

There are 4 lab sections that you can attend each week. You **MUST** attend at least one, and you **MAY** attend multiple sections. Please check the Syllabus on Canvas. You **MAY** ask questions about homework assignments during lab, but **ONLY** if you have completed the lab for that week.

This is an intro lab, with 3 main parts: 1) logging into your linux account, 2) Linux bare basics, and 3) becoming familiar with the C programming language. Note that most students who have taken CSCI 141 and 145 are only familiar with python and java. Some students taking CSCI 247 have also completed CSCI 241 and/or CSCI 301. The labs, homework assignments, and projects in this class do **NOT** assume any experience with C whatsoever.

I. Logging into your Linux account

The language in use in CSCI 247 is C. Unless you have a linux partition (running Ubuntu or CentOS, for example) on your personal computer, you will need to log into your CS account to write, compile, debug, and run your programs that you'll write for this class.

There are many ways that you can log into one of the CS linux machines. Please refer to the *lab1_accessingLinuxPool* pdf file in the files section of Canvas for details how to connect to the CS Linux machines.

All instructions in this lab (and all subsequent labs and homework assignments) assume you are connecting to the linux CS machines. This is the **PREFERRED** way for you to complete your labs and assignments, because gaining familiarity with linux and the command line are among the main objectives of CSCI 247.

You may use an IDE of your choice that supports the C language. jGRASP for example support C. If work on your personal computer using an IDE that supports C, then you are denying yourself the opportunity to learn how to use the command line, which is a skill you need for CS classes later on. In that case, please also practice logging into the CS linux machines.

II. Linux bare basics

Linux is an open-source operating system that can be interacted with by issuing commands via a command line versus by clicking and interacting with icons in a Windows-like environment. Although Linux has a windows-like front-end which can be used to complete this lab, all instructions here are for how to write and run code via the command line only. The commands that are shown below are only a few of the basic commands available to most versions (flavors) of Linux.

The \$ sign and the space to the right of it where you type is called the *command line*. Commands that you type and issue are interpreted by a shell (the default shell, or command line language, in the lab machines is *bash*). It is one of the many shells available in Linux.

One default behavior of bash is that the `$` is prepended with your username and an `@` followed by the name of the computer that you've logged into. For example: `jagodzif@linux-11:~$` specifies the user `jagodzif` logged into the `linux-11` machine.

Nearly all the things that you can do with the mouse when interacting with a windows environment you can do via the command line. In the following steps you'll create a new directory (folder), navigate "into" that folder, copy a file from another directory to your directory, and then access that file using a bare bones editor.

To create a directory, use the `mkdir` command with a single argument that is the name of the directory (folder) that you want to make. Create the directory `csci247`.

```
$ mkdir csci247
```

To confirm that you've made your directory, issue the `ls` command, which will list all contents of the directory where you are currently at.

```
$ ls
```

You should see a list with multiple items, which are the files and/or directories in your account. If done correctly, `csci247` should be listed.

In a windows-like machine, you would double mouse click on a folder to access that folder. In Linux, the command to drill down, or enter, a directory is `cd`, for `change directory`. Enter the `lab1` directory.

```
$ cd csci247
```

Once inside the `csci247` directory, create another directory, `lab1`, and `cd` into it.

Once you've created the directory `csci247`, and the `lab 1` directory inside of it, we now want to fetch a plain text file. One way to do that is to use the `cp` command, which requires two arguments, the first being the source (what you want to copy, the file), and the second being the destination (where you want to copy the data to). The technicalities of copying and moving files among folders and/or computers is beyond the scope of this course, so for the time being simply issue the following command to copy two text files from a remote location to your local (`lab1`) directory. Then issue the `ls` command to make sure the `.txt` files have been fetched.

```
$ cp /home/jagodzif/public_html/teaching/csci247/labs/lab1/helloClass.txt .
```

```
$ ls -l
```

The `ls` command is issued with the `-l` flag, which instructs the output to be displayed in long format, with details about each file.

To inspect (via the command line, without opening the file) the last 10, or first 10 lines of the file seq.txt, issue the following:

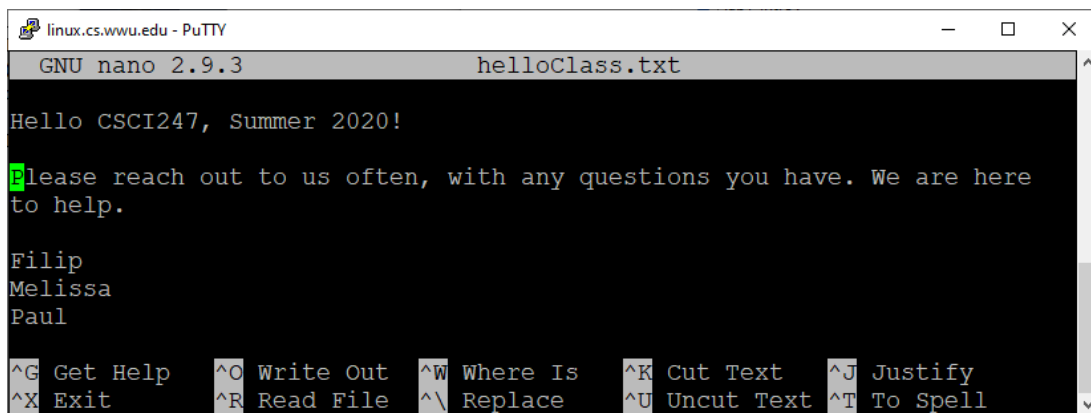
```
$ tail helloClass.txt
```

```
$ head helloClass.txt
```

There are many Linux editors available. The simplest (some say outdated, but simplicity is good!) is *nano*. To open the helloClass.txt file with the nano editor, issue the following:

```
$ nano helloClass.txt
```

You should see something like the following:



Navigate and read the text. At the bottom of the text file, is the “answer” to the lab 1 “quiz” on Canvas.

The “menu” on the bottom of the screen is a shortcut list of commands that you can issue to Exit, Replace, Justify, etc. The ^ symbol means pressing the control button, <ctrl>. Hence to exit nano, you’d press <ctrl> and while holding it down press the x key on the keyboard.

Quit nano (^X), which should return you to the command lint.

Hint : usually, if connecting to a remote computer, it is a good idea to open MULTIPLE connections. One that serves as the command line, and the other as the editor. That way you don’t have to go back and forth saving and exiting and compiling code, and then returning to the editor.

III. Introduction to C

The Objectives of this part of the lab are the following:

- Gain familiarity with using an editor, compiler and debugger
- Gain familiarity with the C programming language constructs and keywords
- Understand basic C data types and their respective sizes
- Gain familiarity with string manipulation in C

In this and future labs, there are a series of question prompts that you should answer. Although you don't submit the answers to these questions, you should still answer the questions. Simply copy and pasting code, and not thinking about the code, will ultimately be detrimental to your learning.

Ask the TAs lots of questions. They are there to help. And, **you CAN work with peers in completing the labs. This is a bit tricky when everybody is remote; you may use slack, email, discord, etc. Up to you.** But, in no circumstances should you SHARE code, which involves emailing code and then submitting it as your own.

Submitting your work

Submit your C program files as the Lab 1 Submission item on Canvas. You must submit your program by the due date/time specified on Canvas.

Create a C source file, compile it, and link it to get an executable

Use the nano editor (or any editor of your choice, such as emacs) to create a file called *lab1Wow.c*, type the following contents and save the file.

```
#include <stdio.h>
int main() {
    printf("Wow! This is my first C Program!\r\n");
}
```

Compile this file into an executable by issuing the following from the command line:

```
$ gcc -o lab1Wow lab1Wow.c
```

What is the gcc Command? What is -o? What do Lab1Wow, and Lab1Wow.c refer to? Ask your TA!

Assuming there are no errors in your code (that the compiler would yell about), run the executable via the following command:

```
$ ./lab1Wow
```

Writing functions in C

Modify the contents of your *lab1Wow.c* file to also include a function `DataTypeSizes`, which takes no arguments, and which prints details about a data type to the screen.

```
/*
Course: CSCI 247 - Summer 2020
File Name: Lab1Wow.c
Name: your name
*/

#include <stdio.h>
#include <string.h>

#define CHAR_DATA_TYPE "char"

void DataTypeSizes(void);

int main() {
    DataTypeSizes();
    return(0);
}

void DataTypeSizes(void){
    printf("\n\"%s\" is a standard C datatype. Size of a \"%s\" data
type is = %ld\r\n", CHAR_DATA_TYPE, CHAR_DATA_TYPE, sizeof(char));
}
```

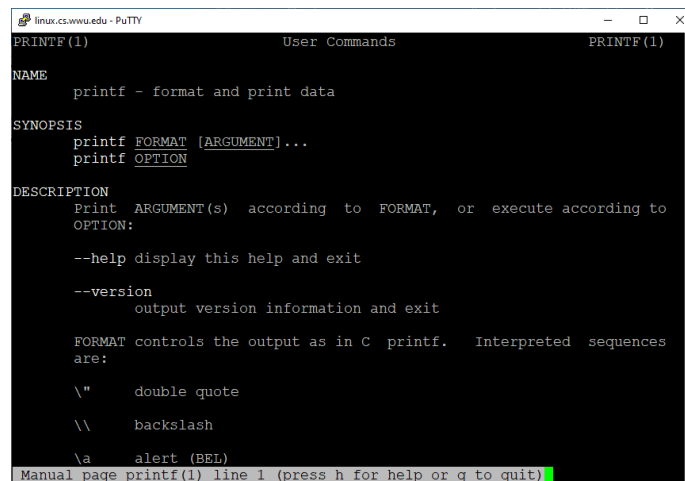
There are several things that you should note. If unsure, ask a TA, or do a web search.

- What are preprocessor directives? – explain what is really happening behind the scenes.
- How are comments (single line, and multi line) entered in C?
- What are forward declarations - Explain why it (they) is/are required.
- What is a macro – Explain some of the advantages of using Macros.
- Functions – Explain why a function call is more advantageous than adding code to the main routine.
- Notice the escape character(\) – Explain why an escape character is needed in a string for some characters
- Operators – what are they in the printf function? How is an operator different from a Macro or Function?
- What is a keyword? What is/are they keywords in this code?
- What does the `printf` function do? What is the significance of the display in the context of your program? (Hint: In Unix based systems all I/O APIs are directed at file based devices. There are 3 file based handles available to a program – `stdin`, `stdout` and `stderr`)

Unsure about `printf`? Do a web search, or, access the manual pages for built-in C functions. To access the manual entry for a function and/or command in C, you would issue from the command line the `man` command, followed by the argument for the manual entry you would want information about.

```
$ man printf
```

The output that you should see is the following:



```
linux.cs.wvu.edu - PuTTY
PRINTF(1)                                User Commands                                PRINTF(1)

NAME
    printf - format and print data

SYNOPSIS
    printf FORMAT [ARGUMENT]...
    printf OPTION

DESCRIPTION
    Print ARGUMENT(s) according to FORMAT, or execute according to
    OPTION:

    --help display this help and exit

    --version
        output version information and exit

    FORMAT controls the output as in C printf.  Interpreted sequences
    are:

    \"    double quote
    \\    backslash
    \a    alert (BEL)

Manual page printf(1) line 1 (press h for help or q to quit)
```

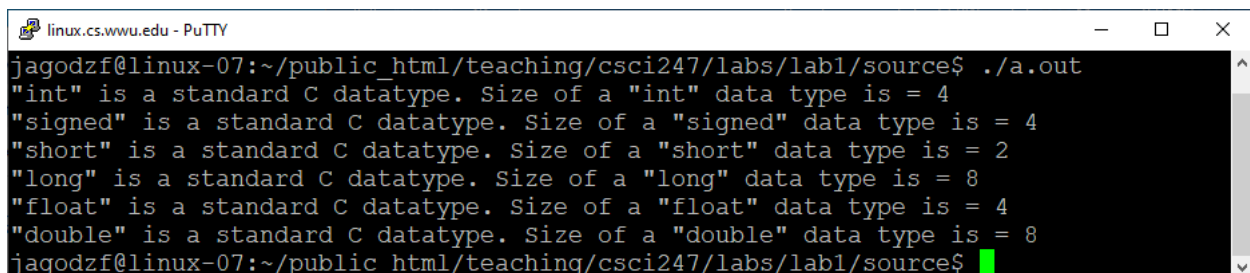
Plain-text manual pages might seem a bit cryptic at first, but they contains lots of good information. Use the space bar to advance to the next screen, and/or use the arrows on your keypad to scroll up or down.

Compile the program, and run it. What is the output?

Basic C data types

Just like python and java have basic data types (`int`, `String`, `float`, etc.), so does C. The data types are a bit different – and there is no `String` data type – but there are many similarities as well.

Modify the `DataTypeSizes` function to display the sizes of the other basic C type specifiers (`int`, `float` and `double`). These data types also allow for modifiers such as “signed”, “unsigned”, “short” and “long”. Experiment with these modifiers and see if it affects the size. Your program should print the sizes of these 6 data types. Sample output is the following:



```
linux.cs.wvu.edu - PuTTY
jagodzif@linux-07:~/public_html/teaching/csci247/labs/lab1/source$ ./a.out
"int" is a standard C datatype. Size of a "int" data type is = 4
"signed" is a standard C datatype. Size of a "signed" data type is = 4
"short" is a standard C datatype. Size of a "short" data type is = 2
"long" is a standard C datatype. Size of a "long" data type is = 8
"float" is a standard C datatype. Size of a "float" data type is = 4
"double" is a standard C datatype. Size of a "double" data type is = 8
jagodzif@linux-07:~/public_html/teaching/csci247/labs/lab1/source$
```

Arrays, passing by reference

The C language does not have a String type. Instead a string is created by allocating memory for a contiguous sequence of characters. The end of a string is identified by a unique character known as the NULL character. You tell the compiler to allocate contiguous memory when you declare an “array” of any basic C type.

In your `main` function declare an array called “str” of 100 characters. Then add a function that uses the `fgets` function to read a string from `stdin`. Call this function from your main program and pass the starting address of the array to this function so it has access to the memory location where your array is located. The function declaration should be as follows:

```
char* GetStringFromStdin( char * str );
```

What does this function return? What does `*` do? Ask the TA!

To prove you have read the user’s input correctly, write another function to display the same string to the console (use `printf`). The function declaration should be as follows:

```
void DisplayString( const char * str);
```

Important: understand the concepts behind passing a variable by reference or by value to a function in the above calls. What is the significance of the “const” qualifier in the above function?

Passing arguments to an executable during invocation

In java, the main method most commonly has the following signature:

```
public static void main(String args[])
```

Where the `args[]` is an array of Strings, and contains the argument(s) that were passed during program invocation. The same thing can be done with C.

All operating systems will have some form of a loader that loads a program into memory and create a stack for it. The stack is a special kind of memory that is used to store variables (function arguments) and return addresses when function calls are made. In C, your “main” program is conceptually not too different from functions you define and call from main and so the loader can pass arguments into your main program. Since all loaders need to follow some standard on what they can pass into the main function, C defines the following function declaration for the “main” function:

```
main(int argc, char* argv[]);
```

The first argument is the count of the number of parameters being passed in and the second argument is an array of character pointers that point to each of the strings in the command line, including the first string that has the name of your program.

If you declare your main function to not have the `argc` and `argv` parameters, you are simply ignoring the values populated on the stack for you by the loader.

Modify your *lab1Wow.c* file to include the `argc` and `argv` parameters and print their values at the very beginning.

Rubric

Remote connecting, lab 1 quiz, entered via canvas	2 points
<i>lab1Wow.c</i> uploaded via Canvas, contains all asked for elements, and runs as intended <ul style="list-style-type: none">• Program Compiles• Function <code>DataTypeSizes</code> written and works as intended; it prints the sizes of the 6 data types• Function <code>GetStringFromStdin</code> written and works as intended, by reading input from the user via the keyboard• Function <code>DisplayString</code> written and works as intended, by printing the user's input back to the screen	8 points