P2T: C Programming under Linux

Lab 3: Introduction to Bash scripting, C arrays and structs

Introduction

This lab will introduce shell scripts in Bash, which let you automate tasks by writing simple programs containing sequences of commands for Bash to run. You will learn to write simple scripts using control structures such as loops and conditional statements.

This lab also introduces various derived types in C, which are collections of data beyond the basic types. Array types are contiguous sequences of values of the same type which can be accessed via their index in order. Structured types ("structs") are collections of named fields of data, which can be of different types. Both kinds of derived type can be combined with each other (you can have arrays of structured data, and structs that have arrays for fields). In addition, C represents text as ordered sequences of "characters", which are conveniently stored in arrays. This lab will cover all of these aspects in a practical manner.

The assessed Linux work in this lab includes a few questions, similar to the previous labs, but you will also be required to write a number of short scripts. The assessed C work comprises two programming activities. Each question or task states the number of marks available.

Getting Started

Start by opening the **Terminal** application and verify that you are in your home directory using the **pwd** command. As with lab 1, the files required for this lab can be obtained using **p2t-get-lab**:

```
p2t-get-lab 3
```

Confirm you have the necessary files by listing the contents of this new directory: **ls linux-lab03**

Hello, World!

It's common for the first program you write in any new language to do nothing more than print "Hello, World!" to the screen and then exit. This lab will follow that tradition.

1. Change to the **linux-lab03** directory and open a new file called **helloworld.sh** in a text editor.
2. Add the following text to the file:

   ```
   #! /bin/bash
   echo "Hello, World!"
   exit 0
   ```

3. Save the file and use the **chmod** command to make it executable.
4. Run the script. You should see the message printed to the terminal.

Congratulations! You've just written and executed your first Bash script. Let's look at this process in a little

more detail.

Bash is an interpreted language. This means that the commands you write are only converted into instructions that the computer can execute when you actually run the script. This is an important difference between Bash and some other programming languages such as C. In C, there is an explicit compilation step when you run **`clang`** to convert your source code into an executable program; in Bash, the only thing you need to do to be able to run your script is to make sure it has the execute permission set. In this way, Bash works similarly to Python which you may be familiar with.

How does Linux know how to execute your script? After all, it's just a text file.

Every shell script begins with the characters **`#!`** (known as the "hashbang", "shebang" or by various other names). This must be the very first thing to appear in the script: even comments or whitespace must not appear before it. The **`#!`** is followed on the same line by the path to the program which will be used to interpret your script, which in this case is Bash itself. This is sometimes called the "interpreter directive", and the space between it and the **`#!`** is optional. Let's see what happens if we change this line.

5. Open helloworld.sh in a text edit and change the first line to the following:

   ```
   #! /bin/cat
   ```

6. Save the file and run it again.

Questions

These questions should be answered in the file **`answers.txt`** which can be found in the **`linux-lab03`** directory created at the start of this lab.

1. What must appear at the start of every Bash script? **(1)**
2. What happens when you run the script after changing the interpreter directive to **`/bin/cat`**? Why do you think this happens? **(2)**
3. What is wrong with the structure of the following script?

   ```
   # Task 0: Script to print welcome message
   #! /bin/bash
   echo "Welcome to P2T"
   exit 0
   ```

   (1)

(4)

The remainder of this lab will introduce some of the features of Bash, and ask you to write a series of short scripts using these features. You will find a brief *Bash Scripting Guide* provided alongside this lab script on Moodle. You may wish to read the following sections of that guide before attempting the next task:

1. Simple scripts
2. Layout and comments

Scripting Task

1. Write a script called **`Task1.sh`** which makes the following directories within the current directory:

   `./data`
   `./data/processed`
   `./docs` **(4)**

Conditionals

Conditional statements allow you to control the flow of a script by running certain commands if a particular condition is met. We are going to look at Bash's **`if`** statement, which is used for the same purposes as its equivalent in C, although the syntax is quite different.

```
1    #! /bin/bash
2    # Example of a conditional statement
3
4    if [[ -f "processed" ]]; then
5        echo "There is a file called processed"
6    fi
7
8    exit 0
```

The conditional statement opens on line 4 with the **`if`** keyword, and is closed on line 6 with **`fi`** keyword. The code within (line 5) is only executed if the statement on line 4 is true; in other words, the message will only be printed out if a file called **`processed`** exists in the directory in which the script is running.

The conditional statement is included within doubled square brackets: **`[[ ]]`**. **`-f`** is a test which returns true if the argument which follows it is a regular file. There are many different tests which you can use, and a list of the most useful ones is included in the *Bash Scripting Guide*.

Note: 0, 1 or 2 square brackets?

Bash has been around since 1989, and in that time its syntax has evolved. There are actually several ways of writing the test in the above example:

`if [[ -f "processed" ]]; then`

`if [ -f "processed" ]; then`

`if test -f "processed"; then`

The first form, using double brackets, is the "new" syntax, first introduced in 1998. It is slightly less portable than the other forms, although it will still work on all but the most outdated versions of Bash, and is supported by several other popular shells as well.

The single-bracket form and the version using the **`test`** command are older and offer the greatest portability. However, they are more rigid in terms of syntax: it is much more likely that a small typo will break the test. Many examples you find online will use these forms of the syntax.

If you desperately need to support computers dating from the dark ages, then you should use either the **`[`** or **`test`** forms. In all other cases, we recommend using **`[ [`** as this form is both more flexible and more forgiving, but you will not lose marks for using a different form as long as your script works!

The **`linux-lab03/examples/conditionals.sh`** file contains some more examples of the use of conditional statements in Bash.

The following sections of the *Bash Scripting Guide* will help with this task:

3. Branching: conditional statements
4. Tests

Scripting Task

2. Make a copy of your script from the previous question called **`Task2.sh`**, and modify it so that it first checks to see if any of the directories exists: if any directory exists, the script should exit with an error code and message; otherwise, the script should create the directories as before. **(5)**

Loops

The ability to use loops makes Bash scripting the perfect tool for many repetitive tasks. There are two main types of loop in Bash, **`for`** loops and **`while`** loops, and we will use the first of these in this lab.

`for` loops

A **`for`** loop is used when you want to do something a certain number of times, or when you want to loop over every file in a particular set. A Bash **`for`** loop looks like this:

```
for value in "One" "Two" "Three"
do
    echo $value
done
```

The **`for.sh`** script in the **`linux-lab03/examples`** directory contains more examples of the use of loops in Bash.

The following sections of the *Bash Scripting Guide* will help with these tasks:

5. **`for`** loops

Scripting Task

3. Write a script called **`Task3.sh`** which uses an appropriate loop to write 100 random numbers to a file called **`random.txt`**.

   You can generate a random number by echoing the value of the special **`$RANDOM`** variable, and can write the values out using redirection. **(5)**

Variables and Command Substitution

You can use variables in your Bash scripts in much the same way as you can use them on the command line:

```
var="Computer"
echo $var
```

One way to control the behaviour of your script is to pass it arguments—also known as parameters—on the command line, like this:

```
./myscript foo bar 1 2
```

Here, **`foo`**, **`bar`**, **`1`** and **`2`** are the script's arguments. To get at these while the script is running, you can use certain special environment variables:

```
#! /bin/bash

echo "The first parameter is $1, and $# parameters were provided in total"

# Loop over all arguments
for arg in $@; do
    echo $arg
done

exit 0
```

The **`variables.sh`** file in the **`linux-lab03/examples`** directory contains further examples of the use of variables in Bash.

The following sections of the *Bash Scripting Guide* will help with these tasks:

6. Variables

Scripting Tasks

4. Write a script called **`Task4.sh`** which is passed a number of names as command-line arguments. For each argument, it should print out whether the name is a regular file, a directory, or whether it does not exist.

   Your script should include a default message for arguments which don't fit into any of the above categories. You should make an effort to lay the script out neatly (try to copy the examples if you are unsure) and include appropriate comments, for example a brief description of what the script does. **(8)**

5. Write a script called **`Task5.sh`** which checks whether a **`.bak`** (back-up) file exists for each **`.data`** file in the directory. In other words, for every **`xxx.data`** file, there should be a copy called **`xxx.data.bak`**. If the corresponding back-up file does not exist, your script should display a suitable message and then create the back-up file by making a copy of the data file.

   You should run your script in the **`linux-lab03/data`** directory to check it works.

   You should lay the script out neatly and include appropriate comments. **(9)**

(17)

Lab continues on next page.

C Strings and Unicode

Note: ASCII, Unicode and C

When C was developed in the 1970s, there were several different competing ways to represent text. In general, storage space was small enough that no-one wanted to use more than 8 bits or so to store each "character", which meant that any given method could only represent as many as 255 different symbols (each with a unique number). Because the same system also needed to be able to contain values to represent things like "make a new line" or "confirm that the printer on the other end received all the text", most character sets represented much fewer than 255 different printable symbols ("glyphs").

In the USA, the dominant encoding was ASCII – the American Standard Code for Information Interchange – which could represent the Latin alphabet (excluding accented letters, or letters needed to write languages other than English), the Arabic numbers 0 to 9, and some useful punctuation, in 8 bits (and 128 total values).

Due to the political and cultural dominance of American interests in computing, ASCII quickly became a "default" encoding for text, despite not representing the languages spoken by the majority of people in the world. Whilst many countries developed their own different encodings for scripts their languages needed (like Big5 for Chinese), a large number developed encodings which were "compatible" with ASCII. (That is, they used 8 bits, and kept the values 0 to 127 the same as ASCII, storing their "new" symbols with the values 128 to 255.) For example: ISCII, developed in India for Hindi and other languages, is ASCII + numbers representing the Devanagari characters.

C, mostly, assumes that you are using ASCII when writing code, and "default" strings – indicated with just `"some text here"` – are usually assumed to contain ASCII values; the "string functions" in **`string.h`** are designed to work on assumptions that are made about ASCII.

The problem with having different "encodings" or "character sets" to represent different writing systems is that… you need to know what encoding was used for a text file before you can read it. Opening an ISCII-encoded file in a text editor which expects IBM Code Page 437 results in unreadable junk.

The problem is even worse if you want to use multiple scripts in a single file!

Unicode

Since 1991, an effort has been made to solve this problem, with the development of a new system called Unicode. Unicode provides more than 1 million potential values to represent symbols – although currently it only uses around 144 thousand of them – and thus can represent all writing scripts (and other printable symbols, like punctuation, mathematical symbols, musical notation, and emoji) in one coherent system. Whilst Unicode isn't perfect – there are still lingering issues with the representation of Chinese, Japanese and Korean scripts, and there are security implications about the fact that there are several similar looking characters (e.g., "A" [capital Latin A] and "A" [capital Greek alpha]) that can be confused in file names – it is still the best thing we have for solving the problems above.

At present ~98% of web pages are encoded using one of the Unicode encodings, UTF-8, which uses sequences of between 1 and 4 bytes to represent each Unicode symbol. UTF-8 is, by design, compatible with ASCII – the first 128 values are represented by a single byte, and agree exactly with what ASCII uses those values for. So, any given ASCII text file is also a valid UTF-8 text file… and any UTF-8 text file which only uses the first 128 Unicode character is also a valid ASCII text file.

The C11 standard adds some features to help support Unicode in C (which will also improve in the C23 standard when it is finalised). In particular, you can write strings like:

`u8"This is Ελληνικά (Greek)"`

and the C compiler will understand that the contents of the string is UTF-8 encoded, not just ASCII or something else. As your text editor uses UTF-8 by default, this usually means that you can just type whatever characters you want into u8 strings, and everything should just "work" – even most of the C string functions will be happy with u8 strings, as they still end with **'\0'**.

Activity 1: Strings and char arrays and a little Unicode

The code below can also be found in ~/examples/p2t/lab03/ex2-orig.c and in the **`clab`** directory within **`linux-lab03`**

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

int main() {

      unsigned char string1[] = u8"This is an implicitly sized
array!";
      char string2[17] = "Explicitly sized";
      char string3[24] = {0}; //zero initialised
      char string4[10] = {0};

      //add more code here

      //this prints out the array string1 char by char
      //[giving the ASCII symbol and the literal numeric value]
      for(int i=0; i<sizeof(string1); i++) printf("String1 element
%d: '%c' (%hhu)\n", i, string1[i], string1[i]);

}
```

4a) Copy the file above to a new file in your own directory space.

Add a line of code that copies **`string2`** into **`string3`**, using a suitable string function.

Also add a line of code that reads a line of text from the terminal into **`string4`**. **(2)**

b) Write lines of code that:

1. Print out each of the strings to the terminal. **(3)**

2. Print out the size of the arrays containing each string, and also the length of the string stored (using a string function for the latter).

   Note: clang may generate some warnings about your code (complaining about `"warning: passing 'unsigned char[24]' to parameter of type 'char *' converts between pointers to integer types where one is of the unique plain 'char' type and the other is not [-Wpointer-sign]"` and similar). This is because, *technically*, "`char`" (the type for characters) and "`unsigned char`" (the explicitly unsigned version of that) are not the same thing.

   However, *in this case*, this is absolutely safe to do, and no bad things will happen as a result of this mixup. In general, you should declare (non-Unicode) strings as "`char`" though, not "`unsigned char`".

c) Compile and run your code. What do you notice about the differences between the two values you print out? Add a comment about this to the source code. **(1)**

5a) Copy the code already given to print out **`string1`** char by char so you also print out the other char arrays element by element as well. **(1)**

b) Compile and run your code. What do you notice about the result of printing out **`string4`** char by char? Why is this the case, compared to the other strings? Add a comment about this to the source code. **(1)**

6a) Change the line that copies **`string2`** into **`string3`**, so that instead it copies **`string1`** into **`string3`**.

   Note: clang may generate some warnings about your code (complaining about `"warning: passing 'unsigned char[24]' to parameter of type 'char *' converts between pointers to integer types where one is of the unique plain 'char' type and the other is not [-Wpointer-sign]"` and similar). This is because, *technically*, "`char`" (the type for characters) and "`unsigned char`" (the explicitly unsigned version of that) are not the same thing.

   However, *in this case*, this is absolutely safe to do, and no bad things will happen as a result of this mixup. In general, you should declare (non-Unicode) strings as "`char`" though, not "`unsigned char`". **(1)**

b) Compile and run your code. What has happened? Why? Add comments about this to the source code. **(1)**

7a) Try replacing some of the text assigned to **`string1`** with characters outside the ASCII range. (If you don't know how to type these, you can use "Character Map" to select and copy them on most operating systems.) **(1)**

b) Compile and run the altered code: what do you notice about the length of **`string1`** according to the string function you used now? What does the loop that prints out chars one by one do for **`string1`** now? Why? Add a comment on this. **(1)**

8. Your code should compile without errors or warnings, even with **`-Wall`** set.

(ignore the warnings you can't avoid above!) **(1)**

Your submission for this activity is one C source code file, with comments for answers where needed.

(13)

Stdint.h

Note: Extra Int types

When C was developed in the 1970s, there was less interest in giving programmers precise control of the "size" of data types they could work with, and more in making sure that, whatever system they compiled their code on, it would "work".

In the modern era, and especially with the need to have reliable data types that can be interchanged between computers, we are more interested in being able to say not just "I want to store an integer", but "I want to store an integer, using exactly this many bits in memory".

Since C99, the C language has supported this via additional integer types. In order to use these, you need to

```
#include <stdint.h>
```

This then makes available a selection of new integer types which have explicit sizes in memory.

`intX_t` – where X is a multiple of 8, is an integer type using exactly X bits (and thus, usually X/8 bytes) to store signed integer values

`uintX_t` – is the same thing, but storing unsigned values rather than signed values.

[There are also other types – such as **`int_fastX_t`**, the "most efficient" integer type that has at least X bits in it – but we won't worry about these as much here.]

For example,

```
uint32_t a = 20;
```

declares **a** as a variable using 32 bits to store unsigned integer values, and stores the value 20 in it.

Using the "explicitly sized" integer types is not necessary for this course, although it may make some questions easier.

On the x86_64 system you are compiling on, "`int`" is the same as "`int32_t`", and "`long int`" and "`short int`" the same as "`int64_t`" and "`int16_t`" respectively. The only way to get an equivalent of `uint8_t` is to use `unsigned chars`. (These statements may not be true if you try to do stuff on an Apple Silicon-based Mac for example.)

Activity 2: building on Lab 2 Activity 2 with arrays and structs

In the previous lab, you wrote a program to explore the convergence of the Newton-Raphson-Seki method to the roots of

$$z^3 - 1 = 0$$

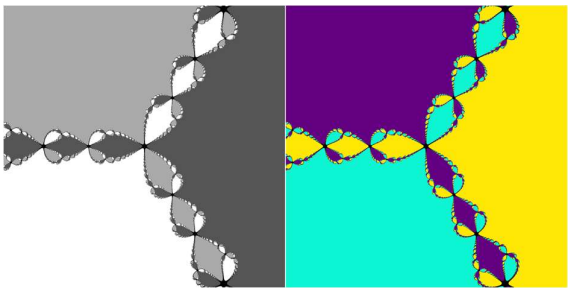
depending on the starting value.

In this activity, we will use our new knowledge of arrays to extend the code you wrote then – storing the values in a two-dimensional array for later processing, and then printing out the array in a later loop.

We will also output the resulting image as a colour image, which can be viewed using the command **`display`** on your terminal.

The "Portable PixMap" format (PPM for short) is a text-based format for describing an image. It's essentially an extension of the PGM you produced in the last lab, except that every pixel now has three values associated with it – representing the amount of Red, Green and Blue in the colour it has. PPM files begin with "P3" rather than "P2", and tend to have **`.ppm`** as a file extension.

That is, whilst our code in Lab 2 gave us an image like the one on the left – we will end up producing a colourised version like the one on the right (with whatever 4 colours we choose as a palette)



Start with a copy of your code from Lab 2 Activity 2 (copy it into the **`clab`** directory for this lab)

9. Storing values in an array.

a) Near the top of your **`main`** function, declare a two-dimensional array to store the results of your loop. The dimensions of the array should be exactly enough that there is one row and one column for every number.

You may choose any suitable data type as the "value type" for this array.

> **Note**: if you made your image very large (bigger than 1000x1000 or so), you will need to make it *smaller* than this for this step to work. (There are limits on the size of an array you can make in the way we've taught you.)

(3)

b) Modify your loop so that, instead of printing out the text representation of values from 0 to 3 to the screen, it instead stores the value (as an integer) in the appropriate element of the array you just declared. **[That is, the "top left" value in the image – the first one we "print",**

(1)

should be in the [0][0] element of the array, for example]. You will need to map your loop variables to appropriate indices.

10. We can represent any colour (or at least, any colour displayable by a computer monitor) by three integers, each between 0 and 255 (where 255 is "full intensity") – one for red, one for green and one for blue. The resulting colour is that produced by mixing "pure" red, green, and blue in the relevant intensities – so pure white is (255 red, 255 green, 255 blue); pure, high-intensity red is (255 red, 0 green, 0 blue); a pale unsaturated yellow is (255 red, 255 green, 150 blue).

a) Declare (above the main function) a suitable structured type for storing colour values, with fields called **r**, **g** and **b** (for **r**ed, **g**reen and **b**lue components of the colour)

The data type for the fields should be the *smallest* integer type capable of representing numbers between 0 and 255. [There are two choices here, depending on if you use the information in the note in blue above or not. `int` is *not* a valid choice – it's not the *smallest* such type.] **(2)**

b) Inside the main function, declare an array of 4 of these new "colour types" you just declared, and assign colour values to its elements directly – these will be used to pick the "colour" for each point in the image, based on the value in the image array there.

Colour 0 should be set to black; you may pick any colours for the other three, as long as they are all different. **(2)**

11. Printing out appropriate colours.

a) Write a second double loop after the first loop in your main function (you can copy parts of the first loop if it helps) to loop over the elements of your image array in order. Remember, array indices start at 0. **(2)**

b) For every element of the array, use its value as an index into the colour array, and print out the resulting colour [as three values, separated by spaces]. (That is, if the value in the image array element is "0", you should use element 0 of your colour array to get the r,g,b values) **(1)**

c) Change the "header printf" in your main function to print the following header:

`P3 <width of image> <height of image> 255`

which tells anything reading the data that this is now a "P3" type image, and that values will go from 0 to 255.

Compile and run your code, this time redirecting output to a file called **`fractal.ppm`**.

Use **`display`** to view the resulting file and confirm that it has the colours you expected.

12. Your code should be appropriately commented and show good layout. **(4)**

13. Your code should compile without errors or warnings, even with **`-Wall`** set. **(1)**

Your submission for this activity is a single C source code file.

(16)

Submitting Your Work

Your submission should include:

- your completed **`answers.txt`**
- your five **`TaskN.sh`** script files
- a subdirectory containing your submission for C Activity 1
- a subdirectory containing your submission for C Activity 2

As with labs 1 and 2 , you should submit this via

```
p2t-submit-lab 3
```

and submit the short code token you are given as your proof of submission to Moodle. As with Lab 1 and 2, you can use p2t-submit-lab as often as you want, but you should make sure that you submit the correct token for the submission you want to be marked.

Summary

Following this lab, you should know:

- How to write a simple Bash script.
- The meaning of the **`#!`** line at the start of a script.
- How to use **`if`** statements to control the flow of execution in your script.
- How to use **`for`** loops to perform repetitive actions.
- How to exit a script with an error code.
- How to take and use information in the form of command-line parameters.
- How to use structs and arrays in C to hold multiple values in a single named variable.
- How to use loops and arrays to usefully repeat a task across multiple values.
- How C strings are stored and manipulated, and the difference between them and arrays of **`char`**s.