CIT 593 - Module 09 Assignment

Introduction to C Programming

Table of Conte

Assignment Overview 3
Learning Objectives 3
Advice 3
Getting Started 4
Codio Setup 4
Starter Code 4
Problem 1 - Compare x86 Assembly to LC4 Assembly 5
Overview 5
Requirements 6
Problem 2 - Read from the Keyboard 7
Overview 7
Requirements 7
Problem 3 - Read User Input and Format Output 8
Overview 8
Requirements 8
Problem 4 - Functions and Pointers in C 10
Overview 10
Requirements 11
Problem 5 - Multi-File Development in C 12
Overview 12
Requirements 14
Problem 6 - Header Files and Makefiles 15
Overview 15
Makefiles 16
Requirements 18
Problem 7 - Pointer Basics 19
Overview 19
Your Task 19
Requirements 19
Problems 8 and 9 - Debugging Basics 20
Overview 20
Your Task 20
Requirements 21
Submission 22
Where to put the files 22
Pre-Submission Test 22

The Actual Submission 22
Grading 22
Important Notes on Plagiarism 22
Hints or FAQs 23
Resources 24

Assignment Overview

In this assignment, you will continue C programming. You will explore how The Stack works in the C language and also start learning how to debug C programs.

Learning Ob

- Write a basic
- Analyze how The Stack
- Create functic
- Debug progra

Advice

- Start early
- Ask for help early
- Do not try to do it all in one day

Getting Started

Codio Setup

Be sure to open Codio from the Codio Assignment page in Canvas.

Starter Code

We have provided a [illegible]arted on this assignment. You will have to create additional files to com[illegible]t.

`program1.c`
This is an example p[illegible] use to compare x86 assembly to LC4 assembly. Submit the unmodified version.

`program7.c`
This program will be used to help you explore how different variables appear on the stack. You will use this to fill out a spreadsheet as part of your submission.

`program8.c`
This is a program that has several bugs. You will need to debug the program using gdb.

`program9.c`
This is a program that has several bugs. You will need to debug the program using gdb.

`Makefile`
This makefile is used to automate compiling the various programs in this assignment.

Problem 1 - Compare x86 Assembly to LC4 Assembly

Overview

Recall in assembly th olved calling a TRAP. `TRAP_PUTS` writes an ASCII string out to the scre ads in an ASCII string from the keyboard. In C, two functions that perform (printing output to the screen and reading a string in from the keyboard) are `printf` and `scanf`. They are actually "wrapper" functions, like the one ast assignment, that invoke OS TRAPs to perform I/O on your behalf.

Let's examine your first proper C program that does I/O and compile it on the Intel x86 (Codio runs on an Intel x86 processor). We won't use LC4 because the functions `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 assignment, we will use the Intel x86 compiler called `clang` instead of our LC4 compiler `lcc`.

In Codio, open the starter program called `program1.c` and 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 to name the output file as `program1`

This compiler will internally generate the assembly of your `program1.c`, but then automatically call the Intel x86 assembler and have it make the executable "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, yo to the ASCII display (the terminal window):

```
Hello World
```

Congratulations, you ks! You've performed the most basic of C I/O: working with the ASC e 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 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 though 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 executable file. Open `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`!

Requirements

- `program1.c` should compile and print `Hello World`
- The `Makefile` should contain a target for `program1.c` (already done for you)
- Be sure to restore `program1.c` to the original before submitting

Problem 2 - Read from the Keyboard

Overview

Let's try reading from the keyboard. Make a new file called `program2.c` and type in the following C code:

```
#include <s                    () {
        char
        print                  name? ") ;
        scanf
        printf ("Hello %s\n", name ) ;
        return (0) ;
}
```


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
```


A few things to notice about our program1 & program2:

- You didn't have to implement `printf` and `scanf`. The assembly that implements these two functions is "linked" to your program (just like in the last assignment). You told the compiler to link them when you placed the line `#include <stdio.h>` at the top of your C source file.
- `printf` and `scanf` actually stand for "**print f**ormatted string" and "**scan f**ormatted string". What does "formatted" mean? Things like new-line characters are considered formatting information; they aren't visible ASCII characters but rather instructions for how to "format" the output. Notice in `program1` we said:

  ```
  printf ("Hello World\n") ;
  ```

  The `'\n'` character indicates to `printf` that you'd like a new-line inserted. There are others, e.g. `'\t'` for tabs. The Resources section has a list of other "escape sequences".

Requirements

- `program2.c` should compile and run the program to read user input and print a formatted string.
- The `Makefile` should contain a target for `program2.c` (already done for you)

Problem 3 - Read User Input and Format Output

Overview

Your (second edition) textbook, pages 485-490, does an excellent job explaining the details of `printf` and `scanf` and try out the examples they give.

Once you have mast- mplement the following program requirements in C and place it in a file c

1. Print "`Welcom` e ASCII display using `printf`, followed by a newline.
2. Ask the user f- eir HW1, HW2, HW3, HW4 scores, their midterm grade, and th- m grade using a single `scanf` statement (and not in a loop). Note that all scores are out of 100.0 not the random number of points in the various rubrics.
3. Have your program average the HW scores and compute their final weighted average (use the CIT 593 syllabus as your guide, consider the final project just another homework).
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 the use of tabs, the alignment of the symbols, and number of decimal places.

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

Requirements

`program3.c` **MUST**:

- print the welcome message followed by a newline character.
- Ask the user to type the name, their HW1, HW2, HW3, HW4 scores (out of 100.0 points), their midterm grade, and their expected final exam grade with a single `scanf` statement that does not exist in a loop.
    - You are not permitted to use any other function in the `scanf` family.
    - Each entry will be separated by a single space.
- Average the homework scores to compute the weighted homework average using the CIT 593 syllabus, and considering the final project to be another homework assignment.
- Display the results in the format described above, paying close attention to the use of tabs, alignment of colons, decimals, and percent sign, and the use of two decimal places.

Problem 4 - Functions and Pointers in C

Overview

As you have learned well in the last assignment, functions in C are translated to Subroutines in Assembly that obey t All implementations of C use The Stack to pass data/hold data/return ctions in C. This is true for the Intel x86's implementation of C

In this section of the periment with setting up functions in C and seeing how data is passed t on.

Create a new file: pr 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. Then 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[illegible] the results. Do `a` and `b` swap values in the function: `swap`? Do `a` and `b` sw[illegible]`in`? Can you explain why? Drawing out the stack may help…

Obtain the template f[illegible]`ent_C-Basics_spreadsheet.xlsx` from Canvas. Download and open [illegible]or a moment your program is running on the LC4, using the stack on th[illegible]register file. Show why these programs differ and explain that using the[illegible]d.

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

Requirements

- `program4a.c` **MUST** show an incorrect swap of a and b
- `program4b.c` **MUST** show a correct swap of a and b using pointers
- You **MUST** show the stack as if it were run on the LC4 in the first tab of the template spreadsheet.

Problem 5 - Multi-File Development in C

Overview

Often in C, we like to not work in one giant file! In fact, doing so is an anti-pattern in programming and co amming practice.

Much like you did in y ment, you can separate functions from one another into separate files. Th nt of programs much easier when you have more than one programme also a good way to set up a project in your own development environ here as well.

Create a new file call 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 cannot 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` function. This is just like your `SUB_FACTORIAL.asm` file in the last assignment after you modified it.

`program5_swap.o` is also considered a library. It is a library of code with only one function, but now it is separate from the `main` function, so anyone can call it from their own program. If you can imagine, there is a library called `stdio.o`, which 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 [illegible]`am5_swap.o`? Create the following file called: `Program5.c` and pla[illegible]nts:

```
#include <s

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" object 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 object files together, which can be done by typing:

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

This will finally produce an executable object file that we can run by typing:

```
./program5
```

Try it now and make sure your program is properly linked together.

Requirements

- You MUST create the files described here with the correct contents, and your program MUST correctly compile.

Problem 6 - Header Files and Makefiles

Overview

Instead of placing declarations of the functions in the file with our `main` function, there is a better way to decoup[illegible]. We can create a separate file called a "header" file.

A header file is a sim[illegible]only function declarations. Let's create one for our `program5_swap.o` lib[illegible]ile called `program5_swap.h` and add only this one line:

```
void swap (int a, [illegible]
```

Notice the extension `.h`, which stands for header. We are creating a simple header file.

Copy the file `program5.c` and call it `program6.c`. Now, edit `program6.c` to be:

```
#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 angled 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 to be sure it is working properly:

```
./program6
```

Makefiles

As you can see, the c the compiler work get a bit cumbersome after a while. And the more f rt of a large project, the more difficult it becomes to manage all the comp sed systems, we use a utility that comes with Linux called `make`.

`make` is a simple utilit some of our compiling duties. It even works with Java files; actually it age – it's language independent.

Open up the file `Make` r code. Scroll down in the file to the line:

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


A `Makefile` is like a cookbook, it contains a recipe for how to create targets like `program5`. The red highlighted part before the colon, `program5`, is called the rule or target. The blue portion is called the recipe and it is the command to execute to make the target.

The items after the target and colon, `program5.o program5_swap.o`, are the target's dependencies, the items that `program5`'s recipe depends upon. These things must exist before the recipe for the target can 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 recursively run the recipe for that target, 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. 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 (the dependencies) don't exist, it will find the target for the missing dependency and 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 `program5` 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
```

The recipe always ex shifted by a **tab**. **Do not use spaces**: `make` is an old utility and requires th u uses spaces, make will complain about missing separators. Disable this keeps happening to you.

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 actually exist (notice that the recipe below does not create a file called `all`). 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 running this command now.

You should see it create all of the programs we've made so far: `program1` through `program6`. It may also say they are "up to date" if no changes in the `.c` source files or the `.h` header 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) is 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 `clean` recipe, and it also deletes the executable object files themselves! This just leaves your `.c` source files and `.h` header 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 re you turn in your assignment, be sure to run `make clobber` and to turn d header files that remain! We will use your `Makefile` to regener nd test them accordingly.

Now that you unders nd prerequisites; create a rule for `program6`. You can use `program5`'s rule as a... 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

Requirements

- Most of the targets and recipes are provided in the starter code. They MUST still work when you submit.
- You MUST create the missing files with the appropriate contents as described above.
- You MUST create a rule for `program6` with the appropriate dependencies and recipe.
- You MUST update the `all` and `clobber` rules for this new program.

Problem 7 - Pointer Basics

Overview

One of the starter code programs is `program7.c`. Examine the contents of the file, then compile this program [illegible] it prints out the actual memory addresses in The x86 Stack! They are [illegible]u'll get a feeling for how the x86's memory and its stack are organized.

Each time you run th[illegible]gram will be given a different place in the x86's data memory to use (this i[illegible]ce layout randomization; take 551 for more details!). So you'll get different [illegible]e you run it and that is completely normal.

Your Task

You have three tasks to complete.

The first is 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. "deref'ed" means "dereferenced", it's just shorthand.

The second task is to complete the spreadsheet that you worked on in Problem 4, which actually has two worksheets within it. Switch to the problem7 worksheet (at the bottom of the Excel window). You need to complete this second spreadsheet.

Use the output of the execution of your program (after you complete the first task) 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!

One important compiler quirk to note is that it likes to put variables on addresses that end with 0 and 8. This is because it makes arithmetic easier. Therefore, you may find that some variables appear to have sizes that vary from run to run. Each type of variable takes the same amount of memory but because of this quirk, some variables may appear to take more space than expected. We do accept a few different values for some of the variables, but we do want you to think carefully about the sizes of each variable.

The last task is to update the Makefile. Add a rule to the Makefile to compile `program7` and also add `program7` to the `clean` recipe.

Requirements

- Add a target for `program7` to your makefile, and include it in your `clobber` recipe.
- Complete `program7` to print all the remaining variables.
- Complete the second worksheet in the Excel template.

Problems 8 and 9 - Debugging Basics

Overview

We have provided two programs: `program8.c` and `program9.c`. While each of these files will compile, they cor[illegible]event them from running correctly. The purpose of these two files is not [illegible]e programs visually, but instead using a debugging tool called GDB (for **G**[illegible]e this debugger, you must compile the program using the `-g` flag. The[illegible]ur program as it normally would, except it keeps all the line numbers and [illegible]e final executable file, which is very helpful for debugging!

Your Task

1. Watch the following tutorial on GDB:
   https://www.youtube.com/watch?v=bWH-nL7v5F4
2. Watch the GDB demo in Canvas.
3. Compile program8.c with the `-g` flag:

   ```
   clang -g program8.c -o program8
   ```

   Do note that we use the `clang` compiler in this course, and other resources may use `gcc`.

4. Add the above recipe to make `program8` to your makefile and include `program8` in the `clean` recipe.

5. 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.

6. Start `program8` inside gdb using the following command:

   ```
   gdb -q -tui ./program8
   ```

   Note that `-tui` enables Text User Interface and `-q` removes the copyright message

7. For grading purposes, we will check how you use gdb. Turn on the gdb logger with the following commands inside gdb:

   ```
   set logging on
   set trace-commands 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 -q -tui -ex "set logging on" -ex "set trace-commands on" ./program8
   ```

8. Now follow the steps in the tutorial to debug `program8.c`. Make the necessary changes such that `program8.c` produces the correct output.
9. Once you finish, repeat this debugging process for `program.9.c`.

Completely unrelated to the assignment, but I couldn't find anywhere else to put this:

You can pass command line arguments with `--args` like this:

```
gdb -q -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 `-Wall` (Warnings All) flag when compiling your programs in C by using the follow

```
clang -Wal                    o program8
```

Seriously, this should n your compile commands forever.

Requirements

- Add `program` ets to your makefile, and also include them in the `clobber` recipe.
- Enable gdb logging each time you start gdb. We expect to see actual debugging including inspection of variables and usage of breakpoints.
    - You will lose points if you fail to use gdb or log correctly, even if the program is bug-free upon submission.
- Correctly debug both programs.
    - These bugs are relatively simple to debug without gdb, but you must show evidence of gdb usage to receive any credit

Submission

Where to put the files

All of your files should be at the top-level directory (where the starter code started).

Be sure to upload J a

Pre-Submissic

There are no pre-sub

The Actual Sub

When you are ready (before the deadline), go to Education -> Mark Complete.

Grading

This assignment is worth 220 points, normalized to 100% for gradebook purposes.

All Problems have partial credit.

Problem 1 is worth 10 points.
Problem 2 is worth 10 points.
Problem 3 is worth 40 points.
Problem 4 is worth 60 points.
Problem 5 is worth 20 points.
Problem 6 is worth 40 points.
Problem 7 is worth 20 points.
Problem 8 is worth 10 points.
Problem 9 is worth 10 points.
There is no extra credit for this assignment.

Important Notes 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).

Hints or FAQs

-

Resources

- Intel's x86 ISA
  http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- Escape Sequ
  https://en.wiki e_sequences_in_C#Table_of_escape_sequences