Problem 2.58 Solution: 10 pts
There are many ways to solve this problem. The basic idea is to create some multibyte datum with different values for the most and least-significant bytes. We then read byte 0 and determine which byte it is.

The following solution creates an int with value 1. We then access its first byte and convert it to an int. This byte will equal 0 on a big-endian machine and 1 on a little-endian machine.

```c
1 int is_little_endian(void) {
2  /* MSB = 0, LSB = 1 */
3  int x = 1;
4
5  /* Return MSB when big-endian, LSB when little-endian */
6  return (int) (* (char *) &x);
7 }
```

Problem 2.61 Solution: 16 pts (2, 2, 4, 8)
These exercises require thinking about the logical operation ! in a nontraditional way. Normally we think of it as logical negation. More generally, it detects whether there is any nonzero bit in a word. In addition, it gives practice with masking.

A. !!x
B. !!~x
C. !!((x & 0xFF)
D. !!~x & ((sizeof(int)-1)<<3))

Problem 2.64 Solution: 10 pts
This problem is very simple, but it reinforces the idea of using different bit patterns as masks.

```c
1 /* Return 1 when any odd bit of x equals 1; 0 otherwise. Assume w=32 */
2 int any_odd_one(unsigned x) {
3  /* Use mask to select odd bits */
4  return (x&0xAAAAAAAA) != 0;
5 }
```

Problem 2.70 Solution: 20 pts
The code is as follows:

```c
1 /*
2 * Return 1 when x can be represented as an n-bit, 2’s complement number;
3 * 0 otherwise
4 * Assume 1 <= n <= w
5 */
6 int fits_bits(int x, int n) {
7  /*
8 * Use left shift then right shift
9 * to sign extend from n bits to full int
10 */
11 int count = (sizeof(int)<<3)-n;
12 int leftright = (x << count) >> count;
13 /* See if still have same value */
```
14 return x == leftright;
15 }

This code uses a common trick, demonstrated in Problem 2.23, of first shifting left by some amount k and then arithmetically shifting right by k. This has the effect of sign-extending from bit $w - k - 1$ leftward.

**Problem 2.72 Solution: 10 pts (5 pts each)**
This code illustrates the hidden dangers of data type size $t$, which is defined to be unsigned on most machines.
A. Since this one data value has type unsigned, the entire expression is evaluated according to the rules of unsigned arithmetic. As a result, the conditional expression will always succeed, since every value is greater or equal to 0.
B. The code can be corrected by rewriting the conditional test:
   if (maxbytes >= sizeof(val))

**Problem 2.76 Solution: 20 pts (4, 4, 6, 6)**
Patterns of the kind shown here frequently appear in compiled code.
A. $K = 17$: $(x\ll 4) + x$
B. $K = -7$: $-(x\ll 3) + x$
C. $K = 60$: $(x\ll 6) - (x\ll 2)$
D. $K = -112$: $-(x\ll 7) + (x\ll 4)$

**Problem 2.80 Solution: 14 pts (6, 8)**
Bit patterns similar to these arise in many applications. Many programmers provide them directly in hexadecimal, but it would be better if they could express them in more abstract ways.
A. $1 \ w - k \ 0^k$
   $\neg((1 \ll k) - 1)$
B. $0^{w-k-1} 1^k 0^j$
   $((1 \ll k) - 1) \ll j$