

南京大学 计算机科学与技术系

Department of Computer Science & Technology, NJU

Chapter 3

程序中操作的描述



刘奇志

程序中操作的描述

- 基本操作及其应用
- 表达式的有关问题
- 复杂操作的描述方法简介

程序中的基本操作（通过操作符operator实现）及其应用

- 算术操作
- 关系操作
- 逻辑操作
- 位操作
- 赋值操作
- 条件操作
- 逗号操作

按功能分类

操作符的目

一个操作符能连接的操作数的个数

➤ 算术操作符

– 取正/取负操作符

– 自增/自减操作符

➤ 关系操作符

➤ 逻辑操作符

➤ 位操作符

➤ 赋值操作符

➤ 条件操作符

➤ 逗号操作

双目

单目

单目

双目

双目/单目

双目/单目

双目

三目

双目

连接两个操作数

连接一个操作数

连接三个操作数

按目分类

● 算术操作主要包括通常意义下的数值运算，由算术操作符来实现

● 包括

➤ 取正/取负：+ -

➤ 四则运算：+ - * / %






➤ 自增/自减：++ --

取负/取正

• -: 将一操作数取负，即由正变负、由负变正

• +: 将一操作数取正，实际上未变，很少用

四则运算（实际上有五种运算）

-  $+$: 两个数相加
-  $-$: 两个数相减
-  $*$: 两个数相乘
-  $/$: 两个数相除
-  $\%$: 求两个整数相除的余数，又叫模运算

加/减法

例3.1 编写C程序，模拟计算器实现两个数相加/减。

```
double x, y, z;
char operatr;
printf("input an expression: x+(-)y\n");
scanf("%lf%c%lf", &x, &operatr, &y);

switch(operatr)
{
    case '+': z = x + y; break;
    case '-': z = x - y; break;
    default: printf("error!\n");
}

printf(" = %f \n", z);
```

10.8+0.13
= 10.930000

//实现连续的加/减法

```
printf("input expression: x+(-)y +(-)z... =\n");
scanf("%lf", &x);
while((operatr = getchar( )) != '=')
{
    scanf("%lf", &y);
    switch(operatr)
    {
        case '+': x += y;    break;
        case '-': x -= y;    break;
        default:  printf("error! \n"); goto END;
    }
}
printf("%f \n", x);
END: ;
```

10.8+0.13-10=
0.930000

//实现连续的加/减法

❁ C语言中乘法用*表示，*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`cmath`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 myPow(int x, int n)
{
    int z = 1;
    while(n >= 1)
    {
        z *= x;    // z = z * x;
        --n;
    }
    return z;
}
```

- C语言中的除法用/表示
- C语言中的除法可以用于两个整数或实数相除（0不能做除数）
- 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。

🌈 例3.2 利用近似公式 计算圆周率，直到最后一项的绝对值小于 10^{-6} 。

```
...  
#include <cmath>  
double myPi();
```

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

```
int main( )  
{  
    printf("圆周率为: %f \n", myPi());  
    return 0;  
}
```

```
double myPi()  
{  
    int sign = 1;  
    double item = 1.0, sum = 1.0;  
    for(int n = 1; fabs(item) > 0.000001; ++n)    // 1e-6  
    {  
        sign = -sign;                                //运用了取负操作  
        item = sign * 1 / (2 * n + 1); //应改成1.0  
        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， ...
}
```

求余数运算（模运算）

- C语言中用%表示计算两个整数相除的余数
 - 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。
- 求余数运算在一些实际问题的处理中，常常能发挥比较巧妙的作用。

● 例3.3 设计程序，求所有的三位水仙花数（一个三位水仙花数等于其各位数字的立方和，比如， $153 = 1^3 + 3^3 + 5^3$ ），要求不用嵌套的循环。

● 分析：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。

```
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);
}
```


例3.4 求两个整数相除的商和余数，并输出。

```
int m, n, q, r;
printf("Please input two numbers: \n");
scanf("%d%d", &m, &n);
while(n == 0)
{
    printf("The divider can not be zero, please input another one: \n");
    scanf("%d", &n);
} //确保输入的除数不为0，才执行后面的除法
q = m / n;
```

```
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 quotient is %d \n", q);
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，再根据商的正负考虑是否添加负号，以避免程序的歧义。

自增/自减操作

++/--: 变量的值自增1/自减1

- **前缀**——前缀操作符置于操作数的前面，除了修改操作数的值外，整个操作结果是操作后操作数的值。

```
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**——后缀操作符置于操作数的后面，除了修改操作数的值外，整个操作结果是操作前操作数的值。

```
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/自减操作符通常在循环语句中单独使用，实现循环变量的高效自增/自减，
 - 或者用于指针类型的操作数，实现内存的高效访问

🌈 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，由关系操作符来实现。

🌈 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!=)

例3.5 假定邮寄包裹的计费标准为：

重量 (克)	收费 (元)
$w < 15$	5
$15 \leq w < 30$	9
$30 \leq w < 45$	12
$45 \leq w < 60$	14 (每满1000公里加收1元)
$w \geq 60$	15 (每满1000公里加收2元)

编写C程序，根据输入的包裹重量 w 和邮寄距离 d ，计算并输出收费数额。

```
...
int Charge(int weight, int distance);

int main( )
{
    int w, d;
    printf("Please input the weight and the distance : \n");
    scanf("%d%d", &w, &d);
    while(w <= 0 || d <= 0)
    {
        printf("The input is wrong! Please input again: \n");
        scanf("%d%d", &w, &d);
    }

    printf("%d \n", Charge(w, d));
    return 0;
}
```

```
int Charge(int weight, int distance)
{
    int money = 0;
    if(weight < 15) money = 5;
    else if(weight < 30) money = 9;
    else if(weight < 45) money = 12;
    else if(weight < 60) money = 14 + distance/1000;
    else money = 15 + (distance/1000) * 2;
    return money;
}
```

```
int Charge(int weight, int distance)
{
    int money = 0;
    switch(weight/15)
    {
        case 0:    money = 5; break;
        case 1:    money = 9; break;
        case 2:    money = 12; break;
        case 3:    money = 14 + distance/1000; break;
        default:   money = 15 + (distance/1000) * 2;
    }
    return money;
}
```

- 在实际应用中，往往需要注意关系操作的边界问题。

```
double myPi()  
{  
    int sign = 1;  
    double item = 1.0, sum = 1.0;  
    for(int n = 1; fabs(item) > 0.000001; ++n)    // 1e-6  
    {  
        sign = -sign;    //运用了取负操作符  
        item = sign * 1.0 / (2 * n + 1);  
        sum += item;  
    }  
    return 4 * sum;  
}
```

`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;
```

逻辑操作

- 逻辑操作指的是命题的逻辑推理，通常用来辅助复杂的条件判断，由逻辑操作符来实现。
- C语言的逻辑操作符包括
 - !（实现逻辑**非**操作，用来判断一个比较操作结果的否命题是否成立，注意与取负操作的区别）
 - &&（实现逻辑**与**操作，用来判断两个比较操作结果是否同时成立）
 - ||（实现逻辑**或**操作，用来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：
 - 成立（true，非0，通常存储为1）
 - 不成立（false，0）

❁ `!(a > b)` 表示a不大于b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吗?

➡ 当age为8、weight为52时成立

❁ `(ch < '0') || (ch > '9')` 表示ch在'0'和'9'之外吗?

➡ 当ch为'7'时不成立

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。
- (不成立 && x) 的结果为 不成立
- (成立 || x) 的结果为 成立

🌈 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。

De Morgan定理


 逻辑操作存在以下操作规律:

- | | | |
|---|---|---|
| <div> <div>➤</div> <div>!(a&&b)</div> </div> | <div> <div></div> <div>等价于</div> </div> | <div> <div></div> <div>$(!a) (!b)$</div> </div> |
| <div> <div>➤</div> <div>!(a b)</div> </div> | <div> <div></div> <div>等价于</div> </div> | <div> <div></div> <div>$(!a) \&\& (!b)$</div> </div> |
| <div> <div>➤</div> <div>!((a&&b) c)</div> </div> | <div> <div></div> <div>等价于</div> </div> | <div> <div></div> <div>$(!a !b) \&\& !c$</div> </div> |

● 例3.6 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？

● 分析：

不定方程趣味数学问题。

采用列举的算法思想，

对每一种可能的组合进行判断。


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

```
for(cock = 0; cock <= 100; ++cock)  
101 { for(hen = 0; hen <= 100; ++hen)  
101 { 101 for(chicken = 0; chicken <= 100; chicken += 3)  
33     if(cock + hen + chicken == 100 &&  
        cock*5 + hen*3 + chicken/3 == 100 )  
        printf("%d %d %d \n", cock, hen, chicken);
```

循环次数的计算:

外循环执行**101**次, 每次中间循环需要执行**101**次,
中间循环执行一次内层循环需要执行**33**次,
因此内层循环体if语句需要判断约**33**万次。

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

优化方案一

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

循环次数的计算:

外循环执行**21**次, 每次中间循环需要执行**33**次,
中间循环执行一次内层循环最多执行**33**次,
因此内层循环体if语句判断< **2**万次。

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

优化方案二

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

结果:

0	25	75
4	18	78
8	11	81
12	4	84

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)`

条件操作的应用-1

- 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));  
}
```

条件操作的应用-2

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！

条件操作的应用-2

- 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，则它会显示下面的信息并终止程序的运行：

```
Assertion failed: x == 1, 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，也容易出现程序出错行


```
#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);
```

```
    sum = (sum-max-min)/(N-2);
```

```
    printf("%.2f \n", sum);
```

```
    return 0;
```

```
}
```

调试方法一

//调试型输出，软件交付前要去掉

```
#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);
    #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

//调试型输出

```
//#define NDEBUG
#include <assert.h>
```

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

```
#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;
}
```

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
- Debug工具软件
- 断言

● 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

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 $z = a+b+c+d$)

- 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。
- 用来将复杂的表达式分开写

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。

$$0 \rightarrow 1, 1 \rightarrow 0$$
$$0 \rightarrow 1, 1 \rightarrow 0$$
$$0 \rightarrow 1, 1 \rightarrow 0$$
$$0 \rightarrow 1, 1 \rightarrow 0$$

&: 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。

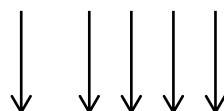
0&0 → 0

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。

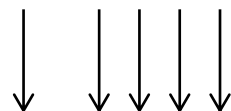
0|0 → 0

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）

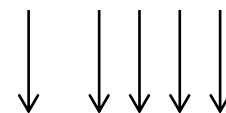
$$0^0 \rightarrow 0$$

$$0^1 \rightarrow 1$$

$$1^0 \rightarrow 1$$

$$1^1 \rightarrow 0$$

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

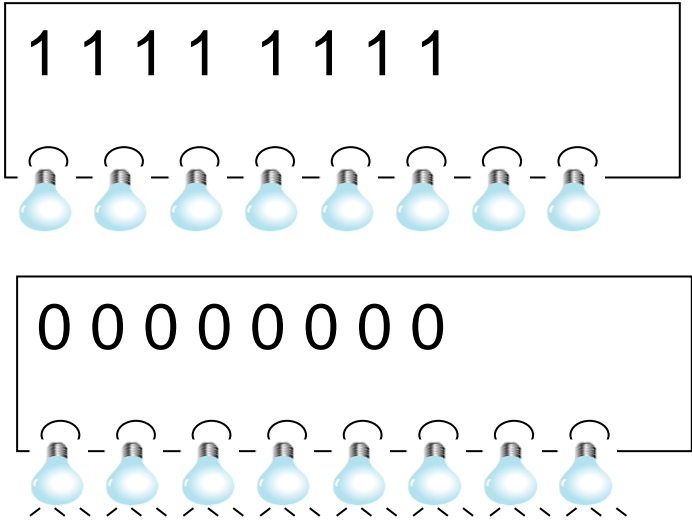
● 注意逻辑位操作与逻辑操作的区别

- 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。



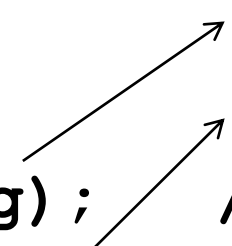
	xxxx	xxxx
&	0110	0010
<hr/>		
	0xx0	00x0

	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 $+$ 、 $-$ 、 $*$ 、 $/$ 、 $\%$ 、 $>>$ 、 $<<$ 、 $\&$ 、 $|$ 、 \wedge

$x \# = y$ 功能上相当于 $x = x \# y$

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

程序中操作的描述

- 基本操作及其应用
- 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简介

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，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
- ...

表达式的有关问题-表达式的值

每个表达式有一个值

- 左值表达式* （一个变量、前缀自增/自减表达式、赋值表达式）
- 操作符的副作用* （=、前缀++/--、后缀++/--）

表达式的值

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
 - ✓ $x=3$ 、 $(x=2)=3$ 、 $++x$ 、 $(x=2)++$ $(++x)++$
 - ✗ $3++$ 、 $(a+b)--$ 、 $++3$ 、 $--(a+b)$ 、 $3=n$ 、 $(!m)=n$ 、 $(x++)++$ 、 $++(x++)$
- ❁ 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， $\underline{3 > 2} > 1$
- 右结合：先让右边的操作符与最近的操作数结合起来， $a = b = 3$

{ 左结合：双目
右结合：{ 单目
三目
赋值

false 0
↑
true 1 > 1
↑

❁ 取正/负与加/减操作的区别

- 功能
- 目
- 优先级
- 结合性

🌈 **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)

🌈 $a/b*c$ 的计算次序为:

$/, *$

(优先级相同, 结合性为左结合)

🌈 $!~a$ 的计算次序为:

$~, !$

(优先级相同, 结合性为右结合)

🌈 $a=b=10$ 的计算次序为:

$b=10, a=b$, 最后 a 、 b 均为10

(优先级相同, 结合性为右结合)

条件操作符的右结合

```
int a = 2;
```

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

注：条件操作符允许嵌套

```
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

🌈 $a = -7\%20 + 3*5 - 4/3$ 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, 加减优先级最低)

- 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-表达式的书写

- 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， $a+b * c$ 与 $a+b*c$ 等价，与 $(a+b) * c$ 不等价。

$(a+b) * c$

$a + b * c$

- 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， $a- --b$ 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。

程序中操作的描述

- 基本操作及其应用
- 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简介

复杂操作的描述方法简介

- ❁ 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
- ❁ 课件第5章~第9章还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...

小结

🌈 程序中的基本操作

- 算术操作符、关系与逻辑操作符、位操作符、赋值操作符、条件操作符等
- 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 \approx 30行
- 继续保持良好的编程习惯
 - 表达式的书写...

Thanks!

