

## 2.56

试着用不同的示例值来运行 `show_bytes` 的代码。

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

```
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, size_t len);
void show_int(int x);
void show_float(float x);
int main() {
    int i=1;
    show_int(i);
    float f = 3.1415926;
    show_float(f);
    return 0;
}
void show_bytes(byte_pointer start, size_t len){
    size_t i;
    for(i=0;i<len;i++){
        printf(" %.2x",start[i]);
        printf("\n");
    }
}
void show_int(int x){
    show_bytes((byte_pointer)&x,sizeof(x));
}
void show_float(float x){
    show_bytes((byte_pointer)&x,sizeof(x));
}
```

```
01 00 00 00
da 0f 49 40
```

## 2.60

假设我们将一个  $w$  位的字中的字节从 0(最低位)到  $w/8-1$ (最高位)编号。写出下面 C 函数的代码，它会返回一个无符号值，其中参数  $x$  的字节  $i$  被替换成字节  $b$ ：

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
```

以下示例，说明了这个函数该如何工作：

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

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

```
unsigned replace_byte (unsigned x, int i, unsigned char b);
int main() {
    printf("%x\n", replace_byte(0x12345678,2,0xAB));
```

```
123456ab
12ab5678
```

```

        printf("%x\n", replace_byte(0x12345678,0,0xAB));
        return 0;
    }
    //小端机有效
    unsigned replace_byte (unsigned x, int i, unsigned char b){
        int a = x;
        unsigned char * p = &a;
        p[i]=b;
        return a;
    }

```

## 2.64

写出代码实现如下函数：

```

/* Return 1 when any odd bit of x equals 1; 0 otherwise.
   Assume w=32 */
int any_odd_one(unsigned x);

```

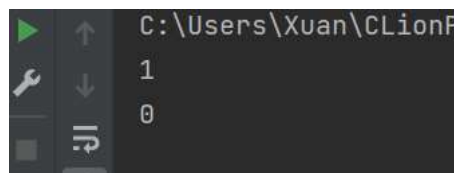
函数应该遵循位级整数编码规则，不过你可以假设数据类型 `int` 有 `w=32` 位。

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

```

int any_odd_one(unsigned x);
int main() {
    unsigned a = 0x3;
    printf("%d\n",any_odd_one(a));
    unsigned b = 0x4;
    printf("%d\n",any_odd_one(b));
    return 0;
    return 0;
}
/*Return 1 when any odd bit of x equals 1;0 otherwise
 * Assume w=32
 * */
int any_odd_one(unsigned x){
    return !(x & 0xaaaaaaaa);
}

```



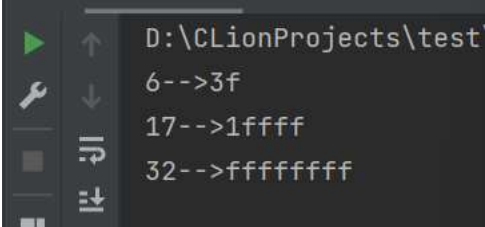
## 2.68

写出具有如下原型的函数的代码：

```
/*
 * Mask with least significant n bits set to 1
 * Examples: n = 6 --> 0x3F, n = 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n);
```

函数应该遵循位级整数编码规则。要注意  $n=w$  的情况。

```
#include <stdio.h>
int lower_one_mask(int n);
int main() {
    printf("%d-->%x\n", 6, lower_one_mask(6));
    printf("%d-->%x\n", 17, lower_one_mask(17));
    printf("%d-->%x\n", 32, lower_one_mask(32));
    return 0;
}
/**
 * Mask with least significant n bits set to 1
 * Examples: n = 6 --> 0x3F, n = 17 --> 0x1ffff
 * Assume 1<= n <=w
 */
int lower_one_mask(int n){
    int w = sizeof(int) << 3;
    return (unsigned)-1 >> (w - n);
}
```



```
D:\CLionProjects\test
6-->3f
17-->1ffff
32-->ffffffff
```

## 2.72

给你一个任务，写一个函数，将整数 `val` 复制到缓冲区 `buf` 中，但是只有当缓冲区中有足够可用的空间时，才执行复制。

你写的代码如下：

```
/* Copy integer into buffer if space is available */
/* WARNING: The following code is buggy */
void copy_int(int val, void *buf, int maxbytes) {
    if (maxbytes-sizeof(val) >= 0)
        memcpy(buf, (void *) &val, sizeof(val));
}
```

这段代码使用了库函数 `memcpy`。虽然在这里用这个函数有点刻意，因为我们只是想复制一个 `int`，但是它说明了一种复制较大数据结构的常见方法。

你仔细地测试了这段代码后发现，哪怕 `maxbytes` 很小的时候，它也能把值复制到缓冲区中。

A. 解释为什么代码中的条件测试总是成功。提示：`sizeof` 运算符返回类型为 `size_t` 的值。

B. 你该如何重写这个条件测试，使之工作正确。

A. `sizeof()` 的结果是一个 `unsigned int`，当计算 `maxbytes-sizeof(val) >= 0` 时，若 `maxbytes`

是一个小于等于 sizeof(val) 的值，那么最后计算的值将是一个很大的 unsigned int，将会执行 if 中的语句，但是在该情况下，本不应该执行。

B. 将函数参数中的 maxbytes 改为 unsigned 类型

## 2.76

库函数 calloc 有如下声明：

```
void *calloc(size_t nmemb, size_t size);
```

根据库文档：“函数 calloc 为一个数组分配内存，该数组有 nmemb 个元素，每个元素为 size 字节。内存设置为 0。如果 nmemb 或 size 为 0，则 calloc 返回 NULL。”

编写 calloc 的实现，通过调用 malloc 执行分配，调用 memset 将内存设置为 0。你的代码应该没有任何由算术溢出引起的漏洞，且无论数据类型 size\_t 用多少位表示，代码都应该正常工作。

作为参考，函数 malloc 和 memset 声明如下：

```
void *malloc(size_t size);
```

```
void *memset(void *s, int c, size_t n);
```

```
void *mycalloc(size_t nmemb, size_t size){
    if(nmemb==0||size==0)return NULL;
    else{
        size_t size1 = nmemb*size;
        if(size1/nmemb==size){
            void* p =malloc(size1);
            if(p!=NULL)
                memset(p,0,size1);
            return p;
        }
    }
}
```

## 2.80

写出函数 threefourths 的代码，对于整数参数 x，计算  $3/4x$  的值，向零舍入。它不会溢出。函数应该遵循位级整数编码规则。

编写 C 表达式产生如下位模式。其中  $a^k$  表示符号  $a$  重复  $k$  次。假设一个  $m$  位的数据类型。代码可

```
int threefourths(int x){
    //判断 x 正负
    int neg_flag = x & INT_MIN;
    //取 x 的前 30 位
    int m30 = x & ~0x3;
    //取 x 的后 2 位
    int l2 = x & 0x3;
    //计算 m30 除以 4 乘以 3
    int m30d4m3 = ((m30 >> 2) << 1) + (m30 >> 2);
    int bias = 3;
    //计算 l2 乘以 3
```

```

int l2m3 = (l2 << 1) + l2;
//如果 x 为负
if(neg_flag)l2m3 = l2m3 + bias;
//计算 l2 乘以 3 除以 4
int l2m3d4 = l2m3 >> 2;

return m30d4m3 + l2m3d4;
}

```

## 2.84

填写下列程序的返回值，这个程序测试它的第一个参数是否小于或者等于第二个参数。假定函数 `f2u` 返回一个无符号 32 位数字，其位表示与它的浮点参数相同。你可以假设两个参数都不是 *NaN*。两种 0，+0 和 -0 被认为是相等的。

```

int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);

    /* Get the sign bits */
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    /* Give an expression using only ux, uy, sx, and sy */
    return
}

return (ux << 1 == 0 && uy << 1 == 0) ||      /* x = y = 0 */

        (sx && !sy) ||      /* x<0 , y>0 */

        (sx && sy && ux >= uy) ||

        (!sx && !sy && ux <= uy);

```

2.88

2.88 考虑下面两个基于 IEEE 浮点格式的 9 位浮点表示。

1. 格式 A

- 有一个符号位。
- 有  $k=5$  个阶码位。阶码偏置量是 15。
- 有  $n=3$  个小数位。

1~30      -14~15

2. 格式 B

- 有一个符号位。
- 有  $k=4$  个阶码位。阶码偏置量是 7。

1~14      -6~7

- 有  $n=4$  个小数位。

下面给出了一些格式 A 表示的位模式，你的任务是把它们转换成最接近的格式 B 表示的值。如果需要舍入，你要向  $+\infty$  舍入。另外，给出用格式 A 和格式 B 表示的位模式对应的值。要么是整数（例如 17），要么是分数（例如  $17/64$  或  $17/2^6$ ）。

格式A		格式B	
位	值	位	值
1 01110 001	$-\frac{9}{16}$	1 0110 0010	$-\frac{9}{16}$
0 10110 101	$13 \cdot 2^4$	0 1110 1010	$13 \cdot 2^4$
1 00111 110	$-7 \cdot 2^{10}$	1 0000 0111	$-7 \cdot 2^{10}$
0 00000 101	$5 \cdot 2^{-14}$	0 0000 0001	$1 \cdot 2^{-6}$
1 11011 000	$-2^{12}$	1 1110 1111	$-13 \cdot 2^4$
0 11000 100	$3 \cdot 2^9$	0 1111 0000	$+\infty$

2.92

遵循位级浮点编码规则，实现具有如下原型的函数：

```
/* Compute -f. If f is NaN, then return f. */
float_bits float_negate(float_bits f);
```

对于浮点数  $f$ ，这个函数计算  $-f$ 。如果  $f$  是 NaN，你的函数应该简单地返回  $f$ 。  
测试你的函数，对参数  $f$  可以取的所有  $2^{32}$  个值求值，将结果与你使用机器的浮点运算得到的结果

相比较。

```
typedef unsigned float_bits;
```

```
float_bits float_negate(float_bits f){
    //符号位
    unsigned sign = f >> 31;
    //阶码
```

```
unsigned exp = f >> 23 & 0xFF;
//尾数
unsigned frac = f & 0x7FFFFFFF;

//是否 NaN
int is_nan = (exp == 0xFF) && (frac != 0);
if (is_nan) {
    return f;
}
return (~sign << 31) | (exp << 23) | frac;
}
```