

**Project #2**  
Getting Started with C  
due at 9pm, Wed 14 Sep 2016

## 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://www.cs.arizona.edu/classes/cs352/fall16/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 different 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 turnin

Turn in the following directory structure using the assignment name `cs352_f16_proj02`:

```
silly/  
    silly.c  
collatz/  
    collatz.c  
primePair/  
    primePair.c
```

## 3 Grading

We 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/cs352/fall16/Assignments/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/cs352/fall16/Assignments/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_collatz
example_primePair

test_collatz_*
test_primePair_*

silly/
    silly.c

collatz/
    collatz.c

primePair/
    primePair.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.

- 20% - Silly
- 40% - Collatz
- 40% - Prime Pairs

#### 3.5.2 Possible Penalties

- -10% - Unnecessary headers in `silly.c`
- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)

## 4 Program 1 - Missing `#include` s

The program `silly.c` (found on Lectura at `/home/cs352/fall16/Assignments/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 greater than 1. If you run it without this integer, it will crash. This is not an error on your part; this is bad program design.

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

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

## 5 Program 2 - The Collatz Conjecture

[https://en.wikipedia.org/wiki/Collatz\\_conjecture](https://en.wikipedia.org/wiki/Collatz_conjecture)

<https://xkcd.com/710/>

<https://bl.ocks.org/cmgiven/231f779f9655025f38b5b4b828f3b7b0>

In this program, you will test the Collatz Conjecture. For each integer in the input, you will run the recursive function ( $3n + 1$  if odd,  $\frac{n}{2}$  if even) until you reach a number which is less than or equal to the starting number. You will print out the numbers in each sequence as you go.

Name your file `collatz.c`

## 5.1 Input

The input is a sequence of positive integers, all encoded in base 10. These numbers may or may not be on the same line.

## 5.2 Behavior

Your program should read in the input numbers, and for each number, test the Collatz Conjecture on it. That is, print the number, then a colon, then a list of the numbers that follow it in the sequence. End when you find any number less than or equal to the starting number. Separate the numbers by spaces, with no trailing space; print a newline at the end of each sequence.

## 5.3 Error Conditions

It is an error if the input contains any non-positive integers or any non-integer values. Non-positive values in the input should not produce any output to `stdout`, but each time an error is encountered, a message should be sent to `stderr`. A non-positive input should not cause the program to exit. However, if the input contains something which cannot be read as an integer, your program should print an error `stderr` and exit.

The exit status of your program should be:

- 0 if there were no errors on the input
- 1 if there were errors on the input (even if the program was able to keep running)

All error messages must be printed to `stderr`.

# 6 Program 3 - Prime Pairs

In this program, you will scan a range, and print out all of the numbers which are the product of exactly two primes.

Name your file `primePair.c`

## 6.1 Input

The input is two positive integers. (If there is any additional information on the input, you may ignore it.)

## 6.2 Behavior

Your program should read in the two input numbers. Then, for each number in the range (including both ends), check to see if the number has exactly two prime factors. If so, then print out the number, a colon, and then both factors, like this:

```
14: 2 7
```

If the number is prime, or has more than 2 prime factors, then print nothing at all.

Note that, in this program, any duplicate factor counts as multiple factors. Thus, 8 should not be printed, because  $8 = 2 \cdot 2 \cdot 2$ , but 4 **should** be printed:

```
4: 2 2
```

### 6.3 Hint 1 - firstFactor()

There are many possible strategies you could use to determine whether a number has exactly two prime factors. One possible strategy - and the one we use in the solution - is to define a function like this:

```
int firstFactor(int val)
```

This function could be used to quickly find factors in any input number!

If you do so, make sure to include a comment for the function - as well as the normal header comment that all files require!

### 6.4 Hint 2 - Simplicity

Don't try to find **all** of the factors of a number - that list could be long, and you'd need an array to store it! (We haven't shown you arrays yet.) Instead, simply write code which can quickly check only this question: "Does the input number have exactly two factors or not?"

### 6.5 Hint 3 - sqrt()

C has a `sqrt()` function. If you want to use it, you may; look up the man page to see how to use it, and what header to include. However, it is possible to implement this program without the use of `sqrt()`; you can do this with multiplication instead.

If you want to call `sqrt()`, take a look at its man page. Near the top, you'll find a special notice:

```
Link with -lm.
```

This means that this function needs a special library (the math library) for your program to run. Take a look at the "Basic C" slide deck, slide 60, to see how to link with this library.

## 6.6 Error Conditions

If the input has fewer than two integers - or anything which cannot be read as an integer by `scanf()` - then print an error message, and end the program immediately. If the input has more inputs than required, you may ignore the extra ones.

If the input has two integers, but the first is more than the second, then print an error message and end the program immediately.

If the input has exactly two integers and they are in order - but the first is less than two, then print an error message and end the program immediately.

The exit status of your program should be:

- 0 if there were no errors on the input
- 1 if there were errors on the input

All error messages must be printed to `stderr`.