CS 352 (Summer 2021): Systems Programming and UNIX

# Project #5.1 (50% grade of complete project 5)

Reading files

due at 11:30pm, Thursday 8 Jul 2021

## 1 Overview

This project has one program - which focuses on a reading files in C.

### 1.1 Standard Requirements

- Your C code should adhere to the coding standards for this class: `https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en`
- 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).

## 2 Turning In Your Code

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

```
compareBytes.c
```

## 3 Grading

We have provided compiled version of the program for you; you should download them and run them (on Lectura) for comparison. The programs can be copied from `/home/marahman/cs352/projects/proj05-1/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.). Note that you can copy the files using the `scp` command. You can find more how to use the `scp` command using the man page. One example to download the files to your current working directory of your non-lectura Linux machine:

```
scp lectura.cs.arizona.edu:/home/marahman/cs352/projects/proj05-1/* .
```

(We have provided the same on Piazza; however, realize that these will only run on a Linux machine running the x86-64 architecture. If don't have a linux machine, you should work on Lectura and only copy the files to your lectura account and your choice of working directory.)

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/marahman/cs352/projects/proj05-1/grade_proj05-1` on Lectura, or from the web.

Note that the grading and testing scripts are executable binaries but only have read permissions! Once you download or copy them to your working directory, you need to modify their permissions to execute them. You may need to review the UNIX lecture from Week 1 and are welcome to use google to see how you can change permissions.

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 test-cases** - try to be creative, and exercise your program in new ways.

**NOTE 1:** This sub-project has one program The test-cases are named `test_<progName>_*`. In order for the grading script to see your new test-cases, please name them according to this standard.

**NOTE 2:** Part of this project uses `stdin` as its input; name those test-cases `test_*.stdin` . Some use command-line arguments; name those test-cases `test_*.args` .

## 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 test-case, 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 test-case either passes, or fails entirely. We do not give partial credit for any test-case - although you may, of course, pass some test-cases but not others.

## 3.2 Running the Grading Script

To run the grading script, make sure you place the following files in the same parent directory (e.g., `proj05-1`) and then run `./grade_proj05-1`

```
grade_proj05-1

example_*
test_*

compareBytes.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 test cases. 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 test case.

  (If you have no test cases 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 test cases that you passed - modified for any deductions.

### 3.3 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

### 3.4 Point Distribution

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

#### 3.4.1 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.

- 60% - `compareBytes`

The last 40% of the score will come from hand grading by the TAs.

## 4 Program - Compare Bytes

This program opens two files, reads them both, and reports the first place where the two bytes diverge.

Name your file `compareBytes.c`

### 4.1 Input

This program reads its parameters from **the command line** - not from `stdin`. The program must have two command-line arguments, which are the names of files; you will open both with `fopen()`.

Read one byte from each file at a time. You can do this using `fscanf()` - but it's generally easier to use a specialized function, such as the `getchar()` family. Check the man page for `getchar()`; as I've done before, the "right" function you need isn't `getchar()` - but it's something related to it, which is on the same man page.

So long as the two files are identical, continue to read them, byte by byte, until either you hit EOF, or some byte which is not the same in the two files. If the files are absolutely identical (meaning that they have the same size **and contents**), then print out a certain message (see the example executable to see what it is).

If the two files are identical up to one point - and then one ends (but not the other), then you will print out a different message; this message includes the next 4 bytes in the longer file (fewer if there are less than 4 bytes left in that file). Again, check the example executable to see how this is formatted.

Finally, if the two files are the same up to some point (but then both continue), you will print a third type of message - along with (up to) the next 4 bytes from the file.

### 4.2 Required Functions

You must write two functions[1]. Use them in your implementation!

[1] You are allowed to write more if you want - but these two are required.

#### 4.2.1 `dumpNext()`

This function must be called, when it's time to report a difference between two files (either because they had different contents, or because one was shorter than the other). It takes three parameters, and returns nothing:

```
void dumpNext(char *filename, FILE *fp, unsigned char c);
```

The first two parameters are obvious: they are the name of the file which is being printed, and the file handle which you have been using to read from it. The third parameter is the **first character which is different** - this is necessary because (of course), your program has **already read** one character from the file - and this was the one that you found was different!

This function should print out a single line, including the newline at the end. If the file has at least 4 bytes left (including the one that was passed as a parameter), then print out the following message:

```
The next 4 bytes of ...
```

(see the example executable for the exact format).

However, if the file has fewer than 4 bytes left (including the one parameter), then you will print out something like this (again, check the example for the exact format):

```
<filename> has only <n> bytes left ...
```

#### 4.2.2 `printChar()`

This function must be called, as a utility function, from `dumpNext()`. It prints out the information about a single byte from the file; `dumpNext()` may call it as many as 4 times. This function takes a single parameter, which is the byte to print:

```
void printChar(unsigned char c);
```

This function will print out the hex value of the character, an equals sign, and then one of three things:

- The character itself, if it is printable.
- A special message if the character is null.
- A different message if the character is non-null but not printable.

As always, check the example executable for the exact format.

### 4.3 Error Conditions

Your program should normally print only to `stdout` (using `printf()`). This includes even output which says that "the files are different," and gives information about that - since having two files which are different is **not an error.**

However, we will **do something unusual with the exit status.** Your program should **only** return 0 exit status if the two input files are **absolutely identical.** If your program detects any difference, then it must report it (as described above), and then **return 1.**

In addition, your program must detect things which are actually errors. Print out a message to `stderr`, and then immediately terminate the program (with exit status of 1), if any one of the following happens:

- There are anything other than exactly 2 command-line arguments (plus the name of the program itself). Note that too many arguments should be handled as an error, just like too few.
- Your program is unable to open one or both of the input files.