# CS 376 Lab 2: Introduction to C

This lab introduces you to the C programming language. It includes exercises that involve separately compiled modules, writing command-line programs, and debugging existing code.

**This lab has a total of 100 possible points.**

# Objectives

Upon completion of the laboratory exercises, you will be able to do the following:

* Copy files to and from the new Virtual Machines and your P drive.
* Write a program that is comprised of separately compiled modules.
* Declare and use one-dimensional arrays in C.
* Access arguments on the command line.
* Manipulate strings in C.
* Gain experience correcting buggy C programs.

**Part 1 - Preliminaries & Learning to Transfer Files**

1. Log into Windows desktop using your campus username and password.
2. After logging in, select the “Engineering Linux Kiosk”
3. You are back at the terminal again this week. To get started for this week, please visit the Moodle site (from within the Engineering Kiosk). Go to the Moodle site for the course and download the starter files for this lab. The name of the starter file is lab2.tar.gz. Download the starter file to the default directory, which is most likely ~/Downloads.
4. Using only the command line, move the lab2.tar.gzstarter files from the ~/Downloads directory to your P\_Drive folder. ***Note:*** ***After you turn off your VM or quit VirtualBox, all files not saved to your P\_Drive folder are deleted. Each new VirtualBox session is started fresh and new; all data gets deleted.***
5. Next, using only the command line again, create a directory called cs376 if you don’t already have one. Create a folder called lab2 inside of your cs376 folder. Move lab2.tar.gz into the lab2 folder.
6. Unarchive the starter files using the following command.

$ **tar –xzvf lab2.tar.gz**

Don’t worry about any possible tar warnings about “Ignoring extended header keyword”. After unarchiving the file, you’ll have a directory called startercode. Inside that directory are two directories, part7 and parts1-4.

If you have problems extracting the files, try:

**$ unzip lab2.zip** (this version of Ubuntu has the unzip utility)

## Checkpoint 1 [10 points]: Show the lab instructor/assistant that the files have been transferred successfully and unarchived.

# Part 2: Make the *main* function the first function in main.c

1. Move into the lab2 directory if you’re not already there. Move into the subdirectory named *parts1-4*. This contains the sorting program from the “Notes on C” document available on the course Moodle site. The main.c file has been modified to include a *printSeparatorLine* function (not yet used), which prints a line with a row of ‘=’ signs.
2. Compile and run the program to confirm that it prints a sorted list of names. Since there is more than one C file, an easy way to compile all the .c files together is to use the Unix wild-card character, ‘\*’:

$ gcc –g \*.c

**What is the name of the output file?**

**Answer:**

1. If you want, you can tell *gcc* to name the executable file something other than the default by using the ‘-o’ switch. For example, this tells *gcc* to name the executable file ‘sort’:

$ gcc –g –o sort \*.c

1. Many C programmers consider it good programming style to have the *main* function be the first function in the file. Open main.c in vim:
2. Move the main function (in main.c) so that its definition comes before the *printSeparatorLine* function.
3. Compile your program (gcc –g –o sortedMain \*.c). Then run the program with the following command. If it does not compile or run, fix the program so that it does.

$ ./sortedMain

## Checkpoint 2 [10 points]: Show the lab instructor/assistant (a) your answer to #2 above and (b) your running program, and (c) let them see your code.

**Part 3: Compiling and Running C Programs Again**

In our Unix environment, we use the gcc command to compile a C program. For example, to compile

myProg.c, we could say:

$ gcc –g myProg.c

The -g indicates that debugging information should be produced. This creates an executable file with the name a.out. This is the default executable file name.

To run the program in Ubuntu Linux, you type ./ followed by the name of the executable file, a.out, followed by the return (or enter) key. When you run the program, it will begin executing in the main function, which should be declared as:

int main(int argc, char \*argv[]) { ... }

1. Type the following gcc command. Notice that you receive an error message because the file myProg.c does not exist.

$ gcc –g myProg.c

1. Create a directory called helloWorld. Navigate into the directory.
2. Use vim to create a C program for printing “Hello, World!”. The program should be named hello.c. Type the following code into the file:

#include <stdio.h>

/\* main program prints Hello, world \*/   
int main (int argc, char\* argv[]) {

printf(“Hello, world!\n”);

return 0;

}

1. Save and quit. Then compile and run the program in the command-line window. Name the output file hello.out. Execute the program.

**What does it do? Is that what you expected?**

**Answer:**

1. Modify the program so that it prints your name 10 times (each on different lines) instead of what the program did before. (Note that the ‘\n’ signifies a newline—it causes each print to occur on a different line.) A *for* loop in C is just like a *for* loop in Java, except you must declare the iteration variable prior to the loop.

int i;

for(i = 0; i < 10; i++) {

/\* body \*/

}

1. Compile and run your modified program.

## Checkpoint 3 [20 points]: Let your lab instructor/assistant see the modified "Hello world" running.

**Part 4: Create and use an array-printing function**

Create a new file printArray.c.

It should define one function, *printStringArray* that:

* 1. takes two arguments, an array of char\* (string) and an *int* that is the size of the array
  2. returns no value
  3. prints the strings in the array from first to last, one per line

Create a new file, printArray.h. It should define a function prototype for *printStringArray*. In main.c, add a preprocessor directive (aka #include) to include the printArray.h file you just created.

Modify the *main* function in main.c so that the array-printing loop is replaced by a call to the *printStringArray* function.

Add a second *printStringArray* call to the *main* function so that the array is printed both before and after the sort. Add a call to *printSeparatorLine* so that the program’s “before sort” and “after sort” listings are separated.

## Checkpoint 4 [25 points]: Show the lab instructor/assistant your running program; let them see your code.

# Part 5: Create a program that prints initial portions of command-line arguments

Recall these facts on accessing the command-line arguments in C programs:

1. If you want to access the command-line arguments from your main function, declare it with two parameters as follows:

int main(int argc, char\* argv[]) { … }

1. The parameter *argc* is an *int* that tells you length of the *argv* array.
2. The parameter *argv* is the array of the command-line arguments:
   1. Index 0 of the array is the name of the command itself
   2. Index 1 contains the first command line argument, etc.

Go back to your *lab2* directory. Create a directory called *part5* and move into it.

## (In this checkpoint, you might find the *atoi* function useful. It converts its string argument to an int; it returns 0 if the string does not denote a valid integer. Be sure to #include <stdlib.h>.)

Create a C program, printEm.c that examines the command-line arguments and prints them as follows:

The first command-line argument should be a positive integer.

1. If it is not a valid integer or it is not positive, print an error message and return.
2. If the first command-line argument is missing, print an error message and return.
3. If the first command-line argument is a valid positive integer (let’s call it N), print the first N characters of each of the subsequent command line arguments, one per line. (If any are shorter than N, just print the entire string.)

For example, if the name of the executable were printEm, then the command

$ ./printEm 3 George Paul Ringo John

should produce the output

Geo Pau Rin Joh

As another example, the command

$ ./printEm Giants Dodgers Reds

should produce an error message, such as

Error: ‘Giants’ must be a positive integer.

As a final example, the command

$ ./printEm

should produce an error message, such as

Error: integer value is missing.

Hint:

int w = 3;

/\* print at most first three characters (safe) \*/

printf("%.\*s\n", w, string);

## Checkpoint 5 [25 points]: Show the lab instructor/assistant your running program; let them see your code

## *Note Part 6 has been removed- I kept the numbering the same because you need to use part 7 in the directory.*

## Part 7: Debugging (yes I intentionally skipped part 6 – did you notice? ;)

Move into the *lab2/part7* directory. Your job is to fix the programs that contain bugs.

When you compile these programs, use the ‘-o’ switch so that the name of each executable is the same as the name of its primary .c file (but without the ‘.c’).

This program should not require fixing.

Program: by\_year

Purpose: To print records of employees hired during (or since) a given year

To compile: gcc -g -o by\_year by\_year.c

Example runs:

by\_year 2000

by\_year 2008

## Checkpoint 7a [5 points]: Show the lab instructor/assistant your running by\_year program.

You will need to fix this program.

Program: in\_order

Purpose: To print records of employees, sorted by first name

To compile: gcc -g -o in\_order in\_order.c sort.c

Example run:

in\_order

(You may want to additionally test it by modifying the code to add more people.)

## Checkpoint 7b [5 points]: Show your lab instructor or assistant your running in\_order program.

You will need to fix this program.

Program: mark\_name

Purpose: To print names of employees, with a particular one marked with a ‘?’

To compile: gcc -g -o mark\_name mark\_name.c

Example runs:

mark\_name 0

marks employee 0 (the first one) with a ‘?’

mark\_name 3

marks employee 3 with a ‘?’

## Checkpoint 7c [5 points]: Show your lab instructor or assistant your running mark\_name program.

## Part 8: Finish up (optional)

Here’s how you can create an archive file of your work, if you wish.

* 1. Move up to the folder containing the lab2 directory (probably a folder named cs376).
  2. Create an archive-file with the contents of the *lab2* directory:

tar -cvf lab2\_update.tar.gz lab2/

* 1. Email the archive file to yourself if you wish.
  2. Close your connection to the remote Linux machine by exiting the Virtual Machine.