匠巢嵌入式

C 语言程序设计

----嵌入式 C 语言编程

成都匠巢职业技能培训学校

地址：成都市成华区府青路二段 18 号新 1 号

网址：www.jcwxpx.com 电话：13880178375

目录

C 语言程序设计

一．数据结构

C 语言有四种基本数据结构，分别为 int 整形，float 单精度浮点型，double 双精度浮点型，char 字符型，其中 float 和 double 型被统称为实型，字符型又可以首尾相接构成字符串型，四种变量在一起构成了复杂缤纷的程序世界，可以用于描述和控制世界的活动规律。

1-1.int 整形变量

整形变量的英文名是 INTEGER，类型标识为'int'，表示整数，如-2，-1，0，1，100，200 等等，整型变量是程序中使用的最多的变量类型之一

1-1-1.整形变量的定义

如：int a;　int b;　int number;

1-1-2.整形变量的初始化

在定义变量的时候，可以对其值进行初始化，初始化后的变量已经被赋予了指定值，定义变量的时候，可以对其初始化，也可以不初始化。

如：int a = 10; int b = 20;

1-1-3.整形变量的赋值

在程序的运行过程中，根据程序的要求，可以随时对整形变量的值进行修改，除了初始化之外的所有修改操作，都叫做赋值操作，当整形变量被重新赋值后，将唯一等于新赋值的值，之前初始化或所赋予的值，已经被丢弃

如：先定义，并初始化或者不初始化整形变量 a 的值，再对其进行修改

初始化不赋值并修改：

```
int a;
a = 10;
```

初始化赋值并修改：

```
int a = 20;
a = 30;
```

1-1-4.整形变量的运算

整形变量在程序中代表具体的数值，可以直接对变量进行加减，来记性对数值的加减

如：将整形变量 a, b 定义并初始化，然后对其进行算术运算

```
Int a = 10;
Int b = 20;
Int c;
c = b – a;
c = a + b;
c = a * b;
```

1-1-5.格式化输出符

c 语言中的不同变量类型，在输出到终端进行显示的时候，需要在代码中使用不同的标志进行表示，int 类型的格式化输出符为%d

1-1-6.printf 函数和格式化输出

printf 是格式化输出函数，它的作用是向终端输出各种信息，并且与格式化输出符配合输出各种变量的值，它的原型是这样：

int printf(char *format, …); 头文件为 stdio.h

如：直接输出字符和中文

```
printf(“hello world\r\n”);
printf(“匠巢嵌入式培训\r\n”);
```

如：输出整形变量的值

```
printf(“a = %d”, a);
```

1-1-7.scanf 函数和格式化输入

既然有了 printf 格式化输出函数，对应的也有从终端获取输入的函数，scanf 函数就是从终端获取输入的字符，它的原型如下：

int scanf(const char * restrict format, ...);

scanf 函数的用法比较讲究，除了需要和 printf 一样结合格式化输出符来进行输入，还需要用到一个&符号，这个符号叫做取地址符，至于是做什么用的先忽略，在指针一章当中我们会详细讲到，现在就先直接用就可以了。

比如我们需要从终端获取一个键盘的输入字符，那么我需要这样写代码：

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      printf("请输入一个整形数字并回车\r\n");
7      scanf("%d", &a);
8      printf("a = %d\r\n", a);
9      return 0;
10 }
11
```

```
E:\example.exe
请输入一个整形数字并回车
34
a = 34

Process returned 0 (0x0)   execution time : 5.007 s
Press any key to continue.
```

注意我选中的第 7 行，就是 scanf 函数的用法，站在计算机原理的角度上，因为键盘上输入的字符并不是数字，而是字符，34 是我们看到的，其实质上是字符'3'和字符'4'，但是 scanf 可以结合格式化输出符，用所见即所得的方式将输入的字符转换成我们需要的数据类型，它的用法是这样的，参数列表的第一个元素是双引号括起来的字符串，在字符串当中我们使用格式化输出符，期望得到某种类型的数据，然后第二个参数输入我们想要接收这个值的变量，变量要提前定义好，而且变量前面还需要带上&取地址符，在程序运行的时候，运行到了 scanf 函数这里，就会阻塞住，程序不会继续往下执行，必须等到按键键盘上的回车键，才会继续执行，我们输入一个数字，按下回车键，这个数字就会被以%d 整形的类型赋值给变量，这样就可以了。

和 printf 一样，scanf 也可以同时输入多个数值，如：

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      int b;
7      int c;
8      printf("请输入一个三个整形数字，以空格分隔并回车\r\n");
9      scanf("%d %d %d,", &a, &b, &c);
10     printf("a = %d  b = %d  c = %d\r\n", a, b, c);
11     return 0;
12 }
13
```

```
选择E:\example.exe
请输入一个三个整形数字，以空格分隔并回车
12 34 56
a = 12  b = 34  c = 56

Process returned 0 (0x0)   execution time : 3.797 s
Press any key to continue.
```

要注意两点，输入的时候，数字与数字之间要用空格分隔，scanf 函数第一个参数的双引号内，格式化输出符之间需要用空格分隔，否则容易出问题，特别是在不同类型变量进行输入的时候

1-1-8.回车符和换行符

回车符号为"\r" 换行符号为 "\n" 在程序中通过 printf 输出的时候，为了得到规则的版面，应该合理使用回车换行符

注：在纯软件环境的 c 语言编程中，不必使用"\r"回车符就可以进行换行，而在嵌入式 STM32 编程中，必须使用换行回车符的接合"\r\n"，才能得到换行回车的效果。"\n"除了换行的功能之外，还有刷新输出缓存，将输出内容打印到终端的作用，在某一些软件运行环境下，如果没有使用"\n"标识，终端可能会显示空白。

1-1-9.整形的长度和计算机存储空间换算

整形在计算机中，使用 4 个字节来进行辨识，字节与位的关系是:

1 字节 = 8bit bit 是计算机最小的存储单位，其单位换算为

1Mbyte = 1024Kyte

1Kbyte = 1024byte

1byte = 8bit

所以，整形的长度为

4byte = 8bit * 4 = 32bit

一个 bit 能够代表 0 和 1 两个指，32 位 bit 组合起来以供可以代表 2 的 32 次方，就是 4,294,967,296 个数字，因为 0 也表示一个数字，所以无符号整形的取值范围为 0-4,294,967,295，有符号型整形因为包含了负数，取值范围有一些变化

1-1-10.短整型

短整型是整形家族中的一员，短整型的标识为 short，可以单独使用，也可以和 int 配合使用，如:

short a;

short a = 10;

short int a;

short int a = 20;

短整型的格式化输出符号：%hd 也可以直接使用%d

短整型的数据长度：2byte = 16bit 无符号短整型的取值范围为：0-65535

1-1-11.长整型

长整型同样是整形家族中的一员，标识为 long，可以单独使用，也可以和 int 配合使用，如:

long a;

long a = 10;

long int a;

long int a = 20;

长整型的格式化输出符号：%ld 也可以直接使用%d

短整型的数据长度：4byte = 32bit 无符号短整型的取值范围为：0-4,294,967,295

长整型有一个特殊之处，在 64 位的操作系统中，可以通过长整型定义 64bit 的整形变量，其定义和初始化方式为:

long long a;

long long a = 10;

这样可以得到一个十分大的数，可以用于内存相关的寻址操作，此时格式化输出符为"%lld"

1-1-12.有符号型和无符号型

之前提到的 int 型赋值范围，都是从 0 开始，这是 int 型的有符号型类型，int 型分为由符号型和无符号行两种类型，在定义整形数值的时候用下面两个标识来进行区分

有符号型的标识为 signed 表示所定义的数据类型其取值范围是从负数开始，到 0，再到正数，其中正数负数各占一半，当不使用 signed 的时候，默认为 singed 类型

无符号型的标识为 unsigned 标识所定义的数据类型取值范围直接从 0 开始，全部在正数范围

其使用方法如下:

有符号型:

signed int a;

signed int a = 10;

或者不使用 signed 关键字，也是有符号型

Int a;

Int a = 10;

无符号型:

Unsigned int a;

Unsigned int a = 10;

有符号整形的取值范围为: -2147483648-2147483647

无符号整形的取值范围为: 0-4,294,967,295

1-2.float 单精度浮点型变量

float 类型的变量用来表示小数，它的中文名叫做单精度浮点型其类型标识为'float'，如 1.2，3.45，0.74，-56.124 这样的数值，全部通过 float 类型来进行表示

1-2-1.单精度浮点型变量的定义与初始化

和整形类似，采用 数据类型 变量名 的方式进行定义，如:

float a; float b; float val;

如果需要进行初始化，则在定义的时候对起进行赋值，如:

Float a = 12.34; float b = -32.56; float val = 23.567478;

可以将整数初始化或者赋值给 float 型的变量，如:

Float a = 10; float b = 0; float c = -25;

但是这样并不是把单精度浮点型定义成了整数，虽然看起来仍然是整数，但是是带小数点的

如 float a = 10.0; float b = 0.0; float c = -25.0;

1-2-2.单精度浮点型变量的精度

既然有小数点，必然涉及到一个精度的问题，单精度浮点型的精度为小数点后 6 位，如果超过 6 位，6 位之后的值是不准确的，不管定义的时候定义了多少个小数位，都是以前 6 位为准，第七位将会四舍五入进位到第六位，如:

Float a = 10.23; 实际上值为 12.230000

Float b = 10; 实际上值为 10.000000

Float c = 21.123456789;实际上值为 21.123457，第七位四舍五入进位到第六位

1-2-3.单精度浮点型变量的格式化输出符

单精度浮点型的格式化输出符为'%f'，当我们采用%f 进行输出的时候，不管定义的时候我们定义的是多少个小数位，中断上都只会输出 6 位，第七位四舍五入进位到第六位。

但是如果我们不想用那么多位，指需要精确到小数点后 1 位，或者 2 位，可以采用如下的方法：

将格式化输出符%f 做一些改变，变为

%.xf 其中 x 代表想精确到小数点后的位数，如

%.2f 为精确到小数点后 2 位

%.4f 为精确到小数点后 4 位

但是最大不能超过 6 位

%.7f 仍然可以在终端输出 7 位小数，但是从第七位开始就是不准确的，是随机值

1-2-4.单精度浮点型变量的运算

单精度浮点型变量的运算符合数学运算的规则，只是在运算的时候，不管定义的小数点位数有多少位，都是采用的 6 位小数点精度进行加减，如：

Float a = 12.5;

Float b = 45.156;

当执行 a + b 操作的时候，实际上是这样执行的：

12.500000 + 45.156000

当执行乘法与除法的时候同样如此，结果也是保留 6 位有效数值

1-2-5.单精度浮点型变量的长度

单精度浮点型变量的长度为 4 个字节，占用 32bit 的内存存储空间，与整形相同，但是单精度变量的存储方式与整形变量不同，是符号位+阶码位+尾数位（针对有符号数）的形式进行存储的，现阶段我们不需要去详细了解它

1-3.double 双精度浮点型变量

双精度浮点型变量的标识是'double'，和单精度浮点型类似，区别在于小数点的精度和数据长度

1-3-1.双精度浮点型变量的定义与初始化与赋值

定义：double a; double b;

初始化：double a = 10.7; double b = 45.123;

赋值：a = -23.54; b = 78.23;

1-3-2.双精度浮点型变量的格式化输出符

双精度浮点型变量的格式化输出符号为'%lf'，使用方法与单精度浮点型相同，如

Double num = 12.03;

Printf("num = %lf", num);

输出的结果同样是 6 位，第七位会被四舍五入到第六位

1-3-3.双精度浮点型变量的长度

双精度浮点型变量的长度是 8 位，是最存储空间最大的一个变量类型，一个 double 类型的变量需要占据 64bit 的内存存储空间

1-3-4.双精度浮点型变量的精度

双精度浮点型变量是精确到小数点后 12 位，虽然在使用%lf 进行格式化输出的时候，只会输出 6 位数字，但是其实际上的精度是 12 位，当使用%.xlf 进行输出的时候，x 的值最大可以到 12，小数点后 12 位以内都是精确值，但是超过 12 位就是随机值，不能使用。
所以，双精度浮点型的小数部分定义可以长达 12 位，如:
lnt number = 876.123456789012;

1-3-5.双精度浮点型变量的运算

运算方式与单精度浮点型相同，不过是整数部分和小数点后 12 位都参与运算，最终结果四舍五入到小数点后 12 位

1-4.char 字符型变量

字符型的标识是'char'，字符型变量和之前的的三种变量都有区别，字符型定义的是字符，而不是数值，计算机屏幕上显示的最小单位的图形图像，其本质都是一个一个的字符，任意能在计算机上显示的字符，都能够用 char 型进行定义。

1-4-1.字符型变量的定义，初始化和赋值

字符型变量的定义，必须用单引号来实现，如:
char par = 'a'; char chr = 'c';
中文描述为：定义一个变量 par，并将其赋值为字符'a'，因为'a'，'c'本身就是字符，这样很好理解，下面来看看错误的定义方式:
char par = a; char chr = c;
这种方式就是错误的，因为字符 a 和 c 没有带单引号的时候，既不是数值，也不是字符，是不能采用这种方式进行定义的。
因为屏幕上同样可以显示单个的数字，所以数字也可以通过定义为字符并赋值给字符型变量，其用法如下:
char par = '0'; char val = '6';
当数字加上双引号之后，就不再代表数值，而是成为了一个字符，这样 par 变量定义并赋值后，它内部存储的就是字符'0'，而不是数值 0。
再来看，数值有正数和负数，char 型类型可以定义为 0~9 的字符，但是却并不能定义为任意负数，如:
char par = '-1'; char val = '-8';
这就是错误的，因为-1 或者-8，这不是一个字符，而是两个，char 类型只能定义单个的字符，不能定义两个和两个以上的字符，同样的，这种方式是错误的:
char par = '12'; char val = 'hello'; char cha = 'world';
虽然不是负数，但是字符数量多余一个，也不能通过字符类型来定义，如果要表示多个字符，需要用到字符串，而字符串和数组又是强相关的，这放到后面数组一章来单独讲

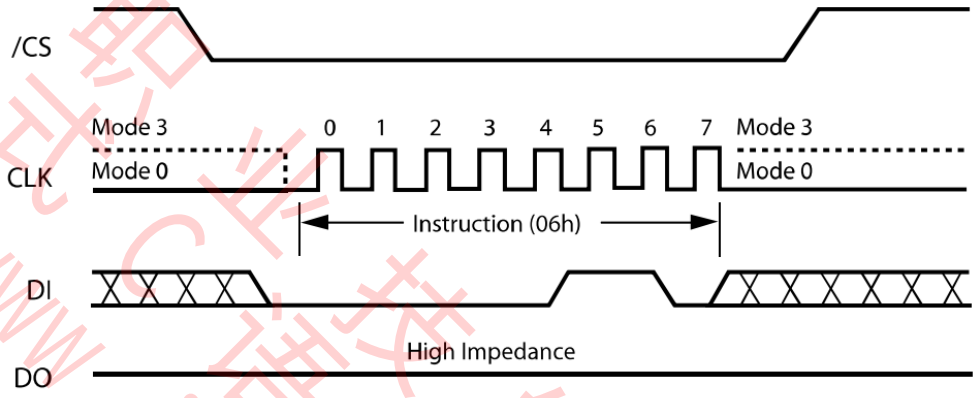
1-4-2.字符型变量的赋值

赋值操作和之前将到的三种类型一样，如:

```
char par; //定义一个字符型变量 par
par = '7'; //将字符型变量赋值为字符'7'
par = 'k'; //将字符型变量的值修改为'k'
```

1-4-3.字符型变量的长度

字符型变量的长度是 1 个字节，占据 8 个 bit，是所有变量类型中长度最小的，但是在嵌入式软件开发中，是我们打交道最细致的一个变量类型，在嵌入式芯片之间传输数据的时候，有的时候，我们要自行定义 8 个 bit 的高低电平分布，以此来表示一个特定的有意义的数值，比如在 STM32 开发的模拟 SPI 开发中，所传输的字节的每一个位都需要我们自行定义，其示意图如下：



图上的 DI，DO 是数据的输入和输出，每一个字节的每一位，都需要我们来控制其高低电平

1-4-5.ASCII 码表

在数字计算机世界中，有一张全世界范围内公用的表，叫做 ASCII 码表，这张表的作用，是用来对计算机上能显示的所有字符进行一个约定俗成的标准，它的原理是这样：
因为数字计算机世界最基本的存储单元是 0 和 1，所有更加复杂的内容，归根到底都是用 0 和 1 的各种组合来进行表示的，如表示十进制的 0，用二进制的 0 和 1 表示也是 0，十进制的 1，用二进制表示也是 1，但是十进制的 2，在数字计算机世界中就是用二进制 10 来表示的，十进制的 3，二进制用 11 表示，十进制的 4，用 100 进行表示，字符虽然不是数字，但是仍然需要一个专门的二进制表示方法，所以为了让全世界的计算机都能够使用同一套字符识别系统，不至于各人用一套标准，相互通讯的时候导致乱套，美国国家标准学会 ANSI 于上世界 50 年代开始提案，最终在 1967 年确定下来的标准，就是 ASCII 标准。
总结来说：ASCII 码表是用来确定字符与数值对应的表，每一个字符，唯一对应于一个数字
接下来我们看一看 ASCII 码表的真容：

ASCII表

(American Standard Code for Information Interchange　美国标准信息交换代码)

高四位 / 低四位		ASCII控制字符												ASCII打印字符											
		0000						0001						0010		0011		0100		0101		0110		0111	
		0						1						2		3		4		5		6		7	
		十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p	
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	◙	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	♀	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。　　2013/08/08

注意看这张表，主要分为两个左右两个区域，左边的区域是控制字符，这些字符是不会显示在终端上的，它代表一些特定的操作，比如之前我们已经用过的'\r'回车符和'\n'换行符，右边的区域是打印字符，是可以在屏幕上进行显示的。这里面有一些细节:

转义字符：转移字符使用两个字符表示，如'\r'，'\n'，注意，虽然这么写，但是仍然表示一个字符，这是特殊情况，要注意区分，转义字符都是代表一些操作，是属于不可打印字符，如

\b	退格	24
\t	横向制表	25
\n	换行	26
\v	纵向制表	27
\f	换页	28
\r	回车	29

当我们在程序当中打印这些字符的时候，打印出来的就是换行，回车，横向制表 tab 等等操作，这些操作在进行 LCD 文本显示的时候十分有用，对于一篇文章，我们不仅要识别里面的字符，还要识别里面的空格，回车换行等等转义字符，才能够在屏幕上按照正常的排版进行显示。

再来看看右侧的打印字符，细节如下:

32		48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r

其中 33 表示'!' 48 表示'0' 64 表示'@' 等等

这里特别要注意一个地方，注意看下面两张细节图:

低四位		十进制	字符	Ctrl	代码	转义字符
0000	0	0		^@	NUL	\0

左边控制字符的 0 表示空白

十进制	字符
32	

右边打印字符的 32 同样表示空白

这两个空白是有极大的区别的，32 表示的空白是空格的意思，而 0 表示的是什么都没有，是空，而且 0 还有一个转义字符为'\0'，虽然在程序中单独打印的效果是一样，但是 0 在字符串当中表示一个字符串的结尾，是十分重要的一个概念，需要我们重点掌握，应用也十分普遍。这个在后面的字符串一节再详细讲到。

1-4-6.字符型变量的格式化输出符

字符型变量的格式化输出符为%c，当我们在程序中使用如下代码的时候，屏幕上将会输出字符:

printf("%c", 'a'); 将输出字符'a'

printf("%c", '@'); 将输出字符'@'

同样的，因为字符本身是由数字代表的，我们用代表 ASCII 的数字来输出，同样会输出对应的字符，如下:

printf("%c", 97); 将输出字符串'a'

printf("%c", 64); 将输出字符串'@'

既然在程序中，字符对应的数值等于字符，那么也可以反向进行输出，输入的是字符，输出的是数字，数字使用的是格式化输出符%d，如:

printf("%d", 'a'); 注意这里的'a'一定要加单引号才代表字符，将输出数字 97

printf("%d", '@'); 注意单引号，将输出数字 64

1-4-7.字符型变量的运算

一般来说，是不会通过字符型变量进行计算的，因为没有这个必要，但是原理上是可以实现的，那就是用字符直接进行加减，本质上，是用代表字符的数字进行加减，如:

printf("%d", 'a' + '@'); 输出结果是 97 + 64 = 161;

也可以先讲字符型变量进行定义和赋值，然后通过变量进行运算，如下:

char a = 'a';

char b = '@';

char c = a + b;

printf("%d", c);

同样会输出 161 的结果，当然也可以这样做

printf("%d", a + b);

这样就省下定义一个变量 c 的空间

1-5.变量类型转换

c 语言的不同变量类型之间可以进行类型的转换，在程序开发当中，有的时候我们需要专门进行类型的转换，这种方式，叫做类型的强制转换，有的时候，我们又需要注意不经意之间类型转换带来的程序出错，这种方式，兼做类型的自动转换

1-5-1.类型的自动转换

当整形变量和单精度浮点型或者双精度浮点型进行加减乘除等算数运算的时候，会得到什么样的结果？我们来测试一下:

首先我定义一个单精度浮点型变量 num

float num = 10.9573;

再定义一个整形变量 val

int val = 2;

接下来我们定义一个 float 型变量 retval 用来接收它的返回值

float retval;

然后我让 num 和 val 做加法运算，并将结果赋值给 retval;

retval = num + val;

因为输出的变量 retval 是单精度浮点型，所以我们用%f 格式化输出符进行输出:

Printf("num + val = %f", retval);

程序代码如下:

```c
#include <stdio.h>

int main()
{
    float num = 10.9573;
    int val = 2;
    float retval;
    retval = num + val;
    printf("num + val = %f\r\n", retval);
    return 0;
}
```

编译并运行，得到结果如下:

```
num + val = 12.957300

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

输出的结果是 12.957300，输出 6 位小数，最后两位补上 0。通过这个程序我们可以发现，整形和浮点型是可以进行算数运算的，但是算数运算的结果，却变成了浮点型。这就是类型强制转

换的特点，整形在和浮点型进行运算的时候，整形变量会自动转变为浮点型变量，与浮点型变量进行运算，最终得到的结果也是浮点型。

但是这里出现了一个疑问，如果我们整形变量来接收返回值呢?

我们测试一下，将 retval 定义成整形，格式化输出符改为%d，来看看结果:

程序代码如下:

```c
#include <stdio.h>

int main()
{
    float num = 10.9573;
    int val = 2;
    int retval;
    retval = num + val;
    printf("num + val = %d\r\n", retval);
    return 0;
}
```

输出的结果为:

```
num + val = 12

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

结果变成 12，为什么呢? 其实这跟上一个代码所说整形与浮点型运算结果是浮点型的并不矛盾，这是因为在这段代码中，类型的自动转换发生了两次，第一次，发生在整形变量 val 和浮点型变量相加的时候，结果为 12.957300，已经进行了一次类型的强制转换，而在程序将这个结果赋值给 retval 变量的时候，因为 retval 现在是整形变量，所以类型进行了第二次自动转换，又从浮点型转换成了整形，所以一行代码，其实是进行了两次数据转换，在程序的编写过程中，一定要注意这一点。

接下来总结一下，发生类型自动转换的地方如下:

发生在数值的运算过程中，他们的关系是

Int 型，包括 long 型，short 型，与 float 或者 double 运算结果都是 float 或 double 型

Int 型(long, short)与 char 运算，结果都是 int 型(long, short)

char 型与 float, double 运算，结果都是 float, double 型

int 型与 short 型，long 型三者之间运算，结果是长度最长那个类型，long>int>short

大家这样理解更方便：从一个不那么规范的角度来感性认识，大家这样来认识:

Double>float>long>int>short>char，当两个或者多个变量运算的时候，最后的结果类型就是最大那个

发生在运算关系式赋值的时候

不管你的关系式当中有几个变量，最终的结果，必须是保存在一个指定类型的变量当中，这个时候也会发生变量类型的自动转换，比如

int num = int * double + char / float;

关系式右边真正得到的数据类型是 double 型，精确到小数点后的 12 位，但是保存在一个 int 型的变量当中，所以小数部分会被截取，只留下整数部分，我们用一段代码来进行演示:

```c
#include <stdio.h>

int main()
{
    double num = 12.9678;
    int retval = num;
    printf("retval = %d", retval);
    return 0;
}
```

代码运行结果如下:

```
retval = 12
Process returned 0 (0x0)   execution time : 0.065 s
Press any key to continue.
```

最后的结果变成了整形变量 12，这里特别要注意一点，双精度浮点型的值为 12.9678，但是整形的结果并没有四舍五入到 13，这是因为，之前说的四舍五入是在双精度浮点型和单精度浮点型通过格式化输出符在进行终端输出的时候会四舍五入，而类型自动转换是直接截断，是不会四舍五入的，一定要注意这一点。

但是在单精度浮点型和双精度浮点型之间进行类型转换仍然会进行四舍五入，再来测试一段代码，双精度浮点型自动转换成单精度浮点型，看看小数部分是否会四舍五入，代码如下，注意 num 小数部分的后六位，专门标记了:

```c
#include <stdio.h>

int main()
{
    double num = 12.967854987654;
    float retval = num;
    printf("retval = %f", retval);
    return 0;
}
```

运行结果如下:

```
retval = 12.967855
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

仍然发生了四舍五入，这一点也必须要注意，在用嵌入式进行民用产品开发的时候，误差大一点或许可以接受，但是如果是医疗，政府，军工，航空航天等高精尖行业的开发，就必须考虑到精度的问题，否则差之毫厘谬以千里，可能导致严重的安全事故。

最后要认识到的是，既然我们已经知道了赋值操作会导致类型自动转换，那么我们在定义接收变量的时候，就要根据情况来定义需要的类型。

1-5-2.类型的强制转换

在代码中，有的时候我们需要对变量类型进行强制转换，比如说 STM32 的 ADC 模数转换数据采集中，我采集到的电压模拟值为 2.7641 伏特，经过转换后，得到 14.5725 的输出值，但是这个值需要与系统传递过来的 int 型修正值进行相减操作，这个时候，这个时候，我就可以进行类型的强制转换，来得到需要的结果，方法如下:

```c
#include <stdio.h>

int main()
{
    float adcdec = 14.5725; //采集模拟电压
    int cmpval = 2; //修正电压
    float outval = adcdec - (float)cmpval;
    printf("outval = %lf", outval);
    return 0;
}
```

注意选中的部分，cmpval 是 int 型变量，我通过在前面括号中加入 float 数据类型的方式，将其转换成了 float 类型，这个时候，它的值就不是 2，而是 2.000000，结果虽然没有差距，但是过程已经变成了两个 float 类型的数值相减。

这段代码的执行结果为:

```
outval = 12.572500
Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

符合我们的预期值

可能大家已经意识到一点了，就我不进行类型的强制转换得到的结果不是一样吗？在这一段代码当中，的确是一样的，分两点说，一是这样编写，才符合代码的编写规范，二是等我们后面讲到数据的存储空间，指针的的时候，就能深刻体会到这里面意义，c 语言是一门十分贴近底层的高级程序设计语言，它通过指针，变量类型转换，可以实现很多细腻的操作，这些细腻的操作，是其他面向对象的高级语言所不能够实现的。

所以，类型的强制转换的格式就是:

（另一种变量类型）变量名

整形，单精度浮点型，双精度浮点型，字符型都可以互相进行强制转换

比如:

double a = 10.234;

char b = (char)a;

又或者:

long a = 'c';

double b = (double)a;

1-6.变量的命名规则

C 语言的变量，遵守一定的命名规则，不符合命名规则变量名就是非法命名，会导致系统不能通过编译。

1-6-1.命名规则

C 语言变量的命名规则如下，一共是六句话:
变量名只能以英文字或者下划线开头;
变量名不能以数字开头，但是可以包含数字;
变量名可以是单个或者多个字符，最大长度 255
变量名中的字母是区分大小写的;
变量名不能是关键字;
变量名中不能包含空格、标点符号和类型说明符（%、&、!、#、@、$）;

1-6-2.变量命名示例

我们来举几个例子:
hello 这是合法变量，全部都是英文字符
hEllK 这是合法变量，大小写英文字符都可以使用
hello 这是合法变量，下划线可以作为开头或结尾
8world 这是非法变量，数字不能开头
wor4ld7 这是合法变量，数字在中间结尾都可以
nu@mber& 这是非法变量，不能包含标点符号和类型说明符
但是在实际的使用过程中，我们应该根据变量的实际意义来对其进行命名
比如可以用 numa，numb，numc 代表不同的整形数值
或者用 adcdec 代表模拟采集量
用 valavg 代表平均值

二．运算符

为了实现各种各样的功能，c 语言支持多种运算符，分别对应于各种运算操作，最常见的是算术运算，还有逻辑运算，移位运算，三木运算等等，每一种运算都有自己对应的运算符，而且每一种运算符之间也区分不同的优先级，我们依次来进行学习。

2-1．算数运算符

算数运算是我们最熟悉的运算，从小学开始我们就已经开始学习，c 语言中的算数运算分为加，减，乘，除，取余，一共是五种，虽然同样是数学的算数运算概念，但是又和数学上的算数运算在实现上有些许的差别，下面我们先看加法运算

2-1-1.加法运算

C 语言中的加法运算和数学上的加法运算完全一致，运算符是'+'，在程序中直接使用即可，如下代码所示:

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 20;
    int c = a + b;
    printf("c = %d\r\n", c);
    return 0;
}
```

结果如下:

```
c = 30

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

上述是整形的加减，同样的，浮点型之间，字符型与浮点型，字符型与整形，整形和浮点型之间都是相同的操作，这在第一章数据类型 5-1 小节中已经详细描述，要注意到数据类型之前自动的或者强制性的自动转换，还要注意数据之间强制类型转换时候的四舍五入。

2-1-2.减法运算

减法运算和数学中的减法运算一样，运算符是'-'，代码如下:

```c
#include <stdio.h>
int main()
{
    int a = 10;
    long b = 4;
    short c = a - b; //整形相减
    printf("c = %d\r\n", c);
    double d = 78.23;
    double e = 98.23;
    double f = d - e; //浮点型相减
    printf("f = %lf\r\n", f);
    char g = '9';
    char h = 'p';
    int i = g - h; //字符型相减
    printf("i = %d\r\n", i);
    return 0;
}
```

直接来看运算结果:

```
c = 6
f = -20.000000
i = -55

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

2-1-3.乘法运算

乘法运算同样遵循数学中的乘法，运算符是'*'，看下面一段代码:

```c
#include <stdio.h>
int main()
{
    float a = 1.1;
    float b = 1.1;
    int c = 2;
    int d = 2;
    char e = 'a'; //字符a二进制值为97
    char f = 'b'; //字符b二进制指为98
    float g = a * b; //浮点型相乘
    int h = c * d; //整形相乘
    int i = e * f; //字符型相乘
    printf("g = %f  h = %d  i = %d\r\n", g, h, i);
    return 0;
}
```

这里要注意代码中选中的地方，因为字符型的长度是 1 个字节 8 个位，最大数值为 255，当字符'a'和字符'b'相乘的时候，数值上实际是 97 * 98，字符型是远远放不下的，所以要用 int 型变量来保存计算后的值。

这是代码的运行结果:

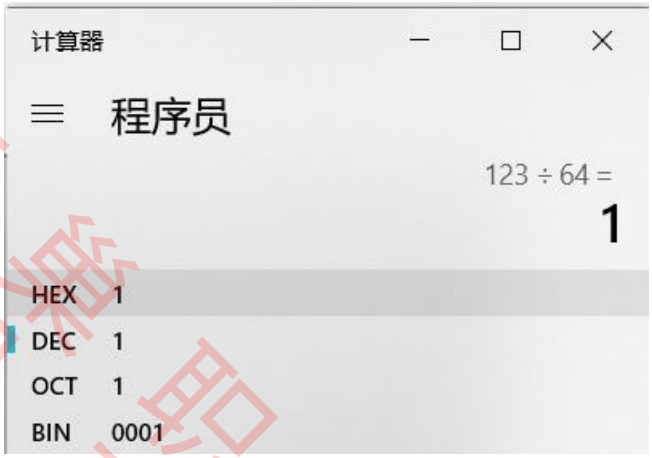
```
g = 1.210000  h = 4  i = 9506

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

2-1-4.除法运算

除法运算数学中的除法就有区别了，符号是'/'；在数学中，除法会根据需要，通过四舍五入精确到小数点后的指定位数，但是在程序当中，对于整形和字符型，或者整形和字符型的混合运算，除法运算的结果，只会保存整数位，而且小数部分不会四舍五入，我们来举一个例子:

在 windows 计算器系统工具中，我们用 123 来除以 64 进行测试，123 / 64，按照科学计算除法，结果如下:

这里我们要注意，结果非常接近 2，按照四舍五入的思想，如果只会一位表示，应该用 2 来进行表示，接下来我们将计算器设置成程序员模式再来进行测试:



结果为 1，这是因为程序员模式的时候，除法结果只会保存整数部分，而不会保存小数部分，也不会有四舍五入，这一点一定要注意，接下来我们在程序中进行演示，代码如下:

```c
#include <stdio.h>

int main()
{
    int a = 123;
    int b = 64;
    int c = a / b; //除法运算
    printf("a / b = %d\r\n", c);
    return 0;
}
```

结果如下，与预测一致:

```
a / b = 1

Process returned 0 (0x0)   execution time : 0.022 s
Press any key to continue.
```

这里又有一个问题，如果我用浮点型来保存结果，不是可以保存小数了吗？我们测试一下:

```c
#include <stdio.h>

int main()
{
    int a = 123;
    int b = 64;
    float c = a / b; //浮点型保存结果试一下
    printf("a / b = %f\r\n", c);
    return 0;
}
```

```
E:\example.exe
a / b = 1.000000

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

结果却仍然是 1，多了 6 个小数位而已，这是因为，虽然是用 float 型来保存结果，但是程序的实际运行过程是右侧的两个整形数值，按照整形数值的运算规律并得出结果 1 后，在类型自动转换并赋值给 float 变量 c，所以结果仍然是 1

但如果我们把计算项其中的一个变成浮点型呢？我们来看一看效果:

```c
#include <stdio.h>

int main()
{
    float a = 123;
    int b = 64;
    float c = a / b; //浮点型保存结果试一下
    printf("a / b = %f\r\n", c);
    return 0;
}
```

```
E:\example.exe
a / b = 1.921875

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

当其中一个计算项是浮点型的时候，结果保存了小数，这是因为计算过程发生了改变，计算的过程中，因为有一个计算项是浮点型，所以另一个整形也会被类型自动转换成浮点型并与之进行除法运算，得到的结果本来就是带小数的，再赋值给 float 类型的变量，打印出来就是这样了。所以关于除法运算的结果，我们又得出一个结论，只要计算项有浮点型，计算的结果本身不会抛弃小数部分，小数部分需不需要，取决于接收计算结果的变量类型，另外我们用类型强制转换来试试这段代码:

```c
#include <stdio.h>

int main()
{
    int a = 123;
    int b = 64;
    float c = a / (float)b; //类型强制转换
    printf("a / b = %f\r\n", c);
    return 0;
}
```

强制转换，效果跟自动转换是一样的，注意代码中选中的部分

2-2-5.取余运算

取余运算就是数学中的计算余数，运算符是'%'，它的作用是得到被除数中不能被除数整除的部分，要注意的是，当我们使用取余运算的时候，运算符的两边不能有 float 型或者 double 型的变量，浮点型不支持取余运算，接下来我们来看代码:

100 可以被 20 整除，是没有余数的，结果为 0，这里要注意一点，%符号是不能直接在 printf 函数中输出的，需要用两个%%，才能被正常输出和显示，注意代码中选中的部分

```c
#include <stdio.h>

int main()
{
    printf("100 %% 20 = %d\r\n", 100 % 20);
    return 0;
}
```

```
E:\example.exe
100 % 87 = 13

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

100 不能被 87 整除，结果是不能整除的部分 13

```c
#include <stdio.h>

int main()
{
    printf("100 %% 87 = %d\r\n", 100 % 87);
    return 0;
}
```

```
E:\example.exe
100 % 87 = 13

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

如果除数大于被除数，结果就是被除数本身，因为无法整除：用 13 取余 21 测试：

```c
#include <stdio.h>

int main()
{
    printf("13 %% 21 = %d\r\n", 13 % 21);
    return 0;
}
```

```
E:\example.exe
13 % 21 = 13

Process returned 0 (0x0)   execution time : 0.022 s
Press any key to continue.
```

如果%两边有浮点型 float 或者 double 变量，是无法进行取余运算的，而且编译都无法通过：

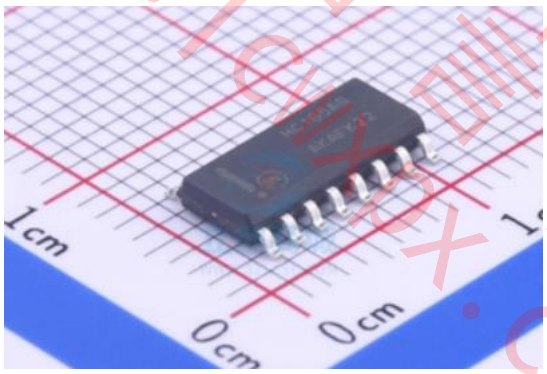
```
1  #include <stdio.h>
2
3  int main()
4  {
5      float a = 100.23;
6      float b = 20.00;
7      printf("a %% b = %d\r\n", a % b);
8      return 0;
9  }
10
```

```
Logs & others
Code::Blocks | 搜索结果 | Cccc | 构建记录 | 构建信息 | CppChecl
文件                行   信息
E:\example.c             In function 'main':
E:\example.c        7    error: invalid operands to binary % (have 'float' and 'float')
                         === 构建 失败: 1 error(s), 0 warning(s) (0 分, 0 秒) ===
```

注意看编译信息，在第 7 行，出现 error，直接就不会通过的

2-2.逻辑运算符与选择结构

逻辑运算符是用来判断数值或者关系式之间的逻辑关系的，有三种，分别是与，或，非逻辑关系，每一种都有专门的符号与之对应，在数字电路的世界当中，逻辑运算是用门电路来实现的，叫做与门，或门与非门，具体到电子元器件芯片上，就是与门芯片，或门芯片和非门芯片，他们长这样，一个这样的芯片可以是三种门，具体看型号:



要注意一点：双目逻辑运算符的优先级小于算术运算符，而单目的逻辑运算符优先级高于算术运算符，关于什么是双目和单目，我们后面再将，先又这么个印象即可

2-2-1.C 语言中的真和假

逻辑运算，必须接合真和假的概念，真假就是字面上的意思，“一天有 24 个小时”，这句话为真，如果“一天有 25 个小时”，这句话就是假

具体到 C 语言里面，就只有简单一句话，0 为假，非零为真。

展开量讲，就是说，数值，变量或者关系式的运算结果为 0，那么这就是假，如果不为零，数值为正数或者负数，就是真

如：数值 0 的逻辑就是假

如：int a = 0; a 就是假，因为 a 的值为 0

如：int a = 0; int b = 12; int c = a * b; c 也是假，因为运算结果是 0

如：数值 198 的逻辑是真

如：a = -1; a 是真，因为不为 0

如：double a = 1.3；a 为真，因为不为 0

如：double a = 0.00；a 为假，因为不为 0

2-2-2.选择结构

C 语言中有各种各种代码执行结构，我们先要把选择结构讲了，才能接着往下讲运算符，选择结构通过三个关键词 if， else， else if 组成，它的作用通俗易懂，就是根据什么条件，执行什么动作，比如如果今天是星期三，那么今天要上课，如果今天是星期日，那么今天不上课，它的结构有三种:

a. If(数值或者关系式) //判断小括号中的数值或者而关系是的真假逻辑关系

```
{
        执行相应的代码;
}
```

在这种结构中，如果数值或者关系式为逻辑真，则执行下方花括号中的内容，如果逻辑为假，则不执行花括号中的内容，在嵌入式软件开发中，我们往往用这种方式来判断是否让 CPU 执行某些动作

b. If(数值或者关系式)

```
{
        执行代码 1
}
else
{
        执行代码 2
}
```

在这个结构中，如果数值或者关系式为逻辑真，则执行代码 1，如果为逻辑假，则执行代码 2

c. If(数值 1 或者关系式 1)

```
{
        执行代码 1
}
else if(数值 2 或者关系式 2)
{
        执行代码 2
}
else
{
        执行代码 3
}
```

在这个结构中，如果数值 1 或者关系式 1 为真，则执行代码 1，如果为假，那就判断数值 2 或者关系式 2，如果为真，则执行代码 2，如果为假，再来执行代码 3，如果代码 1 和代码 2 有任意一个执行了，都不会执行代码 3，if-else if-else 三个为一个整体，三个当中只会执行一个，要注意一点，else if 可以不只一个，多个也是可以的，如下:

If(条件 1){执行代码 1}

else if(条件 2){执行代码 2}

else if(条件 3){执行代码 3}

……

else{}

在这一段代码中，所有的 if-else if-else 都是一个整体，如果条件 1 成立，则执行代码 1，别的都不执行，如果条件 1 不成立条件 2 成立，则执行代码 2，别的都不执行，总之是从上往下每个条件一次判断，找到条件成立那个，执行器相应的代码，然后整个选择语句结束，如果条件 1 和条件 2 都成立，则执行代码 1，条件 2 虽然也成立，但是因为已经执行了代码 1，也不会执行，整个选择语句结束。

接下来是首要注意的几点：

a. If 可以单独使用，不加 else if，也不加 else，如：

```
int main
{
	If(条件)
	{
		执行代码;
	}
	return 0;
}
```

If-else if 可以结合使用，不加 else，如：

```
int main
{
	If(条件 1)
	{
		执行代码 1;
	}
	else if(条件 2)
	{
		执行代码 2;
	}
	return 0;
}
```

else if 和 else 不能单独使用，必须跟 if 结合使用

b. 如果判断多个条件，只要是条件为真就要执行代码，可以用多个 if 来实现，而不用 if-else if-else 的结构，如：

```
int main()
{
	if(条件 1)
	{
		执行代码 1
	}
	If(条件 2)
	{
		执行代码 2
	}
```

```
        ……
        return 0;
    }
```

c. if-else if-else 可以无线嵌套，组成十分复杂的逻辑树关系，要慎用，容易把自己绕晕，如：

```
if(条件 1)
{
    If(条件 2)
    {
        If(条件 3)
        {
            执行代码;
        }
    }
}
```

2-2-3.逻辑与

逻辑与在 C 语言中的符号是 '&&'，既然是判断逻辑关系，就没有具体的算数计算过程，而仅仅是判断运算符两边的数值或者关系式是否为真，如果两边都为真，那么结果就是真，有任意一边为假，或者两边都为假，那么结果就是假

逻辑与在电路中的符号是这样，英文是 and。

它的用法格式如下:

值(变量，关系式) && 值(变量，关系式)

如：1 && 2;

如：val && num;

如：a * b && c * d; 这种写法涉及到运算符优先级，这个我们后面再说

之前提到，如果&&两边的条件为真，那么结果就为真，如果任意一边或者两边为假，那么结果就是假，我们来测试一下

先来看第一段代码,我们直接将逻辑与的结果打印出来:

```c
#include <stdio.h>

int main()
{
    int a = -34;
    int b = -45.32;
    printf("-34 && -45.32  = %d\r\n", -34 && -45.32);
    printf("12 && 42       = %d\r\n", 1 && 2);
    printf("2.324 && 9.54  = %d\r\n", 2.324 && 9.54);
    printf("87 && 0        = %d\r\n", 87 && 0);
    printf("3 + 4 && 0 * 5 = %d\r\n", 3 + 4 && 0 * 5);
    return 0;
}
```

```
E:\example.exe
-34 && -45.32  = 1
12 && 42       = 1
2.324 && 9.54  = 1
87 && 0        = 0
3 + 4 && 0 * 5 = 0

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

通过结果我们发现，不管是变量还是关系式还是数值，只要有一端是 0，那么最后的结果就是 0，就是假，而两边都不为零的数值，变量和关系式，得到的结果就不为 0，是 1，表示真。

这就是逻辑与关系，但是单独的进行逻辑与运算作用不大，我们往往结合 if 选择结构来使用，逻辑与运算可以作为 if 结构的条件，接下来是代码示例:

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a && b)
    {
        printf("结果为真\r\n");
    }
    else
    {
        printf("结果为假\r\n");
    }
}
```

```
E:\example.exe
请输入两个整形数字
23 65
结果为真

Process returned 0 (0x0)   execution time : 6.526 s
Press any key to continue.
```

我输入的是 23 65，回车后提示结果为真，符合我们的要求，再来一个结果为假的

```
E:\example.exe
请输入两个整形数字
0 1000
结果为假

Process returned 0 (0x0)   execution time : 16.488 s
Press any key to continue.
```

因为 0 位假，所以结果肯定就是假了

与运算的两端可以是关系表达式，如下:

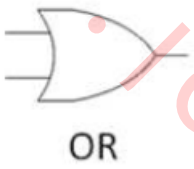
```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a * b && a + b)
    {
        printf("结果为真\r\n");
    }
    else
    {
        printf("结果为假\r\n");
    }
}
```

```
E:\example.exe
请输入两个整形数字
12 34
结果为真

Process returned 0 (0x0)   execution time : 2.685 s
Press any key to continue.
```

这次与运算判断的是 a * b 与 a + b 两个关系式的逻辑关系，程序在执行的时候，会先执行关系式 a * b，再执行关系式 a + b。再用两者的结果进行逻辑与运算，最后 if 选择结构根据与运算的结果来执行代码，因为算术运算符优先级要比双目的逻辑运算符高，所以可以直接这样写，不用加括号，关于什么是双目运算符，后面再讲

2-2-4.逻辑或

逻辑或的运算符是这样'||'，用两个竖线表示，逻辑与跟逻辑或不同，当运算两边的值，变量和关系式有一个为真的时候，或运算的结果就是真，只有两个都为假的时候，输出结果才是假，逻辑或的电路符号如下，用英文 or 来表示



同样的，逻辑运算符的优先级小于算数运算符，我们用代码来进行演示

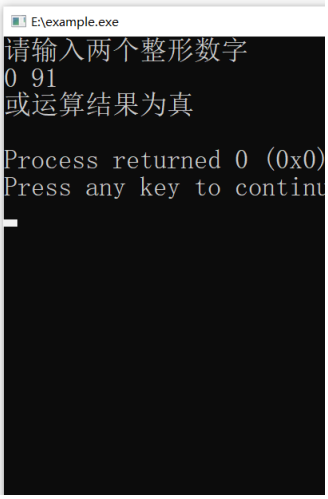
```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a || b)
    {
        printf("或运算结果为真\r\n");
    }
    else
    {
        printf("或运算结果为假\r\n");
    }
}
```

```
E:\example.exe
请输入两个整形数字
12 87
或运算结果为真

Process returned 0 (0x0
Press any key to contin
```

注意程序中选中的地方，运算符已经换成了逻辑或，输入 12，87 并回车，两边都是真，结果为真。

接下来看一边为真的情况:

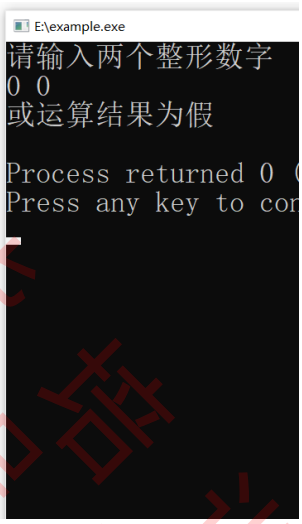
```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a || b)
    {
        printf("或运算结果为真\r\n");
    }
    else
    {
        printf("或运算结果为假\r\n");
    }
}
```



第一个数字我输入 0，第二个数字输入 91，只有一边为真，结果也是为真

最后是逻辑或运算符两边都为假的情况

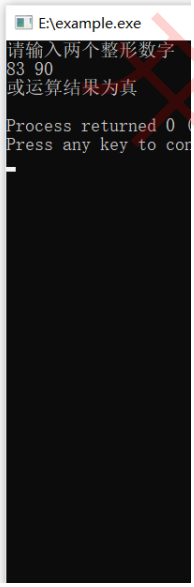
```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a || b)
    {
        printf("或运算结果为真\r\n");
    }
    else
    {
        printf("或运算结果为假\r\n");
    }
}
```



唯一能让逻辑或运算符输出结果假的情况是，运算符两边结果都为假

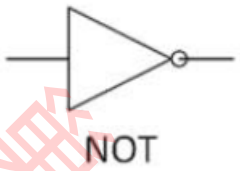
对于两边都是关系式，同样有效:

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个整形数字\r\n");
    scanf("%d %d", &a, &b);
    if(a * b || b * 0)
    {
        printf("或运算结果为真\r\n");
    }
    else
    {
        printf("或运算结果为假\r\n");
    }
}
```



2-2-5.逻辑非

逻辑非，就是反着来，如果一个数值，变量或者关系式的结果是真，那么逻辑非运算后就变成了假，反之，则为真，总之就是刚好相反，逻辑非的电路符号如下，用英文 not 表示，注意三角形右边端点的小圆圈，很多电路简化图上，为了表示逻辑非，直接用小圆圈代替



和逻辑与，逻辑或设置算术运算符都不一样的是，逻辑非是单目运算符，它的优先级比算数运算符要高，比逻辑或或者逻辑与的优先级也要高。

逻辑非的符号是'!'，就是一个单独的感叹号，在代码当中使用时，进行运算的数值，变量或者关系式位于感叹号的右边，感叹号的左边留空，

如：!24;　24 的逻辑非运算，结果是假

如：!val;　变量 val 的非运算，结果取决于 val 的真假，与 val 的真假刚好相反

如: !(a + b)　关系式 a+b 的逻辑非运算，结果取决于 a+b 的结果

需要注意的是，当关系式进行非运算的时候，必须给关系式加上括号，否则逻辑非运算符只会结合离他最近的一个变量或者数值，导致结果出错。接下来我们来看看代码:

```c
#include <stdio.h>

int main()
{
    int a = 0;
    if(!a)
    {
        printf("逻辑非运算 !a的值为真\r\n");
    }
    else
    {
        printf("逻辑非运算 !a的值为假\r\n");
    }
    return 0;
}
```

```
E:\example.exe
逻辑非运算 !a的值为真

Process returned 0 (0x
Press any key to conti
```

很明显，变量 a 的值为 0，非运算后，变成了 1，输出!a 为真

再来看代码:

```c
#include <stdio.h>

int main()
{
    int a = 7;
    int b = -10;
    if(!(a * b))
    {
        printf("逻辑非运算 !(a * b)的值为真\r\n");
    }
    else
    {
        printf("逻辑非运算 !(a * b)的值为假\r\n");
    }
    return 0;
}
```

```
E:\example.exe
逻辑非运算 !(a * b)的值为假

Process returned 0 (0x0)
Press any key to continue.
```

在这段代码中，对关系式 a*b 进行逻辑非运算，a * b 的值为-70，非零为真，取反后为假。

在看看另外的情况：见代码

```c
#include <stdio.h>

int main()
{
    int a = 7;
    int b = -10;
    if(!!(a * b))
    {
        printf("逻辑非运算 !!(a * b)的值为真\r\n");
    }
    else
    {
        printf("逻辑非运算 !!(a * b)的值为假\r\n");
    }
    return 0;
}
```

```
E:\example.exe
逻辑非运算 !!(a * b)的值为真

Process returned 0 (0x0)   ex
Press any key to continue.
```

在这段代码中，我进行了两次非运算，这次运算的顺序是，变量 a 先与 b 相乘，结果为真，相乘后的值与离关系最近的逻辑非运算符相结合进行非运算，结果为假，最后再与最左边的逻辑非运算符结合进行逻辑非运算，结果又反过来了，所以输出真，这段代码证明，逻辑非运算符是可以多次使用的。

2-3.关系运算符

关系运算符就是比较大小的运算符，大小关系有这么几种，分别是大于，大于等于，等于，小于，小于等于，每一种都有自己对应的符号，在 C 语言中，就是字面的意思，比较数值，变量或者关系式的大小，关系运算输出的结果是真或者假，和逻辑运算符一样，也常常结合 if 语句进行使用，关系运算本身不存在数值的计算，在 C 语言中没有单独的用处，我们分别来介绍

2-3-1.大于运算

大于符号的运算符是'>'，如果运算符左边的数值大于右边的数值，则结果为真，如果左边小于或者等于右边，则结果为假，左右两边都可以是数值，变量或者关系式，同样的，我们先来看一下直接输出关系运算的结果：

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 4;
    printf(" a > b      : %d\r\n", a > b);  //变量比较
    printf(" b - a > b  : %d\r\n", b - a > b);  //关系式比较
    printf("10 > 4      : %d\r\n", 10 > 4);  //数值比较
    return 0;
}
```

```
E:\example.exe
 a > b      : 1
 b - a > b  : 0
10 > 4      : 1

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

从上到下分别是变量比较，关系式比较和数值比较，输出的是真假，结果是 0 或者 1，变量的比较可以是 int 型和 char 型和 double 型或者 float 型任意两者之间，但是因为 char 型代表的是字符，所以我们在进行比较的时候，比较的不是字符本身与另外的数值的大小关系，而是字符在 ASCII 码中对应的数字与另一个数字的关系，我们来看一下：

```c
#include <stdio.h>

int main()
{
    int a = 10;
    int b = 4;
    printf(" 10 > 7.65 : %d\r\n", 10 > 7.65); //整形与浮点型比较
    printf(" a  > b    : %d\r\n", 'a' > 'b'); //字符之间比较
    printf(" s  > 72   : %d\r\n", 's' > 72); //字符与整形比较
    return 0;
}
```

```
E:\example.exe
10 > 7.65 : 1
a  > b    : 0
s  > 72   : 1

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

在这段代码中，字符 a 大于字符 b 输出 0，这是因为 a 的 ASCII 数值比 b 要小：

97	a
98	b

而字符 s 大于 72 输出 0 是因为 s 字符的值为 115

115	s

但是关系运算符没有这种用法，这只是演示，关系运算符主要还是与 if 语句进行结合使用，如下代码：

这一段代码，我们通过 if 语句嵌套进行了三个判断

如果 a > b，则进入 if 的执行语句内

进入 if 的执行语句后，判断 a + b 是不是大于 30，如果大于 30，则进入嵌套 if 的执行语句内

如果 a < b，则进入 else 的执行语句内

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入a和b的值并回车，用空格分隔\r\n");
    scanf("%d %d", &a, &b);
    if(a > b) //如果a > b
    {
        printf("a > b\r\n");
        if(a + b > 30) //如果a + b > 30
        {
            printf("a > b and a + b > 30\r\n");
        }
    }
    else //剩下的情况 a小于或者等于b
    {
        printf("a <= b\r\n");
    }
    return 0;
}
```

```
E:\example.exe
请输入a和b的值并回车，用空格分隔
16 7
a > b

Process returned 0 (0x0)   execut
Press any key to continue.
```

我输入 a 为 16，b 为 7，a 大于 b，但是之和小于 30，所以只输出 a > b，接下来换两个数值再试试

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入a和b的值并回车，用空格分隔\r\n");
    scanf("%d %d", &a, &b);
    if(a > b) //如果a > b
    {
        printf("a > b\r\n");
        if(a + b > 30) //如果a + b > 30
        {
            printf("a > b and a + b > 30\r\n");
        }
    }
    else //剩下的情况 a小于或者等于b
    {
        printf("a <= b\r\n");
    }
    return 0;
}
```

```
E:\example.exe
请输入a和b的值并回车，用空格分隔
23 18
a > b
a > b and a + b > 30

Process returned 0 (0x0)   execut
Press any key to continue.
```

这一次不仅 a>b 而且 a+b>30，所以不仅输出了 a > b 而且输出了嵌套 if 中的语句，再来个执行 else 的

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入a和b的值并回车，用空格分隔\r\n");
    scanf("%d %d", &a, &b);
    if(a > b) //如果a > b
    {
        printf("a > b\r\n");
        if(a + b > 30) //如果a + b > 30
        {
            printf("a > b and a + b > 30\r\n");
        }
    }
    else //剩下的情况 a小于或者等于b
    {
        printf("a <= b\r\n");
    }
    return 0;
}
```

```
E:\example.exe
请输入a和b的值并回车，用空格分隔
13 54
a <= b

Process returned 0 (0x0)   execut
Press any key to continue.
```

13 小于 54，所以执行的是 else 中的语句

2-3-2.小于运算

小于运算的符号是'<'，如果运算符左边的数值，变量或者关系式小于右边的，则返回真，如果左边大于或者等于右边，则返回假，因为十分简洁易懂，这里就不进行代码演示了

2-3-3.等于运算

等于运算的符号是'==', 是两个等于，在数学中，一个等于符号表示相等关系，而在 C 语言中，一个等于符号表示的是赋值，所以就用两个等于符号来表示相等，等于运算同样经常集合 if 选择结构使用，代码如下：

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个数字,用空格号分隔:\r\n");
    scanf("%d %d", &a, &b);
    if(a == b)
    {
        printf("a等于b\r\n");
    }
    else
    {
        printf("a与b不相等\r\n");
    }
    return 0;
}
```

```
E:\example.exe
请输入两个数字,用空格号分隔:
23 23
a等于b

Process returned 0 (0x0)   ex
Press any key to continue.
```

在使用等于运算的时候有一点要特别注意，有的时候会忘记写两个等于号，只写了一个，这样程序编译是不会报错的，但是结果会出问题，如下:

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    printf("请输入两个数字,用空格号分隔:\r\n");
    scanf("%d %d", &a, &b);
    if(a = b)
    {
        printf("a等于b\r\n");
    }
    else
    {
        printf("a与b不相等\r\n");
    }
    return 0;
}
```

```
E:\example.exe
请输入两个数字,用空格号分隔:
13 87
a等于b

Process returned 0 (0x0)   ex
Press any key to continue.
```

注意右侧的结果，我输入的是 13 和 87，这两个值是不相等的，但是程序输出仍然是相等，这是因为在 if 的条件当中，我们是进行了一个赋值运算，将 b 的值赋值给 a，然后判断 a 的值，所以只要 a 最终不为零，那么无论如何都会输出 a 等于 b，这种情况一定要注意

2-3-4.大于等于运算

大于等于运算的符号是">="，大于号在左边，等于号在右边，大于等于运算符左边的值，变量或者关系式大于或者等于右边的值，变量或者关系式，那么运行结果为真，否则为假，就是大于运算和等于运算结合在一起。

2-3-5.小于等于运算

小于等于预算的符号是"<="，符号左边的值，变量或者关系式小于或者等于符号右边的值，变量或者关系式，那么运行结果为真，否则为假。

2-4.位运算符

说在最前面：位运算只针对整形(int, short, long)和字符型(char)，不能针对浮点型(float, double)

位运算是针对计算机二进制数字世界的一种操作方法，它精确操作计算机最基本存储单位 bit，可以进行与操作，或操作，异或操作，移动操作等等，在嵌入式软件开发中，是十分有用的运算符，我们在操作寄存器的时候，用移位运算符，会极大的提高工作效率。在讲位运算符之前，我们先来认识一下计算机存储结构和数据在计算机内部的存储方式：我们先来看一个表格:

字节 1	字节 2	字节 3	字节 4	。。。。。。					
								。。。。。。	字节 40

这个表格，我们可以把它理解成计算机内部的存储空间，比如我们的 Stm32F407zgt6 单片机，它的存储空间是 128kbyte，就是有 128 * 1024 这么多个上图这种小格子，每一个格子可以保存一个字节的数据。之前在讲变量的时候，我们提到 1byte 为 8bit，所以我们把上图其中一个格子(一个字节)单独拿出来，看下图，代表一个字节:

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

上面的表格表示一个字节的内部结构，是由 8 个 bit 所组成，每一个 bit 可以有 0 和 1 两种值，从低到高 8 位，所以一个字节可以记录 0 到 255 以供 256 个数字，不过这些数字是用二进制进

行表示的，但是呈现在计算机屏幕终端给我们人看的时候，已经转换成了十进制，同样的，我们往计算机输入一个十进制的数字，真正存储到内存中的时候，已经转换成二进制，写到一个一个的 bit 中去了。

如果我们要存入内存一个数字 3，转换成二进制就是 11，存入内存后占据一个字节，如下：

0	0	0	0	0	0	1	1

如果我们要存入数字 15，转换成二进制是 1111，那么就应该是这样：

0	0	0	0	1	1	1	1

如果我们把这些为全部填满，就是这样：

1	1	1	1	1	1	1	1

这就是二进制数字 11111111，转换成十进制，就是 255，这也是一个字节能存放的最大值，存储空间讲清楚了，接下来我们来看各种位运算符：

2-4-1.左移运算

左移运算符的符号是"<<"，两个向左的折号，它的作用是将存储在内存中的数据向左移动固定的位数，移动后空出来的位，用 0 来补齐，它的用法如下：

数值 << 位数

看下面的代码演示：

```c
#include <stdio.h>

int main()
{
    char num = 7; //定义一个单字节数字7
    char val; //用来接收结果
    val = num << 1; //将7左移一位存入val变量
    printf("val = %d\r\n", val);//打印结果
    return 0;
}
```

选中的这一行，就是移位运算符的用法，我们同样用图片和表格来演示发生了什么：

首先是在 char 型数据类型一个字节的空间内存放了一个数值 7，7 的二进制是 111，所以存放方式是这样的：

0	0	0	0	0	1	1	1

现在让 7 它左移一位，就变成了这样，注意第 0 位变成了 0，7 整体左移一位变成现在到第 4 个位，之前的第 0 位现在用数值 0 来补齐如下表：

0	0	0	0	1	1	1	0

这样移位之后，恰恰数值是增大了一倍，7 变成了 14，二进制 1110 换算成十进制就是 1110。这就是左移运算符的作用。现在我们来看这段代码输出的结果：

```c
#include <stdio.h>

int main()
{
    char num = 7; //定义一个单字节数字7
    char val; //用来接收结果
    val = num << 1; //将7左移一位存入val变量
    printf("val = %d\r\n", val);//打印结果
    return 0;
}
```

```
E:\example.exe
val = 14

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

结果就是 14，和预计值是一样的，满足运算规则。

现在我们来看看另一种情况，如果数值移动之后，装不下了会发生什么？我们用 char 型数值 252 来做测试，252 的二进制位 11111100,我们把它往左边移动 1 位试一下，写法如下：252 << 4；下面是代码和程序运行结果，这里变量用无符号型 unsigned 来定义，是为了去除负号的影响，更符合人的理解习惯：

```c
#include <stdio.h>

int main()
{
    unsigned char num = 252; //定义一个单字节数字7
    unsigned char val; //用来接收结果
    val = num << 1; //将7左移一位存入val变量
    printf("val = %d\r\n", val);//打印结果
    return 0;
}
```

```
E:\example.exe
val = 248

Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

这个结果跟我们预计的不一样，之前说左移动一位数值不是应该增加一倍吗？怎么反而变小称为 248 了？本着这个问题，我们从寄存器的来看看发生了什么：

首先看移位之前 252 在字节当中的存储情况，前 6 位是 1，后 2 位是 0:

1	1	1	1	1	1	0	0

往左移动一位之后，实际上是变成了这样，第 3 位也变成了 0:

1	1	1	1	1	0	0	0

这是因为，因为 char 型的变量只有一位，最大容量是 8 个位，252 本来就顶头了，向左移动一位之后，char 型变量存不下，只好把最高位截去，这种情况很好理解，试想 char 数据类型是一个长方形的盒子，里面的物品向左移动之后装不下了，那就只能丢弃了。而二进制 11111000 的十进制数值就是 248，所以我们在移位的时候，要根据实际的情况来移动。

2-4-2.右移运算

右移运算符的运算符是">>"，它的运算规则和左移运算符一样，只是数据移位方向相反，而且数据移动到右边超出数据类型的范围的时候，超出部分同样会被截取，当移动未超出数据类型范围的时候，每向右移动一位，数据缩小一倍。和左移运算符规则一样的部分，这里就不再画图和代码演示了，这里来演示左移右移扩展的知识点，先用左移来举例

如果我有一个 char 型的数值，我的目的就是需要将它放大两倍，我需要使用放大后的值，但是一旦 char 型左移超出范围，就会被截断，那我应该怎么操作呢？这个时候，我们可以用比 char 型长度更长的变量类型来接收 char 型的返回值，如 short 型，或者 int 型都可以：来看代码演示：

```c
#include <stdio.h>

int main()
{
    unsigned char num = 252; //定义一个单字节数字7
    unsigned short val; //用来接收结果
    val = num << 1; //将7左移一位存入val变量
    printf("val = %d\r\n", val);//打印结果
    return 0;
}
```

```
E:\example.exe
val = 504

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

在这一段代码当中，我用来接收左移结果的变量就是 short 型，这样结果刚好就是原数据的两倍：504，再来通过表格看一看，首先是 char 型变量存放的值 252：

1	1	1	1	1	1	0	0

左移之后，我们是用 short 型变量来接收，就变成了这样：

							1	1	1	1	1	1	0	0	0

用 short 型，就能够装的下了，用 int 型更能装的下。

但是这种方式只能适用于左移，右移不行，因为不管哪种数据变量，数据的长度都是向左扩张，向右都是最低位，所以右移装不下的数据，就是真的丢弃了，在使用中一定要注意这一点。

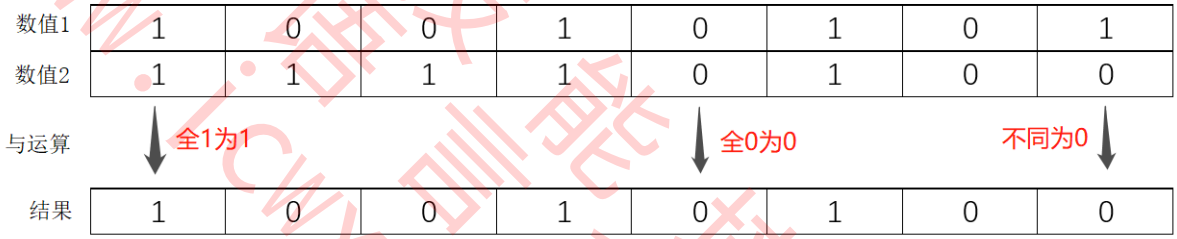
2-4-3.按位与

按位与的符号是'&'，这个符号我们在逻辑与运算当中见识过一次，不过当时是两个符号一起使用，当单个使用这个符号的时候，它就不是逻辑与，而是按位与运算，它的作用，是将符号两端的变量，按位进行与运算，并输出与运算之后的结果，可以说逻辑与是宏观意义上的与运算，而按位与则是微观维度上，按位进行的与运算。

它的写法是这样样子：数值(变量或者关系式) & 数值(变量或者关系式)

它的口诀是：全一为一，其余为零

接下来画图演示它的原理：



本质上，是按位进行与预算，两个数在同一个为必须都为 1，结果才是 1，如果由任意一个不为 1，结构都是 0；图上数值 1 的值为 149，数值 2 的数值为 244，结果为 148，我们用代码来检验一下：

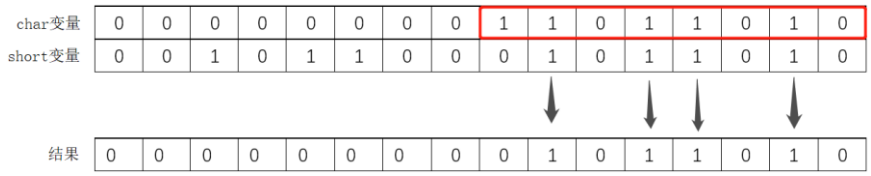
```c
#include <stdio.h>

int main()
{
    unsigned  char a = 149;
    unsigned char b = 244;
    unsigned char c = a & b;
    printf("c = %d\r\n", c);
    return 0;
}
```

```
E:\example.exe
c = 148

Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

结果符合我们的规律

还有这样一个情况，如果一个 char 型和一个 short 型的变量相与，变量类型长度不同的情况下相遇，会将长度短的变量补全为相同长度，补全的部分全部都是 0，

画图演示：



至于应该用什么类型来保存结果呢？这个实际根据情况决定。

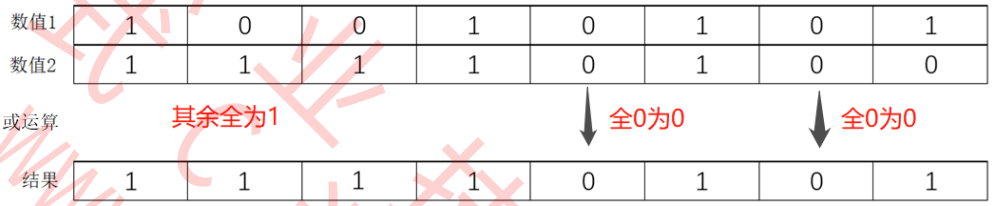
2-4-4.按位或

按位或的运算符是'|'，一根竖线，按位或的操作过程和按位与是类似的，但是运算规则不同，按位或是微观层面的逻辑或。

它的写法是这个样子：数值(变量或者关系式) | 数值(变量或者关系式)

它的口诀是：全零为 0，不同为 1

通过画图来直观说明：



在这张图中，同样是 149 与 244，相或之后，只有同时为 0 的位才变成了 0，其他位全部都变成 1 了。最终的结果，变成了 245，我们用代码来验证一下：

```c
#include <stdio.h>

int main()
{
    int a = 149;
    int b = 244;
    printf("结果是: %d\r\n", a | b);
    return 0;
}
```

```
E:\example.exe
结果是: 245

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

结果符合我们的预计，不同长度的变量类型进行按位或也和按位与一样，是将短的数据类型补齐成长的数据类型，再进行按位或，并根据需要保存在指定的变量类型当中。

2-4-5.按位取反

按位取反的运算符是'~'，它会把数值的每一位的进行取反操作，原来这一位是 1，取反后变成 0，原来是 1，取反后就变成 0，同样是微观层面的逻辑非运算。要取反的数值写在运算符的右边，可以是数值，变量或者关系式，运算符左边留空。

它的写法是这样：~数值(变量或关系式)

我们将数值 72 取反，来看一下结果，十进制 72 转换为二进制为 01001000，如下图：

0	1	0	0	1	0	0	0

按位取反就是每一位都进行取反，结果应该是 10110111。

1	0	1	1	0	1	1	1

我们将 10110111 转换成十进制，就是 183，用代码来验证一下：

```c
#include <stdio.h>

int main()
{
    char a = 72;
    unsigned char b = ~a;
    printf("结果是: %d\r\n", b);
    return 0;
}
```

```
E:\example.exe
结果是: 183

Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

结果符合设计，没有问题，但是如果我们在编程的时候不小心，代码写的不规范，那结果就有可能出问题，我们来看一下出问题的情况，先看代码：

```c
#include <stdio.h>

int main()
{
    char a = 72;
    printf("结果是: %d\r\n", ~a);
    return 0;
}
```

在这一段代码中，注意选中的部分~a，我们不用变量来接收，而是直接将其打印出来，看看结果是多少：

```c
#include <stdio.h>

int main()
{
    char a = 72;
    printf("结果是: %d\r\n", ~a);
    return 0;
}
```

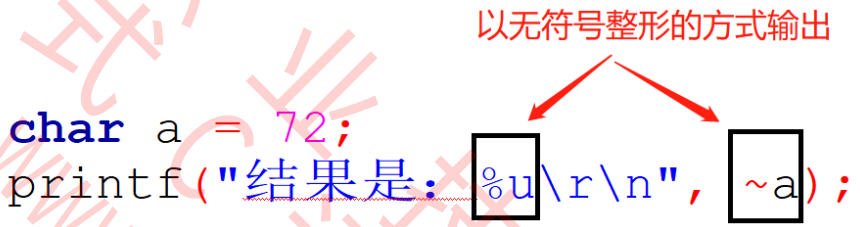
```
E:\example.exe
结果是: -73

Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

结果是-73，跟想象的不太一样，这是为什么呢？我们用无符号型将其打印出来看看是多少：

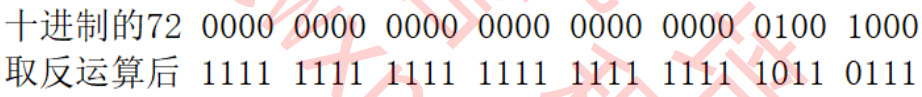
```c
#include <stdio.h>

int main()
{
    char a = 72;
    printf("结果是: %u\r\n", ~a);
    return 0;
}
```

```
E:\example.exe
结果是: 4294967223

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

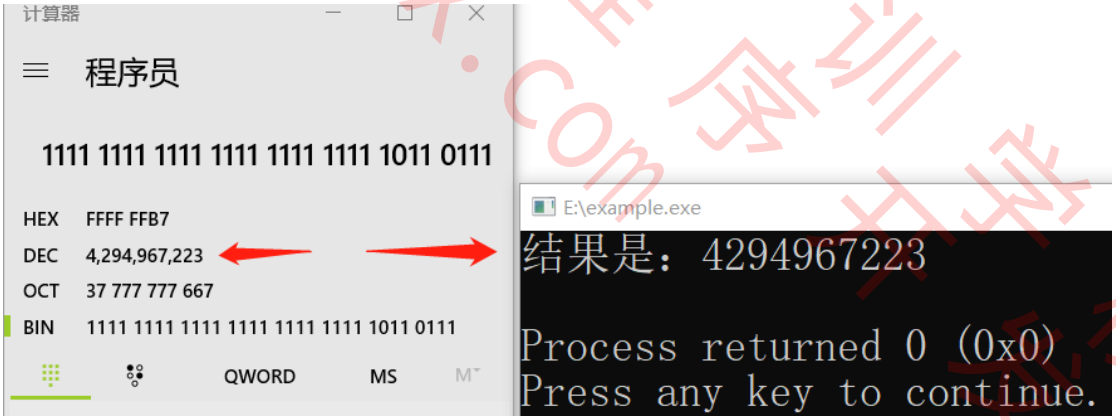
这个结果出乎意料，怎么这么大？这是有原因的，请看下图:



当我们在使用%u 对齐进行输出的时候，表示以无符号整形的方式进行输出，在实际的运算中，就不是将字符型的 a 进行取反，而是现将字符型的 a 扩展成整形，再对整形取反，然后输出，所以输出的结果才那么大，我们用位来进行演示，十进制 72 的二进制是 01001000，将其扩展成 32 位并取反，请看下图



我们将 1111 1111 1111 1111 1111 1111 1011 0111 换算成十进制:



结果跟程序的运行结果一致。

2-4-6.按位异或

按位异或的符号是'^'，这也是从微观层面对其数据进行操作

按位异或的写法如下：数值(变量或者关系式) ^ 数值(变量或者关系式)

按位异或的口诀是：不同为 1，相同为 0

不同为 1 相同为 0 就是说，哪怕两个位的值都是 1，但因为是相同的，所以结果为 0，而两个位

有一个是 1，另一个是 0，那就是不同，结果就为 1。我们用表格来演示，我们用 78 和 173 进行异或运算，78 的二进制是 01001110，173 的二进制是 10101101，将他们进行测试，表格：

数值78	0	1	0	0	1	1	1	0
数值173	1	0	1	0	1	1	0	1
异或运算	↓ 不同为1			↓ 相同为0		↓ 相同为0		
结果	1	1	1	0	0	0	1	1

结果 11100011，十进制值为 227。

我们在代码中演示一下：

```c
#include <stdio.h>

int main()
{
    char a = 78;
    char b = 173;
    unsigned char c = a ^ b;
    printf("result : %d\r\n", c);
    return 0;
}
```

```
E:\example.exe
result : 227

Process returned 0 (0x0)   execution
Press any key to continue.
```

现在我们分别在做一个实验，用 227 来异或上 78 和 173，看看能得出什么样的效果：

```c
#include <stdio.h>

int main()
{
    char a = 227;
    char b = 78;
    unsigned char c = a ^ b;
    printf("result : %d\r\n", c);
    return 0;
}
```

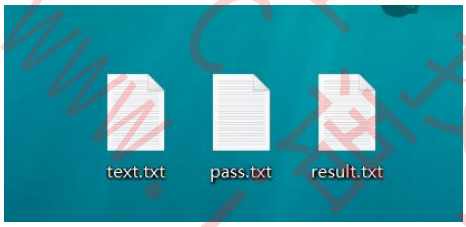
```
E:\example.exe
result : 173

Process returned 0 (0x0)   execution ti
Press any key to continue.
```

当 227 异或上 78 的时候，结果是 23，再来试一下：

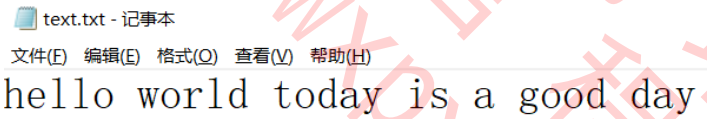
```c
#include <stdio.h>

int main()
{
    char a = 227;
    char b = 173;
    unsigned char c = a ^ b;
    printf("result : %d\r\n", c);
    return 0;
}
```

```
E:\example.exe
result : 78

Process returned 0 (0x0)   execution
Press any key to continue.
```

而异或上 173 的时候，又得到了 78，这就将两个数值都还原了，我们可以利用异或的这个特性，制作密码表，我在 E 盘新建三个文档，第一个文档 text 当中写入一段话，第二个文档用来保存异或后的加密内容，第三个文档用来保存解码后的内容，如下三个文档:



Text 文档中的内容如下，另外连个空白:

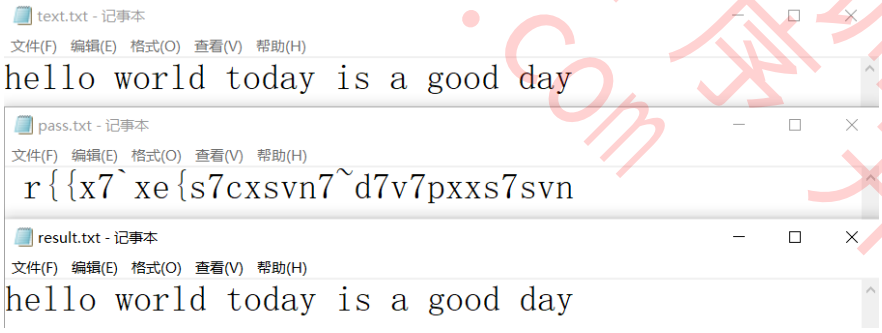


现在我来写代码，这一次的代码当中涉及到了文件操作，这里先不详细讲解文件操作，后面的课程会有，现在先看代码:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char buf;
    char val;
    char par;
    int cnt = 1;
    int fdtext = open("e:/text.txt", O_RDWR);
    int fdpass = open("e:/pass.txt", O_RDWR);
    int fdrult = open("e:/result.txt", O_RDWR);
    if(fdtext < 3 || fdpass < 3 || fdrult < 3)
    {
        printf("read file error\r\n");
    }
    else
    {
        printf("file open success\r\n");
    }
    while(cnt > 0) //循环读取直到读取结束
    {
        printf("--------------------\r\n");
        cnt = read(fdtext, &buf, 1); //以字节为单位读取原数据
        if(cnt > 0) //判断文件末尾
        {
            val = buf ^ 23; //将原数据异或
            write(fdpass, &val, 1); //异或后的值写入pass文件
            par = val ^ 23; //异或后的值还原
            write(fdrult, &par, 1); //还原值写入result文件
        }
    }
    close(fdtext);
    close(fdpass);
    close(fdrult);
    return 0;
}
```

在这一段代码中，我们主要关心 23 到 34 行:

26 行我们以单字节为单位循环读取 text 文件中的数据，在 29 行将其与 23 异或，异或后的值在第 30 行写入 pass 文件，异或后的值在第 31 行再次与 23 异或还原成之前的值，然后在第 32 行写入 result 文件，于是我们得到了异或后的文本 pass 和还原后的文本 result，现在我们来看文本的结果:



从上到下分别是源文件，加密文件和还原文件，这就是异或进行加密的用途，但是在实际的应用中，利用计算机的高速运算速度，采用穷举法，总能把异或加密算出来，时间长短而已，所以这种加密算法并不保险，更靠谱一点的将异或的结果在多次进行异或运算，可以从时间上来消耗破解者的时间。