Section 1 Lesson 1

I. Discuss any new data structures, algorithms, or language features

There is no data used in this very simple application, there aren't any formal data structures either. The algorithm is the simplest sequential case, running line by line. The problem we are solving is to get the student (you) accustomed to using the tools you installed in lesson S1L0. It introduces all of the parts necessary to build an application – tools, code, makefile, and file structure. Once you understand all of the parts of this lesson, the code compiles and runs you are ready to start learning the tools you will use in all of the future applications you write.

Here you will see how light, and fast, gcc and make truly are. Notice how little time it takes to edit, compile, and then run an app under construction. Learn the keystrokes to reuse previously entered commands in your terminal. The up arrow works for me. You can use any command you have typed in that terminal window since you started it, just press the up arrow enough times and you will see (one of the perks of having installed MinGW on your system). Once you get the right command line press enter, and it will run, just as it ran previously. If you want to edit the command simply use the left arrow key (NOT the backspace key, unless you need to erase a character) to find the right spot to insert or delete whatever you want. Then hit enter and the edited command will run.

If you need to change directories use the cd xxxxx command. Oftentimes the xxxxx part of that gets rather long with lots of phrases interspersed with / characters. You can make your life much simpler by learning how to use the * key liberally. For instance: you are currently in your home directory ~ but your file of interest is ~/Bacona Design Solutions/lessons/4 advanced graphics/S2L9_orbit/Particle.cpp Whew, that's long and since there are spaces in it you'll need to quote the whole thing for your cd command to work correctly. Solution? Easy, use the * key liberally by noticing where folder names are the same and where they are different. Use this instead: cd *ions/*ons/4*/S2L9* will get you to the proper directory, with far few keystrokes, while you don't need to quote anything because you have cleverly obfuscated the spaces. Hee, hee, hee :) Just pay attention so you can feel free to be lazy.

II. Describe functions and modules – why

There are only two functions in this trivial application - main() & printf(). main() is the start point for any application run under C. In this simple app it does everything and requires no other files or external functions. main() opens a memory space, starts a process on the CPU, then runs any functions it calls. The only thing this program does is to print the exclamatory sentence -- 'Hello knowledge!' The printf() function is linked from the included library stdio.h Expand your programming vocabulary by using library functions . Soon we will create our own header (.h) files.

III. Compile and run the code

Use the cd (change directory) command to navigate to your project folder. For instance: type **cd /mnt** then type **cd lessons/lesson1** or more simply: **cd /mnt/lessons/lesson1** directs you to the first lesson on my Linux box. Your path will be different.

On my Windows box I launch the command prompt and type **J:** which gets me to a shared drive on my network. Then I type **cd \lessons\lesson1** to get to the folder. The slanty things are backwards :) By using MinGW the direction of the slant is not as important.

Now that you have navigated to the lesson1 folder simply type **make** This calls the makefile in your lesson folder and runs the script creating the executable file "new". On a Linux box type **./new** at the terminal window prompt to run your program. "Hello knowledge!" gets printed to the screen followed by a new line. Linux likes the app name to be preceded by ./ or else it will give you an error. Odd but true.

Under Windows operating systems you can navigate to the application's folder either inside the command prompt window or by using the graphic interface provided by the Windows Explorer. Here your new program is called new.exe and it is listed as an application under type. If you double click the application it will run. But it will run very quickly and then the window will close before you can read it. That is why I advocate using the command prompt window to make things simple. Later we can add code as a trap to keep from jumping right back to the OS again after running. However, if you open the command prompt window, navigate to the correct folder, you only need to type **new** and your application will run.

IV. Describe code line by line – how

The first four lines of the file lesson1.c are comment lines denoted by the leading // on each line. This is the simplest type of comment limited to one line long. If more comments are required you just add // to the beginning of each successive line

I normally name the file, add my by line, and a date along with the company name. In more complicated applications this comment area can be used to describe the app and its features. I also comment by function and by line throughout the code.

The first active line of code is the #include which calls an external library of code stdio.h. This library contains the definition of printf(). By including the library the compiler will know about printf() and how it is used by its declaration fingerprint.

Next comes the main function. This line declares and defines in one step without any need for predeclaration before the main() function. int main(void) says the function main requires no input (void) and returns an integer (int) as output. main() traditionally returns an error code so the operating

system knows whether the app ran correctly or not. Technically the function main() should end with return 0 so the OS knows it ran correctly.

Now the actual work of this app - printf() prints the string of characters inside the quotation marks. "Hello knowledge!\n" prints Hello knowledge! followed by a newline character to tell the cursor to move down to the next line. \n is the way we tell printf() we want a new line. If we had not used the \n character there would have been no space between the end of this app and the next character on the monitor. Use \t and you'll get a tab instead of a new line. The \ is called an escape character telling printf() to use the next letter as an internal code.

Since this is the only work done by this app it now ends with the final } of main(). This tells the OS to free up the memory used by the app and close the process.

V. Describe makefile line by line

The makefile is a script read by the terminal command make. When you type **make** the makefile script's instructions are implemented to build each part, but only if they need to be built. If you have not edited any part of an already compiled file it does not need to be recompiled.

Comments in makefiles are shown with # at the beginning of a line. It is always good to provide typing instructions in your makefile to jog your memory in the future. Comments throughout the script are great for keeping track of design choices. Please start with line 7 because this is where make will find the first instruction it needs to perform. "lesson1.o: lesson1.c" lists what is being created – lesson1.o with lesson1.c naming the file required to build lesson1.o The next line MUST begin with a tab character or else you'll generate an error. make uses the command "gcc -c lesson1.c" to compile the code into an object file. gcc is the command which calls the gcc compiler. -c is a flag telling gcc to compile the next parcel. So gcc -c lesson1.c compiles the C code in lesson1.c creating the object file lesson1.o

Now lines 4 & 5 get called.

"lesson1: lesson1.o" tells make that lesson1 depends on lesson1.o being present.

"gcc -o new lesson1.o" tells gcc to link the application from the object file lesson1.o, -o tells gcc to link the application "new" from the object file "lesson1.o"

Lines 11 – 13 are used for cleaning up. They remove the object file lesson1.o and the application new. Type **make clean** to delete the app and the object file. Then, when you type **make,** gcc is forced to recompile and link every rule of the makefile.

VI. Deeper

If you are running under Windows, the compiler will add .exe to the end of the application name – new.exe If you use the makefile under Linux, the application will be named new without any ending. If you copy an executable between operating systems you will find an .exe file will not run under Linux and an executable compiled under Linux will not run under Windows. Yes, the same code gets compiled a little differently depending on the OS you are using. But by having the source code and the makefile you can compile the code for whichever environment you like. This is true for every lesson in this section of the course. I wanted to make sure students could use the lessons for either OS.

VII. Assignments

1. Change the text in printf() to say something different.
2. Add more printf() statements to say multiple things each time the app is run.
3. Change the printf() statement to use /t instead of /n
4. Change the name of the app in the makefile
5. Change rm lesson1.o to rm *.o for a more realistic use case.
6. Add the flag -Wall at end of makefile line 8

   change file lesson1.c line 8 from int main(void) to void main(void)

   now type **make clean** followed by **make** in your terminal window.

Read the warning message for future reference. Then repair the damage.

VIII. Links

https://gcc.gnu.org/

https://en.wikipedia.org/wiki/GNU_Compiler_Collection

https://www.gnu.org/software/make/

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

https://en.wikipedia.org/wiki/Make_(software)

https://en.wikipedia.org/wiki/Stdio.h

https://www.tutorialspoint.com/c_standard_library/stdio_h.htm