

# C语言程序设计

The C Programming Language



## 第2章 C语言的基本元素

华中科技大学武汉光电国家研究中心

李春花



# 本章目标

---

1. 如何准确地书写各种常量?

2. 如何定义(说明)各种类型的变量?

3. 如何准确地利用各种运算符书写表达式?



# 主要内容

---

- 词法元素
- 标识符和关键字
- 基本数据类型
- 常量与变量
- 运算符和表达式
- 位运算
- 类型转换
- 枚举类型



## 2.1 字符集及词法元素

- 词法元素称为**记号(token)**
- **记号**是程序中**具有语义**的最基本组成单元
- 记号共**分5类**：**标识符、关键字、常量、运算符和标点符号**
- 编译器从左至右收集字符，总是尽量**建立最长的记号**，即使结果并不构成有效的**C**语言程序。相邻记号可以用空白符或注释语句分开。

# 词法分析举例

5种记号：标识符、关键字、常量、运算符和标点符号

■ **sum=x+y**


标识符、运算符

被分解成**sum**、**=**、**x**、**+**和**y** 共5个记号。

■ **int a, b=10;**

关键字、标识符、运算符、常量、标点符号

被分解成**int**、**a**、**,**、**b**、**=**、**10**和**;** 共7个记号

■ **x+++++y**  **x++ ++ +y**      **x++ + ++y**

无效      有效

被分解成**x**、**++**、**++**、**+**、**y** 共5个记号

## 2.2 关键字和标识符

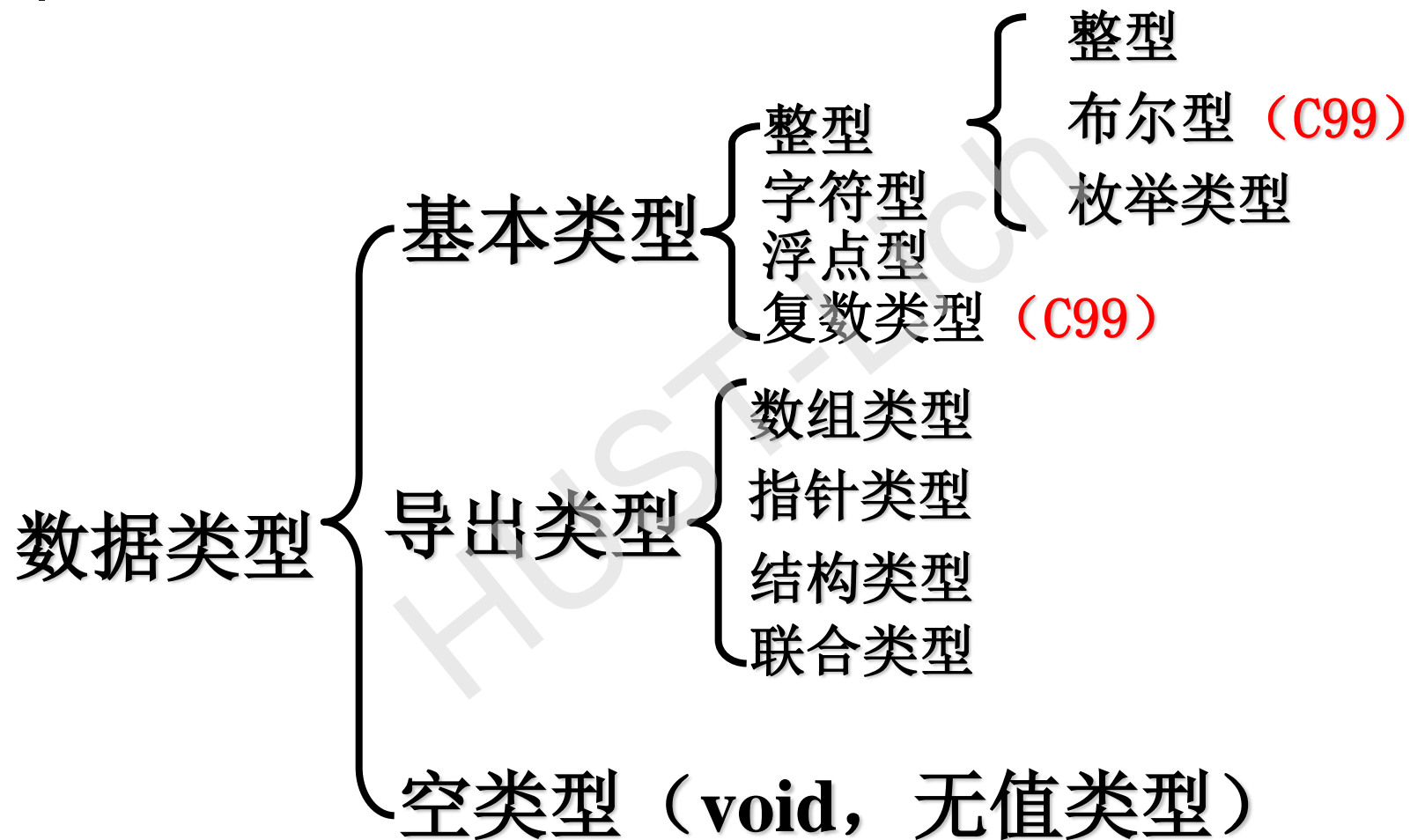
- **关键字**：是被系统赋予特定含义并有专门用途的字，不能作为普通标识符。(表2-1)
  - int、void、return、while、if
  - 小写
- **标识符**：是用来标识用户定义的常量、变量、数据类型和函数等名字的符号。**标识符不能与关键字同名**
  - **命名规则**：由字母(A~Z, a~z)、下划线\_和数字(0~9)组成，且第一个字符必须是字母或下划线\_。字母区分大小写
  - K, \_id, month, time1 ✓
  - 20\_sum, not#me, void ✗



# 注意

- 大小写字母表示不同意义。
- 不能使用类似 **int** 和 **void** 这样的**C**关键字为自己的对象命名，也要避免使用**C**程序库中函数和常量的名称，例如 **scanf** 。
- 良好的编程风格是选择有助于记忆且**有一定含义的标识符**，这样可增强程序的可读性和程序的文档性。

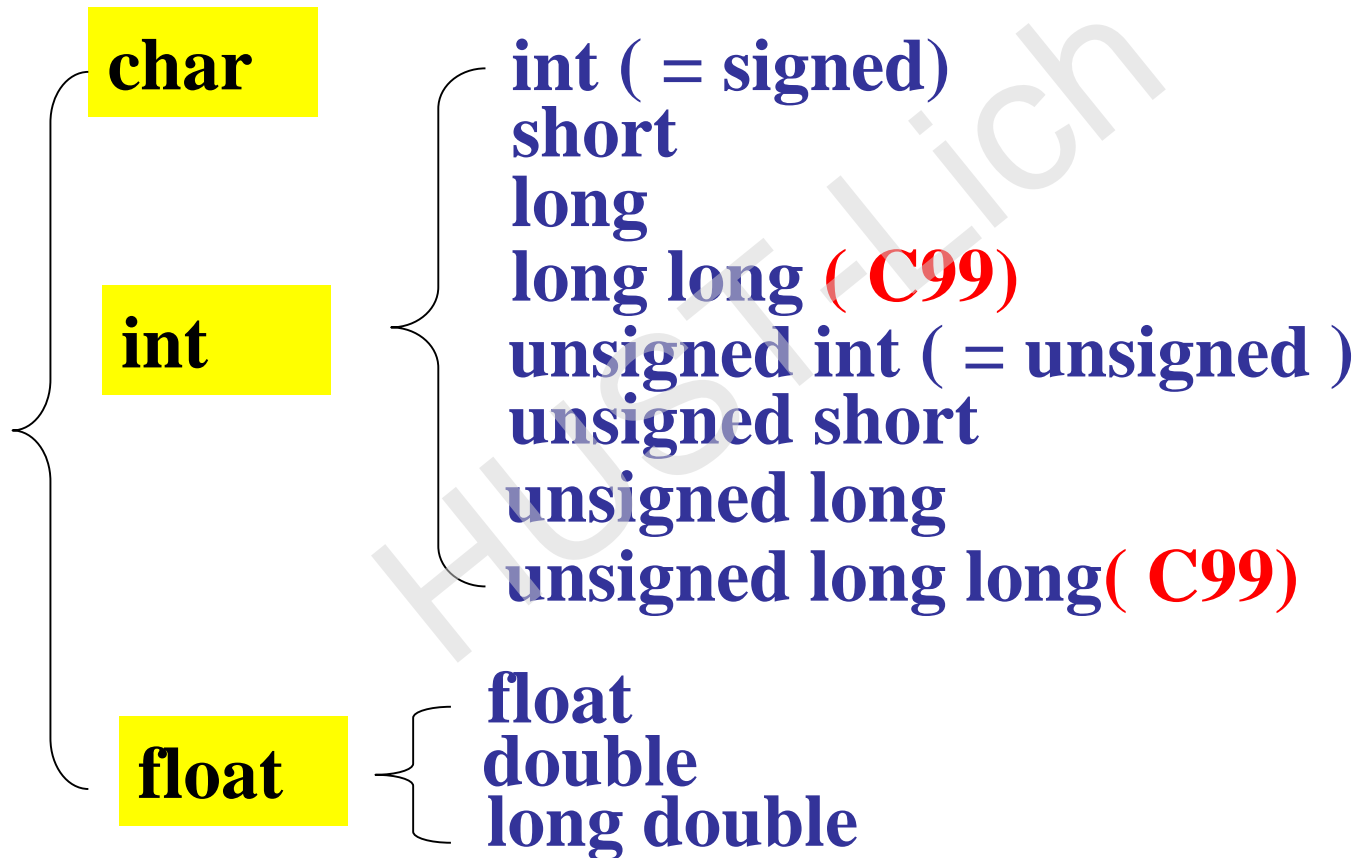
## 2.3 基本数据类型





# 基本类型

## ■ 含字符型、整型、浮点型 (表2-2)



# char类型

- **char**的存储长度是一字节
- 多数系统中**char**与**signed char**同  
取值范围 (-128~127) 即:  $-2^7 \sim 2^7 - 1$
- 字符数据以**ASCII**码存储在内存中(附录A)
- 在不要求大整数的情况下, 可用字符型代替整型。
  - 例如: 字符A的ASCII码为65或0x41, 在内存中表示为:

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

# 整型类型

- **int**型长度一般为一个机器字长.  $-2^{15} \sim 2^{15}-1$
- 假设字长为**2B**, **int**取值范围为-32768~32767,  
**unsigned**取值范围为0 ~ 65535
- 下面的代码是否正确  $0 \sim 2^{16}-1$

```
#define BIG 30000
```

```
int main(void)
```

```
{ short x,y,z;
```

```
    x=y=BIG;
```

```
    z=x+y;    /* 短整数溢出 */
```

```
    .....
```

```
}
```

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

.....

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

.....

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

有符号数 无符号数

32767

32767

$2^{15}-1$

32766

32766

.....

1

1

0

0

-1(补码)

65535

$2^{16}-1$

-2

65534

.....

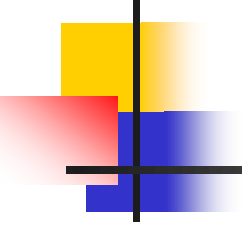
.....

-32767

32769

-32768

32768

- 
- 程序员必须时刻保证整数表达式的值在合理范围内。
  - 引入**short**和**long**的目的是为了提供各种满足实际要求的不同长度的整数。
  - **int**通常反映特定机器的自然大小，**short**一般为**2B**，**long**一般为**4B**。因此，当关心存储时，用**short**；当需要较大的整数值时，用**long**。

# 浮点类型

float  
double  
long double

IEEE754标准

$S$  符号位,  $M$  有效数字,  $E$  指数位

## ■ 浮点数的二进制表示法:

$$V = (-1)^S \times M \times 2^E$$

$$1 \leq M < 2$$

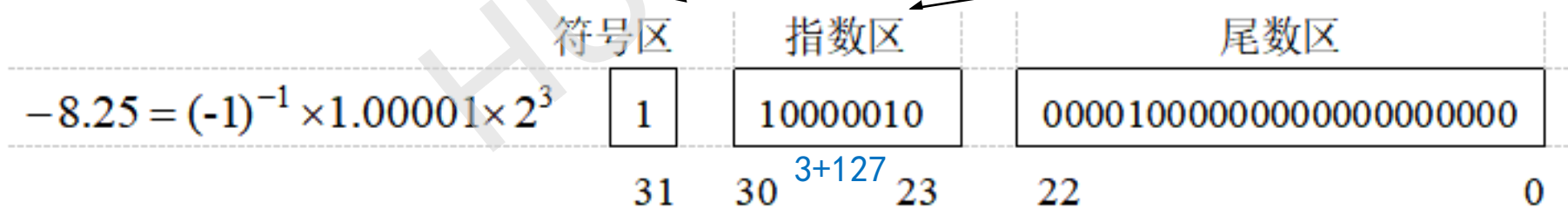
8.25的二进制表示: 1000.01

$$1000.01 = (-1)^0 \times 1.00001 \times 2^3$$

float 的精度约7位;  
double的精度约15位

数符 1b  
尾数 23b  
指数 8b  
阶码

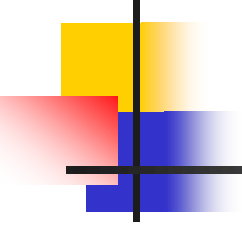
float



指数区用移码表示  $E+127$  尾数区用原码表示且省略1

十进制的整数转二进制: 除2取余, 商0结束, 逆序排列

十进制的小数转二进制: 乘2取整, 小数0结束, 顺序排列<sup>14</sup>

- 
- 尾数所占的位数决定值的精度，指数所占的位数决定值的范围。
  - **float** 占4B，其中符号1b，指数8b，尾数23b，其精度大约为7位，范围约 $10^{-38} \sim 10^{+38}$ 。
  - **double** 占8B，其中符号1b，指数11b，尾数52b，其精度大约为15位，范围约为 $10^{-308} \sim 10^{+308}$ 。
  - 很多编译器将**long double**处理为**double**，在某些系统中，它占用10或12B。很少被使用。

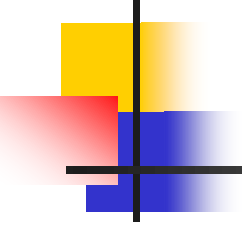
# 浮点数的可表示误差

■  $5.1 = 101.\underline{0011}0011\underline{0011}...$   
 $= 1.010\ 0011\ 0011\ 0011... \times 2^2$

0	10000001	01000110011001100110011
---	----------	-------------------------

在值的可表示范围内，整数一定是精确表示，  
而浮点数的表示可能只是近似的



- 
- 浮点数的表示可能只是近似的。其值与表示法之间的差称为“可表示误差”。
  - 计算也可能造成可表示误差。不能使用 `==` 和 `!=` 运算符比较浮点数据。
  - 可以用两个数值之差同一个小正数 `epsilon` 比较的方法解决这个问题。如：

```
if(abs(a-b)<e-6) printf("a is equal to b\n");
```



# 布尔类型\_Bool

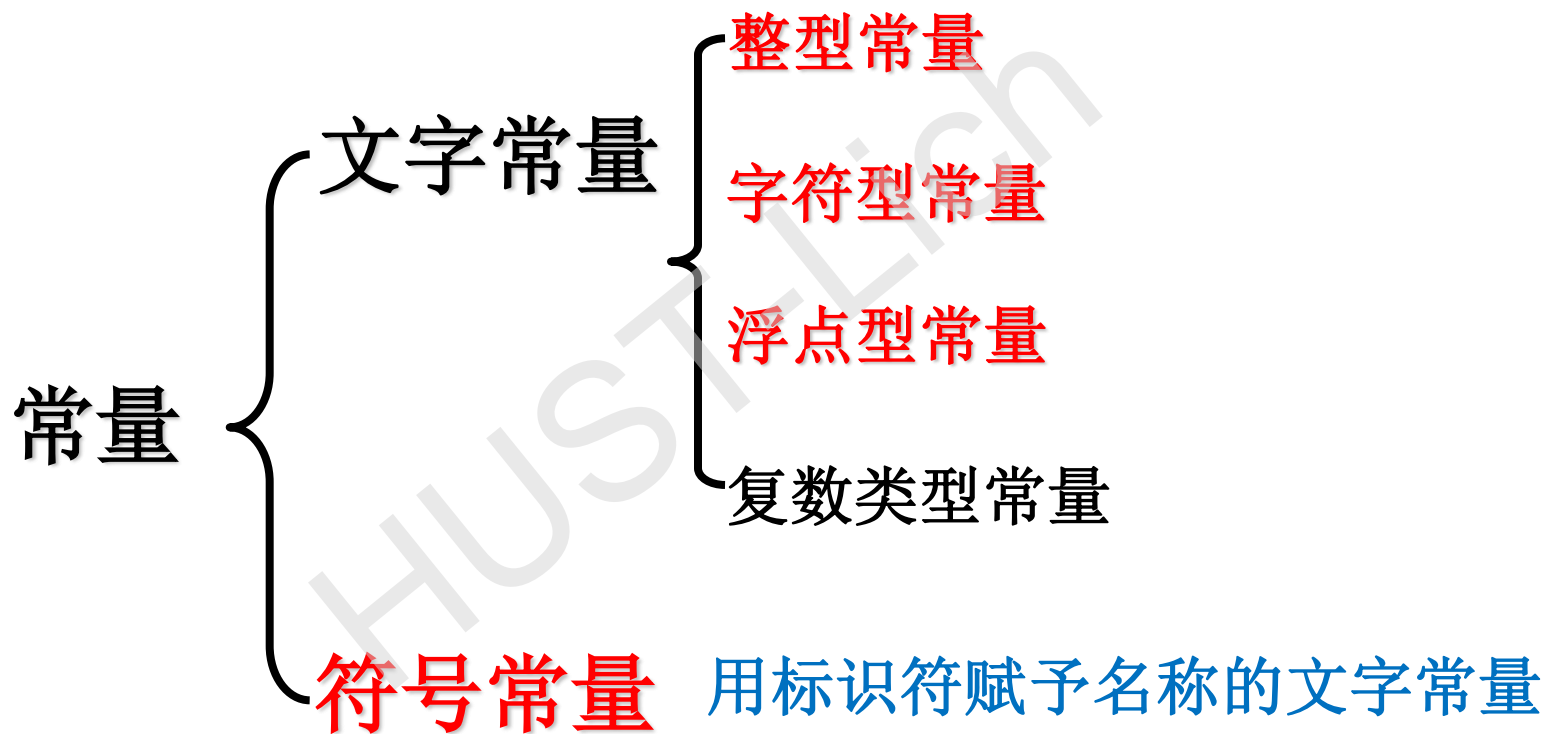
- C语言中可以用任何整数类型表示布尔值，**0**表示假，所有非**0**表示真。
- C99标准新增\_**Bool**类型，**长度为1**，值为**0**和**1**（表示“**false**”与“**true**”），使用\_**Bool**类型更清晰。
- 为了让C和C++兼容，C99增加了头文件**stdbool.h**，里面定义宏名**bool**为\_**Bool**的同义词，定义**false**与**true**分别为**0**和**1**。 `bool flag=true;`
- 将任意非零值赋给\_**Bool**类型变量，都会先转换为**1**，表示真。



## 2.4 常量与变量

- ❖ **常量**：在程序运行过程中，其值一直保持不变的量
  - 常量也区分不同的类型：  
30, 40 为整型, 30.0, 40.0为实型, 'a', 'b'为字符型
  - 编辑器只是根据其表面形式来判断其类型。
- ❖ **变量**：在程序运行过程中，其值可以改变的量
  - 变量在程序的执行中能够赋值，发生变化。
  - 变量有一个名字，**并在使用之前要说明其类型**，**一经说明，就在内存中占据与其类型相应长度的存储单元。**

## 2.4 常量与变量



```
#define PI 3.1415
```

```
#define HUST "Huazhong University of ..."
```

## ❖ 有三种表示方式（通过前缀字符区分）

- **十进制（无前缀）**：例如 56、-100、2004； **023、34A** ×
- **八进制**：以**0**开头，数码取值为0～7。如：017(十进制为15)、0101(十进制为65)、0177777(十进制为65535) **17、084** ×
- **十六进制**：以**0x或0X**开头，其数码取值为0～9，**A～F或a～f**。如：0X2A(十进制为42)、0xA0 (十进制为160)、0xFFFF (十进制为65535)； **5A、0X3H** × 如，31可写成037，也可写成0x1f或0X1F

用后缀 **“L” 或 “l”** 来表示长整型数。如：

158L (十进制为158), 012L (十进制为10) ；

用后缀 **“U” 或 “u”** 来表示无符号整型数，如158U，158UL。

# 整型常量的后缀

用以指定其类型

- 字母u或U表示unsigned，如158u, 158UL
- 字母l或L表示long，如12L, 012L
- 字母ul或UL表示unsigned long
- 字母ll或LL表示long long (C99)
- 字母ull或ULL表示unsigned long long (C99)
- 无后缀时，一般表示int
- 当常量值超出指定类型的范围时，其实际类型取决于数值大小、前缀等，确定类型的规则很复杂，在标准化前的C语言、C89和C99中各不相同

# 浮点型常量

## ❖ 浮点常量

➤ 有两种表示方式:

(可以小数点开头, 也可以小数点结尾)

1) 十进制形式: 23.0 24.5 3.56789 .7 9.

2) 指数形式 (科学计数法): 将指数部分跟在尾数部分后面

$e(E) \pm n$ , 代表  $10 \pm n$   $45e-3 = 45 \times 10^{-3}$ ,  $.15e5 = 0.15 \times 10^5$

e前有数字, 后面必须是整数 1., 1.23, .23e+12, 25E5, +1.5e+31L

345, -E7, 100.-E3, 2.7E, 10e1.5



两条规则:

- 1) 一个浮点数可以无整数部分或小数部分, 但不能二者全无; **E+10** ×
- 2) 一个浮点数可以无小数点或指数部分, 但不能二者全无。



# 浮点型常量的后缀

可以使用**后缀**来指定其类型

- 无后缀: **double**    **12. 10E10**
- 后缀**f或F**: **float**    **.5f 1.2eF** ×
- 后缀**l或L**: **long double**    **1E-123L**



## ❖ 字符常量

(1) 用单引号包含的一个字符是字符常量

‘a’ ‘A’ ‘1’

(2) 只能包含一个字符

‘abc’ “a”



# 字符型数据(char)

- 字符型数据实际上是作为**整型数据**在内存中存储的。
- 计算机是以字符编码的形式处理字符的，因此，我们在计算机内部是以**ASCII码**的形式表示所有字符的。所以7位二进制数即可表示出一个字符，**我们用一个字节的容量（8位）存储一个字符。**
- 例如：字符A的ASCII码为0x41或65，在内存中表示为：

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

在程序中表示为：‘ ’ 内括起来的字符表示该字符的**ASCII码**

`char grade ;`//定义一个字符型的变量空间(1个字节)

`grade= ‘A’ ;` //必须用 ‘ ’ 表示，否则易与标识符混同

➤ 进一步，由于在内存中的形式与整型数据相同，所以，可以直接用其整型值给变量赋值。

```
char grade;  
grade=65;
```

以下的赋值形式均是等同的。

```
grade='A';      grade=65 ;      grade=0x41;      grade=0101;
```

```
#include<stdio.h>  
int main()  
{  
    char a,b;  
    a= 'A' ; //输入ASCII码  
    b=65; //输入十进制数  
    printf("a=%c\n ", a);  
    printf("b=%c\n ", b);  
    return 0;  
}
```

输出：

a=A

b=A

即在内存中的表示均是相同的

0	1	0	0	0	0	0	1
---	---	---	---	---	---	---	---

## ❖ 非打印字符的表示

➤ 有些ASCII的字符代表某些操作，不能打印出来，如回车、退格等，可用两种方式表示这些字符。

1) 用ASCII码的形式      `char re=13;`    回车

2) 用转义字符    `char newln= '\n';` 回车+换行

# 转义序列

■ 以 \ 开头的特殊字符称为转义序列,有两种形式:

➤ 一种是“**字符转义序列**”，反斜线后面跟一个图形符号，用于表示字符集中的非图形符号和一些特殊的图形字符。

'\n' 换行

'\t' 水平制表符

'\' 反斜杠

'\'' 单引号

'\"' 双引号

'\0' 空字符

要输出 c:\tc\tc，则表示为：  
"c:\\tc\\tc"

虽然包含2个或多个字符，但它只代表一个字符。编译系统在见到字符“\”时，会接着找它后面的字符，把它处理成一个字符，在内存中只占一个字节。

➤ 另一种是“**数字转义序列**”

'\101'    '\x41'    '\x61'    '\141'

# 数字转义序列

## ➤ 数字转义序列

- 即 `\ooo` (1~3个八进制数字)

`\xhh` (1~2个十六进制数字)

- 例如,

`'A'`、`'\101'` 和 `'\x41'`

字符A

`'\t'`、`'\11'`、`'\011'`、`'\x9'`和`'\x09'`

水平制表符

## 表2-3 常见的转义字符

转义字符	含 义	ASCII代码
<code>\a</code>	响铃	7
<code>\n</code>	换行，将当前位置移到下一行开头	10
<code>\t</code>	水平制表（跳到下一个tab位置）	9
<code>\b</code>	退格，将当前位置移到前一行	8
<code>\r</code>	回车，将当前位置移到本行开头	13
<code>\f</code>	换页，将当前位置移到下页开头	12
<code>\v</code>	竖向跳格	8
<code>\\</code>	反斜杠字符 “\”	92
<code>\'</code>	单引号（撇号）字符	39
<code>\"</code>	双引号字符	34
<code>\0</code>	空字符	0
<code>\ooo</code>	1到3位8进制数所代表的字符	
<code>\xhh</code>	1到2位16进制数所代表的字符	

表2-3

## 字符常量--转义字符表示

字符类型	字符表示	字符含义	ASCII码值	“\ooo”表示	“\xhh”表示
字母	‘a’	字母 (a)	97	\141	\x61
数字	‘0’	数字 (0)	48	\060	\x30
特殊字符	‘\a’	鸣铃	7	\007	\x07
	‘\’	单引号符 (’)	96	\140	\x60
	‘\”	双引号符 (”)	34	\042	\x22
	‘\\’	反斜线符 (\)	92	\134	\x5C
空格符	‘\n’	回车换行	10	\012	\x0A
	‘\f’	走纸换页	12	\014	\x0C
不能显示的 字符	‘\0’	空字符	0	\000	\x00
	‘\b’	退格	8	\010	\x08
	‘\r’	回车	13	\015	\x0D

八进制数





# 转义序列的应用

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

```
int main(void)
```

```
{
```

```
    printf("\n\n causes \na line feed to occur");
```

```
    printf("\n\\\"causes a double quote (\") to be printed");
```

```
    printf("\n\\a causes the bell,or beep, to sound\a");
```

```
    printf("\n\t can be used to align some numbers to tab");
```

```
    printf("columns \n\t1\t2\t3\n\t4\t5\t6");
```

```
    return 0;
```

```
}
```

```
\n causes
a line feed to occur
\"causes a double quote (") to be printed
\a causes the bell,or beep, to sound
\t can be used to align some numbers to tabcolumns
      1      2      3
      4      5      6
```

## 2.4.4 字符串常量

- 写成用一对**双引号**括住**0**至多个字符的形式。

`"string\n"` /\* 包含7个字符的字符串 \*/

`""` /\* 包含**0**个字符的空字符串\*/

- 字符串中的单引号可以用图形符号表示，但**双引号**和**反斜线**必须用**转义序列**表示。例如：

`"3'40\""` /\* 表示5个字符的字符串：3'40" \*/

`"c:\tc"` /\* 表示4个字符的字符串 \*/

`"c:\\tc"` /\* 表示5个字符的字符串 \*/

# 字符串常量

- 用双引号表示，在内存中顺序存放，以'\0'结束。


如: "CHINA" /\* 包含5个字符的字符串 \*/

C	H	I	N	A	\0
---	---	---	---	---	----

存储时，系统自动在后面补上\0

- 实际上内存是对应字符的ASCII码形式

C	H	I	N	A	
0x43	0x48	0x49	0x55	0x41	\0
01000011	01001000	01001001	01010101	01000001	00000000

- 1) 字符常量由单引号括起来，字符串常量由双引号括起来。
- 2) 字符常量只能是单个字符，字符串常量则可以含一个或多个字符。
- 3) 可以把一个字符常量赋予一个字符变量，但不能把一个字符串常量赋予一个字符变量。
- 4)  字符常量占1B内存空间。字符串常量占的内存空间比字符串实际长度大1。增加的一个字节中存放字符串结束标志' \0' (ASCII码为0)。



# ‘a’与“a”的区别

‘a’：字符常量，占1 B内存空间

“a”：字符串常量，占2B内存空间

a	\0
---	----

- 存储时，系统自动在后面补上\0（空字符，ASCII值为0，作为字符串结束标志）
- 字符串的存储长度比字符串的实际长度大1

有两种方法：

① **行连接**：在前一行的末尾输入续行符（\）再换行。

```
printf("Hello,\n  
how are you");    /* 换行后应紧靠行首 */
```

② **字符串连接**：将字符串分段，分段后的每个字符串用双引号括起来。

```
printf("Hello, "  
"how are you");    /* 换行后不必紧靠行首*/
```

# 求字符串的长度

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

```
int main( )
```

```
{
```

```
    char s[20]="world"; /* 用于保存串 */
```

```
    int counter; /* 计数器变量，保存串的长度 */
```

```
    counter =0; /* 计数器变量清零 */
```

```
    while(s[counter] !='\0')
```

```
        ++counter;
```

```
    printf( "%d", counter);
```

```
    return 0;
```

```
}
```

s[0]	w
s[1]	o
s[2]	r
	l
	d
	\0
	...
s[19]	

# 用函数实现求字符串的长度

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

```
int mystrlen(char s[]);    /* 函数原型 */
```

```
int main(void)
```

```
{
```

```
    char s[20]="world";
```

```
    printf("%d\n",mystrlen(s));    /* 函数调用 */
```

```
    return 0;
```

```
}
```

```
int mystrlen(char s[ ])    /* 函数定义 */
```

```
{
```

```
    int i=0;
```

```
    while(s[i]!='\0') i++;
```

```
    return i;
```

```
}
```

```
char s[20]="world"; /* 用  
int counter; /* 计数器变量,  
counter =0;      /* 计数器变  
while(s[counter] !='\0')  
    ++counter;  
printf( "%d", counter);
```





## 2.4.5 符号常量

用一个标识符表示一个常量. 易于阅读, 便于修改

C语言中有三种定义符号常量的方法:

(1) 用**#define**指令 (常用)

```
#define PI 3.1415926
```

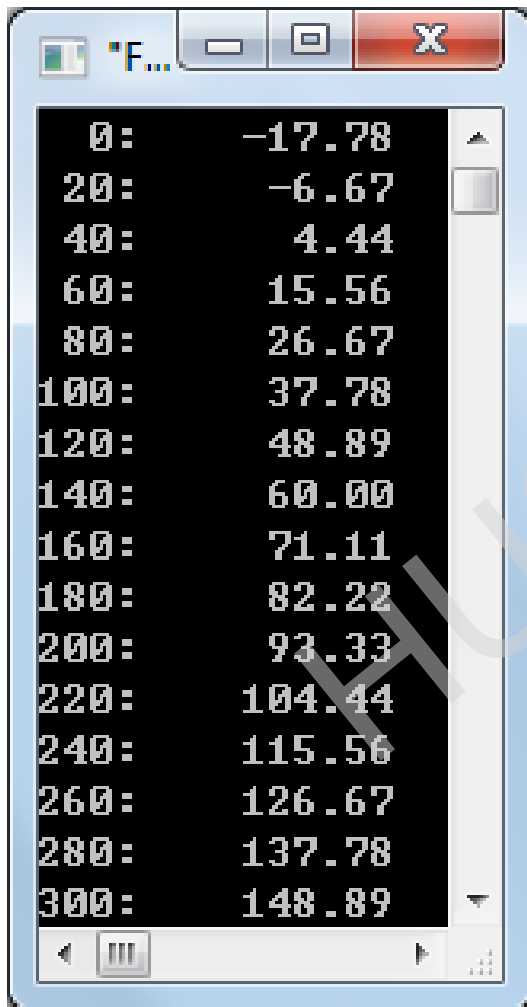
(2) 用**const**声明语句 (只读变量、常变量)

```
const float pi=3.14;
```

(3) 用枚举类型 (在2.9节介绍)

```
enum week { SUN, MON, TUE, WED, THU, FRI, SAT };
```

# 输出华氏和摄氏温度对照表



0:	-17.78
20:	-6.67
40:	4.44
60:	15.56
80:	26.67
100:	37.78
120:	48.89
140:	60.00
160:	71.11
180:	82.22
200:	93.33
220:	104.44
240:	115.56
260:	126.67
280:	137.78
300:	148.89

温度转换公式为:

$$^{\circ}\text{C} = \frac{5}{9} (^{\circ}\text{F} - 32)$$

# while循环实现

```
/* print Fahrenheit-Celsius table */
```

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

```
int main(void)
```

```
{
```

```
    int fahr;
```

$$^{\circ}\text{C} = \frac{5}{9} (^{\circ}\text{F} - 32)$$

```
    fahr=0;
```

```
    while (fahr <= 300) {
```

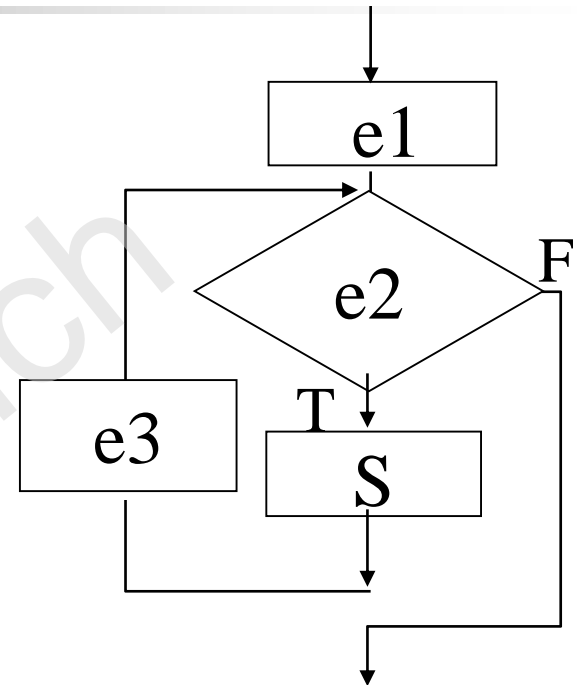
```
        printf( "%3d:  %10.2f\n" , fahr, (5.0/9)*(fahr - 32) );
```

```
        fahr = fahr + 20;
```

```
    }
```

```
    return 0;
```

```
}
```



# 符号常量的应用

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

```
#define LOWER 0      /* 表的下限 */
```

```
#define UPPER 300    /* 表的上限 */
```

```
#define STEP 20      /* 步长 */
```

```
int main(void)
```

```
{
```

```
    int fahr;
```

```
    fahr=LOWER;
```

```
    while (fahr <= UPPER) {
```

```
        printf( “%3d: %10.2f\n” , fahr, (5.0/9)*(fahr - 32) );
```

```
        fahr = fahr + STEP;
```

```
    }
```

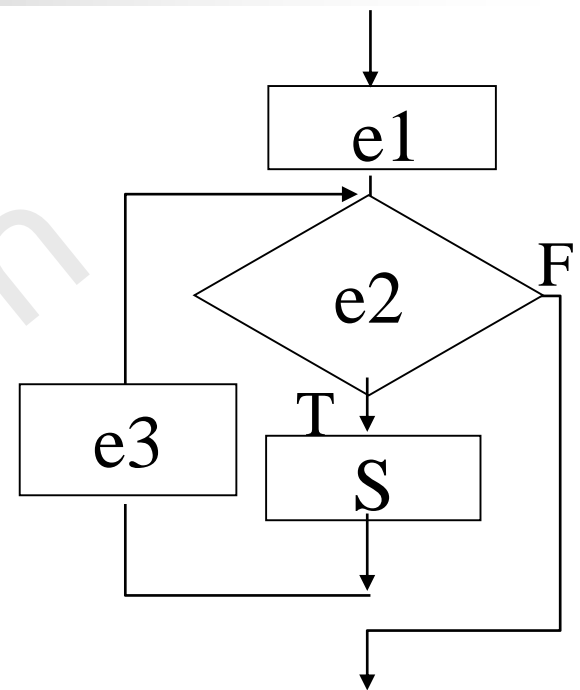
```
    return 0;
```

```
}
```

$$^{\circ}\text{C} = \frac{5}{9} (^{\circ}\text{F} - 32)$$

好处:

含义清晰、一改全改



弊端:

机械式替代, 不做任何语法检查

# for循环实现

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

```
#define LOWER 0    /* 表的下限 */
```

```
#define UPPER 300  /* 表的上限 */
```

```
#define STEP 20    /* 步长 */
```

```
int main(void)
```

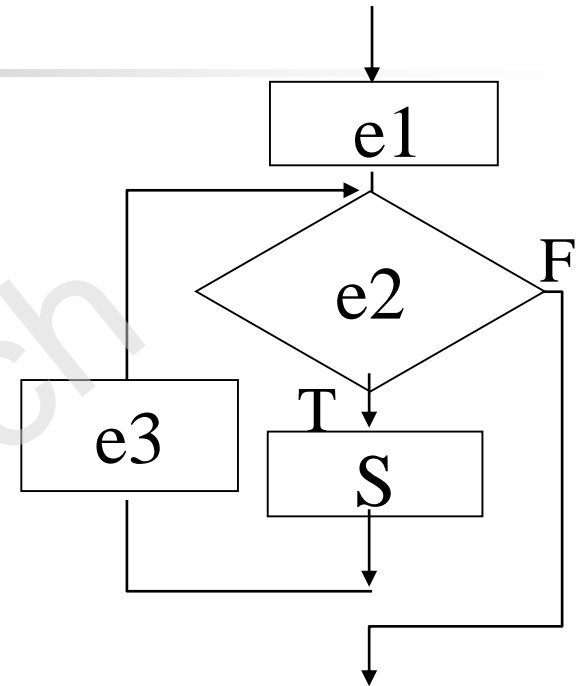
```
{
```

```
    int fahr;
```

```
    for(fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf( "%3d %10.2f\n" , fahr, (5.0/9)*(fahr - 32) );
```

```
    return 0;
```

```
}
```



$$^{\circ}\text{C} = \frac{5}{9} (^{\circ}\text{F} - 32)$$



# 用#define定义符号常量

**#define**是一种编译预处理指令,格式为:

**#define** 标识符 常量



符号常量(一般用大写,以区分变量)

```
#define LOWER 0    /* 表的下限 */  
#define UPPER 300 /* 表的上限 */  
#define STEP 20   /* 步长 */
```



# 用const定义符号常量

**const**是关键字，称为类型限定符。格式为：

**const 类型名 标识符=常量;**

例如：

**const double PI=3.14159;**

**const int DOWN=0x5000; /\* 下光标键的扫描码 \*/**

**const int YES=1, NO=0;**



## const和#define的区别

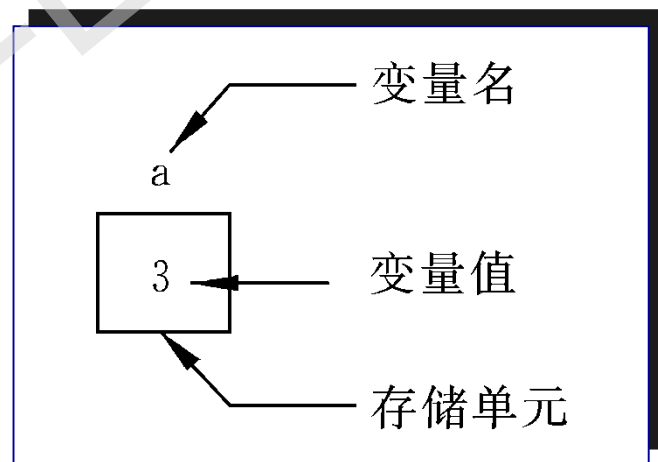
- **const**声明的标识符是一个只读变量，编译时系统会根据定义的类型为该标识符分配存储单元，并把对应的常量值放入其中，该值不能再被更改，此后，程序中每次出现该标识符都是对所代表存储单元的访问。
- **#define**定义的标识符没有对应的存储单元，只是在编译之前由预处理程序进行简单的文本替换。



## 2.4.6 变量

- 变量代表内存中具有特定属性的一个存储单元，它用来存放数据，这就是变量的值，在程序运行期间，这些值是可以改变的。

- `int a=3;`  
`scanf( "%d" ,&a);`



# 变量的定义

- 要求对所有用到的变量作定义，也就是“**先定义，后使用**”

数据类型名 变量表;

**int total, average;** //一行可以定义多个变量

- 变量在声明时可以同时赋一个初值（称为变量的显式初始化） **每个变量必须分别显式初始化。**

**int count=0, sum=0;**

**char alert='\a', c ;**

**int count=sum=0; ❌**



# 思考与练习

(1) 练习例2.9，了解`%3d`，`%10.2f`等域宽控制，然后将其中的“(5.0/9)\*(fahr - 32)”改成“(5/9)\*(fahr-32)”会输出什么结果，思考为什么。

(2) 仿写：实现将字符串反转。

例如 “hello” 反转后为 “olleh”

(3) 仿写：利用for循环实现输出1!，2!，...，10! 的值。

- **运算符**：运算的符号表示，执行对运算对象（称为操作数）的各种操作。如+、-、\*、/、++、&&、=、<=、...
- **表达式**：由**运算符和操作数(运算量)**组成的符合C的语法的**算式**。
- 单个的操作数（包括常量、变量和有返回值的函数调用）是表达式，由运算符和操作数组成的有意义的计算式子更是表达式，如：

`sqrt(b*b-4*a*c)`

`x=x*PI/180`

`fabs(an)>=EPS`

运算符分类：

- 按参与运算的操作数个数：

单目运算符    -   +   ++   --   ~   !

双目运算符    +   -   \*   /   ||   &&   <=

三目运算符    ? :

- 按其功能：

算术运算符、赋值运算符(=、+=)、

关系运算符、逻辑运算符、

位运算符、自增自减运算符、

条件运算符、逗号运算符等等。

- C语言中的表达式分类：

算术表达式、关系表达式、

逻辑表达式、赋值表达式、

条件表达式、逗号表达式

混合表达式等。



注：无论什么表达式，都会返回一个具有确定类型的结果(或值)。

尤其当两个不同类型的操作数进行运算时，会引起数据类型的转换，特别要注意结果值的类型。

# 运算符的优先级和结合性

- 当表达式中包括多个运算符时，C语言会先按优先级规则解释表达式的意义。如：

$1+2*3$  等价于  $1+(2*3)$

- 当一个操作数两侧的运算符优先级别相同时，则按“结合性”规则。

$1+2-3$  等价于  $(1+2)-3$

----从左至右的结合性（左结合）

$-a++$  等价于  $-(a++)$

----从右至左的结合性（右结合）

- 所有运算符的优先级和结合性规则见表2-4



## 2.5.2 算术运算

### ■ 运算符:

双目	单目	
<b>+</b> 加法	取正	$3+6$ , $+3$
<b>-</b> 减法	取负	$6-4$ , $-5$
<b>*</b> 乘法		$3*8$
<b>/</b> 除法		$8/5$ 值为 <b>1</b> $8./5$ 值为 <b>1.6</b>
<b>%</b> 求余		$7\%4$ 值为 <b>3</b>

■ 用算术运算符连接起来的式子是算术表达式

# 注意

两个整型数据相除（结果为整, 舍去小数部分）

$$-5/3 \Rightarrow -1 \quad 1/2 \Rightarrow 0 \quad 1./2 \Rightarrow 0.5$$

C99: 向0取整

使用时千万注意 `int / int` 出现数据丢失。

%操作数必需为整数, 余数的符号与左边数的符号相同

$$-7\%4 \text{ 值为 } -3 \quad 7\%-4 \text{ 值为 } 3 \quad -7\%-4 \text{ 值为?}$$

➤ 优先级与结合性

\*     /     %     +     —





# 求出所有的水仙花数

- “水仙花数”是一个三位数，其各位数字立方和等于该数本身。例如，153是一个“水仙花数”，因为 $153=1^3+5^3+3^3$ 。

如何判断一个三位数  $ijk$  是否为水仙花数呢？

- 判断“水仙花数”的关键是怎样从一个三位数中分离出百位数、十位数和个位数。

$i=x/100;$                       /\* 分解百位数 \*/

$j=x/10\%10;$                 /\* 分解十位数 \*/

$k=x\%10;$                     /\* 分解个位数 \*/



# 输出水仙花数的程序

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

```
int main(void)
```

```
{
```

```
    int x,i,j,k;
```

```
    for(x=100;x<=999;x++) { // 枚举每一个3位数
```

```
        i=x/100;          /* 分解百位数 */
```

```
        j=x/10%10;        /* 分解十位数 */
```

```
        k=x%10;           /* 分解个位数 */
```

```
        if(x==i*i*i+j*j*j+k*k*k) /* 是水仙花数 */
```

```
            printf("%5d",x);
```

```
    }
```

```
    return 0;
```

```
}
```

# 水仙花数程序的函数版

```
#include<stdio.h>
int main(void)
{ int x;
  for(x=100;x<=999;x++) {
    if( isNarcissus(x) )
      /*x是水仙花数*/
      printf("%5d",x);
  }
  return 0;
}
```

```
int  isNarcissus (int m)
{
  int i,j,k;
  i=m/100;
  j=(m-i*100)/10;
  k=m%10;
  if(m==i*i*i+j*j*j+k*k*k)
    return 1;
  else return 0;
}
```

函数**isNarcissus**: 判断一个数**x**是否水仙花数。  
是，返回**1**，不是返回**0**。

- 关系运算用来比较两个操作数之间的关系，因此为双目运算符
- 6个关系运算符：

&lt; (小于)

&lt;= (小于或等于)

== (等于)

&gt; (大于)

&gt;= (大于或等于)

!= (不等于)

用关系运算符将前、后两个操作数连接起来的式子称为“关系表达式”，这两个操作数可以是任意表达式。

如： 'a' != 'b' ;

'2' - '0' > 5



# 关系表达式的类型和值

- 关系表达式的类型: `int`

- 关系表达式的值:

关系成立, 值为1 (代表“真”)

关系不成立, 0 (代表“假”)

- $4 > 3 > 2$  值为 ?

# 关系表达式举例

根据变量说明，给出表达式的值。

```
int x=4, y=3, z=2;
```

```
char c='a';
```

(1) **c == 'A'+32** (1)

(2) **c+1 != 'b'** (0)

(3) **x-y<=10** (1)

(4) **z=x>y** (1)

(5) **x>y>z** (0)



# 常见的编程错误

- 将运算符**`==`** 写成运算符**`=`**（赋值）。可能会因为运行时的逻辑错误而导致不正确的结果。例如，  
**`if (grade == 'A') printf("Very Good! ");`**  
**`/*当成绩的等级为A等时，输出Very Good!*/`**

而：

**`if (grade = 'A') printf("Very Good!");`**  
**`/*不论成绩的等级是几等，总输出Very Good!*/`**

# 注意

- 数学上判断 $x$ 是否在区间 $[a,b]$ 中时，习惯写成：

$$a < x < b$$

- C中“ $a < x < b$ ”的含义与数学中的含义不同，应写成：

$a < x \ \&\& \ x < b$



逻辑运算符



## 2.5.4 逻辑运算

### ■ 3个逻辑运算符

**&&**（逻辑与）、**||**（逻辑或）、**!**（逻辑非）

逻辑表达式：用逻辑运算符将关系表达式  $a \leq x \ \&\& \ x \leq b$   
或逻辑量连接起来的式子  $0 \ \&\& \ 2$ 、 $!a$

- `if(a和b均大于10) x=a+b;`  
`a>10&&b>10`
- `if(a和b有一个大于10) x=a+b;`  
`a>10 || b>10`
- `if(a为0) x=a;`  
`a==0 或 !a`

# 逻辑表达式的类型和值

- 逻辑运算的操作数可以是0和任何非0的数值 **0 && 2**
- 系统最终以0判断属于“假”（0代表“假”）  
以非0判断属于“真”（1代表“真”）
- 逻辑表达式的类型为 **int**  
值为 **1** (“真”) 或 **0** (“假”)
- **if(a和b都非0) x=a+b;**  
 **$a \&\& b \iff a \neq 0 \&\& b \neq 0$**

# 逻辑运算真值表

简单示例:

$X = 0; Y = 2;$

则:

$X \&\& Y$  ---- 0

$X \parallel Y$  ---- 1

$!Y$  ---- 0

$!X$  ---- 1

表达式X	表达式Y	!X	!Y	X&&Y	X    Y
非0	非0	0	0	1	1
非0	0	0	1	0	1
0	非0	1	0	0	1
0	0	1	1	0	0



# 写表达式

(1) 整数a是偶数

$!(a\%2)$  或  $a\%2==0$

(2) 字符c的值是英文字母。

$c>='a' \ \&\& \ c<='z' \ || \ c>='A' \ \&\& \ c<='Z'$

(3) 某一年year是闰年。如果某一年的年份能被4整除但不能被100整除，那么这一年就是闰年，此外，能被400整除的年份也是闰年。

$!(year\%4) \ \&\& \ year\%400 \ || \ !(year\%400)$



# 判闰年的程序

```
#include<stdio.h>
/* 判断闰年,year 是闰年, 返回1; 否则, 返回0 */
int isleap (int year)
{
    if ( !(year%4) && year%400 || !(year%400) )    return 1;
    else      return 0;
}
int main(void)
{
    int year;
    scanf("%d",&year);
    if ( isleap(year) ) printf("%d is a leap year\n",year);
    else printf("%d is not a leap year\n",year);
    return 0;
}
```

# 注意

- 编译程序在处理含有**&&**、**||** 表达式时，往往**采用优化算法**(提高速度)。

**e1 && e2** 一旦发现**e1=0**，不再计算 **e2**

**e1 || e2** 一旦发现**e1=1**，不再计算 **e2**

- 如，

**x >= 0.0 && sqrt(x) <= 7.7**

**/\* 如果x值为负，则不求x的平方根 \*/**

- 单目运算符, **只能用于变量**

- ++ (自增)**, 使变量值加1

- (自减)**, 使变量值减1

- 有**前置式**和**后置式**:

- ++X**

- X++** 相当于 **X=X+1;**

- X**

- X--** 相当于 **X=X-1;**

运算说明:

1. **前置运算: 先变后用**

如:  $x = 1; y = ++x; // x=2, y=2$

分析: 先执行 $x=x+1$ , 再执行 $y=x$ ;

2. **后置运算: 先用后变** //  $x=2, y=1$

如:  $x = 1; y = x++;$

分析: 先使用 $x$ 值, 即 $y=x$ ;

再执行对 $x$ 的加1运算,  $x=x+1$ ;

又如:  $x = 10; y = x++ + x;$

结果:  $x = 11, y = 20? 21? 22?$

**产生副作用**



## 【例2-15】统计正文的字符数和行数

- 正文是一行行字符组成的字符序列，每一行是以换行符为结束标志的一串字符。

Upload file ✓

Special pages ✓

Page information ✓

Ctrl+z ✓

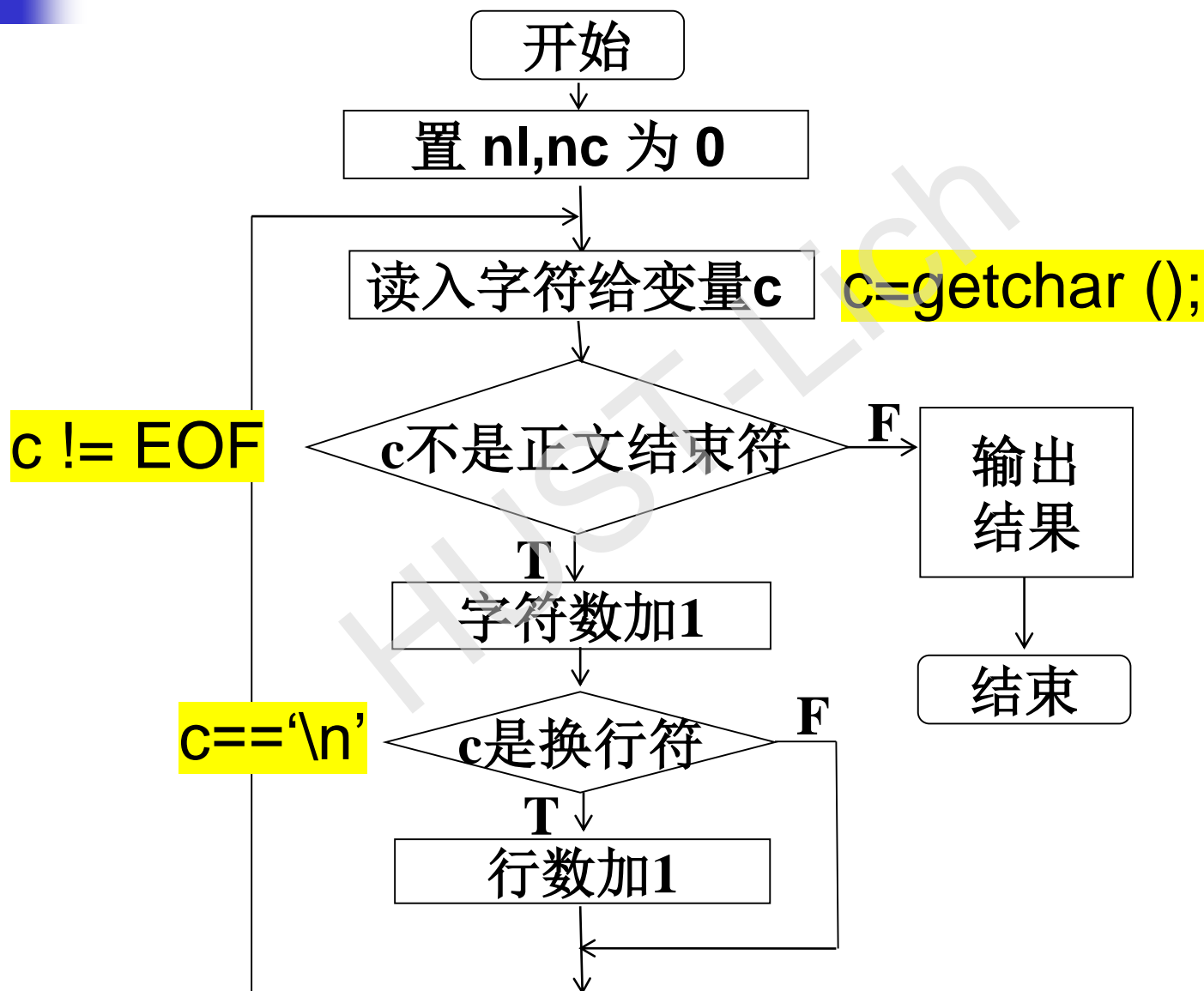




# 文件结束标志

- 输入正文以**Ctrl+Z**（DOS系统）或**Ctrl+D**（UNIX系统）为结束标志（称为文件尾）
- 字符的输入可以使用标准库函数**getchar**或**scanf**，遇文件结束标志时返回**EOF**，
- **EOF**是在头文件<stdio.h>定义的符号常量，其值为**-1**。// #define EOF -1

# 流程图



# 程序

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

```
/*统计输入正文的字符数和行数。*/
```

```
int main(void)
```

```
{
```

```
    char c;
```

```
    int  nc,nl;
```

```
    printf("Input a text end of ctrl+z:\n"); /* 提示语 */
```

```
    nc=nl=0;
```

```
    while((c=getchar())!=EOF)
```

```
    {
```

```
        ++nc; /* 字符数自增1, 可以用nc++代替 */
```

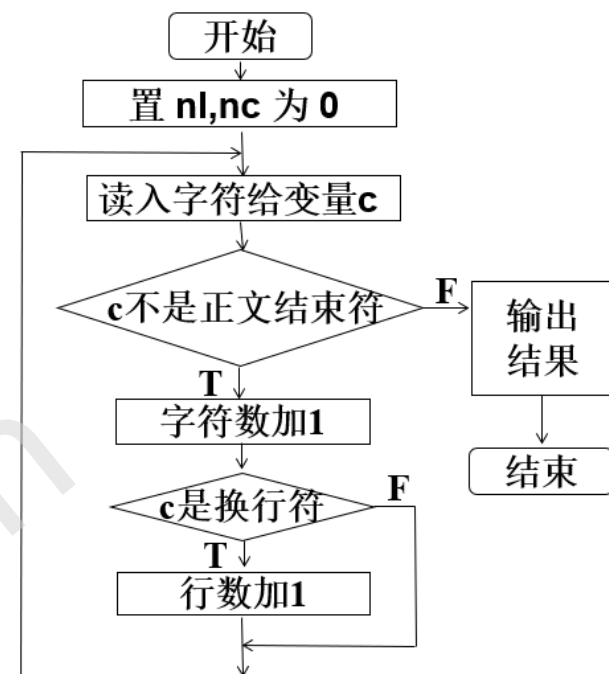
```
        if(c=='\n') ++nl; /* 行数自增1, 可以用nl++代替 */
```

```
    }
```

```
    printf("nc=%d,nl=%d\n",nc,nl);
```

```
    return 0;
```

```
}
```





# 思考与练习

**思考：**将例2.15程序中的**while**循环改为**for**循环

**练习：**

1. 统计输入正文中水平制表符、空格、换行符的个数
2. 把输入的一行字符复制到屏幕输出，但是前导空格不输出。如输入：**Upload file ✓**  
则输出：**Upload file ✓**

均要求用**getchar**函数输入字符

# 计算 $1+2+3+\dots+n$

//计算 $1+2+3+\dots+n$

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

```
int main(void)
```

```
{
```

```
    int i,sum,n;
```

```
    printf("n=");
```

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

```
    sum=i=0;
```

```
    while(++i<=n)
```

?

while(i++<=n)

```
        sum=sum+i; // sum+=i;
```

```
    printf("sum=%d\n",sum);
```

```
    return 0;
```

```
}
```



## 后缀式举例：求表达式的值

**int a=1,b=0;**

**(1) b++ + b++**

值不确定

**(2) a--&& a**

表达式值为0， a为0

**(3) b++ ? b : -b**

表达式值为-1， b为1

# 序列点

程序执行中的点。变量值的改变必须在序列点之前完成，才能保证在序列点之后，是更改后的新值。

C标准中定义的序列点：

- ① **&&**、**||**、**?:** 和 **,** 四个运算符的第一个操作数之后
- ② 完整表达式结束时，即表达式语句的分号处、**return**语句中的表达式、**if**、**switch**或循环语句中的条件表达式（包括**for**语句中的每个表达式）之后

定义序列点，主要是为了尽量消除编译器解释表达式时的二义性

# ++/--运算符举例

求表达式和各变量的值

**&&**、**||**、**?:** 和 **,** 运算符的第一个操作数之后

```
int a=1,b=0;
```

(1)  $a--$  **&&**  $a$

表达式值为0, a为0

$a--$  **||**  $a$

表达式值为1, a为0

$a$  **||**  $--a$

右侧表达式 $--a$ 没有执行

表达式值为1, **a为1**

(2)  $b++$  **?**  $b:-b$

表达式值为-1, b为1

$b$  **?**  $a++:b--$

表达式值为0, b为-1, **a为1**

表达式 $a++$ 没有执行

(3)  $b++ + b++$   
 $++b + b++$

值不确定, **产生副作用**



# 避免使用具有副作用的表达式

- ✓ **表达式的副作用**是指：表达式在求值过程中要改变该表达式中作为操作数的某个变量的值。
- ✓ **具有副作用的运算符**：赋值、复合赋值、自增、自减
- ✓ **C** 标准未规定函数参数的求值顺序，也没有规定除了**&&**、**||** 和 **,** 外的双目运算中两个操作数的计算顺序。由编译器自行规定计算顺序。
- ✓ **由于编译器对求值顺序的不同处理**，当表达式中带有副作用的运算符时，就有可能产生二义性。



# 注意

- 在一个表达式中不要多处出现变量的自增、自减等运算，同时避免与其它运算符连用
- 尽量不要在函数参数中用自增减表达式

- 自增、自减运算符只能用于变量，不可用于常量和表达式

因为表达式在内存内没有具体空间，常量所占的空间不能重新赋值

3++      (x+y)++      (-i)++      ✗

- 当出现同一级别的多个运算符时，按自右至左方式结合

$-i++ \iff -(i++)$

对于  $i+++j \iff (i++) + j$

$*p++ \iff *(p++)$

C语言解析时尽量往长记号进行解析，所以  
+++一般会处理成++ +而不是+ ++

若  $i=3, j=2$   $(i++) + j$  等于 5       $i=4, j=2$

- 简单的赋值运算符 “=”，双目运算符，其一般形式为：

操作数1 = 操作数2; //左值表达式=右值表达式;

左值 (lvalue)：赋值运算符左侧的标识符

变量可以作为左值

而表达式、常量不能作为左值(如a+b)

```
int    i , j ;
char   m , n ;
float  x , y ;
double z ;
```

j = i ;      i、j类型相同,无需转换,直接将i的值赋给j

i = m ;      m由char型向int型转换, 将转换后的值赋给i

z = x \* i ;      x\* i的结果为double型, 然后赋值给z

i = m < n ;      m < n的结果为整型, 无需转换, 直接将值赋给i

i = j = 10 ;      这是一个**多重赋值表达式**, 赋值运算符按从右至左结合, 即相当于 i=(j=10), 先将10赋给j, 而括号中的赋值表达式 (j=10) 的值就是赋值后的j 的值, 再将其赋给i。

## ❖ 多字节→少字节

低位照搬

```
int a=32767;
```

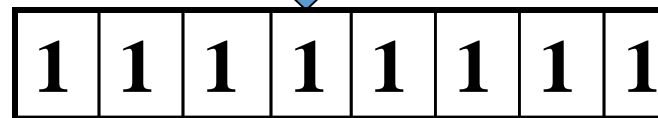
```
short int b;
```

```
b=a;
```



**b=-1**

**b**



# 赋值运算

- 复合的赋值运算符: 在 “ = ” 之前加上一个双目运算符构成。有10种:

$+=$ 、 $-=$ 、 $*=$ 、 $/=$ 、 $\%=$

$\&=$ 、 $|=$ 、 $\wedge=$ 、 $<<=$ 、 $>>=$

如:

$i += j;$

等价于  $i = i + j;$

$x *= y - 5;$

等价于  $x = x * (y - 5)$

$m <<= 2;$

等价于  $m = m << 2$

$x *= k = m + 5$

等价于  $x = x * (k = m + 5)$

## ■ 与左操作数的值和类型相同

```
int x;
```

```
x=5.6; //表达式(x=5.6)的值即x值5, 类型int
```

```
//等价于 x=(int)5.6;
```

- 强制类型转换运算符  
(类型名) 操作数

# 赋值运算符的结合性

- 右结合性，如

**a=b=c=3;**

等价于

**a= (b= (c=3) ) ;**

- C把赋值处理为运算符，其好处是使赋值表达式可以像其它任何表达式一样当作一个数据来处理。 如

**a=2;**

**b=3;**

**x=a+b;**

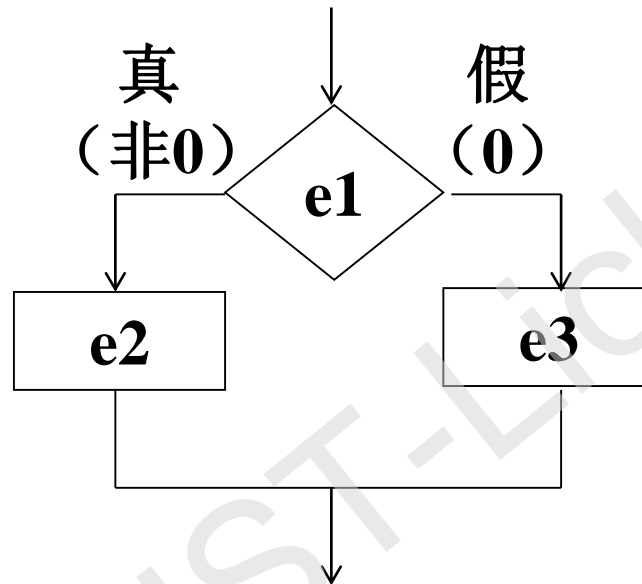
可被简化为：

**x= (a=2) + (b=3);**

## 2.5.7 条件运算

■  $e1 ? e2 : e3$

?: 三目运算符



if(e1)

值=e2;

else

值=e3;

■  $x=10;$

$y=x>9?100:200$  /\* y=100 \*/

■ 如果字符变量c是数字，将其转换为对应的整数，否则，c的值不变。

$c = (c \geq '0' \ \&\& \ c \leq '9') ? c - '0' : c$



从右向左结合

$$y = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

```
if (x>0)
    y=1;
else if(x<0)
    y=-1;
else
    y=0;
```

**$y = x > 0 ? 1 : \underline{x < 0 ? -1 : 0}$**

↓ (因为是右结合性, 括号可省略)

**$y = x > 0 ? 1 : (x < 0 ? -1 : 0)$**

多么简洁的表达!



## 练习

- (1) 一个数如果恰好等于除它本身外的所有因子之和，这个数就称为“完数”。例如 $6=1+2+3$ ，则6是完数。
- (2) 改写例2.8, 统计字符串中十六进制数字字符的个数。
- (3) 改写例2.15, while循环改为for循环。
- (4) 练习例3.1, 3.2, 3.3, 3.4, 熟悉getchar和putchar用法。
- (5) 用getchar输入一段正文，将输入中的双引号（”）和反斜线（\）转换成对应的转义序列（\”，\\）输出, 其他字符原样输出。字符输出用putchar。例如，

输入 123” as

\world

^Z

输出 123\” as

\\world

```
char c=getchar(); putchar(c);
```

- 逗号运算符 ‘,’ 是**双目运算符**，用它构成的逗号表达式形式为：

**$e_1, e_2, e_3, \dots, e_n$**

- **计算规则**：从左往右，依次计算 $e_1$ 、 $e_2$ 、...、 $e_n$ 的值，

**逗号表达式的值和类型与  $e_n$  的值和类型相同**

- **示例1**：若`int a=10, b;` 求以下表达式的结果

`b = (a++, a % 3);`

结果为： `b = 2`

表达式1的结果为10；  
同时计算`a++`，`a`变为11

表达式2的结果为2，即：  
整个逗号表达式的值为2，  
然后把2赋值给`b`。

思考：1)若 `b = a++, a % 3;` `//a=? b=?` 表达式的值为?



2)若 `b = (a++, a++, a % 3);` `//a=? b=?` 表达式的值为?

3) `x=(i=4, i%3);` `//x=?` 表达式的值为?

4) `x=i=4, i%3;` `//x=?` 表达式的值为?

# 逗号表达式的应用

//计算 $1+2+3+\dots+n$

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

```
int main(void)
```

```
{
```

```
    int i,sum,n;
```

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

```
    sum=0;
```

```
    for(i=1;i<=n;i++) {
```

```
        sum+=i;
```

```
    }
```

```
    printf("sum=%d\n",sum);
```

```
    return 0;
```

```
}
```

**for(sum=0, i=1; i<=n; sum+=i, i++)**  
**;**

空语句



# 注意

并不是任何地方出现的逗号都是作为逗号运算符。  
例如函数参数也是用逗号来间隔的。

```
printf(“%d,%d”, 2, 3); /* 输出: 2, 3 */
```

```
printf(“%d,%d”, (2, 3), 3 ); /* 输出: 3, 3 */
```

“2, 3”并不是  
一个逗号表达  
式，它是printf  
函数的2个参数

“ (2, 3) ”  
是一个逗号表  
达式，它的值  
等于3。

# 反转字符串s

```
void Reverse(char s[ ])
```

```
{  
    int i=0, j;    // 首尾字符位置  
    char mid;  
    j=strlen(s)-1; // 最后一个字符位置 */  
    while(i<j) {  
        mid=s[i]; s[i]=s[j]; s[j]=mid; //交换s[i]、s[j]  
        i++;  
        j--; // i、j向中间靠拢  
    }  
}
```

```
for(i=0,j=strlen(s)-1;i<j;i++,j--)  
    mid=s[i],s[i]=s[j],s[j]=mid;
```

逗号运算符特别适用于程序中需要执行多个表达式，而语法上只允许为一个表达式的情况

## 【例2.19】模拟实现scanf("%d",&x) 的功能

- 函数getchar 实现了scanf("%c",&c) 的功能
- 定义函数getint, 实现scanf("%d",&x) 的功能。

实际上输入的是数字字符, 函数读到的是该字符的字符码。

两个步骤:

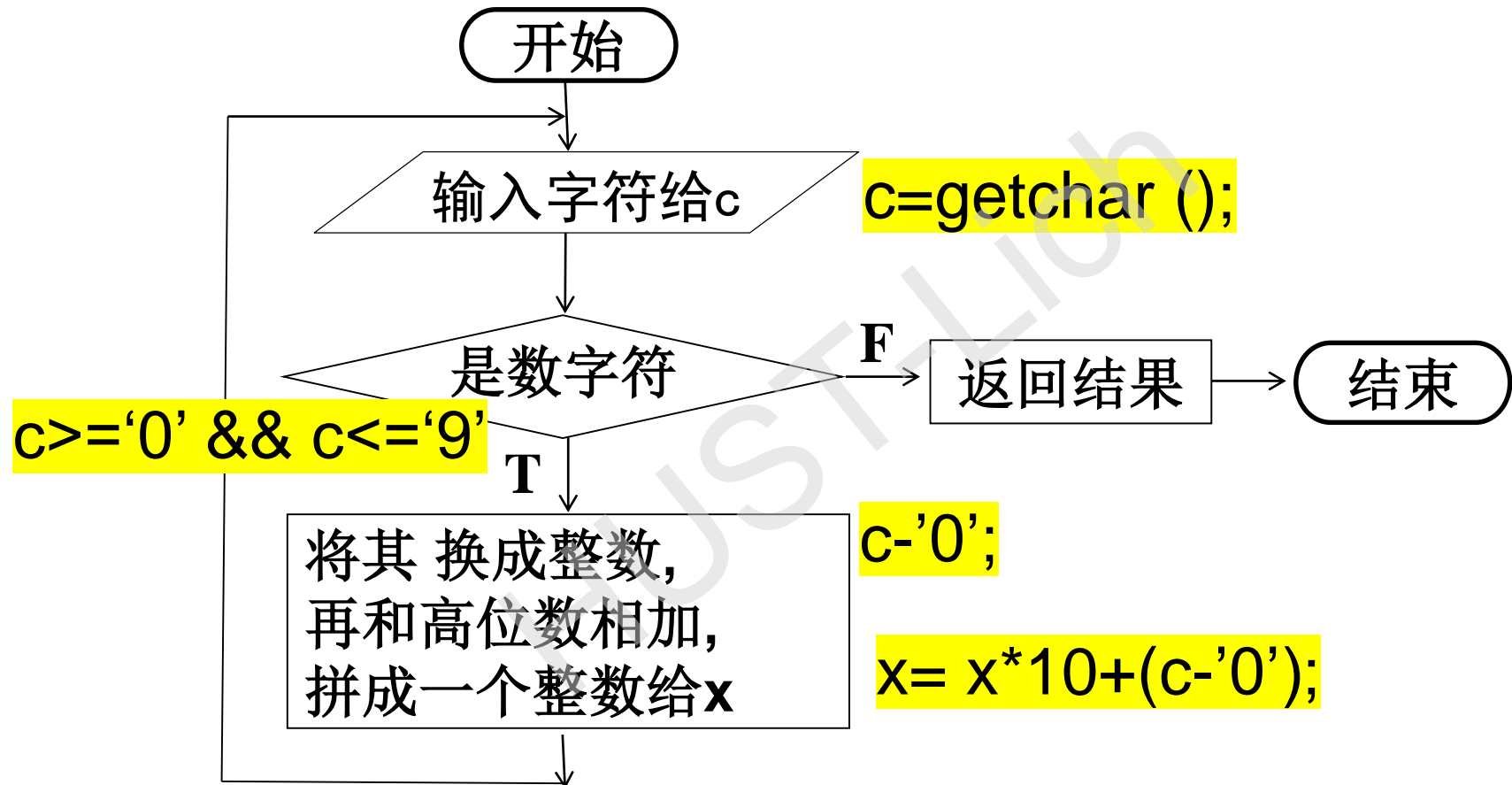
- 数字字符转换成对应的整数

$c - '0'$

- 循环拼数

$x = x * 10 + c - '0'$

# 流程图





# 程序

```
int getint(void)
{
    char c;    /* 用于存放当前输入的字符 */
    int x;     /* 用于存放转换的整数 */
    for(x=0, c=getchar(); c>='0' && c<='9'; c=getchar()) /* 遇非数字字符结束 */
        x=10*x+c-'0';    /* c-'0': 将数字字符转换成对应的一位整数 */
    return x;
}

int main(void)
{
    int score;
    score=getint();
    printf("%d\n", score);
    return 0;
}
```

一个单目运算符，有两种形式

### (1) sizeof (类型名)

计算**指定数据类型**占用的存储字节数

假设 `sizeof(int)=2`; 则

`sizeof(long)=?` */\* 值为4 \*/*

### (2) sizeof 表达式

给出**表达式结果的类型**占用的存储字节数

`double x; sizeof(x)=?` */\* 值为8 \*/*

`short a[10]; sizeof(a)=?` */\* 值为20 \*/*

`int a=1, b=1; sizeof(a+b)=?` */\* 值为2 \*/*

`sizeof(a) + b =?` */\* 值为3 \*/*

先求sizeof a，再和b加，  
值为3 \*/

# 注意

## sizeof 是一个常量表达式

- 其运算是在**编译时执行的**。因此，当**sizeof**的操作数是表达式时，则在编译时分析表达式以确定类型，**运行时不对这个表达式求值**。

例如：

```
short x=1;
```

```
sizeof(++x); /* x不增1，仍为1 */
```

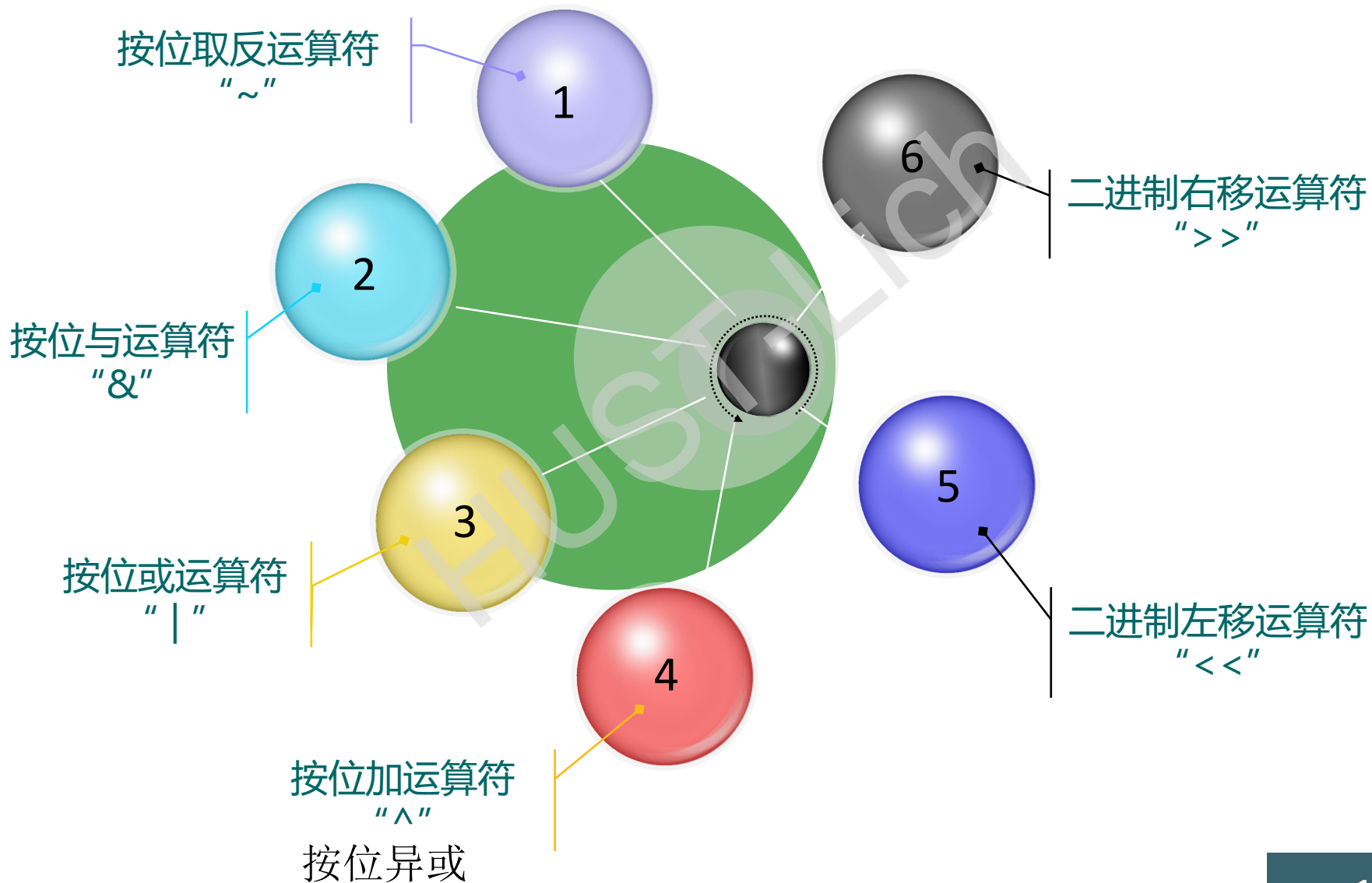
```
short x=1;  
printf("%d,%d,%d",sizeof(short),sizeof(++x),x);
```

2, 2, 1

- **sizeof运算符的结果为无符号整型**

比较有符号和无符号数据类型时，在比较之前会进行隐式转换，最终可能会产生令人惊讶的结果。

```
上机试试: if (sizeof(int)> -1) printf( "TURE" );  
           else                printf( "FALSE" );
```



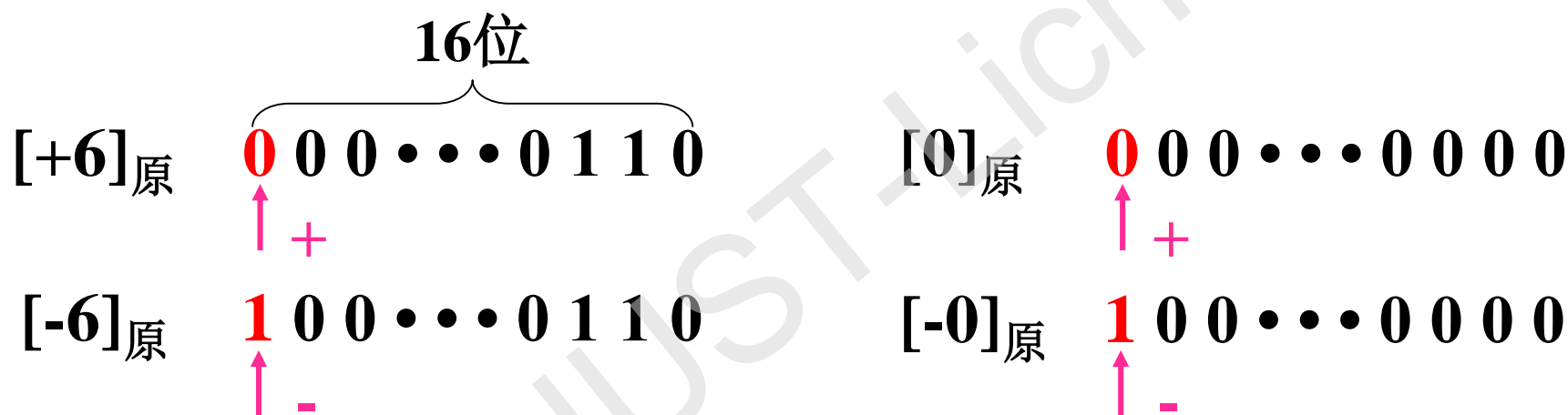
## 2.6.1 数的机器码表示

- **机器数**：一个数在计算机中的二进制表示形式，称为这个数的机器数。机器数的最高位为符号位，其余各位为数值位。  
且规定：正数的符号位为0，负数的符号位为1。  
如：6的机器数：0 0 0 0 0 1 1 0（假设计算机字长为8位）  
-6的机器数：1 0 0 0 0 1 1 0 同时也是 $2^7+6$ 的机器数的形式值
- **真值**：带符号的机器数对应的真实数值称为该机器数的真值。  
即正、负号后跟二进制数的绝对值就构成真值。  
0 0 0 0 0 1 1 0的真值为：+ 0 0 0 0 1 1 0，即+6  
1 0 0 0 0 1 1 0的真值为：- 0 0 0 0 1 1 0，即 -6
- 机器数包含原码、反码和补码三种形式

计算机是以补码形式存放数据的

# 原码、反码、补码

原码：最高位为符号位，其余位表示数值



0的原码表示不惟一

# 原码、反码、补码

- 正数的补码、反码，与原码相同
- 负数的补码 = 反码+1 (符号位参与运算)

负数的反码：在原码基础上，符号位不变，其余各位取反

$[-6]_{\text{原}}$  1 0 0 ··· 0 1 1 0

$[-6]_{\text{反}}$  1 1 1 ··· 1 0 0 1  
+1

$[-6]_{\text{补}}$  1 1 1 ··· 1 0 1 0

$[-0]_{\text{原}}$  1 0 0 ··· 0 0 0 0

$[-0]_{\text{反}}$  1 1 1 ··· 1 1 1 1  
+1

$[-0]_{\text{补}}$  1 0 0 0 ··· 0 0 0 0

1 进位被舍掉

正数在机内以其二进制原码表示  
负数在机内以其二进制补码表示

$\therefore [+0]_{\text{补}} = [-0]_{\text{补}} = 00000000$

## 对操作数的每个二进制位逐位“取反”

- 如：

5的二进制为： 00000000 00000101

~5的二进制为： 11111111 11111010

~5的值为？

若为short型，其(真)值为 **-6**

若为unsigned short型，其(真)值为？ **65530**

```
printf(“~a=%hd”,~5); //解释为有符号，输出 -6
```

```
printf(“~b=%hu”,~5); //解释为无符号，输出 65530
```



对两个操作数逐位 “求与”

- 运算真值表:

位1	位2	位1&位2
0	0	0
0	1	0
1	0	0
1	1	1

- 示例:  $a=0x96$ ,  $b=0x80$ , 求  $a\&b$

$a=0x96$  二进制表示 1001 0110

$\& b=0x80$  二进制表示 1000 0000

---

=  $0x80$  二进制表示 1000 0000

都为1时, 结果为1, 否则为0

(假设计算机字长为8位)

# 按位与的作用

## (1)屏蔽一个整数的某些位（将某些位置零），保留其余位

例：a=0x55, 要保留a的高四位，屏蔽低四位

**0xf0 称为屏蔽码mask  
(逻辑尺、掩码)**

表达式为： a&0xf0

运算过程如下：

a=0x55	二进制表示	0101	0101
& b=0xf0	二进制表示	1111	0000
<hr/>			
=	0x50	二进制表示	0101 0000

**将掩码中与需屏蔽的对应位设为0，要保留的对应位设为1**

## (2)测试一个数的某些位是0，还是1

屏蔽码

例：a=0x55, 判断a的第5位是否为1，将a与2<sup>5</sup>进行&运算

表达式为： (a&32) ? 1 : 0

**将掩码中对应需测试的位设为1，其余位设为0**

对两个操作数逐位“相或”

- 运算真值表

位1	位2	位1 位2
0	0	0
0	1	1
1	0	1
1	1	1

都为0时，结果为0，否则为1

- 示例：

$0x0055 \mid 0xff00 = 0xff55;$  (假设计算机字长为16位)

用掩码  $0xff00$  通过 | 运算，取出了一个数的低8位、高8位全置为1

通过设计合适的掩码，与某数进行 | 运算，可实现二进制串的拼接

## 2.6.2

# 按位加 “^” (异或运算)

将两个操作数逐位 “相加”

- 运算真值表：**不进位的加法运算**  
**相同为0，不同为1**

位1	位2	位1^位2
0	0	0
0	1	1
1	0	1
1	1	0

- 示例:

$a = 0x36$ ,  $b = 0x0f$ , 求  $c = a \wedge b$ .

**对称性、结合性、交换性、自反性**  
 $a \wedge b = b \wedge a$ ;  $a \wedge a = 0$ ;  $a \wedge 0 = a$ ;

$a = 0x36$  二进制为 0011 0110  
 $\wedge$   $b = 0x0f$  二进制为 0000 1111

**若结果再次翻转, 则变回原数a, 即:**  
 $c \wedge b = a$ ;

$=$   $0x39$  二进制为 0011 1001

**翻转了低4位, 高4位不变**

$c = a \wedge b$ ;  
 $a = c \wedge b$ ;  
 $b = c \wedge a$ ;

**掩码设计原则: 某位要翻转就异或1, 某位要保持不变可异或0**

**异或典型应用: 数据加密、数据传输校验、存储数据校验、纠错、数据处理等**

# 按位与(&)、或(|)、加(^)运算

表达式	二进制表示	十六进制值	说明
<b>x</b>	01101000 11010001	0x68d1	待处理数据
<b>mask</b>	11111111 00000000	0xff00	逻辑尺(屏蔽码、掩码)
<b>x &amp; mask</b>	01101000 00000000	0x6800	将x的低字节置为0, 保留高字节
<b>x   mask</b>	11111111 11010001	0xffd1	将x的低字节保留, 高字节置为1
<b>x ^ mask</b>	10010111 11010001	0x97d1	将x的低字节保留, 高字节翻转

# 简单应用

## 【例2.20】判断一个整数是奇数还是偶数

二进制的最低位为0表示该数为偶数，为1表示该数为奇数。

表达式：  $!(x \& 1)$  //偶数表达式结果为1，奇为0

$\text{if}(!(x \& 1)) \text{ printf}("%d \text{ 是偶数}", x);$  //  $x \& 1 == 0$

## 【例2.21】字母的大小写转换

表达式：  $c \wedge 32$

'A' 0100 0001

0x41

只需翻转第5

'a' 0110 0001

0x61

位, 掩码为 $2^5$

## 【例】简单的加密

比如你想对某人说21，但不想让别人知道，于是约定用其生日18作密钥（key）。写出值x的密文表达式  $x \wedge \text{key}$

$21 \wedge 18 = 7$ ，把7告诉Ta。

利用^对称性，Ta再次计算  $7 \wedge 18$ ，得到密文21。

- $e \ll n$  将e的每位向左移n位，低n位填入0，高位丢弃

左移1位相当于该数值乘以2

- $e \gg n$  将e的每位向右移n位，低位丢弃，高n位可能填入？

--e是无符号类型，填入0；

--e是有符号类型，一些机器填入0（即“逻辑移位”），

而另一些机器则填入符号位（即“算术移位”）。

- 在使用右移运算符时，应经常用无符号类型。
- 右移一位相当于该数值除以2

表达式	二进制表示	值	行为
a	00000000 00001011	11	对a进行负号运算 a左移2位，相当于乘以 $2^2$ b右移3位，相当于除以 $2^3$ -a右移2位(填入符号位1，而其他机器可能是0)
b	00000000 00001111	15	
-a	11111111 11110101	-11	
a<<2	00000000 00101100	44	
b>>3	00000000 00000001	1	
-a>>2	11111111 11111101	-3	



# 写表达式

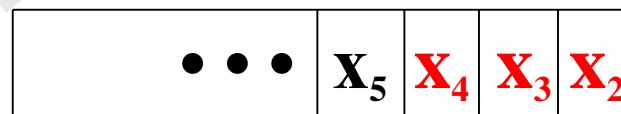
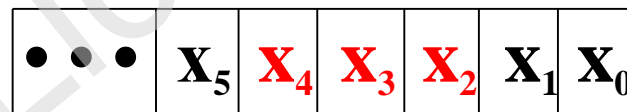
【例2.23】将整数x的第m位起的右n位取出来，作为另一个数

注：一个整数的各个二进制位从右至左依次编号为第0位、第1位、第2位、...

如：m=4 n=3

① 将要取出的那n位移到最右端

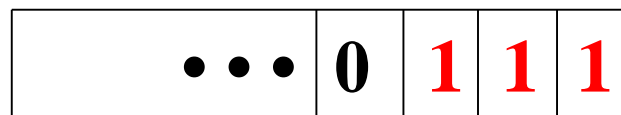
$$x \gg m-n+1$$



② 设计一个逻辑尺：低n位全为1，其余位全为0

$$\sim(\sim 0 \ll n)$$

$$(1 \ll n) - 1$$



③ 将上面二者进行&的运算

$$x \gg (m-n+1) \ \& \ \sim(\sim 0 \ll n)$$

# 位运算应用---压缩存储

【例2.24】可以把表示21世纪日期的日、月和年3个整数压缩成一个16位的整数(short)。写一个表达式，实现压缩存储日、月和年。

分析：日有31个值，月有12个值，年有100个值，因此可以在一个整数中用5b表示日，用4b表示月，用7b表示年

15	11 10	7 6	0
day	month	year	

将day、month左移到新位置，再利用位运算 | 进行拼接

$\text{day} \ll 11 \mid \text{month} \ll 7 \mid \text{year}$



# 压缩函数pack

---

函数名: **pack**

输入参数: **day**-日

**month**-月

**year**-年

返回值: 压缩后的整数

**short pack(short day, short month, short year)**

{

**return(day<<11 | month<<7 | year) ;**

}

# 压缩存储日、月和年

```
short pack(short day, short month, short year); //函数声明语句
```

```
int main()
```

```
{
```

```
    short day,month,year,date;
```

```
    printf("Input year,month,day\n");
```

```
    scanf("%hd%hd%hd", &year,&month,&day);
```

```
    date=pack(day,month,year);    //函数调用
```

```
    printf("%hx\n",date);
```

```
    return 0;
```

```
}
```

```
short pack(short day, short month, short year) //函数定义
```

```
{    return(day<<11 | month<<7 | year) ; }
```

输入/出 short 数据 用 %hd  
输入/出 unsigned short 用 %hu

# 练习：编写解压函数

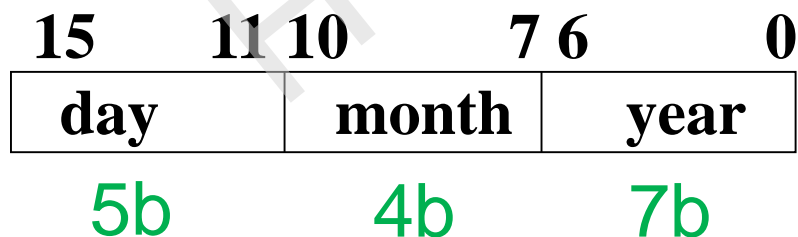
```
void unpack(short x)
```

```
{
```

将x解压为 day, month, year

```
printf : day, month, year
```

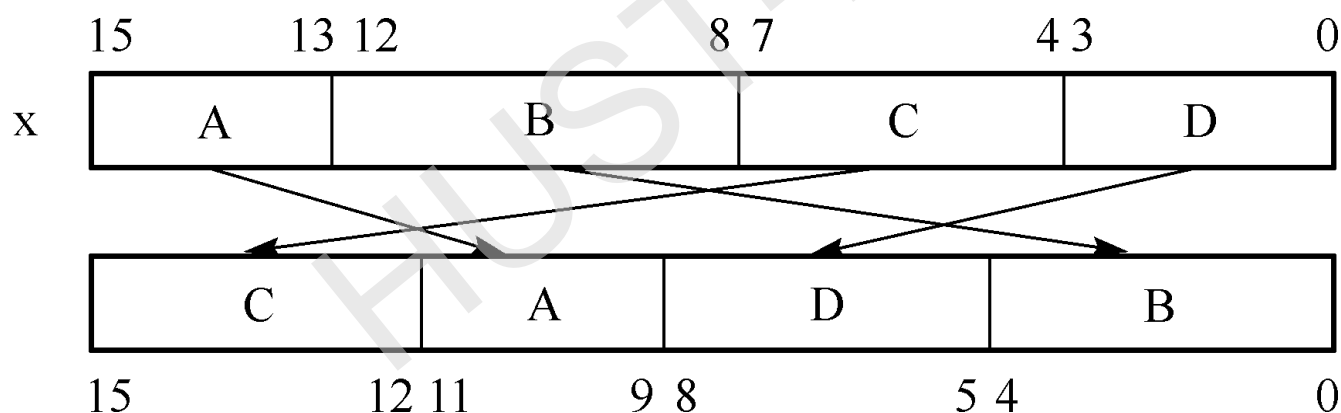
```
}
```



## 【例2.25】位运算应用---加密

将一个短整型数 $x$ 分成4个长度不等的部分：

**A (3b)**、**B (5b)**、**C (4b)** 和 **D (4b)**，然后将它们按照 **CADB** 的顺序重新拼凑在一起，实现对其加密的功能。写一个表达式，求值为 $x$ 的密文。



$(x \& 0xE000) \gg 4 \mid (x \& 0x1F00) \gg 8 \mid (x \& 0xF0) \ll 8 \mid (x \& 0x0F) \ll 5$



# 加密函数

函数名: **encrypt**

函数参数:

**x**-待加密短整数

函数返回值:

加密后的短整数

```
#define AE  0xE000    /* 取出A的逻辑尺 */
#define BE  0x1F00    /* 取出B的逻辑尺 */
#define CE  0x00F0    /* 取出C的逻辑尺 */
#define DE  0x000F    /* 取出D的逻辑尺 */

unsigned short encrypt(unsigned short x)
{
    unsigned short a,b,c,d;
    a=(x&AE)>>4
    b=(x&BE)>>8
    c=(x&CE)<<8
    d=(x&DE)<<5
    return (a|b|c|d);
}
```

## 【例2.27】计算ASCII码字符的奇偶校验位

二进制数据在传送中会发生**误码**（1变成0 或 0变成1），于是就有如何发现并纠正误码的问题。

**解决方法：**在原始数据 (数码位) 基础上增加几位校验 (冗余) 位

**奇偶校验：**分奇校验和偶校验。

**奇校验：**整个码字中奇数个“1”

**偶校验：**整个码字中偶数个“1”

data='A'

	1	0	0	0	0	0	1
--	---	---	---	---	---	---	---

校验位 parity

校验位在每个字符的最高位

若 $a_0 \wedge a_1 \wedge a_2 \wedge \dots \wedge a_n = 1$ ，则data中有奇数个1, 否则偶数个1（据此计算奇偶校验位）



```

int main()
{
    unsigned char data,backup,t;
    int parity=0; /* 设置奇偶校验, 0-偶校验, 1-奇校验 */
    printf("please input a char:\n");
    data=getchar(); /* 被校验的数据 */
    backup=data;
    while (data) {
        t=data&1; /* 取该数的最低位 */
        parity^=t; /* 进行异或操作 */
        data>>=1; /* 为取下一位做准备 */
    } // 计算校验位填充值
    printf("The parity is %#x\n",parity);
    data=backup|(parity<<7); /* 产生8位校验码 */
    printf("The data is %#x\n",backup);
    printf("Parity-Check Code is %#x\n",data); /* 输出校验码 */
    return 0;
}

```

```

please input a char:
1
The parity is 0x1
The data is 0x31
Parity-Check Code is 0xb1

```

```

please input a char:
3
The parity is 0
The data is 0x33
Parity-Check Code is 0x33

```



## 【例2.26】打印整数各位的软件工具

/\* 按位（二进制）显示 int \*/

#include<limits.h>

**void bit\_print(int x)**

{

int i;

int n=**sizeof**(int) \* CHAR\_BIT; /\* CHAR\_BIT 在limits.h中定义 \*/

int mask = 1 << (n-1); /\* 逻辑尺mask=100.....0 \*/

for ( i=1; i<=n; ++i ) {

putchar ( ! ( x & mask ) ? '0': '1');

**x<<=1;**

if ( ! ( i % CHAR\_BIT ) && i<n ) putchar(' '); // 每字节间输出空格

}

}



# 自主练习

- 1、练习例3.8, 3.12, 掌握short、long、double类型数据的输入/出方法。
- 2、编写压缩存储日、月和年的函数，再编写相应的解压函数，在main中测试这两个函数的功能。

```
short pack(short day, short month, short year);  
void unpack(short x);
```

- 3、自学例2.25, 并编写该加密函数

```
short encrypt(short x); /*形参是明文, 返回密文*/
```

- 5、研读例2.27, 用本章后习题2.24的方法重写

- 5、研读例2.26, 仿写: 定义函数hex\_print(int x)以十六进制输出int数, 即模拟实现 printf ("%x", x)。

在main中验证: 分别调用hex\_print和printf以十六进制输出一个int数。

C语言允许双精度、单精度、整型及字符数据之间混合运算

$10 + '1' + 6.5$

但有一个规则：先转换成同一类型, 再计算。

- 当出现下列情况时发生自动类型转换（隐式类型转换）：
  - 表达式中有char和short类型操作数时，将引起整数提升；
  - 当双目运算符的两个操作数类型不相同时，将引起一般算术转换；
  - 当一个值赋予一个不同类型的变量时，将引起赋值转换；
  - 函数调用, 实参与形参类型不同时，引起函数调用转换。（后面章节介绍）
- 强类型转换（显示转换）
  - 将某个值强制转换为另一数据类型

# 自动类型转换：整数提升

- 任何表达式中的 `char`、`unsigned char`、`short` 和 `unsigned short` 都要先转换成 `int` 或 `unsigned`，如果原始类型的所有值可以用 `int` 表示，则转换成 `int`，否则转换成 `unsigned`，把这称为“整数提升”。

- `short a;`

`sizeof(a)=?`      `sizeof(-a)=?`  
2                              4

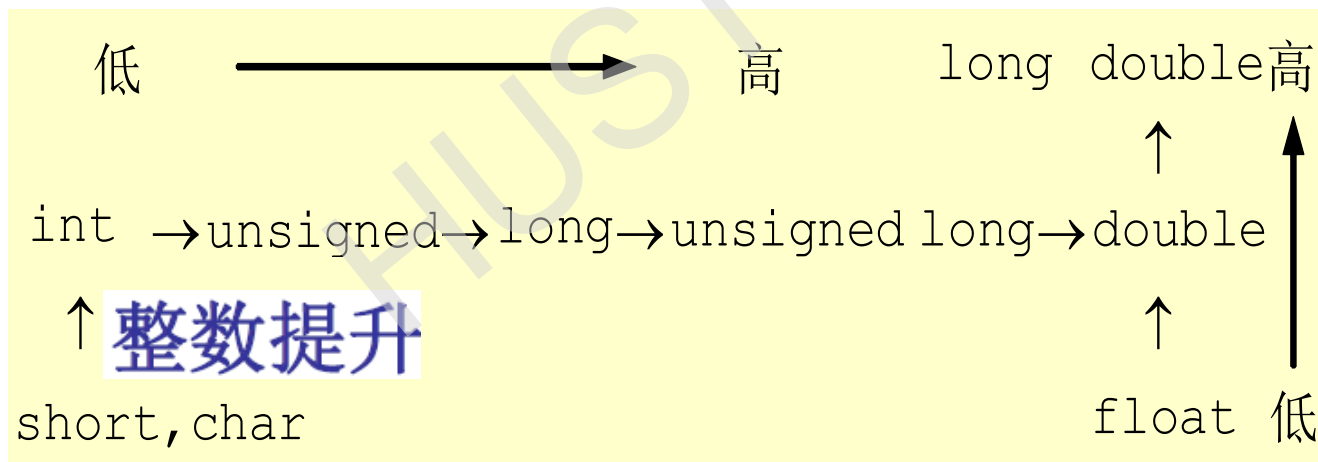
```
int b; short a;
```

```
printf("%d,%d,%d",sizeof(a),sizeof(b),sizeof(-a));
```

# 自动类型转换：算术转换

**转换规则：值域较窄的类型向值域较宽的类型转换**  
(“值域”就是数据类型所能表示的值的最大范围)

算术转换遵循的转换方向如图所示：



# 自动类型转换：赋值转换

- 右操作数的值被转换为左操作数的数据类型
- 例如：

```
short s=5; double d=2.9; long l=0x11118001;
```

则

```
s = d;    /*把d转换为short，再赋给s。值为2*/
```

```
d = s;    /*把s转换为double，再赋给d。值为5.0*/
```

```
s = l;    /* s值为 ? */  低位照搬
```

```
printf("%hx %hd %hu\n",s,s,s);
```

```
// 输出  8001  -32767  32769  
          -215-1  215+1
```

# 强制类型转换（显示转换）

（类型名）操作数

例如，

**(double) i** /\* 将i转换成double，i的类型保持不变\*/

**(long)('a'-32)** /\* 'a' 被自动转换成int，  
相减的结果被强制转换为long\*/

**(float)x+y** /\* 等价于( (float)x)+y\*/

**(double)x=10** /\* 错误\*/

## 类型变换小结：

无论是自动类型转换还是强制类型转换，都只是将变量或常量的值的类型进行暂时的转换，用于参与运算和操作，而变量和常量本身的类型和数值并没有改变。





## 2.8 枚举类型

- 使代码更具可读性，理解清晰，易于维护。
- 它是由用户定义的**若干枚举常量的集合**
- 枚举常量**用标识符命名**，缺省状态下，其值是所列举元素的序号，**序号从0开始**

```
enum color { WHITE,YELLOW,RED,BLUE};
```

```
enum week { SUN, MON, TUE, WED, THU, FRI, SAT };
```

# 枚举类型的定义

```
enum week { SUN, MON, TUE, WED, THU, FRI, SAT };
```

↑ 关键字      ↑ 枚举名      枚举常量

- 在未指定值的缺省情况下，第一个枚举常量的值为0，以后的值依次递增1。
- 可以指定一个或多个枚举常量的值，未指定值的枚举常量的值比前面的值大1。

```
enum sizes { SMALL, MEDIUM=10, BIG, TOO_BIG=20 };
```

enum 后面也可以不出现枚举名：

```
enum { WIN, LOSE, TIE, ERROR};
```



# 用枚举类型定义符号常量

```
#define WIN 0
#define LOSE 1
#define TIE 2
#define ERROR -1
```

- 可用下面的枚举类型定义来代替:

```
enum { WIN, LOSE, TIE, ERROR=-1};
```

给一组相关联的整型常量命名类型



# 枚举变量的声明

两种方式:

(1) 在定义枚举类型的同时说明枚举变量

```
enum color { RED, GREEN, BLUE} c1, c2;
```

(2) 利用枚举名来说明枚举变量

```
enum color { RED, GREEN, BLUE} c1;
```

```
enum color c2;
```

或者

```
enum color { RED, GREEN, BLUE};
```

```
enum color c1, c2;
```

# 枚举变量的输入和输出

```
enum color { RED, GREEN, BLUE } c1, c2;
```

- 一个枚举变量的值是int型整数，其值域仅限于列举的范围，枚举变量值的输入和输出都只能是整数。

```
c1=BLUE;    /* 等价于 c1=2; 使用枚举常量有助于读者理解程序 */
```

```
printf("%d", c1); /* 输出2，而不是BLUE */
```

```
scanf("%d", &c2); /* 可输入0、1、2，不能输入RED等 */
```

```
if(c1==RED) printf("Red" );
```

下面的语句是错误的：

```
c1=3;    /* 变量c1的值域为：0、1和2 */
```

```
printf("%s", GREEN ); // 输出错误的结果，而不是GREEN
```



# 输出对应的英文名

- 一个枚举变量的值是一个**int**整数，它可以出现在整数允许出现的任何地方。如果要输出与枚举值相对应的枚举常量标识符或代表其含义的完整字符串，可以定义一个**字符串数组**，以枚举值作为下标。
- **enum color { RED, GREEN, BLUE};**  
**enum color c1;**  
**char \*colorName[ ]={ "Red", "Green", "Blue"};**  
**c1 = BLUE;**  
**printf("%s",colorName[c1] );**
- 自学例2.30