Flinders UNIVERSITY
College of Science & Engineering

COMP2881 Operating Systems
COMP9881 Operating Systems GE
Semester 1, 2019

CN Workshop01: Bootstrap your C knowledge

Due: Tuesday, 26 March, 2019 (check submission time on FLO)

Lab 1 is comprised of 5 tasks; each task is worth 1 checkpoint with, each checkpoint contributing 1% towards your topic grade. Following completion of the checkpoints, a copy of the source code and output for each task should be compiled into a single text file and uploaded to the relevant submission box on FLO for moderation. Speak to your demonstrator if you are unsure how to do this.

While discussion of programming tasks with fellow students in labs is welcome and encouraged, please be mindful of the University's **Academic Integrity** policy and refrain from simply copying another student's code (even if they agree!). You will gain a much greater understanding of the material if you take the time to work through your own solutions, and you will be expected to be able to explain your code before being awarded checkpoints. Remember, the demonstrators are there to help if you get stuck.

Objectives

This lab is intended to get you started using C in a Linux environment. The exercises should help you understand the process of compiling C code from the command line (manually and via makefiles) as well as provide you with the opportunity to practice some of the mechanics of the language. The lab is separated into five tasks. Each task is worth one checkpoint and 1% towards your topic grade.

Upon completing they lab, you should be able to:

- Extend C source code, adding functionality
- Compile C source code into an executable program
- Use function arguments in the scope of the function to manipulate strings
- Read user input
- Create new functions to process data
- Print out processed data
- Save the output text to a text file to submit as a hand in, in a *.txt file

Learning out comes;
- LO2: Demonstrate the design and implementation of simple process and network level programs

Preparation

You should also be familiar with basic Linux shell commands as discussed in lectures: `ls`, `cd`, `mkdir`, `pwd`, and `mv`. Remember, you can `man` a command to find out more information on its use.

As you work on each task, you should capture the output in a text file to submit to the hand-in for the workshop.

Task 1

The first task will show you how to compile and execute C code from the Linux shell. Before you begin, it is a good idea to think about how you will organise the files you create over the course of the semester. A good approach is to create a top-level directory (folder) named **cnlabs** and then child directories for lab work, assignments, and other topic materials.
Once you have a directory structure in place, download the **lab-01-files** archive from FLO and extract it to a location within your labs directory.

Next, open a **terminal** window (the default shortcut for this in Ubuntu is `Ctrl` + `Alt` + `T`) and change into the directory where you extracted the zip file. List the contents. There should be 5 files:

```
$ ls
circle.c  cnhello.c  hello.c  makefile  triangle.c
```

For this task, you will learn how to compile **hello.c**. Type the following command at the prompt to invoke *gcc*, the C compiler:

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

This will turn the C code in the source file into machine code that the hardware can execute. The `-o` option specifies the *output* file name. If this was omitted, the output file name would default to `a.out`. If you again list the contents of the current directory, you should now see an additional file representing the compiled (executable) **hello** program.

```
$ ls
circle.c  cnhello.c  hello  hello.c  makefile  triangle.c
                      ^^^^^
```

To run **hello**, type `./hello`. The `.` represents the current directory, so the command is equivalent to typing the full path to the program. Verify that the output appears as below.

```
$ ./hello
Success!
You've just completed the first task.
```

Checkpoint 1: show the output to a demonstrator and record the terminal session in a text file for submission.

Task 2

This checkpoint will teach you the basics of command line arguments. To begin, open the provided C file called **cnhello.c**, which should look something like:

```
#include <stdio.h>

int main(int argc, const char *argv[]) {
                ^^^^                ^^^^^^
  /* Replace this comment with your code */
  return 0;
}
```

Take note of the two parameters declared as part of the main function. These are used to process command line arguments passed to the program when it is executed: argc denotes the argument count and argv the collection (or vector) of argument values. Conventionally, the first argument value, argv[0], is the name of the program, so argc will always be at least 1. When processing arguments, you should check the value of argc to verify that an argument you expect has actually been provided. For example:

```
if (argc > 1) {
  // argc is at least 2, so we can safely access argv[1].
  // Remember, array indexing starts from 0.
}
```

Your task is to write your own version of *hello* that greets users with the message “Hello, [name]. Welcome to CN.” Your program should treat the first explicit command line argument passed to it as the name used in the greeting.

Use the **printf** function along with the `%s` format specifier to compose the message with the appropriate *argv* value. `%s` acts as a placeholder for an argument containing a series of characters called a *string*. The actual argument is provided after the format string like so: `printf("I am a %s \n", "string");`. There are many other format specifiers available to handle different data types; visit the link above to learn more. Also, don’t forget the `\n` at the end to insert a new line.

Note: The *printf* function is part of the *stdio* library, which needs to be included before it can be used. This is no different than the need to include the *java.util* package before using the *Scanner* class in Java. For this task, the relevant include statement has been provided for you.

Write your code in **cnhello.c**. When you are ready, compile the program by typing

```
$ gcc cnhello.c -o cnhello -std=c99 -Wall
```

This is the same command you used in the previous task except for the addition of two new compiler options. `-std=c99` tells the compiler to use a more recent version of the C standard, which will be useful for later tasks. The `-Wall` option instructs the compiler to output all warning messages. This is helpful for indentifying problems and is considered good practice.

If the compiler doesn't generate any warnings, run the program, passing *your* name as an argument, and observe the output. It should resemble the following:

```
$ ./cnhello Kim
Hello, Kim. Welcome to CN.
```

Checkpoint 2: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 3

So far you have been compiling your C code by invoking the gcc compiler directly and specifying appropriate arguments and options. However, this can become quite tedious after a while, especially as projects grow to include multiple source files and dependencies.

Makefiles are a means of scripting the compilation process to remove the need to type compiler commands manually. They are executed by the `make` program. Although makefiles themselves can grow to be quite large and complex, we will just focus on the basics for now.

Begin by opening the file named **makefile** from the directory where you extracted the lab files. Using VS Code, you can do this directly from a terminal window by typing the command `code makefile`.
Once the file is open, you should be presented with the following:

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

Makefiles are composed of **rules**. Each rule consists of a **target** followed by a list of **dependencies** and then the **commands** to be executed. The structure is shown below.

```
target: dependencies
[tab] commands
```

Commands must be indented with a tab character and not one or more spaces. If you look at the makefile provided, you will see that a rule for the *hello* program used in task one has already been created. The rule has a single dependency on the *hello.c* source file. The *make* program will check whether this file has been updated when evaluating the rule. If it has, the gcc command will be executed. To run a rule in a makefile, type `make` followed by the name of its target. For example, `make hello`.

Note: The rule at the top of a makefile specifies the default target. If no explicit target is provided when the *make* program is invoked, the rule for the first target will be processed implicitly.

Your task is to add another rule to the makefile for the program you wrote in the previous task. Use **cnhello** as the target name. Before you test your rule, rename the existing *cnhello* executable by typing `mv cnhello cnhello.old` so the *make* program can rebuild it.

Test that your rule works by invoking `make cnhello`. If all goes well, you should see an echo of the rule's gcc command. List the contents of the directory to confirm the existance of a new *cnhello* executable.

Checkpoint 3: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 4

Now that you have some knowledge of working with C code in a Linux environment, you will start writing a few basic programs to practice with the language.

Your task is to create a C program that prints a right-angled triangle using `*` characters (the symbol on the `8` key). Your code should output **5** rows, and each row should contain one more `*` than the last until it too equals **5**. Separate `*` on the same row by a space.

Hint: You will need to use more than one loop. You may use for loops, while loops or both. Paying attention to what you know about the data in advance.

Write your code in the template file named **triangle.c** and ensure you add a new rule to your makefile that will enable the program to be compiled using `make triangle`.

When you are ready, test your program and verify the output resembles the following:

```
$ ./triangle
*
* *
* * *
* * * *
* * * * *
```

Checkpoint 4: show your code, updated makefile, and program output to a demonstrator.

Task 5

The final task for this lab is to write a C program that calculates the **area** and **circumference** of a circle for a given radius. The radius will be read from *stdin* using the **scanf** function. Visit the link to learn more about its use.
Begin by opening the template file named **circle.c**. You will notice that some code has already been provided. Line 5, in particular, declares a **function prototype** for the *area* function that describes its interface in terms of name and signature (return type and paramters). Declaring a function before *main*, which is the entry point to the program, allows the compiler to ensure any calls to it are correct (e.g. provide the right type and number of arguments). The actual implementation of the function then appears below *main*; notice how it shares the same name and signature.
Before you start writing any code, it's a good idea to add a new rule to your makefile to enable the program to be easily compiled with `make circle`. Once you have done that, finish the program by doing the following:

1. complete the implementation of the **area** function so that it returns the correct value. The formula to calculate the area is: **PI** x **radius**^{2}.
2. add a **circumference** function prototype. Like *area*, the circumerference function should return a **double** and take a single **double** parameter named **radius**.
3. implement the **circumference** function so it returns the correct value. The formula to calculate the circumference is: **2** x **PI** x **radius**.

4. add a call to the **circumference** function in *main*.

Executing your finished circle program should produce output similar to:

```
$ ./circle
Enter a radius in metres: 6
Area: 113.10m
Circumference: 37.70m
```

Checkpoint 5: show your code, updated makefile, and output to a demonstrator.