P2T: C Programming under Linux

Lab 3: Arrays, Strings and Structs

Introduction

The purpose of this laboratory is to help you become familiar with the concepts of arrays, strings and structures in C programming. This material has been covered in the Lecture 4 of the C component of P2T: C Programming under Linux. In this lab we will concentrate on arrays and strings while structures will be briefly covered.
This C Lab contains three Challenge Questions, which are the work which you will be marked on. All Challenge Questions, in all C Labs, are worth 10 points in their respective lab, unless explicitly noted. In general, you can assume that you will get at least 1, and up to 3, points for laying out well-presented, well-commented code; the rest of the marks will depend on the question (although partly-functional answers will get marks for the parts which work).

Arrays

An array is a data structure that allows us to group together values of the same type. It is required by the standard to be implemented as a set of consecutive memory locations (so, you can think of it as creating a "row of boxes" for values). A typical array declaration would look like:

```
int data_list[3];
```

The declaration of an array is much like that of a "scalar" variable, with an additional suffix. The declaration includes the type specifier at the beginning (`int` in this case) which is the type of each of the elements in the array. The name of the array follows along with an additional parameter given by suffixing the name with square brackets. The integer value between the square brackets gives the number of elements in the array (in the above example it is 3).

To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array. In the above declaration, for example, `data_list[0]` is the first element of the array, `data_list[1]` - 2^{nd} and `data_list[2]` - 3^{rd} element. In general, the i^{th} element

of any array is refered to as `array[i-1]` because C starts counting indexes from 0. (It may help to think of the index as "the distance from the start of the array", rather than an ordinal number in itself.)

You should bear in mind that arrays are not identical in behaviour to the variables you have met so far. A common mistake is to assume that testing the equality of two arrays will test the pairwise equality of each of their elements - this is not the case! In fact, equality testing for array variables tests if they are stored in the same place... (more on this in due lectures). In order to compare two arrays, therefore, we need to compare each pair of elements in turn. This requires accessing the first element of each array, comparing them and then moving on to the second element of each array, etc. Generally, of course, you would write a loop to perform this test.

The arrays we have covered so far are 1-dimensional arrays in that they only have a single index associated with them. However, data is often multi-dimensional (e.g. matrices, spreadsheets, 3-D coordinate systems). Declaring an array with multiple number of square brackets ([]) after the name produces a multidimensional array. So, if you declare an array with the following:

```
float coordinates[3][3][3];
```

A 3-Dimensional array will be initialised (this example can be visualised as a cube with the sides of length 3). In fact, you can declare as many dimensions as you need, but going higher than 4 dimensions is difficult to visualise/understand and it is not very practical. (In addition, C's implementation of multi-dimensional arrays can become unwieldy with more than a few indices.) As an implementation note, C considers multidimensional arrays to be "built" from right to left - so an array declaration like

```
float twoD[4][2];
```

creates an array of 4 arrays of 2 floating point values each (and never an array of 2 arrays of 4 floating point values). What this means is that the right-most index will always "move between adjacent values" in memory, and that initialising this array can be done like this:

```
float twoD[4][2] = { {1,2}, {3,4}, {5,6}, {7,8} };
```

1 Challenging Questions (1)

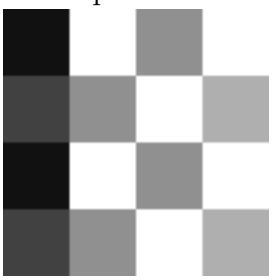
Exercise 1: Write a (grayscale) image

While most image formats, like JPEG, GIF or PNG, are fairly sophisticated formats that we don't know enough C to write yet, there are simple image formats that you can produce in a program with just text output. The PBM, PGM and PPM image formats are text-based representations, describing the colour of each pixel in an image by a number representing the brightness of the point. (The difference between the formats is that PBM only allows black or white pixels, PGM allows shades of grey, and PPM allows you to specify full colour.)

A PGM (grayscale image) file looks, for example, like

```
1 P2 4 4 15
2 0 15 8 15
3 3 8 15 10
4 0 15 8 15
5 3 8 15 10
```

which produces the resulting image (blown up to see the pixels)



where the first line consists of a marker (P2) indicating that this is a PGM file, a number representing the width of the image in pixels, a number representing the height of the image in pixels, and a number giving the maximum value that any pixel will have (which will be drawn as white). The remaining lines are the actual lines of the image, from top to bottom, where each number is a pixel (from left to right on each line).

This exercise is to write a piece of C code which creates a 2d array of integers, each of which represents a pixel in our image. You may choose any image size you like, as long as neither the height nor width are less than 5. Appropriately set the value of elements of the array in order to "draw" a pattern of your choice - in this image, 255 should be white (and 0 will be black). *As the maximum value stored*

will be 255, you could use a "smaller" variable type than **`int`** *as your base type, to avoid wasting space. Remember,* **`char`** *values are exactly a byte in size. Also, remember that you will never store a negative value, so you should appropriately specify this in the variable type.*

Then, using an appropriate, nested, pair of `for` loops, print each line from your code, remembering to output the important first line above before anything else! (Hint: as the first line of the PGM is just a descriptive header, it does not need to be stored in the array itself. In fact, it shouldn't be, as the array just stores the image itself.)

(You can use the bash redirection operator to put the resulting output into a file, e.g. ./mycode > myimage.pgm

You can display the resulting image using: display myimage.pgm)

Strings

As discussed in the lectures, strings in C are represented as an array of characters. Since dealing with text can be tricky and strings are used frequently, C provides some additional features to make the string operations more convenient than most arrays. Firstly, C provides the concept of a *string literal* which is a sequence of characters, enclosed in double-quotes ("). Note: characters are enclosed in single quotes (’). Because of this there are two ways of initialising a string:

```
char string[] = "String.";
```

or

```
char string[] = {'S', 't', 'r', 'i', 'n', 'g', '.', '\0
    '};
```

Note that the end of the string is marked with an additional, special, character, the `null` character (`'\0'`). In the first declaration, the `null` character is automatically appended, but in the second case it needs to be added explicitly. Therefore, to hold a string of n characters, the array needs to be n+1 characters long to hold the extra `'\0'` which marks the end.

Finally (as with other arrays) strings can't be compared, or manipulated like variables. C provides a set of useful functions that can be used to carry out operations such as string comparison. It's important to use the standard string libraries as working with strings is tricky and can lead to errors in our code.

- `strcmp(str1, str2)` function compares two strings for equality. If the strings are equal, function returns `0`.
- `strlen(str)` function returns the length of `str` (a number of characters before a `'\0'` character is encountered).
- `strcpy(str1, str2)` function copies the contents of the `str2` into the `str1`. You have to make sure that `str1` is large enough to store contents of the `str2`, otherwise, you may encounter unexpected behaviour. The C language does not include checks on if the array is large enough to store a particular string (this means that code can run faster, but has to rely on the programmer doing the right thing).

These functions are declared in the file "string.h", so you need to include the following line at the beginning of your C code if you use any of these functions:

```
#include <string.h>
```

There are many more string functions in "string.h". All of them, including the ones mentioned above, rely on the `null` ('\0') character.

Challenging Questions (2)

Exercise 2: Strings

This is a coding question.

Write a program which reads in a string, of up to 100 characters, after prompting the user to do so.

The program should tell the user how long the string they entered was.

The program should then lowercase the string (replacing any uppercase characters with their lowercase equivalents). You can use the two functions `isupper`, which returns true if the `char` it is given is uppercase, and `tolower`, which returns the lowercase equivalent to the `char` it is given. These need `ctype.h` to be included.

If the new string is the same as the old string, the code should tell the user that their string was all lowercase.

Otherwise, it should print the new, all lowercase, string.

As always, add appropriate comments to your code.

Structs

While arrays let us group multiple values of the same type together, it is often the case that there are natural groupings of variables of *different* types into a single record (e.g. phone book, address book, particle physics event, etc.). `structs` (short for structured types) are collections of one or more variables, possibly of different types, grouped together under a single name. The general form of a `struct` definition is:

```
struct structure_name {
  field-type field-name;
  field-type field-name;
  ...
} variable_name;
```

The above definition defines single variable (`variable_name`) as part of the initial definition of a struct type called `structure_name`. Naming your struct type is not necessary, but if you provide it with a name, you can define new variables of that type later on. To do so (assuming the above code also exists), you could write:

```
struct structure_name new_variable;
```

It is also possible to declare an array of structures (and you can have arrays of variables as part of structure definitions):

```
struct structure_name p_array[10];
```

You can initialise a `struct` type variable using the {} braces similarly to arrays (elements of the struct are initialised in the order they are declared in the definition):

```
struct year_book {
  char name[100];
  int age;
  int year;
} student1 = {"John", 23, 2014};
//in case of a new variable
struct year_book student2 = {"Anna", 20, 2014};
```

As with arrays, structs are only really useful if we can address their contained elements. For structs, this is done using the member-access operator, '.' . The syntax is `variable_name.field-name`. For example, to write a value to the "age" element in a year_book struct type variable:

```
student1.age = 22;
```

So, you can also initialise a struct type variable by accessing its members one by one.

C also allows the programmer to define their own variable type names through the `typedef` command. The general form of it is:

```
typedef type-declaration;
```

In other words, we can specify an alias for an existing type (including the struct type). For example,

```
typedef struct year_book new_type;
```

In this example the new type is called `new_type`. From now on, you can declare new struct variables in the following way:

```
new_type student3 = {"Jane", 21, 2014 };
```

This is quite often done in code to simplify the creation of structured data types.

Challenging Questions (3)

Exercise 3: Structures

This is a debugging question.

```
#include <stdio.h>

struct TwoDPoint {
  double x, y;
  };


int main(void) {
  TwoDPoint_t a, b = {0.0,0.0};

  {
    char input[100];
```

```
        puts("Please enter the x and y coordinates of the 1
    st point, separated by a comma.");

        fgets(input, sizeof(input), stdin);
        sscanf(input,"%f,%f", &(a.x), a.y);

        puts("Please enter the x and y coordinates of the 2
    nd point, separated by a comma.");

        fgets(input, sizeof(input), stdin);
        sscanf(input,"%lf,%f", &(b.x), b.y);

    }

    double x_dist = (a.x - b.x), y_dist = (a.y - b.y);

    printf("The square of the distance between the two
     points is: %f\n", (x_dist*x_dist)+(y_dist*y_dist));

    return 0;

}
```

Read the above code. There are multiple issues with its use of `structs`, as well as some other issues.

By adding one, or changing an existing, line of code, fix the first error generated when you try to compile the code. Add a comment explaining the purpose of your change.

Why is the input handling in a separate block? (Does it have to be? What do you think the intent of the author was?) Add comments explaining your reasoning.

Fix the `sscanf`s so that they interpret input in the correct way, and put it in the correct places (there are two kinds of error in the `sscanf` statements). Add comments explaining your changes.

Save a copy of the code so far in a file named "Lab3-Exe3.c"

In a new copy of the file, modify the code above to use an array of structs, rather than two separate structs called `a` and `b`. You should be able to alter the input handling to make it more "generic".

Save this new code to a file called “Lab3-Ex3A.c”