COMP 222
C Project: Hamming Code Calculator (first part)

In this project you will implement a collection of functions that implement various operations associated with Hamming Codes. For simplicity, we will limit the code to 12 bit code words (8 bit data, 4 bit check).
There are a relatively large number of functions, each function performing a very simple task. The project therefore emphasizes program design and incremental testing. Execution efficiency is not an important goal for now.

Representing Bit Patterns in Hex
Use the "unsigned" data type and integer constants in hex (prefix "0x"). Use the printf() format code "%x" to print out a value in hex.

```
      unsigned p = 0xFFE3;    // don't write unsigned p = "0xFFE3";
      printf("p = %x\n",p);
```

Logical Operations in C
C has no bool data type, it is simulated with integer values. When a logical value is computed, 0x1 represents true, and 0x0 represents false. When a logical value is evaluated, 0x0 represents false, and anything other than 0 represents true. Logical operations and values are still very important even though there is no bool or boolean data type because logical values are used to drive all decisions for loops and branches.

Logical Operators: **`&& || !`**

Bitwise Operations in C
Related to logical operations in C are the bitwise operators. These operations and values are generally used to perform computations rather than drive decisions.

Bitwise Operators: **`& | ~ ^ << >>`**

Examples:

```
      unsigned q = 0x5;
      q = q << 1;       // shift the bits to the left by 1 position
                        // q now has value 0xA
      q = q << 3;       // q now has value 0x50

      unsigned x = 0x31;      //    x: 0011 0001
      unsigned y = 0x79;      //    y: 0111 1001
      unsigned z = x & y;     //    z: 0011 0001 = 0x31


      unsigned x = 0x31;      //    0011 0001
      unsigned y = ~x;        //    1100 1110 = 0xCE

      unsigned a = 0x5A;      //    0101 1010
      unsigned b = 0x6B;      //    0110 1011
      unsigned c = a ^ b;     //    0011 0001
```

Masks

Use the following bit operation identities:

```
X | 1 = 1
X | 0 = X
X & 1 = X
X & 0 = 0
```

If you want to set a specific bit in the middle of a bit pattern
- Create a mask that will allow you to clear that bit to 0
  - Create a mask1 with a 0 in the desired position, 1's elsewhere (use << and ~)
  - Compute pattern & mask1
- Create a second mask that will allow to set that bit to a specific value
  - Create a mask2 with the desired bit in the desired position, 0's elsewhere (use <<)
  - Compute result1| mask2
  - Or starting with original pattern, compute (pattern & mask1) | mask2

If you want to read a bit in a specific position in a pattern
- Create a mask with a 1 in the desired position, 0's elsewhere
- Compute pattern & mask
- Shift the result to the right to move the value to the 0 bit position

Separate Compilation and Header Files in C

The simplest C application build process is to put all source code into a single source file. As a project gets large, this becomes unworkable. The main problem is that each small mod requires a complete recompile of all code. At some point, organizing smaller sets of related functions into separate files and recompiling that file only when that file is modified is a better approach. For truly massive applications, separate compilation will reduce recompile time, but will not avoid time to relink. Strategies to minimize relink time are not relevant to this course and are beyond its scope.

But if you break up source into multiple files that will be compiled separately, how to keep them in sync with declarations that must be shared in common? The solution is to use external declarations placed into a header file than can be included in multiple files.

C makes a distinction between a **definition** and a **declaration**. A definition is the **real** definition of a variable or function, it can only occur once in the entire program. A **declaration** is an abbreviation of the real definition. Declarations can occur multiple times in the same program without conflict, so long as all copies of the declaration are consistent with the original definition.

Example:

Suppose a program has a global variable **count**, with function f() storing a value into count, and g() reading a value from count. Furthermore, f() and g() are defined in separate source files.

```
A.c
        int count;

        void f() {
                count = 1;
        }

B.c
        void g() {
                printf("%d",count);    // compiler error: "count" not defined
        }
```

The problem is that B.c will not compile, global variable "count" is not defined. But we can't provide duplicate definitions of the same variable. This won't work either:

```
A.c
        int count;

        void f() {
                count = 1;
        }

B.c
        int count;      // linker error: multiple definitions of "count"

        void g() {
                printf("%d",count);
        }
```

What's needed is a declaration that can be repeated along with the actual definition. The definition can occur in either file so long as it only occurs once.

```
A.c
        int count;                      // okay

        void f() {
                count = 1;
        }

B.c
        extern int count;               // okay

        void g() {
                printf("%d",count);
        }
```

By putting the keyword “extern” in front of the definition, we create a declaration that can be repeated. The meaning of “extern int count;” is “this is heads up that this program will define a variable “int count;” but it is external to the current source file, it should show up eventually when the link step is executed.” When the source file is compiled, only the declaration is needed, the actual definition is not needed until the link step later on.

Recap: the definition “int count;” can appear only once in some source file, but the declaration “extern int count;” can appear multiple times, even multiple times within a single source file. If you correctly use the declaration in multiple files but forget to provide the definition in some source file, the declarations will be enough to allow the source files to compile. But the link step will fail with an “unresolved external reference” error when it fails to find the real definition in any of the files being linked.

```
A.c
        extern int count;

        void f() {
                count = 1;
        }

B.c
        extern int count;               // linker error: unresolved reference

        void g() {
                printf("%d",count);
        }
```

So there are many ways you can arrange this but a common way to keep your large program organized is to put all the global declarations into a header file, then include the header file in all source files. Since declarations can be repeated, this will not cause a conflict, even with the file that contains the actual definitions. Also, to make it easier to maintain, I suggest putting all global definitions into a single file. If you decide to distribute global definitions into multiple files, at least declare them in a block near the top of the file. Otherwise you will waste time later trying to locate them when editing the program.

```
myapp.h
        extern int count;

A.c
        #include "myapp.h"

        void f() {
                count = 1;
        }

B.c
        #include "myapp.h"

        void g() {
                printf("%d",count);
        }

main.c
        #include "myapp.h"

        int count;                      // okay to have decl and def in same file

        int main(void) {
                …
                return 0;
        }
```

What about external functions?
C has the same need to coordinate definitions of external function across multiple source files. But instead of using the extern keyword, we make a distinction between a function **definition** and a function **prototype**. The prototype is the abbreviated definition of the function and can be added multiple times.

Suppose we have a function **sum**() that computes the sum of three doubles and returns a double:

```
double sum(double a, double b, double c) {
      double result = a + b + c;
      return result;
}
```

This is the **real definition** of the function and it can occur only once. But we can create the following **prototype** and use in multiple times:

```
double sum(double,double,double);
```

The **prototype** only specifies the return type, the name of the function, and the types of the inputs. Rather than the body of the function within braces, the prototype is terminated with a semicolon. The keyword "extern" is not needed.

Since prototypes can appear multiple times in different source files, they can also be placed into a header file. If you examine the content of header files, you will find that most of the content is external declarations of global variables (using the extern keyword) and functions (in the form of prototypes), as well as nested includes and define macros.

The C compiler reads a C source file from top to bottom only once, a technique called a one-pass compiler. This means that the order in which definitions appear matters in C in way that it doesn't matter in Java which uses a two-pass compiler. In C, code in the source file must be arranged so that a function definition (or prototype) appears before the function is used or called. If the compiler sees a function call before seeing the definition, it makes an assumption that the function has a return type of int. If this assumption is not correct, it may generate a compiler error, or even worse, it might not generate a compiler error but may generate code that computes the wrong answer due to a mismatch in data types. So when distributing function definitions across multiple files, use function prototypes and header files carefully to make sure all source files have a single consistent set of definitions.

See for example http://www.cprogramming.com/declare_vs_define.html

Part 1: Implement Bit Operations

```
unsigned getBitValueAtIndex(unsigned word, int index);
void setBitValueAtIndex(unsigned *word, int index, unsigned value);

unsigned logicalXor(unsigned a, unsigned b);
unsigned bitwiseXor(unsigned a, unsigned b);

int convertDataIndexToCodeWordIndex(int dataindex);
int convertCodeWordIndexToDataIndex(int codewordindex);
int convertCheckIndexToCodeWordIndex(int checkindex);
int convertCodeWordIndexToCheckIndex(int codewordindex);
```

Implementation

- Create the following files
  - hamming.h
  - utilities.c
  - testing.c
  - hamming.c
- Place the prototypes of all functions into hamming.h
- Place the definitions of the functions into utilities.c
- Place the statement #include "hamming.h" at the top of all source files.
- Place the testing functions in testing.c
- Place the main function in hamming.c. Initially, main should do nothing except call the testing function currently being tested.

Description

```
unsigned getBitValueAtIndex(unsigned word, int index);
```

Returns the bit at position **index** within the bit pattern **word**. C treats the rightmost (least significant bit) in word as bit 0, but Haming code defintions treat it as bit 1. Assume the value of index uses 1-based indexing, but you will have to internally adjust for 0-based indexing.

```
void setBitValueAtIndex(unsigned *word, int index, unsigned value);
```

Sets the bit in word at position index to a specific value. This function uses a pass-by-address parameter so that the parameter value itself is modified. Return value is void.

```
unsigned logicalXor(unsigned a, unsigned b);
```

Computes the logical exclusive OR of a and b.

```
unsigned bitwiseXor(unsigned a, unsigned b);
```

Computes the bitwise exclusive OR of a and b.

```
int convertDataIndexToCodeWordIndex(int dataindex);
```

Data bits are counted from 1 to 8, but those data bits occupy specific positions in the code word from 1 to 12. This function converts values in the range 1-8 into values in the range 1-12. For example, data index 1 should convert to code word position 3.

```
int convertCodeWordIndexToDataIndex(int codewordindex);
```

This function performs the reverse calculation from the previous function. It converts indexes in the range 1-12 to values in the range 1-8. Some code word indexes are not valid data indexes, return 0 for these input values.

```
int convertCheckIndexToCodeWordIndex(int checkindex);
```

Check bits are numbered on the range 1-4 but occupy specific positions in the code word from 1 to 12. This function converts input values in the range 1-4 into values in the range 1-12. For example, check index 4 should convert to code word position 8.

```
int convertCodeWordIndexToCheckIndex(int codewordindex);
```

This function performs the reverse calculation from the previous function. It converts indexes in the range 1-12 to check indexes in the range 1-4. Some code word indexes are not valid check indexes, return 0 for these input values.

Constant Definitions for "hamming.h"

Rather than making the program more general but overly complex, this implementation will be limited to Hamming codes for 8 bit data, 4 bit check, and 12 bit code words. Add the following definitions to your header file:

```
#define DATABITCOUNT 8
#define CHECKBITCOUNT 4
#define CODEWORDBITCOUNT 12
```

Testing

For a complex program, an appropriate testing plan is essential. Since there will be additional functions in the next phase that depend absolutely on the functions defined in this phase, all functions in this phase should be tested individually before moving on to implementing follow on functions that depend on them.

Write a function called **phaseOneTest()** like this and place it in testing.c:

```
void phaseOneTest() {

        // tests for get bit function
        printf("get bit value test -----------\n");

        unsigned a = 0xFE3;
        for (int i=1; i<=CODEWORDBITCOUNT; i++) {
                printf("%d %x\n",i,getBitValueAtIndex(a,i));
        }

        // tests for set bit function
        printf("set bit value test -----------\n");

        unsigned b = 0xFFF;
        unsigned c = b;                         // c is a copy of b

        for (int i=1; i<=12; i++) {
                b = c;
                setBitValueAtIndex(&b,i,0);
                printf("%d %x %x\n",i,c,b);
        }

        // tests for convert data indexes
        printf("convert data index test -----------\n");

        for (int i=1; i<=DATABITCOUNT; i++) {
                printf("%d %d\n",i,convertDataIndexToCodeWordIndex(i));
        }

        for (int i=1; i<=CODEWORDBITCOUNT; i++) {
                printf("%d %d\n",i,convertCodeWordIndexToDataIndex(i));
        }

        // tests for convert check indexes
        printf("convert check index test -----------\n");

        for (int i=1; i<=CHECKBITCOUNT; i++) {
                printf("%d %d\n",i,convertCheckIndexToCodeWordIndex(i));
        }

        for (int i=1; i<=CODEWORDBITCOUNT; i++) {
                printf("%d %d\n",i,convertCodeWordIndexToCheckIndex(i));
        }
}
```

Output is:

```
get bit value test -----------
1 1
2 1
3 0
4 0
5 0
6 1
7 1
8 1
9 1
10 1
11 1
12 1
set bit value test -----------
1 fff ffe
2 fff ffd
3 fff ffb
4 fff ff7
5 fff fef
6 fff fdf
7 fff fbf
8 fff f7f
9 fff eff
10 fff dff
11 fff bff
12 fff 7ff
convert data index test -----------
1 3
2 5
3 6
4 7
5 9
6 10
7 11
8 12
1 0
2 0
3 1
4 0
5 2
6 3
7 4
8 0
9 5
10 6
11 7
12 8
convert check index test -----------
1 1
2 2
3 4
4 8
1 1
2 2
3 0
4 3
5 0
6 0
7 0
8 4
9 0
10 0
11 0
12 0
```

For the logical and bitwise exclusive OR functions, add your own test cases to this test function.

For submitting your work, don’t submit the code for this phase, you will combine it with the 2nd phase to complete the program, and you’ll submit the complete program at the same time.