一. 为什么我们要学位操作

C语言能在汇编语言所保留的级别上工作，这使其成为编写**设备驱动程序**和**嵌入式代码**的首选语言。

1. C可以通过像硬件设备**发送一两个字节**来控制这些设备，其中每个位都有特定的含义
2. 与文件相关的操作系统信息经常被储存，通过使用特定位表明特定项。
3. 许多压缩和加密工作都是直接处理单独的位。

二. 计算机底层基本数据的表示

1. C语言使用**字节**来表示系统字符集所需的大小。注意描述存储器芯片和数据传输率中所用的字节指的是8位字节（可以在limits.h中的CHAR_BIT中查看）
2. 如果使用的是8位一字节，那么一个字节可以表示256个数。
3. 浮点数是怎么表示的：浮点数分两部分组成：mantissa和exponent（其实还有符号）
4. 浮点数是怎么参与运算的：

①如果乘以一个2的幂的数，mantissa不变，exponent加一。

②如果乘以一个不是2的幂的数，会改变mantissa，同时exponent有需要也会改变

（其他的，如八进制，十六进制，反码和补码，实在是过于简单，这里不提了）

三. 按位运算符

1. 按位取反：~

把底层储存的二进制数全部按位取反。

2. 按位与：&

1. 表示方法：val = val & 0377 或 val &= 0377　　// 注意这里0377代表的是八进制数
2. 用法：按位与的用法是最为广泛的。

①mask: 设置MASK，用0隐藏与之运算数的对应的位，用1显示与之运算的对应的位。　　②关闭特定的位（把指定的位设为**可见**，未指定的**隐藏**）：利用掩码

```
flag &= ~MASK
```

③检查位的值：仍然采用掩码，将要比较的位设置为**可见**，其他位统统**隐藏**。

```
if ((flag & MASK) == MASK) puts("HelloWorld!");
```

3. 按位或：|

1. 表示方法： val = val | 0377 或 val &= 0377

2. 用法：打开位——把想要打开的位设置为1，其他的都不管。掩码中1表示要打开，0表示不管

flag |= MASK

4. 按位异或： ^

1. 表示方法： val = val ^ 0377 或 val ^= 0377
2. 用法：切换位——利用掩码，掩码中1表示要切换，0表示不用切换。
3. 还有一种技巧，不引入第三个变量交换数据（字符也行）：

a ^= b ^= a ^= b;　　// 证明，对于每个位一共有4种情况，穷举即可。

四. 移位运算符

1. 左移:<<

1. 表示： a = b << 2 或 a <<= 2;
2. 作用效果：将其左侧运算对象每一位的值都向左移动其右侧运算对象指定的位数。（**移出左末端位的值丢失，用0填充空出的位置。这说明位的长度很重要，下面有例子**）
3. 效果：对于针对2的幂（前提是这个数得小，所以没卵用）提供快速有效的乘法。

number << n　　// number乘以2的n次幂

2. 右移:>>

1. 表示： a = b >> 2 或 a <<= 2;
2. 作用效果：左侧运算对象移出，右末端位的值丢。

对于无符号类型，用0填充空出的位置。对于有符号类型，取决于机器。

3. 效果：同；对于针对2的幂提供快速有效的乘法。

3.实例

1. 计算某个整数的二进制

```
// 注意，这里n是进行操作的数，作为一个临时变量使用
const static int size = CHAR_BIT * sizeof(int);
for (int i = size - 1; i >= 0; i--, n >>= 1) ps[i] = (01 & n) + '0';
ps[size] = '\0';
```

这里用到的技巧：

①直接操作原始的，存在内存中的二进制格式。

②利用和01（八进制数，为了和计算机中表达方式更一致）进行与操作的形式来提取相应的位。　　③每个元素后面加一个'0',让其变成一个字符类型的数。

④最后在数组的末尾添加了一个空字符'\0'(区别'0'和'\0'！)，使其成为一个字符串。

2.获取一个字节的后n位

这种办法我们需要获得特定的掩码，可惜C并不原生支持二进制数（不然也不需要我们用十进制数转化），但我们可以自己利用位操作搞一个

```
int mask = 0; // 这里mask的位数是0
int bitval = 1;

// 这里的bits代表有几位
while (bits-- > 0) {
 mask |= bitval; // 通过|运算让mask的下一位变成1
 bitval <<= 1; // 这里bitval往左移1位，代表处理下一位
}
```

3. 提取较大单元中的一些位，比如一个unsigned long 类型表示颜色值，低阶位字节储存红色的强度，下一字节储存绿色的强度，第三个字节储存蓝色的强度。比如0x002a162f，那我们怎么来提取这几个字节中的内容呢?

```
// 通过设置0xff可以来提取一个字节的内容
// (不知道define后面的行注释或不会被忽略)
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

五. 位字段

1. 存在的意义：用全长（full-size）类型有时候可能会比较浪费内存，比如储存8种状态，那么用一个int就太浪费了，这就有了位字段的产生。

2. 使用方法：

struct {
unsigned int a : 1 ;　　// 占unsigned int中的前1位
unsigned int b : 2 ;　　// 占unsigned int中随后的2位
unsigned int　: 2 ;　　// 这里使用了未命名的字段宽度，产生了一个“洞”
bool c : 1 ;
} stuff;

3. 注意事项：

①所附的值不超出字段可容纳的范围。

②如果声明的总位数超出了unsigned int类型的大小，**编译器会自动移动跨界的字段，保持unsigned int的边界对齐**，不过这样也会在unsigned int末尾的位中留下

一个未命名的洞。　　③注意C以unsigned int和_Bool（引入stdbool后就可以使用bool了）作为位字段结构的基本布局单元。

六. 对齐特性（C11）

1. 什么是对齐？对齐指的是如何安排对象在内存中的位置（坦白地讲就是设置不同类型所占的内存）

2. _Alignof(*type*):类似于sizeof的运算符，返回**储存该类型值相邻地址的字节数。**

size_t d_align = _Alignof(float);

3. _Alignas:关键字，用于声明**储存该类型值相邻地址的字节数。**

_Alignas(8) char c2;　　// 理论上讲，你所声明的字节数应该大于原生字节数

4. 相关的头文件：stdalign.h

作用：可以把alignas和alignof作为_Alignas和_Alignof的别名。可以与C++兼容。