Evil Hangman

Prerequisite Knowledge:

This lab assumes you have completed Computing I at UMass Lowell and that you already have a CS account. If you do not have a CS account then please fill out the appropriate form and give it to your TA. The corresponding lab from Computing I that has bearing on this lab is given in full at the end of this document in case you need to review.

Directory structure:

1. Make a directory called “Spring2018” in your login directory and change to that directory. (relevant commands mkdir and cd).
2. Inside the Spring2018 directory make another directory called “COMP1020” and change to that directory.
3. Finally, make one more directory called “HANGMAN” and change to that directory. Type the command “pwd” and hit enter. pwd stands for print working directory. If you have done the previous steps correctly you should see something like the following but your directory structure will replace the /usr/cs/fac1/dbadams portion:

/usr/cs/fac1/dbadams/Spring2018/COMP1020/HANGMAN

The make utility:

In this portion of the lab we will begin setting up our environment so that we can quickly compile, clean up, test and run our lab project. We will begin by making a simple hello world program. Use your favorite editor (vi or emacs) to create and edit a file named “main.c”.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello world!\n");
   return 0;
}
```

We could compile the program from the command line using the command “gcc main.c” which would create an executable named a.out in the current working directory. Do this now and you should be able to execute the file simply by typing its name with the current directory before it as in “./a.out”. The resulting output should look like:

```
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ ./a.out

Hello world!
```

Remove the executable file using the command “rm a.out”.

For most of the semester we will want to run gcc with the -Wall and --std=c99 options at a minimum. The following command line will create an executable named hello instead of one named a.out.

gcc -Wall --std=c99 -o hello main.c

We can use a utility program called make to make the compilation process much easier than typing everything out every time. It will especially help as we start moving towards projects that use multi-file compilation.

Use your favorite editor to create and edit a file named “Makefile”. Be sure that the name of your file begins with a capital letter M. Edit your Makefile so that it looks like the following:

```
hello: main.c
	gcc -Wall --std=c99 main.c -o hello
```

This forms what the make utility calls a rule. The following is an edited excerpt from the gnu make utility manual:

A simple makefile consists of “rules” with the following shape:

```
target … : prerequisites …
        recipe
        …
        …
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘`clean`’

A *prerequisite* is a file that is used as input to create the target. A target often depends on several files.

A *recipe* is an action that `make` carries out. A recipe may have more than one command, either on the same line or each on its own line. **Please note:** you need to put a tab character at the beginning of every recipe line! This is an obscurity that catches the unwary.

Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites.

If you have created and saved your Makefile correctly you can now type “make” at the command line and it will build the executable named hello which you can run by typing “./hello”.

One of the great features of having a Makefile is that it allows you to only compile the parts of the project that need to be compiled because changes have been made since the last compilation. For large projects this can be a significant savings in time. You can witness this effect by typing make again after you have already built the project and it will tell you that hello is up to date.

```
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ make
gcc -Wall --std=c99 main.c -o hello
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ make
make: `hello' is up to date.
```

Let’s modify the Makefile so that it can split the compilation phase of a file from the linking phase of making an executable. We can do this by asking that the make utility first compile our source code into object code and then make a new rule to turn that object code into an executable. Modify your Makefile so that it looks like the following but remember to use tabs before all of your recipe lines.

```
hello: main.o
	gcc -Wall --std=c99 -o hello main.o
main.o: main.c
	gcc -Wall --std=c99 -c main.c -o main.o
```

The make utitlity, by default, will attempt to make the first target in the Makefile. In our case it will attempt to build the target hello. Since we have changed the rule to say that hello relies on a file named main.o (an object file, not an executable) then it will attempt to build it but since main.o does not yet exist it will look for a rule that describes how to make the target file main.o first. The main difference in the rule logic is that we use the compiler flag -c to tell the compiler that we want to create only object code from the main.c file. The object code for main.c can be compiled independently of all other source files and then linked together at some later time as needed. Type make and give the new Makefile a test run. Once you validate that it is working, let’s clean up the space a little. Right now if I type “ls -l", where the l is a lower case ‘L’, then I get the following output:

```
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ ls -l
total 11
-rwx------ 1 dbadams fac 8550 Jan 18 14:33 hello
-rw------- 1 dbadams fac   99 Jan 18 13:35 main.c
-rw------- 1 dbadams fac 1504 Jan 18 14:33 main.o
-rw------- 1 dbadams fac  107 Jan 18 14:41 Makefile
```

The important information shown is the number in the 5th column. It tells me how many bytes that particular file is taking up on our file system. Notice that my main.c file is fairly small but the executable,

hello, is quite a bit larger and, in practice, can be huge for large projects. If we are not currently testing or running our code then we should do the rest of the users on our system a favor and clean up the files until we need to rebuild. We could do this by typing "rm hello main.o". This will remove the executable named hello and the object file main.o to save space.

Instead of typing the command manually though each time, let's make a rule in our Makefile called clean that will allow us to do this any time we want. Modify your Makefile so that it looks like the following:

```
hello: main.o
	gcc -Wall --std=c99 -o hello main.o
main.o: main.c
	gcc -Wall --std=c99 -c main.c -o main.o
clean:
	rm hello main.o
```

Since the make utility will, by default, try to make the first target in the file each time we can ask it to select a specific target by simply naming it. Type "make clean" at the command prompt now and it will automatically remove the appropriate files. You can type make followed by any of your target names and it will try to make just that.

We can use variables in a Makefile to make writing the rules easier. It is common practice to make a variable for your compiler, your compiler flags, and the object files used in your executable. Modify your Makefile so that it looks like the following:

```
CC = gcc
CFLAGS = -Wall --std=c99
OBJECTS = main.o

hello: $(OBJECTS)
	$(CC) $(CFLAGS) -o hello $(OBJECTS)
main.o: main.c
	$(CC) $(CFLAGS) -c main.c -o main.o
clean:
	rm hello $(OBJECTS)
```

Notice how creating the variable is as simple as giving it a name, an equals sign, and assigning a value. Later in the Makefile you can replace the variable name with its value simply by putting the name in parenthesis and putting a dollar sign in front of it as shown.

TA CHECKPOINT 1: Demonstrate to your TA the following:

A. Display your Makefile for your TA.
B. Demonstrate that you can build the project using make with no arguments and then run the executable.
C. Demonstrate that you can clean up the directory by typing make clean and then show the directory using "ls -l".

Valgrind:

We will be using a tool called valgrind throughout the term to help us test for memory leaks and understand memory management better in C. In order to make use of the tool we have to turn on debugging options for our compiler. We can do this simply by adding the -g option to our CFLAGS variable in our Makefile. Notice that after adding the -g flag to the compiler options that the object code and the executable are slightly bigger.

```
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ make
gcc -g -Wall --std=c99 -c main.c -o main.o
gcc -g -Wall --std=c99 -o hello main.o
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ ls -l
total 14
-rwx------ 1 dbadams fac 9670 Jan 18 15:20 hello
-rw------- 1 dbadams fac   99 Jan 18 13:35 main.c
-rw------- 1 dbadams fac 3384 Jan 18 15:20 main.o
-rw------- 1 dbadams fac  193 Jan 18 15:20 Makefile
```

After you build the project using the make utility run it using valgrind by typing "valgrind ./hello". Your output should look something like the following:

```
dbadams@cs3:~/Spring2018/COMP1020/HANGMAN$ valgrind ./hello
==6945== Memcheck, a memory error detector
==6945== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6945== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==6945== Command: ./hello
==6945==
Hello world!
==6945==
==6945== HEAP SUMMARY:
==6945==     in use at exit: 0 bytes in 0 blocks
==6945==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==6945==
==6945== All heap blocks were freed -- no leaks are possible
==6945==
==6945== For counts of detected and suppressed errors, rerun with: -v
==6945== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The important part of the output comes after the HEAP SUMMARY: You will note that no heap space was in use at the time of exit and the program did not use any heap at all. The most important line is the one that says --no leaks are possible. This is your mantra for the entire semester. You want all of your programs to be able to run and give you the result that no leaks are possible. Valgrind will be your friend. Use it. Use it often. Let's try to write a program with an intentional memory leak to demonstrate. Modify your main.c file so that it looks like the following:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  int *p; //Declaration of a pointer variable.

  p = (int*) malloc(sizeof(int)*100); //First "bookend" allocates space
  printf("Hello world! I have created a dynamic array of 100 integers!\n");

  free(p); //Second "bookend" cleans up the space
  return 0;
}
```

Use the make utility to build your project using the command make and then use valgrind to test it by typing "valgrind ./hello". Your output should look something like the following:

```
…
Hello world! I have created a dynamic array of 100 integers!
==9539==
==9539== HEAP SUMMARY:
==9539==     in use at exit: 0 bytes in 0 blocks
==9539==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==9539==
==9539== All heap blocks were freed -- no leaks are possible
…
```

Now comment out the free(p); line in your main.c program and run valgrind again. Your result should now look something like the following:

```
Hello world! I have created a dynamic array of 100 integers!
==10074==
==10074== HEAP SUMMARY:
==10074==     in use at exit: 400 bytes in 1 blocks
==10074==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==10074==
==10074== LEAK SUMMARY:
==10074==    definitely lost: 400 bytes in 1 blocks
==10074==    indirectly lost: 0 bytes in 0 blocks
==10074==      possibly lost: 0 bytes in 0 blocks
==10074==    still reachable: 0 bytes in 0 blocks
==10074==         suppressed: 0 bytes in 0 blocks
==10074== Rerun with --leak-check=full to see details of leaked memory
```

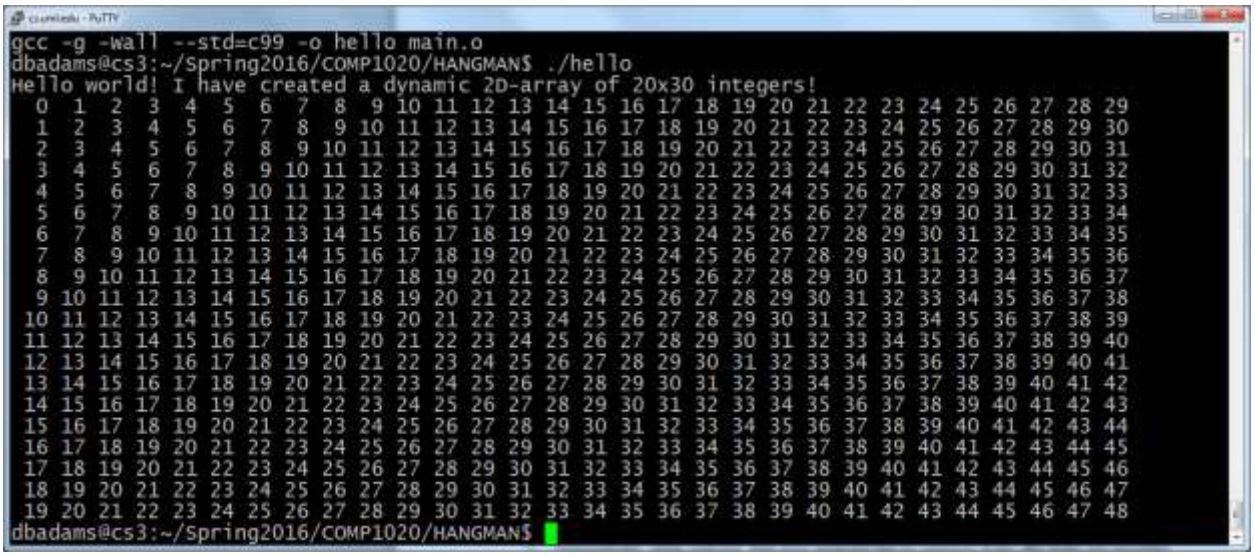
This shows me that there is a loss but it is not clear where the loss is. Sometimes the size of the loss can help you determine what it is and in our case we only have one allocation and 0 frees meaning we didn't try to free it at all but you can get even more information by simply following the instructions to rerun valgrind with the --leak-check=full option. You can do this by typing "valgrind --leak-check=full ./hello".

I just want to draw your attention to these lines:

```
==10721== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==10721==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==10721==    by 0x400595: main (main.c:8)
```

The last line actually tells you what file and what line number (main.c line 8) the malloc was made on that was never released.

Modify your main.c file so that it allocates a two dimentional array of integers so that the array has 20 rows of 30 integers in each row. Fill up the entire array so that the first row contains 0-29 the second row contains 1-30, the third row contains 2-31 and so on until the last row contains 19-48. Print the results in a grid so that your output looks like:



TA CHECKPOINT 2: Demonstrate to your TA the following:

A. Run your program and show the appropriate output.
B. Show your program to your TA so that they can see you have a dynamic two dimensional array using malloc.
C. Demonstrate, using valgrind, that your program has no memory leaks. (No leaks are possible)

The lab below is copied in the case that students enter Computing II without having first had Computing I at UMass Lowell. This is expected knowledge and can be gained simply by following the instructions below. You do not have to show your TA that you have done the Lab 9 information below.

Prerequisite Knowledge Lab:

LAB 9 - Getting around Linux and emacs

- Logging in to Linux
  - Read all of the steps here under *Logging in to Linux* before doing any of them, because some require no delay between steps.
  - You will not be using Windows for this lab. Instead, you'll be logging in to Linux. If your workstation is already running Windows, get to the point where it asks for your username. You'll see a red button in the bottom-right corner, with an arrow attached. Click on the arrow, and then select Restart.
  - You'll see, when the system restarts, that there is a choice of operating systems. By default, Windows will be selected. **Before that screen goes away** (within a few seconds),

you must press the [Up arrow] button a few times, to move the selection up to Linux (it probably says something like "Linux Mint"), and then press [Enter]. Linux will start up. (These workstations are quite slow, and it may take a minute or so before the system is usable.)

- You must use your CS account to log in. You use only the user name (no @ or domain name; you're not entering an email address, only a user name). Your user name is probably the first letter of your first name followed by up to 6 or 7 letters of your last name.
- You may see a pop-up window that welcomes you to Linux. Close that window.
- You should now see a Menu button at the bottom-left corner of the screen.

The following three bullet points, *Tour of Linux*, *Emacs*, and *C Programs*, are three separate activities. Complete one before moving on to the next.

- **Activity 1.** Take a tour of Linux. At the following link, **do not skip the first two sections:** *Typographical Conventions* and *Introduction to the Linux Operating System***.** They provide crucial information that you'll need to know during the tutorials in the later sections. Start a browser (click on the Menu button at the bottom-left corner of the screen. Click on *Internet* and then select the *Chrome* browser). Fully complete http://www.cs.uml.edu/~dlipman/linux-tutorial.

- **Activity 2.** Learn about emacs, your text editor for writing code.
  - At the shell prompt, start emacs. Its command name is **`emacs`**
  - You'll see on the initial screen that it shows you that C-h means to hold down the [control] key, and simultaneously press the [h] key. This key combination is known as "control-h".
  - Complete the built-in emacs tutorial. That initial screen shows you how to start the tutorial.

- **Activity 3.** Working with C programs
  - A clean way of organizing your C projects and assignments is to create a directory for each one. Let's assume that you have an assignment called "project1". Change into your home directory, and create a new directory there called `project1`. Change directory into `project1`.
  - In **LearnCS!**, when you wanted to run a program, you would simply click the Run button. Now, though, you'll have multiple steps.
    1. Run emacs, and create a file called `project1.c`, and in that file, create a very simple program that prints "hello world" to the screen. Save the file, and exit emacs.
    2. You must now compile your C program, which converts the C program text into machine language that can execute on Linux. The command to compile your code is **gcc** and it takes a number of command-line arguments.

- **`-Wall`** which means “all warnings” and helps to find bugs in our code
- **`--std=c99`** which specifies that you’re using the 1999 version of the C language, the most commonly-used version
- **`-o executable_name`** is optional. By default, gcc produces an executable file called `a.out`. Each time you compile a program within any particular directory, the executable file `a.out` is created, possibly overwriting a previously-created executable file of that name. You may use the **`-o`** option to specify a name for the executable file to be created. Typically, the executable file name is the same as the C file name, without “.c”. In the case of this example project, you might then use the arguments **`-o project1`** to specify that the executable file name should be `project1` instead of `a.out`. (Note that there must be a space between **`-o`** and **`project1`**.)
- Usually, the final command-line argument is the name of the C file to be compiled.
- (In the case where you need additional libraries, such as the math library, there may be additional command-line arguments. If your program includes <math.h>, for example then on the command line, you will also need to link the math library into your executable, by appending **`-lm`** to the command line.)

The complete command line, then, to compile your C file (which does not require any libraries), would be:

```
gcc -Wall --std=c99 -o project1 project1.c
```

3. Now that you’ve created an executable file, you have something that can run. As you learned in the Linux tutorials, you can run that program in your current directory, using just its name, for example, **`./project1`**

- When you have completed both the Linux and emacs tutorials, and have successfully compiled your small program, ask your lab instructor for sign-off. Your lab instructor will look at the directory structure you have created during this lab, and ask you to complete a number of tasks in the shell and in emacs to verify that you have enough understanding to be able to work on your Computing I assignments using Linux and emacs henceforth. (Many of your future courses in computer science will be using this same environment -- Linux and emacs -- too, so now is the time to be getting thoroughly familiar with it.)

- Instructor checklist:
    - Ensure that the student successfully printed, and has their printout from Linux Exercise 3B
    - Have student open the science.txt file, and demonstrate at least the following emacs commands:
        - line up, line down
        - page up, page down
        - delete character, delete word
        - undo
        - save file
        - exit without saving
    - Ensure that the student was able to compile and run their hello-world program.