CSE2421 Lab 1 PROBLEM

Getting to know LINUX, your choice of editor and the debugger tool

Due: Tuesday, August 30th , 11:30 p.m.
10 Bonus Points (i.e.,10% bonus) awarded if correctly submitted by Monday, August 29th, noon
Any lab submitted up to 24 hours late will be graded and points earned multiplied by 0.75.
Any lab submitted more than exactly 24 hours from the due date will not be graded and scored as a 0.

Determination of whether or not you have met the due date will be via the timestamp indicated within Carmen.

You must read this lab carefully and following the instructions explicitly to get the correct answers. Do not rush through this lab or you will make likely mistakes and loose points. Pay attention to what is going on in this lab. This lab is set up so that you will understand how to use the tools for all subsequent labs. You can rush through this lab and learn nothing from it, or you can think about what you are doing and make the rest of the semester enormously easier for yourself. Your choice.

UNIX versus LINUX

Read the following two articles, if you haven't already:
http://tldp.org/LDP/intro-linux/html/sect_01_01.html (just this page, you don't need to click to any other pages)
http://www.diffen.com/difference/Linux_vs_Unix

Most common difference: UNIX is propriety system (i.e. you must purchase a license) while Linux is an Open Source system. An Open Source system, however, is not always "free". Why?

UNIX/LINUX – What is it?

- Multi-user and multi-tasking operating system.
- Multi-tasking: Multiple processes can run concurrently.
- Example: different users can read mails, copy files, and print all at once (although, only one process can be running at any given time on a single processor, unless the processor has multiple cores). More about this in Systems II. ☺

Logging on using your CSE username and password: If you are already a major and you have a permanent account, be sure to use your already set/used passwords i.e. do not use the default password to logon. If you are logging on for the first time and/or your CSE account is not a permanent one (which means you have a new account created each term), you will need to use the default password for your CSE account; then, you should be prompted to change your password, keeping in mind that your password is case-sensitive. Also, the cursor doesn't necessarily move when you type in your new password. From the CSE labs, you will need to logon to the Windows environment first, and then use the remote login procedures. Remember that LINUX is case- sensitive. Your Windows and LINUX passwords should be the same and are also identical to your OSU name.# account.

COMPUTING SERVICES

If you ever have questions about or problems with your CSE account, email etshelp@cse.ohio- state.edu or visit the Help Desk area in Baker 392. The hours of operation for the Help Desk are here:

http://www.cse.ohio-state.edu/cs/gethelp/helpdesk.shtml

including how to set up a zoom meeting for a virtual visit.

FYI: If you want to create a personal web space, the directions are here:
http://www.cse.ohio-state.edu/cs/resources/web.shtml

COMPUTING LABS

https://cse.osu.edu/computing-services/computing-labs

THE LINUX ENVIRONMENT – stdlinux

You can log on to stdlinux remotely using the remote access method appropriate for your system, or you can go to a CSE lab. It is my understanding that ETS will be opening these labs with some COVID-19 specific caveats.

Once you have a window up with your "home directory" name. For example:

[jones.5684@cse-fl1 ~] $

You are ready to enter LINUX commands. This is called the "command line prompt".

Below are some other good/common commands for you to learn. The first set of characters up until the first space is the command name, the rest is either a description of what should go next (a command, a directory, an option, etc.) or an actual set of characters to type in. You can always use the **man** command to look up more information for a given command. ***Any command on this list or a command that you would have had to use in order to successfully complete any future lab is a "fair game" question on the mid-term.***

The **man** command: **man**ual pages are on-line manuals which give information about most commands

- Tells you which options a particular command can take
- How each option modifies the behavior of the command
- Type **man *command*** at the command line prompt to read the manual for a command

– Try to find out about the **mkdir** command by typing **man mkdir**

- Press the <enter> key to scroll through the manual information line by line
- Press the space bar to scroll through the manual information page by page
- Read what it says about the different options available to you for this command. There will be a question to answer later in this lab.
- Press the letter q to quit the manual information and return to the LINUX prompt. Note the when you return to the LINUX prompt, the manual page information disappears from your screen.

COMMAND (^ is control)	MEANING
*	match any number of characters
?	match one character
whatis command	brief description of a command
chmod [options] file	change access rights for named file
cd	change to home-directory
cd ~	change to home-directory
cd directory	change to named directory
cd ..	change to parent directory

cp file1 file2	**copy file1 and call it file2**
wc file	**count number of lines/words/characters in file**
cat file	**display a file**
tail file	**display the last few lines of a file**
ls -lag	**list access rights for all files**
ls -a	**list all files and directories**
ls	**list files and directories**
who	**list users currently logged in**
mkdir	**create a directory**
mv file1 file2	**move or rename file1 to file2**
man *command*	**read the online manual page for a command**
rmdir directory	**remove a directory**
rm file	**remove a file**
^C	**kill the job running in the foreground**
^Z	**suspend the job running in the foreground**
od	**Dump files in octal and other formats**
jobs	**list current jobs**
kill %1	**kill job number 1**
kill 26152	**kill process number 26152**
ps	**list current processes**

To create a new directory (i.e. folder) for your C programs called cse2421, type the following command and hit enter:

[jones.5684@cse-fl1 ~] $ **mkdir cse2421**

To change to that new directory created above, type the following command and hit enter:

[jones.5684@cse-fl1 ~] $ **cd cse2421**
[jones.5684@cse-fl1 cse2421] $

Notice that the path name in the LINUX prompt has changed after the command.

Create a new directory called **lab1**:

[jones.5684@cse-fl1 cse2421] $ **mkdir lab1**

Type in the following command to go to the lab1 directory:

[jones.5684@cse-fl1 cse2421**] $ cd lab1**
[jones.5684@cse-fl1 lab1**] $**

Return to your home directory:

[jones.5684@cse-fl1 lab1] **$ cd**
[jones.5684@cse-fl1 ~] **$**

Return to the directory you just left (not everyone knows this trick):
[jones.5684@cse-fl1 ~] **$ cd -** *there is a space between the* ***d*** *and the* **-**
/home/jones.5684/cse2421/SU2021/labs/lab1
[jones.5684@cse-fl1 lab1] **$**

So, if you are flipping between 2 directories, you can just continually use "**cd –** ".

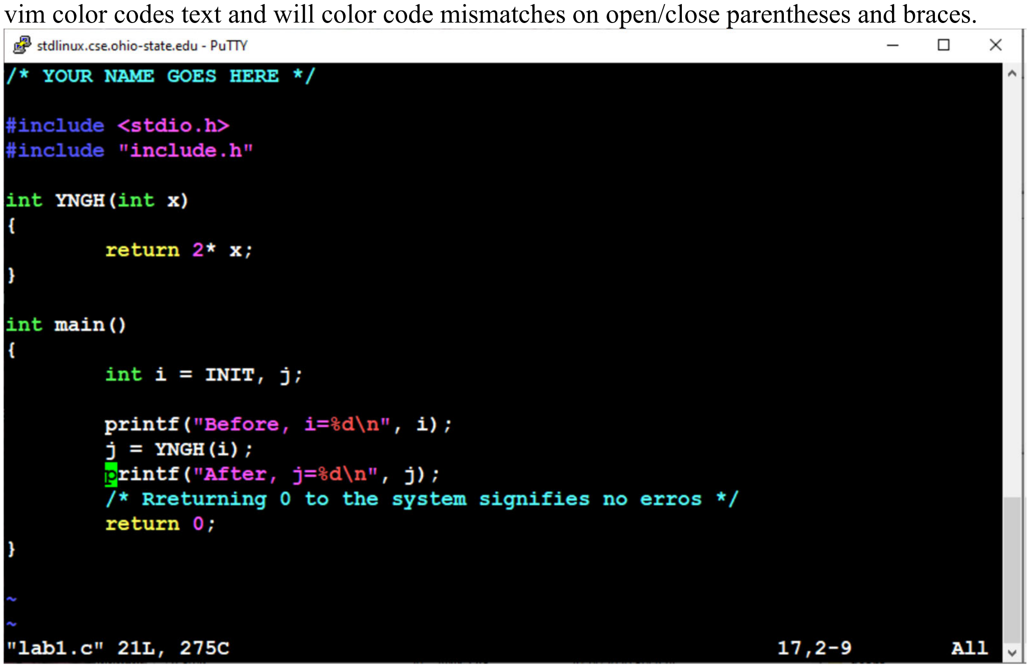
EDITORS
You now need to pick which editor you will use. In piazza in the General Resources section are two vi tutorials . The original editor was vi. That was updated to vim, but "vi" as a command on stdlinux aliases to vim and that is a good thing. There is also gedit.

gedit has appreciably no learning curve. You can look at the interface and figure it out. gedit is by nature graphical, so you need FastX running, or you are doomed.

The vi editor will run in a command window, so no FastX is needed. If you are using putty, vi may be your best choice. I personally develop with 3-4 putty windows open. I use 1 for running make, another for testing code (and using gdb) and 1-2 have vi to edit code.

Tradeoffs:

- vi is strictly keyboard biased, so there is no reaching for a mouse. A decent typist in vi edits more quickly because they never take their fingers off the keyboard. gedit requires constant mouse/keyboard switching.
- vi will run without an X server, so if you have slow link speeds the low resource demand of putty and vi may be your only choice.
- vi has some capabilities that gedit doesn't have, like changing the indent level of an entire block of code with a 2-keystroke sequence **>%** instead of multiple lines of mouse / keyboard /tab / click madness. Or changing every single occurrence of a string with another. For example, **:1,$s/autumn/spring/g** would change each occurrence of the string autumn to spring in the entire file.
- vim color codes text and will color code mismatches on open/close parentheses and braces.



- vi understands tags, allowing you to jump to the definition of a function from any point in the code where the function is called using **^]** (control right bracket).

- gedit is dirt simple but you must teach yourself vi. If you are using vi and forget that you are in edit mode and not insert mode, your keystrokes all have meanings and the act of typing text will do strange and unpleasant things to your file, especially if you look at your keyboard when you type instead of at the screen. vim has multiple undo levels so that's not as scary as it used to be. Using gedit means getting something done right now today but being slower all semester. Using vi means teaching yourself one more thing early on and buying extremely precious time for the rest of the labs.

THE FIRST ENCOUNTER OF A C KIND

The first thing you need to do is get a copy of the lab1.c, lab1_func.c, local_file.h, Makefile and verify files so that you can compile and run lab1 in your stdlinux environment. There are 2 ways to do this: 1) download the files from Piazza or 2) create the files yourself with a text editor as described below. If you have used stdlinux before, option 1 will likely be faster for you. Otherwise, option 2 will likely be faster until you get a little more comfortable with stdlinux. Option 2 will also give you lots of experience using vi, if you choose to learn that text editor. If you would like to try option 1, then skip down this file to the spot where it talks about uploading your program to Carmen. Follow the procedure that explains how to bring a browser up. Go to Piazza rather than Carmen, and then download the needed files.

IMPORTANT: The **&** (ampersand) symbol allows another window to open so that you have a command prompt in one window and your program file in another. This way, you can click on one window to edit and save the program and click on the other window to execute LINUX commands (like compile). The **&** is optional. If you have connected to stdlinux in such a way that you cannot bring up more than one window at a time, DO NOT USE **&** any place below that suggests doing so. Using puTTY is one option that does not allow more than one window at a time.

You can use whatever text editor you would like, but the following directions apply to the gedit editor. Jump to the LINUX EDITORS section at the end of this assignment description for information about other editors; and return to below where you are asked to type in the program. You can use any text editor that you want, just use the command with the filename below and use the **&**.

At the command line prompt, type in:

[jones.5684@cse-fl1 lab1**] $ gedit lab1.c &** [**vi lab1.c &** or some other text editor]

WARNING: Do not ever copy/paste from a .pdf or a docx file to a text file that should contain compliable code because you will likely get some control characters that will cause your program not to work. You'll see lots of compiler errors that don't make any sense on lines of code that look straightforward.

Type the following program exactly as it appears here (except be sure to insert your name where indicated). As you do this, observe how this program is similar/different from what you have seen before in code you have written.

```
/* BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE
  TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH RESPECT TO
  THIS ASSIGNMENT.
  Name:
*/
#include <stdio.h>              /* this tells the preprocessor to copy IO library prototypes
                                   and other information from the file /usr/include/stdio.h    */
#include "local_file.h"         /* this include statement references a file in the current directory    */

int main()
{
```

```
	unsigned int maxEntries;
	int getchar_return_value;		/* note manual page says getchar() returns an integer value	*/

	printf("This program reads in a number, then a series of keyboard characters. The number\n");
	printf("indicates how many characters follows. The number can be no higher than %d.\n", MAX_NUM);
	printf("Then the specified number of characters follow.  These characters can be any\n");
	printf("key on a regular keyboard.\n");

	/* Read the first number entered to know how many entries will follow		*/
	/* We are going to assume that the number will be 1 or 2 digits in length	*/
	/* and it will be followed by a newline,'\n', character			*/

	printf("Please enter the number of entries, followed by the enter/return key: ");
	getchar_return_value = getchar();	/* read the first ASCII character of our max number		*/
	/* if it's not a newline, convert to non-ASCII value, we can assume it is a number		*/
	if (getchar_return_value != '\n'){
		maxEntries = getchar_return_value - '0';
		getchar_return_value = getchar();	/* potentially read the second character of our number	*/
		/* if it's not a newline, convert to non-ASCII value, we can assume it is a number		*/
		if (getchar_return_value != '\n') {
			maxEntries = maxEntries*10 + getchar_return_value - '0';
			getchar();		/* trash the '\n'	*/
		}
	}
	else maxEntries=MAX_NUM;	/* if first ASCII character on the line is a '\n', then assume max		*/

	if (maxEntries > MAX_NUM) {
		printf("Specified number of characters is invalid. It must be between 1 and %d.\n", MAX_NUM);
		return(0);
	}
#ifdef DEBUG
	printf("entering function\n");
#endif
	print_chars(maxEntries);
#ifdef DEBUG
	printf("returned from function\n");
#endif
	return(0);
}
```

NOTES: The quotes are double quotes, not two single quotes side by side

Be sure to hit the enter key after the last curly bracket. The indentation is important for readability and is a programming standard that you should always incorporate when writing your programs – use the tab key to indent. Tabs are typically 8 characters in length; sometimes when there are many levels of indentation, 4 spaces are used rather than a tab.

How is white space (spaces,tabs, new lines/returns,etc.) handled within a program??? They are ignored by the compiler for the most part.

Click the SAVE icon on the secondary menu. Click on the File tab above the secondary menu. A drop down menu appears. Notice there is a “Save As…” option here. Also notice that there are keyboard shortcuts for some of the menu options to the right of each option. Then exit the editor. The window it was in should exit as well.

Use the up arrow key to scroll through previous commands (and the down arrow key to scroll in the opposite direction) to find the **gedit lab1.c &** command then hit the enter key. This will bring your program back up in another terminal window. If you want/need to change the command a little, use the left and right arrow keys to move the cursor through the command characters.

Now create a 2nd file called lab1_func.c with the following text:

```
/* BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE
   TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH RESPECT TO
   THIS ASSIGNMENT.
    Name:
*/
#include <stdio.h>                    /* this tells the preprocessor to copy IO library prototypes
                                         and other information from the file /usr/include/stdio.h     */
#include "local_file.h"               /* this is an include file located in the current directory     */
int getchar_return_value;
void print_chars(unsigned int maxEntries){
int i;
        printf("enter the %u characters: ", maxEntries);

        for(i = 0; i < maxEntries; i++){
                getchar_return_value = getchar();
                if(i==0) printf("The keyboard values are: \n");
                if (getchar_return_value != EOF){              /* we got a valid value          */
                        putchar(getchar_return_value);         /* print it back out             */
                        putchar('\n');

                }
                else{
                        printf("fewer than %u characters entered, number of characters set to %d\n", maxEntries, i);
                        maxEntries = i;
                        break;
                }

        }
        return;
}
```

You'll need a 3rd file called **local_file.h**:

```
/* BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE
   TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH RESPECT TO
   THIS ASSIGNMENT.
   Name:
*/
#define MAX_NUM 25
void print_chars(unsigned int max);
```

To compile the program, type the following command at the LINUX prompt then hit the enter key:

[jones.5684@cse-fl1 lab1] $ gcc -ansi -pedantic -std=c89 -Wimplicit-function-declaration -Wreturn-type –g -o lab1 lab1.c lab1_func.c

gcc -> the compiler command
-ansi -> tells gcc to use the ANSI (American National Standards Institute) C Standard
-pedantic -> tells gcc to strictly adhere to the ANSI C Standard
-std=c89 -> should have same effect as -ansi
-Wimplicit-function-declaration -> warn on implicit declarations*
-Wreturn-type -> warn about return types*
-g -> this tells the gcc command to not delete the symbol table and other debugger important information so that it can be used by gdb
-o <filename> -> this tells the gcc command to put the executable in a file named <filename> rather than

the default filename, a.out.

lab1.c/lab1_func.c -> these are the files that contains the C code the gcc command should compile

* These two items are here because the compiler defaults to being "helpful." Students spend hours of their time and office hour time chasing these things down, so in this class, these two flags are mandatory.

If the compiler gives **errors or warnings**, you've made one or more typos while entering the code above, so go back and correct them, and then resave the edited lab1.c file. Then, recompile. Once the program compiles with **no errors or warnings**, you have an executable which is produced by the compiler, and you can run the program, as explained below. Run the **man** command on **gcc** so that you understand the options used above.

The ansi, pedantic, and the 2 warning flags are **mandatory** for all C code you compile in this class!

Do you think that it would be a good idea to create some sort of tool that would decrease the potential that we could "fat finger" something? Especially if it was going to blow away a file that we had put a lot of work into? There's good news! They did! UNIX and LINUX have a tool called **make**. Not only can it keep us from overwriting our .c files, but it can help us not forget to put something important in our .zip file that we'll be uploading to Carmen later. In order to use the **make** command, we must create a file named **Makefile**. For lab1, the Makefile should look like this (the lines that are indented MUST be indented with a single TAB character. All other lines must start in the first column. Comments within the file are in **blue**. The lines that actually make something happen are in **black**):

```
#comments in a Makefile start with sharp

# BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED
# TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH
# RESPECT TO THIS ASSIGNMENT.

# comments in a Makefile start with sharp
gcc_opt = -ansi -pedantic -Wimplicit-function-declaration -Wreturn-type  -g -c

# target all means all targets currently defined in this file (lab1.zip and lab1)
all: lab1.zip lab1

# this target is the .zip file that must be submitted to Carmen
# this target depends upon 3 items: Makefile lab1.c and LAB1README
lab1.zip: Makefile lab1.c lab1_func.c  local_file.h LAB1README lab1.input1 lab1.input2 verify
#this is the linux command we want make to execute
#if Makefile lab1.c or LAB1README has changed
	zip lab1.zip Makefile lab1.c lab1_func.c local_file.h  LAB1README lab1.input1 lab1.input2 verify
# this target is for the lab1 executable
# the lab1 target gets recreated if lab1.o has changed
lab1: lab1.o lab1_func.o
#this is the linux command we want make to execute
#if lab1.o has changed, note that we only use the -o option here
	gcc lab1.o lab1_func.o -o lab1

# this target is the dependency for lab1.o
# the lab1.o target gets recreated if lab1.c has changed
lab1.o: lab1.c  local_file.h


# this is the linux command we want make to execute
# if lab1.c has changed, note that this is where most of the
# gcc options are and there is now a -c option used
# -c option means that we stop compiling after we create a .o
# file.  We don't compile all the way to the end creating an
# executable.
	gcc $(gcc_opt)  lab1.c

# this target is the dependency for lab1_func.o
# the lab1_func.o target gets recreated if lab1_func.c has changed
lab1_func.o: lab1_func.c local_file.h
```

```
        gcc $(gcc_opt)  lab1_func.c

# this target deletes all files produced from the Makefile
# so that a completely new compile of all items is required
clean:
        rm -rf *.o lab1 lab1.zip

# this target verifies that we have everything we need in the .zip fle
test:
        ./verify
```

The contents of the **verify** file are:

```
echo "files in lab directory"
ls
mkdir test
cp lab1.zip test
unzip lab1.zip -d test
make -r -C test lab1
echo "files in test directory"
ls test
rm -rf test
```

After you have created the verify file, you will have to change the permissions for the file with the following command: **chmod +x verify**. The name of the file should now be green.

Once you have created the Makefile, you will want to create an empty LAB1README file, lab1.input1 and lab1.input2 file. Just use the text editor of your choice to create an empty file by that name. You will put more relevant information into those files later. OR you could use a shell command such as " **>lab1.input1**" The command within the double quotes will create a file with nothing in it.

After you have completed the step above, you can do the following from a linux command line prompt:

```
[jones.5684@cse-fl1 lab1] $   ls
lab1.c  lab1_func.c  lab1.input1  lab1.input2  LAB1README local_file.h  Makefile  verify
[jones.5684@cse-fl1 lab1] $   whatis touch
touch (1)            - change file timestamps
[jones.5684@cse-fl1 lab1] $   make -r                          ALWAYS use the -r option!
zip lab1.zip Makefile lab1.c lab1_func.c local_file.h  LAB1README lab1.input1 lab1.input2 verify
  adding: Makefile (deflated 58%)
  adding: lab1.c (deflated 54%)
  adding: lab1_func.c (deflated 42%)
  adding: local_file.h (deflated 17%)
  adding: LAB1README (deflated 48%)                    The % numbers you see may vary a little
  adding: lab1.input1 (stored 0%)
  adding: lab1.input2 (stored 0%)
  adding: verify (deflated 39%)
gcc -ansi -pedantic -g -c lab1.c
gcc -ansi -pedantic -g -c lab1_func.c
gcc lab1.o lab1_func.o -o lab1
[jones.5684@cse-fl1 lab1] $    ls
lab1  lab1.c  lab1_func.c  lab1_func.o  lab1.input1  lab1.input2  lab1.o  LAB1README  lab1.zip  local_file.h
Makefile  verify
[jones.5684@cse-fl1 lab1] $                            Note that there are now 4 new files that just
                                                       happen to be the two targets of the all
                                                       option of the Makefile and 2 .o files
```

Now say:

```
[jones.5684@cse-fl1 lab1] $    make -r
make: Nothing to be done for `all'.                    No files have changed, so no recompiling
```

jones.5684@cse-fl1 lab1] $ **touch lab1_func.c**
jones.5684@cse-fl1 lab1] $ **make -r**
zip lab1.zip Makefile lab1.c lab1_func.c local_file.h LAB1README lab1.input1 lab1.input2 verify
updating: Makefile (deflated 58%)
updating: lab1.c (deflated 54%)
updating: lab1_func.c (deflated 42%)
updating: local_file.h (deflated 17%)
updating: LAB1README (deflated 48%)
updating: lab1.input1 (stored 0%)
updating: lab1.input2 (stored 0%)
updating: verify (deflated 39%)
gcc -ansi -pedantic -g -c lab1_func.c
gcc lab1.o lab1_func.o -o lab1
[jones.5684@cse-fl1 lab1]$

Notice lab1_func.c was recompiled and the .zip file updated

jones.5684@cse-fl1 lab1] $ **make -r clean**
rm -rf *.o lab1 lab1.zip
[jones.5684@cse-fl1 lab1] $ **ls**
lab1.c lab1_func.c lab1.input1 lab1.input2 LAB1README local_file.h Makefile verify
[jones.5684@cse-fl1 lab1] $

What did the **make -r clean** command do?

We'll work with the **make** command more later in the semester. For now, let's use **make** again so that we've got an executable version of our program again and run the program.

[jones.5684@cse-fl1 lab1**] $ lab1**

*you may have to use **./lab1** rather than **lab1** Which one you must use will depend upon how your $PATH shell variable was set up by ETS.*

You should see the output:

This program reads in a number, then a series of keyboard characters. The number indicates how many characters follow. The number can be no higher than 25. Then the specified number of characters follows. These characters can be any key on a regular keyboard.
Please enter the number of entries, followed by the enter/return key:

on the screen. If you enter **6** at the prompt, you should see:

enter the 6 characters:

then enter **banana** and you should see:

The keyboard values are:
b
a
n
a
n
a
[jones.5684@cse-fl1 lab1]$

WORKING WITH GDB

(Note # beside instructions to better help you answer questions in the readme file)

1. Before moving forward, skim the document from **Piazza -> Resources->General Resources** marked **A better (and condensed) gdb guide** and look at the **Extended_ASCII_Table.pdf** file from the **Piazza-> Resources->General Resources** area. Pay special attention to the hexadecimal values that are related to upper/lower case letters, numeric digits, the space, and the newline characters. Can you determine a relationship between upper- and lower-case alpha characters? Can you determine a relationship between the ASCII representation (in hexadecimal) of digits with their actual value?

2. Enter the following command:
   [jones.5684@cse-fl1 lab1] $ **gdb lab1** ← **if you'd rather use ddd instead of gdb, that's fine. Note this will mean the commands below will change some, but you should be able to do it.**

You will enter the debugger running the lab1 program. You will see a new prompt at which you should enter a new command:

GNU gdb (GDB) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab1...done.
(gdb) **break main**

This command will set a breakpoint at the beginning of your lab1 program. You should see a response similar to the one below:

Breakpoint 1 at 0x400615: file lab1.c, line 15.
(gdb)

The line numbers you see might be a little different than what are described here depending upon exactly how you typed your version of this program in to your lab1.c file. If we knew that main() was on line 15 in our file, we could have used **break 15** rather than **break main**.

3. Now we want to begin running the program so:
   (gdb) **run**

You should see some output that tells you about hitting the breakpoint we set above. It should look similar to:

Starting program: /home/jones.5684/cse2421/SU2021/labs/lab1/lab1

Breakpoint 1, main () at lab1.c:15
15 printf("This program reads in a number, then a series of keyboard characters. The number\n");
(gdb)

We can now run the program instruction by instruction by using the **next** command. Note that the statement printed out is the very first executable statement in the program. The statements prior to this one were declarative in nature. Also note that this statement has not yet executed because we don't have this line of output displayed on the screen yet. If we say **next** at the (gdb) prompt, we will see:

(gdb) **next** **"next" or just "n" both work here**
This program reads in a number, then a series of keyboard characters. The number
16 printf("indicates how many characters follows. The number can be no higher than %d.\n", MAX_NUM);
(gdb)

Notice that the output from the printf statement on Line 15 ***now*** prints out, and then **gdb** prints out the line of code that will be executed *next*.

IMPORTANT: Each time you use the **next** command, a line of code will be printed. This is the line of code that will be executed **next**; it has not yet executed; only the lines of code prior to the one just printed have executed.

NOTE: If you just use the return key at the (gdb) prompt, the last command you executed will be repeated.

Step through the program until you see:

24 printf("Please enter the number of entries, followed by the enter/return key: ");
(gdb)
25 getchar_return_value = getchar(); / read the first ASCII character of our max number */*
(gdb)
Please enter the number of entries, followed by the enter/return key: ***15*** (Enter the **15** here. Note the difference between when the printf() statement is listed in the output vs when the output of that printf statement shows up.)
27 if (getchar_return_value != '\n'){
(gdb)

4. Then say:

(gdb) **print getchar_return_value**
$1 = 49 (Is this is the value you'd expect to see in this variable?)
(gdb) **print /x getchar_return_value**
$2 = 0x31 (Why do you think the output is different?)
(gdb) **print /c getchar_return_value**
$3 = 49 '1' (Does the /c option help you understand the output?)

The **print** command will show us the current value of any **in-scope** variable name. So that we can continuously watch the value of the variable getchar_return_value change over time, let's use a another gdb command called **disp**:

(gdb) **disp /x getchar_return_value**
1: /x getchar_return_value = 0x31
(gdb) **disp /c getchar_return_value**
2: /c getchar_return_value = 49 '1'

Now let's keep going…
(gdb) **next**
28 maxEntries = getchar_return_value - '0';
1: /x getchar_return_value = 0x31
2: /c getchar_return_value = 49 '1'

If we subtract '0' from getchar_return_value, what value do we have? We'll have to use next one more time before the maxEntries variable has a valid value in it. Let's do that, then print maxEntries. What is the value?

What do you think the line:
32 maxEntries = maxEntries*10 + getchar_return_value - '0';

will do?

5. After we say:

(gdb) **next**
33 getchar(); /* trash the '\n' */
We can now check the new value of maxEntries. What is it?

Note that we kept seeing the values for getchar_return_value as we executed each line of this program even when the value doesn't change. So that we don't have to go through the program line by line, we are now going to create a breakpoint at the line in the code where we call print_chars().

(gdb) **break print_chars**

Breakpoint 2 at 0x4006e0: file lab1_func.c, line 11.

Note that even though the code for the print_chars() program is in a different file, gdb can find it. Also, note that gdb says this is Breakpoint 2. We'll use that number later. Now we can use the continue command rather than stepping through the program line by line…

(gdb) **continue** *continue means to run all of the code until the next breakpoint (which in this case, is when we enter print_chars())*

Continuing.
Breakpoint 2, print_chars (maxEntries=15) at lab1_func.c:11
11 printf("enter the %u characters: ", maxEntries);

Note that **gdb** shows us what the value of the parameter we're passing in to print_chars() is. If there is more than one parameter, **gdb** will show us all of them.

(gdb) **next**
13 for(i = 0; i < maxEntries; i++){
(gdb) **next**
14 getchar_return_value = getchar();
gdb)
enter the 15 characters:

Note that the prompt to enter the 15 character doesn't show up until just now even though we executed the printf() statement to print it a few instructions up. This is not uncommon. Use "**1234567 peaches"** as input to the prompt above. You should see the following output:

(gdb)
enter the 15 characters: 1234567 peaches
15 if(i==0) printf("The keyboard values are: \n");
(gdb)

Note that gdb isn't printing out getchar_return_value anymore. This is because the variable we're printing is the one in main(), the variable of the same name in print_chars() is a totally different variable so we must treat it as such.

(gdb) **disp /x getchar_return_value**
3: /x getchar_return_value = 0x31
(gdb) **disp /c getchar_return_value**
4: /c getchar_return_value = 49 '1'

Record the new value of getchar_return_value to put in your lab1Readme file later. Is it what you expected?

6. Now say:
(gdb) break 15
Breakpoint 3 at 0x400708: file lab1_func.c, line 15.
(gdb)

We said break 15 here because we want to break on line 15 of the function we're currently in. We know it i s line 15 because that's what the line number is on the statement above.

(gdb) **continue**

You should see:

```
Continuing.
The keyboard values are:
1

Breakpoint 3, print_chars (maxEntries=15) at lab1_func.c:15
15              if(i==0) printf("The keyboard values are: \n");
3: /x getchar_return_value = 0x32
4: /c getchar_return_value = 50 '2'
(gdb)
```

You can now use the **continue** command repeatedly to repeat the go through the for loop watching <u>and recording</u> each value that is read in. Is it always consistent with what is printed?

After the program reads in the s in peaches, you will see:

```
(gdb)
Continuing.
s                    ←Note that the last character input is printed at this point.
[Inferior 1 (process 24363) exited normally]
(gdb)
```

This means that the program has completed successfully. The process number will change each time you run the program. So, don't be concerned when the process number changes.

7. We will now run the program again, using input **6** and **banana** this time.
(gdb) **run**
Starting program: /home/jones.5684/cse2421/SU2021/labs/lab1/lab1

```
Breakpoint 1, main () at lab1.c:15
15          printf("This program reads in a number, then a series of keyboard characters. The number\n");
1: /x getchar_return_value = 0x0
2: /c getchar_return_value = 0 '\000'
3: /x getchar_return_value = 0x0
4: /c getchar_return_value = 0 '\000'
(gdb)
```

Note that gdb is trying to print values for both the variable getchar_return_value (1 & 2) in the main() function and also the variable getchar_return_value (3&4) in the print_chars() function.

Now, let's get rid of the breakpoint at the beginning of the program by saying:

(gdb) **clear main**   (if we had wanted to get rid of the breakpoint at line 41, we'd use **clear 41**)

Let's set a new breakpoint at line 45.

(gdb) **break 45** (This should be the line in main that calls print_chars())
Deleted breakpoint 1 Breakpoint 4 at 0x4006bf: file lab1.c, line 45.
(gdb)

And let's get rid of the break point at the entry to print_chars(). Remember that gdb called it Breakpoint 2? We can use that number to clear the breakpoint, but we don't use the **clear** instruction, we use **delete** thus:
(gdb) **delete 2**

If you want to get rid of all breakpoints just say **delete** with no specifier. Let's do that now and then reset a breakpoint only on line 45.
(gdb) **delete**
Delete all breakpoints? (y or n) y
(gdb) **break 45**
Breakpoint 5 at 0x4006bf: file lab1.c, line 45.
(gdb)

Use the input **6** and **banana** as the input to the program this time.
(gdb) **run**
The program being debugged has been started already.
Start it from the beginning? (y or n) **y**
Starting program: /home/jones.5684/cse2421/SU2021/labs/lab1/lab1
This program reads in a number, then a series of keyboard characters. The number
indicates how many characters follows. The number can be no higher than 25.
Then the specified number of characters follow. These characters can be any
key on a regular keyboard.
Please enter the number of entries, followed by the enter/return key: **6**

Breakpoint 5, main () at lab1.c:45
45 print_chars(maxEntries);
1: /x getchar_return_value = 0xa
2: /c getchar_return_value = 10 '\n'
3: /x getchar_return_value = 0x0
4: /c getchar_return_value = 0 '\000'
(gdb)

Notice that the next instruction we're going to execute is the call to print_chars().

Let's say next and see what happens.
(gdb) **next**
(gdb) next
enter the 6 characters: banana
The keyboard values are:
b
a
n
a
n
a
49 return(0);
1: /x getchar_return_value = 0xa
2: /c getchar_return_value = 10 '\n'
3: /x getchar_return_value = 0x61
4: /c getchar_return_value = 97 'a'
(gdb)

Notice that we didn't see any of the instructions within the print_chars() function. It's because we used the gdb command **next**. If we want to go through the instructions in a function, we must use a different gdb command called **step**. Let's run the program again and use step instead:

(gdb) **run**
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/jones.5684/cse2421/SU2021/labs/lab1/lab1
This program reads in a number, then a series of keyboard characters. The number
indicates how many characters follows. The number can be no higher than 25.
Then the specified number of characters follow. These characters can be any
key on a regular keyboard.
Please enter the number of entries, followed by the enter/return key: **6**

    Breakpoint 5, main () at lab1.c:45
    45   print_chars(maxEntries);
    1: /x getchar_return_value = 0xa
    2: /c getchar_return_value = 10 '\n'
    3: /x getchar_return_value = 0x0
    4: /c getchar_return_value = 0 '\000'
    (gdb) **step**
    print_chars (maxEntries=6) at lab1_func.c:11
    11   printf("enter the %u characters: ", maxEntries);
    3: /x getchar_return_value = 0x0
    4: /c getchar_return_value = 0 '\000'
    (gdb)

Now we're inside print_chars()! Put the breakpoint back on line 15 and continue:

    (gdb) **break 15**
    Breakpoint 6 at 0x400708: file lab1_func.c, line 15.
    (gdb) **continue**
    Continuing.
    enter the 6 characters: banana

    Breakpoint 6, print_chars (maxEntries=6) at lab1_func.c:15
    15   if(i==0) printf("The keyboard values are: \n");
    3: /x getchar_return_value = 0x62
    4: /c getchar_return_value = 98 'b'
    (gdb)

Let's change the value of the variable.

    (gdb) **set getchar_return_value = 64**
    (gdb) **continue**

What value prints? Is it 'b' or something else?

You can now use the **continue** command repeatedly to repeat the go through the for loop watching <u>and recording</u> each value that is read in and printed.

Exit the debugger:

  (gdb) **quit**

You now know how to enter and exit the debugger, set up breakpoints, and print variables and set variables to values different than they are assigned within the code. (Yes, you really do know how to do all these things.) All of these tools will be useful to you throughout this semester and throughout the rest of your career. If I ask you questions about how to do these things on the midterm or final, you should be able them.

8. Create 2 additional files in your lab1 directory called **lab1.input1** and **lab1.input2**.

lab1.input1 should contain 3 lines:
10
12345
ABCDE

lab1.input2 should contain 2 lines
10
banana

9. From a linux prompt run the following 2 commands:
   [jones.5684@cse-fl1 lab1] $ **lab1 < lab1.input1**
   [jones.5684@cse-fl1 lab1] $ **lab1 < lab1.input2**

Do you get the output that you expected?

10. Go back in to gdb, set breakpoints like you did in prior steps, then, rather than just saying **run** at the gdb prompt, use this command:

(gdb) **run < lab1.input1**

1. This instruction tells the debugger to run the program currently being debugged (**lab1** in this case), but to take all program input from the file **lab1.input1** rather than from the keyboard. Go through the program looking at how the variables **maxEntries** and **getchar_return_value** are set using the input from lab1.input1. Are the values set to the values that you expected? Why or why not? Did you learn anything that you didn't know about how input is interpreted? What value showed up between the **5** and the **A**?

2. Do the same with the file **lab1.input2**. Are the values set to the values that you expected? Why or why not?

3. Now that you've used gdb, exit out of the program and say **ddd lab1.** Can you figure out how to set a breakpoint at this (***printf("The keyboard values are: \n");***) line of the program using the graphical interface?

CREATING LAB1README

Create a text file in the lab1 directory called LAB1README using the template on Piazza. **The filename must be exactly as stated here. Any other filename will not be considered correct, and the content will not be graded**. Answer the questions according to what you read and what you saw while using gdb and the linux commands specified above.

LAB SUBMISSION

Always be sure your linux prompt reflects the correct directory or folder where all of your files to be submitted reside. If you are not in the directory with your files, the following will not work correctly.

You must submit all your lab assignments electronically to Carmen in .zip file format. The format of zip command is as follows:

[jones.5684@cse-fl1 lab1] $ **zip <zip_filename> <files-to-submit>**

where <zip_filename> is the name of the file you want zip to add all of your files to and <files-to-submit> is a list of the file(s) that make up the lab. Remember that you have to be at the correct location (the designated directory) for the command to be able to find your <files- to-submit>. This is why we created a “lab1” directory so that all of the files could be stored in the same place. Thus, when executing the command above, we would need to be in the ~/cse2421/lab1 directory for it to work. **Do not zip at the directory level.**

When you run **make**, this happens automatically.

TEST TO CONFIRM THAT YOUR .zip FILE HAS EVERYTHING IT NEEDS

Before uploading the .zip file to Carmen, you should do the following:

1. Create a new directory in **lab1** called **test**.
2. Copy the lab1.zip file to **test**. (e.g., **cp lab1.zip test)**
3. Enter the **test** directory (e.g., **cd test)**
4. Unzip the file. (e.g. **unzip lab1.zip)**
5. Inspect the files that are now in the directory **test**. Are all the files you want to upload to Carmen present in the directory? If so, then you should be able to execute **make -r lab1** and a lab1 executable should be correctly created in the **test** directory. Now your zipfile is complete and you can return to lab1 and follow the directions below. If **make** didn’t work, then remove all files from the test directory (you want this directory to be empty), go back up and correct your zip command in your **Makefile** and try again. Since you were supplied with the Makefile for this lab, there should not be a problem, but you will want to perform the process for all other labs to ensure your lab submission is complete.

The procedures are in the file called verify. You can use that file, or you can execute each one of the steps as specified here**.**

The file called **verify** does the above steps for you. You can either run it by saying **./verify** or by saying **make test**.

IMPORTANT: YOU SHOULD PERFORM THIS STEP FOR EACH LAB THAT YOU SUBMIT TO ENSURE THAT YOU HAVE INCLUDED ALL FILES YOU NEED. REMEMBER THAT YOU WILL RECEIVE A 0 ON THE LAB IF IT DOESN’T COMPILE WITHOUT ERRORS OR WARNINGS. IF YOU DON’T SUBMIT ALL THE FILES, IT WON’T COMPILE.

UPLOADING FILES TO CARMEN

The easiest way to upload files specific to the stdlinux environment is by opening Carmen from stdlinux and uploading the corresponding .zip file from there. To open Carmen from stdlinux:

i) login to stdlinux using CSE remote access.
ii) Open terminal
iii) Type **firefox &**. (There is a space between x and **&**.) Click **enter**. A new window will open with firefox web browser in it. NOTE: firefox is the browser recommended by Carmen administration. If you are an experienced Linux user, you may use another browser at your own risk. (**google-chrome** is also an option.)
iv) Navigate to carmen.osu.edu from that window.
v) Navigate to the Carmen assignment for which you want to upload files
vi) Use given links to upload assignments to Carmen.

NOTE:

- Your programs MUST be submitted in source code form. Make sure that you zip all the required .c files for the current lab (and .h files when necessary), and any other files specified in the assignment description. Do NOT submit the object files (.o) and/or the executable. The grader will not use executables that you submit anyway. She or he will build/compile your code using the appropriate compile command, and run the executable generated.

- DO NOT zip the whole directory.

- It is YOUR responsibility to make sure your code can compile and run on CSE department server **stdlinux.cse.ohio-state.edu,** using **gcc -ansi -pedantic -std=c89 -Wimplicit-function-declaration -Wreturn-type –g** without generating any errors or warnings or segmentation faults, etc. **Segmentation Faults are applicable to this rule only when valid input is used with your program**. Any program that generates errors or warnings when compiled or does not run without system errors will receive 0 points. No exceptions!
- There is a second way to end up with 0 points on your lab: by not including the certification wording listed above in your README file or any other file you submit for labs. Just to ensure that you know what the certification wording is:

  BY SUBMITTING THIS FILE TO CARMEN, I CERTIFY THAT I HAVE STRICTLY ADHERED TO THE TENURES OF THE OHIO STATE UNIVERSITY'S ACADEMIC INTEGRITY POLICY WITH RESPECT TO THIS ASSIGNMENT.

  This certification must be at the top of any and all files you submit for a lab or homework assignment.

*** More information about lab requirements, point deductions, due dates, late assignments, etc., will be designated with each specific lab.

LOGGING OUT

To exit the terminal window, type the following command at the LINUX prompt then hit the enter key:

% **exit**

Be sure to logoff your account by choosing the menu option "System" tab, then "Logout" from the drop down menu and confirm.

LINUX EDITORS

1. Vim – type vi at the command line prompt

   Home Page: http://www.vim.org/

   Written in: C and Vim script.

2. gedit – type gedit at the command line prompt; can also access gedit through the menu system: Applications , Accessories, then gedit Text Editor

   Home Page: http://projects.gnome.org/gedit/

   Written in: C, Python

3. Nano – type nano at the command line prompt

   Home Page: http://www.nano-editor.org

4. gVim – type gvim at the command line prompt

   Home Page: http://vimdoc.sourceforge.net/htmldoc/gui.html

5. Emacs – type emacs at the command line prompt; can also access xemacs from the menu system:

Applications, Other, Emacs

Home Page: http://www.gnu.org/software/emacs/

MORE LINUX COMMANDS and examples

http://www.thegeekstuff.com/2010/11/50-linux-commands/

UNIX TUTORIAL RESOURCES

- http://www.ee.surrey.ac.uk/Teaching/Unix
- http://www.math.utah.edu/lab/unix/unix-tutorial.html
- http://www2.ocean.washington.edu/unix.tutorial.html

To learn more about the Bash shell

- BASH (Bourne Again SHell) is the Linux default shell. It can support multiple command interpreters.
- http://www.gnu.org/software/bash/manual/bashref.html#Bourne-Shell-Variables