THE UNIVERSITY COLLEGE
OF THE CARIBOO

Computing Science – **COMP213**

Lab Exercises #9 – C – *Bit-Fiddling*

No submission required.

Introduction

Manipulating data at the most basic level is a very powerful aspect of programming, allowing for direct-access to the content in memory, by-passing the data type structures and rules that govern controlled manipulation. Clearly, this technique of data using data can be dangerous and unpredictable…but only if the programmer is careless, and *that never happens!*

The exercises below provide practise with the key concepts of bit manipulation (or *bit-fiddling*) using the C programming language. The goal is to implement solutions using *masks* and *left & right shifts*, although solutions are possible using other programming techniques.

Resources and References

You are **strongly** encouraged to use the suggested references for examples and details on this topic.

Exercises / Programming Problems

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

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;
}
```

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
}
```

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
```

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

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';
   returns (ps);
}
```

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*).

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
   ```

9. Use the functions above (**rotate()** 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.

Conclusion

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