



中山大學
SUN YAT-SEN UNIVERSITY



中山大學
SUN YAT-SEN UNIVERSITY

表达式和运算符

中山大学计算机学院



讲课人：张晓溪



目录

1. 表达式

2. 运算符

1. 算术运算符 (+ - * / % ++ --)

2. 位运算符 (& | ^ ~ << >>)

3. 关系运算符 (< <= > >= == !=)

4. 逻辑运算符 (&& || !)

5. 赋值运算符 (= += -= *= /= %= &= |= ^= ~= <<= >>=)

6. 其他运算符 (sizeof (type) , ?:)

7. 运算符优先级

语句(Statements)

Statements	Explanation	Syntax	Example
复合语句 (Compound Statements)	复合语句又称为块(Block), 是由花括号{}所包围的语句与声明的序列。	<pre>{ expression; or declaration; or statement }</pre>	<pre>int main() { int n = 1; n = n * n; printf("n*n = %d\n", n); return 0; }</pre>
表达式语句 (Expression Statements)	由分号;结尾的表达式。C语言中大多数语句都是表达式语句。	<pre>expression;</pre>	<pre>n = 1; or n = n * n; or printf("n*n = %d\n", n);</pre>
选择语句 (Selection Statements)	选择语句根据表达式的值, 选择数条语句中的一条来执行。	<pre>if (expression) statement else statement</pre>	<pre>if (n > 0) { printf("n > 0\n"); } else { printf("n < 0\n"); }</pre>
循环语句 (Iteration Statements)	循环语句重复执行一条语句。	<pre>while (expression) statement for (init-exp;exp;exp) statement</pre>	<pre>for (int i = 0; i < 10; i++) { int i2 = i*i; printf("%d^2 = %d\n",i,i2); }</pre>
跳转语句 (Jump Statements)	跳转语句无条件地转移程序控制流。	<pre>break; continue; return expression; goto label;</pre>	<pre>return 0;</pre>

语句(Statements)

Statements	Explanation	Syntax	Example
复合语句 (Compound Statements)	复合语句又称为块(Block), 是由花括号{}所包围的语句与声明的序列。	<pre>{ expression; or declaration; or statement }</pre>	<pre>int main() { int n = 1; n = n * n; printf("n*n = %d\n", n); return 0; }</pre>
表达式语句 (Expression Statements)	由分号;结尾的表达式。C语言中大多数语句都是表达式语句。	<pre>expression;</pre>	<pre>n = 1; or n = n * n; or printf("n*n = %d\n", n);</pre>
选择语句 (Selection Statements)	选择语句根据表达式的值, 选择数条语句中的一条来执行。	<pre>if (expression) statement else statement</pre>	<pre>if (n > 0) { printf("n > 0\n"); } else { printf("n < 0\n"); }</pre>
循环语句 (Iteration Statements)	循环语句重复执行一条语句。	<pre>while (expression) statement for (init-exp;exp;exp) statement</pre>	<pre>for (int i = 0; i < 10; i++) { int i2 = i*i; printf("%d^2 = %d\n",i,i2); }</pre>
跳转语句 (Jump Statements)	跳转语句无条件地转移程序控制流。	<pre>break; continue; return expression; goto label;</pre>	<pre>return 0;</pre>



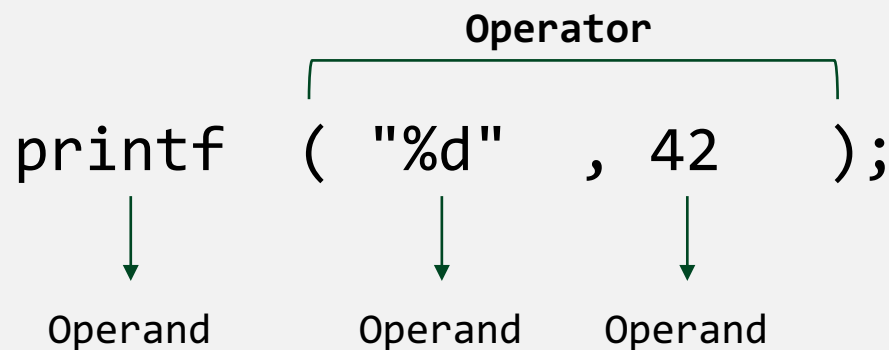
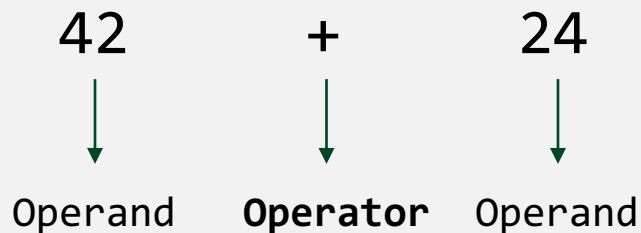
表达式(Expressions)

表达式(Expressions)是由**运算符**(Operators)及其**操作数**(Operands)组成的**序列**，用于指定一组运算。

- 表达式求值通常可以**产生结果**，例如求值`42+24`得到66，其中`+`是运算符，其两边的`42`和`24`分别都是操作数；
- 表达式还可能产生**附效应**(Side Effects)，例如`printf("%d", 42);`会向终端输出数字42，其中`()`是**函数调用**运算符，函数名`printf`以及参数`"%d"`，`42`都可以看作是`()`的操作数。
另一个附效应的例子是`a++`；表达式`a++`求值的结果是`a`，但是它会让变量`a`自增(即`a=a+1;`)

求值结果 (value): 表达式最终计算出来的数值。

附效应 (side effect): 表达式计算过程中对外部状态的修改。





表达式(Expressions)

表达式(Expressions)的操作数(Operand)本身也可以是一个表达式。例如

$$1 + 2 * 3$$

其中，**+**是运算符，1和2*3都是**+**运算符的操作数，操作数1不可继续分解，所以是初等表达式(Primary Expression)；操作数2*3可以继续分解成2 * 3，称为子表达式(Subexpression)。

Primary expression: 3 个 (1、2、3)

Subexpression: 5 个 (1、2、3、2*3、1+2*3)

初等表达式不包含运算符，它可以是：

1. **常量及字面量**，例如**2**或者**"Hello, world"**
2. **已声明的标识符**，例如变量或者函数名。



表达式(Expressions)

表达式(Expressions)的操作数(Operand)本身也可以是一个表达式。例如

$$1 + 2 * 3$$

其中， $+$ 是运算符，1和 $2 * 3$ 都是 $+$ 运算符的操作数，操作数1不可继续分解，所以是初等表达式(Primary Expression)；操作数 $2 * 3$ 可以继续分解成 $2 * 3$ ，称为子表达式(Subexpression)。

那为什么 $1 + 2 * 3$ 不可以分解成 $(1 + 2) * 3$ 呢？



表达式(Expressions)

表达式(Expressions)的操作数(Operand)本身也可以是一个表达式。例如

$$1 + 2 * 3$$

其中，**+**是运算符，1和2*3都是**+**运算符的操作数，操作数1不可继续分解，所以是初等表达式(Primary Expression)；操作数2*****3可以继续分解成2 ***** 3，称为子表达式(Subexpression)。

那为什么1+2*3不可以分解成(1+2)*3呢？
因为*(相乘)的**优先级**比+(相加)更高。



运算符(Operators)

运算符通常可以按照其**功能**划分为以下几类：

- **算术运算符**，用来执行常规算术运算，例如：+ - * / % ++ --
- **位运算符**，用来操作单独的比特运算，例如：& | ^ ~ << >>
- **关系运算符**，用来比较两个操作数的大小，例如：> >= < <= == !=
- **逻辑运算符**，用来进行标准布尔代数运算，例如：&& || !
- **赋值运算符**，例如：= += -= *=
- **其他运算符**，例如：**? :** **sizeof** , **(type)** * (解指针) & (取地址)
[] (数组访问) . (结构体成员) -> (结构体指针的成员)

运算符如果按照其**操作数的个数**则可分为：

- **一元运算符**，例如++，--，!，~，()，sizeof等等
- **二元运算符**，大多数运算符都是二元运算符，连接两个操作数
- **三元运算符**，C语言里只有一个，那就是三元条件运算符 **? :**



算术运算符(Arithmetic Operators)

Operator	Meaning	Example	Result
+	Addition	$x + y$	The sum of x and y
-	Subtraction	$x - y$	The difference of x and y
*	Multiplication	$x * y$	The product of x and y
/	Division	x / y	The quotient of x and y
%	Modulo division	$x \% y$	The remainder of the division x/y
+(Unary)	Positive sign	+x	The value of x
-(Unary)	Negative sign	-x	Negation of x
++	Increment	$x++$ $++x$	x is incremented after evaluation x is incremented before evaluation
--	Decrement	$x--$ $--x$	x is decremented after evaluation x is decremented before evaluation



算术运算符(Arithmetic Operators)

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

```
int main() {  
    int a = 42, b = 5;  
    printf("a+b: %d\n", a+b);  
    printf("a-b: %d\n", a-b);  
    printf("a*b: %d\n", a*b);  
    printf("a/b: %d\n", a/b);  
    printf("a%%b: %d\n", a%b);  
    printf("eval a++: %d\n", a++);  
    printf("after a++, a=%d\n", a);  
    printf("eval ++b: %d\n", ++b);  
    printf("after ++b, b=%d\n", b);  
    return 0;  
}
```

输出:

```
a+b: 47  
a-b: 37  
a*b: 210  
a/b: 8  
a%b: 2  
eval a++: 42  
after a++, a=43  
eval ++b: 6  
after ++b, b=6
```

当操作数都是整型时, a/b 的取值只保留整数部分, 忽略小数部分 (注意不是四舍五入)

%取余运算符的两个操作数都必须是整型。 $a\%b$ 等于 $a - ((a/b)*b)$

$a++$; 和 $++a$; 都有附效应 (Side Effects), 相当于 $a=a+1$;

不过 $a++$ 是先将表达式的取值计算为 a , 然后再将 a 的值加一; $++b$ 则是先将变量 b 自增, 再计算整个表达式的值 (也就是 $b+1$)。

注意自增 (或自减) 运算符的操作数必须是左值 (lvalue)。



左值(lvalue)、右值(rvalue) RECAP

C语言中的表达式(Expressions)分为两种类型:

- **左值(lvalue)**: 是指指向内存中的对象的表达式。赋值符号(=)的左边只能是左值。 (严格来说, 是Modifiable lvalue)
- **右值(rvalue)**: 所有非左值的表达式都是右值(rvalue或者non-lvalue), 它不能出现在赋值符号(=)的左边, 但是可以出现在右边。

```
#include <stdio.h>

int main() {
    int n, a, b, c;
    n = 42;           // n is modifiable lvalue
    10 = 42;          // ERROR! 10 is not lvalue
    const int m = 24; // m is lvalue, but not modifiable
    m = 42;           // ERROR! m is not modifiable lvalue
    a = b = n * 2;     // Both a, b are modifiable lvalue
    a = b + c = n;    // ERROR! b+c is not lvalue
    return 0;
}
```

是**声明**语句,
不是**表达式**语句。

两个变量经过
运算符操作以
后不再是左值

```
a = b = n * 2 ;
-> a = (b = n * 2);
-> a = (n * 2);
```

(赋值符运算顺序是从右到左)

注意: **int a = b = 1;**
来初始化变量a和b是错误的, 因为b还没有被声明。

算术运算符优先级、结合性

优先级	运算符	描述	结合性
1	++ --	自增自减(后缀)	从左到右
2	++ -- + -	自增自减(前缀), 正负号(前缀)	从右到左
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	

a + b * c

→

a + (b * c)

*优先级高于+

a / b * c

→

(a / b) * c

/ *优先级相同, 按照从左到右结合性进行运算

a * 2 + b / 3

→

(a * 2) + (b / 3)

/ *优先级相同且高于+, 优先结合(a*2)和(b/3)

a * - b

→

a * (-b)

-作为负号前缀的优先级高于*

a * b ++

→

a * (b++)

++作为自增后缀优先级高于*



算术运算隐式转换(Implicit Conversion)

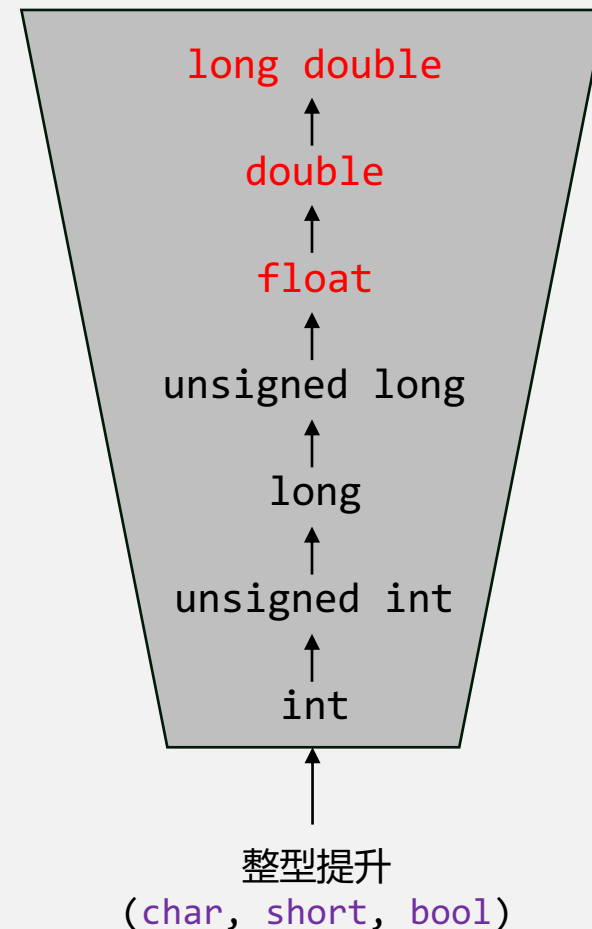
如果算术运算符的两个操作数的数据类型不一致，则会发生**隐式类型转换**。

隐式类型转换通常会将存储空间**较小**的类型转换成存储空间**较大**的类型：

- 如果操作数有大于int类型的，例如long long, float, double, long double, 则另一操作数转换为该高级类型
- 否则，操作数自动转换为int或unsigned int，也就是**整型提升**。

例如：

- `1.0f + 42L;` // long转float, 结果为43.0f
- `'a' + 42;` // 'a'整型提升为int, 结果为97+42=139
- `2 + 3ULL;` // 2转unsigned long long, 结果为5ULL
- `2U - 10;` // 10转unsigned int, 结果为4294967288U
- `2U - 10LL;` // 2U转long long, 结果为-8LL





显式转换(Explicit Cast)

隐式类型转换由编译器自动完成，目的是最大限度地保留信息及精度。

显式类型转换则由程序员手动指定数据类型，利用类型转换运算符`()`强制地将一种类型转换成另一种类型，例如：

- `(double)42L`
- `(int)3.1415926`

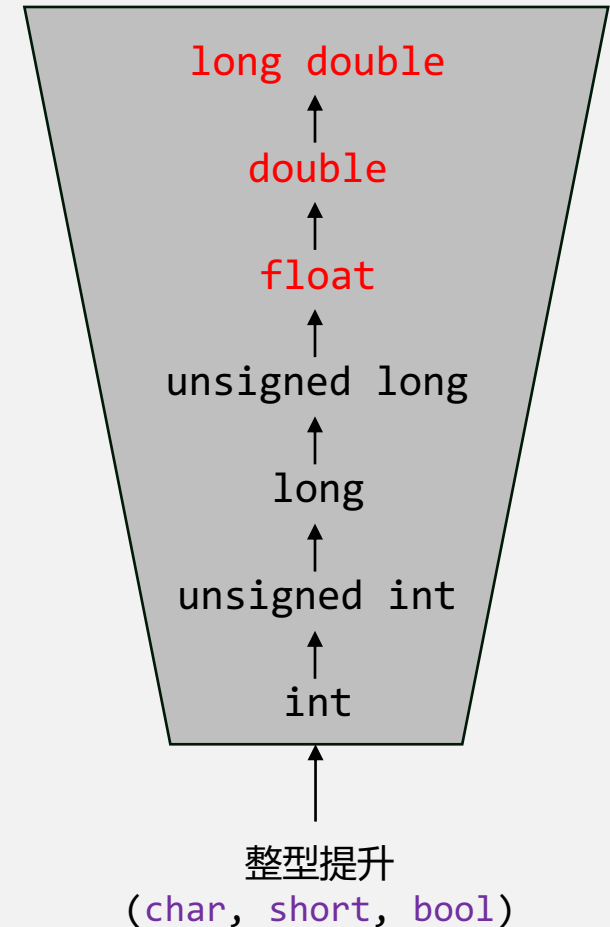
需要注意的是浮点数跟整数之间的转换，当浮点数转换成整数时，其规则为**忽略小数部分**（向零取整），而不是四舍五入。

```
#include <stdio.h>

int main() {
    printf("%f\n", 1.0f + 42L);
    printf("%d\n", (int)3.14);
    printf("%d\n", (int)-3.14);
    return 0;
}
```

输出：

```
43.000000
3
-3
```





位运算符(Bitwise Operators)

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

A	~A
0	1
1	0

```

      0110
&   1100
-----
      0100
    
```

A&B is True only when A AND B are both True.

```

      0110
|   1100
-----
      1110
    
```

A|B is True only when either A OR B is True.

```

      0110
^   1100
-----
      1010
    
```

A^B is True only when either A is true or B is True, **but not BOTH**. (A^B is True only when A and B differs).

```

~   1100
-----
      0011
    
```

~A is True only when A is False.



位运算符(Bitwise Operators)

优先级	运算符	描述	结合性
1	++ --	自增自减(后缀)	从左到右
2	++ -- + - ~	自增自减(前缀), 正负号(前缀), 位运算NOT	从右到左
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	
5	<< >>	位运算左移, 右移	
6	&	位运算AND	
7	^	位运算XOR	
8		位运算OR	

位运算符的结合性是从左到右, 优先级比算术运算符更低。

其中左移、右移 > AND > XOR > OR

位运算NOT(~)因为是一元运算符, 所以其优先级高于其他二元运算符。



位运算符(Bitwise Operators)

```
#include <stdio.h>

int main() {
    unsigned int a = 60; /* 0011 1100 == 60 */
    unsigned int b = 13; /* 0000 1101 == 13 */
    int c = 0;
    c = a & b;           /* 0000 1100 == 12 */
    printf(" a & b: %d\n", c );
    c = a | b;           /* 0011 1101 == 61 */
    printf(" a | b: %d\n", c );
    c = a ^ b;           /* 0011 0001 == 49 */
    printf(" a ^ b: %d\n", c );
    c = ~a;              /* 1100 0011 == -61 */
    printf(" ~a: %d\n", c );
    c = a << 2;          /* 1111 0000 == 240 */
    printf("a << 2: %d\n", c );
    c = a >> 2;          /* 0000 1111 == 15 */
    printf("a >> 2: %d\n", c );
    return 0;
}
```

输出:

```
a & b: 12
a | b: 61
a ^ b: 49
~a: -61
a << 2: 240
a >> 2: 15
```



位运算应用- Mask

对于 `int n = 536912945`，我们想知道它的二进制第13位是1还是0？（第0位是最低位）

0	0	1	0	...	1	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1
31	30	29	28	...	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- `n >> 13`
- `(n >> 13) & 1`



将n右移13位

与1进行AND运算，相当于屏蔽(清零)第0位以外的其他高位
其结果为n的第13位的数字。

&1在这里起到**掩码(Mask)**的作用。
例如，`(n & 0xFF)`的结果是取n低8位的值



位运算应用- Mask

对于int n = 536912945，我们想知道是否它的所有偶数位都为1？

0	0	1	0	...	1	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1
31	30	29	28	...	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- `int mask = 0x55555555;`
- `(n & mask)`
- `(n & mask) == mask`

构建一个选取所有偶数位的mask（5的二进制为0101）

将n与mask进行AND运算，得到n的偶数位

将(n&mask)与mask进行比较，如果相等，那么n偶数位均为1



位运算应用- 取反

对于 `int n = 536912945`，我们想将它的二进制第13位改为0

0	0	1	0	...	1	0	1	0	0	1	0	0	0	0	1	1	0	0	0	1
31	30	29	28	...	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- `1 << 13`
- `int mask = ~(1 << 13);`
- `n = (n & mask);`

将1左移13位，即(0000 ... 0010 0000 0000 0000)
构建一个的mask，其第13位为0，其余为1
即(1111 1111 1111 1111 1101 1111 1111 1111)
将n与mask进行AND运算，则n第13位改为0，其余位不变



位运算应用- 异或(XOR)

如何交换a,b两个数?

借助临时变量c

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d%d", &a, &b);

    c = a;
    a = b;
    b = c;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```



位运算应用- 异或(XOR)

如何交换a,b两个数?

借助临时变量c

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d%d", &a, &b);

    c = a;
    a = b;
    b = c;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

利用算术运算

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);

    a = a + b;    // A=a+b, B=b
    b = a - b;    // A=a+b, B=(a+b)-b=a
    a = a - b;    // A=b, B=a

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```



位运算应用- 异或(XOR)

如何交换a,b两个数?

借助临时变量c

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d%d", &a, &b);

    c = a;
    a = b;
    b = c;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

利用位运算(**异或^**)

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);

    a = a ^ b; // A=a^b, B=b
    b = a ^ b; // A=a^b, B=(a^b)^b=a
    a = a ^ b; // A=b, B=a

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

都是异或^运算, 无需考虑+ -先后顺序



位运算应用- 异或(XOR)

如何交换a,b两个数?

借助临时变量c

```
#include <stdio.h>

int main() {
    int a, b, c;
    scanf("%d%d", &a, &b);

    c = a;
    a = b;
    b = c;

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

利用位运算(异或 \wedge)

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);

    a ^= b;    // A=a^b, B=b
    b ^= a;    // A=a^b, B=(a^b)^b=a
    a ^= b;    // A=b, B=a

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

关系运算符(Relational Operators)

关系运算符(Relational Operators)也叫比较运算符(Comparison Operators), 属于二元运算符, 它们可以测试两个操作数(表达式), 若条件逻辑为真则返回1, 条件逻辑为假则返回0。如果两个操作数的类型不一致, 则会发生**隐式类型转换**。

运算符	运算符名	示例
==	等于	a == b
!=	不等于	a != b
<	小于	a < b
>	大于	a > b
<=	小于或等于	a <= b
>=	大于或等于	a >= b



关系运算符(Relational Operators)

关系运算符(Relational Operators)也叫比较运算符(Comparison Operators), 属于二元运算符, 它们可以测试两个操作数(表达式), 若条件逻辑为真则返回1, 条件逻辑为假则返回0。如果两个操作数的类型不一致, 则会发生**隐式类型转换**。

```
#include <stdio.h>

int main() {
    printf(" 42 > 24: %d\n", 42 > 24);
    printf(" 42 < 24: %d\n", 42 < 24);
    printf("2.0 == 2: %d\n", 2.0 == 2);
    // 2 converted to double 2.0
    printf(" -1 >= 1: %d\n", -1 >= 1);
    printf("-1 >= 1U: %d\n", -1 >= 1U);
    // -1 converted to unsigned int 4294967295
    return 0;
}
```

输出:

```
42 > 24: 1
42 < 24: 0
2.0 == 2: 1
-1 >= 1: 0
-1 >= 1U: 1
```

关系运算符(Relational Operators)

优先级	运算符	描述	结合性
1	++ --	自增自减(后缀)	从左到右
2	++ -- + - ~	自增自减(前缀), 正负号(前缀), 位运算NOT	从右到左
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	
5	<< >>	位运算左移, 右移	
6	< <= > >=	关系运算符(判断大小)	
7	== !=	关系运算符(判断是否相等)	
8	&	位运算AND	
9	^	位运算XOR	
10		位运算OR	

关系运算符的结合性也是从左到右, 优先级比位运算左移右移更低, 比AND, XOR, OR位运算更高。



逻辑运算符(Logical Operators)

逻辑运算符(Logical Operators)应用标准布尔代数运算到其操作数当中。

- 逻辑与`&&`：如果A和B都为真(非0)，则A`&&`B为真，取值为1
- 逻辑或`||`：如果A和B其中一个为真，则A`||`B为真，取值为1
- 逻辑非`!`：如果A为真(非0)，则`!A`为假(0)；如果A为假(0)，则`!A`为真(1)

逻辑运算符比较容易与**位运算符**混淆(`&&` vs `&`, `||` vs `|`, `!` vs `~`)，因为两者都是布尔代数运算的应用。但是位运算是操作数的**每一个比特**进行布尔代数运算，操作数可以是任意整数；而逻辑运算符则把操作数当成只有1和0两种状态，其中0表示**假**，其他任何**非0**的值都为**真**。



逻辑运算符(Logical Operators)

```
#include <stdio.h>
int main() {
    int a = 1, b = 0;
    if (a && b) { // Demonstrating Logical AND (&&)
        printf("Both a and b are true.\n");
    } else {
        printf("Either a or b (or both) are false.\n");
    }
    if (a || b) { // Demonstrating Logical OR (||)
        printf("Either a or b (or both) are true.\n");
    } else {
        printf("Both a and b are false.\n");
    }
    if (!a) { // Demonstrating Logical NOT (!)
        printf("a is false.\n");
    } else {
        printf("a is true.\n");
    }
    return 0;
}
```

输出:

```
Either a or b (or both) are false.
Either a or b (or both) are true.
a is true.
```



逻辑运算符(Logical Operators)

```
#include <stdio.h>
int main() {
    int a = -314, b = 0;
    if (a && b) { // Demonstrating Logical AND (&&)
        printf("Both a and b are true.\n");
    } else {
        printf("Either a or b (or both) are false.\n");
    }
    if (a || b) { // Demonstrating Logical OR (||)
        printf("Either a or b (or both) are true.\n");
    } else {
        printf("Both a and b are false.\n");
    }
    if (!a) { // Demonstrating Logical NOT (!)
        printf("a is false.\n");
    } else {
        printf("a is true.\n");
    }
    return 0;
}
```

将a的值改为其他非0值, 例如
`int a = -314, b = 0;`
逻辑运算符还是把a当作真(True),
程序运行结果不变。

输出:

```
Either a or b (or both) are false.
Either a or b (or both) are true.
a is true.
```



逻辑运算短路求值

逻辑运算符短路求值(Short-circuit Evaluation)。逻辑运算符是**从左到右**对操作数表达式进行求值的，根据逻辑运算的特性，有时候不需要对后面的表达式进行求值或执行：

- 对于逻辑表达式 `exp1 && exp2`，如果 `exp1` 的取值结果为假(0)，那么 `exp2` 则完全不会被执行求值(被短路)，因为 `exp1` 为假的事实已经足够推断出整个表达式也为假(**逻辑与**要求 `exp1` 和 `exp2` 必须都为真)
- 对于逻辑表达式 `exp1 || exp2`，如果 `exp1` 的取值结果为真(非0)，那么 `exp2` 则完全不会被执行求值(被短路)，因为 `exp1` 为真的事实已经足够推断出整个表达式也为真(**逻辑或**只需 `exp1` 和 `exp2` 其中一个为真即可)



逻辑运算符(Logical Operators)

优先级	运算符	描述	结合性
1	++ --	自增自减(后缀)	从左到右
2	++ -- + - ! ~	自增自减(前缀), 正负号(前缀), 逻辑非, 位运算NOT	从右到左
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	
5	<< >>	位运算左移, 右移	
6	< <= > >=	关系运算符(判断大小)	
7	== !=	关系运算符(判断是否相等)	
8	&	位运算AND	
9	^	位运算XOR	
10		位运算OR	
11	&&	逻辑与	
12		逻辑或	

逻辑运算符的结合性也是从左到右，优先级比目前所有运算符都更低。
逻辑非是一元运算符，所以其优先级高于二元运算符。



赋值运算符(Assignment Operators)

赋值运算符是用来给lvalue赋值的运算符。其一般格式是

```
lvalue = expression;  
lvalue op= expression;
```

其中简单赋值运算符`=`的作用是将expression的值赋给lvalue; 复合赋值运算符`**op**=`等价于lvalue = lvalue **op** (expression), 其中**op**可以是任意二元**算术运算符**或二元**位运算符**。例如

- x **+=** 1; 等价于x = x **+** (1);
- x ***=** 1+2; 等价于x = x ***** (1+2);
- x **<<=** 2+3; 等价于x = x **<<** (2+3);



赋值运算符(Assignment Operators)

```
int main() {  
    int x = 10; // Basic assignment using '='  
    printf("Init value of x: %d\n", x);  
    x += 5; // Equivalent to: x = x + 5  
    printf("After x += 5: %d\n", x);  
    x -= 3; // Equivalent to: x = x - 3  
    printf("After x -= 3: %d\n", x);  
    x *= 9; // Equivalent to: x = x * 9  
    printf("After x *= 2: %d\n", x);  
    x %= 5; // Equivalent to: x = x % 5  
    printf("After x %%= 3: %d\n", x);  
    x <<= 2; // Equivalent to: x = x << 2 (bitwise left shift)  
    printf("After x <<= 2: %d\n", x);  
    x >>= 1; // Equivalent to: x = x >> 1 (bitwise right shift)  
    printf("After x >>= 1: %d\n", x);  
    x &= 5; // Equivalent to: x = x & 5 (bitwise AND)  
    printf("After x &= 5: %d\n", x);  
    x ^= 2; // Equivalent to: x = x ^ 2 (bitwise XOR)  
    printf("After x ^= 2: %d\n", x);  
    return 0;  
}
```

$x \text{ op} = y$; 等价于 $x = x \text{ op } y$;

其中 op 可以是任何二元算术运算符或者二元位运算符。

输出:

```
Init value of x: 10  
After x += 5: 15  
After x -= 3: 12  
After x *= 2: 108  
After x %= 3: 3  
After x <<= 2: 12  
After x >>= 1: 6  
After x &= 5: 4  
After x ^= 2: 6
```



赋值运算符(Assignment Operators)

```
#include <stdio.h>

int main() {
    int x, y, z;
    // Demonstrating right-associativity of '='
    x = y = z = 10;
    // Equivalent to: z = 10; y = z; x = y;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
    // Demonstrating right-associativity
    // with compound assignment operators
    x *= y += z -= 5;
    // Equivalent to: z = z - 5; y = y + z; x = x * y;
    printf("After x *= y += z -= 5:\n
           \nx = %d, y = %d, z = %d\n",
           x, y, z);
    return 0;
}
```

赋值运算符的结合性是**从右到左**。
例如 `x *= y += z -= 5;`就等价于

```
z = z - 5;
y = y + z;
x = x * y;
```

输出:

```
x = 10, y = 10, z = 10
After x *= y += z -= 5:
x = 150, y = 15, z = 5
```

赋值运算符(Assignment Operators)

优先级	运算符	描述	结合性
1	++ --	自增自减(后缀)	从左到右
2	++ -- + - ! ~	自增自减(前缀), 正负号(前缀), 逻辑非, 位运算NOT	从右到左
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	
5	<< >>	位运算左移, 右移	
6	< <= > >=	关系运算符(判断大小)	
7	== !=	关系运算符(判断是否相等)	
8	&	位运算AND	
9	^	位运算XOR	
10		位运算OR	
11	&&	逻辑与	
12		逻辑或	
13	= op=	赋值运算符	从右到左

赋值运算符的结合性是**从右到左**, 优先级比目前所有运算符都更低。

其他运算符

Operator	Meaning	Example	Result
<code>? :</code>	三元条件判断	<code>(a>b)?a:b</code>	如果 <code>a>b</code> , 则返回 <code>a</code> ; 否则返回 <code>b</code>
<code>sizeof</code>	查询对象或类型的大小	<code>sizeof(int)</code>	返回 <code>int</code> 类型所占内存空间大小(字节数)
<code>(type)exp</code>	强制类型转换	<code>int a = (int)2.5;</code>	将浮点数 <code>2.5</code> 强制转换成 <code>int</code> 类型
<code>function()</code>	函数调用	<code>printf("Hello");</code>	调用 <code>printf</code> 函数, 参数为 <code>"Hello"</code>
<code>,</code>	逗号运算符	<code>exp1, exp2</code>	先求值 <code>exp1</code> ,再求值 <code>exp2</code>
<code>*</code>	解指针(Dereference)	<code>*p</code>	指针 <code>p</code> 指向的对象(或函数)
<code>&</code>	取地址(Address-of)	<code>&var</code>	指向变量 <code>var</code> 的指针
<code>[]</code>	数组下标	<code>arr[3]</code>	数组 <code>arr</code> 的第4个元素
<code>.</code>	成员访问	<code>s.memb</code>	结构体(或共用体) <code>s</code> 的成员 <code>memb</code>
<code>-></code>	通过指针的成员访问	<code>p->memb</code>	由指针 <code>p</code> 指向的结构体(或共用体)的成员 <code>memb</code>

其他运算符

Operator	Meaning	Example	Result
<code>? :</code>	三元条件判断	<code>(a>b)?a:b</code>	如果 <code>a>b</code> , 则返回 <code>a</code> ; 否则返回 <code>b</code>
<code>sizeof</code>	查询对象或类型的大小	<code>sizeof(int)</code>	返回 <code>int</code> 类型所占内存空间大小(字节数)
<code>(type)exp</code>	强制类型转换	<code>int a = (int)2.5;</code>	将浮点数2.5强制转换成 <code>int</code> 类型
<code>function()</code>	函数调用	<code>printf("Hello");</code>	调用 <code>printf</code> 函数, 参数为"Hello"
<code>,</code>	逗号运算符	<code>exp1, exp2</code>	先求值 <code>exp1</code> ,再求值 <code>exp2</code>
<code>*</code>	解指针(Dereference)	<code>*p</code>	指针 <code>p</code> 指向的对象(或函数)
<code>&</code>	取地址(Address-of)	<code>&var</code>	指向变量 <code>var</code> 的指针
<code>[]</code>	数组下标	<code>arr[3]</code>	数组 <code>arr</code> 的第4个元素
<code>.</code>	成员访问	<code>s.memb</code>	结构体(或共用体) <code>s</code> 的成员 <code>memb</code>
<code>-></code>	通过指针的成员访问	<code>p->memb</code>	由指针 <code>p</code> 指向的结构体(或共用体)的成员 <code>memb</code>



三元条件运算符

三元条件运算符? : 用来组成一个条件表达式(Conditional Expression)。其基本语法为:

Expression1 ? Expression2 : Expression3

该表达式的值取决于Expression1的值(真或假)。如果Expression1为真, 那么该表达式的值即Expression2的值; 如果Expression1为假, 那么该表达式的值即Expression3的值。相当于选择判断语句:

```
if (Expression1) {  
    Expression2;  
} else {  
    Expression3;  
}
```




三元条件运算符

```
#include <stdio.h>

int main() {
    int a = 33, b = 22, c = 44;;
    printf("    Max(a,b): %d\n", a > b ? a : b);
    printf("    Min(a,b): %d\n", a < b ? a : b);
    printf("Max(a,b,c): %d\n",
        a > b ?
        (a > c ? a : c) :
        (b > c ? b : c));
    return 0;
}
```

输出:

```
Max(a,b): 33
Min(a,b): 22
Max(a,b,c): 44
```



三元条件运算符

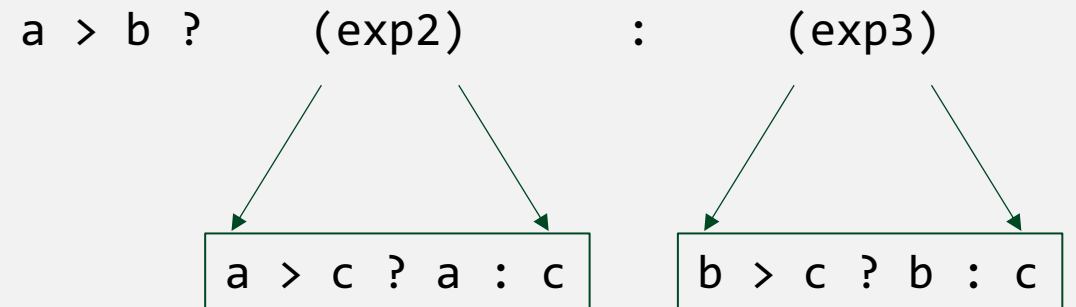
```
#include <stdio.h>

int main() {
    int a = 33, b = 22, c = 44;;
    printf("    Max(a,b): %d\n", a > b ? a : b);
    printf("    Min(a,b): %d\n", a < b ? a : b);
    printf("Max(a,b,c): %d\n",
        a > b ?
        (a > c ? a : c) :
        (b > c ? b : c));
    return 0;
}
```

输出:

```
Max(a,b): 33
Min(a,b): 22
Max(a,b,c): 44
```

嵌套(Nested Ternary Operator):





三元条件运算符

```
#include <stdio.h>

int main() {
    int a = 3, b = 4;
    printf("%d is %s\n", a, a % 2 ? "odd" : "even");
    printf("%d is %s\n", b, b % 2 ? "odd" : "even");
    return 0;
}
```

输出:

```
3 is odd
4 is even
```

条件表达式

$n \% 2 \text{ ? "odd" : "even"}$


当 n 为奇数时, $n \% 2 == 1$, 于是 $(n \% 2)$ 为真, 所以整个三元条件表达式的值为?
后面第一个表达式, 即"odd";
当 n 为偶数时, $n \% 2 == 0$, 于是 $(n \% 2)$ 为假, 所以整个三元条件表达式的值为?
后面第二个表达式, 即"even";



三元条件运算符

```
#include <stdio.h>

int main() {
    int a = 3, b = 4;
    printf("%d is %s\n", a, a % 2 ? "odd" : "even");
    printf("%d is %s\n", b, b % 2 ? "odd" : "even");
    return 0;
}
```



输出:

```
3 is odd
4 is even
```

如果写成对应的if...else...语句，则会显得比较繁琐，但逻辑更加清晰：

```
#include <stdio.h>

int main() {
    int a = 3, b = 4;
    if (a % 2) {
        printf("%d is %s\n", a, "odd");
    } else {
        printf("%d is %s\n", a, "even");
    }
    if (b % 2) {
        printf("%d is %s\n", b, "odd");
    } else {
        printf("%d is %s\n", b, "even");
    }
    return 0;
}
```



if语句 – 多路选择 – 应用

```
#include <stdio.h>

int main() {
    int grade;
    scanf("%d", &grade);
    if (grade >= 90) {
        printf("A\n");
    } else if (grade >= 80) {
        printf("B\n");
    } else if (grade >= 70) {
        printf("C\n");
    } else if (grade >= 60) {
        printf("D\n");
    } else {
        printf("F(ailed)\n");
    }
    return 0;
}
```

运行结果:

```
$ ./grades
98
A
$ ./grades
88
B
$ ./grades
66
D
$ ./grades
59
F(ailed)
```



sizeof运算符

```
#include <stdio.h>

int main() {
    int n = 1;
    printf("    sizeof(char): %lu\n", sizeof(char));
    printf("    sizeof(short): %lu\n", sizeof(short));
    printf("    sizeof(int): %lu\n", sizeof(int));
    printf("    sizeof(long): %lu\n", sizeof(long));
    printf("    sizeof(float): %lu\n", sizeof(float));
    printf("    sizeof(double): %lu\n", sizeof(double));
    printf("    sizeof('A'): %lu\n", sizeof('A'));
    printf("    sizeof(314): %lu\n", sizeof(314));
    printf("    sizeof(3.14): %lu\n", sizeof(3.14));
    printf("    sizeof(n++): %lu\n", sizeof(n++));
    printf("        n: %d\n", n); // n is unchanged
    return 0;
}
```

sizeof运算符是一个编译时(Compile-time)的一元运算符，它通常被用来计算其操作数的大小，即占用内存空间的**字节数**。sizeof的操作数如果是表达式的话，该**表达式不会被执行**。

输出:

```
sizeof(char): 1
sizeof(short): 2
    sizeof(int): 4
    sizeof(long): 8
sizeof(float): 4
sizeof(double): 8
    sizeof('A'): 4
    sizeof(314): 4
    sizeof(3.14): 8
    sizeof(n++): 4
        n: 1
```



逗号运算符,

```
#include <stdio.h>

int main() {
    int a, b, c;
    a = (b = 3, b + 2);
    /* First, b is assigned the value 3;
     * Then, b + 2 is evaluated,
     * and its result (5) is assigned to a. */

    c = (printf("Value of a: %d\n", a), a * 2);
    /* First, the printf statement is executed;
     * Then, a * 2 is evaluated,
     * and its result (10) is assigned to c. */
    printf("Value of b: %d\n", b);
    printf("Value of c: %d\n", c);
    return 0;
}
```

逗号运算符,是所有运算符当中优先级最低的。它表示两个表达式的序列:

exp1, exp2

执行时先求值exp1, 再求值exp2。整个表达式的值是exp2。

因为逗号运算符,的优先级是最低的,所以通常需要使用括号(exp1, exp2)来强制执行这个逗号连接起来的表达式序列。

输出:

```
Value of a: 5
Value of b: 3
Value of c: 10
```



运算符优先级(Precedence)

运算符优先级的基本原则是

一元运算符 > 二元运算符 > 三元运算符

乘除取余 > 加减 > 关系 > 逻辑 > 赋值



运算符优先级(Precedence)

运算符优先级的基本原则是

一元运算符 > 二元运算符 > 三元运算符

乘除取余 > 加减 > 左移右移 > 关系 > 位运算(&^|) > 逻辑 > 赋值

运算符优先级(Precedence)

一元运算符
()函数调用除外

优先级	运算符	描述	结合性
1	++ -- () []	自增自减(后缀), 函数调用, 数组访问	从左到右
2	++ -- + - ! ~	自增自减(前缀), 正负号, 逻辑非, 位运算NOT	从右到左
	(type) sizeof * &	类型转换, 取大小, 取地址, 解指针	
3	* / %	乘法, 除法, 取余数	从左到右
4	+ -	加法, 减法	
5	<< >>	位运算左移、右移	
6	< <= > >=	关系运算符(判断大小)	
7	== !=	关系运算符(判断是否相等)	
8	&	位运算AND	
9	^	位运算XOR	
10		位运算OR	
11	&&	逻辑与	
12		逻辑或	
三元运算符	?:	三元条件运算符	从右到左
	= op=	赋值运算符	
	,	逗号运算符(Comma Operator)	从左到右



小结

重点及难点:

- `a++;`与`++a;`的区别
- `%`取余数运算符的两个操作数必须都是整型类型
- 优先级高的运算符优先求值, 同一优先级按照其结合性(从左到右或从右到左)顺序求值
- 位运算符的优先级, 左移右移优先级高于其他位运算, 中间隔着关系运算符
- (在没有发生溢出的情况下), `a>>N`相当于`a/(2^N)`, `a<<N`相当于`a*(2^N)`
- 异或位运算符(`^`)用来交换两个整数
- 浮点数转换成整数时, 忽略小数部分
- 逻辑运算符的短路求值特性
- 三元条件运算符`?:`的使用
- `sizeof`运算符的使用
- 赋值运算符(`op=`)的结合性是从右到左



中山大學
SUN YAT-SEN UNIVERSITY



中山大學
SUN YAT-SEN UNIVERSITY

谢谢

中山大学计算机学院