第三十四节：移位运算的左移。

【34.1　“左移”运算。】

“左移”运算也是以位为单位进行运算的。位是指二进制中的某一位，位只能是 0 或者 1。欲理解某个数“左移”运算的内部规律，必先把该数展开成二进制的格式，然后才好分析。“左移”运算的符号是“<<”，它的通用格式如下：

```
“保存变量”=“被移数”<<n;
```

运算规律是：“被移数”先被复制一份放到某个隐蔽的临时变量（也称作寄存器），然后对此临时变量展开成二进制的格式，左边是高位，右边是低位，此二进制格式的临时变量被整体由右往左移动了 n 位，原来左边的高 n 位数据被直接覆盖，而右边由于数据位移动而新空出的低 n 位数据被直接填入 0，最后再把移位运算的结果存入“保存变量”。多问一句，这行代码执行完毕后，“保存变量”和“被移数”到底哪个变量发生了变化，哪个变量维持不变？大家记住，只有赋值语句“=”左边的“保存变量”发生数值变化，而右边的“被移数”没有发生变化，因为“被移数”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称寄存器）。这条规律对“加、减、乘、除、与、或、异或、非、取反”等运算都是适用的，重要的事情再重复一次，这条规律就是：只有赋值语句“=”左边的“保存变量”发生数值变化，而赋值语句“=”右边的“运算变量”本身不会发生变化，因为“运算变量”被操作的不是它自己本身，而是它的复制品替身（某个隐蔽的临时变量，也称寄存器）。

上述通用格式中的 n 代表被一次左移的位数，可以取 0，当 n 等于 0 的时候，代表左移 0 位，其实就是数值维持原来的样子没有发生变化。

现在举一个完整的例子来分析“<<”运算的规律。有两个 unsigned char 类型的变量 a 和 b，它们的数值都是十进制的 5，求 a=a<<1 和 b=b<<2 的结果分别是多少？分析步骤如下：

第一步：先把 a 和 b 变量原来的数值以二进制的格式展开。十进制转二进制的方法请参考前面第 14,15,16 节的内容。

```
a 变量是十进制 5，它的二进制格式是：  00000101。
b 变量是十进制 5，它的二进制格式是：  00000101。
```

第二步：将 a 左移 1 位，将 b 左移 2 位。

（1）a=a<<1，就是将 a 左移 1 位。

```
a 左移前是        ->    00000101
a 左移 1 位后是   ->    00001010
```

结果分析：把二进制的 00001010 转换成十六进制是：0x0A。转换成十进制是 10。所以 a 初始值是 5,左移 1 位后的结果是 10。

（2）b=b<<2，就是将 b 左移 2 位。

```
b 左移前是        ->    00000101
b 左移 2 位后是   ->    00010100
```

结果分析：把二进制的 00010100 转换成十六进制是：0x14。转换成十进制是 20。所以 b 初始值是 5,左移 2 位后的结果是 20。

【34.2　“左移”与乘法的关系。】

上面的例子，仔细观察，发现一个规律：5 左移 1 位就变成了 10（相当于 5 乘以 2），5 左移 2 位就变

成了 20（相当于 5 乘以 2 再乘以 2）。这个现象背后的规律是：在左移运算中，只要最高位不发生溢出的现象，那么每左移 1 位就相当于乘以 2，左移 2 位相当于乘以 2 再乘以 2，左移 3 位相当于乘以 2 再乘以 2 再乘以 2.....以此类推。这个规律反过来从乘法的角度看，也是成立的：某个数乘以 2，就相当于左移 1 位，某个数乘以 2 再乘以 2 相当于左移 2 位，某个数乘以 2 再乘以 2 再乘以 2 相当于左移 3 位.....以此类推。那么问题来了，同样是达到乘以 2 的运算结果，从运算速度的角度对比，“左移”和“乘法”哪家强？答案是：一条左移语句的运算速度比一条乘法语句的运算速度要快很多倍。

【34.3 “左移”的常见应用之一：不同数据类型之间的合并。】

比如有两个 unsigned char 单字节的类型数据 H 和 L，H 的初始值是十六进制的 0x12，L 的初始值是十六进制的 0x34，要将两个单字节的 H 和 L 合并成一个 unsigned int 双字节的数据 c，其中 H 是高 8 位字节，L 是低八位字节，合并成 c 后，c 的值应该是十六进制的 0x1234，此程序如何写？就需要用到左移。程序分析如下：

```
unsigned char H=0x12;  //单字节
unsigned char L=0x34;  //单字节
unsigned int c;        //双字节
c=H;                   //c 的低 8 位被 H 覆盖，也就是 c 的低 8 位得到了 H 的值。
c=c<<8;                //及时把 c 的低 8 位移动到高 8 位，同时 c 原来的低 8 位被填入 0
c=c+L;                 //此时 c 再加 L，c 的低 8 位就 L 的值。
```

程序运行结果:c 就等于十六进制的 0x1234，十进制是 4660。

【34.4 “左移”的常见应用之二：聚焦在某个变量的某个位。】

前面第 31 节讲到“或”运算，其中讲到可以对某个变量的某个位置 1，当时是这样讲的，片段如下：

“或”运算最常见的用途是可以指定一个变量的某位置 1，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位置 1，其它位保持不变，只需跟十六进制的 0x01 相“或”:b=b|0x01。
想让第 1 位置 1，其它位保持不变，只需跟十六进制的 0x02 相“或”:b=b|0x02。
想让第 2 位置 1，其它位保持不变，只需跟十六进制的 0x04 相“或”:b=b|0x04。
想让第 3 位置 1，其它位保持不变，只需跟十六进制的 0x08 相“或”:b=b|0x08。
想让第 4 位置 1，其它位保持不变，只需跟十六进制的 0x10 相“或”:b=b|0x10。
想让第 5 位置 1，其它位保持不变，只需跟十六进制的 0x20 相“或”:b=b|0x20。
想让第 6 位置 1，其它位保持不变，只需跟十六进制的 0x40 相“或”:b=b|0x40。
想让第 7 位置 1，其它位保持不变，只需跟十六进制的 0x80 相“或”:b=b|0x80。

但是这样写很多程序员会嫌它不直观，哪里不直观？就是 0x01，0x02，0x04，0x08，0x10，0x20，0x40，0x80 这些数不直观,这些数只是代表了聚焦某个变量不同的位。如果把这些十六进制的数值换成左移的写法，在阅读上就非常清晰直观了。比如：0x01 可以用 1<<0 替代，0x02 可以用 1<<1 替代，0x04 可以用 1<<2 替代.....0x80 可以用 1<<7 替代。左移的 n 位，n 就恰好代表了某个变量的某个位。于是，我们把上面的片段更改成左移的写法后，如下：

“或”运算最常见的用途是可以指定一个变量的某位置 1，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：

想让第 0 位置 1，其它位保持不变，只需:b=b|(1<<0)。
想让第 1 位置 1，其它位保持不变，只需:b=b|(1<<1)。
想让第 2 位置 1，其它位保持不变，只需:b=b|(1<<2)。

想让第 3 位置 1，其它位保持不变，只需:b=b|(1<<3)。
想让第 4 位置 1，其它位保持不变，只需:b=b|(1<<4)。
想让第 5 位置 1，其它位保持不变，只需:b=b|(1<<5)。
想让第 6 位置 1，其它位保持不变，只需:b=b|(1<<6)。
想让第 7 位置 1，其它位保持不变，只需:b=b|(1<<7)。

分析：这样改进后，阅读就很清晰直观了，只是在程序代码的效率速度方面，因为多增加了一条左移指令，意味着要多消耗一条指令的时间，那么到底该选择哪种？其实各有利弊，应该根据个人的编程喜好和实际项目来取舍。很多 32 位的单片机在初始化寄存器的库函数里大量应用这种左移的方法来操作，目的就是为了增加代码可读性。

根据上述规律，假设 d 原来等于十进制的 84（十六进制是 0x54，二进制是 01010100），要想把此数据的第 0 位置 1，只需 d=d|(1<<0)。最终 d 的运算结果是十进制是 85（十六进制是 0x55，二进制是 01010101）。

刚才上面讲到第 31 节的“或”运算，其实在第 30 节的“与”运算中也是可以用这种左移的方法来聚焦，只是要多配合一条“取反”的指令才可以。“与”运算跟“或”运算刚刚相反，它是对某个变量的某个位清零，当时是这样讲的，片段如下：

“与”运算最常见的用途是可以指定一个变量二进制格式的某位清零，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：
想让第 0 位清零，其它位保持不变，只需跟十六进制的 0xfe 相“与”:b=b&0xfe。
想让第 1 位清零，其它位保持不变，只需跟十六进制的 0xfd 相“与”:b=b&0xfd。
想让第 2 位清零，其它位保持不变，只需跟十六进制的 0xfb 相“与”:b=b&0xfb。
想让第 3 位清零，其它位保持不变，只需跟十六进制的 0xf7 相“与”:b=b&0xf7。
想让第 4 位清零，其它位保持不变，只需跟十六进制的 0xef 相“与”:b=b&0xef。
想让第 5 位清零，其它位保持不变，只需跟十六进制的 0xdf 相“与”:b=b&0xdf。
想让第 6 位清零，其它位保持不变，只需跟十六进制的 0xbf 相“与”:b=b&0xbf。
想让第 7 位清零，其它位保持不变，只需跟十六进制的 0x7f 相“与”:b=b&0x7f。

但是这样写很多程序员会嫌它不直观，哪里不直观？就是 0xfe，0xfd，0xfb，0xf7，0xef，0xdf，0xbf，0x7f 这些数不直观,这些数只是代表了聚焦某个变量不同的位。如果把这些十六进制的数值换成左移的写法，在阅读上就非常清晰直观了，但是注意，这里左移之后还要配一条“取反”语句。比如：0xfe 可以用~(1<<0)替代，0xfd 可以用~(1<<1)替代，0xfb 可以用~(1<<2)替代.....0x7f 可以用~(1<<7)替代。左移的 n 位后再取反，n 就恰好代表了某个变量的某个位。于是，我们把上面的片段更改成左移的写法后，如下：

“与”运算最常见的用途是可以指定一个变量二进制格式的某位清零，其它位保持不变。比如一个 unsigned char 类型的变量 b，数据长度一共是 8 位，从右往左：
想让第 0 位清零，其它位保持不变，只需:b=b&(~(1<<0))。
想让第 1 位清零，其它位保持不变，只需:b=b&(~(1<<1))。
想让第 2 位清零，其它位保持不变，只需:b=b&(~(1<<2))。
想让第 3 位清零，其它位保持不变，只需:b=b&(~(1<<3))。
想让第 4 位清零，其它位保持不变，只需:b=b&(~(1<<4))。
想让第 5 位清零，其它位保持不变，只需:b=b&(~(1<<5))。
想让第 6 位清零，其它位保持不变，只需:b=b&(~(1<<6))。
想让第 7 位清零，其它位保持不变，只需:b=b&(~(1<<7))。

分析：这样改进后，阅读就很清晰直观了，只是在程序代码的效率速度方面，因为多增加了一条左移指令和一条取反指令，意味着要多消耗两条指令的时间，那么到底该选择哪种？其实各有利弊，应该根据个人的编程喜好和实际项目来取舍。很多 32 位的单片机在初始化寄存器的库函数里大量应用这种左移的方法来

操作，目的就是为了增加代码可读性。

根据上述规律，假设 e 原来等于十进制的 85（十六进制是 0x55，二进制是 01010101），要想把此数据的第 0 位清零，只需 e=e&(~(1<<0))。最终 e 的运算结果是十进制是 84(十六进制是 0x54，二进制是 01010100)。

【34.5 左移运算的“左移简写”。】

当被移数是“保存变量”时，存在“左移简写”。

```
“保存变量”=“保存变量”<<n;
```

上述左移简写如下：

```
“保存变量”<<=n;
```

比如：

```
unsigned char f=1;
unsigned char g=1;

f<<=1; //就相当于 f=f<<1;
g<<=2; //就相当于 g=g<<2;
```

【34.6 例程练习和分析。】

现在编写一个程序来验证刚才讲到的“左移”运算：

程序代码如下：

```
/*---C 语言学习区域的开始。-----------------------------------------------*/

void main() //主函数
{
     unsigned char a=5;
     unsigned char b=5;

     unsigned char H=0x12; //单字节
     unsigned char L=0x34; //单字节
     unsigned int c;       //双字节

     unsigned char d=84;
     unsigned char e=85;

     unsigned char f=1;
     unsigned char g=1;

     //左移运算中蕴含着乘 2 的规律。
     a=a<<1; //a 左移 1 位，相当于 a=a*2，从原来的 5 变成了 10。
     b=b<<2; //b 左移 2 位，相当于 b=b*2*2，从原来的 5 变成了 20。

     //左移的应用之一：不同变量类型的合并。
```

```
        c=H;     //c 的低 8 位被 H 覆盖，也就是此时 c 的低 8 位得到了 H 的各位值。
        c=c<<8; //及时把 c 的低 8 位移动到高 8 位，同时 c 原来的低 8 位被填入 0
        c=c+L;  //此时 c 再加 L，c 的低 8 位就 L 的值。此时 c 得到了 H 和 L 合并而来的值。

        //左移的应用之二：聚焦在某个变量的某个位。
        d=d|(1<<0);       //对第 0 位置 1。
        e=e&(~(1<<0));    //对第 0 位清零。

        //左移简写。
        f<<=1;  //就相当于 f=f<<1;
        g<<=2;  //就相当于 g=g<<2;

        View(a);              //把第 1 个数 a 发送到电脑端的串口助手软件上观察。
        View(b);              //把第 2 个数 b 发送到电脑端的串口助手软件上观察。
        View(c);              //把第 3 个数 c 发送到电脑端的串口助手软件上观察。
        View(d);              //把第 4 个数 d 发送到电脑端的串口助手软件上观察。
        View(e);              //把第 5 个数 e 发送到电脑端的串口助手软件上观察。
        View(f);              //把第 6 个数 f 发送到电脑端的串口助手软件上观察。
        View(g);              //把第 7 个数 g 发送到电脑端的串口助手软件上观察。

        while(1)
        {
        }
    }

    /*---C 语言学习区域的结束。-----------------------------------------------*/
```

在电脑串口助手软件上观察到的程序执行现象如下：

```
开始...

第 1 个数
十进制:10
十六进制:A
二进制:1010

第 2 个数
十进制:20
十六进制:14
二进制:10100

第 3 个数
十进制:4660
十六进制:1234
```

```
二进制:1001000110100

第 4 个数
十进制:85
十六进制:55
二进制:1010101

第 5 个数
十进制:84
十六进制:54
二进制:1010100

第 6 个数
十进制:2
十六进制:2
二进制:10

第 7 个数
十进制:4
十六进制:4
二进制:100
```

分析：

通过实验结果，发现在单片机上的计算结果和我们的分析是一致的。

【34.7　如何在单片机上练习本章节 C 语言程序？】

直接复制前面章节中第十一节的模板程序，练习代码时只需要更改“C 语言学习区域”的代码就可以了，其它部分的代码不要动。编译后，把程序下载进带串口的 51 学习板，通过电脑端的串口助手软件就可以观察到不同的变量数值，详细方法请看第十一节内容。