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）

抽象与封装（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）

基本操作符（operators）

- 算术操作符　+　-　++　--　+　-　*　/　%
- 关系操作符　>　>=　<　<=　==　!=
- 逻辑操作符　!　&&　||
- 条件操作符　?：
- 位操作符　~　&　|　^　<<　>>
- 赋值操作符　=　+=　-=　*=　/=　%=　<<=　>>=　&=　|=　^=
- 逗号操作符　,

通过基本操作符（operators）描述程序中的基本操作

算术操作的表达和运用

- **乘法操作**
  - C语言中的乘法操作符 * 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;
}
```

除法操作

- 0 不能做除数
- 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，并且不进行四舍五入。
  - 例如：3/2的结果为1； 1/2的结果为0
  - 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数相除而带来的意想不到的错误。***

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

```
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;
}
```

- 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，…
}
```

讨论-1

- 还有什么基本操作也能发挥前述类似的数据映射作用?

求余数操作（模运算）

- 两个操作数只能为*整数*
- 余数的范围：0 ~ 除数-1

```
int id = 0;
scanf("%d", &id);
switch(id % 2)
{
      case 0: printf("... \n"); break;    //若id为偶数，…
      case 1: printf("... \n"); break;    //若id为奇数，…
}
```

例3.1 设计程序，求所有的十进制三位水仙花数（Narcissistic number，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);
}//利用除法和模运算，可以分离出三位数每一位上的数字。
```

- 对于正整数m和n，(m/n)*n+m%n一般等于m，较小的数%较大的数结果为较小的数
- 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，模运算具有歧义。***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);
```

分支语句将负数的模运算统一成：
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。

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，使用的是已改变的值。

```
i = 3;
++i;          //则i的值变为4
--i;          //则i的值变回为3
j = ++i;      //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**

关系与逻辑操作的表达和运用

- **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!==）

`fabs(item) >= 0.000001`
`fabs(item) != 0.000001` 有可能死循环

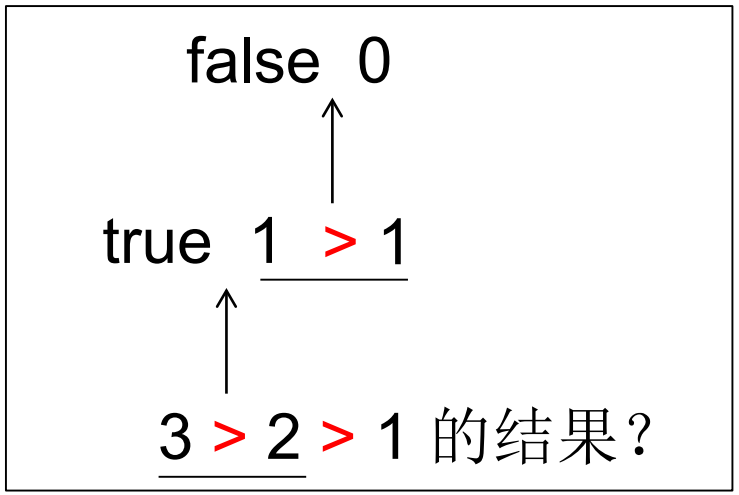
- 为了避免将比较操作符==误写成=，即少写一个等于号，程序员常常将常量写在比较操作符的左边，这样，编译器可以帮助发现这个错误。比如，

```
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;
```

讨论-2



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）
  - 参与逻辑操作的操作数只有两种值：
    - ✓ 真（true，非0 ）
    - ✓ 假（false，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）

```
!(a > b)    表示a不大于b成立吗?
```

  && （逻辑**与**，判断两个比较操作结果是否同时成立）

```
(age < 10) && (weight > 50)
```

  || （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）

```
(ch < '0') || (ch > '9')
```

- 短路求值 (short-circuit evaluation)
  - 对于&&和||，若第一个操作数已能决定结果，则不再处理第二个操作数（保护数据；提高效率）

```
(n != 0) && (1/n > 0.5) //n为0
```

```
(x < 0) || ( x*x > 1) //x为-1
```

- **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`

讨论-3

- 利用逻辑操作，改写 带break 的循环流程

```
int d, sum = 0, i = 1;
while(i <= 10)
{
      scanf("%d", &d);
      if(d <= 0) break;
      sum += d;
      ++i;
}
printf("sum: %d \n", sum);
```

```
int d, sum = 0, i = 1;
scanf("%d", &d);
while(i <= 10 && d > 0)
{
      sum += d;
      ++i;
      scanf("%d", &d);
}
printf("sum: %d \n", sum);
```

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）
  - result = a>b ? a : b
  - 下面程序中利用条件操作符定义了一个带参数的宏:

```
#define max(m, n) (((m)>(n))?(m):(n))
printf("%d", max(i, j));
```

- **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
```

例3.2 设计程序求三个不相等的整数中第二大的数。

方法一：

```
if(i<j)
      if(j<k)
            return j;
      else if(i>k)
            return i;
      else
            return k;
else
      if(j>k)
            return j;
      else if(i<k)
            return i;
      else
            return k;
```

```
if(i<j)
      if(j<k) return j;
      if(k<j) return k;
else
      if(i<k) return i;
      if(k<i) return k;
```

方法二：

```
if((i>j && j>k) || (i<j && j<k))
      return j;
else if((i>j && i<k) || (i<j && i>k))
      return i;
else
      return k;
```

```
if(i<j<k) return j;
else if(j<i<k) return i;
else return k;
```

```
if((i<j<k) || (k<j<i)) return j;
else if((j<i<k) || (k<i<j)) return i;
else return k;
```

方法三：

```
if((i-j)*(j-k) > 0)
        return j;
else if((j-k)*(k-i) > 0)
        return k;
else
        return i;
```

方法四：

```
sum-min-max
```

方法五：

```
return (i>j ? (j>k ? j : (i<k?i:k)) : (j<k ? j : (i>k?i:k)));
```

条件操作的应用

- **替代：**

  - 加入调试信息

```
#ifdef DEBUG
......   //调试信息，主要是输出
#endif
```

    - 调试程序时，定义宏名DEBUG，调试结束，去掉宏名DEBUG的定义。

  - 问题：写输出语句再逐一观察判断输出的值是否正确
    比较麻烦！

- 为了便于调试程序，标准库的头文件assert.h中定义了一个带参数的宏assert（断言）：

```
……
#ifdef NDEBUG
  #define assert(exp) ((void)0)
#else
  #define assert(exp) ( (exp)?(void)0:<输出诊断信息并调用库函数abort> )
#endif
……
```

- 比如，

```
assert(1 == x);
```

- 程序执行到该断言处，如果x的值不等于1，则会显示下面的信息并终止程序的运行：
  `Assertion failed: 1 == x, file XXX, line YYY`
  其中，XXX表示断言所在的源文件名，YYY表示断言所在的行号

- assert的上述功能只有在宏名NDEBUG没有被定义时才有效，否则它什么也不做。

- 优点：程序员写起来不麻烦，也容易发现程序出错行

```
#include <stdio.h>
#define N 5
int main()
{       int score;
        int max;// = -32768;
        int min;// = 32767;
        double sum = 0;
        for(int i=1; i <= N; ++i)
        {       scanf("%d", &score);
                sum += score;
                if(score > max)
                        max = score;
                if(score < min)
                        min = score;
        }
        printf("%d, %d \n", max, min);          //调试型输出，软件交付前要去掉
        sum = (sum-max-min)/(N-2);
        printf("%.2f \n", sum);
        return 0;
}
```

调试方法一

```
#define DEBUG
#include <stdio.h>
#define N 5
int main()
{       int score;
        int max;// = -32768;
        int min;// = 32767;
        double sum = 0;
        for(int i=1; i <= N; ++i)
        {       scanf("%d", &score);
                sum += score;
                if(score > max)
                        max = score;
                if(score < min)
                        min = score;
        }
        #ifdef DEBUG
                printf("%d, %d \n", max, min);          //调试型输出
        #endif
        sum = (sum-max-min)/(N-2);
        printf("%.2f \n", sum);
        return 0;
```

调试方法二：
加上条件编译，
便于后期维护

```
//#define NDEBUG
#include <assert.h>
#include <stdio.h>
#define N 5
int main()
{       int score;
        int max;// = -32768;
        int min;// = 32767;
        double sum = 0;
        for(int i=1; i <= N; ++i)
        {       scanf("%d", &score);
                sum += score;
                if(score > max)
                        max = score;
                if(score < min)
                        min = score;
        }
        assert(6 == max);
        sum = (sum-max-min)/(N-2);
        printf("%.2f \n", sum);
        return 0;
}
```

在头文件包含行之前 加上宏名 NDEBUG 的定义，方可取消断言的作用

调试方法三：
用断言，
更方便

程序调试方法小结

- 注释
- 空语句、空函数
- 调试性输出
- 条件编译
- 断言

- Debug工具软件

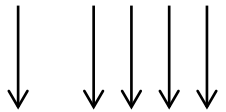
位操作的表达和运用

- 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）
- 操作数只能是整型数，以补码形式参与位操作
- 位操作速度快，节省存储空间

- **~：按位取反**　　0 → 1，1 → 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)

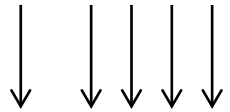
10 (0000 0000 0000 0000 0000 0000 0000 1010) 的结果为:

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 → 0
0^1 → 1
1^0 → 1
1^1 → 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)

- 注意逻辑位操作与逻辑操作区别
  - 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

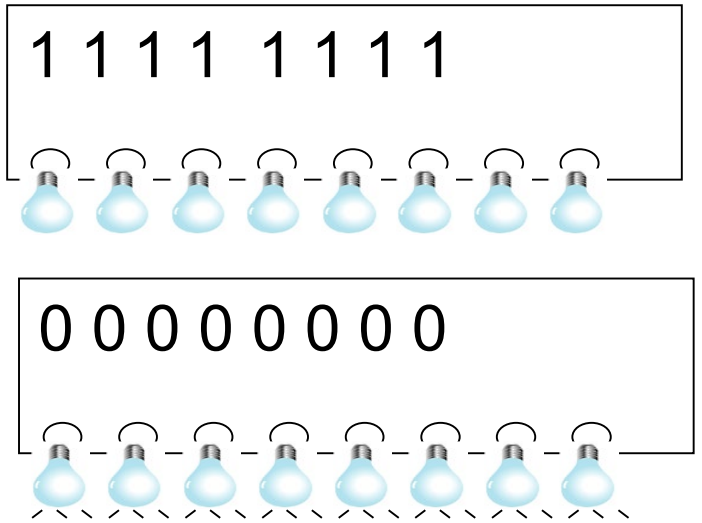
```
~8 为 -9

~ 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)
```

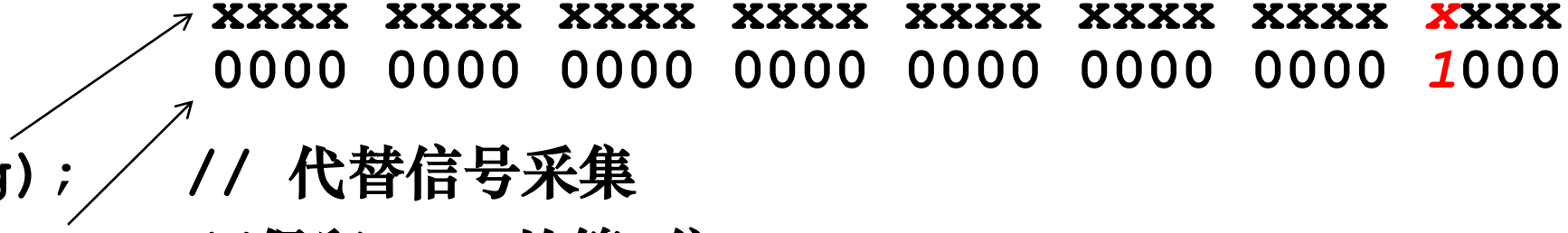
```
8|1为9

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

8||1为1(true)
```

逻辑位操作的用途

- **逻辑位操作用于嵌入式或自动测控系统示例**
  - ~ ： 所有位翻转；
  - & ： 按位清零；
  - | ： 按位置1；
  - ^ ： 特定位的翻转。




```
    xxxx xxxx          xxxx xxxx          xxxx xxxx
&   0110 0010      |   0110 0010      ^   0110 0010
-------------      -------------      -------------
    0xx0 00x0          x11x xx1x          xyyx xxyx
```




```
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");
```

```
#define KEY  0x08
int flag, temp;
scanf("%d", &flag);
temp = flag & KEY;
if(temp == KEY)
    printf("The concerned bit of flag is 1.\n");
else
    printf("The concerned bit of flag is 0.\n");
```

0000 0000 0000 0000 0000 0000 0000 1000

//保留flag的第4位

```
#define KEY  0x04
int flag, temp;
scanf("%d", &flag);
temp = flag & KEY;
if(temp == KEY)
    printf("The concerned bit of flag is 1.\n");
else
    printf("The concerned bit of flag is 0.\n");
```

0000 0000 0000 0000 0000 0000 0000 0100

//保留flag的第3位

左移 (Left Shift)

- 操作规则
  - i << 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)

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)

右移 (Right Shift)

- 操作规则
  - i >> 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**

- 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)

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 ^= 2相当于b = b ^ 2

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。**

操作符的分类

按功能分类

- 算术操作符
  + - ++ --
  + - * / %
- 关系操作符
  > >= < <= == !=
- 逻辑操作符
  !
  && ||
- 条件操作符
  ? :
- 位操作符
  ~
  & | ^ << >>
- 赋值操作符
  = += -= *= /= %= <<= >>= &= |= ^=
- **逗号操作符**
  ,

按目：操作符能连接的操作数的个数

- 单目
- 双目
- 双目
- 单目
- 双目
- 三目
- 单目
- 双目
- 双目
- 双目

表达式（Expression）的有关问题

- **表达式的称谓**
  - 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**

表达式的值

- 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，一般存储在寄存器或内存的临时空间里（C++: 赋值、前缀自增/减的结果存储在操作数中）；
- 关系或逻辑表达式的值存储在寄存器或内存的临时空间里（1或0）；
- 条件表达式的值是第二或第三个子表达式的值，一般存储在寄存器或内存的临时空间里；
- 整个逗号表达式的值是*最后*一个子表达式的值，一般存储在寄存器或内存的临时空间里（eg. `a=3*5, a*4` 这个逗号表达式的值为60）

左值表达式

- 表达式的值存储在操作数中（而不在寄存器或内存的临时空间里），即表达式的值有对应的明确的内存地址。

- 一个对象
  - **变量**
  - （*地址表达式）
- 一个赋值表达式（C++）
- 一个前缀自增/减表达式（C++）

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
- 少数操作符会改变其操作数的值，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

  3++、++3、3=n、(a+b)--、--(a+b)、(!m)=n 、(x++)++、++(x++) ✗

  x=3、++x、x++ ✓

C++： x=3、++x、x++、(x=2)=3、++(x=2)、(x=2)++、++(++x)、(++x)++ ✓

- 对于多个参数的函数，其参数的求值顺序有两种：
  - 自左至右
  - 自右至左
- *C语言标准没有规定该求值次序。*当实参中自增、自减或赋值操作符时，会产生歧义

```
int F(int x, int y)
{     int z;
      if(x > y)  z = 1;
      else z = 0;
      return z;
}
```

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

```
int j = i++;
h = F(i, j);
```
或
```
int j = ++i;
h = F(i, j);
```
可避免歧义

不同开发环境执行结果可能不同（0或1）

- **故在调用函数（包括printf库函数）中尽量不要将带有副作用的操作放在实参中。**

表达式的求值顺序

- 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

  - 首先要遵循操作符本身的处理规则

    （对于&&、||、?: 和，连接的表达式，先执行左边第一个子表达式）

  - *对于相邻的*两个操作，C语言标准规定操作规则为：

    判断两个操作符的优先级高低，先执行优先级高的操作符，加圆括号的操作优先执行；

    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；

  - *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。）

操作符的优先级（precedence）

- 是指操作符的优先处理级别

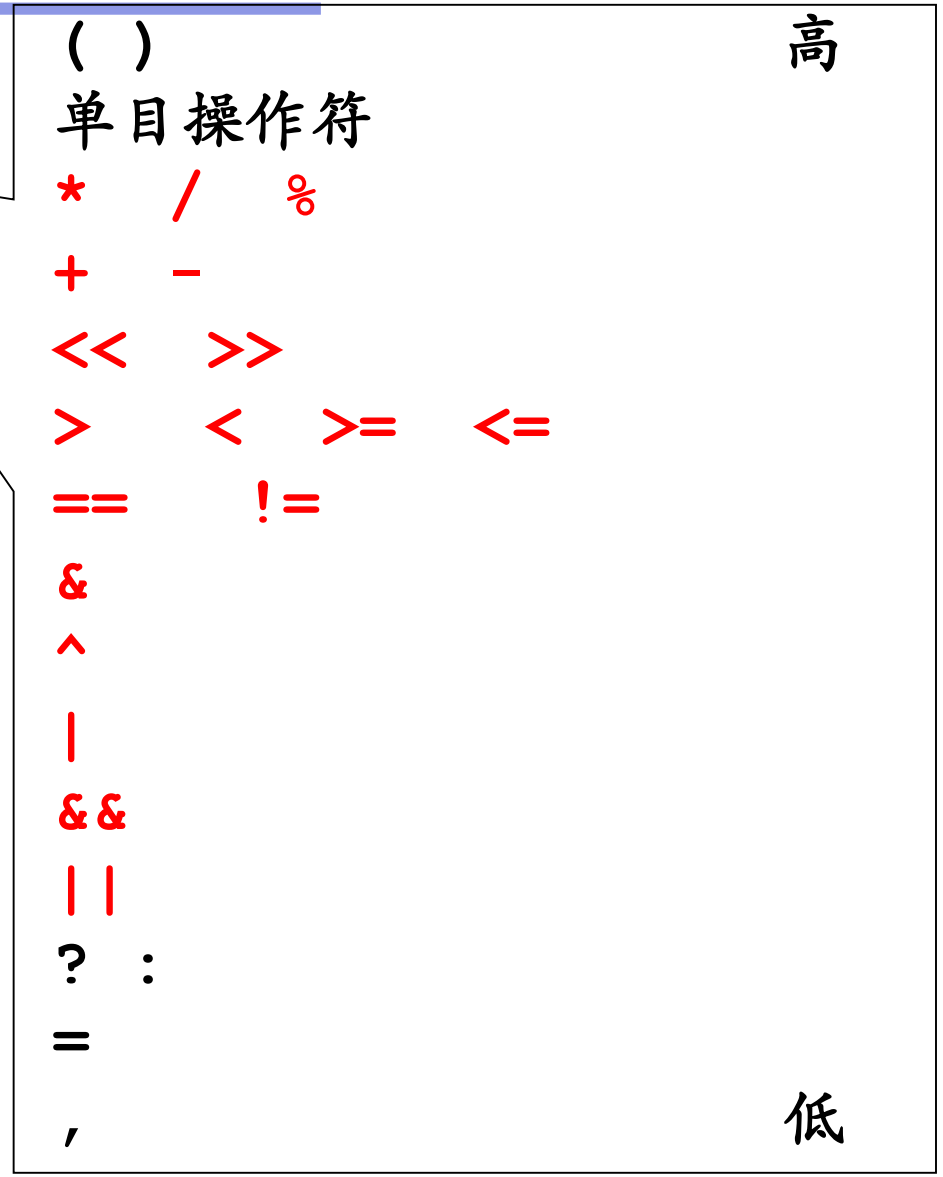
- C语言将基本操作符分成若干个级别
  - 第1级是（）为最高级别，第2级次之，逗号是最低级别

- 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。



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　　双目　`x/y*z`
  - 右结合：先让右边的操作符与最近的操作数结合起来

单目　`!~a`

赋值　`a = b = 3`

三目　`(a==2) ? 1 :((b==2) ? a : b))`

注意：结合之后，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，按自身操作规则进行操作，比如短路求值。

- **取正/负与加/减操作的区别**
  - 功能
  - 目
  - 优先级
  - 结合性

- 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*表达式往往具有歧义*。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);`
- 如果先计算+，则n为22，如果先计算++，则n为24

讨论-4

- 分析下面程序片段的执行结果:

```
int a = 12;
a = a += a -= a*a
printf("%d", a);
```

-264

- x = (a=3), 6*a 的值为:

18

表达式的有关问题-书写

- **程序设计语言中的数值操作符和数学中的运算符不尽相同：**
  - `x ^ 3`

- **良好的表达式*书写习惯*有助于提高其易读性**
  - `x = a + b * c; //a+b *c` 与 `a+b*c` 等价，与 `(a+b)*c` 不等价
  - `x = a - (--b); //a---b` 一般与 `(a--)-b` 等价（多数编译器按*贪婪准则*）

  法宝：加圆括号！

不过，加空格并不影响操作符的优先级

- **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。**
  - `x = a + b, y = c + d, z = x + y`（相当于 `z = a + b + c + d`）

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…

Thanks!