

step by step

进阶

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、指针、结构）

归纳与推广（程序设计的本质）

程序中基本操作的描述

□ 程序中的基本操作（通过操作符operator描述）

■ 算术操作

+ - ++ -- + - * / %

■ 关系操作

> >= < <= == !=

■ 逻辑操作

! && ||

■ 条件操作 ? :

■ 逗号操作 ,

■ 位操作

~ & | ^ << >>

■ 赋值操作

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

算术操作的注意事项及应用

C语言中的乘法操作 (*)

□ * 不可以省略

`i = 2a; // ×`

□ C语言中没有幂运算

- 可以用 $x * x$ 计算平方
- 可以用 $x * x * x$ 计算立方
- 可以调用库函数 `pow(x, y)` 计算 x^y
 - 该函数的说明信息在 `math.h` 中
 - x 、 y 为实数
- 底数与指数均为整数的幂运算，
可以用自行实现的幂函数来求解

```
int PowInt(int x, int n)
{
    int z = 1;
    while(n >= 1)
    {
        z *= x;    // z = z * x;
        --n;
    }
    return z;
}
```

C语言中的除法操作 (/)

❑ 0 不能做除数

❑ / 可用于两个整数或实数相除, 当 (且仅当) 用于两个整数相除时, 结果只取商的整数部分, 小数部分被截去, 并且一般不进行四舍五入。

■ 例如: 3/2的结果为1; 1/2的结果为0; -10/3的结果为-3。

■ 较小的整数除以较大的整数, 结果为0。

■ 编程时, 程序员往往需要采取措施避免因整除而带来的意想不到的错误。

```
double Pi()  
{  
    int sign = 1;  
    double item = 1.0, sum = 1.0;  
    for(int n = 1; fabs(item) >= 0.000001; ++n)  
    {  
        sign = -sign; //运用了取负操作  
        item = sign * 1.0 / (2 * n + 1);  
        sum += item;  
    }  
    return 4 * sum;  
}
```

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

□ C语言整数相除结果的小数部分被截去的特点，在某些场合可以发挥数据映射作用。

```
int score = 0;
scanf("%d", &score);
switch(score / 10)
{
    case 10:          //若score为100，则执行printf("A ...
case 9: printf("A \n"); break;          //若score为90-99， ...
case 8: printf("B \n"); break;          //若score为80-89， ...
case 7: printf("C \n"); break;          //若score为70-79， ...
case 6: printf("D \n"); break;          //若score为60-69， ...
default: printf("Fail \n");             //若score为其他整数， ...
}
```

C语言中的求余数操作（模运算， $\%$ ）

□ 两个操作数只能为**整数**

- 较小的数模较大的数，结果一定为较小的数
- 对于正整数 m 和 n ， $(m/n) * n + m \% n$ 一般等于 m

□ 对于负整数，不同的编译器有不同的实现，操作结果有可能不同，所以**在这种情况下，取余数运算具有歧义性**。程序员要尽量保证所编写的程序没有歧义。

```
if(m >= 0 && n > 0)
    r = m % n;
else if(m < 0 && n < 0)
    r = (-m) % (-n);
else if(m < 0 && n > 0)
    r = -((-m) % n);
else
    r = -(m % (-n));
printf("The remainder is %d \n", r);
```

分支语句将负数的求余数运算统一成：
先求两个正数的余数，
再根据商的正负和实际需求考虑如何添加负号。

□ C语言的求余数运算在一些实际问题的处理中，常常能发挥巧妙的作用。

例3.1 设计程序，求所有的十进制三位水仙花数（这种数等于其各位数字的立方和，例如， $153 = 1^3 + 3^3 + 5^3$ ）。

//穷举法

```
for(int i = 1; i <= 9; ++i)
    for(int j = 0; j <= 9; ++j)
        for(int k = 0; k <= 9; ++k)
            if(i*100 + j*10 + k == i*i*i + j*j*j + k*k*k)
                printf("%d \n", i*100 + j*10 + k);
```

```
for(int n = 100; n <= 999; ++n)
{
    int i = n / 100;           //百位数字
    int j = n / 10 % 10;       //十位数字
    int k = n % 10;            //个位数字
    if(n == i * i * i + j * j * j + k * k * k)
        printf("%d \t", n);
} //利用除法和求余数运算，可以分离出三位数每一位上的数字。
```


C语言中的自增/自减操作 (++)/--)

- 前缀——操作符置于操作数的前面，使用操作数的值之前进行自增/自减操作，使用的是存储在操作数中的值（已改变的值）。

```
i = 3;  
++i;      //相当于i += 1, 也即i = i+1, i的值变为4  
--i;      //相当于i -= 1, 也即i = i-1, i的值变回为3  
j = ++i;   //相当于j = i = i+1; 则i、j的值均变为4
```

- 后缀——操作符置于操作数的后面，使用操作数的值之后进行自增/自减操作，使用的是临时空间中的值（未改变的值）。

```
m = 3;  
m++;      //则m的值变为4  
m--;      //则m的值变回为3  
n = m++;  //则m的值变为4、n的值仍为3
```

□ 自增/自减操作符通常单独使用

- 在循环语句中实现循环变量的高效自增/自减
- 用于指针类型的操作数，实现内存的高效访问

□ 不要用于复杂的表达式，以免产生歧义

关系与逻辑操作的注意事项及应用

关系操作

□ 指的是通常意义下的比较操作，即判断两个数据的大小多少关系**是否**成立

- 结果要么为真（true，存储为整数1），要么为假（false，存储为整数0）

□ C语言的关系操作符

- >（大于）
- <（小于）
- >=（大于或等于）
- <=（小于或等于）
- ==（等于，**切不可与赋值操作符的一个等于号=混淆！！**）
- !=（不等于，不是!=）

<code>fabs(item) >= 0.000001</code>
<code>fabs(item) != 0.000001</code> 有可能死循环

□ 为了避免将比较操作符==误写成=，即少写一个等于号，程序员常常将常量写在比较操作符的左边，这样，编译器可以帮助发现这个错误。比如，

```
if (n == 0)    //若误写成if (n = 0)，编译器不报错，且n变为0，该条件始终不成立
    ++n;
else
    n = 1 / n;
```

可以写成：

```
if (0 == n)    //若误写成if (0 = n)，编译器会报错，因为不能给常量赋值
    ++n;
else
    n = 1 / n;
```

逻辑操作

- 指的是命题的逻辑推理，通常用来辅助复杂关系的判断
 - 结果要么为真（true，存储为整数1），要么为假（false，存储为整数0）
- C语言的逻辑操作符
 - !（逻辑非，判断一个比较操作结果的否命题是否成立）
 - &&（逻辑与，判断两个比较操作结果是否同时成立）
 - ||（实现逻辑或操作，判断两个比较操作结果是否有成立的情况）
- 参与逻辑操作的操作数只有两种值：
 - 真（true，非0）
 - 假（false，0）

□ `!(a > b)` 表示a不大于b成立吗?

■ 当a为3、b为4时成立

□ `(age < 10) && (weight > 50)` 表示age小于10而且weight大于50成立吗?

■ 当age为8、weight为52时成立

□ `(ch < '0') || (ch > '9')` 表示ch在'0'和'9'之外成立吗?

■ 当ch为'7'时不成立

□ De Morgan定理 (逻辑操作存在以下操作规律) :

■ `!(a && b)` 等价于 `(!a) || (!b)`

■ `!(a || b)` 等价于 `(!a) && (!b)`

■ `!((a && b) || c)` 等价于 `(!a || !b) && !c`

短路求值 (short-circuit evaluation)

□ `&&`和`||`

- 如果第一个操作数已能确定操作结果，则不再计算第二个操作数的值。

- (假 `&& x`) 的结果为 假

`(x < 0) && (x*x > 1)`，当`x`为0时

- (真 `|| x`) 的结果为 真

`(x < 0) || (x*x > 1)`，当`x`为-1时

□ 短路求值

- 能够提高逻辑操作的效率

- 有时能为逻辑操作中的其他操作提供一个“**卫士**” (guard)

➤ `(number != 0) && (1/number > 0.5)` 在`number`为0时，不会进行除法运算

□ 例3.2 百鸡问题：鸡翁一值钱五；鸡母一值钱三；鸡雏三值钱一。百钱买百鸡，问鸡翁、鸡母、鸡雏各几何？

```
int cock, hen, chicken;
printf("    ***百鸡问题***\n");

for(cock = 0; cock <= 20; ++cock)
    for(hen = 0; hen <= 33; ++hen)
        for(chicken = 0; chicken <= 100; chicken += 3)
            if(cock + hen + chicken == 100 &&
                cock*5 + hen*3 + chicken/3 == 100 )
                printf("%d %d %d \n", cock, hen, chicken);
```

条件操作

□ ? :

- `d1 ? d2 : d3`

- 如果d1的值为true，则操作结果为d2，否则为d3。

- 如: `result = a > b ? a : b`

- 又比如: `result = a > b ? (a > c ? a : c) : (b > c ? b : c)`

□ 条件操作也遵循短路求值规则

- 如 `int a = 1, b = 2;`

- `int c = (a < b ? (a=3) : (b=4)) ;`

- 则: a、c为3, b仍为2

条件操作的应用

□ 下面程序中利用条件运算符定义了一个带参数的宏：

```
#define max(m, n) m>n?m:n
```

```
//为保险起见，最好写成#define max(m, n) ((m)>(n)?(m):(n))
```

```
//以防参数是复杂的表达式
```

```
int main( )  
{  
    int i, j;  
    scanf("%d", &i, &j);  
    printf("%d", max(i, j));  
}
```

逗号操作

□ 用于将两个表达式连接起来，并从左往右依次计算各表达式的值。

■ 比如，

$x = a+b, y = c+d, z = x+y$ (相当于 $z = a+b+c+d$)

□ 用来将复杂的表达式分开写

□ 并不是任何地方出现的逗号都是逗号操作符，有的是参数分隔符，有的是逗号字符本身。

位操作的注意事项及应用

位操作

- 将整型操作数看作二进制位序列进行操作
- 包括两类：
 - 逻辑位操作：~ & | ^
 - 移位操作：<< >>
- 序列的长度与机器及操作数的类型有关。（以32位机、int类型为例）
- 操作数如果是负数，则以补码形式参与位操作。

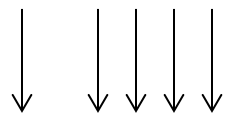
□ ~: 按位取反

$0 \rightarrow 1, 1 \rightarrow 0$

- 用来把一个二进制位序列中的每一位由0变1、由1变0

~ 9 (0000 0000 0000 0000 0000 0000 0000 1001) 的结果为:

-10 (1111 1111 1111 1111 1111 1111 1111 0110)



□ **&**: 按位与

- 对两个二进制位序列逐位进行**与**操作，对应位同时为1，则结果序列的对应位为1，否则为0。

0&0	→	0
0&1	→	0
1&0	→	0
1&1	→	1

9 (0000 0000 0000 0000 0000 0000 0000 1001)

↓ ↓ ↓ ↓ ↓

& 10 (0000 0000 0000 0000 0000 0000 0000 1010) 的结果为:

8 (0000 0000 0000 0000 0000 0000 0000 1000)

□ | : 按位或

对两个二进制位序列逐位进行或操作，对应位有1，则结果序列的对应位为1，否则为0。

0		0	→	0
0		1	→	1
1		0	→	1
1		1	→	1

9 (0000 0000 0000 0000 0000 0000 0000 1001)

| 10 (0000 0000 0000 0000 0000 0000 0000 1010) 的结果为:

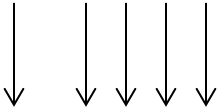
11 (0000 0000 0000 0000 0000 0000 0000 1011)

□ ^: 按位异或

- 对两个二进制位序列逐位进行异或操作，对应位不同，则结果序列的对应位为1，否则为0。
- 一个二进制位与0相异或，保持原值不变，与1相异或，结果和原值相反，所以，相当于按位且无进位的加法（二进制）

$0^0 \rightarrow 0$
$0^1 \rightarrow 1$
$1^0 \rightarrow 1$
$1^1 \rightarrow 0$

9 (0000 0000 0000 0000 0000 0000 0000 1001)



^ 10 (0000 0000 0000 0000 0000 0000 0000 1010) 的结果为:

3 (0000 0000 0000 0000 0000 0000 0000 0011)

逻辑位操作

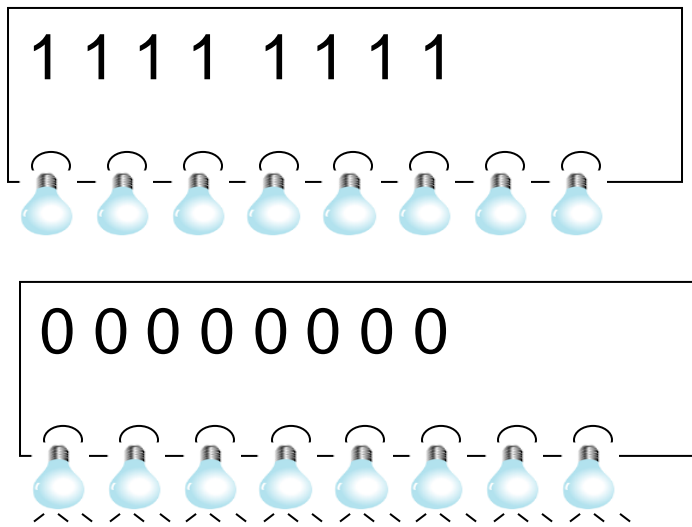
□ 注意逻辑位操作与逻辑操作的区别

- 逻辑位操作结果的含义：是一个数，并且也被看成一个二进制位序列
- 逻辑操作结果的含义：表示是否成立

逻辑位操作的用途

□ 逻辑位操作速度快、效率高、节省存储空间，通常用于嵌入式或自动测控系统。

- ~ : 所有位翻转;
- & : 按位清零;
- | : 按位置1;
- ^ : 特定位的翻转。



	xxxx	xxxx
&	0110	0010
<hr/>		
	0xx0	00x0

	xxxx	xxxx
 	0110	0010
<hr/>		
	x11x	xx1x

	xxxx	xxxx
^	0110	0010
<hr/>		
	xyyx	xyyx

```
int flag, temp;
scanf("%d", &flag); // 代替信号采集
temp = flag & 0x08; // 保留flag的第4位
if(temp == 0x08)
    printf("The concerned bit of flag is 1.\n");
else
    printf("The concerned bit of flag is 0.\n");
```

xxxx xxxx xxxx xxxx xxxx xxxx xxxx **xxxx**
0000 0000 0000 0000 0000 0000 0000 **1000**

```
#define KEY 0x08
```

```
int flag, temp;          0000 0000 0000 0000 0000 0000 0000 1000
```

```
scanf("%d", &flag);
```

```
temp = flag & KEY;      //保留flag的第4位
```

```
if(temp == KEY)
```

```
    printf("The concerned bit of flag is 1.\n");
```

```
else
```

```
    printf("The concerned bit of flag is 0.\n");
```

```
#define KEY 0x04
```

```
int flag, temp;          0000 0000 0000 0000 0000 0000 0000 0100
```

```
scanf("%d", &flag);
```

```
temp = flag & KEY;      //保留flag的第3位
```

```
if(temp == KEY)
```

```
    printf("The concerned bit of flag is 1.\n");
```

```
else
```

```
    printf("The concerned bit of flag is 0.\n");
```

移位操作

- 将左边的整型操作数对应的二进制位序列进行左移或右移操作，移动的位数由右边的整型操作数决定。
- 包括
 - << (左移)
 - >> (右移)

□ 操作举例

5 << 1 的结果为:

(0000 0000 0000 0000 0000 0000 0000 0101)
10 (0000 0000 0000 0000 0000 0000 0000 1010)

5 << 2 的结果为

(0000 0000 0000 0000 0000 0000 0000 0101)
20 (0000 0000 0000 0000 0000 0000 0001 0100)

左移 (Left Shift)

□ 操作规则

- $i \ll n$
- 把*i*各位全部向左移动*n*位
- 最左端的*n*位被移出丢弃
- 最右端的*n*位用0补齐

□ 用法

- 在一定范围内，左移*n*位相当于乘以 2^n
- 操作速度比真正的乘法和幂运算快得多

□ 操作举例

5 >> 1 的结果为:

	(0000	0000	0000	0000	0000	0000	0000	0101)
2	(0000	0000	0000	0000	0000	0000	0000	0010)

5 >> 2 的结果为

	(0000	0000	0000	0000	0000	0000	0000	0101)
1	(0000	0000	0000	0000	0000	0000	0000	0001)

右移 (Right Shift)

□ 操作规则

- $i \gg n$
- 把*i*各位全部向右移动*n*位
- 最右端的*n*位被移出丢弃
- 最左端的*n*位用符号位补齐(算术右移)
- 或最左端的*n*位用0补齐(逻辑右移)，右移操作往往具有歧义

□ 用法

- 在一定范围内，右移*n*位相当于除以 2^n ，并舍去小数部分
- 操作速度比真正的除法快得多

位操作小结

- `~` `&` `^` `|` `<<` `>>`
- 位操作的操作数是二进制位序列
- 位操作速度快，节省存储空间
- 只能对整型数据进行位操作
- 负数以补码形式参与操作
- 注意逻辑位操作与逻辑操作区别

`~8` 为 `-9`

`~` `0...01000`
`(1...10111)`

`!8` 为 `0(false)`

`8&1` 为 `0`

`0...0 1000`
`& 0...0 0001`
`(0...0 0000)`

`8&&1` 为 `1(true)`

`8|1` 为 `9`

`0...0 1000`
`| 0...0 0001`
`(0...0 1001)`

`8||1` 为 `1(true)`

赋值操作

□ 指赋予某变量一个数据

□ 包括

■ = （实现简单赋值操作）

■ # = （实现复合赋值操作，往往能提高效率）

#可以是 +、-、*、/、%、>>、<<、&、|、^

$x \# = y$ 功能上相当于 $x = x \# y$

eg. $a \& = 3$ 相当于 $a = a \& 3$, $b \wedge = 2$ 相当于 $b = b \wedge 2$

表达式的有关问题

表达式的分类

表达式的值（左值表达式，操作符的副作用）

表达式的求值顺序（优先级、结合性、歧义）

表达式的书写

表达式的有关问题-分类

- 多个操作符与操作数连接起来，可以形成较为复杂的表达式（Expression），包括：
 - 逗号表达式
 - 赋值表达式
 - 条件表达式
 - 关系表达式
 - 逻辑表达式
 - 算术表达式
 - 函数调用表达式
 - ...

- 表达式可以作为操作数参加运算，如 $d = d + 1$
 - 最简单的表达式是一个操作数，如一个字面常量1，一个变量d

程序中操作的描述

程序中的基本操作（通过操作符operator实现）

按功能分类

■ 算术操作

+ -

++ --

+ - * / %

■ 关系操作

> >= < <=

== !=

■ 逻辑操作

!

&&

||

■ 条件操作 ? :

■ 逗号操作 ,

■ 位操作

~

& | ^

<< >>

■ 赋值操作 =

+= -= *= /= %=

<<= >>= &= |= ^=

单目
单目
双目

双目
双目

单目
双目
双目

三目
双目

单目
双目
双目

双目

目：操作符能连接的操作数的个数

表达式的有关问题-值

□ 每个表达式有一个值

- 常量表达式（表达式中不含变量）在编译期间可确定其值；
- 算术表达式的值通常是一个整数或小数，具体类型由表达式中操作数的类型决定，一般存储在内存的临时空间里（前缀自增/自减操作的结果存储在操作数中）；
- 关系或逻辑表达式的值一般也是存储在内存的临时空间里，要么为“真”（true，计算机中用1存储），要么为“假”（false，计算机中用0存储）；
- 赋值表达式的值一般存储在左边的操作数中；
- 条件表达式的值是第二个或第三个子表达式的值，一般存储在内存的临时空间里；
- 整个逗号表达式的值是**最后**一个子表达式的值，一般存储在内存的临时空间里（比如 $a=3*5$ ， $a*4$ 这个逗号表达式的值为60，a为15）。

□ 左值表达式

- 表达式的值存储在操作数中（而不在内存的临时空间里），即表达式的值有明确的内存地址。
- 一个变量
- 一个赋值表达式
- 一个前缀自增/自减操作表达式

□ 操作符的副作用

- 一般的基本操作符不改变参与操作的操作数的值
- 少数操作符会改变参与操作的操作数的值，这种操作符通常被认为带有副作用
 - 赋值操作符
 - 自增/自减操作符
- 这类操作符的单个操作数或左边的操作数必须是左值表达式，否则这个副作用的结果无处安放
 - $x=3$ 、 $(x=2)=3$ 、 $++x$ 、 $x++$ 、 $++(x=2)$ 、 $(x=2)++$ 、 $++(++x)$ 、 $(++x)++$ ✓
 - $3++$ 、 $(a+b)--$ 、 $++3$ 、 $--(a+b)$ 、 $3=n$ 、 $(!m)=n$ 、 $(x++)++$ 、 $++(x++)$ ✗
- 这个副作用通常是我们需要的，但是有时候会让代码产生歧义

表达式的有关问题-求值顺序

- 一个表达式可以包含多个操作，先执行哪一个操作呢？
- 系统会依据各个操作符的**功能**及其**优先级**和**结合性**来计算表达式的值
- C语言有以下具体规则：
 - 1) **对于相邻的两个操作**，操作规则为：
 - a) 判断两个操作符的优先级高低，然后先处理优先级高的操作符；
 - b) 如果两个操作符的优先级相同，再判断两个操作符的结合性，
结合性为左结合的先处理左边的操作符，为右结合的先处理右边的操作符；
 - c) 加圆括号的操作优先执行。
 - 2) **对于不相邻的两个操作**，C语言未规定操作顺序，由具体编译器决定
(比如，对于表达式 $(a+b) * (c-d)$ ，C语言没有规定+和-的操作顺序。)
($\&\&$ 、 $||$ 、 $?:$ 和，连接的表达式除外，它们都是先计算左边第一个子表达式)

□ 操作符的优先级 (precedence)

- 是指操作符的优先处理级别
- C语言将基本操作符分成若干个级别
 - 第1级为最高级别，第2级次之，以此类推。
- C语言操作符的优先级一般按“单目、双目、三目、赋值”依次降低，其中双目操作符的优先级按“算术、移位、关系、逻辑位、逻辑”依次降低。

()	高
单目操作符	
* / %	
+ -	
<< >>	
> < >= <=	
== !=	
&	
^	
&&	
? :	
=	
,	低

□ 操作符的结合性 (associativity)

- 是指操作符与操作数的结合特性

- 包括：

- 左结合：先让左边的操作符与最近的操作数结合起来

- 右结合：先让右边的操作符与最近的操作数结合起来

双目 $x / y * z$

单目 $! \sim a$

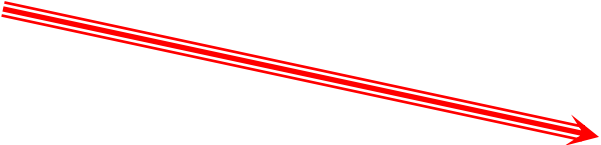
三目

赋值 $a = b = 3$

条件操作符的右结合（条件操作符允许嵌套）

```
int a = 2;
```

```
int tmp = (a==2) ? 1 : 0 ? a++ : a++;
```



```
int a = 2;
```

```
int tmp = ((a==2) ? 1 : 0) ? a++ : a++;
```

tmp为2, a为3

```
int a = 2;
```

```
int tmp = (a==2) ? 1 : (0 ? a++ : a++);
```

tmp为1, a为2

结合之后，赋值表达式右侧是一个操作，而不是相邻的两个操作。

对于相邻的两个操作，加圆括号的操作优先执行。对于一个操作，按自身操作特征进行操作：短路求值

□ 取正/负与加/减操作的区别

- 功能
- 目
- 优先级
- 结合性

□ 不相邻

■ $a = (b=10) / (c=2)$ 的计算次序为: $b=10$ 或 $c=2$, $/$, $a=5$
(圆括号优先级最高, 然后是除, 赋值优先级低, 最后 a 、 b 、 c 的值分别为5, 10, 2)

■ $a = -7\%20 + 3*5 - 4/3$ 的计算次序为: -7 , $7\%20$ 或 $3*5$ 或 $4/3$, $+$, $-$, $=$
(取负优先级最高, 然后是取余、乘、除, 然后是加、减, 赋值优先级最低)

-
- 当表达式中**含有带副作用的操作符**时，由于C语言没有规定不相邻的操作符的操作顺序，不同的编译器可能会得出不同的结果，同一个编译器对不同的表达式还可能采用不同的优化策略。
 - 这样的**表达式具有歧义性**。所以，最好把带有副作用的操作符（++/--、=）作为单独的操作来用，避免将它们用于复杂的表达式中。

□ `int x = 1;`

□ `int tmp = (x + 1) * (x = 10);`

□ 如果先计算`+`，则`tmp`为20，如果先计算`=`，则`tmp`为110。

□ `int m = 5;`

□ `int n = (++m) + (++m) + (++m);`

- 在计算前面两个++后，接下来如果先计算第三个++，然后依次计算两个+，则n为24（TC、VC2008开发环境下可验证）
- 在计算前面两个++后，接下来如果先算第一个+，然后计算第三个++，再计算第二个+，则n为22（Dev C++、VC6.0开发环境下可验证）。

□ 对于多个参数的函数，函数参数的求值顺序有两种：

- 自左至右
- 自右至左

```
int F(int x, int y)
{
    int z;
    if(x > y) z = 7;
    else z = 8;
    return z;
}
```

```
int main()
{
    int i = 1, h;
    h = F(i, i++);
    printf("%d \n", h);
    return 0;
}
```

不同开发环境执行结果可能不同（8或7）

要么

```
int j = i++;
```

```
h = F(i, j);
```

要么

```
int j = ++i;
```

```
h = F(i, j);
```

可避免歧义

□ **标准没有规定该求值次序。**当实参中带有自增、自减或赋值运算符时，会产生歧义，故在调用函数（包括printf库函数）中尽量不要将该类运算放在实参表中。

表达式的有关问题-书写

□ 程序设计语言中的数值运算符和数学中的运算符不尽相同：

- $\sqrt{\quad}$, \times , $^{\quad}$

□ 良好的表达式**书写习惯**有助于表达式的正确求解

- 最好在操作符与操作数之间留有空格，提高可读性，不过，加空格一般不会影响操作符的优先级。比如， $a+b \quad *c$ 与 $a+b*c$ 等价，与 $(a+b)*c$ 不等价。

$(a+b) \quad * \quad c$

$a \quad + \quad b*c$

- 对于连续多个操作符，最好用圆括号来明确操作符的种类和优先级。比如， $a- \quad --b$ 最好写成 $a - (--b)$ ，否则可能会有歧义。多数编译器按**贪婪准则**（尽可能多地自左而右将若干个字符组成一个操作符）确定表达式中的操作符种类和优先级，比如，编译器会把 $a---b$ 解释成 $(a--)-b$ ，而不是 $a- (--b)$ 。
- 编译器对表达式中操作符的数量往往有限制，过长的表达式可以分成几个表达式来写，再用逗号连接。用逗号操作符表示的操作往往更加清晰。

小结

□ 程序中操作的描述

■ 基本操作的描述、注意事项及其应用

■ 表达式的有关问题

➤ 分类

➤ 表达式的值

- ✓ 左值表达式

- ✓ 操作符的副作用

➤ 求值顺序

- ✓ 优先级

- ✓ 结合性

➤ 表达式的书写

■ 复杂操作的描述方法

程序所涉及的操作有时比较复杂，不能直接用基本操作符来表达，需要程序员综合运用基本操作符、流程控制方法和模块设计方法设计特别的算法来实现

分类

穷举

迭代

课程后续内容将结合复杂数据进一步介绍一些常见操作的实现例程

排序

信息检索

更为复杂的操作则需要用专门的方法（比如机器学习）来实现

数据清洗、数据传输、模式识别、隐私保护…

□ 程序中的基本操作符

- 算术操作符、关系与逻辑操作符、位操作符、赋值操作符、条件操作符、逗号操作符
- C语言中的基本操作符除了有其基本含义外，当用于派生数据类型的数据时，其含义可以改变，比如 * 用于指针类型数据时，往往不是乘法操作符，而是取值操作符

□ 要求：

- 了解基本操作符的功能与操作特点
- 掌握恰当选用C语言基本操作符实现简单计算任务的方法
- 会通过恰当的书写方式避免程序存在歧义
 - 一个程序代码量≈30行
- 继续保持良好的编程习惯
 - 表达式的书写...