Exercise 06

Building a Multi-File Project

For this exercise, you get to fill in a few missing parts in a program that's made from two implementation files (and a header). You also get to write a simple Makefile to help automate an efficient build for the program.

On the course homepage, you'll find two source files, a sample input file and an expected output file. You should be able to use the following curl commands to download copies of these files to your current directory.

curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise06/main.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise06/print.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise06/input.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise06/expected.txt

Program Behavior

The job of this program is to read text from standard input and to print it to standard output. As the text is printed, the program should replace all the spaces with dashes. As it does this, it will keep a count of all the spaces it replaced, so it can report this at the end.

When it's complete, you should be able to run the program on the contents of the input.txt file and get the following output:

```
$ ./main < input.txt
This-is-just-a-text-file.
It-has-some-spaces-between-the-words.
---------And,-I-put-a-bunch-of-spaces
at-the-start-of-the-line-above,-just-so
we'd-have-more-spaces-to-replace.

That contained 39 spaces
```

If you want to test your program against the expected output file, you can capture its output to a file, then use diff to make sure you got the right output:

```
$ ./main < input.txt > output.txt
$ diff output.txt expected.txt
```

Completing the Implementation

The main.c file is already complete; you won't need to make any changes to it (in fact, don't make any changes). To finish the program, you need to fill in some missing code in print.c, and create a new header file, "print.h", to go with it.

- In print.c, make a new global variable named spaceCount, to count all the spaces that are replaced by dashes.

- In print.c, fill in the body of the dashesForSpaces() function. It's given a character each time it's called. Most characters will get printed to standard output normally. Spaces will get counted (using the global variable), then printed out as a dash instead.

- Make a new header file, print.h. Its job is to advertise the features of print.c that can be used by other components. It should contain:
  - A prototype for the function defined in print.c
  - An external variable declaration for the global variable defined in print.c

Notice that the print.c file includes its own header file as its first line of code. It doesn't really need to do this in order to compile; it already contains definitions for everything described in the header. However, it's a good practice for every component to include its own header first like this. It helps make sure that the header compiles cleanly (without depending on inclusion of other headers first), and it helps detect inconsistencies in case the prototypes and external variable declarations in the header don't match the definitions in the implementation file.

Once you have these parts implemented, you should be able to compile your program using a command like the following. This compiles both source files and links them together into an executable named main.

```
gcc -Wall -std=c99 -g main.c print.c -o main
```

Creating a Makefile

Create a Makefile (named "Makefile" with no filename extension). Your Makefile should be able to build your program efficiently, first compiling each of the source files into an object file, and then linking the two objects into a complete executable. Building the project in stages like this is a good idea. If one source file changes, we only need to rebuild the targets that depend on it.

In your makefile, instead of defining your own build rule for every target, you can just use the ones built into make. You'll need to set the CFLAGS variable to include the options we normally use during compilation (-Wall, -std=c99 and -g), and you'll need to redefine the CC variable so make uses gcc rather than cc (although these two names actually get you to the same compiler on the common platform systems).

Setting these variables will customize the default build rules, but you also need some dependency lines, to tell make what needs to be built. One of your dependency lines will say you need to rebuild the main target whenever either the object file, main.o changes or the object file print.o changes. For each of the objects, you'll need another dependency line saying what they depend on. The main.o object depends on the main.c implementation file and the print.h header (since it includes that header), and the print.o object file depends on the print.c implementation file and the print.h header.

Since I got make's default rules to do most of the work for me, my Makefile ended up being simple, with only 5 non-empty lines. Once your makefile is working, you should be able to build the project like this. From the output of make, you can see that it's compiling each source file separately (the first two output lines), then linking them together into an executable. (To get make to rebuild everything, you may need to remove your object files and your executable first. Later, we'll see how to write a rule to do this for us.)

```
$ make
gcc -Wall -std=c99 -g   -c -o main.o main.c
gcc -Wall -std=c99 -g   -c -o print.o print.c
gcc   main.o print.o   -o main
```

You can check that your makefile is set up to rebuild your program selectively. Modify one of your source files and make sure it just rebuilds the parts that need to be rebuilt. For example, if I make a change to my print.c file, it just has to rebuild the print object and then relink the executable:

```
$ make
gcc -Wall -std=c99 -g   -c -o print.o print.c
gcc   main.o print.o   -o main
```

Submitting your Work

When you're done, use the exercise_06 assignment on Moodle to submit your four files, **main.c**, **print.h** and **print.c** as well as your **Makefile**.