Lab1: Implementing C Programs

Week 1

Goals:

- Learn how to write the Hello World program in C and learn how to compile your programs using gcc.
- Compile source files to object files and learn how to link them.
- Learn how Makefiles and header files work.

Description:

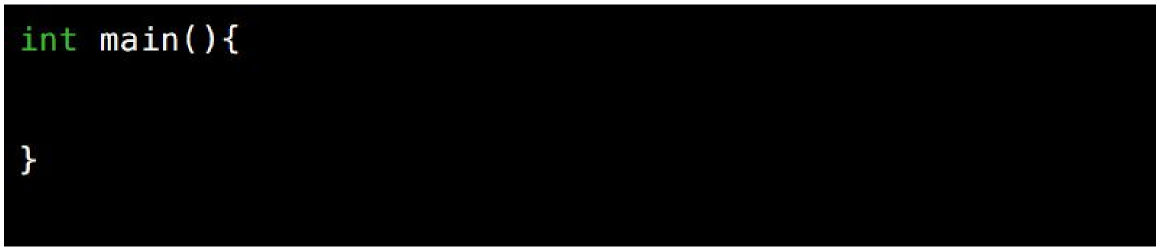
This lab will introduce you to the basics of writing, compiling, and running a program written in C. During this lab, you will be expected to complete 3 different tasks which are associated to each objective stated above.

Task 1: Write your first C program

The first step in learning a new language is learning how to get a basic program to compile and print a basic statement (traditionally "Hello, World!"). In this task, you will write your first C program and compile it with gcc (GNU C compiler).

Steps:

1. Go to ***lab1-src/***
2. Create a new file called "**hello.c**" using the text editor you prefer.
3. Every C program has one main() function. Start with a main function.



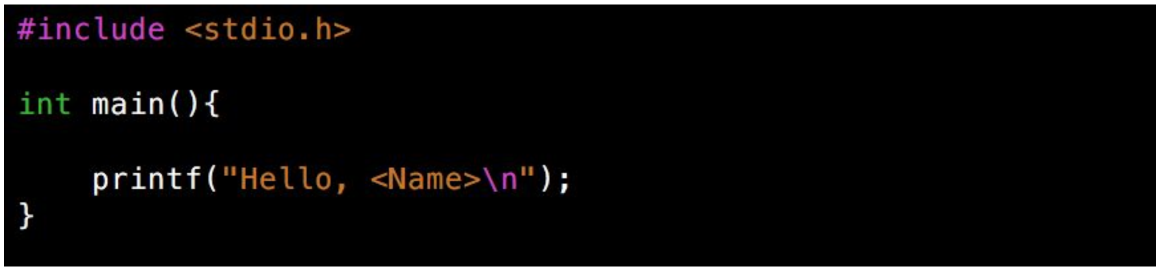
```
int main(){


}
```

➔ Note: In C, main returns an int, referred to as an exit status to the environment that invoked it (this will not concern us). While this form of main() is valid, we will later see how to modify it to accept command line arguments.

4. Print "**Hello! <name>\n**" in main. Change string "**<name>**" to your own name and save the changes you have made.

To print a statement, use **printf()**. In C, all functions must be declared before they are called. To use **printf()** from the C standard library, we need to include **stdio.h** as the header file at the beginning of the file.

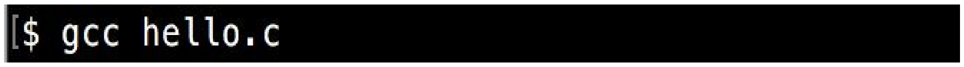


```c
#include <stdio.h>

int main(){

    printf("Hello, <Name>\n");
}
```

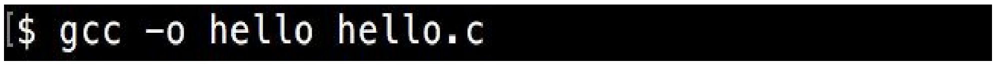
5. Compile your first program using gcc.



```
[$ gcc hello.c
```

- This will generate the default executable file "`a.out`".
- To run the program, simply type "`./a.out`".
- This will print the "**Hello! <name>**".

6. To rename the executable file, use –o option:



```
[$ gcc -o hello hello.c
```

This generates an executable file named **hello**. Similarly, running "`./hello`" will print the same output as" `./a.out`".

➔ Note: there are other useful flags as:

- **-std=gnu99** - Compiles against the GNU C99 standard, which we will use in this course. ▯
- **-g** - Enables debug information, allowing the use of a debugger (gdb) on the executable ▯
- -**Wall** - Enables all warnings ▯
- -**Werror** - Treats all warnings as errors ▯

Compiling with all warnings enabled, and treated as errors will help you catch many bugs as you learn C. With more experience, you will learn when they can safely be ignored. From now on, use **ALL** of these flags when compiling.

Task 2: Linking

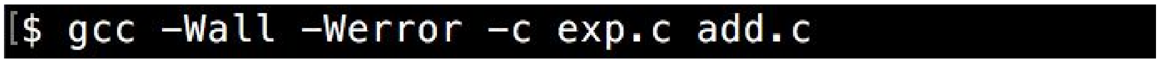
As we know, C Compiler does not compile the source code to executable files at one step. In fact, the C Compiler first compiles the source files to **object files** (**.o**) , then the linker would take the produced **object files** and **libraries** as an input combining them to produce a single (usually executable) file. To learn how the **Compiler** compiles the files further, we will compile source files to the **object files** with the " **–c**" option first , then compile them to final **executable files** with the "**–o**" option .

Given:

exp.c - prints the addition of 2 integers
add.c - adds two integers and returns the result

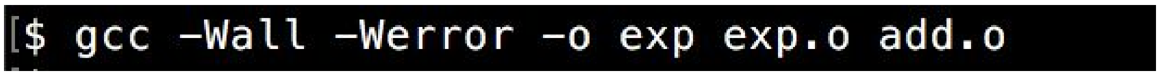
Steps:

1. Go to ***lab1-src/***
2. Compile "**exp.c"** and "**add.c"** to "**.o"** files.



```
[$ gcc -Wall -Werror -c exp.c add.c
```

3. Compile "**exp.o"** and "**add.o"** to the executable "**exp"**.

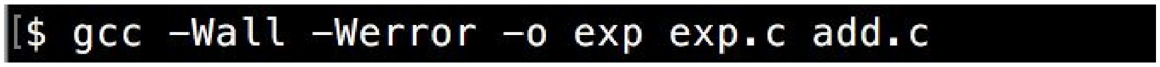


```
[$ gcc -Wall -Werror -o exp exp.o add.o
```

➔ You now have the "**exp"** executable which will produce the expected output from the linked files.

However **gcc** ,by default, both compiles and links files for you. This method is used most commonly because it simplifies the steps above.

4. Remove **.o** files and the executable file "**exp"**.
5. Compile exp_short.c and add.c to an executable by :



```
[$ gcc -Wall -Werror -o exp exp.c add.c
```

➔ You now have the "**exp"** executable, without having to compile the "**.o**" files separately.

Task 3: Makefile and Header File

Part 1: Makefile

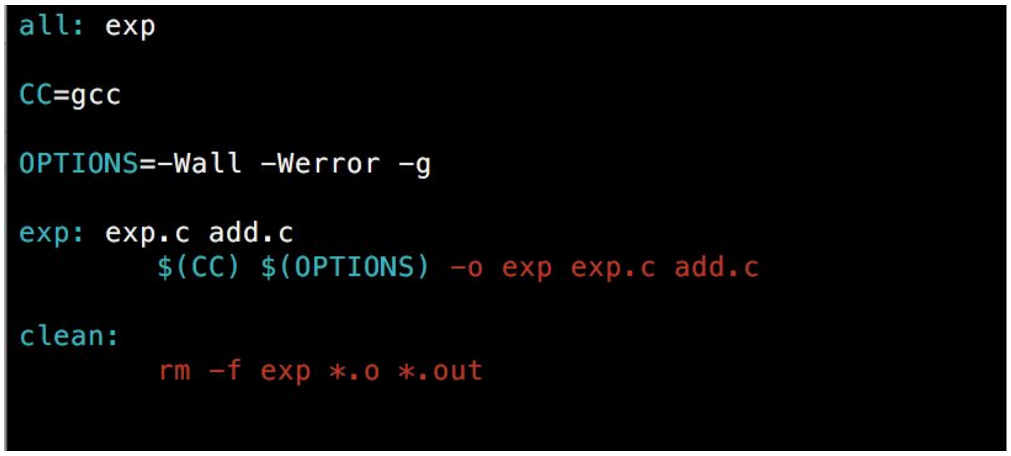
Another way to compile program is to use **make**. Particularly when you are building a large project, you don't want to type duplicate "gcc –o" to compile files every time modifications are made. In this case, **make** will save you loads of time compiling multiple files.

Given:

exp.c - prints the addition of 2 integers
add.c - adds two integers and returns the result
add.h - declare add function defined in **add.c**
Makefile - compile the source files to executable file **exp**

Steps:

1. Go to ***lab1-src/***
2. Use an editor of your choice to open the Makefile and you will see the following:



```
all: exp

CC=gcc

OPTIONS=-Wall -Werror -g

exp: exp.c add.c
	$(CC) $(OPTIONS) -o exp exp.c add.c

clean:
	rm -f exp *.o *.out
```

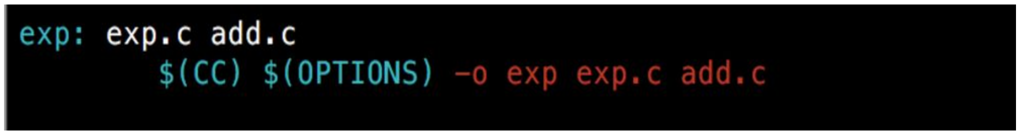
Below is a general makefile format.
Syntax:

```
target: dependencies
<tab> system command to be executed
```

Components:

- Targets (default target of make is 'all'.)
- Dependencies

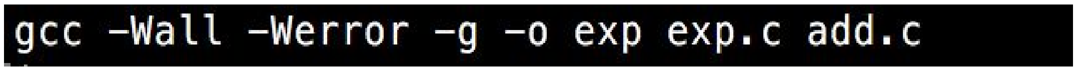
Ex:



```
exp: exp.c add.c
	$(CC) $(OPTIONS) -o exp exp.c add.c
```

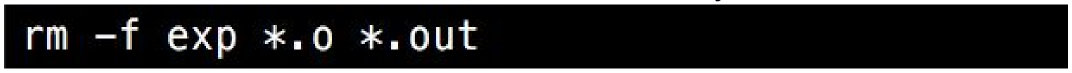
Target: exp is
Dependencies: exp.c, add.c are
System command: gcc -Wall -Werror -g -o exp exp.c add.c

When you run "**make**" (equal to **make all**) in the terminal, you are basically running the corresponding command :



```
gcc -Wall -Werror -g -o exp exp.c add.c
```

Similarly, running "**make clean**" helps clean up the executable file **exp** as well as all the **.o** and **.out** files in the current directory.



```
rm -f exp *.o *.out
```

3. Run **make** and compile the files.
4. Run the generated executable file **exp**.

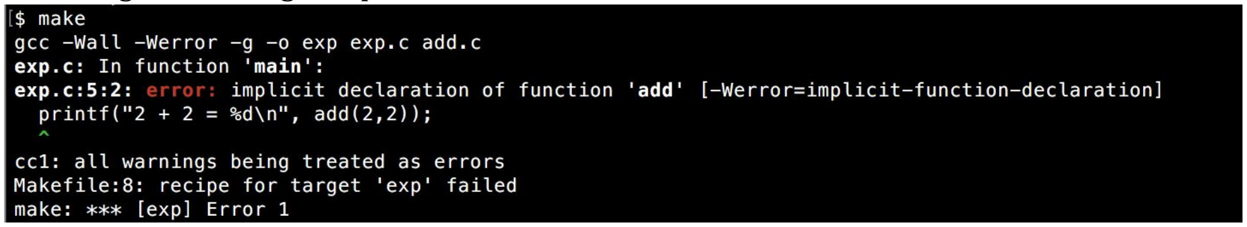
Part 2: Header File

In larger programs, code gets split across many source files. To allow code in one file to call a function in another file, the function can simply be declared in the calling file at the very beginning. To impose order on this situation, **header files** (**.h**) were introduced.

Steps:

1. Open **exp.c** with the editor you like.

2. Notice "`#include add.h`" at the beginning of the file and comment out that line.

3. Save the changes and run **make** again.
   You will get following compile error:
   

```
[$ make
gcc -Wall -Werror -g -o exp exp.c add.c
exp.c: In function 'main':
exp.c:5:2: error: implicit declaration of function 'add' [-Werror=implicit-function-declaration]
  printf("2 + 2 = %d\n", add(2,2));
  ^
cc1: all warnings being treated as errors
Makefile:8: recipe for target 'exp' failed
make: *** [exp] Error 1
```

   This is because you are using **add(int, int)** in the **main()** of **exp.c** without declaring **add(int, int)** first. As we know, the add function is declared in **add.h** and defined in **add.c**. Thus, to use the add function in **exp.c**, we have to include the **header file** where the add function is declared.

4. Now open the exp_short.c and uncomment "`#include add.h`"
5. Run **make** again and you will successfully compile the files.

Grading

Student Name	
Email	
Section	
Grader Name	

Tasks		**Points**	**Given**
1	Demonstration and understanding of Compilation of hello.c	**5**	
1	Correct output of hello.c	**5**	
2	Demonstration and understanding of Linking exp_short.c and add.c	**10**	
3	Demonstration and understanding of the Makefile	**5**	
3	Demonstration and understanding of the Header Files	**5**	
Total		**30**	