Project 0
Warm-Up

INTRODUCTION:

It is not uncommon for students come into this course without basic C software development skills (which should have been developed in the CS111 prerequisites), and invest a great deal of time and frustrating effort in the first two projects before concluding they will not pass the course and must drop. We have created this simple warm-up to determine whether or not students are prepared to work on C programming projects. Most students should find this project to be relatively easy (a few hours of work, mostly understanding the APIs). If you do not find this project to be relatively straight-forward, you may want to reconsider whether or not you are ready to take this course.

RELATION TO READING AND LECTURES:

None. This project requires only C programming skills and basic familiarity with command line interfaces and makefiles that incoming students should already possess. Mastery of the material covered in a class like 35L should be enough to make this project straightforward.

PROJECT OBJECTIVES:

- ensure students have a working Linux development environment.
- ensure students can code, compile, test and debug simple C programs.
- introduce and demonstrate the ability to use basic POSIX file operations.
- introduce and demonstrate the ability to process command line arguments.
- introduce and demonstrate the ability to catch and handle run-time exceptions.
- introduce and demonstrate the ability to return informative exit status.
- demonstrate the ability to research and exploit non-trivial APIs.
- demonstrate the ability to construct a standard `Makefile`.
- demonstrate the ability to write software that conforms to a Command Line Interface (CLI) specification.

DELIVERABLES:

A single compressed tarball (`.tar.gz`) containing:

- a single C source module that compiles cleanly (with gcc, with no errors or warnings).
- a `Makefile` to build the program and the tarball.
- two screen snapshot(s) from a gdb(1) session:
    - `backtrace.png` ... showing a segfault and associated stack-trace
    - `breakpoint.png` ... showing a breakpoint and variable inspection
- a `README` file with identification information, a description of the included files, the smoke-test cases in the `check` target, and any other information about your submission that you would like to bring to our attention (e.g., research, limitations, features, testing methodology).

PROJECT DESCRIPTION:

1. (if you do not already have one) bring up or obtain access to a Linux development environment. Your development environment should include (at least):
    - gcc
    - libc (e.g., glibc or libc6-dev)
    - make
    - gdb

2. (if you are not already familiar with them) study the following manual sections:
    - POSIX file operations ... open(2), creat(2), close(2), dup(2), read(2), write(2), exit(2), signal(2), and this brief tutorial on file descriptor manipulation.

- strerror(3) ... function that interprets the error codes returned from failed system calls.
- getopt(3) ... the framework we will use for argument handling in all projects for this course.
- tar(1) (and the **-z** option) ... program for archiving files in a tarball.
- gdb(1) (and the `run`, `bt`, `list`, `print` and `break` commands in particular) ... a Linux debugger for C/C++ programs.

You will probably find understanding getopt(3) to be the most difficult part of this project. Feel free to seek out other examples/tutorials for these functions, but make sure you cite those sources in your `README`.

3. write a program that copies its standard input to its standard output by read(2)-ing from file descriptor 0 (until encountering an end of file) and write(2)-ing to file descriptor 1. If no errors (other than EOF) are encountered, your program should exit(2) with a return code of 0.

   Your program executable should be called `lab0`, and accept the following (optional) command line arguments (in any combination or order):

   - **--input=**filename ... use the specified file as standard input (making it the new fd0).
     If you are unable to open the specified input file, report the failure (on `stderr`, file descriptor 2) using fprintf(3), and exit(2) with a return code of 2.
   - **--output=**filename ... create the specified file and use it as standard output (making it the new fd1).
     If you are unable to create the specified output file, report the failure (on `stderr`, file descriptor 2) using fprintf(3), and exit(2) with a return code of 3.
   - **--segfault** ... force a segmentation fault (e.g., by calling a subroutine that sets a `char *` pointer to `NULL` and then stores through the null pointer). If this argument is specified, do it immediately, and do not copy from stdin to stdout.
   - **--catch** ... use signal(2) to register a `SIGSEGV` handler that catches the segmentation fault, logs an error message (on `stderr`, file descriptor 2) and exit(2) with a return code of 4.

   When you print out an error message (e.g., because an open failed), your message should include enough information to enable a user to understand not merely the nature of the problem but its cause ... for example:

   - which argument caused the problem ... e.g., **`--input`**
   - which file could not be opened ... e.g., `myfile.txt`
   - the reason it could not be opened ... e.g., no such file

   Do your argument parsing with getopt_long(3). This is, for historical reasons, a somewhat convoluted API, but ...

   - it is very similar APIs are used in many other languages and systems.
   - I want you to gain experience with the very common trial-and-error process of learning how to use a non-trivial API.

   If you encounter an unrecognized argument you should print out an error message including a correct usage line, and exit(2) with a return code of 1.

   To ensure that operations are performed in the right order when multiple arguments are specified, it is suggested that you

   - first process all arguments and store the results in variables
   - then check which options were specified and carry actions out in the correct order:
     1. do any file redirection
     2. register the signal handler
     3. cause the segfault
     4. if no segfault was caused, copy stdin to stdout

   It is relatively easy to generate primitive error messages with perror(3), but if you study the documentation you will see how to get access to the underlying error descriptions, which you could then use with `fprintf(stderr,...` to generate better formatted error messages to the correct file descriptor.

   Note that to use the advanced debugging features of gdb(1) you will need to compile your program with the **-g** option, which adds debugging symbol table information to your program.

4. create a `Makefile` that supports the following targets:
   - (default) ... build the `lab0` executable. To maximize comile-time error checking, you should compile your program with the **`-Wall`** and **`-Wextra`** options.

- `check` ... runs a quick smoke-test on whether or not the program seems to work, supports the required arguments, and properly reports success or failure.
  Please include a brief description (in your `README`) of what checks you chose to include in your smoke-test.
- `clean` ... delete all files created by the Makefile, and return the directory to its freshly untared state.
- `dist` ... build the distribution tarball.

5. run your program (with the **--segfault** argument) under gdb(1)
   - take the fault
   - get a stack backtrace
   - take a screen snapshot (to be included with your submission)

   Putting the code that causes the SEGFAULT in a separate subroutine will make the stack trace a little more interesting.

6. run your program (with the **--segfault** argument) under gdb(1)
   - set a break-point at the bad assignment
   - run the program up to the breakpoint
   - inspect the pointer to confirm that it is indeed NULL
   - take a screen snapshot (to be included with your submission)

   You will not be able to list lines of code, print data, or set breakpoints in gdb unless your program has been compiled with the **-g** (debug symbols) switch.

Summary of exit codes

`0` ... copy successful
`1` ... unrecognized argument
`2` ... unable to open input file
`3` ... unable to open output file
`4` ... caught and received `SIGSEGV`

SUBMISSION:

Your **README** file (for this and every lab) must include lines of the form:

NAME: your name
EMAIL: your email
ID: your student ID

And, if slip days are allowed on this project, and you want to use some, this too must be included in the **README** file:

SLIPDAYS: #days

If, for instance, you wanted to use two slip-days, you would add the following line:

SLIPDAYS: 2

Your name, student ID, and email address should also appear as comments at the top of your `Makefile` and each source file.

Your tarball should have a name of the form `lab0-`yourStudentID`.tar.gz`. You can sanity check your submission with this test script. There will be no manual regrading on this project. Submissions that do not pass the sanity check are likely to receive very low scores.

We will test it on a departmental Linux server. You would be well advised to test all the functionality of your submission on that platform before submitting it.

GRADING:

Points for this project will be awarded:

Value	Feature
	packaging and build

5%	untars expected contents
5%	clean build w/default action
1%	correct `make check`
1%	correct `make clean`
1%	correct `make dist`
2%	reasonability of `README contents`
5%	reasonability of `check` smoke test
	input/output features
10%	correctly copy input to output
5%	correctly implements `--input`
5%	correctly implements `--output`
5%	implements combined `--input` + `--output`
5%	correct handling of un-openable/creatable output file
5%	correct handling of non-existant input file
5%	correct handling of invalid arguments
	fault handling
5%	generate (and die from) SIGSEGV
5%	catch and report SIGSEGV
	gdb use
5%	screen shot showing taking of segfault within gdb
5%	screen shot showing backtrace from segfault
5%	screen shot showing breakpoint stop before fault
5%	screen shot showing inspection of null pointer
	code/package review
2%	correct argument processing
2%	correct file descriptor handling
2%	correct signal handling
4%	misc