C 语言

第五讲、运算符、表达式和语句

张晓平

武汉大学数学与统计学院

2017 年 3 月 22 日

1. 示例程序

2. 基本运算符

3. 其它运算符

4. C 与 C++ 中的逗号

5. 关系运算符与逻辑运算符

6. 表达式和语句

7. 优先级与结合性

1. 示例程序

示例程序 I

```
1  // shoe1.c:
2  #include <stdio.h>
3  #define ADJUST 7.64
4  #define SCALE 0.325
5  int main(void)
6  {
7    double shoe, foot;
8    shoe = 9.0;
9    foot = SCALE * shoe + ADJUST;
10   printf("Shoe size (men\'s)    foot length\n");
11   printf("%10.1f %15.2f inches\n", shoe, foot);
```

示例程序 II

```
12    return 0;
13 }
```

示例程序

```
$ gcc shoe1.c
$ ./a.out
Shoe size (men's)    foot length
       9.0             10.56 inches
```

示例程序 I

```
1  // shoe2.c:
2  #include <stdio.h>
3  #define ADJUST 7.64
4  #define SCALE 0.325
5  int main(void)
6  {
7    double shoe, foot;
8    printf("Shoe size (men\'s)  foot length\n");
9    shoe = 8.0;
10   while (shoe < 18.5)  {
11     foot = SCALE * shoe + ADJUST;
```

示例程序 II

```
12      printf("%10.1f %15.2f inches\n", shoe, foot)
        ;
13      shoe = shoe + 1.0;
14    }
15    printf("If shoes fit, wear it.\n");
16    return 0;
17  }
```

示例程序

```
$ gcc shoe2.c
$ ./a.out
Shoe size (men's)   foot length
      12.0              11.54 inches
      13.0              11.87 inches
      14.0              12.19 inches
      15.0              12.52 inches
      16.0              12.84 inches
      17.0              13.16 inches
      18.0              13.49 inches
If shoes fit, wear it.
```

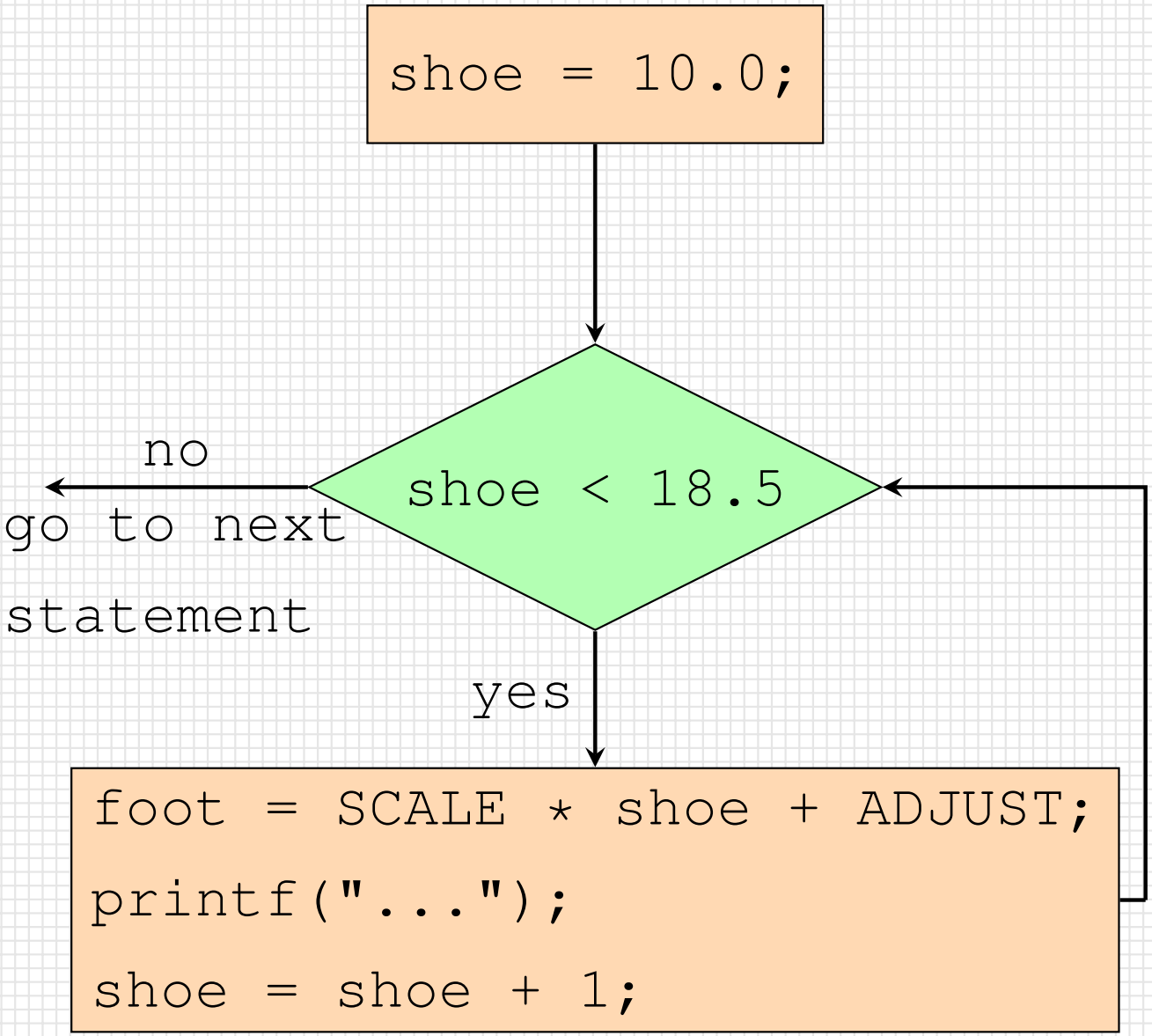
while 循环

```
while (condition)
   statement
```

```
while (condition)
{
   statements
}
```

```
while (condition){
   statements
}
```

while 循环


shoe = 10.0;
shoe < 18.5
no
go to next
statement
yes
foot = SCALE * shoe + ADJUST;
printf("...");
shoe = shoe + 1;

2. 基本运算符

2.1 赋值运算符

赋值运算符

```
int n;
n = 2016;
```

赋值运算符

```
int n;
n = 2016;
```

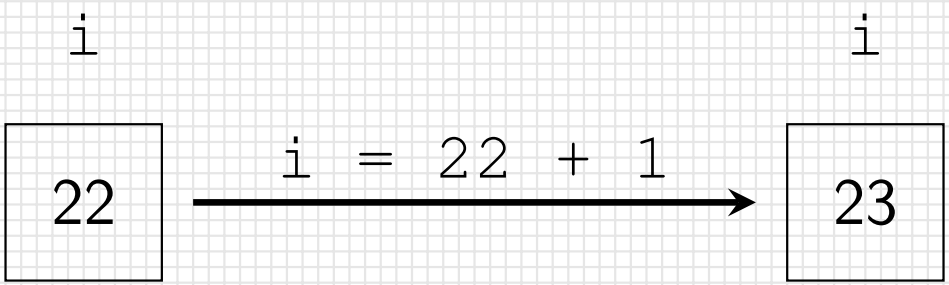
- 在 C 中，= 是一个赋值运算符 (assignment operator)，不表示“相等”。
- 请读成“将值 2016 赋给变量 n”，而不是“n 等于 2016”。
- 赋值运算符的动作是从右到左。

赋值运算符

```
i = i + 1;
```

赋值运算符

```
i = i + 1;
```



赋值运算符

以下语句没有意义：

```
2016 = n;
```

因不能将一个值赋给常量。

赋值运算符

1. 左值：标识的应该是个存储位置，即内存中的位置。

   左值可以是变量名或表达式，但表达式必须表示的是个内存位置。

2. 右值：能赋给可修改的左值的量。

3. 操作数：运算符操作的对象。

赋值运算符

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

赋值运算符

```
$ gcc AssignOpThree.c
$ ./a.out
a = 10, b = 10, c = 10
```

赋值运算符

赋值的过程是从右到左的。先将 10 赋给 c，再将 c 的值赋给 b，最后将 b 的值赋给 a。

2.2 加减运算符

加减运算符

加法运算符 (addition operator) 将其两侧的操作数进行相加。

```
printf("%d", 4+20);
```

两个操作数可以是变量，也可以是常量，如

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

加减运算符

减法运算符 (subtraction operator) 将它前面的数减去后面的数。

```
b = 20.0 - 200.0;
```

+ 和-运算符被称为双目运算符，因为它们都需要两个操作数。

加减运算符

+ 和 - 也可以用作单目运算符。

```
a = -1;
b = -a;
```

此时-表示负号，用于指示或改变一个值的符号。

C99 引入了单目运算符 +，它不改变操作数的值或符号：

```
a = +1;
```

2.3 乘法运算符

乘法运算符

```
mile = 1.6 * km;
```

注意：C 没有提供计算平方的运算符。

乘法运算符

```
1  // square.c:
2  #include <stdio.h>
3  int main(void)
4  {
5     int num=1;
6     while (num < 10) {
7        printf("%4d^2 = %6d\n", num, num*num);
8        num = num + 1;
9     }
10    return 0;
11 }
```

乘法运算符

```
$ gcc squares.c
$ ./a.out
   1^2 =       1
   2^2 =       4
   3^2 =       9
   4^2 =      16
   5^2 =      25
   6^2 =      36
   7^2 =      49
   8^2 =      64
   9^2 =      81
```

2.4 除法运算符

除法运算符

```
1  // divide.c:
2  #include <stdio.h>
3  int main(void)
4  {
5    printf("3/4 = %d\n", 3/4);
6    printf("6/3 = %d\n", 6/3);
7    printf("7/4 = %d\n", 7/4);
8    printf("7./4. = %.2f\n", 7./4.);
9    printf("7./4  = %.2f\n", 7./4);
10   return 0;
11 }
```

除法运算符

```
$ gcc divide.c
$ ./a.out
3/4 = 0
6/3 = 2
7/4 = 1
7./4. = 1.75
7./4  = 1.75
```

除法运算符

- 整数相除，其结果的小数部分会被丢弃，称之为截尾。
- 浮点数相除会得到一个浮点数结果。
- C 允许用一个整数去除浮点数，其结果也是浮点数。在此情况下，做除法运算之前会将整数化为浮点数。

2.5 运算符优先级

运算符优先级

运算符	结合性
()	从左到右
+-(单目运算符)	从右到左
*/	从左到右
+-(双目运算符)	从左到右
=	从右到左

运算符优先级

```
y = 6 * 12 + 5 * 20;
```

运算符优先级

```
y = 6 * 12 + 5 * 20;
```

- 根据优先级规定，两个乘法运算在加法运算之前进行。
- 至于两个乘法运算谁先进行，C 将此选择权留给实现者。

运算符优先级

```
y = 12 / 3 * 2;
```

运算符优先级

```
y = 12 / 3 * 2;
```

- 结合规则适用于共享同一操作数的运算符。
- / 和 * 优先级相同，它们共享操作数 3，按“从左到右”的结合原则，应该先算 12/3。

运算符优先级

```
1 // rules.c:
2 #include <stdio.h>
3 int main(void)
4 {
5   int a, b;
6   b = a = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
7   printf("b = %d\n", b);
8   return 0;
9 }
```

运算符优先级

```
$ gcc rules.c
$ ./a.out
b = -23
```

运算符优先级

```
1. b = a = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
2. b = a = -7 * 6 + (4 + 3 * (2 + 3));
3. b = a = -7 * 6 + (4 + 3 * 5);
4. b = a = -7 * 6 + (4 + 15);
5. b = a = -7 * 6 + 19;
6. b = a = -42 + 19;
7. b = a = -23;
8. b = -23;
```

3. 其它运算符

3.1　取模运算符%

取模运算符%

取模运算符 (modulus operator) 用于计算整数相除所得的余数，只适用于整数运算。

取模运算符%

```
1 // modulus.c:
2 #include <stdio.h>
3 int main(void)
4 {
5   printf("13 %% 5 = %d\n", 13%5);
6   return 0;
7 }
```

```
$ gcc modulus.c
$ ./a.out
13 % 5 = 3
```

取模运算符%I

```
1 // min_sec.c:
2 #include <stdio.h>
3 #define SEC_PER_MIN 60
4 int main(void)
5 {
6   int sec, min, left;
7   printf("Convert seconds to minutes and seconds
    !\n");
8   printf("Enter the number of seconds(<=0 to
    quit):\n");
9   scanf("%d", &sec);
```

取模运算符% II

```
10    while (sec>0) {
11      min = sec / SEC_PER_MIN;
12      left = sec % SEC_PER_MIN;
13      printf("%d seconds is %d minutes, %d seconds
        .\n", sec, min, left);
14      printf("Enter next value (<=0 to quit):\n");
15      scanf("%d", &sec);
16    }
17    printf("Done!\n");
18    return 0;
19  }
```

取模运算符%

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
154
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
```

负数的取模运算 I

```
1  // mod_negative.c:
2  #include <stdio.h>
3  int main(void)
4  {
5    printf(" 11 /  5 = %2d,  11 %  5 = %2d\n",
6           11 / 5, 11 % 5);
7    printf(" 11 / -5 = %2d,  11 % -5 = %2d\n",
8           11 / (-5), 11 % (-5));
9    printf("-11 / -5 = %2d, -11 % -5 = %2d\n",
10          (-11) / (-5), (-11) % (-5));
11   printf("-11 /  5 = %2d, -11 %  5 = %2d\n",
```

负数的取模运算 II

```
12              (-11) / 5, (-11) % 5);
13     return 0;
14 }
```

负数的取模运算

```
 11 /  5 =  2,  11 %  5 =  1
 11 / -5 = -2,  11 % -5 =  1
-11 / -5 =  2, -11 % -5 = -1
-11 /  5 = -2, -11 %  5 = -1
```

负数的取模运算

- C99 规定，整数除法依“趋零截尾”的原则。
- 对于取模运算，
  - 若第一个操作数为负数，则得到的模也为负数；
  - 若第一个操作数为整数，则得到的模也为正数。

3.2 自增自减运算符

自增运算符 ++

自增运算符 (increment operator) 使其操作数的值增加 1。

- 前缀模式：

```
++i;
```

- 后缀模式：

```
i++;
```

两种模式的相似之处在于都使操作数自增 1，区别在于自增这一动作发生的时间不同。

两种模式的相似之处

```
 1 #include <stdio.h>
 2 int main(void)
 3 {
 4   int i = 0, j = 0;
 5   while (i < 5) {
 6     i++;
 7     ++j;
 8     printf("i = %d, j = %d\n",
 9         i, j);
10   }
11   return 0;
12 }
```

两种模式的相似之处

```
i = 1, j = 1
i = 2, j = 2
i = 3, j = 3
i = 4, j = 4
i = 5, j = 5
```

两种模式的相似之处

```
++i;
j++;
```

可以替换为

```
i = i + 1;
j = j + 1;
```

两种模式的相似之处

```
++i;
j++;
```

可以替换为

```
i = i + 1;
j = j + 1;
```

单独使用自增运算符时，前缀模式与后缀模式效果相同。

为什么会创建自增运算符？

使程序更为简洁，可读性更强

```
shoe = 3.0;
while (shoe < 18.5)
{
  foot = SCALE*size + ADJUST;
  printf("%10.1f %20.2f inches\n", shoe, foot);
  ++shoe;
}
```

为什么会创建自增运算符？

进一步简化：

```
shoe = 3.0;
while (++shoe < 18.5)
{
  foot = SCALE*size + ADJUST;
  printf("%10.1f %20.2f inches\n", shoe, foot);
}
```

前缀与后缀模式的不同

```
1  // post_pre.c
2  #include <stdio.h>
3  int main(void)
4  {
5     int a = 1, b = 1;
6     int aplus, bplus;
7     aplus = a++;
8     bplus = ++b;
9     printf("a = %d, aplus = %d\n", a, aplus);
10    printf("b = %d, bplus = %d\n", b, bplus);
11    return 0;
12 }
```

前缀与后缀模式的不同

```
$ gcc post_pre.c
$ ./a.out
a = 2, aplus = 1
b = 2, bplus = 2
```

前缀与后缀模式的不同

```
shoe = 3.0;
while (shoe++ < 18.5)
{
  foot = SCALE * size + ADJUST;
  printf("%10.1f %20.2f inches\n", shoe, foot);
}
```

前缀与后缀模式的不同

- 在使用自增运算符时，请自问一下是否能互换前缀和后缀模式？
- 一个明智的选择是避免那些两种模式将导致不同效果的代码。例如，不要使用

```
b = ++i;
```

可用以下语句代替：

```
++i;
b = i;
```

- 然后有时不那么谨慎会更有趣。

两种模式的不同

观察代码

```
i = 5;
b = ++i;
```

```
i = 5;
b = i++;
```

请分别指出执行后 b 和 i 的值？

自减运算符 –

```
--count; //自减运算符的前缀模式

count--; //自减运算符的后缀模式
```

自增和自减运算符的优先级

自增和自减运算符有很高的优先级，只有圆括号比它们的优先级高。如

```
x * y++
```

等价于

```
x * (y++)
```

而不是

```
(x * y)++    // invalid
```

自增和自减运算符只能作用于变量。

自增和自减运算符的优先级

不要将自增和自减运算符的优先级和求值顺序弄混淆。

自增和自减运算符的优先级

```
1  // inc.c:
2  #include <stdio.h>
3  int main(void)
4  {
5    int y = 2;
6    int n = 3;
7    int nextnum;
8    nextnum = (y + n++)*6;
9    printf("n = %d, nextnum = %d\n", n, nextnum);
10   return 0;
11 }
```

自增和自减运算符的优先级

```
中间过程：nextnum = (2 + 3) * 6 = 5 * 6 = 30
运行结果：n = 4, nextnum = 30
```

自增和自减运算符的优先级

- n++ 表示在使用 n 之后，n 的值才自增。
- 优先级告诉我们 ++ 只属于 n，也告诉我们什么时候使用 n 的值。
- 而自增运算符的性质决定了什么时候改变 n 的值。

自增和自减运算符的优先级

```
1  // inc1.c:
2  #include <stdio.h>
3  int main(void)
4  {
5     int y = 2;
6     int n = 3;
7     int nextnum;
8     nextnum = (y + ++n)*6;
9     printf("n = %d, nextnum = %d\n", n, nextnum);
10    return 0;
11 }
```

自增和自减运算符的优先级

```
中间过程：nextnum = (2 + 4)*6 = 6*6 = 36
运行结果：n = 4, nextnum = 36
```

自增和自减运算符的优先级

- 当 n++ 是表达式的一部分时，它表示：先使用 n，然后将它的值增加
- 当 ++n 是表达式的一部分时，它表示：先将 n 的值增加，然后再使用它。

Don't Be Too Clever

```
1 // inc2.c:
2 #include <stdio.h>
3 int main(void)
4 {
5   int n = 5;
6   printf("n = %d, n^2 = %d\n", n, n*n++);
7   return 0;
8 }
```

Don't Be Too Clever

```
maybe:     n = 5, n^2 = 25
maybe:     n = 6, n^2 = 25
maybe:     n = 6, n^2 = 30
```

Don't Be Too Clever

C 编译器可以选择先计算函数中哪个参数的值。这个自由提高了编译器的效率，但若在函数参数里使用自增自减运算符就会带来麻烦。

Don't Be Too Clever

```
1  // inc3.c:
2  #include <stdio.h>
3  int main(void)
4  {
5    int n = 5;
6    int m;
7    m = n/2 + 5*(1 + n++);
8    printf("n = %d, m = %d\n", n, m);
9    return 0;
10 }
```

Don't Be Too Clever

编译器可能从左到右依次计算，也可能从右到左依次计算，这些都可能导致不可预知的结果。

Don't Be Too Clever

```
1  // inc4.c:
2  #include <stdio.h>
3  int main(void)
4  {
5    int n = 3;
6    int m;
7    m = n++ + n++;
8    printf("n = %d, m = %d\n", n, m);
9    return 0;
10 }
```

Don't Be Too Clever

- 执行后，n 的值为 5，但 m 的值不确定。
- 有的编译器计算 m 时使用 n 的旧值两次，然后将 n 增加两次，从而使 m 的值为 6，n 的值为 5。
- 有的编译器计算 m 时使用 n 的旧值一次，然后增加 n 的值一次，再使用第二个 n 的值，最后第二次增加 n 的值。此方法使 m 的值为 7，n 的值为 5。

请使用以下原则

- 若一个变量出现在同一个函数的多个参数中，不要对它使用自增或自减运算符。
- 当一个变量多次出现在一个表达式时，不要对它使用自增或自减运算符。

4. C 与 C++ 中的逗号

C 与 C++ 中的逗号

在 C 与 C++ 中，逗号有两层含义：

1. 逗号充当运算符。

   逗号运算符为一元运算符，先计算第一个操作数并舍弃之，然后计算第二个操作数并返回该值。逗号运算符具有最低优先级，并且是一个顺序点。

C 与 C++ 中的逗号

```
/* comma as an operator */
int i = (5, 10);  /* 10 is assigned to i*/
int j = (f1(), f2());
 /* f1() is called (evaluated) first followed by
  f2(). The returned value of f2() is assigned
 to j */
```

C 与 C++ 中的逗号

2. 逗号充当分隔符

   逗号作为分隔符，通常用于函数调用与定义，函数宏，变量声明，enum 声明以及结构体中。

```
/* comma as a separator */
int a = 1, b = 2;
void fun(x, y);
```

C 与 C++ 中的逗号

```
/* Comma acts as a separator here and doesn't
enforce any sequence.
    Therefore, either f1() or f2() can be called
     first */
void fun(f1(), f2());
```

C 与 C++ 中的逗号

```
1 // comma1.c:
2 #include<stdio.h>
3 int main()
4 {
5   int x = 10;
6   int y = 15;
7   printf("%d\n", (x, y));
8   return 0;
9 }
```

C 与 C++ 中的逗号

```
1 // comma2.c:
2 #include<stdio.h>
3 int main()
4 {
5   int x = 10;
6   int y = (x++, ++x);
7   printf("%d\n", y);
8   return 0;
9 }
```

C 与 C++ 中的逗号 I

```
1  // comma3.c:
2  #include<stdio.h>
3  int main()
4  {
5    int x = 10, y;
6    // The following is equavalent to y = x++
7    y = (x++, printf("x = %d\n", x),
8          ++x, printf("x = %d\n", x),
9          x++);
10   // Note that last expression is evaluated
11   // but side effect is not updated to y
```

C 与 C++ 中的逗号 II

```
12    printf("y = %d\n", y);
13    printf("x = %d\n", x);
14    return 0;
15 }
```

5. 关系运算符与逻辑运算符

5.1 关系运算符

关系运算符

关系运算符用于比较两个值。

1. 运算符 `==` 检查两个给定的操作数是否相等。若相等，返回 `true`；否则返回 `false`。如 `5 == 5` 返回 `true`。
2. 运算符`!=` 检查两个给定的操作数是否相等。若不相等，返回 `true`；否则返回 `false`。如 `5 != 5` 返回 `false`。

关系运算符

3. 运算符 `>` 检查第一个操作数是否大于第二个操作数。若成立，返回 `true`；否则返回 `false`。如 `6 > 5` 返回 `true`。
4. 运算符 `<` 检查第一个操作数是否小于第二个操作数。若成立，返回 `true`；否则返回 `false`。如 `6 < 5` 返回 `false`。

关系运算符

5. 运算符 `>=` 检查第一个操作数是否大于或等于第二个操作数。若成立，返回 `true`；否则返回 `false`。如 `5 >= 5` 返回 `true`。
6. 运算符 `<=` 检查第一个操作数是否小于或等于第二个操作数。若成立，返回 `true`；否则返回 `false`。如 `5 <= 5` 返回 `true`。

关系运算符 I

```
1 // C program to demonstrate working of
  relational operators
2 #include <stdio.h>
3 int main()
4 {
5   int a=10, b=4;
6   // relational operators
7   // greater than example
8   if (a > b)
9     printf("a is greater than b\n");
10  else
```

关系运算符 II

```
11      printf("a is less than or equal to b\n");
12    // greater than equal to
13    if (a >= b)
14      printf("a is greater than or equal to b\n");
15    else
16      printf("a is lesser than b\n");
17    // less than example
18    if (a < b)
19      printf("a is less than b\n");
20    else
21      printf("a is greater than or equal to b\n");
22    // lesser than equal to
```

关系运算符 III

```
23   if (a <= b)
24     printf("a is lesser than or equal to b\n");
25   else
26     printf("a is greater than b\n");
27   // equal to
28   if (a == b)
29     printf("a is equal to b\n");
30   else
31     printf("a and b are not equal\n");
32   // not equal to
33   if (a != b)
34     printf("a is not equal to b\n");
```

关系运算符 IV

```
35    else
36      printf("a is equal b\n");
37    return 0;
38  }
```

关系运算符

```
Output:
a is greater than b
a is greater than or equal to b
a is greater than or equal to b
a is greater than b
a and b are not equal
a is not equal to b
```

5.2 逻辑运算符

逻辑运算符用于连接两个及以上条件，或对原条件取否。

1. 逻辑与：当两个条件同时满足时，运算符 `&&` 返回 `true`；否则返回 `false`。如，当 `a` 和 `b` 均为 `true` (即非零) 时，`a && b` 返回 `true`。
2. 逻辑或：当至少有一个条件满足时，运算符 `||` 返回 `true`；否则返回 `false`。如，当 `a` 和 `b` 至少有一个为 `true` (即非零) 时，`a || b` 返回 `true`。当然，当 `a` 和 `b` 均为 `true` 时，`a || b` 返回 `true`。
3. 逻辑非：当条件不满足时，运算符`!` 返回 `true` ；否则返回 `false` 。如，若 `a` 为 `false` 时，`a` 返回 `true`。

逻辑运算符 I

```c
// C program to demonstrate working of logical operators
#include <stdio.h>
int main()
{
  int a = 10, b = 4, c = 10, d = 20;
  // logical operators
  // logical AND example
  if (a>b && c==d)
    printf("a is greater than b AND c is equal to d\n");
```

逻辑运算符 II

```
10    else
11      printf("AND condition not satisfied\n");
12    // logical OR example
13    if (a>b || c==d)
14      printf("a is greater than b OR c is equal to
         d\n");
15    else
16      printf("Neither a is greater than b nor c is
         equal to d\n");
17    // logical NOT example
18    if (!a)
19      printf("a is zero\n");
```

逻辑运算符 III

```
20    else
21      printf("a is not zero");
22    return 0;
23  }
```

逻辑运算符

```
AND condition not satisfied
a is greater than b OR c is equal to d
a is not zero
```

5.3 逻辑运算符中的短路现象

逻辑运算符中的短路现象 I

对于逻辑与，若第一个操作数为 false，则第二个操作数将不会被计算。如以下程序将不会打印 Hello。

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 int main()
4 {
5   int a=10, b=4;
6   bool res = ((a == b) && printf("Hello"));
7   return 0;
8 }
```

逻辑运算符中的短路现象

但下面的程序将打印 Hello。

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 int main()
4 {
5   int a=10, b=4;
6   bool res = ((a != b) && printf("Hello"));
7   return 0;
8 }
```

逻辑运算符中的短路现象

对于逻辑或，若第一个操作数为 true，则第二个操作数不会被计算。如以下程序不会打印 Hello。

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 int main()
4 {
5   int a=10, b=4;
6   bool res = ((a != b) || printf("Hello"));
7   return 0;
8 }
```

逻辑运算符中的短路现象

但下面的程序将打印 Hello。

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 int main()
4 {
5   int a=10, b=4;
6   bool res = ((a == b) || printf("Hello"));
7   return 0;
8 }
```

6. 表达式和语句

表达式

定义 **表达式**由运算符和操作数组合而成。最简单的表达式是一个单独的操作数，以此为基础可建立复杂的表达式。

```
4
-6
4 + 21
a * (b + c/d) / 20
q = 5 * 2
x = ++q % 3
q > 3
```

表达式

- 操作数可以是常量、变量，或者是二者的组合。
- 一些表达式是多个较小的表达式的组合，这些小的表达式被称为子表达式 (subexpression)。

表达式

每一个表达式都有一个值

表达式	值
-4 + 6	2
c = 3 + 8	11
5 > 3	1
6 + (c = 3 + 8)	17

最后一个表达式是两个子表达式的和，而每一个子表达式都有值，故它在 C 中是完全合法的，但不建议使用。

语句

- 语句 (statement) 是程序的基本成分。
- 程序 (program) 是一系列语句的集合，一条语句是一条完整的计算机指令。
- 在 C 中，语句以分号结尾。

语句

```
i = 4
```

只是一个表达式，而

```
i = 4;
```

是一条语句。

语句

C 把任何后面带分号的表达式都看做一条语句。所以，C 允许

```
4;
3 + 4;
```

但这些语句不做任何事情。

语句

一般地，语句会改变值和调用函数：

```
x = 25;
++x;
y = sqrt(x);
```

语句

尽管一条语句是一条完整的指令，但不是所有完整的指令都是语句。如

```
x = 6 + (y = 5);
```

在此语句中，子表达式 $y = 5$ 是一个完整的指令，但它只是一条语句的一部分。

语句 I

```
1  // addemup.c:
2  #include <stdio.h>
3  int main(void)   // find sum of first 20 integers
4  {
5     int count, sum;       //declaration statement
6     count = 0;            //assignment statement
7     sum = 0;
8     while (count++ < 20)   //while statement
9        sum = sum + count;
10    printf("sum = %d\n", sum); //function
      statement
```

语句 II

```
11    return 0;
12 }
```

副作用 (side effect)

副作用是对数据对象或文件的修改。如，以下语句

```
i = 50;
```

的副作用是将变量 i 的值设置为 50。

副作用 (side effect)

副作用是对数据对象或文件的修改。如，以下语句

```
i = 50;
```

的副作用是将变量 i 的值设置为 50。Why?

副作用 (side effect)

副作用是对数据对象或文件的修改。如，以下语句

```
i = 50;
```

的副作用是将变量 i 的值设置为 50。Why?

- 从 C 的角度来看，主要目的是求表达式的值。
- 给 C 一个表达式 4+6，C 将计算它的值为 10
- 给 C 一个表达式 i=50，C 将计算它的值为 50，而计算这个表达式的副作用就是把 i 的值改变为 50。

顺序点 (sequence point)

一个顺序点是程序执行中的一点：在该点处，所有的副作用都在进入下一步前被计算。

顺序点 (sequence point)

- 语句中的分号标志了一个顺序点。

  它意味着一条语句中赋值运算符、自增和自减运算符所做的全部改变必须在程序进入下一条语句前发生。

- 任何一个完整表达式的结束也是一个顺序点。

  所谓的完整表达式 (full expression)，它不是一个更大的表达式的子表达式。

  如一个表达式语句中的表达式和在一个 while 循环里作为判断条件的表达式。

顺序点可帮助理解后缀自增动作何时发生

```
while(i++ < 10)
  printf("%d\n", i);
```

- 因 i++ < 10 是 while 循环的判断条件，故它是一个完整表达式，其结束就是一个顺序点。
- C 保证副作用在进入 printf 前发生，同时使用后缀模式保证了 i 在与 10 比较后才增加。

顺序点可帮助理解后缀自增动作何时发生

```
y = (4 + x++) + (6 + x++);
```

- 表达式 4 + x++ 不是一个完整表达式，故 C 不能保证在计算它后立即自增 x。
- 完整表达式是整个赋值语句，并且分号标记了顺序点，故 C 能保证在进入后续语句前 x 被增加两次。
- C 没有指明 x 是在每个子表达式倍计算后增加还是在整个表达式被计算后增加，故建议避免。

复合语句（代码块）

定义　**复合语句**(compound statement) 是使用花括号组织起来的两个或更多的语句；它也被称为一个代码块 (block)。

复合语句（代码块）

比较

```
index = 0;
while (index++ < 10)
  sam = 10 * index + 2;
printf("sam = %d\n", sam);
```

```
index = 0;
while (index++ < 10) {
  sam = 10 * index + 2;
  printf("sam = %d\n", sam);
}
```

7. 优先级与结合性

优先级与结合性

当表达式中有多个不同优先级的运算符，运算符的优先级决定哪个运算符被优先执行。如 `10 + 20 * 30` 等价于 `10 + (20 * 30)`。

结合性用于一个表达式中两个运算符的优先级相同的情形。结合性可能是从左到右或者从右到左。如 `*` 与 `/`有相同的结合性，它们的结合性从左到右，故表达式 `100 / 10 * 10` 等价于 `(100 / 10) * 10`。

优先级与结合性是运算符的两个特征，用于确定子表达式在没有括号时的运算次序。

优先级与结合性

1、结合性仅用于两个或两个以上具有相同优先级的情形。需要指出的是结合性没有定义单运算符中操作数的运算次序。

优先级与结合性 I

```
// Associativity is not used in the below
program. Output
// is compiler dependent.
int x = 0;
int f1()
{
  x = 5;
  return x;
}
int f2()
{
```

优先级与结合性 II

```
  x = 10;
  return x;
}
int main()
{
  int p = f1() + f2();
  printf("%d ", x);
  return 0;
}
```

优先级与结合性

在该程序中，运算符 `+` 的结合性为从左到右，但这并不意味着 `f1()` 总在 `f2()` 之前被调用。

优先级与结合性

2、具有相同优先级的所有运算符有相同的结合性。这是必然的，否则编译器将不知道如何在具有相同优先级和不同结合性的表达式中确定计算次序。如 + 和 − 具有相同的结合性。

优先级与结合性

3、后缀 ++ 与前缀 ++ 的优先级与结合性是不同的。后缀 ++ 的优先级高于前缀 ++，它们的结合性也是不一样的。后缀 ++ 的结合性是从左到右，前缀 ++ 的结合性是从右到左。

优先级与结合性

4、逗号具有最低优先级，使用时需谨慎。

```
#include<stdio.h>
int main()
{
    int a;
    a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
    printf("%d", a);
    return 0;
}
```

优先级与结合性

5、对诸如 `c > b > a` 的表达式，C 将解释为 `(c > b) > a`，而在 Python 中，该表达式将解释为 `a < b and b < c`。

```
#include <stdio.h>
int main()
{
  int a = 10, b = 20, c = 30;
  if (c > b > a)
    printf("TRUE");
  else
    printf("FALSE");
  return 0;
}
```

优先级与结合性

表: 运算符的优先级与结合性

运算符	描述	结合性
()	圆括号 (函数调用)	从左到右
[]	方括号 (数组下标)	
.	通过结构体、共同体等获取成员	
->	通过指针获取成员	
++ --	后缀自增/后缀自减	

优先级与结合性

表: 运算符的优先级与结合性

运算符	描述	结合性
`++ --`	前缀自增/前缀自减	从右到左
`+ -`	正/负	
`!`	逻辑非/位补	
`(type)`	强制类型转换	
`*`	取值运算符	
`&`	取址运算符	
`sizeof`		

优先级与结合性

表: 运算符的优先级与结合性

运算符	描述	结合性
* / %	乘/除/求余	从左到右
+ −	加/减	从左到右
« »	位左移/位右移	从左到右
< <=	小于/小于等于	从左到右
> >=	大于/大于等于	从左到右
== !=	等于/不等于	从左到右

优先级与结合性

表: 运算符的优先级与结合性

运算符	描述	结合性
&	位与	从左到右
^	位异或	从左到右
|	位或	从左到右
&&	逻辑与	从左到右
||	逻辑或	从左到右
?:	三元条件	从右到左

优先级与结合性

表: 运算符的优先级与结合性

运算符	描述	结合性
=	赋值	从右到左
+= -=	加/减赋值	
*= /=	乘/除赋值	
%= &=	求余/位与赋值	
^= !=	位异或/位或赋值	
«= »=	位左移/位右移赋值	
,	逗号 (分离表达式)	从左到右