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:

- Review manual page documentation to learn how specific C library functions work
- Read data from standard input (stdin) and write data to standard output (stdout)
- Use I/O redirection
- Utilise operating system function calls to control process execution
- Manipulate the contents of character arrays.

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

In the previous Lab, you created a greeting program that utilised a command line argument to determine the name used in the welcome message. While arguments are extremely useful for passing short values and configuration options to your program, they are not well suited for large blocks of text.
In this task, you will replicate the Unix *echo* program to explore a method of processing textual data from standard input (stdin) as well as utilise input redirection to substitute keyboard entry for the contents of a file.

1. Begin by creating a new C file called **task1.c** and declaring a main function. If you are using VS Code, you can create a file from the command line by navigating to your lab two folder and typing `code task1.c`. If you need to refresh your memory on the format of the main function, refer to Task 2 of Lab 1.

2. The program should print its input, read from stdin, to the terminal standard output (stdout). To achieve this, you will need to use one function to read character data and another to write it. Read the descriptions of getchar and putchar online or consult their man pages. Note both are part of the **stdio.h** library so you will need to `#include <stdio.h>` at the top of your source file.

3. As both getchar and putchar operate one character at a time, you will require a local variable to store the next character. Declare an **int** variable with a suitable name in your main function.

4. Underneath the variable, define a while loop that will iterate as long as there are still characters to read. The documentation for getchar states the return value will be **EOF** when the end of file stream has been reached (i.e. there are no more characters), so the loop condition should reflect this. A pseudocode implementation would look something like:

```
int nextChar
nextChar = getchar
while (nextChar is not EOF) {
     ...
    nextChar = getchar
}
```

5. Within the while loop, call the putchar function with the variable declared in (3) as the parameter. This will output the character to the terminal (stdout).

6. Compile the program using gcc/make.

7. Run the program with no arguments. You will be presented with a blank prompt. Type a few words and press **Enter**. You should see the words repeated (echoed) underneath like so:

```
$ ./task1
The quick brown fox [Enter]
The quick brown fox
```

8. Press Ctrl+D to manually send EOF to the program and exit. Run the program again but this time invoke it by typing

```
$ ./task1 <OperatingSystem.txt
The addition of < operator will cause the contents of
OperatingSystem.txt to be redirected
as the input to the program instead of the keyboard.
```

9. Observe the output. You should see something similar to the following:

```
$ ./task1 <OperatingSystem.txt
An operating system is a layer of sophisticated software that
manages hardware
resources and provides a common interface for user programs.
Popular desktop
operating systems include:
- Windows
- macOS
- Linux
```

Checkpoint 6: show the output to a demonstrator and record the terminal session in a text file for submission.

Task 2

In this task, you will modify the echo program to wait for a brief interval between outputting each word. Remember to consult the man page (online or via the terminal) of commands/functions to learn more about their use.

1. Create a copy of your task1.c source file and name it **task2.c**. You can copy the file and open it in VS Code from the command line by typing:

```
$ cp task1.c task2.c
$ code task2.c
```

2. Add an additional `#include` statement at the top of the source file for the **unistd.h** library, which provides access to operating system functions.

3. Add an if statement above the call to putchar that checks whether the current character is a space (i.e. between words). Remember, characters are surrounded with single quotes and strings with double quotes.

4. Inside the if statement, add a call to the fflush function, which will ensure the output buffer is written immediately.

```
if (...) {
    fflush(stdout);
}
```

5. Utilise the **usleep** function to add a pause of **0.2 seconds** between printing words.

6. Compile the program using gcc/make.

7. Run the program using input redirection with the same input file as Task 1 (OperatingSystem.txt):

```
$ ./task2 <OperatingSystem.txt
```

Verify there is a short delay between the output of each word.

Checkpoint 7: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 3

In this task, you will once again modify the echo program to output each word on a separate line.

1. Create a copy of task1.c and name it **task3.c**.

2. Add an additional `#include` statement at the top of the source file for the **ctype.h** library, which provides character class test functions.

3. Modify the while loop body so that the program outputs each word on a separate line. You should discard any characters that are not alphabetical, so numbers and punctuation should not appear.

4. Compile the program using gcc/make.

5. Run the program using input redirection with the same input file as Task 1 and 2 (OperatingSystem.txt). Observe the output. It should resemble the following:

```
$ ./task3 <OperatingSystem.txt
An
operating
system
is
a
layer
of
sophisticated
software
that
manages
hardware
resources
and
provides
a
common
interface
for
user
programs
Popular
desktop
operating
systems
include
Windows
macOS
Linux
```

6. So far, you have been redirecting input (stdin) from a text file rather than entering it manually with a keyboard. The output (stdout) can also be redirected instead of being printed to the terminal. The process is similar to input redirection. Run the program again but this time add an additional argument as follows:

```
$ ./task3 <OperatingSystem.txt >Words.txt
                              ^^^^^^^^^^
```

Executing the program with these options produces no output; the output will have been written to a file called Words.txt. View the file using `cat` to verify its contents.

Checkpoint 8: show your code and output to a demonstrator and record the terminal session in a text file for submission.

Task 4

In this task, you will replicate (and improve on) the Unix *wc* utility to output text statistics for an input stream.

1. Copy your task3.c source file and name it **task4.c**. This file already #includes the ctype.h library, which provides functions you may find useful.

2. Add integer variables to record the number of **lines**, **words**, **characters**, **vowels**, **uppercase characters**, and **lowercase characters** in the input stream. As we have discussed in lectures, un-initialised variables in C have undefined values, so ensure all variables are appropriately initialised.

3. In the body of the while loop, implement suitable logic to populate the variables with correct values.

4. Add a series of **printf** statements following the loop to output the values to the terminal (stdout). The numbers should be **right-aligned** with a column width of **3**.

5. Compile the program with gcc/make and run it with OperatingSystem.txt as the input. Verify the output is as follows:

```
$ ./task4 <OperatingSystem.txt
  4 lines
 32 words
209 characters
 65 vowels
166 lowercase
  6 uppercase
```

You can check the number or lines, words, and characters against the built-in wc utility by executing `wc OperatingSystem.txt` .

Checkpoint 9: show your code, updated makefile, and program output to a demonstrator.

Task 5

This task will require some independent investigation. Your goal is to make use of character arrays to find the longest word in Words.txt.

1. Begin by creating a new file named **task5.c** and pasting in the template code below. You will need to fix the while loop so it reads characters from stdin until EOF.

```
#include <stdio.h>

static const int max_word_len = 50;

int main(int argc, const char *argv[]) {
     int c = 0;
     int i = 0;

     while (... != EOF && i < max_word_len-1) {

     }

     return 0;
}
```

2. Declare a new function with the following signature:

```
void copy(char src[], char dst[], int count);
```

   Implement the function definition so it copies the contents of into `dst` until `count` items have been copied or a `'\0'` character is read. The character should be copied.

3. Declare two **char array** variables inside the main function to store the current word and the longest word. Make the size of both equal to `max_word_len` .

4. Declare and initialise a variable inside the main function to record the length of the longest word.

5. Define the body of the while loop so it finds the longest word. A good approach is to continue reading characters into the *current word* array until a sentinel value is read (what separates words in Words.txt?). You can then check whether the length of the current word is greater than the longest word. If it is, store the word in the *longest word* array and remember its length. In either case, you will need to reset any counters you are using in preparation for reading the next word.

Checkpoint 10: show your code, updated makefile, and output to a demonstrator.