COMP26120
Academic Session: 2019-20

Lab Exercise 2: Input/Output; Strings and Program Parameters; Error Handling

Duration: 1 lab session

You should do all your work on the ex2 branch of the COMP26120_2019 repository - see website for further details. You will need to make use of the existing code in the branch as a starting point. Note - the PDF version of these notes may be easier to read as the HTML version is produced automatically.

Important: If you are new to C then you should read the supporting material on how to compile and run C programs first before lab session. Otherwise you will find this lab difficult.

Also Important: After submitting your code log in to COMPjudge to check that you are passing all of the tests. If you are not then you can resubmit.

Learning Objectives

At the end of this lab you should be able to:

1. Compile and link C programs (using *make*) and then run them.
2. Explain some major differences and similarities between C and Java.
3. Find out about different (basic) features of the C language by looking in man pages and other supporting tools.
4. Create and use strings and arrays in C

Introduction

This lab consists of two parts. The first part explores basic functionality of C programs and focuses on input and output. The second part introduces you to C strings (which are fundamentally different from Java strings) and how to deal with command line arguments in C. For your understanding it would be better to get half way through both than to finish one and not start the other. Through both parts we will also explore the idea of *error handling* in C programs.

Part 1 - C Input/Output

In this part you will produce a file called *part1.c*. You will modify this file as you go along so it would be a good idea to keep a working back-up each time you progress to a new step.

Step 1A

Write a C program to read characters one-by-one from standard input (you can use ctrl-D to terminate the input), convert all upper-case characters to lower-case and all lower-case characters to upper-case, and write the result to standard output e.g.:

> Hello World!

would become:

> hELLO wORLD!

You should also count how many characters you have read, and how many of those you have converted in each direction, and output the totals at the end in *exactly* the following format:

> Read 13 characters in total, 8 converted to upper-case, 2 to lower-case

remember a newline at the end of the line.

It is important that you use this exact format as the online marking system will expect it. But note that you can see the result of these tests after you submit and resubmit if the tests are failing – there is no good reason to have failing tests as you can check them yourselves.

As always, it might help to sketch the psuedocode for the algorithm first and then think about what functionality you need to implement it. It might be informative to reflect at this point how you would have done this in Java.

Hints: You will need to use `getchar` and `putchar` for the individual characters, as well as `printf` for the final character counts. You may want to use some of the functions in *ctype.h*, such as `tolower` and `isupper` (e.g. use *man ctype.h*, look at supporting material for what header files are).

Step 1B

Edit your program so that the input is read from a file opened from within your program. For the time being, you can use a fixed file-name such as "input".

Remember to check that the file is correctly opened. Test this by running the program when there is no input file available. If opening the file fails write something relevant to *stderr* - we will return to how to handle errors in C properly later.

Hint: Refer to *man fopen* and *SalaryAnalysis.c*.

Step 1C

Edit your program so that the updated input file is written to a file, using a fixed file-name such as "output".

The summary of what was read and converted should still be output to `stdout` **and not to the output file**.

Remember to check that the file is correctly opened. Test this by running the program when a file of that name exists, but is write protected (e.g. *chmod u-w output*)

Step 1D

Edit your program so that, instead of using fixed filenames such as "input" and "output", both filenames are read from standard input when the program is run (i.e. use `scanf` or similar; I don't want you to use command-line parameters). The online tests will supply the input and output filenames to *stdin* one after the other.

Part2 - Strings and Program Parameters

In this part we want you to start thinking more about how you write your code (note that there are marks for well-formatted and commented code throughout the course) in particular by writing *tests* and performing proper *error handling*.

You will need to create a suitable set of tests for each problem, which both check that your program works correctly given sensible data, and that it detects any possible errors, including silly or missing inputs. In part b we will discuss how you should handle errors properly.

a. Length of Program Parameters

Write a C program *part2a.c* that, given any number of program parameters (command-line parameters), calculates the length of each one, and writes the longest to standard output. In the case of a tie-break the first parameter should be output.

Before you begin, create some interesting tests for your program **in your makefile**[1], so it is easy to run them after each change. For example, you could add something like this (leave a blank line before the "test2a" line, and each of the following command-lines should start with a tab):

```
test2a:
    ./part2a #no parameters
    ./part2a "only␣one␣parameter"
    ./part2a "biggest␣parameter" "at" "start"
    ./part2a "biggest" "parameter" "at" "end" "very␣very␣very␣big␣parameter"
    ./part2a "answer" "somewhere" "in" "the" "middle"
    ./part2a "two" "strings" "the" "same" "length" "ha␣ha␣!"
```

and then run it using *make test2a*. Note that `make` expects an exit code of 0 (you did remember to return 0 didn't you?)

Hints: There are many functions that act on strings in *string.h* You should use one of these to find the length of a string.

But what should we do if there are no inputs? It is a matter of debate whether this is an error or not but that should be part of the specification. For now print the line

```
1 Error: expected some command-line parameters
```

and we will consider a better way to handle errors in the next part.

[1] If you're not sure what a makefile is then go back and read the supporting material.

b. Temperature Conversion

Write a C program *part2b.c* that will convert temperatures between Celsius and Fahrenheit, and is controlled by program parameters. Your program should accept two parameters in the following way:

- `part2b -f number`
  converts the number from Fahrenheit to Celsius
- `part2b -c number`
  converts the number from Celsius to Fahrenheit

where number is any floating point number. e.g.

```
> part2b -f 50.0
10.00°C = 50.00°F

> part2b -c 10
10.00°C = 50.00°F
```

We suggest you achieve the above by first writing the two functions

- `float c2f (float c)`
  which takes a temperature in Celsius and converts it to Fahrenheit (f = 9*c/5 + 32)
- `float f2c (float f)`
  which takes a temperature in Fahrenheit and converts it to Celsius

Your program should handle erroneous input, such as the wrong flag or the wrong number of parameters or temperatures below absolute zero (-273.15°C). In C the standard way to handle errors is to return a non-zero exit status that can be interpreted to understand the error. Throughout this part you must use the following exit statuses.

Status	Description
1	The wrong number of parameters have been supplied
2	Failed to supply either `-f` or `-c` as the first option
3	Malformed number in second parameter
4	Temperatures below absolute zero

Note that you should use the lowest numbered exit status that applies e.g. `-x` should return 1 rather than 2.

Create some interesting tests for your program in your makefile, so it is easy to run them after each change. Note that these tests will be checked when you are marked. To test that the correct exit codes are produced you can use something like

```
./part2b && exit 1 | [ $$? -eq 1 ] #no parameters
```

in your makefile (can you work out what this is doing?).

Note: In C99, the compiler will issue a warning if a function is called before its declaration.

Hint: Use `sscanf` to convert a string into a floating point number. You should check the return result to ensure that you have found a valid floating point number.

Again, so that the automated tests work you must format your output as shown above. Importantly you must:

- Print numbers to two decimal places
- Include a single space on either side of =
- Use the degree symbol ° as specified in *man iso_8859-1* (part of the ISO/IEC 8859 series of ASCII-based standard character encodings).

Submission

Make sure the three files *part1.c*, *part2a.c*, and *part2b.c* are added to your *ex2* branch and then tag (with *ex2_solution*) and push (with tags) – you can use the `submit.sh` script to do this for you. Remember, you should check COMPjudge to see a report that will tell you whether you have passed all of the tests. If you haven't then you can resubmit (which will require you to remove and reapply the tag, the `submit.sh` script handles this for you).

Marking Scheme

The marks are awarded according to the following marking rubric. Similar marking rubrics will be used throughout this course. The marking scheme is formed of a sequence of questions where the answer to each question is associated with a particular mark. These are the questions that your markers will be asking themselves during marking.

Is Part 1 complete?	
All steps have been completed and work as expected (i.e. the tests pass), including error handling in the handling of files	(2)
All steps have been completed but some may have small errors or miss special cases	(1.5)
An attempt has been made to complete at least two of the steps	(0.5)
No attempt made	(0)

Is Part 2 complete?	
Both steps have been completed and work as expected (i.e. the tests pass), including error handling	(2)
Both steps have been completed but there is a small mistake	(1.5)
Both parts have been attempted, with one part complete, and everything completed works as expected (i.e. the tests pass)	(1.5)
Both parts have been attempted and what has been completed mostly works but there are a few mistakes	(1)
A non-trivial attempt has been made at one of the parts	(0.5)
No attempt made	(0)

Does Part 2 have good tests?	
A reasonable number (at least 5) of tests have been written for each part providing good coverage of the functionality. The tests can be run automatically.	(2)
Some tests have been written for each part but they miss important cases or cannot be run automatically	(1.25)
An attempt has been made at writing some tests but they are inadequate	(0.5)
There are no tests	(0)

Does the student understand their code?	
The student can explain how the code works, including what the various library functions do and how the compilation process works	(2)
The student can answer most but not all questions about how the code (etc) works	(1.25)
The student can answer some but not most of the questions about how the code (etc) works	(0.5)
The student does not understand their code	(0)

Is the overall quality of all code good?	
Code is well formatted and commented with appropriate variable names	(2)
Code is mostly well formatted and commented but there are some missing comments or poor variable names	(1.5)
There is at least one comment	(0.5)
There are no comments	(0)
The code is not readable	(0)