Lab Exercise 3: Arrays and Strings

Duration: 1 session

Aims

To encourage you to find out about some of the data-structure features of the C language.

Useful C on-line course themes etc.:

- Differences between Java and C
- Lots of information about Arrays and Strings in Information Representation

Learning outcomes

On successful completion of this exercise, a student will:

- Know how to use arrays and strings in C.
- Know how to use program parameters (command-line parameters) in C.

Summary

Write five C programs that use arrays, strings and program parameters:

- Write `part1.c`, that will find the largest element in arrays of integers;
- Write `part2.c`, that will print chessboard-patterns of any size;
- Write `part3.c` that, given any number of program parameters, writes the longest to standard output;
- Write `part4.c`, that will convert temperatures between Celsius and Fahrenheit, and is controlled by program parameters;
- Write `part5.c`, that will count the different characters in a file;

and create a suitable set of tests for each one, which both check that your program works correctly given sensible data, and that it detects any possible errors, including silly or missing inputs.

The unextended deadline is the end of your scheduled lab.
If you need it, if you attend the lab you will get an automatic extension to the start of the next scheduled COMP26120 lab session (you must use submit to prove you finished in time and get it marked at the start of your next scheduled lab).
You can also get an extension for good reason e.g. medical problems.

Description

For this lab exercise you should do all your work in your `COMP26120/ex3` directory.

Copy the starting `makefile` into your `COMP26120/ex3` directory from `/opt/info/courses/COMP26120/problems/ex3` (use this path).

Debugging

There is every chance that at some point your program will crash with the message: `Segmentation fault`. If you can't work out what happened, use a debugger (e.g. `man gdb` or `man ddd`) e.g. to use `ddd`:

- Run ddd, loading the program that is crashing e.g. `ddd part1`

- Click "Run" on the buttons, or Program->Run
- If you get a pop-up window for arguments etc., just click Run
  The top pane will use a big red arrow to point to just before the line of code where the program crashed. The bottom pane will give more detailed information e.g. about function parameters and the line number.
- Click "Up" on the buttons, or Status->Up
  The top pane will use a big grey arrow to point to just before the line of code where the previous function was called. Again, the bottom pane will give more detailed information.
  Repeat this step if you need to.

I hope this will help give you some idea where to look for the problem. If you can't see it, then at least you know where to start putting `printf` statements. (There is more about debugging in next week's lab script.)

Part 1: 1-Dimensional Arrays

Write a C program `part1.c` that will find the largest element in arrays of integers.

First write a `main` body that creates and initialises a one-dimensional array of integers. Then write a function `largest` that takes two parameters - the array, and its length - and returns the index of the largest element in the array. Make your program call the function and then (back in `main`) print out the answer.

Now modify your `main` to do this for several different test arrays. Create some interesting test arrays to thoroughly test your program.

Part 2: 2-Dimensional Arrays

Write a C program `part2.c` that will print chessboard-patterns of any size.

First write a `main` body that declares a two-dimensional array of a given size to represent a chess board. Then write a function that, given the board and its length and width as parameters, will initialize it so that alternate squares are black and white. Make your program call the function and then (back in `main`) print out the resulting board using suitable characters to represent the black and white squares.

Test your your program by making it print out several boards of different sizes. You should declare several arrays, each with a different length and/or width, and call your function to initialise each in turn.

Hint: Use an enumerated type to define 'black' and 'white'.

Part 3: Strings and Program Parameters

Write a C program `part3.c` that, given any number of program parameters (command-line parameters), calculates the length of each one, and writes the longest to standard output.

Create some interesting tests for your program in your `makefile`, so it is easy to run them after each change. For example, you could add something like this (leave a blank line before the "test3" line, and each of the following command-lines should start with a tab):

```
test3:
        ./part3 #no parameters
        ./part3 "only one parameter"
        ./part3 "biggest parameter" "at" "start"
        ./part3 "biggest" "parameter" "at" "end" "very very very big parameter"
        ./part3 "answer" "somewhere" "in" "the" "middle"
        ./part3 "two" "strings" "the" "same" "length" "ha ha !"
```

and then run it using `make test3`

Hints:
Program Parameters (argv, argc)

`argv[0]` probably isn't what you expect it to be.
There are many functions that act on strings in `string.h` You should use one of these to find the length of a string.

Part 4: Temperature conversion

Write a C program `part4.c` that will convert temperatures between Celsius and Fahrenheit, and is controlled by program parameters.

Write two functions:

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

and a program that accepts two parameters:

- `part4 -f number`
  converts the number from Fahrenheit to Celsius
- `part4 -c number`
  converts the number from Celsius to Fahrenheit

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

```
> part4 -f 50.0
10.00°C = 50.00°F

> part4 -c 10
10.00°C = 50.00°F
```

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).

Create some interesting tests for your program in your `makefile`, so it is easy to run them after each change.

Hints:
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.
You can use e.g. `man iso_8859-1` to find the degree symbol °

Part 5: counting characters

Write a C program `part5.c` that will count the different characters in a file.

Write a program that, given a file-name as a program parameter, will read from it to count how many instances of each different character are in the file, and then output the counts. You should use an array of integers to hold the counts. Only output non-zero counts at the end.

e.g., for the paragraph above, you should get:

```
    1 instance  of character 0x0a (               [i.e. only one new-line]
)
   50 instances of character 0x20 ( )
    3 instances of character 0x2c (,)
    2 instances of character 0x2d (-)
    3 instances of character 0x2e (.)
    1 instance  of character 0x4f (O)
    1 instance  of character 0x57 (W)
    1 instance  of character 0x59 (Y)
   22 instances of character 0x61 (a)
```

```
     8 instances of character 0x63 (c)
     6 instances of character 0x64 (d)
    24 instances of character 0x65 (e)
     7 instances of character 0x66 (f)
     4 instances of character 0x67 (g)
    11 instances of character 0x68 (h)
    10 instances of character 0x69 (i)
     7 instances of character 0x6c (l)
     6 instances of character 0x6d (m)
    19 instances of character 0x6e (n)
    19 instances of character 0x6f (o)
     5 instances of character 0x70 (p)
    17 instances of character 0x72 (r)
     9 instances of character 0x73 (s)
    25 instances of character 0x74 (t)
    11 instances of character 0x75 (u)
     1 instance  of character 0x76 (v)
     2 instances of character 0x77 (w)
     3 instances of character 0x79 (y)
     1 instance  of character 0x7a (z)
```

Create some interesting tests for your program in your `makefile`, so it is easy to run them after each change.

Hint: in C, all characters are represented by small integers in the range 0 to 255.

Marking Process

You must use `labprint` and `submit` as normal. They will look for: `part1.c`, `part2.c`, `part3.c`, `part4.c`, and `part5.c` (If you haven't attempted a part, use e.g. `touch part5.c` to create an empty file for `labprint` and `submit` to use.)

The marks are awarded as follows:

```
2 marks: Part 1 working completely, with sensible tests
2 marks: Part 2 working completely, with sensible tests
2 marks: Part 3 working completely, with sensible tests
2 marks: Part 4 working completely, with sensible tests
2 marks: Part 5 working completely, with sensible tests

Total 10
```