枚举类型

- 程序员用关键词enum构造出来的数据类型，逐个列举出该类型变量所有可能的取值。先构造枚举类型，再定义枚举变量

- 比如，

  enum Color {RED, YELLOW, BLUE};

  Color c1, c2, c3;

  - Color是构造的枚举类型名，花括号里列出了Color类型变量可以取的值
  - c1、c2和c3是枚举变量，这三个变量的取值都只能是RED、YELLOW或BLUE

整形的补码表示

- 补码的简单求法
  - **正数**：同原码
  - **负数**：符号位同原码，其余各位取反，末位加1

真值X：	+1010111	-1010111
$[X]_{原}$(8位)：	01010111	11010111
$[X]_{原}$(16位)：	0000000001010111	1000000001010111
$[X]_{补}$(8位)：	01010111	10101001
$[X]_{补}$(16位)：	0000000001010111	1111111110101001

内容回顾

例：分离出一个整数m的个位、十位和百位

```
…
double x;
void main ()
{
    int m = …;
    x = double(m);
    int a = (x/10-m/10)*10;     // +0.5?
    int b = (x/100-m/100)*10;
    int c = (x/1000-m/1000)*10;
  …
}
```

类型转换！

注意：

C浮点直接转整是截尾取整

（不是四舍五入）

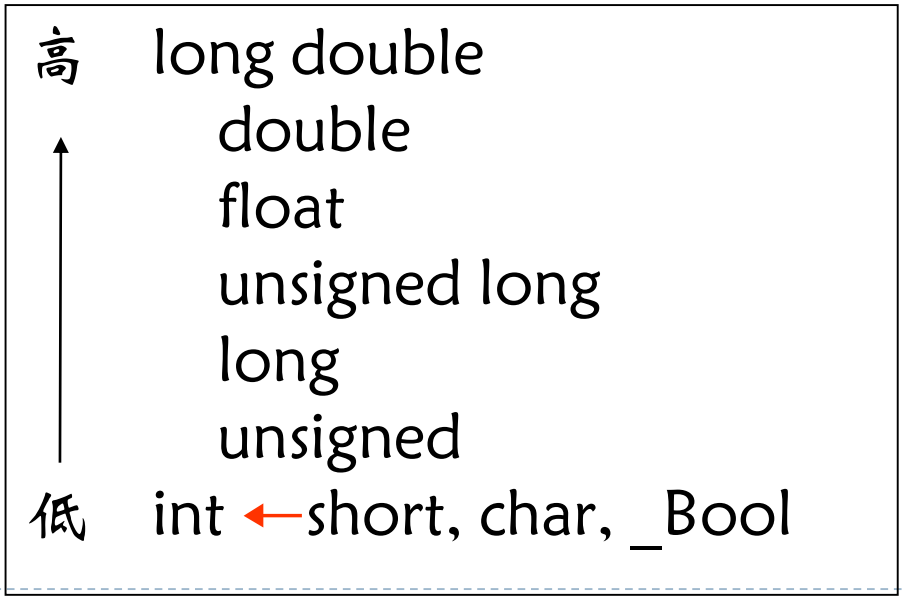
浮点数转整数+0.5

隐式类型转换规则小结

- C程序运行期间，函数和变量的类型以定义的类型为准

实型	去掉小数部分 →	整型
整型	不变 →	实型

- 逻辑操作：
  - 非0数→true，　0→false
- 其他：
  - 精度低→精度高



作 业：

- 输入一个正整数N，判断其为几位数字，按照从高位到低位逐次输出其各位数字。

上机建议:建议大家还是到机房上机!

- 有问题及时讨论、有时候会适当讲解共性的问题
- 经过自己Debug未搞定的问题，老师和助教和大家一起调试
  - 一起调试应该会比直接告诉你错误在那里更有益
  - “直接告诉你错误在那里”不如“告诉你如何寻找错误在那里”
  - “写代码就像一场旅行，要在乎目的地（结果），也重要的是达到“目的”所经历的过程”

上机建议：

- 建议大家还是到机房上机！
  - 有问题及时讨论、有时候会适当讲解共性的问题
  - 经过自己Debug未搞定的问题，老师和助教和大家一起调试
    - 一起调试应该会比直接告诉你错误在那里更有益
    - “直接告诉你错误在那里”不如“告诉你如何寻找错误在那里”
    - “写代码就像一场旅行，要在乎目的地（结果），也重要的是达到“目的”所经历的过程”
- 上机时间未发现的问题，可以微信联系我，我们一起解决
  - 计算机楼1004（My Office）+ 314(TA Lab)

断 点

```
int main()
{
    int a;
    a = 1;
    a ++;
    f(a);
    int b;
    b = 1;
    …
    return
}
```

// F5
// F10 单句（逐句）
// F5 到下一个断点
// F11 进入函数
// Shift+F11 退出函数

5 程序数据描述（III）—操作符与表达式

郭延文

2019级计算机科学与技术系

主要内容

- 操作符
- 表达式

操作符（运算符）

- **操作符**用于描述对数据（**操作数**）的运算，“数据”包括：
  - 常量、变量
  - 函数调用的结果
  - 其它操作符的运算结果

- 例如，在下面的计算式子中，

  a+b-4;

  (-a)*(b+c);

  a/f(10);

  x=a;

  - +、-（减法）、-（取负）、*、/、f（函数调用）以及=（赋值）都是操作符
  - 而a、b、4、c、10、x以及(-a)、(b+c)、f(10)都是操作数

C++操作符的种类

- 算术操作符:+ - * / % ...
- 关系与逻辑操作符:> < && || ...
- 位操作符:<< >> ~ & ^ |
- 赋值操作符:=
- 其它操作符:f ...

算术操作符

- 算术操作符用于实现通常意义下的数值运算。包括：
  - 加“+”、减“-”、乘“*”、除“/”和取余数“%”
  - 取负“-”与取正“+”，例如：-x

除法运算 /

- 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。
  - 需特别注意避免因整除而带来的意想不到的错误!

例：计算圆周率

- 例：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}。

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

```
#include <stdio.h>
#include <math.h>
double myPi();

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

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

```
double myPi()
{
    int sign = 1;
    double item = 1.0, sum = 1.0;
    for(int n = 1; fabs(item) > 1e-6; n++)
    {

        sign = -sign;      //运用了取负操作
        item = sign *1/ (2 * n + 1);   //应改成item = sign * 1.0 / (2 * n + 1);
        sum += item;
    }
    return 4 * sum;
}
```

再看这个例子

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

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) > 1e-6; n++)
    {

        sign = -sign;      //运用了取负操作符
        item = sign * 1 / (2 * n + 1);   //应改成item = sign * 1.0 / (2 * n + 1);
        sum += item;
    }
    return 4 * sum;
}
```

修改为fabs(item) != 1e-6 能行吗？

整数相除特殊应用
例：根据成绩，打印出A(90分以上)，B（80-89），C，D…

- 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:
        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，…
}
```

模运算：求余数 %

- 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

- 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

求余数 %

- 余数的符号由被除数决定置

  例如（VS 2013）：

  -7%5 得到 -2

  7%-5 得到 2

  -7%-5 得到 -2

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。要尽量保证所编写的程序没有歧义！

例：求所有的三位水仙花数

- 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);
}
```

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，操作结果是(自增、自减)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i=i+1;j=i;则i、j的值均变为4
```

先自增；再赋值

  - 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，操作结果是(自增、自减)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++;//相当于n=m;m=m+1;n的值为3， m的值变为4;
```

先赋值再自增；

- 自增/自减操作符可以在循环语句中单独使用，用于循环变量的自增/自减

```
for (int i=0; i < n; i++)
{
        …
}
```

- 或者用于指针类型的操作数(下一章详细介绍)，实现内存的高效访问

关系操作符

- 程序中经常要根据某个条件来决定其后续的动作，这里的条件往往体现为对数据进行比较
- 关系操作符用于对数据进行大小比较，有：

  >，<，>=，<=，==，!=

- 操作数为算术类型和枚举类型，关系操作的结果为bool类型的值：true或false。例如：

  3 > 2的结果为true

  4.3 < 1.2的结果为false

  'A' < 'B'的结果为true

  false < true的结果为true

if（n == 0）V.S. if（0 == n）

- 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该操作符的左边，这样，编译器可以帮助发现这个错误。比如

```
if(n == 0)  //如果误写成if(n = 0)，编译器不会报错，且n变为0
    n++;
else
    n = 1 / n;
```

可以写成：

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

例：求邮包资费

- 假定邮寄包裹的计费标准为：

重量（克）	收费（元）
w < 15	5
15 ≤ w < 30	9
30 ≤ w < 45	12
45 ≤ w < 60	14（每满1000公里加收1元）
w ≥ 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。

主函数

```
#include <stdio.h>
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)  // 数据有效性判断
        {       printf("The input is wrong! Please input again: \n");
                scanf("%d%d", &w, &d);
        }

        printf(  “It costs %d \n RMB.", Charge(w, d) );
        return 0;
}
```

例：求邮包资费 - 再分析

- 假定邮寄包裹的计费标准为：

重量（克）	收费（元）
w < 15	5
15 ≤ w < 30	9
30 ≤ w < 45	12
45 ≤ w < 60	14（每满1000公里加收1元）
w ≥ 60	15（每满1000公里加收2元）

用if else

用距离整除1000

- 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。

计算资费函数 - if else 结构

```
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;
}
```

例：求邮包资费 – 再分析

- 假定邮寄包裹的计费标准为：

重量（克）	收费（元）
w < 15	5
15 ≤ w < 30	9
30 ≤ w < 45	12
45 ≤ w < 60	14（每满1000公里加收1元附加费）
w ≥ 60	15（每满1000公里加收2元附加费）

- 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 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;
}
```

逻辑操作符

- 逻辑操作符实现逻辑运算，用于复杂条件的表示。包括：
  - !（逻辑非）
  - &&（逻辑与）
  - ||（逻辑或）
- 操作数为bool类型，例如：
  - !(a > b)
  - (age < 10) && (weight > 30)
  - (ch < '0') || (ch > '9')
- 结果为bool类型

!true -> false
!false -> true

false && false -> false
false && true -> false
true && false -> false
true && true -> true

false || false -> false
false || true -> true
true || false -> true
true || true -> true

- !(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

- &&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：
  - !(a&&b)　　等价于　　(!a)||(!b)
  - !(a||b)　　等价于　　(!a)&&(!b)
  - !((a&&b)||c)　　等价于　　(!a||!b)&&!c

例：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：

  采用列举的算法思想，
  对每一种可能的组合进行判断
  是否存在多种组合？

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");

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

优化方案一

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");

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

Tips: 当多个表达式逻辑与的时候，
容易做出判断的放前面最好（做pruning）！具体情况具体分析，
顺序是有所谓的；同理：if、else if、else if … else

优化方案二

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

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("%3d %3d %3d \n", cock, hen, chicken);
        }
```

结果：
```
 0  25  75
 4  18  78
 8  11  81
12   4  84
```

逻辑与的两个条件，顺序有关系吗？以上两个调换下顺序更好！

位操作

- 将整型操作数看作二进制位序列进行操作,包括
  - 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

逻辑位操作

- 注意逻辑位操作与逻辑操作的区别
  - 逻辑操作（!、&&、||）结果的含义：表示是否成立
  - 逻辑位操作结果的含义：是一个二进制位序列、

  是一个数，对整数的二进制表示进行操作

按位取反 ~

- 用来把一个二进制位序列中的每一位由0变1、由1变0

~0 → 1，~1 → 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。
- 特点：

一个二进制位与0异或，保持原值不变；

与1异或，结果和原值相反；

与本身异或，为0

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）

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

  ~：所有位翻转

  容易借助&|^实现：

  - &：按位清零
  - |：按位置1
  - ^：特定位的翻转

xxxx xxxx…

& 0110 0010… (0x62…)

0xx0 00x0

xxxx xxxx…

^ 0110 0010… (0x62…)

xyyx xxyx

xxxx xxxx…

| 0110 0010… (0x62…)

x11x xx1x

例：借助 & 保留某个数的某一位

```
#define KEY  8
 int flag, temp;
 scanf("%d", &flag);
 temp = flag & KEY;     //保留flag的第4位（0…0 1000）
 if(temp == KEY)
        printf("The 4th of flag is 1");
 else
        printf("The 4th of flag is 0");
```

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，移动的次数由右边的整型操作数决定

- 包括
  - <<（左移）
  - >>（右移）

左 移

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

例：<< 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

- 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

- 作用
  - 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

例：>> 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)

算数右移 V.S. 逻辑右移

对10101010

- 逻辑左移一位：01010100[0]
- 算数左移一位：01010100[0]
- 逻辑右移一位：[0]10101010
- 算数右移一位：[1]10101010

[ ]表示添加的数字

赋值操作

- 通过赋值操作改编变量的值，例如：

  a = x + y * z;

- 简单赋值操作符

  a = b;

- 复合赋值操作符

  +=，-=，*=，/=，%=，&=，|=，^=，<<=，>>=

  - a#= b 功能上等价于：a = a# (b)

其它操作符

- 条件操作符（?:）

  d1?d2:d3

  - 如果d1的值为true或非零，则运算结果为d2，否则为d3。例如：

    ```
    c = (a>b)?a:b //a和b中的大者赋值给c
    result = a>b ? (a>c ? a : c) : (b>c ? b : c)
    ```

  - 遵循短路求值：

    ```
    int a = 1, b = 2;
    int c = ( a<b ? (a=3) : (b=4) ) ; // 运行后a、c为3，b仍为2
    ```

- 逗号操作符

  ```
  d1, d2, d3,...
  ```

  - 从左至右依次进行各运算，操作结果为最后一个运算的结果

    ```
    x = a+b, y = c+d, z = x+y;
    ```

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 — 双目
  - 取正/取负操作符 — 单目
  - 自增/自减操作符 — 单目
  - 关系操作符 — 双目
  - 逻辑操作符 — 双目/单目
  - 位操作符 — 双目/单目
  - 赋值操作符 — 双目
  - 条件操作符 — 三目

连接两个操作数

连接一个操作数

连接三个操作数

表达式的有关问题

- 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
  - …
- 每个表达式有一个值
- 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
- 良好的表达式书写习惯有助于表达式的正确求解

表达式的值

- 每个表达式有一个值
  - 常量表达式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存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

思考1：

- 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 ;这个表达式的值为：

18

思考2：

c=a--?++a:--a;

相当于：

```
if(a--) // 判断a是否为零，判断后将a自减1；
{
    c=++a; // a自加1后赋值
}
else
{
    c=--a; // a自减1后赋值
}
```

操作符的优先级

- 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

C 语言运算符优先级

一共有十五个优先级：

1 () [] . ->

2 ！～ -（负号） ++ -- &（取变量地址）* （type）(强制类型) sizeof // 单目

3 * / % // 双目 算数

4 + -

5 >> << // 双目 移位

6 > >= < <= // 双目 关系

7 == !=

8 & // 双目 逻辑位

9 ^

10 |

11 && // 双目 逻辑

12 ||

13 ?: // 三目

14 = += -= *= /= %= |= ^= &= >>= <<=

15 ,

C 语言运算符优先级 背诵口诀☺

- 括号成员第一； //括号运算符[]() 成员运算符. ->
- 全体单目第二； //所有的单目运算符比如++、 --、 +(正)、 -(负) 、指针运算*、&乘除余三,加减四； //这个"余"是指取余运算即%
- 移位五，关系六； //移位运算符：<< >> ，关系：> < >= <= 等
- 等于(与)不等排第七; //即== 和!=
- 位与异或和位或； //这几个都是位运算:位与(&)异或(^)位或(|)
- "三分天下"八九十;
- 逻辑或跟与； //逻辑运算符:|| 和 &&
- 十二和十一； //注意顺序:优先级(||) 底于 优先级(&&)
- 条件高于赋值， //三目运算符优先级排到13 位只比赋值运算符和","高
- 逗号运算级最低! //逗号运算符优先级最低

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

一个变量

一个赋值表达式

一个前缀自增/自减操作表达式

- 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

x=3、(x=2)=3、++x、(x=2)++ (++x)++ √

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

操作符的结合性

- 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，

    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

具体结合性参加课本2.5.2的表格！

表达式的操作顺序

- 一个表达式可以包含多个操作，先执行哪一个操作呢？
- C语言有以下规则：

  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，结合性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！

- 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)

回顾 思考1：

- 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 ;这个表达式的值为：

18

C 语言运算符优先级

一共有十五个优先级：

1 () [] . ->

2 ！～ -（负号） ++ -- &（取变量地址）* （type）(强制类型) sizeof // 单目

3 * / % // 双目 算数

4 + -

5 >> << // 双目 移位

6 > >= < <= // 双目 关系

7 == !=

8 & // 双目 逻辑位

9 ^

10 |

11 && // 双目 逻辑

12 ||

13 ?: // 三目

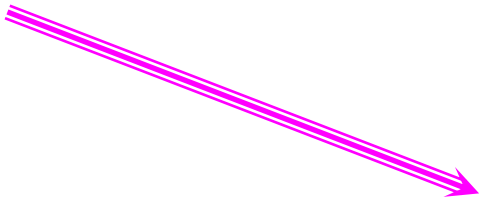
14 = += -= *= /= %= |= ^= &= >>= <<=

15 ,

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，表达式右侧是一个操作。

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（&&、||、?: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）

比如，对于表达式(a+b)*(c-d)，C语言没有规定+和-的操作顺序。

- 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，建议把带有副作用的操作符（++/--、=）作为单独的操作或者放入括号，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 （VS 2013）。

- 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 = 1;
        else z = 0;
        return z;
}
```

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

不同开发环境执行结果可能不同（0或1）

- C语言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。

表达式的书写

- 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）来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

建议！！！

- 以上要掌握知识点，但写程序不建议写“难以理解”的表达式！

例如：

```
int a  = 12;
a = a += a -= a*a;
```

建议写为：

```
a -= a*a;
a += a;
```

建 议！！！

以上要掌握知识点，但写程序不建议写“难以理解”的表达式！

c=a--?++a:--a; 建议写为：

```
if(a--)
{
    c=++a;
}
else
{
    c=--a;
}
```

建 议！！！

以上要掌握知识点，但写程序不建议写“难以理解”的表达式！

```
int x = 1;
 int tmp = (x + 1) * (x = 10); 建议写为：

…
x= 10;
int tmp = (x+1)*x;
```

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
  - 掌握恰当选用基本操作符实现计算任务的方法
  - 会通过恰当的书写方式避免程序存在歧义
  - 保持良好的编程习惯

讨 论

- 交换两个整型变量的值，不引入第三个变量

  - 方法一

```
int a = 5, b = 9;
a = a + b;
b = a - b;
a = a - b;
printf("%d, %d", a, b);
```

如果a、b不是5、9，要注意溢出问题，即在a、b之和（差）溢出时，该方法不能奏效。

方法二：采用位预算

```
int a = 5, b = 9;
a = a ^ b;
b = a ^ b;
a = a ^ b;
printf("%d, %d", a, b);
```

举例：

```
    a   011
(^) b   100
    a   111(a^b的结果赋值给a，a值为7)
(^) b   100
    b   011(b^a的结果赋值给b，b值为3)
(^) a   111
    a   100(a^b的结果赋值给a，a值为4)
```

原理：

a = a^b; b = b^a; 相当于：
b = (a^b)^b = a^(b^b)；
-> b = a^0；
-> b=a;

a = a^b = (a^b)^a = b;

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。
- 特点：

一个二进制位与0异或，保持原值不变；

与1异或，结果和原值相反；

与本身异或，为0

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）

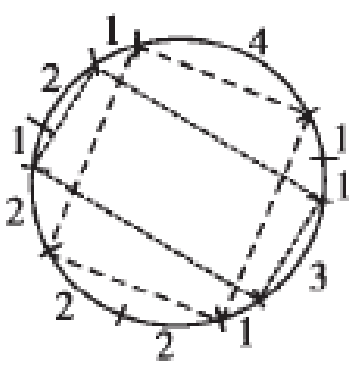
的结果为：

3（0000 0000 0000 0000 0000 0000 0000 0011）

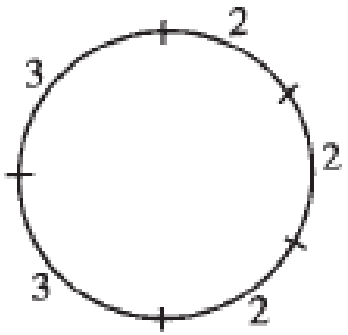
Q & A

上机题目：寻找园内接矩形

- 在圆周上有若干点，已经知道这些点与点之间的弧长，并且依圆弧顺序排列，请写一个程序，找出这些点中有没有4个可以围成长方形，程序返回是否有长方形即可。为了方便计算，弧长均为正整数。



a）



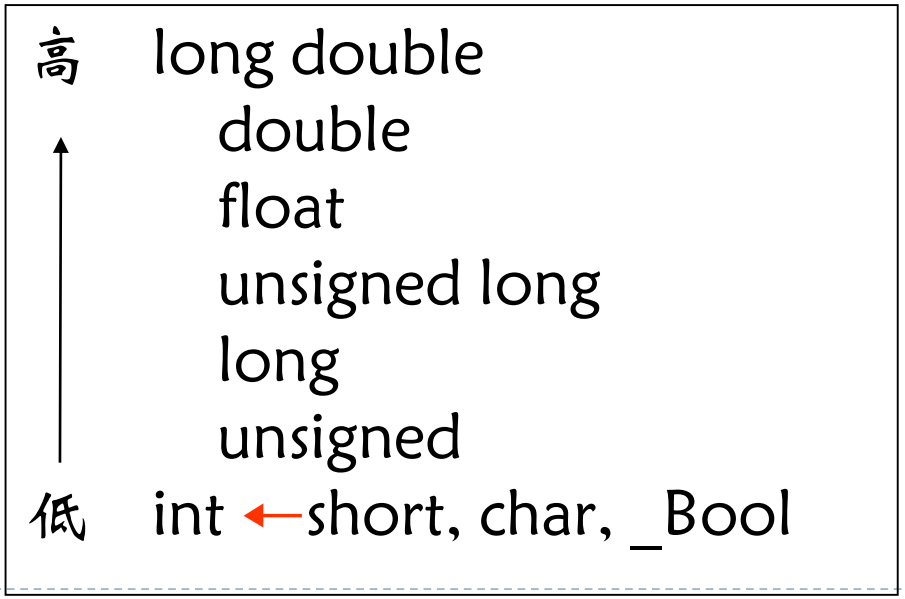
b）

隐式类型转换

- C程序运行期间，函数和变量的类型以定义的类型为准

实型	去掉小数部分 →	整型
整型	不变 →	实型

- 逻辑操作：
  - 非0数→true，　0→false
- 其他：
  - 精度低→精度高



例：生成一定范围内的伪随机数

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define random(x) (rand()%x)

void main()
{
    srand((int)time(0));    // srand(time(NULL));
    for(int x=0;x<10;x++)
        printf("%d/n",random(100));
}
```

例：求所有的三位水仙花数

- 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);
}
```

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，操作结果是(自增、自减)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i=i+1;j=i;则i、j的值均变为4
```

先自增；再赋值

  - 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，操作结果是(自增、自减)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++;//相当于n=m;m=m+1;n的值为3， m的值变为4;
```

先赋值再自增；

逻辑操作符

- 逻辑操作符实现逻辑运算，用于复杂条件的表示。包括：
  - !（逻辑非）
  - &&（逻辑与）
  - ||（逻辑或）
- 操作数为bool类型，例如：
  - !(a > b)
  - (age < 10) && (weight > 30)
  - (ch < '0') || (ch > '9')
- 结果为bool类型

!true -> false
!false -> true

false && false -> false
false && true -> false
true && false -> false
true && true -> true

false || false -> false
false || true -> true
true || false -> true
true || true -> true

位操作 ~ & ^ | << >>

- 位操作(按位取反、与、或、异或)的操作数是整数的二进制位序列表示
- 位操作速度快，节省存储空间
- 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)

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，移动的次数由右边的整型操作数决定

- 包括
  - <<（左移）
  - >>（右移）

左移举例

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）

右移举例（分逻辑和算数右移）

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）

算数右移 V.S. 逻辑右移

对10101010

- 逻辑左移一位：01010100[0]
- 算数左移一位：01010100[0]
- 逻辑右移一位：[0]10101010
- 算数右移一位：[1]10101010

[ ]表示添加的数字