THE UNIVERSITY COLLEGE
OF THE CARIBOO

Computing Science – **COMP213**

Lab Exercises #9 – C – *Bit-Fiddling*

Solutions

Exercises / Programming Problems

1. C permits bit-manipulation and shifting with only integer based types. Why not floating-point types?

   Integer types represent binary numbers using a form that is similar to decimal, with each binary digit (bit) position presenting a 2^x unit value. Any bit manipulation on these types does not change the structure of the data, only the represented value.

   Floating-point types use an advanced structure that segments a word into three: sign, exponent, mantissa. Any bit manipulation or shifting may change bits unpredictably, or shift bits from one segment to another (moving bits from exponents to mantissa), leading to cases where a non-FP value is represents (NaN).

2. What is the output of the following program? You will require the use of an ASCII chart.

```
#include <stdio.h>

typedef struct
{
    char a;
    char b;
} twochar;

int main()
{
   twochar x;
      x.a = 'C'; x.b = 'D';
   short *p = (short*)&x;

   printf ("Value of x.a, x.b: %c, %c\n",x.a, x.b);
   *p = *p >> 1;
   printf ("Value of x.a, x.b: %c, %c\n",x.a, x.b);
   *p = *p << 1;
   printf ("Value of x.a, x.b: %c, %c\n",x.a, x.b);

   return 0;
}
```

   The output of this program is as follows,

```
Value of x.a, x.b: C, D
Value of x.a, x.b: !, "
Value of x.a, x.b: B, D
```

   Initially, the structure contains: 'C','D' or: 01000011,01000100

   Now the pointer points to the structure, but sees the bits as a 2-byte integer: a **short**.
   And recall that on Intel platforms, the **short**'s bytes are in reverse order,
   so the pointer sees the number in binary as: 01000100,01000011

   The short is then shift to the right by 1 bit: 00100010,00100001

   The structure's data is reflected as: 00100001,00100010 or '!',' " '

   Finally, a shift left by 1, using the short view: 01000100,01000010

   The structure's data is now reflected as: 01000010,01000100 or 'B','D'

3. Write the code segment that inverts (complements) the last 3 bits of an **int** variable.

   In the **main()**, initialise an **int** variable with a value, display the numeric value, invert last 3 bits (call function), and display the variable's numeric value again.

   Use the technique of masking with bitwise logic operators (AND &, OR |, NOT ~, XOR ^)

   Compare your solution with the following function that inverts the last **n** bits:

```
int invert_end (int num, int n)
{
   int mask = 0;           // mask
   int bitval = 1;         // bit position of 1

   while (n-- > 0)         // loop to create mask for # of bits
   {
      mask = mask | bitval;   // OR mask with bitval
      bitval = bitval <<= 1;
   }
   return (num^mask);    // rerurn original num XOR mask
}
```

   The key to the program above is building a mask that contains 1's in the appropriate right-hand side (LSB) and then XOR (eXclusive-OR) the mask with the value.

   This, of course, works even simpler if the function already knows the exact number of bits required to invert the number accordingly.

```
int main()
{
   int x = 9;       // original value: 00001001
   int mask = 7;    // 00000111

   printf("x=%d\n",x); // display: 9
   x = x ^ mask;    // x XOR 7: 00001001 XOR 00000111 = 00001110
   printf("x=%d\n",x); // display: 14 (00001110)

   return (0);
}
```

4. Write a function **unitVal()** that determines the *unit value* of the first bit in a **short** with a "1," starting from the *msb* (most-significant bit). The returned value is an **int**. (It really does not matter if the value is positive or negative).

   An example of calling the function,

```
short value   = 407;    // 110010111
int   power2 = 0;

power2 = unitval(value);      // power2 is assigned 2^8 = 256
printf ("%d",unitval(1091));  // 1024 is displayed
```

   Using an AND with a "1" in the appropriate place, any bit can examined for it's value (0 or 1).

   Instead of coding little more than a dozen **if**-statements, a simple loop is probably the simplest and most understandable technique in solving this problem.

   The solution below begins a mask that starts with a "1" in the **msb** (most significant bit) and progressively shifts the bit to a smaller unit value until a bit in the source number is "1"—at this point, the mask directly describes the bit position, and hence unit value of the "right-most" one.

```
int unitval (short n)
{
    unsigned short mask = 32768;   // 1000 0000 0000 0000

     // loop while (n AND mask) is zero, and mask is not zero
    while  ( ((n & mask) == 0) && (mask != 0) )
       mask = mask >> 1;     // shift mask's bit down one unit value

    return ((int)mask);      // returns bit value or zero
}
```

5. Write a function **countOnes()** that returns the number of "1" (or "on") bits in an **int** parameter.

   This function is similar to the previous function **unitval()**, except that rather than determining the first 1, it determines all the ones, and returns the count.

```
int countOnes (int n)
{
    unsigned int mask = 2147483648;  // msb (32) bit = 2147483648
    int i=0,                         // loop control
        count=0;                     // count of 1's

     // loop through: 4 bytes * 8 bits/byte = 32 bits
    for (i=0; i<32; i++)   // loop 32 times
    {
       if ((n & mask) != 0)   // if not a zero, must be a value >0
          count++;              // add one to count

       mask = mask >> 1;     // shift mask's bit down one unit value
    }

    return (count);   // mask contains correct bit location or zero
}
```

6. Write a function **xChange()** that performs an interesting swap of bit values. The function has only a single parameter of type **char** and returns a value of type **char**.

   The bit values are swapped from the source to destination bytes, in the following fashion:

```
source byte:  [0][1][2][3][4][5][6][7]

destination:  [6][7][4][5][2][3][0][1]
```

   Possible solutions can use bit masking, or a structure using a bit field structure within the function.

   In a **main()** test program file, use the function **char2bitstr()** that returns the bits of a value in a char array:

```
// represent bits in val to array ps; ps is then returned
char* char2bitstr(char val, char* ps)
{
   int i;
   int size = (8*sizeof(char));  // calc # of bits in a char

   for (i = size-1; i>=0; i--, val >>= 1)
      ps[i] = (01 & val) + '0';

   ps[size] = '\0';
   return (ps);
}
```

This function is rather simple, if one thing is ignored: *that the data is a character or an integer*. If treated just as a sequence of bits, using ANDs to strip the data, shifts to move bits around, and OR to put it all together, the solution is just step-by-step.

```
//source byte:  [0][1][2][3][4][5][6][7]
//destination:  [6][7][4][5][2][3][0][1]
char xChange (char source)
{
    char dest=0;  // destination byte
     // 4 masks used to strip bit groups from source byte
    char mask1=192, // [0][1]: 11000000
         mask2=48,  // [2][3]: 00110000
         mask3=12,  // [4][5]: 00001100
         mask4=3;   // [6][7]: 00000011

     // output (results) of mask and source byte
    char out1 = (source & mask1),
         out2 = (source & mask2),
         out3 = (source & mask3),
         out4 = (source & mask4);

     // shift output byte to correct bit positions
    out1 = out1 >> 6;  // 6 bits to the right
    out2 = out2 >> 2;  // 2 bits to the right
    out3 = out3 << 2;  // 2 bits to the left
    out4 = out4 << 6;  // 6 bits to the left

     // combine (collapse) output bytes into destination byte
    dest = out1 | out2 | out3 | out4;   // use OR | to combine

    return (dest);
}
```

The following main() show how the function is used.

```
int main()
{
   char original = 73,
        copy = 0;
   char bitz[8];         // output array for displaying bits

   //-------------------------------------------------
   printf ("Original char %c, value: %d, %s\n",
        original,(int)original,char2bitstr(original,bitz));

   copy = xChange (original);

   printf ("    Copy char %c, value: %d, %s\n",
        copy,(int)copy,char2bitstr(copy,bitz));

   return (0);
}
```

7. Modify the function above so that it is a **void**-function, and has only a single **char** parameter that is changed when the function completes (*hint: use a call-by-reference* parameter *and proper function call*).

   The modifications are small and focus strictly on how the argument is passed to the function: it is called through a call-by-reference parameter.

```
void xChange (char *source)
{
    char dest=0;  // destination byte
     // 4 masks used to strip bit groups from source byte
    char mask1=192, // [0][1]: 11000000
         mask2=48,  // [2][3]: 00110000
         mask3=12,  // [4][5]: 00001100
         mask4=3;   // [6][7]: 00000011

     // output (results) of mask and source byte
    char out1 = (*source & mask1),
         out2 = (*source & mask2),
         out3 = (*source & mask3),
         out4 = (*source & mask4);

     // shift output byte to correct bit positions
    out1 = out1 >> 6;  // 6 bits to the right
    out2 = out2 >> 2;  // 2 bits to the right
    out3 = out3 << 2;  // 2 bits to the left
    out4 = out4 << 6;  // 6 bits to the left

     // combine (collapse) output bytes into destination byte
    dest = out1 | out2 | out3 | out4;   // use OR | to combine

    *source = dest;
}
```

   The following main() shows how the function call is changed with a call-by-reference argument.

```
int main()
{
   char original = 73;
   char bitz[8];         // output array for displaying bits

   //-------------------------------------------------
   printf ("Original char %c, value: %d, %s\n",
        original,(int)original,char2bitstr(original,bitz));

   xChange (&original);   // note the "address of" operator: &

   printf ("    Copy char %c, value: %d, %s\n",
        original,(int)original,char2bitstr(original,bitz));

   return (0);
}
```

8. Write a function **rotateInt()** that has three parameters: a **char** for direction ('R' or 'L'), the **int** to rotate, and an **int** for the number of bits to rotate. Modify the method **char2bitstr()** and call it **int2bitstr()** and modify it accordingly; use it to test **rotateInt()**

   Bit-rotation is similar to bit-shifting, except that bits are not lost off the end. Such as, for a byte,

```
10010111  -> rotate right by 2 -> 11100101
01101001  -> rotate left by 3  -> 01001011
```

   Keep in mind, that since it is an **int**, depending on the rotation, the value may change from positive to negative (because of the sign bit on the msb-position).

   Consider the simple examples of calling the function:

```
int x = 2;
x = rotateInt( 'R', x, 1);   // x becomes 1

int y=0;
y = rotateInt('L', 2, 3);    // y becomes 32
```

   A left or right shift, using << or >> respectively, moves an entire sequence of bits in one direction. During the shift, any "extra" bits that get moved past the lsb or msb are lost completely.

   The concept of a "rotate" is similar to a shift, but the lost bits are moved to the other end of the word. There are two (2) approaches that can be considered,

   - 1) create a mask that is the correct size for the appropriate bits that will be lost, use the mask to record the bits, shift the original number, and shift the mask and place the bits on the other end of the word

   - 2) loop for the size of the rotate, recording each bit as it becomes "lost", shift the number, and place the bit at the end; or

   Both techniques are shown below,

   ***** Technique 1.**

```
// rotate bits in source in a direction (L or R), and number of rotates (size)
int rotateInt(char direction, int source, int size)
{
    unsigned int interm = 0; // interm, value returned as result of rotation
    unsigned int mask = 0;   // mask created to strip "lost" bits
    int extra = 0;           // extra bits that are lost during shifting
    int i=0;                 // loop control

    //-----------------------------------------------------------------
    if ((size <= 0) || (size >= 31))  // if size is an invalid number
       interm = 0;                    // return zero
    else                             // size > 0, valid rotate size
    {
        switch (direction)
        {
            case 'L':     // shift left
                 // create appropriate mask
               mask = mask | 2147483648;  // store bit in msb
               for (i=1; i<size; i++)     // count 1..size-1 times
               {
                   mask = mask >> 1;           // shift to the right
                   mask = mask | 2147483648;   // store bit in msb
               }

               extra = source & mask;      // determine bits that would be lost
               extra = extra >> (32-size); // shift extra bits to other end
               interm = source << size;    // shift original value
               interm = interm | extra;    // combine lost bits with original
               break;
```

```
            case 'R':     // shift right
                 // create appropriate mask
               mask = mask | 1;          // store bit in lsb
               for (i=1; i<size; i++)    // count 1..size-1 times
               {
                   mask = mask << 1;       // shift to the left
                   mask = mask | 1;        // store bit in lsb
               }

               extra = source & mask;      // determine bits that would be lost
               extra = extra << (32-size); // shift extra bits to other end
               interm = source >> size;    // shift original value
               interm = interm | extra;    // combine lost bits with original
               break;

            default:      // unknown character
               interm = 0;    // return zero
        }
    }
    return (interm);    // return rotated value
}
```

***** Technique 2.**

```
// rotate bits in source in a direction (L or R), and number of rotates (size)
int rotateInt2(char direction, int source, int size)
{
    unsigned int mask_msb = 2147483648;   // mask for msb
    unsigned int mask_lsb = 1;            // mask for lsb
    unsigned int interm = 0;   // interm, value returned as result of rotation
    unsigned int bit_msb = 0;  // bit value at msb
    unsigned int bit_lsb = 0;  // bit value at lsb

    int i=0;                   // loop control
    //-------------------------------------------------------------
    if ((size <= 0) || (size >= 31))  // if size is an invalid number
       interm = 0;                    // return zero
    else                             // size > 0, valid rotate size
    {
        switch (direction)
        {
            case 'L':     // shift left
               interm = source;        // copy
               for (i=0; i<size; i++)  // count for number of rotates
               {
                   bit_msb = interm & mask_msb;  // get msb bit
                   interm = interm << 1;         // rotate left by 1
                   bit_msb = bit_msb >> 31;      // move bit to other end
                   interm = interm | bit_msb;    // combine bit with number
               }
               break;
            case 'R':     // shift right
               interm = source;        // copy
               for (i=0; i<size; i++)  // count for number of rotates
               {
                   bit_lsb = interm & mask_lsb;  // get lsb bit
                   interm = interm >> 1;         // rotate right by 1
                   bit_lsb = bit_lsb << 31;      // move bit to other end
                   interm = interm | bit_lsb;    // combine bit with number
               }
               break;
            default:      // unknown character
               interm = 0;
        }
    }
    return (interm);
}
```

9. Use the functions above (**rotateInt()** and **int2bitstr()**) to create an interesting pattern on the screen.

   Ask the user for an integer value, then loop 32 times (number of bits in an int), displaying the binary sequence for the integer on a single line, rotating to the left once, and displaying the updated value.

   Essentially, this just requires a loop that calls **rotateInt()** repeatedly with the same parameters, creating a cycle that rotates the bits of the same integer. There should be at least 24 iterations, which is the standard lines on a console screen.

   This is left as a student solution.

Conclusion

There are no submissions required for these exercises. You are encouraged to complete all problems, obtaining help from your instructor or fellow students.