Setting up Codio for this assignment

1) Open the Codio assignment via Coursera
2) From the Codio File-Tree click on: **program1.c**

Problems #1-3: Basic I/O in C

Recall in Assembly that performing I/O involved calling a TRAP. TRAP_PUTS – wrote an ASCII string out to the screen and TRAP_GETS – read in an ASCII string from the keyboard. In C, there are two functions: printf() and scanf() that perform the same basic task (printing output to the screen and reading a string in from the keyboard). They are actually "wrapper" functions – like the one you created in the last HW – that invoke OS TRAPs to perform I/O on your behalf.

Let's examine your first proper C-program (that does I/O) together and compile it on the Intel x86 (Codio runs on an intel x86 processor). We won't use LC4 because the function "printf() and scanf()" don't exist yet…you would have to write them! The C-compiler we'll use on the Intel x86 has many premade functions you are welcome and encouraged to use. So for this HW, we will use the Intel x86 compiler called: *clang*, instead of our LC4 compiler: *lcc*.

On Codio, open the starter file called: **program1.c** in examine the following code:

```
#include <stdio.h>

int main() {

      printf ("Hello World!\n") ;
      return (0) ;
}
```


Open up a terminal window in Codio and *compile* the above code by typing the following:

```
clang program1.c –o program1
```

What does this mean?

`clang` -is the name of our Intel x86 C-compiler

`program1.c` -is the name of the file you want to compile

`-o` *program1* -tells the compiler you'd like to create an **OBJECT** file called *program1*

This compiler will internally generate the Assembly of your program1.c, but then automatically call the Intel x86 Assembler and have it make an OBJECT file for you. So you don't need to manually "assemble" the output file like you did in the last assignment.

Now let's "run" your program on the Intel x86. You would do that by typing in the name of the file directly into the terminal as follows:

```
./program1
```

If things went well, you should see output to the ASCII display (aka – the terminal window):

```
Hello World
```

Congratulations, your first C-program works! You've performed the most basic of C-I/O – working with the ASCII display. All the same steps of a TRAP being called were done in the blink of an eye (privilege being elevated, vector table being called, trap subroutine being called, returning back to the user, etc), and your string is outputted on the ASCII display.

Recall that a compiler should take the high-level language and convert it to assembly. While clang does this step internally, it is possible to force it to show you the assembly it produces. Type in the following command:

```
clang -S program1.c
```


This will generate a file called: program1.s It will contain the x86 Assembly translation for this program! Open this file and you'll see what the Assembly Language looks like for the Intel x86 ISA. Next, re-compile this same program to our LC4 ISA, using the old LCC compiler. The only trouble is that LCC doesn't have the stdio.h library, *so open up the file and comment out the first line (#include <stdio.h>)*. Then type:


```
lcc program1.c
```


Even through our compiler doesn't have the library, it will create the JSR for printf(), it assumes you'll make it someday and you'll link it to create the final object file. Open up program1.asm and compare it to program1.s. Can you find the equivalent of the LC4's JSR command in the program1.s file? *Hint: look for the call to printf!* *(Want to know more about Intel's x86 ISA: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html)*

Let's try reading from the keyboard

Make a new file, called program2.c and type in the following C-code (do NOT copy/paste this, the quotation marks do not copy properly from the PDF):

```
#include <stdio.h>

int main () {

      char name [50] ;
      printf ("What is your name? ") ;

      scanf ("%s", name) ;

      printf ("Hello %s\n", name ) ;
      return (0) ;
}
```


Save the file as **program2.c** and *compile* the above code by typing the following in the Codio's terminal:


```
clang program2.c -o program2
```

Then run it & try it out:


```
./program2
```

Analyzing what we've done:

A few things to notice about our program1 & program2:

- You didn't have to "write" printf() and scanf(), the assembly that implements these two functions is "linked" to your program (just like in the last assignment – recall 'linking'). You told the compiler to link them when you placed the line: #include <stdio.h> at the top of your C-file. STDIO – stands for "standard input/output" functions.
- Printf() and scanf() actually stand for "print formatted string" and "scan formatted string" What does "formatted" mean? Things like new-line characters are considered formatting information – they aren't visible ASCII character exactly, they are instructions for how to 'format' the output. Notice in program1 we said:
- Printf ("Hello World\n") ; the "\n" is tip to the printf() function that you'd like a new-line inserted. There are others… \t for tabs: (https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences)

Doing more with printf() and scanf()

Your textbook, pages 485-490 does an excellent job explaining the details of printf() and scanf(). Read these pages and try out the examples they give. Once you have mastered the examples, write the following program and place it in a file called: **program3.c:**

1) Print "Welcome to CIT 593" to the ASCII display using printf(), followed by a newline

2) Ask the user to type their **name**, their **HW1**, **HW2**, **HW3**, **HW4** scores (out of 100), their **midterm** grade, and their expected **final** exam grade using a SINGLE scanf() statement (not in a loop)

3) Have your program average the HW scores and compute their final weighted average (using the CIT 593 syllabus as your guide).

4) Display their statistics using this *example* format:

Name: Thomas

HW Average	: 100.00%
Midterm Grade	: 100.00%
Final Exam Grade	: 100.00%
Final Average :	: 100.00%

(Notice my use of tabs, the alignment of the decimal points and only two decimal places)

Make certain to compile your program often and test it before submitting program3.c

Problem #4 – Functions and Pointers in C

As you have learned well in the last assignment, functions in C are translated to sub-routines in Assembly that obey the use of the stack. All implementations of C use the stack to pass data/hold data/return data to and from functions in C. This is true for the Intel x86's implementation of C as well. In this section of the HW, you'll experiment with setting up functions in C and seeing how data is passed to & from the function.

Create a new file: program4a.c, and type the following program into it:

```
#include <stdio.h>

void swap (int a, int b) {
     int c = 0 ;
     c = a ; /* swap values of a and b */
     a = b ;
     b = c ;

     printf ("a= %d\n", a) ;
     printf ("b= %d\n", b) ;

     return ;
}

int main() {
     int a = 5 ;
     int b = 10 ;

     printf ("a= %d\n", a) ;
     printf ("b= %d\n", b) ;

     swap (a, b) ;

     printf ("a= %d\n", a) ;
     printf ("b= %d\n", b) ;

     return (0) ;
}
```


Compile the program with clang and view the results. Do "a" and "b" swap values in the function: swap()? Do "a" and "b" swap values back in main()?

Create a new file: program4b.c, and copy and paste the program from program4a.c into this file. Fix the program to perform the swap. You won't need to add any lines of code to this file. Instead, you will only need to use pointers instead. You'll want the variables inside swap to "point" to the variables in main() – instead of being copies (as they are in program4a).

Compile the program with clang and view the results. Do "a" and "b" swap values in the function: swap()? Do "a" and "b" swap values back in main()? Can you explain why? Drawing out the stack may help…

Available on Coursera is a file: CIT593_Assignment_C-Basics_spreadsheet.xlsx. Download and open the file and pretend for a moment your program is running on the LC4, using the stack on the LC4 and the LC4's register file. Show why these programs differ and explain that using the spreadsheet included.

You will upload the completed version of this spreadsheet (after completing problem 7) to the root folder of your Codio workspace for submission

Problem #5 – Multifile development in C

Often in C, we like to not work in one giant file! Much like you did in your previous assignment, you can separate functions from one another into separate files. This makes development of programs much easier when you have more than one programmer on a project. But it's also a good way to setup a project in your own development environment, so we'll do that here as well.

Create a new file called: **program5_swap.c**, and place the first part of program4a.c inside it:

```
#include <stdio.h>

void swap (int a, int b) {
      int c = 0 ;
      c = a ; /* swap values of a and b */
      a = b ;
      b = c ;

      printf ("a= %d\n", a) ;
      printf ("b= %d\n", b) ;

      return ;
}
```


Save the file and try to compile it using the following command:

```
clang program5_swap.c -o program5_swap
```


You'll immediately get an error. Why? Because you don't have a main() function inside this file. Without a main() function, the c-program could not start. But we have another option, try compiling with this command:

```
clang -c program5_swap.c
```

This should work properly (assuming your code is correct), and if you look in your folder you'll see a new file called: program5_swap.o. This new file is called a "partially compiled" object file. It is an object file, but it cannot run on its own, because it doesn't have a main(). This is just like your SUB_FACTORIAL.asm file in the last assignment after you modified it. It cannot run by itself, because it didn't have a main().

program5_swap.o is also considered a library. It is a library of code with only 1 function, but now it's separate from its main(), so anyone could call it from their own program. If you can imagine, there is a library called: stdio.o, it contains the implementation for printf() and scanf() (among other functions). It is just waiting for someone to "link" their program to it and call the functions that are within the library.

So, how do we link a program with program5_swap.o? Create the following file called: **program5.c**:

```
#include <stdio.h>

void swap (int a, int b) ;

int main() {

      int a = 5 ;
      int b = 10 ;

      printf ("a= %d\n", a) ;
      printf ("b= %d\n", b) ;

      swap (a, b) ;

      printf ("a= %d\n", a) ;
      printf ("b= %d\n", b) ;

      return (0) ;
}
```


Notice that new line at the top:

```
      void swap (int a, int b) ;
```

This is known as a function “declaration”, it tells the compiler all about the function swap(): its return type, its name, and the type and order of its arguments. Now, the compiler could actually write the assembly for main, including the call to swap (packing the arguments properly on the stack), even if swap() did not yet exist (this is what your LCC compiler does!).

Save the file and try to compile it using the following command:

```
      clang –c program5.c
```

You’ll see that program5.o gets created. It too is a “partially” compiled file, because it contains the full assembly implementation of main(), but it doesn’t contain the actual implementation of swap()!

We must now “link” the two .o files together, which can be done by typing: `clang`

```
      program5_swap.o program5.o –o program5
```

This will final produce: program5, an object file that we can run by typing: ./program5 Try it now and make sure your program is properly “linked” together.

Problem #6 – Headers Files and Makefiles:

Instead of placing "declarations" of the functions in the file with our main() function, there is a better way to decouple the files a bit more. We can create a separate file, called a "header" file. A header file is a simple file that contains only function declarations. Let's create one for our program5_swap.o library:

Create a new file (called program5_swap.h) and add only this one line:

```
void swap (int a, int b) ;
```

Notice, the extension: ".h" it stands for header. We are creating a simple header file. Copy the file: program5.c, and call it program6.c. Now, edit program6.c

```
#include <stdio.h>
#include "program5_swap.h"

int main() {

      int a = 5 ;
      int b = 10 ;

      printf ("a= %d\n", a) ;
      printf ("b= %d\n", b) ;

      swap (a, b) ;

      printf ("a= %d\n", a) ;
      printf ("b= %d\n", b) ;

      return (0);
}
```


Notice, that we put quotes around "header" files we create, and we use <> - angle brackets, for header files that come with our compiler. Next, try compiling program6 with the following command:

```
clang program5_swap.o program6.c -o program6
```

Notice inside program6.c, we didn't need to put the function declaration, because it is contained inside program5_swap.h. The compiler will open this file (when it is compiling your program6.c) and substitute in the contents of the header file into your program6.c. Try running your program: `./program6` and make sure it's working properly.

Makefiles:

As you can see, the commands that make the compiler work get a bit cumbersome after a while. And the more files you create as part of a large project, the more difficult it becomes to manage all the compiling. So on Linux based systems, we like to use a utility that comes with Linux called: “make” ; It’s a simple utility that can automate some of our compiling duties. It even works with java files, actually any language – it’s language independent.

Open up the file entitled: “Makefile” that was preloaded onto Codio:

Scroll down in the file to the line:

```
program5: program5.o program5_swap.o
	clang program5_swap.c program5.o -o program5
```

This is called the “program5” rule. A Makefile is like a cookbook, it contains a recipe for how to create targets: program5. “program5” is also called the “target” something to make.

The items after the colon: `program5.o program5_swap.o`, are the items that “program5’s” recipe depends upon. These things must exist before the “recipe” for making the program5 target can ever be run. Let’s say the file: program5_swap.o doesn’t exist. The make utility will look for the “target” program5_swap.o. So go ahead and look for that rule yourself:


```
program5_swap.o: program5_swap.c program5_swap.h
	clang -c program5_swap.c
```


If the file: “program5_swap.o” didn’t exist when we were trying to follow the program5 recipe, the make utility will run the rule for program5_swap.o, following the recipe to make the file: program5_swap.o first. Then it will go back to the program5 recipe and continue making it.

try running it as follows. From the terminal type:

```
make program5
```

The make utility automatically looks for a file named: “Makefile” and looks inside that for the target you’ve typed: program5. It tries following the program5 recipe. If any of the “prerequisite” files – aka the files that must exist first, don’t exist, it will follow the recipe for creating them. So it should run the program5_swap.o recipe if it doesn’t yet exist, before it follows the program6 recipe. Once it’s complete, you should have the file: program5 and you can then run it.

The general format for a Makefile *rule* is as follows:

```
target: prerequisite
	recipe
```

You can go target by target and run each of them. OR you can run the target at the top: `all`:

```
program1 program2 program3 program4a program4b program5 program6
```

This is called a "phony" target, because a file called "all" doesn't exist, nor will it ever (because you'll notice it has no recipe underneath it). Instead, if you type: `make all` then the make utility will look for its prerequisite files: program1 program2, etc. and if it doesn't find them, it will follow the recipes to make them

Try typing:

```
make all
```

You should see it create all of programs we've made so far: program1-6. It may also say they are "up to date" if no changes in the .C files (or the .H) files have occurred since the targets were created.

Next, type in:

```
make clean
```


The job of this phony target (as you will see if you look inside the Makefile) it to delete all the ".o" files. The .o files are necessary while you are compiling your program, but they aren't needed when you are running your program. So this target deletes intermediate files when you wish to clean up your folder.

Lastly, if you type in:

```
make clobber
```


This phone target runs the "make clean" recipe, and it also deletes the OBJECT files themselves! This just leaves your .C files and .H files behind. It's handy if you just have too many files in your folder and you want to clean things up a bit. To bring them all back, simply type:

```
make all
```

And it will compile all your files fresh! Before you turn in your HW, be sure to type in: `make clobber` and it turn in only those files that remain! We will use your Makefile to regenerate your object files and test them accordingly.

Now that you understand rules, targets, and prerequisites; create a rule for program6. You can use program5's rule as an example but there are some differences. Remember that program6.c depends upon program5_swap.h. So you'll need to incorporate that into the prerequisite. Once you get the rule working, update the "all" and "clobber" rules accordingly.

Problem #7 – Pointer Basics

Included in Codio is a file: program7.c. Examine the contents of the file and compile this program and then run it in a Codio terminal window. You'll see it prints out the actual memory addresses in the x86 stack! They are in hex, so you'll get a feeling for how the x86's memory and its stack are organized. Each time you run this, your program will be given a different place in the x86's data memory to use. So you'll get different addresses each time your run it (that's normal).

Task #1: Complete the program

Notice that the second part of program7.c is incomplete, the print statements don't actually print the values and memory addresses that they should. Using the first part of the program as an example, update these print statements to print out the things they are supposed to. Hint "deref'ed" means "dereferenced." It's just my short hand.

Task #2: Complete the spreadsheet

The spreadsheet that you worked on in problem #4 actually has two "tabs" within it. Switch to the problem7 tab at the bottom of Excel. The Excel spreadsheet needs to be completed. Use the output of the execution of your program (after you complete Task #1) and reason through what must be on the stack and where it must be on the x86 stack. This is a little tricky, but the memory addresses will help you get a good picture of the Intel x86 stack!

Task #3: Update the Makefile

Add a rule to the Makefile to compile program7 before you turn it in.

Problems #8 & #9 – Debugging Basics

Included in codio are two files: program8.c and program9.c. While each of these files will compile, they contain bugs that will prevent them from running correctly. The purpose of these two files is not to have you debug the programs visually, but instead using a debugging tool called: GDB (for GNU DebBugger). To use this debugger, one must compile their files using the "-g" flag. The -g flag compiles your program as it normally would, except it keeps all the line numbers and variable names in the final executable file, which is very helpful for debugging!

1) Watch the following tutorial on GDB:
   https://www.youtube.com/watch?v=bWH-nL7v5F4

2) Compile program #8 with the -g flag as follows (note, the tutorial uses gcc, we use clang)

   clang -g program8.c -o program8

3) Add the above recipe to make program8 to your makefile and include program8 in the clean recipe

4) Try running the program, you will see that it goes into an infinite loop and doesn't appear to stop, press <control> c to force the program to end.

5) Start program8 inside gdb using the following command:
   (we need to startup a little differently than the tutorial):
   gdb -tui ./program8 ***(-tui brings up the "text user interface")***

6) Important for Grading: Turn on the gdb logger with the following command inside gdb:
   set logging on
   This will output all the debugging output into a file called gdb.txt. Do not delete this file.
   You must do this command each time you launch gdb.
   Alternatively, you can start gdb with logging enabled automatically:
   gdb -tui -ex "set logging on" ./program8

7) Now follow the steps in the tutorial to debug program8.c. Make the necessary changes such that program8.c produces the correct output.

8) Once you finish, repeat this debugging process for program.9.c

As a side note for future HW's, if you happen to write a program that requires command line arguments, you can pass arguments to your program like this:
gdb -tui --args ./program8 argument1 argument2 …

A second wonderful tip for debugging is to have the compiler warn you of poor programming choices! You should always use the "Warnings All" flag when compiling your programs in C by using the following flag:

clang -Wall -g program8.c -o program8

Important Note on Plagiarism:

- We will scan your HW files for plagiarism using an automatic plagiarism detection tool.
- If you are unaware of the plagiarism policy, make certain to check the syllabus to see the possible repercussions of submitting plagiarized work (or letting someone submit yours)