CS 352 (Fall 17): System Programming and UNIX

Project #2

Getting Started with C

due at 9pm, Wed 14 Sep 2017

1 Overview

This project has three small programs. In the first, we'll provide a program which has had all of its `#include` directives removed; you must use the man pages to find out what headers to include so that the program builds with no warnings or errors.

In the other two, we'll assign small problems which will require you to read some input, perform some simple calculations, and then print out a result. You will typically use `scanf()` for reading from input, and `printf()` for writing to the output.

1.1 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://lecturer_russ.appspot.com/classes/cs352/fall17/DOCS/coding-standards.html`

- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  (In bash, you can check the exit status of any command (including your programs by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

  TECHNICAL NOTE: It is possible to terminate a program without returning from main, by calling `exit(int)`; in that case, the exit status is the value that you pass to `exit()`.

2 Turning In Your Code

Turn in your code using D2L. Turn in the following files:

```
silly.c
squares.c
powers.c
```

3 Grading

I have provided compiled versions of these programs for you; you should download them and run them (on Lectura) for comparison. The programs can be copied from `/home/russelll/cs352_website/projects/proj02/` on Lectura; the UNIX commands you've learned, such as `ls, cd, cp` will all be useful here. (We have provided the same on the website; however, realize that these will only run on a Linux machine running the x86-64 architecture.)

We have also provided our grading script. (We'll do this for the first couple of assignments, in order to help you understand the requirements - but we will stop doing this as the programs get more complex.) You can copy it from `/home/russelll/cs352_website/projects/proj02/grade_proj02` on Lectura.

Finally, we have provided a few example inputs to test your program with. (We will use additional ones when we grade your program.) **You should write additional testcases** - try to be creative, and exercise your program in new ways.

NOTE: Since this project has multiple programs, it needs testcases for each one. The testcases are named `test_<progName>_*`. In order for the grading script to see your new testcases, please name them according to this standard. (The first program, `silly.c`, does not have any testcases.)

3.1 Exact Output

In each Project this semester, most of your grade will come from automatic testing of the code. You must match `stdout` **EXACTLY**, byte for byte. Common mistakes include:

- Extra or missing spaces
- Extra or missing blank lines
- Misspelled words
- Different capitalization

Your code must also give the same return value (0 or 1) as the example executable in every case.

`stderr` will be handled a little more loosely. In each testcase, we will check to see if the standard executable reported **some** error message to `stderr` or

not. If it did, then your program must as well; if not, then your program must not print anything, either. However, we will not be comparing the exact error messages.

NOTE: Each testcase either passes, or fails entirely. We do not give partial credit for any testcase - although you might, of course, pass some testcases but not others.

3.2 Lectura

We will be doing our testing on Lectura. You should also test your code on Lectura before you turn it in! Problems porting from one machine to another are kind of rare, but you don't want to lose points because of it!

3.3 Running the Grading Script

To run the grading script, arrange your files like this, and then run `./grade_proj02`

```
grade_proj02

example_silly
example_squares
example_powers

test_squares_*
test_powers_*

silly.c
squares.c
powers.c
```

The grading script does a number of things:

- Confirms that we have an example executable for each program.
- Confirms that each of the required files has the proper name, in a directory with the proper name. If not, you lose all points for that program.
- Compiles your code. If your code doesn't compile at all, then you lose all points for that program. If it compiles but has warnings, then you will get a heavy deduction.
- Runs your code against all of the testcases. In each case, it runs both your code, and also the example executable. It checks to make sure that both have the same return code - and then also checks to make sure that the outputs match. If not, then it will give you a zero for that testcase.

  (If you have no testcases for some program, then the script will give you a zero for that program.)

At the end, the grading script reports a score; this score is determined by the number of testcases that you passed.

3.4 Hand Grading

In addition, each program will be looked at by a human, to check for things which we can't automatically check, like:

- Some of the details of the spec
- Good comments
- Good indentation
- Reasonable variable names

3.5 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

3.5.1 Weight of each program:

If any program compiles and runs, but warnings were produced by the compiler, you will lose half your score for that program.

- 10% - `Silly`
- 30% - `Squares`
- 30% - `Powers`

The last 30% of the score will come from hand grading by the TAs.

4 Program 1 - Missing `#include` s

The program `silly.c` (found on Lectura at `/home/russelll/cs352_website/projects/proj02/`, and on the web) has had all of its `#include` directives mysteriously deleted; as a result, the program no longer compiles. You are to figure out which files you need to include, add the appropriate `#include` directives to the file so that it compiles and runs without any warnings or errors using `gcc -Wall`, and turn in the file so modified. The order in which the include files are listed in the file you turn in is unimportant.

Important: Do not include more files than necessary. Do not change the name of the file.

Comment: You don't need to know what this silly program does, you just need to make it compile without warnings or errors. The missing include files cause the compiler to report syntax errors. Symbols (e.g. `NULL`) used in the program are specified in the missing include files; without them, we get syntax errors. To fix the problem, start with the library functions in the program, figure out what the relevant include files are, and add the appropriate `#include` directives. This will usually take care of defining associated symbols. If you aren't sure whether something is the name of a library function, use `man` (in some cases, you have to specify section 3).

Once again, you do not have to understand or run the program. You just have to get it to compile without warnings or errors by adding the minimal number of `#include` directives. If you **do** want to run it you should know that you need to type the command, followed by an integer. (It uses this integer to initialize the random number generator.)

Here's an example of how to run the program if you want to:

```
gcc -Wall -o silly silly.c
./silly 4
```

5 Program 2 - Squares

This program prints out text squares.

Name your file `squares.c`

5.1 Input

This program takes all of its input from `stdin`, and you must read this using the `scanf()` function.

The first thing in the input must be a decimal number (read this with the `%d` specifier of `scanf()`, and read into an `int`). After that, the input is simply a long series of characters; the program will read them one at a time. (Use the `%c` specifier of `scanf()`, and read into a `char`.)

The first parameter is the size of the squares that this program will print. All of the squares will be the same size, and you will leave exactly one blank line between each square (but do not print a blank line before the first square, or after the last one). Print one square for each character in the input after the size.

The characters that you read will be the "fill" characters, which are placed in the middle of your squares. So if the input is "10 asdf", you will print squares of size 10 - the first will be filled with spaces, the second with the letter `'a'`, the third by the letter `'s'`, and so on.

Your program should terminate when you cannot read any more characters. Return 0 - unless you were not able to read the "size" input.

5.2 Non-Printable Fill Characters

For each character, you must check to see if the character is a "printable" character - meaning one that can be printed to the screen without causing strange artifacts. The C standard library has a function to decide this; use `man 3 isalpha` to read the correct man page to find this. (The function you should use is **not** `isalpha()` - but you'll find it on the same man page.)

If the character is not printable, then print out a message (instead of the square); run the example executable to find out exactly what this message is. (Note that this doesn't count as an error; print the message to `stdout`, and you should still return 0 later on, when the program ends.)

5.3 `drawSquare()`

You must encapsulate all of the drawing logic inside a function named `drawSquare()`. This function must take the size, and the character, as parameters - **no global variables are allowed.**

You **must** call `scanf()` from `main()` - that's where your main `while()` loop will be. For each character that you read, call `drawSquare()`.

The check to see if a character is printable must be **inside** `drawSquare()`.

The code which prints blank lines between the squares must **not** be in `drawSquare()`; do it in `main()`.

5.4 What the Square Looks Like

You should run the example executable, with various size parameters, to see what the square should look like. Pay attention to how it looks when the size is 1, then it is an even number, and when it is odd. Things to look for include:

- What is printed on the edges
- What is printed on the diagonals
- What is printed in the center (and when this piece is skipped!)

5.5 Error Conditions

Your program should normally print only to `stdout` (using `printf()`), and should return 0 from `main()`. However, you must print an error message to `stderr`[1], and return nonzero from `main()`, if any of the following conditions occur:

- You are unable to read a 'size' from the input
- The size is zero or negative

[1] While your error message doesn't have to match the example executable exactly, it should be meaningful. Write a message which would help your user debug the problem. If you don't do this, we will mark you down for style.

6 Program 3 - Powers

This program prints out the powers of a number, and checks to see which of them are multiples of another number. Since large intergers overflow, you may find the results surprising when the numbers get large.

Name your file `powers.c`

6.1 Input

This program takes all of its input from `stdin`, and you must read this using the `scanf()` function.

The input is made up of three integers, separated by whitespace: the "number," "count," and "modulo". The first and third must both be at least 2; the middle can have any value. (In your loop, treat negative count values the same as zero.)

Your program should ignore any input which comes after these three numbers.

6.2 Behavior

Print out `count` many powers of `num`, starting at 1 (the zeroth power). Calculate large powers of `num` using an `int` variable; multiply `num` into it, over and over. Allow overflow to happen - if the number gets too large, continue to just print out whatever is in the `int`.

For each number that you print out, check to see if it is also a multiple of `modulo`; if it is, print out an extra bit of text on the line of output. (Run the example executable to see what this is.)

Finally, when the loop completes, print out a count of how many multiples you found. (Run the example executable to see the format of this line.)

6.3 Error Conditions

Your program should normally print only to `stdout` (using `printf()`), and should return 0 from `main()`. However, you must print an error message to `stderr`, and return nonzero from `main()`, if any of the following conditions occur:

- You are unable to read the three integer parameters
- `num` or `modulo` are less than 2