

第 2 章 数据类型、运算符与表达式

(视频讲解：6 小时)

程序员编写程序就像裁缝做衣服，裁缝需要区分布料的类型，程序员需要区分数据的类型，裁缝需要各种工具对衣服进行加工，程序员需要用加减乘除等运算符对数据进行加工，从而得到自己想要的结果，通过本章，你将掌握以下内容：

- 数据类型有哪些
- 常量与变量的差异
- 整型，浮点型，字符型的原理及使用
- `Scanf` 的原理及使用
- 运算符与表达式

2.1 数据类型

裁缝做衣服，有纤维，纯棉，丝绸等材料，那么我们有哪数据类型呢？请看图 2.1-1

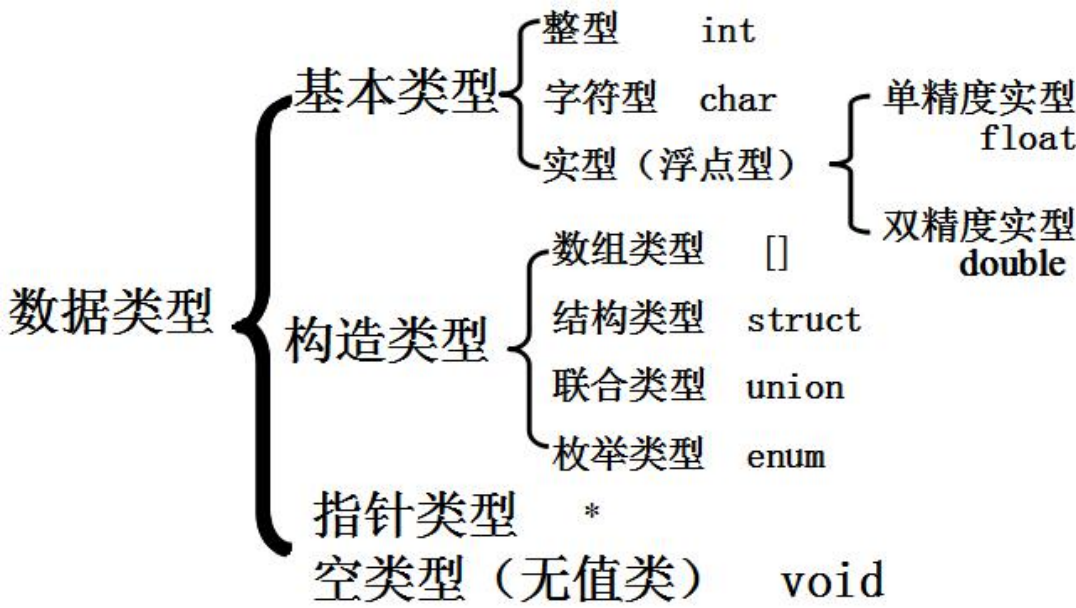


图 2.1-1

C 语言的关键字，在今后的学习中我们将逐步学习这些关键字（不用去记），这里只是列一下，让大家知道关键字有哪些，后面我们会逐个使用到，使用到时我们会详细讲解，这里列出避免大家后面命名变量名时，和我们关键字重名。表 2.1-1 是以字母顺序列出

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

do	if	static	while
----	----	--------	-------

表 2.1-1 C 语言关键字

2.2 常量

本章我们将学习基本类型，整型，字符型，实型（又称之为浮点型），基本数据类型有常量和变量，首先我们先来看一下常量，在程序运行过程中,其值不能被改变的量称为常量。

常量区分为不同的类型：

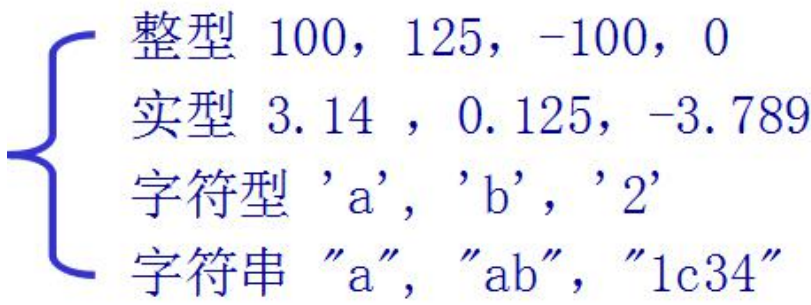


图 2.2-1

整型常量，实型常量，字符型常量在编译时直接编入代码段，字符串常量是存在字符串常量区。大家先初步了解一下，后面内容会深入讲解。“你好”这样双引号中间是汉字的也是字符串常量，无论双引号内放各种 ASCII 码字符，还是汉字，其他国家语言等，都是字符串常量。

2.3 变量

变量代表内存中具有特定属性的一个存储单元，它用来存放数据，这就是变量的值，在程序运行期间，这些值在程序的执行过程中是可以改变的。

变量名实际上是一个以一个名字对应代表一个地址，在对程序编译连接时由编译系统给每一个变量名分配对应的内存地址。从变量中取值，实际上是通过变量名找到相应的内存地址，从该存储单元中读取数据。如图 2.3-1 所示：

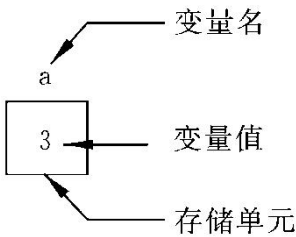


图 2.3-1

变量命名的规定：C 语言规定标识符只能由字母、数字和下划线三种字符组成，且第一个字符必须为字母或下划线。

- 例：sum, _total, month, Student_name,
- lotus_1_2_3 , BASIC, li_ling
- 正确
- M.D.John, ¥123,3D64,a>b
- 错误

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，要求对所有用到的变量作强制定义，也就是“先定义，后使用”，同时选择变量名和其它标识符时，应注意做到“见名知意”，即选有含意的英文单词（或其缩写）作标识符。注意变量名的命名不能与关键字同名！

2.4 整型数据

2.4.1 符号常量

我们通过关键字 `int` 来定义一个整型变量，首先我们来看一个例子吧

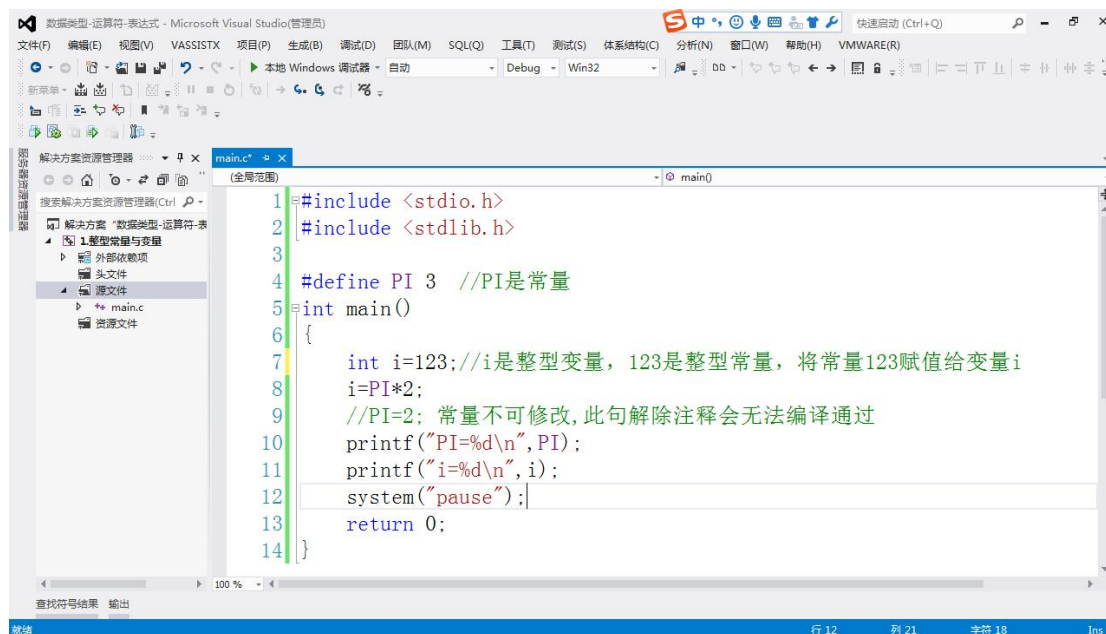


图 2.4.1-1

图 2.4.1-1 中整型变量 `i` 有自己的内存空间，所以可以存储整型常量 `123`，也可以存储 `PI*2`，也就是 `3*2`，其 `2` 实就是 `6`，那通过 `define` 定义的 `PI` 为什么是常量呢，接下来我们通过改变编译设置，来理解什么是预处理，理解了预处理，就明白为什么 `define` 的内容叫常量。右键点击项目，选择属性，设置预处理为“是”，如图 2.4.1-2

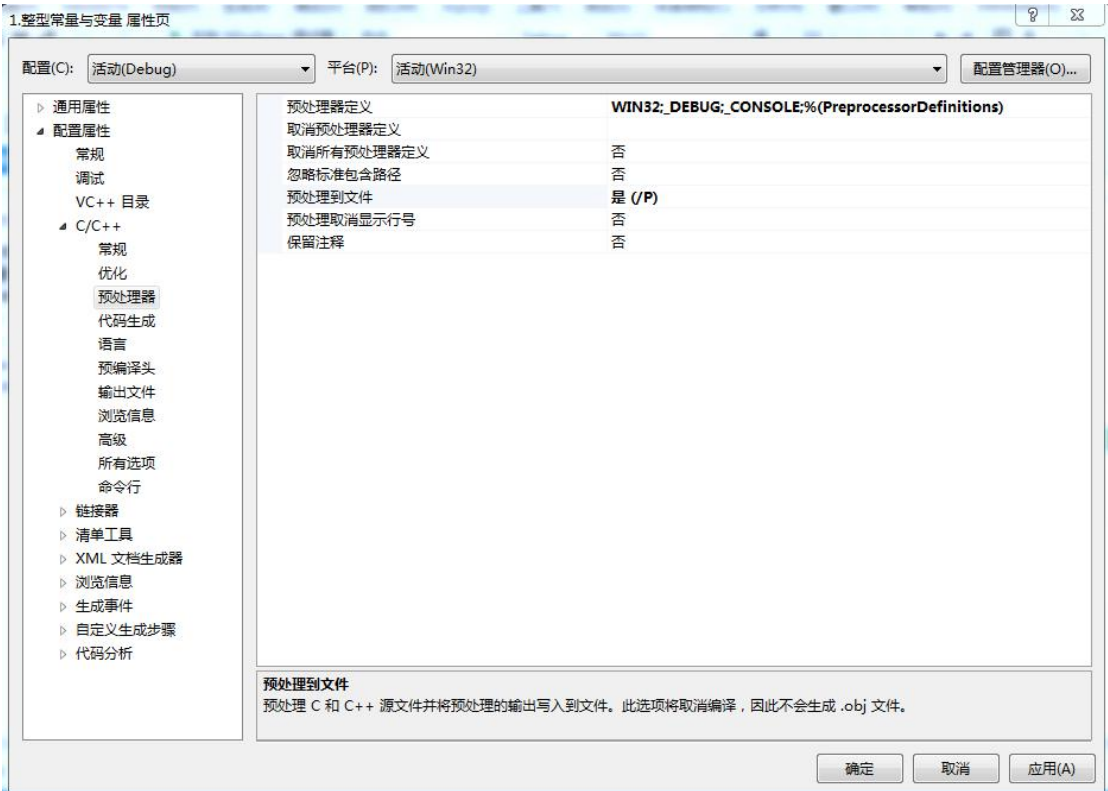


图 2.4.1-2

设置完毕后，重新编译，当然会显示重新编译失败，不用担心，这是因为开启预编译后，VS 只进行预处理，不再进行其他编译动作（VS 针对没有编译出最终可执行程序的操作，就会提示编译失败）。然后如图 2.4.1-3 所示，右键点击 在资源管理器中打开文件夹，然后双击打开 Debug 文件夹，如图 2.4.1-4 所示。

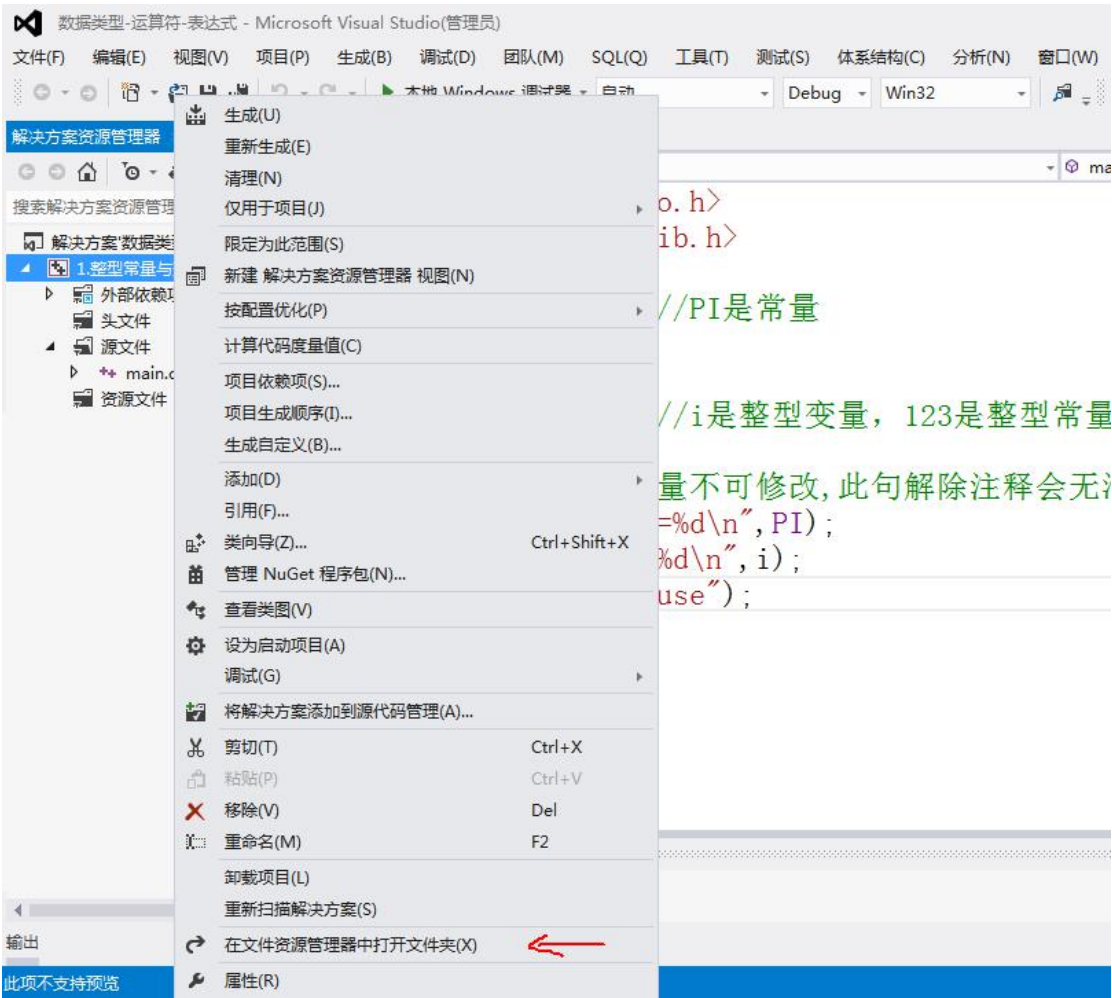


图 2.4.1-3

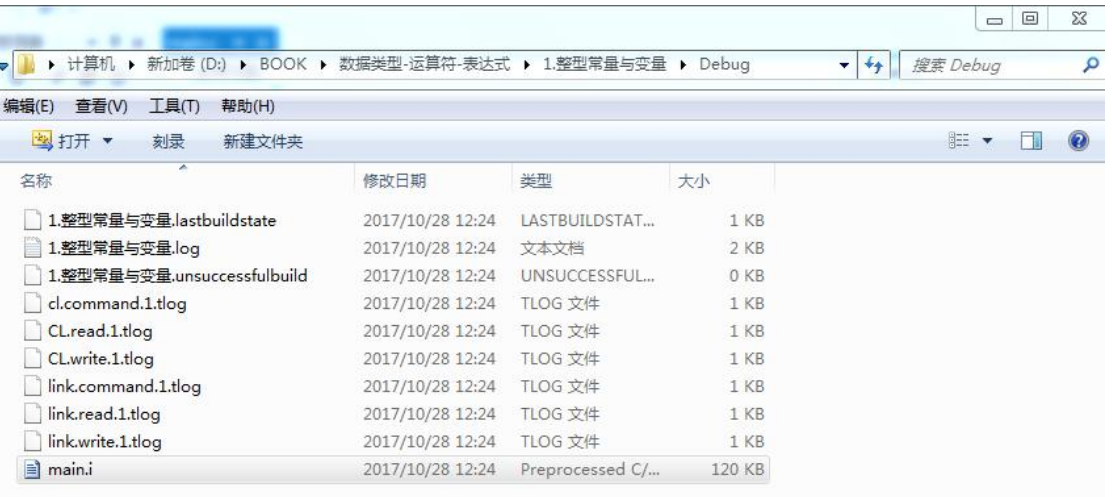


图 2.4.1-4

查看项目文件, 在 Debug 中找到 main.i, 如图 2.4.1-4 所示, 使用写字板或者 NotePad++ 打开 main.i, 直接拖到最下面, 会看到如图 2.4.1-5 内容, 发现 define 已经不见, 是因为预处理时会把所有的 define 消除, 同时我们 define 的 PI 为 3, 编译器在预处理后就会把代码中所有出现常量 PI 的地方, 替换为 3, 所以 PI 是常量, 不可修改, 其作用是我们程序中如果多次用到一个常量, 那么就使用 define 的方法来实现, 便于编写程序时, 当改变多处

同一常量，只需改变 `define` 后符号常量 `PI` 即可。

```

11555 #pragma pack(pop)
11556
11557 #line 933 "d:\\program files\\microsoft visual studio 11.0\\vc\\include\\stdlib.h"
11558
11559 #line 3 "d:\\book\\数据类型-运算符-表达式\\1. 整型常量与变量\\main.c"
11560
11561
11562 int main()
11563 {
11564     int i=123;
11565     i=3*2;
11566
11567     printf("PI=%d\\n", 3);
11568     printf("i=%d\\n", i);
11569     system("pause");
11570     return 0;
11571 }

```

图 2.4.1-5 预处理文件内容

2.4.2 整型常量的不同进制表示

计算机中只能存储二进制，也就是 0 和 1，对应的物理硬件上是高低电平。为了方便观察内存中的二进制情况，除了正常我们使用的十进制外，还提供了十六进制，八进制，下面我们来看一下不同进制的对应关系，首先计算机中 1 个字节=8 位，1 位即二进制的 1 位，存储 0 或者 1。

`int` 型，大小为 4 个字节，即 32 位

二进制数 0100 1100 0011 0001 0101 0110 1111 1110 最低位是 2 的零次方，一直到最高位为 2 的 30 次方，最高位为符号位，具体到补码部分讲解。

八进制数 011414253376 以 0 开头标示，变化范围从 0-7，对应 2 进制为每 3 位二进制变换为 1 位八进制，将上面的二进制每三位隔开，效果如下：

01 001 100 001 100 010 101 011 011 111 110，然后每三位对应 0-7 进行对应转换，就会得到八进制数 011414253376，因为实际编程时，识别八进制前面需要加 0，所以我在前面加了一个 0。

十进制数 1278301950 直接赋值即可

十六进制 0x4C3156FE 以 0x 开头标示，变为范围为 0-9，A-F，A 代表 10，F 代表 15，对应 2 进制为每 4 位二进制转换为 1 位 16 进制，具体可以自行对应

首先我们新建项目整型的进制转换，如何在已有的解决方案内新建项目，如图 2.4.2-1 所示，在解决方案位置，右键点击，找到添加，选择新建项目，填写名字整型的进制转换，同时如图 2.4.2-2 所示，设置“整型的进制转换”为启动项目。

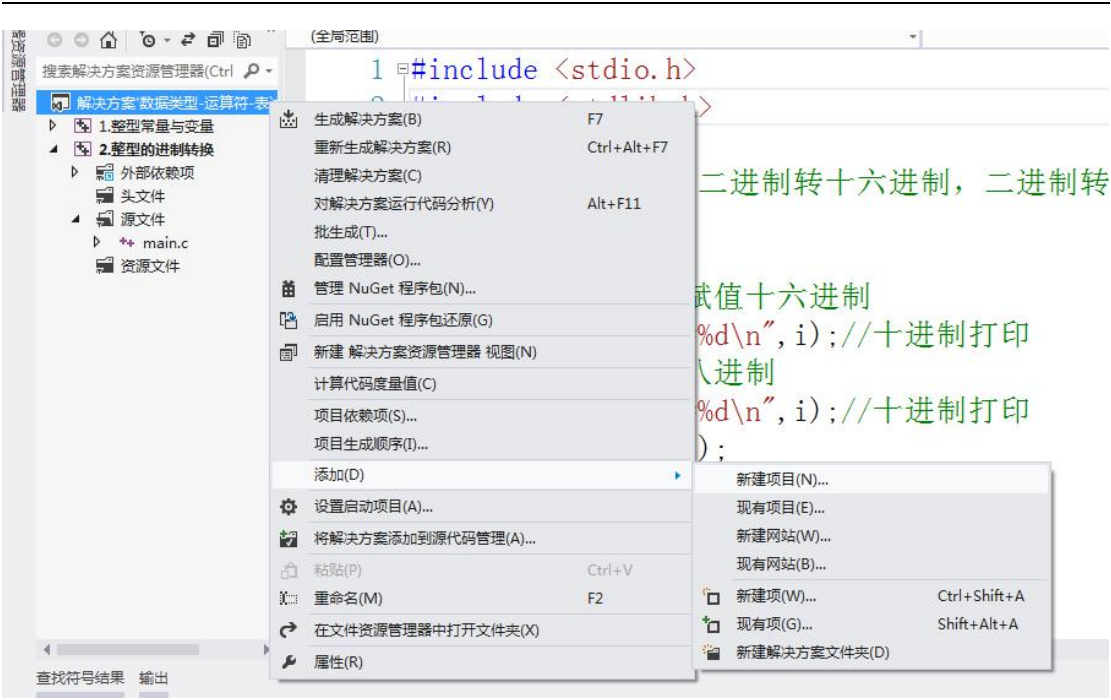


图 2.4.2-1

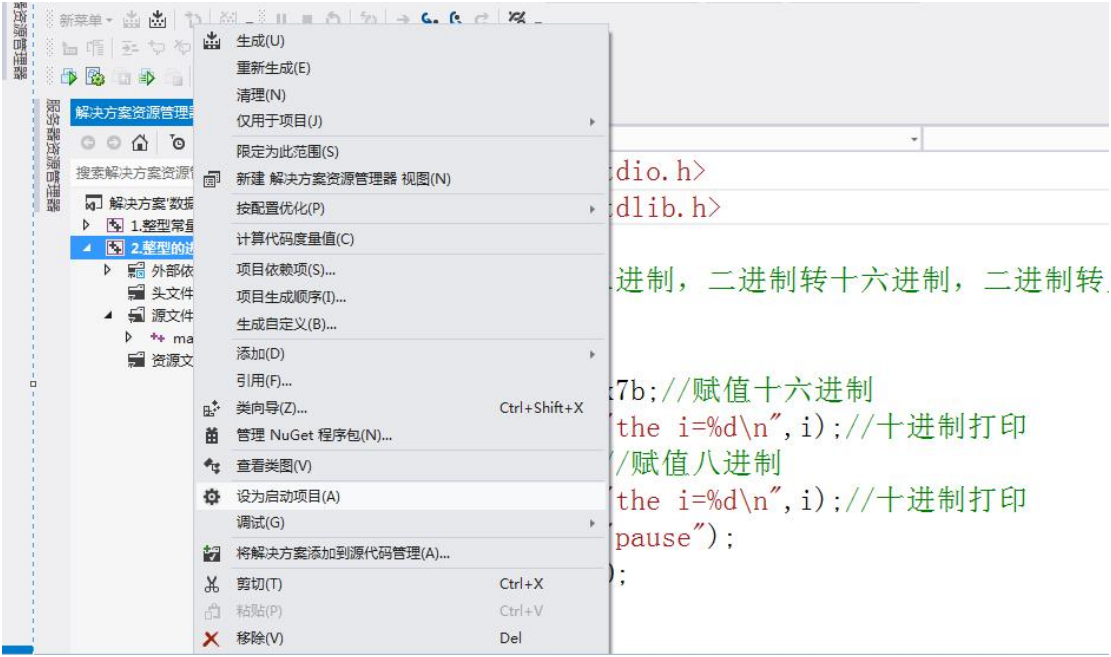


图 2.4.2-2

在第 7 行设置断点，然后点击执行按钮，得到如图 2.4.2-3 效果，我们在监视窗口输入取地址 `i`，即 `&i`，然后得到 `i` 的地址，左键点击拖入右边的内存区域，可以看到 `i` 的内存，我们的 32 位控制台应用程序，地址范围是 0 到 4G，从 `0x00000000` 到 `0xFFFFFFFF`，如图 2.4.2-4，称为进程（程序运行起来叫进程）地址空间，我们程序编译完毕后，开始执行时，会被放入进程地址空间的代码段区域，执行到那句，PC 指针就指向那句代码，比如目前我们执行到 `int i=0x7b`，变量 `i` 会在栈空间上分配空间，大小为 4 个字节，起始地址为 `0x0013FAF8`，我们按 F10，会看到如图 2.4.2-5，`i` 的值变为 7b（我们查看内存以 16 进制方式），其十进制值为 $7 \times 16 + 11 = 123$ 。`i` 的值是 `0x0000007b`，为什么显示效果为 `7b 00 00 00` 呢，这个是因为英特尔的 CPU 为小端存储，所以低位在前，高位在

后。

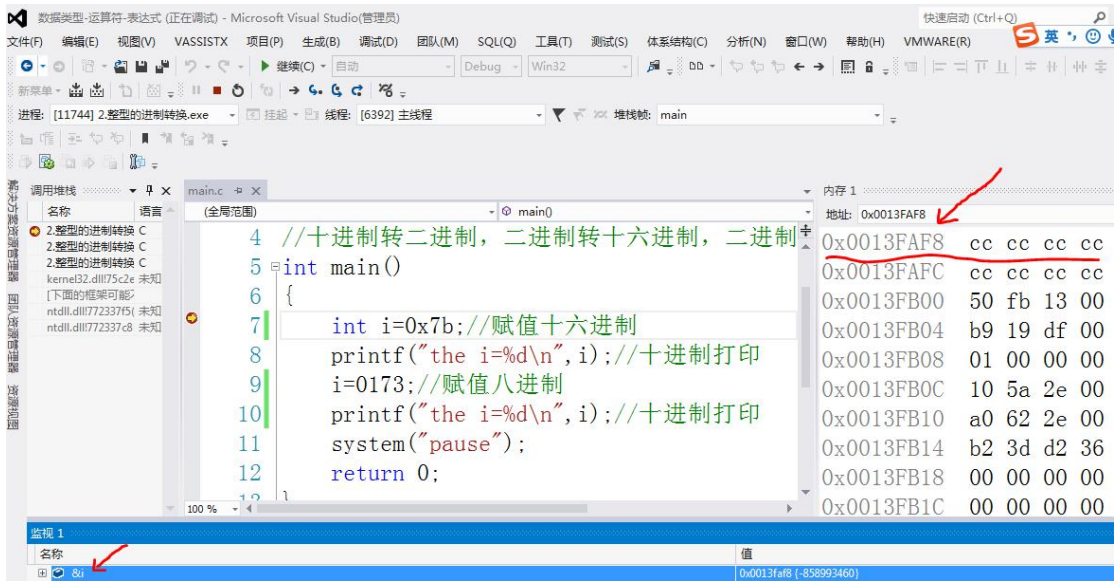


图 2.4.2-3

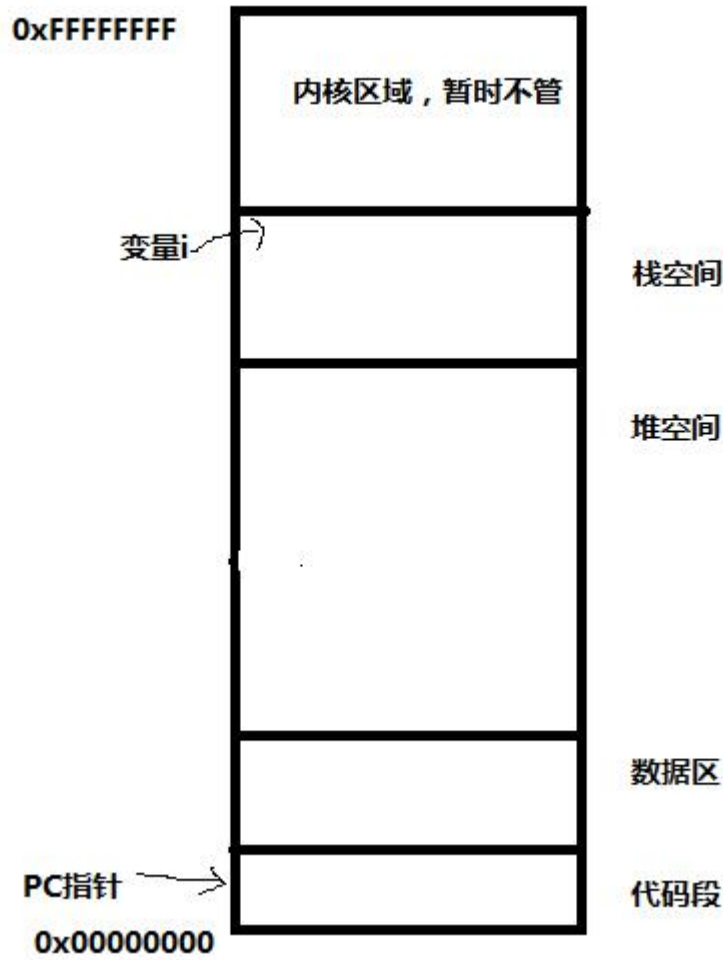


图 2.4.2-4

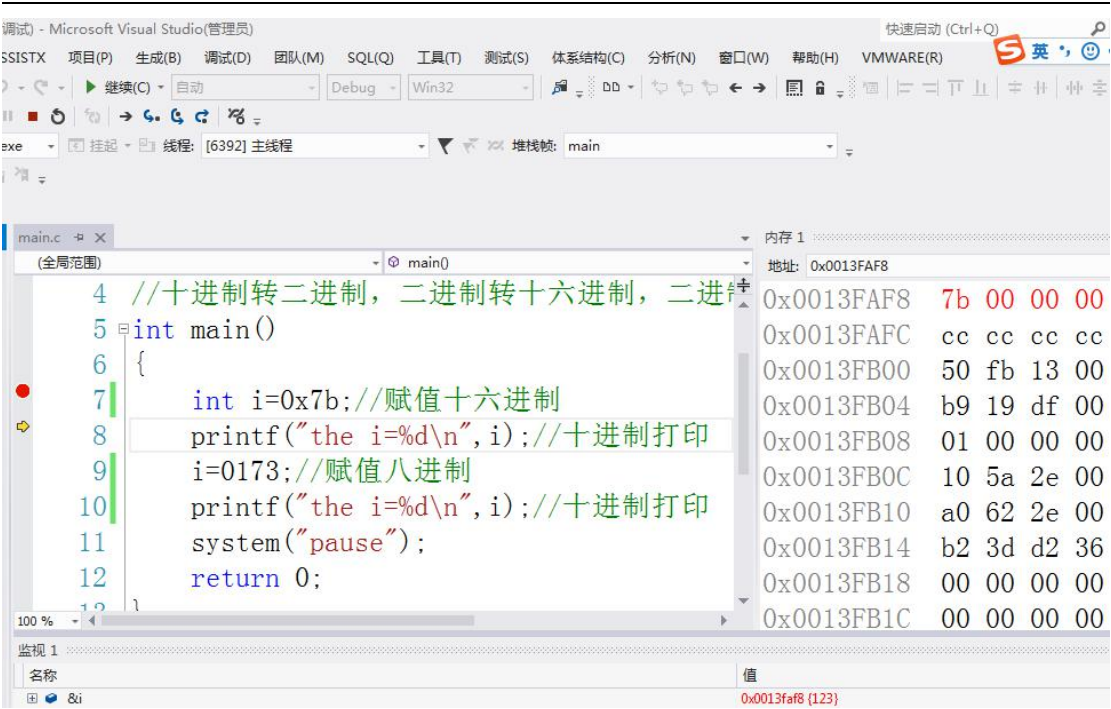


图 2.4.2-5

我们直接按继续按钮，直接运行到最后，得到如图 2.4.2-6 结果，八进制 0173，对应转换为 10 进制方法一直， $1 \times 8^2 + 7 \times 8 + 3 = 123$ ，对应 10 进制 123 如何转换为二进制呢，就是拿 123 不断的除 2，然后把余数写在右边，把商写在下面，然后除，直到商为 1，然后逆向写出，我们得到的二进制为 1111011，详细除的过程如图 2.4.2-7 所示，对应的 16 进制为 7b，十进制对应转十六进制，方法同二进制一直，只是除 16，八进制就是除 8，小伙伴们可以自行练习尝试。

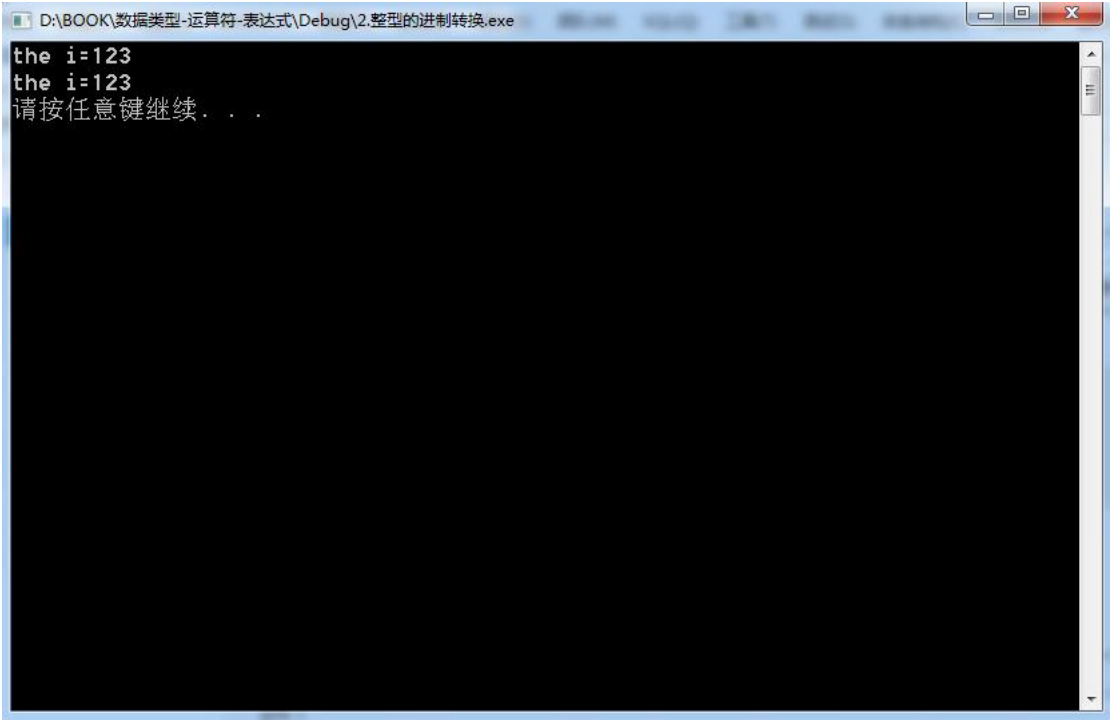


图 2.4.2-6

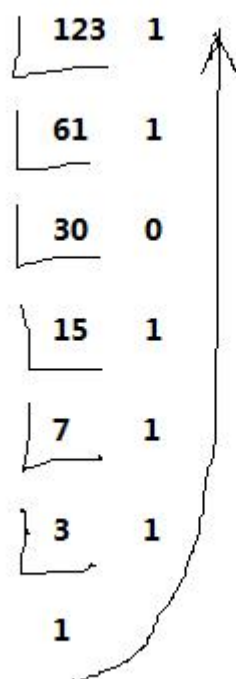


图 2.4.2-7

小技巧:

针对我们手动转换一个数后，不知道对应的进制是否正确，可以在 windows 的“开始”菜单的“附件”命令中打开计算机软件，点击查看菜单中的程序员，得到如图 2.4.2-8 所示的计算器，通过输入一个十进制数以后，点击十六进制，八进制或者二进制，即可得到对应进制的转换结果。



图 2.4.2-8

2.4.3 补码的作用

计算机的 CPU 是无法做减法操作的，只能做加法，其实 CPU 中有一个逻辑单元叫加法器，计算机所做的减法，乘法，除法，都是由科学家将其变化为加法。那么减法是如何实现的呢，其实 2-5，实际所使用的方法是 $2+(-5)$ 进行的，而计算机只能存储 0 和 1，我们编写程序如图 2.4.3-1，来查看计算机如何存储-5，计算机是如何表示-5 的呢，5 的二进制为 101，称为原码，计算机用补码表示-5，补码是对原码取反加 1，也就是计算机表示-5 是对 5（101）的二进制进行取反加 1，如图 2.4.3-2，-5 在内存中的存储是 0xfffffff b，因为 5 取反得到的是 0xfffffff a，然后加 1 就是 0xfffffff b，然后对其加 2，最终得到的结果为 0xfffffff d，见图 2.4.3-3，就是 k 的值，当最高位为 1 时，即代表负数，这个时候我们要得到其原码，才能知道 0xfffffff d 的值，就是对其取反加 1（当然你可以减一取反，效果是一样的），就得到 3，所以其值为 -3。

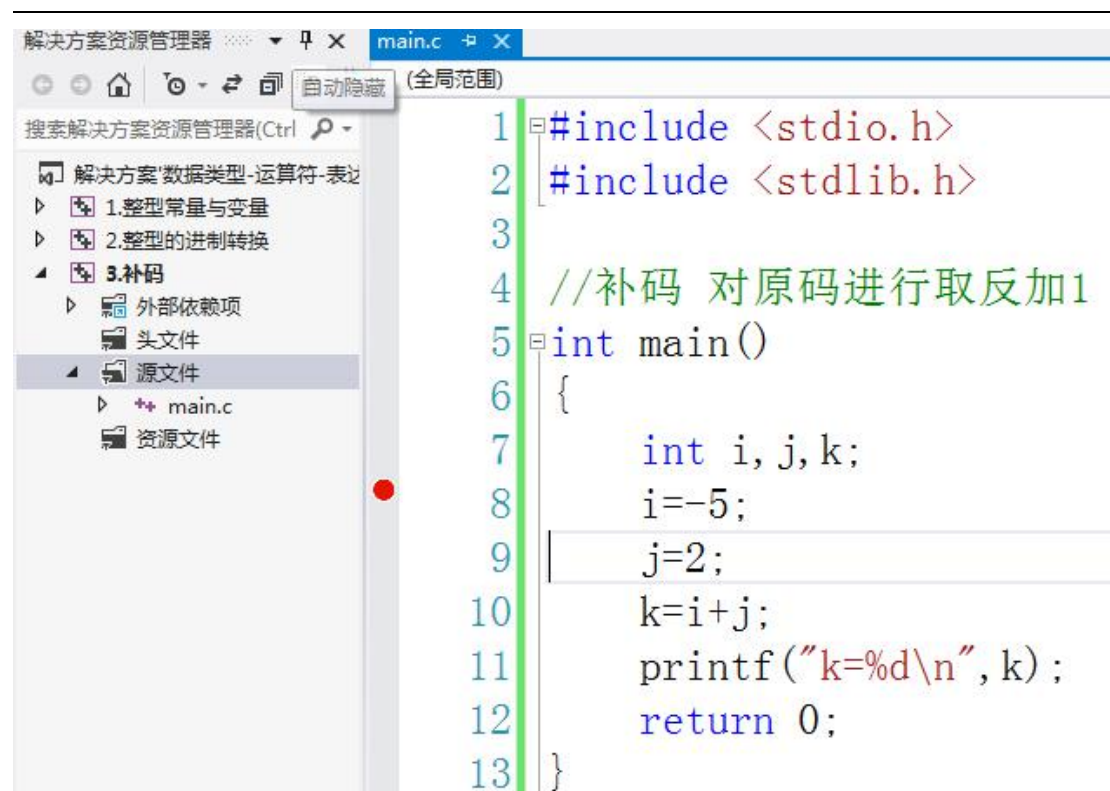


图 2.4.3-1

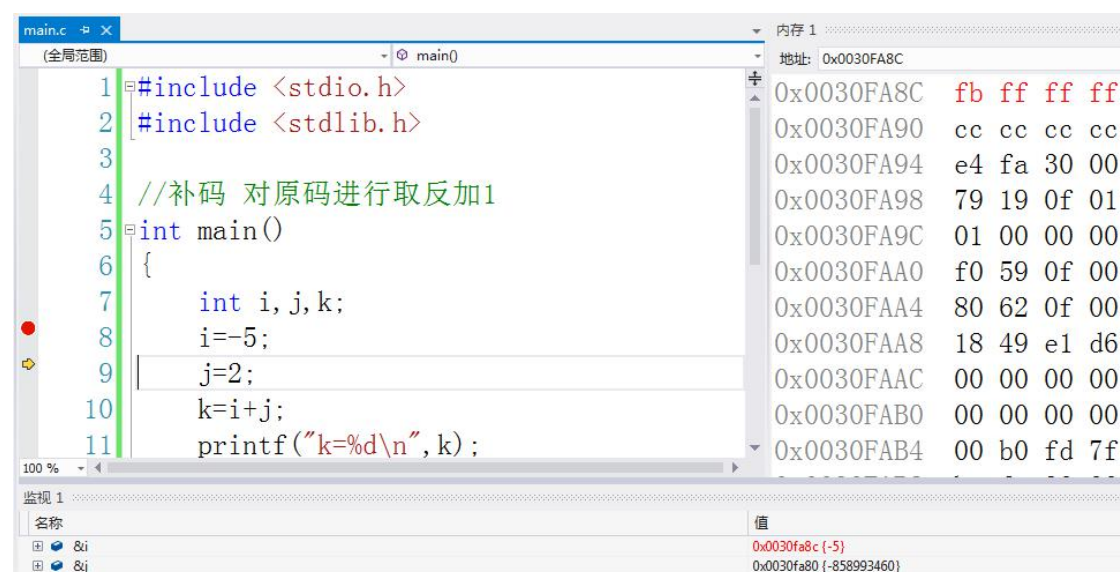


图 2.4.3-2

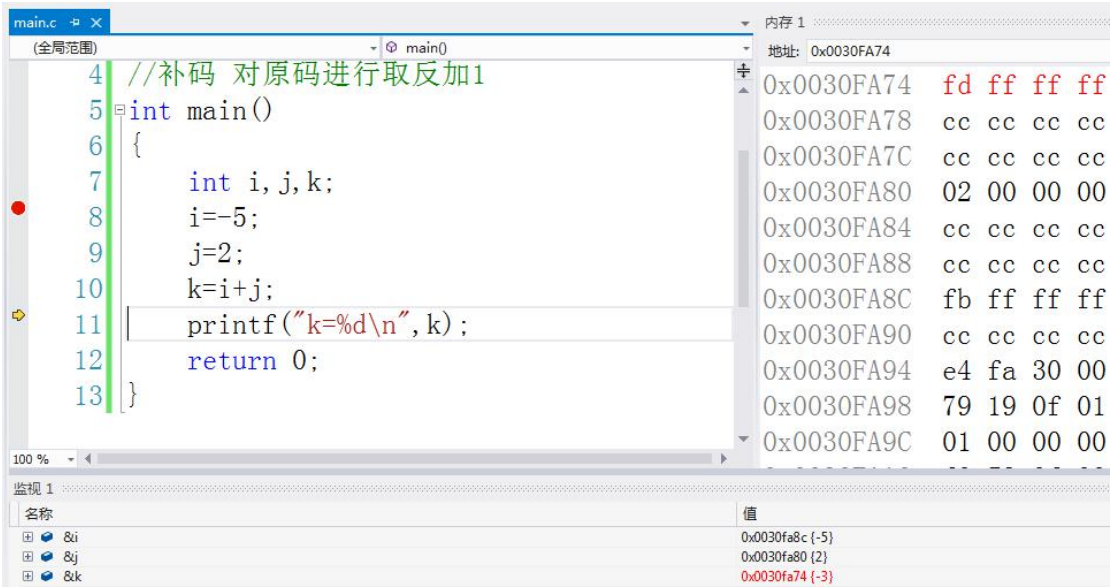


图 2.4.3-3

2.4.4 整型变量

整型变量分为六种类型，如图 2.4.4-1 所示，有符号类型与无符号类型最高位代表意义不同，如图 2.4.4-2 所示，具体对应不同类型变量可以表示的整型数范围如表 2.4.4-1 所示，如果超出数据范围就会发生溢出现象，导致计算出错。

有符号基本整型
有符号短整型
有符号长整型
无符号基本整型
无符号短整型
无符号长整型

(signed)int
(signed)short (int)
(signed) long (int)
unsigned int
unsigned short (int)
unsigned long (int)

注意： 括号表示其中的内容是可选的.

图 2.4.4-1

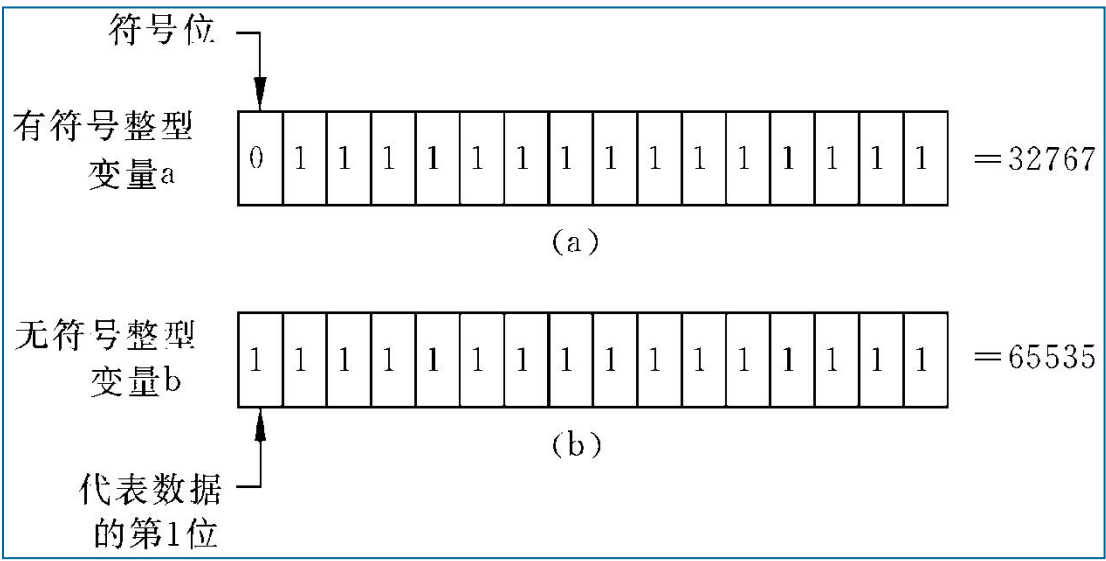
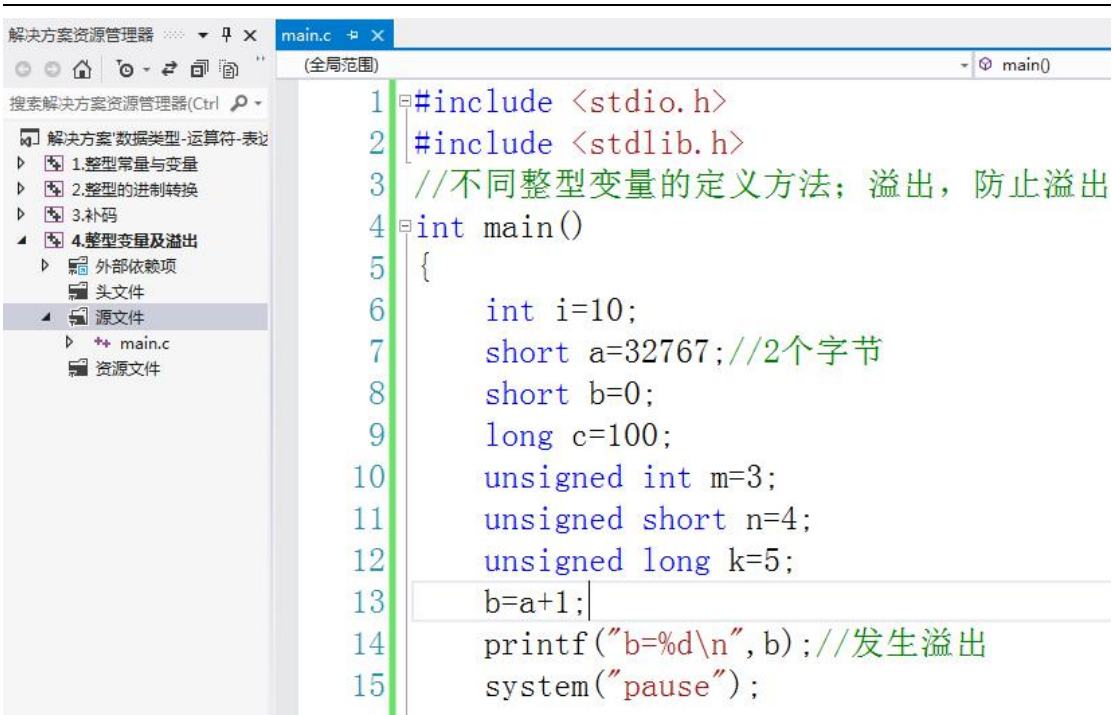


图 2.4.4-2

类型	类型说明符	长度	数的范围
基本型	int	4 字节	$-2^{31} \sim 2^{31}-1$
短整型	short	2 字节	$-2^{15} \sim 2^{15}-1$
长整形	long	4 字节 (64 位为 8 字节)	$-2^{31} \sim 2^{31}-1 / -2^{63} \sim 2^{63}-1$
无符号整型	unsigned int	4 字节	$0 \sim (2^{32}-1)$
无符号短整型	unsigned short	2 字节	$0 \sim (65535)$
无符号长整型	unsigned long	4 字节 (64 位为 8 字节)	$0 \sim (2^{32}-1) / 0 \sim (2^{64}-1)$

表 2.4.4-1 整型不同类型数值变化范围

通过图 2.4.4-3 我们可以看到不同类型的整型变量的定义方法，同时我们来看一下溢出，有符号短整型数可以表示的最大值为 32767，当我们对其加 1 时，b 的值会变为多少呢？实际运行打印得到的是-32768，为什么会这样，因为 32767 对应的 16 进制为 0x7fff，那么加 1 就变为 0x8000，首位为 1，就变为了一个负数，而我们对这个负数取其原码，就是其本身，值为 32768，所以 0x8000 是最小的负数，即-32768，这就发生了溢出，因为我们对 32767 加 1，希望得到的值是 32768，结果得到的值为-32768，就会导致计算结果错误，在使用整型时，一定要注意数值大小，不要超过对应整型数的表示范围，可能有的小伙伴会问，那我开发的系统，大于 $2^{64}-1$ 怎么办，这个时候我们可以自行实现大整数加法，具体到后面数组会解决这个问题。



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 //不同整型变量的定义方法；溢出，防止溢出
4 int main()
5 {
6     int i=10;
7     short a=32767;//2个字节
8     short b=0;
9     long c=100;
10    unsigned int m=3;
11    unsigned short n=4;
12    unsigned long k=5;
13    b=a+1;
14    printf("b=%d\n", b); //发生溢出
15    system("pause");
```

图 2.4.4-3

图 2.4.4-3 代码的执行结果如图 2.4.4-4:



图 2.4.4-4

思考题:

针对上题的溢出问题，我们如何修改，可以让打印出的 b 的值为 32768，而不是-32768 呢？

2.5 浮点型数据

2.5.1 浮点型常量

浮点型常量的表示方法，有两种形式，如下表所示，e 代表 10 的幂次，可正可负

小数形式	0.123
指数形式	3e-3 (为 3×10^{-3} ，即 0.003)
注意:	字母 e(或 E)之前必须有数字，且 e 后面的指数必须为整数
正确示例	1e3、1.8e-3、-123e-6、-.1e-3
错误示例	e3、2.1e3.5、.e3、e

表 2.5.1-1

2.5.2 浮点型变量

我们通过 float 关键字或 double 关键字进行浮点型变量定义，float 类型占据 4 个字节大小的内存空间，double 占据 8 个字节的空间，与整型数据的存储方式不同，浮点型数据是按照指数形式存储的。系统把一个浮点型数据分成小数部分和指数部分，分别存放。指数部分采用规范化的指数形式，指数也分正负，如图 2.5.2-1（考研机试同学不用掌握本节的浮点数存储原理，使用浮点数直接使用 double 即可）。

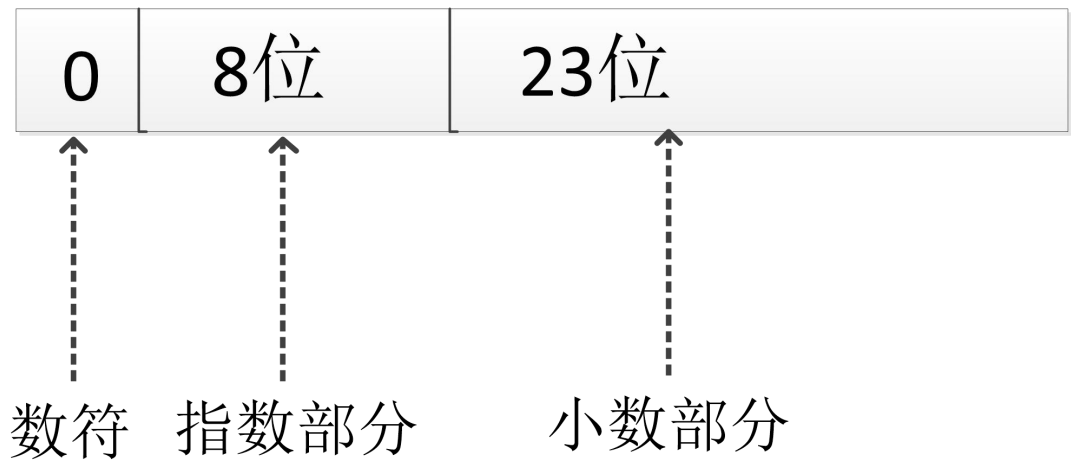


图 2.5.2-1

数符占用 1 位，是 0 就代表正数，1 就代表负数，下面我们用浮点数 4.5 举例来讲一下 IEEE-754 存储标准，

格式	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM
二进制	01000000	10010000	00000000	00000000
16 进制	40	90	00	00

S: 为 0，是个正数。
E: 为 10000001 转为 10 进制为 129， $129-127=2$ ，即实际指数部分为 2。
M: 为 0010 0000 0000 0000 0000 000。这里，在底数左边省略存储了一个 1，使用实际底数表示为 1.001000000000000000000000

计算机并不能够计算 10 的幂次，指数值为 2，代表 2 的 2 次幂，因此将 1.001 向左移动 2 位即可，也就是 100.1，然后转换为 10 进制，整数部分就是 4，然后小数部分是 2^{-1} 刚好等于 0.5，因此为 4.5。浮点数的小数部分，是通过 $2^{-1}+2^{-2}+2^{-3}$ 等去近似一个小数的。

接下来我们来看下浮点数的精度控制，浮点型变量分为单精度（float 型）、双精度（double 型）和长双精度型（long double）三类形式。如表 2.5.2-1 所示，因为浮点数使用的是指数表示法，所以我们不用担心数的范围，以及去看浮点数的内存（自己算起来麻烦），我们需要关注的是浮点数的精度问题，如图 2.5.2-2，我们赋给 a 的值为 1.23456789e10，加 20 后，应该得到的值为 1.234567892e10，但是却是 1.23456788e10，变的更小了，我们把这种现象称为精度丢失，原因就是 float 能够表示的有效数字为 7 位，最多只保证 1.234567e10 的正确性，如果要达到正确，我们需要把 a 和 b 均改为 double 类型。

类型	位数	数的范围	有效数字
float	32	$10^{-37} \sim 10^{38}$	6~7 位
double	64	$10^{-307} \sim 10^{308}$	15~16 位
long double	128	$10^{-4931} \sim 10^{4932}$	18~19 位

表 2.5.2-1 浮点数数值范围及精度

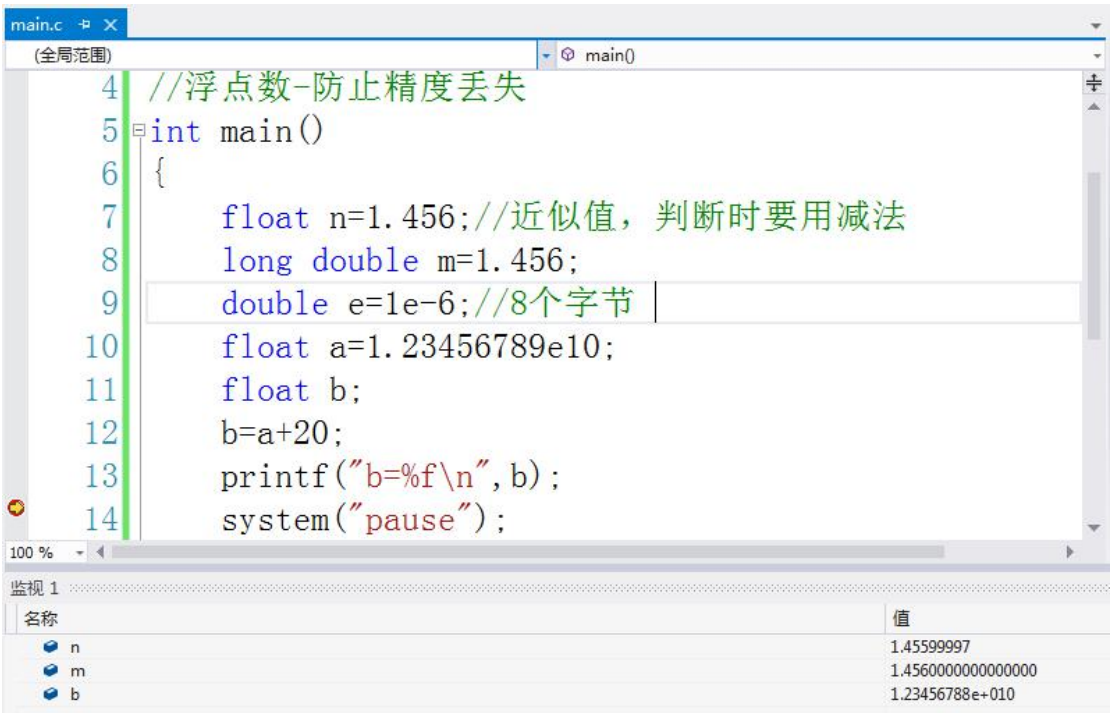


图 2.5.2-2

图 2.5.2-2 的执行结果如图 2.5.2-3:

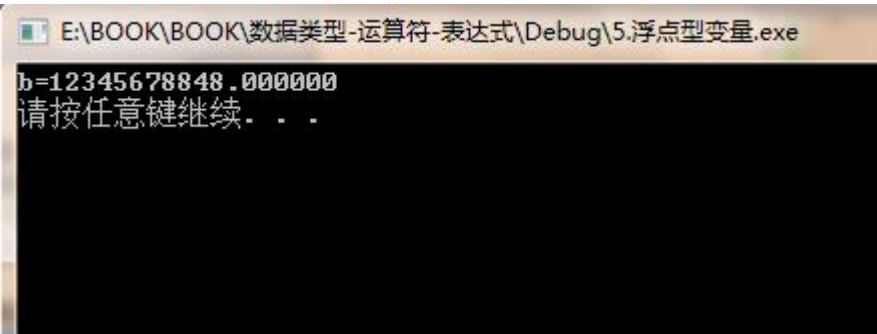


图 2.5.2-3

思考题:

有兴趣的小伙伴可以把 a, b 都改为 double 型, 看看实际求和后 b 的值是否是正确的?

2.6 字符型数据

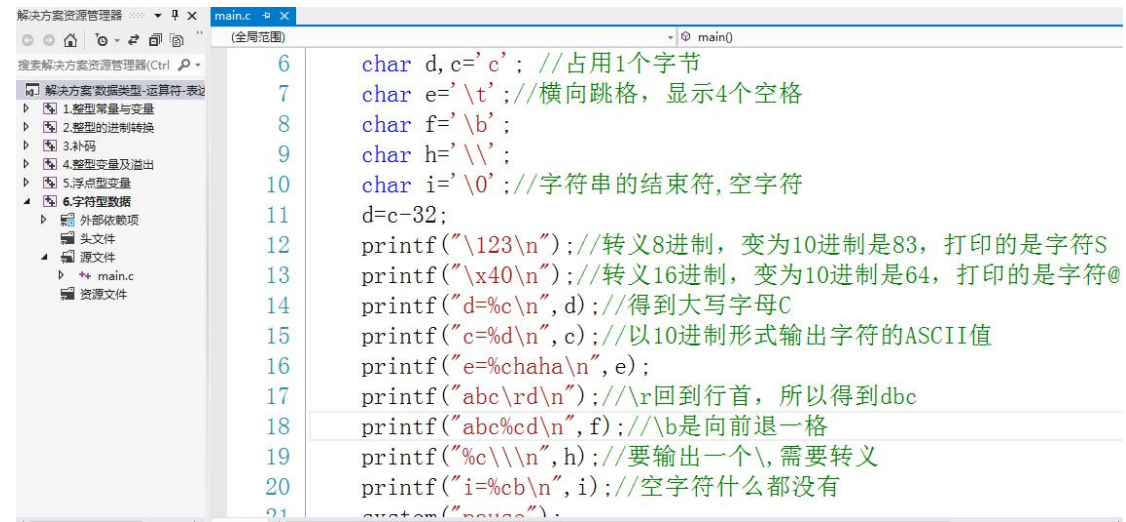
2.6.1 字符型常量

如表 2.6.1-1 所示, 用单引号包含的一个字符是字符型常量, 且只能包含一个字符! 下面是例子示范, 以“\”开头的特殊字符称为转义字符, 转义字符用来表示回车, 退格等功能键。编写图 2.6.1-1 的实例, 运行查看打印效果, 根据自己的想法进行修改, 即可理解转义字符。

正确示例	‘a’ , ‘ A’ , ‘1’ , ‘ ’
错误示例	‘abc’ , “a” , “ ”
转义字符	\n 换行

\t	横向跳格
\r	回车
\\	反斜杠
\b	退格
\0	空字符，用于标示字符串的结尾，不是空格，打印不出，和佛家的空类似
\ddd	ddd 表示 1 到 3 位八进制数字，非常鸡肋，没什么用
\xhh	hh 表示 1 到 2 位十六进制数字，非常鸡肋，没什么用

表 2.6.1-1



```
6 char d, c='c'; //占用1个字节
7 char e='\t'; //横向跳格, 显示4个空格
8 char f='\b';
9 char h='\\';
10 char i='\0'; //字符串的结束符, 空字符
11 d=c-32;
12 printf("\123\n"); //转义8进制, 变为10进制是83, 打印的是字符S
13 printf("\x40\n"); //转义16进制, 变为10进制是64, 打印的是字符@
14 printf("d=%c\n", d); //得到大写字母C
15 printf("c=%d\n", c); //以10进制形式输出字符的ASCII值
16 printf("e=%chaha\n", e);
17 printf("abc\rd\n"); //\r回到行首, 所以得到dbc
18 printf("abc%cd\n", f); //\b是向前退一格
19 printf("%c\\n", h); //要输出一个\, 需要转义
20 printf("i=%cb\n", i); //空字符什么都没有
21 system("pause");
```

图 2.6.1-1

图 2.6.1-1 代码的执行效果如图 2.6.1-2:

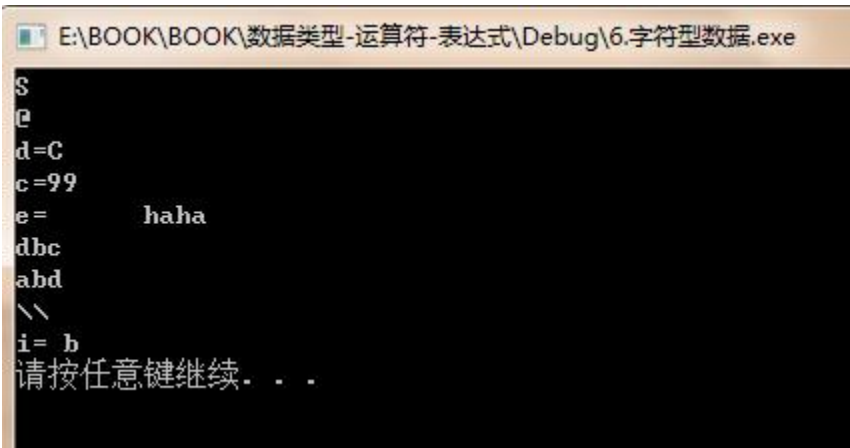


图 2.6.1-2

思考题:

为什么 abc\rd 打印出的效果是 dbc 呢?

2.6.2 字符数据在内存中的存储形式及其使用方法

字符型变量使用关键字 `char` 进行定义，占用 1 个字节大小的空间，一个字符常量存放到一个字符变量中，实际上并不是把该字符的字型放到内存中去，而是将该字符的相应的 ASCII 码值放到存储单元中，每一个字符的 ASCII 码值表详见附录 1，当我们打印字符变量时，如果以字符形式打印，实际过程中拿字符变量的值去 ASCII 码表中查，查到对应的字符后，

显示对应的字符。如图 2.6.2-1 所示，这样使字符型数据和整型数据之间可以通用，一个字符数据既可以以字符形式输出，也可以以整数形式输出。也可以通过运算获取你想要的各种字符效果，详见图 2.6.2-2，各位小伙伴可以通过自己想法修改例子。

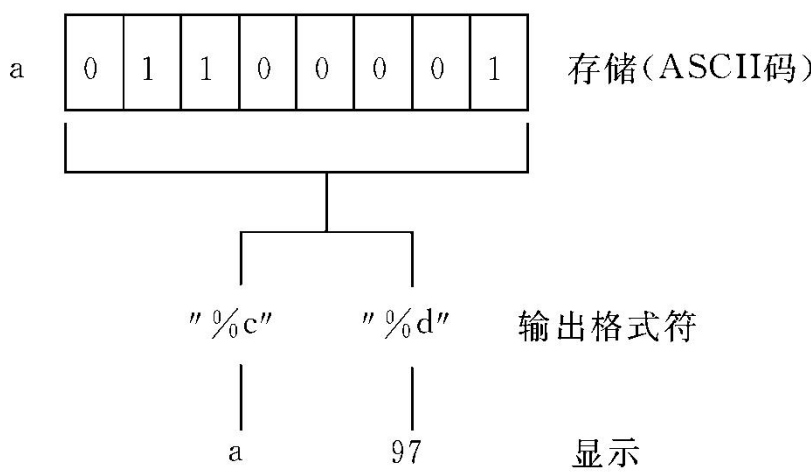


图 2.6.2-1

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     char c, d;
7     c=97;
8     d='a';
9     printf("c=%c, d=%c\n", c, d); //打印都是字母a
10    printf("c=%d, d=%d\n", c, d); //打印都是97
11    c=c-32; //小写字母转换为大写字母
12    printf("c=%c\n", c); //打印字母A
13    system("pause");
14    return 0;
15 }
```

图 2.6.2-2

图 2.6.2-2 的执行结果如图 2.6.2-3 所示，对于字符变量，无论赋值 ASCII 值，还是直接赋值字符，我们通过%c 来打印输出时，得到的是字符，通过%d 打印输出时，得到 ASCII 值。当我们要小写字母转大写字母时，通过附录 1 的 ASCII 表可以发现小写字母与大写字母的差值为 32，因此我们对 c 减去 32，就可以得到大写字母 A。

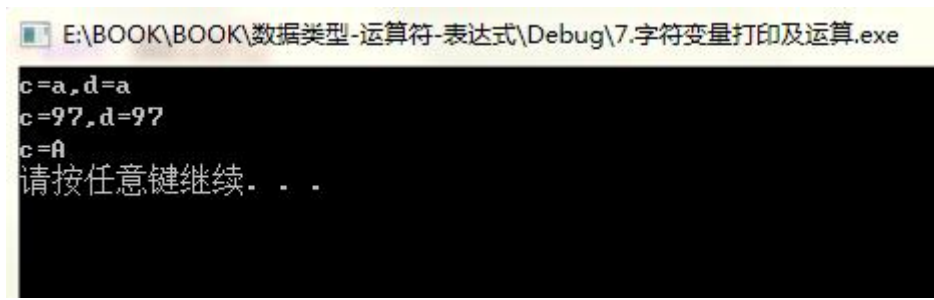


图 2.6.2-3

2.7 字符串常量

字符串常量是一对双撇号（双引号）括起来的字符序列，合法的字符串常量：“How do you do.”，“CHINA”，“a”，“\$123.45”，可以输出一个字符串，如 `printf(“How do you do.”);`；‘a’是字符常量，“a”是字符串常量，二者不同。假如我们定义字符变量 `c`，例如 `char c;`如果 `c="a";c="CHINA"`，这样的赋值都是非法的，不可以将字符串常量赋值给字符变量，C 语言没有定义字符串变量的关键字，具体怎么存字符串，我们到字符数组会进行讲解。

C 规定：在每一个字符串常量的结尾加一个“字符串结束标志”，以便系统据此判断字符串是否结束。C 规定以字符‘\0’作为字符串结束标志。

如果有一个字符串常量“CHINA”，实际上在内存中存储效果如图 2.7-1，它占内存单元不是 5 个字符，而是 6 个字符，即 6 个字节大小，最后一个字符为‘\0’。但在输出时不输出‘\0’，因为‘\0’是显示不出的。

C	H	I	N	A	\0
---	---	---	---	---	----

图 2.7-1

2.8 混合运算

字符型（`char`）、整型（包括 `int,short,long`）、浮点型（包括 `float,double`）可以混合运算。在进行运算时，不同类型的数据要先转换成同一类型,然后进行运算。如图 2.8-1，从短字节到长字节，这种类型转换是由系统自动进行的。同时编译时不会警告，如果反向进行，编译时编译器会有警告提醒。

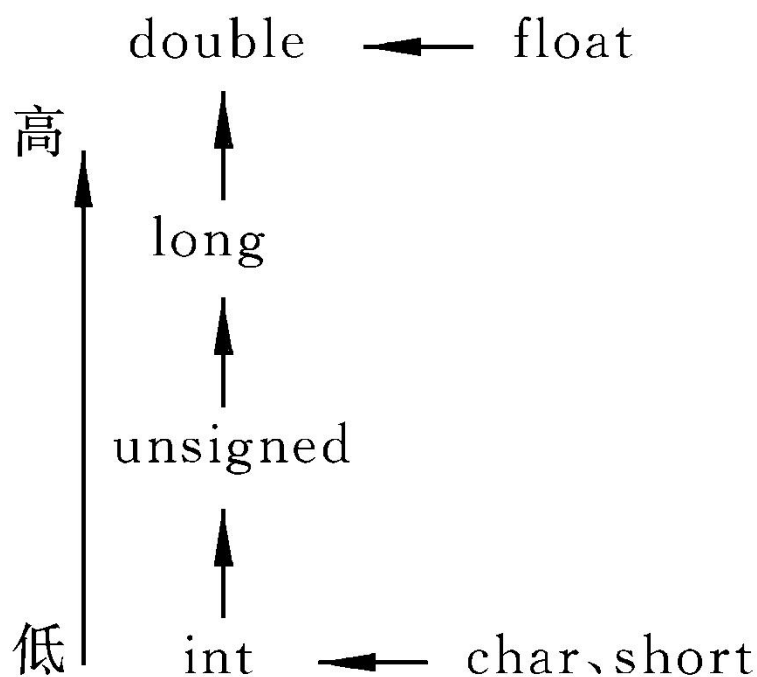


图 2.8-1

2.8.1 数值按 int 运算

C的整型算术运算总是至少以缺省整型类型的精度来进行的。为了获得这个精度，表达式中的字符型和短整型操作数在使用之前被转换为普通整型，这种转换称为整型提升（integral Promotion）。例如，在下面表达式的求值中

```
char a, b, c;
a=b+c;
```

b和c的值被提升为普通整型，然后再执行加法运算。加法运算的结果将被截短，然后再存储于a中，这个例子的结果和使用8位算术的结果是一样的。但在下面【例2.8.1-1】例子中，它的结果就不再相同。

【例2.8.1-1】整型常量默认按int类型运算实例

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

int main()
{
    char b=0x93<<1>>1;
    printf("%x\n",b);//ffffff93
    b=0x93<<1;
    b=b>>1;
    printf("%x\n",b);//13
    system("pause");
}
```

运行结果如下图 2.8.2-1:

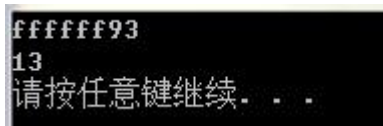


图2.8.2-1

为什么采用 16 进制打印，第一需要让大家了解输出时%x 是取 4 个字节进行输出的，那 b 中存的只有 93，为什么前面却打出了 3 个字节的 ff 呢，也就是 6 个 ff，如果用%d 输出，可以得到一个负值，当我们用%x 输出一个小于 4 个字节的数，前面补的字节是按照对应数据的最高位来看的，因为字符 b 的最高位为 1，所以其他 3 个字节补的都是 1。

为什么把操作分成两步后，b 的值就为 13 呢，因为 0x93 左移一位时，虽然按 4 字节进行，但是最低一个字节值为 0x26，赋值给 b 后，b 内存的就是 0x26，这时再对 b 进行右移时，单个字节拿到寄存器运算是按 4 个字节，但是因为 b 的最高位为零，因此拿到寄存器按 4 个字节运算，前面都是补零，再右移一位是除 2，因此得到的值是 13。

另外一种场景是我们将两个整型常量做乘法，赋值给一个长整型时，对于编译器来讲是按照 int 类型进行的，例子如下，打印结果为零，无论是在 VS 中新建 win32 控制台应用程序，在 32 位下执行，还是在 Linux 下将其编译为 64 位的可执行程序，执行结果均为 0，那怎么解决这种类型问题呢？

【例 2.8.1-2】两个较大的常量相乘溢出实例

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

int main()
{
    long l;
    l=131072*131072;
    printf("%ld\n", l);
    system("pause");
}
```

我们可以在做乘法前，将整型数强转为 long 即可，32 位控制台应用程序代码如【例 2.8.1-3】，因为 32 位下，long long 才是 8 个字节

【例 2.8.1-3】32 位程序两个较大的常量相乘不会溢出实例

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

int main()
{
    long long l;
    l=(long long)131072*131072;
    printf("%lld\n", l);
    system("pause");
}
```

上面代码输出结果如图 2.8.2-2，如果是 64 位程序，如【例 2.8.1-4】，转化为 long 即可。

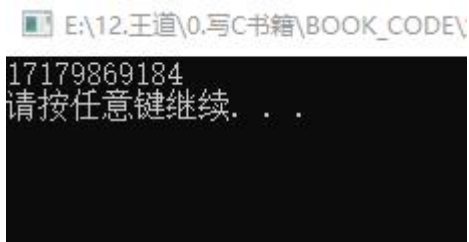


图2.8.2-2

【例 2.8.1-4】64 位程序两个较大的常量相乘不会溢出实例

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

int main()
{
    long l;
    l=(long)131072*131072;
    printf("%ld\n", l);
    system("pause");
}
```

2.8.2 浮点常量默认按 double 类型运算

浮点常量在运算时默认按 8 个字节，请看【例 2.8.2-1】：

【例 2.8.2-1】浮点常量运算实例

```
#include <stdio.h>
#include <stdlib.h>
//浮点常量默认按8个字节运算
int main()
{
    float f=12345678900.0+1;
    double d=f;
    printf("%f\n", f);
    printf("%f\n", 12345678900.0+1);
    system("pause");
    return 0;
}
```

执行结果如图2.8.2-3：

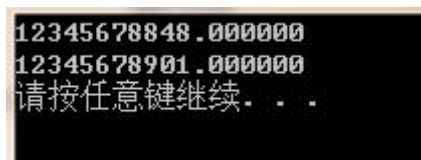


图2.8.2-3

第一个打印只有7位的精度是因为单精度浮点数f只有4个字节的存储空间，能够表示的精度是6-7位，所以只保证1到7是正确的，后面的都是近视。而第二个打印出正确的值是浮点型常量是按8个字节，也就是double类型进行运算的，同时%f会访问寄存器8个字节的空

进行浮点运算，因此可以正常输出。

思考题：

在代码中我们定义了 `double d=f`，请问 `d` 输出会是 12345678901 吗？

2.8.3 类型强转场景

对于整型数进行除法运算时，如果除后为小数，存入浮点数，一定要进行强制类型转换，否则如图 2.8.3-1，`f` 得到的值 2，`g` 得到的值为 2.5

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //混合运算，从长字节到短字节数据需要强制类型转换
5 int main()
6 {
7     int i=5;
8     float f,g;
9     short m;
10    long l=5;
11    f=i/2; //因为没有强制类型转换 i/2 表达式为整型运算
12    g=(float) i/2;
13    m=(short) l+2; //不加 short 强转编译会警告
14    printf("i=%d, f=%f, g=%f, m=%d\n", i, f, g, m);
15    system("pause");
16    return 0;
17 }
```

名称	值
f	2.00000000
g	2.50000000

图 2.8.3-1

图 2.8.3-1 编译结果如 2.8.3-2 所示，11 行的 `warning` 是因为将一个整型表达式赋值给浮点数，有丢失精度的可能，所以警告，这个是我们故意这么做。13 行的 `warning` 是因为我们没有把变量 `l` 强制转换为 `short`，再做加法，所以有警告，如果增加了 `short` 强转，就可以去除 13 行的警告。

```
5 int main()
6 {
7     int i=5;
8     float f,g;
9     short m;
10    long l=5;
11    f=i/2;//因为没有强制类型转换i/2表达式为整型运算
12    g=(float)i/2;
13    m=l+2;//不加short强转编译会警告
14    printf("i=%d, f=%f, g=%f, m=%d\n", i, f, g, m);
15    system("pause");
```

8. 混合运算, 配置: Debug Win32 -----

达式\8. 混合运算\main.c(11): warning C4244: “=”: 从“int”转换到“float”, 可能丢失数据
达式\8. 混合运算\main.c(13): warning C4244: “=”: 从“long”转换到“short”, 可能丢失数据

图 2.8.3-2

代码执行结果如图 2.8.3-3:

```
E:\BOOK\BOOK\数据类型-运算符-表达式\Debug\8.混合运算.exe
i=5,f=2.000000,g=2.500000,m=?
请按任意键继续. . .
```

图 2.8.3-3

思考题:

- 1、假定你有一个程序，它把一个 long 整型变量赋值给一个 short 整型变量。当你编译程序时会发生什么情况？当你运行程序时会发生什么情况？你认为其他编译器的结果是否也是如此？
- 2、假定你有一个程序，它把一个 double 类型变量赋值给一个 float 变量。当你编译程序时会发生什么情况？当你运行程序时会发生什么情况？

2.9 常用的数据输入/输出函数

如图 2.9-1 所示，我们可以给我们的程序输入数据，然后程序处理后，会给我们一个输出，C 语言通过函数库，读取标准输入，然后通过对应函数结果打印到屏幕上，前面我们学习了 printf 函数，理解通过 printf 函数可以将结果输出到控制台窗口。下面我们将详细讲解标准输入读取接口 scanf, getchar, 以及打印到屏幕上的标准输出接口 printf, putchar。



图 2.9-1

2.9.1 scanf 原理

C 语言没有提供输入输出关键字，C 语言的输入和输出通过标准函数库来实现，我们通过 `scanf` 函数读取键盘输入，我们把键盘输入又成为标准输入，当 `scanf` 读取标准输入时，如果我们还没有输入任何内容，那么 `scanf` 会卡主（专业用语为阻塞），下面我们首先来看一个例子（如图 2.9.1-1）

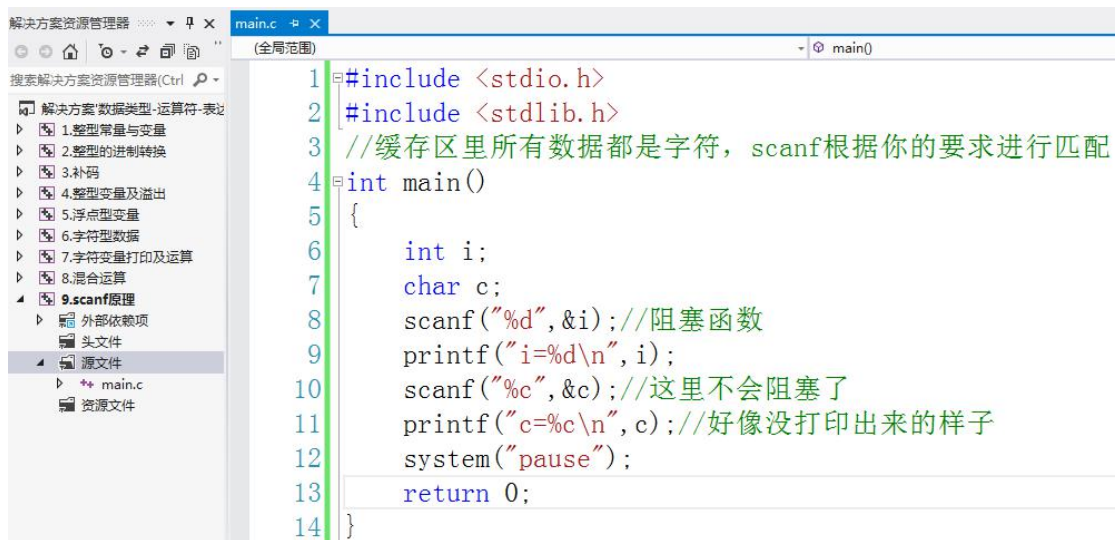


图 2.9.1-1

执行我们输入 20，然后回车，显示结果如图 2.9.1-2，为什么第二个 `scanf` 不会阻塞呢，其实是因为第二个 `scanf` 读取了缓冲区里的 `\n`，被 `scanf(\"%c\", &c)` 读取，打印其实输出了换行，所以不会阻塞。



图 2.9.1-2

接下来我们来学习一下缓冲区原理，缓冲区其实就是一段内存空间，分为读缓冲，写缓冲，接下来我们来看下 C 的缓冲三种特性：

- 1) **全缓冲**：在这种情况下，当填满标准 I/O 缓存后才进行实际 I/O 操作。全缓冲的典型代表是对磁盘文件的读写。
- 2) **行缓冲**：在这种情况下，当在输入和输出中遇到换行符时，执行真正的 I/O 操作。这时，我们输入的字符先存放在缓冲区，等按下回车键换行时才进行实际的 I/O 操作。典型代表是标准输入(`stdin`)和标准输出(`stdout`)。
- 3) **不带缓冲**：也就是不进行缓冲，标准出错情况 `stderr` 是典型代表，这使得出错信息可以直接尽快地显示出来。

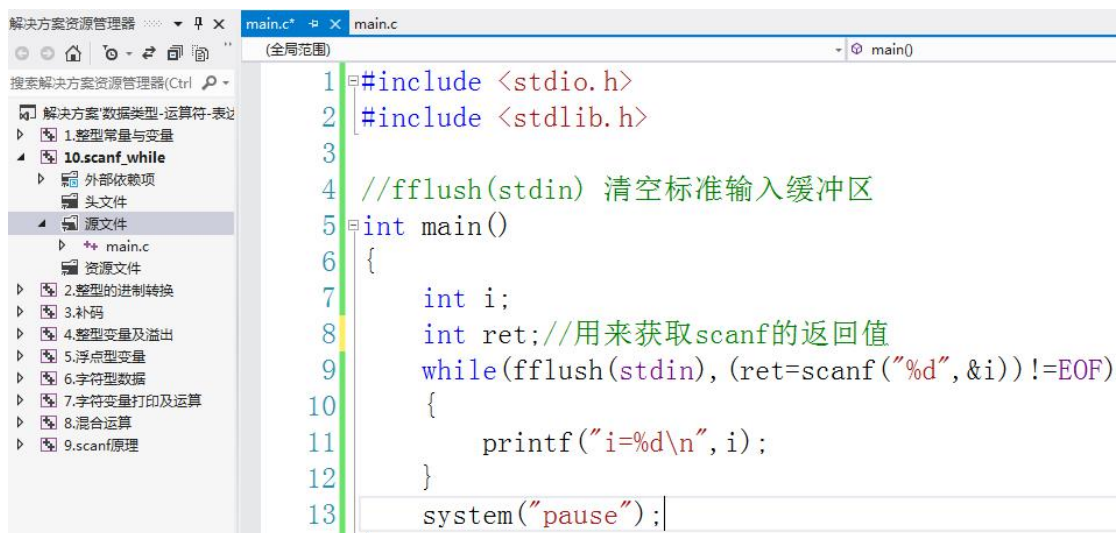
ANSI C (C89) 要求缓存具有下列特征：

- 1) 当且仅当标准输入和标准输出并不涉及交互设备时，它们才是全缓存的。
- 2) 标准出错决不会是全缓存的。

图 2.9.1-1 的例子，我们往标准输入缓冲中放入的字符为 20\n，当我们输入了\n（回车）后，scanf 才开始匹配，scanf 的%d 匹配整型数 20，然后放入变量 i 中，我们进行打印输出，这时候\n，仍然在标准输入缓冲区（stdin）内，如果第二个 scanf 为 scanf("%d",&i)，那么依然会发生阻塞，因为 scanf 在读取整型数，浮点数，字符串（到后面数组讲解字符串）时，会忽略\n（回车），空格等字符（所谓的忽略就是 scanf 执行时会首先删除这些字符然后再阻塞）。scanf 匹配一个字符，就会在缓冲区删除对应字符。因为 scanf("%c",&c)时，不会忽略任何字符，所以 scanf("%c",&c)读取了还在缓冲区中残留的\n。

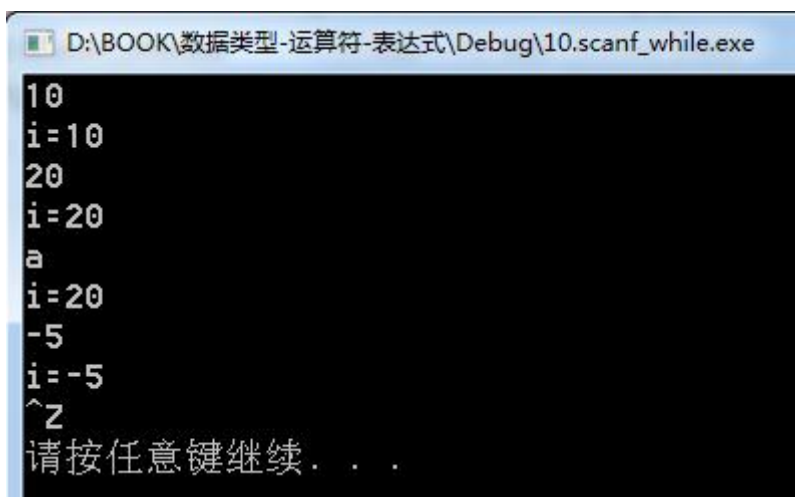
2.9.2 scanf 循环读取

如图 2.9.2-1，如果我们想输入多个整数（每次输入都回车），让 scanf 读取后，并打印输出，那么我们需要一个 while 循环（如果对 while 循环完全没概念，跳过本节，学完第三章以后，再来看本节），为什么要加入 fflush(stdin)呢，因为 fflush 具有刷新（清空）标准输入缓冲区的作用，如果我们输错了，输入的为字符，scanf 无法匹配成功，scanf 没有匹配成功其返回值为 0，也就是 ret 的值为 0，但是并不等于 EOF，因为 EOF 的值为-1，仍然会进入循环，就会造成不断的打印，而我们的实际执行结果如图 2.9.2-2，最后我们输入 ctrl+z，让 scanf 匹配失败，循环结束。各位小伙伴可以自行尝试，去除 fflush(stdin)，然后输入 a，看效果。



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //fflush(stdin) 清空标准输入缓冲区
5 int main()
6 {
7     int i;
8     int ret;//用来获取scanf的返回值
9     while(fflush(stdin), (ret=scanf("%d",&i))!=EOF)
10    {
11        printf("i=%d\n", i);
12    }
13    system("pause");
```

图 2.9.2-1



```
D:\BOOK\数据类型-运算符-表达式\Debug\10.scanf_while.exe
10
i=10
20
i=20
a
i=20
-5
i=-5
^Z
请按任意键继续...
```

图 2.9.2-2

针对 VS2013 到 VS2017 版本，需要按多次 ctrl+z 来结束 while 循环，也就是需要按 3 次 ctrl+z 才能让 scanf 出错返回-1。

接下来我们看一个读取字符串并打印对应字符串的大写字母的例子（如图 2.9.2-3），原理是我们让 scanf 每次读取一个字符，并打印，由于我们一次性输入一个字符串，然后回车，scanf 是循环匹配，所以不能加 fflush(stdin)，加了以后就会导致第一个字符匹配以后，后面的字符被清空。各位小伙伴可以自行尝试一下。

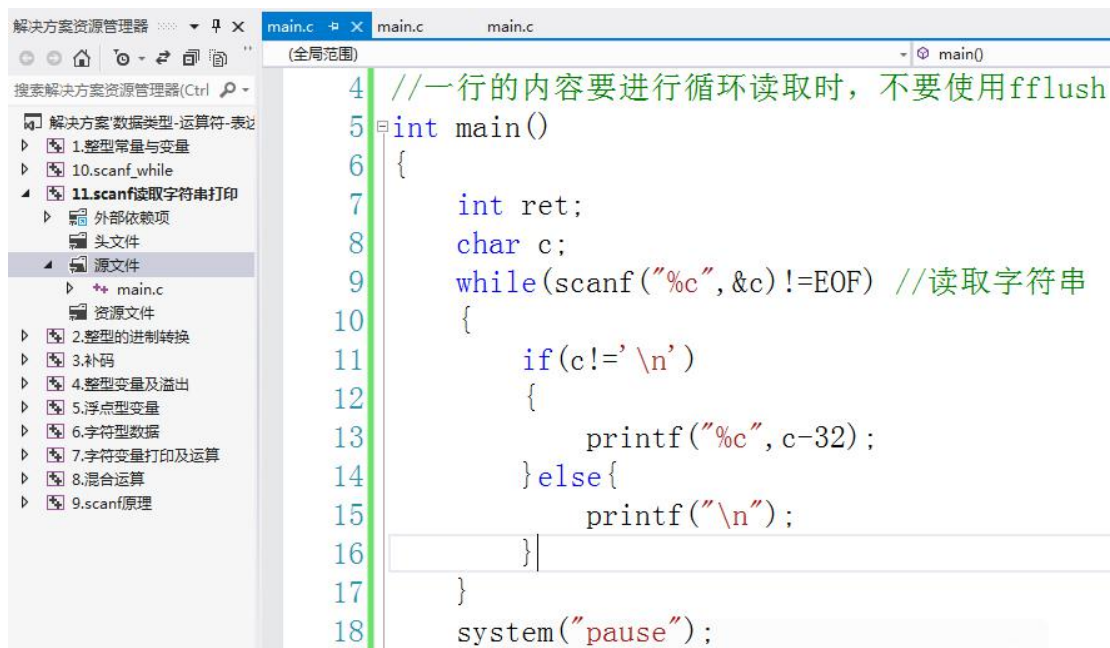


图 2.9.2-3

图 2.9.2-3 执行结果如图 2.9.2-4：

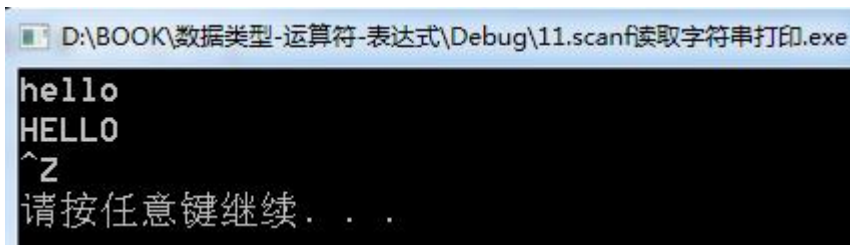


图 2.9.2-4 执行结果

2.9.3 多种数据类型混合输入

当我们让 scanf 一次读取多种类型数据时，对于字符型要格外小心，因为当一行数据中存在字符型数据读取时，读取字符并不会忽略空格，\n(回车)，所以使用方法如图 2.9.3-1，编写代码时，我们需要将%d，与%c之间加入一个空格。输入格式，输出效果如图 2.9.3-2 所示，scanf 匹配成功了 4 个成员，所以返回值为 4，我们可以通过返回值，判断 scanf 匹配成功了几个，中间任何有一个成员匹配出错，那么后面的成员都会匹配出错。

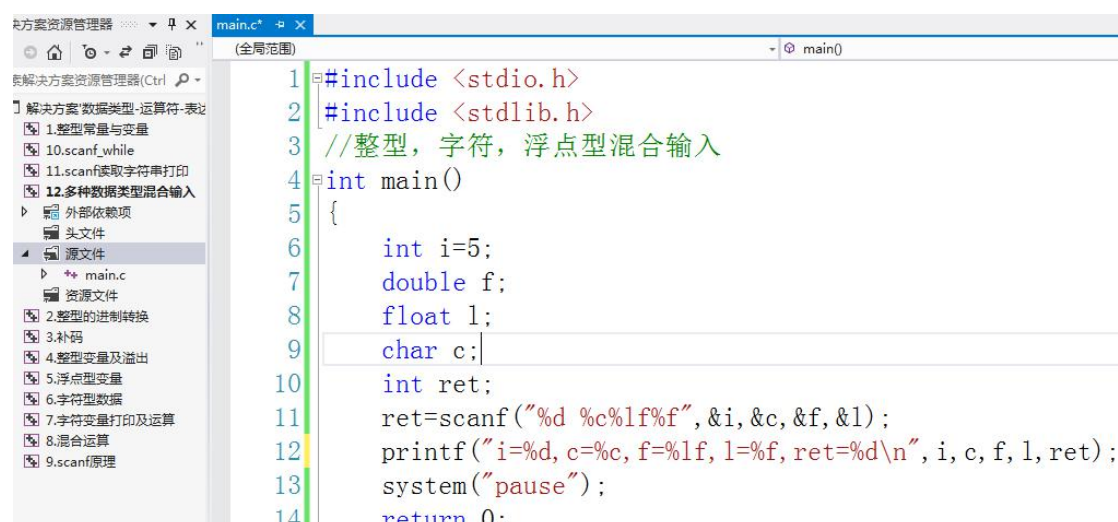


图 2.9.3-1

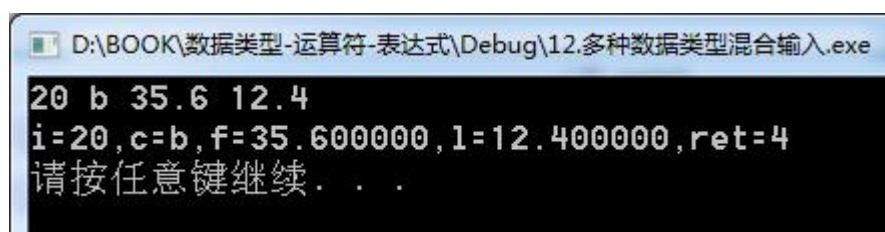


图 2.9.3-2

图 2.9.3-2 可以看到浮点数输出默认带了 6 位小数，看起来非常不美观，修改代码如【例 2.9.3-1】：

【例 2.9.3-1】scanf 读取混合类型输入

```
#include <stdio.h>
#include <stdlib.h>
//整型, 字符, 浮点型混合输入
int main()
{
    int i=5;
    double f;
    float l;
    char c;
    int ret;
    ret=scanf("%d %c%lf%f",&i,&c,&f,&l);
    printf("i=%d, c=%c, f=%5.2lf, l=%5.2f, ret=%d\n", i, c, f, l, ret);
    system("pause");
    return 0;
}
```

5.2 代表总结占用 5 个空格的位置, 2 代表小数点后显示两位, 这时打印输出结果如图 2.9.3-3:

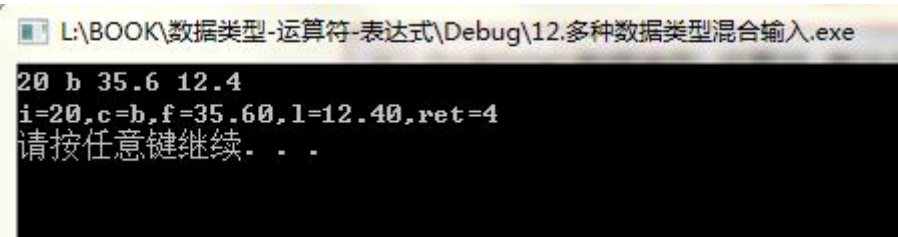


图 2.9.3-3

表 2.9.3-1 是常见的各种数据变量的 scanf 的格式符

数据类型	格式符	使用方法
int	%d	scanf(“%d”,&i)
long	%ld	scanf(“%ld”,&j)
long long	%lld	scanf(“%lld”,&k)
float	%f	scanf(“%f”,&f)
double	%lf	scanf(“%lf”,&d)
char	%c	scanf(“%c”,&c)
字符串(char str[20], 见第 4 章)	%s	scanf(“%s”,str)

表 2.9.3-1 常见数据变量的 scanf 格式符

2.9.4 getchar 讲解

我们通过 getchar 可以一次从标准输入读取一个字符，等价于 `char c,scanf(“%c”,&c)`，语法格式如下：

```
#include <stdio.h>
int getchar( void );
```

通过下面代码我们可以用 getchar 读取一个字符

【例 2.9.4-1】getchar 使用

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char c;
    c=getchar();
    printf("you input alphabet=%c\n",c);
    system("pause");
}
```

getchar 每次只能读取一个字符。【例 2.9.4-1】执行效果如图 2.9.4-1：



图 2.9.4-1

思考题：

在上面代码中的 printf 之后，再加一个 getchar，请问 getchar 是否会阻塞？

2.9.5 putchar 讲解

输出字符数据使用 `putchar` 函数，作用是向显示设备输出一个字符，语法格式如下：

```
#include <stdio.h>
int putchar( int ch );
```

参数 `ch` 为要输出的字符，可以是字符变量，可以是整型变量，也可以是常量。输出字符 `H` 的代码如下：

```
putchar( 'H' );
```

下面代码通过 `putchar` 实现，变量，转义字符的打印：

【例 2.9.5-1】`putchar` 使用

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char a,b,c;
    a='a';
    b='b';
    c='c';
    putchar(a);
    putchar('\b');//输出转义字符
    putchar(b);
    putchar(c);
    putchar('\n');//输出转义字符
    system("pause");
}
```

输出结果如图 2.9.5-1：

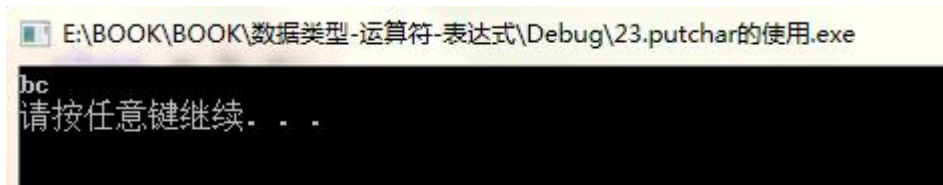


图 2.9.5-1

思考题：

如果想输出结果为 `ac`，而不是 `bc`，如果改动？

2.9.6 printf 讲解

`printf` 函数可以输出各种类型数据，整型，浮点型，字符型，字符串等，实际原理是 `printf` 将这些类型的数据格式化为字符串后，放入标准输出缓冲区，然后通过 `\n`，刷新标准输出，推到屏幕上。

语法如下：

```
#include <stdio.h>
int printf( const char *format, ... );
```

`printf()` 函数根据 *format* (格式) 给出的格式打印输出到 STDOUT (标准输出) 和其它参数中.

字符串 *format* (格式) 由两类项目组成 - 显示到屏幕上的字符和定义 `printf()` 显示的其它参数. 基本上, 你可以指定一个包含文本在内的 *format* (格式) 字符串, 也可以是映射到 `printf()` 其它参数的“特殊”字符. 例如本代码

```
int age = 21;
printf( "Hello %s, you are %d years old\n", "Bob", age );
```

显示下列输出:

```
Hello Bob, you are 21 years old
```

`%s` 表示, “在这里插入首个参数, 一个字符串.” `%d` 表示第二个参数 (一个整数) 应该放置在那里. 不同的“*%-codes*”表示不同的变量类型, 也可以限制变量的长度.

Code	格式
<code>%c</code>	字符
<code>%d</code>	带符号整数
<code>%i</code>	带符号整数
<code>%e</code>	科学计数法, 使用小写“e”
<code>%E</code>	科学计数法, 使用大写“E”
<code>%f</code>	浮点数
<code>%o</code>	八进制
<code>%s</code>	一串字符
<code>%u</code>	无符号整数
<code>%x</code>	无符号十六进制数, 用小写字母
<code>%X</code>	无符号十六进制数, 用大写字母
<code>%p</code>	一个指针
<code>%%</code>	一个“%”符号

表 2.9.6-1 `printf` 的 Code 格式

一个位于一个%和格式化命令间的整数担当着一个最小字段宽度说明符, 并且加上足够的空格或0使输出足够长. 如果你想填充0, 在最小字段宽度说明符前放置0. 你可以使用一个精度修饰符, 它可以根据使用的格式代码而有不同的含义.

- 用`%e`, `%E` 和 `%f`, 精度修饰符让你指定想要的小数位数. 例如,

```
%5.2f
```

将会至少显示 5 位数字, 并带有 2 位小数的浮点数.

- 用 %s, 精度修饰符简单的表示一个**最大**的最大长度, 以补充句点前的最小字段长度.

所有的 printf() 的输出都是右对齐的, 除非你在 % 符号后放置了负号. 例如,

`%-5.2f`

将会显示 5 位字符, 2 位小数位的浮点数并且**左对齐**

请看【例 2.9.6-1】:

【例 2.9.6-1】printf 输出对齐

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i=10;
    float f=96.3;
    printf("student number=%3d score=%5.2f\n", i, f);
    printf("student number=%-3d score=%5.2f\n", i, f);
    printf("%10s\n", "hello");
    system("pause");
}
```

运行结果如图 2.9.6-1, 可以看到整型数 10, 不加负号靠右对齐, 加负号靠左对齐, 针对 %10s 代表字符串总计占用 10 个字符的位置, 因为默认靠右对齐, 所以 hello 字符串相对于左边起始位置有 5 个空格的距离, 掌握这些对于后面做项目学生管理系统, 对于打印格式的控制就会手到擒来。

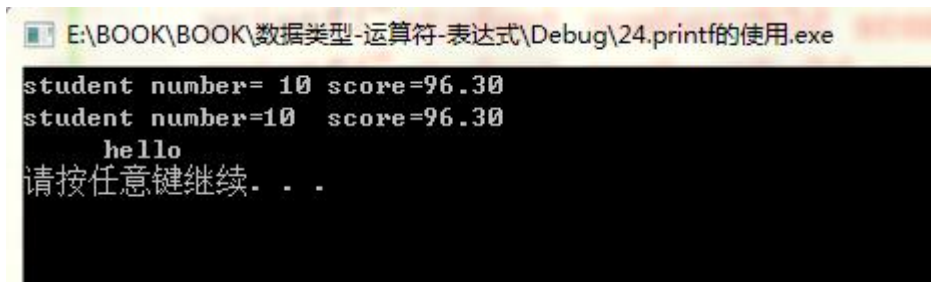


图 2.9.6-1

2.10 运算符与表达式

2.10.1 运算符分类

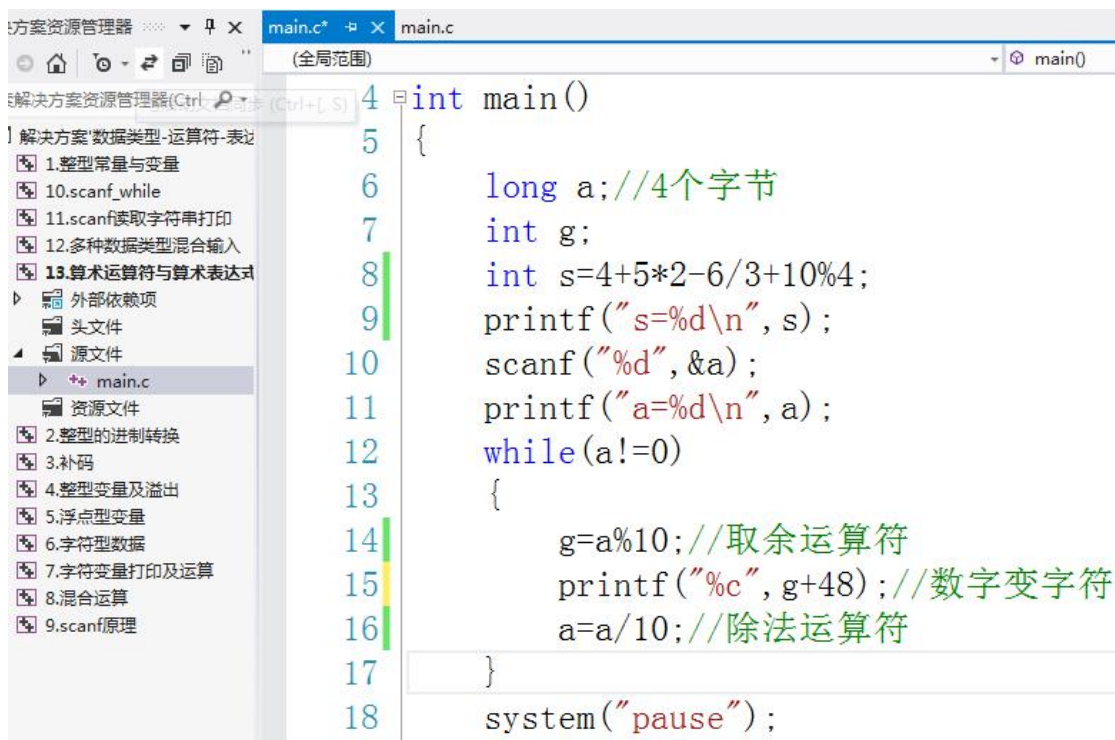
C 语言提供了 13 种类型的运算符, 如下:

- (1) 算术运算符 (+ - * / %)
- (2) 关系运算符 (> < == >= <= !=)
- (3) 逻辑运算符 (! && ||)
- (4) 位运算符 (<< >> ~ | ^ &)
- (5) 赋值运算符 (= 及其扩展赋值运算符)

- (6)条件运算符 (?:)
- (7)逗号运算符 (,)
- (8)指针运算符 (*和&)
- (9)求字节数运算符 (sizeof)
- (10)强制类型转换运算符 ((类型)) //在混合运算已经讲解
- (11)分量运算符 (. ->)
- (12)下标运算符 ([])
- (13)其他 (如函数调用运算符())

2.10.2 算术运算符及算术表达式

算术运算符包含+ - * / %, 当一个表达式同时出现这 5 种运算符时, 先进行乘 (*), 除 (/), 取余 (%), 取余也叫取模, 后进行加 (+), 减 (-), 也就是乘除取余的运算符的优先级高于加减运算符。除了%运算符, 其余几个运算符即适用于浮点数类型又适用于整型类型, 当操作符的两个操作数都是整型数时, 它执行整除运算, 在其他情况下执行浮点数除法。%为取模操作符, 它接受两个整型操作数, 把左操作数除以右操作数, 但它的值返回的是余数而不是商。在 VS 下, 如果运算除零, 或者模零, 都会造成编译不通, 如果考研机试编译器是 GCC, 会编译警告, 同时执行时程序会崩溃。例子 (如图 2.10.2-1) 除了变量 s 用于大家理解算术运算符优先级之外, 其他的为一道华为面试题, 输入一个整数, 然后逆序将其输出。由算术运算符组成的式子我们称为算术表达式。表达式一定有一个值。



```
1 int main()
2 {
3     long a; //4个字节
4     int g;
5     int s=4+5*2-6/3+10%4;
6     printf("s=%d\n", s);
7     scanf("%d", &a);
8     printf("a=%d\n", a);
9     while(a!=0)
10    {
11        g=a%10; //取余运算符
12        printf("%c", g+48); //数字变字符
13        a=a/10; //除法运算符
14    }
15    system("pause");
16 }
```

图 2.10.2-1

图 2.10.2-1 的执行结果如图 2.10.2-2:

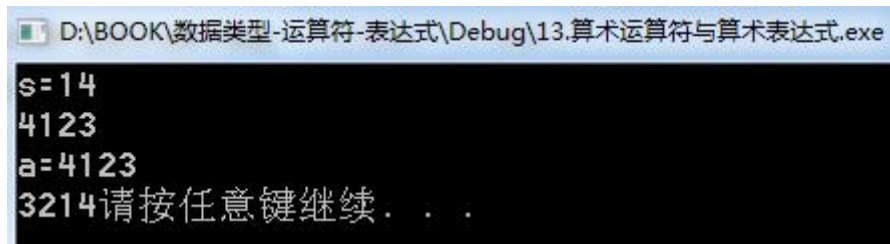


图 2.10.2-2

思考题 1:

下面我们出一道思考题，我们有两个整型变量 **a** 与 **b**，假如不使用第三个变量，交换变量 **a** 和 **b** 的值，如何做？

思考题 2:

因为我们的 CPU 做浮点运算能力不够，王者荣耀中，不再做小数运算，全部用分数代替，如何比较两个分数的大小（不能用除法，除成小数比较）？

思考题 3:

如果输入一个整数，判断该整数是不是对称数，如何做，比如 12321，是对称数，123321 也是对称数，但是 456 就不是对称数。

2.10.3 关系运算符与关系表达式

关系运算符 (**>** **<** **==** **>=** **<=** **!=**) 依次为大于，小于，是否等于，大于等于，小于等于，不等于六个运算符，由关系运算符组成的表达式，我们称为**关系表达式**，关系表达式的最终值，只有真和假，**对应的值为 1 和 0**，因为 C 语言是没有布尔类型的，在 C 语言中，**0 值代表假，非 0 即为真**。例如 **3>4**，这个关系表达式不成立，那么为假，那么其整体的值为 0，而 **5>2** 这个关系表达式成立，所以为真，其整体的值为 1。**关系运算符的优先级低于算术运算符**，运算符的详细的优先级情况见附录 2。实例（如图 2.10.3-1）演示了如何比较一个浮点数是否等于某个值的方法，因为关系运算符优先级低，所以 **f-234.56** 不用加括号，因为浮点数中存的是对应数的近似值，只保证精度为 7 位，有兴趣的小伙伴，可以注释掉下面一句，打开上面一句，会发现不相等。

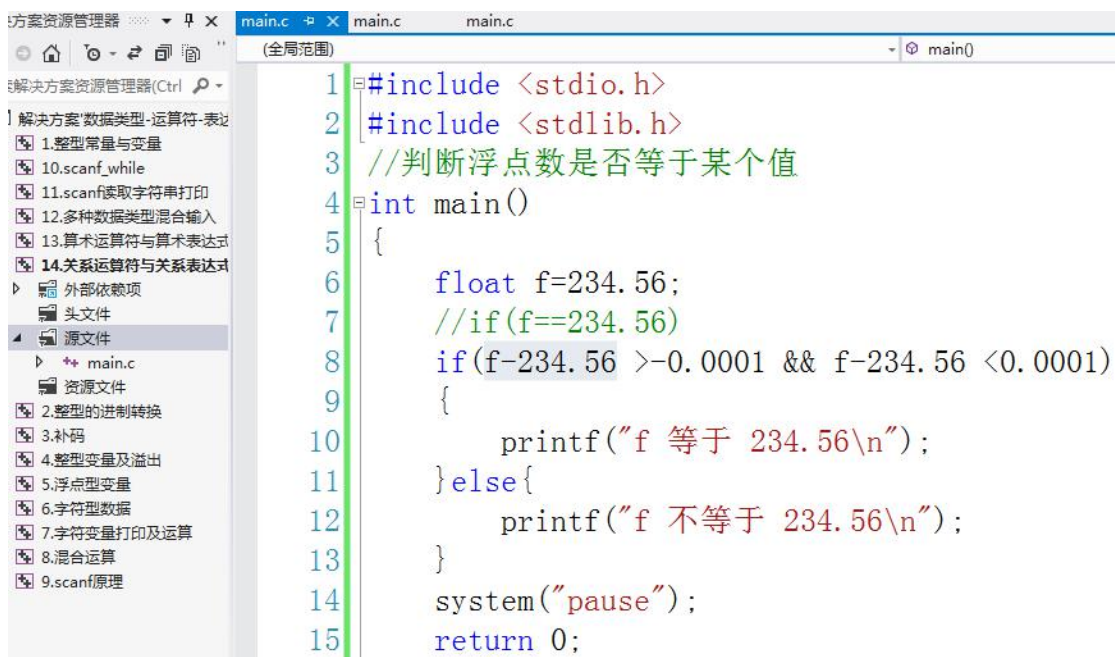


图 2.10.3-1

图 2.10.3-1 的程序执行结果如图 2.10.3-2:

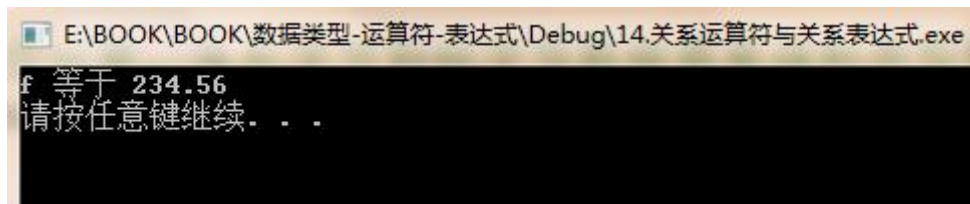


图 2.10.3-2

在工作中，因为很多程序员容易将两个等号，写成一个等号，所以当我们判断整型变量 i 是否等于 3 时，我们要这样写 $3==i$ ，把常量写在前面，这是为了防止不小心将两个等号写为一个等号时，变量在前面就会编译不通，从而快速发现错误。（这种写法属于华为内的一条编程规范）

同时编写程序时，当我们判断三个数是否相等时，绝对不可以写 $\text{if}(5==5==5)$ ，这种无论如何都是为假，为什么，因为首先 $5==5$ 得到的结果为 1，然后 $1==5$ 得到的结果为 0，因为毕竟 3 个变量 a, b, c 是否相等，不能写 $a==b==c$ ，而应该写成 $a==b \ \&\& \ b==c$ 来判断三个数是否相等。

【例 2.10.3-1】关系运算符的使用

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

```
#include <stdlib.h>
```

```
//不能用数学上的连续判断大小来判断某个数
```

```
int main()
{
    int a;
    while(scanf("%d", &a) != EOF)
    {
        if(3 < a < 10)
        {
            printf("a在3和10之间\n");
        } else {
            printf("a 不在3和10之间\n");
        }
    }
    system("pause");
}
```

上面代码执行结果如图 2.10.3-3:

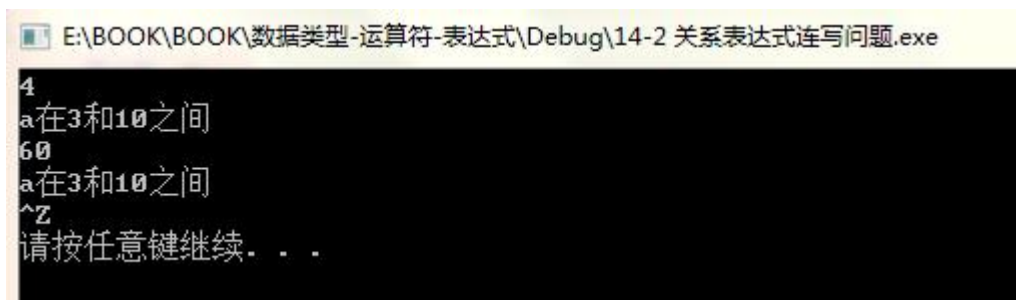


图 2.10.3-3

如【例 2. 10. 3-1】所示，如果我们想判断变量 a 大于 3，同时小于 10，不能写为 $3 < a < 10$ ，在数学上的确是可以这么写的，但是编程中，是不可以的，首先无论 a 是大于 3，还是小于 3，对于 $3 < a$ 这个表达式只有 1 或者 0，两个值，然后 1 和 0 都是小于 10 的，无论 a 值为多少，所以表达式本身始终为真，因此我们想判断变量 a 小于 3，同时大于 10 时，要写成 $a > 3 \ \&\& \ a < 10$ ，这样才是正确的写法。

思考题：

爱动手的同学把及时修改一下上面的代码，让输出的 a 为 60 时，打印出 a 不在 3 和 10 之间

2.10.4 逻辑运算符与逻辑表达式

逻辑运算符（! && ||）依次为逻辑非，逻辑与，逻辑或，和大家学的数学上的与或非是一致的，逻辑非的优先级高于算术运算符，逻辑与和逻辑或的优先级低于关系运算符。逻辑表达式的最终值，只有真和假，对应的值为 1 和 0，如何使用见表 2.10.4-1。

运算符	含义	语法	返回值
&&	与	$a \&\& b$	a 和 b 都为真，返回真，其他情况都为假
	或	$a b$	a 和 b 都为假（即 a 和 b 的值都是零），则返回假，其他情况均为真
!	非	$!a$	如果 a 为真，则返回假 如果 a 为假，则返回真

表 2.10.4-1

下面的代码【例 2. 10. 4-1】为如何计算一年是否为闰年的例子，因为重复测试，所以我们用了一个循环。在 while 循环后，我们演示了什么是短路运算， $j=1$ 时，判断 $j==0$ 表达式为假，中间为与，无论后面真假，整体都为假，所以后面的 printf 函数表达式不会再执行，因此看不到打印输出， $j=0$ 时， $j==0$ 为真，所以后面的 printf 得不到打印，工作中我们经常用短路运算避免使用 if 判断，降低代码量。执行结果如图 2.10.4-1 所示。

针对代码【例 2. 10. 4-1】中的逻辑非，首先给 j 赋值为 10，因为 j 是非 0，所以 $!j$ 为 0，然后逻辑非是单目运算符，从右至左，所以 $!j$ 得到的值为 1，因为对 0 取非，得到的值为 1，对非 0 值取非，得到的值为 0。

【例 2. 10. 4-1】逻辑运算符的使用

```
#include <stdio.h>
#include <stdlib.h>
//逻辑与，逻辑或的短路运算
int main()
{
    int i=0, j;
    while(scanf("%d", &i) != EOF)
    {
        if(i%4==0 && i%100!=0 || i%400==0)
        {
            printf("i is leap year\n");
        }else{
            printf("i is not leap year\n");
        }
    }
}
```

```

    }
}
j=1;//当j不为0时，j==0为假，后面一个表达式不会得到执行
j==0&&printf("system is error\n");
j=0;//当j为0时，j==0为真，后面一个表达式不会得到执行
j==0||printf("system is error\n");
j=10;//下面我们来看一下逻辑非运算符
i=!j;
printf("i的值=%d\n",i);
system("pause");
return 0;
}

```

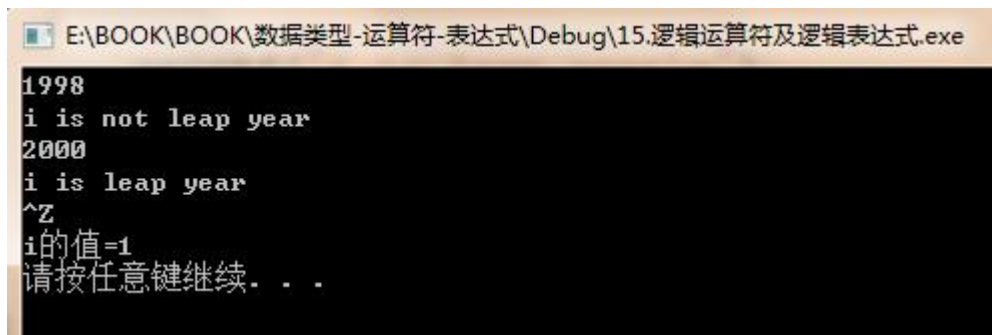


图 2.10.4-1

思考题：

如果想看到逻辑与和逻辑或后面的 system is error 打印，如果修改？

2.10.5 位运算符

位运算符（<<>>~|&）包含左移，右移，按位取反，按位或，按位异或，按位与。

左移为高位丢弃，低位补 0，相当于乘 2，工作中很多时候申请内存会用左移，例如申请 1G 大小的空间，使用 malloc（1<<30），malloc 接口的使用我们后面会讲；**右移**为低位丢弃，正数（无符号数最高位无论是 0 或者 1，都认为是正数），高位补 0，负数，高位补 1，相当于除 2，移位比乘法和除法的效率要高，负数右移，对于偶数来说是除 2，但是对于奇数来说是先减 1，然后再除 2，例如-8>>1，得到的是-4，但是对于-7>>1 并不是-3，而是-4，另外对于-1 来说，无论右移多少位，值永远为-1；

异或 相同的数，异或为 0，任何数和 0 异或就为本身。

按位取反即对于的位是 1 的变为 0，位是 0 的变为 1，按位与和按位或，就是用两个数的每一位进行与和或。我们看图 2.10.5-1 的例子，执行结果如图 2.10.5-2，有兴趣的小伙伴可以自己改动一下。

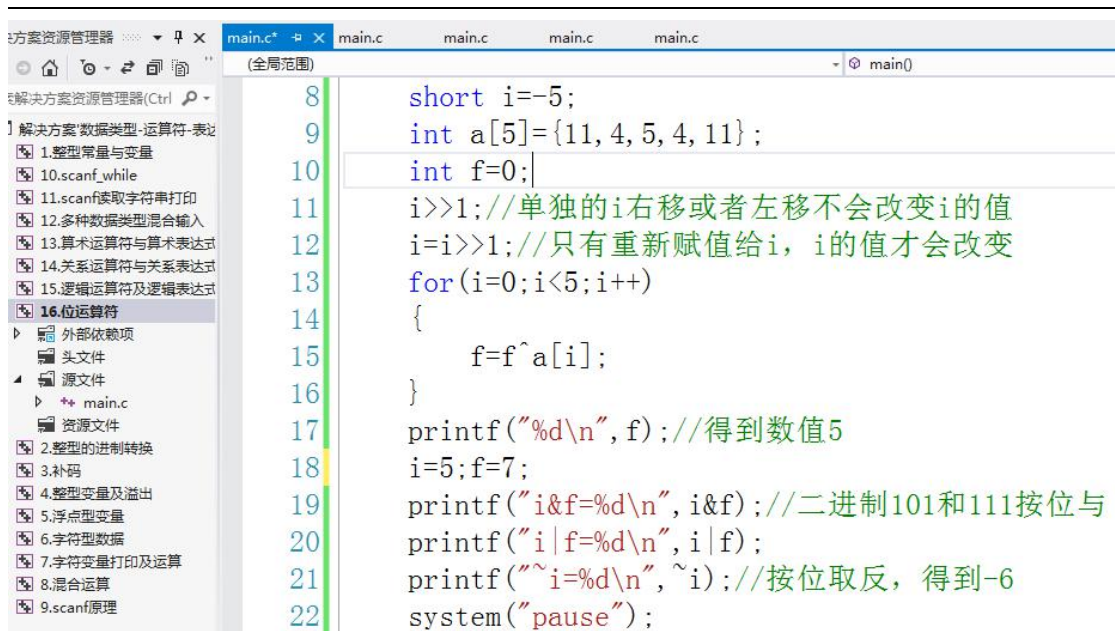


图 2.10.5-1

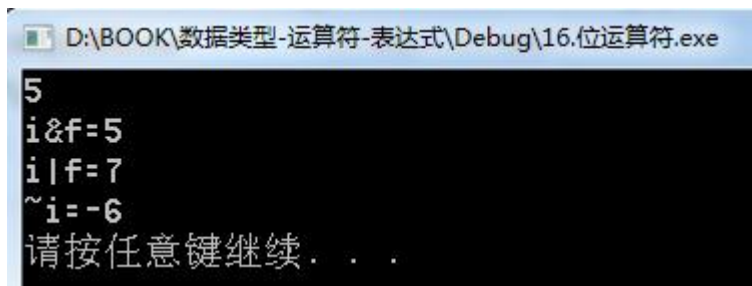


图 2.10.5-2

思考题 1:

我们有两个变量 **a** 与 **b**, 假如不使用第三个变量, 交换变量 **a** 和 **b** 的值, 通过异或操作来交换两个数, 这种交换相对于之前的加法交换有何优势?

思考题 2:

如何通过位运算找到一个数的最低位为 1 的那一位 (注意不可使用循环), 复杂度 $O(1)$

思考题 3:

C 语言没有提供循环移位的运算符, 想挑战的小伙伴可以自行实现一个能够循环移位的函数

2.10.6 赋值运算符

为了理解有些操作符存在的限制, 你必须理解左值(L-value)和右值(R-value)之间的区别。这两个术语是多年前由编译器设计者所创造并沿用至今, 尽管它们的定义并不与 C 语言严格吻合。

左值就是那些能够出现在赋值符号左边的东西。右值就是那些可以出现在赋值符号右边的东西。 这里有个例子:

```
a = b + 25;
```

a 是个左值, 因为它标识了一个可以存储结果值的地点, **b + 25** 是个右值, 因为它指定了一个值。

它们可以互换吗?

```
b + 25 = a;
```


原先用作左值的 `a` 此时也可以当作右值，因为每个位置都包含一个值。然而，`b+25` 不能作为左值，因为它并未标识一个特定的位置（并不对应特定的内存空间）。因此，这条赋值语句是非法的。

通过看附录 2，可以看到赋值运算符的优先级是非常低的，仅高于逗号运算符，因为如果我们想通过 `getchar` 循环读取并打印每一个字符，写法如【例 2.10.6-1】：

【例 2.10.6-1】赋值运算符使用

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char c;
    while((c=getchar())!=EOF)
    {
        printf("%c",c);
    }
    system("pause");
}
```

执行结果如图 2.10.6-1，我们输入 hello，得到输出结果 hello，然后按 `ctrl+z` 结束输入：

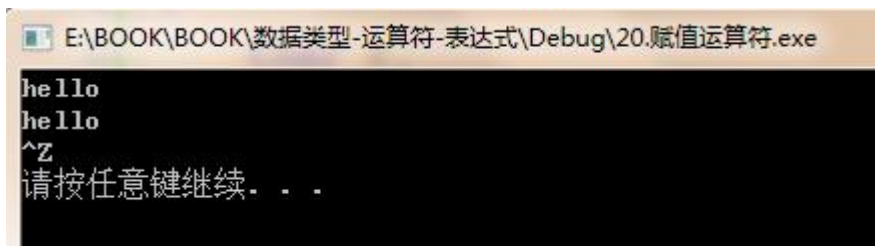


图 2.10.6-1

为什么需要对 `c=getchar()` 整体括起来再判断是否与 `EOF` 相等呢，因为赋值运算符的优先级小于关系运算符，如果不加括号，那么 `c` 的值只有 0 或者 1 两种情况。

思考题：

有兴趣的小伙伴可以把括号去掉，单步观察一下上面程序的打印输出。

接下来我们看一下复合赋值运算符，复合赋值运算符操作是一种缩写形式，使得对变量的赋值操作变的更加简洁。例如在程序中对变量赋值

```
iNum=iNum+5;
```

这个语句使对变量 `iNum` 的赋值操作，值为这个变量本身与一个整型变量 5 相加的结果值。使用复合语句可以实现同样的操作。例如，上面的语句可以修改为：

```
iNum+=5;
```

赋值运算符与复合赋值运算符的区别在于后者：

- 1、简化了程序，使程序精炼，阅读速度提升。
- 2、提高了编译效率

请看实例【例 2.10.6-1】：

【例 2.10.6-2】加后赋值与乘后赋值使用

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



```
{  
    int iNum, iResult;  
    iNum=10;  
    iResult=3;  
  
    iNum+=5;  
    iResult*=iNum;  
    printf("iNum=%d\n", iNum);  
    printf("iResult=%d\n", iResult);  
    system("pause");  
}
```

从上面程序代码可以看到，`iNum+=5`，代表 `iNum` 加 5 后再赋值给 `iNum`，因此最终 `iNum` 的值为 15，而 `iResult` 的值为其自身在乘以 `iNum` 的值，所以最终得到的结果为 45。程序运行结果如图 2.10.6-2：



图 2.10.6-2

2.10.7 条件运算符与逗号运算符

条件运算符 C 语言中唯一的三目运算符，三目运算符代表有三个操作数，双目运算符就是两个操作数，比如我们的逻辑与就是双目运算符，单目运算符就是一个操作数，比如逻辑非就是单目运算符，运算符我们也可以称为操作符。逗号运算符的优先级最低，我们需要掌握的是逗号表达式整体的值是最后一个表达式的值。下面我们来看一个实例(如图 2.10.6-1)，通过条件运算符我们可以快速得到 3 个数中间的最大值，避免了很多 if 判断，通过逗号运算符，我们可以先做一些准备操作，而最终 while 循环是否结束，依赖 `scanf("%d%d%d",&a,&b,&c)!=EOF` 这个关系表达式的真假。

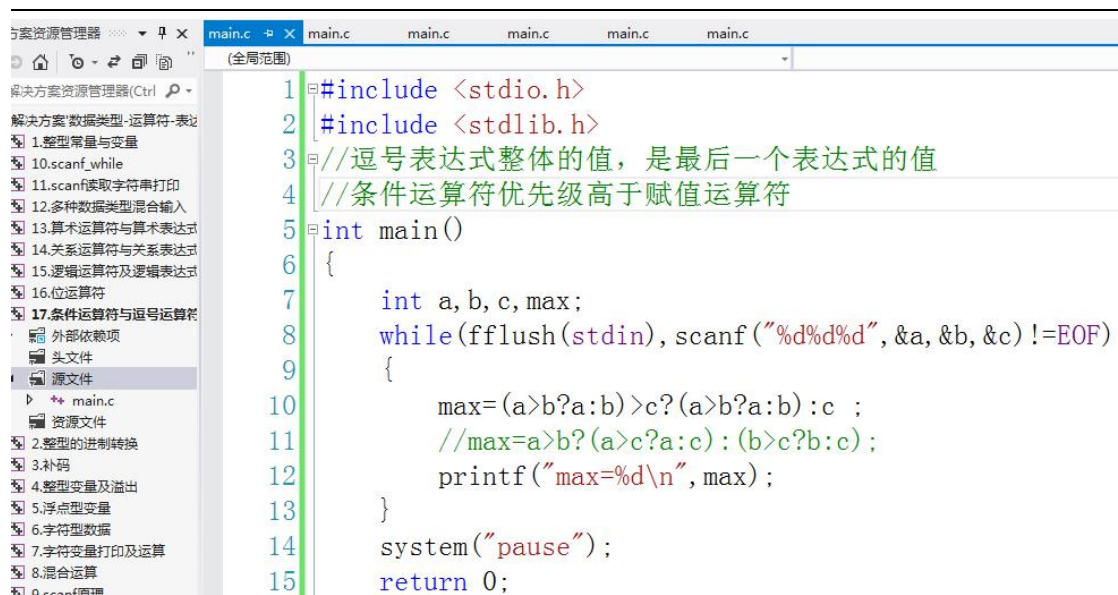


图 2.10.6-1

2.10.8 自增自减运算符及求字节运算符

自增自减运算符和其他运算符有很大的区别，因为其他运算符除了赋值运算符可以改变变量本身的值以外，其他的运算符不会有这种效果，自增自减就是对变量进行加 1 和减 1 操作，有加法和减法运算符，为什么还要发明这个呢，其实是因为自增和自减来源于 B 语言，当时汤姆逊和里奇（C 语言的发明者）为了不改变当时程序员的编写习惯，所以在 C 中保留了下来。因为自增自减会改变变量的值，所以自增和自减是不能应用于常量的！

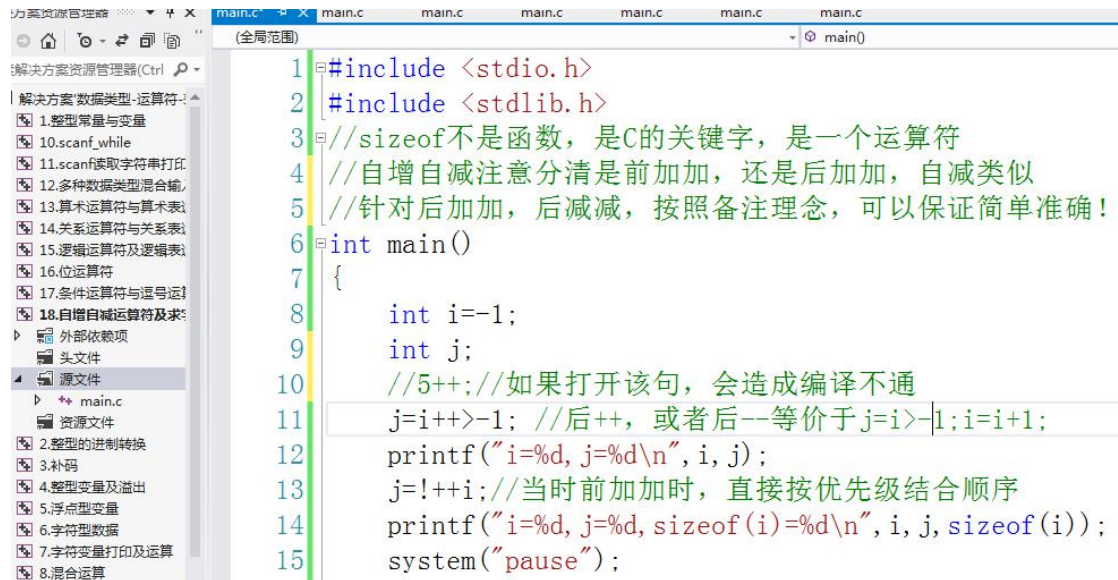
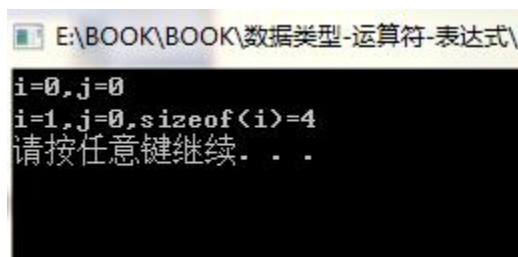


图 2.10.6-1

图 2.10.6-1 代码运行结果如图 2.10.6-2 所示：



```
E:\BOOK\BOOK\数据类型-运算符-表达式\  
i=0, j=0  
i=1, j=0, sizeof<i>=4  
请按任意键继续. . .
```

图 2.10.6-2

如何掌握自增自减运算符，需要做到的就是分开两条语句来算，就肯定不会出错，图 2.10.6-1 中的例子 `j=i++>-1`，对于后++或者后--，首先我们需要去掉++或者减减运算符，也就是首先计算 `j=i>-1`，因为 `i` 本身等于 -1，所以得到 `j` 的值为 0，接着在单独算 `i++`，也就是对 `i` 加 1，所以 `i` 从 -1 加 1，得到 0，因此 `printf("i=%d,j=%d\n",i,j)`；得到的结果是 0 和 0。