Flinders UNIVERSITY
College of Science & Engineering

COMP2881 Operating Systems
COMP9881 Operating Systems GE
Semester 1, 2019

CN Workshop02: Understanding Input/Output

Due: Tuesday, 2 April, 2019 (check submission time on FLO)

Lab 2 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

Upon completing this lab, you should be able to:

- Format C code according to established conventions and best practice
- Determine the memory size of basic data types
- Use pointers to access and modify array elements
- Create unit tests

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 C compiler doesn't care how whitespace is used within a source file; as long as the syntax is correct, the program will compile. However, poor use of whitespace has a negative effect on code readability. If you find yourself working on a project as part of a team, you will be expected to follow conventions and consistently format your code to ensure it is legible by others.

1. Open the file named **task1.c**. The source code is unformatted and intentionally terse. While the code does compile, there is a small logic error you will need to fix. Formatting the code should help you find it.
2. Reformat the code so it is consistently indented and spaced. This will involve proper use of whitespace to separate symbols and indent/break lines as appropriate.

3. Once you have reformatted the code, find and fix the logic error.

4. Compile the program with make/gcc. The correct output should be the first 15 **Fibonacci** numbers:

```
$ ./task1
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
```

Checkpoint 11: show the output to a demonstrator and record the terminal session in a text file for submission.

Task 2

When working on programs with memory constraints, it is useful to understand the amount of space occupied by the various types available within C. For this task, you will make use of the **sizeof** operator to explore this.

1. Create a new C file called **task2.c**. Add a main function and #include the **stdio.h** library.

2. Write a C program to output the size of the following data types for your machine using the **sizeof** operator.

   ```
   char
   short
   int
   long
   float
   double
   char *
   int *
   double *
   ```

   Format your output so the size is reported in **bits**. What do you notice about the pointer types?

3. Inside a comment block underneath your code, write down what you *think* the size of the following will be:

   ```
   double darr[7]
   "Flinders University"
   ```

4. Extend your program to check your assumptions. As before, report the size in **bits**.

5. Is there a way you could determine the length of an array by utilising the sizeof operator?

Checkpoint 12: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 3

Write a C program that duplicates the functionality of the **strlen** function to calculate the length of a string.

1. Create a new C file called **task3.c**. Add a main function and #include the **stdio.h** library.

2. Copy the following string literal declarations into your main function:
   const char *a = "";
   ```
   const char *b = "Tonsley";
   const char *c = "Flinders University";
   ```

3. Declare a new function with the following signature:

```
int string_length(const char *str)
```

Following good programming practice, *str* is declared *const* as the function does not need to modify its contents in order to count characters.

4. Implement the function definition so it calculates and returns the length of the string. Consider how you can determine when there are no more characters left to be counted.

5. Call your function from main three times, once with each of the three string literals, and print the result to *stdout*. Your output should include the string value surrounded by single quotes in addition to the length, e.g.

   ```
   4 characters in 'Unix'
   ```

6. Compile the program with make/gcc and observe your output reports the correct number of characters.

7. Answer the following questions:
   What would happen if the string was not properly terminated with a null character?
   What would be the length of the string literal
   `"Operating\0System"`

Checkpoint 13: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 4

Write a C program that reverses the contents of a string using pointers.

1. Make a copy of your task3.c source file and name it **task4.c**.
2. Declare a new function with the following signature:

```
void reverse(char *str)
```

3. Implement the function definition so it reverses the characters in *str*. You may wish to utilise your `string_length` function from the previous task to help.
4. Modify your main function to pass the three string literals as arguments to *reverse*.
5. Print each reversed string to *stdout* surrounded by single quotes. The expected output is as follows:

```
$ ./task3
''
'yelsnoT'
'ytisrevinU srednilF'
```

Checkpoint 4: show your code, updated makefile, and program output to a demonstrator. Add to your text file for submission

Task 5

When we start to build modules in our C programs, we would like to know that when we make small changes, everything still works as expected.

Programmers across a variety of languages have implemented unit testing frame works to make sure that the possibility of side-effects from code changes are reduced. The Test Anything Protocol [1] support numerous languages[2], and in particular we would like to use CTAP[3].

With a standardised approach to testing, we can integrate a test phase in our build system within our make files. Part of this process is as follows;

```
make clean
make task1
make test
```

If we set up the dependencies in the make file correctly then the make test item can depend on the previous two. Additionally we can include a clean rule to remove files for us. We can create the dependencies like so;

```
test:     clean task5

clean:
     rm task5 task5.o reverse.o
```

Compiling the source code into an object file will require the compiler not to perform the linking stage, we pass the compiler the -c flag like so;

```
task5:    task5.o
     gcc -o task5 task5.o

task5.o:  task5.c ctap.h reverse.h
     gcc -c -o task5.o task5.c

reverse.o:     reverse.c reverse.h
     gcc -c -o reverse.o reverse.c
```

1. Download the workshop files from the FLO page.
2. In your makefile, implement the build rules for test5 and task5 to compile the respective .c files into .o files, then compile the test1 program from the .o object files.
3. Copy the function you created in task4.c, into a new file reverse.c and use it to create a revised string
4. Modify the function to return a "string" and then match the output text with an "is" rule[3] like the following;

```
const char *a = "Tonsley";
char *b = reverse(a);
```

```
// test 1
ok(string_length(a) == string_length(b), \
    "Strings are the same size");
// test 2
is("yelsnoT", b, "Strings match")
```

[1] https://en.wikipedia.org/wiki/Test_Anything_Protocol
[2] http://testanything.org/
[3] https://github.com/jhunt/ctap

Checkpoint 15: show your code, makefile, and output to a demonstrator. Add to your text file for submission