University of California San Diego

ECE15: Lab #2

This lab relates to the material covered in Lecture Units 4 and 5 in class, and in Chapters 3 and 5 of the Kernighan and Ritchie textbook. Similar material is also covered in Chapters 5, 6, 7 of the Kochan book. Here are several instruction specific to this lab:

- Throughout this lab, you ***should not assume*** that the input provided by the user has the expected *type*. Specifically, if you are using the **`scanf()`** function to read the input, you should check the value *returned* by this function. If this value is less than what it should be when all the conversions in the format string of **`scanf()`** complete successfully, your program should terminate with an appropriate error message. When using the **`getchar()`** function to read the input, you should check that the value it returns is different from **`EOF`**. Otherwise, the program should again terminate with an appropriate error message.
- You ***can assume*** that when the input does have the expected type, it also fits within the range appropriate for this type. For example, if your program prompts the user to enter an integer and the value returned by **`scanf()`** indicates successful conversion, you can assume that this integer does fit in a memory cell of type **`int`**.
- Unless specifically instructed otherwise in a problem, you ***should not assume*** that the *values* entered by the user are valid. For example, if your program prompts the user to enter an integer between 0 and 10, and the value returned by **`scanf()`** indicates successful conversion to an integer, you should ***verify*** that this integer is in the correct range. If it is not, the program should terminate with an appropriate error message.
- Throughout this lab, you ***should not use*** any functions from the C standard library, ***except*** for those functions that are declared in the file `<stdio.h>`, such as **`printf`** and **`scanf`**.
- For each of the four programs below, you ***should use*** the file `template.c` provided on the course website at `http://ece15.ucsd.edu/Main/Homeworks.html`. Try to make your programs easy to follow and understand: choose meaningful names for identifiers, use proper indentation, and provide comments wherever they might be helpful.

This lab will be graded out of 100 points, with each of the problems worth 25 points. You should submit it by following the instructions posted at `http://ece15.ucsd.edu/Labs/TurnIn2.html`.

Problem 1.

Write a C program, called `linear_solver.c`, that solves single-variable linear equations. The program should prompt the user to enter a linear equation of the form

`aY + b = c`

where **`a`**, **`b`**, and **`c`** are real numbers of type **`double`**. The program should then output the value of **`Y`** that solves the equation, if such a value exists (see the note about verifying that **`a`** in nonzero). If the input provided by the user is not valid, the program should terminate with an appropriate error message.

Notes:

- The character **`Y`** should appear in the input as a capital letter. Introduce this character into the program as a symbolic constant using **`#define`**, as follows:

  ```
  #define VARIABLE_NAME 'Y'
  ```

  When reading the equation using **`scanf()`**, you can use the **`%c`** conversion specification to read the character typed-in by the user, then compare this character with **`VARIABLE_NAME`**.

- There should be no spaces in the input between the number **`a`** and the character **`Y`**. However, there *may* or *may not* be spaces around the **`+`** and **`=`** characters.

- If there is additional text after the equation, but the equation itself is valid, ignore the additional text and solve the equation as described above. For an example of this, see the second sample run.

- Verify that the value of **`a`** is not zero. If this is not the case, you should print an appropriate error message and terminate the program using **`return 1`**. See the third sample run.

- Print the value of **`Y`** that solves the equation with three digits of accuracy after the decimal point.

Here are six sample runs of this program, assuming that the executable file is called `linear_solver`.

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  3Y+5=2
Y = -1.000
```

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  2Y + -12    = 0.5 bla blah...
Y = 6.250
```

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  0Y+2=1
Invalid equation!
```

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  2 Y + 5 = 2
Invalid input!
```

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  3Y-2=1
Invalid input!
```

```
/home/userXYZ/ECE15/Lab2> linear_solver
Enter a linear equation:  2y + 12 = 0.5
Invalid input!
```

Problem 2.

An integer $p > 1$ is said to be *prime* if p is divisible only by 1 and by p itself (strictly speaking, a prime p is divisible by ± 1 and $\pm p$, but this is not relevant for the problem at hand). Every integer $n > 1$ can be written in a *unique way* as a product of primes in nondecreasing order. The nondecreasing sequence of primes in this product is called the *prime factorization* of n. Here are a few examples:

- Since $6 = 2{\cdot}3$, the prime factorization of 6 is $2, 3$.
- Since $12 = 2{\cdot}2{\cdot}3$, the prime factorization of 12 is $2, 2, 3$.
- The prime factorization of 7 is 7.
- The prime factorization of 300300 is $2, 2, 3, 5, 5, 7, 11, 13$.

Write a C program, called `prime_factorization.c`, that prompts the user to enter an integer and then prints its prime factorization. The program program should keep doing this repeatedly as long as the user enters an integer greater than 1. If the user enters an integer less than or equal to 1, the program should simply terminate. If the user enters a non-integer, the program should terminate with an appropriate error message. Here is one sample run of such a program, assuming that the executable file is called `prime_factorization`.

```
/home/userXYZ/ECE15/Lab2> prime_factorization
Enter an integer:  75
The prime factorization of 75 is 3 5 5

Enter an integer:  18
The prime factorization of 18 is 2 3 3

Enter an integer:  17
The prime factorization of 17 is 17

Enter an integer:  1
/home/userXYZ/ECE15/Lab2>
```

Notes:

- ▶ Print the prime factors in nondecreasing order, as in the sample run above.
- ▶ If a prime factor appears in the factorization m times, it should appear in the output m times as well. For example, in the sample run above, 5 appears twice in the prime factorization of 75.
- ▶ Verify successful execution of the input function. If its execution is not successful, print an appropriate error message and terminate the program using **`return 1`**.

Hints:

- ▶ You can use the following *algorithm* to compute the prime factorization of an integer $n > 1$.

  Step 1. Initialize by setting $i = 2$.

  Step 2. If i divides n, adjoin i to the prime factorization and set $n = n/i$; else increase i by 1.

  Step 3. If $i > n$ terminate, otherwise go back to Step 2.

- ▶ It is a good idea to convince yourself, first, that the foregoing algorithm is correct. For your information, a list of the first few prime numbers can be found at `www.prime-numbers.org`.

Problem 3.

The ABRACADABRA language has just five lowercase letters **a,b,c,d,r**, and every combination of these letters (but no other letters) is a word in the ABRACADABRA language. For example, **abba** is a word in the language, but **aBBa** and **abbe** are not. The good people of ABRACADABRA often *encrypt* their words, so that they can send each other messages which their adversaries cannot understand. To this end, they use the following simple encryption process, called a *substitution cipher*.

First consider the *natural* (or *lexicographic*) order of the five letters, where **a** is the first letter, **b** is the second letter, and so on, with **r** being the last letter. Two parties who wish to communicate with each other agree in advance on a a secret order of the five letters in the language, also called the *encryption key* (for example **r,b,c,d,a**). To encrypt a message, they create a table consisting of two rows. The first row lists the letters **a,b,c,d,r** in their natural order, while the second row lists these same letters in the secret order of the encryption key. Here is an example of such a table:

natural order	**a**	**b**	**c**	**d**	**r**
encryption key	**r**	**b**	**c**	**d**	**a**

An arbitrary word in the ABRACADABRA language can be now encrypted by replacing every letter in the word by an encrypted version thereof that appears in the second row of the table. For example, if the encryption key is **rbcda** as above, every **a** is replaced by **r** and every **r** is replaced by **a**, while the letters **b,c,d** remain unchanged. Thus the word **abba** gets encrypted **rbbr**. Warm up exercises:

- If the secret key is **abcdr** and the source word is **abbaca**, what is the encrypted word?
- If the secret key is **arcdb** and the source word is **abbacar**, what is the encrypted word?

Write a C program, called `abracadabra_encoder.c`, that encrypts words in the ABRACADABRA language. The program should prompt the user to enter a secret key and a word. You ***can assume*** that the secret key will consist of the five characters **a,b,c,d,r** listed in some order, each appearing exactly once, with no spaces. The leftmost letter is the first letter of the secret key (hence it should replace the letter **a** in your encryption), and so on. The word should be a valid word in the ABRACADABRA language, consisting of ***seven letters***. If the user enters anything *after* the seven letters, you should ignore this input. On the other hand, if the first seven characters typed in by the user do not constitute a valid word in the ABRACADABRA language, the program should print an appropriate error message and terminate (remember to verify the success of the input functions **scanf()** and/or **getchar()**). The output of the program should be the encrypted word. Here are two sample runs of this program, assuming that the executable file is named `abracadabra_encoder`.

```
/home/userXYZ/ECE15/Lab2> abracadabra_encoder
     Enter key:  rbcda
    Enter word:  abbacar
Encrypted word:  rbbrcra
```

```
/home/userXYZ/ECE15/Lab2> abracadabra_encoder
     Enter key:  rdbca
    Enter word:  acrAbar
You did not speak in ABRACADABRA to me!
```

Problem 4.

The program requested in the previous problem assumes that a word has exactly seven letters. However, in reality, words in the ABRACADABRA language may have arbitrary length. Thus it would be nice to modify this program so that it can support words of arbitrary length. Moreover, most ABRACADABRA speakers have not taken ECE15 and cannot write C code themselves. Therefore, you are requested to write a C program, called `write_encoder.c`, that creates *by itself* the `abracadbra_encoder` program for any given word length. Your program should prompt the user for the desired word length, and then output a C code of the `abracadbra_encoder` program for the specified length.

To see how one C program can output another C program, consider the following program, called herein `write_hello.c`, that prints to the screen the famous "**`hello world!`**" program:

```
#include <stdio.h>

int main()
{
   printf("#include <stdio.h>\n\n");
   printf("int main()\n");
   printf("{\n");
   printf("   printf(\"hello world!\\n\");\n");
   printf("   return 0;\n");
   printf("}");
   return 0;
}
```

Notes:

- Note that the fourth **`printf`** statement above contains the sequences `\"` and `\\`. The reason is that inside a **`printf`** statement, the characters `"` and `\` have special meanings: the former signifies the start and the end of the **`printf`** format string, while the latter modifies the meaning of the character that follows. For example, **`\n`** is the new-line character, rather than the character **`'n'`**. To "turn off" these special meanings, one should precede `"` and `\` by the backslash `\`. Thus `\"` just prints a `"` while `\\` just prints a `\`. You might want to test those yourself.

- The `%` character also has a special meaning inside the **`printf`** format string (what is its meaning?). Thus, in order to print the `%` character itself, you should write `%%`. For example, check the output of **`printf("Up to 50%% discount!")`**

- To verify the correctness of the above program you should redirect its output to a file. For example, if the executable file is called `write_hello`, you should invoke the program as follows:

```
/home/userXYZ/ECE15/Lab2> write_hello > hello.c
```

If `write_hello.c` is written correctly, then the above should create a file called `hello.c`. The file `hello.c` can then be compiled to generate an executable file which, when run, should print **`hello world!`** to the screen. It is recommended to actually try this!

- You should test the correctness of the program `write_encoder.c`, that you are writing in this problem, in the same way. For example:

```
/home/userXYZ/ECE15/Lab2> write_encoder > mytest1.c
```

  Make sure that the output is redirected to a file with a `.c` suffix, so that it can be compiled afterward. As you develop your code, you may want to create several such `mytest.c` programs.

- Observe that when you follow the example above, the prompt "**`Enter the desired word length:`**" will be printed to the file `mytest1.c` and ***not*** to the screen. This implies two things. First, you will have to type-in the desired word length, without being prompted for it. Second, before you can compile the file `mytest1.c` you should manually delete the first line in this file. You can avoid these complications, during most of the development, by leaving out from your program the part that deals with prompting the user and reading the desired word length. You could add this part when you are sure that the rest of your code is correct.

- In this problem, you ***may assume*** that the user will enter a positive integer when prompted for the desired word length. Moreover, even though words in the ABRACADABRA language can have arbitrary length, you ***may assume*** that the user will ***not*** request a length greater than 200.

Here is a sample run of the `write_encoder.c` program, assuming that the executable file is called `write_encoder`. In this example, the user enters 100 as the desired word length.

```
/home/userXYZ/ECE15/Lab2> write_encoder
Enter the desired word length:  100

#include <stdio.h>

int main()
{
   char ...
   printf("     Enter key:  ");
   :

   return 0;
}
```

Comment: For those of you who are curious about C programs that output C programs, the program `self_replicator.c` outputs to the screen an ***exact copy of itself***. This program can be found at `http://ece15.ucsd.edu/Misc/self_replicator.c`. The program does *not* precisely conform to the **C89** or **C99** standards (doing this would be much more difficult), but should work on most compilers. It also involves certain C tricks that have not (yet) been covered in the course. Nevertheless, it is fun! Try to download it, compile it, run it with redirection of output, and then compare the file thereby created with the source code file `self_replicator.c`.