

09、C 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 杂项运算符

本章将逐一介绍算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和其他运算符。

下表显示了 C 语言支持的所有算术运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
*	把两个操作数相乘	A * B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
--	自减运算符，整数值减少 1	A-- 将得到 9

注释事项

% 操作数必须是整数

/ 两个整数相除，得到整数。如果是小数，舍弃小数部分

关于溢出问题，会与类型范围取余

+ - 优先级低于 * / %

在优先级相同的情况下考虑结合律

❤ 结合律

赋值运算符 (=) 以及扩展的复合赋值运算符如 += 、 -= 、 *= 等) 具有从右到左的结合律。

关系运算符 (如 > 、 < 、 >= 、 <= 、 == 、 !=) 具有从左到右的结合律。

逻辑与运算符 (&&) : 具有从左到右的结合律。

逻辑或运算符 (||) : 也是从左到右的结合律。

按位与运算符 (&) 、 **按位或运算符** (|) 、 **按位异或运算符** (^) : 这些位运算符都遵循从左到右的结合律。

条件运算符 (? :) : 具有从右到左的结合律。

逗号运算符 (,) : 具有从左到右的结合律

前置自增和前置自减运算符与它们所在表达式中的其他运算符进行运算时，具有从右到左的结合律

实例

请看下面的实例，了解 C 语言中 + - * / % 算术运算符：

```
#include <stdio.h>
/**
 * 算术运算符:
 *      + - * / %
 *      - 二元运算符，因为有2个操作数，所以被称为二元运算符
 *      - 操作数1 运算符 操作数2
 *
 *      char   1  窄字符
 *      short  2
```

```

*      int    4
*      long   4
*      long long 8 宽字符
*      float
*      double
*
*      - 窄类型与宽类型进行运算得到的结果一定是宽类型
*      - 需要注意的2个点：
*          - / 如果除数与被除数都是整数，得到的结果一定是整数，如果如果是
小数则截断小数部分
*          - % 取余运算符，参与取余运算的操作数必须是整数类型
*      关于算术运算符的优先级：
*          + - 优先级低于 * / %
*
*      关于阅读程序执行顺序的时候：
*          1. 运算符的优先级
*          2. 在优先级相同的情况，考虑结合律
*              - 从左往右 左结合
*              - 从右往左 右结合 = 从右往左
*/
int main()
{
    int a = 7;
    int b = 3;
    double c = 3.0;
    printf("%d\n", a % b); // 1
    // printf("%d\n", a % c); // 错误，参与取余运算的操作数必须是整数类
型
}
int main03()
{
    int a = 10;
    int b = 20;
    double c = 40;
    printf("%d\n", a * b * c); // 为什么输出的是0，因为从浮点数的存储
中，低位开始取32位
    printf("%lf\n", a * b * c);

```

```

    printf("%d\n", a / b); // 两个整数相除，得到的结果一定是整数，如果出现小数，小数部分会被舍弃
    printf("%lf\n", a / c);
}
int main02()
{
    // 整型与浮点数运算
    int a = 100;
    double b = 3.14;
    printf("%f\n", a + b);
    printf("%lf - size:%d\n", a - b, sizeof(a - b));
}
int main01()
{
    int a = 1;
    int b = 2;
    char c1 = 1;
    char c2 = 2;
    int c = a + b;      // 加法
    char c3 = c1 + c2; // 加法，赋值过程是包含一个类型转换的

    printf("%d\n", c);
    printf("%d size:%d\n", c3, sizeof(c3));
    printf("%d size:%d\n", c1 + c2, sizeof(c1 + c2));
    printf("%d size:%d\n", a + c2, sizeof(a + c2));

    // 使用 short 类型与 int 类型进行运算，得到的结果是 int 类型
    short s1 = 1;
    int i1 = 1000;
    long long l1 = 100000000;
    printf("%d size:%d\n", s1 + i1, sizeof(s1 + i1));
    printf("%u size:%u\n", l1 + i1, sizeof(l1 + i1));
}

```

请看下面的实例，了解 C 语言中 ++、-- 运算符示例：

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

```
/**
```

```
 * 变量++
```

```
 *      - 先返回再加加
```

```
 *  ++变量
```

```
 *      - 先加加再返回
```

```
 * 变量--
```

```
 *      - 先返回再减减
```

```
 *  --变量
```

```
 *      - 先减减再返回
```

```
 */
```

```
int main()
```

```
{
```

```
    int a = 1;
```

```
    int b = 2;
```

int c = a++; // 先返回a变量的值，然后a变量在自加1， ;表示语句，语句能够保证下一条语句得到是当前语句执行后的结果

// 这个概念被称为 序列点，序列点能够保证下一条语句得到是当前语句执行后的结果

```
    int d = ++a;
```

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

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

```
    int x = 1;
```

```
    int y = 2;
```

```
    int z = x++ + ++y;
```

```
    printf("x=%d; y=%d; z=%d\n", x, y, z);
```

```
    int q = 1;
```

```
    /***** 思考这个问题:  start *****/
```

```
    // 第一情况: w = 1 + 2
```

```
    // 第二情况: w = 1 + 3 (在q++执行后的结果2的基础上++)
```

```
    // int w = q++ + ++q;
```

```

/***** 思考这个问题:  end *****/
/***** C语言将这种行为称为, 未定义行为 *****/
int w = ++q + 0 + q++; // C 语言并不保证表达式中操作数执行顺序, 所以这种写法是错误的, 规避这个问题
printf("q=%d;w=%d\n", q, w);
}

```

下表显示了 C 语言支持的所有关系运算符。假设变量 A 的值为 10, 变量 B 的值为 20, 则:

运算符	描述	实例
A > B	判断A是否大于B, 成立返回1否则返回0	0
A >= B	判断A是否大于等于B, 成立返回1否则返回0	0
A < B	判断A是否小于B, 成立返回1否则返回0	1
A <= B	判断A是否小于等于B, 成立返回1否则返回0	1
A == B	判断A是否等于B, 成立返回1否则返回0	0
A != B	判断A是否不等于B, 成立返回1否则返回0	1

实例

请看下面的实例, 了解 C 语言中所有可用的关系运算符:

```

#include <stdio.h>
#include <stdbool.h>
/**
 * 比较运算符:
 *      - >  大于
 *      - <  小于
 *      - >= 大于等于
 *      - <= 小于等于
 *      - == 等于
 *      - != 不等于
 *
 * 需要注意的问题: 条件运算符避免连续书写, 语义不对 (与数学上的语义是不同的)

```

```

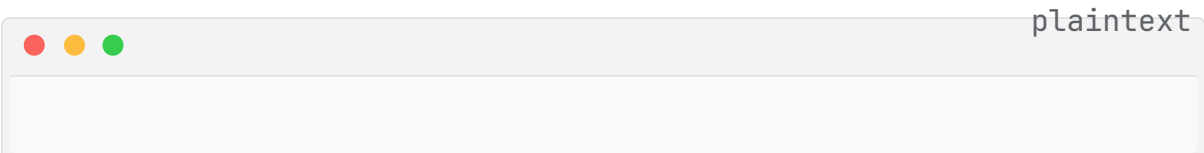
*
*      == 表示判断是否相等
*      = 表示赋值
*
*
*      逻辑值:
*          - 真 true 1
*          - 假 false 0
*/
int main()
{
    int a = 40;
    int b = 20;
    int c = 30;

    printf("布尔类型的 true: %d\n", true);
    printf("布尔类型的 false: %d\n", false);
    printf("条件运算符表达式: %d\n", a > b);
    printf("条件运算符表达式: %d\n", a < b);
    printf("条件运算符表达式: %d\n", a >= b);
    printf("条件运算符表达式: %d\n", a <= b);
    printf("条件运算符表达式: %d\n", a == b);
    printf("条件运算符表达式: %d\n", a != b);

    // 问题:
    // 想要使用C语言表达 b 大于 a 并且 b 小于 c 语义
    a < b < c; // 这个表达式的语义: a < b 的结果在与C比较大小
    // 条件表达式允许连续书写条件表达式, 但是语义不对
    printf("a < b < c 的表达式: %d\n", a < b < c);
    return 0;
}

```

当上面的代码被编译和执行时, 它会产生下列结果:



下表显示了 C 语言支持的所有关系逻辑运算符。假设变量 A 的值为 1，变量 B 的值为 0，则：

运算符	描述	实例
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(A && B) 为假
	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A B) 为真
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(A && B) 为真

注意事项

&& 逻辑与

expression1 && expression2

仅在expression1和expression2都成立（非0即真，0即假），否则返回0

短路结构：

是在expression1不成立情况下，expression2不需要执行

|| 逻辑或

expression1 || expression2

仅在expression1和expression2都不成立（0），否则返回1

短路结构：

是在expression1成立的情况下，expression2不需要执行

! 逻辑非

!expression

表示取反

真转换为假，假转换为真

0 变 1, 1 变 0

非0取反即为0，0取反即为1

数字0和空字符'\0'

!!变量 ---> 可以用来表示当前值对应的逻辑值(0或1)

非0和非空字符转换为1

0和空字符转为0

优先级: ! > && > ||

实例

请看下面的实例，了解 C 语言中所有可用的逻辑运算符：

```
1  #include <stdio.h>
2  /**
3   *   逻辑运算符:
4   *       - && 逻辑与
5   *           expression1 && expression2
6   *               - 仅在expression1和expression2都成立 (非0
   即真, 0即假), 否则返回0
7   *               - 短路结构:
8   *                   - 是在expression1不成立情况下,
   expression2不需要执行
9   *       - || 逻辑或
10  *           expression1 || expression2
11  *               - 仅在expression1和expression2都不成立
   (0), 否则返回1
12  *               - 短路结构:
13  *                   - 是在expression1成立的情况下,
   expression2不需要执行
14  *       - !   逻辑非
15  *           !expression
16  *               - 表示取反
17  *               - 真转换为假, 假转换为真
18  *               - 0 变 1, 1 变 0
19  *
20  *               - 非0取反即为0, 0取反即为1
   *
```

```

21  *           数字0和空字符'\0'
22  *
23  *           !!变量 ---> 可以用来表示当前值对应的逻辑值(0或1)
24  *           - 非0和非空字符转换为1
25  *           - 0和空字符转为0
26  *
27  *
28  *           优先级: ! > && > ||
29  */
30  int main()
31  {
32      int a = 20;
33      int b = 10;
34
35      printf("逻辑运算符 &&: %d\n", a && b++);
36      printf("b++ 是否执行: %d\n", b); // 是10, 表示b++未执
37  行, 是11表示b++执行了, 验证短路结构

38      int x = 0;
39      int y = 2;
40      printf("逻辑运算符 ||: %d\n", x || y++);
41      printf("y++ 是否执行: %d\n", y); // 是2, 表示y++未执行,
42  是3表示y++执行了, 验证短路结构

43      // 逻辑运算符与条件运算结合经常用来表示复杂逻辑判断
44
45      int r = 10;
46      int p = 20;
47      int q = 30;
48      // 想要表达的义: p 大于 r 并且 p 小于 q
49      printf("%d\n", r < p && p < q); // 逻辑运算符的表达式
50
51      char m = '\0'; // 类型转换
52      int n = 0;
53
54      printf("!m = %d\n", !m);
55      printf("!n = %d\n", !n);
56

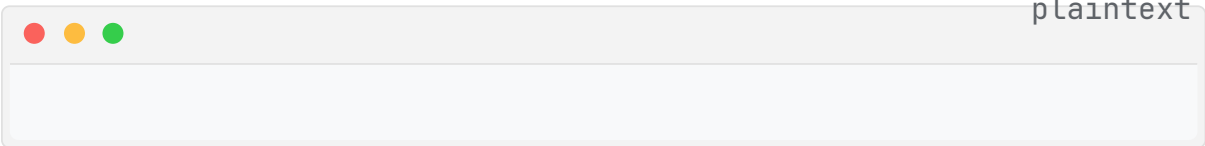
```

```

57     return 0;
58 }

```

当上面的代码被编译和执行时，它会产生下列结果：



位运算符作用于位，并逐位执行操作。 $\&$ 、 $|$ 和 \wedge 的真值表如下所示：

p	q	$p \& q$	$p q$	$p \wedge q$
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 $A = 60$ ，且 $B = 13$ ，现在以二进制格式表示，它们如下所示：

$A = 0011\ 1100$

$B = 0000\ 1101$

$A \& B = 0000\ 1100$

$A | B = 0011\ 1101$

$A \wedge B = 0011\ 0001$

$\sim A = 1100\ 0011$

下表显示了 C 语言支持的位运算符。假设变量 A 的值为 60 ，变量 B 的值为 13 ，则：

运算符	描述	实例
$\&$	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	$(A \& B)$ 将得到 12 ，即为 $0000\ 1100$
$ $	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	$(A B)$ 将得到 61 ，即为 $0011\ 1101$
\wedge	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	$(A \wedge B)$ 将得到 49 ，即为 $0011\ 0001$
\sim	二进制补码运算符是一元运算符，具有“翻转”位效果。	$(\sim A)$ 将得到 -61 ，即为 $1100\ 0011$ ， 2 的补码形式，带符号的二进制数。

运算符	描述	实例
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

实例

请看下面的实例，了解 C 语言中所有可用的位运算符：

```
#include <stdio.h>
int main(){
    unsigned int a = 60; /* 60 = 0011 1100 */
    unsigned int b = 13; /* 13 = 0000 1101 */
    int c = 0;
    c = a & b;          /* 12 = 0000 1100 */
    printf("Line 1 - c 的值是 %d\n", c );
    c = a | b;          /* 61 = 0011 1101 */
    printf("Line 2 - c 的值是 %d\n", c );
    c = a ^ b;          /* 49 = 0011 0001 */
    printf("Line 3 - c 的值是 %d\n", c );
    c = ~a;             /* -61 = 1100 0011 */
    printf("Line 4 - c 的值是 %d\n", c );
    c = a << 2;         /* 240 = 1111 0000 */
    printf("Line 5 - c 的值是 %d\n", c );
    c = a >> 2;         /* 15 = 0000 1111 */
    printf("Line 6 - c 的值是 %d\n", c );
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

下表列出了 C 语言支持的赋值运算符：

运算符	描述	实例
=	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
+=	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
-=	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
=	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C = A 相当于 C = C * A
/=	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
%=	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
<<=	左移且赋值运算符	C <<= 2 等同于 C = C << 2
>>=	右移且赋值运算符	C >>= 2 等同于 C = C >> 2
&=	按位与且赋值运算符	C &= 2 等同于 C = C & 2
^=	按位异或且赋值运算符	C ^= 2 等同于 C = C ^ 2
=	按位或且赋值运算符	C = 2 等同于 C = C 2

实例

请看下面的实例，了解 C 语言中所有可用的赋值运算符：

```
#include <stdio.h>
int main() {
    int a = 21;
    int c;
    c = a;
    printf("Line 1 - = 运算符实例, c 的值 = %d\n", c );
    c += a;
    printf("Line 2 - += 运算符实例, c 的值 = %d\n", c );
    c -= a;
    printf("Line 3 - -= 运算符实例, c 的值 = %d\n", c );
    c *= a;
    printf("Line 4 - *= 运算符实例, c 的值 = %d\n", c );
    c /= a;
    printf("Line 5 - /= 运算符实例, c 的值 = %d\n", c );
    c = 200;
    c %= a;
    printf("Line 6 - %= 运算符实例, c 的值 = %d\n", c );
    c <<= 2;
```

```

printf("Line 7 - <=<= 运算符实例, c 的值 = %d\n", c );
c >>= 2;
printf("Line 8 - >>= 运算符实例, c 的值 = %d\n", c );
c &= 2;
printf("Line 9 - &= 运算符实例, c 的值 = %d\n", c );
c ^= 2;
printf("Line 10 - ^= 运算符实例, c 的值 = %d\n", c );
c |= 2;
printf("Line 11 - |= 运算符实例, c 的值 = %d\n", c );
}

```

当上面的代码被编译和执行时，它会产生下列结果：

plaintext

```

Line 1 - = 运算符实例, c 的值 = 21
Line 2 - += 运算符实例, c 的值 = 42
Line 3 - -= 运算符实例, c 的值 = 21
Line 4 - *= 运算符实例, c 的值 = 441
Line 5 - /= 运算符实例, c 的值 = 21
Line 6 - %= 运算符实例, c 的值 = 11
Line 7 - <=<= 运算符实例, c 的值 = 44
Line 8 - >>= 运算符实例, c 的值 = 11
Line 9 - &= 运算符实例, c 的值 = 2
Line 10 - ^= 运算符实例, c 的值 = 0
Line 11 - |= 运算符实例, c 的值 = 2

```

下表列出了 C 语言支持的其他一些重要的运算符，包括 **sizeof** 和 **? :**。

运算符	描述	实例
sizeof()	返回变量的大小。	sizeof(a) 将返回 4，其中 a 是整数。
&	返回变量的地址。	&a；将给出变量的实际地址。
*a	指向一个变量。	*a；将指向一个变量。
? :	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

实例

请看下面的实例，了解 C 语言中所有可用的杂项运算符：

```

#include <stdio.h>
int main() {
    int a = 4;
    short b;
    double c;
    int* ptr;
    /* sizeof 运算符实例 */
    printf("Line 1 - 变量 a 的大小 = %d\n", sizeof(a) );
    printf("Line 2 - 变量 b 的大小 = %d\n", sizeof(b) );
    printf("Line 3 - 变量 c 的大小 = %d\n", sizeof(c) );
    /* & 和 * 运算符实例 */
    ptr = &a;    /* 'ptr' 现在包含 'a' 的地址 */
    printf("a 的值是 %d\n", a);
    printf("*ptr 是 %d\n", *ptr);
    /* 三元运算符实例 */
    a = 10;
    b = (a == 1) ? 20: 30;
    printf("b 的值是 %d\n", b );
    b = (a == 10) ? 20: 30;
    printf("b 的值是 %d\n", b );
}

```

当上面的代码被编译和执行时，它会产生下列结果：

```

a 的值是 4
*ptr 是 4
b 的值是 30
b 的值是 20

```

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 $x = 7 + 3 * 2$ ，在这里， x 被赋值为 13，而不是 20，因为运算符 $*$ 具有比 $+$ 更高的优先级，所以首先计算乘法 $3 * 2$ ，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	() [] -> . ++ --	从左到右
一元	+ - ! ~ ++ -- (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR		从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR		从左到右
条件	?:	从右到左
赋值	= += -= *= /= %>>= <<= &= ^= =	从右到左
逗号	,	从左到右

实例

请看下面的实例，了解C语言中运算符的优先级：

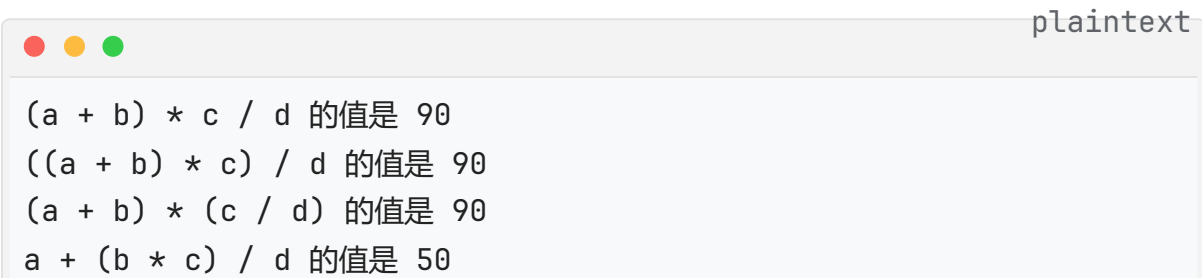
C

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



```
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
e = (a + b) * c / d;      // ( 30 * 15 ) / 5
printf("(a + b) * c / d 的值是 %d\n", e );
e = ((a + b) * c) / d;    // (30 * 15 ) / 5
printf("((a + b) * c) / d 的值是 %d\n" , e );
e = (a + b) * (c / d);    // (30) * (15/5)
printf("(a + b) * (c / d) 的值是 %d\n", e );
e = a + (b * c) / d;      // 20 + (150/5)
printf("a + (b * c) / d 的值是 %d\n" , e );
return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：



```
(a + b) * c / d 的值是 90
((a + b) * c) / d 的值是 90
(a + b) * (c / d) 的值是 90
a + (b * c) / d 的值是 50
```