }
```

输出结果：

```
20      345     700     22
56720   9999    20098   2
233     205     1       6666
34      0       23      23006783
```

这样写起来更加方便，即使改变某个数字，也无需修改 printf() 语句，增加或者减少空格数目。

`%-9d` 中，`d` 表示以十进制输出，`9` 表示最少占 9 个字符的宽度，宽度不足以空格补齐，`-`表示左对齐。综合起来，`%-9d` 表示以十进制输出，左对齐，宽度最小为 9 个字符。大家可以亲自试试`%9d` 的输出效果。

printf() 格式控制符的完整形式如下：

```
%[flag][width][.precision]type
```

[ ] 表示此处的内容可有可无，是可以省略的。

1) type 表示输出类型，比如 %d、%f、%c、%lf，type 就分别对应 d、f、c、lf；再如，%-9d 中 type 对应 d。

type 这一项必须有，这意味着输出时必须要知道是什么类型。

2) width 表示最小输出宽度，也就是至少占用几个字符的位置；例如，%-9d 中 width 对应 9，表示输出结果最少占用 9 个字符的宽度。

当输出结果的宽度不足 width 时，以空格补齐（如果没有指定对齐方式，默认会在左边补齐空格）；当输出结果的宽度超过 width 时，width 不再起作用，按照数据本身的宽度来输出。

下面的代码演示了 width 的用法：

```
1.  #include <stdio.h>
2.  int main() {
3.      int n = 234;
4.      float f = 9.8;
5.      char c = '@';
6.      char *str = "http://c.biancheng.net";
7.      printf("%10d%12f%4c%8s", n, f, c, str);
8.
9.      return 0;
10. }
```

运行结果：

```
       234    9.800000    @http://c.biancheng.net
```

对输出结果的说明：

- n 的指定输出宽度为 10，234 的宽度为 3，所以前边要补上 7 个空格。
- f 的指定输出宽度为 12，9.800000 的宽度为 8，所以前边要补上 4 个空格。
- str 的指定输出宽度为 8，"http://c.biancheng.net" 的宽度为 22，超过了 8，所以指定输出宽度不再起作用，而是按照 str 的实际宽度输出。

3) .precision 表示输出精度，也就是小数的位数。

- 当小数部分的位数大于 precision 时，会按照四舍五入的原则丢掉多余的数字；
- 当小数部分的位数小于 precision 时，会在后面补 0。

另外，.precision 也可以用于整数和字符串，但是功能却是相反的：

- 用于整数时，.precision 表示最小输出宽度。与 width 不同的是，整数的宽度不足时会在左边补 0，而不是补空格。
- 用于字符串时，.precision 表示最大输出宽度，或者说截取字符串。当字符串的长度大于 precision 时，会截掉

多余的字符；当字符串的长度小于 precision 时，.precision 就不再起作用。

请看下面的例子：

```
#include <stdio.h>
int main() {
    int n = 123456;
    double f = 882.923672;
    char *str = "abcdefghi";
    printf("n: %.9d  %.4d\n", n, n);
    printf("f: %.2lf  %.4lf  %.10lf\n", f, f, f);
    printf("str: %.5s  %.15s\n", str, str);
    return 0;
}
```

运行结果：

```
n: 000123456  123456
f: 882.92  882.9237  882.9236720000
str: abcde  abcdefghi
```

对输出结果的说明：

- 对于 n，.precision 表示最小输出宽度。n 本身的宽度为 6，当 precision 为 9 时，大于 6，要在 n 的前面补 3 个 0；当 precision 为 4 时，小于 6，不再起作用。
- 对于 f，.precision 表示输出精度。f 的小数部分有 6 位数字，当 precision 为 2 或者 4 时，都小于 6，要按照四舍五入的原则截断小数；当 precision 为 10 时，大于 6，要在小数的后面补四个 0。
- 对于 str，.precision 表示最大输出宽度。str 本身的宽度为 9，当 precision 为 5 时，小于 9，要截取 str 的前 5 个字符；当 precision 为 15 时，大于 9，不再起作用。

4) flag 是标志字符。例如，`%#x` 中 flag 对应 #，`%-9d` 中 flags 对应`-`。下表列出了 printf() 可以用的 flag：

标志字符	含 义
-	`-`表示左对齐。如果没有，就按照默认的对齐方式，默认一般为右对齐。
+	用于整数或者小数，表示输出符号（正负号）。如果没有，那么只有负数才会输出符号。
空格	用于整数或者小数，输出值为正时冠以空格，为负时冠以负号。
#	➢ 对于八进制（%o）和十六进制（%x / %X）整数，# 表示在输出时添加前缀；八进制的前缀是 0，十六进制的前缀是 0x / 0X。 ➢ 对于小数（%f / %e / %g），# 表示强迫输出小数点。如果没有小数部分，默认是不输出小数点的，加上 # 以后，即使没有小数部分也会带上小数点。

请看下面的例子：

```
#include <stdio.h>
int main() {
    int m = 192, n = -943;
    float f = 84.342;
```

```
5.      printf("m=%10d, m=%-10d\n", m, m);  //演示 - 的用法
6.      printf("m=%+d, n=%+d\n", m, n);  //演示 + 的用法
7.      printf("m=% d, n=% d\n", m, n);  //演示空格的用法
8.      printf("f=%.0f, f=%#.0f\n", f, f);  //演示#的用法
9.
10.     return 0;
11. }
```

运行结果：

```
m=       192, m=192
m=+192, n=-943
m= 192, n=-943
f=84, f=84.
```

对输出结果的说明：

- 当以%10d 输出 m 时，是右对齐，所以在 192 前面补七个空格；当以%-10d 输出 m 时，是左对齐，所以在 192 后面补七个空格。
- m 是正数，以%+d 输出时要带上正号；n 是负数，以%+d 输出时要带上负号。
- m 是正数，以% d 输出时要在前面加空格；n 是负数，以% d 输出时要在前面加负号。
- %.0f 表示保留 0 位小数，也就是只输出整数部分，不输出小数部分。默认情况下，这种输出形式是不带小数点的，但是如果有了#标志，那么就要在整数的后面“硬加上”一个小数点，以和纯整数区分开。

printf() 不能立即输出的问题

printf() 有一个尴尬的问题，就是有时候不能立即输出，请看下面的代码：

```
1.  #include<stdio.h>
2.  #include<unistd.h>
3.  int main()
4.  {
5.      printf("C语言中文网");
6.      sleep(5);  //程序暂停5秒钟
7.      printf("http://c.biancheng.net\n");
8.
9.      return 0;
10. }
```

这段代码使用了两个 printf() 语句，它们之间有一个 sleep() 函数，该函数的作用是让程序暂停 5 秒，然后再继续执行。sleep() 是 Linux 和 Mac OS 下特有的函数，不能用于 Windows。当然，Windows 下也有功能相同的暂停函数，叫做 Sleep()，稍后我们会讲解。

在 Linux 或者 Mac OS 下运行该程序，会发现第一个 printf() 并没有立即输出，而是等待 5 秒以后，和第二个 printf() 一起输出了，请看下面的动图演示：

> 注意：本节的动态图片在 Word 和 PDF 中不能播放，请保存到本地后再播放。

我们不妨修改一下代码，在第一个 printf() 的最后添加一个换行符，如下所示：

```
printf("C 语言中文网\n");
```

再次编译并运行程序，发现第一个 printf() 首先输出（程序运行后立即输出），等待 5 秒以后，第二个 printf() 才输出，请看下面的动图演示：

为什么一个换行符\n就能让程序的表现有天壤之别呢？按照通常的逻辑，程序运行后第一个 printf() 应该立即输出，而不是等待 5 秒以后再和第二个 printf() 一起输出，也就是说，第二种情形才符合我们的惯性思维。然而，第一种情形该如何理解呢？

其实，这一切都是输出缓冲区（缓存）在作怪！

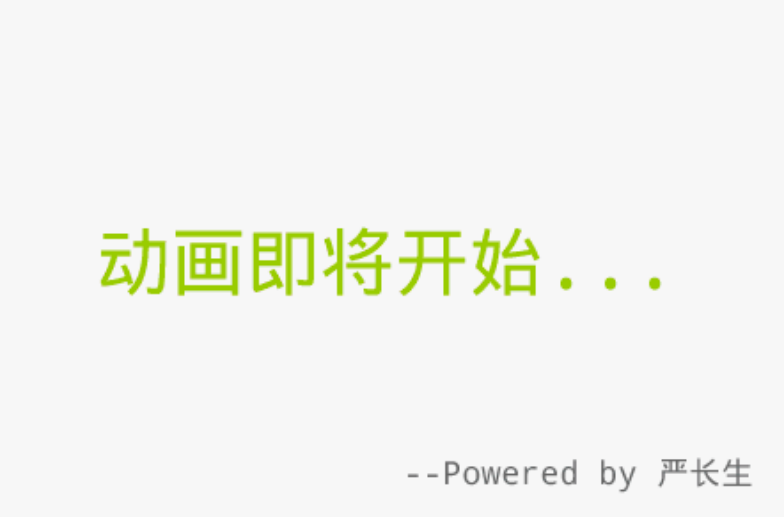
从本质上讲，printf() 执行结束以后数据并没有直接输出到显示器上，而是放入了缓冲区，直到遇见换行符\n才将缓冲区中的数据输出到显示器上。更加深入的内容，我们将在本章的《进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症》中详细讲解。

以上测试的是 Linux 和 Mac OS，我们不妨再测试一下 Windows，请看下面的代码：

```
#include<stdio.h>
#include<Windows.h>
int main()
{
    printf("C语言中文网");
    Sleep(5000);  //程序暂停5秒钟
    printf("http://c.biancheng.net\n");

    return 0;
}
```

在 Windows 下，想让程序暂停可以使用 Windows.h 头文件中的 Sleep() 函数(S 要大写)，它和 Linux 下的 sleep() 功能相同。不过，sleep() 要求的时间单位是秒，而 Sleep() 要求的时间单位是毫秒，1 秒等于 1000 毫秒。这段代码中，我们要求程序暂停 5000 毫秒，也即 5 秒。

编译并运行程序，会发现第一个 printf() 首先输出（程序运行后立即输出），等待 5 秒以后，第二个 printf() 才输出，请看下面的动画演示：



在第一个 printf() 的最后添加一个换行符，情况也是一样的，第一个 printf() 从来不会和第二个 printf() 一起输出。

你看，Windows 和 Linux、Mac OS 的情况又不一样。这是因为，Windows 和 Linux、Mac OS 的缓存机制不同。更加深入的内容，我们将在本章的《进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症》中详细讲解。

要想破解 printf() 输出的问题，必须要了解缓存，它能使你对输入输出的认识上升到一个更高的层次，以后不管遇到什么疑难杂症，都能迎刃而解。可以说，输入输出的“命门”就在于缓存。

总结

对于初学者来说，上面讲到的 printf() 用法已经比较复杂了，基本满足了实际开发的需求，相信大家也需要一段时间才能熟悉。但是，受到所学知识的限制，本文也未能讲解 printf() 的所有功能，后续我们还会逐步深入。

printf() 的这些格式规范不是“小把戏”，优美的输出格式随处可见，例如，dos 下的 dir 命令，会整齐地列出当前

目录下的文件，这明显使用了右对齐，还指定了宽度。

> 再次提醒：本节的动态图片在 Word 和 PDF 中不能播放，请保存到本地后再播放。

4.2 C 语言在屏幕的任意位置输出字符，开发小游戏的第一步

4.3 使用 scanf 读取从键盘输入的数据（含输入格式汇总表）

程序是人机交互的媒介，有输出必然也有输入，第三章我们讲解了如何将数据输出到显示器上，本章我们开始讲解如何从键盘输入数据。在 C 语言中，有多个函数可以从键盘获得用户输入：

- scanf()：和 printf() 类似，scanf() 可以输入多种类型的数据。
- getchar()、getche()、getch()：这三个函数都用于输入单个字符。
- gets()：获取一行数据，并作为字符串处理。

scanf() 是最灵活、最复杂、最常用的输入函数，但它不能完全取代其他函数，大家都要有所了解。

本节我们只讲解 scanf()，其它的输入函数将在下节讲解。

scanf()函数

scanf 是 scan format 的缩写，意思是格式化扫描，也就是从键盘获得用户输入，和 printf 的功能正好相反。

我们先来看一个例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a = 0, b = 0, c = 0, d = 0;
5.      scanf("%d", &a);  //输入整数并赋值给变量a
6.      scanf("%d", &b);  //输入整数并赋值给变量b
```

```
7.      printf("a+b=%d\n", a + b);  //计算a+b的值并输出
8.      scanf("%d %d", &c, &d);  //输入两个整数并分别赋值给c、d
9.      printf("c*d=%d\n", c*d);  //计算c*d的值并输出
10.
11.     return 0;
12. }
```

运行结果：

12↙
60↙
a+b=72
10 23↙
c*d=230

↙表示按下回车键。

从键盘输入 12，按下回车键，scanf() 就会读取输入数据并赋值给变量 a；本次输入结束，接着执行下一个 scanf() 函数，再从键盘输入 60，按下回车键，就会将 60 赋值给变量 b，都是同样的道理。

第 8 行代码中，scanf() 有两个以空格分隔的%d，后面还跟着两个变量，这要求我们一次性输入两个整数，并分别赋值给 c 和 d。注意"%d %d"之间是有空格的，所以输入数据时也要有空格。对于 scanf()，输入数据的格式要和控制字符串的格式保持一致。

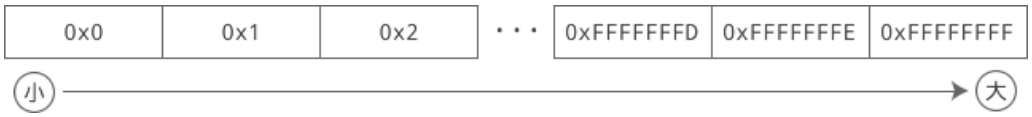
其实 scanf 和 printf 非常相似，只是功能相反罢了：

```
1.  scanf("%d %d", &a, &b);  // 获取用户输入的两个整数，分别赋值给变量 a 和 b
2.  printf("%d %d", a, b);  // 将变量 a 和 b 的值在显示器上输出
```

它们都有格式控制字符串，都有变量列表。不同的是，scanf 的变量前要带一个&符号。&称为取地址符，也就是获取变量在内存中的地址。

在《数据在内存中的存储》一节中讲到，数据是以二进制的形式保存在内存中的，字节（Byte）是最小的可操作单位。为了便于管理，我们给每个字节分配了一个编号，使用该字节时，只要知道编号就可以，就像每个学生都有学号，老师会随机抽取学号来让学生回答问题。字节的编号是有顺序的，从 0 开始，接下来是 1、2、3……

下图是 4G 内存中每个字节的编号（以十六进制表示）：



这个编号，就叫做地址（Address）。int a;会在内存中分配四个字节的空间，我们将第一个字节的地址称为变量 a 的地址，也就是&a 的值。对于前面讲到的整数、浮点数、字符，都要使用 & 获取它们的地址，scanf 会根据地址把读取到的数据写入内存。

我们不妨将变量的地址输出看一下：

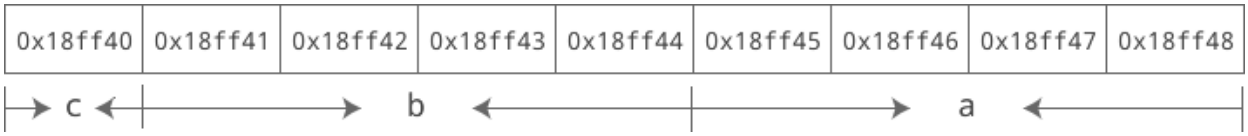
```
1.  #include <stdio.h>
2.  int main()
3.  {
```

```
4.      int a = 'F';
5.      int b = 12;
6.      int c = 452;
7.      printf("&a=%p, &b=%p, &c=%p\n", &a, &b, &c);
8.
9.      return 0;
10. }
```

输出结果：

&a=0x18ff48, &b=0x18ff44, &c=0x18ff40

%p 是一个新的格式控制符，它表示以十六进制的形式（带小写的前缀）输出数据的地址。如果写作%P，那么十六进制的前缀也将变成大写形式。



图：a、b、c 的内存地址

> 注意：这里看到的地址都是假的，是虚拟地址，并不等于数据在物理内存中的地址。虚拟地址是现代计算机因内存管理的需要才提出的概念，我们将在《C 语言内存精讲》专题中详细讲解。

再来看一个 scanf 的例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a, b, c;
5.
6.      scanf("%d %d", &a, &b);
7.      printf("a+b=%d\n", a + b);
8.
9.      scanf("%d   %d", &a, &b);
10.     printf("a+b=%d\n", a + b);
11.
12.     scanf("%d, %d, %d", &a, &b, &c);
13.     printf("a+b+c=%d\n", a + b + c);
14.
15.     scanf("%d is bigger than %d", &a, &b);
16.     printf("a-b=%d\n", a - b);
17.
18.     return 0;
19. }
```

运行结果：

```
10    20↙
a+b=30
```

```
100 200↙
a+b=300
56,45,78↙
a+b+c=179
25 is bigger than 11↙
a-b=14
```

第一个 scanf() 的格式控制字符串为"%d %d"，中间有一个空格，而我们却输入了 10　　20，中间有多个空格。第二个 scanf() 的格式控制字符串为"%d　　%d"，中间有多个空格，而我们却输入了 100 200，中间只有一个空格。这说明 scanf() 对输入数据之间的空格的处理比较宽松，并不要求空格数严格对应，多几个少几个无所谓，只要有空格就行。

第三个 scanf() 的控制字符串为"%d, %d, %d"，中间以逗号分隔，所以输入的整数也要以逗号分隔。

第四个 scanf() 要求整数之间以 is bigger than 分隔。

用户每次按下回车键，程序就会认为完成了一次输入操作，scanf() 开始读取用户输入的内容，并根据格式控制字符串从中提取有效数据，只要用户输入的内容和格式控制字符串匹配，就能够正确提取。

本质上讲，用户输入的内容都是字符串，scanf() 完成的是从字符串中提取有效数据的过程。

连续输入

在本节第一段示例代码中，我们一个一个地输入变量 a、b、c、d 的值，每输入一个值就按一次回车键。现在我们改变输入方式，将四个变量的值一次性输入，如下所示：

```
12 60 10 23↙
a+b=72
c*d=230
```

可以发现，两个 scanf() 都能正确读取。合情合理的猜测是，第一个 scanf() 读取完毕后没有抛弃多余的值，而是将它们保存在了某个地方，下次接着使用。

如果我们多输入一个整数，会怎样呢？

```
12 60 10 23 99↙
a+b=72
c*d=230
```

这次我们多输入了一个 99，发现 scanf() 仍然能够正确读取，只是 99 没用罢了。

如果我们少输入一个整数，又会怎样呢？

```
12 60 10↙
a+b=72
23↙
```

```
c*d=230
```

输入三个整数后，前两个 scanf() 把前两个整数给读取了，剩下一个整数 10，而第三个 scanf() 要求输入两个整数，一个单独的 10 并不能满足要求，所以我们还得继续输入，凑够两个整数以后，第三个 scanf() 才能读取完毕。

从本质上讲，我们从键盘输入的数据并没有直接交给 scanf()，而是放入了缓冲区中，直到我们按下回车键，scanf() 才到缓冲区中读取数据。如果缓冲区中的数据符合 scanf() 的要求，那么就读取结束；如果不符合要求，那么就继续等待用户输入，或者干脆读取失败。我们将在本章的《进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症》《结合 C 语言缓冲区谈 scanf 函数》两节中详细讲解缓冲区。

注意，如果缓冲区中的数据不符合 scanf() 的要求，要么继续等待用户输入，要么就干脆读取失败，上面我们演示了“继续等待用户输入”的情形，下面我们对代码稍作修改，演示“读取失败”的情形。

```
#include <stdio.h>
int main()
{
    int a = 1, b = 2, c = 3, d = 4;  //修改处：给变量赋予不同的初始值
    scanf("%d", &a);
    scanf("%d", &b);
    printf("a=%d, b=%d\n", a, b);
    scanf("%d %d", &c, &d);
    printf("c=%d, d=%d\n", c, d);

    return 0;
}
```

运行结果：

```
12 60 a10↙
a=12, b=60
c=3, d=4
```

前两个整数被正确读取后，剩下了 a10，而第三个 scanf() 要求输入两个十进制的整数，a10 无论如何也不符合要求，所以只能读取失败。输出结果也证明了这一点，c 和 d 的值并没有被改变。

这说明 scanf() 不会跳过不符合要求的数据，遇到不符合要求的数据会读取失败，而不是再继续等待用户输入。

总而言之，正是由于缓冲区的存在，才使得我们能够多输入一些数据，或者一次性输入所有数据，这可以认为是缓冲区的一点优势。然而，缓冲区也带来了一定的负面影响，甚至会导致很奇怪的行为，请看下面的代码：

```
#include <stdio.h>
int main()
{
    int a = 1, b = 2;
    scanf("a=%d", &a);
    scanf("b=%d", &b);
    printf("a=%d, b=%d\n", a, b);

    return 0;
}
```

```
10.  }
```

输入示例：

```
a=99↙
a=99, b=2
```

输入 a=99，按下回车键，程序竟然运行结束了，只有第一个 scanf() 成功读取了数据，第二个 scanf() 仿佛没有执行一样，根本没有给用户任何机会去输入数据。

如果我们换一种输入方式呢？

```
a=99b=200↙
a=99, b=200
```

这样 a 和 b 都能够正确读取了。注意，a=99b=200 中间是没有任何空格的。

肯定有好奇的小伙伴又问了，如果 a=99b=200 两个数据之间有空格又会怎么样呢？我们不妨亲试一下：

```
a=99 b=200↙
a=99, b=2
```

你看，第二个 scanf() 又读取失败了！在前面的例子中，输入的两份数据之前都是有空格的呀，为什么这里不能带空格呢，真是匪夷所思。好吧，这个其实还是跟缓冲区有关系，我将在《结合 C 语言缓冲区谈 scanf()函数》中深入讲解。

要想破解 scanf() 输入的问题，一定要学习缓冲区，它能使你对输入输出的认识上升到一个更高的层次，以后不管遇到什么疑难杂症，都能迎刃而解。可以说，输入输出的“命门”就在于缓冲区。

输入其它数据

除了输入整数，scanf() 还可以输入单个字符、字符串、小数等，请看下面的演示：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      char letter;
5.      int age;
6.      char url[30];
7.      float price;
8.
9.      scanf("%c", &letter);
10.     scanf("%d", &age);
11.     scanf("%s", url); //可以加&也可以不加&
12.     scanf("%f", &price);
13.
14.     printf("26个英文字母的最后一个是 %c。\n", letter);
15.     printf("C语言中文网已经成立%d年了，网址是 %s，开通VIP会员的价格是%g。\n", age, url, price);
16.
17.     return 0;
```

```
18.  }
```

运行示例：

```
z↙
6↙
http://c.biancheng.net↙
159.9↙
26 个英文字母的最后一个是 z。
C 语言中文网已经成立 6 年了，网址是 http://c.biancheng.net，开通 VIP 会员的价格是 159.9。
```

scanf() 和 printf() 虽然功能相反，但是格式控制符是一样的，单个字符、整数、小数、字符串对应的格式控制符分别是 %c、%d、%f、%s。

对读取字符串的说明

在《在 C 语言中使用英文字符》一节中，我们谈到了字符串的两种定义形式，它们分别是：

```
char str1[] = "http://c.biancheng.net";
char *str2 = "C 语言中文网";
```

这两种形式其实是有区别的，第一种形式的字符串所在的内存既有读取权限又有写入权限，第二种形式的字符串所在的内存只有读取权限，没有写入权限。printf()、puts() 等字符串输出函数只要求字符串有读取权限，而 scanf()、gets() 等字符串输入函数要求字符串有写入权限，所以，第一种形式的字符串既可以用于输出函数又可以用于输入函数，而第二种形式的字符串只能用于输出函数。

另外，对于第一种形式的字符串，在[ ]里面要指明字符串的最大长度，如果不指明，也可以根据=后面的字符串来自动推算，此处，就是根据"http://c.biancheng.net"的长度来推算的。但是在前一个例子中，开始我们只是定义了一个字符串，并没有立即给它赋值，所以没法自动推算，只能手动指明最大长度，这也就是为什么一定要写作 char url[30]，而不能写作 char url[]的原因。

读者还要注意第 11 行代码，这行代码用来输入字符串。上面我们说过，scanf() 读取数据时需要的是数据的地址，整数、小数、单个字符都要加&取地址符，这很容易理解；但是对于此处的 url 字符串，我们并没有加 &，这是因为，字符串的名字会自动转换为字符串的地址，所以不用再多此一举加 & 了。当然，你也可以加上，这样虽然不会导致错误，但是编译器会产生警告，至于为什么，我们将会在《数组和指针绝不等价，数组是另外一种类型》《数组到底在什么时候会转换为指针》中讲解。

关于字符串，后续章节我们还会专门讲解，这里只要求大家会模仿，不要彻底理解，也没法彻底理解。

最后需要注意的一点是，scanf() 读取字符串时以空格为分隔，遇到空格就认为当前字符串结束了，所以无法读取含有空格的字符串，请看下面的例子：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      char author[30], lang[30], url[30];
5.      scanf("%s %s", author, lang);
```

```
6.      printf("author:%s \nlang: %s\n", author, lang);
7.      scanf("%s", url);
8.      printf("url: %s\n", url);
9.      return 0;
10. }
```

运行示例：

```
YanChangSheng C-Language↙
author:YanChangSheng
lang: C-Language
http://c.biancheng.net http://biancheng.net↙
url: http://c.biancheng.net
```

对于第一个 scanf()，它将空格前边的字符串赋值给 author，将空格后边的字符串赋值给 lang；很显然，第一个字符串遇到空格就结束了，第二个字符串到了本行的末尾结束了。

或许第二个 scanf() 更能说明问题，我们输入了两个网址，但是 scanf() 只读取了一个，就是因为这两个网址以空格为分隔，scanf() 遇到空格就认为字符串结束了，不再继续读取了。

scanf() 格式控制符汇总

格式控制符	说明
%c	读取一个单一的字符
%hd、%d、%ld	读取一个十进制整数，并分别赋值给 short、int、long 类型
%ho、%o、%lo	读取一个八进制整数（可带前缀也可不带），并分别赋值给 short、int、long 类型
%hx、%x、%lx	读取一个十六进制整数（可带前缀也可不带），并分别赋值给 short、int、long 类型
%hu、%u、%lu	读取一个无符号整数，并分别赋值给 unsigned short、unsigned int、unsigned long 类型
%f、%lf	读取一个十进制形式的小数，并分别赋值给 float、double 类型
%e、%le	读取一个指数形式的小数，并分别赋值给 float、double 类型
%g、%lg	既可以读取一个十进制形式的小数，也可以读取一个指数形式的小数，并分别赋值给 float、double 类型
%s	读取一个字符串（以空白符为结束）

4.4 C 语言输入字符和字符串（所有函数大汇总）

C 语言有多个函数可以从键盘获得用户输入，它们分别是：

- scanf()：和 printf() 类似，scanf() 可以输入多种类型的数据。
- getchar()、getche()、getch()：这三个函数都用于输入单个字符。

- gets()：获取一行数据，并作为字符串处理。

scanf() 是最灵活、最复杂、最常用的输入函数，上节我们已经进行了讲解，本节接着讲解剩下的函数，也就是字符输入函数和字符串输入函数。

输入单个字符

输入单个字符当然可以使用 scanf() 这个通用的输入函数，对应的格式控制符为%c，上节已经讲到了。本节我们重点讲解的是 getchar()、getche() 和 getch() 这三个专用的字符输入函数，它们具有某些 scanf() 没有的特性，是 scanf() 不能代替的。

1) getchar()

最容易理解的字符输入函数是 getchar()，它就是 scanf("%c", c)的替代品，除了更加简洁，没有其它优势了；或者说，getchar() 就是 scanf() 的一个简化版本。

下面的代码演示了 getchar() 的用法：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      char c;
5.      c = getchar();
6.      printf("c: %c\n", c);
7.
8.      return 0;
9.  }
```

输入示例：

```
@↙
c: @
```

你也可以将第 4、5 行的语句合并为一个，从而写作：

```
char c = getchar();
```

2) getche()

getche() 就比较有意思了，它没有缓冲区，输入一个字符后会立即读取，不用等待用户按下回车键，这是它和 scanf()、getchar() 的最大区别。请看下面的代码：

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main()
4.  {
5.      char c = getche();
6.      printf("c: %c\n", c);
7.
8.      return 0;
```

```
9.  }
```

输入示例：

```
@c: @
```

输入@后，getche() 立即读取完毕，接着继续执行 printf() 将字符输出，所以没有按下回车键程序就运行结束了。

注意，getche() 位于 conio.h 头文件中，而这个头文件是 Windows 特有的，Linux 和 Mac OS 下没有包含该头文件。换句话说，getche() 并不是标准函数，默认只能在 Windows 下使用，不能在 Linux 和 Mac OS 下使用。

3) getch()

getch() 也没有缓冲区，输入一个字符后会立即读取，不用按下回车键，这一点和 getche() 相同。getch() 的特别之处是它没有回显，看不到输入的字符。所谓回显，就是在控制台上显示出用户输入的字符；没有回显，就不会显示用户输入的字符，就好像根本没有输入一样。

回显在大部分情况下是有必要的，它能够与用户及时交互，让用户清楚地看到自己输入的内容。但在某些特殊情况下，我们却不希望有回显，例如输入密码，有回显是非常危险的，容易被偷窥。

getch() 使用举例：

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  int main()
4.  {
5.      char c = getch();
6.      printf("c: %c\n", c);
7.
8.      return 0;
9.  }
```

输入@后，getch() 会立即读取完毕，接着继续执行 printf() 将字符输出。但是由于 getch() 没有回显，看不到输入的@字符，所以控制台上最终显示的内容为 c: @。

注意，和 getche() 一样，getch() 也位于 conio.h 头文件中，也不是标准函数，默认只能在 Windows 下使用，不能在 Linux 和 Mac OS 下使用。

对三个函数的总结

函数	缓冲区	头文件	回显	适用平台
getchar()	有	stdio.h	有	Windows、Linux、Mac OS 等所有平台
getche()	无	conio.h	有	Windows
getch()	无	conio.h	无	Windows

关于缓冲区，我们将在下节《进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症》中展开讲解。

输入字符串

输入字符串当然可以使用 scanf() 这个通用的输入函数，对应的格式控制符为%s，上节已经讲到了；本节我们重点讲解的是 gets() 这个专用的字符串输入函数，它拥有一个 scanf() 不具备的特性。

gets() 的使用也很简单，请看下面的代码：

```
#include <stdio.h>
int main()
{
    char author[30], lang[30], url[30];
    gets(author);
    printf("author: %s\n", author);
    gets(lang);
    printf("lang: %s\n", lang);
    gets(url);
    printf("url: %s\n", url);

    return 0;
}
```

运行结果：

```
YanChangSheng↙
author: YanChangSheng
C-Language↙
lang: C-Language
http://c.biancheng.net http://biancheng.net↙
url: http://c.biancheng.net http://biancheng.net
```

gets() 是有缓冲区的，每次按下回车键，就代表当前输入结束了，gets() 开始从缓冲区中读取内容，这一点和 scanf() 是一样的。gets() 和 scanf() 的主要区别是：

- scanf() 读取字符串时以空格为分隔，遇到空格就认为当前字符串结束了，所以无法读取含有空格的字符串。
- gets() 认为空格也是字符串的一部分，只有遇到回车键时才认为字符串输入结束，所以，不管输入了多少个空格，只要不按下回车键，对 gets() 来说就是一个完整的字符串。

也就是说，gets() 能读取含有空格的字符串，而 scanf() 不能。

总结

C 语言中常用的从控制台读取数据的函数有五个，它们分别是 scanf()、getchar()、getche()、getch() 和 gets()。其中 scanf()、getchar()、gets() 是标准函数，适用于所有平台；getche() 和 getch() 不是标准函数，只能用于 Windows。

scanf() 是通用的输入函数，它可以读取多种类型的数据。

getchar()、getche() 和 getch() 是专用的字符输入函数，它们在缓冲区和回显方面与 scanf() 有着不同的特性，是 scanf() 不能替代的。

gets() 是专用的字符串输入函数，与 scanf() 相比，gets() 的主要优势是可以读取含有空格的字符串。

scanf() 可以一次性读取多份类型相同或者不同的数据，getchar()、getche()、getch() 和 gets() 每次只能读取一份特定类型的数据，不能一次性读取多份数据。

4.5 进入缓冲区（缓存）的世界，破解一切与输入输出有关的疑难杂症

4.6 结合 C 语言缓冲区谈 scanf 函数，那些奇怪的行为其实都有章可循

4.7 C 语言清空（刷新）缓冲区，从根本上消除那些奇怪的行为

4.8 C 语言 scanf 的高级用法，原来 scanf 还有这么多新技能

4.9 C 语言模拟密码输入（显示星号）

4.10 C 语言非阻塞式键盘监听，用户不输入数据程序也能继续执行

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

第 05 章 循环结构和选择结构

C 语言中有三大结构，分别是顺序结构、选择结构和循环结构（分支结构）：

- C 语言顺序结构就是让程序按照从头到尾的顺序依次执行每一条 C 语言代码，不重复执行任何代码，也不跳过任何代码。
- C 语言选择结构也称分支结构，就是让程序“拐弯”，有选择性的执行代码；换句话说，可以跳过没用的代码，只执行有用的代码。
- C 语言循环结构就是让程序“杀个回马枪”，不断地重复执行同一段代码。

顺序结构很好理解，无需多说，本章重点讲解选择结构和循环结构。

本章目录：

1. C 语言 if else 语句详解
2. C 语言关系运算符详解
3. C 语言逻辑运算符详解
4. C 语言 switch case 语句详解
5. C 语言条件运算符（? :）详解
6. C 语言 while 循环和 do while 循环详解
7. C 语言 for 循环（for 语句）详解
8. C 语言跳出循环（break 和 continue）
9. C 语言循环嵌套详解
10. 对选择结构和循环结构的总结
11. 谈编程思维的培养，初学者如何实现自我突破（非常重要）
12. 用 C 语言写一个内存泄露的例子，让计算机内存爆满

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

5.1 C 语言 if else 语句详解

前面我们看到的代码都是顺序执行的，也就是先执行第一条语句，然后是第二条、第三条……一直到最后一条语句，这称为顺序结构。

但是对于很多情况，顺序结构的代码是远远不够的，比如一个程序限制了只能成年人使用，儿童因为年龄不够，没有权限使用。这时候程序就需要做出判断，看用户是否是成年人，并给出提示。

在 C 语言中，使用 if 和 else 关键字对条件进行判断。请先看下面的代码：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int age;
5.      printf("请输入你的年龄：");
6.      scanf("%d", &age);
7.      if (age >= 18) {
8.          printf("恭喜，你已经成年，可以使用该软件！\n");
9.      }
10.     else {
11.         printf("抱歉，你还未成年，不宜使用该软件！\n");
12.     }
13.     return 0;
14. }
```

可能的运行结果：

```
请输入你的年龄：23↙
恭喜，你已经成年，可以使用该软件！
```

或者：

```
请输入你的年龄：16
抱歉，你还未成年，不宜使用该软件！
```

这段代码中，age>=18 是需要判断的条件，>=表示“大于等于”，等价于数学中的≥。

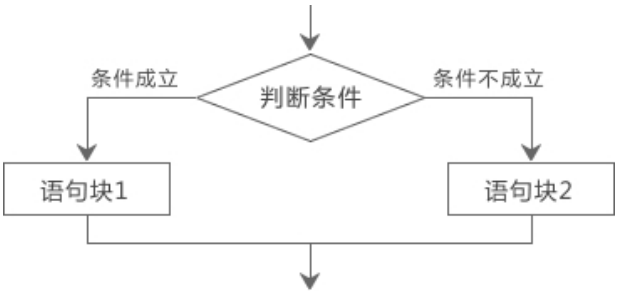
如果条件成立，也即 age 大于或者等于 18，那么执行 if 后面的语句（第 8 行）；如果条件不成立，也即 age 小于 18，那么执行 else 后面的语句（第 10 行）。

if 和 else 是两个新的关键字，if 意为“如果”，else 意为“否则”，用来对条件进行判断，并根据判断结果执行不同的语句。总结起来，if else 的结构为：

```
if(判断条件){
    语句块 1
}else{
    语句块 2
```

```
}
```

意思是，如果判断条件成立，那么执行语句块 1，否则执行语句块 2 。其执行过程可表示为下图：



所谓语句块（Statement Block），就是由{}包围的一个或多个语句的集合。如果语句块中只有一个语句，也可以省略{}，例如：

```
1.  if (age >= 18) printf("恭喜，你已经成年，可以使用该软件！\n");
2.  else printf("抱歉，你还未成年，不宜使用该软件！\n");
```

由于 if else 语句可以根据不同的情况执行不同的代码，所以也叫分支结构或选择结构，上面的代码中，就有两个分支。

求两个数中的较大值：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a, b, max;
5.      printf("输入两个整数：");
6.      scanf("%d %d", &a, &b);
7.      if (a>b) max = a;
8.      else max = b;
9.      printf("%d和%d的较大值是：%d\n", a, b, max);
10.     return 0;
11. }
```

运行结果：
输入两个整数：34 28↙
34 和 28 的较大值是：34

本例中借助变量 max，用 max 来保存较大的值，最后将 max 输出。

只使用 if 语句

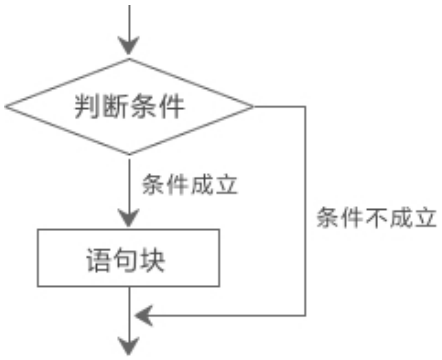
有的时候，我们需要在满足某种条件时进行一些操作，而不满足条件时就不进行任何操作，这个时候我们可以只使用 if 语句。也就是说，if else 不必同时出现。

单独使用 if 语句的形式为：

```
if(判断条件){
```

```
    语句块
}
```

意思是，如果判断条件成立就执行语句块，否则直接跳过。其执行过程可表示为下图：



只使用 if 语句来求两个数中的较大值：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int a, b, max;
5.      printf("输入两个整数：");
6.      scanf("%d %d", &a, &b);
7.      max = b;  // 假设b最大
8.      if (a>b) max = a;  // 如果a>b，那么更改max的值
9.      printf("%d和%d的较大值是：%d\n", a, b, max);
10.     return 0;
11. }
```

运行结果：
输入两个整数：34 28
34 和 28 的较大值是：34

本例程序中，输入两个数 a、b。把 b 先赋予变量 max，再用 if 语句判别 max 和 b 的大小，如 max 小于 b，则把 b 赋予 max。因此 max 中总是大数，最后输出 max 的值。

多个 if else 语句

if else 语句也可以多个同时使用，构成多个分支，形式如下：

```
if(判断条件 1){
    语句块 1
} else   if(判断条件 2){
    语句块 2
}else   if(判断条件 3){
    语句块 3
}else   if(判断条件 m){
```

```
    语句块 m
}else{
    语句块 n
}
```

意思是，从上到下依次检测判断条件，当某个判断条件成立时，则执行其对应的语句块，然后跳到整个 if else 语句之外继续执行其他代码。如果所有判断条件都不成立，则执行语句块 n，然后继续执行后续代码。

也就是说，一旦遇到能够成立的判断条件，则不再执行其他的语句块，所以最终只能有一个语句块被执行。

例如，使用多个 if else 语句判断输入的字符的类别：

```
#include <stdio.h>
int main() {
    char c;
    printf("Input a character:");
    c = getchar();
    if (c<32)
        printf("This is a control character\n");
    else if (c >= '0'&&c <= '9')
        printf("This is a digit\n");
    else if (c >= 'A'&&c <= 'Z')
        printf("This is a capital letter\n");
    else if (c >= 'a'&&c <= 'z')
        printf("This is a small letter\n");
    else
        printf("This is an other character\n");
    return 0;
}
```

运行结果：

Input a character:e↙

This is a small letter

本例要求判别键盘输入字符的类别。可以根据输入字符的 ASCII 码来判别类型。由 ASCII 码表可知 ASCII 值小于 32 的为控制字符。在“0”和“9”之间的为数字，在“A”和“Z”之间为大写字母，在“a”和“z”之间为小写字母，其余则为其它字符。这是一个多分支选择的问题，用多个 if else 语句编程，判断输入字符 ASCII 码所在的范围，分别给出不同的输出。例如输入为“e”，输出显示它为小写字符。

在使用 if 语句时还应注意以下两点：

- 在 if 语句中，判断条件必须用括号括起来。
- 语句块由{}包围，但要注意的是在}之后不需要再加分号;（当然加上也没错）。

if 语句的嵌套

if 语句也可以嵌套使用，例如：

```
#include <stdio.h>
int main() {
    int a, b;
    printf("Input two numbers:");
    scanf("%d %d", &a, &b);
    if (a != b) {  //!=表示不等于
        if (a>b) printf("a>b\n");
        else printf("a<b\n");
    }
    else {
        printf("a=b\n");
    }
    return 0;
}
```

运行结果：

Input two numbers:12 68

a<b

if 语句嵌套时，要注意 if 和 else 的配对问题。C 语言规定，else 总是与它前面最近的 if 配对，例如：

```
if (a != b)  // ①
if (a>b) printf("a>b\n");  // ②
else printf("a<b\n");  // ③
```

③和②配对，而不是和①配对。

5.2 C 语言关系运算符详解

在上节《C 语言 if else 语句》中看到，if 的判断条件中使用了<=、>、!=等符号，它们专门用在判断条件中，让程序决定下一步的操作，称为关系运算符（Relational Operators）。

关系运算符在使用时，它的的两边都会有一个表达式，比如变量、数值、加减乘除运算等，关系运算符的作用就是判明这两个表达式的大小关系。注意，是判明大小关系，不是其他关系。

C 语言提供了以下关系运算符：

关系运算符	含　义	数学中的表示
<	小于	<
<=	小于或等于	≤
>	大于	>
>=	大于或等于	≥
==	等于	=

!=	不等于	≠

关系运算符都是双目运算符，其结合性均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。在六个关系运算符中，<、<=、>、>=的优先级相同，高于==和!=，==和!=的优先级相同。

在 C 语言中，有的运算符有两个操作数，例如 10+20，10 和 20 都是操作数，+ 是运算符。我们将这样的运算符称为双目运算符。同理，将有一个操作数的运算符称为单目运算符，将有三个操作数的运算符称为三目运算符。

常见的双目运算符有 +、-、*、/ 等，单目运算符有 ++、-- 等，三目运算符只有一个，就是 ?:，我们将在《C 语言条件运算符》中详细介绍。

关系运算符的两边可以是变量、数据或表达式，例如：
1) a+b > c-d
2) x > 3/2
3) 'a'+1 < c
4) -i-5*j == k+1

关系运算符也可以嵌套使用，例如：
1) a > (b > c)
2) a != (c == d)

关系运算符的运算结果只有 0 或 1。当条件成立时结果为 1，条件不成立结果为 0。例如：
- 5>0 成立，其值为 1；
- 34-12>100 不成立，其值为 0；
- (a=3)>(b=5) 由于 3>5 不成立，故其值为 0。

我们将运算结果 1 称为“真”，表示条件成立，将 0 称为“假”，表示条件不成立。

下面的代码会将关系运算符的结果输出：

```
1.  #include <stdio.h>
2.  int main() {
3.      char c = 'k';
4.      int i = 1, j = 2, k = 3;
5.      float x = 3e+5, y = 0.85;
6.      int result_1 = 'a' + 5<c, result_2 = x - 5.25 <= x + y;
7.      printf("%d, %d\n", result_1, -i - 2 * j >= k + 1);
8.      printf("%d, %d\n", 1<j<5, result_2);
9.      printf("%d, %d\n", i + j + k == -2 * j, k == j == i + 5);
10.     return 0;
11. }
```

运行结果：
1, 0
1, 1
0, 0

对于含多个关系运算符的表达式，如 k==j==i+5，根据运算符的左结合性，先计算 k==j，该式不成立，其值为 0，再计算 0==i+5，也不成立，故表达式值为 0。

需要提醒的是，==才表示等于，而=表示赋值，大家要注意区分，切勿混淆。

再谈 if 语句的判断条件

if 语句的判断条件中不是必须要包含关系运算符，它可以是赋值表达式，甚至也可以是一个变量，例如：

```
1.  //情况①
2.  if (b) {
3.      //TODO:
4.  }
5.  //情况②
6.  if (b = 5) {  //情况①
7.      //TODO:
8.  }
```

都是允许的。只要整个表达式的值为非 0，条件就成立。

上面两种情况都是根据变量 b 的最终值来判断的，如果 b 的值为非 0，那么条件成立，否则不成立。

又如，有程序段：

```
1.  if (a = b)
2.  printf("%d", a);
3.  else
4.  printf("a=0");
```

意思是，把 b 的值赋予 a，如果为非 0 则输出该值，否则输出 “a=0” 字符串。这种用法在后面的程序中会经常出现。

5.3 C 语言逻辑运算符详解

现在假设有这样一种情况，我们的软件比较特殊，要求使用者必须成年，并且成绩大于等于 60，该怎么办呢？

或许你会想到使用嵌套的 if 语句，类似下面这样的代码：

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int age;
5.      float score;
6.      printf("请输入你的年龄和成绩：");
7.      scanf("%d %f", &age, &score);
8.      if (age >= 18) {
9.          if (score >= 60) {
```

```
            printf("你满足条件，欢迎使用该软件\n");
        }
        else {
            printf("抱歉，你的成绩不及格，不能使用该软件\n");
        }
    }
    else {
        printf("抱歉，你还未成年，不能使用该软件！\n");
    }
    return 0;
}
```

这种方法虽然能够行得通，但不够简洁和专业，我们可以将其压缩为一条 if else 语句：

```
#include <stdio.h>
int main()
{
    int age;
    float score;
    printf("请输入你的年龄和成绩：");
    scanf("%d %f", &age, &score);
    if (age >= 18 && score >= 60) {
        printf("你满足条件，欢迎使用该软件\n");
    }
    else {
        printf("抱歉，你还未成年，或者成绩不及格，不能使用该软件！\n");
    }
    return 0;
}
```

&&是一个新的运算符，称为逻辑运算符，表示 age>=18 和 score>=60 两个条件必须同时成立才能执行 if 后面的代码，否则就执行 else 后面的代码。

在高中数学中，我们就学过逻辑运算，例如 p 为真命题，q 为假命题，那么“p 且 q”为假，“p 或 q”为真，“非 q”为真。在 C 语言中，也有类似的逻辑运算：

运算符	说明	结合性	举例
&&	与运算，双目，对应数学中的“且”	左结合	1&&0、(9>3)&&(b>a)
||	或运算，双目，对应数学中的“或”	左结合	1||0、(9>3)||(b>a)
!	非运算，单目，对应数学中的“非”	右结合	!a、!(2<5)

逻辑运算的结果

在编程中，我们一般将零值称为“假”，将非零值称为“真”。逻辑运算的结果也只有“真”和“假”，“真”对应的值为 1，“假”对应的值为 0。

1) 与运算(&&)

参与运算的两个表达式都为真时，结果才为真，否则为假。例如：

```
5&&0
```

5 为真，0 为假，相与的结果为假，也就是 0。

```
(5>0) && (4>2)
```

5>0 的结果是 1，为真，4>2 结果是 1，也为真，所以相与的结果为真，也就是 1。

2) 或运算(||)

参与运算的两个表达式只要有一个为真，结果就为真；两个表达式都为假时结果才为假。例如：

```
10 || 0
```

10 为真，0 为假，相或的结果为真，也就是 1。

```
(5>0) || (5>8)
```

5>0 的结果是 1，为真，5>8 的结果是 0，为假，所以相或的结果为真，也就是 1。

3) 非运算(!)

参与运算的表达式为真时，结果为假；参与运算的表达式为假时，结果为真。例如：

```
!0
```

0 为假，非运算的结果为真，也就是 1。

```
!(5>0)
```

5>0 的结果是 1，为真，非运算的结果为假，也就是 0。

输出逻辑运算的结果：

```
#include <stdio.h>
int main() {
    int a = 0, b = 10, c = -6;
    int result_1 = a&&b, result_2 = c || 0;
    printf("%d, %d\n", result_1, !c);
    printf("%d, %d\n", 9 && 0, result_2);
    printf("%d, %d\n", b || 100, 0 && 0);
    return 0;
}
```

运行结果：

```
0, 0
0, 1
1, 0
```

优先级

逻辑运算符和其它运算符优先级从低到高依次为：

赋值运算符(=) < &&和|| < 关系运算符 < 算术运算符 < 非(!)

&& 和 || 低于关系运算符，！高于算术运算符。

按照运算符的优先顺序可以得出：

- a>b && c>d 等价于 (a>b)&&(c>d)
- !b==c||d<a 等价于 ((!b)==c)||(d<a)
- a+b>c&&x+y<b 等价于 ((a+b)>c)&&((x+y)<b)

另外，逻辑表达式也可以嵌套使用，例如 a>b && b || 9>c，a || c>d && !p。

逻辑运算符举例：

```
#include <stdio.h>
int main() {
    char c = 'k';
    int i = 1, j = 2, k = 3;
    float x = 3e+5, y = 0.85;
    printf("%d,%d\n", !x*!y, !!!x);
    printf("%d,%d\n", x || i&&j - 3, i<j&&x<y);
    printf("%d,%d\n", i == 5 && c && (j = 8), x + y || i + j + k);
    return 0;
}
```

运行结果：

0,0
1,0
0,1

本例中!x 和!y 分别为 0，!x*!y 也为 0，故其输出值为 0。由于 x 为非 0，故!!!x 的逻辑值为 0。对 x|| i && j-3 式，先计算 j-3 的值为非 0，再求 i && j-3 的逻辑值为 1，故 x||i&&j-3 的逻辑值为 1。对 i<j&&x<y 式，由于 i<j 的值为 1，而 x<y 为 0 故表达式的值为 1，0 相与，最后为 0，对 i==5&&c&&(j=8)式，由于 i==5 为假，即值为 0，该表达式由两个与运算组成，所以整个表达式的值为 0。对于式 x+ y||i+j+k 由于 x+y 的值为非 0，故整个或表达式的值为 1。

5.4 C 语言 switch case 语句详解

C 语言虽然没有限制 if else 能够处理的分支数量，但当分支过多时，用 if else 处理会不太方便，而且容易出现 if else 配对出错的情况。例如，输入一个整数，输出该整数对应的星期几的英文表示：

```
#include <stdio.h>
int main() {
    int a;
    printf("Input integer number:");
    scanf("%d", &a);
    if (a == 1) {
        printf("Monday\n");
```

```
    }
    else if (a == 2) {
        printf("Tuesday\n");
    }
    else if (a == 3) {
        printf("Wednesday\n");
    }
    else if (a == 4) {
        printf("Thursday\n");
    }
    else if (a == 5) {
        printf("Friday\n");
    }
    else if (a == 6) {
        printf("Saturday\n");
    }
    else if (a == 7) {
        printf("Sunday\n");
    }
    else {
        printf("error\n");
    }
    return 0;
}
```

运行结果：

Input integer number:3↙

Wednesday

对于这种情况，实际开发中一般使用 switch 语句代替，请看下面的代码：

```
#include <stdio.h>
int main() {
    int a;
    printf("Input integer number:");
    scanf("%d", &a);
    switch (a) {
    case 1: printf("Monday\n"); break;
    case 2: printf("Tuesday\n"); break;
    case 3: printf("Wednesday\n"); break;
    case 4: printf("Thursday\n"); break;
    case 5: printf("Friday\n"); break;
    case 6: printf("Saturday\n"); break;
    case 7: printf("Sunday\n"); break;
    default:printf("error\n"); break;
    }
```

```
16.       return 0;
17.   }
```

运行结果：

Input integer number:4↙

Thursday

switch 是另外一种选择结构的语句，用来代替简单的、拥有多个分枝的 if else 语句，基本格式如下：

```
switch(表达式){
    case 整型数值 1: 语句 1;
    case 整型数值 2: 语句 2;
    ……
    case 整型数值 n: 语句 n;
    default: 语句 n+1;
}
```

它的执行过程是：

1) 首先计算“表达式”的值，假设为 m。

2) 从第一个 case 开始，比较“整型数值 1”和 m，如果它们相等，就执行冒号后面的所有语句，也就是从“语句 1”一直执行到“语句 n+1”，而不管后面的 case 是否匹配成功。

3) 如果“整型数值 1”和 m 不相等，就跳过冒号后面的“语句 1”，继续比较第二个 case、第三个 case……一旦发现和某个整型数值相等了，就会执行后面所有的语句。假设 m 和“整型数值 5”相等，那么就会从“语句 5”一直执行到“语句 n+1”。

4) 如果直到最后一个“整型数值 n”都没有找到相等的值，那么就执行 default 后的“语句 n+1”。

需要重点强调的是，当和某个整型数值匹配成功后，会执行该分支以及后面所有分支的语句。例如：

```
1.  #include <stdio.h>
2.  int main() {
3.      int a;
4.      printf("Input integer number:");
5.      scanf("%d", &a);
6.      switch (a) {
7.      case 1: printf("Monday\n");
8.      case 2: printf("Tuesday\n");
9.      case 3: printf("Wednesday\n");
10.     case 4: printf("Thursday\n");
11.     case 5: printf("Friday\n");
12.     case 6: printf("Saturday\n");
13.     case 7: printf("Sunday\n");
14.     default:printf("error\n");
15.     }
```

```
16.       return 0;
17.  }
```

运行结果：

```
Input integer number:4↙
Thursday
Friday
Saturday
Sunday
error
```

输入 4，发现和第四个分支匹配成功，于是就执行第四个分支以及后面的所有分支。这显然不是我们想要的结果，我们希望只执行第四个分支，而跳过后面的其他分支。为了达到这个目标，必须要在每个分支最后添加 break;语句。

break 是 C 语言中的一个关键字，专门用于跳出 switch 语句。所谓“跳出”，是指一旦遇到 break，就不再执行 switch 中的任何语句，包括当前分支中的语句和其他分支中的语句；也就是说，整个 switch 执行结束了，接着会执行整个 switch 后面的代码。

使用 break 修改上面的代码：

```
1.   #include <stdio.h>
2.   int main() {
3.       int a;
4.       printf("Input integer number:");
5.       scanf("%d", &a);
6.       switch (a) {
7.       case 1: printf("Monday\n"); break;
8.       case 2: printf("Tuesday\n"); break;
9.       case 3: printf("Wednesday\n"); break;
10.      case 4: printf("Thursday\n"); break;
11.      case 5: printf("Friday\n"); break;
12.      case 6: printf("Saturday\n"); break;
13.      case 7: printf("Sunday\n"); break;
14.      default:printf("error\n"); break;
15.      }
16.      return 0;
17.  }
```

运行结果：

```
Input integer number:4↙
Thursday
```

由于 default 是最后一个分支，匹配后不会再执行其他分支，所以也可以不添加 break;语句。

最后需要说明的两点是：

1) case 后面必须是一个整数，或者是结果为整数的表达式，但不能包含任何变量。请看下面的例子：

```
1.   case 10: printf("..."); break;  //正确
```

```
2.  case 8 + 9: printf("..."); break;  //正确
3.  case 'A': printf("..."); break;  //正确，字符和整数可以相互转换
4.  case 'A' + 19: printf("..."); break;  //正确，字符和整数可以相互转换
5.  case 9.5: printf("..."); break;  //错误，不能为小数
6.  case a: printf("..."); break;    //错误，不能包含变量
7.  case a + 10: printf("..."); break;  //错误，不能包含变量
```

2) default 不是必须的。当没有 default 时，如果所有 case 都匹配失败，那么就什么都不执行。

5.5 C 语言条件运算符（? :）详解

如果希望获得两个数中最大的一个，可以使用 if 语句，例如：

```
1.  if (a>b) {
2.      max = a;
3.  }
4.  else {
5.      max = b;
6.  }
```

不过，C 语言提供了一种更加简单的方法，叫做条件运算符，语法格式为：

```
表达式 1？表达式 2：表达式 3
```

条件运算符是 C 语言中唯一的一个三目运算符，其求值规则为：如果表达式 1 的值为真，则以表达式 2 的值作为整个条件表达式的值，否则以表达式 3 的值作为整个条件表达式的值。条件表达式通常用于赋值语句之中。

上面的 if else 语句等价于：

```
max = (a>b) ? a : b;
```

该语句的语义是：如 a>b 为真，则把 a 赋予 max，否则把 b 赋予 max。

> 读者可以认为条件运算符是一种简写的 if else，完全可以用 if else 来替换。

使用条件表达式时，还应注意以下几点：
1) 条件运算符的优先级低于关系运算符和算术运算符，但高于赋值符。因此

```
max=(a>b) ? a : b;
```

可以去掉括号而写为

```
max=a>b ? a : b;
```

2) 条件运算符?和：是一对运算符，不能分开单独使用。

3) 条件运算符的结合方向是自右至左。例如：

```
a>b ? a : c>d ? c : d;
```

应理解为：

```
a>b ? a : ( c>d ? c : d );
```

这也就是条件表达式嵌套的情形，即其中的表达式又是一个条件表达式。

用条件表达式重新编程，输出两个数中的最大值：

```
1.  #include <stdio.h>
2.  int main() {
3.      int a, b;
4.      printf("Input two numbers:");
5.      scanf("%d %d", &a, &b);
6.      printf("max=%d\n", a>b ? a : b);
7.      return 0;
8.  }
```

运行结果：
Input two numbers:23 45
max=45

5.6 C 语言 while 循环和 do while 循环详解

在 C 语言中，共有三大常用的程序结构：

- 顺序结构：代码从前往后执行，没有任何“拐弯抹角”；
- 选择结构：也叫分支结构，重点要掌握 if else、switch 以及条件运算符；
- 循环结构：重复执行同一段代码。

前面讲解了顺序结构和选择结构，本节开始讲解循环结构。所谓循环（Loop），就是重复地执行同一段代码，例如要计算 1+2+3+……+99+100 的值，就要重复进行 99 次加法运算。

while 循环

while 循环的一般形式为：

```
while(表达式){
    语句块
}
```

意思是，先计算“表达式”的值，当值为真（非 0）时，执行“语句块”；执行完“语句块”，再次计算表达式的值，如果为真，继续执行“语句块”……这个过程会一直重复，直到表达式的值为假（0），就退出循环，执行 while 后面的代码。

我们通常将“表达式”称为循环条件，把“语句块”称为循环体，整个循环的过程就是不停判断循环条件、并执行循环体代码的过程。

用 while 循环计算 1 加到 100 的值：

```
#include <stdio.h>
int main() {
    int i = 1, sum = 0;
    while (i <= 100) {
        sum += i;
        i++;
    }
    printf("%d\n", sum);
    return 0;
}
```

运行结果：
5050

代码分析：
1) 程序运行到 while 时，因为 i=1，i<=100 成立，所以会执行循环体；执行结束后 i 的值变为 2，sum 的值变为 1。

2) 接下来会继续判断 i<=100 是否成立，因为此时 i=2，i<=100 成立，所以继续执行循环体；执行结束后 i 的值变为 3，sum 的值变为 3。

3) 重复执行步骤 2)。

4) 当循环进行到第 100 次，i 的值变为 101，sum 的值变为 5050；因为此时 i<=100 不再成立，所以就退出循环，不再执行循环体，转而执行 while 循环后面的代码。

while 循环的整体思路是这样的：设置一个带有变量的循环条件，也即一个带有变量的表达式；在循环体中额外添加一条语句，让它能够改变循环条件中变量的值。这样，随着循环的不断执行，循环条件中变量的值也会不断变化，终有一个时刻，循环条件不再成立，整个循环就结束了。

如果循环条件中不包含变量，会发生什么情况呢？

1) 循环条件成立时的话，while 循环会一直执行下去，永不结束，成为“死循环”。例如：

```
#include <stdio.h>
int main() {
    while (1) {
        printf("1");
    }
    return 0;
}
```

运行程序，会不停地输出“1”，直到用户强制关闭。

2) 循环条件不成立的话，while 循环就一次也不会执行。例如：

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

```
3.        while (0) {
4.            printf("1");
5.        }
6.        return 0;
7.    }
```

运行程序，什么也不会输出。

再看一个例子，统计从键盘输入的一行字符的个数：

```
1.    #include <stdio.h>
2.    int main() {
3.        int n = 0;
4.        printf("Input a string:");
5.        while (getchar() != '\n') n++;
6.        printf("Number of characters: %d\n", n);
7.        return 0;
8.    }
```

运行结果：
Input a string:c.biancheng.net↙
Number of characters: 15

本例程序中的循环条件为 getchar()!='\n'，其意义是，只要从键盘输入的字符不是回车就继续循环。循环体 n++;完成对输入字符个数计数。

do-while 循环

除了 while 循环，在 C 语言中还有一种 do-while 循环。

do-while 循环的一般形式为：

```
do{
    语句块
}while(表达式);
```

do-while 循环与 while 循环的不同在于：它会先执行“语句块”，然后再判断表达式是否为真，如果为真则继续循环；如果为假，则终止循环。因此，do-while 循环至少要执行一次“语句块”。

用 do-while 计算 1 加到 100 的值：

```
1.    #include <stdio.h>
2.    int main() {
3.        int i = 1, sum = 0;
4.        do {
5.            sum += i;
6.            i++;
7.        } while (i <= 100);
8.        printf("%d\n", sum);
```

```
9.      return 0;
10. }
```

运行结果：
5050

> 注意 while(i<=100);最后的分号;，这个必须要有。

while 循环和 do-while 各有特点，大家可以适当选择，实际编程中使用 while 循环较多。

5.7 C 语言 for 循环（for 语句）详解

除了 while 循环，C 语言中还有 for 循环，它的使用更加灵活，完全可以取代 while 循环。

上节我们使用 while 循环来计算 1 加到 100 的值，代码如下：

```
1.  #include <stdio.h>
2.  int main() {
3.      int i, sum = 0;
4.      i = 1;  //语句①
5.      while (i <= 100 /*语句②*/) {
6.          sum += i;
7.          i++;  //语句③
8.      }
9.      printf("%d\n", sum);
10.     return 0;
11. }
```

可以看到，语句①②③被放到了不同的地方，代码结构较为松散。为了让程序更加紧凑，可以使用 for 循环来代替，如下所示：

```
12. #include <stdio.h>
1.  int main() {
2.      int i, sum = 0;
3.      for (i = 1/*语句①*/; i <= 100/*语句②*/; i++/*语句③*/) {
4.          sum += i;
5.      }
6.      printf("%d\n", sum);
7.      return 0;
8.  }
```

在 for 循环中，语句①②③被集中到了一起，代码结构一目了然。

for 循环的一般形式为：

```
for(表达式 1; 表达式 2; 表达式 3){
    语句块
}
```

它的运行过程为：

1) 先执行“表达式 1”。

2) 再执行“表达式 2”，如果它的值为真（非 0），则执行循环体，否则结束循环。

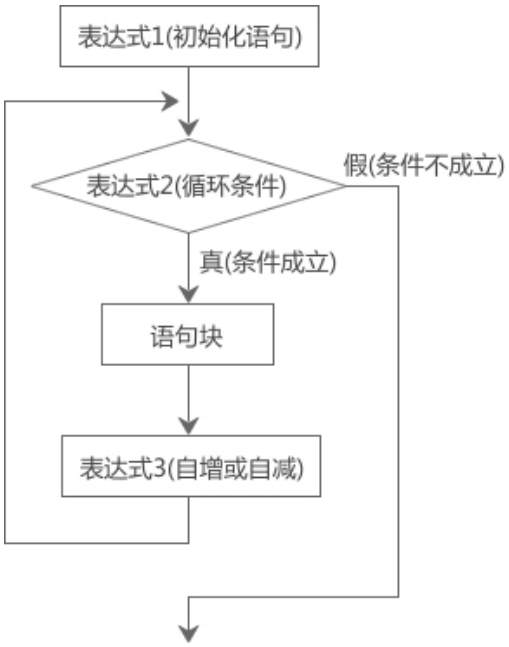
3) 执行完循环体后再执行“表达式 3”。

4) 重复执行步骤 2) 和 3)，直到“表达式 2”的值为假，就结束循环。

上面的步骤中，2) 和 3) 是一次循环，会重复执行，for 语句的主要作用就是不断执行步骤 2) 和 3)。

“表达式 1”仅在第一次循环时执行，以后都不会再执行，可以认为这是一个初始化语句。“表达式 2”一般是一个关系表达式，决定了是否还要继续下次循环，称为“循环条件”。“表达式 3”很多情况下是一个带有自增或自减操作的表达式，以使循环条件逐渐变得“不成立”。

for 循环的执行过程可用下图表示：



我们再来分析一下“计算从 1 加到 100 的和”的代码：

```
1.  #include <stdio.h>
2.  int main() {
3.      int i, sum = 0;
4.      for (i = 1; i <= 100; i++) {
5.          sum += i;
6.      }
7.      printf("%d\n", sum);
8.      return 0;
9.  }
```

运行结果：

5050

代码分析：
1) 执行到 for 语句时，先给 i 赋初值 1，判断 i<=100 是否成立；因为此时 i=1，i<=100 成立，所以执行循环体。循环体执行结束后（sum 的值为 1），再计算 i++。

2) 第二次循环时，i 的值为 2，i<=100 成立，继续执行循环体。循环体执行结束后（sum 的值为 3），再计算 i++。

3) 重复执行步骤 2)，直到第 101 次循环，此时 i 的值为 101，i<=100 不成立，所以结束循环。

由此我们可以总结出 for 循环的一般形式：

```
for(初始化语句; 循环条件; 自增或自减){
    语句块
}
```

C 语言 for 循环中的三个表达式

for 循环中的“表达式 1（初始化条件）”、“表达式 2(循环条件)”和“表达式 3（自增或自减）”都是可选项，都可以省略（但分号;必须保留）。

1) 修改“从 1 加到 100 的和”的代码，省略“表达式 1（初始化条件）”：

```
1.  int i = 1, sum = 0;
2.  for (; i <= 100; i++) {
3.      sum += i;
4.  }
```

可以看到，将 i=1 移到了 for 循环的外面。

2) 省略了“表达式 2(循环条件)”，如果不做其它处理就会成为死循环。例如：

```
1.  for (i = 1; ; i++) sum = sum + i;
```

相当于：

```
1.  i = 1;
2.  while (1) {
3.      sum = sum + i;
4.      i++;
5.  }
```

所谓死循环，就是循环条件永远成立，循环会一直进行下去，永不结束。死循环对程序的危害很大，一定要避免。

3) 省略了“表达式 3(自增或自减)”，就不会修改“表达式 2(循环条件)”中的变量，这时可在循环体中加入修改变量的语句。例如：

```
1.  for (i = 1; i <= 100; ) {
2.      sum = sum + i;
3.      i++;
4.  }
```

4) 省略了“表达式 1(初始化语句)”和“表达式 3(自增或自减)”。例如：

```
for (; i <= 100; ) {
    sum = sum + i;
    i++;
}
```

相当于：

```
while (i <= 100) {
    sum = sum + i;
    i++;
}
```

5) 3 个表达式可以同时省略。例如：

```
for(;;)  语句
```

相当于：

```
while(1)  语句
```

6) “表达式 1”可以是初始化语句，也可以是其他语句。例如：

```
for (sum = 0; i <= 100; i++)  sum = sum + i;
```

7) “表达式 1”和“表达式 3”可以是一个简单表达式也可以是逗号表达式。

```
for (sum = 0, i = 1; i <= 100; i++)  sum = sum + i;
```

或：

```
for (i = 0, j = 100; i <= 100; i++, j--)  k = i + j;
```

8) “表达式 2”一般是关系表达式或逻辑表达式，但也可是数值或字符，只要其值非零，就执行循环体。例如：

```
for (i = 0; (c = getchar()) != '\n'; i += c);
```

又如：

```
for (; (c = getchar()) != '\n'; )
    printf("%c", c);
```

5.8 C 语言跳出循环（break 和 continue）

使用 while 或 for 循环时，如果想提前结束循环（在不满足结束条件的情况下结束循环），可以使用 break 或 continue 关键字。

break 关键字

在《C 语言 switch case 语句》一节中，我们讲到了 break，用它来跳出 switch 语句。

当 break 关键字用于 while、for 循环时，会终止循环而执行整个循环语句后面的代码。break 关键字通常和 if 语句一起使用，即满足条件时便跳出循环。

使用 while 循环计算 1 加到 100 的值：

```
#include <stdio.h>
int main() {
    int i = 1, sum = 0;
    while (1) {  //循环条件为死循环
        sum += i;
        i++;
        if (i>100) break;
    }
    printf("%d\n", sum);
    return 0;
}
```

运行结果：
5050

while 循环条件为 1，是一个死循环。当执行到第 100 次循环的时候，计算完 i++;后 i 的值为 101，此时 if 语句的条件 i> 100 成立，执行 break;语句，结束循环。

在多层循环中，一个 break 语句只向外跳一层。例如，输出一个 4*4 的整数矩阵：

```
#include <stdio.h>
int main() {
    int i = 1, j;
    while (1) {  // 外层循环
        j = 1;
        while (1) {  // 内层循环
            printf("%-4d", i*j);
            j++;
            if (j>4) break;  //跳出内层循环
        }
        printf("\n");
        i++;
        if (i>4) break;  // 跳出外层循环
    }

    return 0;
}
```

运行结果：

```
1   2   3   4
2   4   6   8
```

```
3   6   9   12
4   8   12  16
```

当 j>4 成立时，执行 break;，跳出内层循环；外层循环依然执行，直到 i>4 成立，跳出外层循环。内层循环共执行了 4 次，外层循环共执行了 1 次。

continue 语句

continue 语句的作用是跳过循环体中剩余的语句而强制进入下一次循环。continue 语句只用在 while、for 循环中，常与 if 条件语句一起使用，判断条件是否成立。

来看一个例子：

```
1.  #include <stdio.h>
2.  int main() {
3.      char c = 0;
4.      while (c != '\n') {  //回车键结束循环
5.          c = getchar();
6.          if (c == '4' || c == '5') {  //按下的是数字键4或5
7.              continue;  //跳过当次循环，进入下次循环
8.          }
9.          putchar(c);
10.     }
11.     return 0;
12. }
```

运行结果：

0123456789↙

01236789

程序遇到 while 时，变量 c 的值为'\0'，循环条件 c!='\n'成立，开始第一次循环。getchar() 使程序暂停执行，等待用户输入，直到用户按下回车键才开始读取字符。

本例我们输入的是 0123456789，当读取到 4 或 5 时，if 的条件 c=='4'||c=='5'成立，就执行 continue 语句，结束当前循环，直接进入下一次循环，也就是说 putchar(c);不会被执行到。而读取到其他数字时，if 的条件不成立，continue 语句不会被执行到，putchar(c);就会输出读取到的字符。

break 与 continue 的对比：break 用来结束所有循环，循环语句不再有执行的机会；continue 用来结束本次循环，直接跳到下一次循环，如果循环条件成立，还会继续循环。

5.9 C 语言循环嵌套

在 C 语言中，if-else、while、do-while、for 都可以相互嵌套。所谓嵌套（Nest），就是一条语句里面还有另一条语句，例如 for 里面还有 for，while 里面还有 while，或者 for 里面有 while，while 里面有 if-else，这都是允许的。

if-else 的嵌套在《C 语言 if else 语句》一节中已经进行了讲解，本节主要介绍循环结构的嵌套。

示例 1：for 嵌套执行的流程。

```
#include <stdio.h>
int main()
{
    int i, j;
    for (i = 1; i <= 4; i++) {  //外层for循环
        for (j = 1; j <= 4; j++) {  //内层for循环
            printf("i=%d, j=%d\n", i, j);
        }
        printf("\n");
    }
    return 0;
}
```

运行结果：

```
i=1, j=1
i=1, j=2
i=1, j=3
i=1, j=4

i=2, j=1
i=2, j=2
i=2, j=3
i=2, j=4

i=3, j=1
i=3, j=2
i=3, j=3
i=3, j=4

i=4, j=1
i=4, j=2
i=4, j=3
i=4, j=4
```

本例是一个简单的 for 循环嵌套，外层循环和内层循环交叉执行，外层 for 每执行一次，内层 for 就要执行四次。

在 C 语言中，代码是顺序、同步执行的，当前代码必须执行完毕后才能执行后面的代码。这就意味着，外层 for 每次循环时，都必须等待内层 for 循环完毕（也就是循环 4 次）才能进行下次循环。虽然 i 是变量，但是对于内层 for 来说，每次循环时它的值都是固定的。

示例 2：输出一个 4×4 的整数矩阵。

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

```
2.  int main()
3.  {
4.      int i, j;
5.      for (i = 1; i <= 4; i++) {  //外层for循环
6.          for (j = 1; j <= 4; j++) {  //内层for循环
7.              printf("%-4d", i*j);
8.          }
9.          printf("\n");
10.     }
11.     return 0;
12. }
```

运行结果：

```
1   2   3   4
2   4   6   8
3   6   9   12
4   8   12  16
```

外层 for 第一次循环时，i 为 1，内层 for 要输出四次 1*j 的值，也就是第一行数据；内层 for 循环结束后执行 printf("\n")，输出换行符；接着执行外层 for 的 i++ 语句。此时外层 for 的第一次循环才算结束。

外层 for 第二次循环时，i 为 2，内层 for 要输出四次 2*j 的值，也就是第二行的数据；接下来执行 printf("\n") 和 i++，外层 for 的第二次循环才算结束。外层 for 第三次、第四次循环以此类推。

可以看到，内层 for 每循环一次输出一个数据，而外层 for 每循环一次输出一行数据。

示例 3：输出九九乘法表。

```
1.  #include <stdio.h>
2.  int main() {
3.      int i, j;
4.      for (i = 1; i <= 9; i++) {  //外层for循环
5.          for (j = 1; j <= i; j++) {  //内层for循环
6.              printf("%d*%d=%-2d  ", i, j, i*j);
7.          }
8.          printf("\n");
9.      }
10.
11.     return 0;
12. }
```

运行结果：

```
1*1=1
2*1=2   2*2=4
3*1=3   3*2=6   3*3=9
```

```
4*1=4   4*2=8   4*3=12  4*4=16
5*1=5   5*2=10  5*3=15  5*4=20  5*5=25
6*1=6   6*2=12  6*3=18  6*4=24  6*5=30  6*6=36
7*1=7   7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49
8*1=8   8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64
9*1=9   9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

和示例 2 一样，内层 for 每循环一次输出一条数据，外层 for 每循环一次输出一行数据。

需要注意的是，内层 for 的结束条件是 j<=i。外层 for 每循环一次，i 的值就会变化，所以每次开始内层 for 循环时，结束条件是不一样的。具体如下：

- 当 i=1 时，内层 for 的结束条件为 j<=1，只能循环一次，输出第一行。
- 当 i=2 时，内层 for 的结束条件是 j<=2，循环两次，输出第二行。
- 当 i=3 时，内层 for 的结束条件是 j<=3，循环三次，输出第三行。
- 当 i=4、5、6... 时，以此类推。

九九乘法表还有更多输出方式，请查看：C 语言输出九九乘法表(5 种解法)

5.10 对选择结构和循环结构的总结

C 语言中常用的编程结构有三种（其它编程语言也是如此），它们分别是：

- **顺序结构**：代码从前往后依次执行，没有任何“拐弯抹角”，不跳过任何一条语句，所有的语句都会被执行到。
- **选择结构**：也叫分支结构。代码会被分成多个部分，程序会根据特定条件（某个表达式的运算结果）来判断到底执行哪一部分。
- **循环结构**：程序会重新执行同一段代码，直到条件不再满足，或者遇到强行跳出语句（break 关键字）。

选择结构

选择结构（分支结构）涉及到的关键字包括 if、else、switch、case、break，还有一个条件运算符?:（这是 C 语言中唯一的一个三目运算符）。其中，if...else 是最基本的结构，switch...case 和?:都是由 if...else 演化而来，它们都是为了让程序员书写更加方便。

你可以只使用 if，也可以 if...else 配对使用。另外要善于使用 switch...case 和?:，有时候它们看起来更加清爽。

if...else 可以嵌套使用，原则上嵌套的层次（深度）没有限制，但是过多的嵌套层次会让代码结构混乱。

循环结构

C 语言中常用的循环结构有 while 循环和 for 循环，它们都可以用来处理同一个问题，一般可以互相代替。

除了 while 和 for，C 语言中还有一个 goto 语句，它也能构成循环结构。不过由于 goto 语句很容易造成代码混乱，维护和阅读困难，饱受诟病，不被推荐，而且 goto 循环完全可以被其他循环取代，所以后来的很多编程语言都取消了 goto 语句，我们也不再讲解。

> 国内很多大学仍然讲解 goto 语句，但也仅仅是完成教材所设定的课程，goto 语句在实际开发中很难见到。

对于 while 和 do-while 循环，循环体中应包括使循环趋于结束的语句。

对于 while 和 do-while 循环，循环变量的初始化操作应该在 while 和 do-while 语句之前完成，而 for 循环可以在内部实现循环变量的初始化。

for 循环是最常用的循环，它的功能强大，一般都可以代替其他循环。

最后还要注意 break 和 continue 关键字用于循环结构时的区别：
- break 用来跳出所有循环，循环语句不再有执行的机会；
- continue 用来结束本次循环，直接跳到下一次循环，如果循环条件成立，还会继续循环。

此外，break 关键字还可以用于跳出 switch...case 语句。所谓“跳出”，是指一旦遇到 break，就不再执行 switch 中的任何语句，包括当前分支中的语句和其他分支中的语句；也就是说，整个 switch 执行结束了，接着会执行整个 switch 后面的代码。

5.11 谈编程思维的培养，初学者如何实现自我突破（非常重要）

5.12 写一个内存泄露的例子，让计算机内存爆满

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

第 06 章 C 语言数组

数组（Array）就是一些列具有相同类型的数据的集合，这些数据在内存中依次挨着存放，彼此之间没有缝隙。

数组不是 C 语言的专利，Java、C++、C#、JavaScript、PHP 等其他编程语言也有数组。

C 语言数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或是构造类型。因此按数组元素的类型不同，数组又可分为数值数组、字符数组、指针数组、结构数组等各种类别。

本章目录：

1. 什么是数组？
2. C 语言二维数组
3. 【实例】判断数组中是否包含某个元素
4. C 语言字符数组和字符串
5. C 语言字符串的输入和输出
6. C 语言字符串处理函数
7. C 语言数组是静态的，不能插入或删除元素
8. C 语言数组的越界和溢出
9. C 语言变长数组：使用变量指明数组的长度
10. C 语言对数组元素进行排序（冒泡排序法）
11. 对 C 语言数组的总结

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

6.1 什么是数组？

在《C 语言数据输出大汇总以及轻量进阶》一节中我们举了一个例子，是输出一个 4×4 的整数矩阵，代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a1 = 20, a2 = 345, a3 = 700, a4 = 22;
    int b1 = 56720, b2 = 9999, b3 = 20098, b4 = 2;
    int c1 = 233, c2 = 205, c3 = 1, c4 = 6666;
    int d1 = 34, d2 = 0, d3 = 23, d4 = 23006783;

    printf("%-9d %-9d %-9d %-9d\n", a1, a2, a3, a4);
    printf("%-9d %-9d %-9d %-9d\n", b1, b2, b3, b4);
    printf("%-9d %-9d %-9d %-9d\n", c1, c2, c3, c4);
    printf("%-9d %-9d %-9d %-9d\n", d1, d2, d3, d4);

    system("pause");
    return 0;
}
```

运行结果：

```
20        345       700       22
56720     9999      20098     2
233       205       1         6666
34        0         23        23006783
```

矩阵共有 16 个整数，我们为每个整数定义了一个变量，也就是 16 个变量。那么，为了减少变量的数量，让开发更有效率，能不能为多个数据定义一个变量呢？比如，把每一行的整数放在一个变量里面，或者把 16 个整数全部都放在一个变量里面。答案当然是肯定的，办法就是使用数组（Array）。

数组的概念和定义

我们知道，要想把数据放入内存，必须先要分配内存空间。放入 4 个整数，就得分配 4 个 int 类型的内存空间：

```
int a[4];
```

这样，就在内存中分配了 4 个 int 类型的内存空间，共 4×4=16 个字节，并为它们起了一个名字，叫 a。

我们把这样的一组数据的集合称为**数组（Array）**，它所包含的每一个数据叫做数组元素（Element），所包含的数据的个数称为数组长度（Length），例如 int a[4];就定义了一个长度为 4 的整型数组，名字是 a。

数组中的每个元素都有一个序号，这个序号从 0 开始，而不是从我们熟悉的 1 开始，称为下标（Index）。使用数组元素时，指明下标即可，形式为：

```
arrayName[index]
```

arrayName 为数组名称，index 为下标。例如，a[0] 表示第 0 个元素，a[3] 表示第 3 个元素。

接下来我们就把第一行的 4 个整数放入数组：

```
a[0]=20;
a[1]=345;
a[2]=700;
a[3]=22;
```

这里的 0、1、2、3 就是数组下标，a[0]、a[1]、a[2]、a[3] 就是数组元素。

在学习过程中，我们经常会使用循环结构将数据放入数组中（也就是为数组元素逐个赋值），然后再使用循环结构输出（也就是依次读取数组元素的值），下面我们就来演示一下如何将 1~10 这十个数字放入数组中：

```
#include <stdio.h>
int main() {
    int nums[10];
    int i;

    //将1~10放入数组中
    for (i = 0; i<10; i++) {
        nums[i] = (i + 1);
    }

    //依次输出数组元素
    for (i = 0; i<10; i++) {
        printf("%d ", nums[i]);
    }

    return 0;
}
```

运行结果：

```
1 2 3 4 5 6 7 8 9 10
```

变量 i 既是数组下标，也是循环条件；将数组下标作为循环条件，达到最后一个元素时就结束循环。数组 nums 的最大下标是 9，也就是不能超过 10，所以我们规定循环的条件是 i<10，一旦 i 达到 10 就得结束循环。

更改上面的代码，让用户输入 10 个数字并放入数组中：

```
1.  #include <stdio.h>
2.  int main() {
3.      int nums[10];
4.      int i;
5.
6.      //从控制台读取用户输入
7.      for (i = 0; i<10; i++) {
8.          scanf("%d", &nums[i]);  //注意取地址符 &，不要遗忘哦
9.      }
10.
11.     //依次输出数组元素
12.     for (i = 0; i<10; i++) {
13.         printf("%d ", nums[i]);
14.     }
15.
16.     return 0;
17. }
```

运行结果：

```
22 18 928 5 4 82 30 10 666 888↙
22 18 928 5 4 82 30 10 666 888
```

第 8 行代码中，scanf() 读取数据时需要一个地址（地址用来指明数据的存储位置），而 nums[i] 表示一个具体的数组元素，所以我们要在前边加 & 来获取地址。

最后我们来总结一下数组的定义方式：

```
dataType  arrayName[length];
```

dataType 为数据类型，arrayName 为数组名称，length 为数组长度。例如：

```
1.  float m[12];  //定义一个长度为 12 的浮点型数组
2.  char ch[9];  //定义一个长度为 9 的字符型数组
```

需要注意的是：

1) 数组中每个元素的数据类型必须相同，对于 int a[4];，每个元素都必须为 int。

2) 数组长度 length 最好是整数或者常量表达式，例如 10、20*4 等，这样在所有编译器下都能运行通过；如果 length 中包含了变量，例如 n、4*m 等，在某些编译器下就会报错，我们将在《C 语言变长数组：使用变量指明数组的长度》一节专门讨论这点。

3) 访问数组元素时，下标的取值范围为 0 ≤ index < length，过大或过小都会越界，导致数组溢出，发生不可预测的情况，我们将在《C 语言数组的越界和溢出》一节重点讨论，请大家务必要引起注意。

数组内存是连续的

数组是一个整体，它的内存是连续的；也就是说，数组元素之间是相互挨着的，彼此之间没有一点点缝隙。下图演示了 int a[4];在内存中的存储情形：

「数组内存是连续的」这一点很重要，所以我使用了一个大标题来强调。连续的内存为指针操作（通过指针来访问数组元素）和内存处理（整块内存的复制、写入等）提供了便利，这使得数组可以作为缓存（临时存储数据的一块内存）使用。大家暂时可能不理解这句话是什么意思，等后边学了指针和内存自然就明白了。

数组的初始化

上面的代码是先定义数组再给数组赋值，我们也可以在定义数组的同时赋值，例如：

```
int a[4] = {20, 345, 700, 22};
```

数组元素的值由{ }包围，各个值之间以,分隔。

对于数组的初始化需要注意以下几点：

1) 可以只给部分元素赋值。当{ }中值的个数少于元素个数时，只给前面部分元素赋值。例如：

```
int a[10]={12, 19, 22 , 993, 344};
```

表示只给 a[0]~a[4] 5 个元素赋值，而后面 5 个元素自动初始化为 0。

当赋值的元素少于数组总体元素的时候，剩余的元素自动初始化为 0：

- 对于 short、int、long，就是整数 0；
- 对于 char，就是字符 '\0'；
- 对于 float、double，就是小数 0.0。

我们可以通过下面的形式将数组的所有元素初始化为 0：

```
int nums[10] = {0};
char str[10] = {0};
float scores[10] = {0.0};
```

由于剩余的元素会自动初始化为 0，所以只需要给第 0 个元素赋值为 0 即可。

2) 只能给元素逐个赋值，不能给数组整体赋值。例如给 10 个元素全部赋值为 1，只能写作：

```
int a[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
```

而不能写作：

```
int a[10] = 1;
```

3) 如给全部元素赋值，那么在定义数组时可以不给出数组长度。例如：

```
int a[] = {1, 2, 3, 4, 5};
```

等价于

```
int a[5] = {1, 2, 3, 4, 5};
```

最后，我们借助数组来输出一个 4×4 的矩阵：

```
#include <stdio.h>
int main()
{
    int a[4] = { 20, 345, 700, 22 };
    int b[4] = { 56720, 9999, 20098, 2 };
    int c[4] = { 233, 205, 1, 6666 };
    int d[4] = { 34, 0, 23, 23006783 };

    printf("%-9d %-9d %-9d %-9d\n", a[0], a[1], a[2], a[3]);
    printf("%-9d %-9d %-9d %-9d\n", b[0], b[1], b[2], b[3]);
    printf("%-9d %-9d %-9d %-9d\n", c[0], c[1], c[2], c[3]);
    printf("%-9d %-9d %-9d %-9d\n", d[0], d[1], d[2], d[3]);

    return 0;
}
```

6.2 C 语言二维数组

上节讲解的数组可以看作是一行连续的数据，只有一个下标，称为一维数组。在实际问题中有很多数据是二维的或多维的，因此 C 语言允许构造多维数组。多维数组元素有多个下标，以确定它在数组中的位置。本节只介绍二维数组，多维数组可由二维数组类推而得到。

二维数组的定义

二维数组定义的一般形式是：

```
dataType arrayName[length1][length2];
```

其中，dataType 为数据类型，arrayName 为数组名，length1 为第一维下标的长度，length2 为第二维下标的长度。

我们可以将二维数组看做一个 Excel 表格，有行有列，length1 表示行数，length2 表示列数，要在二维数组中定位某个元素，必须同时指明行和列。例如：

```
int a[3][4];
```

定义了一个 3 行 4 列的二维数组，共有 3×4=12 个元素，数组名为 a，即：

```
a[0][0], a[0][1], a[0][2], a[0][3]
```

```
a[1][0], a[1][1], a[1][2], a[1][3]
a[2][0], a[2][1], a[2][2], a[2][3]
```

如果想表示第 2 行第 1 列的元素，应该写作 a[2][1]。

也可以将二维数组看成一个坐标系，有 x 轴和 y 轴，要想在一个平面中确定一个点，必须同时知道 x 轴和 y 轴。

二维数组在概念上是二维的，但在内存中是连续存放的；换句话说，二维数组的各个元素是相互挨着的，彼此之间没有缝隙。那么，如何在线性内存中存放二维数组呢？有两种方式：

- 一种是按行排列，即放完一行之后再放入第二行；
- 另一种是按列排列，即放完一列之后再放入第二列。

在 C 语言中，二维数组是按行排列的。也就是先存放 a[0] 行，再存放 a[1] 行，最后存放 a[2] 行；每行中的 4 个元素也是依次存放。数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 4×(3×4)=48 个字节。

你可以这样认为，二维数组是由多个长度相同的一维数组构成的。

【实例 1】一个学习小组有 5 个人，每个人有 3 门课程的考试成绩，求该小组各科的平均分和总平均分。

--	Math	C	English
张涛	80	75	92
王正华	61	65	71
李丽丽	59	63	70
赵圈圈	85	87	90
周梦真	76	77	85

对于该题目，可以定义一个二维数组 a[5][3] 存放 5 个人 3 门课的成绩，定义一个一维数组 v[3] 存放各科平均分，再定义一个变量 average 存放总平均分。最终编程如下：

```
1.  #include <stdio.h>
2.  int main() {
3.      int i, j;  //二维数组下标
4.      int sum = 0;  //当前科目的总成绩
5.      int average;  //总平均分
6.      int v[3];  //各科平均分
7.      int a[5][3];  //用来保存每个同学各科成绩的二维数组
8.      printf("Input score:\n");
9.      for (i = 0; i<3; i++) {
10.         for (j = 0; j<5; j++) {
11.             scanf("%d", &a[j][i]);  //输入每个同学的各科成绩
12.             sum += a[j][i];  //计算当前科目的总成绩
13.         }
14.         v[i] = sum / 5;  // 当前科目的平均分
15.         sum = 0;
```

```
    }
    average = (v[0] + v[1] + v[2]) / 3;
    printf("Math: %d\nC Languag: %d\nEnglish: %d\n", v[0], v[1], v[2]);
    printf("Total: %d\n", average);
    return 0;
}
```

运行结果：

```
Input score:
80 61 59 85 76 75 65 63 87 77 92 71 70 90 85↙
Math: 72
C Languag: 73
English: 81
Total: 75
```

程序使用了一个嵌套循环来读取所有学生所有科目的成绩。在内层循环中依次读入某一门课程的各个学生的成绩，并把这些成绩累加起来，退出内层循环（进入外层循环）后再把该累加成绩除以 5 送入 v[i] 中，这就是该门课程的平均分。外层循环共循环三次，分别求出三门课各自的平均成绩并存放在数组 v 中。所有循环结束后，把 v[0]、v[1]、v[2] 相加除以 3 就可以得到总平均分。

二维数组的初始化（赋值）

二维数组的初始化可以按行分段赋值，也可按行连续赋值。

例如，对于数组 a[5][3]，按行分段赋值应该写作：

```
int a[5][3]={ {80,75,92}, {61,65,71}, {59,63,70}, {85,87,90}, {76,77,85} };
```

按行连续赋值应该写作：

```
int a[5][3]={80, 75, 92, 61, 65, 71, 59, 63, 70, 85, 87, 90, 76, 77, 85};
```

这两种赋初值的结果是完全相同的。

【实例 2】和“实例 1”类似，依然求各科的平均分和总平均分，不过本例要求在初始化数组的时候直接给出成绩。

```
#include <stdio.h>
int main() {
    int i, j;  //二维数组下标
    int sum = 0;  //当前科目的总成绩
    int average;  //总平均分
    int v[3];  //各科平均分
    int a[5][3] = { { 80,75,92 }, { 61,65,71 }, { 59,63,70 }, { 85,87,90 }, { 76,77,85 } };

    for (i = 0; i<3; i++) {
        for (j = 0; j<5; j++) {
            sum += a[j][i];  //计算当前科目的总成绩
        }
        v[i] = sum / 5;  // 当前科目的平均分
```

```
14.             sum = 0;
15.         }
16.
17.         average = (v[0] + v[1] + v[2]) / 3;
18.         printf("Math: %d\nC Languag: %d\nEnglish: %d\n", v[0], v[1], v[2]);
19.         printf("Total: %d\n", average);
20.
21.         return 0;
22.  }
```

运行结果：
Math: 72
C Languag: 73
English: 81
Total: 75

对于二维数组的初始化还要注意以下几点：

1) 可以只对部分元素赋值，未赋值的元素自动取“零”值。例如：

```
int a[3][3] = {{1}, {2}, {3}};
```

是对每一行的第一列元素赋值，未赋值的元素的值为 0。赋值后各元素的值为：
1 0 0
2 0 0
3 0 0

再如：

```
int a[3][3] = {{0,1}, {0,0,2}, {3}};
```

赋值后各元素的值为：
0 1 0
0 0 2
3 0 0

2) 如果对全部元素赋值，那么第一维的长度可以不给出。例如：

```
int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

可以写为：

```
int a[][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

3) 二维数组可以看作是由一维数组嵌套而成的；如果一个数组的每个元素又是一个数组，那么它就是二维数组。当然，前提是各个元素的类型必须相同。根据这样的分析，一个二维数组也可以分解为多个一维数组，C 语言允许这种分解。

例如，二维数组 a[3][4]可分解为三个一维数组，它们的数组名分别为 a[0]、a[1]、a[2]。

这三个一维数组可以直接拿来使用。这三个一维数组都有 4 个元素，比如，一维数组 a[0] 的元素为 a[0][0]、a[0][1]、a[0][2]、a[0][3]。

6.3【实例】判断数组中是否包含某个元素

在实际开发中，经常需要查询数组中的元素。例如，学校为每位同学分配了一个唯一的编号，现在有一个数组，保存了实验班所有同学的编号信息，如果有家长想知道他的孩子是否进入了实验班，只要提供孩子的编号就可以，如果编号和数组中的某个元素相等，就进入了实验班，否则就没进入。

不幸的是，C 语言标准库没有提供与数组查询相关的函数，所以我们只能自己编写代码。

对无序数组的查询

所谓无序数组，就是数组元素的排列没有规律。无序数组元素查询的思路也很简单，就是用循环遍历数组中的每个元素，把要查询的值挨个比较一遍。请看下面的代码：

```
#include <stdio.h>
int main() {
    int nums[10] = { 1, 10, 6, 296, 177, 23, 0, 100, 34, 999 };
    int i, num, thisindex = -1;

    printf("Input an integer: ");
    scanf("%d", &num);
    for (i = 0; i<10; i++) {
        if (nums[i] == num) {
            thisindex = i;
            break;
        }
    }
    if (thisindex < 0) {
        printf("%d isn't  in the array.\n", num);
    }
    else {
        printf("%d is  in the array, it's index is %d.\n", num, thisindex);
    }

    return 0;
}
```

运行结果：

```
Input an integer: 100↙
100 is  in the array, it's index is 7.
```

或者

```
Input an integer: 28↙
```

```
28 isn't  in the array.
```

这段代码的作用是让用户输入一个数字，判断该数字是否在数组中，如果在，就打印出下标。

第 10~15 行代码是关键，它会遍历数组中的每个元素，和用户输入的数字进行比较，如果相等就获取它的下标并跳出循环。

注意：数组下标的取值范围是非负数，当 thisindex >= 0 时，该数字在数组中，当 thisindex < 0 时，该数字不在数组中，所以在定义 thisindex 变量时，必须将其初始化为一个负数。

对有序数组的查询

查询无序数组需要遍历数组中的所有元素，而查询有序数组只需要遍历其中一部分元素。例如有一个长度为 10 的整型数组，它所包含的元素按照从小到大的顺序（升序）排列，假设比较到第 4 个元素时发现它的值大于输入的数字，那么剩下的 5 个元素就没必要再比较了，肯定也大于输入的数字，这样就减少了循环的次数，提高了执行效率。

请看下面的代码：

```
1.  #include <stdio.h>
2.  int main() {
3.      int nums[10] = { 0, 1, 6, 10, 23, 34, 100, 177, 296, 999 };
4.      int i, num, thisindex = -1;
5.
6.      printf("Input an integer: ");
7.      scanf("%d", &num);
8.      for (i = 0; i<10; i++) {
9.          if (nums[i] == num) {
10.             thisindex = i;
11.             break;
12.         }
13.         else if (nums[i] > num) {
14.             break;
15.         }
16.     }
17.     if (thisindex < 0) {
18.         printf("%d isn't  in the array.\n", num);
19.     }
20.     else {
21.         printf("%d is  in the array, it's index is %d.\n", num, thisindex);
22.     }
23.
24.     return 0;
25. }
```

与前面的代码相比，这段代码的改动很小，只增加了一个判断语句，也就是 12~14 行。因为数组元素是升序排列的，所以当 nums[i] > num 时，i 后边的元素也都大于 num 了，num 肯定不在数组中了，就没有必要再继续比较

了，终止循环即可。

6.4 C 语言字符数组和字符串

用来存放字符的数组称为**字符数组**，例如：

```
1.  char a[10];  //一维字符数组
2.  char b[5][10];  //二维字符数组
3.  char c[20] = { 'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a','m' };  // 给部分数组元素赋值
4.  char d[] = { 'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm' };  //对全体元素赋值时可以省去长度
```

字符数组实际上是一系列字符的集合，也就是**字符串（String）**。在 C 语言中，没有专门的字符串变量，没有 string 类型，通常就用一个字符数组来存放一个字符串。

C 语言规定，可以将字符串直接赋值给字符数组，例如：

```
1.  char str[30] = { "c.biancheng.net" };
2.  char str[30] = "c.biancheng.net";  //这种形式更加简洁，实际开发中常用
```

数组第 0 个元素为'c'，第 1 个元素为'.'，第 2 个元素为'b'，后面的元素以此类推。

为了方便，你也可以不指定数组长度，从而写作：

```
1.  char str[] = { "c.biancheng.net" };
2.  char str[] = "c.biancheng.net";  //这种形式更加简洁，实际开发中常用
```

给字符数组赋值时，我们通常使用这种写法，将字符串一次性地赋值（可以指明数组长度，也可以不指明），而不是一个字符一个字符地赋值，那样做太麻烦了。

这里需要留意一个坑，字符数组只有在定义时才能将整个字符串一次性地赋值给它，一旦定义完了，就只能一个字符一个字符地赋值了。请看下面的例子：

```
1.  char str[7];
2.  str = "abc123";  //错误
3.  //正确
4.  str[0] = 'a'; str[1] = 'b'; str[2] = 'c';
5.  str[3] = '1'; str[4] = '2'; str[5] = '3';
```

字符串结束标志（划重点）

字符串是一系列连续的字符的组合，要想在内存中定位一个字符串，除了要知道它的开头，还要知道它的结尾。找到字符串的开头很容易，知道它的名字（字符数组名或者字符串名）就可以；然而，如何找到字符串的结尾呢？C 语言的解决方案有点奇妙，或者说有点奇葩。

在 C 语言中，字符串总是以'\0'作为结尾，所以'\0'也被称为字符串结束标志，或者字符串结束符。

> '\0'是 ASCII 码表中的第 0 个字符，英文称为 NUL，中文称为“空字符”。该字符既不能显示，也没有控制功能，输出该字符不会有任何效果，它在 C 语言中唯一的作用就是作为字符串结束标志。

C 语言在处理字符串时，会从前往后逐个扫描字符，一旦遇到'\0'就认为到达了字符串的末尾，就结束处理。'\0'至

关重要，没有'\0'就意味着永远也到达不了字符串的结尾。

由" "包围的字符串会自动在末尾添加'\0'。例如，"abc123"从表面看起来只包含了 6 个字符，其实不然，C 语言会在最后隐式地添加一个'\0'，这个过程是在后台默默地进行的，所以我们感受不到。

下图演示了"C program"在内存中的存储情形：

C		p	r	o	g	r	a	m	\0

需要注意的是，逐个字符地给数组赋值并不会自动添加'\0'，例如：

```
char str[] = {'a', 'b', 'c'};
```

数组 str 的长度为 3，而不是 4，因为最后没有'\0'。

当用字符数组存储字符串时，要特别注意'\0'，要为'\0'留个位置；这意味着，字符数组的长度至少要比字符串的长度大 1。请看下面的例子：

```
char str[7] = "abc123";
```

"abc123"看起来只包含了 6 个字符，我们却将 str 的长度定义为 7，就是为了能够容纳最后的'\0'。如果将 str 的长度定义为 6，它就无法容纳'\0'了。

> 当字符串长度大于数组长度时，有些较老或者不严格的编译器并不会报错，甚至连警告都没有，这就为以后的错误埋下了伏笔，读者自己要多多注意。

有些时候，程序的逻辑要求我们必须逐个字符地为数组赋值，这个时候就很容易遗忘字符串结束标志'\0'。下面的代码中，我们将 26 个大写英文字符存入字符数组，并以字符串的形式输出：

```
1.  #include <stdio.h>
2.  int main() {
3.      char str[30];
4.      char c;
5.      int i;
6.      for (c = 65, i = 0; c <= 90; c++, i++) {
7.          str[i] = c;
8.      }
9.      printf("%s\n", str);
10.
11.     return 0;
12. }
```

在 VS2015 下的运行结果：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ 口口口口 i 口口 0？
```

口表示无法显示的特殊字符。

大写字母在 ASCII 码表中是连续排布的，编码值从 65 开始，到 90 结束，使用循环非常方便。

在《C 语言变量的定义位置以及初始值》一节中我们讲到，在很多编译器下，局部变量的初始值是随机的，是垃圾

值，而不是我们通常认为的“零”值。局部数组（在函数内部定义的数组，本例中的 str 数组就是在 main() 函数内部定义的）也有这个问题，很多编译器并不会把局部数组的内存都初始化为“零”值，而是放任不管，爱是什么就是什么，所以它们的值也是没有意义的，也是垃圾值。

在函数内部定义的变量、数组、结构体、共用体等都称为局部数据。在很多编译器下，局部数据的初始值都是随机的、无意义的，而不是我们通常认为的“零”值。这一点非常重要，大家一定要谨记，否则后面会遇到很多奇葩的错误。

本例中的 str 数组在定义完成以后并没有立即初始化，所以它所包含的元素的值都是随机的，只有很小的概率会是“零”值。循环结束以后，str 的前 26 个元素被赋值了，剩下的 4 个元素的值依然是随机的，不知道是什么。

printf() 输出字符串时，会从第 0 个元素开始往后检索，直到遇见'\0'才停止，然后把'\0'前面的字符全部输出，这就是 printf() 输出字符串的原理。本例中我们使用 printf() 输出 str，按理说到了第 26 个元素就能检索到'\0'，就到达了字符串的末尾，然而事实却不是这样，由于我们并未对最后 4 个元素赋值，所以第 26 个元素不是'\0'，第 27 个也不是，第 28 个也不是……可能到了第 50 个元素才遇到'\0'，printf() 把这 50 个字符全部输出出来，就是上面的样子，多出来的字符毫无意义，甚至不能显示。

数组总共才 30 个元素，到了第 50 个元素不早就超出数组范围了吗？是的，的确超出范围了！然而，数组后面依然有其它的数据，printf() 也会将这些数据作为字符串输出。

你看，不注意'\0'的后果有多严重，不但不能正确处理字符串，甚至还会毁坏其它数据。

要想避免这些问题也很容易，在字符串的最后手动添加'\0'即可。修改上面的代码，在循环结束后添加'\0'：

```
1.  #include <stdio.h>
2.  int main() {
3.      char str[30];
4.      char c;
5.      int i;
6.      for (c = 65, i = 0; c <= 90; c++, i++) {
7.          str[i] = c;
8.      }
9.      str[i] = 0;  //此处为添加的代码，也可以写作 str[i] = '\0';
10.     printf("%s\n", str);
11.
12.     return 0;
13. }
```

第 9 行为新添加的代码，它让字符串能够正常结束。根据 ASCII 码表，字符'\0'的编码值就是 0。

但是，这样的写法貌似有点业余，或者说不够简洁，更加专业的做法是将数组的所有元素都初始化为“零”值，这样才能够从根本上避免问题。再次修改上面的代码：

```
1.  #include <stdio.h>
2.  int main() {
3.      char str[30] = { 0 };  //将所有元素都初始化为 0，或者说 '\0'
4.      char c;
```

```
5.      int i;
6.      for (c = 65, i = 0; c <= 90; c++, i++) {
7.          str[i] = c;
8.      }
9.      printf("%s\n", str);
10.
11.     return 0;
12. }
```

还记得《什么是数组》一节中强调过的吗？如果只初始化部分数组元素，那么剩余的数组元素也会自动初始化为“零”值，所以我们只需要将 str 的第 0 个元素赋值为 0，剩下的元素就都是 0 了。

字符串长度

所谓字符串长度，就是字符串包含了多少个字符（不包括最后的结束符'\0'）。例如"abc"的长度是 3，而不是 4。

在 C 语言中，我们使用 string.h 头文件中的 strlen() 函数来求字符串的长度，它的用法为：

```
length strlen(strname);
```

strname 是字符串的名字，或者字符数组的名字；length 是使用 strlen() 后得到的字符串长度，是一个整数。

下面是一个完整的例子，它输出《C 语言入门教程》网址的长度：

```
1.  #include <stdio.h>
2.  #include <string.h>  //记得引入该头文件
3.
4.  int main() {
5.      char str[] = "http://c.biancheng.net/c/";
6.      long len = strlen(str);
7.      printf("The lenth of the string is %ld.\n", len);
8.
9.      return 0;
10. }
```

运行结果：
The lenth of the string is 25.

6.5 C 语言字符串的输入和输出

其实在《C 语言输入输出》一章中我们已经提到了如何输入输出字符串，但是那个时候我们还没有讲解字符串，大家理解的可能不透彻，所以本节我们有必要再深入和细化一下。

字符串的输出

在 C 语言中，有两个函数可以在控制台（显示器）上输出字符串，它们分别是：

- puts()：输出字符串并自动换行，该函数只能输出字符串。
- printf()：通过格式控制符%s 输出字符串，不能自动换行。除了字符串，printf() 还能输出其他类型的数据。

这两个函数相信大家已经非常熟悉了，这里不妨再演示一下，请看下面的代码：

```
#include <stdio.h>
int main() {
    char str[] = "http://c.biancheng.net";
    printf("%s\n", str);  //通过字符串名字输出
    printf("%s\n", "http://c.biancheng.net");  //直接输出
    puts(str);  //通过字符串名字输出
    puts("http://c.biancheng.net");  //直接输出

    return 0;
}
```

运行结果：

http://c.biancheng.net
http://c.biancheng.net
http://c.biancheng.net
http://c.biancheng.net

注意，输出字符串时只需要给出名字，不能带后边的[]，例如，下面的两种写法都是错误的：

```
printf("%s\n", str[]);
puts(str[10]);
```

字符串的输入

在 C 语言中，有两个函数可以让用户从键盘上输入字符串，它们分别是：

- scanf()：通过格式控制符%s输入字符串。除了字符串，scanf() 还能输入其他类型的数据。
- gets()：直接输入字符串，并且只能输入字符串。

但是，scanf() 和 gets() 是有区别的：

- scanf() 读取字符串时以空格为分隔，遇到空格就认为当前字符串结束了，所以无法读取含有空格的字符串。
- gets() 认为空格也是字符串的一部分，只有遇到回车键时才认为字符串输入结束，所以，不管输入了多少个空格，只要不按下回车键，对 gets() 来说就是一个完整的字符串。换句话说，gets() 用来读取一整行字符串。

请看下面的例子：

```
#include <stdio.h>
int main() {
    char str1[30] = { 0 };
    char str2[30] = { 0 };
    char str3[30] = { 0 };

    //gets() 用法
    printf("Input a string: ");
    gets(str1);
```

```
10.
11.     //scanf() 用法
12.     printf("Input a string: ");
13.     scanf("%s", str2);
14.     scanf("%s", str3);
15.
16.     printf("\nstr1: %s\n", str1);
17.     printf("str2: %s\n", str2);
18.     printf("str3: %s\n", str3);
19.
20.     return 0;
21. }
```

运行结果：

```
Input a string: C C++ Java Python↙
Input a string: PHP JavaScript↙

str1: C C++ Java Python
str2: PHP
str3: JavaScript
```

第一次输入的字符串被 gets() 全部读取，并存入 str1 中。第二次输入的字符串，前半部分被第一个 scanf() 读取并存入 str2 中，后半部分被第二个 scanf() 读取并存入 str3 中。

注意，scanf() 在读取数据时需要的是数据的地址，这一点是恒定不变的，所以对于 int、char、float 等类型的变量都要在前边添加&以获取它们的地址。但是在本段代码中，我们只给出了字符串的名字，却没有在前边添加&，这是为什么呢？因为字符串名字或者数组名字在使用的过程中一般都会转换为地址，所以再添加&就是多此一举，甚至会导致错误了。

就目前学到的知识而言，int、char、float 等类型的变量用于 scanf() 时都要在前面添加&，而数组或者字符串用于 scanf() 时不用添加&，它们本身就会转换为地址。读者一定要谨记这一点。

至于数组名字（字符串名字）和地址的转换细节，以及数组名字什么时候会转换为地址，我们将在《数组到底在什么时候会转换为指针》一节中详细讲解，大家暂时“死记硬背”即可。

其实 scanf() 也可以读取带空格的字符串

以上是 scanf() 和 gets() 的一般用法，很多教材也是这样讲解的，所以大部分初学者都认为 scanf() 不能读取包含空格的字符串，不能替代 gets()。其实不然，scanf() 的用法还可以更加复杂和灵活，它不但可以完全替代 gets() 读取一整行字符串，而且比 gets() 的功能更加强大。比如，以下功能都是 gets() 不具备的：

- scanf() 可以控制读取字符的数目；
- scanf() 可以只读取指定的字符；
- scanf() 可以不读取某些字符；
- scanf() 可以把读取到的字符丢弃。

这些我们已经在《scanf 的高级用法，原来 scanf 还有这么多新技能》讲解过了，本节就不再赘述了。

6.6 C 语言字符串处理函数

C 语言提供了丰富的字符串处理函数，可以对字符串进行输入、输出、合并、修改、比较、转换、复制、搜索等操作，使用这些现成的函数可以大大减轻我们的编程负担。

用于输入输出的字符串函数，例如 printf、puts、scanf、gets 等，使用时要包含头文件 stdio.h，而使用其它字符串函数要包含头文件 string.h。

string.h 是一个专门用来处理字符串的头文件，它包含了很多字符串处理函数，由于篇幅限制，本节只能讲解几个常用的，有兴趣的读者请猛击这里查阅所有函数。

字符串连接函数 strcat()

strcat 是 string catenate 的缩写，意思是把两个字符串拼接在一起，语法格式为：

```
strcat(arrayName1, arrayName2);
```

arrayName1、arrayName2 为需要拼接的字符串。

strcat() 将把 arrayName2 连接到 arrayName1 后面，并删除原来 arrayName1 最后的结束标志'\0'。这意味着，arrayName1 必须足够长，要能够同时容纳 arrayName1 和 arrayName2，否则会越界（超出范围）。

strcat() 的返回值为 arrayName1 的地址。

下面是一个简单的演示：

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[100] = "The URL is ";
    char str2[60];
    printf("Input a URL: ");
    gets(str2);
    strcat(str1, str2);
    puts(str1);

    return 0;
}
```

运行结果：
Input a URL: http://c.biancheng.net/cpp/u/jiaocheng/↙
The URL is http://c.biancheng.net/cpp/u/jiaocheng/

字符串复制函数 strcpy()

strcpy 是 string copy 的缩写，意思是字符串复制，也即将字符串从一个地方复制到另外一个地方，语法格式为：

```
strcpy(arrayName1, arrayName2);
```

strcpy() 会把 arrayName2 中的字符串拷贝到 arrayName1 中，字符串结束标志'\0'也一同拷贝。请看下面的例子：

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[50] = "《C语言变怪兽》";
    char str2[50] = "http://c.biancheng.net/cpp/u/jiaocheng/";
    strcpy(str1, str2);
    printf("str1: %s\n", str1);

    return 0;
}
```

运行结果：

str1: http://c.biancheng.net/cpp/u/jiaocheng/

你看，将 str2 复制到 str1 后，str1 中原来的内容就被覆盖了。

另外，strcpy() 要求 arrayName1 要有足够的长度，否则不能全部装入所拷贝的字符串。

字符串比较函数 strcmp()

strcmp 是 string compare 的缩写，意思是字符串比较，语法格式为：

```
strcmp(arrayName1, arrayName2);
```

arrayName1 和 arrayName2 是需要比较的两个字符串。

字符本身没有大小之分，strcmp() 以各个字符对应的 ASCII 码值进行比较。strcmp() 从两个字符串的第 0 个字符开始比较，如果它们相等，就继续比较下一个字符，直到遇见不同的字符，或者到字符串的末尾。

返回值：若 arrayName1 和 arrayName2 相同，则返回 0；若 arrayName1 大于 arrayName2，则返回大于 0 的值；若 arrayName1 小于 arrayName2，则返回小于 0 的值。

对 4 组字符串进行比较：

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[] = "aBcDeF";
    char b[] = "AbCdEf";
    char c[] = "aacdef";
    char d[] = "aBcDeF";
    printf("a VS b: %d\n", strcmp(a, b));
    printf("a VS c: %d\n", strcmp(a, c));
    printf("a VS d: %d\n", strcmp(a, d));

```

```
12.     return 0;
13. }
```

运行结果：
a VS b: 32
a VS c: -31
a VS d: 0

6.7 C 语言数组是静态的，不能插入或删除元素

6.8 C 语言数组的越界和溢出

6.9 C 语言变长数组：使用变量指明数组的长度

6.10 C 语言对数组元素进行排序（冒泡排序法）

在实际开发中，有很多场景需要我们将数组元素按照从大到小（或者从小到大）的顺序排列，这样在查阅数据时会更加直观，例如：

- 一个保存了班级学号的数组，排序后更容易分区好学生和坏学生；
- 一个保存了商品单价的数组，排序后更容易看出它们的性价比。

对数组元素进行排序的方法有很多种，比如冒泡排序、归并排序、选择排序、插入排序、快速排序等，其中最经典最需要掌握的是「冒泡排序」。

以从小到大排序为例，冒泡排序的整体思想是这样的：

- 从数组头部开始，不断比较相邻的两个元素的大小，让较大的元素逐渐往后移动（交换两个元素的值），直到数组的末尾。经过第一轮的比较，就可以找到最大的元素，并将它移动到最后一个位置。

- 第一轮结束后，继续第二轮。仍然从数组头部开始比较，让较大的元素逐渐往后移动，直到数组的倒数第二个元素为止。经过第二轮的比较，就可以找到次大的元素，并将它放到倒数第二个位置。
- 以此类推，进行 n-1（n 为数组长度）轮“冒泡”后，就可以将所有的元素都排列好。

整个排序过程就好像气泡不断从水里冒出来，最大的先出来，次大的第二出来，最小的最后出来，所以将这种排序方式称为冒泡排序（Bubble Sort）。

下面我们以“3 2 4 1”为例对冒泡排序进行说明。

第一轮 排序过程
3 2 4 1 （最初）
2 3 4 1 （比较 3 和 2，交换）
2 3 4 1 （比较 3 和 4，不交换）
2 3 1 4 （比较 4 和 1，交换）
第一轮结束，最大的数字 4 已经在最后面，因此第二轮排序只需要对前面三个数进行比较。

第二轮 排序过程
2 3 1 4 （第一轮排序结果）
2 3 1 4 （比较 2 和 3，不交换）
2 1 3 4 （比较 3 和 1，交换）
第二轮结束，次大的数字 3 已经排在倒数第二个位置，所以第三轮只需要比较前两个元素。

第三轮 排序过程
2 1 3 4 （第二轮排序结果）
1 2 3 4 （比较 2 和 1，交换）

至此，排序结束。

算法总结及实现

对拥有 n 个元素的数组 R[n] 进行 n-1 轮比较。

第一轮，逐个比较 (R[1], R[2]), (R[2], R[3]), (R[3], R[4]), ……. (R[N-1], R[N])，最大的元素被移动到 R[n] 上。

第二轮，逐个比较 (R[1], R[2]), (R[2], R[3]), (R[3], R[4]), ……. (R[N-2], R[N-1])，次大的元素被移动到 R[n-1] 上。
。。。。。。
以此类推，直到整个数组从小到大排序。

具体的代码实现如下所示：

```
1.  #include <stdio.h>
2.  int main() {
3.      int nums[10] = { 4, 5, 2, 10, 7, 1, 8, 3, 6, 9 };
4.      int i, j, temp;
5.
6.      //冒泡排序算法：进行 n-1 轮比较
```

```
    for (i = 0; i<10 - 1; i++) {
        //每一轮比较前 n-1-i 个，也就是说，已经排序好的最后 i 个不用比较
        for (j = 0; j<10 - 1 - i; j++) {
            if (nums[j] > nums[j + 1]) {
                temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }

    //输出排序后的数组
    for (i = 0; i<10; i++) {
        printf("%d ", nums[i]);
    }
    printf("\n");

    return 0;
}
```

运行结果：

1 2 3 4 5 6 7 8 9 10

优化算法

上面的算法是大部分教材中提供的算法，其中有一点是可以优化的：当比较到第 i 轮的时候，如果剩下的元素已经排序好了，那么就不用再继续比较了，跳出循环即可，这样就减少了比较的次数，提高了执行效率。

未经优化的算法一定会进行 n-1 轮比较，经过优化的算法最多进行 n-1 轮比较，高下立判。

优化后的算法实现如下所示：

```
#include <stdio.h>
int main() {
    int nums[10] = { 4, 5, 2, 10, 7, 1, 8, 3, 6, 9 };
    int i, j, temp, isSorted;

    //优化算法：最多进行 n-1 轮比较
    for (i = 0; i<10 - 1; i++) {
        isSorted = 1;  //假设剩下的元素已经排序好了
        for (j = 0; j<10 - 1 - i; j++) {
            if (nums[j] > nums[j + 1]) {
                temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
                isSorted = 0;  //一旦需要交换数组元素，就说明剩下的元素没有排序好
```

```
15.                 }
16.             }
17.             if (isSorted) break; //如果没有发生交换，说明剩下的元素已经排序好了
18.         }
19.
20.         for (i = 0; i<10; i++) {
21.             printf("%d ", nums[i]);
22.         }
23.         printf("\n");
24.
25.         return 0;
26.     }
```

我们额外设置了一个变量 isSorted，用它作为标志，值为“真”表示剩下的元素已经排序好了，值为“假”表示剩下的元素还未排序好。

每一轮比较之前，我们预先假设剩下的元素已经排序好了，并将 isSorted 设置为“真”，一旦在比较过程中需要交换元素，就说明假设是错的，剩下的元素没有排序好，于是将 isSorted 的值更改为“假”。

每一轮循环结束后，通过检测 isSorted 的值就知道剩下的元素是否排序好。

6.11 对 C 语言数组的总结

数组（Array）是一系列相同类型的数据的集合，可以是一维的、二维的、多维的；最常用的是一维数组和二维数组，多维数组较少用到。

对数组的总结

1) 数组的定义格式为：

```
type arrayName[length]
```

type 为数据类型，arrayName 为数组名，length 为数组长度。需要注意的是：

- 数组长度 length 最好是常量表达式，例如 10、20*4 等，这样在所有编译器下都能运行通过；如果 length 中包含了变量，例如 n、4*m 等，在某些编译器下就会报错，我们已在《C 语言变长数组：使用变量指明数组的长度》一节专门讨论了这点。
- 数组是一个整体，它的内存是连续的；也就是说，数组元素之间是相互挨着的，彼此之间没有一点点缝隙。
- 一般情况下，数组名会转换为数组的地址，需要使用地址的地方，直接使用数组名即可。

2) 访问数组元素的格式为：

```
arrayName[index]
```

index 为数组下标。注意 index 的值必须大于等于零，并且小于数组长度，否则会发生数组越界，出现意想不到的错误，我们已在《C 语言数组的越界和溢出》一节重点讨论过。

3) 可以对数组中的单个元素赋值，也可以整体赋值，例如：

```
1.  // 对单个元素赋值
2.  int a[3];
3.  a[0] = 3;
4.  a[1] = 100;
5.  a[2] = 34;
6.
7.  // 整体赋值（不指明数组长度）
8.  float b[] = { 23.3, 100.00, 10, 0.34 };
9.  // 整体赋值（指明数组长度）
10. int m[10] = { 100, 30, 234 };
11.
12. // 字符数组赋值
13. char str1[] = "http://c.biancheng.net";
14.
15. // 将数组所有元素都初始化为0
16. int arr[10] = { 0 };
17. char str2[20] = { 0 };
```

4) 字符串是本章的重点内容，大家要特别注意字符串结束标志'\0'，各种字符串处理函数在定位字符串时都把'\0'作为结尾，没有'\0'就到达不了字符串的结尾。

关于查找和排序

学完了数组，有两项内容大家可以深入研究了，分别是查找（Search）和排序（Sort），它们在实际开发中都经常使用，比如：

- 给你 10 个打乱顺序的整数，要能够按照从小到大或者从大到小的顺序输出；
- 给定一个字符串 str1，以及一个子串 str2，要能够判断 str2 是否在 str1 中。

本章我们讲解了最简单的查找和排序算法，分别是顺序查找（遍历数组查找某个元素）和冒泡排序，这些都是最基本的，有兴趣的读者也可以深入研究，下面我给列出了几篇文章：

- C 语言快速排序算法以及代码
- C 语言选择排序算法以及代码
- C 语言插入排序算法及代码
- C 语言归并排序（合并排序）算法以及代码
- C 语言顺序查找算法以及代码
- C 语言二分查找（折半查找）算法以及代码

第 07 章 C 语言函数

函数就是一段封装好的，可以重复使用的代码，它使得我们的程序更加模块化，不需要编写大量重复的代码。

函数可以提前保存起来，并给它起一个独一无二的名字，只要知道它的名字就能使用这段代码。函数还可以接收数

据，并根据数据的不同做出不同的操作，最后再把处理结果反馈给我们。

本章目录：

1. 什么是函数？
2. C 语言函数定义（C 语言自定义函数）
3. C 语言形参和实参（非常详细）
4. C 语言函数返回值（return 关键字）精讲
5. C 语言函数调用详解（从中发现程序运行的秘密）
6. C 语言函数声明以及函数原型
7. C 语言全局变量和局部变量（带实例讲解）
8. C 语言变量的作用域（加深对全局变量和局部变量的理解）
9. C 语言块级变量（在代码块内部定义的变量）
10. C 语言递归函数（递归调用）详解[带实例演示]
11. C 语言中间递归函数（比较复杂的一种递归）
12. C 语言多层递归函数（最烧脑的一种递归）
13. 递归函数的致命缺陷：巨大的时间开销和内存开销（附带优化方案）
14. 忽略语法细节，从整体上理解函数

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

7.1 什么是函数？

从表面上看，函数在使用时必须带上括号，有必要的话还要传递参数，函数的执行结果也可以赋值给其它变量。例如，strcmp() 是一个用来比较字符串大小的函数，它的用法如下：

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[] = "http://c.biancheng.net";
    char str2[] = "http://www.baidu.com";
    //比较两个字符串大小
    int result = strcmp(str1, str2);
    printf("str1  - str2 = %d\n", result);

    return 0;
}
```

str1 和 str2 是传递给 strcmp() 的参数，strcmp() 的处理结果赋值给了变量 result。

我们不妨设想一下，如果没有 strcmp() 函数，要想比较两个字符串的大小该怎么写呢？请看下面的代码：

```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[] = "http://c.biancheng.net";
    char str2[] = "http://www.baidu.com";
    int result, i;
```

```
7.      //比较两个字符串大小
8.      for (i = 0; (result = str1[i] - str2[i]) == 0; i++) {
9.          if (str1[i] == '\0' || str2[i] == '\0') {
10.             break;
11.         }
12.     }
13.
14.     printf("str1  - str2 = %d\n", result);
15.     return 0;
16. }
```

比较字符串大小是常用的功能，一个程序可能会用到很多次，如果每次都写这样一段重复的代码，不但费时费力、容易出错，而且交给别人时也很麻烦，所以 C 语言提供了一个功能，允许我们将常用的代码以固定的格式封装（包装）成一个独立的模块，只要知道这个模块的名字就可以重复使用它，这个模块就叫做**函数（Function）**。

函数的本质是一段可以重复使用的代码，这段代码被提前编写好了，放到了指定的文件中，使用时直接调取即可。下面我们就来演示一下如何封装 strcmp() 这个函数。

```
1.  #include <stdio.h>
2.
3.  //将比较字符串大小的代码封装成函数，并命名为strcmp_alias
4.  int strcmp_alias(char *s1, char *s2) {
5.      int i, result;
6.      for (i = 0; (result = s1[i] - s2[i]) == 0; i++) {
7.          if (s1[i] == '\0' || s2[i] == '\0') {
8.              break;
9.          }
10.     }
11.
12.     return result;
13. }
14.
15. int main() {
16.     char str1[] = "http://c.biancheng.net";
17.     char str2[] = "http://www.baidu.com";
18.     char str3[] = "http://data.biancheng.net";
19.     //重复使用strcmp_alias()函数
20.     int result_1_2 = strcmp_alias(str1, str2);
21.     int result_1_3 = strcmp_alias(str1, str3);
22.     printf("str1  - str2 = %d\n", result_1_2);
23.     printf("str1  - str3 = %d\n", result_1_3);
24.
25.     return 0;
26. }
```

为了避免与原有的 strcmp 产生命名冲突，我将新函数命名为 strcmp_alias。

这是我们自己编写的函数，放在了当前源文件中（函数封装和函数使用在同一个源文件中），所以不需要引入头文件；而 C 语言自带的 strcmp() 放在了其它的源文件中（函数封装和函数使用不在同一个源文件中），并在 string.h 头文件中告诉我们如何使用，所以我们必须引入 string.h 头文件。

我们自己编写的 strcmp_alias() 和原有的 strcmp() 在功能和格式上都是一样的，只是存放的位置不同，所以一个需要引入头文件，一个不需要引入。

本章我们重点讲解的内容就是如何将一段代码封装成函数，以及封装以后如何使用。

C 语言中的函数和数学中的函数

美国人将函数称为“Function”。Function 除了有“函数”的意思，还有“功能”的意思，中国人将 Function 译为“函数”而不是“功能”，是因为 C 语言中的函数和数学中的函数在使用形式上有些类似，例如：

- C 语言中有 length = strlen(str)
- 数学中有 y = f(x)

你看它们是何其相似，都是通过一定的操作或规则，由一份数据得到另一份数据。

不过从本质上看，将 Function 理解为“功能”或许更恰当，C 语言中的函数往往是独立地实现了某项功能。一个程序由多个函数组成，可以理解为「一个程序由多个小的功能叠加而成」。

本教程重在实践，不咬文嚼字，不死扣概念，大家理解即可，不必在此深究。

库函数和自定义函数

C 语言在发布时已经为我们封装好了很多函数，它们被分门别类地放到了不同的头文件中（暂时先这样认为），使用函数时引入对应的头文件即可。这些函数都是专家编写的，执行效率极高，并且考虑到了各种边界情况，各位读者请放心使用。

C 语言自带的函数称为库函数（Library Function）。库（Library）是编程中的一个基本概念，可以简单地认为它是一系列函数的集合，在磁盘上往往是一个文件夹。C 语言自带的库称为标准库（Standard Library），其他公司或个人开发的库称为第三方库（Third-Party Library）。

> 关于库的概念，我们已在《不要这样学习 C 语言，这是一个坑！》中进行了详细介绍。

除了库函数，我们还可以编写自己的函数，拓展程序的功能。自己编写的函数称为自定义函数。自定义函数和库函数在编写和使用方式上完全相同，只是由不同的机构来编写。

参数

函数的一个明显特征就是使用时带括号()，有必要的话，括号中还要包含数据或变量，称为参数（Parameter）。参数是函数需要处理的数据，例如：

- strlen(str1)用来计算字符串的长度，str1 就是参数。
- puts("C 语言中文网")用来输出字符串，"C 语言中文网"就是参数。

返回值

既然函数可以处理数据，那就有必要将处理结果告诉我们，所以很多函数都有返回值（Return Value）。所谓返回值，就是函数的执行结果。例如：

```
char str1[] = "C Language";
int len = strlen(str1);
```

strlen() 的处理结果是字符串 str1 的长度，是一个整数，我们通过 len 变量来接收。

函数返回值有固定的数据类型（int、char、float 等），用来接收返回值的变量类型要一致。

7.2 C 语言函数定义（C 语言自定义函数）

函数是一段可以重复使用的代码，用来独立地完成某个功能，它可以接收用户传递的数据，也可以不接收。接收用户数据的函数在定义时要指明参数，不接收用户数据的不需要指明，根据这一点可以将函数分为有参函数和无参函数。

将代码段封装成函数的过程叫做**函数定义**。

C 语言无参函数的定义

如果函数不接收用户传递的数据，那么定义时可以不带参数。如下所示：

```
dataType  functionName(){
    //body
}
```

- dataType 是返回值类型，它可以是 C 语言中的任意数据类型，例如 int、float、char 等。
- functionName 是函数名，它是标识符的一种，命名规则和标识符相同。函数名后面的括号()不能少。
- body 是函数体，它是函数需要执行的代码，是函数的主体部分。即使只有一个语句，函数体也要由{}包围。
- 如果有返回值，在函数体中使用 return 语句返回。return 出来的数据的类型要和 dataType 一样。

例如，定义一个函数，计算从 1 加到 100 的结果：

```
1.  int sum() {
2.      int i, sum = 0;
3.      for (i = 1; i <= 100; i++) {
4.          sum += i;
5.      }
6.      return sum;
7.  }
```

累加结果保存在变量 sum 中，最后通过 return 语句返回。sum 是 int 型，返回值也是 int 类型，它们一一对应。

return 是 C 语言中的一个关键字，只能用在函数中，用来返回处理结果。

将上面的代码补充完整：

```
#include <stdio.h>

int sum() {
    int i, sum = 0;
    for (i = 1; i <= 100; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int a = sum();
    printf("The sum is %d\n", a);
    return 0;
}
```

运行结果：
The sum is 5050

函数不能嵌套定义，main 也是一个函数定义，所以要将 sum 放在 main 外面。函数必须先定义后使用，所以 sum 要放在 main 前面。

> 注意：main 是函数定义，不是函数调用。当可执行文件加载到内存后，系统从 main 函数开始执行，也就是说，系统会调用我们定义的 main 函数。

无返回值函数

有的函数不需要返回值，或者返回值类型不确定（很少见），那么可以用 void 表示，例如：

```
void hello() {
    printf("Hello,world \n");
    //没有返回值就不需要 return 语句
}
```

void 是 C 语言中的一个关键字，表示“空类型”或“无类型”，绝大部分情况下也就意味着没有 return 语句。

C 语言有参函数的定义

如果函数需要接收用户传递的数据，那么定义时就要带上参数。如下所示：

```
dataType  functionName( dataType1 param1, dataType2 param2 ... ){
    //body
}
```

dataType1 param1, dataType2 param2 ...是参数列表。函数可以只有一个参数，也可以有多个，多个参数之间由,分隔。参数本质上也是变量，定义时要指明类型和名称。与无参函数的定义相比，有参函数的定义仅仅是多了一个参数列表。

数据通过参数传递到函数内部进行处理，处理完成以后再通过返回值告知函数外部。

更改上面的例子，计算从 m 加到 n 的结果：

```
int sum(int m, int n) {
    int i, sum = 0;
    for (i = m; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

参数列表中给出的参数可以在函数体中使用，使用方式和普通变量一样。

调用 sum() 函数时，需要给它传递两份数据，一份传递给 m，一份传递给 n。你可以直接传递整数，例如：

```
int result = sum(1, 100);  //1 传递给 m，100 传递给 n
```

也可以传递变量：

```
int begin = 4;
int end = 86;
int result = sum(begin, end);  //begin 传递给 m，end 传递给 n
```

也可以整数和变量一起传递：

```
int num = 33;
int result = sum(num, 80);  //num 传递给 m，80 传递给 n
```

函数定义时给出的参数称为形式参数，简称形参；函数调用时给出的参数（也就是传递的数据）称为实际参数，简称实参。函数调用时，将实参的值传递给形参，相当于一次赋值操作。

原则上讲，实参的类型和数目要与形参保持一致。如果能够进行自动类型转换，或者进行了强制类型转换，那么实参类型也可以不同于形参类型，例如将 int 类型的实参传递给 float 类型的形参就会发生自动类型转换。

将上面的代码补充完整：

```
#include <stdio.h>

int sum(int m, int n) {
    int i, sum = 0;
    for (i = m; i <= n; i++) {
        sum += i;
    }
    return sum;
}

int main() {
    int begin = 5, end = 86;
```

```
13.     int result = sum(begin, end);
14.     printf("The sum from %d to %d is %d\n", begin, end, result);
15.     return 0;
16. }
```

运行结果：

The sum from 5 to 86 is 3731

定义 sum() 时，参数 m、n 的值都是未知的；调用 sum() 时，将 begin、end 的值分别传递给 m、n，这和给变量赋值的过程是一样的，它等价于：

```
m = begin;
n = end;
```

函数不能嵌套定义

强调一点，C 语言不允许函数嵌套定义；也就是说，不能在一个函数中定义另外一个函数，必须在所有函数之外定义另外一个函数。main() 也是一个函数定义，也不能在 main() 函数内部定义新函数。

下面的例子是错误的：

```
1.  #include <stdio.h>
2.
3.  void func1() {
4.      printf("http://c.biancheng.net");
5.
6.      void func2() {
7.          printf("C语言小白变怪兽");
8.      }
9.  }
10.
11. int main() {
12.     func1();
13.     return 0;
14. }
```

有些初学者认为，在 func1() 内部定义 func2()，那么调用 func1() 时也就调用了 func2()，这是错误的。

正确的写法应该是这样的：

```
1.  #include <stdio.h>
2.
3.  void func2() {
4.      printf("C语言小白变怪兽");
5.  }
6.
7.  void func1() {
8.      printf("http://c.biancheng.net");
```

```
9.      func2();
10. }
11.
12. int main() {
13.     func1();
14.     return 0;
15. }
```

func1()、func2()、main() 三个函数是平行的，谁也不能位于谁的内部，要想达到「调用 func1() 时也调用 func2()」的目的，必须将 func2() 定义在 func1() 外面，并在 func1() 内部调用 func2()。

> 有些编程语言是允许函数嵌套定义的，例如 JavaScript，在 JavaScript 中经常会使用函数的嵌套定义。

7.3 函数的形参和实参（非常详细）

如果把函数比喻成一台机器，那么参数就是原材料，返回值就是最终产品；从一定程度上讲，函数的作用就是根据不同的参数产生不同的返回值。

这一节我们先来讲解 C 语言函数的参数，下一节再讲解 C 语言函数的返回值。

C 语言函数的参数会出现在两个地方，分别是函数定义处和函数调用处，这两个地方的参数是有区别的。

形参（形式参数）

在函数定义中出现的参数可以看做是一个占位符，它没有数据，只能等到函数被调用时接收传递进来的数据，所以称为**形式参数**，简称**形参**。

实参（实际参数）

函数被调用时给出的参数包含了实实在在的数据，会被函数内部的代码使用，所以称为**实际参数**，简称**实参**。

形参和实参的功能是传递数据，发生函数调用时，实参的值会传递给形参。

形参和实参的区别和联系

1) 形参变量只有在函数被调用时才会分配内存，调用结束后，立刻释放内存，所以形参变量只有在函数内部有效，不能在函数外部使用。

2) 实参可以是常量、变量、表达式、函数等，无论实参是何种类型的数据，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参，所以应该提前用赋值、输入等办法使实参获得确定值。

3) 实参和形参在数量上、类型上、顺序上必须严格一致，否则会发生“类型不匹配”的错误。当然，如果能够进行自动类型转换，或者进行了强制类型转换，那么实参类型也可以不同于形参类型。

4) 函数调用中发生的数据传递是单向的，只能把实参的值传递给形参，而不能把形参的值反向地传递给实参；换句话说，一旦完成数据的传递，实参和形参就再也没有瓜葛了，所以，在函数调用过程中，形参的值发生改变并不会影响实参。

请看下面的例子：

```
#include <stdio.h>

//计算从m加到n的值
int sum(int m, int n) {
    int i;
    for (i = m + 1; i <= n; ++i) {
        m += i;
    }
    return m;
}

int main() {
    int a, b, total;
    printf("Input two numbers: ");
    scanf("%d %d", &a, &b);
    total = sum(a, b);
    printf("a=%d, b=%d\n", a, b);
    printf("total=%d\n", total);

    return 0;
}
```

运行结果：

Input two numbers: 1 100↙

a=1, b=100

total=5050

在这段代码中，函数定义处的 m、n 是形参，函数调用处的 a、b 是实参。通过 scanf() 可以读取用户输入的数据，并赋值给 a、b，在调用 sum() 函数时，这份数据会传递给形参 m、n。

从运行情况看，输入 a 值为 1，即实参 a 的值为 1，把这个值传递给函数 sum() 后，形参 m 的初始值也为 1，在函数执行过程中，形参 m 的值变为 5050。函数运行结束后，输出实参 a 的值仍为 1，可见实参的值不会随形参的变化而变化。

以上调用 sum() 时是将变量作为函数实参，除此以外，你也可以将常量、表达式、函数返回值作为实参，如下所示：

```
total = sum(10, 98);  //将常量作为实参
total = sum(a + 10, b - 3);  //将表达式作为实参
total = sum(pow(2, 2), abs(-100));  //将函数返回值作为实参
```

5) 形参和实参虽然可以同名，但它们之间是相互独立的，互不影响，因为实参在函数外部有效，而形参在函数内部有效。

更改上面的代码，让实参和形参同名：

```
#include <stdio.h>

//计算从m加到n的值
int sum(int m, int n) {
    int i;
    for (i = m + 1; i <= n; ++i) {
        m += i;
    }
    return m;
}

int main() {
    int m, n, total;
    printf("Input two numbers: ");
    scanf("%d %d", &m, &n);
    total = sum(m, n);
    printf("m=%d, n=%d\n", m, n);
    printf("total=%d\n", total);

    return 0;
}
```

运行结果：

Input two numbers: 1 100

m=1, n=100

total=5050

调用 sum() 函数后，函数内部的形参 m 的值已经发生了变化，而函数外部的实参 m 的值依然保持不变，可见它们是相互独立的两个变量，除了传递参数的一瞬间，其它时候是没有瓜葛的。

74. 函数返回值（return 关键字）精讲

函数的返回值是指函数被调用之后，执行函数体中的代码所得到的结果，这个结果通过 **return** 语句返回。

return 语句的一般形式为：

```
return 表达式;
```

或者：

```
return (表达式);
```

有没有()都是正确的，为了简明，一般也不写()。例如：

```
return max;
return a+b;
return (100+200);
```

对 C 语言返回值的说明：

1) 没有返回值的函数为空类型，用 void 表示。例如：

```
void func() {
    printf("http://c.biancheng.net\n");
}
```

一旦函数的返回值类型被定义为 void，就不能再接收它的值了。例如，下面的语句是错误的：

```
int a = func();
```

为了使程序有良好的可读性并减少出错，凡不要求返回值的函数都应定义为 void 类型。

2) return 语句可以有多个，可以出现在函数体的任意位置，但是每次调用函数只能有一个 return 语句被执行，所以只有一个返回值（少数的编程语言支持多个返回值，例如 Go 语言）。例如：

```
//返回两个整数中较大的一个
int max(int a, int b) {
    if (a > b) {
        return a;
    }
    else {
        return b;
    }
}
```

如果 a>b 成立，就执行 return a，return b 不会执行；如果不成立，就执行 return b，return a 不会执行。

3) 函数一旦遇到 return 语句就立即返回，后面的所有语句都不会被执行到了。从这个角度看，return 语句还有强制结束函数执行的作用。例如：

```
//返回两个整数中较大的一个
int max(int a, int b) {
    return (a>b) ? a : b;
    printf("Function is performed\n");
}
```

第 4 行代码就是多余的，永远没有执行的机会。

下面我们定义了一个判断素数的函数，这个例子更加实用：

```
#include <stdio.h>

int prime(int n) {
    int is_prime = 1, i;

    //n一旦小于0就不符合条件，就没必要执行后面的代码了，所以提前结束函数
    if (n < 0) { return -1; }

    for (i = 2; i<n; i++) {
        if (n % i == 0) {
```

```
11.                 is_prime = 0;
12.                 break;
13.             }
14.         }
15.
16.         return is_prime;
17.     }
18.
19.     int main() {
20.         int num, is_prime;
21.         scanf("%d", &num);
22.
23.         is_prime = prime(num);
24.         if (is_prime < 0) {
25.             printf("%d is a illegal number.\n", num);
26.         }else if (is_prime > 0) {
27.             printf("%d is a prime number.\n", num);
28.         }else {
29.             printf("%d is not a prime number.\n", num);
30.         }
31.
32.         return 0;
33.     }
```

prime() 是一个用来求素数的函数。素数是自然数，它的值大于等于零，一旦传递给 prime() 的值小于零就没有意义了，就无法判断是否是素数了，所以一旦检测到参数 n 的值小于 0，就使用 return 语句提前结束函数。

return 语句是提前结束函数的唯一办法。return 后面可以跟一份数据，表示将这份数据返回到函数外面；return 后面也可以不跟任何数据，表示什么也不返回，仅仅用来结束函数。

更改上面的代码，使得 return 后面不跟任何数据：

```
1.  #include <stdio.h>
2.
3.  void prime(int n){
4.      int is_prime = 1, i;
5.
6.      if(n < 0){
7.          printf("%d is a illegal number.\n", n);
8.          return;  //return后面不带任何数据
9.      }
10.
11.     for(i=2; i<n; i++){
12.         if(n % i == 0){
13.             is_prime = 0;
14.             break;
```

```
15.         }
16.     }
17.
18.     if(is_prime > 0){
19.         printf("%d is a prime number.\n", n);
20.     }else{
21.         printf("%d is not a prime number.\n", n);
22.     }
23. }
24.
25. int main(){
26.     int num;
27.     scanf("%d", &num);
28.     prime(num);
29.
30.     return 0;
31. }
```

prime() 的返回值是 void，return 后面不能带任何数据，直接写分号即可。

7.5 函数调用详解（从中发现程序运行的秘密）

所谓**函数调用（Function Call）**，就是使用已经定义好的函数。函数调用的一般形式为：

```
functionName(param1, param2, param3 ...);
```

functionName 是函数名称，param1, param2, param3 ...是实参列表。实参可以是常数、变量、表达式等，多个实参用逗号,分隔。

在 C 语言中，函数调用的方式有多种，例如：

```
1.  //函数作为表达式中的一项出现在表达式中
2.  z = max(x, y);
3.  m = n + max(x, y);
4.  //函数作为一个单独的语句
5.  printf("%d", a);
6.  scanf("%d", &b);
7.  //函数作为调用另一个函数时的实参
8.  printf( "%d", max(x, y) );
9.  total( max(x, y), min(m, n) );
```

函数的嵌套调用

函数不能嵌套定义，但可以嵌套调用，也就是在一个函数的定义或调用过程中允许出现对另外一个函数的调用。

【示例】计算 sum = 1! + 2! + 3! + ... + (n-1)! + n!

分析：可以编写两个函数，一个用来计算阶乘，一个用来计算累加的和。

```
#include <stdio.h>

//求阶乘
long factorial(int n){
    int i;
    long result=1;
    for(i=1; i<=n; i++){
        result *= i;
    }
    return result;
}

// 求累加的和
long sum(long n){
    int i;
    long result = 0;
    for(i=1; i<=n; i++){
        //在定义过程中出现嵌套调用
        result += factorial(i);
    }
    return result;
}

int main(){
    printf("1!+2!+...+9!+10! = %ld\n", sum(10));  //在调用过程中出现嵌套调用
    return 0;
}
```

运行结果：

1!+2!+...+9!+10! = 4037913

sum() 的定义中出现了对 factorial() 的调用，printf() 的调用过程中出现了对 sum() 的调用，而 printf() 又被 main() 调用，它们整体调用关系为：

```
main() --> printf() --> sum() --> factorial()
```

如果一个函数 A() 在定义或调用过程中出现了对另外一个函数 B() 的调用，那么我们就称 A() 为主调函数或主函数，称 B() 为被调函数。

当主调函数遇到被调函数时，主调函数会暂停，CPU 转而执行被调函数的代码；被调函数执行完毕后再返回主调函数，主调函数根据刚才的状态继续往下执行。

一个 C 语言程序的执行过程可以认为是多个函数之间的相互调用过程，它们形成了一个或简单或复杂的调用链条。这个链条的起点是 main()，终点也是 main()。当 main() 调用完了所有的函数，它会返回一个值（例如 `return 0;`）来结束自己的生命，从而结束整个程序。

函数是一个可以重复使用的代码块，CPU 会一条一条地挨着执行其中的代码，当遇到函数调用时，CPU 首先要记录下当前代码块中下一条代码的地址（假设地址为 0X1000），然后跳转到另外一个代码块，执行完毕后再回来继续执行 0X1000 处的代码。整个过程相当于 CPU 开了一个小差，暂时放下手中的工作去做点别的事情，做完了再继续刚才的工作。

从上面的分析可以推断出，在所有函数之外进行加减乘除运算、使用 if...else 语句、调用一个函数等都是没有意义的，这些代码位于整个函数调用链条之外，永远都不会被执行到。C 语言也禁止出现这种情况，会报语法错误，请看下面的代码：

```
#include <stdio.h>

int a = 10, b = 20, c;
//错误：不能出现加减乘除运算
c = a + b;

//错误：不能出现对其他函数的调用
printf("c.biancheng.net");

int main(){
    return 0;
}
```

7.6 函数声明以及函数原型

C 语言代码由上到下依次执行，原则上函数定义要出现在函数调用之前，否则就会报错。但在实际开发中，经常会在函数定义之前使用它们，这个时候就需要提前声明。

所谓**声明（Declaration）**，就是告诉编译器我要使用这个函数，你现在没有找到它的定义不要紧，请不要报错，稍后我会把定义补上。

函数声明的格式非常简单，相当于去掉函数定义中的函数体，并在最后加上分号;，如下所示：

```
dataType  functionName( dataType1 param1, dataType2 param2 ... );
```

也可以不写形参，只写数据类型：

```
dataType  functionName( dataType1, dataType2 ... );
```

函数声明给出了函数名、返回值类型、参数列表（重点是参数类型）等与该函数有关的信息，称为**函数原型（Function Prototype）**。函数原型的作用是告诉编译器与该函数有关的信息，让编译器知道函数的存在，以及存在的形式，即使函数暂时没有定义，编译器也知道如何使用它。

有了函数声明，函数定义就可以出现在任何地方了，甚至是其他文件、静态链接库、动态链接库等。

【实例 1】定义一个函数 sum()，计算从 m 加到 n 的和，并将 sum() 的定义放到 main() 后面。

```
#include <stdio.h>

```

```
//函数声明
int sum(int m, int n);  //也可以写作int sum(int, int);

int main(){
    int begin = 5, end = 86;
    int result = sum(begin, end);
    printf("The sum from %d to %d is %d\n", begin, end, result);
    return 0;
}

//函数定义
int sum(int m, int n){
    int i, sum=0;
    for(i=m; i<=n; i++){
        sum+=i;
    }
    return sum;
}
```

我们在 main() 函数中调用了 sum() 函数，编译器在它前面虽然没有发现函数定义，但是发现了函数声明，这样编译器就知道函数怎么使用了，至于函数体到底是什么，暂时可以不用操心，后续再把函数体补上就行。

【实例 2】定义两个函数，计算 1! + 2! + 3! + … + (n-1)! + n!的和。

```
#include <stdio.h>

// 函数声明部分
long factorial(int n);  //也可以写作 long factorial(int);
long sum(long n);  //也可以写作 long sum(long);

int main(){
    printf("1!+2!+...+9!+10! = %ld\n", sum(10));
    return 0;
}

//函数定义部分
//求阶乘
long factorial(int n){
    int i;
    long result=1;
    for(i=1; i<=n; i++){
        result *= i;
    }
    return result;
}
```

```
// 求累加的和
long sum(long n){
    int i;
    long result = 0;
    for(i=1; i<=n; i++){
        result += factorial(i);
    }
    return result;
}
```

运行结果：
1!+2!+...+9!+10! = 4037913

初学者编写的代码都比较简单，顶多几百行，完全可以放在一个源文件中。对于单个源文件的程序，通常是将函数定义放到 main() 的后面，将函数声明放到 main() 的前面，这样就使得代码结构清晰明了，主次分明。

使用者往往只关心函数的功能和函数的调用形式，很少关心函数的实现细节，将函数定义放在最后，就是尽量屏蔽不重要的信息，凸显关键的信息。将函数声明放到 main() 的前面，在定义函数时也不用关注它们的调用顺序了，哪个函数先定义，哪个函数后定义，都无所谓了。

然而在实际开发中，往往都是几千行、上万行、百万行的代码，将这些代码都放在一个源文件中简直是灾难，不但检索麻烦，而且打开文件也很慢，所以必须将这些代码分散到多个文件中。对于多个文件的程序，通常是将函数定义放到源文件（.c 文件）中，将函数的声明放到头文件（.h 文件）中，使用函数时引入对应的头文件就可以，编译器会在链接阶段找到函数体。

前面我们在使用 printf()、puts()、scanf() 等函数时引入了 stdio.h 头文件，很多初学者认为 stdio.h 中包含了函数定义（也就是函数体），只要有了头文件就能运行，其实不然，头文件中包含的都是函数声明，而不是函数定义，函数定义都放在了其它的源文件中，这些源文件已经提前编译好了，并以动态链接库或者静态链接库的形式存在，只有头文件没有系统库的话，在链接阶段就会报错，程序根本不能运行。

关于编译链接的原理，以及如果将代码分散到多个文件中，我们将在《C 语言多文件编程》专题中详细讲解。

除了函数，变量也有定义和声明之分。实际开发过程中，变量定义需要放在源文件（.c 文件）中，变量声明需要放在头文件（.h 文件）中，在链接程序时会将它们对应起来，这些我们也将在《C 语言多文件编程》专题中详细讲解。

学完《C 语言多文件编程》，你对 C 语言的认识将会有质的提升，瞬间豁然开朗，轻松超越 90% 的 C 语言程序员。

函数参考手册

最后再补充一点，函数原型给出了使用该函数的所有细节，当我们不知道如何使用某个函数时，需要查找的是它的原型，而不是它的定义，我们往往不关心它的实现。

www.cplusplus.com 是一个非常给力的网站，它提供了所有 C 语言标准函数的原型，并给出了详细的介绍和使用示例，可以作为一部权威的参考手册。

7.7 全局变量和局部变量（带实例讲解）

在《C 语言形参和实参的区别》中提到，形参变量要等到函数被调用时才分配内存，调用结束后立即释放内存。这说明形参变量的作用域非常有限，只能在函数内部使用，离开该函数就无效了。所谓作用域（Scope），就是变量的有效范围。

不仅对于形参变量，C 语言中所有的变量都有自己的作用域。决定变量作用域的是变量的定义位置。

局部变量

定义在函数内部的变量称为**局部变量（Local Variable）**，它的作用域仅限于函数内部，离开该函数后就是无效的，再使用就会报错。例如：

```
1.  int f1(int a){
2.      int b,c;  //a,b,c仅在函数f1()内有效
3.      return a+b+c;
4.  }
5.  int main(){
6.      int m,n;  //m,n仅在函数main()内有效
7.      return 0;
8.  }
```

几点说明：

1) 在 main 函数中定义的变量也是局部变量，只能在 main 函数中使用；同时，main 函数中也不能使用其它函数中定义的变量。main 函数也是一个函数，与其它函数地位平等。

2) 形参变量、在函数体内定义的变量都是局部变量。实参给形参传值的过程也就是给局部变量赋值的过程。

3) 可以在不同的函数中使用相同的变量名，它们表示不同的数据，分配不同的内存，互不干扰，也不会发生混淆。

4) 在语句块中也可定义变量，它的作用域只限于当前语句块。

全局变量

在所有函数外部定义的变量称为**全局变量（Global Variable）**，它的作用域默认是整个程序，也就是所有的源文件，包括 .c 和 .h 文件。例如：

```
1.  int a, b;  //全局变量
2.  void func1(){
3.      //TODO:
4.  }
5.
6.  float x,y;  //全局变量
7.  int func2(){
8.      //TODO:
9.  }
10.
```

```
int main(){
    //TODO:
    return 0;
}
```

a、b、x、y 都是在函数外部定义的全局变量。C 语言代码是从前往后依次执行的，由于 x、y 定义在函数 func1() 之后，所以在 func1() 内无效；而 a、b 定义在源程序的开头，所以在 func1()、func2() 和 main() 内都有效。

局部变量和全局变量的综合示例

【示例 1】输出变量的值：

```
#include <stdio.h>

int n = 10;  //全局变量

void func1(){
    int n = 20;  //局部变量
    printf("func1 n: %d\n", n);
}

void func2(int n){
    printf("func2 n: %d\n", n);
}

void func3(){
    printf("func3 n: %d\n", n);
}

int main(){
    int n = 30;  //局部变量
    func1();
    func2(n);
    func3();
    //代码块由{}包围
    {
        int n = 40;  //局部变量
        printf("block n: %d\n", n);
    }
    printf("main n: %d\n", n);

    return 0;
}
```

运行结果：

```
func1 n: 20
func2 n: 30
```

```
func3 n: 10
block n: 40
main n: 30
```

代码中虽然定义了多个同名变量 n，但它们的作用域不同，在内存中的位置（地址）也不同，所以是相互独立的变量，互不影响，不会产生重复定义（Redefinition）错误。

1) 对于 func1()，输出结果为 20，显然使用的是函数内部的 n，而不是外部的 n；func2() 也是相同的情况。

当全局变量和局部变量同名时，在局部范围内全局变量被“屏蔽”，不再起作用。或者说，变量的使用遵循就近原则，如果在当前作用域中存在同名变量，就不会向更大的作用域中去寻找变量。

2) func3() 输出 10，使用的是全局变量，因为在 func3() 函数中不存在局部变量 n，所以编译器只能到函数外部，也就是全局作用域中去寻找变量 n。

3) 由{}包围的代码块也拥有独立的作用域，printf() 使用它自己内部的变量 n，输出 40。

4) C 语言规定，只能从小的作用域向大的作用域中去寻找变量，而不能反过来，使用更小的作用域中的变量。对于 main() 函数，即使代码块中的 n 离输出语句更近，但它仍然会使用 main() 函数开头定义的 n，所以输出结果是 30。

【示例 2】根据长方体的长宽高求它的体积以及三个面的面积。

```
#include <stdio.h>

int s1, s2, s3;  //面积

int vs(int a, int b, int c){
    int v;  //体积
    v = a * b * c;
    s1 = a * b;
    s2 = b * c;
    s3 = a * c;
    return v;
}

int main(){
    int v, length, width, height;
    printf("Input length, width and height: ");
    scanf("%d %d %d", &length, &width, &height);
    v = vs(length, width, height);
    printf("v=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);

    return 0;
}
```

运行结果：

```
Input length, width and height: 10 20 30↙
v=6000, s1=200, s2=600, s3=300
```

根据题意，我们希望借助一个函数得到三个值：体积 v 以及三个面的面积 s1、s2、s3。遗憾的是，C 语言中的函数只能有一个返回值，我们只能将其中的一份数据，也就是体积 v 放到返回值中，而将面积 s1、s2、s3 设置为全局变量。全局变量的作用域是整个程序，在函数 vs() 中修改 s1、s2、s3 的值，能够影响到包括 main() 在内的其它函数。

7.8 C 语言变量的作用域（加深对全局变量和局部变量的理解）

所谓**作用域（Scope）**，就是变量的有效范围，就是变量可以在哪个范围以内使用。有些变量可以在所有代码文件中使用，有些变量只能在当前的文件中使用，有些变量只能在函数内部使用，有些变量只能在 for 循环内部使用。

变量的作用域由变量的定义位置决定，在不同位置定义的变量，它的作用域是不一样的。本节我们只讲解两种变量，一种是只能在函数内部使用的变量，另一种是可以在所有代码文件中使用的变量。

在函数内部定义的变量（局部变量）

在函数内部定义的变量，它的作用域也仅限于函数内部，出了函数就不能使用了，我们将这样的变量称为局部变量（Local Variable）。函数的形参也是局部变量，也只能在函数内部使用。请看下面的例子：

```
#include <stdio.h>

int sum(int m, int n){
    int i, sum=0;
    //m、n、i、sum 都是局部变量，只能在 sum() 内部使用
    for(i=m; i<=n; i++){
        sum+=i;
    }
    return sum;
}

int main(){
    int begin = 5, end = 86;
    int result = sum(begin, end);
    //begin、end、result 也都是局部变量，只能在 main() 内部使用
    printf("The sum from %d to %d is %d\n", begin, end, result);

    return 0;
}
```

m、n、i、sum 是局部变量，只能在 sum() 内部使用；begin、end、result 也是局部变量，只能在 main() 内部使用。

对局部变量的两点说明：

- main() 也是一个函数，在 main() 内部定义的变量也是局部变量，只能在 main() 函数内部使用。

- 形参也是局部变量，将实参传递给形参的过程，就是用实参给局部变量赋值的过程，它和 a=b;sum=m+n;这样的赋值没有什么区别。

在所有函数外部定义的变量（全局变量）

C 语言允许在所有函数的外部定义变量，这样的变量称为全局变量（Global Variable）。

全局变量的默认作用域是整个程序，也就是所有的代码文件，包括源文件（.c 文件）和头文件（.h 文件）。如果给全局变量加上 **static** 关键字，它的作用域就变成了当前文件，在其它文件中就无效了。我们目前编写的代码都是在一个源文件中，所以暂时不用考虑 static 关键字，后续我将会在《C 语言多文件编程》专题中详细讲解。

【实例】定义一个函数，根据长方体的长宽高求它的体积以及三个面的面积。

```
#include <stdio.h>

//定义三个全局变量，分别表示三个面的面积
int s1 = 0, s2 = 0, s3 = 0;

int vs(int length, int width, int height){
    int v;  //体积
    v = length * width * height;
    s1 = length * width;
    s2 = width * height;
    s3 = length * height;
    return v;
}

int main(){
    int v = 0;
    v = vs(15, 20, 30);
    printf("v=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);
    v = vs(5, 17, 8);
    printf("v=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);

    return 0;
}
```

运行结果：

```
v=9000, s1=300, s2=600, s3=450
v=680, s1=85, s2=136, s3=40
```

根据题意，我们希望借助一个函数得到四份数据：体积 v 以及三个面的面积 s1、s2、s3。遗憾的是，C 语言中的函数只能有一个返回值，我们只能将其中的一份数据（也就是体积 v）放到返回值中，其它三份数据（也就是面积 s1、s2、s3）只能保存到全局变量中。

C 语言代码从前往后依次执行，变量在使用之前必须定义或者声明，全局变量 s1、s2、s3 定义在程序开头，所以在 vs() 和 main() 中都有效。

在 vs() 中将求得的面积放到 s1、s2、s3 中，在 main() 中能够顺利取得它们的值，这说明：在一个函数内部修改全局变量的值会影响其它函数，全局变量的值在函数内部被修改后并不会自动恢复，它会一直保留该值，直到下次被修改。

全局变量也是变量，变量只能保存一份数据，一旦数据被修改了，原来的数据就被冲刷掉了，再也无法恢复了，所以不管是全局变量还是局部变量，一旦它的值被修改，这种影响都会一直持续下去，直到再次被修改。

关于变量的命名

每一段可运行的 C 语言代码都包含了多个作用域，即使最简单的 C 语言代码也是如此。

```
1.  int main(){
2.      return 0;
3.  }
```

这就是最简单的、可运行的 C 语言代码，它包含了两个作用域，一个是 main() 函数内部的局部作用域，一个是 main() 函数外部的全局作用域。

C 语言规定，在同一个作用域中不能出现两个名字相同的变量，否则会产生命名冲突；但是在不同的作用域中，允许出现名字相同的变量，它们的作用范围不同，彼此之间不会产生冲突。这句话有两层含义：

- 不同函数内部可以出现同名的变量，不同函数是不同的局部作用域；
- 函数内部和外部可以出现同名的变量，函数内部是局部作用域，函数外部是全局作用域。

1) 不同函数内部的同名变量是两个完全独立的变量，它们之间没有任何关联，也不会相互影响。请看下面的代码：

```
1.  #include <stdio.h>
2.
3.  void func_a(){
4.      int n = 100;
5.      printf("func_a: n = %d\n", n);
6.      n = 86;
7.      printf("func_a: n = %d\n", n);
8.  }
9.
10. void func_b(){
11.     int n = 29;
12.     printf("func_b: n = %d\n", n);
13.     func_a(); //调用func_a()
14.     printf("func_b: n = %d\n", n);
15. }
16.
17. int main(){
18.     func_b();
19.     return 0;
20. }
```

运行结果：

```
func_b: n = 29
func_a: n = 100
func_a: n = 86
func_b: n = 29
```

func_a() 和 func_b() 内部都定义了一个变量 n，在 func_b() 中，n 的初始值是 29，调用 func_a() 后，n 值还是 29，这说明 func_b() 内部的 n 并没有影响 func_a() 内部的 n。这两个 n 是完全不同的变量，彼此之间根本“不认识”，只是起了个相同的名字而已，这就好像明星撞衫，北京和云南都有叫李红的，赶巧了而已。

2) 函数内部的局部变量和函数外部的全局变量同名时，在当前函数这个局部作用域中，全局变量会被“屏蔽”，不再起作用。也就是说，在函数内部使用的是局部变量，而不是全局变量。

变量的使用遵循就近原则，如果在当前的局部作用域中找到了同名变量，就不会再去更大的全局作用域中查找。另外，只能从小的作用域向大的作用域中去寻找变量，而不能反过来，使用更小的作用域中的变量。

下面我们通过一个具体的例子来说明：

```
#include <stdio.h>

int n = 10;  //全局变量

void func1(){
    int n = 20;  //局部变量
    printf("func1 n: %d\n", n);
}

void func2(int n){
    printf("func2 n: %d\n", n);
}

void func3(){
    printf("func3 n: %d\n", n);
}

int main(){
    int n = 30;  //局部变量
    func1();
    func2(n);
    func3();
    printf("main n: %d\n", n);

    return 0;
}
```

运行结果：

```
func1 n: 20
func2 n: 30
```

```
func3 n: 10
main n: 30
```

代码中虽然定义了多个同名变量 n，但它们的作用域不同，所有不会产生命名冲突。

下面是对输出结果的分析：

- 对于 func1()，输出结果为 20，显然使用的是 func1() 内部的 n，而不是外部的 n。
- 调用 func2() 时，会把 main() 中的实参 n 传递给 func2() 中的形参 n，此时形参 n 的值变为 30。形参 n 也是局部变量，所以就使用它了。
- func3() 输出 10，使用的是全局变量，因为在 func3() 中不存在局部变量 n，所以编译器只能到函数外部，也就是全局作用域中去寻找变量 n。
- main() 中 printf() 语句输出 30，说明使用的是 main() 中的 n，而不是外部的 n。

7.9 C 语言块级变量（在代码块内部定义的变量）

所谓代码块，就是由{}包围起来的代码。代码块在 C 语言中随处可见，例如函数体、选择结构、循环结构等。不包含代码块的 C 语言程序根本不能运行，即使最简单的 C 语言程序（上节已经进行了展示）也要包含代码块。

C 语言允许在代码块内部定义变量，这样的变量具有块级作用域；换句话说，在代码块内部定义的变量只能在代码块内部使用，出了代码块就无效了。

上节我们已经讲解了函数，在函数内部定义的变量叫做局部变量，这节我们接着讲解选择结构和循环结构。

【实例 1】定义一个函数 gcd()，求两个整数的最大公约数。

```
#include <stdio.h>

//函数声明
int gcd(int a, int b);  //也可以写作 int gcd(int, int);

int main(){
    printf("The greatest common divisor is %d\n", gcd(100, 60));
    return 0;
}

//函数定义
int gcd(int a, int b){
    //若a<b，那么交换两变量的值
    if(a < b){
        int temp1 = a;  //块级变量
        a = b;
        b = temp1;
    }

    //求最大公约数
```

```
    while(b!=0){
        int temp2 = b;  //块级变量
        b = a % b;
        a = temp2;
    }

    return a;
}
```

运行结果：
The greatest common divisor is 20

读者暂时不用理解 gcd() 函数的思路，只需要关注 temp1 和 temp2 这两个变量，它们都是在代码块内部定义的块级变量，temp1 的作用域是 if 内部，temp2 的作用域是 while 内部。

在 for 循环条件里面定义变量

遵循 C99 标准的编译器允许在 for 循环条件里面定义新变量，这样的变量也是块级变量，它的作用域仅限于 for 循环内部。例如，计算从 m 累加到 n 的和：

```
#include <stdio.h>

int sum(int m, int n);

int main(){
    printf("The sum from 1 to 100 is %d\n", sum(1, 100));
    return 0;
}

int sum(int m, int n){
    int sum = 0;
    for(int i=m; i<=n; i++){  //i是块级变量
        sum += i;
    }
    return sum;
}
```

变量 i 定义在循环条件里面，所以是一个块级变量，它的作用域就是当前 for 循环，出了 for 循环就无效了。

如果一个变量只在 for 循环内部使用，就可以将它定义在循环条件里面，这样做可以避免在函数开头定义过多的变量，使得代码结构更加清晰，所以我鼓励大家这样做，当然，前提是你的编译器支持。

【实例 2】定义一个函数 strchar()，查看给定的字符是否位于某个字符串中。

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

int strchar(char *str, char c);
```

```
5.
6.  int main(){
7.      char url[] = "http://c.biancheng.net";
8.      char letter = 'c';
9.      if(strchar(url, letter) >= 0){
10.         printf("The letter is in the string.\n");
11.     }else{
12.         printf("The letter is not in the string.\n");
13.     }
14.     return 0;
15. }
16.
17. int strchar(char *str, char c){
18.     for(int i=0,len=strlen(str); i<len; i++){  //i和len都是块级变量
19.         if(str[i] == c){
20.             return i;
21.         }
22.     }
23.
24.     return -1;
25. }
```

循环条件里面可以定义一个或者多个变量，这段代码我们就定义了两个变量，分别是 i 和 len，它们都是块级变量，作用域都是当前 for 循环。

单独的代码块

C 语言还允许出现单独的代码块，它也是一个作用域。请看下面的代码：

```
1.  #include <stdio.h>
2.  int main(){
3.      int n = 22;  //编号①
4.      //由{ }包围的代码块
5.      {
6.          int n = 40;  //编号②
7.          printf("block n: %d\n", n);
8.      }
9.      printf("main n: %d\n", n);
10.
11.     return 0;
12. }
```

运行结果：

block n: 40

main n: 22

这里有两个 n，它们位于不同的作用域，不会产生命名冲突。{} 的作用域比 main() 更小，{} 内部的 printf() 使用

的是编号为②的 n，main() 内部的 printf() 使用的是编号为①的 n。

再谈作用域

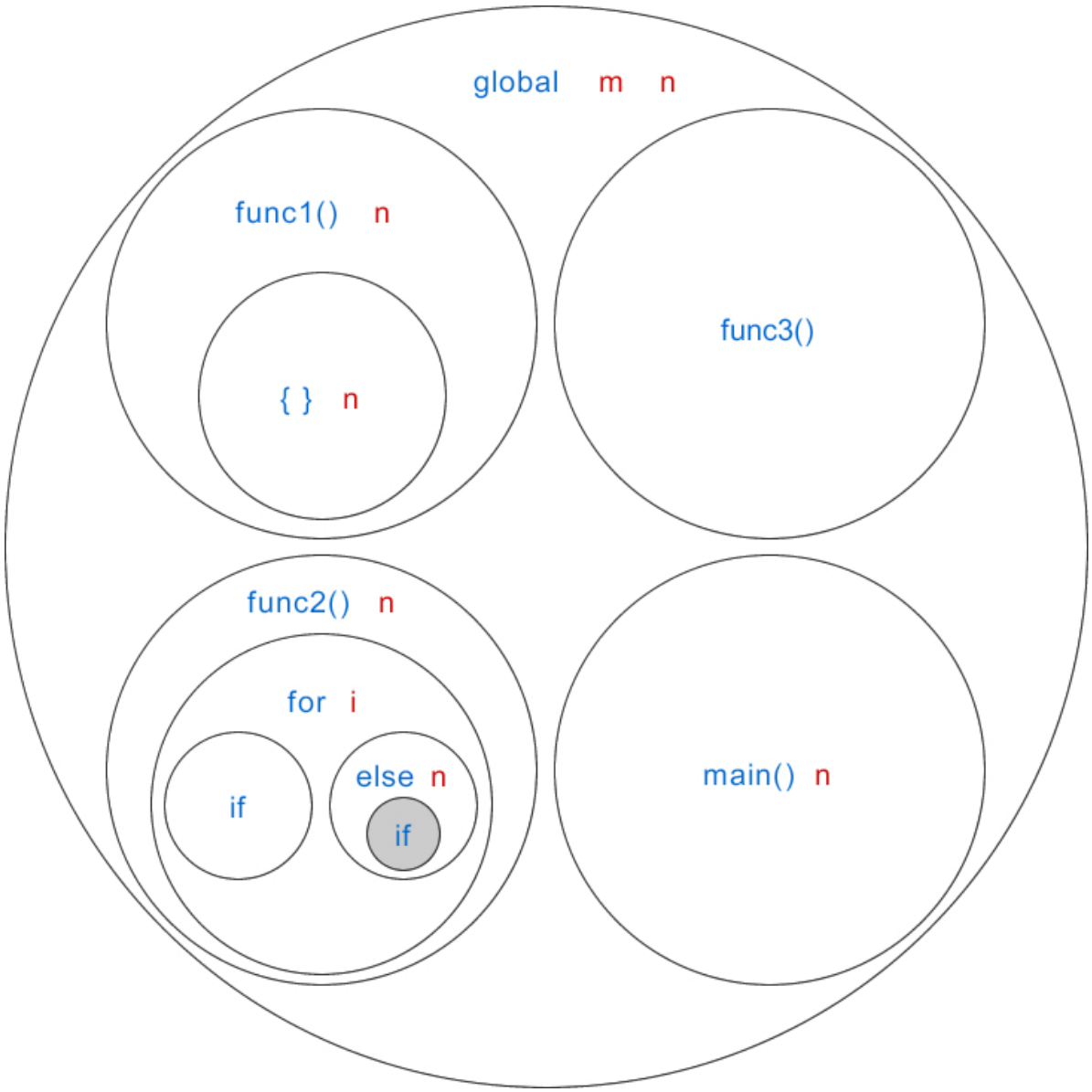
每个 C 语言程序都包含了多个作用域，不同的作用域中可以出现同名的变量，C 语言会按照从小到大的顺序、一层一层地去父级作用域中查找变量，如果在最顶层的全局作用域中还未找到这个变量，那么就会报错。

下面我们通过具体的代码来演示：

```
#include <stdio.h>

int m = 13;
int n = 10;

void func1(){
    int n = 20;
    {
        int n = 822;
        printf("block1 n: %d\n", n);
    }
    printf("func1 n: %d\n", n);
}

void func2(int n){
    for(int i=0; i<10; i++){
        if(i % 5 == 0){
            printf("if m: %d\n", m);
        }else{
            int n = i % 4;
            if(n<2 && n>0){
                printf("else m: %d\n", m);
            }
        }
    }
    printf("func2 n: %d\n", n);
}

void func3(){
    printf("func3 n: %d\n", n);
}

int main(){
    int n = 30;
    func1();
    func2(n);
    func3();
```

```
38.     printf("main n: %d\n", n);
39.
40.     return 0;
41. }
```

下图展示了这段代码的作用域：



蓝色表示作用域的名称，红色表示作用域中的变量，global 表示全局作用域。在灰色背景的作用域中，我们使用到了 m 变量，而该变量位于全局作用域中，所以得穿越好几层作用域才能找到 m。

7.10 C 语言递归函数（递归调用）详解[带实例演示]

一个函数在它的函数体内调用它自身称为**递归调用**，这种函数称为**递归函数**。执行递归函数将反复调用其自身，每调用一次就进入新的一层，当最内层的函数执行完毕后，再一层一层地由里到外退出。

> 递归函数不是 C 语言的专利，Java、C#、JavaScript、PHP 等其他编程语言也都支持递归函数。

下面我们通过一个求阶乘的例子，看看递归函数到底是如何运作的。阶乘 n! 的计算公式如下：

$$n! = \begin{cases} 1 & (n = 0, 1) \\ n * (n-1)! & (n > 1) \end{cases}$$

根据公式编写如下的代码：

```
#include <stdio.h>

//求n的阶乘
long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return factorial(n - 1) * n;  // 递归调用
    }
}

int main() {
    int a;
    printf("Input a number: ");
    scanf("%d", &a);
    printf("Factorial(%d) = %ld\n", a, factorial(a));

    return 0;
}
```

运行结果：

Input a number: 5↙

Factorial(5) = 120

factorial() 就是一个典型的递归函数。调用 factorial() 后即进入函数体，只有当 n==0 或 n==1 时函数才会执行结束，否则就一直调用它自身。

由于每次调用的实参为 n-1，即把 n-1 的值赋给形参 n，所以每次递归实参的值都减 1，直到最后 n-1 的值为 1 时再作递归调用，形参 n 的值也为 1，递归就终止了，会逐层退出。

要想理解递归函数，重点是理解它是如何逐层进入，又是如何逐层退出的，下面我们以 5! 为例进行讲解。

递归的进入

1) 求 5!，即调用 factorial(5)。当进入 factorial() 函数体后，由于形参 n 的值为 5，不等于 0 或 1，所以执行 factorial(n-1) * n，也即执行 factorial(4) * 5。为了求得这个表达式的结果，必须先调用 factorial(4)，并暂停其他操作。换句话说，在得到 factorial(4) 的结果之前，不能进行其他操作。这就是第一次递归。

2) 调用 factorial(4) 时，实参为 4，形参 n 也为 4，不等于 0 或 1，会继续执行 factorial(n-1) * n，也即执行 factorial(3) * 4。为了求得这个表达式的结果，又必须先调用 factorial(3)。这就是第二次递归。

3) 以此类推，进行四次递归调用后，实参的值为 1，会调用 factorial(1)。此时能够直接得到常量 1 的值，并把结果 return，就不需要再次调用 factorial() 函数了，递归就结束了。

下表列出了逐层进入的过程

层次/层数	实参/形参	调用形式	需要计算的表达式	需要等待的结果
1	n=5	factorial(5)	factorial(4) * 5	factorial(4) 的结果
2	n=4	factorial(4)	factorial(3) * 4	factorial(3) 的结果
3	n=3	factorial(3)	factorial(2) * 3	factorial(2) 的结果
4	n=2	factorial(2)	factorial(1) * 2	factorial(1) 的结果
5	n=1	factorial(1)	1	无

递归的退出

当递归进入到最内层的时候，递归就结束了，就开始逐层退出了，也就是逐层执行 return 语句。

1) n 的值为 1 时达到最内层，此时 return 出去的结果为 1，也即 factorial(1) 的调用结果为 1。

2) 有了 factorial(1) 的结果，就可以返回上一层计算 `factorial(1) * 2` 的值了。此时得到的值为 2，return 出去的结果也为 2，也即 factorial(2) 的调用结果为 2。

3) 以此类推，当得到 factorial(4) 的调用结果后，就可以返回最顶层。经计算，factorial(4) 的结果为 24，那么表达式 `factorial(4) * 5` 的结果为 120，此时 return 得到的结果也为 120，也即 factorial(5) 的调用结果为 120，这样就得到了 5! 的值。

下表列出了逐层退出的过程

层次/层数	调用形式	需要计算的表达式	从内层递归得到的结果 （内层函数的返回值）	表达式的值 （当次调用的结果）
5	factorial(1)	1	无	1
4	factorial(2)	factorial(1) * 2	factorial(1) 的返回值，也就是 1	2
3	factorial(3)	factorial(2) * 3	factorial(2) 的返回值，也就是 2	6
2	factorial(4)	factorial(3) * 4	factorial(3) 的返回值，也就是 6	24
1	factorial(5)	factorial(4) * 5	factorial(4) 的返回值，也就是 24	120

至此，我们已经对递归函数 factorial() 的进入和退出流程做了深入的讲解，把看似复杂的调用细节逐一呈献给大家，即使你是初学者，相信你也能解开谜团。

递归的条件

每一个递归函数都应该只进行有限次的递归调用，否则它就会进入死胡同，永远也不能退出了，这样的程序是没有意义的。

要想让递归函数逐层进入再逐层退出，需要解决两个方面的问题：

- 存在限制条件，当符合这个条件时递归便不再继续。对于 factorial()，当形参 n 等于 0 或 1 时，递归就结束了。
- 每次递归调用之后越来越接近这个限制条件。对于 factorial()，每次递归调用的实参为 n - 1，这会使得形参 n 的值逐渐减小，越来越趋近于 1 或 0。

更多关于递归函数的内容

factorial() 是最简单的一种递归形式——尾递归，也就是递归调用位于函数体的结尾处。除了尾递归，还有更加烧脑的两种递归形式，分别是中间递归和多层递归：

- 中间递归：发生递归调用的位置在函数体的中间；
- 多层递归：在一个函数里面多次调用自己。

递归函数也只是一种解决问题的技巧，它和其它技巧一样，也存在某些缺陷，具体来说就是：递归函数的时间开销和内存开销都非常大，极端情况下会导致程序崩溃。

我们将在接下来的三节课程里面讲解这些进阶内容：

- C 语言中间递归函数（比较复杂的一种递归）
- C 语言多层递归函数（最烧脑的一种递归）
- 递归函数的致命缺陷：巨大的时间开销和内存开销（附带优化方案）

7.11 C 语言中间递归函数（比较复杂的一种递归）

7.12 C 语言多层递归函数（最烧脑的一种递归）

7.13 递归函数的致命缺陷：巨大的时间开销和内存开销（附带优化方案）

7.14 忽略语法细节，从整体上理解函数

从整体上看，C 语言代码是由一个一个的函数构成的，除了定义和说明类的语句（例如变量定义、宏定义、类型定义等）可以放在函数外面，所有具有运算或逻辑处理能力的语句（例如加减乘除、if else、for、函数调用等）都要放在函数内部。

例如，下面的代码就是错误的：

```
#include <stdio.h>

int a = 10;
int b = a + 20;

int main(){
    return 0;
}
```

`int b = a + 20;`是具有运算功能的语句，要放在函数内部。

但是下面的代码就是正确的：

```
#include <stdio.h>

int a = 10;
int b = 10 + 20;

int main(){
    return 0;
}
```

`int b = 10 + 20;`在编译时会被优化成 `int b = 30;`，消除加法运算。

在所有的函数中，main() 是入口函数，有且只能有一个，C 语言程序就是从这里开始运行的。

C 语言不但提供了丰富的库函数，还允许用户定义自己的函数。每个函数都是一个可以重复使用的模块，通过模块间的相互调用，有条不紊地实现复杂的功能。可以说，C 程序的全部工作都是由各式各样的函数完成的，函数就好比一个一个的零件，组合在一起构成一台强大的机器。

标准 C 语言（ANSI C）共定义了 15 个头文件，称为“C 标准库”，所有的编译器都必须支持，如何正确并熟练的使用这些标准库，可以反映出一个程序员的水平。

- 合格程序员：<stdio.h>、<ctype.h>、<stdlib.h>、<string.h>
- 熟练程序员：<assert.h>、<limits.h>、<stddef.h>、<time.h>
- 优秀程序员：<float.h>、<math.h>、<error.h>、<locale.h>、<setjmp.h>、<signal.h>、<stdarg.h>

以上各类函数不仅数量众多，而且有的还需要硬件知识才能使用，初学者要想全部掌握得需要一个较长的学习过程。我的建议是先掌握一些最基本、最常用的函数，在实践过程中再逐步深入。由于课时关系，本教程只介绍了很少一部分库函数，其余部分读者可根据需要查阅 C 语言函数手册，网址是 http://www.cplusplus.com。

还应该指出的是，C 语言中所有的函数定义，包括主函数 main() 在内，都是平行的。也就是说，在一个函数的函数体内，不能再定义另一个函数，即不能嵌套定义。但是函数之间允许相互调用，也允许嵌套调用。习惯上把调用者称为主调函数，被调用者称为被调函数。函数还可以自己调用自己，称为递归调用。

main() 函数是主函数，它可以调用其它函数，而不允许被其它函数调用。因此，C 程序的执行总是从 main() 函数开始，完成对其它函数的调用后再返回到 main() 函数，最后由 main() 函数结束整个程序。

第 08 章 C 语言预处理命令

在编译和链接之前，还需要对源文件进行一些文本方面的操作，比如文本替换、文件包含、删除部分代码等，这个过程叫做预处理，由预处理程序完成。

较之其他编程语言，C/C++ 语言更依赖预处理器，所以在阅读或开发 C/C++ 程序过程中，可能会接触大量的预处理指令，比如 #include、#define 等。

本章目录：

1. C 语言预处理命令是什么？
2. C 语言#include 的用法（文件包含命令）
3. C 语言宏定义（#define 的用法）
4. C 语言带参数的宏定义
5. C 语言带参宏定义和函数的区别
6. C 语言宏参数的字符串化和宏参数的连接
7. C 语言中几个预定义宏
8. C 语言条件编译
9. C 语言#error 命令，阻止程序编译
10. C 语言预处理命令总结

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

8.1 C 语言预处理命令是什么？

前面各章中，已经多次使用过#include 命令。使用库函数之前，应该用#include 引入对应的头文件。这种以#号开头的命令称为预处理命令。

C 语言源文件要经过编译、链接才能生成可执行程序：
1) 编译（Compile）会将源文件（.c 文件）转换为目标文件。对于 VC/VS，目标文件后缀为.obj；对于 GCC，目标文件后缀为.o。

> 编译是针对单个源文件的，一次编译操作只能编译一个源文件，如果程序中有多个源文件，就需要多次编译操作。

2) 链接（Link）是针对多个文件的，它会将编译生成的多个目标文件以及系统中的库、组件等合并成一个可执行程序。

关于编译和链接的过程、目标文件和可执行文件的结构、.h 文件和 .c 文件的区别，我们将在《C 语言多文件编程》专题中讲解。

在实际开发中，有时候在编译之前还需要对源文件进行简单的处理。例如，我们希望自己的程序在 Windows 和 Linux 下都能够运行，那么就要在 Windows 下使用 VS 编译一遍，然后在 Linux 下使用 GCC 编译一遍。但是现在有个问题，程序中要实现的某个功能在 VS 和 GCC 下使用的函数不同（假设 VS 下使用 a()，GCC 下使用 b()），VS 下的函数在 GCC 下不能编译通过，GCC 下的函数在 VS 下也不能编译通过，怎么办呢？

这就需要在编译之前先对源文件进行处理：如果检测到是 VS，就保留 a() 删除 b()；如果检测到是 GCC，就保留 b() 删除 a()。

这些在编译之前对源文件进行简单加工的过程，就称为**预处理**（即预先处理、提前处理）。

预处理主要是处理以#开头的命令，例如#include <stdio.h>等。预处理命令要放在所有函数之外，而且一般都放在源文件的前面。

预处理是 C 语言的一个重要功能，由**预处理程序**完成。当对一个源文件进行编译时，系统将自动调用预处理程序对源程序中的预处理部分作处理，处理完毕自动进入对源程序的编译。

编译器会将预处理的结果保存到和源文件同名的.i 文件中，例如 main.c 的预处理结果在 main.i 中。和.c 一样，.i 也是文本文件，可以用编辑器打开直接查看内容。

C 语言提供了多种预处理功能，如宏定义、文件包含、条件编译等，合理地使用它们会使编写的程序便于阅读、修改、移植和调试，也有利于模块化程序设计。

实例

下面我们举个例子来说明预处理命令的实际用途。假如现在要开发一个 C 语言程序，让它暂停 5 秒以后再输出内容，并且要求跨平台，在 Windows 和 Linux 下都能运行，怎么办呢？

这个程序的难点在于，不同平台下的暂停函数和头文件都不一样：

- Windows 平台下的暂停函数的原型是 void Sleep(DWORD dwMilliseconds)（注意 S 是大写的），参数的单位是“毫秒”，位于 <windows.h> 头文件。
- Linux 平台下暂停函数的原型是 unsigned int sleep (unsigned int seconds)，参数的单位是“秒”，位于 <unistd.h> 头文件。

不同的平台下必须调用不同的函数，并引入不同的头文件，否则就会导致编译错误，因为 Windows 平台下没有 sleep() 函数，也没有 <unistd.h> 头文件，反之亦然。这就要求我们在编译之前，也就是预处理阶段来解决这个问题。请看下面的代码：

```
#include <stdio.h>

//不同的平台下引入不同的头文件
#if _WIN32  //识别windows平台
#include <windows.h>
#elif __linux__  //识别linux平台
#include <unistd.h>
#endif

int main() {
    //不同的平台下调用不同的函数
    #if _WIN32  //识别windows平台
    Sleep(5000);
    #elif __linux__  //识别linux平台
    sleep(5);
    #endif

    puts("http://c.biancheng.net/");

    return 0;
}
```

#if、#elif、#endif 就是预处理命令，它们都是在编译之前由预处理程序来执行的。这里我们不讨论细节，只从整体上来理解。

对于 Windows 平台，预处理以后的代码变成：

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

int main() {
    Sleep(5000);
    puts("http://c.biancheng.net/");

    return 0;
}
```

对于 Linux 平台，预处理以后的代码变成：

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

int main() {
    sleep(5);
    puts("http://c.biancheng.net/");

    return 0;
}
```

你看，在不同的平台下，编译之前（预处理之后）的源代码都是不一样的。这就是预处理阶段的工作，它把代码当成普通文本，根据设定的条件进行一些简单的文本替换，将替换以后的结果再交给编译器处理。

8.2 C 语言#include 的用法（文件包含命令）

#include 叫做文件包含命令，用来引入对应的头文件（.h 文件）。#include 也是 C 语言预处理命令的一种。

#include 的处理过程很简单，就是将头文件的内容插入到该命令所在的位置，从而把头文件和当前源文件连接成一个源文件，这与复制粘贴的效果相同。

#include 的用法有两种，如下所示：

```
#include <stdHeader.h>
#include "myHeader.h"
```

使用尖括号< >和双引号" "的区别在于头文件的搜索路径不同：

- 使用尖括号< >，编译器会到系统路径下查找头文件；
- 而使用双引号" "，编译器首先在当前目录下查找头文件，如果没有找到，再到系统路径下查找。

也就是说，使用双引号比使用尖括号多了一个查找路径，它的功能更为强大。

前面我们一直使用尖括号来引入标准头文件，现在我们也可以使用双引号了，如下所示：

```
#include "stdio.h"
#include "stdlib.h"
```

stdio.h 和 stdlib.h 都是标准头文件，它们存放于系统路径下，所以使用尖括号和双引号都能够成功引入；而我们自己编写的头文件，一般存放于当前项目的路径下，所以不能使用尖括号，只能使用双引号。

当然，你也可以把当前项目所在的目录添加到系统路径，这样就可以使用尖括号了，但是一般没人这么做，纯粹多此一举，费力不讨好。

> 关于系统路径和当前路径，还有更多的细节需要读者了解，我们将在《细说 C 语言头文件的路径》一文中深入探讨。

在以后的编程中，大家既可以使用尖括号来引入标准头文件，也可以使用双引号来引入标准头文件；不过，我个人

的习惯是使用尖括号来引入标准头文件，使用双引号来引入自定义头文件（自己编写的头文件），这样一眼就能看出头文件的区别。

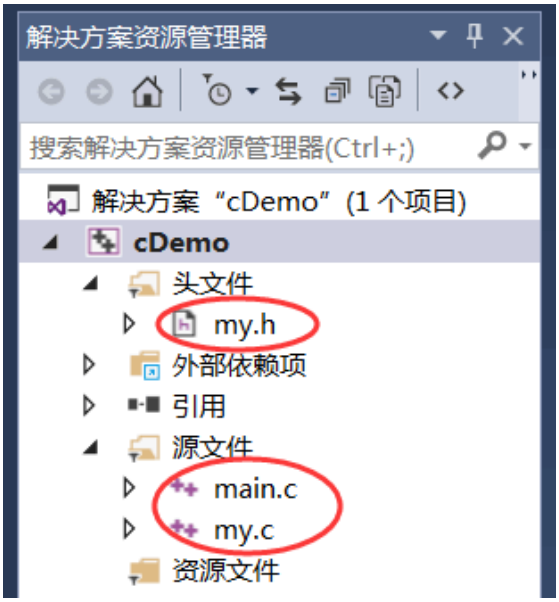
关于 #include 用法的注意事项：

- 一个 #include 命令只能包含一个头文件，多个头文件需要多个 #include 命令。
- 同一个头文件可以被多次引入，多次引入的效果和一次引入的效果相同，因为头文件在代码层面有防止重复引入的机制，具体细节我们将在《防止 C 语言头文件被重复包含》一文中深入探讨。
- 文件包含允许嵌套，也就是说在一个被包含的文件中又可以包含另一个文件。

#include 用法举例

我们早就学会使用 #include 引入标准头文件了，但是如何使用 #include 引入自定义的头文件呢？下面我们就通过一个例子来简单地演示一下。

本例中需要创建三个文件，分别是 main.c、my.c 和 my.h，如下图所示：



my.c 所包含的代码：

```
1.  //计算从m加到n的和
2.  int sum(int m, int n) {
3.      int i, sum = 0;
4.      for (i = m; i <= n; i++) {
5.          sum += i;
6.      }
7.      return sum;
8.  }
```

my.h 所包含的代码：

```
1.  //声明函数
2.  int sum(int m, int n);
```

main.c 所包含的代码：

```
1.  #include <stdio.h>
2.  #include "my.h"
3.
4.  int main() {
5.      printf("%d\n", sum(1, 100));
6.      return 0;
7.  }
```

我们在 my.c 中定义了 sum() 函数，在 my.h 中声明了 sum() 函数，这可能与很多初学者的认知发生了冲突：函数不是在头文件中定义的吗？为什么头文件中只有声明？

「在头文件中定义定义函数和全局变量」这种认知是原则性的错误！不管是标准头文件，还是自定义头文件，都只能包含变量和函数的声明，不能包含定义，否则在多次引入时会引起重复定义错误。

此外，可能还有初学者会问，main.c 只是引入了 my.h，没有引入 my.c，程序在编译时应该找不到函数定义呀，然而当我们亲自去运行程序的时候，却发现运行结果是正确的，这是怎么回事呢？

C 语言多文件编程涉及到很多细节，需要深入理解编译和链接的原理，本节我们仅做演示，不做更多讲解，有兴趣的读者请阅读《C 语言多文件编程》，届时你将解开以上各种谜团。

8.3 C 语言宏定义（#define 的用法）

#define 叫做宏定义命令，它也是 C 语言预处理命令的一种。所谓宏定义，就是用一个标识符来表示一个字符串，如果在后面的代码中出现了该标识符，那么就全部替换成指定的字符串。

我们先通过一个例子来看一下 #define 的用法：

```
1.  #include <stdio.h>
2.
3.  #define N 100
4.
5.  int main(){
6.      int sum = 20 + N;
7.      printf("%d\n", sum);
8.      return 0;
9.  }
```

运行结果：
120

注意第 6 行代码 `int sum = 20 + N`，`N` 被 `100` 代替了。

`#define N 100` 就是宏定义，`N` 为宏名，`100` 是宏的内容（宏所表示的字符串）。在预处理阶段，对程序中所有出现

的“宏名”，预处理器都会用宏定义中的字符串去代换，这称为“宏替换”或“宏展开”。

宏定义是由源程序中的宏定义命令#define完成的，宏替换是由预处理程序完成的。

宏定义的一般形式为：

```
#define  宏名  字符串
```

#表示这是一条预处理命令，所有的预处理命令都以 # 开头。宏名是标识符的一种，命名规则和变量相同。字符串可以是数字、表达式、if 语句、函数等。

> 这里所说的字符串是一般意义上的字符序列，不要和 C 语言中的字符串等同，它不需要双引号。

程序中反复使用的表达式就可以使用宏定义，例如：

```
#define M (n*n+3*n)
```

它的作用是指定标识符 M 来表示(y*y+3*y)这个表达式。在编写代码时，所有出现 (y*y+3*y) 的地方都可以用 M 来表示，而对源程序编译时，将先由预处理程序进行宏代替，即用 (y*y+3*y) 去替换所有的宏名 M，然后再进行编译。

将上面的例子补充完整：

```
1.  #include <stdio.h>
2.
3.  #define M (n*n+3*n)
4.
5.  int main(){
6.      int sum, n;
7.      printf("Input a number: ");
8.      scanf("%d", &n);
9.      sum = 3*M+4*M+5*M;
10.     printf("sum=%d\n", sum);
11.     return 0;
12. }
```

运行结果：

Input a number: 10↙

sum=1560

程序的开头首先定义了一个宏 M，它表示 (n*n+3*n) 这个表达式。在 9 行代码中使用了宏 M，预处理程序将它展开为下面的语句：

```
sum=3*(n*n+3*n)+4*(n*n+3*n)+5*(n*n+3*n);
```

需要注意的是，在宏定义中表达式(n*n+3*n)两边的括号不能少，否则在宏展开以后可能会产生歧义。下面是一个反面的例子：

```
#difine M n*n+3*n
```

在宏展开后将得到下述语句：

```
s=3*n*n+3*n+4*n*n+3*n+5*n*n+3*n;
```

这相当于：

```
3n2+3n+4n2+3n+5n2+3n
```

这显然是不正确的。所以进行宏定义时要注意，应该保证在宏替换之后不发生歧义。

对 #define 用法的几点说明

1) 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单粗暴的替换。字符串中可以含任何字符，它可以是常数、表达式、if 语句、函数等，预处理程序对它不作任何检查，如有错误，只能在编译已被宏展开后的源程序时发现。

2) 宏定义不是说明或语句，在行末不必加分号，如加上分号则连分号也一起替换。

3) 宏定义必须写在函数之外，其作用域为宏定义命令起到源程序结束。如要终止其作用域可使用#undef 命令。例如：

```
#define PI 3.14159

int main(){
    // Code
    return 0;
}

#undef PI

void func(){
    // Code
}
```

表示 PI 只在 main() 函数中有效，在 func() 中无效。

4) 代码中的宏名如果被引号包围，那么预处理程序不对其作宏代替，例如：

```
#include <stdio.h>
#define OK 100
int main(){
    printf("OK\n");
    return 0;
}
```

运行结果：

```
OK
```

该例中定义宏名 OK 表示 100，但在 printf 语句中 OK 被引号括起来，因此不作宏替换，而作为字符串处理。

5) 宏定义允许嵌套，在宏定义的字符串中可以使用已经定义的宏名，在宏展开时由预处理程序层层代换。例如：

```
#define PI 3.1415926
#define S PI*y*y      /* PI 是已定义的宏名*/
```

对语句：

```
printf("%f", S);
```

在宏代换后变为：

```
printf("%f", 3.1415926*y*y);
```

6) 习惯上宏名用大写字母表示，以便于与变量区别。但也允许用小写字母。

7) 可用宏定义表示数据类型，使书写方便。例如：

```
#define UINT unsigned int
```

在程序中可用 UINT 作变量说明：

```
UINT a, b;
```

应注意用宏定义表示数据类型和用 typedef 定义数据说明符的区别。宏定义只是简单的字符串替换，由预处理器来处理；而 typedef 是在编译阶段由编译器处理的，它并不是简单的字符串替换，而给原有的数据类型起一个新的名字，将它作为一种新的数据类型。

请看下面的例子：

```
#define PIN1 int *
typedef int *PIN2;  //也可以写作 typedef int (*PIN2);
```

从形式上看这两者相似， 但在实际使用中却不相同。

下面用 PIN1，PIN2 说明变量时就可以看出它们的区别：

```
PIN1 a, b;
```

在宏代换后变成：

```
int * a, b;
```

表示 a 是指向整型的指针变量，而 b 是整型变量。然而：

```
PIN2 a,b;
```

表示 a、b 都是指向整型的指针变量。因为 PIN2 是一个新的、完整的数据类型。由这个例子可见，宏定义虽然也可表示数据类型， 但毕竟只是简单的字符串替换。在使用时要格外小心，以避出错。

8.4 C 语言带参数的宏定义

C 语言允许宏带有参数。在宏定义中的参数称为“形式参数”，在宏调用中的参数称为“实际参数”，这点和函数有些类似。

对带参数的宏，在展开过程中不仅要进行字符串替换，还要用实参去替换形参。

带参宏定义的一般形式为：

```
#define 宏名(形参列表) 字符串
```

在字符串中可以含有各个形参。

带参宏调用的一般形式为：

```
宏名(实参列表);
```

例如：

```
#define M(y) y*y+3*y   //宏定义
// TODO:
k=M(5);   //宏调用
```

在宏展开时，用实参 5 去代替形参 y，经预处理程序展开后的语句为 `k=5*5+3*5`。

【示例】输出两个数中较大的数。

```
1.  #include <stdio.h>
2.  #define MAX(a,b) (a>b) ? a : b
3.  int main(){
4.      int x , y, max;
5.      printf("input two numbers: ");
6.      scanf("%d %d", &x, &y);
7.      max = MAX(x, y);
8.      printf("max=%d\n", max);
9.      return 0;
10. }
```

运行结果：
input two numbers: 10 20
max=20

程序第 2 行定义了一个带参数的宏，用宏名 `MAX` 表示条件表达式`(a>b) ? a : b`，形参 a、b 均出现在条件表达式中。程序第 7 行 `max = MAX(x, y)`为宏调用，实参 x、y 将用来代替形参 a、b。宏展开后该语句为：

```
max=(x>y) ? x : y;
```

对带参宏定义的说明

1) 带参宏定义中，形参之间可以出现空格，但是宏名和形参列表之间不能有空格出现。例如把：

```
#define MAX(a,b) (a>b)?a:b
```

写为：

```
#define MAX   (a,b)   (a>b)?a:b
```

将被认为是无参宏定义，宏名 MAX 代表字符串`(a,b) (a>b)?a:b`。宏展开时，宏调用语句：

```
max = MAX(x,y);
```

将变为：

```
max = (a,b)(a>b)?a:b(x,y);
```

这显然是错误的。

2) 在带参宏定义中，不会为形式参数分配内存，因此不必指明数据类型。而在宏调用中，实参包含了具体的数据，要用它们去替换形参，因此实参必须要指明数据类型。

这一点和函数是不同的：在函数中，形参和实参是两个不同的变量，都有自己的作用域，调用时要把实参的值传递给形参；而在带参数的宏中，只是符号的替换，不存在值传递的问题。

【示例】输入 n，输出 (n+1)^2 的值。

```
1.  #include <stdio.h>
2.  #define SQ(y) (y)*(y)
3.  int main(){
4.      int a, sq;
5.      printf("input a number: ");
6.      scanf("%d", &a);
7.      sq = SQ(a+1);
8.      printf("sq=%d\n", sq);
9.      return 0;
10. }
```

运行结果：

```
input a number: 9
sq=100
```

第 2 行为宏定义，形参为 y。第 7 行宏调用中实参为 a+1，是一个表达式，在宏展开时，用 a+1 代换 y，再用 (y)*(y) 代换 SQ，得到如下语句：

```
sq=(a+1)*(a+1);
```

这与函数的调用是不同的，函数调用时要把实参表达式的值求出来再传递给形参，而宏展开中对实参表达式不作计算，直接按照原样替换。

3) 在宏定义中，字符串内的形参通常要用括号括起来以避免出错。例如上面的宏定义中 (y)*(y) 表达式的 y 都用括号括起来，因此结果是正确的。如果去掉括号，把程序改为以下形式：

```
1.  #include <stdio.h>
2.  #define SQ(y) y*y
3.  int main(){
4.      int a, sq;
5.      printf("input a number: ");
6.      scanf("%d", &a);
7.      sq = SQ(a+1);
8.      printf("sq=%d\n", sq);
9.      return 0;
```

```
10.  }
```

运行结果为：

input a number: 9

sq=19

同样输入 9，但结果却是不一样的。问题在哪里呢？这是由于宏展开只是简单的符号替换的过程，没有任何其它的处理。宏替换后将得到以下语句：

```
sq=a+1*a+1;
```

由于 a 为 9，故 sq 的值为 19。这显然与题意相违，因此参数两边的括号是不能少的。即使在参数两边加括号还是不够的，请看下面程序：

```
1.  #include <stdio.h>
2.  #define SQ(y) (y)*(y)
3.  int main(){
4.      int a,sq;
5.      printf("input a number: ");
6.      scanf("%d", &a);
7.      sq = 200 / SQ(a+1);
8.      printf("sq=%d\n", sq);
9.      return 0;
10. }
```

与前面的代码相比，只是把宏调用语句改为：

```
sq = 200/SQ(a+1);
```

运行程序后，如果仍然输入 9，那么我们希望的结果为 2。但实际情况并非如此：

input a number: 9

sq=200

为什么会得这样的结果呢？分析宏调用语句，在宏展开之后变为：

```
sq=200/(a+1)*(a+1);
```

a 为 9 时，由于 “/” 和 “*” 运算符优先级和结合性相同，所以先计算 200/(9+1)，结果为 20，再计算 20*(9+1)，最后得到 200。

为了得到正确答案，应该在宏定义中的整个字符串外加括号：

```
1.  #include <stdio.h>
2.  #define SQ(y) ((y)*(y))
3.  int main(){
4.      int a,sq;
5.      printf("input a number: ");
6.      scanf("%d", &a);
7.      sq = 200 / SQ(a+1);
8.      printf("sq=%d\n", sq);
9.      return 0;
```

```
10.  }
```

由此可见，对于带参宏定义不仅要在参数两侧加括号，还应该在整个字符串外加括号。

8.5 C 语言带参宏定义和函数的区别

带参数的宏和函数很相似，但有本质上的区别：宏展开仅仅是字符串的替换，不会对表达式进行计算；宏在编译之前就被处理掉了，它没有机会参与编译，也不会占用内存。而函数是一段可以重复使用的代码，会被编译，会给它分配内存，每次调用函数，就是执行这块内存中的代码。

【示例①】用函数计算平方值。

```
1.  #include <stdio.h>
2.
3.  int SQ(int y){
4.    return ((y)*(y));
5.  }
6.
7.  int main(){
8.      int i=1;
9.      while(i<=5){
10.         printf("%d^2 = %d\n", (i-1), SQ(i++));
11.     }
12.     return 0;
13. }
```

运行结果：

```
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

【示例②】用宏计算平方值。

```
1.  #include <stdio.h>
2.
3.  #define SQ(y) ((y)*(y))
4.
5.  int main(){
6.      int i=1;
7.      while(i<=5){
8.          printf("%d^2 = %d\n", i, SQ(i++));
9.      }
10.     return 0;
11. }
```

在 Visual Studio 和 C-Free 下的运行结果（其它编译器的运行结果可能不同，这个++运算的顺序有关）：

3^2 = 1
5^2 = 9
7^2 = 25

在示例①中，先把实参 i 传递给形参 y，然后再自增 1，这样每循环一次 i 的值增加 1，所以最终要循环 5 次。

在示例②中，宏调用只是简单的字符串替换，SQ(i++) 会被替换为 ((i++)*(i++))，这样每循环一次 i 的值增加 2，所以最终只循环 3 次。

由此可见，宏和函数只是在形式上相似，本质上是完全不同的。

带参数的宏也可以用来定义多个语句，在宏调用时，把这些语句又替换到源程序中，请看下面的例子：

```
1.  #include <stdio.h>
2.  #define SSSV(s1, s2, s3, v) s1 = length * width; s2 = length * height; s3 = width * height; v = width * length * height;
3.  int main(){
4.      int length = 3, width = 4, height = 5, sa, sb, sc, vv;
5.      SSSV(sa, sb, sc, vv);
6.      printf("sa=%d, sb=%d, sc=%d, vv=%d\n", sa, sb, sc, vv);
7.      return 0;
8.  }
```

运行结果：
sa=12, sb=15, sc=20, vv=60

8.6 宏参数的字符串化和宏参数的连接

8.7 C 语言中几个预定义宏

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

8.8 C 语言条件编译

假如现在要开发一个 C 语言程序，让它输出红色的文字，并且要求跨平台，在 Windows 和 Linux 下都能运行，怎么办呢？

这个程序的难点在于，不同平台下控制文字颜色的代码不一样，我们必须要能够识别出不同的平台。

Windows 有专有的宏_WIN32，Linux 有专有的宏__linux__，以现有的知识，我们很容易就想到了 if else，请看下面的代码：

```
#include <stdio.h>
int main(){
    if(_WIN32){
        system("color 0c");
        printf("http://c.biancheng.net\n");
    }else if(__linux__){
        printf("\033[22;31mhttp://c.biancheng.net\n\033[22;30m");
    }else{
        printf("http://c.biancheng.net\n");
    }

    return 0;
}
```

但这段代码是错误的，在 Windows 下提示 __linux__ 是未定义的标识符，在 Linux 下提示 _Win32 是未定义的标识符。对上面的代码进行改进：

```
#include <stdio.h>
int main(){
    #if _WIN32
        system("color 0c");
        printf("http://c.biancheng.net\n");
    #elif __linux__
        printf("\033[22;31mhttp://c.biancheng.net\n\033[22;30m");
    #else
        printf("http://c.biancheng.net\n");
    #endif

    return 0;
}
```

#if、#elif、#else 和 #endif 都是预处理命令，整段代码的意思是：如果宏 _WIN32 的值为真，就保留第 4、5 行代码，删除第 7、9 行代码；如果宏 __linux__ 的值为真，就保留第 7 行代码；如果所有的宏都为假，就保留第 9 行代码。

这些操作都是在预处理阶段完成的，多余的代码以及所有的宏都不会参与编译，不仅保证了代码的正确性，还减小了编译后文件的体积。

这种能够根据不同情况编译不同代码、产生不同目标文件的机制，称为条件编译。条件编译是预处理程序的功能，不是编译器的功能。

条件编译需要多个预处理命令的支持，下面一一讲解。

#if 的用法

#if 用法的一般格式为：

```
#if 整型常量表达式 1
    程序段 1
#elif 整型常量表达式 2
    程序段 2
#elif 整型常量表达式 3
    程序段 3
#else
    程序段 4
#endif
```

它的意思是：如常“表达式 1”的值为真（非 0），就对“程序段 1”进行编译，否则就计算“表达式 2”，结果为真的话就对“程序段 2”进行编译，为假的话就继续往下匹配，直到遇到值为真的表达式，或者遇到 #else。这一点和 if else 非常类似。

需要注意的是，#if 命令要求判断条件为“整型常量表达式”，也就是说，表达式中不能包含变量，而且结果必须是整数；而 if 后面的表达式没有限制，只要符合语法就行。这是 #if 和 if 的一个重要区别。

#elif 和 #else 也可以省略，如下所示：

```
1.  #include <stdio.h>
2.  int main(){
3.      #if _WIN32
4.          printf("This is Windows!\n");
5.      #else
6.          printf("Unknown platform!\n");
7.      #endif
8.
9.      #if __linux__
10.         printf("This is Linux!\n");
11.     #endif
12.
13.     return 0;
14. }
```

#ifdef 的用法

#ifdef 用法的一般格式为：

```
#ifdef  宏名
    程序段 1
```

```
#else
    程序段 2
#endif
```

它的意思是，如果当前的宏已被定义过，则对“程序段 1”进行编译，否则对“程序段 2”进行编译。

也可以省略 #else：

```
#ifdef  宏名
    程序段
#endif
```

VS/VC 有两种编译模式，Debug 和 Release。在学习过程中，我们通常使用 Debug 模式，这样便于程序的调试；而最终发布的程序，要使用 Release 模式，这样编译器会进行很多优化，提高程序运行效率，删除冗余信息。

为了能够清楚地看到当前程序的编译模式，我们不妨在程序中增加提示，请看下面的代码：

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main(){
4.      #ifdef _DEBUG
5.          printf("正在使用 Debug 模式编译程序...\n");
6.      #else
7.          printf("正在使用 Release 模式编译程序...\n");
8.      #endif
9.
10.     system("pause");
11.     return 0;
12. }
```

当以 Debug 模式编译程序时，宏 _DEBUG 会被定义，预处器会保留第 5 行代码，删除第 7 行代码。反之会删除第 5 行，保留第 7 行。

#ifndef 的用法

#ifndef 用法的一般格式为：

```
#ifndef 宏名
    程序段 1
#else
    程序段 2
#endif
```

与 #ifdef 相比，仅仅是将 #ifdef 改为了 #ifndef。它的意思是，如果当前的宏未被定义，则对“程序段 1”进行编译，否则对“程序段 2”进行编译，这与 #ifdef 的功能正好相反。

三者之间的区别

最后需要注意的是，#if 后面跟的是“整型常量表达式”，而 #ifdef 和 #ifndef 后面跟的只能是一个宏名，不能是其他的。

例如，下面的形式只能用于 #if：

```
#include <stdio.h>
#define NUM 10
int main(){
    #if NUM == 10 || NUM == 20
        printf("NUM: %d\n", NUM);
    #else
        printf("NUM Error\n");
    #endif
    return 0;
}
```

运行结果：
NUM: 10

再如，两个宏都存在时编译代码 A，否则编译代码 B：

```
#include <stdio.h>
#define NUM1 10
#define NUM2 20
int main(){
    #if (defined NUM1 && defined NUM2)
        //代码A
        printf("NUM1: %d, NUM2: %d\n", NUM1, NUM2);
    #else
        //代码B
        printf("Error\n");
    #endif
    return 0;
}
```

运行结果：
NUM1: 10, NUM2: 20

#ifdef 可以认为是 #if defined 的缩写。

8.9 #error 命令，阻止程序编译

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能

8.10 C 语言预处理命令总结

预处理指令是以#号开头的代码行，# 号必须是该行除了任何空白字符外的第一个字符。# 后是指令关键字，在关键字和 # 号之间允许存在任意个数的空白字符，整行语句构成了一条预处理指令，该指令将在编译器进行编译之前对源代码做某些转换。

下面是本章涉及到的部分预处理指令：

指令	说明
#	空指令，无任何效果
#include	包含一个源代码文件
#define	定义宏
#undef	取消已定义的宏
#if	如果给定条件为真，则编译下面代码
#ifdef	如果宏已经定义，则编译下面代码
#ifndef	如果宏没有定义，则编译下面代码
#elif	如果前面的#if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个#if……#else 条件编译块

预处理功能是 C 语言特有的功能，它是在对源程序正式编译前由预处理程序完成的，程序员在程序中用预处理命令来调用这些功能。

宏定义可以带有参数，宏调用时是以实参代换形参，而不是“值传送”。

为了避免宏代换时发生错误，宏定义中的字符串应加括号，字符串中出现的形式参数两边也应加括号。

文件包含是预处理的一个重要功能，它可用来把多个源文件连接成一个源文件进行编译，结果将生成一个目标文件。

条件编译允许只编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销并提高了程序的效率。

使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。

第 09 章 C 语言指针（精讲版）

没学指针就是没学 C 语言！指针是 C 语言的精华，也是 C 语言的难点，破解 C 语言指针，会让你的 C 语言水平突飞猛进。

所谓指针，也就是内存的地址；所谓指针变量，也就是保存了内存地址的变量。不过，人们往往不会区分两者的概念，而是混淆在一起使用，在必要的情况下，大家也要注意区分。

「C 语言指针专题」是本套教程的精华所在，文中演示了指针的各种玩法，阅读完本专题你将不再惧怕任何指针，再复杂的指针在你面前都是小菜一碟。

本章目录：

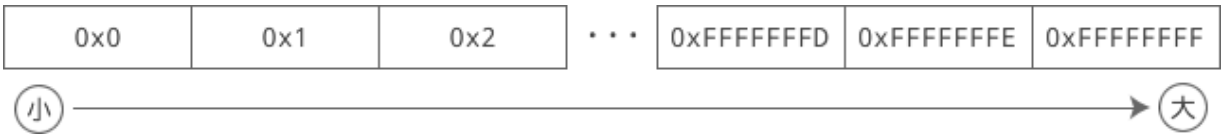
1. 1 分钟彻底理解 C 语言指针的概念
2. C 语言指针变量的定义和使用（精华）
3. C 语言指针变量的运算（加法、减法和比较运算）
4. C 语言数组指针（指向数组的指针）
5. C 语言字符串指针（指向字符串的指针）
6. C 语言数组灵活多变的访问形式
7. C 语言指针变量作为函数参数
8. C 语言指针作为函数返回值
9. C 语言二级指针（指向指针的指针）
10. C 语言空指针 NULL 以及 void 指针
11. 数组和指针绝不等价，数组是另外一种类型
12. 数组到底在什么时候会转换为指针
13. C 语言指针数组（数组每个元素都是指针）
14. 一道题目玩转指针数组和二级指针
15. C 语言二维数组指针（指向二维数组的指针）
16. C 语言函数指针（指向函数的指针）
17. 只需一招，彻底攻克 C 语言指针，再复杂的指针都不怕
18. main()函数的高级用法：接收用户输入的数据
19. 对 C 语言指针的总结

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

9.1 1 分钟彻底理解指针的概念

计算机中所有的数据都必须放在内存中，不同类型的数据占用的字节数不一样，例如 int 占用 4 个字节，char 占用 1 个字节。为了正确地访问这些数据，必须为每个字节都编上号码，就像门牌号、身份证号一样，每个字节的编号是唯一的，根据编号可以准确地找到某个字节。

下图是 4G 内存中每个字节的编号（以十六进制表示）：



我们将内存中字节的编号称为地址（Address）或指针（Pointer）。地址从 0 开始依次增加，对于 32 位环境，程序能够使用的内存为 4GB，最小的地址为 0，最大的地址为 0XFFFFFFFF。

下面的代码演示了如何输出一个地址：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      int a = 100;
5.      char str[20] = "c.biancheng.net";
6.      printf("%#X, %#X\n", &a, str);
7.      return 0;
8.  }
```

运行结果：
0X28FF3C, 0X28FF10

%#X 表示以十六进制形式输出，并附带前缀 0X。a 是一个变量，用来存放整数，需要在前面加&来获得它的地址；str 本身就表示字符串的首地址，不需要加&。

C 语言中有一个控制符%p，专门用来以十六进制形式输出地址，不过 %p 的输出格式并不统一，有的编译器带 0x 前缀，有的不带，所以此处我们并没有采用。

一切都是地址

C 语言用变量来存储数据，用函数来定义一段可以重复使用的代码，它们最终都要放到内存中才能供 CPU 使用。

数据和代码都以二进制的形式存储在内存中，计算机无法从格式上区分某块内存到底存储的是数据还是代码。当程序被加载到内存后，操作系统会给不同的内存块指定不同的权限，拥有读取和执行权限的内存块就是代码，而拥有读取和写入权限（也可能只有读取权限）的内存块就是数据。

CPU 只能通过地址来取得内存中的代码和数据，程序在执行过程中会告知 CPU 要执行的代码以及要读写的数据的地址。如果程序不小心出错，或者开发者有意为之，在 CPU 要写入数据时给它一个代码区域的地址，就会发生内存访问错误。这种内存访问错误会被硬件和操作系统拦截，强制程序崩溃，程序员没有挽救的机会。

CPU 访问内存时需要的是地址，而不是变量名和函数名！变量名和函数名只是地址的一种助记符，当源文件被编译和链接成可执行程序后，它们都会被替换成地址。编译和链接过程的一项重要任务就是找到这些名称所对应的地址。

假设变量 a、b、c 在内存中的地址分别是 0X1000、0X2000、0X3000，那么加法运算 c = a + b;将会被转换成类似下面的形式：

```
0X3000 = (0X1000) + (0X2000);
```

()表示取值操作，整个表达式的意思是，取出地址 0X1000 和 0X2000 上的值，将它们相加，把相加的结果赋值给地址为 0X3000 的内存

变量名和函数名为我们提供了方便，让我们在编写代码的过程中可以使用易于阅读和理解的英文字符串，不用直接面对二进制地址，那场景简直让人崩溃。

需要注意的是，虽然变量名、函数名、字符串名和数组名在本质上是一样的，它们都是地址的助记符，但在编写代码的过程中，我们认为变量名表示的是数据本身，而函数名、字符串名和数组名表示的是代码块或数据块的首地址。

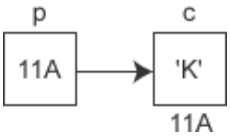
关于程序内存、编译链接、可执行文件的结构以及如何找到名称对应的地址，我们将在《C 语言内存精讲》和《C 语言多文件编程》专题中深入探讨。

9.2 C 指针变量的定义和使用（精华）

数据在内存中的地址也称为指针，如果一个变量存储了一份数据的指针，我们就称它为指针变量。

在 C 语言中，允许用一个变量来存放指针，这种变量称为指针变量。指针变量的值就是某份数据的地址，这样的一份数据可以是数组、字符串、函数，也可以是另外的一个普通变量或指针变量。

现在假设有一个 char 类型的变量 c，它存储了字符 'K'（ASCII 码为十进制数 75），并占用了地址为 0X11A 的内存（地址通常用十六进制表示）。另外有一个指针变量 p，它的值为 0X11A，正好等于变量 c 的地址，这种情况我们就称 p 指向了 c，或者说 p 是指向变量 c 的指针。



定义指针变量

定义指针变量与定义普通变量非常类似，不过要在变量名前面加星号*，格式为：

```
datatype *name;
```

或者

```
datatype *name = value;
```

*表示这是一个指针变量，datatype 表示该指针变量所指向的数据的类型 。例如：

```
int *p1;
```

p1 是一个指向 int 类型数据的指针变量，至于 p1 究竟指向哪一份数据，应该由赋予它的值决定。再如：

```
int a = 100;
int *p_a = &a;
```

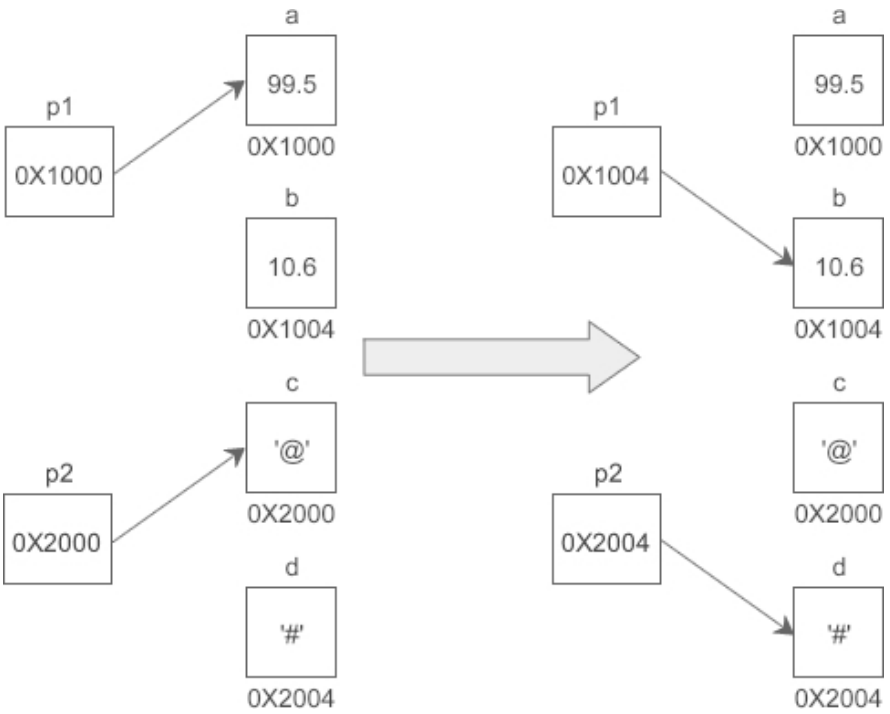
在定义指针变量 p_a 的同时对它进行初始化，并将变量 a 的地址赋予它，此时 p_a 就指向了 a。值得注意的是，p_a 需要的一个地址，a 前面必须要加取地址符&，否则是不对的。

和普通变量一样，指针变量也可以被多次写入，只要你想，随时都能够改变指针变量的值，请看下面的代码：

```
1.  //定义普通变量
2.  float a = 99.5, b = 10.6;
3.  char c = '@', d = '#';
4.  //定义指针变量
5.  float *p1 = &a;
6.  char *p2 = &c;
7.  //修改指针变量的值
8.  p1 = &b;
9.  p2 = &d;
```

是一个特殊符号，表明一个变量是指针变量，定义 p1、p2 时必须带。而给 p1、p2 赋值时，因为已经知道了它是一个指针变量，就没必要多此一举再带上*，后边可以像使用普通变量一样来使用指针变量。也就是说，定义指针变量时必须带*，给指针变量赋值时不能带*。

假设变量 a、b、c、d 的地址分别为 0X1000、0X1004、0X2000、0X2004，下面的示意图很好地反映了 p1、p2 指向的变化：



需要强调的是，p1、p2 的类型分别是 float*和 char*，而不是 float 和 char，它们是完全不同的数据类型，读者要引起注意。

指针变量也可以连续定义，例如：

```
1.  int *a, *b, *c;  //a、b、c 的类型都是 int*
```

注意每个变量前面都要带*。如果写成下面的形式，那么只有 a 是指针变量，b、c 都是类型为 int 的普通变量：

```
1.  int *a, b, c;
```

通过指针变量取得数据

指针变量存储了数据的地址，通过指针变量能够获得该地址上的数据，格式为：

```
*pointer;
```

这里的*称为指针运算符，用来取得某个地址上的数据，请看下面的例子：

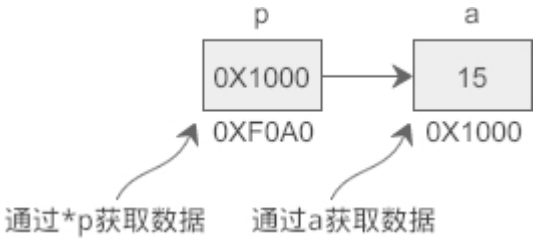
```
#include <stdio.h>

int main(){
    int a = 15;
    int *p = &a;
    printf("%d, %d\n", a, *p);  //两种方式都可以输出a的值
    return 0;
}
```

运行结果：
15, 15

假设 a 的地址是 0X1000，p 指向 a 后，p 本身的值也会变为 0X1000，*p 表示获取地址 0X1000 上的数据，也即变量 a 的值。从运行结果看，*p 和 a 是等价的。

上节我们说过，CPU 读写数据必须要知道数据在内存中的地址，普通变量和指针变量都是地址的助记符，虽然通过 *p 和 a 获取到的数据一样，但它们的运行过程稍有不同：a 只需要一次运算就能够取得数据，而 *p 要经过两次运算，多了一层“间接”。

假设变量 a、p 的地址分别为 0X1000、0XF0A0，它们的指向关系如下图所示：



程序被编译和链接后，a、p 被替换成相应的地址。使用 *p 的话，要先通过地址 0XF0A0 取得变量 p 本身的值，这个值是变量 a 的地址，然后再通过这个值取得变量 a 的数据，前后共有两次运算；而使用 a 的话，可以通过地址 0X1000 直接取得它的数据，只需要一步运算。

也就是说，使用指针是间接获取数据，使用变量名是直接获取数据，前者比后者的代价要高。

指针除了可以获取内存上的数据，也可以修改内存上的数据，例如：

```
#include <stdio.h>

int main(){
    int a = 15, b = 99, c = 222;
    int *p = &a;  //定义指针变量
```

```
6.      *p = b;  //通过指针变量修改内存上的数据
7.      c = *p;  //通过指针变量获取内存上的数据
8.      printf("%d, %d, %d, %d\n", a, b, c, *p);
9.      return 0;
10. }
```

运行结果：
99, 99, 99, 99

*p 代表的是 a 中的数据，它等价于 a，可以将另外的一份数据赋值给它，也可以将它赋值给另外的一个变量。

*在不同的场景下有不同的作用：*可以用在指针变量的定义中，表明这是一个指针变量，以和普通变量区分开；使用指针变量时在前面加*表示获取指针指向的数据，或者说表示的是指针指向的数据本身。

也就是说，定义指针变量时的*和使用指针变量时的*意义完全不同。以下面的语句为例：

```
1.  int *p = &a;
2.  *p = 100;
```

第 1 行代码中*用来指明 p 是一个指针变量，第 2 行代码中*用来获取指针指向的数据。

需要注意的是，给指针变量本身赋值时不能加*。修改上面的语句：

```
1.  int *p;
2.  p = &a;
3.  *p = 100;
```

第 2 行代码中的 p 前面就不能加*。

指针变量也可以出现在普通变量能出现的任何表达式中，例如：

```
1.  int x, y, *px = &x, *py = &y;
2.  y = *px + 5;  //表示把x的内容加5并赋给y，*px+5相当于(*px)+5
3.  y = ++*px;  //px的内容加上1之后赋给y，++*px相当于++(*px)
4.  y = *px++;  //相当于y=(*px)++
5.  py = px;  //把一个指针的值赋给另一个指针
```

【示例】通过指针交换两个变量的值。

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      int a = 100, b = 999, temp;
5.      int *pa = &a, *pb = &b;
6.      printf("a=%d, b=%d\n", a, b);
7.      /*****开始交换*****/
8.      temp = *pa;  //将a的值先保存起来
9.      *pa = *pb;  //将b的值交给a
10.     *pb = temp;  //再将保存起来的a的值交给b
11.     /*****结束交换*****/
```

```
12.     printf("a=%d, b=%d\n", a, b);
13.     return 0;
14. }
```

运行结果：
a=100, b=999
a=999, b=100

从运行结果可以看出，a、b 的值已经发生了交换。需要注意的是临时变量 temp，它的作用特别重要，因为执行*pa = *pb;语句后 a 的值会被 b 的值覆盖，如果不先将 a 的值保存起来以后就找不到了。

关于 * 和 & 的谜题

假设有一个 int 类型的变量 a，pa 是指向它的指针，那么*&a 和&*pa 分别是什么意思呢？

&a 可以理解为(&a)，&a 表示取变量 a 的地址（等价于 pa），*(&a)表示取这个地址上的数据（等价于 *pa），绕来绕去，又回到了原点，*&a 仍然等价于 a。

&*pa 可以理解为&(*pa)，*pa 表示取得 pa 指向的数据（等价于 a），&(*pa)表示数据的地址（等价于 &a），所以&*pa 等价于 pa。

对星号*的总结

在我们目前所学到的语法中，星号*主要有三种用途：
- 表示乘法，例如 int a = 3, b = 5, c; c = a * b;，这是最容易理解的。
- 表示定义一个指针变量，以和普通变量区分开，例如 int a = 100; int *p = &a;。
- 表示获取指针指向的数据，是一种间接操作，例如 int a, b, *p = &a; *p = 100; b = *p;。

9.3 指针变量的运算（加法、减法和比较运算）

指针变量保存的是地址，而地址本质上是一个整数，所以指针变量可以进行部分运算，例如加法、减法、比较等，请看下面的代码：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      int    a = 10,   *pa = &a, *paa = &a;
5.      double b = 99.9, *pb = &b;
6.      char   c = '@',  *pc = &c;
7.      //最初的值
8.      printf("&a=%#X, &b=%#X, &c=%#X\n", &a, &b, &c);
9.      printf("pa=%#X, pb=%#X, pc=%#X\n", pa, pb, pc);
10.     //加法运算
11.     pa++; pb++; pc++;
12.     printf("pa=%#X, pb=%#X, pc=%#X\n", pa, pb, pc);
```

```
    //减法运算
    pa -= 2; pb -= 2; pc -= 2;
    printf("pa=%#X, pb=%#X, pc=%#X\n", pa, pb, pc);
    //比较运算
    if(pa == paa){
        printf("%d\n", *paa);
    }else{
        printf("%d\n", *pa);
    }
    return 0;
}
```

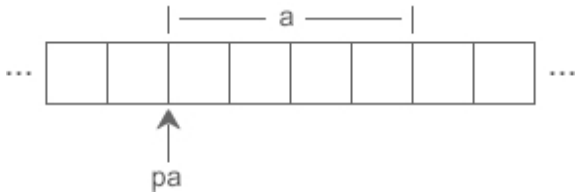
运行结果：

```
&a=0X28FF44, &b=0X28FF30, &c=0X28FF2B
pa=0X28FF44, pb=0X28FF30, pc=0X28FF2B
pa=0X28FF48, pb=0X28FF38, pc=0X28FF2C
pa=0X28FF40, pb=0X28FF28, pc=0X28FF2A
2686784
```

从运算结果可以看出：pa、pb、pc 每次加 1，它们的地址分别增加 4、8、1，正好是 int、double、char 类型的长度；减 2 时，地址分别减少 8、16、2，正好是 int、double、char 类型长度的 2 倍。

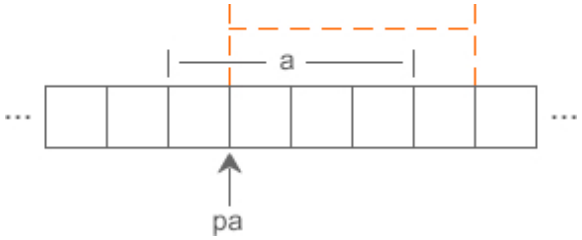
这很奇怪，指针变量加减运算的结果跟数据类型的长度有关，而不是简单地加 1 或减 1，这是为什么呢？

以 a 和 pa 为例，a 的类型为 int，占用 4 个字节，pa 是指向 a 的指针，如下图所示：



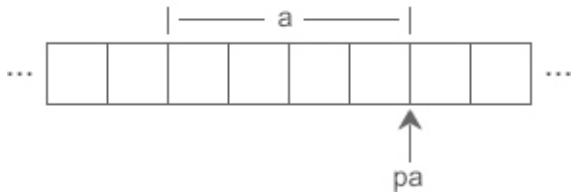
刚开始的时候，pa 指向 a 的开头，通过 *pa 读取数据时，从 pa 指向的位置向后移动 4 个字节，把这 4 个字节的内容作为要获取的数据，这 4 个字节也正好是变量 a 占用的内存。

如果 pa++;使得地址加 1 的话，就会变成如下图所示的指向关系：



这个时候 pa 指向整数 a 的中间，*pa 使用的是橙色虚线画出的 4 个字节，其中前 3 个是变量 a 的，后面 1 个是其它数据的，把它们“搅和”在一起显然没有实际的意义，取得的数据也会非常怪异。

如果 pa++;使得地址加 4 的话，正好能够完全跳过整数 a，指向它后面的内存，如下图所示：



我们知道，数组中的所有元素在内存中是连续排列的，如果一个指针指向了数组中的某个元素，那么加 1 就表示指向下一个元素，减 1 就表示指向上一个元素，这样指针的加减运算就具有了现实的意义，我们将在《C 语言数组指针》一节中深入探讨。

不过 C 语言并没有规定变量的存储方式，如果连续定义多个变量，它们有可能是挨着的，也有可能是分散的，这取决于变量的类型、编译器的实现以及具体的编译模式，所以对于指向普通变量的指针，我们往往不进行加减运算，虽然编译器并不会报错，但这样做没有意义，因为不知道它后面指向的是什么数据。

下面的例子是一个反面教材，警告读者不要尝试通过指针获取下一个变量的地址：

```
#include <stdio.h>

int main(){
    int a = 1, b = 2, c = 3;
    int *p = &c;
    int i;
    for(i=0; i<8; i++){
        printf("%d, ", *(p+i) );
    }
    return 0;
}
```

在 VS2010 Debug 模式下的运行结果为：

```
3, -858993460, -858993460, 2, -858993460, -858993460, 1, -858993460,
```

可以发现，变量 a、b、c 并不挨着，它们中间还参杂了别的辅助数据。

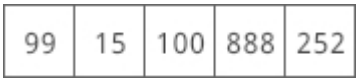
指针变量除了可以参与加减运算，还可以参与比较运算。当对指针变量进行比较运算时，比较的是指针变量本身的值，也就是数据的地址。如果地址相等，那么两个指针就指向同一份数据，否则就指向不同的数据。

上面的代码（第一个例子）在比较 pa 和 paa 的值时，pa 已经指向了 a 的上一份数据，所以它们不相等。而 a 的上一份数据又不知道是什么，所以会导致 printf() 输出一个没有意义的数，这正好印证了上面的观点，不要对指向普通变量的指针进行加减运算。

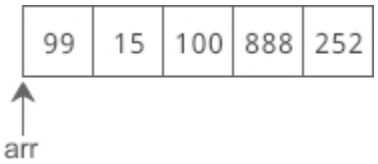
另外需要说明的是，不能对指针变量进行乘法、除法、取余等其他运算，除了会发生语法错误，也没有实际的含义。

9.4 C 语言数组指针（指向数组的指针）

数组（Array）是一系列具有相同类型的数据的集合，每一份数据叫做一个数组元素（Element）。数组中的所有元素在内存中是连续排列的，整个数组占用的是一块内存。以 `int arr[] = { 99, 15, 100, 888, 252 };` 为例，该数组在内存中的分布如下图所示：



定义数组时，要给出数组名和数组长度，数组名可以认为是一个指针，它指向数组的第 0 个元素。在 C 语言中，我们将第 0 个元素的地址称为数组的首地址。以上面的数组为例，下图是 arr 的指向：



> 数组名的本意是表示整个数组，也就是表示多份数据的集合，但在使用过程中经常会转换为指向数组第 0 个元素的指针，所以上面使用了“认为”一词，表示数组名和数组首地址并不总是等价。初学者可以暂时忽略这个细节，把数组名当做指向第 0 个元素的指针使用即可，我们将在 VIP 教程《数组和指针绝不等价，数组是另外一种类型》和《数组到底在什么时候会转换为指针》中再深入讨论这一细节。

下面的例子演示了如何以指针的方式遍历数组元素：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      int arr[] = { 99, 15, 100, 888, 252 };
5.      int len = sizeof(arr) / sizeof(int);  //求数组长度
6.      int i;
7.      for(i=0; i<len; i++){
8.          printf("%d  ", *(arr+i) );  //*(arr+i)等价于arr[i]
9.      }
10.     printf("\n");
11.     return 0;
12. }
```

运行结果：

```
99  15  100  888  252
```

第 5 行代码用来求数组的长度，sizeof(arr) 会获得整个数组所占用的字节数，sizeof(int) 会获得一个数组元素所占用的字节数，它们相除的结果就是数组包含的元素个数，也即数组长度。

第 8 行代码中我们使用了`*(arr+i)`这个表达式，arr 是数组名，指向数组的第 0 个元素，表示数组首地址，arr+i 指向数组的第 i 个元素，*(arr+i) 表示取第 i 个元素的数据，它等价于 arr[i]。

> arr 是 int*类型的指针，每次加 1 时它自身的值会增加 sizeof(int)，加 i 时自身的值会增加 sizeof(int)*i，这在《C 语言指针变量的运算》中已经进行了详细讲解。

我们也可以定义一个指向数组的指针，例如：

```
1.  int arr[] = { 99, 15, 100, 888, 252 };
2.  int *p = arr;
```

arr 本身就是一个指针，可以直接赋值给指针变量 p。arr 是数组第 0 个元素的地址，所以`int *p = arr;`也可以写作`int *p = &arr[0];`。也就是说，arr、p、&arr[0] 这三种写法都是等价的，它们都指向数组第 0 个元素，或者说指向

数组的开头。

> 再强调一遍，“arr 本身就是一个指针”这种表述并不准确，严格来说应该是“arr 被转换成了一个指针”。这里请大家先忽略这个细节，我们将在 VIP 教程《数组和指针绝不等价，数组是另外一种类型》和《数组到底在什么时候会转换为指针》中深入讨论。

如果一个指针指向了数组，我们就称它为数组指针（Array Pointer）。

数组指针指向的是数组中的一个具体元素，而不是整个数组，所以数组指针的类型和数组元素的类型有关，上面的例子中，p 指向的数组元素是 int 类型，所以 p 的类型必须也是 int *。

反过来想，p 并不知道它指向的是一个数组，p 只知道它指向的是一个整数，究竟如何使用 p 取决于程序员的编码。

更改上面的代码，使用数组指针来遍历数组元素：

```
#include <stdio.h>

int main(){
    int arr[] = { 99, 15, 100, 888, 252 };
    int i, *p = arr, len = sizeof(arr) / sizeof(int);

    for(i=0; i<len; i++){
        printf("%d  ", *(p+i) );
    }
    printf("\n");
    return 0;
}
```

数组在内存中只是数组元素的简单排列，没有开始和结束标志，在求数组的长度时不能使用 sizeof(p) / sizeof(int)，因为 p 只是一个指向 int 类型的指针，编译器并不知道它指向的到底是一个整数还是一系列整数（数组），所以 sizeof(p) 求得的是 p 这个指针变量本身所占用的字节数，而不是整个数组占用的字节数。

也就是说，根据数组指针不能逆推出整个数组元素的个数，以及数组从哪里开始、到哪里结束等信息。不像字符串，数组本身也没有特定的结束标志，如果不知道数组的长度，那么就无法遍历整个数组。

上节我们讲到，对指针变量进行加法和减法运算时，是根据数据类型的长度来计算的。如果一个指针变量 p 指向了数组的开头，那么 p+i 就指向数组的第 i 个元素；如果 p 指向了数组的第 n 个元素，那么 p+i 就是指向第 n+i 个元素；而不管 p 指向了数组的第几个元素，p+1 总是指向下一个元素，p-1 也总是指向上一个元素。

更改上面的代码，让 p 指向数组中的第二个元素：

```
#include <stdio.h>

int main(){
    int arr[] = { 99, 15, 100, 888, 252 };
    int *p = &arr[2];  //也可以写作 int *p = arr + 2;

```

```
7.      printf("%d, %d, %d, %d, %d\n", *(p-2), *(p-1), *p, *(p+1), *(p+2) );
8.      return 0;
9.  }
```

运行结果：
99, 15, 100, 888, 252

引入数组指针后，我们就有两种方案来访问数组元素了，一种是使用下标，另外一种是使用指针。

1) 使用下标

也就是采用 arr[i] 的形式访问数组元素。如果 p 是指向数组 arr 的指针，那么也可以使用 p[i] 来访问数组元素，它等价于 arr[i]。

2) 使用指针

也就是使用 *(p+i) 的形式访问数组元素。另外数组名本身也是指针，也可以使用 *(arr+i) 来访问数组元素，它等价于 *(p+i)。

不管是数组名还是数组指针，都可以使用上面的两种方式来访问数组元素。不同的是，数组名是常量，它的值不能改变，而数组指针是变量（除非特别指明它是常量），它的值可以任意改变。也就是说，数组名只能指向数组的开头，而数组指针可以先指向数组开头，再指向其他元素。

更改上面的代码，借助自增运算符来遍历数组元素：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      int arr[] = { 99, 15, 100, 888, 252 };
5.      int i, *p = arr, len = sizeof(arr) / sizeof(int);
6.
7.      for(i=0; i<len; i++){
8.          printf("%d  ", *p++ );
9.      }
10.     printf("\n");
11.     return 0;
12. }
```

运行结果：
99 15 100 888 252

第 8 行代码中，*p++ 应该理解为 *(p++)，每次循环都会改变 p 的值（p++ 使得 p 自身的值增加），以使 p 指向下一个数组元素。该语句不能写为 *arr++，因为 arr 是常量，而 arr++ 会改变它的值，这显然是错误的。

关于数组指针的谜题

假设 p 是指向数组 arr 中第 n 个元素的指针，那么 *p++、*++p、(*p)++ 分别是什么意思呢？

*p++ 等价于 *(p++)，表示先取得第 n 个元素的值，再将 p 指向下一个元素，上面已经进行了详细讲解。

*++p 等价于 *(++p)，会先进行 ++p 运算，使得 p 的值增加，指向下一个元素，整体上相当于 *(p+1)，所以会获得第 n+1 个数组元素的值。

(*p)++ 就非常简单了，会先取得第 n 个元素的值，再对该元素的值加 1。假设 p 指向第 0 个元素，并且第 0 个元素的值为 99，执行完该语句后，第 0 个元素的值就会变为 100。

9.5 C 语言字符串指针（指向字符串的指针）

C 语言中没有特定的字符串类型，我们通常是将字符串放在一个字符数组中，这在《C 语言字符数组和字符串》中已经进行了详细讲解，这里不妨再来演示一下：

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

int main(){
    char str[] = "http://c.biancheng.net";
    int len = strlen(str), i;
    //直接输出字符串
    printf("%s\n", str);
    //每次输出一个字符
    for(i=0; i<len; i++){
        printf("%c", str[i]);
    }
    printf("\n");
    return 0;
}
```

运行结果：
http://c.biancheng.net
http://c.biancheng.net

字符数组归根结底还是一个数组，上节讲到的关于指针和数组的规则同样也适用于字符数组。更改上面的代码，使用指针的方式来输出字符串：

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

int main(){
    char str[] = "http://c.biancheng.net";
    char *pstr = str;
    int len = strlen(str), i;

    //使用*(pstr+i)
    for(i=0; i<len; i++){
        printf("%c", *(pstr+i));
    }
```

```
    printf("\n");
    //使用pstr[i]
    for(i=0; i<len; i++){
        printf("%c", pstr[i]);
    }
    printf("\n");
    //使用*(str+i)
    for(i=0; i<len; i++){
        printf("%c", *(str+i));
    }
    printf("\n");

    return 0;
}
```

运行结果：

http://c.biancheng.net

http://c.biancheng.net

http://c.biancheng.net

除了字符数组，C 语言还支持另外一种表示字符串的方法，就是直接使用一个指针指向字符串，例如：

```
char *str = "http://c.biancheng.net";
```

或者：

```
char *str;
str = "http://c.biancheng.net";
```

字符串中的所有字符在内存中是连续排列的，str 指向的是字符串的第 0 个字符；我们通常将第 0 个字符的地址称为字符串的首地址。字符串中每个字符的类型都是 char，所以 str 的类型也必须是 char *。

下面的例子演示了如何输出这种字符串：

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

int main(){
    char *str = "http://c.biancheng.net";
    int len = strlen(str), i;

    //直接输出字符串
    printf("%s\n", str);
    //使用*(str+i)
    for(i=0; i<len; i++){
        printf("%c", *(str+i));
    }
    printf("\n");
```

```
15.     //使用str[i]
16.     for(i=0; i<len; i++){
17.         printf("%c", str[i]);
18.     }
19.     printf("\n");
20.
21.     return 0;
22. }
```

运行结果：
http://c.biancheng.net
http://c.biancheng.net
http://c.biancheng.net

这一切看起来和字符数组是多么地相似，它们都可以使用%s 输出整个字符串，都可以使用*或[ ]获取单个字符，这两种表示字符串的方式是不是就没有区别了呢？

有！它们最根本的区别是在内存中的存储区域不一样，字符数组存储在全局数据区或栈区，第二种形式的字符串存储在常量区。全局数据区和栈区的字符串（也包括其他数据）有读取和写入的权限，而常量区的字符串（也包括其他数据）只有读取权限，没有写入权限。

> 关于全局数据区、栈区、常量区以及其他的内存分区，我们将在《C 语言内存精讲》专题中详细讲解，相信你必将有所顿悟，从根本上理解 C 语言。

内存权限的不同导致的一个明显结果就是，字符数组在定义后可以读取和修改每个字符，而对于第二种形式的字符串，一旦被定义后就只能读取不能修改，任何对它的赋值都是错误的。

我们将第二种形式的字符串称为字符串常量，意思很明显，常量只能读取不能写入。请看下面的演示：

```
1.  #include <stdio.h>
2.  int main(){
3.      char *str = "Hello World!";
4.      str = "I love C!";  //正确
5.      str[3] = 'P';  //错误
6.
7.      return 0;
8.  }
```

这段代码能够正常编译和链接，但在运行时会出现段错误（Segment Fault）或者写入位置错误。

第 4 行代码是正确的，可以更改指针变量本身的指向；第 5 行代码是错误的，不能修改字符串中的字符。

到底使用字符数组还是字符串常量

在编程过程中如果只涉及到对字符串的读取，那么字符数组和字符串常量都能够满足要求；如果有写入（修改）操作，那么只能使用字符数组，不能使用字符串常量。

获取用户输入的字符串就是一个典型的写入操作，只能使用字符数组，不能使用字符串常量，请看下面的代码：

```
#include <stdio.h>
int main(){
    char str[30];
    gets(str);
    printf("%s\n", str);

    return 0;
}
```

运行结果：

C C++ Java Python JavaScript

C C++ Java Python JavaScript

最后我们来总结一下，C 语言有两种表示字符串的方法，一种是字符数组，另一种是字符串常量，它们在内存中的存储位置不同，使得字符数组可以读取和修改，而字符串常量只能读取不能修改。

9.6 C 语言数组灵活多变的访问形式

9.7 指针变量作为函数参数

在 C 语言中，函数的参数不仅可以是整数、小数、字符等具体的数据，还可以是指向它们的指针。用指针变量作函数参数可以将函数外部的地址传递到函数内部，使得在函数内部可以操作函数外部的数据，并且这些数据不会随着函数的结束而被销毁。

像数组、字符串、动态分配的内存等都是一系列数据的集合，没有办法通过一个参数全部传入函数内部，只能传递它们的指针，在函数内部通过指针来影响这些数据集合。

有的时候，对于整数、小数、字符等基本类型数据的操作也必须要借助指针，一个典型的例子就是交换两个变量的值。

有些初学者可能会使用下面的方法来交换两个变量的值：

```
#include <stdio.h>

void swap(int a, int b){
    int temp;  //临时变量
    temp = a;
    a = b;
    b = temp;
```

```
8.  }
9.
10. int main(){
11.     int a = 66, b = 99;
12.     swap(a, b);
13.     printf("a = %d, b = %d\n", a, b);
14.     return 0;
15. }
```

运行结果：
a = 66, b = 99

从结果可以看出，a、b 的值并没有发生改变，交换失败。这是因为 swap() 函数内部的 a、b 和 main() 函数内部的 a、b 是不同的变量，占用不同的内存，它们除了名字一样，没有其他任何关系，swap() 交换的是它内部 a、b 的值，不会影响它外部（main() 内部） a、b 的值。

改用指针变量作参数后就很容易解决上面的问题：

```
1.  #include <stdio.h>
2.
3.  void swap(int *p1, int *p2){
4.      int temp;  //临时变量
5.      temp = *p1;
6.      *p1 = *p2;
7.      *p2 = temp;
8.  }
9.
10. int main(){
11.     int a = 66, b = 99;
12.     swap(&a, &b);
13.     printf("a = %d, b = %d\n", a, b);
14.     return 0;
15. }
```

运行结果：
a = 99, b = 66

调用 swap() 函数时，将变量 a、b 的地址分别赋值给 p1、p2，这样 *p1、*p2 代表的就是变量 a、b 本身，交换 *p1、*p2 的值也就是交换 a、b 的值。函数运行结束后虽然会将 p1、p2 销毁，但它对外部 a、b 造成的影响是“持久化”的，不会随着函数的结束而“恢复原样”。

需要注意的是临时变量 temp，它的作用特别重要，因为执行`*p1 = *p2;`语句后 a 的值会被 b 的值覆盖，如果不先将 a 的值保存起来以后就找不到了。

这就好比拿来一瓶可乐和一瓶雪碧，要想把可乐倒进雪碧瓶、把雪碧倒进可乐瓶里面，就必须先找一个杯子，将两者之一先倒进杯子里面，再从杯子倒进瓶子里面。这里的杯子，就是一个“临时变量”，虽然只是倒倒手，但是也不可或缺。

用数组作函数参数

数组是一系列数据的集合，无法通过参数将它们一次性传递到函数内部，如果希望在函数内部操作数组，必须传递数组指针。下面的例子定义了一个函数 max()，用来查找数组中值最大的元素：

```
1.  #include <stdio.h>
2.
3.  int max(int *intArr, int len){
4.      int i, maxValue = intArr[0];  //假设第0个元素是最大值
5.      for(i=1; i<len; i++){
6.          if(maxValue < intArr[i]){
7.              maxValue = intArr[i];
8.          }
9.      }
10.
11.     return maxValue;
12. }
13.
14. int main(){
15.     int nums[6], i;
16.     int len = sizeof(nums)/sizeof(int);
17.     //读取用户输入的数据并赋值给数组元素
18.     for(i=0; i<len; i++){
19.         scanf("%d", nums+i);
20.     }
21.     printf("Max value is %d!\n", max(nums, len));
22.
23.     return 0;
24. }
```

运行结果：

12 55 30 8 93 27↙

Max value is 93!

参数 intArr 仅仅是一个数组指针，在函数内部无法通过这个指针获得数组长度，必须将数组长度作为函数参数传递到函数内部。数组 nums 的每个元素都是整数，scanf() 在读取用户输入的整数时，要求给出存储它的内存的地址，nums+i 就是第 i 个数组元素的地址。

用数组做函数参数时，参数也能够以“真正”的数组形式给出。例如对于上面的 max() 函数，它的参数可以写成下面的形式：

```
1.  int max(int intArr[6], int len){
2.      int i, maxValue = intArr[0];  //假设第0个元素是最大值
3.      for(i=1; i<len; i++){
4.          if(maxValue < intArr[i]){
5.              maxValue = intArr[i];
6.          }
7.      }
8.      return maxValue;
9.  }
```

int intArr[6]好像定义了一个拥有 6 个元素的数组，调用 max() 时可以将数组的所有元素“一股脑”传递进来。

读者也可以省略数组长度，把形参简写为下面的形式：

```
1.  int max(int intArr[], int len){
2.      int i, maxValue = intArr[0];  //假设第0个元素是最大值
3.      for(i=1; i<len; i++){
4.          if(maxValue < intArr[i]){
5.              maxValue = intArr[i];
6.          }
7.      }
8.      return maxValue;
9.  }
```

int intArr[]虽然定义了一个数组，但没有指定数组长度，好像可以接受任意长度的数组。

实际上这两种形式的数组定义都是假象，不管是 int intArr[6]还是 int intArr[]都不会创建一个数组出来，编译器也不会为它们分配内存，实际的数组是不存在的，它们最终还是会转换为 int *intArr 这样的指针。这就意味着，两种形式都不能将数组的所有元素“一股脑”传递进来，大家还得规规矩矩使用数组指针。

int intArr[6]这种形式只能说明函数期望用户传递的数组有 6 个元素，并不意味着数组只能有 6 个元素，真正传递的数组可以有少于或多于 6 个的元素。

需要强调的是，不管使用哪种方式传递数组，都不能在函数内部求得数组长度，因为 intArr 仅仅是一个指针，而不是真正的数组，所以必须要额外增加一个参数来传递数组长度。

C 语言为什么不允许直接传递数组的所有元素，而必须传递数组指针呢？

参数的传递本质上是一次赋值的过程，赋值就是对内存进行拷贝。所谓内存拷贝，是指将一块内存上的数据复制到另一块内存上。

对于像 int、float、char 等基本类型的数据，它们占用的内存往往只有几个字节，对它们进行内存拷贝非常快速。而数组是一系列数据的集合，数据的数量没有限制，可能很少，也可能成千上万，对它们进行内存拷贝有可能是一个漫长的过程，会严重拖慢程序的效率，为了防止技艺不佳的程序员写出低效的代码，C 语言没有从语法上支持数据集合的直接赋值。

除了 C 语言，C++、Java、Python 等其它语言也禁止对大块内存进行拷贝，在底层都使用类似指针的方式来实现。

9.8 指针作为函数返回值

C 语言允许函数的返回值是一个指针(地址)，我们将这样的函数称为指针函数。下面的例子定义了一个函数 strlong()，用来返回两个字符串中较长的一个：

```
1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  char *strlong(char *str1, char *str2){
5.      if(strlen(str1) >= strlen(str2)){
6.          return str1;
7.      }else{
8.          return str2;
9.      }
10. }
11.
12. int main(){
13.     char str1[30], str2[30], *str;
14.     gets(str1);
15.     gets(str2);
16.     str = strlong(str1, str2);
17.     printf("Longer string: %s\n", str);
18.
19.     return 0;
20. }
```

运行结果：

C Language↙

c.biancheng.net↙

Longer string: c.biancheng.net

用指针作为函数返回值时需要注意的一点是，函数运行结束后会销毁在它内部定义的所有局部数据，包括局部变量、局部数组和形式参数，函数返回的指针请尽量不要指向这些数据，C 语言没有任何机制来保证这些数据会一直有效，它们在后续使用过程中可能会引发运行时错误。请看下面的例子：

```
1.  #include <stdio.h>
2.
3.  int *func(){
4.      int n = 100;
5.      return &n;
```

```
6.  }
7.
8.  int main(){
9.      int *p = func(), n;
10.     n = *p;
11.     printf("value = %d\n", n);
12.     return 0;
13. }
```

运行结果：
value = 100

n 是 func() 内部的局部变量，func() 返回了指向 n 的指针，根据上面的观点，func() 运行结束后 n 将被销毁，使用 *p 应该获取不到 n 的值。但是从运行结果来看，我们的推理好像是错误的，func() 运行结束后 *p 依然可以获取局部变量 n 的值，这个上面的观点不是相悖吗？

为了进一步看清问题的本质，不妨将上面的代码稍作修改，在第 9~10 行之间增加一个函数调用，看看会有什么效果：

```
1.  #include <stdio.h>
2.
3.  int *func(){
4.      int n = 100;
5.      return &n;
6.  }
7.
8.  int main(){
9.      int *p = func(), n;
10.     printf("c.biancheng.net\n");
11.     n = *p;
12.     printf("value = %d\n", n);
13.     return 0;
14. }
```

运行结果：
c.biancheng.net
value = -2

可以看到，现在 p 指向的数据已经不是原来 n 的值了，它变成了一个毫无意义的甚至有些怪异的值。与前面的代码相比，该段代码仅仅是在 *p 之前增加了一个函数调用，这一细节的不同却导致运行结果有天壤之别，究竟是为什么呢？

前面我们说函数运行结束后会销毁所有的局部数据，这个观点并没错，大部分 C 语言教材也都强调了这一点。但是，这里所谓的销毁并不是将局部数据所占用的内存全部抹掉，而是程序放弃对它的使用权限，弃之不理，后面的代码可以随意使用这块内存。对于上面的两个例子，func() 运行结束后 n 的内存依然保持原样，值还是 100，如果使用及时也能够得到正确的数据，如果有其它函数被调用就会覆盖这块内存，得到的数据就失去了意义。

> 关于函数调用的原理以及函数如何占用内存的更多细节，我们将在《C 语言内存精讲》专题中深入探讨，相信你必将有所顿悟，解开心中的谜团。

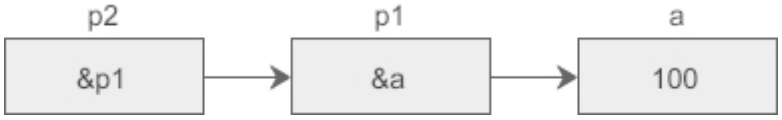
第一个例子在调用其他函数之前使用 *p 抢先获得了 n 的值并将它保存起来，第二个例子显然没有抓住机会，有其他函数被调用后才使用 *p 获取数据，这个时候已经晚了，内存已经被后来的函数覆盖了，而覆盖它的究竟是一份什么样的数据我们无从推断（一般是一个没有意义甚至有些怪异的值）。

9.9 二级指针（指向指针的指针）

指针可以指向一份普通类型的数据，例如 int、double、char 等，也可以指向一份指针类型的数据，例如 int *、double *、char * 等。

如果一个指针指向的是另外一个指针，我们就称它为二级指针，或者指向指针的指针。

假设有一个 int 类型的变量 a，p1 是指向 a 的指针变量，p2 又是指向 p1 的指针变量，它们的关系如下图所示：



将这种关系转换为 C 语言代码：

```
int a =100;
int *p1 = &a;
int **p2 = &p1;
```

指针变量也是一种变量，也会占用存储空间，也可以使用&获取它的地址。C 语言不限制指针的级数，每增加一级指针，在定义指针变量时就得增加一个星号*。p1 是一级指针，指向普通类型的数据，定义时有一个*；p2 是二级指针，指向一级指针 p1，定义时有两个*。

如果我们希望再定义一个三级指针 p3，让它指向 p2，那么可以这样写：

```
int ***p3 = &p2;
```

四级指针也是类似的道理：

```
int ****p4 = &p3;
```

实际开发中会经常使用一级指针和二级指针，几乎用不到高级指针。

想要获取指针指向的数据时，一级指针加一个*，二级指针加两个*，三级指针加三个*，以此类推，请看代码：

```
#include <stdio.h>

int main(){
    int a =100;
    int *p1 = &a;
    int **p2 = &p1;
    int ***p3 = &p2;

```

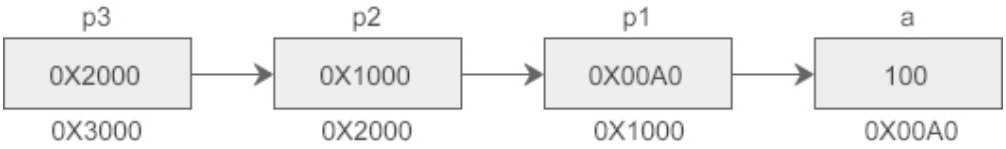
```
9.      printf("%d, %d, %d, %d\n", a, *p1, **p2, ***p3);
10.     printf("&p2 = %#X, p3 = %#X\n", &p2, p3);
11.     printf("&p1 = %#X, p2 = %#X, *p3 = %#X\n", &p1, p2, *p3);
12.     printf(" &a = %#X, p1 = %#X, *p2 = %#X, **p3 = %#X\n", &a, p1, *p2, **p3);
13.     return 0;
14. }
```

运行结果：

```
100, 100, 100, 100
&p2 = 0X28FF3C, p3 = 0X28FF3C
&p1 = 0X28FF40, p2 = 0X28FF40, *p3 = 0X28FF40
 &a = 0X28FF44, p1 = 0X28FF44, *p2 = 0X28FF44, **p3 = 0X28FF44
```

以三级指针 p3 为例来分析上面的代码。***p3** 等价于*(*(*p3))。*p3 得到的是 p2 的值，也即 p1 的地址；*(*p3) 得到的是 p1 的值，也即 a 的地址；经过三次“取值”操作后，*(*(*p3)) 得到的才是 a 的值。

假设 a、p1、p2、p3 的地址分别是 0X00A0、0X1000、0X2000、0X3000，它们之间的关系可以用下图来描述：



方框里面是变量本身的值，方框下面是变量的地址。

9.10 空指针 NULL 以及 void 指针

9.11 数组和指针绝不等价，数组是另外一种类型

9.12 数组到底在什么时候会转换为指针

9.13 C 语言指针数组（数组每个元素都是指针）

如果一个数组中的所有元素保存的都是指针，那么我们就称它为指针数组。指针数组的定义形式一般为：

```
dataType *arrayName[length];
```

[ ]的优先级高于*，该定义形式应该理解为：

```
dataType *(arrayName[length]);
```

括号里面说明 arrayName 是一个数组，包含了 length 个元素，括号外面说明每个元素的类型为 dataType *。

除了每个元素的数据类型不同，指针数组和普通数组在其他方面都是一样的，下面是一个简单的例子：

```
1.  #include <stdio.h>
2.  int main(){
3.      int a = 16, b = 932, c = 100;
4.      //定义一个指针数组
5.      int *arr[3] = {&a, &b, &c};//也可以不指定长度，直接写作 int *parr[]
6.      //定义一个指向指针数组的指针
7.      int **parr = arr;
8.      printf("%d, %d, %d\n", *arr[0], *arr[1], *arr[2]);
9.      printf("%d, %d, %d\n", **(parr+0), **(parr+1), **(parr+2));
10.
11.     return 0;
12. }
```

运行结果：
16, 932, 100
16, 932, 100

arr 是一个指针数组，它包含了 3 个元素，每个元素都是一个指针，在定义 arr 的同时，我们使用变量 a、b、c 的地址对它进行了初始化，这和普通数组是多么地类似。

parr 是指向数组 arr 的指针，确切地说是指向 arr 第 0 个元素的指针，它的定义形式应该理解为 int *(*parr)，括号中的*表示 parr 是一个指针，括号外面的 int *表示 parr 指向的数据的类型。arr 第 0 个元素的类型为 int *，所以在定义 parr 时要加两个 *。

第一个 printf() 语句中，arr[i] 表示获取第 i 个元素的值，该元素是一个指针，还需要在前面增加一个 * 才能取得它指向的数据，也即 *arr[i] 的形式。

第二个 printf() 语句中，parr+i 表示第 i 个元素的地址，*(parr+i) 表示获取第 i 个元素的值（该元素是一个指针），**(parr+i) 表示获取第 i 个元素指向的数据。

指针数组还可以和字符串数组结合使用，请看下面的例子：

```
#include <stdio.h>
int main(){
    char *str[3] = {
        "c.biancheng.net",
        "C语言中文网",
        "C Language"
    };
    printf("%s\n%s\n%s\n", str[0], str[1], str[2]);
    return 0;
}
```

运行结果：
c.biancheng.net
C 语言中文网
C Language

需要注意的是，字符数组 str 中存放的是字符串的首地址，不是字符串本身，字符串本身位于其他的内存区域，和字符数组是分开的。

也只有当指针数组中每个元素的类型都是 char *时，才能像上面那样给指针数组赋值，其他类型不行。

为了便于理解，可以将上面的字符串数组改成下面的形式，它们都是等价的。

```
#include <stdio.h>
int main(){
    char *str0 = "c.biancheng.net";
    char *str1 = "C语言中文网";
    char *str2 = "C Language";
    char *str[3] = {str0, str1, str2};
    printf("%s\n%s\n%s\n", str[0], str[1], str[2]);
    return 0;
}
```

9.14 一道题目玩转指针数组和二级指针

9.15 二维数组指针（指向二维数组的指针）

二维数组在概念上是二维的，有行和列，但在内存中所有的数组元素都是连续排列的，它们之间没有“缝隙”。以下面的二维数组 a 为例：

```
int a[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
```

从概念上理解，a 的分布像一个矩阵：

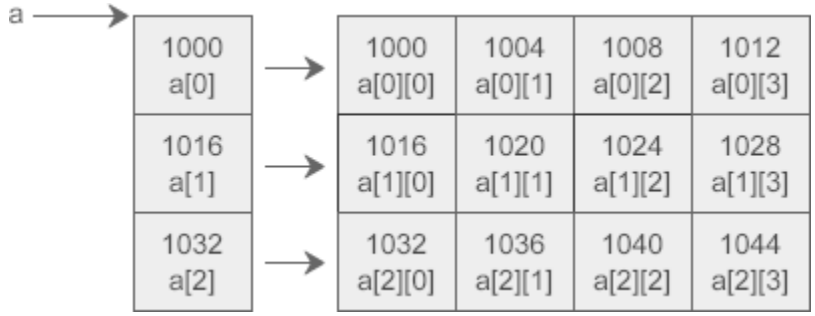
```
0   1   2   3
4   5   6   7
8   9   10  11
```

但在内存中，a 的分布是一维线性的，整个数组占用一块连续的内存：

C 语言中的二维数组是按行排列的，也就是先存放 a[0] 行，再存放 a[1] 行，最后存放 a[2] 行；每行中的 4 个元素也是依次存放。数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 4×(3×4) = 48 个字节。

C 语言允许把一个二维数组分解成多个一维数组来处理。对于数组 a，它可以分解成三个一维数组，即 a[0]、a[1]、a[2]。每一个一维数组又包含了 4 个元素，例如 a[0] 包含 a[0][0]、a[0][1]、a[0][2]、a[0][3]。

假设数组 a 中第 0 个元素的地址为 1000，那么每个一维数组的首地址如下图所示：



为了更好的理解指针和二维数组的关系，我们先来定义一个指向 a 的指针变量 p：

```
int (*p)[4] = a;
```

括号中的*表明 p 是一个指针，它指向一个数组，数组的类型为 int [4]，这正是 a 所包含的每个一维数组的类型。

[ ]的优先级高于*，( )是必须要加的，如果赤裸裸地写作 int *p[4]，那么应该理解为 int *(p[4])，p 就成了一个指针数组，而不是二维数组指针，这在《C 语言指针数组》中已经讲到。

对指针进行加法（减法）运算时，它前进（后退）的步长与它指向的数据类型有关，p 指向的数据类型是 int [4]，那么 p+1 就前进 4×4 = 16 个字节，p-1 就后退 16 个字节，这正好是数组 a 所包含的每个一维数组的长度。也就是说，p+1 会使得指针指向二维数组的下一行，p-1 会使得指针指向数组的上一行。

数组名 a 在表达式中也会被转换为和 p 等价的指针！

下面我们就来探索一下如何使用指针 p 来访问二维数组中的每个元素。按照上面的定义：
1) p 指向数组 a 的开头，也即第 0 行；p+1 前进一行，指向第 1 行。

2) *(p+1)表示取地址上的数据，也就是整个第 1 行数据。注意是一行数据，是多个数据，不是第 1 行中的第 0 个

元素，下面的运行结果有力地证明了这一点：

```
#include <stdio.h>
int main(){
    int a[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
    int (*p)[4] = a;
    printf("%d\n", sizeof(*(p+1)));

    return 0;
}
```

运行结果：
16

3) *(p+1)+1 表示第 1 行第 1 个元素的地址。如何理解呢？

*(p+1)单独使用时表示的是第 1 行数据，放在表达式中会被转换为第 1 行数据的首地址，也就是第 1 行第 0 个元素的地址，因为使用整行数据没有实际的含义，编译器遇到这种情况都会转换为指向该行第 0 个元素的指针；就像一维数组的名字，在定义时或者和 sizeof、& 一起使用时才表示整个数组，出现在表达式中就会被转换为指向数组第 0 个元素的指针。

4) *(*(p+1)+1)表示第 1 行第 1 个元素的值。很明显，增加一个 * 表示取地址上的数据。

根据上面的结论，可以很容易推出以下的等价关系：

```
a+i == p+i
a[i] == p[i] == *(a+i) == *(p+i)
a[i][j] == p[i][j] == *(a[i]+j) == *(p[i]+j) == *(*(a+i)+j) == *(*(p+i)+j)
```

【实例】使用指针遍历二维数组。

```
#include <stdio.h>
int main(){
    int a[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
    int(*p)[4];
    int i,j;
    p=a;
    for(i=0; i<3; i++){
        for(j=0; j<4; j++) printf("%2d  ",*(*(p+i)+j));
        printf("\n");
    }

    return 0;
}
```

运行结果：

```
 0   1   2   3
```

```
4   5   6   7
8   9   10  11
```

指针数组和二维数组指针的区别

指针数组和二维数组指针在定义时非常相似，只是括号的位置不同：

```
int *(p1[5]);  //指针数组，可以去掉括号直接写作 int *p1[5];
int (*p2)[5];  //二维数组指针，不能去掉括号
```

指针数组和二维数组指针有着本质上的区别：指针数组是一个数组，只是每个元素保存的都是指针，以上面的 p1 为例，在 32 位环境下它占用 4×5 = 20 个字节的内存。二维数组指针是一个指针，它指向一个二维数组，以上面的 p2 为例，它占用 4 个字节的内存。

9.16 函数指针（指向函数的指针）

一个函数总是占用一段连续的内存区域，函数名在表达式中有时也会被转换为该函数所在内存区域的首地址，这和数组名非常类似。我们可以把函数的这个首地址（或称入口地址）赋予一个指针变量，使指针变量指向函数所在的内存区域，然后通过指针变量就可以找到并调用该函数。这种指针就是函数指针。

函数指针的定义形式为：

```
returnType (*pointerName)(param list);
```

returnType 为函数返回值类型，pointerNmae 为指针名称，param list 为函数参数列表。参数列表中可以同时给出参数的类型和名称，也可以只给出参数的类型，省略参数的名称，这一点和函数原型非常类似。

注意( )的优先级高于*，第一个括号不能省略，如果写作 returnType *pointerName(param list);就成了函数原型，它表明函数的返回值类型为 returnType *。

【实例】用指针来实现对函数的调用。

```
#include <stdio.h>

//返回两个数中较大的一个
int max(int a, int b){
    return a>b ? a : b;
}

int main(){
    int x, y, maxval;
    //定义函数指针
    int (*pmax)(int, int) = max;  //也可以写作int (*pmax)(int a, int b)
    printf("Input two numbers:");
    scanf("%d %d", &x, &y);
    maxval = (*pmax)(x, y);
    printf("Max value: %d\n", maxval);
```

```
16.
17.     return 0;
18. }
```

运行结果：
Input two numbers:10 50↙
Max value: 50

第 14 行代码对函数进行了调用。pmax 是一个函数指针，在前面加 * 就表示对它指向的函数进行调用。注意()的优先级高于*，第一个括号不能省略。

9.17 只需一招，彻底攻克 C 语言指针，再复杂的指针都不怕

9.18 main()函数的高级用法：接收用户输入的数据

您好，您正在阅读高级教程，即将认识到 C 语言的本质，并掌握一些“黑科技”。阅读高级教程能够醍醐灌顶，颠覆三观，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

9.19 对 C 语言指针的总结

指针（Pointer）就是内存的地址，C 语言允许用一个变量来存放指针，这种变量称为指针变量。指针变量可以存放基本类型数据的地址，也可以存放数组、函数以及其他指针变量的地址。

程序在运行过程中需要的是数据和指令的地址，变量名、函数名、字符串名和数组名在本质上是一样的，它们都是地址的助记符：在编写代码的过程中，我们认为变量名表示的是数据本身，而函数名、字符串名和数组名表示的是代码块或数据块的首地址；程序被编译和链接后，这些名字都会消失，取而代之的是它们对应的地址。

常见指针变量的定义

定 义	含 义
int *p;	p 可以指向 int 类型的数据，也可以指向类似 int arr[n] 的数组。
int **p;	p 为二级指针，指向 int * 类型的数据。
int *p[n];	p 为指针数组。[ ] 的优先级高于 *，所以应该理解为 int *(p[n]);

int (*p)[n];	p 为二维数组指针。
int *p();	p 是一个函数，它的返回值类型为 int *。
int (*p)();	p 是一个函数指针，指向原型为 int func() 的函数。

1) 指针变量可以进行加减运算，例如 p++、p+i、p-=i。指针变量的加减运算并不是简单的加上或减去一个整数，而是跟指针指向的数据类型有关。

2) 给指针变量赋值时，要将一份数据的地址赋给它，不能直接赋给一个整数，例如 int *p = 1000;是没有意义的，使用过程中一般会导致程序崩溃。

3) 使用指针变量之前一定要初始化，否则就不能确定指针指向哪里，如果它指向的内存没有使用权限，程序就崩溃了。对于暂时没有指向的指针，建议赋值 NULL。

4) 两个指针变量可以相减。如果两个指针变量指向同一个数组中的某个元素，那么相减的结果就是两个指针之间相差的元素个数。

5) 数组也是有类型的，数组名的本意是表示一组类型相同的数据。在定义数组时，或者和 sizeof、& 运算符一起使用时数组名才表示整个数组，表达式中的数组名会被转换为一个指向数组的指针。

第 10 章 C 语言结构体

C 语言结构体（Struct）从本质上讲是一种自定义的数据类型，只不过这种数据类型比较复杂，是由 int、char、float 等基本类型组成的。你可以认为结构体是一种聚合类型。

在实际开发中，我们可以将一组类型不同的、但是用来描述同一件事物的变量放到结构体中。例如，在校学生有姓名、年龄、身高、成绩等属性，学了结构体后，我们就不需要再定义多个变量了，将它们都放到结构体中即可。

此外，本章还讲解了与位操作有关的知识点，比如位域、位运算等。

本章目录：

1. 什么是结构体？
2. 结构体数组（带实例演示）
3. 结构体指针（指向结构体的指针）
4. C 语言枚举类型（enum 用法）
5. C 语言共用体（union 用法）
6. 大端小端以及判别方式
7. C 语言位域（位段）
8. C 语言位运算详解
9. 使用位运算对数据或文件内容进行加密

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

10.1 什么是结构体？

前面的教程中我们讲解了数组（Array），它是一组具有相同类型的数据的集合。但在实际的编程过程中，我们往往还需要一组类型不同的数据，例如对于学生信息登记表，姓名为字符串，学号为整数，年龄为整数，所在的学习小组为字符，成绩为小数，因为数据类型不同，显然不能用一个数组来存放。

在 C 语言中，可以使用**结构体（Struct）**来存放一组不同类型的数据。结构体的定义形式为：

```
struct 结构体名{
    结构体所包含的变量或数组
};
```

结构体是一种集合，它里面包含了多个变量或数组，它们的类型可以相同，也可以不同，每个这样的变量或数组都称为结构体的成员（Member）。请看下面的一个例子：

```
1.  struct stu{
2.      char *name;  //姓名
3.      int num;  //学号
4.      int age;  //年龄
5.      char group;  //所在学习小组
6.      float score;  //成绩
7.  };
```

stu 为结构体名，它包含了 5 个成员，分别是 name、num、age、group、score。结构体成员的定义方式与变量和数组的定义方式相同，只是不能初始化。

> 注意大括号后面的分号;不能少，这是一条完整的语句。

结构体也是一种数据类型，它由程序员自己定义，可以包含多个其他类型的数据。

像 int、float、char 等是由 C 语言本身提供的数据类型，不能再进行分拆，我们称之为基本数据类型；而结构体可以包含多个基本类型的数据，也可以包含其他的结构体，我们将它称为复杂数据类型或构造数据类型。

结构体变量

既然结构体是一种数据类型，那么就可以用它来定义变量。例如：

```
struct stu stu1, stu2;
```

定义了两个变量 stu1 和 stu2，它们都是 stu 类型，都由 5 个成员组成。注意关键字 struct 不能少。

stu 就像一个“模板”，定义出来的变量都具有相同的性质。也可以将结构体比作“图纸”，将结构体变量比作“零件”，根据同一张图纸生产出来的零件的特性都是一样的。

你也可以在定义结构体的同时定义结构体变量：

```
1.  struct stu{
2.      char *name;  //姓名
3.      int num;  //学号
```

```
    int age;  //年龄
    char group;  //所在学习小组
    float score;  //成绩
} stu1, stu2;
```

将变量放在结构体定义的最后即可。

如果只需要 stu1、stu2 两个变量，后面不需要再使用结构体名定义其他变量，那么在定义时也可以不给出结构体名，如下所示：

```
struct{  //没有写 stu
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在学习小组
    float score;  //成绩
} stu1, stu2;
```

这样做书写简单，但是因为没有结构体名，后面就没法用该结构体定义新的变量。

理论上讲结构体的各个成员在内存中是连续存储的，和数组非常类似，例如上面的结构体变量 stu1、stu2 的内存分布如下图所示，共占用 4+4+4+1+4 = 17 个字节。

但是在编译器的具体实现中，各个成员之间可能会存在缝隙，对于 stu1、stu2，成员变量 group 和 score 之间就存在 3 个字节的空白填充（见下图）。这样算来，stu1、stu2 其实占用了 17 + 3 = 20 个字节。

关于成员变量之间存在“裂缝”的原因，我们将在《C 语言内存精讲》专题中的《C 语言内存对齐，提高寻址效率》一节中详细讲解。

成员的获取和赋值

结构体和数组类似，也是一组数据的集合，整体使用没有太大的意义。数组使用下标[]获取单个元素，结构体使用点号.获取单个成员。获取结构体成员的一般格式为：

```
结构体变量名.成员名;
```

通过这种方式可以获取成员的值，也可以给成员赋值：

```
#include <stdio.h>
int main(){
    struct{
        char *name;  //姓名
        int num;  //学号
        int age;  //年龄
        char group;  //所在小组
```

```
8.          float score;  //成绩
9.      } stu1;
10.
11.     //给结构体成员赋值
12.     stu1.name = "Tom";
13.     stu1.num = 12;
14.     stu1.age = 18;
15.     stu1.group = 'A';
16.     stu1.score = 136.5;
17.
18.     //读取结构体成员的值
19.     printf("%s的学号是%d，年龄是%d，在%c组，今年的成绩是%.1f！\n", stu1.name, stu1.num, stu1.age,
    stu1.group, stu1.score);
20.
21.     return 0;
22. }
```

运行结果：
Tom 的学号是 12，年龄是 18，在 A 组，今年的成绩是 136.5！

除了可以对成员进行逐一赋值，也可以在定义时整体赋值，例如：

```
1.  struct{
2.      char *name;  //姓名
3.      int num;  //学号
4.      int age;  //年龄
5.      char group;  //所在小组
6.      float score;  //成绩
7.  } stu1, stu2 = { "Tom", 12, 18, 'A', 136.5 };
```

不过整体赋值仅限于定义结构体变量的时候，在使用过程中只能对成员逐一赋值，这和数组的赋值非常类似。

需要注意的是，结构体是一种自定义的数据类型，是创建变量的模板，不占用内存空间；结构体变量才包含了实实在在的数据，需要内存空间来存储。

10.2 结构体数组（带实例演示）

所谓结构体数组，是指数组中的每个元素都是一个结构体。在实际应用中，C 语言结构体数组常被用来表示一个拥有相同数据结构的群体，比如一个班的学生、一个车间的职工等。

在 C 语言中，定义结构体数组和定义结构体变量的方式类似，请看下面的例子：

```
1.  struct stu{
2.      char *name;  //姓名
3.      int num;  //学号
4.      int age;  //年龄
5.      char group;  //所在小组
```

```
    float score;  //成绩
}class[5];
```

表示一个班级有 5 个学生。

结构体数组在定义的同时也可以初始化，例如：

```
struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
}class[5] = {
    {"Li ping", 5, 18, 'C', 145.0},
    {"Zhang ping", 4, 19, 'A', 130.5},
    {"He fang", 1, 18, 'A', 148.5},
    {"Cheng ling", 2, 17, 'F', 139.0},
    {"Wang ming", 3, 17, 'B', 144.5}
};
```

当对数组中全部元素赋值时，也可不给出数组长度，例如：

```
struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
}class[] = {
    {"Li ping", 5, 18, 'C', 145.0},
    {"Zhang ping", 4, 19, 'A', 130.5},
    {"He fang", 1, 18, 'A', 148.5},
    {"Cheng ling", 2, 17, 'F', 139.0},
    {"Wang ming", 3, 17, 'B', 144.5}
};
```

结构体数组的使用也很简单，例如，获取 Wang ming 的成绩：

```
class[4].score;
```

修改 Li ping 的学习小组：

```
class[0].group = 'B';
```

【示例】计算全班学生的总成绩、平均成绩和以及 140 分以下的人数。

```
#include <stdio.h>

struct{
```

```
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
}class[] = {
    {"Li ping", 5, 18, 'C', 145.0},
    {"Zhang ping", 4, 19, 'A', 130.5},
    {"He fang", 1, 18, 'A', 148.5},
    {"Cheng ling", 2, 17, 'F', 139.0},
    {"Wang ming", 3, 17, 'B', 144.5}
};

int main(){
    int i, num_140 = 0;
    float sum = 0;
    for(i=0; i<5; i++){
        sum += class[i].score;
        if(class[i].score < 140) num_140++;
    }
    printf("sum=%.2f\naverage=%.2f\nnum_140=%d\n", sum, sum/5, num_140);
    return 0;
}
```

运行结果：

```
sum=707.50
average=141.50
num_140=2
```

10.3 结构体指针（指向结构体的指针）

当一个指针变量指向结构体时，我们就称它为**结构体指针**。C 语言结构体指针的定义形式一般为：

```
struct 结构体名 *变量名;
```

下面是一个定义结构体指针的实例：

```
//结构体
struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
} stu1 = { "Tom", 12, 18, 'A', 136.5 };
//结构体指针
struct stu *pstu = &stu1;
```

也可以在定义结构体的同时定义结构体指针：

```
struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
} stu1 = { "Tom", 12, 18, 'A', 136.5 }, *pstu = &stu1;
```

注意，结构体变量名和数组名不同，数组名在表达式中会被转换为数组指针，而结构体变量名不会，无论在任何表达式中它表示的都是整个集合本身，要想取得结构体变量的地址，必须在前面加&，所以给 pstu 赋值只能写作：

```
struct stu *pstu = &stu1;
```

而不能写作：

```
struct stu *pstu = stu1;
```

还应该注意，结构体和结构体变量是两个不同的概念：结构体是一种数据类型，是一种创建变量的模板，编译器不会为它分配内存空间，就像 int、float、char 这些关键字本身不占用内存一样；结构体变量才包含实实在在的数据，才需要内存来存储。下面的写法是错误的，不可能去取一个结构体名的地址，也不能将它赋值给其他变量：

```
struct stu *pstu = &stu;
struct stu *pstu = stu;
```

获取结构体成员

通过结构体指针可以获取结构体成员，一般形式为：

```
(*pointer).memberName
```

或者：

```
pointer->memberName
```

第一种写法中，.的优先级高于*，(*pointer)两边的括号不能少。如果去掉括号写作*pointer.memberName，那么就等效于*(pointer.memberName)，这样意义就完全不对了。

第二种写法中，->是一个新的运算符，习惯称它为“箭头”，有了它，可以通过结构体指针直接取得结构体成员；这也是->在 C 语言中的唯一用途。

上面的两种写法是等效的，我们通常采用后面的写法，这样更加直观。

【示例】结构体指针的使用。

```
#include <stdio.h>
int main(){
    struct{
        char *name;  //姓名
        int num;  //学号
        int age;  //年龄
```

```
        char group;  //所在小组
        float score;  //成绩
    } stu1 = { "Tom", 12, 18, 'A', 136.5 }, *pstu = &stu1;

    //读取结构体成员的值
    printf("%s的学号是%d，年龄是%d，在%c组，今年的成绩是%.1f！\n", (*pstu).name, (*pstu).num, (*pstu).age, (*pstu).group, (*pstu).score);
    printf("%s的学号是%d，年龄是%d，在%c组，今年的成绩是%.1f！\n", pstu->name, pstu->num, pstu->age, pstu->group, pstu->score);

    return 0;
}
```

运行结果：

Tom 的学号是 12，年龄是 18，在 A 组，今年的成绩是 136.5！
Tom 的学号是 12，年龄是 18，在 A 组，今年的成绩是 136.5！

【示例】结构体数组指针的使用。

```
#include <stdio.h>

struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
}stus[] = {
    {"Zhou ping", 5, 18, 'C', 145.0},
    {"Zhang ping", 4, 19, 'A', 130.5},
    {"Liu fang", 1, 18, 'A', 148.5},
    {"Cheng ling", 2, 17, 'F', 139.0},
    {"Wang ming", 3, 17, 'B', 144.5}
}, *ps;

int main(){
    //求数组长度
    int len = sizeof(stus) / sizeof(struct stu);
    printf("Name\t\tNum\tAge\tGroup\tScore\t\n");
    for(ps=stus; ps<stus+len; ps++){
        printf("%s\t%d\t%d\t%c\t%.1f\n", ps->name, ps->num, ps->age, ps->group, ps->score);
    }

    return 0;
}
```

运行结果：

```
Name            Num     Age     Group   Score
Zhou ping       5       18      C       145.0
Zhang ping      4       19      A       130.5
Liu fang        1       18      A       148.5
Cheng ling      2       17      F       139.0
Wang ming       3       17      B       144.5
```

结构体指针作为函数参数

结构体变量名代表的是整个集合本身，作为函数参数时传递的整个集合，也就是所有成员，而不是像数组一样被编译器转换成一个指针。如果结构体成员较多，尤其是成员为数组时，传送的时间和空间开销会很大，影响程序的运行效率。所以最好的办法就是使用结构体指针，这时由实参传向形参的只是一个地址，非常快速。

【示例】计算全班学生的总成绩、平均成绩和以及 140 分以下的人数。

```
#include <stdio.h>

struct stu{
    char *name;  //姓名
    int num;  //学号
    int age;  //年龄
    char group;  //所在小组
    float score;  //成绩
}stus[] = {
    {"Li ping", 5, 18, 'C', 145.0},
    {"Zhang ping", 4, 19, 'A', 130.5},
    {"He fang", 1, 18, 'A', 148.5},
    {"Cheng ling", 2, 17, 'F', 139.0},
    {"Wang ming", 3, 17, 'B', 144.5}
};

void average(struct stu *ps, int len);

int main(){
    int len = sizeof(stus) / sizeof(struct stu);
    average(stus, len);
    return 0;
}

void average(struct stu *ps, int len){
    int i, num_140 = 0;
    float average, sum = 0;
    for(i=0; i<len; i++){
        sum += (ps + i) -> score;
```

```
30.          if((ps + i)->score < 140) num_140++;
31.      }
32.      printf("sum=%.2f\naverage=%.2f\nnum_140=%d\n", sum, sum/5, num_140);
33.  }
```

运行结果：
sum=707.50
average=141.50
num_140=2

10.4 C 语言枚举类型（enum 关键字）

在实际编程中，有些数据的取值往往是有限的，只能是非常少量的整数，并且最好为每个值都取一个名字，以方便在后续代码中使用，比如一个星期只有七天，一年只有十二个月，一个班每周有六门课程等。

以每周七天为例，我们可以使用#define 命令来给每天指定一个名字：

```
1.  #include <stdio.h>
2.
3.  #define Mon 1
4.  #define Tues 2
5.  #define Wed 3
6.  #define Thurs 4
7.  #define Fri 5
8.  #define Sat 6
9.  #define Sun 7
10.
11. int main(){
12.      int day;
13.      scanf("%d", &day);
14.      switch(day){
15.          case Mon: puts("Monday"); break;
16.          case Tues: puts("Tuesday"); break;
17.          case Wed: puts("Wednesday"); break;
18.          case Thurs: puts("Thursday"); break;
19.          case Fri: puts("Friday"); break;
20.          case Sat: puts("Saturday"); break;
21.          case Sun: puts("Sunday"); break;
22.          default: puts("Error!");
23.      }
24.      return 0;
25. }
```

运行结果：
5↙
Friday

#define 命令虽然能解决问题，但也带来了不小的副作用，导致宏名过多，代码松散，看起来总有点不舒服。C 语言提供了一种**枚举（Enum）类型**，能够列出所有可能的取值，并给它们取一个名字。

枚举类型的定义形式为：

```
enum typeName{ valueName1, valueName2, valueName3, ..... };
```

enum 是一个新的关键字，专门用来定义枚举类型，这也是它在 C 语言中的唯一用途；typeName 是枚举类型的名字；valueName1, valueName2, valueName3, ...是每个值对应的名字的列表。注意最后的;不能少。

例如，列出一个星期有几天：

```
enum week{ Mon, Tues, Wed, Thurs, Fri, Sat, Sun };
```

可以看到，我们仅仅给出了名字，却没有给出名字对应的值，这是因为枚举值默认从 0 开始，往后逐个加 1（递增）；也就是说，week 中的 Mon、Tues ..... Sun 对应的值分别为 0、1 ..... 6。

我们也可以给每个名字都指定一个值：

```
enum week{ Mon = 1, Tues = 2, Wed = 3, Thurs = 4, Fri = 5, Sat = 6, Sun = 7 };
```

更为简单的方法是只给第一个名字指定值：

```
enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun };
```

这样枚举值就从 1 开始递增，跟上面的写法是等效的。

枚举是一种类型，通过它可以定义枚举变量：

```
enum week a, b, c;
```

也可以在定义枚举类型的同时定义变量：

```
enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun } a, b, c;
```

有了枚举变量，就可以把列表中的值赋给它：

```
enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun };
enum week a = Mon, b = Wed, c = Sat;
```

或者：

```
enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun } a = Mon, b = Wed, c = Sat;
```

【示例】判断用户输入的是星期几。

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun } day;
5.      scanf("%d", &day);
6.      switch(day){
```

```
7.          case Mon: puts("Monday"); break;
8.          case Tues: puts("Tuesday"); break;
9.          case Wed: puts("Wednesday"); break;
10.         case Thurs: puts("Thursday"); break;
11.         case Fri: puts("Friday"); break;
12.         case Sat: puts("Saturday"); break;
13.         case Sun: puts("Sunday"); break;
14.         default: puts("Error!");
15.     }
16.     return 0;
17. }
```

运行结果：

```
4↙
Thursday
```

需要注意的两点是：

1) 枚举列表中的 Mon、Tues、Wed 这些标识符的作用范围是全局的（严格来说是 main() 函数内部），不能再定义与它们名字相同的变量。

2) Mon、Tues、Wed 等都是常量，不能对它们赋值，只能将它们的值赋给其他的变量。

枚举和宏其实非常类似：宏在预处理阶段将名字替换成对应的值，枚举在编译阶段将名字替换成对应的值。我们可以将枚举理解为编译阶段的宏。

对于上面的代码，在编译的某个时刻会变成类似下面的样子：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun } day;
5.      scanf("%d", &day);
6.      switch(day){
7.          case 1: puts("Monday"); break;
8.          case 2: puts("Tuesday"); break;
9.          case 3: puts("Wednesday"); break;
10.         case 4: puts("Thursday"); break;
11.         case 5: puts("Friday"); break;
12.         case 6: puts("Saturday"); break;
13.         case 7: puts("Sunday"); break;
14.         default: puts("Error!");
15.     }
16.     return 0;
17. }
```

Mon、Tues、Wed 这些名字都被替换成了对应的数字。这意味着，Mon、Tues、Wed 等都不是变量，它们不占用数据区（常量区、全局数据区、栈区和堆区）的内存，而是直接被编译到命令里面，放到代码区，所以不能用&取

得它们的地址。这就是枚举的本质。

> 关于程序在内存中的分区以及各个分区的作用，我们将在《C 语言内存精讲》专题中的《Linux 下 C 语言程序的内存布局（内存模型）》一节中详细讲解。

我们在《C 语言 switch case 语句》一节中讲过，case 关键字后面必须是一个整数，或者是结果为整数的表达式，但不能包含任何变量，正是由于 Mon、Tues、Wed 这些名字最终会被替换成一个整数，所以它们才能放在 case 后面。

枚举类型变量需要存放的是一个整数，我猜测它的长度和 int 应该相同，下面来验证一下：

```
#include <stdio.h>

int main(){
    enum week{ Mon = 1, Tues, Wed, Thurs, Fri, Sat, Sun } day = Mon;
    printf("%d, %d, %d, %d, %d\n", sizeof(enum week), sizeof(day), sizeof(Mon), sizeof(Wed), sizeof(int) );
    return 0;
}
```

运行结果：
4, 4, 4, 4, 4

10.5 C 语言共用体（union 关键字）

通过前面的讲解，我们知道结构体（Struct）是一种构造类型或复杂类型，它可以包含多个类型不同的成员。在 C 语言中，还有另外一种和结构体非常类似的语法，叫做**共用体（Union）**，它的定义格式为：

```
union 共用体名{
    成员列表
};
```

> 共用体有时也被称为联合或者联合体，这也是 Union 这个单词的本意。

结构体和共用体的区别在于：结构体的各个成员会占用不同的内存，互相之间没有影响；而共用体的所有成员占用同一段内存，修改一个成员会影响其余所有成员。

结构体占用的内存大于等于所有成员占用的内存的总和（成员之间可能会存在缝隙），共用体占用的内存等于最长的成员占用的内存。共用体使用了内存覆盖技术，同一时刻只能保存一个成员的值，如果对新的成员赋值，就会把原来成员的值覆盖掉。

共用体也是一种自定义类型，可以通过它来创建变量，例如：

```
union data{
    int n;
    char ch;
    double f;
```

```
};
union data a, b, c;
```

上面是先定义共用体，再创建变量，也可以在定义共用体的同时创建变量：

```
union data{
    int n;
    char ch;
    double f;
} a, b, c;
```

如果不再定义新的变量，也可以将共用体的名字省略：

```
union{
    int n;
    char ch;
    double f;
} a, b, c;
```

共用体 data 中，成员 f 占用的内存最多，为 8 个字节，所以 data 类型的变量（也就是 a、b、c）也占用 8 个字节的内存，请看下面的演示：

```
#include <stdio.h>

union data{
    int n;
    char ch;
    short m;
};

int main(){
    union data a;
    printf("%d, %d\n", sizeof(a), sizeof(union data) );
    a.n = 0x40;
    printf("%X, %c, %hX\n", a.n, a.ch, a.m);
    a.ch = '9';
    printf("%X, %c, %hX\n", a.n, a.ch, a.m);
    a.m = 0x2059;
    printf("%X, %c, %hX\n", a.n, a.ch, a.m);
    a.n = 0x3E25AD54;
    printf("%X, %c, %hX\n", a.n, a.ch, a.m);

    return 0;
}
```

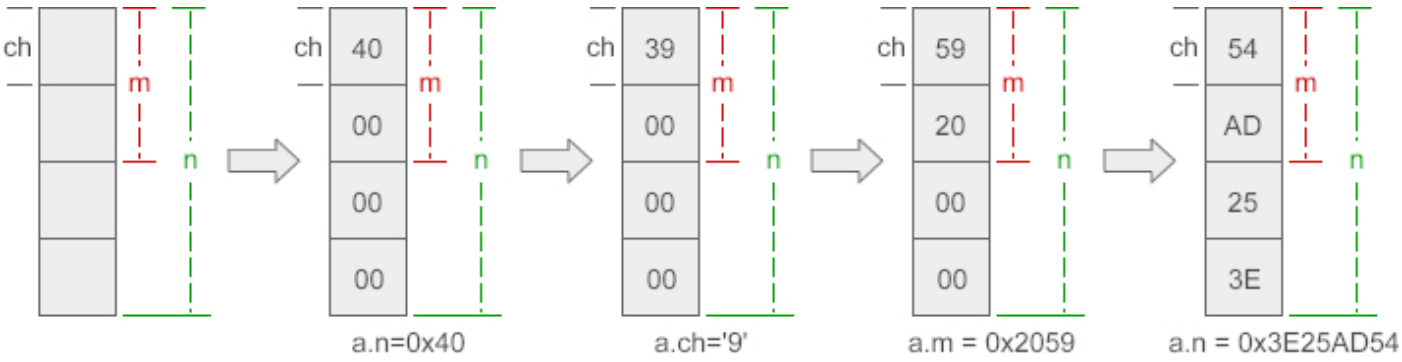
运行结果：

```
4, 4
40, @, 40
39, 9, 39
```

```
2059, Y, 2059
3E25AD54, T, AD54
```

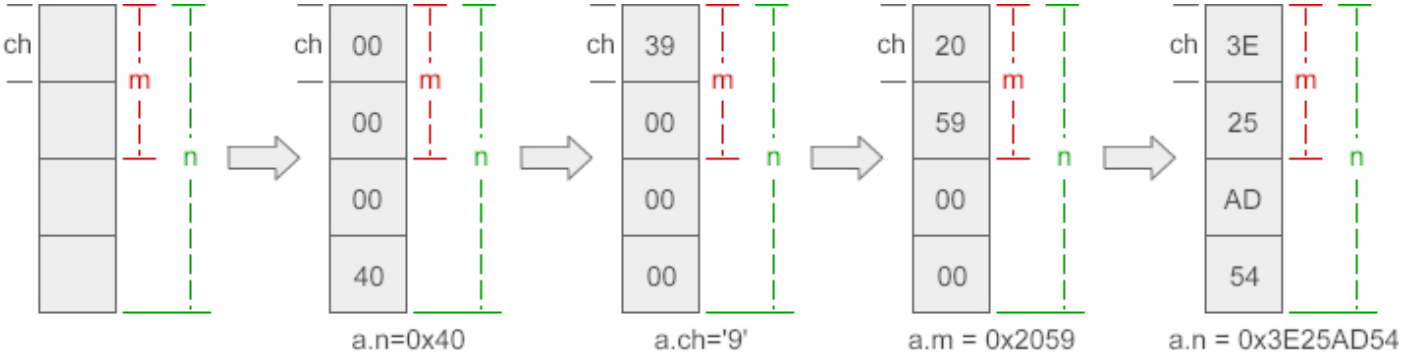
这段代码不但验证了共用体的长度，还说明共用体成员之间会相互影响，修改一个成员的值会影响其他成员。

要想理解上面的输出结果，弄清成员之间究竟是如何相互影响的，就得了解各个成员在内存中的分布。以上面的 data 为例，各个成员在内存中的分布如下：



成员 n、ch、m 在内存中“对齐”到一头，对 ch 赋值修改的是前一个字节，对 m 赋值修改的是前两个字节，对 n 赋值修改的是全部字节。也就是说，ch、m 会影响到 n 的一部分数据，而 n 会影响到 ch、m 的全部数据。

上图是在绝大多数 PC 机上的内存分布情况，如果是 51 单片机，情况就会有所不同：



为什么不同的机器会有不同的分布情况呢？这跟机器的存储模式有关，我们将在 VIP 教程《大端小端以及判别方式》一节中展开探讨。

共用体的应用

共用体在一般的编程中应用较少，在单片机中应用较多。对于 PC 机，经常使用到的一个实例是： 现有一张关于学生信息和教师信息的表格。学生信息包括姓名、编号、性别、职业、分数，教师的信息包括姓名、编号、性别、职业、教学科目。请看下面的表格：

Name	Num	Sex	Profession	Score / Course
HanXiaoXiao	501	f	s	89.5
YanWeiMin	1011	m	t	math
LiuZhenTao	109	f	t	English
ZhaoFeiYan	982	m	s	95.0

f 和 m 分别表示女性和男性，s 表示学生，t 表示教师。可以看出，学生和教师所包含的数据是不同的。现在要求把这些信息放在同一个表格中，并设计程序输入人员信息然后输出。

如果把每个人的信息都看作一个结构体变量的话，那么教师和学生的前 4 个成员变量是一样的，第 5 个成员变量可能是 score 或者 course。当第 4 个成员变量的值是 s 的时候，第 5 个成员变量就是 score；当第 4 个成员变量的值是 t 的时候，第 5 个成员变量就是 course。

经过上面的分析，我们可以设计一个包含共用体的结构体，请看下面的代码：

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

#define TOTAL 4  //人员总数

struct{
    char name[20];
    int num;
    char sex;
    char profession;
    union{
        float score;
        char course[20];
    } sc;
} bodys[TOTAL];

int main(){
    int i;
    //输入人员信息
    for(i=0; i<TOTAL; i++){
        printf("Input info: ");
        scanf("%s %d %c %c", bodys[i].name, &(bodys[i].num), &(bodys[i].sex), &(bodys[i].profession));
        if(bodys[i].profession == 's'){  //如果是学生
            scanf("%f", &bodys[i].sc.score);
        }else{  //如果是老师
            scanf("%s", bodys[i].sc.course);
        }
        fflush(stdin);
    }

    //输出人员信息
    printf("\nName\t\tNum\tSex\tProfession\tScore / Course\n");
    for(i=0; i<TOTAL; i++){
        if(bodys[i].profession == 's'){  //如果是学生
            printf("%s\t%d\t%c\t%c\t\t%f\n", bodys[i].name, bodys[i].num, bodys[i].sex,
bodys[i].profession, bodys[i].sc.score);
```

```
        }else{  //如果是老师
            printf("%s\t%d\t%c\t%c\t\t%s\n", bodys[i].name, bodys[i].num, bodys[i].sex,
    bodys[i].profession, bodys[i].sc.course);
        }
    }
    return 0;
}
```

运行结果：

```
Input info: HanXiaoXiao 501 f s 89.5↙
Input info: YanWeiMin 1011 m t math↙
Input info: LiuZhenTao 109 f t English↙
Input info: ZhaoFeiYan 982 m s 95.0↙

Name            Num     Sex     Profession      Score / Course
HanXiaoXiao     501     f       s               89.500000
YanWeiMin       1011    m       t               math
LiuZhenTao      109     f       t               English
ZhaoFeiYan      982     m       s               95.000000
```

10.6 大端小端以及判别方式

10.7 C 语言位域（位段）

有些数据在存储时并不需要占用一个完整的字节，只需要占用一个或几个二进制位即可。例如开关只有通电和断电两种状态，用 0 和 1 表示足以，也就是用一个二进位。正是基于这种考虑，C 语言又提供了一种叫做位域的数据结构。

在结构体定义时，我们可以指定某个成员变量所占用的二进制位数（Bit），这就是位域。请看下面的例子：

```
struct bs{
    unsigned m;
    unsigned n: 4;
    unsigned char ch: 6;
};
```

:后面的数字用来限定成员变量占用的位数。成员 m 没有限制，根据数据类型即可推算出它占用 4 个字节（Byte）

的内存。成员 n、ch 被:后面的数字限制，不能再根据数据类型计算长度，它们分别占用 4、6 位（Bit）的内存。

n、ch 的取值范围非常有限，数据稍微大些就会发生溢出，请看下面的例子：

```
#include <stdio.h>

int main(){
    struct bs{
        unsigned m;
        unsigned n: 4;
        unsigned char ch: 6;
    } a = { 0xad, 0xE, '$'};
    //第一次输出
    printf("%#x, %#x, %c\n", a.m, a.n, a.ch);
    //更改值后再次输出
    a.m = 0xb8901c;
    a.n = 0x2d;
    a.ch = 'z';
    printf("%#x, %#x, %c\n", a.m, a.n, a.ch);

    return 0;
}
```

运行结果：
0xad, 0xe, $
0xb8901c, 0xd, :

对于 n 和 ch，第一次输出的数据是完整的，第二次输出的数据是残缺的。

第一次输出时，n、ch 的值分别是 0xE、0x24（'$' 对应的 ASCII 码为 0x24），换算成二进制是 1110、10 0100，都没有超出限定的位数，能够正常输出。

第二次输出时，n、ch 的值变为 0x2d、0x7a（'z' 对应的 ASCII 码为 0x7a），换算成二进制分别是 10 1101、111 1010，都超出了限定的位数。超出部分被直接截去，剩下 1101、11 1010，换算成十六进制为 0xd、0x3a（0x3a 对应的字符是 :）。

C 语言标准规定，位域的宽度不能超过它所依附的数据类型的长度。通俗地讲，成员变量都是有类型的，这个类型限制了成员变量的最大长度，:后面的数字不能超过这个长度。

例如上面的 bs，n 的类型是 unsigned int，长度为 4 个字节，共计 32 位，那么 n 后面的数字就不能超过 32；ch 的类型是 unsigned char，长度为 1 个字节，共计 8 位，那么 ch 后面的数字就不能超过 8。

我们可以这样认为，位域技术就是在成员变量所占用的内存中选出一部分位宽来存储数据。

C 语言标准还规定，只有有限的几种数据类型可以用于位域。在 ANSI C 中，这几种数据类型是 int、signed int 和 unsigned int（int 默认就是 signed int）；到了 C99，_Bool 也被支持了。

> 关于 C 语言标准以及 ANSI C 和 C99 的区别，我们已在付费教程《C 语言的三套标准：C89、C99 和 C11》中进行了讲解。

但编译器在具体实现时都进行了扩展，额外支持了 char、signed char、unsigned char 以及 enum 类型，所以上面的代码虽然不符合 C 语言标准，但它依然能够被编译器支持。

位域的存储

C 语言标准并没有规定位域的具体存储方式，不同的编译器有不同的实现，但它们都尽量压缩存储空间。

位域的具体存储规则如下：
1) 当相邻成员的类型相同时，如果它们的位宽之和小于类型的 sizeof 大小，那么后面的成员紧邻前一个成员存储，直到不能容纳为止；如果它们的位宽之和大于类型的 sizeof 大小，那么后面的成员将从新的存储单元开始，其偏移量为类型大小的整数倍。

以下面的位域 bs 为例：

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      struct bs{
5.          unsigned m: 6;
6.          unsigned n: 12;
7.          unsigned p: 4;
8.      };
9.      printf("%d\n", sizeof(struct bs));
10.
11.     return 0;
12. }
```

运行结果：
4

m、n、p 的类型都是 unsigned int，sizeof 的结果为 4 个字节（Byte），也即 32 个位（Bit）。m、n、p 的位宽之和为 6+12+4 = 22，小于 32，所以它们会挨着存储，中间没有缝隙。

> sizeof(struct bs) 的大小之所以为 4，而不是 3，是因为要将内存对齐到 4 个字节，以便提高存取效率，这将在《C 语言内存精讲》专题的《C 语言内存对齐，提高寻址效率》一节中详细讲解。

如果将成员 m 的位宽改为 22，那么输出结果将会是 8，因为 22+12 = 34，大于 32，n 会从新的位置开始存储，相对 m 的偏移量是 sizeof(unsigned int)，也即 4 个字节。

如果再将成员 p 的位宽也改为 22，那么输出结果将会是 12，三个成员都不会挨着存储。

2) 当相邻成员的类型不同时，不同的编译器有不同的实现方案，GCC 会压缩存储，而 VC/VS 不会。

请看下面的位域 bs：

```
#include <stdio.h>

int main(){
    struct bs{
        unsigned m: 12;
        unsigned char ch: 4;
        unsigned p: 4;
    };
    printf("%d\n", sizeof(struct bs));

    return 0;
}
```

在 GCC 下的运行结果为 4，三个成员挨着存储；在 VC/VS 下的运行结果为 12，三个成员按照各自的类型存储（与不指定位宽时的存储方式相同）。

> m 、ch、p 的长度分别是 4、1、4 个字节，共计占用 9 个字节内存，为什么在 VC/VS 下的输出结果却是 12 呢？这个疑问将在《C 语言和内存》专题的《C 语言内存对齐，提高寻址效率》一节中为您解开。

3) 如果成员之间穿插着非位域成员，那么不会进行压缩。例如对于下面的 bs：

```
struct bs{
    unsigned m: 12;
    unsigned ch;
    unsigned p: 4;
};
```

在各个编译器下 sizeof 的结果都是 12。

通过上面的分析，我们发现位域成员往往不占用完整的字节，有时候也不处于字节的开头位置，因此使用&获取位域成员的地址是没有意义的，C 语言也禁止这样做。地址是字节（Byte）的编号，而不是位（Bit）的编号。

无名位域

位域成员可以没有名称，只给出数据类型和位宽，如下所示：

```
struct bs{
    int m: 12;
    int  : 20;  //该位域成员不能使用
    int n: 4;
};
```

无名位域一般用来作填充或者调整成员位置。因为没有名称，无名位域不能使用。

上面的例子中，如果没有位宽为 20 的无名成员，m、n 将会挨着存储，sizeof(struct bs) 的结果为 4；有了这 20 位作为填充，m、n 将分开存储，sizeof(struct bs) 的结果为 8。

10.8 C 语言位运算详解

所谓位运算，就是对一个比特（Bit）位进行操作。在《数据在内存中的存储》一节中讲到，比特（Bit）是一个电子元器件，8 个比特构成一个字节（Byte），它已经是粒度最小的可操作单元了。

C 语言提供了六种位运算符：

运算符	&	|	^	~	<<	>>
说明	按位与	按位或	按位异或	取反	左移	右移

按位与运算（&）

一个比特（Bit）位只有 0 和 1 两个取值，只有参与&运算的两个位都为 1 时，结果才为 1，否则为 0。例如 1&1 为 1，0&0 为 0，1&0 也为 0，这和逻辑运算符&&非常类似。

C 语言中不能直接使用二进制，&两边的操作数可以是十进制、八进制、十六进制，它们在内存中最终都是以二进制形式存储，&就是对这些内存中的二进制位进行运算。其他的位运算符也是相同的道理。

例如，9 & 5 可以转换成如下的运算：

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
& 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0001   (1 在内存中的存储)
```

也就是说，按位与运算会对参与运算的两个数的所有二进制位进行&运算，9 & 5 的结果为 1。

又如，-9 & 5 可以转换成如下的运算：

```
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
& 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
```

-9 & 5 的结果是 5。

> 关于正数和负数在内存中的存储形式，我们已在 VIP 教程《整数在内存中是如何存储的，为什么它堪称天才般的设计》中进行了讲解。

再强调一遍，&是根据内存中的二进制位进行运算的，而不是数据的二进制形式；其他位运算符也一样。以-9&5 为例，-9 的在内存中的存储和 -9 的二进制形式截然不同：

```
 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
-0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (-9 的二进制形式，前面多余的 0 可以抹掉)
```

按位与运算通常用来对某些位清 0，或者保留某些位。例如要把 n 的高 16 位清 0，保留低 16 位，可以进行 n

& 0XFFFF 运算（0XFFFF 在内存中的存储形式为 0000 0000 -- 0000 0000 -- 1111 1111 -- 1111 1111）。

【实例】对上面的分析进行检验。

```
#include <stdio.h>

int main(){
    int n = 0X8FA6002D;
    printf("%d, %d, %X\n", 9 & 5, -9 & 5, n & 0XFFFF);
    return 0;
}
```

运行结果：

1, 5, 2D

按位或运算（|）

参与|运算的两个二进制位有一个为 1 时，结果就为 1，两个都为 0 时结果才为 0。例如 1|1 为 1，0|0 为 0，1|0 为 1，这和逻辑运算中的||非常类似。

例如，9 | 5 可以转换成如下的运算：

```
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   （9 在内存中的存储）
| 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   （5 在内存中的存储）
-------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1101   （13 在内存中的存储）
```

9 | 5 的结果为 13。

又如，-9 | 5 可以转换成如下的运算：

```
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   （-9 在内存中的存储）
| 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   （5 在内存中的存储）
-------------------------------------------------------------------------------
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   （-9 在内存中的存储）
```

-9 | 5 的结果是 -9。

按位或运算可以用来将某些位置 1，或者保留某些位。例如要把 n 的高 16 位置 1，保留低 16 位，可以进行 n | 0XFFFF0000 运算（0XFFFF0000 在内存中的存储形式为 1111 1111 -- 1111 1111 -- 0000 0000 -- 0000 0000）。

【实例】对上面的分析进行校验。

```
#include <stdio.h>

int main(){
    int n = 0X2D;
    printf("%d, %d, %X\n", 9 | 5, -9 | 5, n | 0XFFFF0000);
```

```
6.      return 0;
7.  }
```

运行结果：
13, -9, FFFF002D

按位异或运算（^）

参与^运算两个二进制位不同时，结果为 1，相同时结果为 0。例如 0^1 为 1，0^0 为 0，1^1 为 0。

例如，9 ^ 5 可以转换成如下的运算：

```
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001   (9 在内存中的存储)
^  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1100   (12 在内存中的存储)
```

9 ^ 5 的结果为 12。

又如，-9 ^ 5 可以转换成如下的运算：

```
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111   (-9 在内存中的存储)
^  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0101   (5 在内存中的存储)
--------------------------------------------------------------------------------
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0010   (-14 在内存中的存储)
```

-9 ^ 5 的结果是 -14。

按位异或运算可以用来将某些二进制位反转。例如要把 n 的高 16 位反转，保留低 16 位，可以进行 n ^ 0XFFFF0000 运算（0XFFFF0000 在内存中的存储形式为 1111 1111 -- 1111 1111 -- 0000 0000 -- 0000 0000）。

【实例】对上面的分析进行校验。

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      unsigned n = 0X0A07002D;
5.      printf("%d, %d, %X\n", 9 ^ 5, -9 ^ 5, n ^ 0XFFFF0000);
6.      return 0;
7.  }
```

运行结果：
12, -14, F5F8002D

取反运算（~）

取反运算符~为单目运算符，右结合性，作用是对参与运算的二进制位取反。例如~1 为 0，~0 为 1，这和逻辑运算中的!非常类似。。

例如，~9 可以转换为如下的运算：

```
~ 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001    (9 在内存中的存储)
--------------------------------------------------------------------------------
  1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0110    (-10 在内存中的存储)
```

所以~9 的结果为 -10。

例如，~-9 可以转换为如下的运算：

```
~ 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111    (-9 在内存中的存储)
--------------------------------------------------------------------------------
  0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1000    (9 在内存中的存储)
```

所以~-9 的结果为 8。

【实例】对上面的分析进行校验。

```
1.  #include <stdio.h>
2.
3.  int main(){
4.      printf("%d, %d\n", ~9, ~-9 );
5.      return 0;
6.  }
```

运行结果：

```
-10, 8
```

左移运算（<<）

左移运算符<<用来把操作数的各个二进制位全部左移若干位，高位丢弃，低位补 0。

例如，9<<3 可以转换为如下的运算：

```
<< 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001    (9 在内存中的存储)
--------------------------------------------------------------------------------
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0100 1000    (72 在内存中的存储)
```

所以 9<<3 的结果为 72。

又如，(-9)<<3 可以转换为如下的运算：

```
<< 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111    (-9 在内存中的存储)
--------------------------------------------------------------------------------
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1011 1000    (-72 在内存中的存储)
```

所以(-9)<<3 的结果为 -72

如果数据较小，被丢弃的高位不包含 1，那么左移 n 位相当于乘以 2 的 n 次方。

【实例】对上面的结果进行校验。

```
#include <stdio.h>

int main(){
    printf("%d, %d\n", 9<<3, (-9)<<3 );
    return 0;
}
```

运行结果：
72, -72

右移运算（>>）

右移运算符>>用来把操作数的各个二进制位全部右移若干位，低位丢弃，高位补 0 或 1。如果数据的最高位是 0，那么就补 0；如果最高位是 1，那么就补 1。

例如，9>>3 可以转换为如下的运算：

```
>> 0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 1001    (9 在内存中的存储)
--------------------------------------------------------------------------------
   0000 0000 -- 0000 0000 -- 0000 0000 -- 0000 0001    (1 在内存中的存储)
```

所以 9>>3 的结果为 1。

又如，(-9)>>3 可以转换为如下的运算：

```
>> 1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 0111    (-9 在内存中的存储)
--------------------------------------------------------------------------------
   1111 1111 -- 1111 1111 -- 1111 1111 -- 1111 1110    (-2 在内存中的存储)
```

所以(-9)>>3 的结果为 -2

如果被丢弃的低位不包含 1，那么右移 n 位相当于除以 2 的 n 次方（但被移除的位中经常会包含 1）。

【实例】对上面的结果进行校验。

```
#include <stdio.h>

int main(){
    printf("%d, %d\n", 9>>3, (-9)>>3 );
    return 0;
}
```

运行结果：
1, -2

10.9 使用位运算对数据或文件内容进行加密

第 11 章 C 语言重要知识点补充

本章我们来补充一下前面没有讲到的 C 语言知识。

这些知识虽然很重要，但是比较零散，不能单独构成一章，也没法划分到其它章节，所以才把这些重要知识汇总在本章集中讲解。

本章目录：

1. C 语言 typedef 的用法
2. C 语言 const 的用法
3. C 语言随机数：rand()和 srand()函数

11.1 C 语言 typedef 的用法

C 语言允许为一个数据类型起一个新的别名，就像给人起“绰号”一样。

起别名的目的不是为了提高程序运行效率，而是为了编码方便。例如有一个结构体的名字是 stu，要想定义一个结构体变量就得这样写：

```
struct stu stu1;
```

struct 看起来就是多余的，但不写又会报错。如果为 struct stu 起了一个别名 STU，书写起来就简单了：

```
STU stu1;
```

这种写法更加简练，意义也非常明确，不管是在标准头文件中还是以后的编程实践中，都会大量使用这种别名。

使用关键字 **typedef** 可以为类型起一个新的别名。typedef 的用法一般为：

```
typedef  oldName  newName;
```

oldName 是类型原来的名字，newName 是类型新的名字。例如：

```
1.  typedef int INTEGER;
2.  INTEGER a, b;
3.  a = 1;
4.  b = 2;
```

INTEGER a, b;等效于 int a, b;。

typedef 还可以给数组、指针、结构体等类型定义别名。先来看一个给数组类型定义别名的例子：

```
typedef char ARRAY20[20];
```

表示 ARRAY20 是类型 char [20]的别名。它是一个长度为 20 的数组类型。接着可以用 ARRAY20 定义数组：

```
ARRAY20 a1, a2, s1, s2;
```

它等价于：

```
char a1[20], a2[20], s1[20], s2[20];
```

注意，数组也是有类型的。例如 char a1[20];定义了一个数组 a1，它的类型就是 char [20]，这一点已在 VIP 教程《数组和指针绝不等价，数组是另外一种类型》中讲解过。

又如，为结构体类型定义别名：

```
1.  typedef struct stu{
2.      char name[20];
3.      int age;
4.      char sex;
5.  } STU;
```

STU 是 struct stu 的别名，可以用 STU 定义结构体变量：

```
STU body1,body2;
```

它等价于：

```
struct stu body1, body2;
```

再如，为指针类型定义别名：

```
typedef int (*PTR_TO_ARR)[4];
```

表示 PTR_TO_ARR 是类型 int * [4]的别名，它是一个二维数组指针类型。接着可以使用 PTR_TO_ARR 定义二维数组指针：

```
PTR_TO_ARR p1, p2;
```

按照类似的写法，还可以为函数指针类型定义别名：

```
typedef int (*PTR_TO_FUNC)(int, int);
PTR_TO_FUNC pfunc;
```

【示例】为指针定义别名。

```
1.  #include <stdio.h>
2.
3.  typedef char (*PTR_TO_ARR)[30];
4.  typedef int (*PTR_TO_FUNC)(int, int);
5.
6.  int max(int a, int b){
7.      return a>b ? a : b;
```

```
8.  }
9.
10. char str[3][30] = {
11.     "http://c.biancheng.net",
12.     "C语言中文网",
13.     "C-Language"
14. };
15.
16. int main(){
17.     PTR_TO_ARR parr = str;
18.     PTR_TO_FUNC pfunc = max;
19.     int i;
20.
21.     printf("max: %d\n", (*pfunc)(10, 20));
22.     for(i=0; i<3; i++){
23.         printf("str[%d]: %s\n", i, *(parr+i));
24.     }
25.
26.     return 0;
27. }
```

运行结果：

max: 20

str[0]: http://c.biancheng.net

str[1]: C 语言中文网

str[2]: C-Language

需要强调的是，typedef 是赋予现有类型一个新的名字，而不是创建新的类型。为了“见名知意”，请尽量使用含义明确的标识符，并且尽量大写。

typedef 和 #define 的区别

typedef 在表现上有时候类似于 #define，但它和宏替换之间存在一个关键性的区别。正确思考这个问题的方法就是把 typedef 看成一种彻底的“封装”类型，声明之后不能再往里面增加别的东西。

1) 可以使用其他类型说明符对宏类型名进行扩展，但对 typedef 所定义的类型名却不能这样做。如下所示：

```
#define INTERGE int
unsigned INTERGE n;   //没问题

typedef int INTERGE;
unsigned INTERGE n;   //错误，不能在 INTERGE 前面添加 unsigned
```

2) 在连续定义几个变量的时候，typedef 能够保证定义的所有变量均为同一类型，而 #define 则无法保证。例如：

```
#define PTR_INT int *
PTR_INT p1, p2;
```

经过宏替换以后，第二行变为：

```
int *p1, p2;
```

这使得 p1、p2 成为不同的类型：p1 是指向 int 类型的指针，p2 是 int 类型。

相反，在下面的代码中：

```
typedef int * PTR_INT
PTR_INT p1, p2;
```

p1、p2 类型相同，它们都是指向 int 类型的指针。

11.2 C 语言 const 的用法

有时候我们希望定义这样一种变量，它的值不能被改变，在整个作用域中都保持固定。例如，用一个变量来表示班级的最大人数，或者表示缓冲区的大小。为了满足这一要求，可以使用 const 关键字对变量加以限定：

```
const int MaxNum = 100;  //班级的最大人数
```

这样 MaxNum 的值就不能被修改了，任何对 MaxNum 赋值的行为都将引发错误：

```
MaxNum = 90;  //错误，试图向 const 变量写入数据
```

我们经常将 const 变量称为常量（Constant）。创建常量的格式通常为：

```
const type name = value;
```

const 和 type 都是用来修饰变量的，它们的位置可以互换，也就是将 type 放在 const 前面：

```
type const name = value;
```

但我们通常采用第一种方式，不采用第二种方式。另外建议将常量名的首字母大写，以提醒程序员这是个常量。

由于常量一旦被创建后其值就不能再改变，所以常量必须在定义的同时赋值（初始化），后面的任何赋值行为都将引发错误。一如既往，初始化常量可以使用任意形式的表达式，如下所示：

```
1.  #include <stdio.h>
2.
3.  int getNum(){
4.      return 100;
5.  }
6.
7.  int main(){
8.      int n = 90;
9.      const int MaxNum1 = getNum();  //运行时初始化
10.     const int MaxNum2 = n;  //运行时初始化
11.     const int MaxNum3 = 80;  //编译时初始化
12.     printf("%d, %d, %d\n", MaxNum1, MaxNum2, MaxNum3);
```

```
13.
14.     return 0;
15. }
```

运行结果：
100, 90, 80

const 和指针

const 也可以和指针变量一起使用，这样可以限制指针变量本身，也可以限制指针指向的数据。const 和指针一起使用会有几种不同的顺序，如下所示：

```
1.  const int *p1;
2.  int const *p2;
3.  int * const p3;
```

在最后一种情况下，指针是只读的，也就是 p3 本身的值不能被修改；在前面两种情况下，指针所指向的数据是只读的，也就是 p1、p2 本身的值可以修改（指向不同的数据），但它们指向的数据不能被修改。

当然，指针本身和它指向的数据都有可能是只读的，下面的两种写法能够做到这一点：

```
1.  const int * const p4;
2.  int const * const p5;
```

const 和指针结合的写法多少有点让初学者摸不着头脑，大家可以这样来记忆：const 离变量名近就是用来修饰指针变量的，离变量名远就是用来修饰指针指向的数据，如果近的和远的都有，那么就同时修饰指针变量以及它指向的数据。

const 和函数形参

在 C 语言中，单独定义 const 变量没有明显的优势，完全可以使用#define 命令代替。const 通常用在函数形参中，如果形参是一个指针，为了防止在函数内部修改指针指向的数据，就可以用 const 来限制。

在 C 语言标准库中，有很多函数的形参都被 const 限制了，下面是部分函数的原型：

```
1.  size_t strlen ( const char * str );
2.  int strcmp ( const char * str1, const char * str2 );
3.  char * strcat ( char * destination, const char * source );
4.  char * strcpy ( char * destination, const char * source );
5.  int system (const char* command);
6.  int puts ( const char * str );
7.  int printf ( const char * format, ... );
```

我们自己在定义函数时也可以使用 const 对形参加以限制，例如查找字符串中某个字符出现的次数：

```
1.  #include <stdio.h>
2.
3.  size_t strnchr(const char *str, char ch){
4.      int i, n = 0, len = strlen(str);
5.
6.      for(i=0; i<len; i++){
```

```
7.          if(str[i] == ch){
8.              n++;
9.          }
10.     }
11.
12.     return n;
13. }
14.
15. int main(){
16.     char *str = "http://c.biancheng.net";
17.     char ch = 't';
18.     int n = strnchr(str, ch);
19.     printf("%d\n", n);
20.     return 0;
21. }
```

运行结果：
3

根据 strnchr() 的功能可以推断，函数内部要对字符串 str 进行遍历，不应该有修改的动作，用 const 加以限制，不但可以防止由于程序员误操作引起的字符串修改，还可以给用户一个提示，函数不会修改你提供的字符串，请你放心。

const 和非 const 类型转换

当一个指针变量 str1 被 const 限制时，并且类似 const char *str1 这种形式，说明指针指向的数据不能被修改；如果将 str1 赋值给另外一个未被 const 修饰的指针变量 str2，就有可能发生危险。因为通过 str1 不能修改数据，而赋值后通过 str2 能够修改数据了，意义发生了转变，所以编译器不提倡这种行为，会给出错误或警告。

也就是说，const char *和 char *是不同的类型，不能将 const char *类型的数据赋值给 char *类型的变量。但反过来是可以的，编译器允许将 char *类型的数据赋值给 const char *类型的变量。

这种限制很容易理解，char *指向的数据有读取和写入权限，而 const char *指向的数据只有读取权限，降低数据的权限不会带来任何问题，但提升数据的权限就有可能发生危险。

C 语言标准库中很多函数的参数都被 const 限制了，但我们在以前的编码过程中并没有注意这个问题，经常将非 const 类型的数据传递给 const 类型的形参，这样做从未引发任何副作用，原因就是上面讲到的，将非 const 类型转换为 const 类型是允许的。

下面是一个将 const 类型赋值给非 const 类型的例子：

```
1.  #include <stdio.h>
2.
3.  void func(char *str){ }
4.
5.  int main(){
```

```
6.      const char *str1 = "c.biancheng.net";
7.      char *str2 = str1;
8.      func(str1);
9.      return 0;
10. }
```

第 7、8 行代码分别通过赋值、传参（传参的本质也是赋值）将 const 类型的数据交给了非 const 类型的变量，编译器不会容忍这种行为，会给出警告，甚至直接报错。

11.3 C 语言随机数：rand()和 srand()函数

在实际编程中，我们经常需要生成随机数，例如，贪吃蛇游戏中在随机的位置出现食物，扑克牌游戏中随机发牌。

在 C 语言中，我们一般使用 <stdlib.h> 头文件中的 rand() 函数来生成随机数，它的用法为：

```
int rand (void);
```

void 表示不需要传递参数。

> C 语言中还有一个 random() 函数可以获取随机数，但是 random() 不是标准函数，不能在 VC/VS 等编译器通过，所以比较少用。

rand() 会随机生成一个位于 0 ~ RAND_MAX 之间的整数。

RAND_MAX 是 <stdlib.h> 头文件中的一个宏，它用来指明 rand() 所能返回的随机数的最大值。C 语言标准并没有规定 RAND_MAX 的具体数值，只是规定它的值至少为 32767。在实际编程中，我们也不需要知道 RAND_MAX 的具体值，把它当做一个很大的数来对待即可。

下面是一个随机数生成的实例：

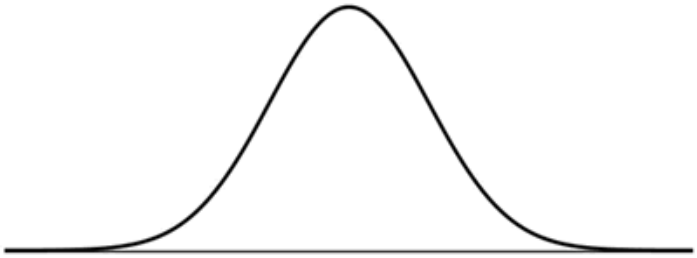
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main(){
4.      int a = rand();
5.      printf("%d\n",a);
6.      return 0;
7.  }
```

运行结果举例：
193

随机数的本质

多次运行上面的代码，你会发现每次产生的随机数都一样，这是怎么回事呢？为什么随机数并不随机呢？

实际上，rand() 函数产生的随机数是伪随机数，是根据一个数值按照某个公式推算出来的，这个数值我们称之为“种子”。种子和随机数之间的关系是一种正态分布，如下图所示：



种子在每次启动计算机时是随机的，但是一旦计算机启动以后它就不再变化了；也就是说，每次启动计算机以后，种子就是定值了，所以根据公式推算出来的结果（也就是生成的随机数）就是固定的。

重新播种

我们可以通过 srand() 函数来重新“播种”，这样种子就会发生改变。srand() 的用法为：

```
void srand (unsigned int seed);
```

它需要一个 unsigned int 类型的参数。在实际开发中，我们可以用时间作为参数，只要每次播种的时间不同，那么生成的种子就不同，最终的随机数也就不同。

使用 <time.h> 头文件中的 time() 函数即可得到当前的时间（精确到秒），就像下面这样：

```
srand((unsigned)time(NULL));
```

有兴趣的读者请猛击这里自行研究 time() 函数的用法，本节我们不再过多讲解。

对上面的代码进行修改，生成随机数之前先进行播种：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int a;
    srand((unsigned)time(NULL));
    a = rand();
    printf("%d\n", a);
    return 0;
}
```

多次运行程序，会发现每次生成的随机数都不一样了。但是，这些随机数会有逐渐增大或者逐渐减小的趋势，这是因为我们以时间为种子，时间是逐渐增大的，结合上面的正态分布图，很容易推断出随机数也会逐渐增大或者减小。

生成一定范围内的随机数

在实际开发中，我们往往需要一定范围内的随机数，过大或者过小都不符合要求，那么，如何产生一定范围的随机数呢？我们可以利用取模的方法：

```
int a = rand() % 10;      //产生 0~9 的随机数，注意 10 会被整除
```

如果要规定上下限：

```
int a = rand() % 51 + 13;      //产生 13~63 的随机数
```

分析：取模即取余，rand()%51+13 我们可以看成两部分：rand()%51 是产生 0~50 的随机数，后面+13 保证 a 最小只能是 13，最大就是 50+13=63。

最后给出产生 13~63 范围内随机数的完整代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    int a;
    srand((unsigned)time(NULL));
    a = rand() % 51 + 13;
    printf("%d\n",a);
    return 0;
}
```

连续生成随机数

有时候我们需要一组随机数（多个随机数），该怎么生成呢？很容易想到的一种解决方案是使用循环，每次循环都重新播种，请看下面的代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int a, i;
    //使用for循环生成10个随机数
    for (i = 0; i < 10; i++) {
        srand((unsigned)time(NULL));
        a = rand();
        printf("%d ", a);
    }

    return 0;
}
```

运行结果举例：
8 8 8 8 8 8 8 8 8 8

运行结果非常奇怪，每次循环我们都重新播种了呀，为什么生成的随机数都一样呢？

这是因为，for 循环运行速度非常快，在一秒之内就运行完成了，而 time() 函数得到的时间只能精确到秒，所以每次循环得到的时间都是一样的，这样一来，种子也就是一样的，随机数也就一样了。

那么，该如何解决呢？难道就没有办法连续生成随机数了吗？当然有，我们将在《C 语言连续生成多个随机数》一

节中给出一种巧妙的解决方案。

第 12 章 C 语言文件操作

C 语言具有操作文件的能力，比如打开文件、读取和追加数据、插入和删除数据、关闭文件、删除文件等。

与其他编程语言相比，C 语言文件操作的接口相当简单和易学。在 C 语言中，为了统一对各种硬件的操作，简化接口，不同的硬件设备也都被看成一个文件。对这些文件的操作，等同于对磁盘上普通文件的操作。

本章目录：

1. C 语言中的文件是什么？
2. C 语言打开文件：fopen()函数的用法
3. 文本文件和二进制文件到底有什么区别？
4. 以字符形式读写文件
5. 以字符串的形式读写文件
6. 以数据块的形式读写文件
7. 格式化读写文件
8. 随机读写文件
9. C 语言实现文件复制功能(包括文本文件和二进制文件)
10. C 语言 FILE 结构体以及缓冲区深入探讨
11. C 语言获取文件大小（长度）
12. C 语言插入、删除、更改文件内容

蓝色链接是初级教程，能够让你快速入门；红色链接是高级教程，能够让你认识到 C 语言的本质。

12.1 C 语言中的文件是什么？

我们对文件的概念已经非常熟悉了，比如常见的 Word 文档、txt 文件、源文件等。文件是数据源的一种，最主要的作用是保存数据。

在操作系统中，为了统一对各种硬件的操作，简化接口，不同的硬件设备也都被看成一个文件。对这些文件的操作，等同于对磁盘上普通文件的操作。例如：

- 通常把显示器称为标准输出文件，printf 就是向这个文件输出数据；
- 通常把键盘称为标准输入文件，scanf 就是从这个文件读取数据。

常见硬件设备所对应的文件

文件	硬件设备
stdin	标准输入文件，一般指键盘；scanf()、getchar() 等函数默认从 stdin 获取输入。
stdout	标准输出文件，一般指显示器；printf()、putchar() 等函数默认向 stdout 输出数据。

stderr	标准错误文件，一般指显示器；perror() 等函数默认向 stderr 输出数据（后续会讲到）。
stdprn	标准打印文件，一般指打印机。

> 我们不去探讨硬件设备是如何被映射成文件的，大家只需要记住，在 C 语言中硬件设备可以看成文件，有些输入输出函数不需要你指明到底读写哪个文件，系统已经为它们设置了默认的文件，当然你也可以更改，例如让 printf 向磁盘上的文件输出数据。

操作文件的正确流程为：打开文件 --> 读写文件 --> 关闭文件。文件在进行读写操作之前要先打开，使用完毕要关闭。

所谓打开文件，就是获取文件的有关信息，例如文件名、文件状态、当前读写位置等，这些信息会被保存到一个 FILE 类型的结构体变量中。关闭文件就是断开与文件之间的联系，释放结构体变量，同时禁止再对该文件进行操作。

在 C 语言中，文件有多种读写方式，可以一个字符一个字符地读取，也可以读取一整行，还可以读取若干个字节。文件的读写位置也非常灵活，可以从文件开头读取，也可以从中间位置读取。

文件流

在《载入内存，让程序运行起来》一文中提到，所有的文件（保存在磁盘）都要载入内存才能处理，所有的数据必须写入文件（磁盘）才不会丢失。数据在文件和内存之间传递的过程叫做文件流，类似水从一个地方流动到另一个地方。数据从文件复制到内存的过程叫做输入流，从内存保存到文件的过程叫做输出流。

文件是数据源的一种，除了文件，还有数据库、网络、键盘等；数据传递到内存也就是保存到 C 语言的变量（例如整数、字符串、数组、缓冲区等）。我们把数据在数据源和程序（内存）之间传递的过程叫做数据流(Data Stream)。相应的，数据从数据源到程序（内存）的过程叫做输入流(Input Stream)，从程序（内存）到数据源的过程叫做输出流(Output Stream)。

输入输出（Input output，IO）是指程序（内存）与外部设备（键盘、显示器、磁盘、其他计算机等）进行交互的操作。几乎所有的程序都有输入与输出操作，如从键盘上读取数据，从本地或网络上的文件读取数据或写入数据等。通过输入和输出操作可以从外界接收信息，或者是把信息传递给外界。

我们可以说，打开文件就是打开了一个流。

12.2 C 语言打开文件：fopen()函数的用法

在 C 语言中，操作文件之前必须先打开文件；所谓“打开文件”，就是让程序和文件建立连接的过程。

打开文件之后，程序可以得到文件的相关信息，例如大小、类型、权限、创建者、更新时间等。在后续读写文件的过程中，程序还可以记录当前读写到了哪个位置，下次可以在此基础上继续操作。

> 标准输入文件 stdin（表示键盘）、标准输出文件 stdout（表示显示器）、标准错误文件 stderr（表示显示器）是由系统打开的，可直接使用。

使用 <stdio.h> 头文件中的 fopen() 函数即可打开文件，它的用法为：

```
FILE *fopen(char *filename, char *mode);
```

filename 为文件名（包括文件路径），mode 为打开方式，它们都是字符串。

fopen() 函数的返回值

fopen() 会获取文件信息，包括文件名、文件状态、当前读写位置等，并将这些信息保存到一个 FILE 类型的结构体变量中，然后将该变量的地址返回。

FILE 是 <stdio.h> 头文件中的一个结构体，它专门用来保存文件信息。我们不用关心 FILE 的具体结构，只需要知道它的用法就行。

如果希望接收 fopen() 的返回值，就需要定义一个 FILE 类型的指针。例如：

```
FILE *fp = fopen("demo.txt", "r");
```

表示以“只读”方式打开当前目录下的 demo.txt 文件，并使 fp 指向该文件，这样就可以通过 fp 来操作 demo.txt 了。fp 通常被称为文件指针。

再来看一个例子：

```
FILE *fp = fopen("D:\\demo.txt","rb+");
```

表示以二进制方式打开 D 盘下的 demo.txt 文件，允许读和写。

判断文件是否打开成功

打开文件出错时，fopen() 将返回一个空指针，也就是 NULL，我们可以利用这一点来判断文件是否打开成功，请看下面的代码：

```
1.  FILE *fp;
2.  if( (fp=fopen("D:\\demo.txt","rb") == NULL ){
3.      printf("Fail to open file!\n");
4.      exit(0);  //退出程序（结束程序）
5.  }
```

我们通过判断 fopen() 的返回值是否和 NULL 相等来判断是否打开失败：如果 fopen() 的返回值为 NULL，那么 fp 的值也为 NULL，此时 if 的判断条件成立，表示文件打开失败。

以上代码是文件操作的规范写法，读者在打开文件时一定要判断文件是否打开成功，因为一旦打开失败，后续操作就都没法进行了，往往以“结束程序”告终。

fopen() 函数的打开方式

不同的操作需要不同的文件权限。例如，只想读取文件中的数据的话，“只读”权限就够了；既想读取又想写入数据的话，“读写”权限就是必须的了。

另外，文件也有不同的类型，按照数据的存储方式可以分为二进制文件和文本文件，它们的操作细节是不同的。

在调用 fopen() 函数时，这些信息都必须提供，称为“文件打开方式”。最基本的文件打开方式有以下几种：

控制读写权限的字符串（必须指明）	
打开方式	**说明**
"r"	以“只读”方式打开文件。只允许读取，不允许写入。文件必须存在，否则打开失败。
"w"	以“写入”方式打开文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么清空文件内容（相当于删除原文件，再创建一个新文件）。
"a"	以“追加”方式打开文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么将写入的数据追加到文件的末尾（文件原有的内容保留）。
"r+"	以“读写”方式打开文件。既可以读取也可以写入，也就是随意更新文件。文件必须存在，否则打开失败。
"w+"	以“写入/更新”方式打开文件，相当于 w 和 r+叠加的效果。既可以读取也可以写入，也就是随意更新文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么清空文件内容（相当于删除原文件，再创建一个新文件）。
"a+"	以“追加/更新”方式打开文件，相当于 a 和 r+叠加的效果。既可以读取也可以写入，也就是随意更新文件。如果文件不存在，那么创建一个新文件；如果文件存在，那么将写入的数据追加到文件的末尾（文件原有的内容保留）。
控制读写方式的字符串（可以不写）	
打开方式	**说明**
"t"	文本文件。如果不写，默认为"t"。
"b"	二进制文件。

调用 fopen() 函数时必须指明读写权限，但是可以不指明读写方式（此时默认为"t"）。

读写权限和读写方式可以组合使用，但是必须将读写方式放在读写权限的中间或者尾部（换句话说，不能将读写方式放在读写权限的开头）。例如：
- 将读写方式放在读写权限的末尾："rb"、"wt"、"ab"、"r+b"、"w+t"、"a+t"
- 将读写方式放在读写权限的中间："rb+"、"wt+"、"ab+"

整体来说，文件打开方式由 r、w、a、t、b、+ 六个字符拼成，各字符的含义是：
- r(read)：读
- w(write)：写
- a(append)：追加
- t(text)：文本文件
- b(banary)：二进制文件
- +：读和写

关闭文件

文件一旦使用完毕，应该用 fclose() 函数把文件关闭，以释放相关资源，避免数据丢失。fclose() 的用法为：

```
int fclose(FILE *fp);
```

fp 为文件指针。例如：

```
fclose(fp);
```

文件正常关闭时，fclose() 的返回值为 0，如果返回非零值则表示有错误发生。

实例演示

最后，我们通过一段完整的代码来演示 fopen 函数的用法，这个例子会一行一行地读取文本文件的所有内容：

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

#define N 100

int main() {
    FILE *fp;
    char str[N + 1];

    //判断文件是否打开失败
    if ( (fp = fopen("d:\\demo.txt", "rt")) == NULL ) {
        puts("Fail to open file!");
        exit(0);
    }

    //循环读取文件的每一行数据
    while( fgets(str, N, fp) != NULL ) {
        printf("%s", str);
    }

    //操作结束后关闭文件
    fclose(fp);
    return 0;
}
```

读者只需要关心文件打开部分的代码，暂时不用关心文件读取部分的代码，后续我们会逐一讲解。

12.3 文本文件和二进制文件到底有什么区别？

12.4 以字符形式读写文件

在 C 语言中，读写文件比较灵活，既可以每次读写一个字符，也可以读写一个字符串，甚至是任意字节的数据（数据块）。本节介绍以字符形式读写文件。

以字符形式读写文件时，每次可以从文件中读取一个字符，或者向文件中写入一个字符。主要使用两个函数，分别是 fgetc() 和 fputc()。

字符读取函数 fgetc

fgetc 是 file get char 的缩写，意思是从指定的文件中读取一个字符。fgetc() 的用法为：

```
int fgetc (FILE *fp);
```

fp 为文件指针。fgetc() 读取成功时返回读取到的字符，读取到文件末尾或读取失败时返回 EOF。

EOF 是 end of file 的缩写，表示文件末尾，是在 stdio.h 中定义的宏，它的值是一个负数，往往是 -1。fgetc() 的返回值类型之所以为 int，就是为了容纳这个负数（char 不能是负数）。

> EOF 不绝对是 -1，也可以是其他负数，这要看编译器的实现。

fgetc() 的用法举例：

```
1.  char ch;
2.  FILE *fp = fopen("D:\\demo.txt", "r+");
3.  ch = fgetc(fp);
```

表示从 D:\\demo.txt 文件中读取一个字符，并保存到变量 ch 中。

在文件内部有一个位置指针，用来指向当前读写到的位置，也就是读写到第几个字节。在文件打开时，该指针总是指向文件的第一个字节。使用 fgetc() 函数后，该指针会向后移动一个字节，所以可以连续多次使用 fgetc() 读取多个字符。

注意 :这个文件内部的位置指针与 C 语言中的指针不是一回事。位置指针仅仅是一个标志，表示文件读写到的位置，也就是读写到第几个字节，它不表示地址。文件每读写一次，位置指针就会移动一次，它不需要你在程序中定义和赋值，而是由系统自动设置，对用户是隐藏的。

【示例】在屏幕上显示 D:\\demo.txt 文件的内容。

```
1.  #include<stdio.h>
2.  int main(){
3.      FILE *fp;
4.      char ch;
5.
6.      //如果文件不存在，给出提示并退出
7.      if( (fp=fopen("D:\\demo.txt","rt")) == NULL ){
8.          puts("Fail to open file!");
9.          exit(0);
```

```
10.     }
11.
12.     //每次读取一个字节，直到读取完毕
13.     while( (ch=fgetc(fp)) != EOF ){
14.         putchar(ch);
15.     }
16.     putchar('\n');  //输出换行符
17.
18.     fclose(fp);
19.     return 0;
20. }
```

在 D 盘下创建 demo.txt 文件，输入任意内容并保存，运行程序，就会看到刚才输入的内容全部都显示在屏幕上。

该程序的功能是从文件中逐个读取字符，在屏幕上显示，直到读取完毕。

程序第 13 行是关键，while 循环的条件为(ch=fgetc(fp)) != EOF。fget() 每次从位置指针所在的位置读取一个字符，并保存到变量 ch，位置指针向后移动一个字节。当文件指针移动到文件末尾时，fget() 就无法读取字符了，于是返回 EOF，表示文件读取结束了。

对 EOF 的说明

EOF 本来表示文件末尾，意味着读取结束，但是很多函数在读取出错时也返回 EOF，那么当返回 EOF 时，到底是文件读取完毕了还是读取出错了？我们可以借助 stdio.h 中的两个函数来判断，分别是 feof() 和 ferror()。

feof() 函数用来判断文件内部指针是否指向了文件末尾，它的原型是：

```
int feof ( FILE * fp );
```

当指向文件末尾时返回非零值，否则返回零值。

ferror() 函数用来判断文件操作是否出错，它的原型是：

```
int ferror ( FILE *fp );
```

出错时返回非零值，否则返回零值。

需要说明的是，文件出错是非常少见的情况，上面的示例基本能够保证将文件内的数据读取完毕。如果追求完美，也可以加上判断并给出提示：

```
1.  #include<stdio.h>
2.  int main(){
3.      FILE *fp;
4.      char ch;
5.
6.      //如果文件不存在，给出提示并退出
7.      if( (fp=fopen("D:\\demo.txt","rt")) == NULL ){
8.          puts("Fail to open file!");
9.          exit(0);
10.     }
```

```
11.
12.     //每次读取一个字节，直到读取完毕
13.     while( (ch=fgetc(fp)) != EOF ){
14.         putchar(ch);
15.     }
16.     putchar('\n');  //输出换行符
17.
18.     if(ferror(fp)){
19.         puts("读取出错");
20.     }else{
21.         puts("读取成功");
22.     }
23.
24.     fclose(fp);
25.     return 0;
26. }
```

这样，不管是出错还是正常读取，都能够做到心中有数。

字符写入函数 fputc

fputc 是 file output char 的所以，意思是向指定的文件中写入一个字符。fputc() 的用法为：

```
int fputc ( int ch, FILE *fp );
```

ch 为要写入的字符，fp 为文件指针。fputc() 写入成功时返回写入的字符，失败时返回 EOF，返回值类型为 int 也是为了容纳这个负数。例如：

```
fputc('a', fp);
```

或者：

```
char ch = 'a';
fputc(ch, fp);
```

表示把字符 'a' 写入 fp 所指向的文件中。

两点说明

1) 被写入的文件可以用写、读写、追加方式打开，用写或读写方式打开一个已存在的文件时将清除原有的文件内容，并将写入的字符放在文件开头。如需保留原有文件内容，并把写入的字符放在文件末尾，就必须以追加方式打开文件。不管以何种方式打开，被写入的文件若不存在时则创建该文件。

2) 每写入一个字符，文件内部位置指针向后移动一个字节。

【示例】从键盘输入一行字符，写入文件。

```
1.  #include<stdio.h>
2.  int main(){
3.      FILE *fp;
```

```
    char ch;

    //判断文件是否成功打开
    if( (fp=fopen("D:\\demo.txt","wt+")) == NULL ){
        puts("Fail to open file!");
        exit(0);
    }

    printf("Input a string:\n");
    //每次从键盘读取一个字符并写入文件
    while ( (ch=getchar()) != '\n' ){
        fputc(ch,fp);
    }
    fclose(fp);
    return 0;
}
```

运行程序，输入一行字符并按回车键结束，打开 D 盘下的 demo.txt 文件，就可以看到刚才输入的内容。

程序每次从键盘读取一个字符并写入文件，直到按下回车键，while 条件不成立，结束读取。

12.5 以字符串的形式读写文件

fgetc() 和 fputc() 函数每次只能读写一个字符，速度较慢；实际开发中往往是每次读写一个字符串或者一个数据块，这样能明显提高效率。

读字符串函数 fgets

fgets() 函数用来从指定的文件中读取一个字符串，并保存到字符数组中，它的用法为：

```
char *fgets ( char *str, int n, FILE *fp );
```

str 为字符数组，n 为要读取的字符数目，fp 为文件指针。

返回值：读取成功时返回字符数组首地址，也即 str；读取失败时返回 NULL；如果开始读取时文件内部指针已经指向了文件末尾，那么将读取不到任何字符，也返回 NULL。

注意，读取到的字符串会在末尾自动添加 '\0'，n 个字符也包括 '\0'。也就是说，实际只读取到了 n-1 个字符，如果希望读取 100 个字符，n 的值应该为 101。例如：

```
#define N 101
char str[N];
FILE *fp = fopen("D:\\demo.txt", "r");
fgets(str, N, fp);
```

表示从 D:\\demo.txt 中读取 100 个字符，并保存到字符数组 str 中。

需要重点说明的是，在读取到 n-1 个字符之前如果出现了换行，或者读到了文件末尾，则读取结束。这就意味着，

不管 n 的值多大，fgets() 最多只能读取一行数据，不能跨行。在 C 语言中，没有按行读取文件的函数，我们可以借助 fgets()，将 n 的值设置地足够大，每次就可以读取到一行数据。

【示例】一行一行地读取文件。

```
#include <stdio.h>
#include <stdlib.h>
#define N 100
int main(){
    FILE *fp;
    char str[N+1];
    if( (fp=fopen("d:\\demo.txt","rt")) == NULL ){
        puts("Fail to open file!");
        exit(0);
    }

    while(fgets(str, N, fp) != NULL){
        printf("%s", str);
    }

    fclose(fp);
    return 0;
}
```

将下面的内容复制到 D:\\demo.txt：

```
C 语言中文网
http://c.biancheng.net
一个学习编程的好网站！
```

那么运行结果为：

fgets() 遇到换行时，会将换行符一并读取到当前字符串。该示例的输出结果之所以和 demo.txt 保持一致，该换行的地方换行，就是因为 fgets() 能够读取到换行符。而 gets() 不一样，它会忽略换行符。

写字符串函数 fputs

fputs() 函数用来向指定的文件写入一个字符串，它的用法为：

```
int fputs( char *str, FILE *fp );
```

str 为要写入的字符串，fp 为文件指针。写入成功返回非负数，失败返回 EOF。例如：

```
char *str = "http://c.biancheng.net";
FILE *fp = fopen("D:\\demo.txt", "at+");
fputs(str, fp);
```

表示把把字符串 str 写入到 D:\\demo.txt 文件中。

【示例】向上例中建立的 d:\\demo.txt 文件中追加一个字符串。

```
#include<stdio.h>
int main(){
    FILE *fp;
    char str[102] = {0}, strTemp[100];
    if( (fp=fopen("D:\\demo.txt", "at+")) == NULL ){
        puts("Fail to open file!");
        exit(0);
    }
    printf("Input a string:");
    gets(strTemp);
    strcat(str, "\n");
    strcat(str, strTemp);
    fputs(str, fp);
    fclose(fp);
    return 0;
}
```

运行程序，输入 C C++ Java Linux Shell，打开 D:\\demo.txt，文件内容为：

```
C 语言中文网
http://c.biancheng.net
一个学习编程的好网站！
C C++ Java Linux Shell
```

12.6 以数据块的形式读写文件

fgets() 有局限性，每次最多只能从文件中读取一行内容，因为 fgets() 遇到换行符就结束读取。如果希望读取多行内容，需要使用 fread() 函数；相应地写入函数为 fwrite()。

对于 Windows 系统，使用 fread() 和 fwrite() 时应该以二进制的形式打开文件，具体原因我们已在《文本文件和二进制文件到底有什么区别》一文中进行了说明。

fread() 函数用来从指定文件中读取块数据。所谓块数据，也就是若干个字节的数据，可以是一个字符，可以是一个字符串，可以是多行数据，并没有什么限制。fread() 的原型为：

```
size_t fread ( void *ptr, size_t size, size_t count, FILE *fp );
```

fwrite() 函数用来向文件中写入块数据，它的原型为：

```
size_t fwrite ( void * ptr, size_t size, size_t count, FILE *fp );
```

对参数的说明：

- ptr 为内存区块的指针，它可以是数组、变量、结构体等。fread() 中的 ptr 用来存放读取到的数据，fwrite() 中的 ptr 用来存放要写入的数据。
- size：表示每个数据块的字节数。
- count：表示要读写的数据块的块数。
- fp：表示文件指针。
- 理论上，每次读写 size*count 个字节的数据。

size_t 是在 stdio.h 和 stdlib.h 头文件中使用 typedef 定义的数据类型，表示无符号整数，也即非负数，常用来表示数量。

返回值：返回成功读写的块数，也即 count。如果返回值小于 count：

- 对于 fwrite() 来说，肯定发生了写入错误，可以用 ferror() 函数检测。
- 对于 fread() 来说，可能读到了文件末尾，可能发生了错误，可以用 ferror() 或 feof() 检测。

【示例】从键盘输入一个数组，将数组写入文件再读取出来。

```
#include<stdio.h>
#define N 5
int main(){
    //从键盘输入的数据放入a，从文件读取的数据放入b
    int a[N], b[N];
    int i, size = sizeof(int);
    FILE *fp;

    if( (fp=fopen("D:\\demo.txt", "rb+")) == NULL ){  //以二进制方式打开
        puts("Fail to open file!");
        exit(0);
    }

    //从键盘输入数据 并保存到数组a
    for(i=0; i<N; i++){
        scanf("%d", &a[i]);
    }
    //将数组a的内容写入到文件
    fwrite(a, size, N, fp);
    //将文件中的位置指针重新定位到文件开头
    rewind(fp);
    //从文件读取内容并保存到数组b
    fread(b, size, N, fp);
    //在屏幕上显示数组b的内容
    for(i=0; i<N; i++){
        printf("%d ", b[i]);
```

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

    fclose(fp);
    return 0;
}
```

运行结果：
23 409 500 100 222↙
23 409 500 100 222

打开 D:\\demo.txt，发现文件内容根本无法阅读。这是因为我们使用"rb+"方式打开文件，数组会原封不动地以二进制形式写入文件，一般无法阅读。

数据写入完毕后，位置指针在文件的末尾，要想读取数据，必须将文件指针移动到文件开头，这就是 rewind(fp);的作用。更多关于 rewind 函数的内容请点击：C 语言 rewind 函数。

> 文件的后缀不一定是 .txt，它可以是任意的，你可以自己命名，例如 demo.ddd、demo.doc、demo.diy 等。

【示例】从键盘输入两个学生数据，写入一个文件中，再读出这两个学生的数据显示在屏幕上。

```
#include<stdio.h>

#define N 2

struct stu{
    char name[10]; //姓名
    int num;  //学号
    int age;  //年龄
    float score;  //成绩
}boya[N], boyb[N], *pa, *pb;

int main(){
    FILE *fp;
    int i;
    pa = boya;
    pb = boyb;
    if( (fp=fopen("d:\\demo.txt", "wb+")) == NULL ){
        puts("Fail to open file!");
        exit(0);
    }

    //从键盘输入数据
    printf("Input data:\n");
    for(i=0; i<N; i++,pa++){
        scanf("%s %d %d %f",pa->name, &pa->num,&pa->age, &pa->score);
    }
```

```
    //将数组 boya 的数据写入文件
    fwrite(boya, sizeof(struct stu), N, fp);
    //将文件指针重置到文件开头
    rewind(fp);
    //从文件读取数据并保存到数据 boyb
    fread(boyb, sizeof(struct stu), N, fp);

    //输出数组 boyb 中的数据
    for(i=0; i<N; i++,pb++){
        printf("%s  %d  %d  %f\n", pb->name, pb->num, pb->age, pb->score);
    }
    fclose(fp);
    return 0;
}
```

运行结果：

```
Input data:
Tom 2 15 90.5↙
Hua 1 14 99↙
Tom   2   15   90.500000
Hua   1   14   99.000000
```

12.7 格式化读写文件

fscanf() 和 fprintf() 函数与前面使用的 scanf() 和 printf() 功能相似，都是格式化读写函数，两者的区别在于 fscanf() 和 fprintf() 的读写对象不是键盘和显示器，而是磁盘文件。

这两个函数的原型为：

```
int fscanf ( FILE *fp, char * format, ... );
int fprintf ( FILE *fp, char * format, ... );
```

fp 为文件指针，format 为格式控制字符串，... 表示参数列表。与 scanf() 和 printf() 相比，它们仅仅多了一个 fp 参数。例如：

```
FILE *fp;
int i, j;
char *str, ch;
fscanf(fp, "%d %s", &i, str);
fprintf(fp,"%d %c", j, ch);
```

fprintf() 返回成功写入的字符的个数，失败则返回负数。fscanf() 返回参数列表中被成功赋值的参数个数。

【示例】用 fscanf 和 fprintf 函数来完成对学生信息的读写。

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

```
2.
3.  #define N 2
4.
5.  struct stu{
6.      char name[10];
7.      int num;
8.      int age;
9.      float score;
10. } boya[N], boyb[N], *pa, *pb;
11.
12. int main(){
13.     FILE *fp;
14.     int i;
15.     pa=boya;
16.     pb=boyb;
17.     if( (fp=fopen("D:\\demo.txt","wt+")) == NULL ){
18.         puts("Fail to open file!");
19.         exit(0);
20.     }
21.
22.     //从键盘读入数据，保存到boya
23.     printf("Input data:\n");
24.     for(i=0; i<N; i++,pa++){
25.         scanf("%s %d %d %f", pa->name, &pa->num, &pa->age, &pa->score);
26.     }
27.     pa = boya;
28.     //将boya中的数据写入到文件
29.     for(i=0; i<N; i++,pa++){
30.         fprintf(fp,"%s %d %d %f\n", pa->name, pa->num, pa->age, pa->score);
31.     }
32.     //重置文件指针
33.     rewind(fp);
34.     //从文件中读取数据，保存到boyb
35.     for(i=0; i<N; i++,pb++){
36.         fscanf(fp, "%s %d %d %f\n", pb->name, &pb->num, &pb->age, &pb->score);
37.     }
38.     pb=boyb;
39.     //将boyb中的数据输出到显示器
40.     for(i=0; i<N; i++,pb++){
41.         printf("%s  %d  %d  %f\n", pb->name, pb->num, pb->age, pb->score);
42.     }
43.
44.     fclose(fp);
45.     return 0;
46. }
```

运行结果：

```
Input data:
Tom 2 15 90.5↙
Hua 1 14 99↙
Tom  2  15  90.500000
Hua  1  14  99.000000
```

打开 D:\\demo.txt，发现文件的内容是可以阅读的，格式非常清晰。用 fprintf() 和 fscanf() 函数读写配置文件、日志文件会非常方便，不但程序能够识别，用户也可以看懂，可以手动修改。

如果将 fp 设置为 stdin，那么 fscanf() 函数将会从键盘读取数据，与 scanf 的作用相同；设置为 stdout，那么 fprintf() 函数将会向显示器输出内容，与 printf 的作用相同。例如：

```
#include<stdio.h>
int main(){
    int a, b, sum;
    fprintf(stdout, "Input two numbers: ");
    fscanf(stdin, "%d %d", &a, &b);
    sum = a + b;
    fprintf(stdout, "sum=%d\n", sum);
    return 0;
}
```

运行结果：
Input two numbers: 10 20↙
sum=30

12.8 随机读写文件

前面介绍的文件读写函数都是顺序读写，即读写文件只能从头开始，依次读写各个数据。但在实际开发中经常需要读写文件的中间部分，要解决这个问题，就得先移动文件内部的位置指针，再进行读写。这种读写方式称为随机读写，也就是说从文件的任意位置开始读写。

实现随机读写的关键是要按要求移动位置指针，这称为文件的定位。

文件定位函数 rewind 和 fseek
移动文件内部位置指针的函数主要有两个，即 rewind() 和 fseek()。

rewind() 用来将位置指针移动到文件开头，前面已经多次使用过，它的原型为：

```
void rewind ( FILE *fp );
```

fseek() 用来将位置指针移动到任意位置，它的原型为：

```
int fseek ( FILE *fp, long offset, int origin );
```

参数说明：

1) fp 为文件指针，也就是被移动的文件。

2) offset 为偏移量，也就是要移动的字节数。之所以为 long 类型，是希望移动的范围更大，能处理的文件更大。offset 为正时，向后移动；offset 为负时，向前移动。

3) origin 为起始位置，也就是从何处开始计算偏移量。C 语言规定的起始位置有三种，分别为文件开头、当前位置和文件末尾，每个位置都用对应的常量来表示：

起始点	常量名	常量值
文件开头	SEEK_SET	0
当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

例如，把位置指针移动到离文件开头 100 个字节处：

```
fseek(fp, 100, 0);
```

值得说明的是，fseek() 一般用于二进制文件，在文本文件中由于要进行转换，计算的位置有时会出错。

文件的随机读写

在移动位置指针之后，就可以用前面介绍的任何一种读写函数进行读写了。由于是二进制文件，因此常用 fread() 和 fwrite() 读写。

【示例】从键盘输入三组学生信息，保存到文件中，然后读取第二个学生的信息。

```
#include<stdio.h>

#define N 3

struct stu{
    char name[10]; //姓名
    int num;  //学号
    int age;  //年龄
    float score;  //成绩
}boys[N], boy, *pboys;

int main(){
    FILE *fp;
    int i;
    pboys = boys;
    if( (fp=fopen("d:\\demo.txt", "wb+")) == NULL ){
        printf("Cannot open file, press any key to exit!\n");
        getch();
        exit(1);
    }
```

```
21.
22.     printf("Input data:\n");
23.     for(i=0; i<N; i++,pboys++){
24.         scanf("%s %d %d %f", pboys->name, &pboys->num, &pboys->age, &pboys->score);
25.     }
26.     fwrite(boys, sizeof(struct stu), N, fp);  //写入三条学生信息
27.     fseek(fp, sizeof(struct stu), SEEK_SET);  //移动位置指针
28.     fread(&boy, sizeof(struct stu), 1, fp);  //读取一条学生信息
29.     printf("%s  %d  %d %f\n", boy.name, boy.num, boy.age, boy.score);
30.
31.     fclose(fp);
32.     return 0;
33. }
```

运行结果：

```
Input data:
Tom 2 15 90.5↙
Hua 1 14 99↙
Zhao 10 16 95.5↙
Hua   1   14 99.000000
```

12.9 C 语言实现文件复制功能(包括文本文件和二进制文件)

12.10 FILE 结构体以及缓冲区深入探讨

12.11 C 语言获取文件大小（长度）

12.12 C 语言插入、删除、更改文件内容

第 13 章 C 语言调试教程（非常详细）

所谓调试（Dubug），就是跟踪程序的运行过程，从而发现程序的逻辑错误（思路错误），或者隐藏的缺陷（Bug）。

在调试的过程中，我们可以监控程序的每一个细节，包括变量的值、函数的调用过程、内存中数据、线程的调度等，从而发现隐藏的错误或者低效的代码。

我敢保证，每个人都会遇到逻辑错误，而且会经常遇到，初学者更是错的离谱，所以，必须掌握调试技能，没有选择的余地，没有学会调试就是没有学会编程！

13.1 调试的概念以及调试器的选择

13.2 设置断点，开始调试

13.3 查看和修改变量的值

13.4 单步调试（逐语句调试和逐过程调试）

13.5 即时窗口的使用

13.6 查看、修改运行时的内存

13.7 有条件断点的设置

13.8 assert 断言函数

13.9 调试信息的输出

13.10 VS 调试的总结以及技巧

第 14 章 C 语言内存精讲，让你彻底明白 C 语言的运行机制！

当你决定学习「内存」，你已然超越了 99% 的程序员！

程序是在内存中运行的，一名合格的程序员必须了解内存，学习 C 语言是了解内存布局的最简单、最直接、最有效的途径，C 语言简直是为内存而生的，它比任何一门编程语言都贴近内存。

本专题将为你解开以下谜团：

- C 语言中使用的地址为什么是假的，计算机又是如何通过假的地址访问到真实的物理内存的？
- 一个 C 语言程序在内存中是如何分布的？函数放在哪里？变量放在哪里？字符串放在哪里？
- 为什么全局变量在整个程序中都可以使用，而局部变量只能在函数内部使用？
- 一个 C 语言程序可以使用多大的内存？
- 操作系统和用户程序之间是如何协作的？
- 堆和栈都是什么，它们在程序运行过程中起到什么作用？为什么栈内存的分配效率要高于堆？
- 栈溢出是怎么回事，如何利用栈溢出进行攻击？
- 内存泄漏、野指针、非法内存访问、段错误都是怎么产生的？
- 内存池、线程池、连接池等这些莫名其妙的“池子”是怎么回事？

14.1 一个程序在计算机中到底是如何运行的？

14.2 虚拟内存到底是什么？为什么我们在 C 语言中看到的地址是假的？

14.3 虚拟地址空间以及编译模式

14.4 C 语言内存对齐，提高寻址效率

14.5 内存分页机制，完成虚拟地址的映射

14.6 分页机制究竟是如何实现的？

14.7 MMU 部件以及对内存权限的控制

14.8 Linux 下 C 语言程序的内存布局（内存模型）

14.9 Windows 下 C 语言程序的内存布局（内存模型）

14.10 用户模式和内核模式

14.11 栈（Stack）是什么？栈溢出又是怎么回事？

14.12 一个函数在栈上到底是怎样的？

14.13 函数调用惯例(Calling Convention)

14.14 用一个实例来深入剖析函数进栈出栈的过程

14.15 栈溢出攻击的原理是什么？

14.16 C 语言动态内存分配

14.17 malloc 函数背后的实现原理——内存池

14.18 C 语言野指针以及非法内存操作

14.19 C 语言内存泄露（内存丢失）

14.20 C 语言变量的存储类别和生存期

第 15 章 C 语言头文件的编写（多文件编程）

多文件编程就是把多个头文件（.h 文件）和源文件（.c 文件）组合在一起构成一个程序，这是 C 语言的重点，也是 C 语言的难点。C 语言头文件的编写是其中的重点内容，有很多细节需要注意，有的甚至会让你感觉奇怪。

多文件编程既涉及到了内存，也涉及到了编译原理，市面上的绝大部分资料对此也语焉不详，所以很多初学者对此都非常困惑。

学会了多文件编程，你就可以使用 C 语言来开发中大型项目了，对初学者来说，这简直是跨域了一大步。

15.1 从 extern 关键字开始谈 C 语言多文件编程

15.2 那些被编译器隐藏了的过程

15.3 目标文件和可执行文件里面都有什么？

15.4 到底什么是链接，它起到了什么作用？

15.5 符号——链接的粘合剂

15.6 强符号和弱符号

15.7 强引用和弱引用

15.8 C 语言模块化编程中的头文件

15.9 C 语言标准库以及标准头文件

15.10 细说 C 语言头文件的路径

15.11 防止 C 语言头文件被重复包含

15.12 C 语言 static 变量和函数

15.13 一个比较规范的 C 语言多文件编程的例子

第 16 章 C 语言项目实战（带源码和解析）

学知识容易，用知识难！编程是一门不断实践的技术，读者不但要阅读教程，还要自己动手去开发项目，将知识运用到实际中。

初学者往往有这样的困惑：教程已经阅读过了，其中的知识点也都理解了，但是真正编写代码的时候却感觉无从下手，甚至连数组排序、文件复制、百钱买白鸡这样的小程序都不能完成。究其原因，就是缺少实践，没有培养起编程思维，没有处理相关问题的经验。编程能力和你的代码量是成正比的！

现在，我们就带大家实践一下，做几个小项目。如下所示，每个项目都给出了规范的源码、清晰的思路、丰富的注释以及透彻的解析。

16.1 贪吃蛇游戏（彩色版）【带源码和解析】

C 语言中文网提供的贪吃蛇游戏不依赖 TC 环境，不依赖任何第三方库，可以在 VC 6.0、VS、C-Free 等常见 IDE 中编译通过。

设计贪吃蛇游戏的主要目的是让大家夯实 C 语言基础，训练编程思维，培养解决问题的思路，领略多姿多彩的 C 语言。

游戏开始后，会在中间位置出现一条只有三个节点的贪吃蛇，并随机出现一个食物，如下图所示：

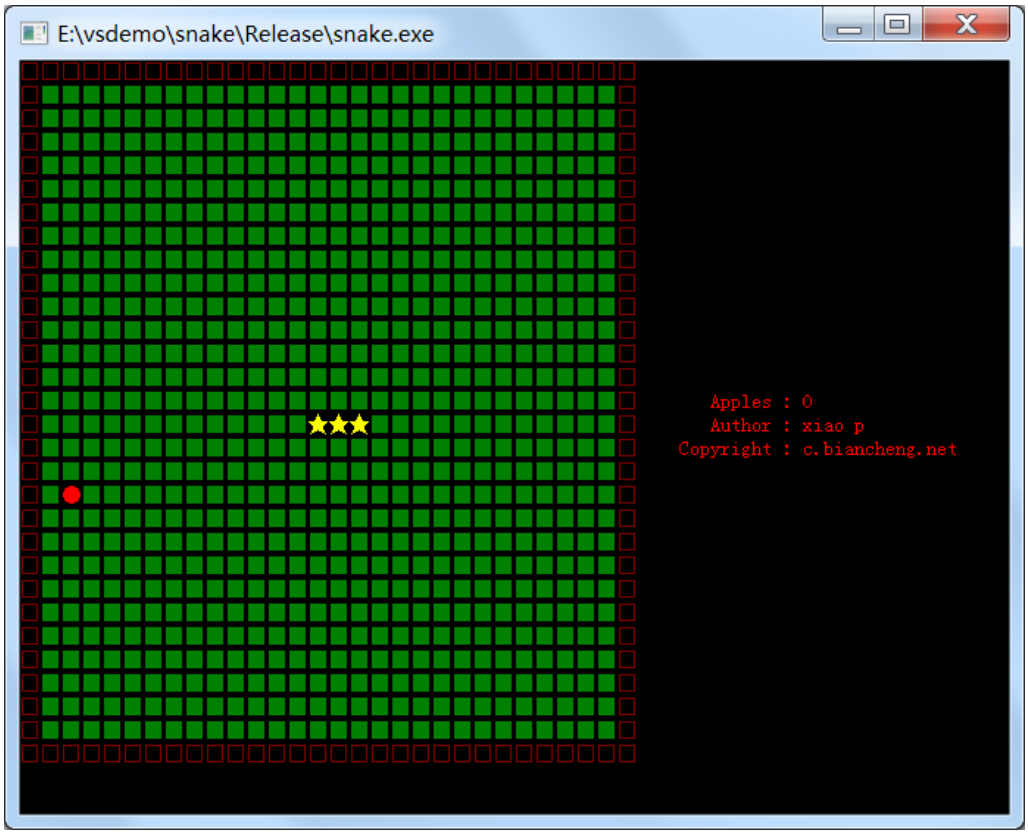


图 1：游戏初始化

按下键盘上的任意一个键，贪吃蛇开始移动。和大部分游戏一样，你可以通过 W、A、S、D 四个键来控制移动方向，如下图所示：

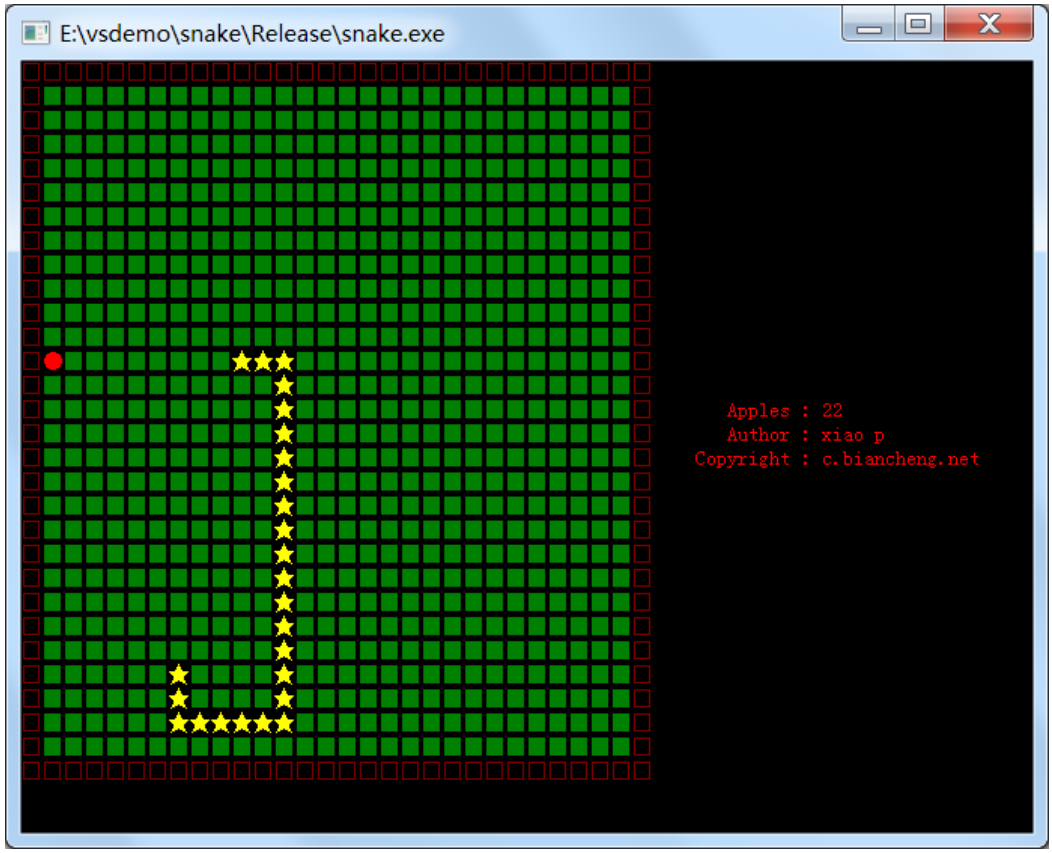


图 2：游戏进行中

当贪吃蛇出界或者撞到自己时，游戏结束，如下图所示：

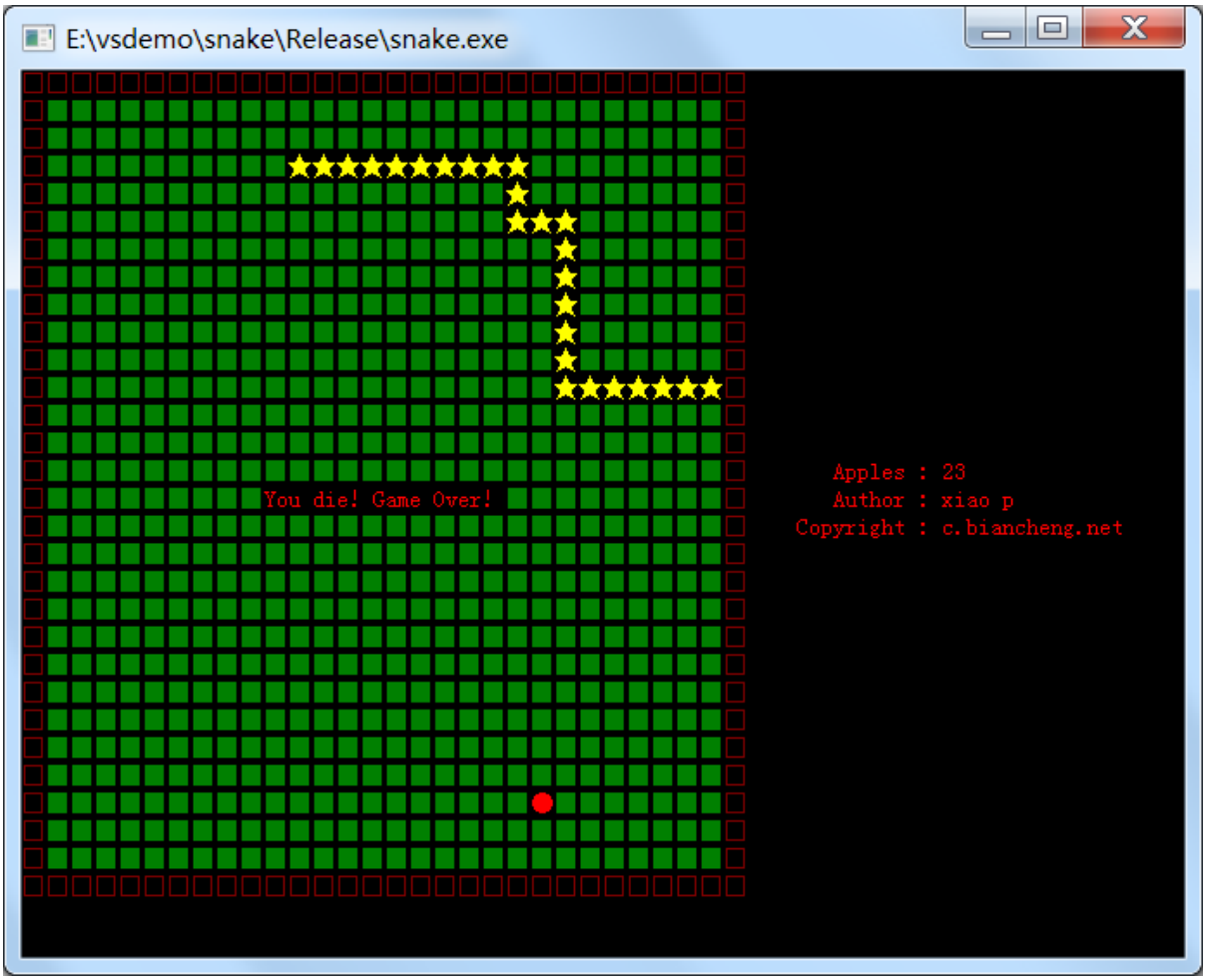


图 3：游戏结束

演示程序百度网盘下载地址：https://pan.baidu.com/s/1pMTS3NH　密码：u5ee

注意：上面的下载地址仅仅提供了已经编译好的贪吃蛇程序，如果你希望查看游戏源码，或者想了解贪吃蛇是如何编写的，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

16.2 2048 小游戏【带源码和解析】

2048 游戏是风靡一时的小游戏，我们提供的 2048 小游戏不依赖 TC 环境，不依赖任何第三方库，可以在 VS、CodeBlocks、DEV C++ 等常见 IDE 中编译通过。

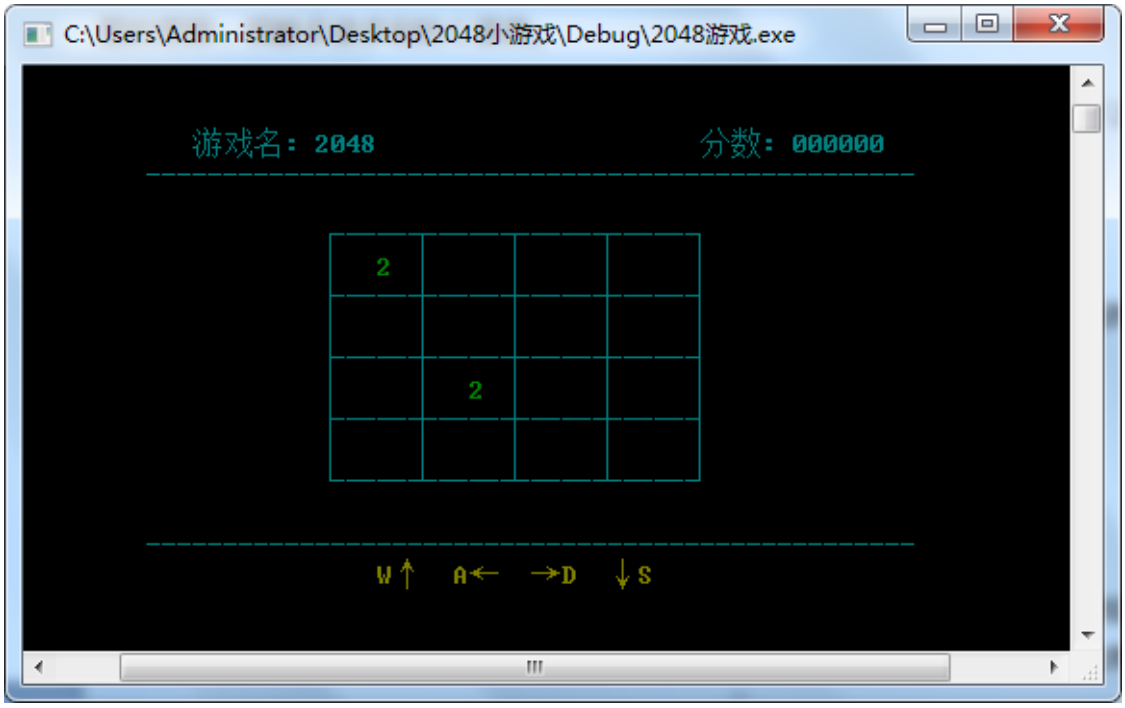
我们提供给大家的 2048 小游戏，不要求大家精通 C 语言，也不会涉及到指针的使用，只需要学会以下几个知识点即可：

- 会使用变量（int、char）和二维数组；
- 能够懂得函数的声明和使用；

- 掌握 C 语言的分支结构（if-else 和 switch 语句）和循环结构（while 和 fo() 循环）；

设计 2048 小游戏的主要目的是让大家夯实 C 语言基础，训练编程思维，培养解决问题的思路，领略多姿多彩的 C 语言。

游戏开始，会生成一个 4 行 4 列的初始界面，界面中有任意两个位置会出现数字 2 或者 4，如下图所示：



游戏采用 W\A\S\D 或者 ↑↓←→ 来控制，直到整个界面中没有可以合并的数字，游戏结束，如下图所示：

演示程序百度网盘下载地址：https://pan.baidu.com/s/1bqkUQ8v 密码：3id4

注意：上面的下载地址仅仅提供了已编译好的 2048 小游戏，如果你希望查看游戏源码，或者想了

16.3 推箱子小游戏（彩色版）【带源码和解析】

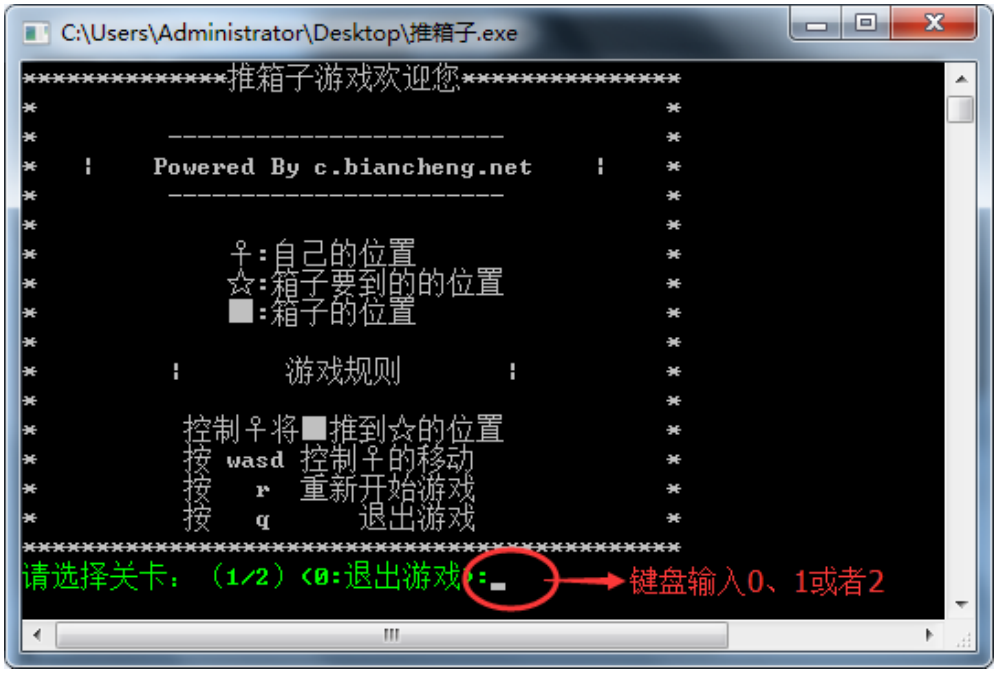
C 语言中文网提供的推箱子游戏不依赖 TC 环境，不依赖任何第三方库，可以在 VS、CodeBlocks、Dev C++ 等常见 IDE 中编译通过。

我们提供给大家的推箱子游戏，不要求大家精通 C 语言，也不会涉及到指针的使用，只需要学会以下几个知识点即可：

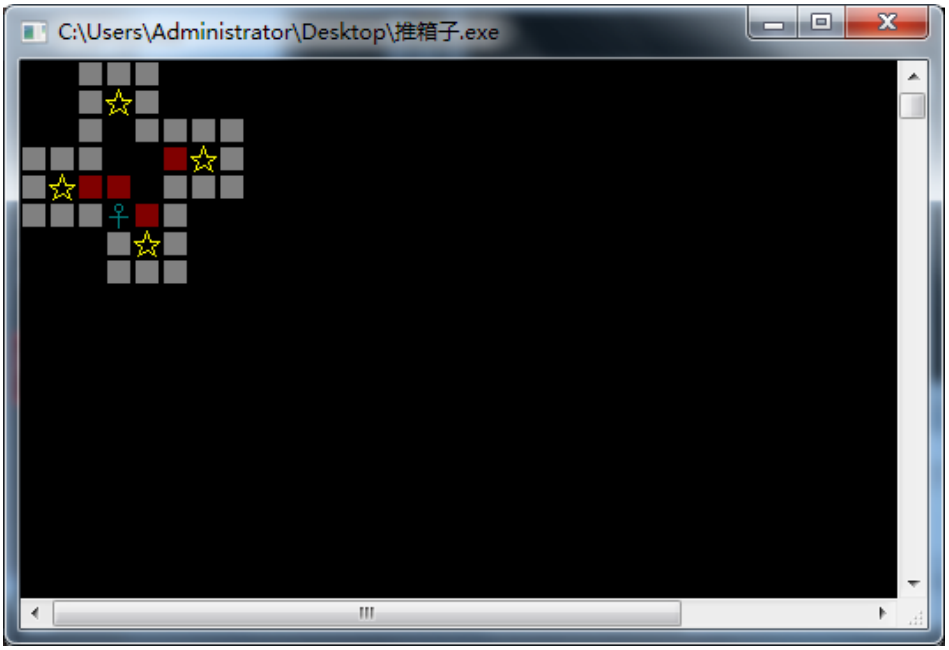
- 会使用变量（int、char）和二维数组；
- 能够懂得函数的声明和使用；
- 掌握 C 语言的分支结构（if-else 和 switch 语句）和循环结构（while 和 fo() 循环）；

设计推箱子游戏的主要目的是让大家夯实 C 语言基础，训练编程思维，培养解决问题的思路，领略多姿多彩的 C 语言。

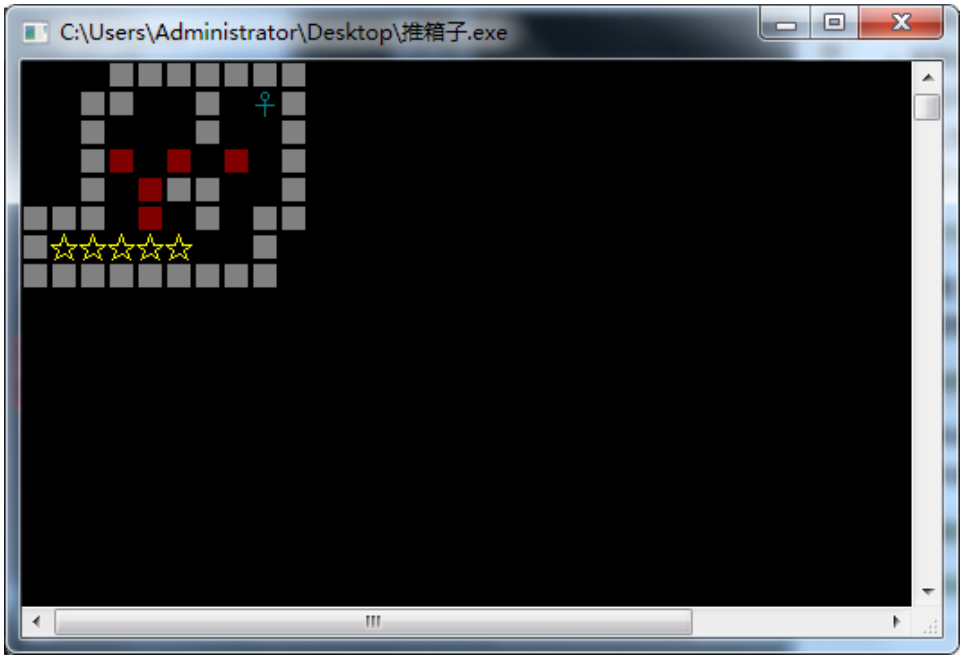
游戏开始后，首先会提示游戏规则，由用户键盘输入选择游戏关卡，如下图所示：



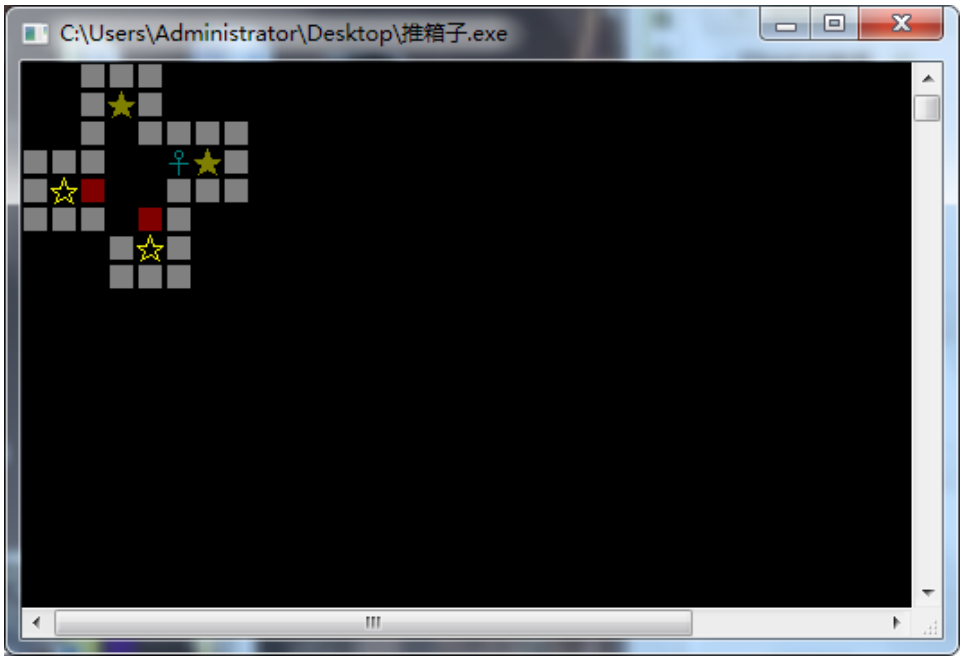
输入 1，即选择第 1 关，游戏初始状态如下图所示：



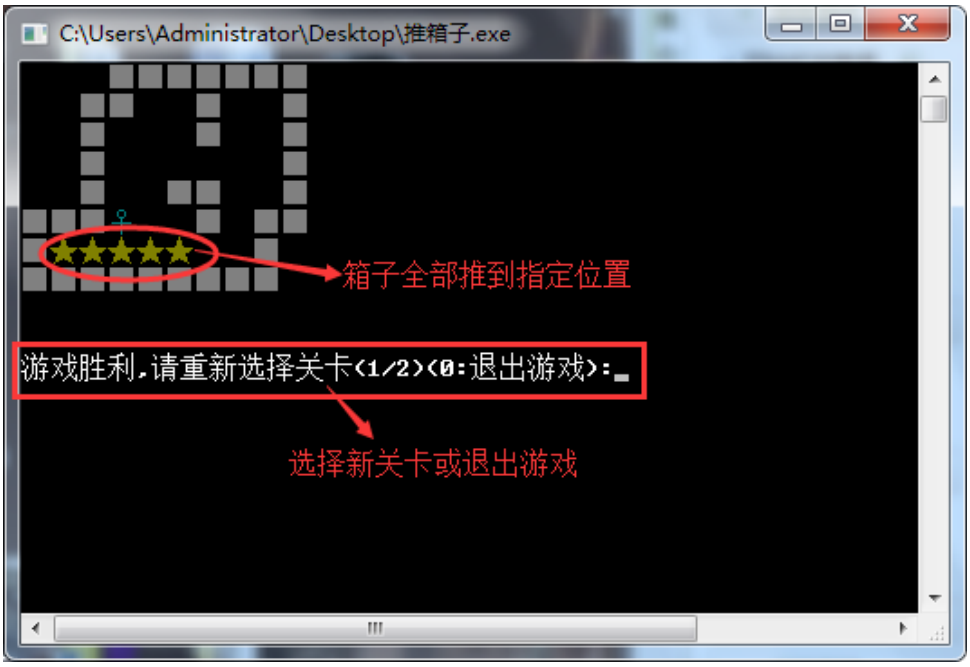
输入 2，即选择第 2 关，游戏初始状态如下图所示：



游戏开始后，你可以通过W、A、S、D四个键来控制人物的移动方向，如下图所示：



当所有红色箱子全部推到黄色星星处，游戏挑战成功，如下图所示：



演示程序百度网盘下载地址：https://pan.baidu.com/s/1o9LLSTG 密码: dzsp

注意：上面的下载地址仅提供了已编译好的推箱子小游戏，如果你希望查看游戏源码，或者想了解推箱子小游戏是如何编写的，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

16.4 扫雷游戏【带源码和解析】

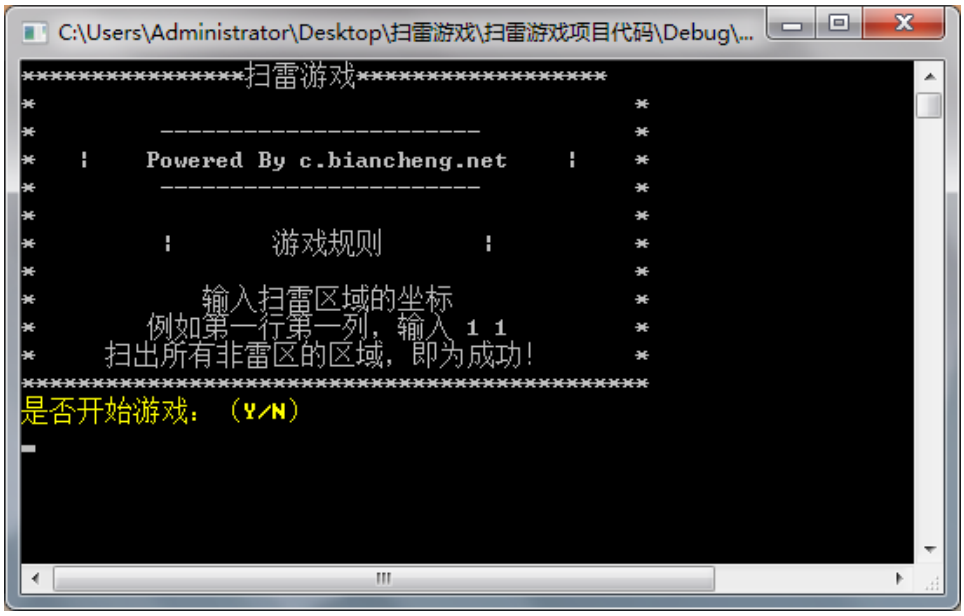
C 语言中文网提供的扫雷游戏不依赖 TC 环境，不依赖第三方库，可以在 VC6.0、VS、CodeBlocks、Dev C++ 编译通过。

我们提供给大家的扫雷游戏，不要求大家精通 C 语言，也不会涉及到指针的使用，只需要学会以下几个知识点即可：

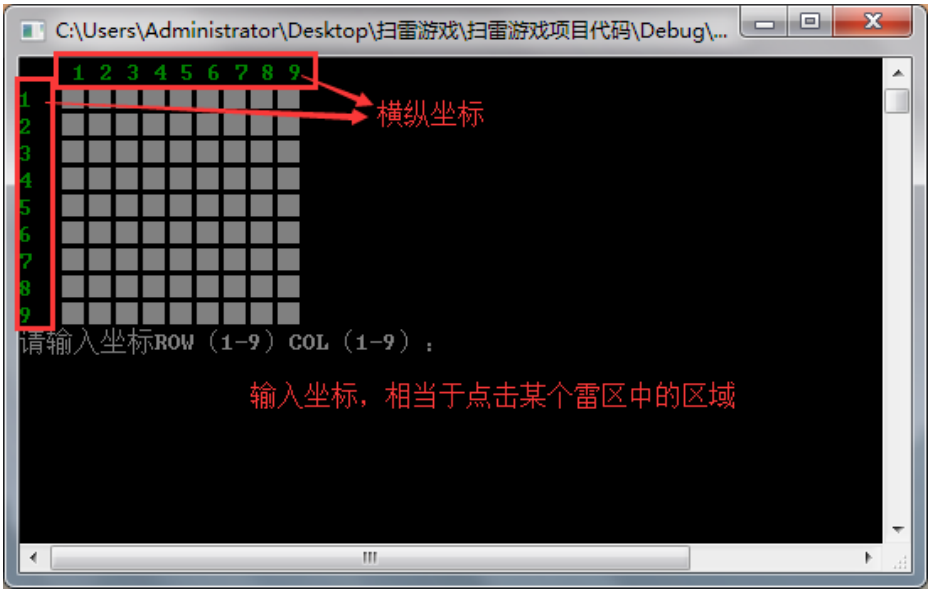
- 会使用变量（int、char）和二维数组；
- 能够懂得函数的声明和使用；
- 掌握 C 语言的分支结构（if-else 和 switch 语句）和循环结构（while 和 fo() 循环）；

设计扫雷游戏的主要目的是让大家夯实 C 语言基础，训练编程思维，培养解决问题的思路，领略多姿多彩的 C 语言。

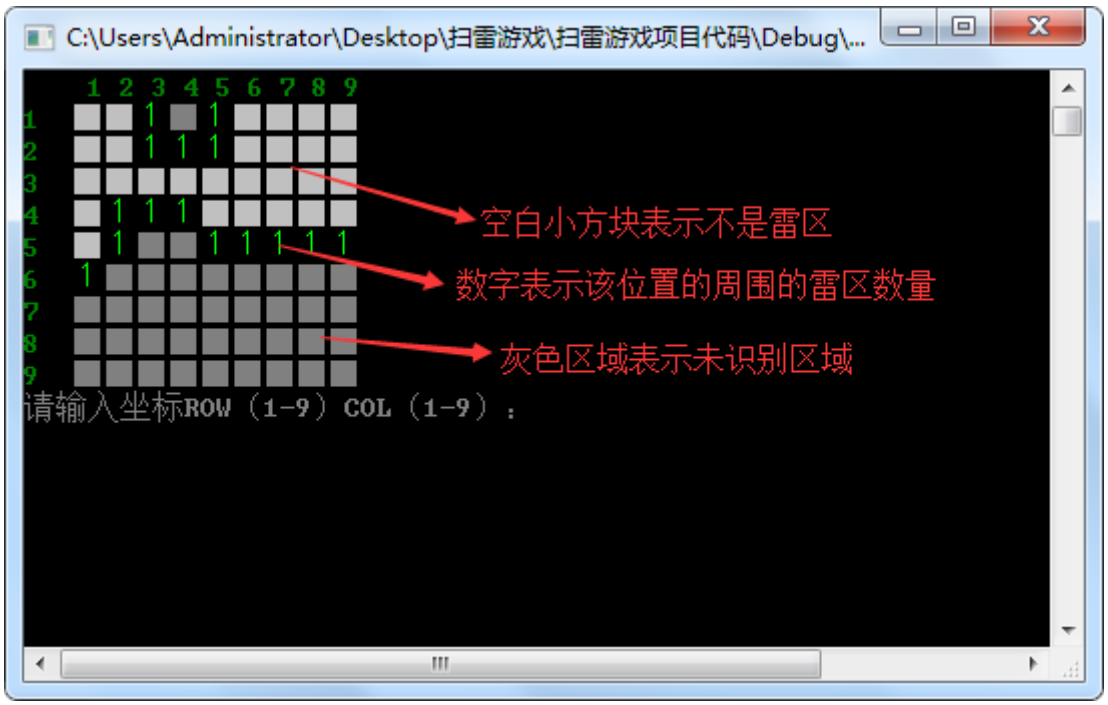
游戏开始后，会出现一个有关扫雷游戏的介绍，同时由用户选择，游戏是否开始。用户输入 “y” 或者 “Y” 都可开始游戏，输入 “n” 或 “N” 或其他字符会直接退出游戏。如下图所示：



输入 “y” 或 “Y”，游戏开始，进入扫雷游戏的初始界面，如下图所示：



玩家通过输入坐标选定某个区域，例如，输入 （1,1），游戏结果如下图所示：



结果显示，（1,1）位置不是雷区，且其周围也没有雷区，同时还顺带为玩家显示了一部分同样不是雷区，且其周围也无雷区的区域。

若游戏中探测到雷区时，则 game over！如下图所示：

当所有雷区全部成功躲避，将所有安全区域全部扫出来时，游戏成功！如下图所示：

扫雷游戏演示程序百度网盘下载地址：https://pan.baidu.com/s/1pM2Nrw3 密码: qe2r

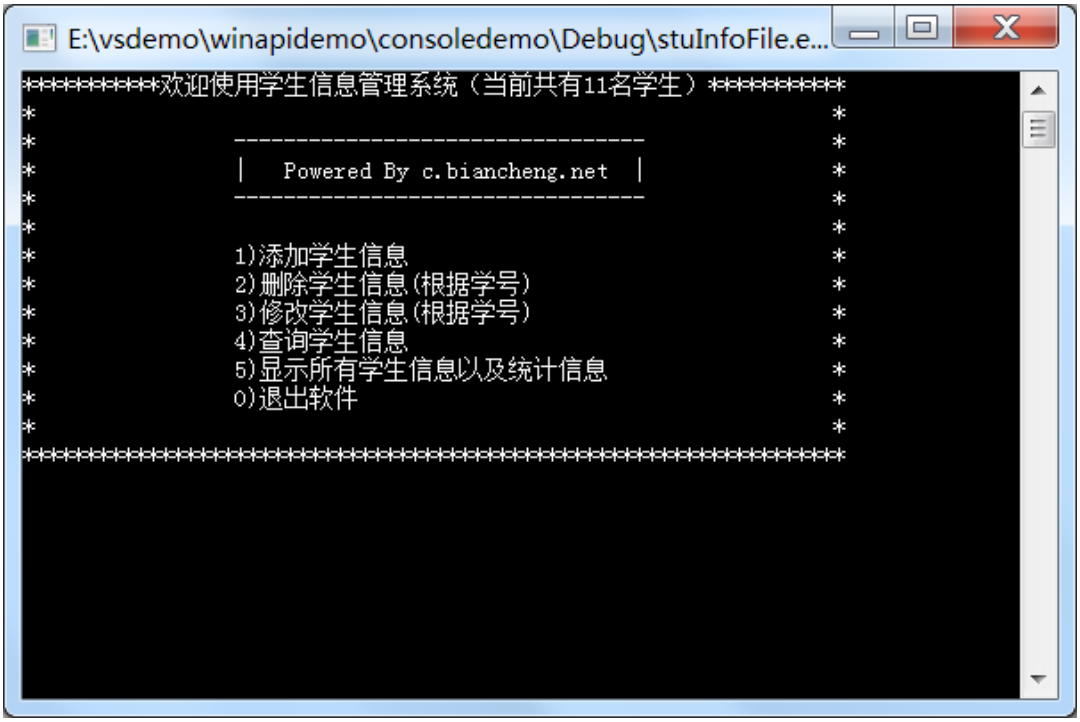
注意：上面的下载地址仅提供了已编译好的扫雷游戏，如果你希望查看游戏源码，或者想了解扫雷游戏是如何编写的，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

16.5 学生信息管理系统（文件版）【带源码和解析】

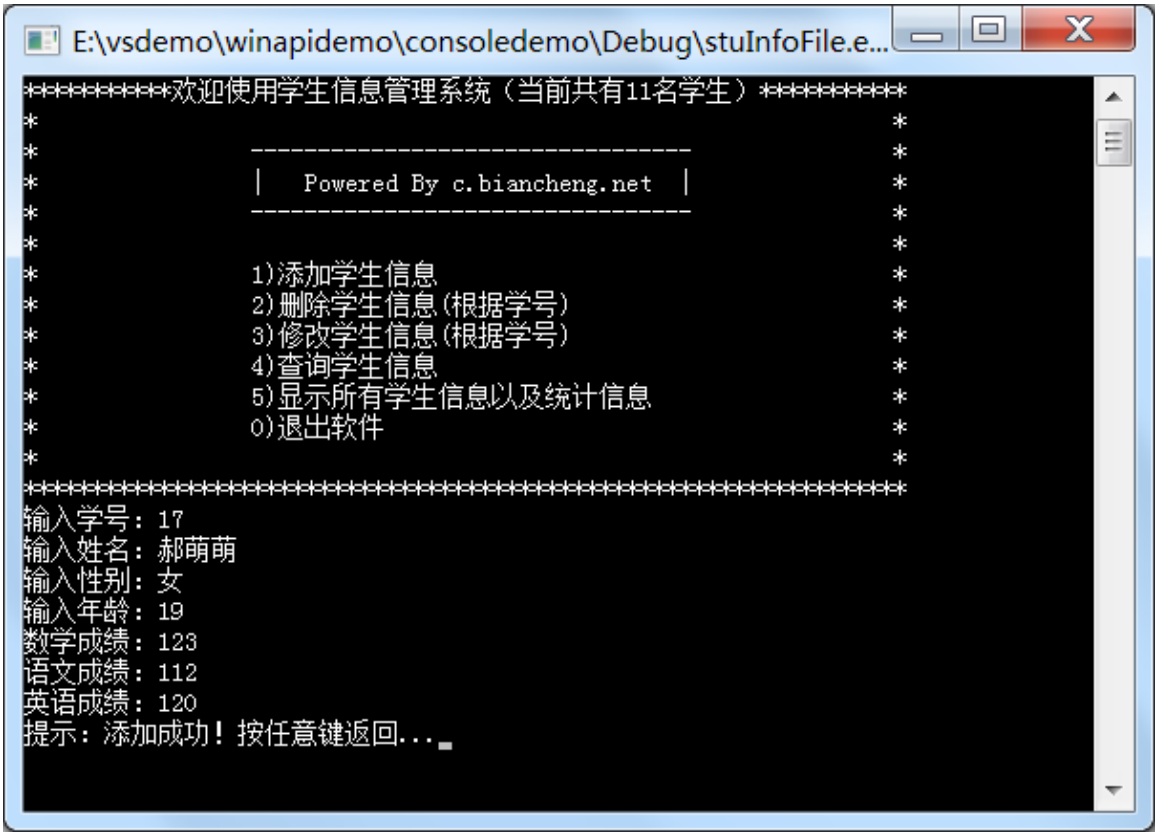
学生管理系统可以用来管理班级学生，能够对学生信息（包括姓名、性别、年龄、成绩等）进行增加、删除、更改、查询等操作。

该软件将学生信息保存在文件中，直接对文件中的数据进行增删改查操作，除了能够培养您的编程思维，还能让你深入了解文件操作。

软件运行后，首先会显示主菜单，让用户选择要进行的操作，如下图所示：

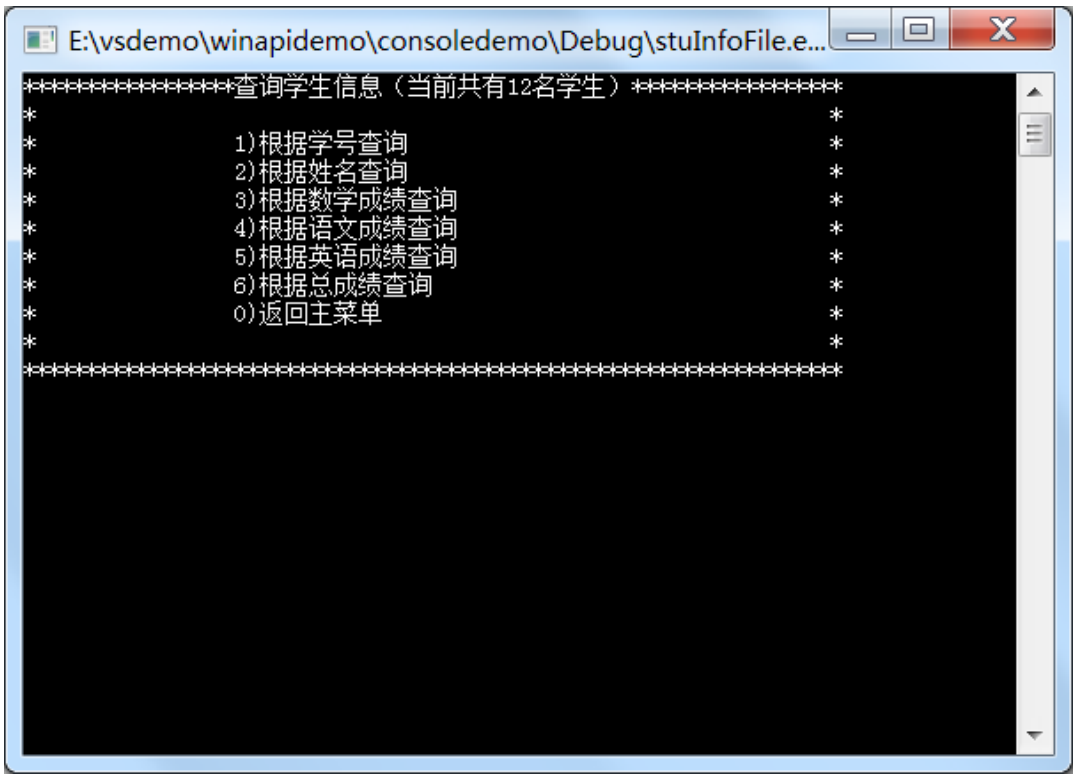


输入数字 1~5，或者 0，就可以进行相应的操作。例如输入数字 1，就可以添加学生信息，如下所示：



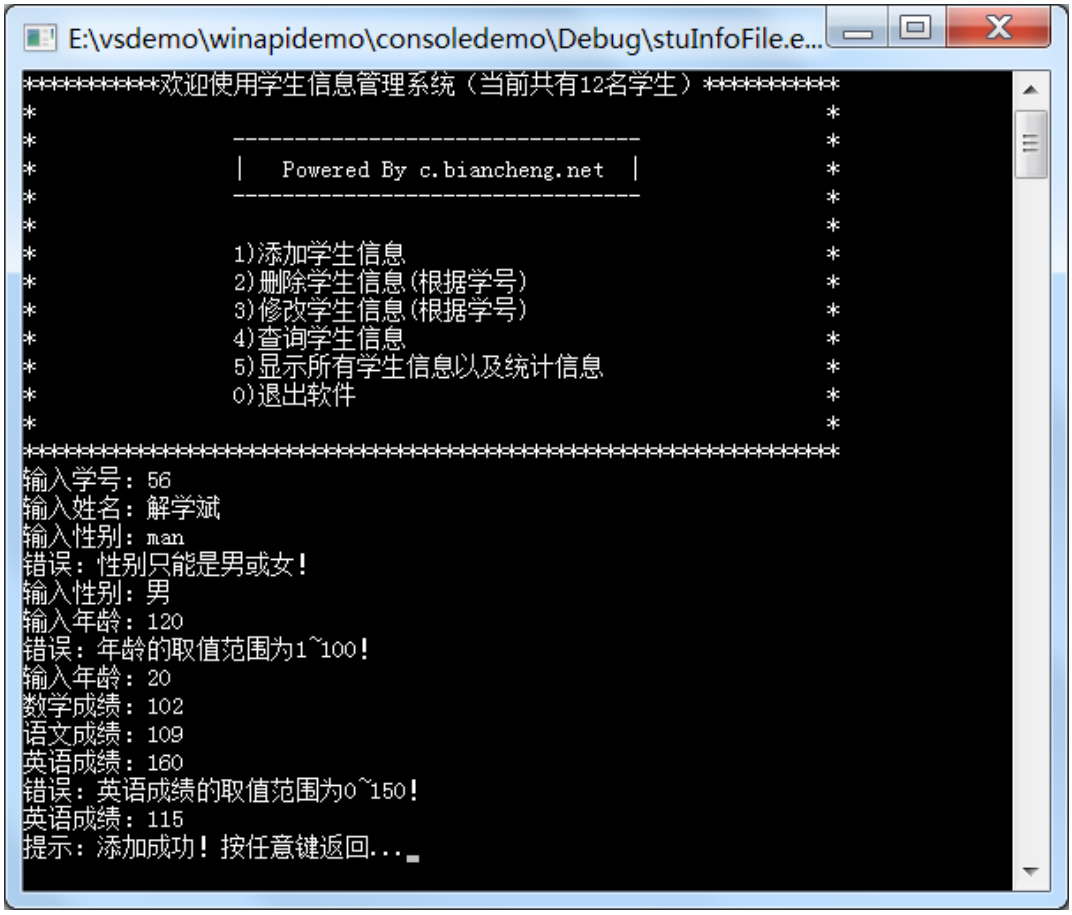
添加成功后，会给出提示信息。

输入数字 4，会显示子菜单，可以用来查询学生信息，如下图所示：



输入数字 3，根据数学成绩来查询，如下图所示：

软件还会对用户输入的信息进行校验，不符合规则的数据会给出提示。例如，输入不合法的用户数据：



演示程序下载地址：http://pan.baidu.com/s/1pK6hflV　　提取密码：neve

注意：上面的下载地址仅仅提供了已编译好的学生信息管理系统，如果你希望查看它的源码，或者想了解它是如何编写的，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。

16.6 学生信息管理系统（数据结构版）【带源码和解析】

C 语言中文网提供了三个版本的学生信息管理系统：文件版、数据结构版和密码版。

数据结构版和文件版的学生管理系统在使用是上完全相同的，已在《学生信息管理系统演示和说明（文件版）》进行了演示和说明，请大家点击链接查看，这里不再赘述。

文件版的学生管理系统将学生信息保存在文件中，重点是如何对文件进行增删改查操作，能够加深大家对文件操作的理解。而数据结构版的学生管理系统重点是维护链表，能够将大家学到的数据结构的知识运用到实际中。

演示程序下载地址：http://pan.baidu.com/s/1dDS5WIX　　提取密码：9qv4

注意：上面的下载地址仅仅提供了已编译好的学生信息管理系统，如果你希望查看它的源码，或者

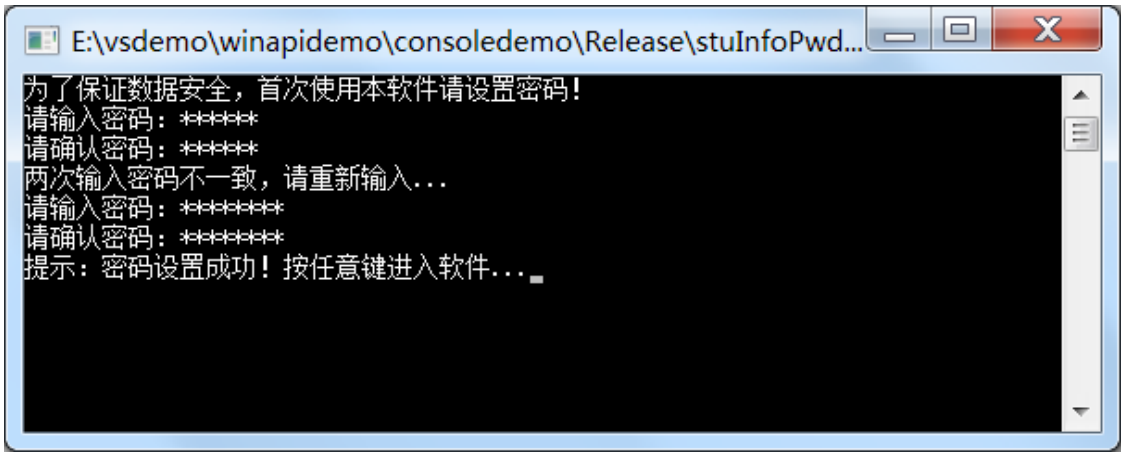
16.7 学生信息管理系统（密码版）【带源码和解析】

在《学生信息管理系统演示和说明（文件版）》和《学生信息管理系统演示和说明（数据结构版）》中我们演示了文件版和数据结构版的学生信息管理系统，最终都将学生信息存储到文件中。不过，这些数据并不安全，任何人都可以查看和修改。

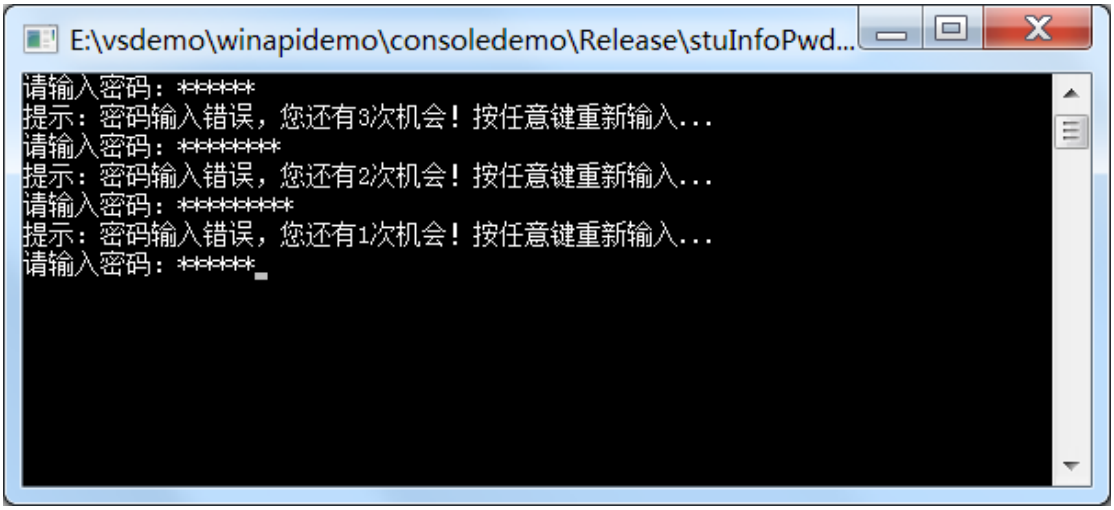
我们有必要增强程序的安全性，对学生信息进行加密，用户只有输入正确的密码才能查看和使用学生信息。我们不妨将该版本的学生管理系统称为密码版。

密码版的学生管理系统和文件版、数据结构版的学生管理系统大部分操作都相同，只是增加了与密码相关的选项。

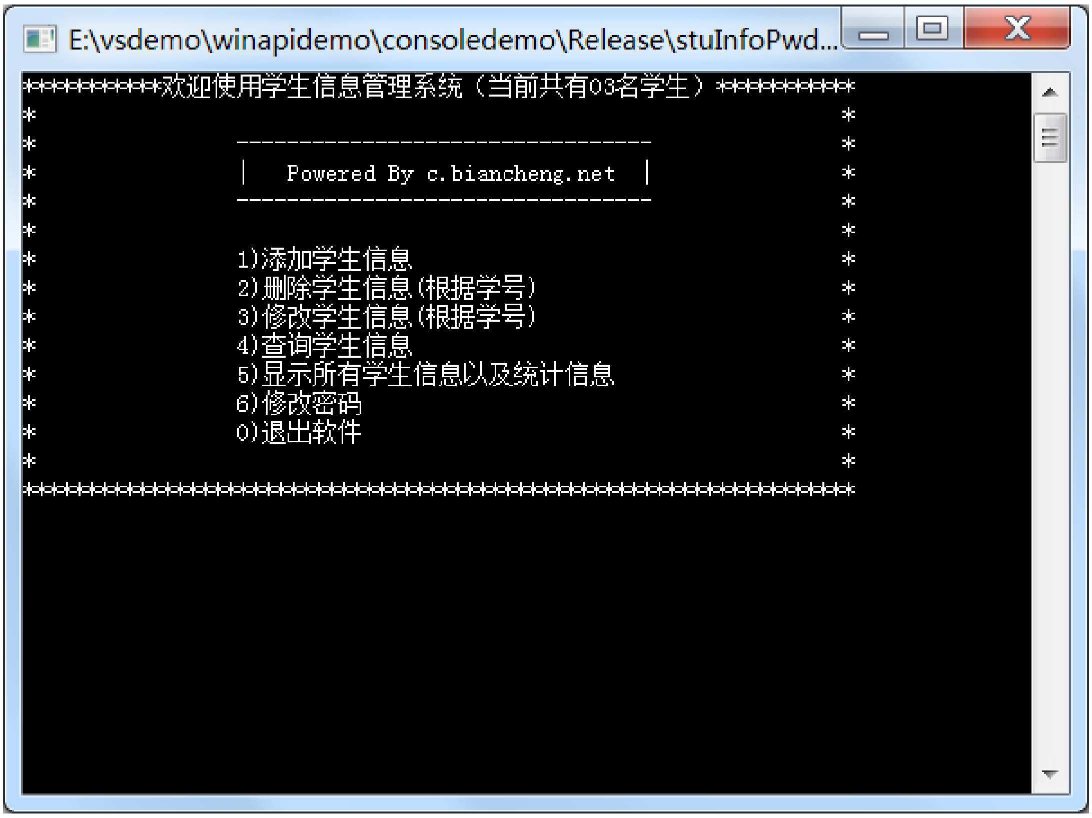
第一次运行软件，会让用户先设置密码再使用，如下图所示：



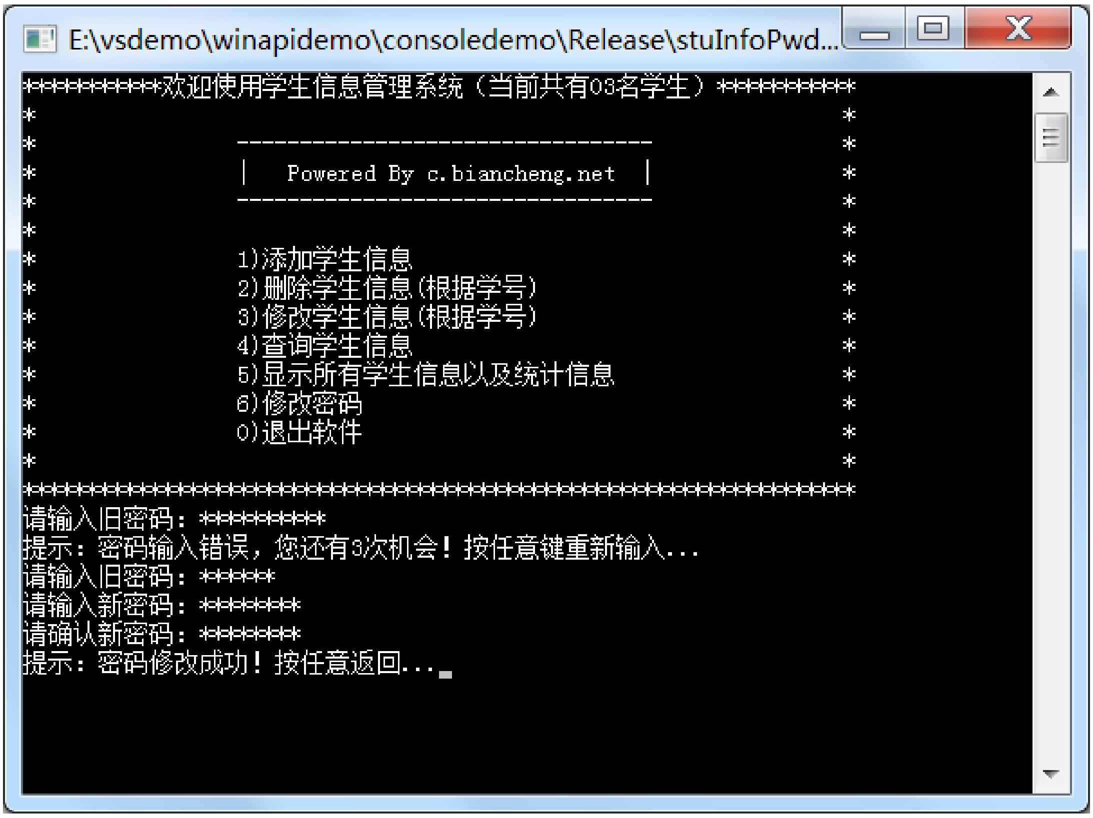
如果不是第一次运行软件，会让用户输入密码校验，如下图所示：



输入密码后，进入主界面，显示主菜单，如下图所示：



与前面两个版本相比，多了一个“修改密码”的选项。输入数字 6，进行修改密码，如下图所示：



演示程序下载地址：http://pan.baidu.com/s/1dDZrBBr 提取密码：kp96

注意：上面的下载地址仅仅提供了已编译好的学生信息管理系统，如果你希望查看它的源码，或者想了解它是如何编写的，请开通 VIP 会员（提供 QQ 一对一答疑，并赠送 1TB 编程资料）。