

一、操作符

1. 算数操作符

+ - * / %

1. 除了%操作符之外，其他的几个操作符可以作用于整数和浮点数。
2. 对于/操作符，如果两个操作数都为整数，执行整数除法。而只要有浮点数执行的就是浮点数除法。
3. %操作符的两个操作数必须为**整数**。返回的是整数之后的余数。

若除数与被除数都是整数，结果也是整数。

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
int main() {
    int a = 5 / 2; //商2余1
    printf("a=%d\n", a);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

a=2

操作符讲解 (上) (全局范围)

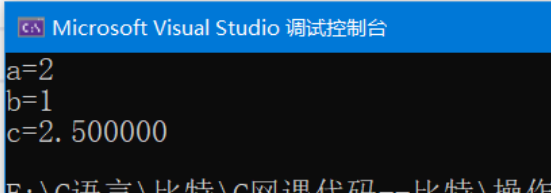
```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include<stdio.h>
3  int main() {
4      int a = 5 / 2; //商2余1
5      int b = 5 % 2;
6      printf("a=%d\n", a);
7      printf("b=%d\n", b);
8      return 0;
9  }
```

选择Microsoft Visual Studio 调试控制台

a=2
b=1

若除数与被除数有一个是浮点数，得到的就是小数。

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include<stdio.h>
3  int main() {
4      int a = 5 / 2; //商2余1
5      int b = 5 % 2;
6      double c = 5 / 2.0;
7      printf("a=%d\n", a);
8      printf("b=%d\n", b);
9      printf("c=%lf\n", c);
10     return 0;
11 }
```

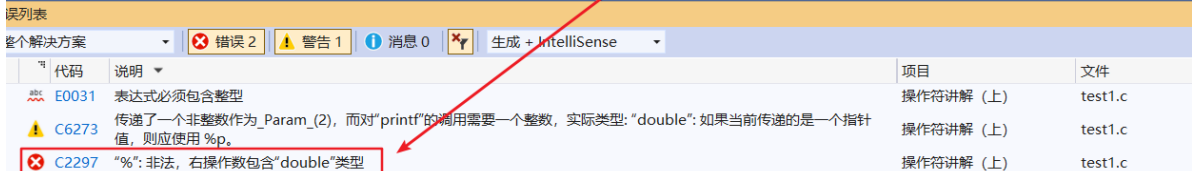


Microsoft Visual Studio 调试控制台

```
a=2
b=1
c=2.500000
```

%操作符两侧有一个不是整数时

```
1  #define _CRT_SECURE_NO_WARNINGS
2  #include<stdio.h>
3  int main() {
4      int a = 5 / 2; //商2余1
5      int b = 5 % 2;
6      double c = 5 / 2.0;
7      double d = 5 % 2.0;
8
9      printf("a=%d\n", a);
10     printf("b=%d\n", b);
11     printf("c=%lf\n", c);
12     return 0;
13 }
```



错误列表

代码	说明	项目	文件
E0031	表达式必须包含整型	操作符讲解 (上)	test1.c
C6273	传递了一个非整数作为 _Param_(2), 而对 "printf" 的调用需要一个整数, 实际类型: "double"; 如果当前传递的是一个指针值, 则应使用 %p。	操作符讲解 (上)	test1.c
C2297	"%": 非法, 右操作数包含 "double" 类型	操作符讲解 (上)	test1.c

2. 移位操作符

>>右移操作符 -->移动的是二进制位

<<左移操作符

(1) 右移操作符

举例

看一个例子：（正整数）

```
int a = 16;
int b = a >> 1;
printf("%d\n", b);
```

```
14 int a = 16;
15 // >>右移操作符
16 // 移动的是二进制位
17 // 16 --> 0000 0000 0000 0000 0000 0000 0001 0000
18 int b = a >> 1;
```

结果如何呢？

我们先来分析一下（内存中以补码存储）：

16 ---> 0000 0000 0000 0000 0000 0000 0001 0000
右移一位： 0000 0000 0000 0000 0000 0000 0001 0000
左侧补0： 0000 0000 0000 0000 0000 0000 0001 0000

右移操作符：

1、算术右移

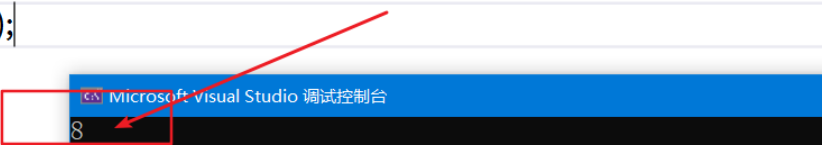
16是正数，符号位（最高位）是0，右边丢弃了一位（0），左边补原符号位（0）。

2、逻辑右移

右边直接丢弃，左边补0。

看一下运行结果：

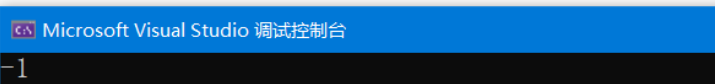
```
14 int a = 16;
15 // >>右移操作符
16 // 移动的是二进制位
17 // 16 --> 0000 0000 0000 0000 0000 0000 0001 0000
18
19 int b = a >> 1;
20 // 右移一位： 0000 0000 0000 0000 0000 0000 0000 1000 --> 2^3=8
21 printf("%d\n", b);
22
23 return 0;
```



补充

正数无论是算术右移还是逻辑右移，左侧补得都是0，我们用负数来测试一下电脑是算术右移还是逻辑右移：

```
25  int a = -1;
26  // -1 --> 1000 0000 0000 0000 0000 0000 0000 0000
27  // 算术右移: 1000 0000 0000 0000 0000 0000 0000 0000
28  // 1000 0000 0000 0000 0000 0000 0000 0000 --> -1
29  //逻辑右移: 1000 0000 0000 0000 0000 0000 0000 0000
30  // 0000 0000 0000 0000 0000 0000 0000 0000 --> +1
31  int b = a >> 1;
32  printf("%d\n", b);
```



可以看到，结果是-1，即：使用的是算术移位。

知识补充

整数的二进制表示有：原码、反码、补码。

存储到内存中的是补码。

(1) 正数的原码、反码、补码都是一样的，没有差别。

(2) 来看一下负数：（以-1为例）

最高位是符号位，正数为0，负数为1。

那么就可以写出-1的原码：（int类型4个字节，1个字节8个bit，那么int类型就是32bit）

1000 0000 0000 0000 0000 0000 0000 0001

反码 就是原码的符号位（最高位）不变，其余取反（1变0，0变1）：

1111 1111 1111 1111 1111 1111 1111 1110

补码 就是反码+1：

1111 1111 1111 1111 1111 1111 1111 1111

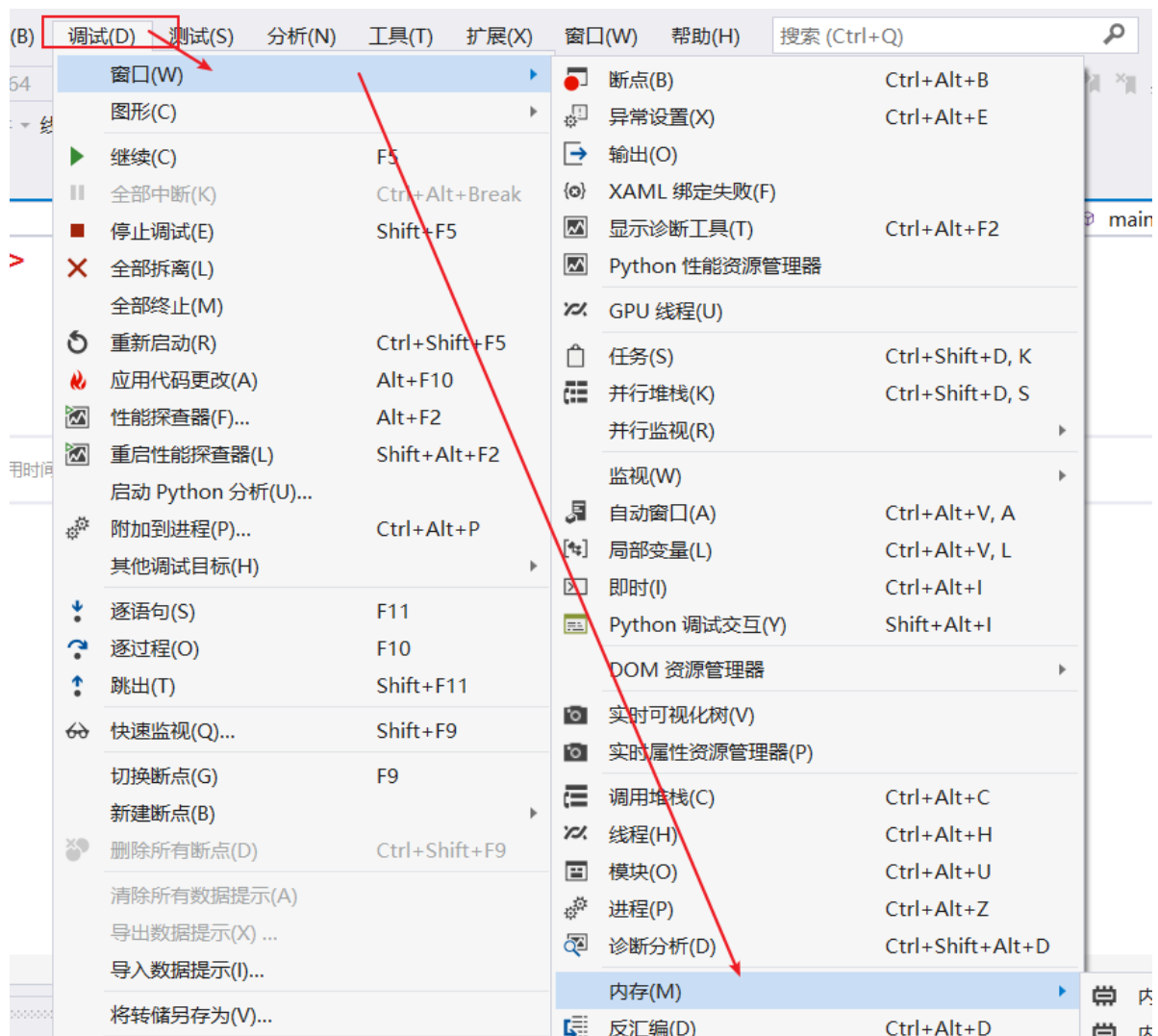
我们不妨去内存中看一下-1的补码：

按一下 F10，调出调试面板：

继续按 F10，当箭头走到这一行。

```
1  #include<stdio.h>
2  int main() {
3      int a = -1;
4      int b = a >> 1; 已用时间 <= 1ms
5      printf("%d\n");
6      return 0;
7  }
```

然后我们调出内存。（调试-->窗口-->内存）



四个可以选择一个。

点开之后输入 &a，将a的地址提取出来：

内存 1	
地址:	<code>&a</code>
<code>0x000000C93DCFFC64</code>	<code>ff ff ff ff d3 8e c2 87 f9 7f</code>
<code>0x000000C93DCFFC77</code>	<code>00 00 00 00 00 00 00 00 00 00</code>
<code>0x000000C93DCFFC8A</code>	<code>00 00 00 00 00 00 00 29 22 19</code>
<code>0x000000C93DCFFC9D</code>	<code>7f 00 00 39 22 19 dc 39 39</code>

敲回车：

内存 1	
地址:	<code>0x000000C93DCFFC64</code>
<code>0x000000C93DCFFC64</code>	<code>ff ff ff ff d3 8e c2 87 f9 7f 00 00</code>
<code>0x000000C93DCFFC77</code>	<code>00 00 00 00 00 00 00 00 00 00</code>
<code>0x000000C93DCFFC8A</code>	<code>00 00 00 00 00 00 00 29 22 19 dc</code>

可以看见，有8个f。

一个f是15，15换成二进制是1111。

8个f，就是32个1。

再回到之前的运算。

-1的补码是1111 1111 1111 1111

然后右移一位，最右边的1溢出，最左边补符号位（前面已经验证电脑默认算术右移），然后就可以得到：

1111 1111 1111 1111

所以最终输出-1。（输出的是原码）

(2) 左移操作符

举例

同样来看个例子：

```
int a = 5;
int b = a << 1;
printf("%d\n", b);
```

分析

写出5的二进制：

0000 0000 0000 0000 0000 0000 0000 0101

现在将它左移一位，最高位溢出，右边补0即可。

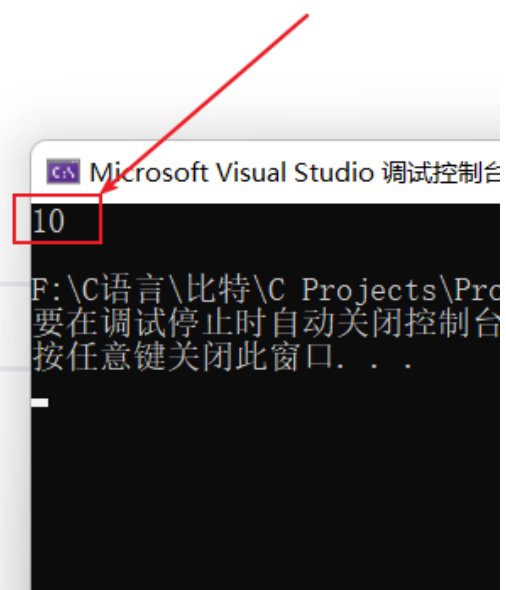
现在就可以得到：

0000 0000 0000 0000 0000 0000 0000 1010

转为十进制，就是10。

来看一下输出结果：

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



(3) 警告

对于移位运算符，不要移动负数位，这个是标准未定义的。

例如：

```
int num=10;
num>>-1;
```

Error!!!

3.位操作符

& 按位与

| 按位或

^ 按位异或

它们的操作数必须是整数。“位”是“二进制位”。

(1) 按位与

举个例子：

```
int a = 3;
int b = 5;
int c = a & b;
```

运算：

两个有一个为0即为0，两个同时为1才为1。

先写出3和5的二进制位：

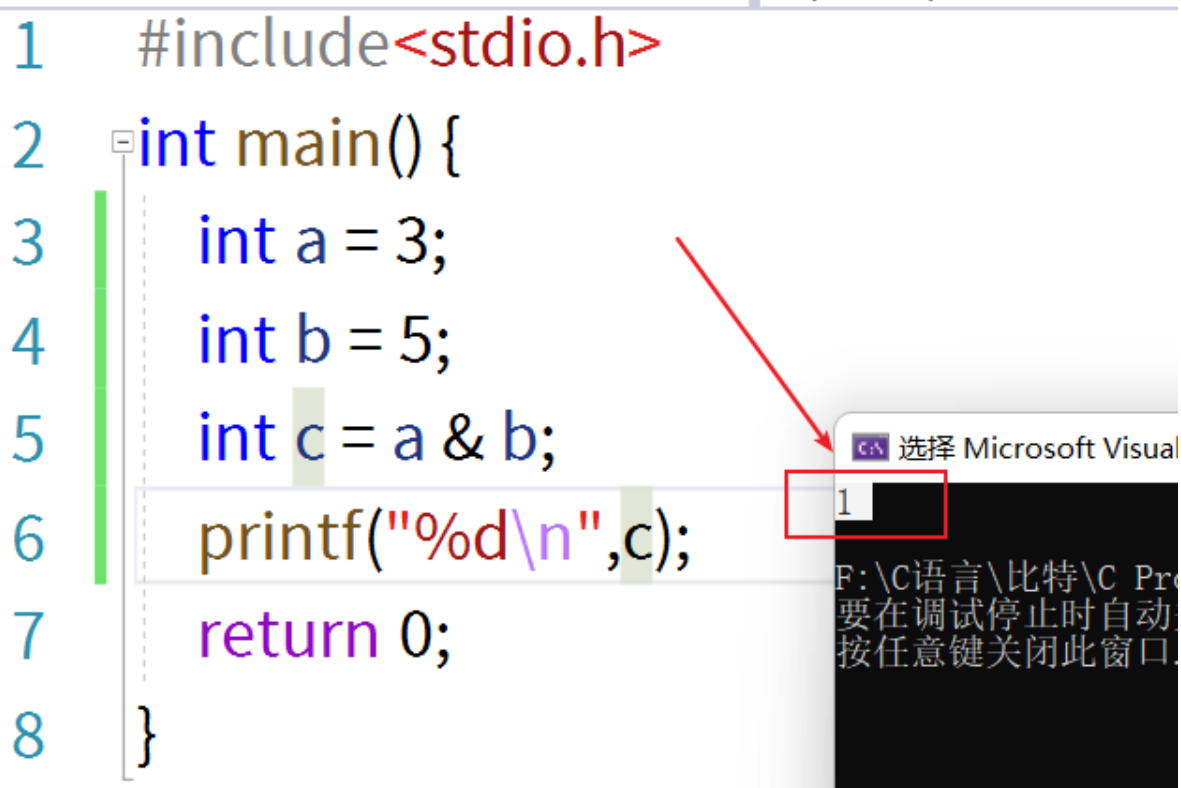
3: 0000 0000 0000 0000 0000 0000 0000 0011

5: 0000 0000 0000 0000 0000 0000 0000 0101

0000 0000 0000 0000 0000 0000 0000 0001

计算结果为1。

输出结果看一下：

The image shows a screenshot of a C program being executed. On the left, the source code is displayed with line numbers 1 through 8. The code includes <stdio.h>, defines a main function, and declares three integer variables: a (value 3), b (value 5), and c (value a & b). It then prints the value of c using printf and returns 0. On the right, a console window shows the output of the program, which is the number 1. A red arrow points from the variable 'c' in the code to the output '1' in the console. The console window also shows a prompt to '选择 Microsoft Visual' and some file paths.

⊗我们都是用补码运算，如果给的是负数，需要算出负数的补码，然后再进行运算。

(2) 按位或

举个例子：

```
int a =3;  
int b=5;  
int c=a | b;
```

运算：

两个有一个为1则为1，同时为0才为0。

先写出3和5的二进制位：

3: 0000 0000 0000 0000 0000 0000 0000 0011

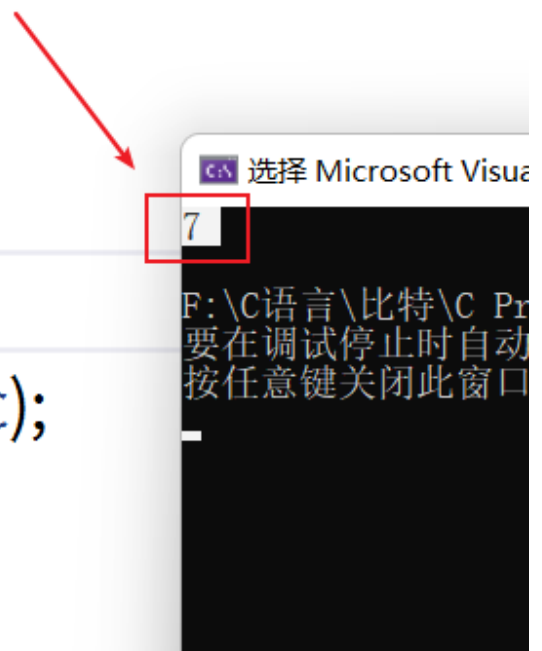
5: 0000 0000 0000 0000 0000 0000 0000 0101

0000 0000 0000 0000 0000 0000 0000 0111

计算结果为7。

输出结果看一下：

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



(3) 按位异或

举个例子：

```
int a=3;
int b=5;
int c=a^b;
```

运算：

对应二进制位若相同则为0，不同则为1。

先写出3和5的二进制位：

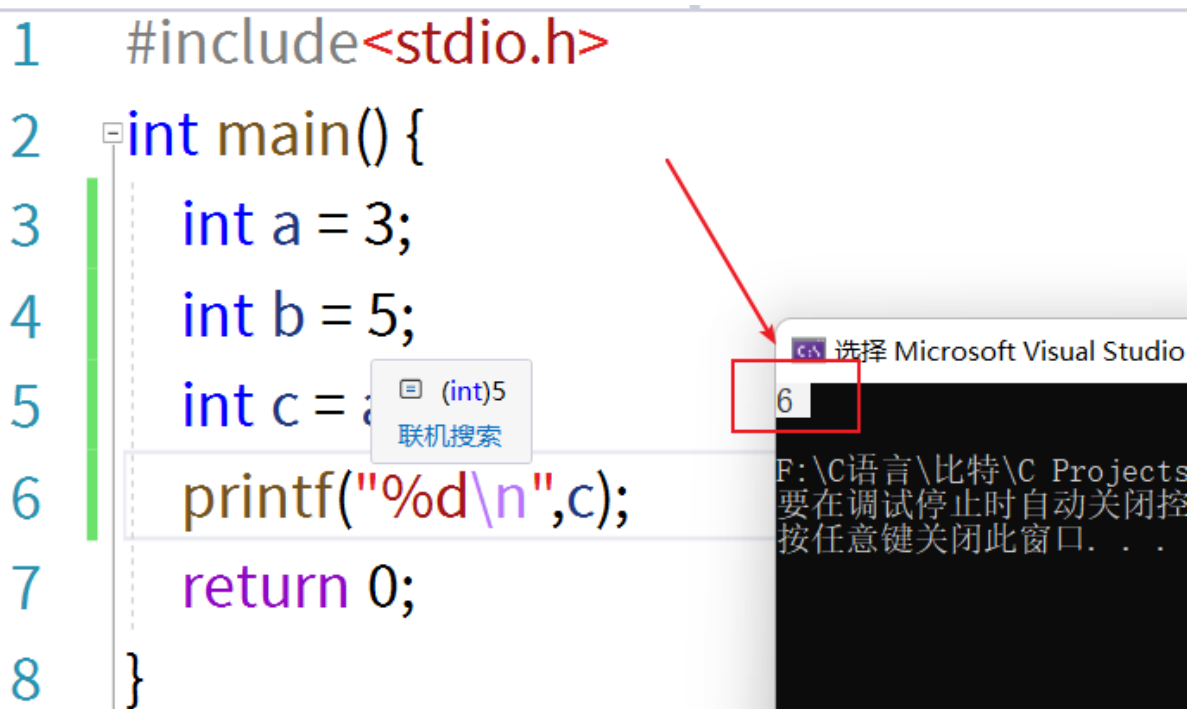
3: 0000 0000 0000 0000 0000 0000 0000 0011

5: 0000 0000 0000 0000 0000 0000 0000 0101

0000 0000 0000 0000 0000 0000 0000 0110

计算结果为6。

输出结果看一下：



The screenshot shows a C program in Visual Studio. The code is as follows:

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

A red arrow points from the code to a console window. The console window shows the output '6'. A tooltip for the variable 'c' shows its value as '(int)5' and a '联机搜索' (Online Search) button.

(4) 案例

案例一

这里讲一道面试题：

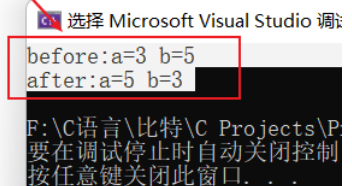
不能创建临时变量（第三个变量），实现两个数的交换。

如果 创建临时变量，是最基础的运算：

```
int a=3;
int b=5;
int temp=0;
printf("before:a=%d b=%d\n",a,b);
temp=a;
a=b;
b=temp;
printf("after:a=%d b=%d\n",a,b);
```

看一下结果：

```
1  #include<stdio.h>
2  int main() {
3      int a = 3;
4      int b = 5;
5      int temp = 0;
6      printf("before:a=%d b=%d\n", a, b);
7      temp = a;
8      a = b;
9      b = temp;
10     printf("after:a=%d b=%d\n", a, b);
11     return 0;
```



选择 Microsoft Visual Studio 调
before:a=3 b=5
after:a=5 b=3
F:\C语言\比特\C Projects\P
要在调试停止时自动关闭控制
按任意键关闭此窗口...

上面我们创建了临时变量temp来进行运算。

现在不用临时变量，该如何计算呢？

1) 加减法 计算

```
int a=3;
int b=5;
int temp=0;
printf("before:a=%d b=%d\n",a,b);
a=a+b;
b=a-b;
a=a-b;
printf("after:a=%d b=%d\n",a,b);
```

其实原理很简单。

将a与b全部倒进a，此时a就是它们的 和。

这时候拿这个 和，减去b，就可以得到a的原来的值，存入b中。

然后再拿这个和，减去此时的b（最开始的a值），就得到了b最开始的值，存入a中。

2) 异或法 计算

上面我们用的是整形值，占4个字节，有限空间。

当a与b的值特别大，相加之后超出了整形的最大值，这时候就会有值丢失。

所以第一种方法会有溢出的可能。

```
int a=3;
int b=5;
int temp=0;
printf("before:a=%d b=%d\n",a,b);
a=a^b;
b=a^b;
a=a^b;
printf("after:a=%d b=%d\n",a,b);
```

写出二进制，这里就写后三项了。

- 第一步：

a: 011

b: 101

a=a^b: 110

- 第二步：

a: 110

b: 101

b=a^b: 011 --> 3

- 第三步

a: 110


b: 011

a=a^b: 101 --> 5

可以这样理解：

a与b异或产生密码，存入a。

这个密码如果和b异或可以得到原来a的值，这个密码如果和现在的b（原来的a）异或可以得到原来的b。

 注意：

在异或运算中，不可能产生溢出。因为没有进位的可能。相同为0，相异为1，产生的结果无非是1或者0。

现实中写代码，这种方法，执行效率不高，可读性差。不建议用这种方法。

为了应付面试，只能了解啦。（可以不用，但不能不会）

案例二

再来看一道题：

编写代码实现：求一个整数存储在内存中的二进制中1的个数。

1) 错误示范

乍一看有点懵，不过之前我们会遇到这种题目：

输出123的每一位。

123是十进制数，我们可以用 $123\%10$ 得到123的个位数 3，然后用 $123/10$ 可以得到 12（将3去除）。

然后循环操作，就可以分别输出1、2、3了。

那么这一题，可以输出二进制数，然后做一个判断，如果是1，就计数。输出方法和上面一样，只不过除的是2而已。

```
int num =0;
int count =0;
scanf("%d",&num);    //3--011

//统计num中的补码有几个1
while(num){ //num不为0就继续执行
    if(num%2==1){
        count++;
        num=num/2;
    }
}
printf("%d\n",count);
```

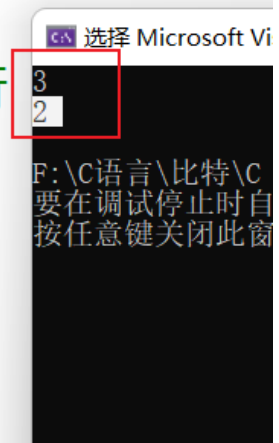
如果输入3，那么最终输出（二进制1的个数）就是2。

看一下：

```
int num = 0;
int count = 0;
scanf("%d", &num); //3--011
```

//统计num中的补码有几个1

```
while (num) { //num不为0就继续执行
    if (num % 2 == 1) {
        count++;
        num = num / 2;
    }
}
```



其实上面的方法是有缺陷的，如果是负数，那么就会出错。（不解释了，自己算一下就知道了）

换一种方法。

2) 方法1

既然要数出来二进制的1，那么能否将二进制都拿到手呢？

以3为例，3的二进制为：0000 0000 0000 0000 0000 0000 0000 0011

此时将它和1按位与运算。

1的二进制为：0000 0000 0000 0000 0000 0000 0000 0001

3&1：0000 0000 0000 0000 0000 0000 0000 0001

注意观察最后一位的结果变化。

可以发现一个规律：**如果该位置与1按位与运算，如果得到结果为1，那么该位置就一定是1。如果该位置是0，那么结果就是0。**

计算第一次就可以判断最后一位是不是1。（num>>0）

接下来想看一下倒数第二位，就可以让该二进制右移一位（num>>1），再与1做按位与运算。

然后想看一下倒数第三位，就可以让该二进制右移两位（num>>2），再与1做按位与运算。

如果按位与的结果是1，就表示该位置是1，记录一下即可。

代码：

```

int num =0;
int count =0;
scanf("%d",&num);    //3--011
int i=0;
for(i=0;i<32;i++){
    if(1==((num>>i)&1)){
        count++;
    }
}
printf("%d\n",count);

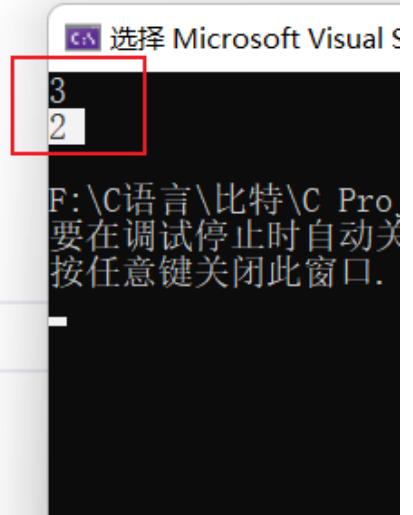
```

看一下最后结果：（以3为例）

```

int num = 0;
int count = 0;
scanf("%d", &num); //3--011
int i = 0;
for (i = 0; i < 32; i++) {
    if (1 == ((num >> i) & 1)) {
        count++;
    }
}
printf("%d\n", count);

```



再看一下-1:

```

int num = 0;
int count = 0;
scanf("%d", &num); //3--011
int i = 0;
for (i = 0; i < 32; i++) {
    if (1 == ((num >> i) & 1)) {
        count++;
    }
}
printf("%d\n", count);
return 0;
}

```

选择 Microsoft V

-1
32

F:\C语言\比特\C
要在调试停止时自
按任意键关闭此窗

3) 方法2

其实还有一种方法。

```

int num=-1;
int i=0;
int count=0;//计数
while(num){
    count++;
    num=num&(num-1);
}
printf("二进制中1的个数=%d\n",count);

```

4.赋值操作符

(1) 赋值操作符

赋值操作符是一个很棒的操作符，它可以让你得到一个你之前不满意的值。也就是你可以给自己重新赋值。

变量在创建的时候给它一个值，叫 **初始化**。当变量已经有了的时候，再给它一个值，叫 **赋值**。

初始化会开辟新的空间，而赋值不会。

举个最简单的例子：

```
int weight=120;  
weight=89;//不满意就赋值  
double salary=10000.0;  
salary=20000.0;//使用赋值操作符赋值
```

赋值操作符可以连续使用：

```
int a=10;  
int x=0;  
int y=20;  
a=x=y+1;//连续赋值
```

不建议连续赋值，不易于理解。（连续赋值语法上是支持的）

这样的代码更加简洁易于调试：

```
x=y+1;  
a=x;
```

赋值是一个`=`，而判断是两个`==`。

(2) 复合赋值符

`+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `|=` `^=`

这些运算都可以写成复合的效果。

比如：

```
int a=10;  
a=a+2;  
a+=2;//复合赋值符（和上面一行的意思一致）  
a=a>>1;  
a>>=1;//复合赋值符（和上面一行的意思一致）
```

5.单目操作符

！ 逻辑反操作

- 负值

+ 正值

& 取地址

sizeof 操作数的类型长度（以字节为单位）

~ 对一个数的二进制按位取反

-- 前置、后置--

++ 前置、后置++

* 间接访问操作符（解引用操作符）

(类型) 强制类型转换

单目操作符：只有一个操作数。

双目操作符：有两个操作数。比如 `a+b`，`+` 左右两侧都有数值。

(1) 逻辑反操作


! 逻辑反操作（真变假，假变真）

例如：

```
int a=0;
printf("%d\n", !a);
```

得到的结果就是1：

```
int a = 0;
printf("%d\n", !a);
return 0;
```



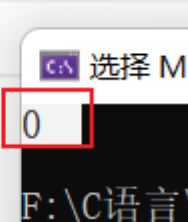
这里判断的数值不是局限于1和0，非假（0）即真。

比如此时a等于10，那么 `!a` 的数值也是0：

```
int a=10;
printf("%d\n", !a);
```

看一下结果：

```
int a = 10;
printf("%d\n", !a);
return 0;
```



所以这里的 `!` 可以这样来用（为语句条件做判断）：

①a为假打印输出：

```
if(!a){ //a为假, !a为真。如果a为假, 就可以打印输出。  
    printf("haha");  
}
```

②a为真打印输出:

```
if(a){  
    printf("haha");//a为真就打印输出  
}
```

(2) 正值与负值

正值 +

负值 -

这个很简单, 正值一般不写。

```
int a=-5;  
a=-a;
```

(3) 取地址和解引用

& 取地址 (取出任意变量或符号的地址)

* 解引用

取地址:

```
int a=10;  
&a; //取地址操作符
```

解引用:

```
int a=10;  
int* p=&a;  
*p; //解引用操作符
```

取地址操作符一般和指针一起使用。

```
int a=10;
int* p=&a; //将a变量的地址拿出来，存储到p变量里面。p是存放地址的变量，叫指针变量，类型为int*
*p=20; //解引用操作符，通过p找到所指的对象，即a。然后将20赋值给a。
```

(4) sizeof

sizeof 操作数的类型长度（以字节为单位）

sizeof 计算的是变量所占内存空间的大小，单位是字节。

```
int a=10;
char c='r';
char* p=&c;
int arr[10]={0};
printf("%d\n",sizeof(a)); //4
printf("%d\n",sizeof(c)); //1
printf("%d\n",sizeof(p)); //指针大小4字节（32平台）或者8字节（64平台）
printf("%d\n",sizeof(arr)); //数组里面10个元素，每个元素是整形，一个整形是4个字节，10个就是40个字节
```

输出看一下：

```
int a = 10;
char c = 'r';
char* p = &c;
int arr[10] = { 0 };
printf("%d\n", sizeof(a)); //4
printf("%d\n", sizeof(c)); //1
printf("%d\n", sizeof(p)); //指针
printf("%d\n", sizeof(arr)); //数
```

C:\ 选择 Microsoft Visual Studio 调

4
1
8
40

F:\C语言\比特\C Projects\
要在调试停止时自动关闭控制
按任意键关闭此窗口...

当然，sizeof 里面也可以写类型。

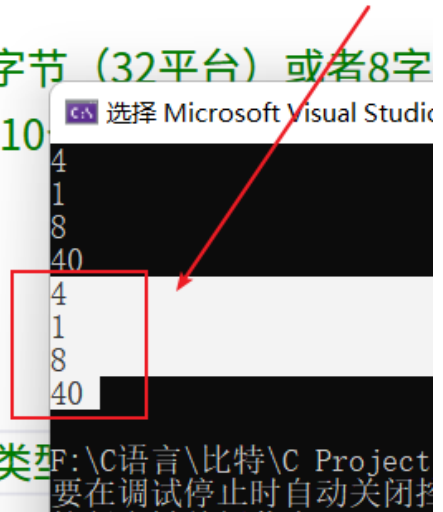
比如：

```
printf("%d\n",sizeof(int)); //4
printf("%d\n",sizeof(char)); //1
printf("%d\n",sizeof(char*)); //4或8
printf("%d\n",sizeof(int[10])); //数组类型就是去除数组名字之后 4*10=40
```

输出看一下：

```
char p = &c,  
int arr[10] = { 0 };  
printf("%d\n", sizeof(a)); //4  
printf("%d\n", sizeof(c)); //1  
printf("%d\n", sizeof(p)); //指针大小4字节（32平台）或者8字  
printf("%d\n", sizeof(arr)); //数组里面10
```

```
printf("%d\n", sizeof(int)); //4  
printf("%d\n", sizeof(char)); //1  
printf("%d\n", sizeof(char*)); //4或8  
printf("%d\n", sizeof(int[10])); //数组类型
```

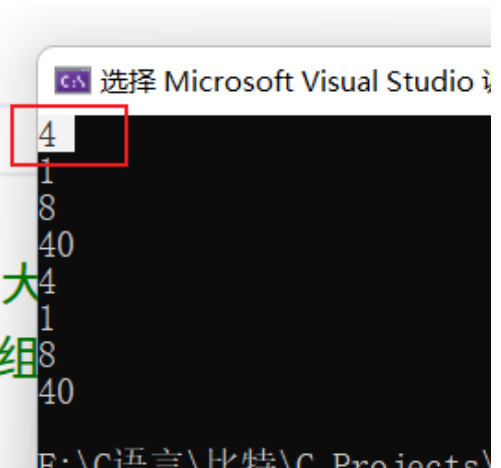


可以通过 类型 或者 变量名 来计算大小。

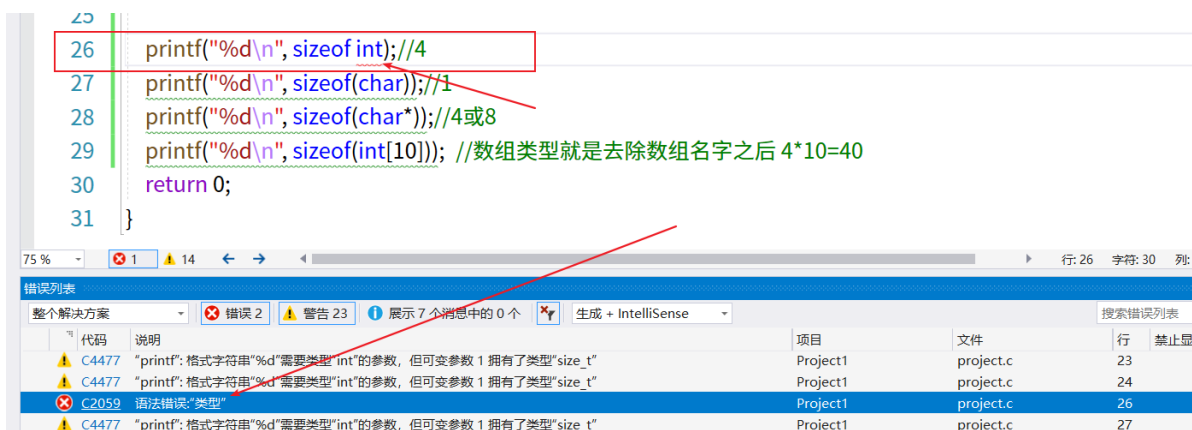
注意：如果 sizeof 计算的是变量名，则可以省略后面的括号。如果计算的是类型，则不可以。

变量名：

```
char p = &c,  
int arr[10] = { 0 };  
printf("%d\n", sizeof a); //4  
printf("%d\n", sizeof(c)); //1  
printf("%d\n", sizeof(p)); //指针大  
printf("%d\n", sizeof(arr)); //数组
```



类型：



案例:

补充一个小例子:

```
short s=0;
int a=10;
printf("%d\n", sizeof(s=a+5));
printf("%d\n", s);
```

不管a是什么类型, 加了5之后, 放进s里面, 都是2个字节。

sizeof 表达式不会真实运算, 仅仅只是摆设。

s的值不会发生变化。

printf() 会参与运算, sizeof() 里面的表达式不会参与运算。

看一下最后输出结果:

```
short s = 0;
int a = 10;
printf("%d\n", sizeof(s = a + 5));
printf("%d\n", s);
```



(5) 按位取反

~ 按位取反

按位取反, 对于负数的符号位也要取反!

看个例子:

```
int a=0;
printf("%d\n", ~a);
```

来分析一下：

~ 按位取反，是按照**二进制**取反。

a的值为0，二进制表示为：0000 0000 0000 0000 0000 0000 0000 0000

在C语言中，0相当于正数，原码、反码、补码都一样。

然后我们取反就得到：1111 1111 1111 1111 1111 1111 1111 1111（补码）

当我们输出打印的时候，是按照**原码**打印输出的。

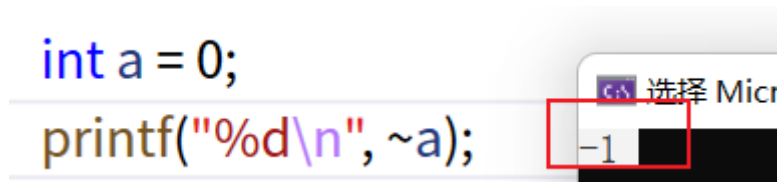
所以现在要求出它的原码。

原码--（0变1，1变0）-->反码---（反码+1）-->补码

反过来求得上面的反码是：1111 1111 1111 1111 1111 1111 1111 1110（反码）

原码就是：1000 0000 0000 0000 0000 0000 0000 0001（原码）--> -1

最终输出结果就是-1。



案例：

接下来，我们来看一个小案例。

11的二进制是：0000 0000 0000 0000 0000 0000 0000 1011

现在**想让倒数第三位变成1**。

之前学了**按位或**，**两个有一个为1则为1，同时为0才为0。**

那么我们可以让11**按位或**这个**数**：0000 0000 0000 0000 0000 0000 0000 0100

既然这样，我们就可以轻松改变一个二进制位置上的数。

按位或的这个**数**，相当于把1左移了两位：

0000 0000 0000 0000 0000 0000 0000 0000 (1)

0000 0000 0000 0000 0000 0000 0000 0100 ($1 \ll 2$)

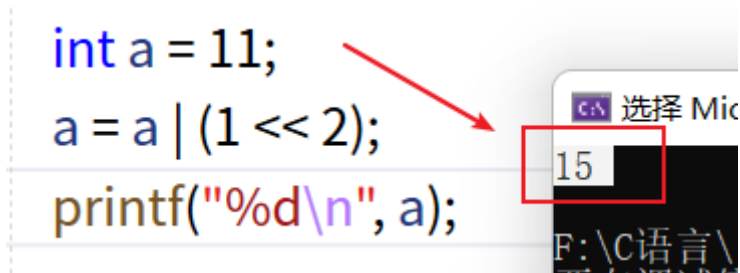
所以代码可以这样来写：

```
int a=11;
a=a|(1<<2);
```

最终得到结果：

0000 0000 0000 0000 0000 0000 0000 1111 (15)

输出看一下就是15:



看到这里，有小伙伴就要问了，这和 按位取反 有什么关系？

别急，我们现在想让刚才变化的位置，就是15倒数第三个位置上的1，变回去，变成0。

怎么办呢？

之前学了 按位与，两个有一个为0即为0，两个同时为1才为1。

我们将这个位置，按位与一个0，就可以变回去了！

那么我们可以让15 按位与 这个数：1111 1111 1111 1111 1111 1111 1111 1011

而按位与的这个数，相当于把1左移了两位，然后 按位取反：

0000 0000 0000 0000 0000 0000 0000 0001 (1)

0000 0000 0000 0000 0000 0000 0000 0100 (1<<2)

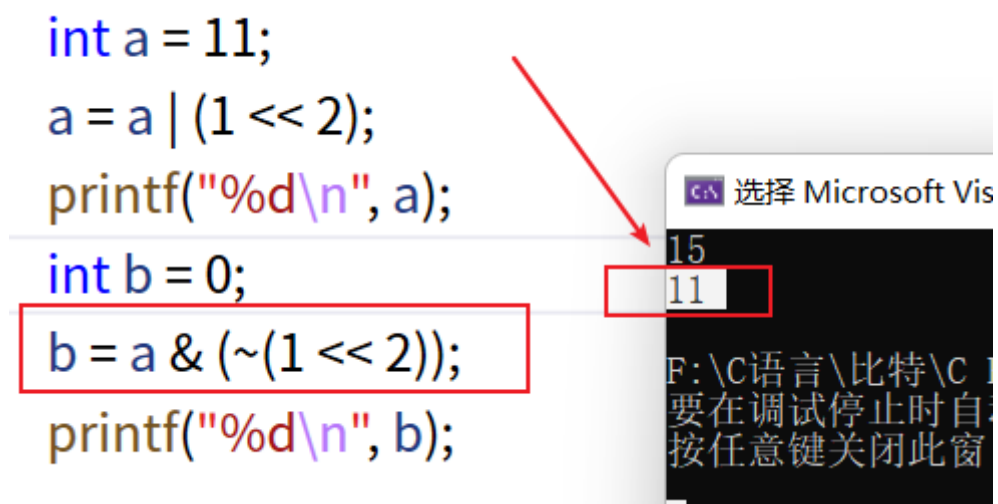
1111 1111 1111 1111 1111 1111 1111 1011 (~ (1<<2))

所以代码可以这样来写：

```
b=a&(~(1<<2));
```

最终得到：

0000 0000 0000 0000 0000 0000 0000 1011 (11)



(6) ++和--

-- 前置、后置--

++ 前置、后置++

①前置++：先加后使用。

```
int a=10;  
printf("%d\n", ++a);
```

输出结果：

```
int a = 10;  
printf("%d\n", ++a);
```

11

②后置++：先使用后加

```
int a=10;  
printf("%d\n", a++);
```

输出结果：

```
int a = 10;  
printf("%d\n", a++);
```

10

③前置--：先减后使用。

```
int a=10;  
printf("%d\n", --a);
```

输出结果：

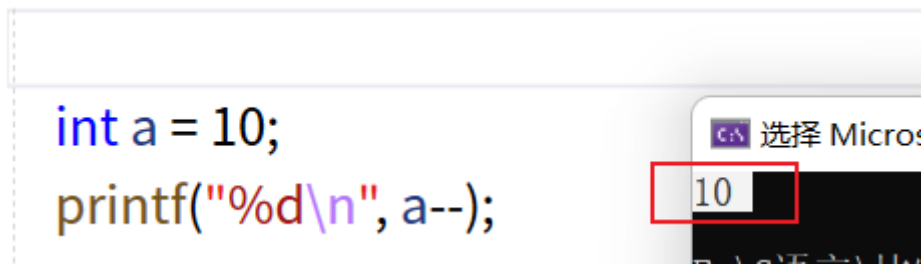
```
int a = 10;  
printf("%d\n", --a);
```

9

④后置--：先使用后减。

```
int a=10;  
printf("%d\n", a--);
```

输出结果：



(7) 强制类型转换

(类型) 强制类型转换

这个很简单，举个例子。

将3.14存入int类型的变量中，直接存是会报错的。

这时候，我们需要将它强制类型转换成int类型。

```
int a=(int)3.14;
```

(8) 小案例

巩固一下刚才学到的东西，我们做一个小案例。

```
#include<stdio.h>
void test1(int arr[]){
    printf("%d\\n",sizeof(arr)); //(3)
}
void test2(char ch[]){
    printf("%d\\n",sizeof(ch)); //(4)
}
int main(){
    int arr[10]={0};
    char ch[10]={0};
    printf("%d\\n",sizeof(arr)); //(1)
    printf("%d\\n",sizeof(ch)); //(2)
    test1(arr);
    test2(ch);
}
```

经过上面的学习，这一道题目变得非常简单。

第一空，arr数组里面有10个元素，每个元素都是int类型（4个字节），10个元素就是40字节。答案是40。

第二空，ch数组里面有10个元素，每个元素都是char类型（1个字节），10个元素就是10字节。答案是10。

第三空，将数组传递到函数，数组名相当于数组首元素地址，test1接收的时候，是用指针接收的，那么指针的大小是4（32平台）或8（64平台）。第四空一样。

输出看一下结果：

```
int arr[10] = { 0 };
char ch[10] = { 0 };
printf("%d\n", sizeof(arr)); //(1)
printf("%d\n", sizeof(ch)); //(2)
test1(arr);
```



6.关系操作符

> >=

< <=

!= 用于测试“不相等”

== 用于测试“相等”

这些关系运算符比较简单。

在编程的过程中，要小心 == 和 = 混淆出错。

7.逻辑操作符

&& 逻辑与

|| 逻辑或

区分**逻辑与**与**按位与**，**逻辑或**与**按位或**。

按位与和**按位或**是拿它们的二进制对应的位置进行**与**和**或**。

逻辑与和**逻辑或**是关注这个数的本身，是真还是假。

①逻辑与

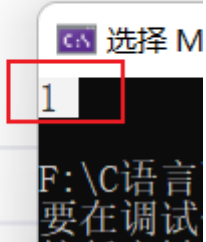
举个小例子：

```
int a=3;
int b=5;
int c=a && b;
```

分析：

a为真，b也为真。用 逻辑与 运算之后，结果为真（1）。

```
int a = 3;
int b = 5;
int c = a && b;
printf("%d\n",c);
```


A screenshot of a debugger window showing a variable 'c' with a value of '1'. The value is highlighted with a red box. The debugger interface includes a '选择 M' (Select M) button and a file path 'F:\C语言'.

有一个为假，结果就是假。两个同时为真才为真。比如：

```
int a=0;
int b=5;
int c=a && b;
```

看一下结果：

```
int a = 0;
int b = 5;
int c = a && b;
printf("%d\n",c);
```

A screenshot of a debugger window showing a variable 'c' with a value of '0'. The value is highlighted with a red box. The debugger interface includes a '选' (Select) button and a file path 'F:\C语言'.

②逻辑或

同样来举个小例子：

有一个为真，结果就为真。两个同时是假才为假。

```
int a=0;
int b=5;
int c=a || b;
```

看一下结果：

```
int a = 0;
int b = 5;
int c = a || b;
printf("%d\n",c);
```



一起来看个练习：

```
#include<stdio.h>
int main(){
    int i=0,a=0,b=2,c=3,d=4;
    i=a++ && ++b && d++;
    //i=a++ || ++b || d++;
    printf("a=%d\n b=%d\n c=%d\n d=%d\n",a,b,c,d);
    return 0;
}
```

①先看逻辑与那一行。

对于 逻辑与，左边的表达式如果计算结果是假，右边表达式无论算不算都是假。

所以此时只算了第一个表达式，后边的表达式都没有运算。

因此，a等于1，后边的都是原值。

看一下结果：

```
int i = 0, a = 0, b = 2, c = 3, d = 4;
i = a++ && ++b && d++;
//i=a++ || ++b || d++;
printf("a=%d\n b=%d\n c=%d\n d=%d\n", a, b, c, d);
```



如果将a的初始值换成1，像这样：

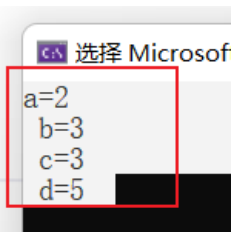
```
#include<stdio.h>
int main(){
    int i=0,a=1,b=2,c=3,d=4;
    i=a++ && ++b && d++;
    printf("a=%d\n b=%d\n c=%d\n d=%d\n",a,b,c,d);
    return 0;
}
```

最终所有表达式都要运算。

```
int i = 0, a = 1, b = 2, c = 3, d = 4;
```

```
i = a++ && ++b && d++;
```

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



②再看逻辑或那一行。

对于逻辑或，左边的表达式如果计算结果是真，右边表达式无论算不算都是真。

第一个a++结果是0，需要继续算。第二个++b结果是3，为真，后边不用算了。

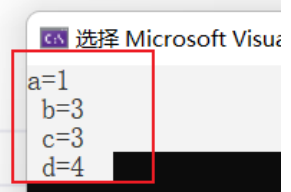
看一下：

```
int i = 0, a = 0, b = 2, c = 3, d = 4;
```

```
// i = a++ && ++b && d++;
```

```
i = a++ || ++b || d++;
```

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



8. 条件操作符

三目操作符。

```
exp1?exp2:exp3
```

表达式1结果为真，就计算表达式2的结果。

如果表达式1结果为假，就计算表达式3的结果。

做个小练习：

```
if(a>5){  
    b=3;  
}else{  
    b=-3;  
}
```

将上面的代码用条件运算符来写：

```
b=(a>5?3:-3);
```

再举个例子：

```
int a=10;
int b=20;
int max=0;
if(a>b){
    max=a;
}else{
    max=b;
}
```

将上面的代码用条件运算符来写：

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

9.逗号表达式

exp1,exp2,exp3,...expn

逗号表达式，就是用逗号隔开的多个表达式。

从左到右依次执行，整个表达式的结果是最后一个表达式的结果。

来看代码：

```
int a=1;
int b=2;
int c=(a>b, a=b+10, a, b=a+1);
```

上面代码结果，c是多少呢？

第一个表达式 `a>b` 执行没有结果，第二个 `a=b+10` 算出结果 `a=12`，第三个表达式也是执行了没有结果，第四个得出 `b=13`。

整个逗号表达式的结果，是最后一个表达式的结果，即13。

输出看一下：

```
int a = 1;
int b = 2;
int c = (a > b, a = b + 10, a, b = a + 1);
printf("%d\n", c);
```



10.下标引用、函数调用和结构成员

(1) 下标引用操作符

[] 下标引用操作符

操作数：一个数组名+一个索引值

看个小案例：

```
int arr[10];    //创建数组
arr[9]=10;    //下标引用操作符
[]的两个操作数是arr和9。
```

这个操作符的用法在数组广泛应用，可以移步去[\(31条消息\) C语言基础--数组 雨翼轻尘的博客-CSDN博客](#)那一节。

(2) 函数调用操作符

() 函数调用操作符

接受一个或者多个操作数：第一个操作数是函数名，剩余的操作数就是传递给函数的参数。

看个小案例：

```
get_max(int x,int y){
    return x>y?x:y;
}
int main(){
    int a=10;
    int b=20;
    //调用函数的时候()就是函数调用操作符
    //()的操作数有三个：函数名get_max和a和b
    int max=get_max(a,b);
    printf("max=%d\n",max);
    return 0;
}
```

这个操作符在函数广泛应用，后期再对函数专门写一篇，别着急，先了解一下。

(3) 访问一个结构的成员

.结构体,成员名

-> 结构体指针->成员名

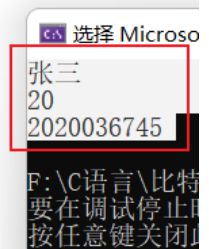
1)

举个小例子：


```
//创建一个结构体类型--struct Stu
struct Stu{
    //成员变量
    char name[20];
    int age;
    char id[20];
};
int main(){
    //使用struct Stu这个类型创建了一个学生对象s1，并初始化
    struct Stu s1={"张三",20,"2020036745"};
    printf("%s\n",s1.name);
    printf("%d\n",s1.age);
    printf("%s\n",s1.id);
    //结构体变量.成员名
    return 0;
}
```

输出看一下结果:

```
struct Stu {
    char name[20];
    int age;
    char id[20];
};
int main() {
    //使用struct Stu这个类型创建了一个学生对象s1，并初始化
    struct Stu s1 = { "张三",20,"2020036745" };
    printf("%s\n",s1.name);
    printf("%d\n",s1.age);
    printf("%s\n",s1.id);
}
```



选择 Microsoft
张三
20
2020036745
F:\C语言\比特
要在调试停止时
按任意键关闭以

2)

刚才我们是使用 `结构体变量.成员名` 来调用成员变量的。

还有一种方法:

```
ps=&s1; //将s1的地址找出来，存入指针变量ps中。
struct Stu* ps //ps是指针变量，类型是struct Stu。
```

合起来就是:

```
struct Stu* ps=&s1;
```

既然存好了，就拿出来用。

怎么调用呢?

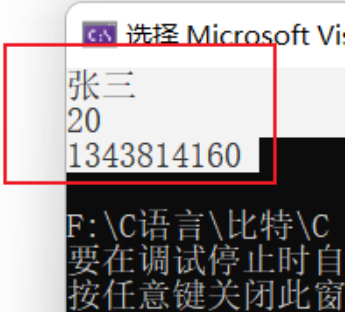
`(*ps).name` //ps是指针变量，*ps就是解引用。然后我们用它.成员变量，就可以调用了

看一下整体的代码：

```
//创建一个结构体类型--struct Stu
struct Stu{
    //成员变量
    char name[20];
    int age;
    char id[20];
};
int main(){
    //使用struct Stu这个类型创建了一个学生对象s1，并初始化
    struct Stu s1={"张三",20,"2020036745"};
    struct Stu* ps=&s1;
    printf("%s\n",(*ps).name);
    printf("%d\n",(*ps).age);
    printf("%d\n",(*ps).id);
    //结构体变量.成员名
    return 0;
}
```

看一下输出结果：

```
int main() {
    //使用struct Stu这个类型创建了一个学生对象s1，并初始
    struct Stu s1 = { "张三",20,"2020036745" };
    struct Stu* ps = &s1;
    printf("%s\n", (*ps).name);
    printf("%d\n", (*ps).age);
    printf("%d\n", (*ps).id);
    //结构体变量.成员名
}
```



3)

刚才 `(*ps).name` 的调用方式非常繁琐。

C语言给我们提供了另外一种方法：

`ps->name;` //指针->对象

和刚才的方法没有任何区别。

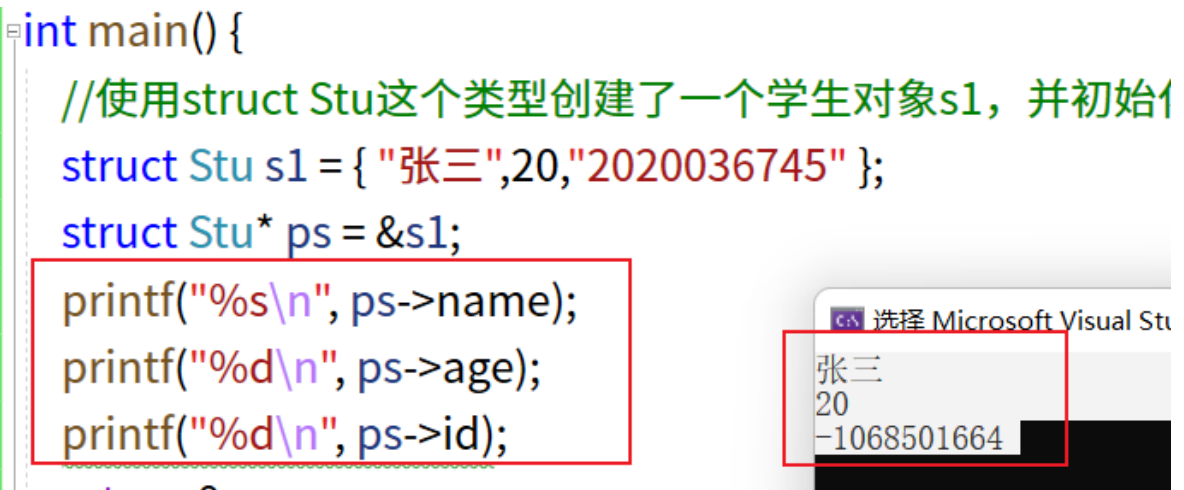
然后我们就可以这样调用了：

```
printf("%s\n",ps->name);
```

看一下整体的代码：

```
struct Stu{
    //成员变量
    char name[20];
    int age;
    char id[20];
};
int main(){
    //使用struct Stu这个类型创建了一个学生对象s1，并初始化
    struct Stu s1={"张三",20,"2020036745"};
    struct Stu* ps=&s1;
    printf("%s\n",ps->name);
    printf("%d\n",ps->age);
    printf("%d\n",ps->id);
    //结构体指针->成员名
    return 0;
}
```

看一下输出结果：



```
int main() {
    //使用struct Stu这个类型创建了一个学生对象s1，并初始化
    struct Stu s1 = { "张三",20,"2020036745" };
    struct Stu* ps = &s1;
    printf("%s\n", ps->name);
    printf("%d\n", ps->age);
    printf("%d\n", ps->id);
}
```

选择 Microsoft Visual Stu

张三
20
-1068501664

这个操作符在结构体那一节广泛应用，后期再对结构体专门写一篇，别着急，先了解一下。

需要的小伙伴可以关注后续文章。

二、表达式求值

学了这么多操作符，主要还是用于表达式求值。

表达式求值的顺序一部分是由操作符的 [优先级](#) 和 [结合性](#) 决定。

同样，有些表达式的操作数在求值的过程中可能需要转换为其他类型。

1.隐式类型转换

隐式类型转换

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**。

整型提升的意义：

表达式的整型运算要在CPU的相应运算器件内执行，CPU内整型运算器(ALU)的操作数的字节长度一般就是int的字节长度，同时也是CPU的通用寄存器的长度。

因此，即使两个char类型的相加，在CPU执行时实际上也要先转换为CPU内整型操作数的标准长度。

通用CPU (general-purpose CPU) 是难以直接实现两个8比特字节直接相加运算 (虽然机器指令中可能有这种字节相加指令)。所以，表达式中各种长度可能小于int长度的整型值，都必须先转换为int或unsigned int，然后才能送入CPU去执行运算。

(1) 案例一

来看一个案例：

```
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。

加法运算完成之后，结果将被截断，然后再存储于a中。

这样可能比较抽象，不易于理解，下面再来看一个例子：

```
int main(){
    char a=3;
    char b=127;
    char c=a+b;
    printf("%d\n",c);
}
```

输出结果是多少呢？

①来看第一行代码：

```
char a=3;
```

3是一个整数，32bit位，那么我们就可以写出它的二进制表达形式：

0000 0000 0000 0000 0000 0000 0000 0011

但是现在我们要将3存入char类型的变量中，char类型是1个字节，8bit位。

所以我们只能将32bit位截断，规则是挑最低位的字节。

那么就可以得到：

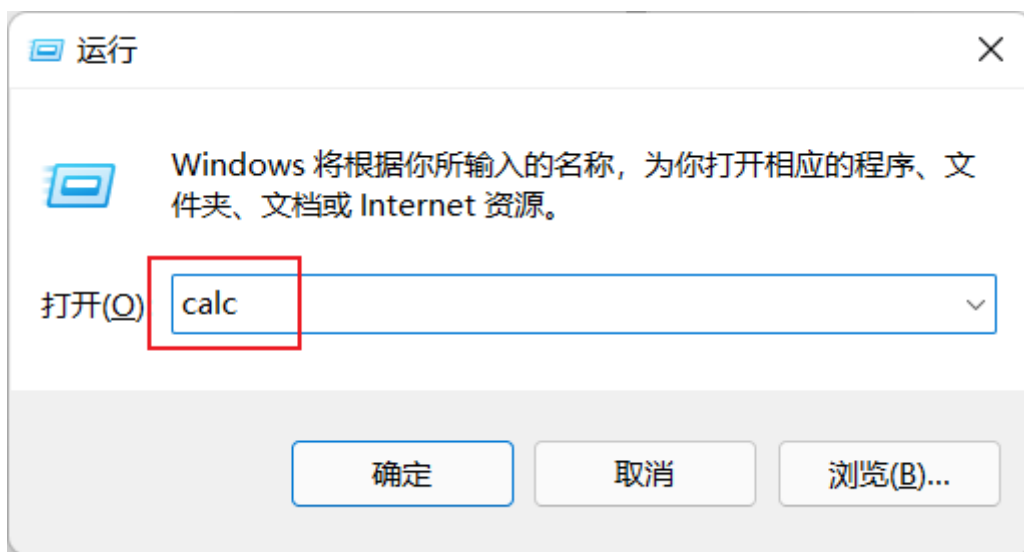
0000 0011

②再来看第二行代码：

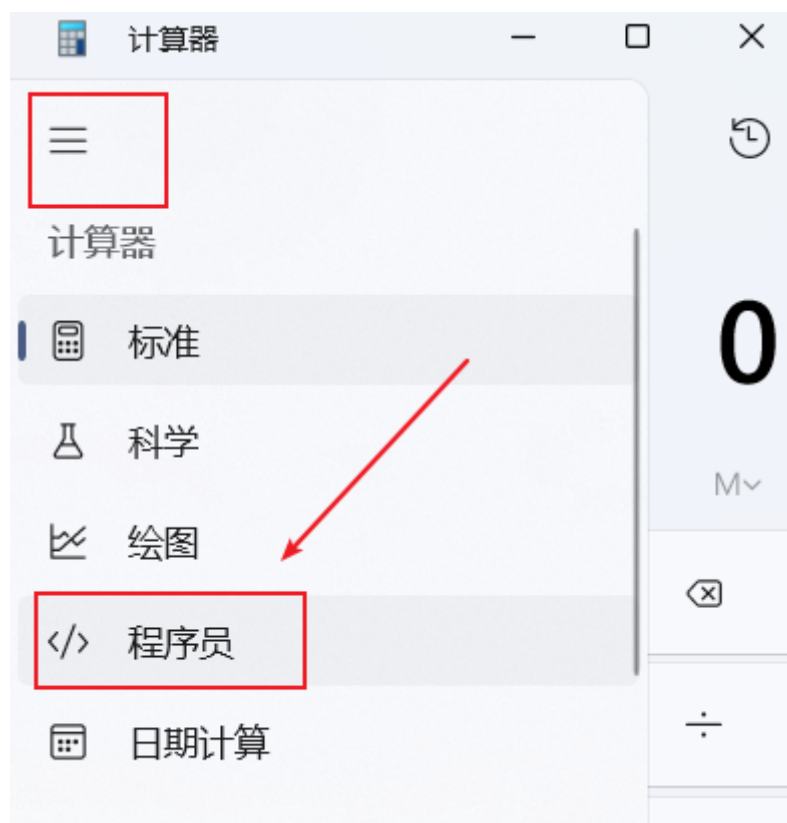
```
char b=127;
```

现在将127转为二进制表示。

可以借助电脑来帮我们计算，按一下 Windows + R 键，输入 calc 调出计算器。



调整到程序员模式。



输入127:



可以看到二进制:



所以127的二进制的表示为：

0000 0000 0000 0000 0000 0000 0111 1111

但是现在我们要将127存入char类型（1字节--8bit）的变量中，截断之后得到：

0111 1111

③再来看第三行代码：

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

a和b如何相加呢？

整型提升：为了获得精度，表达式中的字符和短整型操作数在使用之前被转换为普通整型。

👉 如何进行**整型提升**？

整型提升是按照变量的数据类型的符号来提升的。

<1>负数的整型提升

```
char c1=-1;
```

变量c1的二进制位（补码）中只有8个比特位：1111 1111

因为char是有符号的char，所以整型提升的时候，**高位补充符号位**，即1。

提升之后的结果是：

1111 1111 1111 1111 1111 1111 1111 1111

<2>正数的整型提升

```
char c2=1;
```

变量c2的二进制位（补码）中只有8个比特位：0000 0001

因为char是有符号的char，所以整型提升的时候，**高位补充符号位**，即0。

提升之后的结果是：

0000 0000 0000 0000 0000 0000 0000 0001

<3>无符号整数提升

高位补0。

再回到本案例。

既然现在的char是有符号的char。

那么现在我们可以认为8bit位的**最高位是符号位**。

例如a截断之后是：0000 0011

最高位是符号位，是0。

那么进行**整型提升**的时候，提升的是符号位（这里是0），所以前面24bit位补0。

就成了这样：

0000 0000 0000 0000 0000 0000 0000 0011

b截断之后是：0111 1111

最高位是符号位，是0。

整型提升之后：

0000 0000 0000 0000 0000 0000 0111 1111

然后再将a与b相加。

```
0000 0000 0000 0000 0000 0000 0000 0011
```

+

```
0000 0000 0000 0000 0000 0000 0111 1111
```

=

```
0000 0000 0000 0000 0000 0000 1000 0010
```

粗略看一下进位（二进制里面逢2进1）：

$$\begin{array}{r} 0011 \\ + 0111 \\ \hline 10000010 \end{array}$$

现在得到这样的结果：

```
0000 0000 0000 0000 0000 0000 1000 0010
```

再将这个二进制放入c中。

c也是char类型变量，只能存8个bit位。

截断之后：1000 0010

④最后输出c

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

最后我们输出的c是以%d形式输出。

刚才得到的c是：1000 0010

最高位是符号位，即1。

现在需要对它**整型提升**，前面要补符号位1，就得到：

1111 1111 1111 1111 1111 1111 1000 0010（补码）

现在得到的是内存中存储的 **补码**！

真正打印出来的是 **原码**。

原码--（取反）--->反码---（+1）-->补码

算一下：

1111 1111 1111 1111 1111 1111 1000 0001 (反码)

1000 0000 0000 0000 0000 0000 0111 1110 (原码, 符号位不要动)

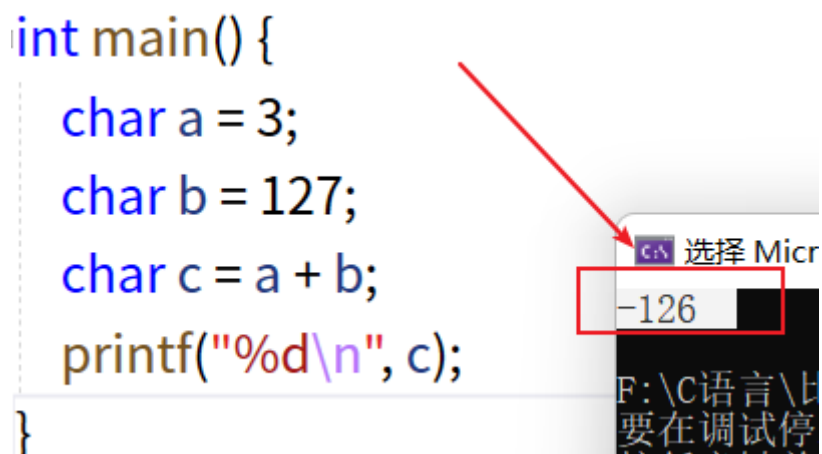
用计算器看一下:



最高位是1, 负数。

所以最终输出 -126。

在编译器中运行看一下:



(2) 案例二

上面我们已经了解了**整型提升**的知识, 下面来做个案例巩固一下。

```
int main(){
    char a=0xb6;
    short b=0xb600;
    int c=0xb6000000;
    if(a==0xb6){
        printf("a");
    }
}
```

```

    }
    if(b==0xb600){
        printf("b");
    }
    if(c==0xb6000000){
        printf("c");
    }
    return 0;
}

```

①先看a

```
char a=0xb6;
```

对于十六进制 (0~9, a~f) , a是10, 那么往后数, b就是11。

11的二进制为: 1011

6的二进制为: 0110

前面的0x是十六进制的标志而已, 不用管。

这样合在一起, 就可以得到a的二进制: 1011 0110

下面a要与0xb6比较。

```

if(a==0xb6){
    printf("a");
}

```

比较也是一种运算。

a也要发生**整型提升**。

a的二进制最高位是1, 进行整型提升的时候, 前面补最高位 (符号位) , 所以得到:

1111 1111 1111 1111 1111 1111 1011 0110

这样的话, 怎么可能和0xb6一样呢?

不能输出。

②再看b

```
short b=0xb600;
```

b是short类型, 最后比较的时候也要进行整型提升。

所以后边也不可能和0xb600相等。

也不能输出。

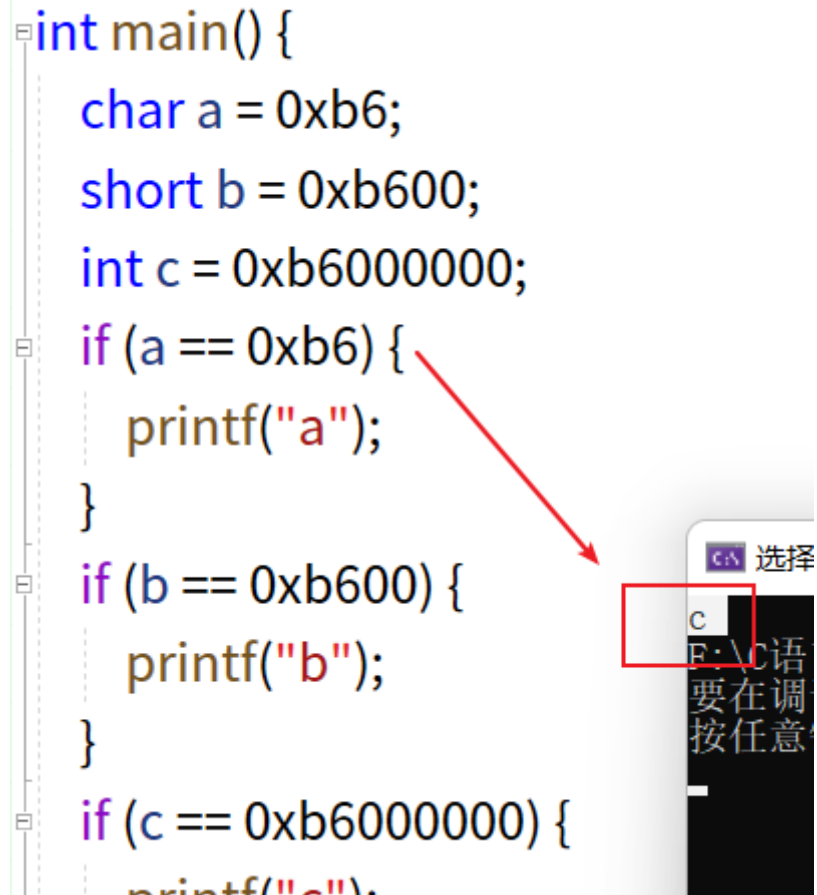
③再看c

```
int c=0xb6000000;
```

c本来就是整型，不需要进行整型提升。

所以最后可以输出。

看一下编辑器最后结果：



```
int main() {  
    char a = 0xb6;  
    short b = 0xb600;  
    int c = 0xb6000000;  
    if (a == 0xb6) {  
        printf("a");  
    }  
    if (b == 0xb600) {  
        printf("b");  
    }  
    if (c == 0xb6000000) {  
        printf("c");  
    }  
}
```

(3) 案例三

最后再来看一道例题。

```
int main(){  
    char c=1;  
    printf("%u\n",sizeof(c));  
    printf("%u\n",sizeof(+c));  
    printf("%u\n",sizeof(-c));  
    printf("%u\n",sizeof(!c));  
    return 0;  
}
```

c只要参与表达式运算，就会发生**整型提升**。

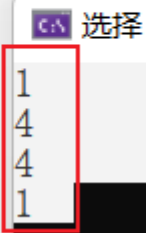
表达式 `+c`，会发生整型提升，计算的就是整型大小，所以 `sizeof(+c)` 是4个字节。

表达式 `-c`，会发生整型提升，所以 `sizeof(-c)` 也是4个字节。

但是 `sizeof(c)` 就是1个字节。

看一下编辑器结果：

```
int main() {
    char c = 1;
    printf("%u\n", sizeof(c));
    printf("%u\n", sizeof(+c));
    printf("%u\n", sizeof(-c));
    printf("%u\n", sizeof(!c));
}
```



2.算术转换

算术转换

如果某个操作符的各个操作数属于不同的类型，那么除非其中一个操作数的转换为另一个操作数的类型，否则操作就无法进行。下面的层次体系称为**寻常算术转换**。

```
long double
double
float
unsigned long int
long int
unsigned int
int
```

如果某个操作数的类型在上面这个列表中排名较低，那么首先要转换为另外一个操作数的类型后执行运算。

警告：但是算术转换要合理，要不然会有一些潜在的问题。

简单来说，如果float类型和double类型运算，需要先把float转换为double类型，再进行运算。

```
float f=3.14;
int num=f; //隐式转换，会有精度丢失
```

三、操作符的属性

复杂表达式的求值有三个影响的因素。

- 1.操作符的优先级
- 2.操作符的结合性
- 3.是否控制求值顺序

两个相邻的操作符先执行哪一个？取决于它们的优先级，如果两者的优先级相同，取决于它们的结合性。

1.操作符优先级

比如：

```
int main(){
    int a=10;
    int b=20;
    int c=b+a*3;
}
```

很明显，在表达式 `b+a*3` 中，乘法优先级高于加法，所以先计算 `a*3`，再计算加法。

然后再来看：

```
int main(){
    int a=10;
    int b=20;
    int c=b+a+3;
}
```

在表达式 `b+a+3` 中，优先级相同，自左向右。

初级运算符(`()`、`[]`、`->`、`.` 高于 **单目运算符** 高于 **算数运算符**（先乘除后加减） 高于 **关系运算符** 高于 **逻辑运算符**（不包括`!`） 高于 **条件运算符** 高于 **赋值运算符** 高于 **逗号运算符**。

位运算符的优先级比较分散。

除了**赋值运算符**、条件运算符、单目运算符三类的平级运算符之间的结合顺序是**从右至左**，其他都是从左至右。

C语言运算符优先级

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	--
	()	圆括号	(表达式) /函数名(形参表)	左到右	--
	.	成员选择 (对象)	对象.成员名	左到右	--
	->	成员选择 (指针)	对象指针->成员名	左到右	--
2	-	负号运算符	-表达式	右到左	单目运算符
	~	按位取反运算符	~表达式	右到左	
	++	自增运算符	++变量名/变量名++	右到左	
	--	自减运算符	--变量名/变量名--	右到左	
	*	取值运算符	*指针变量	右到左	
	&	取地址运算符	&变量名	右到左	
	!	逻辑非运算符	!表达式	右到左	
	(类型)	强制类型转换	(数据类型)表达式	右到左	--
	sizeof	长度运算符	sizeof(表达式)	右到左	--
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式	左到右	
	%	余数 (取模)	整型表达式%整型表达式	左到右	
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式	左到右	
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式	左到右	
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式	左到右	
	<	小于	表达式<表达式	左到右	
	<=	小于等于	表达式<=表达式	左到右	

优先级	运算符	名称或含义	使用形式	结合方向	说明
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式	左到右	
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10		按位或	表达式 表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12		逻辑或	表达式 表达式	左到右	双目运算符
13	?:	条件运算符	表达式1?表达式2: 表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	--
	/=	除后赋值	变量/=表达式	右到左	--
	=	乘后赋值	变量=表达式	右到左	--
	%=	取模后赋值	变量%=表达式	右到左	--
	+=	加后赋值	变量+=表达式	右到左	--
	-=	减后赋值	变量-=表达式	右到左	--
	<<=	左移后赋值	变量<<=表达式	右到左	--
	>>=	右移后赋值	变量>>=表达式	右到左	--
	&=	按位与后赋值	变量&=表达式	右到左	--
	^=	按位异或后赋值	变量^=表达式	右到左	--
	=	按位或后赋值	变量	=表达式	
15	,	逗号运算符	表达式,表达式,...	左到右	--

说明:

同一优先级的运算符，运算次序由结合方向所决定。

简单记就是：！ > 算术运算符 > [关系运算符](#) > && > || > 赋值运算符

操作符	描述	用法示例	结果类型	结合性	是否控制求值顺序
()	聚组	(表达式)	与表达式同	N/A	否
()	函数调用	rexp (rexp , ...,rexp)	rexp	L-R	否
[]	下标引用	rexp[rexp]	lexp	L-R	否
.	访问结构成员	lexp.member_name	lexp	L-R	否
—	后缀自减	lexp —	rexp	L-R	否
!	逻辑反	! rexp	rexp	R-L	否
~	按位取反	~ rexp	rexp	R-L	否
+	单目，表示正值	+ rexp	rexp	R-L	否
-	单目，表示负值	- rexp	rexp	R-L	否
++	前缀自增	++lexp	rexp	R-L	否
--	前缀自减	--lexp	rexp	R-L	否
*	间接访问	* rexp	lexp	R-L	否
&	取地址	& lexp	rexp	R-L	否
sizeof	取其长度，以字节表示	sizeof rexp sizeof(类型)	rexp	R-L	否
(类型)	类型转换	(类型) rexp	rexp	R-L	否
*	乘法	rexp * rexp	rexp	L-R	否
/	除法	rexp / rexp	rexp	L-R	否

*	乘法	rexp * rexp	rexp	L-R	否
/	除法	rexp / rexp	rexp	L-R	否
%	整数取余	rexp % rexp	rexp	L-R	否
+	加法	rexp + rexp	rexp	L-R	否
-	减法	rexp - rexp	rexp	L-R	否
<<	左移位	rexp << rexp	rexp	L-R	否
>>	右移位	rexp >> rexp	rexp	L-R	否
>	大于	rexp > rexp	rexp	L-R	否
>=	大于等于	rexp >= rexp	rexp	L-R	否
<	小于	rexp < rexp	rexp	L-R	否
<=	小于等于	rexp <= rexp	rexp	L-R	否
==	等于	rexp == rexp	rexp	L-R	否
!=	不等于	rexp != rexp	rexp	L-R	否
&	位与	rexp & rexp	rexp	L-R	否
^	位异或	rexp ^ rexp	rexp	L-R	否

操作	描述	语法示例	结果类	结合	是否控制求值
	位或	rexp rexp	rexp	L-R	否
&&	逻辑与	rexp && rexp	rexp	L-R	是
	逻辑或	rexp rexp	rexp	L-R	是
?:	条件操作符	rexp ? rexp : rexp	rexp	N/A	是
=	赋值	lexp = rexp	rexp	R-L	否
+=	以...加	lexp += rexp	rexp	R-L	否
-=	以...减	lexp -= rexp	rexp	R-L	否
*=	以...乘	lexp *= rexp	rexp	R-L	否
/=	以...除	lexp /= rexp	rexp	R-L	否
%=	以...取模	lexp %= rexp	rexp	R-L	否

<<=	以...左移	lexp <<= rexp	rexp	R-L	否
>>=	以...右移	lexp >>= rexp	rexp	R-L	否
&=	以...与	lexp &= rexp	rexp	R-L	否
^=	以...异或	lexp ^= rexp	rexp	R-L	否
=	以...或	lexp = rexp	rexp	R-L	否
,	逗号	rexp , rexp	rexp	L-R	是

2.有问题的表达式

表达式的求值部分由操作符的优先级决定。

(1) 表达式1

```
a*b+c*d+e*f
```

该表达式在计算的时候，由于 `*` 比 `+` 优先级高，只能保证 `*` 的计算比 `+` 早，但是优先级并不能决定第三个 `*` 比第一个 `+` 早执行。

所以表达式的计算机顺序可能是：

```
a*b
```

```
c*d
```

```
a*b+c*d
```

```
e*f
```

```
a*b+c*d+e*f
```

或者：

```
a*b
```

```
c*d
```

```
e*f
```

```
a*b+c*d
```

```
a*b+c*d+e*f
```

(2) 表达式2

```
c + --c;
```

操作符的优先级只能决定自减 `--` 的运算在 `+` 运算的前面，但是我们并没有办法得知，`+` 操作符的左操作数的获取在右操作数之前还是之后求值。

所以结果是不可预测的，有歧义。

(3) 表达式3

```
int main(){
    int i=10;
    i=i-- - --i*(i=-3)* i++ + ++i;
    printf("i=%d\n",i);
    return 0;
}
```

该表达式，在不同编译器中的结果：（非法表达式程序的结果）

值	编译器
-128	Tandy 6000 Xenix 3.2
-95	Think C 5.02(Macintosh)
-86	IBM PowerPC AIX 3.2.5
-85	Sun Sparc cc(K&C编译器)
-63	gcc , HP_UX 9.0 , Power C 2.0.0
4	Sun Sparc acc(K&C编译器)
21	Turbo C/C++ 4.5
22	FreeBSD 2.1 R
30	Dec Alpha OSF1 2.0
36	Dec VAX/VMS
42	Microsoft C 5.1

(4) 表达式4

```

int fun(){
    static int count=1;
    return ++count;
}
int main(){
    int answer;
    answer=fun()-fun()*fun();
    printf("%d\n",answer); //输出多少?
    return 0;
}

```

虽然在大多数编译器上求得结果相同，但是这是存在问题的。

上述表达式 `answer=fun()-fun()*fun()` 中，我们只能通过操作符的优先级得知：先算乘法，再算减法。

但是函数的调用先后顺序无法通过操作符的优先级确定。

(5) 表达式5

```

#include<stdio.h>
int main(){
    int i=1;
    int ret=(++i)+(++i)+(++i);
    printf("%d\n",ret);
    printf("%d\n",i);
    return 0;
}

```

Linux环境的结果：

```

[root@centos7net test]# ./a.out
10
4

```

VS2013环境的结果：

C:\WINDOWS\system32\cmd.exe

12

4

请按任意键继续. . .

看看同样的代码产生了不同的结果，这是为什么？

简单看一下汇编代码，就可以分析清楚。

这段代码中的第一个 + 在执行的时候，第三个 ++ 是否执行，这个是不确定的，因为依靠操作符的优先级和结合性是无法决定第一个 + 和第三个前置 ++ 的先后顺序。

总结：我们写出的表达式如果不能通过操作符的属性确定唯一的计算路径，那这个表达式就是存在问题的。