Notes on Computer Project #5

Comments about the assignment and responses to frequently asked questions will be added to this file as necessary.

***** comments added on 09/24/18 *****

~~1)~~ Please note the following statement from the assignment handout:

```
  The deliverables for this assignment are:

    proj05.makefile  -- the makefile which produces "proj05"
    proj05.support.c -- the source code for your conversion module
    proj05.driver.c  -- the source code for your driver module

  Be sure to submit your files for grading via the "handin" program.
```

It is possible to submit your solution files multiple times: the last files that you submit will be graded.

~~2)~~ As stated on the assignment handout, you are required to create a makefile which controls the translation of your program, and the name of your executable file must be "proj05".

If you are not familiar with the "make" utility, an overview is available on the course website:

  Course Information ==> intro.make.pdf

In addition, there is a longer tutorial available on the course website:

  Tutorials ==> Make Tutorial ==> make.tutorial.pdf

3) Please note the following from the assignment handout:

  4. Your driver module may not be written as an interactive program, where the user supplies input in response to prompts. Instead, your test cases will be included in the source code as literal constants.

An interactive program is one in which the user enters inputs in response to prompts. Since that approach is prohibited for this assignment, your driver module will NOT contain code segments such as the following:

```
  printf ) "Enter an angle in radians:" );
  scanf( "%f", &AngleRadians );
```

Instead of prompting the user to enter test cases, you will embed the test cases in the driver module as constants.

That approach (embedding the test cases in the driver module) will facilitate incremental development of the driver module and the support module.

For example, if you were going to test "sin" (from the math library), you might use a series of statements such as:

```
  printf( "%f\n", sin( 0.0 ) );
  printf( "%f\n", sin( 0.5 ) );
  printf( "%f\n", sin( 1.0 ) );
```

Obviously, that code segment is too crude -- the test cases aren't particularly good, and the output isn't labeled or formatted. However, it illustrates the general approach for a driver module which doesn't accept any input.

4) For your convenience, I have supplied the following file:

```
  /user/cse320/Projects/project05.support.h
```

That file contains one source code statement: the declaration of function "convert". Since a function must be declared before it is used, your driver module should contain the statement:

```
  int convert( const char[], int, int* );
```

or the statement:

```
  #include "/user/cse320/Projects/project05.support.h"
```

To permit the compiler to do as much error checking as possible, your support module should also contain one of the two statements above.

Please note that you should not copy "project05.support.h" into your account, since you will not be able to submit that file as part of your solution.

5) In general, preprocessor "include" directives should only be used with interface files (".h" files), not C source code files (".c" files).

Consider the the following preprocessor directives:

```
  #include <stdio.h>
  #include "/user/cse320/Projects/project05.support.h"
  #include "proj05.support.c"
```

The first and second directives could appear in your "proj05.driver.c", but the third is not appropriate.

6) Make sure that you understand the difference between the value of a variable and how that value is represented.

Consider the value twelve. That value can be represented in a number of different ways in mathematics:

```
  30 base 4
  14 base 8
  12 base 10
  C base 16
```

Similarly, there are three different literals to represent twelve in C:

```
  014  /* octal literal */
  12   /* decimal literal */
  0xC  /* hexadecimal literal */
```

The "printf" function is capable of producing octal, decimal and hexadecimal representations of a given value. The twos complement notation cannot be directly displayed, but it is easy to generate through shifting and masking (the "bitlib" functions).

Consider the following example:

```
  #include <stdio.h>
  #include "/user/cse320/lib/bitlib.h"

  int main()
  {
    int A;

    A = 12;
    printf( "A:  %o (base 8)\n",  A );
    printf( "A:  %d (base 10)\n", A );
    printf( "A:  %x (base 16)\n", A );
    printf( "A:  %32s (twos complement)\n", bit32(A) );
  }

  <prompt> gcc source.c /user/cse320/lib/bitlib.o
  <prompt> a.out

  A:  14 (base 8)
  A:  12 (base 10)
  A:  c (base 16)
  A:  00000000000000000000000000001100 (twos complement)
```

Please note that the "%o" and "%x" format specs display integer values as if they were of type "unsigned int". Thus, when "A" is changed to a negative value, the output is quite different:

```
  #include <stdio.h>
  #include "/user/cse320/lib/bitlib.h"

  int main()
  {
    int A;

    A = -17;
    printf( "A:  %o (base 8)\n",  A );
    printf( "A:  %d (base 10)\n", A );
    printf( "A:  %x (base 16)\n", A );
    printf( "A:  %32s (twos complement)\n", bit32(A) );
  }

  A:  37777777757 (base 8)
  A:  -17 (base 10)
  A:  ffffffef (base 16)
  A:  11111111111111111111111111101111 (twos complement)
```

Summary: the external representation of an integer value is some series of characters used to convey meaning to a human (or a compiler); the internal representation of an integer value is a bit pattern which is meaningful in the twos complement system.

7) A polynomial can be evaluated without computing the powers of the base; the technique is called "nesting". For example:

$$ar^4 + br^3 + cr^2 + dr + e = (((((\ 0\ )r + a)r + b)r + c)r + d)r + e$$

Note that only one multiply and one addition are needed for each digit.

Of course, there are additional considerations when converting from external representation to internal representation.

Consider the problem of converting the character sequence "-325" into the value negative three hundred twenty five (as a twos complement number). One method:

```
  int Result = - ((’3’-’0’)*10*10 + (’2’-’0’)*10 + (’5’-’0’));
```

Since the machine uses the twos complement system, the end result of the series of operations is that "Result" contains the twos complement notation for the value negative three hundred twenty five:

```
  11111111111111111111111010111011
```

Clearly, this method is not general enough, since it only converts one specific value. However, it shows the kinds of operations which could be used to convert from external representation to internal representation.

--M. McCullen