

# C 语言

## 程序设计和 C 语言简介

### 计算机程序

所谓程序,就是一组计算机能识别和执行的指令。每一条指令使计算机执行特定的操作。只要让计算机执行这个程序,计算机就会“自动地”执行各条指令,有条不紊地进行工作。一个特定的指令序列,用来完成一定的功能。为了使计算机系统能实现各种功能,需要成千上万个程序。这些程序大多数是由计算机软件设计人员根据需要设计好的,作为计算机的软件系统的一部分提供给用户使用。此外,用户还可以根据自己的实际需要设计一些应用程序,例如学生成绩统计程序、财务管理程序、工程中的计算程序等。

### 计算机语言

**机器语言** 计算机工作基于二进制,从根本上说,计算机只能识别和接受由 0 和 1 组成的指令。在计算机发展的初期,一般计算机的指令长度为 16,即以 16 个二进制数(0 或 1)组成一条指令,16 个 0 和 1 可以组成各种排列组合。例如,用

1011011000000000

让计算机进行一次加法运算。人要使计算机知道和执行自己的意图,就要编写许多条由 0 和 1 组成的指令。然后要用纸带穿孔机以人工的方法在特制的黑色纸带上穿孔,在指定的位置上有孔代表 1,无孔代表 0。一个程序往往需要一卷长长的纸带。在需要运行此程序时就将此纸带装在光电输入机上,当光电输入机从纸带读入信息时,有孔处产生一个电脉冲,指令变成电信号,让计算机执行各种操作。

这种计算机能直接识别和接受的二进制代码称为机器指令(machine instruction)。机器指令的集合就是该计算机的机器语言(machine language)。在语言的规则中规定各种指令的表现形式及其作用

**符号语言** 为了克服机器语言的上述缺点,人们创造出符号语言(symbolic language),它用一些英文字母和数字表示一个指令,例如用 ADD 代表“加”,SUB 代表“减”,LD 代表“传送”等。如上面介绍的那条机器指令可以改用符号指令代替:

ADD A,B (执行  $A+B\rightarrow A$ , 将寄存器 A 中的数与寄存器 B 中的数相加, 放到寄存器 A 中)

显然,计算机并不能直接识别和执行符号语言的指令,需要用一种称为汇编程序的软件,把符号语言的指令转换为机器指令。一般,一条符号语言的指令对应转换为一条机器指令。转换的过程称为“代真”或“汇编”,因此,符号语言又称为符号汇编语言(symbolic assembler language)或汇编语言(assembler language)。

虽然汇编语言比机器语言简单好记一些,但仍然难以普及,只在专业人员中使用。

不同型号的计算机的机器语言和汇编语言是互不通用的。用甲机器的机器语言编写的程序在乙机器上不能使用。机器语言和汇编语言是完全依赖于具体机器特性的,是面向机器的语言。由于它“贴近”计算机,或者说离计算机“很近”,称为计算机低级语言(low level language)。

**高级语言** 为了克服低级语言的缺点,20世纪50年代创造出了第一个计算机高级语言——FORTRAN语言。它很接近于人们习惯使用的自然语言和数学语言。程序中用到的语句和指令是用英文单词表示的,程序中所用的运算符和运算表达式和人们日常所用的数学式子差不多,很容易理解。程序运行的结果用英文和数字输出,十分方便。例如在FORTRAN语言程序中,想计算和输出  $3.5 \times 6 \sin(\pi/3)$ ,只须写出下面这样一个语句:

```
PRINT *, 3.5 * 6 * SIN(3.1415926/3)
```

即可得到计算结果。显然这是很容易理解和使用的。

这种语言功能很强,且不依赖于具体机器,用它写出的程序对任何型号的计算机都适用(或只须作很少的修改),它与具体机器距离较远,故称为计算机高级语言(high level language)。

当然,计算机也是不能直接识别高级语言程序的,也要进行“翻译”。用一种称为编译程序的软件把用高级语言写的程序(称为源程序,source program)转换为机器指令的程序(称为目标程序,object program),然后让计算机执行机器指令程序,最后得到结果。高级语言的一个语句往往对应多条机器指令。

## C 语言的特点

(1) 语言简洁、紧凑,使用方便、灵活。C语言一共只有37个关键字(见附录B)、9种控制语句,程序书写形式自由,主要用小写字母表示,压缩了一切不必要的成分。C语言程序比其他许多高级语言简练,源程序短,因此输入程序时工作量少。

实际上,C是一个很小的内核语言,只包括极少的与硬件有关的成分,C语言不直接提供输入和输出语句、有关文件操作的语句和动态内存管理的语句等(这些操作是由编译系统所提供的库函数来实现的),C的编译系统相当简洁。

(2) 运算符丰富。C 语言的运算符包含的范围很广泛,共有 34 种运算符(见附录 C)。C 语言把括号、赋值和强制类型转换等都作为运算符处理,从而使 C 语言的运算类型极其丰富,表达式类型多样化。灵活使用各种运算符可以实现在其他高级语言中难以实现的运算。

(3) 数据类型丰富。C 语言提供的数据类型包括:整型、浮点型、字符型、数组类型、指针类型、结构体类型和共用体类型等,C 99 又扩充了复数浮点类型、超长整型(long long)和布尔类型(bool)等。尤其是指针类型数据,使用十分灵活和多样化,能用来实现各种复杂的数据结构(如链表、树、栈等)的运算。

(4) 具有结构化的控制语句(如 if…else 语句、while 语句、do…while 语句、switch 语句和 for 语句)。用函数作为程序的模块单位,便于实现程序的模块化。C 语言是完全模块化和结构化的语言。

(5) 语法限制不太严格,程序设计自由度大。例如,对数组下标越界不进行检查,由程序编写者自己保证程序的正确。对变量的类型使用比较灵活,例如,整型量与字符型数据以及逻辑型数据可以通用。一般的高级语言语法检查比较严,能检查出几乎所有的语法错误,而 C 语言允许程序编写者有较大的自由度,因此放宽了语法检查。程序员应当仔细检查程序,保证其正确,而不要过分依赖 C 语言编译程序查错。“限制”与“灵活”是一对矛盾。限制严格,就失去灵活性;而强调灵活,就必然放松限制。对于不熟练的人员,编一个正确的 C 语言程序可能会比编一个其他高级语言程序难一些。也就是说,对用 C 语言的人,要求更高一些。

(6) C 语言允许直接访问物理地址,能进行位(bit)操作,能实现汇编语言的大部分功能,可以直接对硬件进行操作。因此 C 语言既具有高级语言的功能,又具有低级语言的许多功能,可用来编写系统软件。C 语言的这种双重性,使它既是成功的系统描述语言,又是通用的程序设计语言。

(7) 用 C 语言编写的程序可移植性好。由于 C 的编译系统相当简洁,因此很容易移植到新的系统。而且 C 编译系统在新的系统上运行时,可以直接编译“标准链接库”中的大部分功能,不需要修改源代码,因为标准链接库是用可移植的 C 语言写的。因此,几乎在所有的计算机系统中都可以使用 C 语言。

(8) 生成目标代码质量高,程序执行效率高。

# 简单的 c 语言程序举例

例 1.1 要求在屏幕上输出以下一行信息。

This is a C program.

解题思路：在主函数中用 printf 函数原样输出以上文字。

编写程序：

```
# include <stdio.h> //这是编译预处理指令
int main() //定义主函数
{
    printf ("This is a C program.\n"); //输出所指定的一行信息
    return 0; //函数执行完毕时返回函数值 0
}
```

运行结果：

```
This is a C program.
Press any key to continue...
```

程序分析：先看第 2 行，其中 main 是函数的名字，表示“主函数”，main 前面的 int 表示此函数的类型是 int 类型(整型)。在执行主函数后会得到一个值(即函数值)，其值为整型。程序第 5 行“return 0；”的作用是：当 main 函数执行结束前将整数 0 作为函数值，返回到调用函数处<sup>①</sup>。每一个 C 语言程序都必须有一个 main 函数。函数体由花括号()括起来。本例中主函数内有两个语句，程序第 4 行是一个输出语句，printf 是 C 编译系统提供的函数库中的输出函数(详见第 4 章)。printf 函数中双撇号内的字符串“This is a C program.”按原样输出。\\n 是换行符，即在输出“This is a C program.”后，显示屏上的光标位置移到下一行的开头。这个光标位置称为输出的当前位置，即下一个输出的字符出现在此位置上。每个语句最后都有一个分号，表示语句结束。

在使用函数库中的输入输出函数时，编译系统要求程序提供有关此函数的信息(例如对这些输入输出函数的声明和宏的定义、全局量的定义等)，程序第 1 行“# include <stdio.h>”

<sup>①</sup> C 99 建议把 main 函数指定为 int 型(整型)，它要求函数带回一个整数值。在 main 函数中，在执行的最后设置一个“return 0；”语句。当主函数正常结束时，得到的函数值为 0，当执行 main 函数过程中出现异常或错误时，函数值为一个非 0 的整数。这个函数值是返回给调用 main 函数的操作系统的。程序员可以利用操作指令检查 main 函数的返回值，从而判断 main 函数是否已正常执行，并据此决定以后的操作。如果在程序中不写“return 0；”语句，有的 C 编译系统会在目标程序中自动加上这一语句，因此。主函数正常结束时，也能使函数值为 0。为使程序规范和可移植性，希望读者写的程序一律将 main 函数指定为 int 型，并在 main 函数的最后加一个“return 0；”语句。

的作用就是用来提供这些信息的。stdio.h 是系统提供的一个文件名，stdio 是“standard input & output”的缩写，文件后缀.h 的意思是头文件(header file)，因为这些文件都是放在程序各文件模块的开头的。输入输出函数的相关信息已事先放在 stdio.h 文件中。现在，用 #include 指令把这些信息调入供使用。对编译预处理指令 #include，在此读者可先不必深究，只要记住：在程序中如要用到标准函数库中的输入输出函数，应该在本文件模块的开头写上下面一行：

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

在以上程序各行的右侧，如果有//，则表示从到本行结束是“注释”，用来对程序有关部分进行必要的说明。在写 C 程序时应当多用注释，以方便自己和别人理解程序各部分的作用。在程序进行预编译处理时将每个注释替换为一个空格，因此在编译时注释部分不产生目标代码，注释对运行不起作用。注释只是给人看的，而不是让计算机执行的。

说明：C 语言允许用两种注释方式：

(1) 以//开始的单行注释。如上介绍的注释。这种注释可以单独占一行，也可以出现在一行中其他内容的右侧。此种注释的范围从//开始，以换行符结束。也就是说这种注释不能跨行。如果注释内容一行内写不下，可以用多个单行注释，如下面两行是连续的注释行：

```
//如注释内容一行内写不下  
//可以在下一行重新用“//”，然后继续写注释。
```

(2) 以/\* 开始，以 \*/ 结束的块式注释。这种注释可以包含多行内容。它可以单独占一行(在行开头以/\* 开始，行末以 \*/结束)，也可以包含多行。编译系统在发现一个/\* 后，会开始找注释结束符 \*/，把二者间的内容作为注释。

但应注意的是在字符串中的//和/\* 都不作为注释的开始。而是作为字符串的一部分。如：

```
# include <stdio.h>  
//主函数  
int main() //定义主函数  
{ //主函数体开始  
    • 8 •
```

www.TopSage.com

```
int max(int x,int y); //对被调用函数 max 的声明  
int a,b,c; //定义变量 a,b,c  
scanf("%d,%d",&a,&b); //输入变量 a 和 b 的值  
c=max(a,b); //调用 max 函数,将得到的值赋给 c  
printf("max=%d\n",c); //输出 c 的值  
return 0; //返回函数值为 0  
} //主函数体结束  
  
//求两个整数中的较大者的 max 函数  
int max(int x,int y) //定义 max 函数,函数值为整型,形式参数 x 和 y 为整型  
{  
    int z; //max 函数中的声明部分,定义本函数中用到的变量 z 为整型  
    if(x>y)z=x; //若 x>y 成立,将 x 的值赋给变量 z  
    else z=y; //否则(即 x>y 不成立),将 y 的值赋给变量 z  
    return(z); //将 z 的值作为 max 函数值,返回到调用 max 函数的位置  
}
```

**程序分析：**本程序包括两个函数：①主函数 main；②被调用的函数 max。

max 函数的作用是将 x 和 y 中较大的值赋给变量 z。第 16 行 return 语句将 z 的值作为 max 的函数值，返回给调用 max 函数的函数（即主函数 main）。返回值是通过函数名 max 带回到 main 函数中去的（带回到程序第 7 行，main 函数调用 max 函数处）。

程序第 4 行是对被调用函数 max 的声明（declaration）。为什么要作这个函数声明呢？因为在主函数中要调用 max 函数（主函数中第 8 行“c=max(a,b);”），而 max 函数的定义却在 main 函数之后，对程序的编译是自上而下进行的，在对程序第 7 行进行编译时，编译系统无法知道 max 是什么，因而无法把它作为函数调用处理。为了使编译系统能识别 max 函数，就要在调用 max 函数之前用“int max(int x,int y);”对 max 函数进行“声明”，所谓声明，通俗地说就是告诉编译系统 max 是什么，以及它的有关信息。有关函数的声明详见第 7 章。

程序第 6 行 scanf 是输入函数的名字（scanf 和 printf 都是 C 的标准输入输出函数）。该 scanf 函数的作用是输入变量 a 和 b 的值。scanf 后面圆括号中包括两部分内容：一是双撇号中的内容，它指定输入的数据按什么格式输入。“%d”的含义是十进制整数形式。二是输入的数据准备放到哪里，即赋给哪个变量。现在，scanf 函数中指定的是 a 和 b，在 a 和 b 的前面各有一个 &，在 C 语言中“&”是地址符，&a 的含义是“变量 a 的地址”，&b 是“变量 b 的地址”。执行 scanf 函数，从键盘读入两个整数，送到变量 a 和 b 的地址处，然后把这两个整数分别赋给变量 a 和 b。

程序第 7 行用 max(a,b) 调用 max 函数。在调用时将 a 和 b 作为 max 函数的参数（称为实际参数）的值分别传送给 max 函数中的参数 x 和 y（称为形式参数），然后执行 max 函数的函数体（程序第 12~17 行），使 max 函数中的变量 z 得到一个值（即 x 和 y 中大者的值），return(z) 的作用是把 z 的值作为 max 函数值带回到程序第 7 行 = 的右侧（主函数调用 max 函数的位置），取代 max(a,b)，然后把这个值赋给变量 c。

第 8 行输出结果。在执行 printf 函数时，对双撇号括起来的 max=%d\n 是这样处理的：将 max= 原样输出，%d 由变量 c 的值取代之，\n 执行换行。

**注意：**本例程序中两个函数，都有 return 语句，请注意它们的异同。两个函数都定义为整型，都有函数值，都需要用 return 语句为函数指定返回值。但是 main 函数中的 return 语句指定的返回值一般为 0，而 max 函数的返回值是 max 函数中求出的二数中的最大值 z，只有通过 return 语句才能把求出的 z 值作为函数的值并返回调用它的位置上（即 main 函数，程序第 7 行）。不要以为在 max 函数中求出最大值 z 后就会自动地作为函数值返回调用处，必须用 return 语句指定将哪个值作为函数值。也不要不加分析地在所有函数的最后都写上“return 0;”。

# C 语言程序的结构

(1) 一个程序由一个或多个源程序文件组成。一个规模较小的程序,往往只包括一个源程序文件,如例 1.1 和例 1.2 是一个源程序文件中只有一个函数(main 函数),例 1.3 中有两个函数,属于同一个源程序文件。在一个源程序文件中可以包括 3 个部分:

① **预处理指令**。如`#include <stdio.h>`(还有一些其他预处理指令,如`#define` 等)。C 编译系统在对源程序进行“翻译”以前,先由一个“预处理器”(也称“预处理程序”、“预编译器”)对预处理指令进行预处理,对于`#include <stdio.h>`指令来说,就是将 stdio.h 头文件的内容读进来,放在`#include` 指令行,取代了`#include <stdio.h>`。由预处理得到的结果与程序其他部分一起,组成一个完整的、可以用来编译的最后的源程序,然后由编译程序对该源程序正式进行编译,才得到目标程序。

② **全局声明**。即在函数之外进行的数据声明。例如可以把例 1.2 程序中的“`int a, b, sum;`”放到 main 函数的前面,这就是全局声明,在函数外面声明的变量称为全局变量。如果是在程序开头(定义函数之前)声明的变量,则在整个源程序文件范围内有效。在函数中声明的变量是局部变量,只在函数范围内有效。关于全局变量和局部变量的概念和用法见本书第 7 章。在本章的例题中没有用全局声明,只有在函数中定义的局部变量。

③ **函数定义**。如例 1.1、例 1.2 和例 1.3 中的 main 函数和例 1.3 中的 max 函数,每个函数用来实现一定的功能。在调用这些函数时,会完成函数定义中指定的功能。

(2) **函数是 C 程序的主要组成部分**。程序的几乎全部工作都是由各个函数分别完成的,函数是 C 程序的基本单位,在设计良好的程序中,每个函数都用来实现一个或几个特定的功能。编写 C 程序的工作主要就是编写一个个函数。

一个 C 语言程序是由一个或多个函数组成的,其中必须包含一个 main 函数(且只能有一个 main 函数)。例 1.1 和例 1.2 中的程序只由一个 main 函数组成,例 1.3 程序由一个 main 函数和一个 max 函数组成,它们组成一个源程序文件,在进行编译时对整个源程序文件统一进行编译。

一个小程序只包含一个源程序文件,在一个源程序文件中包含若干个函数(其中有一个

• 10 •

[www.TopSage.com](http://www.TopSage.com)

main 函数)。当程序规模较大时,所包含的函数的数量较多,如果把所有的函数都放在同一个源程序文件中,则此文件显得太大,不便于编译和调试。为了便于调试和管理,可以使一个程序包含若干个源程序文件,每个源程序文件又包含若干个函数。一个源程序文件就是一个程序模块,即将一个程序分成若干个程序模块。

在进行编译时是以源程序文件为对象进行的。在分别对各源程序文件进行编译并得到相应的目标程序后,再将这些目标程序连接成为一个统一的二进制的可执行程序。

C 语言的这种特点使得容易实现程序的模块化。

在程序中被调用的函数,可以是系统提供的库函数(例如 printf 和 scanf 函数),也可以是用户自己编写设计的函数(例如例 1.3 中的 max 函数)。C 的函数库十分丰富。

### (3) 一个函数包括两个部分。

① 函数首部。即函数的第 1 行，包括函数名、函数类型、函数属性、函数参数(形式参数)名、参数类型。

例如，例 1.3 中的 max 函数的首部为

int	max	(int	x,	int	y)
↓	↓	↓	↓	↓	↓
函数类型	函数名	函数参数类型	函数参数名	函数参数类型	函数参数名

一个函数名后面必须跟一对圆括号，括号内写函数的参数名及其类型。如果函数没有参数，可以在括号中写 void，也可以是空括号，如：

```
int main(void)
```

或

```
int main()
```

② 函数体。即函数首部下面的花括号内的部分。如果在一个函数中包括有多层花括号，则最外层的一对花括号是函数体的范围。

函数体一般包括以下两部分。

- **声明部分**。声明部分包括：定义在本函数中所用到的变量，如例 1.3 中在 main 函数中定义变量“int a,b,c;”；对本函数所调用函数进行声明，如例 1.3 中在 main 函数中对 max 函数的声明“int max(int x,int y); ”。
- **执行部分**。由若干个语句组成，指定在函数中所进行的操作。

在某些情况下也可以没有声明部分(例如例 1.1)，甚至可以既无声明部分也无执行部分。如：

```
void dump()  
{}
```

(4) 程序总是从 main 函数开始执行的,而不论 main 函数在整个程序中的位置如何

• 11 •

www.TopSage.com

(main 函数可以放在程序最前头,也可以放在程序最后,或在一些函数之前、另一些函数之后)。

(5) 程序中对计算机的操作是由函数中的 C 语句完成的。如赋值、输入输出数据的操作都是由相应的 C 语句实现的。C 程序书写格式是比较自由的。一行内可以写几个语句,一个语句可以分写在多行上,但为清晰起见,习惯上每行只写一个语句。

(6) 在每个数据声明和语句的最后必须有一个分号。分号是 C 语句的必要组成部分。如

c=a+b;

其中的分号是不可缺少的。

(7) C 语言本身不提供输入输出语句。输入和输出的操作是由库函数 scanf 和 printf 等函数来完成的。C 对输入输出实行“函数化”。由于输入输出操作涉及具体的计算机设备,把输入输出操作用库函数实现,就可以使 C 语言本身的规模较小,编译程序简单,很容易在各种机器上实现,程序具有可移植性。

(8) 程序应当包含注释。一个好的、有使用价值的源程序都应当加上必要的注释,以增加程序的可读性。

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)

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

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)。

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。

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。**

# 运行 C 程序的步骤与方法

(1) 上机输入和编辑源程序。通过键盘向计算机输入程序,如发现有错误,要及时改正。最后将此源程序以文件形式存放在自己指定的文件夹内(如果不特别指定,一般存放在用户当前目录下),文件用.c作为后缀,生成源程序文件,如 f.c。

(2) 对源程序进行编译,先用 C 编译系统提供的“预处理器”(又称“预处理程序”或“预编译器”)对程序中的预处理指令进行编译预处理。例如,对于 #include <stdio.h> 指令来说,就是将 stdio.h 头文件的内容读进来,取代 #include <stdio.h> 行。由预处理得到的信息与程序其他部分一起,组成一个完整的、可以用来进行正式编译的源程序,然后由编译系统对该源程序进行编译。

编译的作用首先是对源程序进行检查,判定它有无语法方面的错误,如有,则发出“出错信息”,告诉编程人员认真检查改正。修改程序后重新进行编译,如有错,再发出“出错信息”。如此反复进行,直到没有语法错误为止。这时,编译程序自动把源程序转换为二进制形式的目标程序(在 Visual C++ 中后缀为.obj,如 f.obj)。如果不特别指定,此目标程序一般也存放在用户当前目录下,此时源文件没有消失。

在用编译系统对源程序进行编译时,自动包括了预编译和正式编译两个阶段,一气呵成。用户不必分别发出二次指令。

(3) 进行连接处理。经过编译所得到的二进制目标文件(后缀为.obj)还不能供计算机直接执行。前面已说明:一个程序可能包含若干个源程序文件,而编译是以源程序文件为对象的,一次编译只能得到与一个源程序文件相对应的目标文件(也称目标模块),它只是整个程序的一部分。必须把所有的编译后得到的目标模块连接装配起来,再与函数库相连接成一个整体,生成一个可供计算机执行的目标程序,称为可执行程序(executive program),在 Visual C++ 中其后缀为.exe,如 f.exe。

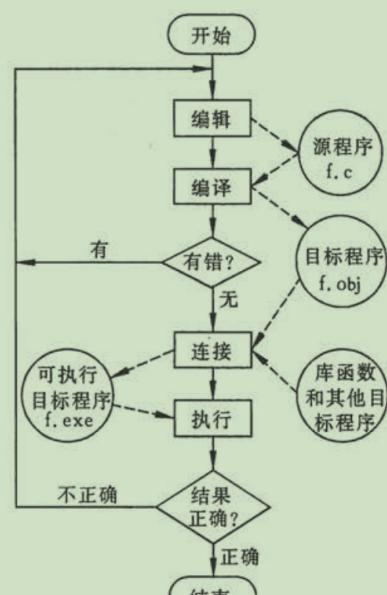
即使一个程序只包含一个源程序文件,编译后得到的目标程序也不能直接运行,也要经过连接阶段,因为要与函数库进行连接,才能生成可执行程序。

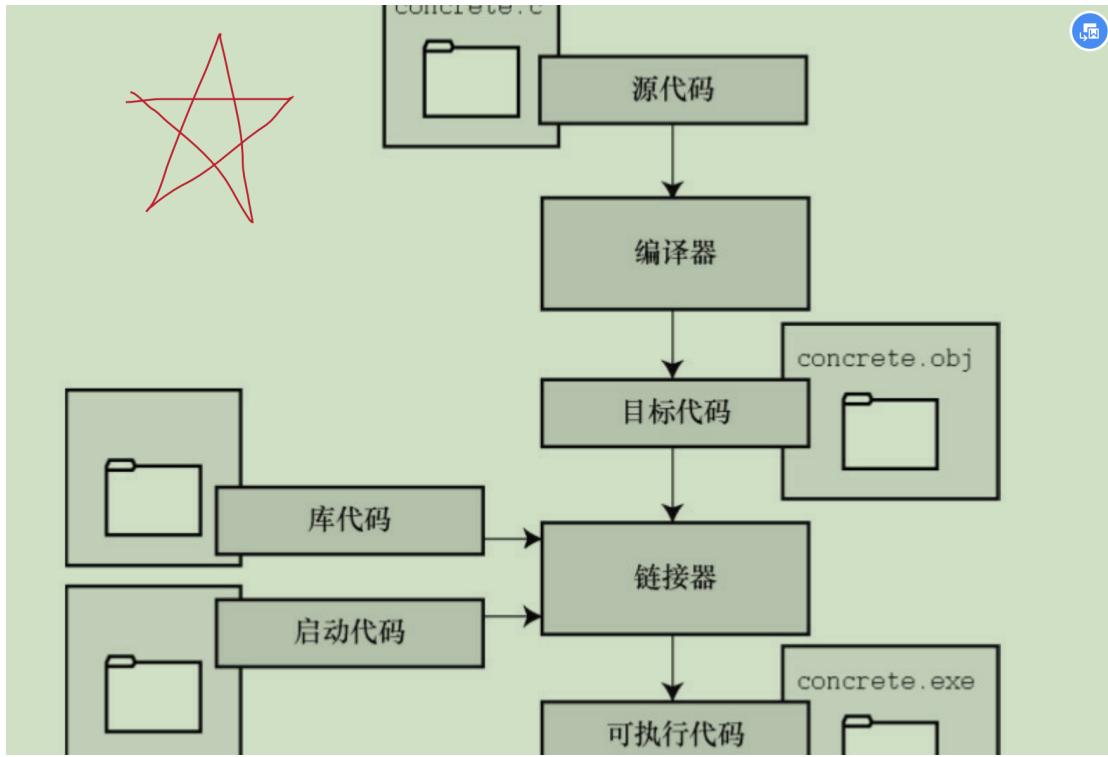
以上连接的工作是由一个称为“连接编辑程序(linkage editor)”的软件来实现的。

(4) 运行可执行程序,得到运行结果。

以上过程如图 1.2 所示。其中实线表示操作流程,虚线表示文件的输入输出。例如,编辑后得到一个源程序文件 f.c,然后在进行编译时再将源程序文件 f.c 输入,经过编译得到目标程序文件 f.obj,再将所有目标模块输入计算机,与系统提供的库函数等进行连接,得到可执行的目标程序 f.exe,最后把 f.exe 输入计算机,并使之运行,得到结果。

一个程序从编写到运行成功,并不是一次成功的。





## 程序设计的任务

(1) **问题分析**。对于接手的任务要进行认真的分析,研究所给定的条件,分析最后应达到的目标,找出解决问题的规律,选择解题的方法。在此过程中可以忽略一些次要的因素,使问题抽象化,例如用数学式子表示问题的内在特性。这就是建立模型。

(2) **设计算法**。即设计出解题的方法和具体步骤。例如要解一个方程式,就要选择用什么方法求解,并且把求解的每一个步骤清晰无误地写出来。一般用流程图来表示解题的步骤。

(3) **编写程序**。根据得到的算法,用一种高级语言编写出源程序。

(4) **对源程序进行编辑、编译和连接**,得到可执行程序。

(5) **运行程序,分析结果**。运行可执行程序,得到运行结果。能得到运行结果并不意味着程序正确,要对结果进行分析,看它是否合理。例如把“ $b=a;$ ”错写为“ $a=b;$ ”,程序不存在语法错误,能通过编译,但运行结果显然与预期不符。因此要对程序进行调试(debug)。调试的过程就是通过上机发现和排除程序中故障的过程。经过调试,得到了正确的结果,但是工作不应到此结束。不要只看到某一次结果是正确的,就认为程序没有问题。例如,求  $c=b/a$ ,当  $a=4, b=2$  时,求出  $c$  的值为 0.5,是正确的,但是当  $a=0, b=2$  时,就无法求出  $c$  的值。说明程序对某些数据能得到正确结果,对另外一些数据却得不到正确结果,程序还有漏洞,因此,还要对程序进行测试(test)。所谓测试,就是设计多组测试数据,检查程序对不同数据的运行情况,从中尽量发现程序中存在的漏洞,并修改程序,使之能适用于各种情况,作为商品提供使用的程序,是必须经过严格测试的。

在本书的配套书《C 程序设计(第四版)学习辅导》中对程序的调试和测试做了进一步的说明,读者可以参考。

(6) **编写程序文档**。许多程序是提供给别人使用的,如同正式的产品应当提供产品说明书一样,正式提供给用户使用的程序,必须向用户提供程序说明书(也称为用户文档)。内容应包括:程序名称、程序功能、运行环境、程序的装入和启动、需要输入的数据,以及使用

# 算法——程序的灵魂

## 什么是算法

计算机算法可分为两大类别：数值运算法和非数值运算法。数值运算的目的是求数值解，例如求方程的根、求一个函数的定积分等，都属于数值运算范围。非数值运算包括的面十分广泛，最常见的是用于事务管理领域，例如对一批职工按姓名排序、图书检索、人事管理和行车调度管理等。目前，计算机在非数值运算方面的应用远远超过了在数值运算方面的应用。

由于数值运算往往有现成的模型，可以运用数值分析方法，因此对数值运算的算法的研究比较深入，算法比较成熟。对各种数值运算都有比较成熟的算法可供选用。人们常常把这些算法汇编成册（写成程序形式），或者将这些程序存放在磁盘或光盘上，供用户调用。例如有的计算机系统提供“数学程序库”，使用起来十分方便。

非数值运算的种类繁多，要求各异，难以做到全部都有现成的答案，因此只有一些典型的非数值运算法（例如排序算法、查找搜索算法等）有现成的、成熟的算法可供使用。许多问题往往需要使用者参考已有的类似算法的思路，重新设计解决特定问题的专门算法。本

## 算法的特性

(1) **有穷性**。一个算法应包含有限的操作步骤，而不能是无限的。例如例 2.4 的算法，如果将 S8 步骤改为：“若  $deno > 0$ ，返回 S4”，则循环永远不会停止，这不是有穷的步骤。事实上，“有穷性”往往指“在合理的范围之内”。如果让计算机执行一个历时 1000 年才结束的算法，这虽然是有穷的，但超过了合理的限度，人们也不把它视为有效算法。究竟什么算“合理限度”，由人们的常识和需要判定。

(2) **确定性**。算法中的每一个步骤都应当是确定的，而不应当是含糊的、模棱两可的。例如，有一个健身操的动作要领，其中有一个动作：“手举过头顶”，这个步骤就是不确定的，含糊的。是双手都举过头？还是左手或右手？举过头顶多少厘米？不同的人可以有不同的理解。算法中的每一个步骤应当不致被解释成不同的含义，而应是明确无误的。如例 2.5 中的 S3 步骤如果写成“ $n$  被一个整数除，得余数  $r$ ”，这也是“不确定”的，它没有说明  $n$  被哪个整数除，因此无法执行。也就是说，算法的含义应当是唯一的，而不应当产生“歧义性”。所谓“歧义性”，是指可以被理解为两种（或多种）的可能含义。

(3) **有零个或多个输入**。所谓输入是指在执行算法时需要从外界取得必要的信息。例如，在执行例 2.5 算法时，需要输入  $n$  的值，然后判断  $n$  是否素数。也可以有两个或多个输入，例如，求两个整数  $m$  和  $n$  的最大公约数，则需要输入  $m$  和  $n$  的值。一个算法也可以没有输入，例如，例 2.1 在执行算法时不需要输入任何信息，就能求出  $5!$ 。

(4) 有一个或多个输出。算法的目的是为了求解，“解”就是输出。如例 2.5 求素数的算法，最后输出的“是素数”或“不是素数”就是输出的信息。但算法的输出并不一定就是计算机的打印输出或屏幕输出，一个算法得到的结果就是算法的输出。没有输出的算法是没有意义的。

(5) 有效性。算法中的每一个步骤都应当能有效地执行，并得到确定的结果。例如，若  $b=0$ ，则执行  $a/b$  是不能有效执行的。

对于一般最终用户来说，他们并不需要在处理每一个问题时都要自己设计算法和编写程序，可以使用别人已设计好的现成算法和程序，只须根据已知算法的要求给予必要的输入，就能得到输出的结果。对使用者来说，算法如同一个“黑箱子”一样，他们可以不了解“黑箱子”中的结构，只是从外部特性上了解算法的作用，即可方便地使用算法。例如，对一个“输入 3 个数，求其中最大值”的算法，可以用图 2.2 表示，只要输入  $a, b, c$  这 3 个数，执行算法后就能得到其中最大的数。

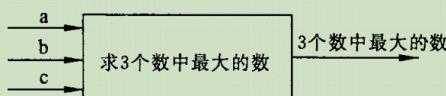


图 2.2

## 三种基本结构和算法表示方式

(1) 顺序结构。如图 2.14 所示，虚线框内是一个顺序结构。其中 A 和 B 两个框是顺序执行的。即：在执行完 A 框所指定的操作后，必然接着执行 B 框所指定的操作。顺序结构是最简单的一种基本结构。

(2) 选择结构。选择结构又称选取结构或分支结构，如图 2.15 所示。虚线框内是一个选择结构。此结构中必包含一个判断框。根据给定的条件  $p$  是否成立而选择执行 A 框或 B 框。例如  $p$  条件可以是  $x \geq 0$  或  $x > y, a + b < c + d$  等。

(3) 循环结构。又称重复结构，即反复执行某一部分的操作。有两类循环结构。

① 当型(while 型)循环结构。当型循环结构如图 2.17(a)所示。它的作用是：当给定的条件  $p_1$  成立时，执行 A 框操作，执行完 A 后，再判断条件  $p_1$  是否成立，如果仍然成立，再执行 A 框，如此反复执行 A 框，直到某一次  $p_1$  条件不成立为止，此时不执行 A 框，而从 b 点脱离循环结构。

② 直到型(until 型)循环结构。直到型循环结构如图 2.17(b)所示。它的作用是：先执行 A 框，然后判断给定的  $p_2$  条件是否成立，如果  $p_2$  条件不成立，则再执行 A，然后再对  $p_2$  条件作判断，如果  $p_2$  条件仍然不成立，又执行 A……如此反复执行 A，直到给定的  $p_2$  条件成立为止，此时不再执行 A，从 b 点脱离本循环结构。

NS 流程图

N-S 流程图用以下的流程图符号。

(1) 顺序结构。顺序结构用图 2.24 形式表示。A 和 B 两个框组成一个顺序结构。

(2) 选择结构。选择结构用图 2.25 表示。它与图 2.15 所表示的意思是相同的。当条件成立时执行 A 操作, p 不成立则执行 B 操作。注意: 图 2.25 是一个整体, 代表一个基本结构。

(3) 循环结构。当型循环结构用图 2.26 形式表示, 当  $p_1$  条件成立时反复执行 A 操作, 直到  $p_1$  条件不成立为止。

直到型循环结构用图 2.27 形式表示。



图 2.24



图 2.25

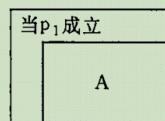


图 2.26

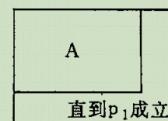
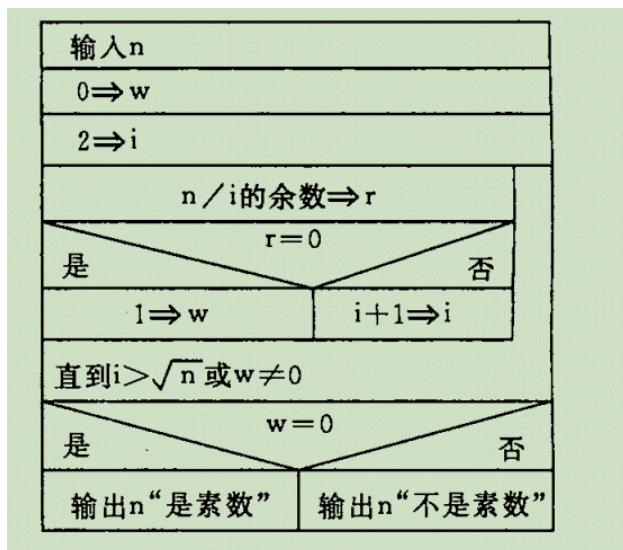
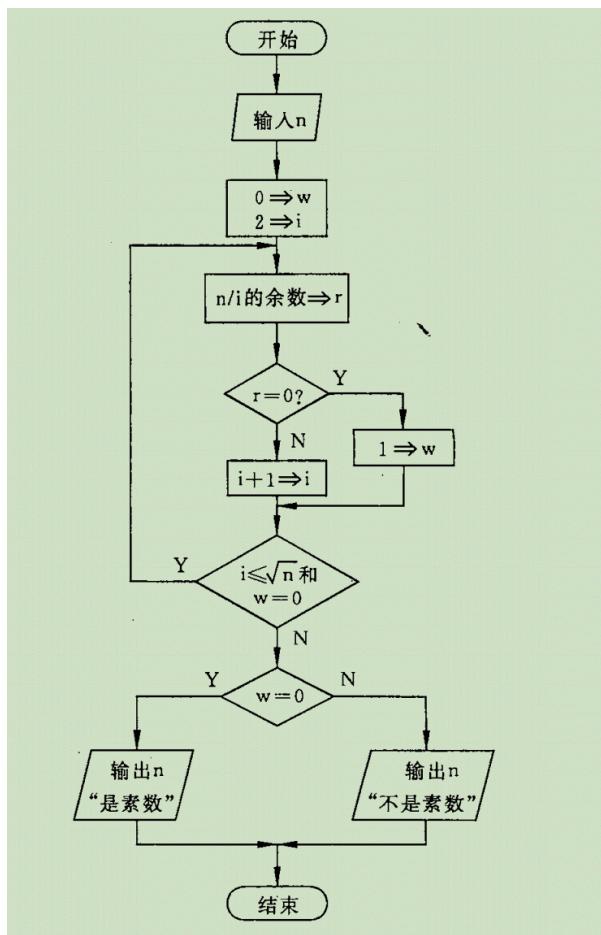


图 2.27

在初学时, 为清楚起见, 可如图 2.26 和图 2.27 那样, 写明“当  $p_1$ ”或“直到  $p_2$ ”, 待熟练之后, 可以不写“当”和“直到”字样, 只写“ $p_1$ ”和“ $p_2$ ”。从图的形状即可知道是当型还是直到型。

#### 实例：判断素数

在例 2.10 中用传统流程图(图 2.12)。可以看出, 图 2.12 不是由 3 种基本结构组成的。图中间的循环部分有两个出口(一个从第 1 个判断框右面出口, 另一个在第 2 个判断框下边出口), 不符合基本结构的特点。由于不能分解为 3 种基本结构, 就无法直接用 N-S 流程图的 3 种基本结构的符号来表示。因此, 应当先对图 2.12 作必要的变换。要将第 1 个判断框(“ $r=0?$ ”)的两个出口汇合在一点, 以解决两个出口问题。当  $r=0$  时意味着  $n$  为非素数, 但此时不马上输出  $n$ “不是素数”的信息, 而只使标志变量  $w$  的值由 0 改为 1( $w$  的值为  $w=0$ )。如果  $r \neq 0$ , 则保持  $w=0$ , 见图 2.34。



前者循环结构是当型后者是直到，条件互否  
伪代码

用传统的流程图和 N-S 图表示算法直观易懂,但画起来比较费事,在设计一个算法时,可能要反复修改,而修改流程图是比较麻烦的。因此,流程图适于表示一个算法,但在设计算法过程中使用不是很理想(尤其是当算法比较复杂、需要反复修改时)。为了设计算法时方便,常用一种称为伪代码(pseudo code)的工具。

**例 2.17** 求  $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{99} - \frac{1}{100}$ 。

用伪代码表示的算法如下：

```
begin
    1⇒sum
    2⇒deno
    1⇒sign
    while deno≤100
    {
        (-1) * sign⇒sign
        sign * 1/deno⇒term
        sum + term⇒sum
        deno+1⇒deno
    }
    print sum
end
```

## 结构化程序设计方法

具体说，采取以下方法来保证得到结构化的程序：

(1) 自顶向下；(2) 逐步细化；(3) 模块化设计；(4) 结构化编码。

应当掌握自顶向下、逐步细化的设计方法。这种设计方法的过程是将问题求解由抽象逐步具体化的过程。如图 2·3·6 所示，最开始拿到的题目是作“工作报告”，这是一个很笼统而抽象的任务，经过初步考虑之后把它分成 4 个大的部分。这就比刚才具体一些了，但还不够具体。这一步只是粗略地划分，称为“顶层设计”。然后一步一步细化，依次称为第 2 层、第 3 层设计，直到不需要细分为止。

用这种方法便于验证算法的正确性，在向下一层展开之前应仔细检查本层设计是否正确，只有上一层是正确的才能向下细化。如果每一层设计都没有问题，则整个算法就是正确的。由于每一层向下细化时都不太复杂，因此容易保证整个算法的正确性。检查时也是由上而下逐层检查，这样做，思路清楚，有条不紊地一步一步地进行，既严谨又方便。

在程序设计中常采用模块设计的方法，尤其当程序比较复杂时，更有必要。在拿到一个程序模块（实际上是程序模块的任务书）以后，根据程序模块的功能将它划分为若干个子模块，如果这些子模块的规模还嫌大，可以再划分为更小的模块。这个过程采用自顶向下的方法来实现。

程序中的子模块在 C 语言中通常用函数来实现（有关函数的概念将在第 7 章中介绍）。程序中的子模块一般不超过 50 行，即把它打印输出时不超过一页，这样的规模便于组织，也便于阅读。划分子模块时应注意模块的独立性，即使用一个模块完成一项功能，耦合性愈少愈好。模块化设计的思想实际上是一种“分而治之”的思想，把一个大任务分为若干个子任务。

结构化程序设计方法用来解决人脑思维能力的局限性和被处理问题的复杂性之间的矛盾。在设计好一个结构化的算法之后，还要善于进行结构化编码（coding）。所谓编码就是将已设计好的算法用计算机语言来表示，即根据已经细化的算法正确地写出计算

机程序。结构化的语言（如 Pascal，C，Visual Basic 等）都有与 3 种基本结构对应的语句，进行结构化编程序是不困难的。

## 进制及其互相转换

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： $6+7=D$ 、 $18+BA=D2$ 、 $595+792=D27$ 、 $2F87+F8A=3F11$

当前位满 16，向高位进 1，然后再减去 16，就是当前位的和	进位得到的 1，运算时要加上。当前位满 16，向高位进 1，然后再减去 16，就是当前位的和	进位得到的 1，运算时要加上	
$  \begin{array}{r}  6 \\  + 7 \\  \hline  = D  \end{array}  $	$  \begin{array}{r}  1 \quad 8 \\  + B_1 \quad A \\  \hline  = D \quad 2  \end{array}  $	$  \begin{array}{r}  5 \quad 9 \quad 5 \\  + 7_1 \quad 9 \quad 2 \\  \hline  = D \quad 2 \quad 7  \end{array}  $	$  \begin{array}{r}  2 \quad F \quad 8 \quad 7 \\  + 0_1 \quad F_1 \quad 8_1 \quad A \\  \hline  = 3 \quad F \quad 1 \quad 1  \end{array}  $

图5：十六进制加法示意图

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

当前位不够减，向高位借 1，当做 16 使用	被低位借走 1 后，当前位就不够减了，还得再向高位借 1，并当做 16 使用	被低位借走的 1，运算时要减去	
当前位本来就不够减，还被低位借走 1，所以必须向高位借 1 了，并且借到后当做 16 使用			
$  \begin{array}{r}  D \\  - 3 \\  \hline  = A  \end{array}  $	$  \begin{array}{r}  5^1 \quad 2 \\  - 2 \quad F \\  \hline  = 2 \quad 3  \end{array}  $	$  \begin{array}{r}  E^1 \quad 0 \quad 7 \\  - 1 \quad 4 \quad 1 \\  \hline  = C \quad C \quad 6  \end{array}  $	$  \begin{array}{r}  7^1 \quad C^1 \quad A^1 \quad 0 \\  - 1 \quad C \quad B \quad 1 \\  \hline  = 5 \quad F \quad E \quad F  \end{array}  $

图6：十六进制减法示意图

# 其它转十

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：

$$53627 = 5 \times 8^4 + 3 \times 8^3 + 6 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 = 22423 \text{ (十进制)}$$

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：

$$9FA8C = 9 \times 16^4 + 15 \times 16^3 + 10 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 653964 \text{ (十进制)}$$

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：

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

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：

$$423.5176 = 4 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 + 5 \times 8^{-1} + 1 \times 8^{-2} + 7 \times 8^{-3} + 6 \times 8^{-4} = 275.65576171875 \text{ (十进制)}$$

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：

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

小数部分和整数部分相反，要从左往右看，第1位的位权为  $2^{-1}=1/2$ ，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  $2^{-m}$ 。

# 十转其它

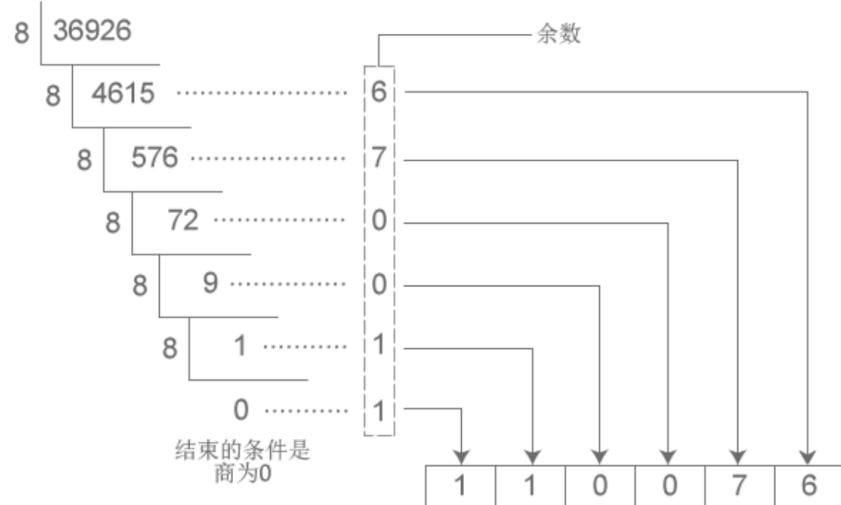
## 1) 整数部分

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：

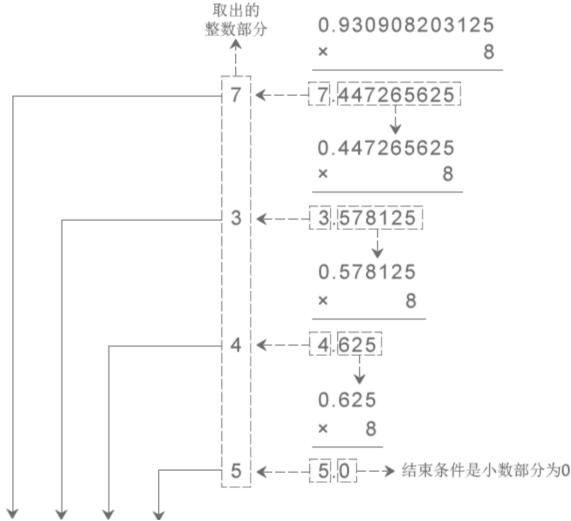


十进制小数转换成 N 进制小数采用“乘 N 取整，顺序排列”法。具体做法是：

- 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：



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

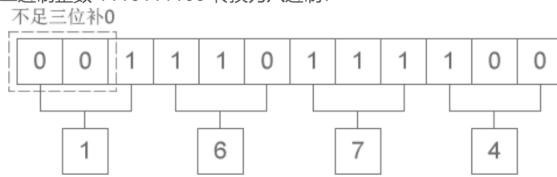
注意，十进制小数转换成其他进制小数时，结果有可能是一个无限位的小数。请看下面的例子：

- 十进制 0.51 对应的二进制为 0.10000101000111101011100001010001111010111...，是一个循环小数；
- 十进制 0.72 对应的二进制为 0.1011100001010001111010111000010100011110...，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。

## 二，八，十六间的转换（简便方法）

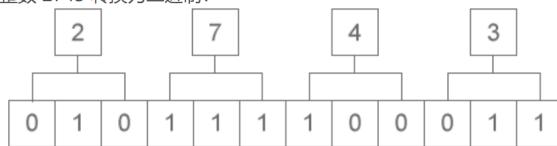
### 1) 二进制整数和八进制整数之间的转换

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：



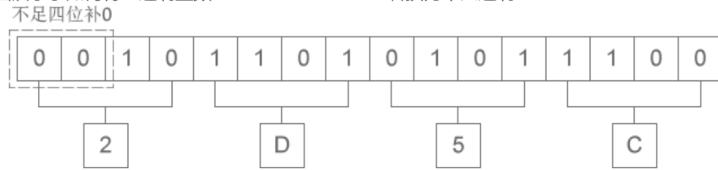
从图中可以看出，二进制整数 1110111100 转换为八进制的结果为 1674。

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：



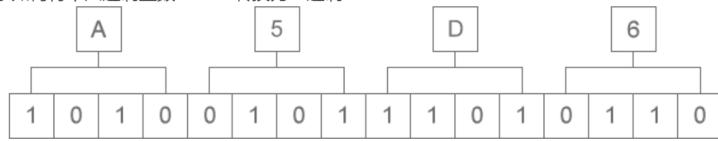
### 2) 二进制整数和十六进制整数之间的转换

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 10110101011100 转换为十六进制：



从图中可以看出，二进制整数 10110101011100 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：



## 二，八，十六进制的表示

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：

```
01. //合法的二进制
02. int a = 0b101; //换算成十进制为 5
03. int b = -0b110010; //换算成十进制为 -50
04. int c = 0B100001; //换算成十进制为 33
05.
06. //非法的二进制
07. int m = 101010; //无前缀 0B，相当于十进制
08. 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。

### 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：

```
01. //合法的八进制数
02. int a = 015; //换算成十进制为 13
03. int b = -0101; //换算成十进制为 -65
04. int c = 0177777; //换算成十进制为 65535
05.
06. //非法的八进制
07. int m = 256; //无前缀 0，相当于十进制
08. int n = 03A2; //A不是有效的八进制数字
```

[纯文本](#) [复制](#)

### 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：

```
01. //合法的十六进制
02. int a = 0X2A; //换算成十进制为 42
03. int b = -0XA0; //换算成十进制为 -160
04. int c = 0xffff; //换算成十进制为 65535
05.
06. //非法的十六进制
07. int m = 5A; //没有前缀 0X，是一个无效数字
08. int n = 0X3H; //H不是有效的十六进制数字
```

## 存储和编码相关

### 数据在内存中的存储（二进制形式存储）

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。

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：

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

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。

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

## ASCII 编码

American Standard Code for Information Interchange：美国信息交换标准代码  
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。

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。

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 ( $2^7 = 128$ )，所以还会空闲下一个比特位，它就被浪费了。

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，C 语言有时候使用 ASCII 编码，有时候却不是

## ASCII 码一览表及各字符解释

## 其它编码

随着计算机发展，各国已经不满足于单纯用 ASCII 码；  
对于我们来说能在计算机中显示中文字符是至关重要的，所以我们还需要一张关于中文和数字对应的关系表；

一个字节 8 位二进制，只能最多表示 256 个字符，要处理中文显然一个字节是不够的；  
所以我们需要采用两个字节来表示，而且还不能和 ASCII 编码冲突；

所以 1980 年中国制定了 GB2312 编码，国家简体中文字符集，兼容 ASCII；

1995 年制定了 GBK 编码，GB2312 的扩展字符集，支持繁体字，兼容 GB2312。

注：在 GBK 和 GB2312 中，一个中文字符占两个字节，16 个二进制位，4 个十六进制位。

输出汉字： `printf("%c%c",b[0],b[1]);` 两个字符一个字

如何兼容 ASCII：

如何区别连在一起的 2 个字节是代表 2 个英文字母，还是一个中文汉字呢？

如果 2 个字节连在一起，且每个字节的第 1 位(也就是相当于 128 的那个 2 进制位)如果是

**1，就代表这是个中文，这个首位是 128 的字节被称为高字节。也就是 2 个高字节连在一起，必然就是一个中文。**

因为 0-127 已经表示了英文的绝大部分字符，128-255 是 ASCII 的扩展表，表示的都是极特殊的字符，一般没什么用。

所以 0-127 位 ASCII 码，GB2312 就直接拿来用了。

## 二、编码 Unicode

全世界有上百种语言，日本把日文编到 Shift\_JIS 里，韩国把韩文编到 Euc-kr 里；

各国有各国的标准，就会不可避免地出现冲突，结果就是，在多语言混合的文本中，显示出来会有乱码。

因此，1991 年国际标准组织统一标准字符集，编码 Unicode 应运而生。

最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要 4 个字节）

## 三、编码 UTF-8

如果统一成 Unicode 编码，乱码问题从此消失了；

但是，Unicode 编码最少用两个字节，ASCII 码中英文是一个字节；

如果文本基本上全部是英文，用 Unicode 编码需要多一倍存储空间，存储和传输十分费劲。

1992 年创建 UTF-8 编码，是一种针对 Unicode 的可变长度字符编码，又称万国码；

UTF-8 编码把一个 Unicode 字符根据不同的数字大小编码成 1-6 个字节（每 8 位缩减），从而兼容所有编码，

**英文字符 1 字节，欧洲字符 2 字节，中文字符 3 字节，只有很生僻的字符才会被编码成 4-6 个字节。**

# 环境配置和编译器基础

## 有关编译器的几个概念

### 1) 源文件 (Source File)

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）。

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。

源文件其实只是纯文本文件，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语言的肩膀上发展起来的，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 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) 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 VC 6.0 下，这叫做一个“工程”，而在 VS 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。

### 3) 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。

上节我们讲到，源代码经过编译(Compile)后就变成了可执行文件，其实这种说法有点笼统，甚至从严格意义上来说是错误的。源代码要经过编译(Compile)和链接(Link)两个过程才能变成可执行文件。

编译器一次只能编译一个源文件，如果当前程序包含了多个源文件，那么就需要编译多次。编译器每次编译的结果是产生一个中间文件（可以认为是一种临时文件），而不是最终的可执行文件。中间文件已经非常接近可执行文件了，它们都是二进制格式，内部结构也非常相似。

将当前程序的所有中间文件以及系统库（暂时可以理解为系统中的一些组件）组合在一起，才能形成最终的可执行文件，这个组合的过程就叫做**链接 (Link)**。完成链接功能的软件叫做**链接器 (Linker)**。

如果程序只包含了一个源文件，是不是就不需要链接了呢？不是的！

经过编译后程序虽然只有一个中间文件，不再需要和其它的中间文件组合了，但是这个唯一的中间文件还需要和系统库组合，这个过程也是链接。也就是说，不管有多少个源文件，都必须经过编译和链接两个过程才能生成可执行文件。

简单讲，编译器就是将“一种语言（通常为高级语言）”翻译为“另一种语言（通常为低级语言）”的程序。一个现代编译器的主要工作流程：源代码(source code) → 预处理器(preprocessor) → 编译器(compiler) → 目标代码(object code) → 链接器(Linker) → 可执行程序(executables)

高级计算机语言便于人编写，阅读交流，维护。机器语言是计算机能直接解读、运行的。编译器将汇编或高级计算机语言**源程序 (Source program)**作为输入，翻译成目标语言(**Target language**)机器代码的等价程序。源代码一般为高级语言(High-level language)，如Pascal、C、C++、Java、**汉语编程**等或**汇编语言**，而目标则是机器语言的**目标代码 (Object code)**，有时也称作机器代码(Machine code)。

对于C#、VB等高级语言而言，此时编译器完成的功能是把源码(SourceCode)编译成通用中间语言(MSIL/CIL)的字节码(ByteCode)。最后运行的时候通过通用语言运行库的转换，编程最终可以被CPU直接计算的机器码(NativeCode)。

# GCC 编译器简明教程（Linux 下 C 语言开发环境的搭建）

Linux 下使用最广泛的 C/C++ 编译器是 GCC，大多数的 Linux 发行版本都默认安装，不管是开发人员还是初学者，一般都将 GCC 作为 Linux 下首选的编译工具。本教程也毫不犹豫地使用 GCC 来编译 C 语言程序。

GCC 仅仅是一个编译器，没有界面，必须在命令行模式下使用。通过 `gcc` 命令就可以将源文件编译成可执行文件。

## 1) 生成可执行程序

最简单的生成可执行文件的写法为：

```
$ cd demo #进入源文件所在的目录  
$ gcc main.c #在 gcc 命令后面紧跟源文件名
```

打开 demo 目录，会看到多了一个名为 `a.out` 的文件，这就是最终生成的可执行文件，如下图所示：



`a.out`

这样就一次性完成了编译和链接的全部过程，非常方便。

注意：不像 Windows，Linux 不以文件后缀来区分可执行文件，Linux 下的可执行文件后缀理论上可以是任意的，这里的 `.out` 只是用来表明它是 GCC 的输出文件。不管源文件的名字是什么，GCC 生成的可执行文件的默认名字始终是 `a.out`。

如果不想使用默认的文件名，那么可以通过 `-o` 选项来自定义文件名，例如：

```
$ gcc main.c -o main.out
```

这样生成的可执行程序的名字就是 `main.out`。

因为 Linux 下可执行文件的后缀仅仅是一种形式上的，所以可执行文件也可以不带后缀，例如：

```
$ gcc main.c -o main
```

通过 `-o` 选项也可以将可执行文件输出到其他目录，并不一定非得在当前目录下，例如：

```
$ gcc main.c -o ./out/main.out
```

或者

```
$ gcc main.c -o out/main.out
```

表示将可执行文件输出到当前目录下的 `out` 目录，并命名为 `main.out`。`./` 表示当前目录，如果不写，默认也是当前目录。

注意：`out` 目录必须存在，如果不存在，`gcc` 命令不会自动创建，而是抛出一个错误。

## 2) 运行可执行程序

上面我们生成了可执行程序，那么该如何运行它呢？很简单，在控制台中输入程序的名字就可以，如下所示：

```
$ ./a.out
```

`./` 表示当前目录，整条命令的意思是运行当前目录下的 `a.out` 程序。如果不写 `./`，Linux 会到系统路径下查找 `a.out`，而系统路径下显然不存在这个程序，所以会运行失败。

所谓系统路径，就是环境变量指定的路径，我们可以通过修改环境变量添加自己的路径，或者删除某个路径。很多时候，一条 Linux 命令对应一个可执行程序，如果执行命令时没有指明路径，那么就会到系统路径下查找对应的程序。

上面讲解的是通过 `gcc` 命令一次性完成编译和链接的整个过程，这样最方便，大家在学习C语言的过程中一般都这么做。实际上，`gcc` 命令也可以将编译和链接分开，每次只完成一项任务。

## 1) 编译 (Compile)

将源文件编译成目标文件需要使用 `-c` 选项，例如：

```
gcc -c main.c
```

就将 `main.c` 编译为 `main.o`。打开 `demo` 目录，就会看到 `main.o`：



对于微软编译器（内嵌在 Visual C++ 或者 Visual Studio 中），目标文件的后缀为 `.obj`；对于 GCC 编译器，目标文件的后缀为 `.o`。

一个源文件会生成一个目标文件，多个源文件会生成多个目标文件，源文件数目和目标文件数目是一样的。通常情况下，默认的目标文件名字和源文件名字是一样的。

如果希望自定义目标文件的名字，那么可以使用 `-o` 选项，例如：

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

这样生成的目标文件的名字就是 `a.o`。

## 2) 链接 (Link)

在 `gcc` 命令后面紧跟目标文件的名字，就可以将目标文件链接成为可执行文件，例如：

```
gcc main.o
```

就将 `main.o` 链接为 `a.out`。打开 `demo` 目录，就会看到 `a.out`。

在 `gcc` 命令后面紧跟源文件名字或者目标文件名字都是可以的，`gcc` 命令能够自动识别到底是源文件还是目标文件：如果是源文件，那么要经过编译和链接两个步骤才能生成可执行文件；如果是目标文件，只需要链接就可以了。

使用 `-o` 选项仍然能够自定义可执行文件的名字，例如：

```
gcc main.o -o main.out
```

这样生成的可执行文件的名字就是 `main.out`。

# 数据类型

C 语言允许使用的类型见图 3.4, 图中有 \* 的是 C 99 所增加的。



其中基本类型（包括整型和浮点型）和枚举类型变量的值都是数值，统称为算术类型 (arithmetic type)。算术类型和指针类型统称为纯量类型 (scalar type)，因为其变量的值是以数字来表示的。枚举类型是程序中用户定义的整数类型。数组类型和结构体类型统称为组合类型 (aggregate type)，共用体类型不属于组合类型，因为在同一时间内只有一个成员具有值。函数类型用来定义函数，描述一个函数的接口，包括函数返回值的数据类型和参数的类型。

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。

## 整形数据

编译系统分配给 int 型数据 2 个字节或 4 个字节(由具体的 C 编译系统自行决定)。如 Turbo C 2.0 为每一个整型数据分配 2 个字节(16 个二进位)，而 Visual C++ 为每一个整型数据分配 4 个字节(32 位)。在存储单元中的存储方式是：用整数的补码(complement)形式存放。一个正数的补码是此数的二进制形式，如 5 的二进制形式是 101，如果用两个字节存放一个整数，则在存储单元中数据形式如图 3.5 所示。如果是一个负数，则应先求出负数的补码。求负数的补码的方法是：先将此数的绝对值写成二进制形式，然后对其后面所有各二进位按位取反，再加 1。如 -5 的补码见图 3.6。

5 的补码 

0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1

图 3.5

5 的原码 

0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1

 (a)

按位取反 

1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	0

 (b)

再加 1  
(-5 的补码) 

1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1

 (c)

图 3.6

在存放整数的存储单元中，最左面一位是用来表示符号的，如果该位为 0，表示数值为正；如果该位为 1，表示数值为负。

### 负数补码：取反加一

**说明：**如果给整型变量分配 2 个字节，则存储单元中能存放的最大值为 0111111111111111，第 1 位为 0 代表正数，后面 15 位为全 1，此数值是  $(2^{15}-1)$ ，即十进制数 32767。最小值为 1000000000000000，此数是  $-2^{15}$ ，即 -32768。因此一个整型变量的值的范围是 -32768~32767。超过此范围，就出现数值的“溢出”，输出的结果显然不正确。如果给整型变量分配 4 个字节(Visual C++)，其能容纳的数值范围为  $-2^{31} \sim (2^{31}-1)$ ，即 -2147483648~2147483647。

#### (2) 短整型(short int)

类型名为 short int 或 short。如用 Visual C++ 6.0，编译系统分配给 int 数据 4 个字节，短整型 2 个字节。存储方式与 int 型相同。一个短整型变量的值的范围是 -32768~32767。

#### (3) 长整型(long int)

类型名为 long int 或 long。一个 long int 型变量的值的范围是  $2^{31} \sim (2^{31}-1)$ ，即

#### (4) 双长整型(long long int)

类型名为 long long int 或 long long,一般分配 8 个字节。这是 C 99 新增的类型,但许多 C 编译系统尚未实现。

说明: C 标准没有具体规定各种类型数据所占用存储单元的长度,这是由各编译系统自行决定的。C 标准只要求 long 型数据长度不短于 int 型,short 型不长于 int 型。即

`sizeof(short)≤sizeof(int)≤sizeof(long)≤sizeof(long long)`

sizeof 是测量类型或变量长度的运算符。在 Turbo C 2.0 中,int 型和 short 型数据都是 2 个字节(16 位),而 long 型数据是 4 个字节(32 位)。在 Visual C++ 6.0 中,short 数据的长度为 2 字节,int 数据的长度为 4 字节,long 数据的长度为 4 字节。通常的做法是:把 long 定为 32 位,把 short 定为 16 位,而 int 可以是 16 位,也可以是 32 位。读者应了解所用系统的规定。在将一个程序从 A 系统移到 B 系统时,需要注意这个区别。例如,在 A 系统,整型数据占 4 个字节,程序中将整数 50000 赋给整型变量 price 是合法的、可行的。但在 B 系统,整型数据占 2 个字节,将整数 50000 赋给整型变量 price 就超过整型数据的范围,出现“溢出”。这时应当把 int 型变量改为 long 型,才能得到正确的结果。

表 3.2 整型数据常见的存储空间和值的范围

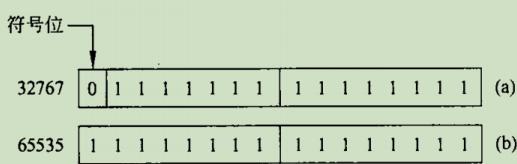
类 型	字节数	取 值 范 围
int(基本整型)	2	$-32768 \sim 32767$ , 即 $-2^{15} \sim (2^{15}-1)$
	4	$-2147483648 \sim 2147483647$ , 即 $-2^{31} \sim (2^{31}-1)$
unsigned int(无符号基本整型)	2	$0 \sim 65535$ , 即 $0 \sim (2^{16}-1)$
	4	$0 \sim 4294967295$ , 即 $0 \sim (2^{32}-1)$
short(短整型)	2	$-32768 \sim 32767$ , 即 $-2^{15} \sim (2^{15}-1)$
unsigned short(无符号短整型)	2	$0 \sim 65535$ , 即 $0 \sim (2^{16}-1)$
long(长整型)	4	$-2147483648 \sim 2147483647$ , 即 $-2^{31} \sim (2^{31}-1)$
unsigned long(无符号长整型)	4	$0 \sim 4294967295$ , 即 $0 \sim (2^{32}-1)$
long long(双长型)	8	$-9223372036854775808 \sim 9223372036854775807$ 即 $-2^{63} \sim (2^{63}-1)$
unsigned long long (无符号双长整型)	8	$0 \sim 18446744073709551615$ , 即 $0 \sim (2^{64}-1)$

### 十万以上尽量用 long

有符号整型数据存储单元中最高位代表符号(0 为正,1 为负)。如果指定 unsigned(为无符号)型,存储单元中全部二进位(b)都用作存放数值本身,而没有符号。无符号型变量只能存放不带符号的整数,如 123,4687 等,而不能存放负数,如 -123,-3。由于左面最高位不再用来表示符号,而用来表示数值,因此无符号整型变量中可以存放的正数的范围比一般整型变量中正数的范围扩大一倍。如果在程序中定义 a 和 b 两个短整型变量(占 2 个字节),其中 b 为无符号短整型:

```
short a; //a 为有符号短整型变量  
unsigned short b; //b 为无符号短整型变量
```

则变量 a 的数值范围为  $-32768 \sim 32767$ ,而变量 b 的数值范围为  $0 \sim 65535$ 。图 3.7(a)表示有符号整型变量 a 的最大值(32767),图 3.7(b)表示无符号整型变量 b 的最大值(65535)。



### 说明：

- (1) 只有整型(包括字符型)数据可以加 signed 或 unsigned 修饰符, 实型数据不能加。  
(2) 对无符号整型数据用“%u”格式输出。%u 表示用无符号十进制数的格式输出。如：

```
unsigned short price = 50; //定义 price 为无符号短整型变量  
printf("%u\n", price); //指定用无符号十进制数的格式输出
```

## 字符型数据

注意：字符'1'和整数 1 是不同的概念，字符'1'只是代表一个形状为'1'的符号，在需要时按原样输出，在内存中以 ASCII 码形式存储，占 1 个字节，见图 3.10(a)，而整数 1 是以整数存储方式(二进制补码方式)存储的，占 2 个或 4 个字节，见图 3.10(b)。

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

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

(a)

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

(b)

图 3.9

图 3.10

整数运算  $1+1$  等于整数 2，而字符'1'+'1'并不等于整数 2 或字符'2'。

字符型数据的存储空间和值的范围见表 3.3。

表 3.3 字符型数据的存储空间和值的范围

类 型	字节数	取 值 范 围
signed char(有符号字符型)	1	$-128 \sim 127$ , 即 $-2^7 \sim (2^7 - 1)$
unsigned char(无符号字符型)	1	$0 \sim 255$ , 即 $0 \sim (2^8 - 1)$

说明：在使用有符号字符型变量时，允许存储的值为  $-128 \sim 127$ ，但字符的代码不可能为负值，所以在存储字符时实际上只用到  $0 \sim 127$  这一部分，其第 1 位都是 0。

如果将一个负整数赋给有符号字符型变量是合法的，但它不代表一个字符，而作为一字节整型变量存储负整数。如：

```
signed char c=-6;
```

① 前面已介绍：127 个基本字符用 7 个二进位存储，如果系统只提供 127 个字符，那么就将 char 型变量的第一位二进位设置为 0，用后面 7 位存放 0 到 127，即 127 个字符的代码。在这种情况下，系统提供的 char 类型相当于 signed char。但是在实际应用中，往往觉得 127 个字符不够用，希望能多提供一些可用的字符。根据此需要，有的系统提供了扩展的字符集。把可用的字符由 127 个扩展为 255 个，即扩大了一倍。怎么解决这个问题呢？就是把本来不用的第一位用起来。把 char 变量改为 unsigned char，即第一位并不固定设为 0，而是把 8 位都用来存放字符代码。这样，可以存放  $(2^8 - 1)$  即 255 个字符代码。附录 B 中 ASCII 代码的 128~255 部分就是某系统扩展的 ASCII 字符，它并不适用于所有的系统。

读者可以用以下语句检查 ASCII 代码从 128 到 255 部分的扩展字符。

```
unsigned char c=128; //定义 c 为无符号字符变量  
printf("%d:%c\n",c,c); //输出 ASCII 代码为 128 的字符
```

观察是否输出附录 B 中代码为 128 的字符。可以用类似方法检查其他扩展字符。

编码在 127 以上的都显示为“?”。

这是因为 编号在 128~255 的是扩展的编码，原本就不是作为显示用的，当然在不同的终端上显示就不一致，这完全取决于不同的显示终端的处理。有些 ic 厂商会重新做液晶显示的驱动，将大于 127 的 ascii 码做成他们要的图标，比如一个充满电的电池的图标，一个表示加锁的图标。

通常情况下出现这个问题的原因是控制台使用了中文代码页，要显示扩展 ASCII 码，则将执行这个程序的控制台的代码页改为 437 (OEM—美国) 即可！

### 转义字符

转义字符	字符值	输出结果
\'	一个单撇号(')	具有此八进制码的字符
\"	一个双撇号(")	输出此字符
\?	一个问号(?)	输出此字符
\\"	一个反斜线(\)	输出此字符
\a	警告(alert)	产生声音或视觉信号
\b	退格(backspace)	将当前位置后退一个字符
\f	换页(form feed)	将当前位置移到下一页的开头
\n	换行	将当前位置移到下一行的开头
\r	回车(carriage return)	将当前位置移到本行的开头
\t	水平制表符	将当前位置移到下一个 tab 位置
\v	垂直制表符	将当前位置移到下一个垂直制表对齐点
\o、\oo 或\ooo 其中 o 代表一个八进制数字	与该八进制码对应的 ASCII 字符	与该八进制码对应的字符
\xh[h...] 其中 h 代表一个十六进制数字	与该十六进制码对应的 ASCII 字符	与该十六进制码对应的字符

表 3.1 中列出的字符称为“转义字符”，意思是将“\”后面的字符转换成另外的意义。如 “\n”中的“n”不代表字母 n 而作为“换行”符。

表 3.1 中倒数第 2 行是一个以八进制数表示的字符，例如'\101'代表八进制数 101 的 ASCII 字符，即'A'(八进制数 101 相当于十进制数 65，从附录 B 可以看到 ASCII 码(十进制数)为 65 的字符是大写字母'A')。'\012'代表八进制数 12(即十进制数的 10)的 ASCII 码所对应的字符“换行”符。表 3.1 中倒数第 1 行是一个以十六进制数表示的 ASCII 字符，如 '\x41'代表十六进制数 41 的 ASCII 字符，也是'A'(十六进制数 41 相当于十进制数 65)。用表 3.1 中的方法可以表示任何可显示的字母字符、数字字符、专用字符、图形字符和控制字符。如'\033'或'\x1B'代表 ASCII 代码为 27 的字符，即 ESC 控制符。'\0'或'\000'是代表 ASCII 码为 0 的控制字符，即“空操作”字符，它常用在字符串中。

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母"A"，\102 表示字母"B"，\134 表示反斜线，\XOA 表示换行等。

### 字符变量在内存中的存储形式及使用方法

每个字符变量被分配一个字节的内存空间，因此只能存放一个字符。字符值是以ASCII码的形式存放在变量的内存单元之中的。

如x的十进制ASCII码是120，y的十进制ASCII码是121。对字符变量a、b赋予'x'和'y'值：

a='x';

b='y';

实际上是在a、b两个单元内存放120和121的二进制代码：

a:

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

b:

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

所以也可以把它们看成是整型量。C语言允许对整型变量赋以字符值，也允许对字符变量赋以整型值。在输出时，允许把字符变量按整型量输出，也允许把整型量按字符量输出。

整型量为二字节数，字符量为单字节数，当整型量按字符型量处理时，只有低八位字节参与处理。

## 字符串常量

字符串是由一对双引号括起的字符序列。例如：“CHINA”，“C program”，“\$12.5”等都是合法的字符串。字符串和字符不同，它们之间主要有以下区别：

- 字符由单引号括起来，字符串由双引号括起来。
- 字符只能是单个字符，字符串则可以含一个或多个字符。
- 可以把一个字符型数据赋予一个字符变量，但不能把一个字符串赋予一个字符变量。在C语言中没有相应的字符串变量，也就是说不存在这样的关键字，将一个变量声明为字符串。但是可以用一个字符数组来存放一个字符串，这将在[数组](#)一章内予以介绍。
- 字符占一个字节的内存空间。字符串占的内存字节数等于字符串中字节数加1。增加的一个字节中存放字符"\0"（ASCII码为0）。这是字符串结束的标志。

例如，字符串 "C program" 在内存中所占的字节为：

C	p	r	o	g	r	a	m	\0
---	---	---	---	---	---	---	---	----

字符'a'和字符串"a"虽然都只有一个字符，但在内存中的情况是不同的。

'a'在内存中占一个字节，可表示为：

a
---

"a"在内存中占二个字节，可表示为：

a	\0
---	----

在《[在C语言中使用英文字符串](#)》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。

## 符号常量

(5) **符号常量**。用`#define`指令，指定用一个符号名称代表一个常量。如：

```
#define PI 3.1416 //注意行末没有分号
```

经过以上的指定后，本文件中从此行开始所有的PI都代表3.1416。在对程序进行预编译前，预处理器先对PI进行处理，把所有PI全部置换为3.1416。这种用一个符号名称代表一个常量的，称为**符号常量**。在预编译后，符号常量已全部变成字面常量(3.1416)。使用符号常量有以下好处。

① 含义清楚。看程序时从PI就可大致知道它代表圆周率。在定义符号常量时，应尽量考虑“见名知意”。在一个规范的程序中不提倡使用很多的常数，如：`sum=15*3+43`，在检查程序时搞不清各个常数究竟代表什么。应尽量使用“见名知意”的变量表示常量。

② 在需要改变程序中多处用到的同一个常量时，能做到“一改全改”。例如：

注意：要区分符号常量和变量，不要把符号常量误认为变量。符号常量不占内存，只是一个临时符号，在预编译后这个符号就不存在了，故不能对符号常量赋以新值。为与变量名相区别，习惯上符号常量用大写表示，如PI，PRICE等。

## 变量和标识符

```
const int a=3;
```

表示a被定义为一个整型变量，指定其值为3，而且在变量存在期间其值不能改变。

常变量与常量的异同是：常变量具有变量的基本属性：有类型，占存储单元，只是不允许改变其值。可以说，常变量是有名字的不变量，而常量是没有名字的不变量。有名字就便于在程序中被引用。

符号常量 Pi 和常变量 pi 都代表 3.1415926，在程序中都能使用。但二者性质不同：定义符号常量用 #define 指令，它是预编译指令，它只是用符号常量代表一个字符串，在预编译时仅是进行字符替换，在预编译后，符号常量就不存在了（全置换成 3.1415926 了），对符号常量的名字是不分配存储单元的。而常变量要占用存储单元，有变量值，只是该值不改变而已。从使用的角度看，常变量具有符号常量的优点，而且使用更方便。有了常变量以后，可以不必多用符号常量。

在计算机高级语言中，用来对变量、符号常量名、函数、数组、类型等命名的有效字符序列统称为标识符(identifier)。简单地说，标识符就是一个对象的名字。前面用到的变量名 p1, p2, c, f, 符号常量名 PI, PRICE, 函数名 printf 等都是标识符。

C 语言规定标识符只能由字母、数字和下划线 3 种字符组成，且第 1 个字符必须为字母或下划线。下面列出的是合法的标识符，可以作为变量名：

sum, average, \_total, Class, day, month, Student\_name, lotus\_1\_2\_3, BASIC, li\_ling.

下面是不合法的标识符和变量名：

M. D. John, ¥123, #33, 3D64, a>b

注意：编译系统将大写字母和小写字母认为是两个不同的字符。因此，sum 和 SUM 是两个不同的变量名，同样，Class 和 class 也是两个不同的变量名。一般而言，变量名用小写字母表示，与人们日常习惯一致，以增加可读性。

## 浮点型数据

种形式，如 3.1416 可以表示为： $3.14159 \times 10^0$ ,  $0.314159 \times 10^1$ ,  $0.0314159 \times 10^2$ ,  $31.4159 \times 10^{-1}$ ,  $314.159 \times 10^{-2}$  等，它们代表同一个值。可以看到：小数点的位置是可以在 31416 几个数字之间和之前或之后（加 0）浮动的，只要在小数点位置浮动的同时改变指数的值，就可以保证它的值不会改变。由于小数点位置可以浮动，所以实数的指数形式称为浮点数。

在指数形式的多种表示方式中把小数部分中小数点前的数字为 0、小数点后第 1 位数字不为 0 的表示形式称为规范化的指数形式，如  $0.314159 \times 10^1$  就是 3.14159 的规范化的指数形式。一个实数只有一个规范化的指数形式，在程序以指数形式输出一个实数时，必然以规范化的指数形式输出，如  $0.314159e001$ 。

浮点数类型包括 float(单精度浮点型)、double(双精度浮点型)、long double(长双精度浮点型)。

(1) **float 型**(单精度浮点型)。编译系统为每一个 float 型变量分配 4 个字节,数值以规范化的二进制数指数形式存放在存储单元中。在存储时,系统将实型数据分成小数部分和指数部分两个部分,分别存放。小数部分的小数点前面的数为 0。如 3.14159 在内存中的存放形式可以用图 3.11 表示。

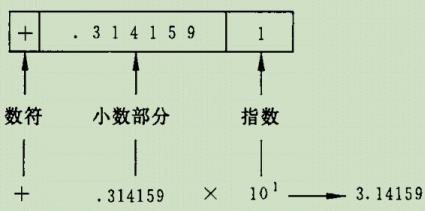


图 3.11

图 3.11 是用十进制数来示意的,实际上在计算机中是用二进制数来表示小数部分以及用 2 的幂次来表示指数部分的。在 4 个字节(32 位)中,究竟用多少位来表示小数部分,多少位来表示指数部分,C 标准并无具体规定,由各

C 语言编译系统自定。有的 C 语言编译系统以 24 位表示小数部分(包括符号),以 8 位表示指数部分(包括指数的符号)。由于用二进制形式表示一个实数以及存储单元的长度是有限的,因此不可能得到完全精确的值,只能存储成有限的精确度。小数部分占的位(bit)数愈多,数的有效数字愈多,精度也就愈高。指数部分占的位数愈多,则能表示的数值范围愈大。float 型数据能得到 6 位有效数字,数值范围为  $-3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$ 。

(2) **double 型**(双精度浮点型)。为了扩大能表示的数值范围,用 8 个字节存储一个 double 型数据,可以得到 15 位有效数字,数值范围为  $-1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$ 。在 C 语言中进行浮点数的算术运算时,将 float 型数据都自动转换为 double 型,然后进行运算。

(3) **long double 型**(长双精度)型,不同的编译系统对 long double 型的处理方法不同,Turbo C 对 long double 型分配 16 个字节。而 Visual C++ 6.0 则对 long double 型和 double 型一样处理,分配 8 个字节。请读者在使用不同的编译系统时注意其差别。

表 3.4 实型数据的有关情况

类型	字节数	有效数字	数值范围(绝对值)
float	4	6	$0 \text{ 以及 } 1.2 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	8	15	$0 \text{ 以及 } 2.3 \times 10^{-308} \sim 1.7 \times 10^{308}$
long double	8	15	$0 \text{ 以及 } 2.3 \times 10^{-308} \sim 1.7 \times 10^{308}$
	16	19	$0 \text{ 以及 } 3.4 \times 10^{-4932} \sim 1.1 \times 10^{4932}$

说明:由于用有限的存储单元存储一个实数,不可能完全精确地存储,例如 float 型变量能存储的最小正数为  $1.2 \times 10^{-38}$ ,不能存放绝对值小于此值的数,例如  $10^{-40}$ 。float 型变量能存储的范围见图 3.12。

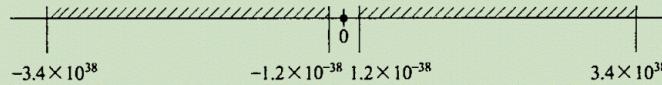


图 3.12

即数值可以在 3 个范围内:(1)  $-3.4 \times 10^{38}$  到  $-1.2 \times 10^{-38}$ ;(2) 0;(3)  $1.2 \times 10^{-38}$  到  $3.4 \times 10^{38}$ 。

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；
- %IE 以指数形式输出 double 类型，输出结果中的 E 大写。

, %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 %ee 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

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。

对于十进制小数，整数部分转换成二进制使用“展除法”（就是不断除以 2，直到余数为 0），一个有限位数的整数一定能转换成有限位数的二进制。但是小数部分就不一定了，小数部分转换成二进制使用“乘二取整法”（就是不断乘以 2，直到小数部分为 0），一个有限位数的小数并不一定能转换成有限位数的二进制，**只有末位是 5 的小数才有可能转换成有限位数的二进制，其它的小数都不行。**

float 和 double 的尾数部分是有限的，固然不能容纳无限的二进制；即使小数能够转换成有限的二进制，也有可能会超出尾数部分的长度，此时也不能容纳。这样就必须“四舍五入”，将多余的二进制“处理掉”，只保留有效长度的二进制，这就涉及到了精度的问题也就是说，浮点数不一定能保存真实的小数，很有可能保存的是一个近似值。

对于 float，尾数部分有 23 位，再加上一个隐含的整数 1，一共是 24 位。最后一位可能是精确数字，也可能是近似数字（由四舍五入、向零舍入等不同方式得到）；除此以外，剩余的 23 位都是精确数字。从二进制的角度看，这种浮点格式的小数，最多有 24 位有效数字，但是能保证的是 23 位；也就是说，整体的精度为 23~24 位。如果转换成十进制， $2^{24} = 16\ 777\ 216$ ，一共 8 位；也就是说，最多有 8 位有效数字，但是能保证的是 7 位，从而得出整体精度为 **7~8** 位。

对于 double，同理可得，二进制形式的精度为 52~53 位，十进制形式的精度为 **15~16 位**。

### 1) 舍入到最接近的值

就是将结果舍入为最接近且可以表示的值，这是默认的舍入模式。最近舍入模式和我们平时所见的“四舍五入”非常类似，但有一个细节不同。

**对于最近舍入模式，IEEE 754 规定，当有两个最接近的可表示的值时首选“偶数”值；而对四舍五入模式，当有两个最接近的可表示的值时要选较大的值。**以十进制为例，就是对.5 的舍入上采用偶数的方式，请看下面的例子。

最近舍入模式：Round(0.5) = 0、Round(1.5) = 2、Round(2.5) = 2

四舍五入模式：Round(0.5) = 1、Round(1.5) = 2、Round(2.5) = 3

### 2) 向 $+\infty$ 方向舍入（向上舍入）

会将结果朝正无穷大的方向舍入。标准库函数 ceil() 使用的就是这种舍入模式，例如，ceil(1.324) = 2，Ceil(-1.324) = -1。

### 3) 向 $-\infty$ 方向舍入（向下舍入）

会将结果朝负无穷大的方向舍入。标准库函数 `floor()` 使用的就是这种舍入模式，例如，`floor(1.324) = 1`, `floor(-1.324) = -2`。

#### 4) 向 0 舍入（直接截断）

会将结果朝接近 0 的方向舍入，也就是将多余的位数直接丢掉。C 语言中的类型转换使用的就是这种舍入模式，例如，`(int)1.324 = 1`, `(int) -1.324 = -1`。

牺牲精度换取取值范围

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     short b;
6     long c;
7     long long d;
8     float e;
9     double f;
10    long double g;
11    unsigned char h = 130;
12    printf("%d : %c\n", h, h);
13    printf("%zu,%zu,%zu,%zu,%zu,%zu,%zu", sizeof(a), sizeof(b), sizeof(c), s
14
15    return 0;
16 }
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Code +  
fengsc@ubuntu:~/Desktop/<c> \$ cd "/home/fengsc/Desktop/<c>/" && gcc shuju.c -o shuju && "/home/fengsc/Desktop/<c>/shuju  
130 : @  
4,2,8,8,4,8,16,1fengsc@ubuntu:~/Desktop/<c> \$

`%zu` 专门用来输出 `sizeof` 的结果，一般用 `%d`，本机环境提示用 `%ld` 或 `%lu`（无符号时两者等价）

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。

`size_t`是标准C库中定义的，在64位系统中为`long long unsigned int`，非64位系统中为`long unsigned int`。

数据类型“`socklen_t`”和`int`应该具有相同的长度，否则就会破坏BSD套接字层的填充。POSIX开始的时候用的是`size_t`，Linus Torvalds（他希望有更多的人，但显然不是很多）努力向他们解释使用`size_t`是完全错误的，因为在64位结构中`size_t`和`int`的长度是不一样的，而这个参数的长度必须和`int`一致，因为这是BSD套接字接口标准。最终POSIX的那帮家伙找到了解决的办法，那就是创造了一个新的类型“`socklen_t`”。Linus Torvalds说这是由于他们发现了自己的错误但又不好意思向大家伙儿承认，所以另外创造了一个新的数据类型。

-std 选项的使用方式很简单，其基本格式如下：

gcc/g++ -std=编译标准

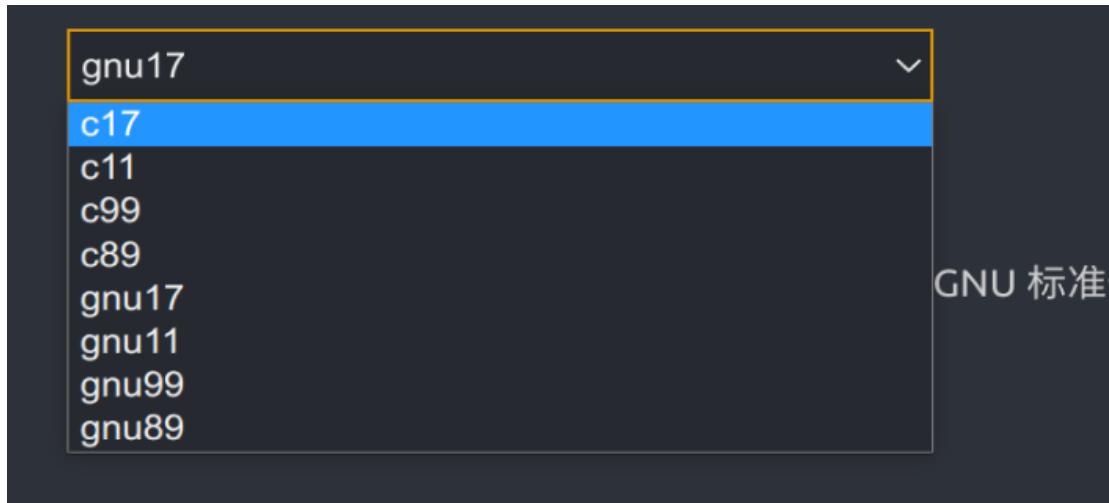
由于数据的长度和平台相关，所以基于 64 位系统比较。

	Windows	Linux
<code>unsigned int</code>	32 bits/4294967295	32 bits
<code>unsigned long</code>	32 bits	64 bits/18446744073709551615
<code>unsigned long long</code>	64 bits	64 bits
<code>size_t</code>	32 bits	64 bits

`size_t`与`unsigned long`同步  
格式控制(`printf`)

length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

```
fengsc@ubuntu:~/Desktop/<c $ gcc -E -dM -</dev/null | grep "STDC_VERSION"
#define __STDC_VERSION__ 201710L
GNU17 标准
```



## 浮点数比较大小

float, double 分别遵循 R32-24,R64-53 的标准，

他们的位数分别是 23,52，

即误差在  $2^{-23}, 2^{-52}$ ；

所以 float 的精度误差在  $1e-6$ , double 精度误差在  $1e-15$ 。

所以要判断一个单精度浮点数：则是 if( fabs(f\_float) <= 1e-6);

要判断一个双精度浮点数：则是 if( fabs(d\_double) <= 1e-15 );

判断两个浮点数相等：

```
if( fabs(a_float-b_float) <= 1e-6);
```

```
if( fabs(x_double-y_double) <= 1e-15);
```

类似的 判断大于的时候，就是 if(a>b && fabs(a-b)>1e-6)。

判断小于的时候，就是 if(a<b&&fabs(a-b)>1e-6)。

类似可实现大于等于

常量浮点数存储时误差是固定的，与用式子计算出来值的误差一般不相等

小数常量一般是 double 类型，要用 double 型变量与之比较。一般浮点数定义都用 double 最好

## 确定常量类型

**整型常量。**不带小数点的数值是整型常量,但应注意其有效范围。如在 Turbo C 中,系统为整型数据分配 2 个字节,其表值范围为 -32768~32767,如果在程序中出现数值常量 23456,系统把它作为 int 型处理,用 2 个字节存放。如果出现 49875,由于超过 32768,2 个字节放不下,系统会把它作为长整型(long int)处理,分配 4 个字节。在 Visual C++ 中,凡在 -2147483648~2147483647 之间的不带小数点的数都作为 int 型,分配 4 个字节,在此范围外的整数,而又在 long long 型数的范围内的整数,作为 long long 型处理。

在一个整数的末尾加大写字母 L 或小写字母 l,表示它是长整型(long int)。例如 123L,234l 等。但在 Visual C++ 中由于对 int 和 long int 型数据都分配 4 个字节,因此没有必要用 long int 型。

**浮点型常量。**凡以小数形式或指数形式出现的实数,是浮点型常量,在内存中都以指数形式存储。如: 10 是整型常量,10.0 是浮点型常量。那么对浮点型常量是按单精度处理还是按双精度处理呢? C 编译系统把浮点型常量都按双精度处理,分配 8 个字节。

注意: C 程序中的实型常量都是双精度浮点型常量。

如果有:

```
float a=3.14159;
```

在进行编译时,对 float 变量分配 4 个字节,但对于浮点型常量 3.14159,则按双精度处理,分配 8 个字节。编译系统会发出“警告”(warning: truncation from 'const double' to 'float')。意为“把一个双精度常量转换为 float 型”,提醒用户注意这种转换可能损失精度。

可以在常量的末尾加专用字符,强制指定常量的类型。如在 3.14159 后面加字母 F 或 f,就表示是 float 型常量,分配 4 个字节。如果在实型常量后面加大写或小写的 L,指定此常量为 long double 类型。如:

```
float a=3.14159f;           //把此 3.14159 按单精度浮点常量处理,编译时不出现“警告”。  
long double a = 1.23L       //把此 1.23 作为 long double 处理
```

# 运算符和表达式

## 基本

- 由于键盘无 $\times$ 号,运算符 $\times$ 以 $*$ 代替。
- 由于键盘无 $\div$ 号,运算符 $\div$ 以 $/$ 代替。两个实数相除的结果是双精度实数,两个整数相除的结果为整数,如 $5/3$ 的结果值为 1,舍去小数部分。但是,如果除数或被除数中有一个为负值,则舍入的方向是不固定的。例如, $-5/3$ ,有的系统中得到的结果为 $-1$ ,在有的系统中则得到结果为 $-2$ 。多数 C 编译系统(如 Visual C++)采取“向零取整”的方法,即 $5/3=1,-5/3=-1$ ,取整后向零靠拢。
- %运算符要求参加运算的运算对象(即操作数)为整数,结果也是整数。如 $8\%3$ ,结果为 2。
- 除%以外的运算符的操作数都可以是任何算术类型。

$++i,--i$  (在使用 i 之前,先使 i 的值加(减)1)

$i++,i--$  (在使用 i 之后,使 i 的值加(减)1)

粗略地看, $++i$  和  $i++$  的作用相当于  $i=i+1$ 。但  $++i$  和  $i++$  不同之处在于  $++i$  是先执行  $i+i+1$  后,再使用 i 的值;而  $i++$  是先使用 i 的值后,再执行  $i=i+1$ 。如果 i 的原值等于 3,请分析下面的赋值语句:

①  $j=++i;$  (i 的值先变成 4,再赋给 j,j 的值为 4)

②  $j=i++;$  (先将 i 的值 3 赋给 j,j 的值为 3,然后 i 变为 4)

又例如:

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

输出 4。若改为

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

则输出 3。

**注意:** 自增运算符( $++$ )和自减运算符( $--$ )只能用于变量,而不能用于常量或表达式,如 $5++$ 或 $(a+b)++$ 都是不合法的。因为 5 是常量,常量的值不能改变。 $(a+b)++$ 也不可能实现,假如 $a+b$ 的值为 5,那么自增后得到的 6 放在什么地方呢?无变量可供存放。

自增(减)运算符常用于循环语句中,使循环变量自动加 1;也用于指针变量,使指针指向下一个地址。这些将在以后的章节中介绍。

# 优先性、结合性和强制类型转换

用算术运算符和括号将运算对象(也称操作数)连接起来的、符合 C 语法规则的式子，称为 **C 算术表达式**。运算对象包括常量、变量、函数等。例如，下面是一个合法的 C 算术表达式：

```
a * b/c - 1.5 + 'a'
```

C 语言除了规定了运算符的优先级外，还规定了运算符的结合性。在表达式求值时，先按运算符的优先级别顺序执行，例如先乘除后加减。如表达式  $a - b * c$ ， $b$  的左侧为减号，右侧为乘号，而乘号优先级高于减号，因此，相当于  $a - (b * c)$ 。

如果在一个运算对象两侧的运算符的优先级别相同，如  $a - b + c$ ，则按规定的“结合方向”处理。C 语言规定了各种运算符的结合方向(结合性)，算术运算符的结合方向都是“自左至右”，即先左后右，因此  $b$  先与减号结合，执行  $a - b$  的运算，然后再执行加  $c$  的运算。“自左至右的结合方向”又称“左结合性”，即运算对象先与左面的运算符结合。以后可以看到有些运算符的结合方向为“自右至左”，即右结合性(例如，赋值运算符，若有  $a = b = c$ ，按从右到左顺序，先把变量  $c$  的值赋给变量  $b$ ，然后把变量  $b$  的值赋给变量  $a$ )。关于“结合性”的概念在其他一些高级语言中是没有的，是 C 语言的特点之一，希望能弄清楚。附录 D 列出了所有运算符以及它们的优先级别和结合性。

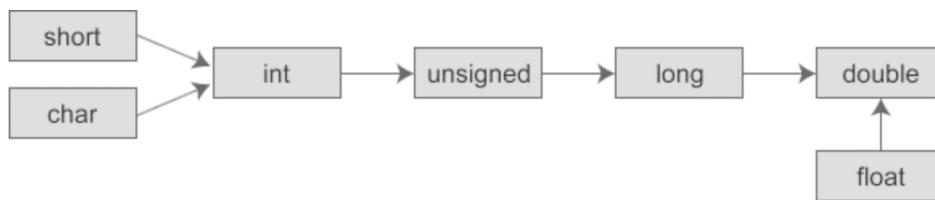
在程序中经常会遇到不同类型的数据进行运算，如  $5 * 4.5$ 。如果一个运算符的两侧的数据类型不同，则先自动进行类型转换，使二者具有同一种类型，然后进行运算。因此整型、实型、字符型数据间可以进行混合运算。规律为：

(1)  $+$ 、 $-$ 、 $*$ 、 $/$  运算的两个数中有一个数为 float 或 double 型，结果是 double 型，因为系统将所有 float 型数据都先转换为 double 型，然后进行运算。

(2) 如果 int 型与 float 或 double 型数据进行运算，先把 int 型和 float 型数据转换为 double 型，然后进行运算，结果是 double 型。

(3) 字符(char)型数据与整型数据进行运算，就是把字符的 ASCII 代码与整型数据进行运算。如： $12 + 'A'$ ，由于字符 A 的 ASCII 代码是 65，相当于  $12 + 65$ ，等于 77。字符数据可以直接与整型数据进行运算。如果字符型数据与实型数据进行运算，则将字符的 ASCII 代码转换为 double 型数据，然后进行运算。

以上的转换是编译系统自动完成的，用户不必过问。



**例 3.3** 给定一个大写字母,要求用小写字母输出。

**解题思路:** 前已介绍:字符数据以 ASCII 码存储在内存的,形式与整数的存储形式相同。所以字符型数据和其他算术型数据之间可以互相赋值和运算。

要进行大小写字母之间的转换,就要找到一个字母的大写形式和小写形式之间有什么内在联系。从附录 B 中可以找到其内在规律:同一个字母,用小写表示的字符的 ASCII 代码比用大写表示的字符的 ASCII 代码大 32。例如字符'a'的 ASCII 代码为 97,而'A'ASCII 代码为 65。将'A'的 ASCII 代码加 32,就能得到'a'的 ASCII 代码。有此思路就可以编写程序了。

## 5. 强制类型转换运算符

可以利用强制类型转换运算符将一个表达式转换成所需类型。例如:

```
(double)a      (将 a 转换成 double 类型)  
(int)(x+y)    (将 x+y 的值转换成 int 型)  
(float)(5%3)  (将 5%3 的值转换成 float 型)
```

其一般形式为

**(类型名)(表达式)**

注意,表达式应该用括号括起来。如果写成

```
(int)x+y
```

则只将 x 转换成整型,然后与 y 相加。

需要说明的是,在强制类型转换时,得到一个所需类型的中间数据,而原来变量的类型未发生变化。例如:

```
a=(int)x
```

如果已定义 x 为 float 型变量,a 为整型变量,进行强制类型运算(int)x 后得到一个 int 类型的临时值,它的值等于 x 的整数部分,把它赋给 a,注意 x 的值和类型都未变化,仍为 float 型。该临时值在赋值后就不再存在了。

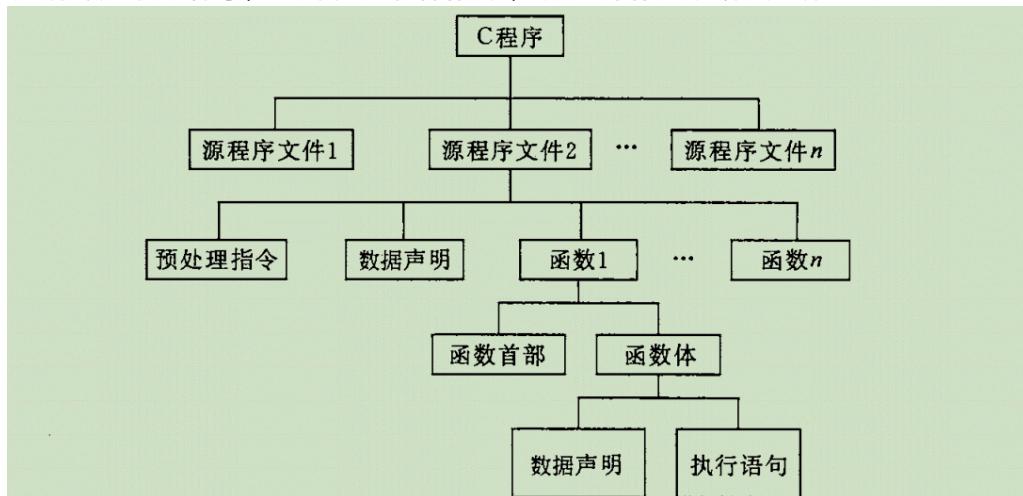
换。如%运算符要求其两侧均为整型量,若 x 为 float 型,则 x%3 不合法,必须用(int)x%3。从附录 D 可以查到,强制类型转换运算优先于%运算,因此先进行(int)x 的运算,得到一个整型的中间变量,然后再对 3 求余。此外,在函数调用时,有时为了使实参与形参类型一致,可以用强制类型转换运算符得到一个所需类型的参数。

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。

# c 语句

## 作用和分类

一个函数包含声明部分和执行部分， 执行部分是由语句组成的， 语句的作用是向计算机系统发出操作指令， 要求执行相应的操作。一个 C 语句经过编译后产生若干条机器指令。声明部分不是语句， 它不产生机器指令， 只是对有关数据的声明。



c 语句分为（一）：控制语句；（二）：函数调用语句（一个函数加一个分号）；（三）：表达式语句（一个表达式加一个分号）；（四）：空语句（一个分号）；（五）：复合语句(用 {把一些语句和声明括起来成为复合语句（又称语句块）}

可以在复合语句中包含声明部分（如上面的第 2 行），C 99 允许将声明部分放在复合语句中的任何位置，但习惯上把它放在语句块开头位置。复合语句常用在 if 语句或循环中，此时程序需要连续执行一组语句。

## 赋值表达式

### 变量 赋值运算符 表达式

赋值表达式的作用是将一个表达式的值赋给一个变量，因此赋值表达式具有计算和赋值的双重功能。如  $a=3 * 5$  是一个赋值表达式。对赋值表达式求解的过程是：先求赋值运算符右侧的“表达式”的值，然后赋给赋值运算符左侧的变量。既然是一个表达式，就应该有一个值。例如，赋值表达式  $a=3 * 5$  的值为 15，对表达式求解后，变量  $a$  的值和表达式的值都是 15。

赋值运算符左侧应该是一个可修改的“左值”(left value，简写为 lvalue)。左值的意思是它可以出现在赋值运算符的左侧，它的值是可以改变的。并不是任何形式的数据都可以作为左值的，变量可以作为左值，而算术表达式  $a+b$  就不能作为左值，常量也不能作为左值，因为常量不能被赋值。能出现在赋值运算符右侧的表达式称为“右值”(right value，简写为 rvalue)。显然左值也可以出现在赋值运算符右侧，因而凡是左值都可以作为右值。例如：

```
b=a;                                //b 是左值  
c=b;                                //b 也是右值
```

## 赋值过程中的类型转换

将一个 double 型数据赋给 float 变量时，先将双精度数转换为单精度，即只取 6、7 位有效数字，存储到 float 变量的 4 个字节中。应注意双精度数值的大小不超出量的数值范围

(5) 将一个占字节多的整型数据赋给一个占字节少的整型变量或字符变量(例如把占 4 个字节的 int 型数据赋给占 2 个字节的 short 变量或占 1 个字节的 char 变量)时，只将其低字节原封不动地送到被赋值的变量(即发生“截断”)。例如：

```
int i=289;  
char c='a';  
c=i;
```

赋值情况见图 3.15。c 的值为 33，如果用“%c”输出 c，将得到字符“!”(其 ASCII 码为 33)。

只要知道整型数据之间的赋值，按存储单元中的存储形式直接传送。实型数据之间以及整型与实型之间的赋值，是先转换(类型)后赋值。

在 C 程序中,赋值语句是用得最多的语句。在 3.3.1 节的 C 语句分类中,并没有看到赋值语句,实际上,C 语言的赋值语句属于表达式语句,由一个赋值表达式加一个分号组成。其他一些高级语言(如 BASIC,FORTRAN,COBOL,Pascal 等)有赋值语句,而无“赋值表达式”这一概念。这是 C 语言的一个特点,使之应用灵活方便。

前面已经提到,在一个表达式中可以包含另一个表达式。赋值表达式既然是表达式,那么它就可以出现在其他表达式之中。例如:

```
if ((a=b)>0) max=a;
```

按一般理解,if 后面的括号内应该是一个“条件”,例如可以是

```
if (a>0) max=a;
```

现在,在 a 的位置上换上一个赋值表达式 a=b,其作用是:先进行赋值运算(将 b 的值赋给 a),然后判断 a 是否大于 0,如大于 0,执行 max=a。请注意,在 if 语句中的 a=b 不是赋值语句,而是赋值表达式。如果写成

```
if ((a=b;>0) max=a; //“a=b;”是赋值语句
```

就错了。在 if 的条件中可以包含赋值表达式,但不能包含赋值语句。由此可以看到,C 语言把赋值语句和赋值表达式区别开来,增加了表达式的种类,使表达式的应用几乎“无孔不入”,能实现其他语言中难以实现的功能。

注意: 要区分赋值表达式和赋值语句。

赋值表达式的末尾没有分号,而赋值语句的末尾必须有分号。在一个表达式中可以包含一个或多个赋值表达式,但绝不能包含赋值语句。

## 变量赋初值

```
int a, b, c=5;
```

指定 a, b, c 为整型变量,但只对 c 初始化, c 的初值为 5

如果对几个变量赋予同一个初值,应写成

```
int a=3,b=3,c=3;
```

表示 a,b,c 的初值都是 3。不能写成

```
int a=b=c=3;
```

一般变量初始化不是在编译阶段完成的(只有在静态存储变量和外部变量的初始化是在编译阶段完成的),而是在程序运行时执行本函数时赋予初值的,相当于执行一个赋值语句。例如:

```
int a=3;
```

相当于:

```
int a; //指定 a 为整型变量  
a=3; //赋值语句,将 3 赋给 a
```

又如:

```
int a,b,c=5;
```

相当于:

```
int a,b,c; //指定 a,b,c 为整型变量  
c=5; //将 5 赋给 c
```

## 输入输出、

### printf 和 scanf

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。

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

输出%，用%%，其它用\，%+-d，表示显示正负号且左对齐

读取多行字符串时 scanf() 是以第一个非空白字符开始读入的，故第一次读入字符所输入的'\n'，在第二个 scanf() 函数开始时被忽略掉了。（“%d” 同理），%c 时不会忽略，可以用 getchar() 处理

整数补零用 .precision.

(1) 对于 printf() 来说二者没有区别

(2) 使用 scanf() 输入数据时应该区分 %f 和 %lf，当输入数据是 float 时用 %f，当输入数据是 double 时用 %lf

都是默认输出六位小数

```
1 #include <stdio.h>
2 int main()
3 {
4     int a1=1,a2=2,a3=3,b1=4,b2=5,b3=6,c1=7,c2=8,c3=9;
5     printf("%-6d %-6d %-6d\n",a1,a2,a3);
6     printf("%-6d %-6d %-6d\n",b1,b2,b3);
7     printf("%-6d %-6d %-6d\n",c1,c2,c3);
8     return 0;
9 }
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gctop/c/"juzhen
1      2      3
4      5      6
7      8      9
```

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。

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。

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
-	<code>-</code> 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。
#	<ul style="list-style-type: none"><li>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。</li><li>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。</li></ul>

```
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。

本来计算的理论值应为 3333.3333333…，但由于 float 型数据只能保证 6~7 位有效数字，因此虽然程序输出了 6 位小数，但从左面开始的第 7 位数字以后的数字并不保证是绝对正确的。

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：

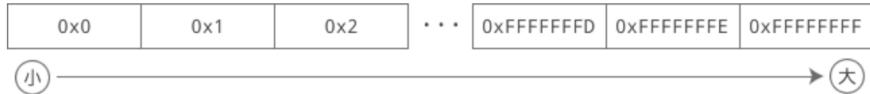
- `scanf()`：和 `printf()` 类似，`scanf()` 可以输入多种类型的数据。
- `getchar()`、`getche()`、`getch()`：这三个函数都用于输入单个字符。
- `gets()`：获取一行数据，并作为字符串处理。

`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。

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。

`%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。

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

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；如果不符要求，那么就继续等待用户输入，或者干脆读取失败

`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

`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

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：

```
01. #include <stdio.h>
02. int main()
03. {
04.     char letter;
05.     int age;
06.     char url[30];
07.     float price;
08.
09.     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 is %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. }
```

## 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）

# scanf 高级用法

%2d 表示最多读取两位整数；

%10s 表示读取的字符串的最大长度为 10，或者说，最多读取 10 个字符。

限制读取数据的长度在实际开发中非常有用，最典型的一个例子就是读取字符串：我们为字符串分配的内存是有限的，用户输入的字符串过长就存放不下了，就会冲刷掉其它的数据，从而导致程序出错甚至崩溃；如果被黑客发现了这个漏洞，就可以构造栈溢出攻击，改变程序的执行流程，甚至执行自己的恶意代码，这对服务器来说简直是灭顶之灾。

在用 gets() 函数读取字符串的时候，有一些编译器会提示不安全，建议替换为 gets\_s() 函数，就是因为 gets() 不能控制读取到的字符串的长度，风险极高。

%s 控制符会匹配除空白符以外的所有字符，它有两个缺点：

- %s 不能读取特定的字符，比如只想读取小写字母，或者十进制数字等，%s 就无能为力；
- %s 读取到的字符串中不能包含空白符，有些情况会比较尴尬，例如，无法将多个单词存放到一个字符串中，因为单词之间就是以空格为分隔的，%s 遇到空格就读取结束了。

要想解决以上问题，可以使用 scanf() 的另外一种字符匹配方式，就是%[xxx]，[ ]包围起来的是需要读取的字符集合。例如，%[abcd] 表示只读取字符 abcd，遇到其它的字符就读取结束；注意，这里并不强调字符的顺序，只要字符在 abcd 范围内都可以匹配成功，所以你可以输入 abcd、dcba、ccdc、bdcca 等。

为了简化字符集合的写法，scanf() 支持使用连字符来表示一个范围内的字符，例如 %[a-z]、%[0-9] 等。

连字符左边的字符对应一个 ASCII 码，连字符右边的字符也对应一个 ASCII 码，位于这两个 ASCII 码范围以内的字符就是要读取的字符。注意，连字符左边的 ASCII 码要小于右边的，如果反过来，那么它的行为是未定义的。

常用的连字符举例：

- %[a-z] 表示读取 abc...xyz 范围内的字符，也即小写字母；
- %[A-Z] 表示读取 ABC...XYZ 范围内的字符，也即大写字母；
- %[0-9] 表示读取 012...789 范围内的字符，也即十进制数字。

你也可以将它们合并起来，例如：

- %[a-zA-Z] 表示读取大写字母和小写字母，也即所有英文字母；
- %[a-zA-Z0-9] 表示读取所有的英文字母和十进制数字；
- %[0-9a-f] 表示读取十六进制数字。

## 不匹配某些字符

假如现在有一种需求，就是读取换行符以外的所有字符，或者读取 0~9 以外的所有字符，该怎么实现呢？总不能把剩下的字符都罗列出来吧，一是麻烦，二是不现实。

C 语言的开发者们早就考虑到这个问题了，scanf() 允许我们在%[ ]中直接指定某些不能匹配的字符，具体方法就是在不匹配的字符前面加上^，例如：

- %[^\\n] 表示匹配除换行符以外的所有字符，遇到换行符就停止读取；
- %[^0-9] 表示匹配除十进制数字以外的所有字符，遇到十进制数字就停止读取。

## 可以用来读取一整行数据

### 3) 丢弃读取到的字符

在前面的代码中，每个格式控制符都要对应一个变量，把读取到的数据放入对应的变量中。其实你也可以不这样做，`scanf()` 允许把读取到的数据直接丢弃，不往变量中存放，具体方法就是在 % 后面加一个\*，例如：

- `%*d` 表示读取一个整数并丢弃；
- `%*[a-z]` 表示读取小写字母并丢弃；
- `%[^\\n]` 表示将换行符以外的字符全部丢弃。

`scanf()` 控制字符串的完整写法为：

`%{*} {width} type`

其中，{} 表示可有可无。各个部分的具体含义是：

- type 表示读取什么类型的数据，例如 `%d`、`%s`、`%[a-z]`、`%[^\\n]` 等；type 必须有。
- width 表示最大读取宽度，可有可无。
- \* 表示丢弃读取到的数据，可有可无。

## 1. `scanf("%[^\\n]", str);` 接受带有空格的字符

## printf 控制输出字符颜色

格式:`printf("\033[显示方式;字背景颜色;字体颜色m字符串\033[0m")`;只改变颜色只需输入字体颜色项即可，显示方式可以写多项（**前景背景数字可交换位置，并都不能为变量**）

字背景颜色范围: 40--49

40: 黑  
41: 红  
42: 绿  
43: 黄  
44: 蓝  
45: 紫  
46: 深绿  
47: 白色

字颜色: 30--39

30: 黑  
31: 红  
32: 绿  
33: 黄  
34: 蓝  
35: 紫  
36: 深绿  
37: 白色

`\033[0m` 关闭所有属性

显示方式 :0 (默认值) 、 1 (高亮) 、 22 (非粗体) 、 4 (下划线) 、 24 (非下划线) 、 5 (闪烁) 、 25 (非闪烁) 、 7 (反显) 、 27 (非反显)

# 其它输入输出函数

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：

```
01. #include <stdio.h>
02. int main()
03. {
04.     char c;
05.     c = getchar();
06.     printf("c: %c\n", c);
07.
08.     return 0;
09. }
```

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();
```

无论操作系统实际使用何种方法检测文件结尾，在 C 语言中，用 `getchar()` 读取文件检测到文件结尾时将返回一个特殊的值，即 EOF（end of file 的缩写）。`scanf()` 函数检测到文件结尾时也返回 EOF。通常，EOF 定义在 `stdio.h` 文件中：

```
#define EOF (-1)
```

为什么是 -1？因为 `getchar()` 函数的返回值通常都介于 0~127，这些值对应标准字符集。但是，如果系统能识别扩展字符集，该函数的返回值可能在 0~255 之间。无论哪种情况，-1 都不对应任何字符，所以，该值可用于标记文件结尾。

某些系统也许把 EOF 定义为 -1 以外的值，但是定义的值一定与输入字符所产生的返回值不同。如果包含 `stdio.h` 文件，并使用 EOF 符号，就不必担心 EOF 值不同的问题。这里关键要理解 EOF 是一个值，标志着检测到文件结尾，并不是在文件中找得到的符号

那么，如何在程序中使用 EOF？把 `getchar()` 的返回值和 EOF 作比较。如果两值不同，就说明没有到达文件结尾。也就是说，可以使用下面这样的表达式：

```
while ((ch = getchar()) != EOF)
```

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：

```
01. #include <stdio.h>
02. int main()
03. {
04.     char author[30], lang[30], url[30];
05.     gets(author);
06.     printf("author: %s\n", author);
07.     gets(lang);
08.     printf("lang: %s\n", lang);
09.     gets(url);
10.     printf("url: %s\n", url);
11.
12.     return 0;
13. }
```

- `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。`(getchar` 也以空格为分隔)
- `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。

## gets 与 fgets

`fgets` 函数用来从文件中读入字符串。`fgets` 函数的调用形式如下：`fgets (str, n, fp)`；此处，`fp` 是文件指针；`str` 是存放在字符串的起始地址；`n` 是一个 `int` 类型变量。函数的功能是从 `fp` 所指文件中读入 `n-1` 个字符放入 `str` 为起始地址的空间内；如果在未读满 `n-1` 个字符之时，已读到一个换行符或一个 EOF（文件结束标志），则结束本次读操作，读入的字符串中最后包含读到的换行符。因此，确切地说，调用 `fgets` 函数时，最多只能读入 `n-1` 个字符。读入结束后，系统将自动在最后加 `\0`，并以 `str` 作为函数值返回。

其中：`s` 代表要保存到的内存空间的首地址，可以是字符数组名，也可以是指向字符数组的字符指针变量名。`size` 代表的是读取字符串的长度。`stream` 表示从何种流中读取，可以是标准输入流 `stdin`，也可以是文件流，即从某个文件中读取，这个在后面讲文件的时候再详细介绍。标准输入流就是前面讲的输入缓冲区。所以如果是从键盘读取数据的话就是从输入缓冲区中读取数据，即从标准输入流 `stdin` 中读取数据，所以第三个参数为 `stdin`。

虽然用 `gets()` 时有空格也可以直接输入，但是 `gets()` 有一个非常大的缺陷，即它不检查预留存储区是否能够容纳实际输入的数据，换句话说，如果输入的字符数目大于数组的长度 `gets` 无法检测到这个问题，就会发生内存越界，所以编程时建议使用 `fgets()`。

`fget()` 函数中的 `size` 如果小于字符串的长度，那么字符串将会被截取；如果 `size` 大于字符串的长度则多余的部分系统会自动用 `'\0'` 填充。所以假如你定义的字符数组长度为 `n`，那么 `fgets()` 中的 `size` 就指定为 `n-1`，留一个给 `'\0'` 就行了。

## gets 会抛弃换行符

但是需要注意的是，如果输入的字符串长度没有超过 `n-1`，那么系统会将最后输入的换行符 `'\n'` 保存进来，保存的位置是紧跟输入的字符，然后剩余的空间都用 `'\0'` 填充。所以此时输出该字符串时 `printf` 中就不需要加换行符 `'\n'` 了，因为字符串中已经有了。

## putchar

`putchar` 函数的基本格式为：`putchar(c)`。

(1) 当 `c` 为一个被单引号（英文状态下）引起来的字符时，输出该字符（注：该字符也可

为转义字符)；

(2) 当 c 为一个介于 0~127 (包括 0 及 127) 之间的十进制整型数时，它会被视为对应字符的 ASCII 代码，输出该 ASCII 代码对应的字符；

(3) 当 c 为一个事先用 char 定义好的字符型变量时，输出该变量所指向的字符

```
#include <stdio.h>
int main ()
{
    int a=66,b=79,c=89;           //定义 3 个整型变量，并初始化
    putchar(a);                  //向显示器输出字符 B
    putchar(b);                  //向显示器输出字符 O
    putchar(c);                  //向显示器输出字符 Y
    putchar ('\n');              //向显示器输出一个换行符
    return 0;
}
```

## 重定向输入输出

### 1.重定向输入

假设已经编译了 echo\_eof.c 程序，并把可执行版本放入一个名为 echo\_eof (或者在 Windows 系统中名为 echo\_eof.exe) 的文件中。运行该程序，输入可执行文件名：

```
echo_eof
```

```
#include <stdio.h>
int main(void)
{
int ch;
while ((ch = getchar()) != EOF)
putchar(ch);
return 0;
}
```

使用该程序进行键盘输入，要设法输入 EOF 字符。不能只输入字符 EOF，也不能只输入 -1 (输入 -1 会传送两个字符：一个连字符和一个数字 1)。正确的方法是，必须找出当前系统的要求。例如，在大多数 UNIX 和 Linux 系统中，在一行开始处按下 Ctrl+D 会传输文件结尾信号。许多微型计算机系统都把一行开始处的 Ctrl+Z 识别为文件结尾信号，一些系统把任意位置的 Ctrl+Z 解释成文件结尾信号。

现在，假设你希望制作一份 mywords 文件的副本，并命名为 savewords。

只需输入以下命令即可：

```
echo_eof< mywords > savewords
```

下面的命令也起作用，因为命令与重定向运算符的顺序无关：

```
echo_eof> savewords < mywords
```

注意：在一条命令中，输入文件名和输出文件名不能相同。

```
512echo_eof< mywords > mywords....<--错误
```

原因是> mywords 在输入之前已导致原 mywords 的长度被截断为 0。

总之，在 UNIX、Linux 或 Windows/DOS 系统中使用两个重定向运算符

（<和>）时，要遵循以下原则。

重定向运算符连接一个可执行程序（包括标准操作系统命令）和一个数据文件，不能用于连接一个数据文件和另一个数据文件，也不能用于连接一个程序和另一个程序。

使用重定向运算符不能读取多个文件的输入，也不能把输出定向至多个文件

```
fengsc@ubuntu:~/Desktop/c/func $ cd "/home/fengsc/Desktop/c/func/" && gcc recursive4.c -o recursive4 -lm && "/home/fengsc/Desktop/c/func/"recursive4<input>output
fengsc@ubuntu:~/Desktop/c/func $ cat output
yes
fengsc@ubuntu:~/Desktop/c/func $
```

等价于./recursive4<input>output

## 缓冲区

缓冲区（Buffer）又称为缓存（Cache），是内存空间的一部分。也就是说，计算机在内存中预留了一定的存储空间，用来暂时保存输入或输出的数据，这部分预留的空间就叫做缓冲区（缓存）。

有时候，从键盘输入的内容，或者将要输出到显示器上的内容，会暂时进入缓冲区，待时机成熟，再一股脑将缓冲区中的所有内容“倒出”，我们才能看到变量的值被刷新，或者屏幕产生变化。

缓冲区是为了让低速的输入输出设备和高速的用户程序能够协调工作，并降低输入输出设备的读写次数。

用户程序的执行速度可以看做 CPU 的运行速度，如果没有各种硬件的阻碍，理论上它们是同步的。例如，我们都知道硬盘的速度要远低于 CPU，它们之间有好几个数量级的差距，当向硬盘写入数据时，程序需要等待，不能做任何事情，就好像卡顿了一样，用户体验非常差。计算机上绝大多数应用程序都需要和硬件打交道，例如读写硬盘、向显示器输出、从键盘输入等，如果每个程序都等待硬件，那么整台计算机也将变得卡顿。

但是有了缓冲区，就可以将数据先放入缓冲区中（内存的读写速度也远高于硬盘），然后程序可以继续往下执行，等所有的数据都准备好了，再将缓冲区中的所有数据一次性地写入硬盘，这样程序就减少了等待的次数，变得流畅起来。

缓冲区的另外一个好处是可以减少硬件设备的读写次数。其实我们的程序并不能直接读写硬件，它必须告诉操作系统，让操作系统内核（Kernel）去调用驱动程序，只有驱动程序才能真正的操作硬件。从用户程序到硬件设备要经过好几层的转换，每一层的转换都有时间和空间的开销，而且开销不一定小；一旦用户程序需要密集的输入输出操作，这种开销将变得非常大，会成为制约程序性能的瓶颈。

这个时候，分配缓冲区就是必不可少的。每次调用读写函数，先将数据放入缓冲区，等数据都准备好了再进行真正的读写操作，这就大大减少了转换的次数。实践证明，合理的缓冲区设置能成倍提高程序性能。缓冲区其实就是一个内存空间，它用在硬件设备和用户程序之间，用来缓存数据，目的是让快速的 CPU 不必等待慢速的输入输出设备，同时减少操作硬件的次数。

### 1) 全缓冲

在这种情况下，当缓冲区被填满以后才进行真正的输入输出操作。缓冲区的大小都有限制的，比如 1KB、4MB 等，数据量达到最大值时就清空缓冲区。

在实际开发中，将数据写入文件后，打开文件并不能立即看到内容，只有清空缓冲区，或者关闭文件，或者关闭程序后，才能在文件中看到内容。这种现象，就是缓冲区在作怪。

### 2) 行缓冲

在这种情况下，当在输入或者输出的过程中遇到换行符时，才执行真正的输入输出操作。行缓冲的典型代表就是标准输入设备（也即键盘）和标准输出设备（也即显示器）。

② 对于 `scanf()`，不管用户输入多少内容，只要不按下回车键，就不进行真正的读取。这是因为 `scanf()` 是带有行缓冲的，用户输入的内容会先放入缓冲区，直到用户按下回车键，产生换行符`\n`，才会刷新缓冲区，进行真正的读取。

### 3) 不带缓冲

不带缓冲区，数据就没有地方缓存，必须立即进行输入输出。

`getche()`、`getch()` 就不带缓冲区，输入一个字符后立即就执行了，根本不用按下回车键。

Windows 下的 `printf()` 也不带缓冲区，不管最后有没有换行符`\n`，都会立即输出，所以对于类似的输出代码，错误信息输出函数 `perror()` 也没有缓冲区。错误信息必须刻不容缓、立即、马上显示出来，

C语言标准规定，输入输出缓冲区要具有以下特征：

- 当且仅当输入输出不涉及交互设备时，它们才可以是全缓冲的。
- 错误显示设备不能带有缓冲区。

现代计算机已经没有了专门的错误显示设备，所有的信息都显示到一个屏幕上，这里的错误显示设备只能是计算机的显示器。上面提到的 perror() 其实就是向错误显示设备上输出信息，但是现代计算机已经把显示器作为了错误显示设备，所以 perror() 也是向显示器上输出内容。

所谓交互设备，就是现代计算机上的显示器和键盘。C标准虽然规定它们不能是全缓冲的，但并没有规定它们到底是行缓冲还是不带缓冲，这就导致不同的平台有不同的实现。

### 1) 输入设备

scanf()、getchar()、gets() 就是从输入设备（键盘）上读取内容。对于输入设备，没有缓冲区将导致非常奇怪的行为，比如，我们本来想输入一个整数 947，没有缓冲区的话，输入 9 就立即读取了，根本没有机会输入 47，所以，没有输入缓冲区是不能接受的。Windows、Linux、Mac OS 在实现时都给输入设备带上了行缓冲，所以 scanf()、getchar()、gets() 在每个平台下的表现都一致。

但是在某些特殊情况下，我们又希望程序能够立即响应用户按键，例如在游戏中，用户按下方向键人物要立即转向，而且越快越好，这肯定就不能带有缓冲区了。Windows 下特有的 getch() 和 getche() 就是为这种特殊需求而设计的，它们都不带缓冲区。

### 2) 输出设备

printf()、puts()、putchar() 就是向输出设备（显示器）上显示内容。对于输出设备，有没有缓冲区其实影响没有那么大，顶多是晚一会看到内容，不会有功能性的障碍，所以 Windows 和 Linux、Mac OS 采用了不同的方案：

- Windows 平台下，输出设备是不带缓冲区的；
- Linux 和 Mac OS 平台下，输出设备带有行缓冲区。

## 缓冲区的刷新（清空）

所谓刷新缓冲区，就是将缓冲区中的内容送达到目的地。缓冲区的刷新遵循以下的规则：

- 不管是行缓冲还是全缓冲，缓冲区满时会自动刷新；
- 行缓冲遇到换行符\n时会刷新；
- 关闭文件时会刷新缓冲区；
- 程序关闭时一般也会刷新缓冲区，这个是由标准库来保障的；
- 使用特定的函数也可以手动刷新缓冲区

`scanf()` 是带有缓冲区的。遇到 `scanf()` 函数，程序会先检查输入缓冲区中是否有数据：

- 如果没有，就等待用户输入。用户从键盘输入的每个字符都会暂时保存到缓冲区，直到按下回车键，输入结束，`scanf()` 再从缓冲区中读取数据，赋值给变量。
- 如果有数据，哪怕是一个字符，`scanf()` 也会直接读取，不会等待用户输入。

### 回车之后没有数据才会等待

程序执行到第一个 `scanf()`，由于缓冲区中没有数据，所以会等待用户输入。从键盘输入100 200 300后按下回车键，输入就结束了，`scanf()` 开始从缓冲区中读取数据。由于控制字符串是“%d”，所以它会读取一个整数，这里匹配到的整数是100。接下来将100赋值给变量 a，并将100从缓冲区中删除，此时缓冲区中剩下200 300。

**注意：`scanf()` 匹配到想要的数据后，会将匹配到的数据从缓冲区中删除，而没有匹配到的数据仍然会留在缓冲区中。**

执行到第二个 `scanf()` 时，检测到缓冲区中有内容，所以不会等待用户输入，而是直接从缓冲区中读取。此时缓冲区中的内容为200 300，`scanf()` 会匹配到整数200，并将200从缓冲区中删除，剩下300。

执行到第三个 `scanf()` 时，同理会匹配到300，并将300赋值给变量 c。

匹配失败意味着不会从缓冲区中删除任何数据。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a=0, b=0;
6     scanf("a=%d", &a);
7     scanf("b=%d", &b);
8     printf("a=%d, b=%d\n", a, b);
9
10    system("pause");
11    return 0;
12 }
```

登录后复制



运行结果：

```
a=100↙
a=100, b=0
```

遇到第一个 `scanf()`，输入a=100并回车，就会将100赋值给a，并将a=100从缓冲区中删除。遇到第二个 `scanf()` 时，缓冲区中不是没有内容了吗，为什么不会等待用户输入呢？

其实当用户按下回车键时，回车换行符也会被保存到缓冲区，只是大多数情况下 `scanf()` 会忽略，前面的两个例子就是这样。但是当控制字符串不是以 %xxx 开头时，回车换行符就起作用了，`scanf()` 会对它进行匹配，只是匹配失败而已。该例中第二个 `scanf()` 就是匹配回车换行符失败，所以既不等待用户输入，也不给b赋值。

即不会被忽略

可以删去 b=

读取字符时也会不忽略并且将换行符匹配为字符，可以在 scanf 后使用 getchar () ；去（读）掉缓存中的那个换行符（读取遵循先来后到），更简单的方法是清空缓冲区

```
10     int a = 0, b = 0;char c;
11     scanf("a=%d", &a);
12     c=getchar();
13     scanf("b=%d", &b);
14     printf("a=%d, b=%d\n, c=%c(%d)\n", a, b,c,c);
15
16     return 0;
17 }
```

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/"
c/"char
a=2
b=3
a=2, b=3
,c=
(10)
```

总而言之，换行符的影响出现在换行符后的 scanf 语句中的控制语句不是%开头或者需要读取字符

清空缓冲区

只能用于 windows: fflush(stdin); 或 rewind(stdin);

linux 可用：1, [setbuf](#)(stdin, NULL); 2, scanf("%\*[^\n]%\*c"); \*表示该输入项读入后不赋予任何变量

scanf("%\*[^\n]");//清理（格式化）输入缓冲区中第一个\n之前的所有字符

scanf("%\*c");//清理输入缓冲区中第一个字符，也就是上次遗留下的\n

scanf("%\*[^\n]%\*c") 是把这两句放在一起,当缓冲区只有\n时前第一个式子将失败致使跳过第二个式子（是与关系），此时需要分开使用。

3, while((c = getchar()) != '\n' && c != EOF) (不断读取，读到 EOF(End of file)可以替换为其它) 或\n再跳出，需要用 int 或 char 声明 c)

system 是包含在 [stdlib.h](#) 头文件中的，要实现类似功能的方法，要么编程

system("read") (system("read -p 'Press Enter to continue...' var");, -p 内容可选

;要么不用 system 命令，直接 getchar(); 不过跟 system("pause") 的区别是，后者是按任意键继续，但是前面两种方法都是按回车键继续的。

使用 system 浪费系统资源，如有需要一般用 getchar 有时需要两个，缓冲区可能还有一个\n.

功能：system() 函数调用 “/bin/sh -c command” 执行特定的命令，阻塞当前进程直到 command 命令执行完毕

原型：

int system(const char \*command);

返回值：

如果无法启动 shell 运行命令，system 将返回 127；出现不能执行 system 调用的其他错误时返回-1。如果 system 能够顺利执行，返回那个命令的退出码。在该 command 执行期间，SIGINT 和 SIGQUIT 是被忽略的

6. 请编程将“China”译成密码，密码规律是：用原来的字母后面第4个字母代替原来的字母。例如，字母“A”后面第4个字母是“E”，用“E”代替“A”。因此，“China”应译为“Glmre”。请编一程序，用赋初值的方法使c1,c2,c3,c4,c5这5个变量的值分别为'C','h','i','n','a'，经过运算，使c1,c2,c3,c4,c5分别变为'G','l','m','r','e'。分别用putchar函数和printf函数输出这5个字符。

```
1 #include <stdio.h>
2 int main()
3 {
4     char a,b,c,d,e;
5     scanf("%c%c%c%c%c", &a, &b, &c, &d, &e);
6     a=a+4, b=b+4, c=c+4, d=d+4, e=e+4;
7     printf("%c%c%c%c\n", a, b, c, d, e);
8     putchar(a),putchar(b),putchar(c),putchar(d),putchar(e);
9     return 0;
10 }
11 }
```

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL 2: Code

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc xitil.c -o xitil & p/c/"xitil
China
Glmre
Glmrefengsc@ubuntu:~/Desktop/c $
```

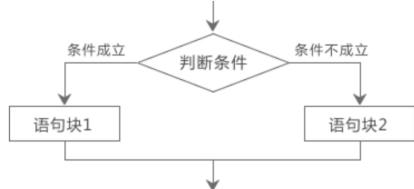
# 分支结构和循环结构

## if

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：



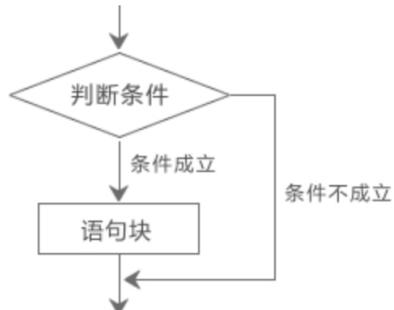
所谓语句块 (Statement Block)，就是由 {} 包围的一个或多个语句的集合。如果语句块中只有一个语句，也可以省略 {}，例如：

```
01. if(age>=18) printf("恭喜，你已经成年，可以使用该软件！\n");  
02. else printf("抱歉，你还未成年，不宜使用该软件！\n");
```

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：



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：

```
01. if(a!=b) // ①  
02. if(a>b) printf("a>b\n"); // ②  
03. else printf("a<b\n"); // ③
```

③和②配对，而不是和①配对。

平方：

直接用两个数(或变量)相乘就可以表示平方，比如 $x*x$

不过如果，需要求 $m$ 的 $n$ 次方，就需要用到 $\text{pow}(x,y)$ 乘方(包括开方)这个库函数了，使用 $\text{pow}(x,y)$ 这个库函数，需要 $\text{math.h}$ 头文件

其中 $x$ 和 $y$ 都是双精度浮点(double)型

标准库的大部分函数通常放在文件 `libc.a` 中 (文件名后缀 `.a` 代表 “achieve”，译为“获取”)，或者放在用于共享的动态链接文件 `libc.so` 中 (文件名后缀 `.so` 代表 “share object”，译为“共享对象”)。这些链接库一般位于 `/lib/` 或 `/usr/lib/`，或者位于 [GCC](#) 默认搜索的其他目录。

当使用 `GCC` 编译和链接程序时，`GCC` 默认会链接 `libc.a` 或者 `libc.so`，但是对于其他的库 (例如非标准库、第三方库等)，就需要手动添加。

令人惊讶的是，标准头文件 `<math.h>` 对应的数学库默认也不会被链接，如果没有手动将它添加进来，就会发生函数未定义错误。

如果我们不使用 `-l` 选项：

```
[root@bogon demo]# gcc main.c  
/tmp/ccYfkZJk.o: In function `main':  
main.c:(.text+0x34): undefined reference to `cos'  
collect2: ld returned 1 exit status
```

显然，`GCC` 编译器无法找到 `cos()` 这个函数。为了编译这个 `main.c`，必须使用 `-l` 选项，以链接数学库：

```
[root@bogon demo]# gcc main.c -o main.out -lm
```

数学库的文件名是 `libm.a`。前缀 `lib` 和后缀 `.a` 是标准的，`m` 是基本名称，`GCC` 会在 `-l` 选项后紧跟着的基本名称的基础上自动添加这些前缀、后缀，本例中，基本名称为 `m`。

```

18     int main()
19     {
20         double a, b, c, disc, x1, x2, realpart, imagpart;
21         printf("please input a equation like ax^2+bx+c=0\n");
22         scanf("%lf x^2 %lf x %lf = 0", &a, &b, &c);
23         printf("the equation ");
24         if (fabs(a) <= 1e-6)
25         {
26             printf("is not a quadratic\n");
27         }
28         else
29         {
30             disc = (b * b - 4 * a * c);
31
32             if (fabs(disc) <= 1e-6)
33                 printf("two equal roots: x1=x2=%8.4f\n", -b / (2 * a));
34             else if (disc > 1e-6)
35             {
36                 x1 = (-b + sqrt(disc)) / (2 * a);

```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    2: Code

```

fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc xiti2.c -o xiti2
sktop/c/xiti2
please input a equation like ax^2+bx+c=0
666x^2+777x-888=0
the equation has distinct real roots : x1= 0.7103, x2= -1.8770
the disc is 2969361.0000
fengsc@ubuntu:~/Desktop/c $

```

直接判断与 0 的关系在系数为分数时可能会出现误差

## 关系运算符

关系运算符在使用时，它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	$\leq$
>	大于	>
>=	大于或等于	$\geq$
==	等于	=
!=	不等于	$\neq$

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。

判断时，等号左边写常量，等号右边写变量，防止少写一个等号带来的严重 BUG

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。

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：

```
01. //情况①
02. if (b) {
03.     //TODO:
04. }
05. //情况②
06. if (b=5) { //情况①
07.     //TODO:
08. }
```

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：

```
01. if (a=b)
02.     printf("%d", a);
03. else
04.     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。

## 逻辑运算符

运算符	说明	结合性	举例
<code>&amp;&amp;</code>	与运算，双目，对应数学中的“且”	左结合	<code>1&amp;&amp;0、(9&gt;3)&amp;&amp;(b&gt;a)</code>
<code>  </code>	或运算，双目，对应数学中的“或”	左结合	<code>1  0、(9&gt;3)   (b&gt;a)</code>
<code>!</code>	非运算，单目，对应数学中的“非”	右结合	<code>!a、!(2&lt;5)</code>

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。逻辑运算符和其它运算符优先级从低到高依次为：

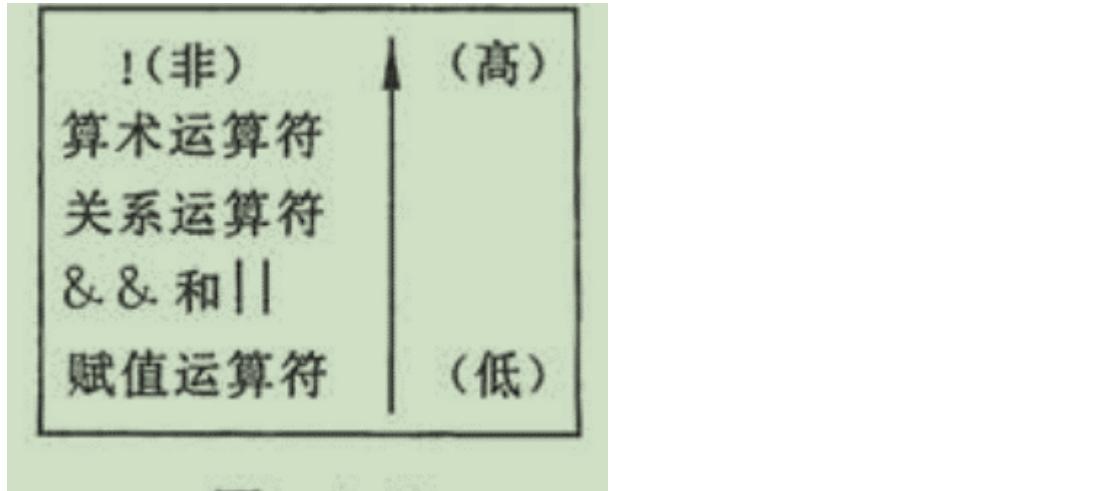
赋值运算符(=) < && 和 || < 关系运算符 < 算术运算符 < 非(!)

&& 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  $a > b \ \&\& b || 9 > c$ ， $a || c > d \ \&\& !p$ 。



5>3 && 8<4 - !0

表达式自左至右扫描求解。首先处理“5>3”（因为关系运算符优先于逻辑运算符&&）。在关系运算符>两侧的5和3作为数值参加关系运算，“5>3”的值为1（代表真）。再进行“1 && 8<4 - !0”的运算，8的左侧为“&&”，右侧为“<”运算符，根据优先规则，应先进行“<”的运算，即先进行“8<4 - !0”的运算。现在4的左侧为“<”，右侧为“-”运算符，而“-”优先于“<”，因此应先进行“4 - !0”的运算，由于“!”的级别最高，因此先进行“!0”的运算，得到结果1。然后进行“4 - 1”的运算，得到结果3，再进行“8<3”的运算，得0，最后进行“1&&0”的运算，得0。

实际上，逻辑运算符两侧的运算对象不但可以是0和1，或者是0和非0的整数，也可以是字符型、浮点型、枚举型或指针型的纯量型数据。系统最终以0和非0来判定它们属于“真”或“假”。例如：'c' && 'd'的值为1（因为'c'和'd'的ASCII值都不为0，按“真”处理），所以1 && 1的值为1。

从左至右，两两比较

(1)  $a \& \& b \& \& c$ 。只有  $a$  为真(非 0)时,才需要判别  $b$  的值。只有当  $a$  和  $b$  都为真的情况下才需要判别  $c$  的值。如果  $a$  为假,就不必判别  $b$  和  $c$ (此时整个表达式已确定为假)。如果  $a$  为真, $b$  为假,不判别  $c$ ,见图 4.7。

(2)  $a \parallel b \parallel c$ 。只要  $a$  为真(非 0),就不必判断  $b$  和  $c$ 。只有  $a$  为假,才判别  $b$ 。 $a$  和  $b$  都为假才判别  $c$ ,见图 4.8。



逻辑型变量、

```
51 #include <stdio.h>
52 #include <stdbool.h>
53 int main()
54 {
55
56     bool a, b, c, d;
57     int e, f;
58     a = 3 > 4, b = 4 > 3, c = e = 2, d = f = 0;
59     printf("%d %d %d %d\n", a, b, c, d);
60     if (a == false && b == true)
61         printf("yes");
62     return 0;
63 }
64
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" &
sktop/c/"xit2
0 1 1 0
yesfengsc@ubuntu:~/Desktop/c $
```

# 条件运算符

如果希望获得两个数中最大的一个，可以使用 if 语句，例如：

```
01. if(a>b) {  
02.     max = a;  
03. } else {  
04.     max = b;  
05. }
```

纯文本 复制

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：

```
01. 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

```
01. max=(a>b) ? a : b;
```

可以去掉括号而写为

```
01. 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：

```
01. a>b ? a : c>d ? c : d;
```

应理解为：

```
01. 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。

flag=(x>-5 && x<0)?1:(x>=0 && x<5)?2:(x>=5 && x<10)?3:4;

实现分段函数

(3) 上面的例子是利用了条件表达式的值,把它赋给一个变量 max。实际上也可以不把条件表达式的值赋予一个变量。表达式加一个分号,就成为一个独立的语句。如:

```
a>b ? (max=a) : (max=b); //表达式 2 和表达式 3 是赋值表达式,赋值表达式两侧有括号。  
分析为什么?
```

相当于:

```
if (a>b) max=a;  
else max=b;
```

条件表达式还可以写成以下形式:

```
a>b ? printf("%d",a) : printf ("%d",b)
```

即“表达式 2”和“表达式 3”不仅可以是数值表达式,还可以是赋值表达式或函数表达式。上

## 先执行表达式,再赋值

```
#include <stdio.h>  
  
int main()  
{  
    char ch;  
    scanf("%c", &ch);  
    ch=(ch>='A' && ch<='Z') ? (ch+32) : ch;  
    printf("%c\n", ch);  
    return 0;  
}
```

字符比较大小即对应的 ASCII 值比较

字符串用 strcmp 比较

# switch

**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”。

**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：

```
01. #include <stdio.h>  
02. int main(){  
03.     int a;  
04.     printf("Input integer number:");  
05.     scanf("%d",&a);  
06.     switch(a){  
07.         case 1: printf("Monday\n"); break;  
08.         case 2: printf("Tuesday\n"); break;  
09.         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. }
```

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。

```
68     printf("please input a grade:\n");
69     scanf("%d", &grade);
70     switch ((int)(grade / 10))
71     {
72     case 0:
73     case 1:
74     case 2:
75     case 3:
76     case 4:
77     case 5:
78         printf("%d is rank E\n", grade);
79         break;
80     case 6:
81         printf("%d is rank D\n", grade);
82         break;
83     case 7:
84         printf("%d is rank C\n", grade);
85         break;
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL 2: Code

```
xiti2.c:81:44: error: break statement not within loop or switch
81 |     case 10:printf("%d is rank A\n",grade);break;
|          ^
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc xiti2.c -o xiti2
sktop/c/"xiti2
please input a grade:
66
66 is rank D
```

## (do) while

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。

再看一个例子，统计从键盘输入的一行字符的个数：

```
01. #include <stdio.h>
02. int main(){
03.     int n=0;
04.     printf("Input a string:");
05.     while(getchar() != '\n') n++;
06.     printf("Number of characters: %d\n", n);
07.     return 0;
08. }
```

纯文本 复制

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”。

注意 `while(i<=100);` 最后的分号；，这个必须要有。

## for

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
}
```

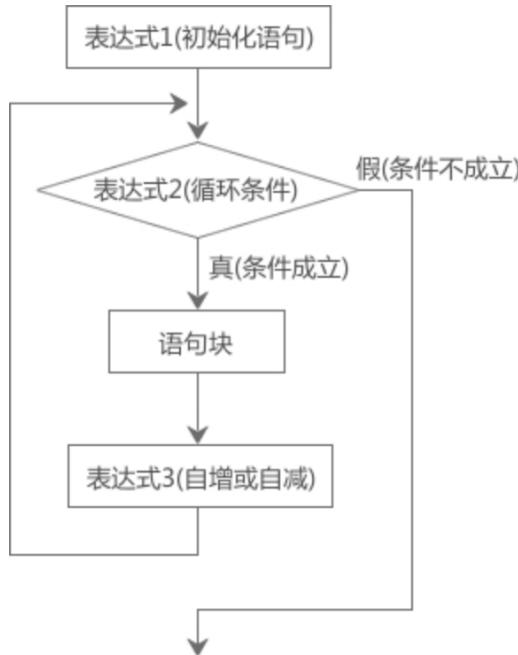
它的运行过程为：

- 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：



1) 修改“从1加到100的和”的代码，省略“表达式1（初始化条件）”：

```
01. int i = 1, sum = 0;
02. for( ; i<=100; i++) {
03.     sum+=i;
04. }
```

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：

```
01. for(i=1; ; i++) sum=sum+i;
```

相当于：

```
01. i=1;
02. while(1) {
03.     sum=sum+i;
04.     i++;
05. }
```

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：

```
01. for( i=1; i<=100; ) {
02.     sum=sum+i;
03.     i++;
04. }
```

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：

```
01. for( ; i<=100 ; ){
02.     sum=sum+i;
03.     i++;
04. }
```

相当于：

```
01. while(i<=100) {
02.     sum=sum+i;
03.     i++;
04. }
```

6) “表达式1”可以是初始化语句，也可以是其他语句。例如：

```
01. 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。

```
01. for( sum=0, i=1; i<=100; i++ )    sum=sum+i;
```

或：

```
01. 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，但也可用数值或字符，只要其值非零，就执行循环体。例如：

```
01. for( i=0; (c=getchar()) != '\n'; i+=c );
```

又如：

```
01. for( ; (c=getchar()) != '\n' ; )
02.     printf("%c", c);
```

## break continue

C 语言中 **break** 语句有以下两种用法：

1. 当 **break** 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。
2. 它可用于终止 **switch** 语句中的一个 **case**。

如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），**break** 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

1. **break** 语句对 **if-else** 的条件语句不起作用。
2. 在多层循环中，一个 **break** 语句只向外跳一层  
只作用于 **loop** 和 **switch**，**if** 中的 **break** 会跳出包含 **if** 的内层循环（如果有的话）

```

int i,j,n=0;
for (i=1;i<=4;i++)
    for (j=1;j<=5;j++,n++)
        { if(n%5==0)printf("\n");
          if (i==3 && j==1)break; //遇到第3行第1列,终止内循环
          printf("%d\t",i*j);
        }
    printf("\n");
    return 0;
}

```

请读者分析,输出结果会怎样。实际的输出如下:

1	2	3	4	5
2	4	6	8	10
4	8	12	16	20

第3行空白,即不输出第3行的5个数据。原因是:当i等于3和j等于1时,执行break语句,提前终止执行内循环,流程进入下一次外循环,即开始第4次外循环,i等于4。

```
if (i==3 && j==1) continue;
```

请分析运行情况。实际的输出如下:

1	2	3	4	5
2	4	6	8	10
6	9	12	15	
4	8	12	16	20

原来第3行第1个数据3没有输出,从第3行第2个数据6开始输出,由于没有执行“printf(“%d\t”,i\*j);”,所以少输出一次“\t”,后面4个数据向左移动了一个位置。应当注意的是continue语句只是跳过其后的“printf(“%d\t”,i\*j);”结束了当i=3,j=1时的那次内循环,而接着执行i=3,j=2时的内循环。

## 小结和举例

### 九九乘法表的五种输出形式

```

for(i=1;i<=9;i++) {
    for(j=1;j<=9;j++)
        // %2d 控制宽度为两个字符,且右对齐;如果改为 %-2d 则为左对齐
        // \t为tab缩进
        printf("%d*%d=%2d\t", i, j, i*j);
}

```

```

for(i=1;i<=9;i++) {
    for(j=1;j<=9;j++) {
        if(j<i)
            //打印八个空格,去掉空格就是左上三角形
            printf("         ");
        else
            printf("%d*%d=%2d    " i i i*j);
    }
}

```

```

for(i=1;i<=9;i++) {
    // 将下面的for循环注释掉，就输出左下三角形
    for(n=1; n<=9-i; n++)
        printf("      ");
}

for(j=1;j<=i;j++)
    printf("%d*%d=%2d ", i, j, i*j);

```

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（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。

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。  
防止 else 与就近的 if 结合可为 if 加入一个 else 空语句，从 if 语句中退出可用 exit (n) 或 return (n) 需要头文件 stdlib.h

## 1、exit函数和return函数的主要区别是：

- 1) exit用于在程序运行的过程中随时结束程序，exit的参数是返回给OS的。main函数结束时也会隐式地调用exit函数。exit函数运行时首先会执行由atexit()函数登记的函数，然后会做一些自身的清理工作，同时刷新所有输出流、关闭所有打开的流并且关闭通过标准I/O函数tmpfile()创建的临时文件。exit是结束一个进程，它将删除进程使用的内存空间，同时把错误信息返回父进程；而return是返回函数值并退出函数。通常情况：exit(0)表示程序正常，exit(1)和exit(-1)表示程序异常退出，exit(2)表示表示系统找不到指定的文件。在整个程序中，只要调用exit就结束（当前进程或者在main时候为整个程序）。
- 2) **return是语言级别的，它表示了调用堆栈的返回；** return( )是当前函数返回，当然如果是在主函数main，自然也就结束当前进程了，如果不是，那就是退向上一层调用。在多个进程时。如果有时要检测上个进程是否正常退出。就要用到上个进程的返回值，依次类推。而**exit是系统调用级别的，它表示了一个进程的结束。**
- 3) exit函数是退出应用程序，并将应用程序的一个状态返回给OS，这个状态标识了应用程序的一些运行信息。
- 4) 和机器和操作系统有关的一般是：0为正常退出，非0为非正常退出；

## 2、进程环境与进程控制

exit(int n)其实就是直接退出程序，因为默认的标准程序入口为 int main(int argc, char\*\* argv)，返回值是int型的。一般在shell下面，运行一个程序，然后使用命令echo \$?就能得到该程序的返回值，也就是退出值，在main()里面，你可以用return n，也能够直接用exit(n)来做。unix默认的正确退出是返回0，错误返回非0。理论上exit可以返回小于256的任何整数。返回的不同数值主要是给调用者作不同处理的。

单独的进程是返回给操作系统的。如果是多进程，是返回给父进程的。父进程里面调用waitpid()等函数得到子进程退出的状态，以便作不同处理。根据相应的返回值来让调用者作出相应的处理。总的说来，**exit()就是当前进程把控制权返回给调用该程序的程序，括号里的是返回值，告诉调用程序该程序的运行状态。**

自增自减的区别只体现在赋值的时候，执行完后结果一样，故在 for 中没有区别  
注释嵌套外层应该用

```
#if 0
    #ednif
用/**/会就近匹配，不能嵌套
```

```
01. #include<stdio.h>
02. int main()
03. {
04.     int a, i, b, n;
05.     printf("There are following friendly--numbers pair smaller than 3000:\n");
06.     for( a=1; a<3000; a++ ) /*穷举3000以内的全部整数*/
07.     {
08.         for( b=0, i=1; i<=a/2; i++ ) /*计算数a的各因子，各因子之和存放于b*/
09.             if( !(a%i) )
10.                 b+=i;
11.         for( n=0, i=1; i<=b/2; i++ ) /*计算b的各因子，各因子之和存于n*/
12.             if( !(b%i) )
13.                 n+=i;
14.         if( n==a && a<b ) /*使每对亲密数只输出一次*/
15.             printf("%4d--%4d    ", a, b); /*若n=a，则a和b是一对亲密数，输出*/
16.     }
17.
18.     return 0;
19. }
```

## 内存泄漏

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(){
4     while(1){ //死循环
5         malloc(1024); //分配1024个字节的内存
6     }
7
8     return 0;
9 }
```

登录后复制

while 循环的条件是 1，始终成立，循环会一直进行下去，永无休止，所以是一个“死循环”。

每次循环，程序都会向计算机申请 1024 个字节 (1KB) 的内存，并且不会释放；循环到第 1024 次时，程序就占用了  $1024 \times 1024$  个字节 (1MB) 的内存；循环到  $1024 \times 1024$  次时，程序就占用了  $1024 \times 1024 \times 1024$  个字节 (1GB) 的内存。

不要害怕，亲自跑一下试试，打开 Windows 下的任务管理器，可以看到内存的使用率会飙升，稍等片刻后程序会被终止。Windows 的内存管理机制发现我们的程序占用内存太多，会让它崩溃，防止系统卡死（其它的操作系统也有相应的措施）。

内存泄漏是C语言程序很常见的一个问题，当程序规模比较大时，有时候找到这个 Bug 就变得很困难，所以大家一定要养成良好的内存使用习惯，及时释放不再使用的内存。

# 数组

## 基本概念

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：

```
01. float m[12]; //定义一个长度为 12 的浮点型数组  
02. 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  $0 \leq \text{index} < \text{length}$ ，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

```
01. #include <stdio.h>  
02. int main() {  
03.     int nums[10];  
04.     int i;  
05.  
06.     //从控制台读取用户输入  
07.     for(i=0; i<10; i++) {  
08.         scanf("%d", &nums[i]); //注意取地址符 &，不要遗忘哦  
09.     }  
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. }
```

纯文本 复制

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：

a[0]	a[1]	a[2]	a[3]
------	------	------	------

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};
```

#### //数组的指定初始化

```
int array1[6] = {[0]=3,[1]=2,[4]=6};  
display(6,array1); //输出: 3 2 0 0 6 0
```

可以使用 `[m ... n]`

```
1 #include <stdio.h>//Fibonacci sequence
2 int main()
3 {
4     int i,f[20]={1,1};
5     for(i=2;i<20;i++)
6         f[i]=f[i-1]+f[i-2];
7     for(i=0;i<20;i++)
8     {
9         if(i%5==0)
10            printf("\n");
11         printf("%-10d",f[i]);
12     }
13     return 0;
14 }
15 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc -o f f.c
4
1      1      2      3      5
8     13     21     34     55
89    144    233    377    610
987   1597   2584   4181   6765      fengsc@ubuntu
```

数组的赋值：

```
int arr[5];
arr[5] = {1,2,3,4,5}; //error arr[i]这种写法是访问数组元素，并且 arr[5]越界了
arr = {1,2,3,4,5}; //error arr 是数组名，我们目的是给数组中的每一个元素赋值
```

所以：赋值时只能依次给每个元素赋值，使用 for 循环来实现

## 初始化空串

在C语言编程中，当我们声明一个字符串数组的时候，常常需要把它初始化为空串。总结起来有以下三种方式：

- (1) char str[10]="";
- (2) char str[10]={'\0'};
- (3) char str[10]; str[0]='\0' ;

第(1)(2)种方式是将str数组的所有元素都初始化为'\0'，而第(3)种方式是只将str数组的第一个元素初始化为'\0'。如果数组的size非常大，那么前两种方式将会造成很大的开销。所以，除非必要（即我们需要将str数组的所有元素都初始化为0的情况），我们都应该选用第(3)种方式来初始化字符串数组。

## 二维数组

```
dataType arrayName[length1][length2];
```

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

```
for (i=0; i<3; i++) {  
    for (j=0; j<5; j++) {  
        scanf("%d", &a[j][i]); //输入每个同学的各科成绩  
        sum += a[j][i]; //计算当前科目的总成绩  
    }  
}
```

(1) 分行给二维数组赋初值。例如：

```
int a[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

这种赋初值方法比较直观，把第 1 个花括号内的数据给第 1 行的元素，第 2 个花括号内的数据赋给第 2 行的元素……即按行赋初值。

(2) 可以将所有数据写在一个花括号内，按数组元素在内存中的排列顺序对各元素赋初值。例如：

```
int a[3][4]={ 1,2,3,4,5,6,7,8,9,10,11,12 },
```

效果与前相同。但以第(1)种方法为好，一行对一行，界限清楚。用第(2)种方法如果数据多，则会写成一大片，容易遗漏，也不易检查。

(3) 可以对部分元素赋初值。例如：

```
int a[3][4]={ {1},{5},{9} };
```

(3) 可以对部分元素赋初值。例如：

```
int a[3][4]={{1},{5},{9}};
```

它的作用是只对各行第 1 列(即序号为 0 的列)的元素赋初值,其余元素值自动为 0。初始化后数组各元素为:

1	0	0	0
5	0	0	0
9	0	0	0

也可以对各行中的某一元素赋初值,例如:

```
int a[3][4]={{1},{0,6},{0,0,11}};
```

初始化后的数组元素如下:

1	0	0	0
0	6	0	0
0	0	11	0

这种方法对非 0 元素少时比较方便,不必将所有的 0 都写出来,只须输入少量数据。

也可以只对某几行元素赋初值:

```
int a[3][4]={{1},{5,6}};
```

也可以对第 2 行不赋初值,例如:

```
int a[3][4]={{1},{ },{9}};
```

(4) 如果对全部元素都赋初值(即提供全部初始数据),则定义数组时对第 1 维的长度可以不指定,但第 2 维的长度不能省。例如:

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

与下面的定义等价:

```
int a[][][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

系统会根据数据总个数和第 2 维的长度算出第 1 维的长度。数组一共有 12 个元素,每行 4 列,显然可以确定行数为 3。

在定义时也可以只对部分元素赋初值而省略第 1 维的长度,但应分行赋初值。例如:

```
int a[][][4]={{{0,0,3},{ }},{0,10}};
```

这样的写法,能通知编译系统;数组共有 3 行。数组各元素为

0	0	3	0
0	0	0	0
0	10	0	0

# 数组元素查询

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：

```
01. #include <stdio.h>
02. int main(){
03.     int nums[10] = {1, 10, 6, 296, 177, 23, 0, 100, 34, 999};
04.     int i, num, thisindex = -1;
05.
06.     printf("Input an integer: ");
07.     scanf("%d", &num);
08.     for(i=0; i<10; i++) {
09.         if(nums[i] == num) {
10.             thisindex = i;
11.             break;
12.         }
13.     }
14.     if(thisindex < 0) {
15.         printf("%d isn't in the array.\n", num);
16.     } else{
17.         printf("%d is in the array, it's index is %d.\n", num, thisindex);
18.     }
19.
20.     return 0;
21. }
```

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 个元素就没有必要再比较了，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。

查询到后 `break` 跳出

# 字符数组

## 基本

用来存放字符的数组称为**字符数组**, 例如:

```
01. char a[10]; //一维字符数组
02. char b[5][10]; //二维字符数组
03. char c[20]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'}; // 给部分数组元素赋值
04. 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:

```
01. char str[30] = {"c.biancheng.net"};
02. 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:

```
01. char str[] = {"c.biancheng.net"};
02. 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:

```
01. char str[7];
02. str = "abc123"; //错误
03. //正确
04. str[0] = 'a'; str[1] = 'b'; str[2] = 'c';
05. str[3] = '1'; str[4] = '2'; str[5] = '3';
```

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：

```
01. 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：

```
01. 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。

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'：

```
01. #include <stdio.h>
02. int main() {
03.     char str[30];
04.     char c;
05.     int i;
06.     for(c=65, i=0; c<=90; c++, i++) {
07.         str[i] = c;
08.     }
09.     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。

The screenshot shows a code editor interface with a dark theme. On the left is the code editor pane containing a C program. On the right is the terminal pane showing the execution of the program and its output.

```
76 #include <stdio.h>
77 int main()
78 {
79     char str[27]={0},c;//需要大于26，使后面的'0'被识别从而输出字符串；
80     int i;
81     for(i=0,c=65; c<=90;i++,c++)
82     {
83         str[i] = c;
84     }
85     printf("%s\n",str);
86     return 0;
87 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc xiti4.c -o x
4
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

将数组的某一部分赋值，其余部分都将成为零

由于系统自动在最后一个字符后面加了一个'\0',因此 c 数组的存储情况如下:

C		p	r	o	g	r	a	m	.	\0
---	--	---	---	---	---	---	---	---	---	----

若想用一个新的字符串代替原有的字符串"C program.",从键盘输入"Hello"分别赋给 c 数组中前面 5 个元素。如果不加'\0'的话,字符数组中的字符如下:

H	e	l	l	o	g	r	a	m	.	\0
---	---	---	---	---	---	---	---	---	---	----

新字符串和老字符串连成一片,无法区分开。如果想输出字符数组中的字符串,则会连续输出:

Hellogram.

如果在"Hello"后面加一个'\0',它取代了第 6 个字符"g"。在数组中的存储情况为

H	e	l	l	o	\0	r	a	m	.	\0
---	---	---	---	---	----	---	---	---	---	----

'\0'是字符串结束标志,如果用以下语句输出数组 c 中的字符串:

```
printf("%s\n",c); //输出数组 c 中的字符串
```

在输出字符数组中的字符串时,遇'\0'就停止输出,因此只输出了字符串"Hello"。而不会

- (1) 输出的字符中不包括结束符'\0'。
- (2) 用"%s"格式符输出字符串时,printf 函数中的输出项是字符数组名,而不是数组元素名。写成下面这样是不对的:

```
printf("%s",c[0]);
```

- (3) 如果数组长度大于字符串的实际长度,也只输出到遇'\0'结束。例如:

```
char c[10]={"China"}; //字符串长度为 5,连'\0'共占 6 个字节
printf("%s",c);
```

只输出字符串的有效字符"China",而不是输出 10 个字符。这就是用字符串结束标志的好处。

- (4) 如果一个字符数组中包含一个以上'\0',则遇第一个'\0'时输出就结束。

- (5) 可以用 scanf 函数输入一个字符串。例如:

```
scanf("%s",c);
```

scanf 函数中的输入项 c 是已定义的字符数组名,输入的字符串应短于已定义的字符数组的长度。例如,已定义:

```
char c[6];
```

从键盘输入:

China ↵

系统会自动在 China 后面加一个'\0'结束符。如果利用一个 scanf 函数输入多个字符串,则

## 字符串的输入输出

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。

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。

scanf 读取带空格字符串用 scanf(“%[^\\n]”,str)

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。

## 字符串处理函数

`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，感兴趣的读者请[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。

## 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：

```
01. #include <stdio.h>
02. #include <string.h>
03. int main(){
04.     char str1[50] = "《C语言变怪兽》";
05.     char str2[50] = "http://c.biancheng.net/cpp/u/jiaocheng/";
06.     strcpy(str1, str2);
07.     printf("str1: %s\n", str1);
08.
09.     return 0;
10. }
```

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。

(3) 如果在复制前未对 str1 数组初始化或赋值，则 str1 各字节中的内容是无法预知的，复制时将 str2 中的字符串和其后的'\0'一起复制到字符数组 1 中，取代字符数组 1 中的前面 6 个字符，最后 4 个字符并不一定是'\0'，而是 str1 中原有的最后 4 个字节的内容。

(4) 不能用赋值语句将一个字符串常量或字符数组直接给一个字符数组。如下面两行都是不合法的：

```
str1="China";           //企图用赋值语句将一个字符串常量直接赋给一个字符数组
str1=str2;             //企图用赋值语句将一个字符数组直接赋给另一个字符数组
```

只能用 strcpy 函数将一个字符串复制到另一个字符数组中去。用赋值语句只能将一个字符赋给一个字符型变量或字符数组元素。如下面的语句是合法的：

```
char a[5],c1,c2;
c1='A';c2='B';
a[0]='C'; a[1]='h'; a[2]='i'; a[3]='n'; a[4]='a';
```

(5) 可以用 strncpy 函数将字符串 2 中前面 n 个字符复制到字符数组 1 中去。例如：

```
strncpy(str1,str2,2);
```

作用是将 str2 中最前面 2 个字符复制到 str1 中，取代 str1 中原有的最前面 2 个字符。但复制的字符个数 n 不应多于 str1 中原有的字符(不包括'\0')。

## 6. **strlen** 函数——测字符串长度的函数

其一般形式为

**strlen** (字符数组)

**strlen** 是 STRing LENgth(字符串长度)的缩写。它是测试字符串长度的函数。函数的值为字符串中的实际长度(不包括'\0'在内)。例如：

```
char str[10] = "China";
printf("%d", strlen(str));
```

输出结果不是 10,也不是 6,而是 5。也可以直接测试字符串常量的长度,例如：

```
strlen("China");
```

## 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。

**strchr** 函数功能为在一个串中查找给定字符的第一个匹配之处。函数原型为：char \***strchr**(const char \*str, int c), 即在参数 str 所指向的字符串中搜索第一次出现字符 c (一个无符号字符) 的位置。

返回一个指向该字符串中第一次出现的字符的指针,如果字符串中不包含该字符则返回 NULL 空指针。

```
find = strchr(st, '\n'); // 查找换行符 if (find) // 如果地址不是 NULL, *find = '\0'; // 在此处放置一个空字符 else while (getchar() != '\n') continue; // 处理剩余输入行 }
```

## 实例

统计单词数

```
# include <stdio. h>
int main()
{
    char string[81];
    int i, num=0, word=0;
    char c;
    gets(string);                                //输入一个字符串给字符数组 string
    for (i=0;(c=string[i])!='\0';i++)
        if(c==' ') word=0;                      //只要字符不是'\0'就继续执行循环
        else if(word==0)                         //如果是空格字符,使 word 置 0
            {word=1;                            //如果不是空格字符且 word 原值为 0
             num++;                           //使 word 置 1
             num++;                           //num 累加 1,表示增加一个单词
            }
    printf("There are %d words in this line.\n",num); //输出单词数
    return 0;
}
```

```
1 #include <stdio.h>
2 #include <string.h>
3 int main()
4 {
5     char str1[3][20],str[20];
6     for(int i=0;i<3;i++)
7         fgets(str1[i],20,stdin);
8         printf("%s",str1[0]);
9         if(strcmp(str1[0],str1[1])>0)
10            strcpy(str,str1[0]);
11        else
12            strcpy(str,str1[1]);
13        if(strcmp(str1[2],str)>0)
14            strcpy(str,str1[2]);
15        printf("the largest str is %s\n",str);
16        return 0;
17
18
19 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
fhtjj
the largest str is stghj

fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop"
uuu
lll
mmm
uuu
the largest str is uuu
```

## gets 和 fgets 函数及其区别

我们知道，对于 gets 函数，它的任务是从 stdin 流中读取字符串，直至接收到换行符或 EOF 时停止，并将读取的结果存放在 buffer 指针所指向的字符数组中。这里需要注意的是，换行符不作为读取串的内容，读取的换行符被转换为 null('0') 值，并由此来结束字符串。即换行符会被丢弃，然后在末尾添加 null('0') 字符。其函数的原型如下：

```
01. char* gets(char* buffer);
```

如果读入成功，则返回与参数 buffer 相同的指针；如果读入过程中遇到 EOF 或发生错误，返回 NULL 指针。因此，在遇到返回值为 NULL 的情况，要用 perror 或 feof 函数检查是发生错误还是遇到 EOF。



函数 gets 可以无限读取，不会判断上限，所以程序员应该确保 buffer 的空间足够大，以便在执行读操作时不发生溢出。也就是说，gets 函数并不检查缓冲区 buffer 的空间大小，事实上它也无法检查缓冲区的空间。

如果函数的调用者提供了一个指向堆栈的指针，并且 gets 函数读入的字符数量超过了缓冲区的空间（即发生溢出），gets 函数会将多出来的字符继续写入堆栈中，这样就覆盖了堆栈中原来的内容，破坏一个或多个不相关变量的值。如下面的示例代码

根据运行结果，当用户在键盘上输入的字符个数大于缓冲区 buffer 的最大界限时，gets 函数也不会对其进行任何检查，因此我们可以将恶意代码多出来的数据写入堆栈。由此可见，gets 函数是极其不安全的，可能成为病毒的入口，因为 gets 函数没有限制输入的字符串长度。所以我们应该使用 fgets 函数来替换 gets 函数，实际上这也是大多程序员所推荐的做法。

相对于 gets 函数，fgets 函数最大的改进就是能够读取指定大小的数据，从而避免 gets 函数从 stdin 接收字符串而不检查它所复制的缓冲区空间大小导致的缓存溢出问题。当然，fgets 函数主要是为文件 I/O 而设计的（注意，不能用 fgets 函数读取二进制文件，因为 fgets 函数会把二进制文件当成文本文件来处理，这势必会产生乱码等不必要的麻烦）。其中，fgets 函数的原型如下：

```
01. char *fgets(char *buf, int bufsize, FILE *stream);
```

该函数的第二个参数 bufsize 用来指示最大读入字符数。如果这个参数值为 n，那么 fgets 函数就会读取最多 n-1 个字符或者读完一个换行符为止，在这两者之中，最先满足的那个条件用于结束输入。

与 gets 函数不同的是，如果 fgets 函数读到换行符，就会把它存储到字符串中，而不是像 gets 函数那样丢弃它。即给定参数 n，fgets 函数只能读取 n-1 个字符（包括换行符）。如果有一行超过 n-1 个字符，那么 fgets 函数将返回一个不完整的行（只读取该行的前 n-1 个字符）。但是，缓冲区总是以 null('0') 字符结尾，对 fgets 函数的下一次调用会继续读取该行。

也就是说，每次调用时，fgets 函数都会把缓冲区的最后一个字符设为 null('0')，这意味着最后一个字符不能用来存放需要的数据。所以如果某一行含有 size 个字符（包括换行符），要想把这行读入缓冲区，要把参数 n 设为 size+1，即多留一个位置存储 null('0')。

最后，它还需要第 3 个参数来说明读取哪个文件。如果是从键盘上读入数据，可以使用 stdin 作为该参数，如下面的代码所示

所以用 fgets 读取的字符串长度比 gets 读取的大一 (\n)

存储长度为 n 的字符串用 fgets 读取时长度应定义为 n+2（结尾为 \n\0），输出时会自动换行

fgets 只能读取 N-1 个字符，包括最后的 '\n'，读完结束后系统将自动在最后加 '\0'（gets 读完结束后系统自动会将 '\n' 置换成 '\0'）。

说到这里就有两种情况了：

一：当你从键盘上输入 <=N-1 个字符（包括 '\n'）时，那么字符串 str 会以 '\n\0' 结尾。这就造成了 strlen(str) 比你想象的大 1，

当然你可以通过下面代码将 '\n' 去掉。

```
1 if(str[strlen(str) - 1] == '\n') { // 去掉换行符
2   str[strlen(str) - 1] = '\0';
3 }
```

二：当你从键盘上输入>N-1个字符（包括'\n'）时，那么字符串 str 会以'\0'结尾。  
正常读取时返回值是字符串指针，异常是 NULL

```
char *ret_val;
char *find;

/*fgets获取的是一行字符串，遇到换行符或读取到n-1个字符时结束读取，前者将换行符保存进字
/*注意字符串中的空格字符不是'\0'*/
ret_val = fgets(st, n, stdin);
/*成功读取数据为返回str，不成功则返回NULL空指针 */
if (ret_val != NULL)
{
    /*str是专门搜索字符串中的字符的，遇到'\0'或者找到目标字符就会停止*/
    /*查找st所指向的字符串中的换行符'\n'，返回指向该字符的指针，未找到返回NULL指针*/
    find = strchr(st, '\n');
    /*表示找到了换行符*/
    if (find != NULL)
    {
        *find = '\0';
    }
    /*else表示未找到'\n'，但是已经搜寻到了字符串末尾，这是因为fgets获取了n-1个字符就
    else
    {
        /*此时fgets获取了n-1个字符，屏幕中的输入字符没有读完故使用getchar继续读取，目
         的是为了避免，接下来继续读取字符时，续接之前的输入数据，这实质上是丢弃了第n-1
         取输入字符要从下一行开始*/
        /*getchar()函数是从标准输入(屏幕)读取下一个字符*/
        while (getchar() != '\n')
        {
            continue;
        }
    }
}
```

# 数组的下标越界与内存溢出

## 1. 下标越界

在引用数组元素时，使用的下标超过了该数组下标的应有范围，但应注意的是：

C/C++不对数组做边界检查。可以重写数组的每一端，并写入一些其他变量的数组或者甚至是写入程序的代码。不检查下标是否越界可以有效提高程序运行的效率，因为如果你检查，那么编译器必须在生成的目标代码中加入额外的代码用于程序运行时检测下标是否越界，这就会导致程序的运行速度下降，所以为了程序的运行效率，C / C++才不检查下标是否越界。发现如果数组下标越界了，那么它会自动接着那块内存往后写。

关于C/C++为什么不对数组的下标是否越界做检查，可以参考：

<http://www.xuebuyuan.com/967089.html>

因为编译器不会自动检测你的数组下标是否越界，而是把这个任务交给了程序员自己，所以在写程序，引用数组元素时，一定注意不要让数组的下标越界。

还有，初学者一定不能忘了数组的下标是从0开始的，不是常识中的从1开始。

## 2. 内存溢出

在初始化数组(给数组元素赋值)时，**初始化(赋值)**元素的个数超过了数组定义时元素的个数。这里的元素个数就是在定义数组时那个方框框里的数字，对于多维数组来说，元素个数 = 每个方框框里的数字之积。

当然，求数组元素个数可以用公式：

数组元素个数 = sizeof(数组名)/sizeof(数组任意一个元素)

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



程序输出如下

```
0022FF48 0022FF34 0022FF30
2 2 2 2
```

存储空间图示如下，可以看到b恰好在数组的前面，而a正好在数组的后面。

0022FF30	0022FF34	0022FF38	0022FF3C	0022FF40	0022FF44	0022FF48
b	s[0]	s[1]	s[2]	s[3]	s[4]	a

$s[i] = *(s+i)$ ，所以 $s[-1]$ 正好是b， $s[5]$ 正好是a。而-1,5两个下标不在数组的有效下标中。在现实中经常可以看到编程人员控制不好 $s[i]$ 中的i，导致i越界，有些语言（如Java）运行时刻会对此进行检查，但是C不会。

```
1 #include <stdio.h>
2 int main(void)
3 {
4     char a, s[5], b;
5
6     a = b = '@';
7     gets(s);
8     printf("%p %p %p\n", &a, s, &b);
9     printf("%c %s %c\n", a, s, b);
10
11    return 0;
12 }
```



输入下面的内容

abcdefg

程序的输出如下

0022FF4F 0022FF4A 0022FF49

f abcdefg @

存储空间图示如下，字符数组s容纳不下"abcdefg"，字符a的内容被重写成了f。

0022FF49	0022FF4A	0022FF4B	0022FF4C	0022FF4D	0022FF4E	0022FF4F
b	s[0]	s[1]	s[2]	s[3]	s[4]	a

## C 语言变长数组：使用变量指明数组的长度

int m = 10, n;scanf("%d", &n);int a[m], b[n]; 在实际编程中，有时数组的长度不能提前确定，如果这个变化范围小，那么使用常量表达式定义一个足够大的数组就可以，如果这个变化范围很大，就可能会浪费内存，这时就可以使用变长数组。

变量的值在编译期间并不能确定，只有等到程序运行后，根据计算结果才能知道它的值到底是什么，所以数组长度中一旦包含了变量，那么数组长度在编译期间就不能确定了，也就不能为数组分配内存了，只有等到程序运行后，得到了变量的值，确定了具体的长度，才能给数组分配内存，我们将这样的数组称为变长数组(VLA, Variable Length Array)。普通数组(固定长度的数组)是在编译期间分配内存的，而变长数组是在运行期间分配内存的。注意，变长数组是说数组的长度在定义之前可以改变，一旦定义了，就不能再改变了，所以变长数组的容量也是不能扩大或缩小的，它仍然是静态数组（不能插入或删除，只能修改或读取）。以上面的代码为例，第 8 行代码是数组定义，此时就确定了数组的长度，在此之前长度可以随意改变，在此之后长度就固定了。

C99/C11标准规定，可以省略原型中的形参名，但是在这种情况下，必须用星号来代替省略的维度：

```
int sum2d(int, int, int ar[*][*]); // ar是一个变长数组（VLA），省略了维度  
形参名
```

## 一种“自欺欺人”的写法

有些初学者在使用变长数组时会像下面一样书写代码：

int n; int arr[n]; scanf("%d", &n); 先定义一个变量 n 和一个数组 arr，然后用 scanf() 读入 n 的值。有些初学者认为，scanf() 输入结束后，n 的值就确认下来了，数组 arr 的长度也随即确定了。这种想法看似合理，其实是蛮不讲理，毫无根据。从你定义数组的那一刻起(也就是第二行代码)，数组的长度就确定下来了，以后再也不会改变了；改变 n 的值并不会影响数组长度，它们之间没有任何“联动”关系。用 scanf() 读入 n 的值，影响的也仅仅是 n 本身，不会影响数组。那么，上面代码中数组的长度究竟是什么呢？鬼知道！n 是未初始化的局部变量，它的值是未知的。修改上面的代码，将数组的定义放到最后就没问题了：

```
for (int i = 0; i < row1; i++) // 初始化变长数组，不能直接初始化  
{  
    for (int j = 0; j < col2; j++)  
        mat3[i][j] = 0;  
}
```

变长数组不可以初始化，编译会出错，如果对变长数组赋值只能使用循环语句进行循环赋值

VM 不能具有文件作用域，存储连续性只能为 auto，这是因为编译器通常把全局对象存放于数据段，对象的完整信息必须在编译期内确定。

## C 语言对数组元素进行排序（冒泡排序法）

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）。

## 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。

```
1 #include <stdio.h>
2 int main()
3 {
4     int a[10], i, j, k;
5     for (i = 0; i < 10; i++)
6         scanf("%d", &a[i]); // 输入
7     for (j = 0; j < 9; j++) // n-1 次
8     {
9         for (i = 0; i < 9; i++) // 注意 i+1 的范围，否则会越界
10        {
11            if (a[i] > a[i + 1])
12            {
13                k = a[i + 1];
14                a[i + 1] = a[i];
15                a[i] = k; // 进行数值交换
16            }
17        }
18    }
19    for (i = 0; i < 10; i++)
20    {
21        printf("%d ", a[i]);
22    }
23 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```
23
45
34
456
234
34
1
2
3
4
1 2 3 4 23 34 45 234 456 fengsc@ubuntu:~/Desktop/c/xiti $ 
```

优化后的算法实现如下所示：

```
01. #include <stdio.h>
02. int main() {
03.     int nums[10] = {4, 5, 2, 10, 7, 1, 8, 3, 6, 9};
04.     int i, j, temp, isSorted;
05.
06.     //优化算法：最多进行 n-1 轮比较
07.     for(i=0; i<10-1; i++) {
08.         isSorted = 1; //假设剩下的元素已经排序好了
09.         for(j=0; j<10-1-i; j++) {
10.             if(nums[j] > nums[j+1]) {
11.                 temp = nums[j];
12.                 nums[j] = nums[j+1];
13.                 nums[j+1] = temp;
14.                 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. }
```

## 选择排序

```
2  public static void sort(int arr[])
3  {
4      for(int i=0;i<arr.length;i++){
5          int min = i;//最小元素的下标
6          for(int j=i+1;j<arr.length;j++){
7              if(arr[j] < arr[min]){
8                  min = j;//找最小值
9              }
10         }
11         //交换位置
12         int temp = arr[i];
13         arr[i] = arr[min];
14         arr[min] = temp;
15     }
}
```

冒泡排序是左右两个数相比较，而选择排序是用后面的数和每一轮的第一个数相比较；

冒泡排序每轮交换的次数比较多，而选择排序每轮只交换一次；

冒泡排序是通过数去找位置，选择排序是给定位置去找数；

当一个数组遇到相同的数时，冒泡排序相对而言是稳定的，而选择排序便不稳定；（序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中两个5的相对前后顺序就被破坏了，所以选择排序是一个不稳定的排序算法）

在时间效率上，选择排序优于冒泡排序。

## C 语言获取随机数

```
 srand(time(NULL)); rand();
```

运行结果：



```
1494945062
Process exited after 0.2199
请按任意键继续. . .
```

图4

我们看到输出的是一个很大的整数，那么这个整数代表的是什么呢？

它代表的意义是从1970年1月1日0时0分0秒0毫秒到你点击运行按钮时所经过的时间毫秒数（记住，单位是毫秒）。这个数有一个名字，叫做**时间戳**。

从这里我们可以知道rand()函数实际上需要一个整数的实参。而函数time(NULL)所返回的整数就是我们所说的随机种子。将此整数作为函数rand()的实参，实际上就是一个初始化随机种子的过程。

运行结果（每次运行结果都不一样）：

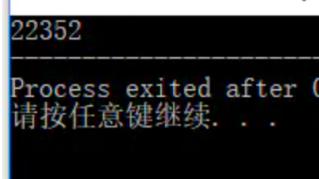


图6

我们看到结果也是一个整数（0~32767之间的整数）每调用一次rand函数产生的数都将是随机的（本质就是将随机种子进行数学运算（此算法没必要去了解）得到一个数，并将此数返回）。

如果不设置初始化种子，默认初始化种子为1。

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

编译后再运行几次，你会发现产生的随机数是相同的。实际上，rand() 函数产生的随机数是伪随机数，是根据一个数按照某个公式推算出来的，这个数我们称之为“种子”，但是这个种子在系统启动之后就是一个定值，我们需要用 srand() 来进行播种，即在int a前加一句：

```
1. srand((unsigned)time(NULL)); //这里利用时间进行播种，需要time.h
```

这样，我们就能得到不同的随机数，其实C语言中还有一个 random() 函数可以获取随机数，但是 random() 函数不是ANSI C标准，不能在VC等编译器通过，所以比较少用。

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  $50+13=63$ 。

从 X 到 Y，有  $Y-X+1$  个数，所以要产生从 X 到 Y 的数，只需要这样写：

```
k=rand()% (Y-X+1)+X;
```

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：

```
01. #include <stdio.h>
02. #include <stdlib.h>
03. #include <time.h>
04. int main() {
05.     int a, i;
06.     // 使用for循环生成10个随机数
07.     for (i = 0; i < 10; i++) {
08.         srand((unsigned)time(NULL));
09.         a = rand();
10.         printf("%d ", a);
11.     }
12.
13.     return 0;
14. }
```

纯文本 复制

运行结果举例：

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。

**推荐：**如果把 srand () 放在循环外，就能产生不同的不同的序列

或者 在 循 环 内 用 随 机 数 加 时 间 戳 做 种： srand((unsigned)time(NULL) + (unsigned)rand());

## 复制函数（可复制任意内容）

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n)
```

这两个函数都从 s2 指向的位置拷贝 n 字节到 s1 指向的位置，而且都返回 s1 的值。所不同的是， memcpy() 的参数带关键字 restrict，即 memcpy() 假设两个内存区域之间没有重叠；而 memmove() 不作这样的假设，所以拷贝过程类似于先把所有字节拷贝到一个临时缓冲区，然后再拷贝到最终目的地。如果使用 memcpy() 时，两区域出现重叠会怎样？其行为是未定义的，这意味着该函数可能正常工作，也可能失败。

由于这两个函数设计用于处理任何数据类型，所有它们的参数都是两个指向 void 的指针。  
**C 允许把任何类型的指针赋给 void \*类型的指针。如此宽容导致函数无法知道待拷贝数据的类型。因此，这两个函数使用第 3 个参数 指明待拷贝的字节数。**注意，对数组而言，字节数一般与元素个数不同。如果要拷贝数组中 10 个 double 类型的元素，要使用 10\*sizeof(double)，而不是 10

通常在复制字符串时用 strcpy()，而需要复制其他类型数据时则一般用 memcpy();

# C 语言函数

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：

```
01. #include <stdio.h>
02. #include <string.h>
03. int main(){
04.     char str1[] = "http://c.biancheng.net";
05.     char str2[] = "http://www.baidu.com";
06.     int result, i;
07.     //比较两个字符串大小
08.     for(i=0; (result = str1[i] - str2[i]) == 0; i++) {
09.         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。

```
01. #include <stdio.h>
02.
03. //将比较字符串大小的代码封装成函数，并命名为strcmp_alias
04. int strcmp_alias(char *s1, char *s2) {
05.     int i, result;
06.     for(i=0; (result = s1[i] - s2[i]) == 0; i++) {
07.         if(s1[i] == '\0' || s2[i] == '\0'){
08.             break;
09.         }
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。

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。

# C 语言函数定义

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。

当我们想修改一个函数或者变量的名字时候，我们只需把光标放到函数或者变量名上，然后按下 **F2**，这样这个函数或者变量出现的地方就都会被修改。

将代码段封装成函数的过程叫做**函数定义**。

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。

```
01. #include <stdio.h>
02.
03. int sum() {
04.     int i, sum=0;
05.     for(i=1; i<=100; i++) {
06.         sum+=i;
07.     }
08.     return sum;
09. }
10.
11. int main() {
12.     int a = sum();
13.     printf("The sum is %d\n", a);
14.     return 0;
15. }
```

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。

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：

```
1. void hello(){
2.     printf ("Hello,world \n");
3.     //没有返回值就不需要 return 语句
4. }
```

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。

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。

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。

```
01. #include <stdio.h>
02.
03. int sum(int m, int n) {
04.     int i, sum=0;
05.     for(i=m; i<=n; i++) {
06.         sum+=i;
07.     }
08.     return sum;
09. }
10.
11. int main() {
12.     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. }
```

纯文本 复制

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;
```

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：

```
01. #include <stdio.h>
02.
03. void func1() {
04.     printf("http://c.biancheng.net");
05.
06.     void func2() {
07.         printf("C语言小白变怪兽");
08.     }
09. }
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：

```
01. #include <stdio.h>
02.
03. void func2() {
04.     printf("C语言小白变怪兽");
05. }
06.
07. void func1() {
08.     printf("http://c.biancheng.net");
09.     func2();
10. }
11.
12. int main() {
13.     func1();
14.     return 0;
15. }
```

纯文本 复制

`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。

## 需要定义形参的原因

在 ANSI C 标准之前，声明函数的方案有缺陷，因为只需要声明函数的类型，不用声明任何参数。下面我们看一下使用旧式的函数声明会导致什么问题。

下面是 ANSI 之前的函数声明，告知编译器 `imin()` 返回 `int` 类型的值：

```
int imin();
```

然而，以上函数声明并未给出 `imin()` 函数的参数个数和类型。因此，如果调用 `imin()` 时使用的参数个数不对或类型不匹配，编译器根本不会察觉出来。

由于不同系统的内部机制不同，所以出现问题的

具体情况也不同。下面介绍的是使用 PC 和 VAX 的情况。主调函数把它的参数储存在被称为栈（stack）的临时存储区，被调函数从栈中读取这些参数。

对于该例，这两个过程并未相互协调。主调函数根据函数调用中的实际参数决定传递的类型，而被调函数根据它的形式参数读取值。因此，函数调用 `imax(3)` 把一个整数放在栈中。当 `imax()` 函数开始执行时，它从栈中读取两个整数。而实际上栈中只存放了一个待读取的整数，所以读取的第 2 个值是当时恰好在栈中的其他值。

第 2 次使用 `imax()` 函数时，它传递的是 `float` 类型的值。这次把两个 `double` 类型的值放在栈中（回忆一下，当 `float` 类型被作为参数传递时会被升级为 `double` 类型）。在我们的系统中，两个 `double` 类型的值就是两个 64 位的值，所以 128 位的数据被放在栈中。当 `imax()` 从栈中读取两个 `int` 类型的值时，它从栈中读取前 64 位。在我们的系统中，每个 `int` 类型的变量占用 32 位。这些数据对应两个整数，其中较大的是 3886。

一个支持 ANSI C 的编译器会假定用户没有用函数原型来声明函数，它将不会检查参数。为了表明函数确实没有参数，应该在圆括号中使用 `void` 关键字：

```
void print_name(void);
```

支持 ANSI C 的编译器解释为 `print_name()` 不接受任何参数。然后在调用该函数时，编译器会检查以确保没有使用参数。

一些函数接受（如，`printf()` 和 `scanf()`）许多参数。例如对于 `printf()`，第 1 个参数是字符串，但是其余参数的类型和数量都不固定。对于这种情况，

ANSI C 允许使用部分原型。例如，对于 `printf()` 可以使用下面的原型：

```
int printf(const char *, ...);
```

这种原型表明，第 1 个参数是一个字符串（第 11 章中将详细介绍），可能还有其他未指定的参数。

C 库通过 `stdarg.h` 头文件提供了一个定义这类（形参数量不固定的）函数的标准方法

## 形参和实参

### 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。

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。**

**当 float 类型被作为参数传递时会被升级为 double 类型**

```
09.     return m;
10. }
11.
12. int main() {
13.     int a, b, total;
14.     printf("Input two numbers: ");
15.     scanf("%d %d", &a, &b);
16.     total = sum(a, b);
17.     printf("a=%d, b=%d\n", a, b);
18.     printf("total=%d\n", total);
19.
20.     return 0;
21. }
```

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：

1. total = sum(10, 98); //将常量作为实参
2. total = sum(a+10, b-3); //将表达式作为实参
3. 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：

```
01. #include <stdio.h>
02.
03. //计算从m加到n的值
04. int sum(int m, int n) {
05.     int i;
06.     for (i = m + 1; i <= n; ++i) {
07.         m += i;
08.     }
09.     return m;
10. }
11.
12. int main() {
13.     int m, n, total;
```

## 返回值和 EXIT

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。（可以用来赋值）

**return** 语句的一般形式为：

**return** 表达式；

**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：

```
1. void func(){
2.     printf("http://c.biancheng.net\n");
3. }
```

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。**

**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）

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：

```
1. //返回两个整数中较大的一个
2. int max(int a, int b){
3.     return (a>b) ? a : b;
4.     printf("Function is performed\n");
5. }
```

第 4 行代码就是多余的，永远没有执行的机会。

**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

1. 返回值类型应注意与定义函数时的 datatype 保持一致，如果类型不一样，会强制类型转换为定义函数地类型的值然后返回。

C 语言的规则, 所有没有函数原型的函数返回值都为 int。

exit() 函数关闭所有打开的文件并结束程序。exit() 的参数被传递给一些 操作系统，包括 UNIX、Linux、Windows 和 MS-DOS，以供其他程序使用。通常的惯例是：正常结束的程序传递 0，异常结束的程序传递非零值。不同的退出值可用于区分程序失败的不同原因，这也是 UNIX 和 DOS 编程的通常做法。但是，并不是所有的操作系统都能识别相同范围内的返回值。因此，C 标准规定了一个最小的限制范围。尤其是，标准要求 0 或宏 EXIT\_SUCCESS 用于表明成功结束程序，宏 EXIT\_FAILURE 用于表明结束程序失败。这些宏和 exit() 原型都位于 stdlib.h 头文件中。根据 ANSI C 的规定，在最初调用的 main() 中使用 return 与调用 exit() 的效果相同。因此，在 main()，下面的语句：return 0; 和下面这条语句的作用相同：exit(0); 但是要注意，我们说的是“最初的调用”。如果 main() 在一个递归程序中，exit() 仍然会终止程序，但是 return 只会把控制权交给上一级递归，直至最初的一级。然后 return 结束程序。return 和 exit() 的另一个区别是，即使在其 他函数中（除 main() 以外）调用 exit() 也能结束整个程序。

## 函数调用

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：

```
01. //函数作为表达式中的一项出现在表达式中
02. z = max(x, y);
03. m = n + max(x, y);
04. //函数作为一个单独的语句
05. printf("%d", a);
06. scanf("%d", &b);
07. //函数作为调用另一个函数时的实参
08. printf( "%d", max(x, y) );
09. total( max(x, y), min(m, n) );
```

```

1 #include <stdio.h>
2 long factorial(int n)
3 {
4     long result = 1;
5     for (; n >= 1; n--)
6         result *= n;
7     return result;
8 }
9 long sum(int n)
10 {
11     long sum = 0;
12     for (int i = 1; i <= n; i++)
13         sum += factorial(i);
14     return sum;
15 }
16 int main()
17 {
18     int n;
19     scanf("%d", &n);
20     printf("从1到%d的阶乘之和为%ld\n", n, sum(n));
21     return 0;
22 }
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

```

cd "/home/fengsc/Desktop/c/func/" && gcc func2.c -o func2
esktop/c $ cd "/home/fengsc/Desktop/c/func/" && gcc func2.c
10
从1到10的阶乘之和为4037913

```

调试时遵循从外到内

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。

## 函数声明

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**

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。**

**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。**

```
#include <stdio.h>
long factorial(int);
long sum(int);
int main()
{
    int n;
    scanf("%d", &n);
    printf("从1到%d的阶乘之和为%ld\n", n, sum(n));
    return 0;
}
```

## 全局变量和局部变量

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。

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

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（块级变量），它的作用域只限于当前语句块（例如 for 里面）。

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

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。

在函数中不存在局部变量时，编译器只能到函数外部，也就是全局作用域中去寻找变量。

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。

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（C 语言中函数可以修改全局变量的值，前提是 **没有定义同名变量**）。

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。

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。

## 递归函数

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。

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：

$$n! = \begin{cases} 1 & (n = 0, 1) \\ n * (n - 1)! & (n > 1) \end{cases}$$

其递归公式为  $f(n)=n*f(n-1)(n!=0,1)$

```
1. long factorial(int n) {
2.     if (n == 0 || n == 1) {
3.         return 1;
4.     }
5.     else {
6.         return factorial(n - 1) * n; // 递归调用
7.     }
8. }
```

## 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  $\text{factorial}(n-1) * n$ ，也即执行  $\text{factorial}(4) * 5$ 。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  $\text{factorial}(4)$  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。

## 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。

### 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。

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。

即—fun1()调用

fun2()、fun2()调用 fun3()、fun3()调用 fun4()。当 fun4()结束时，控制传回

fun3()；当 fun3()结束时，控制传回

fun2()；当 fun2()结束时，控制传回

fun1()。递归的情况与此类似，只不过 fun1()、fun2()、fun3()和 fun4()都是相同的函数

```
5 |     int fact(int n)
6 |     {
7 |         if (n < 0)
8 |             return 0;
9 |         else if (n == 0)
10 |             return 1;
11 |         else if (n == 1)
12 |             return 1;
13 |         else
14 |             return n * fact(n - 1);
15 |     }
16 |
17 |     int facttail(int n, int a)
18 |     {
19 |
20 |         if (n < 0)
21 |             return 0;
22 |         else if (n == 0)
23 |             return 1;
24 |         else if (n == 1)
25 |             return a;
26 |         else
27 |             return facttail(n - 1, n * a);
28 |     }

```

## 例题

1,

有一只猴子，第一天摘了若干个桃子，当即吃了一半，但还觉得不过瘾，就又多吃了1个。第2天早上又将剩下的桃子吃掉一半，还是觉得不过瘾，就又多吃了2个。以后每天早上都吃了前一天剩下的一半加1个（例如，第5天吃了前一天剩下的一般加5个）。到第n天早上再想吃的时候，就只剩下一个桃子了。

输入：

```
1 | 天数n
```

输出：

```
1 | 第一天的桃子个数
```

```
1 #include<stdio.h>
2
3 int n; // 定义全局变量n，记录输入天数
4
5 int Taozi( int day ) // 递归函数。
6 {
7     int num;
8     if( day == n )
9     {
10         return 1; // 递归终点。
11     }
12     else
13     {
14         num = 2 * ( Taozi( day + 1 ) + day ); // 递推公式
15     }
16     return num; // 返回num的值
17 }
18 int main()
19 {
20     int num; // 此处num是局部变量，与递归函数中的num 意义不同
21     scanf("%d", &n); // 输入全局变量n的值
22     num = Taozi( 1 ); // 因为是从第一天开始算，传入的参数值为 1。
23     if(num > 1)
24         printf("The monkey got %d peaches in first day.\n", num);
25     else
26         printf("The monkey got %d peach in first day.\n", num); // 等于1的情况
}

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: Code

```
fengsc@ubuntu:~/Desktop/c/func $ cd "/home/fengsc/Desktop/c/func/" && gcc func3.c -o /c/func/"func3
5
The monkey got 290 peaches in first day.
```

2, 请使用递归算法求下列序列的前 n 项之和。

$1 + 1/2 - 1/3 + 1/4 - 1/5 \dots$

```
3 double plus(int x) // 这是一个返回值为浮点数的函数，写在后面的话要进行函数申明。
4 {
5     double sum = 0;
6     if(x == 1) // 递归终点
7     {
8         return 1;
9     }
10    else
11    {
12        if(x % 2 == 0) // 当为偶数项时，符号为正
13            sum = 1.0 / x + plus(x - 1);
14        else // 当为奇数项时，符号为负（首项除外）
15            sum = -1.0 / x + plus(x - 1);
16    }
17    return sum;
18 }
```

3,

请编写一个递归函数 reverse(char str[], int start, int end) , 该函数的功能是将串 str 中下标从 start 开始到 end 结束的字符颠倒顺序。假设 start 和 end 都在合理的取值范围。

例如：

执行前：str[]="0123456"; start=1 ; end=4

执行后：str[]="0432156"

```
if(end <= start)
{
    return; //递归终点
}
else
{
    temp = str[start];
    str[start] = str[end];
    str[end] = temp; //交换字符
    reverse(str, start + 1, end - 1); //递归调用
}
```

4, 判断回文字符串

```
if (length == 1 || length == 0)//(length<=1)
    return 1;
else
{
    if (arr[low] == arr[high]) //判断对应位置的字符是否相同
    {
        return (palindrome(low + 1, high - 1, str, length - 2));
        //这里返回的是下一次调用的返回值，这样只有当所有对应字符相同时才返回 1。
    }
    else
        return 0;
}
int main()
{
    int length, low = 0, high;
    fgets(arr, 10000, stdin);
    length = strlen(arr);
    high = length - 2; //注意fgets
    //if (arr[length - 1] == '\n')
    // arr[length - 1] = '\0';//对此程序无影响，但最好加上
    if (palindrome(low, high, arr, length))
    {
        printf("Yes\n");
    }
}
```

判断回文数

```
if(n==0)
    return 1;
else
{
    i *= 10;
    i += n%10;
    isPalindrom(n/10);
}

void main()
{
    int number;
    scanf("%d",&number);
    isPalindrom(number);

    if(i==number)
```

当一个非常简单的程序使用递归实现时，系统会分配大量的内存。

这是因为，每一次递归的实现中，系统都会重新为变量分配空间而不是覆盖原来的空间。

因此，当问题没有特别复杂，并不一定需要使用到递归程序时，应当避免使用递归程序，尤其是递归次数多的程序，可能会造成内存分配的崩溃。

判断回文数最终版

```
int main(void)
{
    int a[10], b; //未初始化时
    for (int j = 0; j < 10; j++)
    {
        //scanf("%d", &a[j]);
        b = scanf("%d", &a[j]);
        switch (b)
        {
        case 0:
        {
            scanf("%*[^\n]*c");
            printf("error\n");
            break;
        }
        case EOF:
            break ;
        default:
        {
            if (invert(a[j]) == a[j])
                printf("yes");
            else
                printf("no");
            printf("\n");
            i = 0;
        }
    }
}
```

## scanf 返回值和数组默认值

数组使用时一般应初始化为 0，否则会出现一些奇怪的现象

以 a[10]为例，其默认值为

-224804920

32623

1112941008

21871

0

0

1112940672

21871

-1471595872

32764

0 是奇怪 yes 的来源

```
no
no
no
yes
yes
no
no
no
no
```

返回成功匹配的个数，输入 scanf 不能识别的数据类型时 scanf 会关闭输入流，每次都会读到 a，都不能赋值，直到 a 被取走，返回值为 0

```
a
error
error
error
error
error
error
error
error
error
```

如果遇到错误或遇到 end of file，返回值为 EOF，且返回值为 int 型（一般为 -1），同时关闭输入流。

```
int a[10], b;//未初始化时
for (int j = 0; j < 10; j++)
{
    //scanf("%d", &a[j]);
    b = scanf("%d", &a[j]);
    printf("%d", b);
    if (b == 0)
    {
        //printf("%d\n", b);
        scanf("%*[^\n]%*c");//清空缓冲区中的不能接收字符，重新打开输入流
        printf("error\n");
        continue;           //if (scanf("%d", &a[j]) == EOF)//用文件作为输入源时去除空行的影响
    }
    else
    {
        if (a[j] == a[j])
            //do something
    }
}
```

# main 函数详解和命令行参数

在最新的 C99 标准中，只有以下两种定义方式是正确的：

```
int main( void ) /* 无参数形式 */
{
    ...
    return 0;
}
int main( int argc, char *argv[] ) /* 带参数形式 */
{
    ...
    return 0;
}
```

int 指明了 main() 函数的返回类型，函数名后面的圆括号一般包含传递给函数的信息。void 表示没有给函数传递参数。关于带参数的形式，我们等会讨论。

**void main()**

有些编译器允许这种形式，但是还没有任何标准考虑接受它。C++ 之父 Bjarne Stroustrup 在他的主页上的 FAQ 中明确地表示：void main() 的定义从来就不存在于 C++ 或者 C。所以，编译器不必接受这种形式，并且很多编译器也不允许这么写。

坚持使用标准的意义在于：**当你把程序从一个编译器移到另一个编译器时，照样能正常运行。**

&& 的含义是：如果 && 前面的程序正常退出，则继续执行 && 后面的程序，否则不执行。所以，要是把 a.c 里面的 return 0; 删除或者改为 return 99;，那么你只能看到 I love you.。也就是说，程序 b.c 就不执行了。现在，大家该明白 return 0; 的作用了吧。（不会影响单个程序的运行）**用\$?查看返回值**

C 编译器允许 main() 函数没有参数，或者有两个参数（有些实现允许更多的参数，但这只是对标准的扩展）。这两个参数，一个是 int 类型，一个是字符串类型。第一个参数是命令行中的字符串数。按照惯例（但不是必须的），这个 int 参数被称为 argc (argument count)。大家或许现在才明白这个形参为什么要取这么个奇怪的名字吧，呵呵！至于英文的意思，自己查字典吧。第二个参数是一个指向字符串的指针数组。命令行中的每个字符串被存储到内存中，并且分配一个指针指向它。按照惯例，这个指针数组被称为 argv (argument value)。系统使用空格把各个字符串隔开。一般情况下，把程序本身的名字赋值给 argv[0]，接着，把最后的第一个字符串赋给 argv[1]，等等。

linux 中使用方法：./xxx argv[1] argv[2] .....

repeat "I am hungry" now

**这行命令把字符串"I am hungry"赋给 argv[1]，把"now"赋给 argv[2]**

如果运行该程序时没有提供命令行参数，那么 argc < 2 为真，程序给出一条提示信息后结束。如果 times 为 0 或负数，情况也是如此。C 语言逻辑运算符的求值顺序保证了如果 argc < 2，就不会对 atoi(argv[1]) 求值。

**使用 if (argc>1) 等可以使未输入参数而又需要对参数进行操作时不报错**

## 将字符串转化为数字 (atoi)

如果字符串仅以整数开头， atoi() 函数也能处理，它只把开头的整数转换为字符。例如 atoi("42regular") 将返回整数 42。如果在命令行输入 hello what 会怎样？在我们所用的 C 实现中，如果命令行参数不是数字， atoi() 函数返回 0。然而 C 标准规定，这种情况下的是未定义的。因此，使用有错误检测功能的 strtol() 函数（马上介绍）会更安全。

该程序中包含了 stdlib.h 头文件，因为从 ANSI C 开始，该头文件中包含了 atoi() 函数的原型。除此之外，还包含了 atof() 和 atol() 函数的原型。atof() 函数把字符串转换成 double 类型的值，atol() 函数把字符串转换成 long 类型的值。atof() 和 atol() 的工作原理和 atoi() 类似，因此它们分别返回 double 类型和 long 类型。

ANSI

C 还提供一套更智能的函数：strtol() 把字符串转换成 long 类型的值，strtoul() 把字符串转换成 unsigned long 类型的值，strtod() 把字符串转换成 double 类型的值。这些函数的智能之处在于识别和报告字符串中的首字符是否是数字。而且，strtol() 和 strtoul() 还可以指定数字的进制。

下面的程序示例中涉及 strtol() 函数，其原型如下：

```
long strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

这里，nptr 是指向待转换字符串的指针，endptr 是一个指针的地址，该指针被设置为标识输入数字结束字符的地址，base 表示以什么进制写入数字

首先注意，当 base 分别为 10 和 16 时，字符串 "10" 分别被转换成数字 10 和 16。还要注意，如果 end 指向一个字符，\*end 就是一个字符。因此，第 1 次转换在读到空字符时结束，此时 end 指向空字符。打印 end 会显示一个空字符串，以 %d 转换说明输出 \*end 显示的是空字符的 ASCII 码。

对于第 2 个输入的字符串，当 base 为 10 时，end 的值是 'a' 字符的地址。所以打印 end 显示的是字符串 "atom"，打印 \*end 显示的是 'a' 字符的 ASCII 码。然而，当 base 为 16 时，'a' 字符被识别为一个有效的十六进制数，strtol() 函数把十六进制数 10a 转换成十进制数 266

## 将数字转化为字符串(sprintf)(linux 没有 itoa)

sprintf() 函数用于将格式化的数据写入字符串，其原型为：

```
int sprintf(char *str, char * format [, argument, ...]);
```

sprintf() 最常见的应用之一莫过于把整数打印到字符串中，如：

```
sprintf(s, "%d", 123); // 把整数 123 打印成一个字符串保存在 s 中
```

```
sprintf(s, "%8x", 4567); // 小写 16 进制，宽度占 8 个位置，右对齐
```

sprintf 的作用是将一个格式化的字符串输出到一个目的字符串中，而 printf 是将一个格式化的字符串输出到屏幕。sprintf 的第一个参数应该是目的字符串，如果不指定这个参数，执行过程中出现 "该程序产生非法操作,即将被关闭...." 的提示。

如果成功，则返回写入的字符总数，不包括字符串追加在字符串末尾的空字符。如果失败，则返回一个负数。

## 将数字 0~9 和字符转化（切记切记）

方法：‘0’ + 需要转换的数字

例：将数字 6 转换成字符 ‘6’

'0' + 6 = '6'

0 的 ASCII 码是 48

方法：需要转换的数字 - ‘0’

例：将字符 ‘6’ 转换成数字 6

'6' - '0' = 6

## assert 函数详解

断言函数，用于在调试过程中捕捉程序的错误。

“断言”在语文中的意思是“断定”、“十分肯定地说”，在编程中是指对某种假设条件进行检测，如果条件成立就不进行任何操作，如果条件不成立就捕捉到这种错误，并打印出错误信息，终止程序执行。

assert() 会对表达式 expression 进行检测：

- 如果 expression 的结果为 0（条件不成立），那么断言失败，表明程序出错，assert() 会向标准输出设备（一般是显示器）打印一条错误信息，并调用 abort() 函数终止程序的执行。
- 如果 expression 的结果为非 0（条件成立），那么断言成功，表明程序正确，assert() 不进行任何操作。

要打印的错误信息的格式和内容没有统一的规定，这和标准库的具体实现有关（也可以说和编译器有关），但是错误信息至少应该包含以下几个方面的信息：

- 断言失败的表达式，也即 expression；
- 源文件名称；
- 断言失败的代码的行号。

大部分编译器的格式如下所示：

Assertion failed: expression, file filename, line number

assert() 的用法很简单，我们只要传入一个表达式，它会计算这个表达式的结果：如果表达式的結果为“假”，assert() 会打印出断言失败的信息，并调用 abort() 函数终止程序的执行；如果表达式的結果为“真”，assert() 就什么也不做，程序继续往后执行。

一旦定义了 NDEBUG 宏，assert() 就无效了。

NDEBUG 是“No Debug”的意思，也即“非调试”。有的编译器（例如 Visual Studio）

在发布（Release）模式下会定义 NDEBUG 宏，在调试（Debug）模式下不会定义这个宏；有的编译器（例如 Xcode）在发布模式和调试模式下都不会定义 NDEBUG 宏，这样当我们以发布模式编译程序时，就必须自己在编译参数中增加 NDEBUG 宏，或者在包含 <assert.h> 头文件之前定义 NDEBUG 宏。

调试模式是程序员在测试代码期间使用的编译模式，发布模式是将程序提供给用户时使用的编译模式。**在发布模式下，我们不应该再依赖 assert() 宏，因为程序一旦出错，assert() 会抛出一段用户看不懂的提示信息，并毫无预警地终止程序执行，这样会严重影响软件的用户体验，所以在发布模式下应该让 assert() 失效。**

C11 新增了一个特性：**\_Static\_assert** 声明，可以在编译时检查 assert() 表达式。因此，assert() 可以导致正在运行的程序中止，而 \_Static\_assert() 可以导致程序无法通过编译。  
**\_Static\_assert()** 接受两个参数。第 1 个参数是整型常量表达式，第 2 个参数是一个字符串。如果第 1 个表达式求值为 0（或 **\_False**），编译器会显示字符串，而且不编译该程序。  
`_Static_assert(sizeof(double) == 2 * sizeof(int), "double not twice int size");`

## C 语言可变参数函数

C 语言中最常用的可变参数函数例子是 **printf ()** 和 **scanf ()**。这两个函数都有一个强制参数，即格式化字符串。格式化字符串中的转换修饰符决定了可选参数的数量和类型。

对于每一个强制参数来说，函数头部都会显示一个适当的参数，像普通函数声明一样。参数列表的格式是强制性参数在前，后面跟一个逗号和省略号 (...)，这个省略号代表可选参数。

可变参数函数要获取可选参数时，必须通过一个类型为 **va\_list** 的对象，它包含了参数信息。这种类型的对象也称为参数指针（argument pointer），它包含了栈中至少一个参数的位置。可以使用这个参数指针从一个可选参数移动到下一个可选参数，由此，函数就可以获取所有的可选参数。**va\_list** 类型被定义在头文件 **stdarg.h** 中。

调用 **va\_end(ap)** 后，只有用 **va\_start** 重新初始化 **ap** 后，才能使用变量 **ap**。  
因为 **va\_arg()** 不提供退回之前参数的方法，所以有必要保存 **va\_list** 类型变量的副本。  
`va_copy(apcopy, ap); // 把 apcopy 作为 ap 的副本`

当编写支持参数数量可变的函数时，必须用 va\_list 类型定义参数指针，以获取可选参数。在下面的讨论中，va\_list 对象被命名为 argptr。可以用 4 个宏来处理该参数指针，这些宏都定义在头文件 stdarg.h 中：

```
01. void va_start(va_list argptr, lastparam);
```

宏 va\_start 使用第一个可选参数的位置来初始化 argptr 参数指针。该宏的第二个参数必须是该函数最后一个有名称参数的名称。必须先调用该宏，才可以开始使用可选参数。

```
01. type va_arg(va_list argptr, type);
```

展开宏 va\_arg 会得到当前 argptr 所引用的可选参数，也会将 argptr 移动到列表中的下一个参数。宏 va\_arg 的第二个参数是刚刚被读入的参数的类型。

```
01. void va_end(va_list argptr);
```

当不再需要使用参数指针时，必须调用宏 va\_end。如果想使用宏 va\_start 或者宏 va\_copy 来重新初始化一个之前用过的参数指针，也必须先调用宏 va\_end。

```
01. void va_copy(va_list dest, va_list src);
```

宏 va\_copy 使用当前的 src 值来初始化参数指针 dest。然后就可以使用 dest 中的备份获取可选参数列表，从 src 所引用的位置开始。

```
01. // 函数add() 计算可选参数之和
02. // 参数：第一个强制参数指定了可选参数的数量，可选参数为double类型
03. // 返回值：和值，double类型
04. double add( int n, ... )
05. {
06.     int i = 0;
07.     double sum = 0.0;
08.     va_list argptr;
09.     va_start( argptr, n );           // 初始化argptr
10.    for ( i = 0; i < n; ++i )       // 对每个可选参数，读取类型为double的参数，
11.        sum += va_arg( argptr, double ); // 然后累加到sum中
12.    va_end( argptr );
13.    return sum;
14. }
```

va\_arg 代表每个参数，运行一次后 arg\_list 会指向下一个参数

## 预处理命令（宏定义和条件编译）

### 预处理命令

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。

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）。

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。

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：

```
01. #include <stdio.h>
02.
03. //不同的平台下引入不同的头文件
04. #if _WIN32 //识别windows平台
05. #include <windows.h>
06. #elif __linux__ //识别linux平台
07. #include <unistd.h>
08. #endif
09.
10. int main() {
11.     //不同的平台下调用不同的函数
12.     #if _WIN32 //识别windows平台
13.         Sleep(5000);
14.     #elif __linux__ //识别linux平台
15.         sleep(5);
16.     #endif
17.
18.     puts("http://c.biancheng.net/");
19.
20.     return 0;
21. }
```

#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。

## \_WIN32

## \_\_linux\_\_

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。

# #include

#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。

一般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。
- 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。

my.c 所包含的代码：

```
01. //计算从m加到n的和
02. int sum(int m, int n) {
03.     int i, sum = 0;
04.     for (i = m; i <= n; i++) {
05.         sum += i;
06.     }
07.     return sum;
08. }
```

my.h 所包含的代码：

```
01. //声明函数
02. int sum(int m, int n);
```

main.c 所包含的代码：

```
01. #include <stdio.h>
02. #include "my.h"
03.
04. int main() {
05.     printf("%d\n", sum(1, 100));
06.     return 0;
07. }
```

在头文件中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。

# 宏定义

#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

#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替换，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。

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，这只是一个简单粗暴的替换。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：

```
01. #define PI 3.14159
02.
03. int main() {
04.     // Code
05.     return 0;
06. }
07.
08. #undef PI
09.
10. void func() {
11.     // Code
12. }
```

纯文本

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

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。

### 带参宏定义

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。

相当于定义一个函数

【示例】输出两个数中较大的数。

```
01. #include <stdio.h>  
02. #define MAX(a,b) (a>b) ? a : b  
03. int main()  
04. {  
05.     int x , y, max;  
06.     printf("input two numbers: ");  
07.     scanf("%d %d", &x, &y);  
08.     max = MAX(x, y);  
09.     printf("max=%d\n", max);  
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;
```

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。

```
01. #include <stdio.h>
02. #define SQ(y) (y)*(y)
03. int main() {
04.     int a, sq;
05.     printf("input a number: ");
06.     scanf("%d", &a);
07.     sq = SQ(a+1);
08.     printf("sq=%d\n", sq);
09.     return 0;
10. }
```

纯文本 复制

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：

```
01. #include <stdio.h>
02. #define SQ(y) y*y
03. int main() {
04.     int a, sq;
05.     printf("input a number: ");
06.     scanf("%d", &a);
07.     sq = SQ(a+1);
08.     printf("sq=%d\n", sq);
09.     return 0;
10. }
```

纯文本 复制

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：

```
01. #include <stdio.h>
02. #define SQ(y) (y)*(y)
03. int main()
04. {
05.     int a, sq;
06.     printf("input a number: ");
07.     scanf("%d", &a);
08.     sq = 200 / SQ(a+1);
09.     printf("sq=%d\n", sq);
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：

```
01. #include <stdio.h>
02. #define SQ(y) ((y)*(y))
03. int main()
04. {
05.     int a, sq;
06.     printf("input a number: ");
07.     scanf("%d", &a);
08.     sq = 200 / SQ(a+1);
09.     printf("sq=%d\n", sq);
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。

## 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②】用宏计算平方值。

```
01. #include <stdio.h>
02.
03. #define SQ(y) ((y)*(y))
04.
05. int main() {
06.     int i=1;
07.     while(i<=5) {
08.         printf("%d^2 = %d\n", i, SQ(i++));
09.     }
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，不同项中间用；隔开

## 宏参数中#和##的用法

# 用来将宏参数转换为字符串，也就是在宏参数的开头和末尾添加引号。例如有如下宏定义：

```
#define STR(s) #s
```

那么：

```
printf("%s", STR(c.biancheng.net));
printf("%s", STR("c.biancheng.net"));
```

分别被展开为：

```
printf("%s", "c.biancheng.net");
printf("%s", "\"c.biancheng.net\"");
```

## 称为连接符，用来将宏参数或其他的串连接起来。例如有如下的宏定义：

```
#define CON1(a, b) a##e##b  
#define CON2(a, b) a##b##00
```

那么：

```
printf("%f\n", CON1(8.5, 2));  
printf("%d\n", CON2(12, 34));
```

将被展开为：

```
printf("%f\n", 8.5e2);  
printf("%d\n", 123400);
```

## 几个预定义宏

- `__LINE__`: 表示当前源代码的行号；
- `__FILE__`: 表示当前源文件的名称；
- `__DATE__`: 表示当前的编译日期；
- `__TIME__`: 表示当前的编译时间；
- `__STDC__`: 当要求程序严格遵循 ANSI C 标准时该标识被赋值为 1；
- `__cplusplus`: 当编写 C++ 程序时该标识符被定义。

```
printf("Date : %s\n", __DATE__);  
printf("Time : %s\n", __TIME__);  
printf("File : %s\n", __FILE__);  
printf("Line : %d\n", __LINE__);  
system("pause");  
return 0;  
}
```

VS下的输出结果：

Date : Mar 6 2016

Time : 11:47:15

File : main.c

Line : 8

# 条件编译详解

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。

#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。

即为假的不进行编译，if 语句内容都会被编译

#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。

#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。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifndef WIN32
    #include <windows.h>
    #include <io.h>
#else
    #include <unistd.h>
    #include <sys/time.h>
    #inclu.....
```

再如，两个宏都存在时编译代码A，否则编译代码B：

```
01. #include <stdio.h>
02. #define NUM1 10
03. #define NUM2 20
04. int main(){
05.     #if (defined NUM1 && defined NUM2)
06.         //代码A
07.         printf("NUM1: %d, NUM2: %d\n", NUM1, NUM2);
08.     #else
09.         //代码B
10.         printf("Error\n");
11.     #endif
12.     return 0;
13. }
```

纯文本 复制

运行结果：

NUM1: 10, NUM2: 20

#ifdef 可以认为是 #if defined 的缩写。

#error 指令用于在编译期间产生错误信息，并阻止程序的编译，其形式如下：

```
#error error_message
```

例如，我们的程序针对Linux编写，不保证兼容Windows，那么可以这样做：

```
#ifdef WIN32
#error This programme cannot compile at Windows Platform
#endif
```

WIN32 是Windows下的预定义宏。当用户在Windows下编译该程序时，由于定义了WIN32这个宏，所以会执行

#error 命令，提示用户发生了编译错误，错误信息是：

```
This programme cannot compile at Windows Platform
```

## 小结

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。

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。

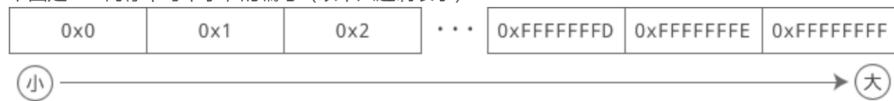
[#if 和 #ifdef 的区别](#)

# 指针

## 定义和使用

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：

```
01. #include <stdio.h>
02.
03. int main() {
04.     int a = 100;
05.     char str[20] = "c.biancheng.net";
06.     printf("%#X, %#X\n", &a, str);
07.     return 0;
08. }
```

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 &。

### 或者用%p

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

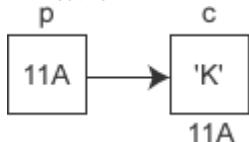
变量名和函数名为我们提供了方便，让我们在编写代码的过程中可以使用易于阅读和理解的英文字符串，不用直接面对二进制地址，那场景简直让人崩溃。

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（不用&）**。

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：

1. 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：

1. int a = 100;

2. 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

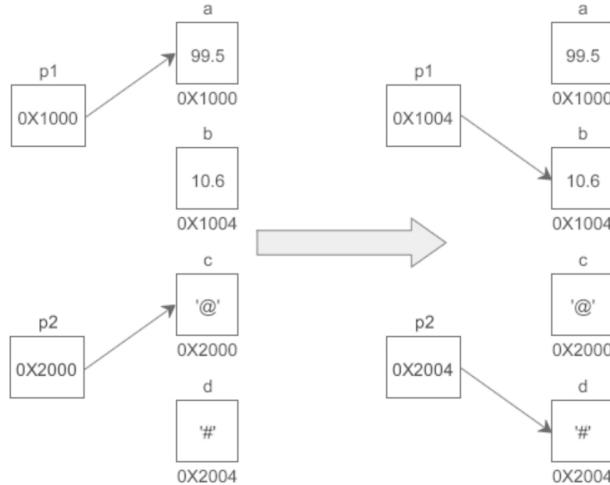
```

01. //定义普通变量
02. float a = 99.5, b = 10.6;
03. char c = '@', d = '#';
04. //定义指针变量
05. float *p1 = &a;
06. char *p2 = &c;
07. //修改指针变量的值
08. p1 = &b;
09. 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：

```
01. 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：

```
01. int *a, b, c;
```

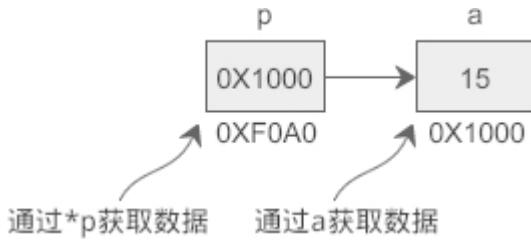
**即使\*靠近 int，b，c 也是普通变量，多个定义时必须按 01 方式，单个无所谓**指针变量存储了数据的地址，通过指针变量能够获得该地址上的数据，格式为：

**\*pointer;**

这里的\*称为指针运算符（对应解引用操作），用来取得某个地址上的数据

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` 本身的数据，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：

```

01. #include <stdio.h>
02.
03. int main() {
04.     int a = 15, b = 99, c = 222;
05.     int *p = &a; //定义指针变量
06.     *p = b; //通过指针变量修改内存上的数据
07.     c = *p; //通过指针变量获取内存上的数据
08.     printf("%d, %d, %d, %d\n", a, b, c, *p);
09.     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。

需要注意的是，**给指针变量本身赋值时不能加\***

指针变量也可以出现在普通变量能出现的任何表达式中

假设有一个 `int` 类型的变量 `a`，`pa` 是指向它的指针，那么`*&a` 和 `&*&a` 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**。

`&*&a` 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`。

在我们目前所学到的语法中，星号\*主要有三种用途：

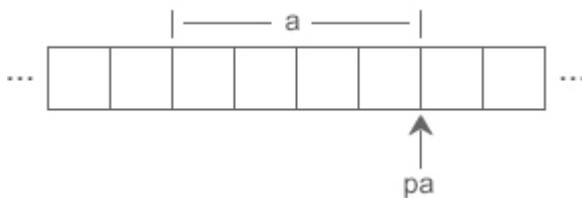
- 表示乘法，例如 `int a = 3, b = 5, c; c = a * b;`，这是最容易理解的。
- 表示定义一个指针变量，以和普通变量区分开，例如 `int a = 100; int *p = &a;`。
- 表示获取指针指向的数据，是一种间接操作，例如 `int a, b, *p = &a; *p = 100; b = *p;`。

## 指针变量的运算

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

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。

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 就表示指向下一个元素，这样指针的加减运算就具有了现实的意义

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。

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。

数据一般不要对指向普通变量的指针进行加减运算。

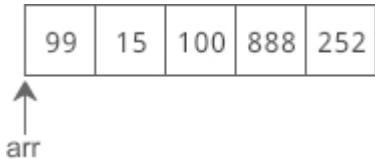
另外需要说明的是，不能对指针变量进行乘法、除法、取余等其他运算，除了会发生语法错误，也没有实际的含义

## 数组指针详解

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：

99	15	100	888	252
----	----	-----	-----	-----

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（在用 `sizeof` 求长度的时候会有区别）。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

下面的例子演示了如何以指针的方式遍历数组元素：

```

01. #include <stdio.h>
02.
03. int main() {
04.     int arr[] = { 99, 15, 100, 888, 252 };
05.     int len = sizeof(arr) / sizeof(int); //求数组长度
06.     int i;
07.     for(i=0; i<len; i++) {
08.         printf("%d ", *(arr+i) ); /*(arr+i) 等价于 arr[i]
09.     }
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。

[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。

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` 也总是指向下一个元素。

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。

```
int *p = &arr[2]; //也可以写作 int *p = arr + 2;
```

### 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：

```
01. #include <stdio.h>
02.
03. int main() {
04.     int arr[] = { 99, 15, 100, 888, 252 };
05.     int i, *p = arr, len = sizeof(arr) / sizeof(int);
06.
07.     for(i=0; i<len; i++) {
08.         printf("%d ", *p++);
09.     }
10.     printf("\n");
11.     return 0;
12. }
```

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。

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。

## 数组与指针关系

C 语言标准规定，当数组名作为 **数组定义的标识符**（也就是定义或声明数组时）、**sizeof** 或 **&** 的操作数时，它才表示整个数组本身，在其他的表达式中，数组名会被转换为指向第 0 个元素的指针（地址）。

C 语言标准还规定，数组下标与指针的偏移量相同。通俗地理解，就是对数组下标的引用总是可以写成“一个指向数组的起始地址的指针加上偏移量”

读者可以通过以下任何一种方式来访问 a[i]：

p = a;	p = a; *(p + i);	p = a + i; *p;
--------	---------------------	-------------------

对数组的引用 a[i] 在编译时总是被编译器改写成 **\*(a+i)** 的形式，C 语言标准也要求编译器必须具备这种行为。

取下标操作符 [] 是建立在指针的基础上，它的作用是使一个指针和一个整数相加，产生出一个新的指针，然后从这个新指针（新地址）上取得数据；假设指针的类型为 T \*，所产生的结果的类型就是 T。

取下标操作符的两个操作数是可以交换的，它并不在意操作数的先后顺序，就像在加法中 3+5 和 5+3 并没有什么不一样。以上面的数组 a 为例，如果希望访问第 3 个元素，那么可以写作 **a[3]**，也可以写作 **3[a]**，这两种形式都是正确的，只不过后面的形式从不曾使用，它除了可以把初学者搞晕之外，实在没有什么实际的意义。

## 字符串指针

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 **使用一个指针直接指向字符串**，例如：

1. char \*str = "http://c.biancheng.net";

或者：

1. char \*str;
2. str = "http://c.biancheng.net";

字符串中的所有字符在内存中是连续排列的，（=的意思是使）str 指向字符串的第 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：

```
01. #include <stdio.h>
02. #include <string.h>
03.
04. int main() {
05.     char *str = "http://c.biancheng.net";
06.     int len = strlen(str), i;
07.
08.     //直接输出字符串
09.     printf("%s\n", str);
10.     //使用*(str+i)
11.     for(i=0; i<len; i++) {
12.         printf("%c", *(str+i));
13.     }
14.     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. }
```

纯文本 复制

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。

```
3     char str[20] = "c.biancheng.net";
4     char *s1 = str;
5     char *s2 = str+2;
6     char c1 = str[4];
7     char c2 = *str;
8     char c3 = *(str+4);
9     char c4 = *str+2;
.0    char c5 = (str+1)[5];
.1
.2    int num1 = *str+2;
.3    long num2 = (long)str;
.4    long num3 = (long)(str+2);
.
```

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：

```
01. #include <stdio.h>
02. int main() {
03.     char *str = "Hello World!";
04.     str = "I love C!"; //正确
05.     str[3] = 'P'; //错误
06.
07.     return 0;
08. }
```

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。

### 只能改变指向，不能改变字符串中的内容

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。

1) str 既是数组名称，也是一个指向字符串的指针；指针可以参加运算，加 1 相当于数组下标加 1。

printf() 输出字符串时，要求给出一个起始地址，并从这个地址开始输出，直到遇见字符串结束标志\0。s1 为字符串 str 第 0 个字符的地址，s2 为第 2 个字符的地址，所以 printf() 的结果分别为 c.biancheng.net 和 biancheng.net。

2) 指针可以参加运算，str+4 表示第 4 个字符的地址，c3 = \*(str+4) 表示第 4 个字符，即 'a'。

3) 其实，数组元素的访问形式可以看做 address[offset]，address 为起始地址，offset 为偏移量：c1 = str[4] 表示 str 的地址加 4 再取值，为 'a'；c5 = (str+1)[5] 表示以地址 str+1 为起点，向后偏移 5 个字符，等价于 str[6]，为 'c'。

+n=[n]

作为实参它传递的是 str 的副本，不是它本身，函数调用结束后副本销毁，所指的没有变化，没有初始化时会报错，

做实参时最好使用字符数组

```
char c[20];
```

```
char *p=c; //初始化，使指针指向数组c的首地址
```

```
gets(*p); //赋值
```

2、

```
char *p="ascd"; //初始化，这里会分配一个存储空间存储字符串"ascd",然后将该存储空间地址赋值给p. 这样初始化之后，访问*p得到的内容就是字符串"ascd"了，
```

3、

```
char *p = (char*)malloc(sizeof(char)*20); //初始化，分配了20个char类型的存储空间，p指向首地址
```

```
gets(*p); //赋值
```

如果想给指针所指向的地址赋值的话，那么，就必须先给指针初始化，使指针指向一个地址，然后利用指针修改该地址的内容。

# 指针变量和数组作为函数参数

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：

```
01. #include <stdio.h>
02.
03. void swap(int a, int b) {
04.     int temp; //临时变量
05.     temp = a;
06.     a = b;
07.     b = temp;
08. }
09.
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：

```
01. #include <stdio.h>
02.
03. void swap(int *p1, int *p2) {
04.     int temp; //临时变量
05.     temp = *p1;
06.     *p1 = *p2;
07.     *p2 = temp;
08. }
09.
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. }
```

纯文本 复制

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”。

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**

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，也可以省略数组长度**

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（当一维数组作为函数参数（形参，实参）的时候，编译器总是把它解析成一个指向其首元素首地址的指针）** int intArr[] 提醒读者指针 ar 指向的不仅仅

**一个 int 类型值，还是一个 int 类型数组的元素。**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。

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（编译可以通过，因为本质都是传递一个首地址，但结果可能会异常），所以并无实际意义。

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。

，数组名就是数组的首地址。因此在**数组名作函数参数时所进行的传送只是地址的传送，也就是说把实参数组的首地址赋予形参数组名(\*表示将实参的地址赋给形参)**。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上是**形参数组和实参数组为同一数组共同拥有一段内存空间，形参数组的值会影响实参数组**

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。

函数要处理数组必须知道何时开始、何时结束。sum()函数使用一个指针形参标识数组的一个整数形参表明待处理数组的元素个数（指针形参也表明了数组中的数据类型）。但是这并不是给函数传递必备信息的唯一方法。还有一种方法是传递两个指针，第1个指针指明数组的开始处（与前面用法相同），第2个指针指明数组的结束处。程序清单10.11演示了这种

方法，同时该程序也表明了指针形参是变量，这意味着可以用索引表明访问数组中的哪一个元素。

```
/* 使用指针算法 */
int sump(int * start, int * end)
{
    int total = 0;
    while (start < end)
    {
        total += *start; // 把数组元素的值加起来
        start++; // 让指针指向下一个元素
    }
    return total;
}
```

因为数组名是该数组首元素的地址，作为实际参数的数组名要求形式参数是一个与之匹配的指针。只有在这种情况下，C才会把int ar[]和int \* ar理解成一样。也就是说，ar是指向int的指针。由于函数原型可以省略参数名，所以下面4种原型都是等价的：

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

但是，在函数定义中不能省略参数名。

## 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。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。

```
01. #include <stdio.h>
02.
03. int *func() {
04.     int n = 100;
05.     return &n;
06. }
07.
08. int main() {
09.     int *p = func(), n;
10.     n = *p;
11.     printf("value = %d\n", n);
12.     return 0;
13. }
```

纯文本 复

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，这个方面的观点不是相悖吗？

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，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。

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）。

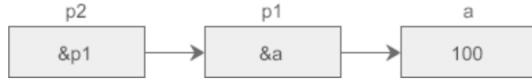
可以设为输入函数的返回指针

## 二级指针

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：

```
01. int a = 100;
02. int *p1 = &a;
03. 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：

```
01. int ***p3 = &p2;
```

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：

```
01. #include <stdio.h>
02.
03. int main() {
04.     int a = 100;
05.     int *p1 = &a;
06.     int **p2 = &p1;
07.     int ***p3 = &p2;
08.
09.     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. }
```

纯文本 复制

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：



对于数组指针，有 `char **p=char *p[]`

p 都是二级指针

## 空指针和 void 指针

在C语言中，如果一个指针不指向任何数据，我们就称之为 空指针，用 NULL 表示。例如：

```
int *p = NULL;
```

注意区分大小写，null 没有任何特殊含义，只是一个普通的标识符。

NULL 是一个宏定义，在 stdio.h 被定义为：

```
#define NULL ((void *)0)
```

(void \*)0 表示把数值 0 强制转换为 void \* 类型，最外层的 () 把宏定义的内容括起来，我们自己进行宏定义时也推荐这么做，防止发生歧义。

总之，p 的值为 0。你可以输出 p 的值：

它的值是随机的，是垃圾值，如果不小心使用了它，运行时一般会引起段错误，导致程序退出，甚至会不知不觉地修改数据。

p 经过定义，就一定在内存中分配了4个字节（32位环境）的空间，只是它的值是随机的，不像 int 会被初始化为 0，但是它确实指向了一段正常使用的内存。使用 p 时，操作的就是这段内存的数据，幸运的话能够正常运行，不过大部分情况下这段内存是无权操作的。

NULL 使 p 指向地址 0。大多数系统中都将 0 作为不被使用的地址，所以误用 p 也不会毁坏数据。

但并非总是如此，也有系统会使用地址 0，而将 NULL 定义为其他值，所以不要把 NULL 和 0 等同起来，下面的写法也是不专业的：

```
int *p = 0;
```

而应该坚持写为：

```
int *p = NULL;
```

注意 NULL 和 NUL 的区别：NULL表示空指针，是一个宏定义，可以在代码中直接使用。而 NUL 表示字符 '\0'，也就是字符串结束标志，它是ASCII码表中的第 0 个字符。NUL 没有在C语言中定义，仅仅是对 '\0' 的称呼 不能在代码中直接使用

### void 指针

C语言还有一种 void 指针类型，即可以定义一个指针变量，但不说明它指向哪一种类型数据。例如：

```
void *p = malloc(2);
```

在内存中分配2个字节的空间，但不确定它保存什么类型的数据。

注意，void 指针与空指针 NULL 不同：NULL 说明指针不指向任何数据，是“空的”；而 void 指针实实在在地指向一块内存，只是不知道这块内存中是什么类型的数据。

当把一个空指针常量转换为指针类型时，所得到的结果就是空指针（null pointer）。空指针常量（null pointer constant）是一个值为 0 的整数常量表达式，或者是一个 void\* 类型的表达式。在头文件 stdlib.h、stdio.h 以及其他头文件中，宏 NULL 被定义为空指针常量。

### 可以通过与 NULL 比较来检查指针是否为空指针

空指针有别于其他指向对象或函数的有效指针。因此，当返回值为指针的函数出现执行失败的情况时，它通常会使用空指针作为返回值。标准函数 `fopen()` 正是这样的一个例子，如果在指定的模式下打开某文件失败时，该函数会返回一个空指针。

`void` 指针是一种特殊的指针，表示为“无类型指针”，在 ANSI C 中使用它来代替“`char*`”作为通用指针的类型。由于 `void` 指针没有特定的类型，因此它可以指向任何类型的数据。也就是说，任何类型的指针都可以直接赋值给 `void` 指针，而无需进行其他相关的强制类型转换，如下面的示例代码所示：

```
1. void *p1;
2. int *p2;
3. ...
4. p1 = p2;
```

虽然如此，但这并不意味着可以无需任何强制类型转换就将 `void` 指针直接赋给其他类型的指针，因为“空类型”可以包容“有类型”，而“有类型”则不能包容“空类型”。正如我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”一样。因此下面的示例代码将编译出错，如果在 VC++2010 中，将提示“`a value of type "void*" cannot be assigned to an entity of type "int**"` 的错误信息。

```
1. void *p1;
2. int *p2;
3. ...
4. p2 = p1;
```

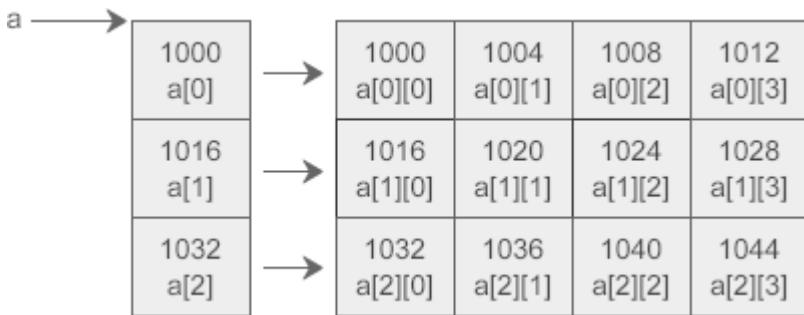
由此可见，要将 `void` 指针赋值给其他类型的指针，必须进行强制类型转换。前面提到，`void` 指针可以指向任意类型的数据，同时任何类型的指针都可以直接赋值给 `void` 指针，而无需进行其他相关的强制类型转换。因此，在编程中，如果函数的参数可以是任意类型指针，那么应该使用 `void` 指针作为函数的形参，这样函数就可以接受任意数据类型的指针作为参数。

## 使用指针访问二维数组

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  $4 \times (3 \times 4) = 48$  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：



`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  $4 \times 4 = 16$  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

`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)`

## 将二维数组作为函数参数

二维数组作为函数的参数，实参可以直接使用二维数组名，在被调用函数中可以定义形参所有维数的大小，也可以省略以为大小的说明。例如：

```
1 void find (char a[3][10]);
2 void find (char a[ ][10]);
```

也可以使用数组指针来作为函数参数，例如：

```
1 void find (char (*p)[10]);
```

**空的方括号表示 p 是指针；最终都会转化为指针数组（也是实参（数组名）的数据类型 `(int (*)[(sizetype)(n)])`），维数必须另外传参进去**

**二维数组名是一个指向第一行的指针**

指针数组是一个数组，这个数组的每一个元素都是一个指针。

**数组指针**是一个指针，这个指针指向的是一个数组，这里要注意的是，是指向整个数组，而不是指向数组的第一个元素

```
void Func(int array[3][10]);
void Func(int array[][10]);
```

变为：

```
void Func(int **array, int m, int n);
```

如果不确定二维数组的维数的话，可以用下面的方法：

手工转变寻址方式

对于数组 int p[m][n];

如果要取  $p[i][j]$  的值 ( $i \geq 0 \&& m \leq 0 \&& j \geq 0 \&& n \leq 0$ )，编译器是这样寻址的，它的地址为：

$p + i * n + j;$

从以上可以看出，如果我们省略了第二维或者更高维的大小，编译器将不知道如何正确的寻址。但是我们在编写程序的时候却需要用到各个维数都不固定的二维数组作为参数，这就难办了，编译器不能识别阿，怎么办呢？不要着急，编译器虽然不能识别，但是我们完全可以不把它当作一个二维数组，而是把它当作一个普通的指针，再另外加上两个参数指明各个维数，然后我们为二维数组手工寻址，这样就达到了将二维数组作为函数的参数传递的目的，根据这个思想，我们可以把维数固定的参数变为维数随即的参数

在转变后的函数中， $array[i][j]$  这样的式子是不对的，因为编译器不能正确的为它寻址，所以我们需要模仿编译器的行为把  $array[i][j]$  这样的式子手工转变为： $* ( (int*)a + n*i + j);$   
 $Func((int**)a, 3, 3);$

假如，有如下代码：

```
int a[2][3]; // 一个二维数组
function(a); // 将二维数组传参给一个函数
```

上述代码中，就是将一个二维数组传递给一个函数，那么这个函数怎么接受这个参数呢，答案是这样的：

```
void function(int a[2][3]) // 这是对的
{
void function(int (*a)[3]) // 这也是对的
{}
```

千万别写成这样：

```
void function(int **a) // 这是错的
{}
```

究其原因，还是回到以前提过多次的数组与指针的结论：任何数组，都将被一律视为一个指向其首元素的指针。因此以下两行代码是等价的：

```
function(a);
function(&a[0]);
```

对于二维数组  $int a[2][3]$  而言，其首元素就是一个  $int [3]$ ，因此  $\&a[0]$  就是一个指向  $int [3]$  的指针，其类型当然是  $int (*)[3]$  了。这样再来看上述的  $function$  两种正确定义，就不难理解了。

最后一种只是无法通过  $a[]$  寻址，需要用  $* ( (int*)a + n*i + j);$ （类似用一级指针模拟二级指针的取值方式）切记实参和寻址表达式需要强制转换（分别转换为  $**$  和  $*$ ），传时当成二级指针，找时当成一级指针，不能用  $*(*(p+i)+j)$ ，这里的  $p$  是一个指针名，指向  $int[j]$ ，而  $a$  是指向  $int *[i]$ ；

## 动态数组

二级指针,其实是指向指针的指针,在上面我用二级指针指向一个指针数组是合法的,其值就是指针数组第一个元素的地址,而第一个元素是一个指针,那么就是指向指针的地址,在通过下标找寻元素的时候,其实质是先找到所谓"行"(找到一个指针),然后通过第二个下标找到该指针指向的第几号元素完成索引。

```
/*int **matrix;
matrix=(int **)malloc(n*sizeof(int *));
for (int i = 0; i < n; ++i)
{
    matrix[i] = (int *)malloc(sizeof(int) * n);
}*/
```

等价于 int \*matrix[n];然后循环分配

```
..... }
```

```
..... free(matrix[i]);
```

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

```
..... }
```

```
..... free(matrix);
```

: 先申请空间——>判断空间是否申请成功——>使用申请成功的空间——>free 掉申请的空间。不过需要注意的是,二维数组的逐层申请和释放空间的顺序。

在申请空间时,遵循由外向里的逐层申请次序;

在释放空间时,遵循由里向外的逐层释放次序。

只有这样可以用二级指针传参并在函数内用 a[][] 取值

定义数组用二级指针传参的方法见最上面

用一级指针模拟二级指针

```
arr=(int*)malloc(n*n*sizeof(int))
```

```
取值用* (arr+n*i+j)
```

```
struct film * movies; /* 指向结构的指针 */  
...  
printf("Enter the maximum number of movies you'll enter:\n");  
scanf("%d", &n);  
movies = (struct film *) malloc(n * sizeof(struct film));
```

第12章介绍过，可以像使用数组名那样使用指针movies。

```
struct film * movies[FMAX]; /* 结构指针数组 */
```

```
int i;
```

```
...
```

```
movies[i] = (struct film *) malloc (sizeof (struct film));
```

## 函数指针

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，pointerName 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。

```
01. #include <stdio.h>
02.
03. //返回两个数中较大的一个
04. int max(int a, int b) {
05.     return a>b ? a : b;
06. }
07.
08. int main() {
09.     int x, y, maxval;
10.     //定义函数指针
11.     int (*pmax)(int, int) = max; //也可以写作int (*pmax)(int a, int b)
12.     printf("Input two numbers:");
13.     scanf("%d %d", &x, &y);
14.     maxval = (*pmax)(x, y);
```

由于函数名是指针，那么指针和函数名可以互换使用

```
double (*pdf)(double);
```

1121

```
double x;

pdf = sin;

x = (*pdf)(1.2); // 调用sin(1.2)

x = pdf(1.2); // 同样调用 sin(1.2)
```

## 指针数组

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：

```
01. #include <stdio.h>
02. int main(){
03.     int a = 16, b = 932, c = 100;
04.     //定义一个指针数组
05.     int *arr[3] = {&a, &b, &c}; //也可以不指定长度，直接写作 int *arr[]
06.     //定义一个指向指针数组的指针
07.     int **parr = arr;
08.     printf("%d, %d, %d\n", *arr[0], *arr[1], *arr[2]);
09.     printf("%d, %d, %d\n", *(parr+0), *(parr+1), *(parr+2));
10.
11.     return 0;
12. }
```

指针数组还可以和字符串数组结合使用，请看下面的例子：

```
01. #include <stdio.h>
02. int main(){
03.     char *str[3] = {
04.         "c.biancheng.net",
05.         "C语言中文网",
06.         "C Language"
07.     };
08.     printf("%s\n%s\n%s\n", str[0], str[1], str[2]);
09.     return 0;
10. }
```

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，不能修改字符串中的元素，例如给`*str[0]`赋值

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。

```
01. #include <stdio.h>
02. int main(){
03.     char *str0 = "c.biancheng.net";
04.     char *str1 = "C语言中文网";
05.     char *str2 = "C Language";
06.     char *str[3] = {str0, str1, str2};
07.     printf("%s\n%s\n%s\n", str[0], str[1], str[2]);
08.     return 0;
09. }
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     char *lines[5] = {
6         "COSC1283/1284",
7         "Programming",
8         "Techniques",
9         "is",
10        "great fun"
11    };
12
13    char *str1 = lines[1];
14    char *str2 = *(lines + 3);
15    char c1 = *(*(lines + 4) + 6);
16    char c2 = (*lines + 5)[5];
17    char c3 = *lines[0] + 2;
18
19    printf("str1 = %s\n", str1);
20    printf("str2 = %s\n", str2);
21    printf("c1 = %c\n", c1);
22    printf("c2 = %c\n", c2);
23    printf("c3 = %c\n", c3);
24    return EXIT_SUCCESS;
25 }

```



这样，或许会清晰很多，`char *lines[5]`; 定义了一个指针数组，数组的每一个元素都是指向 `char` 类型的指针。最后 5 行，为数组的每一个元素赋值，都是直接赋给指针。

而 `lines`，是一个指向第 0 个指针元素（值为该指针的地址（指针定义））的指针，它的类型为 `char **`，所以 `*lines` 是一个指向字符串的指针，`**lines` 是一个具体的字符（相当于 `(*lines+0)` 指向具体字符串后才有意义）。这一点很重要，一定要明白。

指针是可以进行运算的，`lines` 为 `lines[5]` 数组的首地址，即 第 0 个元素的地址；`lines+0`, `lines+1`, `lines+2` ... 分别是第 0, 1, 2 ... 个元素的 首地址，`*(lines+0)` 或 `lines[0]`, `*(lines+1)` 或 `lines[1]`, `*(lines+2)` 或 `lines[2]` ... 分别是字符串 `str0`, `str1`, `str2` ... 的首地址，所以：

`*lines == *(lines+0) == lines[0] == str0`

`*(lines+1) == lines[1] == str1`

`*(lines+n) = (*lines)[n] = lines[0][n]` (第 0 个字符串中的第 n 个字符)

`p[n] = *(p+n)`

`p+n = &p[n]`

- `lines[1]`: 它是一个指针，指向字符串 `string1`，即 `string1` 的首地址。
- `*(lines + 3)`: `lines + 3` 为 `lines[5]` 数组第 3 个元素的地址，`*(lines + 3)` 为第 3 个元素，它是一个指针，指向字符串 `string3`。
- `*(*(lines + 4) + 6)`: `*(*(lines + 4) + 6) == lines[4] + 6 == string4 + 6`，为字符串 `string4` 第 6 个字符的地址，即 `f` 的地址，`*(*(lines + 4) + 6)` 就表示字符 `f`。
- `(*lines + 5)[5]`: `*lines + 5` 为字符串 `string0` 第 5 个字符的地址，即 `2` 的地址，`(*lines + 5)[5]` 等价于 `*(*lines + 5 + 5)`，表示第 10 个字符，即 `2`。
- `*lines[0] + 2`: `*lines[0]` 为字符串 `string0` 第 0 个字符的地址，即 `C` 的地址。字符与整数运算，首先转换为该字符对应的 ASCII 码值，然后再运算，所以 `*lines[0] + 2 = 67 + 2 = 69`。不过要求输出字符，所以还要转换成 69 所对应的字符，即 `E`。

字符数组与字符指针处理字符串有何不同

占用空间不同。数组所占空间取决于数组的长度，而指针只占用 4 字节，用以存放字符串

的首地址。

赋值方式不同指针本身是变量，所以可以这样 `char *ps; ps = "C language!"`; 赋值，而数组不能这样 `char A[20];A = "C language!"`; 赋值，而要逐个赋值（只有定义的时候可以整体赋值）。

p 肯定是指针变量，它也一定是地址，但是别忘了在 C 语言中，系统都会默认在字符串末给我们加上\0 用于结束字符串，所以用 `printf()` 输出时候，只要你打印格式是%`s`，系统知道我们要输出字符串，而且 **系统也可以确定字符串的长度**，所以当你用%`s` 打印字符指针 p 的时候就会输出字符串（对地址进行解析），如果你是以地址打印格式输出的话，系统就会给我们打印出地址（即输出 p 的值（字符串的地址）），**本质都是对一个地址进行操作**，如果你是解指针（`*p`）那么系统就取 p 指针指向的内容，此时看你自己以什么格式输出，就用什么格式打印。即**输出 p 的值（字符串的地址）或地址**

函数使用指向字符串首字符的指针来表示待处理的字符串。通常，对应的实际参数是数组名、指针变量或用双引号括起来的字符串。无论是哪种情况，传递的都是首字符的地址。一般而言，没必要传递字符串的长度，因为函数可以通过末尾的空字符确定字符串的结束

`&a[0]` 和 `&a` 的值是相同的。但是要注意，尽管它们的结果相同，但其所表达的意义却完全不相同，这一点一定要注意。

因为**数组名包含数组的首地址（即数组第一个元素的地址）**，或者说**数组名指向数组的首地址（或第一个元素）**，所以，对于 `&a`，表示取数组 a 的首地址；而对于 `&a[0]`，它表示取数组**首元素 a[0] 的首地址**（两者一样）。这就好像陕西的省政府在西安，而西安的市政府同样也在西安。虽然两个政府机构都在西安，但其代表的意义完全不同。

## const 函数详解

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：

```
01. 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**

`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 本身的数据不能被修改；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 本身的数据可以修改（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。

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：

```
01. size_t strlen ( const char * str );
02. int strcmp ( const char * str1, const char * str2 );
03. char * strcat ( char * destination, const char * source );
04. char * strcpy ( char * destination, const char * source );
05. int system (const char* command);
06. int puts ( const char * str );
07. 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：

```
01. #include <stdio.h>
02.
03. size_t strnchr(const char *str, char ch) {
04.     int i, n = 0, len = strlen(str);
05.
06.     for(i=0; i<len; i++) {
07.         if(str[i] == ch) {
08.             n++;
09.         }
10.    }
```

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。

## define 和 const 使用时的注意点

在 C 语言中，`const` 不是一个真真正正的常量，其代表的含义仅仅是只读。使用 `const` 声明的对象是一个[运行时对象](#)，无法使用其作为某个量的初值、数组的长度  
由 `const` 修饰的类型在内存中占空间而 `#define` 不占空间，`#define` 只是在编译前将要编译的源文件中相应的部分使用字符串替换例如前面的代码在编译前会被预处理  
`const` 定义的常量[不能作为函数形参和变成数组长度](#)

## 复合字面量

假设给带 `int` 类型形参的函数传递一个值，要传递 `int` 类型的变量，但是也可以传递 `int` 类型常量，如 5。在 C99 标准以前，对于带数组形参的函数，情况不同，可以传递数组，但是没有等价的数组常量。C99 新增了复合字面量（compound literal）。字面量是除符号常量外的常量。例如，5 是 `int` 类型字面量，81.3 是 `double` 类型的字面量，'Y' 是 `char` 类型的字面量，"elephant" 是字符串字面量。发布 C99 标准的委员会认为，如果有代表数组和结构内容的复合字面量，在编程时会更方便。

对于数组，复合字面量类似数组初始化列表，前面是用括号括起来的类型名。例如，下面是一个普通的数组声明：

```
int diva[2] = {10, 20};
```

下面的复合字面量创建了一个和diva数组相同的匿名数组，也有两个int类型的值：

```
(int [2]) {10, 20} // 复合字面量
```

注意，去掉声明中的数组名，留下的int [2]即是复合字面量的类型名。

初始化有数组名的数组时可以省略数组大小，复合字面量也可以省略大小，编译器会自动计算数组当前的元素个数：

```
(int []) {50, 20, 90} // 内含3个元素的复合字面量
```

因为复合字面量是匿名的，所以不能先创建然后再使用它，必须在创建的同时使用它。使用指针记录地址就是一种用法。也就是说，可以这样用：

```
int * pt1;
```

```
pt1 = (int [2]) {10, 20};
```

注意，该复合字面量的字面常量与上面创建的diva数组的字面常量完全相同。与有数组名的数组类似，复合字面量的类型名也代表首元素的地址，所以可以把它赋给指向int的指针。然后便可使用这个指针。例如，本例

中\*pt1是10，pt1[1]是20

还可以把复合字面量作为实际参数传递给带有匹配形式参数的函数：

```
int sum(const int ar[], int n);
```

```
...
```

```
int total3;
```

```
total3 = sum((int []) {4, 4, 4, 5, 5, 5}, 6);
```

这里，第1个实参是内含6个int类型值的数组，和数组名类似，这同时也是该数组首元素的地址。这种用法的好处是，把信息传入函数前不必先创建数组，这是复合字面量的典型用法。

可以把这种用法应用于二维数组或多维数组。例如，下面的代码演示了如何创建二维int数组并储存其地址：

```
int (*pt2)[4]; // 声明一个指向二维数组的指针，
```

```
// 每个元素是内含4个int类型值的数组
```

```
pt2 = (int [2][4]) { {1, 2, 3, -9}, {4, 5, 6, -8} };
```

如上所示，该复合字面量的类型是int [2][4]，即一个 $2 \times 4$ 的int数组。

复合字面量是提供只临时需要的值的一种手段。复合字面量具有块作用域，这意味着一旦离开定义复合字面量的块，程序将无法保证该字面量是否存在。也就是说，复合字面量的定义在最内层的花括号中

# 指针初始化

## 1. char类型的指针

```
char *names[]={"zhangsan","lisi","wangwu"};//可以这样初始化
```

这样，每个指针指向对应的字符串

## 2. 其他类型的指针

```
int *nums[]={1, 2, 3, 4, 5};//不可以这样初始化
```

原因：指针的初始化或赋值可以使用0值、常量表达式、和类型匹配的对象的地址。//指针需要用地址初始化

如上，字符串字面值常量类型为`const char *`，与指针类型匹配，可赋值。

整形字面值常量可以为int, long等类型，详见《C++ Primer》第二章2.2节。但给指针赋值应该是地址，即&i，VS2010下编译错误也会提示：不可将int类型赋给int \*类型。

在C语言中，使用赋值操作符时，赋值操作符左边和右边的表达式类型应该是同样的，假设不是，赋值操作符将试图把右边表达式的值转换为左边的类型。所以假设写出`int *p = 0x12345678;`；这条语句编译器会报错：`'=' : cannot convert from ' const int ' to ' int * '`，由于赋值操作符左边和右边的表达式的类型应该同样，而`0x12345678`是int型常量，p是一个指向int型的指针，两者类型不同，**所以正确的方式是：**`int *p = (int *) 0x12345678;`；对指针进行初始化时经常使用的有下面几种方式：

1. 采用NULL或空指针常量，如：`int *p = NULL;`或`char *p = 0;`或`float *p = 0;`或`int *a[n]={0}`
2. 取一个对象的地址然后赋给一个指针，如：`int i = 3; int *ip = &i;`
3. 将一个指针常量赋给一个指针，如：`long *p = (long *)0xffffffff0;`
4. 将一个T类型数组的名字赋给一个同样类型的指针，如：`char ary[100]; char *cp = ary;`
5. 将一个指针的地址赋给一个指针，如：`int i = 3; int *ip = &i; int **pp = &ip;`
6. 将一个字符串常量赋给一个字符指针，如：`char *cp = "abcdefg";`

对指针进行初始化或赋值的实质是将一个地址或同类型(或相兼容的类型)的指针赋给它，而无论这个地址是怎么取得的。要注意的是：对于一个不确定要指向何种类型的指针，在定义它之后最好把它初始化为NULL，并在解引用这个指针时对它进行检验，防止解引用空指针。另外，为程序中不论什么新创建的变量提供一个合法的初始值是一个好习惯，它能够帮你避免一些不必要的麻烦。

其实变量和指针，编译器都是平等对待的。VC中，全局变量初始化为0，全局指针初始化为NULL；局部变量（包括指针变量）如果未显式初始化，其值都是未定的。

```
13 void PRINT(int *,int);
14 //char *a;
15 int *a[L];
16 int main()
17 {
18     int *p[L];
19     PRINT(a[0],L);
20     // PRINT(a,L);
21     PRINT(p[0],L);
22     // PRINT(p,L);
23     return 0;
24 }
25 void PRINT(int *s,int n)
26 {
27     printf("%p",&s);
28     for(int i=0;i<n;i++)
29         printf("%p",s+4*i);
30     printf("\n");
31 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c"
1
0x7fff558aad68(nil)0x100x200x300x400x500x600x700
0x7fff558aad680x55e0b38590400x55e0b38590500x55e0
0b38590b00x55e0b38590c00x55e0b38590d0
fengsc@ubuntu:~/Desktop/c $
```

这个错误，因为传入的只是 p[0]的值：0x0 (nil) ， 结果相当于 16 进制加法， +i=+4i

```
15 int *a[L];
16 int main()
17 {
18     int *p[L];
19     //PRINT(a[0],L);
20     PRINT(a,L);
21     // PRINT(p[0],L);
22     PRINT(p,L);
23     return 0;
24 }
25 void PRINT(int **s,int n)
26 {
27     printf("%p",&s);
28     for(int i=0;i<n;i++)
29         printf("%p",s[i]);
30     printf("\n");
31 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fengsc@ubuntu:~/Desktop/c $ cd "/home/fengsc/Desktop/c/" && gcc test1.c -o
1
0xffffca361268(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil)
0xffffca3612680x55f1741bb0400xf0b5ff0xc20x7ffffca3612d70x7ffffca3612d60x55f17
55f1741bc0a0
fengsc@ubuntu:~/Desktop/c $
```

用一级指针作参数会警告，因为%p 是用来输出指针的值的 (void \*)

## Segmentation Fault 错误原因总结

# 值传递和地址传递

值传递使用变量、常量、数组元素作为函数参数，实际是将实参的值复制到形参相应的存储单元中，即形参和实参分别占用不同的存储单元，这种传递方式称为“参数的值传递”或者“函数的传值调用”。

值传递的特点是单向传递，即主调函数调用时给形参分配存储单元，把实参的值传递给形参，在调用结束后，形参的存储单元被释放，而形参值的任何变化都不会影响到实参的值，实参的存储单元仍保留并维持数值不变。

地址传递使用数组名或者指针作为函数参数，传递的是该数组的首地址或指针的值，而形参接收到的是地址，即指向实参的存储单元，形参和实参占用相同的存储单元，这种传递方式称为“参数的地址传递”。

地址传递的特点是形参并不存在存储空间，编译系统不为形参数组分配内存。数组名或指针就是一组连续空间的首地址。因此在数组名或指针作函数参数时所进行的传送只是地址传送，形参在取得该首地址之后，与实参共同拥有一段内存空间，**形参的变化也就是实参的变化。形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作**

指针的传递也遵守上述规则，对自身值的操作就是值操作，对指向的地址的操作为地址操作

要改变\*p 的值，则需要把&p (\*\*型) 传进去，再取值操作

## 指针常见错误

### 1) 引用未初始化的指针变量

试图引用未初始化的指针变量是初学者最容易犯的错误。未初始化的指针变量就是“野”指针，它指向的是无效的地址。

有些书上说：“如果指针变量不初始化，那么它可能指向内存中的任何一个存储单元，这样就会很危险。如果正好指向存储着重要数据的内存单元，而且又不小心向这个内存单元中写入了数据，把原来的重要数据给覆盖了，这样就会导致系统崩溃。”这种说法是不正确的！如果真是这样的话就是编译器的一个严重的 BUG！

编译器的设计人员是不会允许这么大的 BUG 存在的。那么如果指针变量未初始化，编译器的设计人员是如何处理这个问题的呢？肯定不可能让它乱指。以 VC++6.0 这个编译器为例，如果指针变量未初始化，那么编译器会让它指向一个固定的、不用的地址。

程序中，j 是 int\* 型的指针变量。j 中存放的应该是内存空间的地址，然后“变量 i 赋给 \*j”表示将变量 i 中的值放到该地址所指向的内存空间中。但是现在 j 中并没有存放一个地址，程序中并没有给它初始化，那么它指向的就是 0XCCCCCCCC 这个内存单元。这个内存单元是不允许访问的，即不允许往里面写数据。而把 i 赋给 \*j 就是试图往这个内存空间中写数据，程序执行时就会出错。但这种错误在编译的时候并不会报错，只有在执行的时候才会出错，即传说中的“段错误”。所以，**一定要确保指针变量在引用之前已经被初始化为指向有效的地址。**

能不能使用 `scanf` (或其他输入函数) 给指针变量初始化? 指针变量里面存放的是地址, 而内存中有数不清的单元, 每个单元都有一个地址, 你知道每个单元的地址吗? 你知道哪些地址是空闲可用的, 而哪些地址正存储着重要数据不能用吗? 不知道的话怎么用 `scanf` 给它初始化呢? 万一随便写一个地址正好是存储着非常重要的数据的内存单元地址, 那系统就真的崩溃了!

## 2) 往一个存放 NULL 地址的指针变量里面写入数据

之前是没有给指针变量 `j` 初始化, 现在初始化了, 但是将它初始化为指向 `NULL`。`NULL` 也是一个指针变量。`NULL` 指向的是内存中地址为 0 的内存空间。以 32 位操作系统为例, 内存单元地址的范围为 `0x00000000~0xffff ffff`。其中 `0x00000000` 就是 `NULL` 所指向的内存单元的地址。但是在操作系统中, 该内存单元是不可用的。凡是试图往该内存单元中写入数据的操作都会被视为非法操作, 从而导致程序错误。同样, 这种错误在编译的时候也不会报错, 只有在执行的时候才会出错。这种错误也属于“段错误”。

然而虽然这么写是错误的, 但是将一个指针变量初始化为指向 `NULL`, 这在实际编程中是经常使用的。就跟前面讲普通变量在定义时给它初始化为 0 一样, 指针变量如果在定义时不知道指向哪里就将其初始化为指向 `NULL`。只是此时要注意的是, 在该指针变量指向有效地址之前不要往该地址中写入数据。也就是说, 该指针变量还要二次赋值。

既然不能往里面写数据, 而且还容易犯错, 为什么还要这样给它初始化呢? 直接同前面定义普通变量时一样, 在定义时也不初始化, 等到后面知道该给它赋什么值时再给它赋值不行吗? 可以! 但还是建议大家将它初始化为 `NULL`, 就同前面将普通变量在定义时初始化为 0 一样。这是很好的一种编程习惯。

## 小结

C 语言标准规定, 对于一个符号的定义, 编译器总是从它的名字开始读取, 然后按照优先级顺序依次解析。从名字开始, 不是从开头也不是从末尾, 这是理解复杂指针的关键!

它们的优先级从高到低依次是: () [] \*(右结合性)

`char *(* c[10])(int **p);`

这个定义有两个名字, 分别是 `c` 和 `p`, 乍看起来 `p` 是指针变量的名字, 不过很遗憾这是错误的。如果 `p` 是指针变量名, `c[10]` 这种写法就又定义了一个新的名字, 这让人匪夷所思。

以 `c` 作为变量的名字, 先来看括号内部部分(`* c[10]`):

`char *(*c[10])(int **p);`

[`] 的优先级高于 *`, 编译器先解析 `c[10]`, `c` 首先是一个数组, 它前面的`*`表明每个数组元素都是一个指针, 只是还不知道指向什么类型的数据。整体上来看, (`* c[10]`) 说明 `c` 是一个指针数组, 只是指针指向的数据类型尚未确定。

跳出括号, 根据优先级规则 ((`)` 的优先级高于 `*`) 应该先看右边(`int **p`):

`char *(*c[10])(int **p);`

(`)` 说明是一个函数, `int **p` 是函数参数。

再看左边 `char *:`

`char *(*c[10])(int **p);`

(`*c[10]`) 表明 `c` 是一个指针数组, 其他部分表明指针指向的数据类型, 合起来就是: `c` 是一个拥有 10 个元素的指针数组, 每个指针指向一个原型为 `char *func(int **p);` 的函数。

```
int (*(*(pfunc)(int *))[5])(int *)
```

1. `(*pfunc)` 一个指针
2. `(*(*pfunc)(int *))`: 根据优先级规则应该先看右边的 `(int *)`, 它表明这是一个函数, `int *` 是参数列表。再看左边的\*, **它表明函数的返回值是一个指针, 只是指针指向的数据类型尚未确定** (下一步就是定义函数的返回值)。
3. 由第二步可知, 它表明 `pfunc` 是一个指向函数的指针, 现在函数的参数列表确定了, 也知道返回值是一个指针了 (只是不知道它指向什么类型的数据)
4. `(*(*(*pfunc)(int *)))[5]):[ ]` 的优先级高于 \*, 先看右边, `[5]` 表示这是一个数组, 再看左边, \* 表示数组的每个元素都是指针。也就是说, `* [5]` 是一个指针数组, 函数返回的指针就指向这样一个数组。
5. 指针数组中的指针指向原型为 `int func(int *)`; 的函数。
6. 将上面的三部分合起来就是: `pfunc` 是一个函数指针, 该函数的返回值是一个指针, 它指向一个指针数组 (, 指针数组中的指针指向原型为 `int func(int *)`; 的函数。

#### 常见指针变量的定义

定 义	含 义
<code>int *p;</code>	<code>p</code> 可以指向 <code>int</code> 类型的数据, 也可以指向类似 <code>int arr[n]</code> 的数组。
<code>int **p;</code>	<code>p</code> 为二级指针, 指向 <code>int *</code> 类型的数据。
<code>int *p[n];</code>	<code>p</code> 为指针数组。 <code>[ ]</code> 的优先级高于 *, 所以应该理解为 <code>int *(p[n]);</code>
<code>int (*p)[n];</code>	<code>p</code> 为 <b>二维数组</b> 指针。
<code>int *p();</code>	<code>p</code> 是一个函数, 它的返回值类型为 <code>int *</code> 。
<code>int (*p)();</code>	<code>p</code> 是一个 <b>函数指针</b> , 指向原型为 <code>int func()</code> 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。

## 结构体和其它数据类型

### struct 用法详解

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）**

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。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**。

。如果把结构声明置于一个函数的内部，它的标记就只限于该函数内部使用。如果把结构声明置于函数的外部，那么该声明之后的所有函数都能使用它的标记。

```
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。**

**可以在定义结构体的同时定义结构体变量，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，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。

name	num	age	group	score
------	-----	-----	-------	-------

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。

name	num	age	group	score
------	-----	-----	-------	-------

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；**

```
//给结构体成员赋值  
stu1.name = "Tom";  
stu1.num = 12;  
stu1.age = 18;  
stu1.group = 'A';  
stu1.score = 136.5;
```

name 是字符数组则不能直接赋值，需要 strcpy 函数

```
struct {  
    char *name; //姓名  
    int num; //学号  
    int age; //年龄  
    char group; //所在小组  
    float score; //成绩  
} stu1, stu2 = { "Tom", 12, 18, 'A', 136.5 };
```

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。

大括号内定义时用分号，初始化用逗号

{ } (整体赋值)这种语法不能用于结构体的赋值，只能用于 **初始化**  
**指定初始化**

只初始化 book 结构的 value 结构，可以这样做

```
struct book surprise = { .value = 10.99};
```

可以按照任意顺序使用指定初始化器：

```
struct book gift = { .value = 25.99,  
.author = "James Broadfool",  
.title = "Rue for the Toad"};
```

与数组类似，在指定初始化器后面的普通初始化器，为指定成员后面的成员提供初始值。另外，对特定成员的最后一次赋值才是它实际获得的值。

例如，考虑下面的代码：

```
struct book gift = { .value = 18.90,  
.author = "Philionna Pestle",  
0.25};
```

赋给 value 的值是 0.25，因为它在结构声明中紧跟在 author 成员之后。新值 0.25 取代了之前的 18.9。在学习了结构的基本知识后，可以进一步了解结构的一些相关类型。

**在结构体变量内部，在成员运算符前面加或者不加结构体变量名都是可以的。**

## 结构体数组

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：

```
01. struct stu{  
02.     char *name; //姓名  
03.     int num; //学号  
04.     int age; //年龄  
05.     char group; //所在小组  
06.     float score; //成绩  
07. }class[5];
```

表示一个班级有5个学生。

定义后里面元素不能单独赋值，因为它是数组，即不能用 class[5].xxx=xxx,形式  
结构体数组在定义的同时也可以初始化，例如：

```
01. struct stu{  
02.     char *name; //姓名  
03.     int num; //学号  
04.     int age; //年龄  
05.     char group; //所在小组  
06.     float score; //成绩  
07. }class[5] = {  
08.     {"Li ping", 5, 18, 'C', 145.0},  
09.     {"Zhang ping", 4, 19, 'A', 130.5},  
10.     {"He fang", 1, 18, 'A', 148.5},  
11.     {"Cheng ling", 2, 17, 'F', 139.0},  
12.     {"Wang ming", 3, 17, 'B', 144.5}  
13. };
```

当对数组中全部元素赋值时，也可不给出数组长度，例如：

```
01. struct stu{  
02.     char *name; //姓名  
03.     int num; //学号  
04.     int age; //年龄  
05.     char group; //所在小组  
06.     float score; //成绩  
07. }class[] = {  
08.     {"Li ping", 5, 18, 'C', 145.0},  
09.     {"Zhang ping", 4, 19, 'A', 130.5},  
10.     {"He fang", 1, 18, 'A', 148.5},  
11.     {"Cheng ling", 2, 17, 'F', 139.0},  
12.     {"Wang ming", 3, 17, 'B', 144.5}  
13. };
```

```
3 struct student array1[1000] ;  
4 int main(void)  
5 {  
6     int i ;  
7     for(i = 0 ; i < 1000 ; i++)  
8     {  
9         array[i].a = 1 ;  
10        array[i].b = 2 ;  
11        array[i].c = 3 ;  
12    }
```

```
struct student array[1000] = {  
    [0 ... 999] = {  
        .a = 1 ,  
        .b = 2 ,  
        .c = 3 ,  
    }  
};
```

```
struct Structure ptStct[10]={  
    [2].b = 0x2B, [2].a = 0x2A,  
    [0].a = 0x0A };
```

#### 嵌套结构

首先，注意如何在结构声明中创建嵌套结构。和声明 int 类型变量一样，进行简单的声明：  
struct names handle;

该声明表明 handle 是一个 struct names 类型的变量。当然，文件中也应包含结构 names 声明。其次，注意如何访问嵌套结构的成员，这需要使用两次点运算符：

```
printf("Hello, %s!\n", fellow.handle.first);
```

从左往右解释 fellow.handle.first:

(fellow.handle).first

也就是说，找到 fellow，然后找到 fellow 的 handle 的成员，再找到 handle 的 first 成员

## 结构体指针

使用结构体指针的原因：

第一，就像指向数组的指针比数组本身更容易操控（如，排序问题）一样，指向结构的指针通常比结构本身更容易操控。

第二，在一些早期的 C 实现中，结构不能作为参数传递给函数，但是可以传递指向结构的指针。第三，即使能传递一个结构，传递指针通常更有效率。第四，一些用于表示数据的结构中包含指向其他结构的指针

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 结构体变量名 \*变量名；

也可以在定义结构体的同时定义结构体指针 struct 结构体名 {xxx} 结构体变量名={xxx}, \*变量名=&xxx；

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 stu 赋值只能写作：

struct stu \*pststu = &stu1;

而不能写作：

struct stu \*pststu = 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。

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。

## 结构体数组指针及函数

设 ps 为指向结构体数组的指针变量，则 ps 也指向该结构体数组的第 0 个元素，ps+1 指向第一个元素，ps+i 则指向第 i 元素，这与普通数组的情况是一样的。

应该注意的是，一个结构体指针变量虽然可以用来访问**结构体变量或结构体数组元素的成员，但是，不能使它指向一个成员**。也就是说不允许取一个成员的地址来赋予它。所以，下面的赋值是错误的：

```
ps=&boy[1].sex;
```

而只能是：

```
ps=boy; // 赋予数组首地址
```

或者是：

```
ps=&boy[0]; // 赋予 0 号元素首地址
```

```
(stus+i)->name=stus[i].name
```

结构体变量代表的是结构体本身这个整体，而不是首地址，作为函数参数时传递的整个结构体，也就是所有成员。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，严重降低程序的效率。所以最好的办法就是使用指针，也就是用指针变量作为函数参数。这时由实参传向形参的只是地址，非常快速

例如 void average(**struct stu** \*ps)

## 枚举类型

枚举类型的定义形式为：

```
enum typeName{ valueName1, valueName2, valueName3, ..... };
```

枚举常量只能以标识符形式表示，而不能是整型、字符型等文字常量

各枚举常量的值可以重复。

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：**

枚举变量的值只能取枚举常量表中所列的值（标识符），就是整型数的一个子集（普通变

量需要强制类型转换后赋值)

枚举变量占用内存的大小与整型数相同。

枚举变量只能参与赋值和关系运算以及输出操作，参与运算时用其本身的整数值。

```
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;
```

enum feline {cat, lynx = 10, puma, tiger}; 那么，**cat** 的值是 0（默认），lynx、puma 和 tiger 的值分别是 10、11、12。

```
enum day weekend;
```

```
weekend = ( enum day ) a; //类型转换
```

```
//weekend = a; //错误
```

enum 可以通过枚举名加点通过枚举的元素访问枚举数值。例：

```
int i = (int)ASN.MON; 枚举值虽然是 int 类型的，但是也必须进行显示的强制转换。
```

枚举类型的是为了**提高程序的可读性和可维护性**。如果要处理颜色，使用 red 和 blue 比使用 0 和 1 更直观。注意，**枚举类型只能在内部使用**。如果要输入 color 中 orange 的值，只能输入 1，而不是单词 orange。或者，让程序先读入字符串 "orange"，再将其转换为 orange 代表的值。因为枚举类型是整数类型，所以可以在表达式中以使用整数变量的方式使用 enum 变量。它们用在 case 语句中很方便

虽然枚举符（如 red 和 blue）是 int 类型，但是枚举变量可以是任意整数类型，前提是该整数类型可以储存枚举常量。例如，spectrum 的枚举符范围是 0~5，所以编译器可以用 unsigned char 来表示 color 变量。

C 语言使用名称空间（namespace）标识程序中的各部分，即通过名称来识别。作用域是名称空间概念的一部分：两个不同作用域的同名变量不冲突；两个相同作用域的同名变量冲突。名称空间是分类别的。**在特定作用域中的结构标记、联合标记和枚举标记都共享相同的名称空间，该名称空间与普通变量使用的空间不同。这意味着在相同作用域中变量和标记的名称可以相同，不会引起冲突，但是不能在相同作用域中声明两个同名标签或同名变**

量。例如，在 C 中，下面的代码不会产生冲突： struct rect { double x; double y; }; int rect; // 在 C 中不会产生冲突 尽管如此，以两种不同的方式使用相同的标识符会造成混乱。

另外，**C++不允许这样做，因为它把标记名和变量名放在相同的名称空间中。**

```
typedef enum {false, true} bool
```

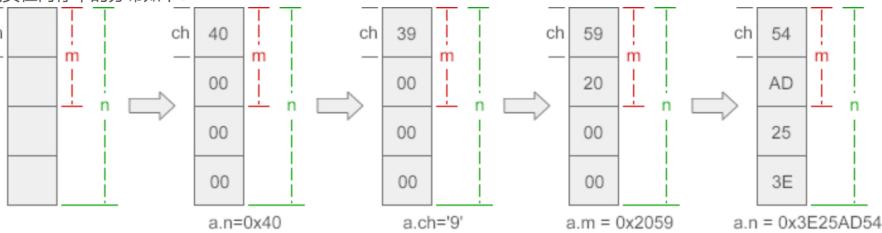
## 共用体

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。

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”到一起，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。

当tag=0元素结点由标志域，值域构成，当tag=1时，表结点由标志域，头指针域和尾指针域三部分组成。头指针域指向原子或者广义表结点，尾指针域为空或者指向本层中的下一个广义表结点。

```
typedef struct GenealNode{
    int tag;
    union{
        Datatype data;
        struct{
            struct GenealNode *hp,*tp;
        }ptr;
    };
}*Glist;
```

## 存储类别，链接和内存管理

### 存储类别

从硬件方面来看，被储存的每个值都占用一定的物理内存，C 语言把这样的一块内存称为对象（object）（在内存中的一段有意义的区域）。对象可以储存一个或多个值。一个对象可能并未储存实际的值，但是它在储存适当的值时一定具有相应的大小（面向对象编程中的对象指的是类对象，其定义包括数据和允许对数据进行的操作，C 不是面向对象编程语言）

从软件方面来看，程序需要一种方法访问对象。这可以通过声明变量来完成：int entity = 3; 该声明创建了一个名为 entity 的标识符（identifier）。标识符是一个名称，在这种情况下，标识符可以用来指定（designate）特定对象的内容。标识符遵循变量的命名规则。在该例中，标识符 entity 即是软件（即 C 程序）指定硬件内存中的对象的方式。该声明还提供了储存在对象中的值

第 1 行声明中，pt 是一个标识符，它指定了一个储存地址的对象。但是，表达式\*pt 不是标识符，因为它不是一个名称。然而，它确实指定了一个对象，在这种情况下，它与

`entity` 指定的对象相同。一般而言，那些指定对象的表达式被称为左值。所以，`entity` 既是标识符也是左值；`*pt` 既是表达式也是左值。按照这个思路，`ranks + 2 * entity` 既不是标识符（不是名称），也不是左值（它不指定内存位置上的内容）。但是表达式`(ranks + 2 * entity)`是一个左值，因为它的确指定了特定内存位置的值，即 `ranks` 数组的第 7 个元素。顺带一提，`ranks` 的声明创建了一个可容纳 10 个 `int` 类型元素的对象，该数组的每个元素也是一个对象。

```
const char * pc = "Behold a string literal!";
```

程序根据该声明把相应的字符串字面量储存在内存中，内含这些字符值的数组就是一个对象。由于数组中的每个字符都能被单独访问，所以每个字符也是一个对象。该声明还创建了一个标识符为 `pc` 的对象，储存着字符串的地址。由于可以设置 `pc` 重新指向其他字符串，所以标识符 `pc` 是一个可修改的左值。`const` 只能保证被 `pc` 指向的字符串内容不被修改，但是无法保证 `pc` 不指向别的字符串。由于`*pc` 指定了储存'B'字符的数据对象，所以`*pc` 是一个左值，但不是一个可修改的左值。与此类似，因为字符串字面量本身指定了储存字符串的对象，所以它也是一个左值，但不是可修改的左值。

可以用存储期（storage duration）描述对象，所谓存储期是指对象在内存中保留了多长时间。标识符用于访问对象，可以用作用域（scope）和链接（linkage）描述标识符，标识符的作用域和链接表明了程序的哪些部分可以使用它。不同的存储类别具有不同的存储期、作用域和链接。标识符可以在源代码的多文件中共享、可用于特定文件的任意函数中可仅限于特定函数中使用，甚至只在函数中的某部分使用。对象可存在于程序的执行期，也可以仅存在于它所在函数的执行期。对于并发编程，对象可以在特定线程的执行期存在。可以通过函数调用的方式显式分配和释放内存。

## 作用域

作用域描述程序中可访问标识符的区域。一个 C 变量的作用域可以是块作用域、函数作用域、函数原型作用域或文件作用域。到目前为止，本书程序示例中使用的变量几乎都具有块作用域。块是用一对花括号括起来的代码区域。例如，整个函数体是一个块，函数中的任意复合语句也是一个块。定义在块中的变量具有块作用域（block scope），块作用域变量的可见范围是从定义处到包含该定义的块的末尾。另外，虽然函数的形式参数声明在函数的左花括号之前，但是它们也具有块作用域，属于函数体这个块。所以到目前为止，我们使用的局部变量（包括函数的形式参数）都具有块作用域

声明在内层块中的变量，其作用域仅局限于该声明所在的块

以前，具有块作用域的变量都必须声明在块的开头。C99 标准放宽了这一限制，允许在块中的任意位置声明变量。因此，对于 `for` 的循环头，现在可以这样写：`for (int i = 0; i < 10; i++) printf("A C99 feature: i = %d", i);` 为适应这个新特性，C99 把块的概念扩展到包括 `for` 循环、`while` 循环、`do while` 循环和 `if` 语句所控制的代码，即使这些代码没有用花括号括起来，也算是块的一部分。所以，上面 `for` 循环中的变量 `i` 被视为 `for` 循环块的一部分，它的作用域仅限于 `for` 循环。一旦程序离开 `for` 循环，就不能再访问 `i`。

函数作用域（function scope）仅用于 `goto` 语句的标签。这意味着即使一个标签首次出现在函数的内层块中，它的作用域也延伸至整个函数。如果在两个块中使用相同的标签会很混乱，标签的函数作用域防止了这样的事情发生。

函数原型作用域（function prototype scope）用于函数原型中的形参名变量名），如下所示：`int mighty(int mouse, double large);` 函数原型作用域的范围是从形参定义处到原型声明结束。这意味着，编译器在处理函数原型中的形参时只关心它的类型，而形参名（如果有的话）通常无关紧要。而且，即使有形参名，也不必与函数定义中的形参名相匹

配。只有在变长数组中，形参名才有用： void use\_a\_VLA(int n, int m, ar[n][m]); 方括号中必须使用在函数原型中已声明的名称（或者用\*号代替）

变量的定义在函数的外面，具有文件作用域（file scope）。具有文件作用域的变量，从它的定义处到该定义所在文件的末尾均可见，由于这样的变量可用于多个函数，所以文件作用域变量也称为全局变量（global variable）。

你认为的多个文件在编译器中可能以一个文件出现。例如，通常在源代码（.c 扩展名）中包含一个或多个头文件（.h 扩展名）。头文件会依次包含其他头文件，所以会包含多个单独的物理文件。但是，C 预处理实际上是用包含的头文件内容替换#include 指令。所以，编译器源代码文件和所有的头文件都看成是一个包含信息的单独文件。这个文件被称为翻译单元（translation unit）。描述一个具有文件作用域的变量时，它的实际可见范围是整个翻译单元。如果程序由多个源代码文件组成（一般都是由多个组成的），那么该程序也将由多个翻译单元组成。每个翻译单元均对应一个源代码文件和它所包含的文件，分别编译，然后链接

在 C 语言术语中，翻译单元指 C 编译器产生目标文件（object file）的最终输入。在非正式使用情况下，翻译单元也叫编译单元。一个编译单元大致由一个经过 C 预处理器处理过的源文件组成，意味着由#include 指令列出的头文件会被正确的包含进来，由#ifndef 指令包含的代码会被包含进来，定义的宏会被展开。

由单元组成的 C 程序叫源文件（或者叫预处理文件），源文件除了源代码以外，还包括 C 预处理指令。一个源文件经过预处理器处理后的输出叫做翻译单元。

预处理主要包括将一个源文件中由#include 指令声明的文件（通常是头文件，也可能是其它源文件）递归地替换，产生的结果是一个预处理翻译单元。接下来包括对#define 指令进行宏展开，对#ifndef 指令进行条件编译等等；这一步便将预处理翻译单元转换成一个翻译单元。编译器从翻译单元产生一个目标文件，目标文件经过后续处理后链接（可能需要其它目标文件）成一个可执行程序。（库函数会自动链接，而自定义函数链接时需要添加语句）

需要注意的是预处理器是语言无关的，只是一个词法处理器，只在词法分析级别，它并不做语法分析，所以它不能处理具体的 C 语法。编译单元做为编译器的输入，它将不会看到任何预处理指令，因为在编译之前预处理指令已经被预处理器处理了。一个翻译单元根本上是基于一个文件，实际输入编译器的源代码可能和程序员所看到的大不一样，特别是递归包含的头文件。

## 存储期

作用域和链接描述了标识符的可见性。存储期描述了通过这些标识符访问的对象的生存期。C 对象有 4 种存储期：静态存储期、线程存储期、自动存储期、动态分配存储期。

如果对象具有静态存储期，那么它在程序的执行期间一直存在。文件作用域变量具有静态存储期。注意，对于文件作用域变量，关键字 static 表明了其链接属性，而非存储期。以 static 声明的文件作用域变量具有内部链接。但是无论是内部链接还是外部链接，所有的文件作用域变量都具有静态存储期

1) 全局变量是不显式用 static 修饰的全局变量，全局变量默认是有外部链接性的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 extern 全局变量名的声明，就可以使用全局变量。

2) 全局静态变量是显式用 static 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 extern 声明也不能使用

静态局部变量有以下特点：

该变量在全局数据区分配内存；

静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；

静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；  
它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束；

静态函数不能被其它文件所用；

其它文件中可以定义相同名字的函数，不会发生冲突；

块作用域变量也能具有静态存储期。为了创建这样的变量，要把 变量声明在块中，且在声明前面加上关键字 static：

变量 ct 储存在静态内存中，它从程序被载入到程序结束期间都存在。但是，它的作用域定义在 more() 函数块中。只有在执行该函数时，程序 才能使用 ct 访问它所指定的对象（但是，该函数可以给其他函数提供该存储区的地址以便间接访问该对象，例如通过指针形参或返回值）

线程存储期用于并发程序设计，程序执行可被分为多个线程。具有线程存储期的对象，从被声明时到线程结束一直存在。以关键字 \_Thread\_local 声明一个对象时，每个线程都获得该变量的私有备份

块作用域的变量通常都具有自动存储期。当程序进入定义这些变量的块 时，为这些变量分配内存；当退出这个块时，释放刚才为变量分配的内存。这种做法相当于把自动变量占用的内存视为一个可重复使用的工作区或暂存区。例如，一个函数调用结束后，其变量占用的内存可用于储存下一个被调用函数的变量。变长数组稍有不同，它们的存储期从声明处到块的末尾，而不是从块的开始处到块的末尾。

表12.1 5种存储类别

存储类别	存储期	作用域	链接	声明方式
自动	自动	块	无	块内
寄存器	自动	块	无	块内， 使用关键字 register
静态外部链接	静态	文件	外部	所有函数外
静态内部链接	静态	文件	内部	所有函数外， 使用关键字 static
静态无链接	静态	块	无	块内， 使用关键字 static

## 自动变量

属于自动存储类别的变量具有 **自动存储期、块作用域且无链接**。默认情况下，声明在 **块或函数头中的任何变量**都属于自动存储类别。为了更清楚地 表达你的意图（例如，为了表明有意覆盖一个外部变量定义，或者强调不要 把该变量改为其他存储类别），可以显式使用关键字 **auto**。  
块作用域和无链接意味着只有在变量定义所在的块中才能通过变量名访问该变量（当然，参数用于传递变量的值和地址给另一个函数，但是这是间接的方法）。另一个函数可以使用同名变量，但是该变量是储存在不同内存位置上的另一个变量。变量具有自动存储期意味着，程序在进入该变量声明所在的块时变量存在，程序在退出该块时变量消失。原来该变量占用的内存位置现在可做他用。

块中声明的变量仅限于该块及其包含的块使用，如果内层块中声明的变量与外层块中的变量同名会怎样？内层块会隐藏 外层块的定义。但是离开内层块后，外层块变量的作用域又回到了原来的作用域

我们使用的编译器在创建 while 循环体中的 x 时，并未复用内层块中 x 占用的内存，但是有些

编译器会这样做。

#### 自动变量不会初始化，除非显式初始化它

可以用非常量表达式（non-constant expression）初始化自动变量，前提是所用的变量已在前面定义过

## 寄存器变量

寄存器变量储存在 CPU 的寄存器中，或者概括地说，储存在最快的可用内存中。与普通变量相比，访问和处理这些变量的速度更快。由于寄存器变量储存在寄存器而非内存中，所以无法获取寄存器变量的地址。绝大多数方面，寄存器变量和自动变量都一样。也就是说它们都是块作用域、无链接和自动存储期。使用存储类别说明符 `register` 便可声明寄存器变量

编译器必须根据寄存器或最快可用内存的数量衡量你的请求，或者直接忽略你的请求，所以可能不会如你所愿。在这种情况下，寄存器变量就变成普通的自动变量。即使是这样，仍然不能对该变量使用地址运算符

可声明为 `register` 的数据类型有限。例如，处理器中的寄存器可能没有足够大的空间来储存 `double` 类型的值。

## 静态变量

静态的意思是该变量在内存中原地不动，并不是说它的值不变。具有文件作用域的变量自动具有（也必须是）静态存储期。前面提到过，可以创建具有静态存储期、块作用域的局部变量。这些变量和自动变量一样，具有相同的作用域，但是程序离开它们所在的函数后这些变量不会消失。也就是说，这种变量具有块作用域、无链接，但是具有静态存储期。计算机在多次函数调用之间会记录它们的值。在块中（提供块作用域和无链接）以存储类别说明符 `static`（提供静态存储期）声明这种变量。

#### 不能在函数的形参中使用 static

外部链接的静态变量具有文件作用域、外部链接和静态存储期。该类别有时称为外部存储类别（external storage class），属于该类别的变量称为外部变量（external variable）。把变量的定义性声明（defining declaration）放在所有函数的外面便创建了外部变量。当然，为了指出该函数使用了外部变量，可以在函数中用关键字 `extern` 再次声明。如果一个源代码文件使用的外部变量定义在另一个源代码文件中，则必须用 `extern` 在该文件中声明该变量。

外部变量具有静态存储期。因此，无论程序执行到 `main()`、`next()` 还是其他函数，数组 `Up` 及其值都一直存在。

外部变量和自动变量类似，也可以被显式初始化。与自动变量不同的是，如果未初始化外部变量，它们会被自动初始化为 0。这一原则也适用于外部定义的数组元素。与自动变量的情况不同，只能使用常量表达式初始化文件作用域变量（只要不是变长数组，`sizeof` 表达式可被视为常量表达式。）

这里，`tern` 被声明了两次。第 1 次声明为变量预留了存储空间，该声明构成了变量的定义。第 2 次声明只告诉编译器使用之前已创建的 `tern` 变量，所以这不是定义。第 1 次声明被称为定义式声明（defining declaration），第 2 次声明被称为引用式声明（referencing declaration）。关键字 `extern` 表明该声明不是定义，因为它指示编译器去别处查询其定义

编译器会假设 `tern` 实际的定义在该程序的别处，也许在别的文件中。该声明并不会引起分配存储空间。因此，不要用关键字 `extern` 创建外部定义，只用它来引用现有的外部定

义。

## 内部链接的静态变量

该存储类别的变量具有静态存储期、文件作用域和内部链接。在所有函数外部（这点与外部变量相同），用存储类别说明符 `static` 定义的变量具有这种存储类别。普通的外部变量可用于同一程序中任意文件中的函数，但是内部链接的静态变量 **只能用于同一个文件中的函数**。可以使用存储类别说明符 `extern`，在函数中 **重复声明任何具有文件作用域的变量**。这样的声明**并不会改变其链接属性**。

## 多文件

只有当程序由多个翻译单元组成时，才体现区别内部链接和外部链接的 **重要性**。C 通过在一个文件中进行定义式声明，然后在其他文件中进行引用式声明来实现共享。也就是说，除了一个定义式声明外，其他声明都要使用 `extern` 关键字。而且，只有定义式声明才能初始化变量。

注意，如果外部变量定义在一个文件中，那么其他文件在使用该变量之前必须先声明它（用 `extern` 关键字）。也就是说，在某文件中对外部变量进行定义式声明只是单方面允许其他文件使用该变量，**其他文件在用 `extern` 声明之前不能直接使用它**。

## 说明符小结

`auto` 说明符表明变量是自动存储期，只能用于块作用域的变量声明中。由于在块中声明的变量本身就具有自动存储期，所以**使用 `auto` 主要是为了明确表达要使用与外部变量同名的局部变量的意图**。

`register` 说明符也只用于块作用域的变量，它把变量归为寄存器存储类别，请求最快速度访问该变量。**同时，还保护了该变量的地址不被获取**。

用 `static` 说明符创建的对象具有静态存储期，**载入程序时创建对象，当程序结束时对象消失**。如果 `static` 用于文件作用域声明，**作用域受限于该文件**。如果 `static` 用于块作用域声明，作用域则受限于该块。因此，只要程序在运行对象就存在并保留其值，但是只有在执行块内的代码时，才能通过标识符访问。**块作用域的静态变量无链接。文件作用域的静态变量具有内部链接**。

`extern` 说明符表明声明的变量定义在别处。如果包含 `extern` 的声明具有文件作用域，则引用的变量**必须具有外部链接**。如果包含 `extern` 的声明具有块作用域，则引用的变量可能具有外部链接或内部链接，这取决于该变量的定义式声明。

自动变量具有块作用域、无链接、自动存储期。它们是局部变量，属于其定义所在块（通常指函数）私有。寄存器变量的属性和自动变量相同，但是编译器会使用更快的内存或寄存器储存它们。不能获取寄存器变量的地址。具有静态存储期的变量可以具有外部链接、内部链接或无链接。在同一个文件所有函数的外部声明的变量是外部变量，具有文件作用域、外部链接和静态存储期。如果在这种声明前面加上关键字 `static`，那么其声明的变量具有文件作用域、内部链接和静态存储期。如果在函数中用 `static` 声明一个变量，则该变量具有块作用域、无链接、静态存储期。具有自动存储期的变量，程序在进入该变量的声明所在块时才为其分配内存，在退出该块时释放之前分配的内存。如果未初始化，自动变量中是垃圾值。程序在编译时为具有静态存储期的变量分配内存，并在程序的运行过程中一直保留这块内存。如果未初始化，这样的变量会被设置为 0。**具有块作用域的变量是局部的，属于包含该声明的块私有。具有文件作用域的变量对文件（或翻译单元）中位于其**

声明后面的所有函数可见。具有外部链接的文件作用域变量，可用于该程序的其他翻译单元。具有内部链接的文件作用域变量，只能用于其声明所在的文件内。

#### 函数存储类别

因为以 static 存储类别说明符创建的函数属于特定模块私有。这样做 避免了名称冲突 的问题，在其它文件中可以引用。

通常的做法是：用 extern 关键字声明定义在其他文件中的函数。这样做 是为了表明当前文件中使用的函数被定义在别处。除非使用 static 关键字，否则一般函数声明都默认为 extern。

对于“使用哪种存储类别”的回答绝大多数是“自动存储类别”，要知道默认类别就是自动存储类别。初学者会认为外部存储类别很不错，为何不把所有的变量都设置成外部变量这样就不必使用参数和指针在函数间传递信息了。然而，这背后隐藏着一个陷阱。如果这样做，A()函数可能违背你的意图，私下修改 B()函数使用的变量。多年来，无数程序员的经验表明，随意使用外部存储类别的变量导致的后果远远超过了它所带来的便利。唯一例外的是 const 数据。因为它们在初始化后就不会被修改，所以不用担心它们被意外篡改。

保护性程序设计的黄金法则是：“按需知道”原则。尽量在函数内部解决该函数的任务，只共享那些需要共享的变量。除自动存储类别外，其他存储类别也很有用。不过，在使用某类别之前先要考虑一下是否有必要这样做。

## 内存分配

malloc() 函数，该函数接受一个参数：所需的内存字节数。malloc() 函数会找到合适的空闲内存块，这样的内存是匿名的。也就是说，malloc() 分配内存，但是不会为其赋名。然而，它确实 返回动态分配内存块的首字节地址。因此，可以把该地址赋给一个指针变量，并使用指针访问这块内存。因为 char 表示 1 字节，malloc() 的返回类型通常被定义为指向 char 的指针。然而，从 ANSI C 标准开始，C 使用一个新的类型：指向 void 的指针。该类型相当于一个“通用指针”。malloc() 函数可用于返回指向数组的指针、指向结构的指针等，所以通常该函数的返回值会被强制转换为匹配的类型。在 ANSI C 中，应该坚持使用强制类型转换，提高代码的可读性（便于转换为 c++ 程序）。然而，把指向 void 的指针赋给任意类型的指针完全不用考虑类型匹配的问题。如果 malloc() 分配内存失败，将返回空指针。

数组名是该数组首元素的地址。因此，如果让 ptd 指向这个块的首元素，便可像使用数组名一样使用它。也就是说，可以使用表达式 ptd[0] 访问该块的首元素，ptd[1] 访问第 2 个元素，以此类推。根据前面所学的知识，可以使用数组名来表示指针，也可以用指针来表示数组。

声明数组时，用常量表达式表示数组的维度，用数组名访问数组的元素。可以用静态内存或自动内存创建这种数组。声明变长数组（C99 新增的特性）时，用变量表达式表示数组的维度，用数组名访问数组的元素。具有这种特性的数组只能在自动内存中创建。声明一个指针，调用 malloc()，将其返回值赋给指针，使用指针访问数组的元素。该指针可以是静态的或自动的。

通常，malloc() 要与 free() 配套使用。free() 函数的参数是之前 malloc() 返回的地址，该函数释放之前 malloc() 分配的内存。因此，动态分配内存的存储期从调用 malloc() 分配内存到调用 free() 释放内存为止。设想 malloc() 和 free() 管理着一个内存池。每次调用 malloc()

分配内存给程序使用，每次调用 free() 把内存归还内存池中，这样便可重复使用这些内存。  
free() 的参数应该是一个指针，指向由 malloc() 分配的一块内存。不能用 free() 释放通过其他方式（如，声明一个数组）分配的内存。malloc() 和 free() 的原型都在 stdlib.h 头文件中

使用 malloc()，程序可以在运行时才确定数组大小。free() 函数位于程序的末尾，它释放了 malloc() 函数分配的内存。free() 函数只释放其参数指向的内存块。一些操作系统在程序结束时会自动释放动态分配的内存，但是有些系统不会。**为保险起见，请使用 free()，不要依赖操作系统来清理。**

第 1 次调用 gobble() 时，它创建了指针 temp，并调用 malloc() 分配了 16000 字节的内存（假设 double 为 8 字节）。假设如代码注释所示，遗漏了 free()。当函数结束时，作为自动变量的指针 temp 也会消失。但是它所指向的 16000 字节的内存却仍然存在。由于 temp 指针已被销毁，**所以无法访问这块内存，它也不能被重复使用**，因为代码中没有调用 free() 释放这块内存

#### calloc

newmem = (long \*)calloc(100, sizeof (long)); 和 malloc() 类似，在 ANSI 之前，calloc() 也返回指向 char 的指针；在 ANSI 之后，返回指向 void 的指针。如果要储存不同的类型，应使用强制类型转换运算符。calloc() 函数接受两个无符号整数作为参数（ANSI 规定是 size\_t 类型）。**第 1 个参数是所需的存储单元数量，第 2 个参数是存储单元的大小（以字节为单位）**

calloc() 函数还有一个特性：它把块中的所有位都设置为 0（注意，在某些硬件系统中，不是把所有位都设置为 0 来表示浮点值 0）。free() 函数也可用于释放 calloc() 分配的内存。

使用 malloc() 函数的程序开始时（内存空间还没有被重新分配）能正常运行，但经过一段时间后（内存空间已被重新分配）可能会出现问题，因此在使用它之前必须先进行初始化（可用 memset 函数 对其初始化为 0），但调用 calloc() 函数分配到的空间在分配时就已经被初始化为 0 了。当你在 calloc() 函数和 malloc() 函数之间作选择时，你需考虑是否要初始化所分配的内存空间，从而来选择相应的函数。

#### realloc

指针名 = (数据类型\*) realloc（要改变内存大小的指针名，新的大小）。

新的大小可大可小（如果新的大小大于原内存大小，则新分配部分不会被初始化；如果新的大小小于原内存大小，可能会导致数据丢失 [1-2]）

## 类型限定符

### (1) const

对象的类型如果采用了限定符 const，则该对象就是常量。在定义该对象之后，程序无法修改它。

### (2) volatile

对象的类型如果采用了限定符 volatile，则该对象可以被其他进程或事件修改。关键字 volatile 告诉编译器在每次使用该对象值时，都要重新读取它，即便程序本身自上一次获取后再没有修改过它的值。这种限定符通常用于硬件接口编程，以**防止变量值被外部事件修改后，未能及时更新。**

### (3) restrict

限定符 `restrict` 只适用于对象指针类型。这种类型限定符是 C99 新增加的，用来告诉编译器，一个被指针所引用的对象，如果它可以被修改，那么只能被 `restrict` 限定的指针修改，**不能被除该指针以外的其他任何方式修改，无论是直接方式还是间接方式。** 该特性允许编译器采用特定优化技术，而该技术可能需要 `restrict` 特性的支撑。编译器也有可能会忽略限定符 `restrict`，而不对程序结果造成任何影响。

```
int ar[10]; int * restrict restar = (int *) malloc(10 * sizeof(int)); int * par = ar;
for (n = 0; n < 10; n++) {par[n] += 5; restar[n] += 5; ar[n] *= 2; par[n] += 3; restar[n] += 3; }由于之前声明了 restar 是访问它所指向的数据块的唯一且初始的方式，编译器可以把涉及 restar 的两条语句替换成下面这条语句，效果相同： restar[n] += 8; /* 可以进行替换 */
/* 但是，如果把与 par 相关的两条语句替换成下面的语句，将导致计算错误： par[n] += 8;
/* 给出错误的结果 */ 这是因为 for 循环在 par 两次访问相同的数据之间，用 ar 改变了该数据的值。在本例中，如果未使用 restrict 关键字，编译器就必须假设最坏的情况（即，在两次使用指针之间，其他的标识符可能已经改变了数据）。如果用了 restrict 关键字，编译器就可以选择捷径优化计算。（其它的标识符不可能改变数据）
```

### (4) \_Atomic

采用类型限定符 `_Atomic` 声明的对象是一个原子对象（atomic object）。数组不能是原子对象。对原子对象的支持是可选的：C11 实现版本中，如果定义了宏 `_STDC_NO_ATOMICS_`，则表示程序不能声明原子对象。**当一个线程对一个原子类型的对象执行原子操作时，其他线程不能访问该对象**

编译器也可以将 `const` 对象存储在内存中的只读区域，但是 `volatile` 对象不能。如果程序不使用 `volatile` 对象的地址，也有可能编译器干脆不为该对象分配内存。

对象如果同时使用了限定符 `const` 和 `volatile`，如下例中 `ticks` 对象所示，它不能被程序本身修改，但可以被其他事件修改，例如时钟芯片的中断处理程序 `const volatile int ticks;`

## 小结

内存用于存储程序中的数据，由存储期、作用域和链接表征。存储期可以是静态的、自动的或动态分配的。如果是静态存储期，在程序开始执行时分配内存，并在程序运行时都存在。如果是自动存储期，在程序进入变量定义所在块时分配变量的内存，在程序离开块时释放内存。如果是动态分配存储期，在调用 `malloc()`（或相关函数）时分配内存，在调用 `free()` 函数时释放内存。作用域决定程序的哪些部分可以访问某数据。定义在所有函数之外的变量具有文件作用域，对位于该变量声明之后的所有函数可见。定义在块或作为函数形参内的变量具有块作用域，只对该块以及它包含的嵌套块可见。链接描述定义在程序某翻译单元中的变量可被链接的程度。具有块作用域的变量是局部变量，无链接。具有文件作用域的变量可以是内部链接或外部链接。内部链接意味着只有其定义所在的文件才能使用该变量。外部链接意味着其他文件使用也可以使用该变量。下面是 C 的 5 种存储类别

（不包括线程的概念）。自动——在块中不带存储类别说明符或带 `auto` 存储类别说明符声明的变量（或作为函数头中的形参）属于自动存储类别，具有自动存储期、块作用域、无链接。如果未初始化自动变量，它的值是未定义的。寄存器——在块中带 `register` 存储类别说明符声明的变量（或作为函数头中的形参）属于寄存器存储类别，具有自动存储期、块作用域、无链接，且**无法获取其地址**。把一个变量声明为寄存器变量即请求编译器将其

储存到访问速度最快的区域。如果未初始化寄存器变量，它的值是未定义的。静态、无链接——在块中带 static 存储类别说明符声明的变量属于“静态、无链接”存储类别，具有静态存储期、块作用域、无链接。只在编译时被初始化一次。如果未显式初始化，它的字节都被设置为 0。

静态、外部链接——在所有函数外部且没有使用 static 存储类别说明符声明的变量属于“静态、外部链接”存储类别，具有静态存储期、文件作用域、外部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。静态、内部链接——在所有函数外部且使用了 static 存储类别说明符明的变量属于“静态、内部链接”存储类别，具有静态存储期、文件作用域、内部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。动态分配的内存由 malloc()（或相关）函数分配，该函数返回一个指向指定字节数内存块的指针。这块内存被 free() 函数释放后便可重复使用，free() 函数以该内存块的地址作为参数。类型限定符 const、volatile、restrict 和\_atomic。const 限定符限定数据在程序运行时不能改变。对指针使用 const 时，可限定指针本身不能改变或指针指向的数据不能改变，这取决于 const 在指针声明中的位置。volatile 限定符表明，限定的数据除了被当前程序修改外还可以被其他进程修改。该限定符的目的是警告编译器不要进行假定的优化不要自以为是）。restrict 限定符也是为了方便编译器设置优化方案。restrict 限定的指针是访问它所指向数据的唯一途径。

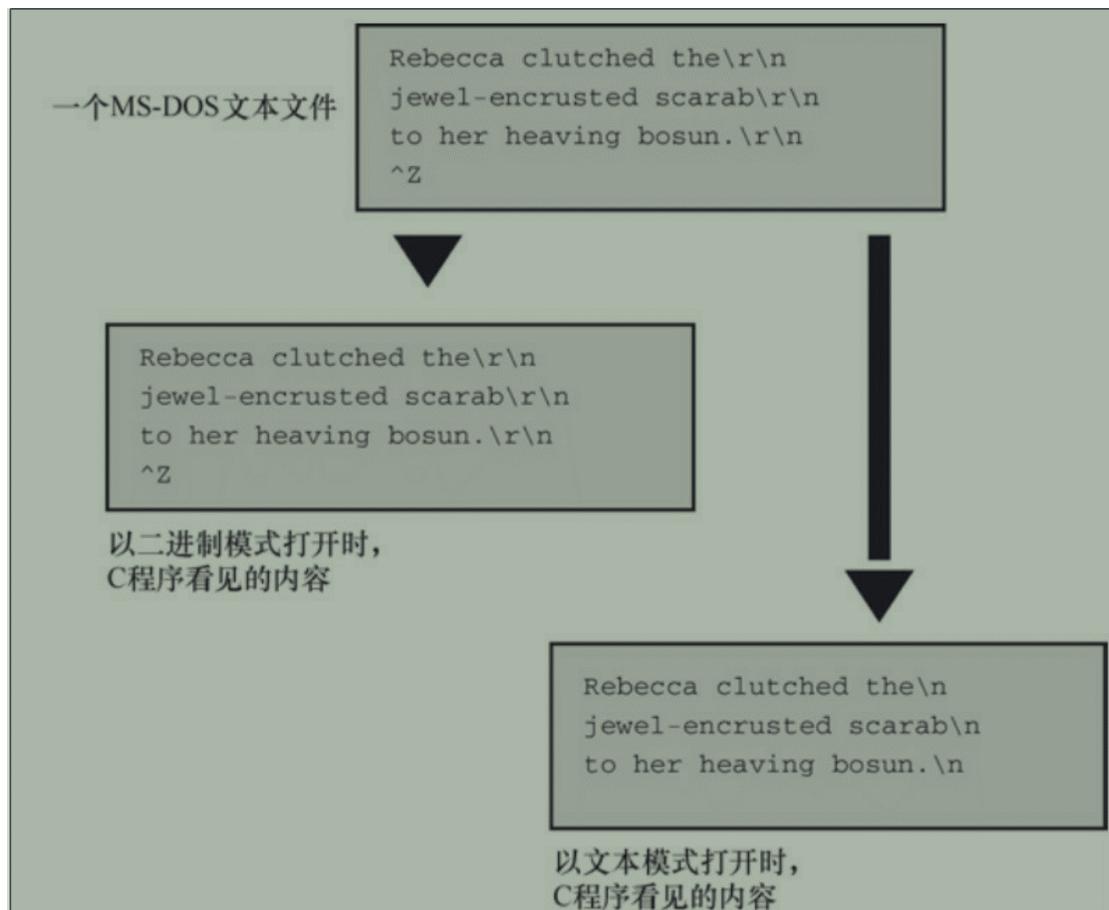
## 文件输入/输出

文件（file）通常是在磁盘或固态硬盘上的一段已命名的存储区

C 把文件看作是一系列连续的字节，每个字节都能被单独读取。这与 UNIX 环境中（C 的发源地）的文件结构相对应。由于其他环境中可能无法完全对应这个模型，C 提供两种文件模式：文本模式和二进制模式。

所有文件的内容都以二进制形式（0 或 1）储存。但是，如果文件最初使用二进制编码的字符（例如，ASCII 或 Unicode）表示文本（就像 C 字符串那样），该文件就是文本文件，其中包含文本内容。如果文件中的二进制值代表机器语言代码或数值数据（使用相同的内部表示，假设，用于 long 或 double 类型的值）或图片或音乐编码，该文件就是二进制文件，其中包含二进制内容。

为了规范文本文件的处理，C 提供两种访问文件的途径：二进制模式和文本模式。在二进制模式中，程序可以访问文件的每个字节。而在文本模式中，程序所见的内容和文件的实际内容不同。程序以文本模式读取文件时，把本地环境表示的行末尾或文件结尾映射为 C 模式。例如，C 程序在旧式 Macintosh 中以文本模式读取文件时，把文件中的\r 转换成\n；以文本模式写入文件时，把\n 转换成\r。或者，C 文本模式程序在 MS-DOS 平台读取文件时，把\r\n 转换成\n；写入文件时，把\n 转换成\r\n。在其他环境中编写的文本模式程序也会做类似的转换。



## 标准 I/O

除了选择文件的模式，大多数情况下，还可以选择 I/O 的两个级别（即处理文件访问的两个级别）。**底层 I/O**（low-level I/O）使用操作系统提供的**基本 I/O 服务**。**标准高级 I/O**（standard high-level I/O）使用 C 库的标准包和 stdio.h 头文件定义。因为无法保证所有的操作系统都使用相同的底层 I/O 模型，C 标准只支持**标准 I/O 包**。有些实现会提供底层库，但是 C 标准建立了可移植的 I/O 模型，我们主要讨论这些 I/O。

C 程序会自动打开 3 个文件，它们被称为**标准输入**（standard input）、**标准输出**（standard output）和**标准错误输出**（standard error output）。在默认情况下，标准输入是系统的普通输入设备，通常为键盘；标准输出和标准错误输出是系统的普通输出设备，通常为显示屏。

与底层 I/O 相比，标准 I/O 包除了可移植以外还有两个好处。第一，标准 I/O 有许多专门的函数简化了处理不同 I/O 的问题。例如，printf() 把不同形式的数据转换成与终端相适应的字符串输出。第二，输入和输出都是缓冲的。也就是说，一次转移一大块信息而不是一字节信息（通常至少 512 字节）。例如，当程序读取文件时，一块数据被拷贝到缓冲区（一块中介存储区域）。这种缓冲极大地提高了数据传输速率。程序可以检查缓冲区中的字节。stdout（标准输出），输出方式是行缓冲。输出的字符会先存放在缓冲区，等按下回车键时才进行实际的 I/O 操作。

stderr（标准出错），是不带缓冲的，这使得出错信息可以直接尽快地显示出来。

# 打开和关闭文件

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()` 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。文件指针（该例中是 `fp`）的类型是指向 `FILE` 的指针，`FILE` 是一个定义在 `stdio.h` 中的派生类型。文件指针 `fp` 并不指向实际的文件，它指向一个包含文件信息的数据对象，其中包含操作文件的 I/O 函数所用的缓冲区信息。因为标准库中的 I/O 函数使用缓冲区，所以它们 不仅要知道缓冲区的位置，还要知道缓冲区被填充的程度以及操作哪一个文件。标准 I/O 函数根据这些信息在必要时决定再次填充或清空缓冲区。`fp` 指向的数据对象包含了这些信息

`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。

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：

```
FILE *fp;
if( (fp=fopen("D:\\demo.txt","rb") == NULL ){
    printf("Fail to open file!\\n");
    exit(0); //退出程序（结束程序）
}
```

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。

与文件是否存在无关

```
fopen("文件路径","r",stdin);
fopen("文件路径", "w",stdout);
```

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(binary)：二进制文件

+：更新

像 UNIX 和 Linux 这样只有一种文件类型的系统，带 b 字母的模式和不带 b 字母的模式相同。

新的 C11 新增了带 x 字母的写模式，与以前的写模式相比具有更多特性。第一，如果以传统的一种写模式打开一个现有文件，fopen() 会把该文件的长度截为 0，这样就丢失了该文件的内容。但是使用带 x 字母的写模式，即使 fopen() 操作失败，原文件的内容也不会被删除。第二，如果环境允许，x 模式的独占特性使得其他程序或线程无法访问正在被打开的文件。

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。

## 读写文件

### 以字符形式

`fgetc` 是 `file get char` 的缩写，意思是从此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：

```
char ch;
```

```
FILE *fp = fopen("D:\\demo.txt", "r+");
```

```
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。

```
//每次读取一个字节，直到读取完毕
```

```
while( (ch=fgetc(fp)) != EOF){
```

```
    putchar(ch);
```

}赋值语句的返回值就是所赋的值

`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

```
if(ferror(fp)){
    puts("读取出错");
} else{
    puts("读取成功");
}
```

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。**除了读写(r+)**，**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。

## 以字符串形式

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。

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。

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。

和 puts() 函数不同， fputs() 在打印字符串时不会在其末尾添加换行符

## 以数据块形式（慎用）

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()**。

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，它可以指向数组、变量、结构体，一块分配过的内存等。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。（每次读的字节数，有时需设为 1，此时效率比较低）

count：表示要读写的数据块的块数。（读的次数）

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 perror() 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 perror() 或 feof() 检测。

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

虽然结构中有些内容是文本，但是 value 成员不是文本。如果使用文本编辑器 查看 book.dat，该结构本文部分的内容显示正常，但是 **数值部分的内容不可读，甚至会导致文本编辑器出现乱码**

## 格式化读写（重点）

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, ... );
```

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

`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。

在空白符这个意义上讲，`fscanf` 对空格、制表符、换行符是一视同仁的，不加区分的；`%s` 会跳过前面的空白符，但是不会跳过后面的空白符；`%c` 不会跳过空白符。

2. `*` 表示读取一个域，但是不赋值给变量。

3. `[]` 表示只读取中括号内的字符，`[]` 表示不读取中括号内的字符，值得注意的是`%[]s` 将不会跳过前面的空白符。

4. 如果还没有任何一个域匹配成功或者任何一个匹配失败发生之前，就达到了文件流末尾，就算出错；或者读取文件流出错。这两种情况下，`fscanf` 返回 EOF。

当作为空白符时，`format str` 中的空格、制表符以及换行符是一样的，可以相互替代！

`scanf()` 默认把空白作为分割多个输入的标志，所以当你输入空白的时候，`scanf()` 会认为你将要进行下一个输入，从而停下来等你输入下一个非空白数据。可以用来读取多个间隔数据中的字符（不用空格会读取数据间隔里的空白符）；

## 随机读写

`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()` 函数可以移动文件的读写指针到指定的位置，即移动当前文件的位置指针，其原型为：

```
int fseek(FILE * stream, long offset, int fromwhere);
```

参数 `offset` 为根据参数 `fromwhere` 来移动读写位置的位移数。参数 `fromwhere` 为下列其中一种：

`SEEK_SET`: 从文件开始处向后移动 `offset`;

`SEEK_CUR`: 以目前的读写位置往后增加 `offset` 个位移量；前面加上 `rewind(fp)` 就相当于 `SEEK_SET`；

`SEEK_END`: 将读写位置指向文件尾后再增加 `offset` 个位移量。

`offset` 为正时，向后移动；`offset` 为负时，向前移动。

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。根据情况调整

```
fseek(fp, sizeof(struct stu), SEEK_SET); //移动位置指针
```

## 实现文件复制功能

实现文件复制的主要思路是：开辟一个缓冲区，不断从原文件中读取内容到缓冲区，每读取完一次就将缓冲区中的内容写入到新建的文件，直到把原文件的内容读取完。

这里有两个关键的问题需要解决：

1) 开辟多大的缓冲区合适？缓冲区过小会造成读写次数的增加，过大也不能明显提高效率。

目前大部分磁盘的扇区都是 4K 对齐的，如果读写的数据不是 4K 的整数倍，就会跨扇区读取，降低效率，所以我们开辟 4K 的缓冲区。

2) 缓冲区中的数据是没有结束标志的，如果缓冲区填充不满，如何确定写入的字节数？最好的办法就是每次读取都能返回读取到的字节数。

fread() 的原型为：

```
size_t fread ( void *ptr, size_t size, size_t count, FILE *fp );
```

它返回成功读写的块数，该值小于等于 count。如果我们让参数 size 等于 1，那么返回的就是读取的字节数。

注意：fopen()一定要以二进制的形式打开文件，不能以文本形式打开，否则系统会对文件进行一些处理，如果是文本文件，像.txt 等，可能没有问题，但如果是其他格式的文件，像.mp4, .rmvb, .jpg 等，复制后就会出错，无法读取

```
33. FILE *fpRead; // 指向要复制的文件  
34. FILE *fpWrite; // 指向复制后的文件  
35. int bufferLen = 1024*4; // 缓冲区长度  
36. char *buffer = (char*)malloc(bufferLen); // 开辟缓存  
37. int readCount; // 实际读取的字节数  
38.  
39. if( (fpRead=fopen(fileRead, "rb")) == NULL || (fpWrite=fopen(fileWrite, "wb")) == NULL ){  
40. printf("Cannot open file, press any key to exit!\n");  
41. getch();  
42. exit(1);  
43. }  
44.  
45. // 不断从fileRead读取内容，放在缓冲区，再将缓冲区的内容写入fileWrite  
46. while( (readCount=fread(buffer, 1, bufferLen, fpRead)) > 0 ){  
47. fwrite(buffer, readCount, 1, fpWrite);  
48. }  
49.  
50. free(buffer);  
51. fclose(fpRead);
```

## 获取文件大小

以 fopen 打开的文件，通过 fseek 可以定位到文件尾，这时使用 **fseek** 函数，返回的文件指针偏移值，就是文件的实际大小。

```
fseek(fp, 0, SEEK_END); //定位文件指针到文件尾。  
size=f.tell(fp); //获取文件指针偏移量，即文件大小。
```

## 标准 I/O 机理

通常，使用标准 I/O 的第 1 步是调用 `fopen()` 打开文件（前面介绍过，C 程序会自动打开 3 种标准文件）。`fopen()` 函数不仅打开一个文件，还创建了一个缓冲区（在读写模式下会创建两个缓冲区）以及一个包含文件和缓冲区数据的结构。另外，`fopen()` 返回一个指向该结构的指针，以便其他函数知道如何找到该结构。假设把该指针赋给一个指针变量 `fp`，我们说 `fopen()` 函数“打开一个流”。如果以文本模式打开该文件，就获得一个文本流；如果以二进制模式打开该文件，就获得一个二进制流。这个结构通常包含一个指定流中当前位置的文件位置指示器。除此之外，它还包含错误和文件结尾的指示器、一个指向缓冲区开始处的指针、一个文件标识符和一个计数（统计实际拷贝进缓冲区的字节数）。我们主要考虑文件输入。通常，使用标准 I/O 的第 2 步是调用一个定义在 `stdio.h` 中的输入函数，如 `fscanf()`、`getc()` 或 `fgets()`。一调用这些函数，文件中的数据块就被拷贝到缓冲区中。缓冲区的大小因实现而异，一般是 512 字节或是它的倍数，如 4096 或 16384（随着计算机硬盘容量越来越大，缓冲区的大小也越来越大）。最初调用函数，除了填充缓冲区外，还要设置 `fp` 所指向的结构中的值。尤其要设置流中的当前位置和拷贝进缓冲区的字节数。通常，当前位置从字节 0 开始。在初始化结构和缓冲区后，输入函数按要求从缓冲区中读取数据。在它读取数据时，文件位置指示器被设置为指向刚读取字符的下一个字符。由于 `stdio.h` 系列的所有输入函数都使用相同的缓冲区，所以调用任何一个函数都将从上一次函数停止调用的位置开始。当输入函数发现已读完缓冲区中的所有字符时，会请求把下一个缓冲大小的数据块从文件拷贝到该缓冲区中。以这种方式，输入函数可以读取文件中的所有内容，直到文件结尾。函数在读取缓冲区中的最后一个字符后，把结尾指示器设置为真。于是，下一次被调用的输入函数将返回 EOF。输出函数以类似的方式把数据写入缓冲区。当缓冲区被填满时，数据将被拷贝至文件中。

# 位运算

## 原码，补码，反码

### (1) 原码

将一个整数转换成二进制形式，就是其原码。例如`short a = 6;` a的原码就是0000 0000 0000 0110;更改a的值`a = -18,` 此时a的原码就是1000 0000 0001 0010。

通俗的理解，原码就是一个整数本来的二进制形式

### (2) 反码

谈到反码，需要将正数和负数区别对待，因为它们的反码不一样。

对于正数，它的反码就是其原码（原码和反码相同）；负数的反码就是将原码中除符号位以外的所有位（数值位）取反，也就是0变成1,1变成0.。例如`short a = 6,` a的原码和反码都是0000 0000 0000 0110；更改a的值`a = -18,` 此时a的反码是1111 1111 1110 1101.

### (3) 补码

正数和负数的补码也不一样，也要区别对待。

对于正数，它的补码就是其原码（原码、反码、补码都相同）；负数的补码是其反码加 1。例如`short a = 6;`, a 的原码、反码、补码都是0000 0000 0000 0110；更改 a 的值`a = -18;`，此时 a 的补码是1111 1111 1110 1110。

可以认为，补码是在反码的基础上打了一个补丁，进行了一下修正，所以叫“补码”。

原码、反码、补码的概念只对负数有实际意义，对于正数，它们都一样。

小数减去大数，结果为负数，之前（负数从反码转换为补码要加 1）加上的 1，后来（负数从补码转换为反码要减 1）还要减去，正好抵消掉，所以不会受影响。

而大数减去小数，结果为正数，之前（负数从反码转换为补码要加 1）加上的 1，后来（正数的补码和反码相同，从补码转换为反码不用减 1）就没有再减去，不能抵消掉，这就相当于给计算结果多加了一个 1。

**为了解决原码做减法的问题，出现了反码：**

计算十进制的表达式： $1 - 1 = 0$

$$1 - 1 = 1 + (-1) = [0000\ 0001]_{\text{原}} + [1000\ 0001]_{\text{原}} = [0000\ 0001]_{\text{反}} + [1111\ 1110]_{\text{反}} = [1111\ 1111]_{\text{反}} = [1000\ 0000]_{\text{原}} = -0$$

发现用反码计算减法，结果的真值部分是正确的。而唯一的问题其实就出现在"0"这个特殊的数值上，虽然人们理解上+0和-0是一样的，但是0带符号是没有任何意义的，而且会有[0000 0000]原和[1000 0000]原两个编码表示0。

**于是补码的出现，解决了0的符号问题以及0的两个编码问题：**

$$1 - 1 = 1 + (-1) = [0000\ 0001]_{\text{原}} + [1000\ 0001]_{\text{原}} = [0000\ 0001]_{\text{补}} + [1111\ 1111]_{\text{补}} = [1\ 0000\ 0000]_{\text{补}} = [0000\ 0000]_{\text{原}} \text{ 注意：进位1不在计算机字长里。}$$

**这样0用[0000 0000]表示，而以前出现问题的-0则不存在了。而且可以用[1000 0000]表示-128：-128的由来如下：**

表15.2 十进制、十六进制和等价的二进制

十进制	十六进制	等价二进制	十进制	十六进制	等价二进制
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

## 大端小端和内存对齐

1) 大端模式 (Big-endian) 是指将数据的低位 (比如 1234 中的 34 就是低位) 放在内存的高地址上，而数据的高位 (比如 1234 中的 12 就是高位) 放在内存的低地址上。这种存储模式有点儿类似于把数据当作字符串顺序处理，地址由小到大增加，而数据从高位往低位存放。

2) 小端模式 (Little-endian) 是指将数据的低位放在内存的低地址上，而数据的高位放在内存的高地址上。这种存储模式将地址的高低和数据的大小结合起来，高地址存放数值较大的部分，低地址存放数值较小的部分，这和我们的思维习惯是一致，比较容易理解。

对于一次能处理多个字节的CPU，必然存在着如何安排多个字节的问题，也就是大端和小端模式。以 int 类型的 0x12345678 为例，它占用 4 个字节，如果是小端模式 (Little-endian)，那么在内存中的分布情况为（假设从地址 0x4000 开始存放）：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

如果是大端模式 (Big-endian)，那么分布情况正好相反：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

我们的 PC 机上使用的是 X86 结构的 CPU，它是小端模式；51 单片机是大端模式；很多 ARM、DSP 也是小端模式（部分 ARM 处理器还可以由硬件来选择是大端模式还是小端模式）。

```

01. #include <stdio.h>
02. int main(){
03.     union{
04.         int n;
05.         char ch;
06.     } data;
07.
08.     data.n = 0x00000001; //也可以直接写作 data.n = 1;
09.     if(data.ch == 1){
10.         printf("Little-endian\n");
11.     }else{
12.         printf("Big-endian\n");
13.     }
14.     return 0;
15. }
```

共用体的各个成员是共用一段内存的。1 是数据的低位，如果 1 被存储在 data 的低字节（结构体中位域的申明一般是先声明高位），就是小端模式，这个时候 data.ch 的值也是 1 (0x01)。如果 1 被存储在 data 的高字节，就是大端模式，这个时候 data.ch 的值就是 0。

输出 int 时，先输出高位，再是低位。若是小端就是高地址->低地址，若是大端，低地址->高地址。

CPU 通过地址总线来访问内存，一次能处理几个字节的数据，就命令地址总线读取几个字节的数据。32 位的 CPU 一次可以处理 4 个字节的数据，那么每次就从内存读取 4 个字节的数据；少了浪费主频，多了没用。64 位的处理器也是这个道理，每次读取 8 个字节。以 32 位的 CPU 为例，实际寻址的步长为 4 个字节，也就是只对编号为 4 的倍数的内存寻址，例如 0、4、8、12、1000 等，而不会对编号为 1、3、11、1001 的内存寻址。

对于程序来说，一个变量最好位于一个寻址步长的范围内，这样一次就可以读取到变量的值；如果跨步长存储，就需要读取两次，然后再拼接数据，效率显然降低了。

例如一个 int 类型的数据，如果地址为 8，那么很好办，对编号为 8 的内存寻址一次就可以。如果编号为 10，就比较麻烦，CPU 需要先对编号为 8 的内存寻址，读取 4 个字节，得到该数据的前半部分，然后再对编号为 12 的内存寻址，读取 4 个字节，得到该数据的后半部分，再将这两部分拼接起来，才能取得数据的值。

如果不考虑内存对齐，结构体变量 t 所占内存应该为  $4+1+4=9$  个字节。考虑到内存对齐，虽然成员 b 只占用 1 个字节，但它所在的寻址步长内还剩下 3 个字节的空间，放不下下一个 int 型的变量了，所以要把成员 c 放到下一个寻址步长。剩下的这 3 个字节，作为内存填充浪费掉了。

将一个数据尽量放在一个步长之内，避免跨步长存储，这称为内存对齐。在 32 位编译模式下，默认以 4 字节对齐；在 64 位编译模式下，默认以 8 字节对齐。对于全局变量，GCC 在 Debug 和 Release 模式下都会进行内存对齐，而 VS 只有在 Release 模式下才会进行对

齐。而对于局部变量，GCC 和 VS 都不会进行对齐，不管是 Debug 模式还是 Release 模式。

## 位运算符

& 按位与 如果两个相应的二进制位都为1，则该位的结果值为1，否则为0  
| 按位或 两个相应的二进制位中只要有一个为1，该位的结果值为1  
^ 按位异或 若参加运算的两个二进制位值相同则为0，否则为1  
~ 取反 ~是一元运算符，用来对一个二进制数按位取反，即将0变1，将1变0  
<< 左移 用来将一个数的各二进制位全部左移N位，右补0  
>> 右移 将一个数的各二进制位右移N位，移到右端的低位被舍弃，对于无符号数，高位补0

按位与运算通常用来对某些位清0（掩盖），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）。

### 用0清零，用1保留

把掩码中的0看作不透明，1看作透明。表达式 $flags \& MASK$ 相当于用掩码覆盖在flags的位组合上，只有MASK为1的位才可见。

用 $\&=$ 运算符可以简化前面的代码，如下所示：`flags &= MASK;`下面这条语句是按位与的一种常见用法：`ch &= 0xff; /* 或者 ch &= 0377; */`前面介绍过0xff的二进制形式是11111111，八进制形式是0377。这个掩码保持ch中最后8位不变，其他位都设置为0。无论ch原来是8位、16位或是其他更多位，最终的值都被修改为1个8位字节。在该例中，**掩码的宽度为8位**。

### 检查位的值

覆盖flags中的其他位，只用1号位和MASK比较：`if ((flags & MASK) == MASK)`  
`puts("Wow!");`由于按位运算符的优先级比==低，所以必须在`flags & MASK`周围加上圆括号。为了避免信息漏过边界，**掩码至少要与其覆盖的值宽度相同**。`flag`取反检查0号位是否为0；(MASK)的0号位是1；

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）。

### 用1置1，用0保留（打开特定位）

`flags |= MASK;`同样，这种方法根据MASK中为1的位，把flags中对应的位设置为1，其他位不变

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）。

### 用1反转（切换），用0保留

值与MASK为0的位相对应的位不变。要切换flags中的1号位，可以使用下面两种方法：  
`flags = flags ^ MASK;` `flags ^= MASK;`例如，假设flags是00001111，MASK是10110110。表达式：`flags ^ MASK`即是：`(00001111) ^ (10110110)`// 表达式其结果为：`(10111001)`// 结果值flags中与MASK为1的位相对应的位都被切换了，MASK为0的位相对应的位不变。

```
ooosw = sweet >> 3; // ooosw = 2, sweet 的值仍然为 16
```

```
sweet >>=3; // sweet 的值为 2
```

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。

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。

对于有符号数，某些机器将对左边空出的部分

用符号位填补（即“算术移位”），而另一些机器则对左边空出的部分用 0 填补（即“逻辑移位”）。注意：对无符号数，右移时左边高位移入 0；对于有符号的值，如果原来符号位为 0（该数为正），则左边也是移入 0。如果符号位原来为 1（即负数），则左边移入 0 还是 1，要取决于所用的计算机系统。

```
0x7fffffffdfcac: 111111111111111111111111111111110
```

```
0x7fffffffdfca8: 1111111111111111111111111111111101
```

一般都是算数移位

内存查看 gdb

gdb 查看指定地址的内存地址的值：examine 简写 x----使用 gdb> help x 来查看使用方式  
x/ (n,f,u 为可选参数)

n: 需要显示的内存单元个数，也就是从当前地址向后显示几个内存单元的内容，一个内存单元的大小由后面的 u 定义

f: 显示格式

x(hex) 按十六进制格式显示变量。

d(decimal) 按十进制格式显示变量。

u(unsigned decimal) 按十进制格式显示无符号整型。

o(octal) 按八进制格式显示变量。

t(binary) 按二进制格式显示变量。

a(address) 按十六进制格式显示变量。

c(char) 按字符格式显示变量。

f(float) 按浮点数格式显示变量

u: 每个单元的大小，按字节数来计算。**默认是 4 bytes**。GDB 会从指定内存地址开始读取指定字节，并把其当作一个值取出来，并使用格式 f 来显示

b:1 byte h:2 bytes w:4 bytes g:8 bytes

比如 x/3uh 0x54320 表示从内存地址 0x54320 读取内容，h 表示以双字节为单位，3 表示输出 3 个单位，u 表示按照十六进制显示。

在 vscode 中使用需要加前缀-exec，可以放在监视窗口，查看一个 int 的二进制可以用 -exec x/t &a;

查看当前程序栈的内容: x/10x \$sp-->打印 stack 的前 10 个元素

查看当前程序栈的信息: info frame---list general info about the frame

查看当前程序栈的参数: info args---lists arguments to the function

查看当前程序栈的局部变量: info locals---list variables stored in the frame

查看当前寄存器的值: info registers(不包括浮点寄存器) info all-registers(包括浮点寄存器)

**二进制转换**

```

char * itobs(int n, char * ps)

{
    int i;

    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)

        ps[i] = (01 & n) + '0';

    ps[size] = '\0';

    return ps;
}

```

用  $1 \& n$  或  $01 \& n$  都可以。我们用八进制 1 而不是十进制 1，只是为了更接近计算机的表达方式

使用 limits.h 中的 CHAR\_BIT 宏，该宏表示 char 中的位数。sizeof 运算符返回 char 的大小，所以表达式 CHAR\_BIT \* sizeof(int) 表示 int 类型的位数。bin\_str 数组的元素个数是 CHAR\_BIT \* sizeof(int) + 1，留出一个位置给末尾的空字符

## 异或运算和文件加密

异或是一种基于二进制的位运算，用符号 XOR 或者  $\wedge$  表示，其运算法则是对运算符两侧数的每一个二进制位，同值取 0，异值取 1。它与布尔运算的区别在于，当运算符两侧均为 1 时，布尔运算的结果为 1，异或运算的结果为 0。

简单理解就是不进位加法，如  $1+1=0$ ,  $0+0=0$ ,  $1+0=1$ 。

性质

- 1、交换律
- 2、结合律（即  $(a \wedge b) \wedge c == a \wedge (b \wedge c)$ ）
- 3、对于任何数  $x$ ，都有  $x \wedge x = 0$ ,  $x \wedge 0 = x$
- 4、自反性  $A \text{ XOR } B \text{ XOR } B = A \text{ xor } 0 = A$

不设中间变量交换值

$A=A \text{ XOR } B \text{ (a XOR b)}$

$B=B \text{ XOR } A \text{ (b XOR a XOR b = a) } B=B \wedge (A \wedge B)=A$

A=A XOR B (a XOR b XOR a = b) A=(A^B)^A=B

1-1000 放在含有 1001 个元素的数组中，只有唯一的一个元素值重复，其它均只出现一次。每个数组元素只能访问一次，设计一个算法，将它找出来；不用辅助存储空间，能否设计一个算法实现？

解法一、显然已经有人提出了一个比较精彩的解法，将所有数加起来，减去  $1+2+\dots+1000$  的和。

这个算法已经足够完美了，相信出题者的标准答案也就是这个算法，唯一的问题是，如果数列过大，则可能会导致溢出。

解法二、异或就没有这个问题，并且性能更好。

将所有的数全部异或，得到的结果与  $1^2^3\dots^1000$  的结果进行异或，得到的结果就是重复数。

但是这个算法虽然很简单，但证明起来并不是一件容易的事情。这与异或运算的几个特性有关系。

首先是异或运算满足交换律、结合律。

所以， $1^2\dots^n\dots^1000$ ，无论这两个 n 出现在什么位置，都可以转换成为  $1^2\dots^{1000}(n^n)$  的形式。

其次，对于任何数 x，都有  $x^x=0$ ,  $x^0=x$ 。

所以  $1^2\dots^n\dots^1000 = 1^2\dots^{1000}(n^n) = 1^2\dots^{1000}0 = 1^2\dots^{1000}$  (即序列中除了 n 的所有数的异或)。

令， $1^2\dots^{1000}$  (序列中不包含 n) 的结果为 T

则  $1^2\dots^{1000}$  (序列中包含 n) 的结果就是  $T^n$ 。

$T^n(T^n)=n$ 。

所以，将所有的数全部异或，得到的结果与  $1^2^3\dots^1000$  的结果进行异或，得到的结果就是重复数。

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

int main(){
    char plaintext = 'a'; // 明文
    char secretkey = '!'; // 密钥
    char ciphertext = plaintext ^ secretkey; // 密文
    char decodetext = ciphertext ^ secretkey; // 解密后的字符
    char buffer[9];

    printf("      char  ASCII\n");
    // itoa()用来将数字转换为字符串，可以设定转换时的进制（基数）
    // 这里将字符对应的 ascii 码转换为二进制
    printf(" plaintext %c  %7s\n", plaintext, itoa(plaintext, buffer, 2));
    printf(" secretkey %c  %7s\n", secretkey, itoa(secretkey, buffer, 2));
    printf("ciphertext %c  %7s\n", ciphertext, itoa(ciphertext, buffer, 2));
    printf("decodetext %c  %7s\n", decodetext, itoa(decodetext, buffer, 2));

    return 0;
}
```

}

运行结果：

```
char ASCII  
plaintext a 1100001  
secretkey ! 100001  
ciphertext @ 1000000  
decodetext a 1100001
```

plaintext 与 decodetext 相同，也就是说，两次异或运算后还是原来的结果。

这就是加密的关键技术：

通过一次异或运算，生成密文，密文没有可读性，与原文风马牛不相及，这就是加密；

密文再经过一次异或运算，就会还原成原文，这就是解密的过程；

加密和解密需要相同的密钥，如果密钥不对，是无法成功解密的。

**如果加密和解密的密钥不同，则称为非对称加密算法。在非对称算法中，加密的密钥称为公钥，解密的密钥称为私钥，只知道公钥是无法解密的，还必须知道私钥。**

```
bool XorEncrypt(void* bufPtr, unsigned int bufferSize, const char* key, unsigned  
int keySize)  
{  
    if (!bufPtr || !key || keySize == 0)  
    {  
        return false;  
    }  
    char* ptr = (char*)bufPtr;  
  
    unsigned int index;  
    for (unsigned int i = 0; i < bufferSize; i++)  
    {  
        index = i%keySize;  
        ptr[i] ^= key[index];  
    }  
  
    return true;  
}
```

## 高级数据表示

### 抽象数据类型 (Abstract Data Type)

- 数据类型
  - 数据对象集
  - 数据集合相关联的操作集
- 抽象：描述数据类型的方法不依赖于具体实现
  - 与存放数据的机器无关
  - 与数据存储的物理结构无关
  - 与实现操作的算法和编程语言均无关

只描述数据对象集和相关操作集“**是什么**”，并不涉及“**如何做到**”的问题

## 复杂度和 time.h

运行效率体现在两方面：

算法的运行时间。（称为“时间复杂度”）

运行算法所需的内存空间大小。（称为“空间复杂度”）

好算法的标准就是：在符合算法本身的要求的基础上，使用算法编写的程序运行的时间短  
运行过程中占用的内存空间少，就可以称这个算法是“好算法”。

计算一个算法的时间复杂度，不可能把所有的算法都编写出实际的程序出来让计算机跑，这样会做很多无用功，效率太低。实际采用的方法是估算算法的时间复杂度。

在学习C语言的时候讲过，程序由三种结构构成：顺序结构、分支结构和循环结构。顺序结构和分支结构中的每段代码只运行一次；循环结构中的代码的运行时间要看循环的次数。

由于是估算算法的时间复杂度，相比而言，循环结构对算法的执行时间影响更大。所以，算法的时间复杂度，主要看算法中使用到的循环结构中代码循环的次数（称为“频度”）。次数越少，算法的时间复杂度越低。

例如：

- a) `++x; s=0;`
- b) `for (int i=1; i<=n; i++) { ++x; s+=x; }`
- c) `for (int i=1; i<=n; i++) { for (int j=1; j<=n; j++) { ++x; s+=x; } }`

上边这个例子中，a 代码的运行了 1 次，b 代码的运行了  $n$  次，c 代码运行了  $n^2$  次。

## 时间复杂度的表示

算法的时间复杂度的表示方式为：

$O(\text{频度})$

这种表示方式称为 大 “O” 记法。

### 常见的复杂度

执行次数	复杂度	非正式术语
12	$O(1)$	常数阶
$2n + 3$	$O(n)$	线性阶
$4n^2 + 2n + 6$	$O(n^2)$	平方阶
$4\log_2 n$	$O(\log n)$	对数阶
$3n + 2n\log_2 n + 15$	$O(n \log n)$	$n \log n$ 阶
$4n^3 + 3n^2 + 22n + 100$	$O(n^3)$	立方阶
$2^n$	$O(2^n)$	指数阶

注意： $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

- 若两段算法分别有复杂度  $T_1(n) = \Theta(f_1(n))$  和  $T_2(n) = \Theta(f_2(n))$ , 则
  - $T_1(n) + T_2(n) = \max(\Theta(f_1(n)), \Theta(f_2(n)))$
  - $T_1(n) \times T_2(n) = \Theta(f_1(n) \times f_2(n))$
- 若  $T(n)$  是关于  $n$  的  $k$  阶多项式, 那么  $T(n) = \Theta(n^k)$
- 一个 **for** 循环的时间复杂度等于循环次数乘以循环体代码的复杂度
- **if-else** 结构的复杂度取决于 **if** 的条件判断复杂度和两个分枝部分的复杂度, 总体复杂度取三者中最大

### 例3: 写程序计算给定多项式在给定点 $x$ 处的值

$$f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$$

```
double f( int n, double a[], double x )
{ int i;
  double p = a[0];
  for ( i=1; i<=n; i++ )
    p += (a[i] * pow(x, i));
  return p;
}
```

$$f(x) = a_0 + x(a_1 + x(\cdots(a_{n-1} + x(a_n))\cdots))$$

```
double f( int n, double a[], double x )
{ int i;
  double p = a[n];
  for ( i=n; i>0; i-- )
    p = a[i-1] + x*p;
  return p;
}
```

C/C++中的计时函数是clock()，而与其相关的数据类型是clock\_t。在MSDN中，查得对clock函数定义如下：

```
clock_t clock( void );
```

这个函数返回从“开启这个程序进程”到“程序中调用clock()函数”时之间的CPU时钟计时单元（clock tick）数，在MSDN中称之为挂钟时间（wall-clock）。其中clock\_t是用来保存时间的数据类型，在time.h文件中，我们可以找到对它的定义：

```
#ifndef _CLOCK_T_DEFINED  
typedef long clock_t;  
#define _CLOCK_T_DEFINED  
#endif
```

很明显，clock\_t是一个长整形数。在time.h文件中，还定义了一个常量CLOCKS\_PER\_SEC，它用来表示一秒钟会有多少个时钟计时单元，其定义如下：

```
#define CLOCKS_PER_SEC ((clock_t)1000)
```

可以看到每过千分之一秒（1毫秒），调用clock()函数返回的值就加1。下面举个例子，你可以使用公式clock() / CLOCKS\_PER\_SEC来计算一个进程自身的运行时间：

```
clock_t start, finish;  
duration = (double)(finish - start) / CLOCKS_PER_SEC;  
毫秒级  
difftime()函数，但它只能精确到秒  
time_t start,end;  
start = time(NUL);  
end = time(NUL);  
difftime(end,start)  
time_t lt;  
lt = time(NUL); //或者直接是 lt = time(&lt) time 返回当前日历时间
```

## 线性表

将具有“一对一”关系的数据“线性”地存储到物理空间中，这种存储结构就称为线性存储结构（简称线性表）。

线性表存储结构可细分为顺序存储结构和链式存储结构。

某一元素的左侧相邻元素称为“直接前驱”，位于此元素左侧的所有元素都统称为“前驱元素”；

某一元素的右侧相邻元素称为“直接后继”，位于此元素右侧的所有元素都统称为“后继元素”；

```

typedef struct Table {
    int * head;
    int length;
    int size;
} table;
table initTable() {
    table t;
    t.head = (int*)malloc(Size * sizeof(int));
    if (!t.head)
    {
        printf("初始化失败");
        exit(0);
    }
    t.length = 0;
    t.size = Size;
    return t;
}

```

其实就是一个嵌在结构体里的动态数组，用最后一个元素标记时初始化为-1；

```

table addTable(table t, int elem, int add)
{
    int i;
    if (add > t.length + 1 || add < 1) {
        printf("插入位置有问题");
        return t;
    }
    if (t.length >= t.size) {
        t.head = (int*)realloc(t.head, (t.size + 1) * sizeof(int));
        if (!t.head) {
            printf("存储分配失败");
        }
        t.size += 1;
    }
    for (i = t.length - 1; i >= add - 1; i--) {
        t.head[i + 1] = t.head[i];
    }
    t.head[add - 1] = elem;
    t.length++;
    return t;
}

```

```

.... if(L->Last==MAXSIZE-1)
.... {
....     printf("FULL");
....     return false;
.... }
.... if(P<0||P>L->Last+1)//注意范围，可以在尾部插入
.... {
....     printf("ILLEGAL POSITION");
....     return false;
.... }

```

指针名= (数据类型\*) realloc (要改变内存大小的指针名，新的大小)。

新的大小可大可小 (如果新的大小大于原内存大小，则新分配部分不会被初始化；如果新的大小小于原内存大小，可能会导致数据丢失 [1-2] )

```

table delTable(table t, int add) {
    int i;
    if (add > t.length || add < 1) {
        printf("被删除元素的位置有误");
        exit(0);
    }
    for (i = add; i < t.length; i++) {
        t.head[i - 1] = t.head[i];
    }
    t.length--;
    return t;
}

int selectTable(table t, int elem) {
    int i;
    for (i = 0; i < t.length; i++) {
        if (t.head[i] == elem) {
            return i + 1;
        }
    }
    return -1;
}

table amendTable(table t, int elem, int newElem) {
    int add = selectTable(t, elem);
    t.head[add - 1] = newElem;
    return t;
}

```

## 从数组到链表

链表是一种常见的基础数据结构，结构体指针在这里得到了充分的利用。链表可以动态的进行存储分配，也就是说，链表是一个功能极为强大的数组，他可以在节点中定义多种数据类型，还可以根据需要随意增添，删除，插入节点。链表都有一个头指针，一般以 head 来表示，存放的是一个地址。链表中的节点分为两类，头结点和一般节点，头结点是没有数据域的。链表中每个节点都分为两部分，一个数据域，一个是指针域。说到这里你应该就明白了，链表就如同车链子一样，head 指向第一个元素：第一个元素又指向第二个元素；……，直到最后一个元素，该元素不再指向其它元素，它称为“表尾”，它的地址部分放一个“NULL”（表示“空地址”），链表到此结束。

作为有强大功能的链表，对他的操作当然有许多，比如：链表的创建，修改，删除，插入输出，排序，反序，清空链表的元素，求链表的长度等等。

```
struct film
{
    char title[TSIZE];
    int rating;
    struct film *next; /* 指向链表中的下一个结构 */
};

struct film *head = NULL;
struct film *prev, *current;
char input[TSIZE];
/* 收集并储存信息 */ puts("Enter first movie title:");
while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
{
    current = (struct film *)malloc(sizeof(struct film));
    if (head == NULL) /* 第1个结构 */
        head = current;
    else /* 后续的结构 */
        prev->next = current;
    current->next = NULL;
    strcpy(current->title, input);
    puts("Enter your rating <0-10>:");
    scanf("%d", &current->rating);
    while (getchar() != '\n')
        continue;
    puts("Enter next movie title (empty line to stop):");
    prev = current;
} /* 显示电影列表 */
```

```
if (head == NULL)
|    printf("No data entered. ");
else
|    printf("Here is the movie list:\n");
current = head;
while (current != NULL)
{
|    printf("Movie: %s Rating: %d\n", current->title, current->rating);
|    current = current->next;
```

```
} /* 完成任务，释放已分配的内存 */
current = head;
while (current != NULL)
{
|    current = head;
|    head = current->next;
|    free(current);
}
```

这里将 head 当作一个中间变量从而释放 current 并通过 current 找到下一个指针

**头指针：**一个普通的指针，它的特点是永远指向链表第一个节点的位置。很明显，头指针用于指明链表的位置，便于后期找到链表并使用表中的数据；

**节点：**链表中的节点又细分为**头节点、首元节点和其他节点**：

**头节点：**其实就是一个不存任何数据的空节点，通常作为链表的第一个节点。对于链表来说，**头节点不是必须的**，它的作用只是为了方便解决某些实际问题；

**首元节点：**由于头节点（也就是空节点）的缘故，链表中称第一个存有数据的节点为首元节点。首元节点只是对链表中第一个存有数据节点的一个称谓，没有实际意义；

**其他节点：**链表中其他的节点；

---

如果想创建一个存储 {1,2,3,4} 且含头节点的链表，则 C 语言实现代码为：

```
01. link * initLink()
02. {
03.     int i;
04.     link * p=(link*)malloc(sizeof(link));//创建一个头结点
05.     link * temp=p;//声明一个指针指向头结点,
06.     //生成链表
07.     for (i=1; i<5; i++) {
08.         link *a=(link*)malloc(sizeof(link));
09.         a->elem=i;
10.         a->next=NULL;
11.         temp->next=a;
12.         temp=temp->next;
13.     }
14. }
```

p 指向的节点没有数据，操作时只需 next 即可，没有头节点的话部分操作对首元节点要分类

```
01. void display(link *p) {
02.     link* temp = p;//将temp指针重新指向头结点
03.     //只要temp指针指向的结点的next不是Null，就执行输出语句。
04.     while (temp) {
05.         printf("%d ", temp->elem);
06.         temp = temp->next;
07.     }
08.     printf("\n");
09. }
```

注意，如果使用带有头节点创建链表的方式，则输出链表的 display 函数需要做适当地修改：

```
01. void display(link *p) {
02.     link* temp=p;//将temp指针重新指向头结点
03.     //只要temp指针指向的结点的next不是Null，就执行输出语句。
04.     while (temp->next) {
05.         temp=temp->next;
06.         printf("%d",temp->elem);
07.     }
08.     printf("\n");
09. }
```

## 链表操作

“`typedef struct node{} Node;`” 定义，后续定义结构体变量就能采用“`Node node1`”这种形式了，即 `Node` 等价为 `struct node`，这样定义变量十分简洁。

同顺序表一样，向链表中增添元素，根据添加位置不同，可分为以下 3 种情况：

插入到链表的头部（头节点之后），作为首元节点；

插入到链表中间的某个位置；

插入到链表的最末端，作为链表中最后一个数据元素；

虽然新元素的插入位置不固定，但是链表插入元素的思想是固定的，只需做以下两步操作即可将新元素插入到指定的位置：

将新结点的 `next` 指针指向插入位置后的结点；

将插入位置前结点的 `next` 指针指向插入结点；

如果结构体中含有指针，是很容易出问题的，从上面的结果中（高亮）部分可以看到字符串的输出结果是不一样的，这说明，在进行写入文件的时候，`char*` 所指向的字符串没有写入文件，只是将指针写入，当从文件中读出结构体，再次得到这个指针的时候，由于程序运行的内存位置变化，所以原来指针所指向的内容也变了，所以输出也变了

```

//首先找到要插入位置的上一个结点
for (i = 1; i < add; i++) {
    if (temp == NULL) {
        printf("插入位置无效\n");
        return p;
    }
    temp = temp->next;
}
//创建插入结点c
c = (link*)malloc(sizeof(link));
c->elem = elem;
//向链表中插入结点
c->next = temp->next;
temp->next = c;
return p;

```

从链表中删除指定数据元素时，实则就是将存有该数据元素的节点从链表中摘除，但作为一名合格的程序员，要对存储空间负责，对不再利用的存储空间要及时释放。因此，从链表中删除数据元素需要进行以下 2 步操作：

将结点从链表中摘下来；

手动释放掉结点，回收被结点占用的存储空间；

其中，从链表上摘除某节点的实现非常简单，只需找到该节点的直接前驱节点 temp，执行一程序：（**没有头节点的可以新建并初始化一个头节点**）

```

temp->next=temp->next->next;
06.     //temp指向被删除结点的上一个结点
07.     for (i = 1; i < add; i++) {
08.         temp = temp->next;
09.     }
10.     del = temp->next;//单独设置一个指针指向被删除结点，以防丢失
11.     temp->next = temp->next->next;//删除某个结点的方法就是更改前一个结点的指针域
12.     free(del);//手动释放该结点，防止内存泄漏
13.     return p;
...
....struct ListNode* head=(struct ListNode*)malloc(sizeof(struct ListNode)),*nhead,*temp;
....head->next=L;nhead=head;
....while(head->next)
....{
....    if(head->next->data==m)
....    {
....        temp=head->next;//记录下一个节点
....        head->next=head->next->next;
....        free(temp);
....    }
....    else
....        head=head->next;
....}
....return nhead->next;

```

在链表中查找指定数据元素，最常用的方法是：从表头依次遍历表中节点，用被查找元素与各节点数据域中存储的数据元素进行比对，直至比对成功或遍历至链表最末端的

NULL (比对失败的标志)。

```
06.     //由于头节点的存在，因此while中的判断为t->next
07.     while (t->next) {
08.         t = t->next;
09.         if (t->elem == elem) {
10.             return i;
11.         }
12.         i++;
13.     }
14.     //程序执行至此处，表示查找失败
15.     return -1;
16. }
```

注意，遍历有头节点的链表时，需避免头节点对测试数据的影响，因此在遍历链表时，建立使用上面代码中的遍历方法，直接越过头节点对链表进行有效遍历。

(1)对带头结点的链表，在表的任何结点之前插入结点或删除表中任何结点，所要做的都是修改前一结点的指针域，因为任何元素结点都有前驱结点。若链表没有头结点，则首元素结点没有前驱结点，在其前插入结点或删除该结点时操作会复杂些。

(2)对带头结点的链表，表头指针是指向头结点的非空指针，因此空表与非空表的处理是一样的。

## 单链表反转

迭代反转

```
ListNode* reverseList(ListNode* head) {
    ListNode* temp; // 保存cur的下一个节点
    ListNode* cur = head;
    ListNode* pre = NULL;
    while(cur) {
        temp = cur->next; // 保存一下 cur的下一个节点，因为接下来要改变cur->next
        cur->next = pre; // 翻转操作
        // 更新pre 和 cur指针
        pre = cur;
        cur = temp;
    }
    return pre;
}
```

```
link *reverseElem(link *p) //迭代反转
{
    if (p->next == NULL || p->next->next == NULL) //如果没有头节点则对应的两项为p和p->next.
        return p;
    link *pre = NULL; //逆转后的结束标志
    link *cur = p->next;
    link *temp ;
    while (1)
    {
        temp = cur->next;//暂存下个节点的位置
        cur->next = pre; //将下个节点改为前一个
        if (!temp)
        {
            temp = (link *)malloc(sizeof(link)); /*为最后的temp分配空间，因为它此时没有得到属于原链表的空间
            temp->next = cur;
            return temp;
        }
        pre=cur; /*右移*/
        cur=temp;
    }
}
```

直接移动 Head 也行

头插法与就地逆置和迭代反转没区别

```
link *reverseElem2(link *p)
{
    if (p->next == NULL || p->next->next == NULL)
        return p;
    p = p->next;
    link *beg = p;
    link *end = p->next;
    while (1)
    {
        //将 end 从链表中摘除
        beg->next = end->next;
        //将 end 移动至链表头
        end->next = p;
        p = end;
        //调整 end 的指向，另其指向 beg 后的一个节点，为反转下一个节点做准备
        end = beg->next;
        if (end == NULL)
        {
            end = (link *)malloc(sizeof(link));
            end->next = p;
            return end;
        }
    }
}
```

递归反转

```
link *reverseElem1(link *p)
{
    if (p->next == NULL || p == NULL)//递归进入的终点和节点过少无需逆转的情况
    {
        link *newhead = (link *)malloc(sizeof(link));//新建头节点
        newhead->next = p;
        return newhead;
    }
    else
    {
        link *reverse = reverseElem1(p->next);//逐层进入
        /*逐层退出*/
        p->next->next = p;//逆转下一个节点的指向
        p->next = NULL;//逆转后最后一个节点的指向
        return reverse;//*返回值一直是newhead
    }
}
```

这个将跳过头节点放在了函数外，实参是 head->next

```
link *reverseElem1(link *p)
{
    p= p->next;
    if (p->next == NULL || p == NULL)//递归进入的终点和节点过少无需逆转的情况
    {
        link *newhead = (link *)malloc(sizeof(link));//新建头节点
        newhead->next = p;
        return newhead;
    }
    else
    {
        link *reverse = reverseElem1(p);//逐层进入
        /*逐层退出*/
        p->next->next = p;//逆转下一个节点的指向
        p->next = NULL;//逆转后最后一个节点的指向
        return reverse;//*返回值一直是newhead
    }
}
```

小结：

如果没有头节点则不需最后的初始化新头节点和最初的越过头节点

## 判断单链表相交

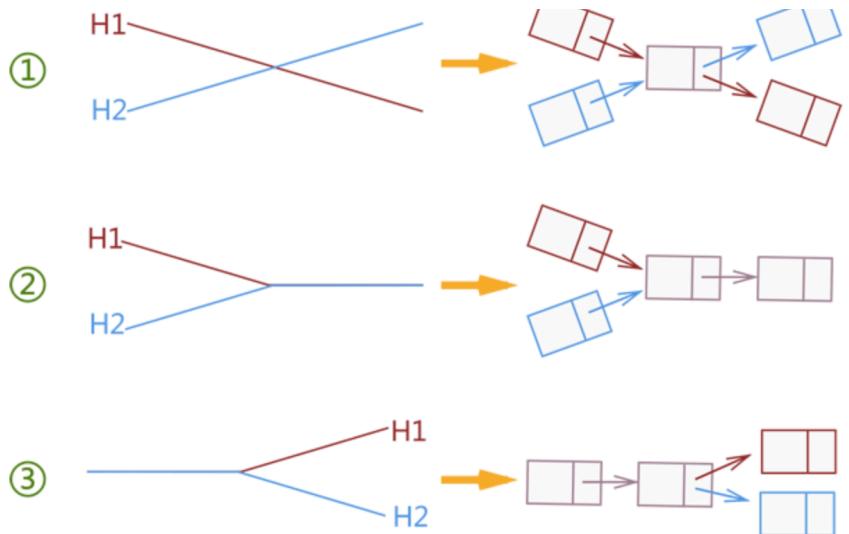


图 1 链表相交

注意，结合“单链表中每个节点有且仅有 1 个指针域”的特性，如 1 所示的这 3 种情况下，只有第 2 种情况符合单链表的特性，另外 2 种情况则破坏了此特性。经过以上的分析，本节要验证 2 个单链表是否相交，实际上等同于判断 2 个单链表是否和图 1② 所示的存储结构相同。

### 第一个相交节点之后的部分也相交

```
01. //自定义的 bool 类型
02. typedef enum bool
03. {
04.     False = 0,
05.     True = 1
06. }bool;
07. //L1 和 L2 为 2 个单链表，函数返回 True 表示链表相交，返回 False 表示不相交
08. bool LinkIntersect(link * L1, link * L2) {
09.     link * p1 = L1;
10.     link * p2 = L2;
11.     //逐个遍历 L1 链表中的各个节点
12.     while (p1)
13.     {
14.         //遍历 L2 链表，针对每个 p1，依次和 p2 所指节点做比较
15.         while (p2) {
16.             //p1、p2 中记录的就是各个节点的存储地址，直接比较即可
17.             if (p1 == p2) {
18.                 return True;
19.             }
20.             p2 = p2->next;
21.         }
22.         p1 = p1->next;
23.     }
24.     return False;
25. }
```

纯文本

return p 就可以将位置传出去

```
01. //L1 和 L2 为 2 个单链表，函数返回 True 表示链表相交，返回 False 表示不相交
02. bool LinkIntersect(link * L1, link * L2) {
03.     link * p1 = L1;
04.     link * p2 = L2;
05.     //找到 L1 链表中的最后一个节点
06.     while (p1->next) {
07.         p1 = p1->next;
08.     }
09.     //找到 L2 链表中的最后一个节点
10.     while (p2->next)
11.     {
12.         p2 = p2->next;
13.     }
14.     //判断 L1 和 L2 链表最后一个节点是否相同
15.     if (p1 == p2) {
16.         return True;
17.     }
18.     return False;
19. }
```

纯文本

时间复杂度小一些，但是无法得到开始相交的节点

## 静态链表

数据域：用于存储数据元素的值；

游标：其实就是数组下标，表示直接后继元素所在数组中的位置；

```
typedef struct {
```

```
    int data;//数据域
```

```
    int cur;//游标
```

```
}component;
```

备用链表的作用是回收数组中未使用或之前使用过（目前未使用）的存储空间，留待后期使用。也就是说，静态链表使用数组申请的物理空间中，存有两个链表，一条连接数据，另一条连接数组中未使用的空间。

通常，备用链表的表头位于数组下标为 0 (a[0]) 的位置，而数据链表的表头位于数组下标为 1 (a[1]) 的位置。

静态链表中设置备用链表的好处是，可以清楚地知道数组中是否有空闲位置，以便数据链表添加新数据时使用。比如，若静态链表中数组下标为 0 的位置上存有数据，则证明数组已满。

在数据链表未初始化之前，数组中所有位置都处于空闲状态，因此都应被链接在备用链表上

当向静态链表中添加数据时，需提前从备用链表中摘除节点，以供新数据使用。

备用链表摘除节点最简单的方法是摘除 a[0] 的直接后继节点；同样，向备用链表中添加空闲节点也是添加作为 a[0] 新的直接后继节点。因为 a[0] 是备用链表的第一个节点，我们知道它的位置，操作它的直接后继节点相对容易，无需遍历备用链表，耗费的时间复杂度为

O(1)。

从备用链表中摘除一个节点，用于存储元素 4；

找到表中第 2 个节点（添加位置的前一个节点，这里是数据元素 2），将元素 2 的游标赋值给新元素 4；

将元素 4 所在数组中的下标赋值给元素 2 的游标；

静态链表中删除指定元素，只需实现以下 2 步操作：

将存有目标元素的节点从数据链表中摘除；

将摘除节点添加到备用链表，以便下次再用；

```
void insert(component *array, int n, int body) //添加元素
{
    int j = mallocArr(array); //申请一个闲置节点
    //找到要插入位置的上一个结点在数组中的位置
    int temp = body;
    for (int i = 1; i < n - 1; i++)
    {
        temp = array[temp].cur;
    }
    array[j].cur = array[temp].cur; //将前置节点的游标赋格给新节点
    array[temp].cur = j;           //将新节点的游标值赋给前置节点
    array[j].data = 666;
}

void delete (component *array, int n, int body)
{
    //找到要删除位置的上一个结点在数组中的位置
    int temp = body;
    for (int i = 1; i < n-1; i++)
    {
        temp = array[temp].cur;
    }
    array[temp].cur = array[array[temp].cur].cur;      //
    freeArr(array, array[temp].cur);
}

//备用链表回收空间的函数，其中array为存储数据的数组，k表示未使用节点所在数组的下标
void freeArr(component *array, int k)
{
    array[k].cur = array[0].cur;
    array[0].cur = k;
}
```

## 循环链表

和它名字的表意一样，只需要将表中最后一个结点的指针指向头结点，链表就能成环儿



图1 循环链表

\*--+

判断是否有环

定义两个指针 tortoise 与 rabbit，让它们一开始均指向 head 头节点，之后，tortoise 每次向后移动一个节点，rabbit 每次向后移动 2 个节点，只要这个链表是有环的，它们必定会在某一次移动完后相遇，什么？你问我为什么？我们来思考这样一个问题，两个人在运

动场跑步，他们的起始位置都是一样的，当开跑后，只有在两种情况下，他们的位置会重合，第一就是他们的速度始终一致，第二就是跑得快的那个人套圈

我们假设两位跑步的同学速度始终不变，即 tortoise 以  $V$  的速度进行移动，rabbit 以  $2V$  的速度进行移动，在经过了相同的时间  $T$  后，他们相遇了，此时 tortoise 移动的距离为  $VT$ ，而 rabbit 移动的距离为  $2VT$ ，他们移动的距离差  $VT$ ，即为这个链表中“环”的周长，如上图所示，节点 A 表示为环入口，节点 B 表示他们第一次相遇，我们将 head 头节点至节点 A 的距离记为  $a$ ，将节点 A 至他们第一次相遇的节点 B 的距离记为  $b$ ，通过我们刚才的分析，不难得出，tortoise 移动的距离  $VT = a + b$ ，等量代换，他们移动的距离差也为  $a + b$ ，所以链表中环的周长为  $a + b$ ，现在已知节点 A 至节点 B 的距离为  $b$ ，那么节点 B 至节点 A 的距离便为  $a$ ，至此，发现什么了？head 头节点到节点 A 的距离同样为  $a$ ，我们建立一个指针 start 指向头节点，它与 B 节点处的 tortoise 同时以一个节点的速度进行移动，一段时间后，它们将在环入口相遇。我们不光能证明一个链表是否有环，还能找到环的入口。

## 双向链表

双向链表，简称双链表。从名字上理解双向链表，即链表是“双向”的，如图 1 所示：

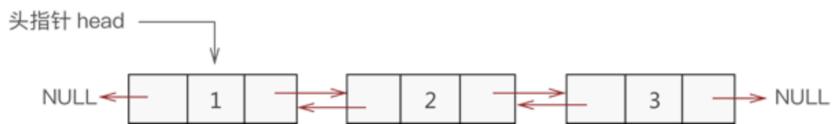


图 1 双向链表结构示意图

所谓双向，指的是各节点之间的逻辑关系是双向的，但通常头指针只设置一个，除非实际情况需要，可以为最后一个节点再设置一个“头指针”。

根据图 1 不难看出，双向链表中各节点包含以下 3 部分信息（如图 2 所示）：

1. 指针域：用于指向当前节点的直接前驱节点；
2. 数据域：用于存储数据元素；
3. 指针域：用于指向当前节点的直接后继节点。



图 2 双向链表的节点构成

初始化时只是多了个 `cur->prev=pre;` 其它操作注意 `pre` 指针域的改变即可



## 栈和队列

栈是一种只能从表的一端存取数据且遵循 "先进后出" 原则的线性存储结构。

通常，栈的开口端被称为栈顶；相应地，封口端被称为栈底。因此，栈顶元素指的就是距离栈顶最近的元素

基于栈结构的特点，在实际应用中，通常只会对栈执行以下两种操作：

向栈中添加元素，此过程被称为"进栈"（入栈或压栈）； push

从栈中提取出指定元素，此过程被称为"出栈"（或弹栈）； pop

栈是一种 "特殊" 的线性存储结构，因此栈的具体实现有以下两种方式：

顺序栈：采用顺序存储结构可以模拟栈存储数据的特点，从而实现栈存储结构；

链栈：采用链式存储结构实现栈结构；

即在顺序表中设定一个实时指向栈顶元素的变量（一般命名为 top），**top 初始值为 -1**，表示栈中没有存储任何数据元素，及栈是"空栈"。一旦有数据元素进栈，则 top 就做 +1 操作；反之，如果数据元素出栈，top 就做 -1 操作。

```
01. //元素elem进栈，a为数组，top值为当前栈的栈顶位置
02. int push(int* a, int top, int elem) {
03.     a[++top]=elem;
04.     return top;
05. }
```

```
01. //数据元素出栈
02. int pop(int * a, int top) {
03.     if (top== -1) {
04.         printf("空栈");
05.         return -1;
06.     }
07.     printf("弹栈元素: %d\n", a[top]);
08.     top--;
09.     return top;
10. }
```

链表的头部作为栈顶，意味着：

- 在实现数据“入栈”操作时，需要将数据从链表的头部插入；
- 在实现数据“出栈”操作时，需要删除链表头部的首元节点；

因此，链栈实际上就是一个只能采用头插法插入或删除数据的链表。

## 链栈元素入栈

例如，将元素 1、2、3、4 依次入栈，等价于将各元素采用头插法依次添加到链表中，每个数据元素的添加过程如图 2 所示：

- 刚开始： head  $\rightarrow$  NULL
- 添加元素 1： head  $\rightarrow$  1  $\rightarrow$  NULL
- 添加元素 2： head  $\rightarrow$  2  $\rightarrow$  1  $\rightarrow$  NULL
- 添加元素 3： head  $\rightarrow$  3  $\rightarrow$  2  $\rightarrow$  1  $\rightarrow$  NULL
- 添加元素 4： head  $\rightarrow$  4  $\rightarrow$  3  $\rightarrow$  2  $\rightarrow$  1  $\rightarrow$  NULL

图 2 链栈元素依次入栈过程示意图

## 进制转换

```
n2 = func1(num, (int)strlen(num), base1);
while (n2 / base2)
{
    push(n2 % base2);
    n2 /= base2;
}
push(n2);
printf("the result is:\n");
while (top != -1)
{
    pop();
    printf("\n");
```

## 括号匹配

```
for (i=0; i<length; i++) {
    //如果是左括号，直接压栈
    if (bracket[i]=='(' || bracket[i]=='{') {
        push(a, bracket[i]);
    } else {
        //如果是右边括号，判断与栈顶元素是否匹配，如果匹配，栈顶元素弹栈，程序继续运行；否则，发现括号不匹配，输出结果直接退出
        if (bracket[i]==')') {
            if (visit(a)=='(') {
                pop(a);
            } else {
                printf("括号不匹配");
                return 0;
```

## 求运算式

以  $3!$  为例， $!$  为运算符， $3$  为运算项，因此  $3!$  本身就是一个后缀表达式；再以  $4^2$  为例， $*$  为运算符， $4$  和  $2$  作为它的运算项，其对应的后缀表达式为  $4\ 2\ +$ 。

在此基础上，我们试着将  $3!+4^2/(1-5)^2$  转换成分后缀表达式，其过程也就是将表达式中所有运算符放置在它的运算项之后：

- $!$  运算符对应的运算项为  $3$ ，转换后得到  $3\ !\ ;$
- $+$  运算符对应的运算项是  $3!$  和  $4^2/(1-5)^2$ ，转换之后得到：  $3!\ 4^2/(1-5)^2\ +\ ;$
- $*$  运算符对应的运算项是  $4$  和  $2$ ，转换之后得到  $4\ 2\ *$ ；
- $/$  运算符对应的运算项是  $4\ 2\ *$  和  $(1-5)^2$ ，转换后得到  $4\ 2\ *\ (1-5)^2\ /\ ;$
- $-$  运算符对应的运算项是  $1$  和  $5$ ，转换后得到  $1\ 5\ -\ ;$
- $^$  运算符对应的运算项是  $1\ 5\ -$  和  $2$ ，转换后得到  $1\ 5\ -\ 2\ ^\ ;$

整合之后，整个普通表达式就转换成了  $3!\ 4\ 2\ *\ 1\ 5\ -\ 2\ ^\ /+\ ;$ ，这就是其对应的后缀表达式。

不难发现，后缀表达式完全舍弃了表达式本该有的可读性，但有失必有得，相比普通表达式，后缀表达式的值可以轻松借助栈存储结构求得。具体求值的过程是：当用户给定一个后缀表达式时，按照从左到右的顺序依次扫描表达式中的各个运算项和运算符，对它们进行如下处理：

1. 遇到运算项时，直接入栈；
2. 遇到运算符时，将位于栈顶的运算项出栈，对于  $!$  运算符，取栈顶 1 个运算项；其它运算符，取栈顶 2 个运算项，第一个取出的运算项作为该运算符的右运算项，另一个作为左运算项。求此表达式的值并将其入栈。

经过以上操作，直到栈中仅存在一个运算项为止，此运算项即为整个表达式的值。

```
typedef struct
{
    double data[MAXSIZE];
    int top;
} Stack_num;

void InitStack_num(Stack_num **s)
{
    *s = (Stack_num *)malloc(sizeof(Stack_num));
    (*s)->top = -1;
}
```

### 通过结构体方式定义和初始化

```
bool Pop_num(Stack_num **s, double *e)
{
    if ((*s)->top == -1)
        return false;
    *e = (*s)->data[(*s)->top];
    (*s)->top--;
    return true;
}
```

### 通过传地址直接修改值

在一个数组中实现两个堆栈

```

Stack* CreateStack(int MaxSize)
{
    Stack* S=(Stack)malloc(sizeof(struct SNode));
    S->Data=(int*)malloc(sizeof(int)*MaxSize);
    S->Top1=-1;
    S->Top2=MaxSize;
    S->MaxSize=MaxSize;
    return S;
}

```

满标志：if(S->Top2-S->Top1==1)

空标志

(S->Top1==-1&&Tag==1)|(S->Top2==S->MaxSize&&Tag==2)

## 顺序队列

由于顺序队列的底层使用的是数组，因此需预先申请一块足够大的内存空间初始化顺序队列。除此之外，为了满足顺序队列中数据从队尾进，队头出且先进先出的要求，我们还需要定义两个指针（top 和 rear）分别用于指向顺序队列中的队头元素和队尾元素

由于顺序队列初始状态没有存储任何元素，因此 top 指针和 rear 指针重合，且由于顺序队列底层实现靠的是数组，因此 top 和 rear 实际上是两个变量，它的值分别是队头元素和队尾元素所在数组位置的下标。

在图 1 的基础上，当有数据元素进队列时，对应的实现操作是将其存储在指针 rear 指向的数组位置，然后 rear+1；当需要队头元素出队时，仅需做 top+1 操作。

int front,rear;

//设置队头指针和队尾指针，**当队列中没有元素时，队头和队尾指向同一块地址**

front=rear=0;

顺序队列整体后移造成的影响是：

顺序队列之前的数组存储空间将无法再被使用，造成了空间浪费；

如果顺序表申请的空间不足够大，则直接造成程序中数组 a 溢出，产生溢出错误；

**改进，使用环状结构**

```

int enqueue(int *a, int front, int rear, int data) {
    //添加判断语句，如果rear超过max，则直接将其从a[0]重新开始存储，如果rear+1和front重合，则表示数组已满
    if ((rear+1)%max==front) {
        printf("空间已满");
        return rear;
    }
    a[(rear%max)]=data;
    rear++;
    return rear;
}

```

```

int deQueue(int *a, int front, int rear) {
    //如果front==rear, 表示队列为空
    if(front==rear%max) {
        printf("队列为空");
        return front;
    }
    printf("%d ", a[front]);
    //front不再直接 +1, 而是+1后同max进行比较, 如果=max, 则直接跳转到 a[0]
    front=(front+1)%max;
    return front;
}

```

### 链式队列

进队列：尾部插入

出队列：头部删除（最好使用带头节点的）

细节不表

## KMP

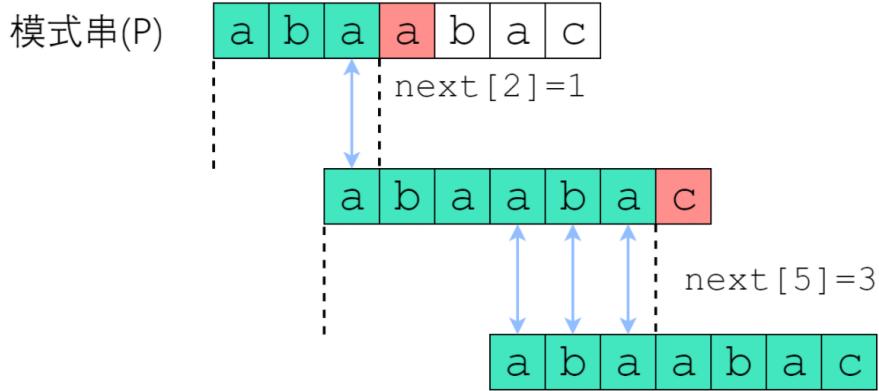
KMP 算法的核心，是一个被称为部分匹配表(Partial Match Table)的数组

如果字符串 A 和 B，存在 A=BS，其中 S 是任意的非空字符串，那就称 B 为 A 的前缀。例如，“Harry”的前缀包括{“H”，“Ha”，“Har”，“Harr”}，我们把所有前缀组成的集合，称为字符串的前缀集合。同样可以定义后缀 A=SB，其中 S 是任意的非空字符串那就称 B 为 A 的后缀，例如，“Potter”的后缀包括{“otter”，“tter”，“ter”，“er”，“r”}，然后把所有后缀组成的集合，称为字符串的后缀集合。要注意的是，字符串本身并不是自己的后缀/前缀。有了这个定义，就可以说明 PMT 中的值的意义了。PMT 中的值是字符串的前缀集合与后缀集合的交集中最长元素的长度。例如，对于“aba”，它的前缀集合为{“a”，“ab”}，后缀集合为{“ba”，“a”}。两个集合的交集为{“a”}，那么长度最长的元素就是字符串“a”了，长度为 1，所以对于“aba”而言，它在 PMT 表中对应的值就是 1。再比如，对于字符串“ababa”，它的前缀集合为{“a”，“ab”，“aba”，“abab”}，它的后缀集合为{“baba”，“aba”，“ba”，“a”}，两个集合的交集为{“a”，“aba”}，其中最长的元素为“aba”，长度为 3。

我们看到如果是在 j 位失配，那么影响 j 指针回溯的位置的其实是第 j-1 位的 PMT 值，所以为了编程的方便，我们不直接使用 PMT 数组，而是将 PMT 数组向后偏移一位。我们把新得到的这个数组称为 next 数组。下面给出根据 next 数组进行字符串匹配加速的字符串匹配程序。其中要注意的一个技巧是，在把 PMT 进行向右偏移时，第 0 位的值，我们将其设成了 -1，这只是为了编程的方便，并没有其他的意义。

next 数组是对于模式串而言的。P 的 next 数组定义为：next[i] 表示 P[0] ~ P[i-1] 这一个子串，使得前 k 个字符恰等于后 k 个字符的最大的 k。特别地，k 不能取 i+1（因为这个子串一共才 i+1 个字符，自己肯定与自己相等，就没有意义了）。图中的 next 是 PMT

主串(S) a b a b a a b a a b a c ... ...



在  $S[0]$  尝试匹配，失配于  $S[3] \leftrightarrow P[3]$  之后，我们直接把模式串往右移了两位，让  $S[3]$  对准  $P[1]$ 。接着继续匹配，失配于  $S[8] \leftrightarrow P[6]$ ，接下来我们把  $P$  往右平移了三位，把  $S[8]$  对准  $P[3]$ 。此后继续匹配直到成功。

我们应该如何移动这把标尺？很明显，如图中蓝色箭头所示，旧的后缀要与新的前缀一致（如果不一致，那就肯定没法匹配上了）！

回忆  $\text{next}$  数组的性质： $P[0]$  到  $P[i]$  这一段子串中，前  $\text{next}[i]$  个字符与后  $\text{next}[i]$  个字符一模一样。既然如此，如果失配在  $P[r]$ ，那么  $P[0] \sim P[r-1]$  这一段里面，前  $\text{next}[r]$  个字符恰好和后  $\text{next}[r]$  个字符相等——也就是说，我们可以拿长度为  $\text{next}[r]$  的那一段前缀，来顶替当前后缀的位置，让匹配继续下去！您可以验证一下上面的匹配例子： $P[3]$  失配后，把  $P[PMT[3-1]]$  也就是  $P[1]$  对准了主串刚刚失配的那一位； $P[6]$  失配后，把  $P[PMT[6-1]]$  也就是  $P[3]$  对准了主串刚刚失配的那一位。

$i-1$  是因为计算的时某一位之前（0 到  $i-1$  位子串）的最大相等前后缀，从一个最大前缀开始匹配失败后就要从与它相等的后缀开始（中间部分确定对不上头，对上也没前缀对的长，而后缀确定能对上（且是已知能对上最多的）），并将这个后缀（ $\text{next}(j)$ ）位作为新的前缀，直接从失配处（ $P[\text{next}(j)]$ ）继续匹配。

失配时，模式串向右移动的位数为：已匹配字符数 - 失配字符的上一位字符所对应的最大长度值，等于已配减去新前缀（老后缀）

结构不够对称时  $\text{next}$  的值就会小，跳过的部分就少（右移位数就大），极限情况下就会接近普通算法

主串影响失配的位置和多少，模式串影响回溯的步数（普通算法就是回溯到串首）

#### next 计算

求  $\text{next}$  数组的过程完全可以看成字符串匹配的过程，即以模式字符串为主字符串，以模式字符串的前缀为目标字符串，一旦字符串匹配成功，那么当前的  $\text{next}$  值就是匹配成功的字符串的长度。具体来说，就是从模式字符串的第一位（注意，不包括第 0 位）开始对自身进行匹配运算。在任一位置，能匹配的最长长度就是当前位置的  $\text{next}$  值。

```
void GetNext(char* p,int next[])
{
    int pLen = strlen(p);
    next[0] = -1; //用去判断时不会用到
```

```

int k = -1;
int j = 0;
while (j < pLen - 1)
{
    //p[k]表示前缀, p[j]表示后缀
    if (k == -1 || p[j] == p[k]) //相等或者初始状态下最大前后缀才会增长
    {
        ++k;
        ++j;
        next[j] = k; //Next[1]=0;
    }
    else //否则 k 不断减小直到重新匹配上 p[j], 或者匹配不上退到-1使 next[j]=0;
    {
        k = next[k];
    }
}
}

```

若能在前缀 “ $p_0 p_{k-1} p_k$ ” 中不断的递归前缀索引  $k = \text{next}[k]$ ，找到一个字符  $p_k'$  也为 D，代表  $p_k' = p_j$ ，且满足  $p_0 p_{k'-1} p_{k'} = p_j p_{k'-1} p_j$ ，则最大相同的前缀后缀长度为  $k' + 1$ ，从而  $\text{next}[j+1] = k' + 1 = \text{next}[k'] + 1$

那为何递归前缀索引  $k = \text{next}[k]$ ，就能找到长度更短的相同前缀后缀呢？这又归根到  $\text{next}$  数组的含义。我们拿前缀  $p_0 p_{k-1} p_k$  去跟后缀  $p_j p_{k-1} p_j$  匹配，如果  $p_k$  跟  $p_j$  失配，下一步就是用  $p[\text{next}[k]]$  去跟  $p_j$  继续匹配，如果  $p[\text{next}[k]]$  跟  $p_j$  还是不匹配，则需要寻找长度更短的相同前缀后缀，即下一步用  $p[\text{next}[\text{next}[k]]]$  去跟  $p_j$  匹配。此过程相当于模式串的自我匹配，所以不断的递归  $k = \text{next}[k]$ ，直到要么找到长度更短的相同前缀后缀，要么没有长度更短的相同前缀后缀。

判断次长的前缀字符串 "aa" 和后缀字符串 "ac" 是否相等，这一步的实现可以用  $j = \text{next}[j]$  来实现

```

Next(T,next); //根据模式串 T, 初始化 next 数组
int i=1;
int j=1;
while (i<=strlen(S)&&j<=strlen(T)) {
    //j==0: 代表模式串的第一个字符就和当前测试的字符不相等; S[i-1]==T[j-1], 如果
    对应位置字符相等, 两种情况下, 指向当前测试的两个指针下标 i 和 j 都向后移
    if (j==0 || S[i-1]==T[j-1]) {
        i++;
        j++;
    }
    else{
        j=next[j]; //如果测试的两个字符不相等, i 不动, j 变为当前测试字符串的 next
    值
    }
}
if (j>strlen(T)) { //如果条件为真, 说明匹配成功

```

# 稀疏矩阵存储

结合数据结构压缩存储的思想，我们可以使用一维数组存储对称矩阵。由于矩阵中沿对角线两侧的数据相等，因此数组中只需存储对角线一侧（包含对角线）的数据即可。

对称矩阵的实现过程是，若存储下三角中的元素，只需将各元素所在的行标  $i$  和列标  $j$  代入下面的公式：

$$k = \frac{i \times (i - 1)}{2} + j - 1$$

存储上三角的元素要将各元素的行标  $i$  和列标  $j$  代入另一个公式：

$$k = \frac{j \times (j - 1)}{2} + i - 1$$

最终求得的  $k$  值即为该元素存储到数组中的位置（矩阵中元素的行标和列标都从 1 开始）。

例如，在数组  $skr[6]$  中存储图 1 中的对称矩阵，则矩阵的压缩存储状态如图 3 所示（存储上三角和下三角的结果相同）：

skr	1	2	4	3	5	6
	0	1	2	3	4	5

图 3 对称矩阵的压缩存储示意图

注意，以上两个公式既是用来存储矩阵中元素的，也用来从数组中提取矩阵相应位置的元素。例如，如果想从图 3 中的数组提取矩阵中位于  $(3,1)$  处的元素，由于该元素位于下三角，需用下三角公式获取元素在数组中的位置，即：

$$k = \frac{i \times (i - 1)}{2} + j - 1 = \frac{3 \times (3 - 1)}{2} + 1 - 1 = 3$$

结合图 3，数组下标为 3 的位置存储的是元素 3，与图 1 对应。

## 三角阵只需要存储非零元素

如果矩阵中分布有大量的元素 0，即非 0 元素非常少，这类矩阵称为稀疏矩阵。

压缩存储稀疏矩阵的方法是：只存储矩阵中的非 0 元素，与前面的存储方法不同，稀疏矩阵非 0 元素的存储需同时存储该元素所在矩阵中的行标和列标。

$(1,1,1)$ ：数据元素为 1，在矩阵中的位置为  $(1,1)$ ；

$(3,3,1)$ ：数据元素为 3，在矩阵中的位置为  $(3,1)$ ；

$(5,2,3)$ ：数据元素为 5，在矩阵中的位置为  $(2,3)$ ；

除此之外，还要存储矩阵的行数 3 和列数 3；

由此，可以成功存储一个稀疏矩阵。

行逻辑链接的顺序表和三元组顺序表的实现过程类似，它们存储矩阵的过程完全相同，都是将矩阵中非 0 元素的三元组（行标、列标和元素值）存储在一维数组中。但为了提高提取数据的效率，前者在存储矩阵时比后者多使用了一个数组，专门记录矩阵中每行第一个非 0 元素在一维数组中的位置。

```

typedef struct
{
    int row, col; //行,列
    int ele;      //元素
} triple;
typedef struct
{
    triple data[20]; //三元组
    int n, m, num;   //长,宽,非零元素数量
    int index[20];   //首个非零元素位置
} stu;
int main()
{
    stu matrix;
    matrix.num = 0;

```

, 如果想从行逻辑链接的顺序表中提取元素, 则可以借助 rpos 数组提高遍历数组的效率。

```

for (i = 1; i <= matrix.m; i++)
{
    flag = 1;
    matrix.index[i] = matrix.num; // *首个非零元素位置初始位置是行首
    for (j = 1; j <= matrix.n; j++)
    {
        scanf("%d", &temp);
        if (temp)
        {
            matrix.data[matrix.num].row = i;
            matrix.data[matrix.num].col = j;
            matrix.data[matrix.num++].ele = temp;

            if (flag)
            {
                matrix.index[i] = matrix.num - 1;
                flag = 0;
            }
        }
    }
    matrix.index[i] = matrix.num;//最后一行行首
}

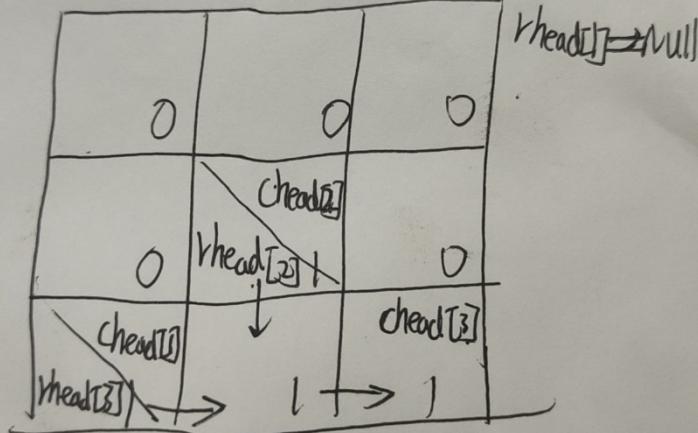
```

```

for (i = 1; i <= matrix.m; i++)
{
    for (j = 1; j <= matrix.n; j++)
    {
        value = 1;
        for (int k = matrix.index[i]; k < matrix.index[i + 1]; k++) //!重点: 分布是连续的, 之间的元素就是
        {
            if (matrix.data[k].row == i && matrix.data[k].col == j)
            {
                printf(" %d ", matrix.data[k].ele);
                value = 0;
                break;
            }
        }
        if (value)
            printf("0 ");
    }
    printf("\n");
}

```

## 十字链表



```
typedef struct CLnode // 节点结构体
{
    int r, c, ele;
    // 位置和元素
    struct CLnode *prow, *pcol; // 行链指针，列链指针
} CLnode, *Clink;

typedef struct // 矩阵信息
{
    int row, col, sum; // 总行数，总列数，非零元素数 ;
    Clink *rhead, *chead; // 行列头指针动态数组
} *Crosslist;
```

```
if (! (M->rhead[m]) || n <= M->rhead[m]->c) // ! 在(首)节点前插入
{
    p->prow = M->rhead[m];
    M->rhead[m] = p; // 首指针重新指向第一列
}
else
{
    q = M->rhead[m];
    while (q)
    {
        if (!(q->prow) || q->prow->c >= n) // ! 在节点后插入, 注意条件顺序, 这样只有在非空的时候才会
        {
            p->prow = q->prow;
            q->prow = p;
        }
        q = q->prow;
    }
}
```