

Lab Objective

- Build familiarity with double pointers
- Understand how strings work in C.
- Practice passing data by reference
- Working with multiple files

Introduction

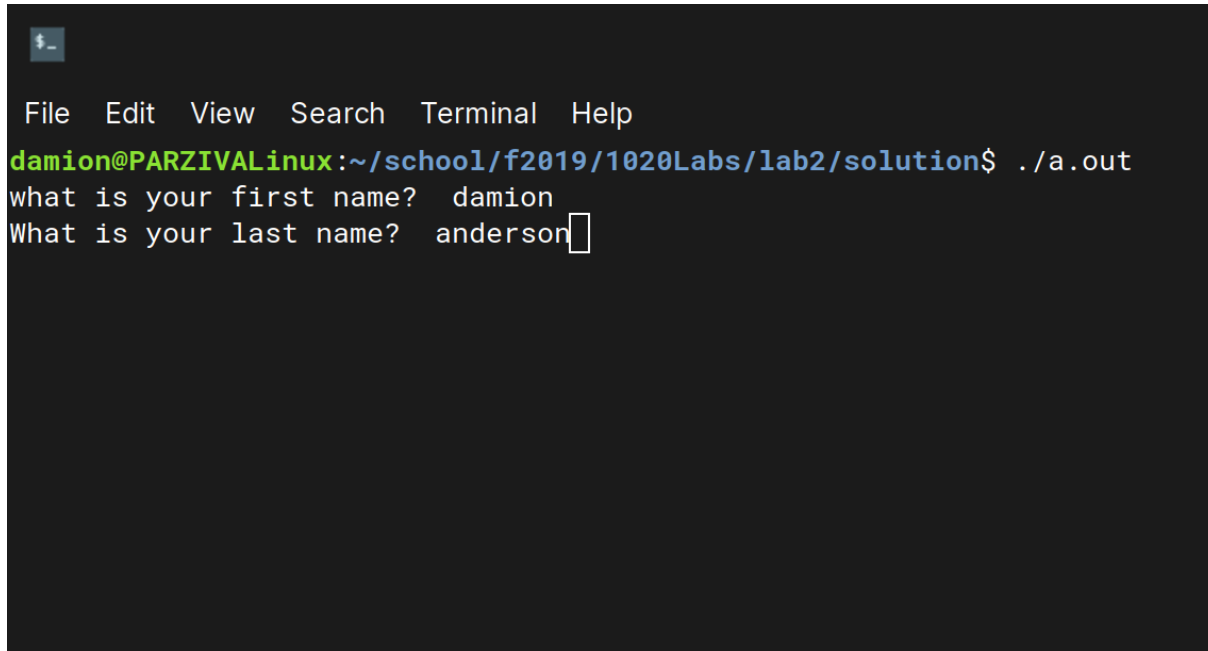
The purpose of this lab is to continue to build familiarity with pointers. In this lab, we will use a double pointer to hold a first and last name. You will then print how long the first and last name is. Lastly, you will print the first and last name in reverse.

You are required to implement the functions as they are given. While you are working through the various steps, you are free to comment out the functions you have not implemented, so you can compile and test your program as you work your way through the lab.

Step 1: Allocate Space for 2D Pointer

To begin this lab, you need to malloc space for a 2D pointer. The size of your 2D pointer will be 2 X 100. Each row of columns will hold a name. The first index of the pointer will hold the first name and the second will hold the last name. Once you have properly allocated space for the pointer, ask the user for input and store the input into the arrays.

Match this format:

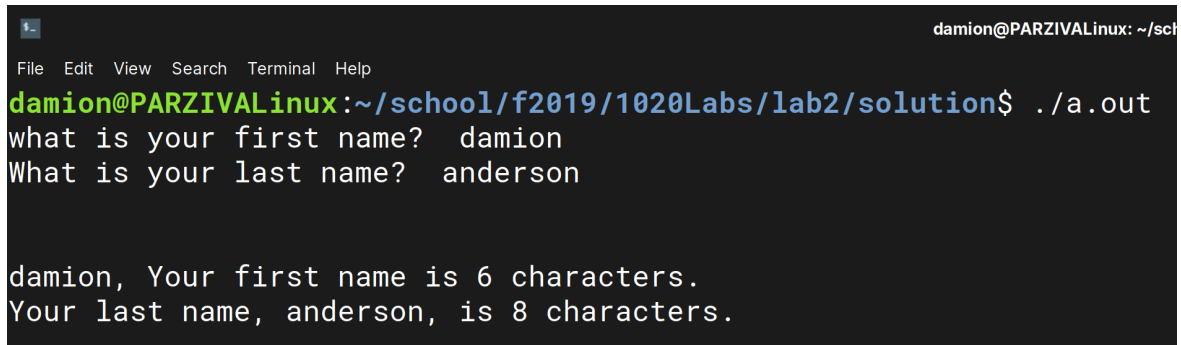
A screenshot of a terminal window with a dark background. At the top left, there is a small icon of a terminal window. Below it, a menu bar contains the words "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal prompt is "damion@PARZIVALinux:~/school/f2019/1020Labs/lab2/solution\$". The user has entered the command "./a.out". The program then asks "what is your first name?" and the user has entered "damion". It then asks "What is your last name?" and the user has entered "anderson". A cursor is visible at the end of the last input.

```
File Edit View Search Terminal Help
damion@PARZIVALinux:~/school/f2019/1020Labs/lab2/solution$ ./a.out
what is your first name? damion
What is your last name? anderson
```

Step 2: Implement the SizeOfName() function.

This function should parse through the text and grab the length of the first name and the last name. You should recall that scanf will add a null character at the end of a captured string.

When you have calculated the length of a string, **pass these variables by reference to printSizeOfName() function**. Inside this function, you should print how long their first and name is. Match the following output:

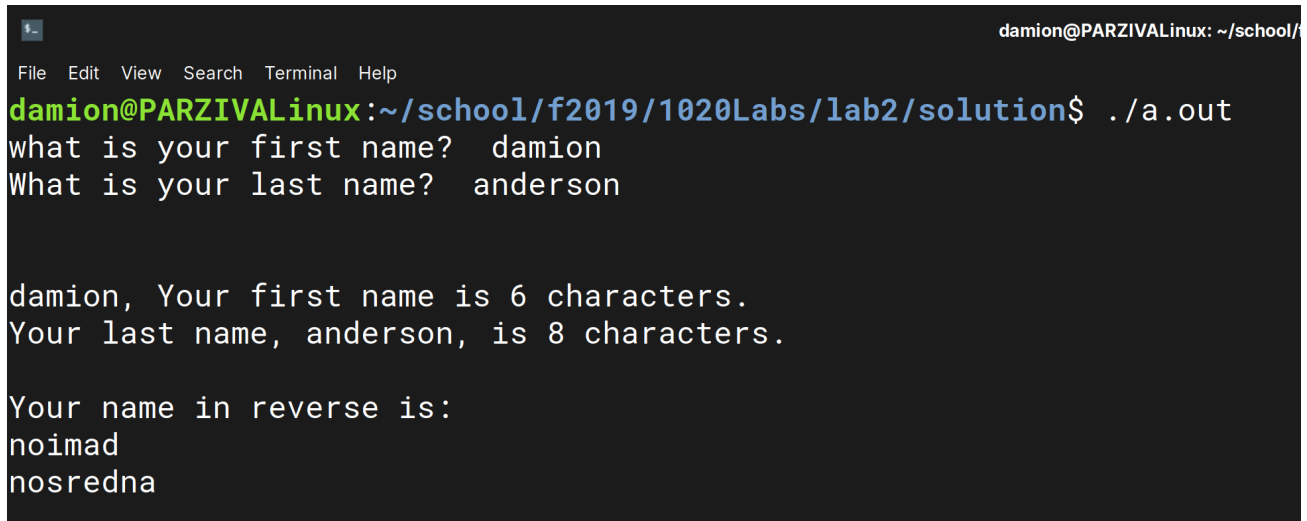
A terminal window with a dark background. The title bar shows 'damion@PARZIVALinux: ~/sch'. The menu bar includes 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The prompt is 'damion@PARZIVALinux:~/school/f2019/1020Labs/lab2/solution\$'. The user runs './a.out'. The program asks 'what is your first name?' and the user enters 'damion'. It then asks 'What is your last name?' and the user enters 'anderson'. The program outputs 'damion, Your first name is 6 characters.' and 'Your last name, anderson, is 8 characters.'

```
damion@PARZIVALinux: ~/sch
File Edit View Search Terminal Help
damion@PARZIVALinux:~/school/f2019/1020Labs/lab2/solution$ ./a.out
what is your first name? damion
What is your last name? anderson

damion, Your first name is 6 characters.
Your last name, anderson, is 8 characters.
```

Final Step: Implement reverseString() Function

Now that you have the functions above implemented and working, your last step is to implement the final function to print the string in reverse order. Here is an example of what the final program should look like.



```
damion@PARZIVALinux: ~/school/
File Edit View Search Terminal Help
damion@PARZIVALinux:~/school/f2019/1020Labs/lab2/solution$ ./a.out
what is your first name? damion
What is your last name? anderson

damion, Your first name is 6 characters.
Your last name, anderson, is 8 characters.

Your name in reverse is:
noimad
nosredna
```

Other Requirements:

In this lab you will be introduced to the use of multi-file programs. We have provided you with main.c. Using the information from this file you will produce 3 files, described below:

1. main.c – This is the driver file. The “main” function will be implemented in this file.
2. functions.h – You will need to move the function prototypes to this file. You are also required to use header guards in this file. Below I have provided a information about header guards and why you need them.
3. functions.c – You will implement the functions in this file.

Multiple Files and Header Guards

At this point you should have already encountered multi-file programs. For those of you that have not we will give you a quick explanation of multiple files and header guards.

The early years of computer programming, where programs were relatively small compared to programs today, is long gone. Writing all code in the driver file does not exhibit good programming practice. It is important that you get experience developing multi-file programs.

Rather than putting your function prototypes in the driver file above the main, you will need to put the function prototypes in the **functions.h** file. So, where do you think the function implementation should go? If you are thinking **functions.c**, you would be correct. Now the next question you should be considering, is how does the driver program (main function) or “Mr. Compiler” know where the function prototypes live. Well we are going to tell them using a #include statement. The syntax of this is as follows:

```
#include “functions.h”
```

This statement should be put in the main and any “.c” file that will use the functions listed in the file.

Since we are providing the implementation of our functions in the **functions.c** file we need to also include the functions.h in the functions.c file.

While working with multi-file programs has many advantages, there is one issue that needs to be addressed. It is not uncommon that a file could be included in another file multiple times. As an example, suppose we include the following in main.c:

```
#include <stdio.h>
#include "functions.h"
#include "anotherFunctions.h"
```

Now suppose anotherFunctions.h includes functions.h

By including functions.h and anotherFunctions.h in your main you have inadvertently included functions.h twice. Which could cause a nasty compile error. This can be avoided by using **header guards** in your “.h” files.

From the following website:

<https://www.learncpp.com/cpp-tutorial/header-guards/>

Header guards are conditional compilation directives that take the following form:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE
```

```
// YOUR PROTOTYPES GO HERE
```

```
#endif
```

#ifndef stands for **if not defined** then you should **define** this file as

#endif means this is the end of the if statement

You should use these header guards in all of your “.h” files.

FORMATTING

You will need to add a header to each of your files like the following:

```
/******
*Your name
*CPSC 1021, your Lab Section, F19
*Your email
*Course Instructor
*****/
```

Your program should compile with no warnings and no errors. If your program will not compile you will receive a **0** on the lab.

- Your code should be well documented. (comments)
- There should be no lines of code longer than 80 characters.
- You should use proper and consistent indentation.

Any infractions of the above listed requirements will result in a 5 point deduction for each infraction category.

Turning in Your Assignment

Handin

You will use the School of Computing **handin** system to submit your lab files.

Prior to handing in your assignment make sure you test your program on the SOC servers. We will grade your assignment on the SOC server and will not accept the excuse that “it ran on my computer”. Your lab must run on the SOC servers.

Most labs will require you to use the **tar** utility. The term “tar” stands for tape archive, an archiving file format. You will use the **tar** utility to “tar zip” (tar.gz) all files prior to submitting through **handin**. Make sure you are inside the current lab2 directory and files are inside the same directory. Using the terminal, **tar** your file as follows:

```
tar -czvf <username>_lab2.tar.gz *
```

The <username> is your username.

Let’s break this down:

tar – the tar utility command

-czvf are flags used by the tar utility

 c – create a new archive containing the specified items

 z – (c mode only) Compress the resulting archive with **gzip**

 v – verbose output (tar will list each file name as it is read from or written to the archive)

 f – read the archive from or write the archive to the specified file

username_lab2.tar.gz - this is the file name. This **must** come before the list of names of the files to **tar**.

*- this is the wildcard. This will archive all files in the current directory. If you do not use the wildcard, you must type the name of each file you want archived.

Once you have tarred your file you should always check the files in your tarred file to determine if they are all there and in workable order. The command to **untar** a file is as follow.

```
tar -zxvf <username>_lab2.tar.gz
```

Useful Linux commands

Command	Purpose
<code>ls</code>	list all contents in your current directory
<code>cd</code>	takes you back to your home directory
<code>cd ..</code>	takes you back one directory
<code>cd dir_path</code>	takes you to the directory specified by the path provided
<code>mv src_file dest_file</code>	renames <i>src_file</i> to <i>dest_file</i>
<code>mv src_file dest_path</code>	moves <i>src_file</i> to the directory specified by the <i>dest_path</i>
<code>mv src_file dest_dir/dest_file</code>	moves <i>src_file</i> to the directory specified by <i>dest_path</i> and gives it the name <i>dest_file</i>
<code>cp src_file dest_file</code>	copies <i>src_file</i> to <i>dest_file</i>
<code>cp src_file dest_path</code>	copies <i>src_file</i> to the directory specified by <i>dest_path</i>
<code>cp src_file dest_dir/dest_file</code>	copies <i>src_file</i> to the directory specified by <i>dest_path</i> and gives it the name <i>dest_file</i>
<code>rm file_name</code>	deletes the file named <i>file_name</i>
<code>mkdir dir_name</code>	creates a directory name <i>dir_name</i> in your current directory
<code>rmdir dir_name</code>	deletes the directory named <i>dir_name</i> if it is empty
<code>rm -rf dir_name</code>	deletes the directory named <i>dir_name</i> (be careful when using <i>-rf</i>)
<code>cat src_file</code>	shows the contents of the file on the screen without opening it in an editor
<code>control-c</code>	send the terminate signal to a running process (kills the current process); good to use if your program is stuck in an infinite loop, for example
<code>ps</code>	shows a listing of processes that are running
<code>kill -9 [pid]</code>	kills the process you specify with the pid (process id #) which is shown when you type ps
<code>man unix_command</code>	displays the manual page (help page) for the specified Unix command
<code>logout</code>	logs you out