C Programming

These accompanying programming exercises and challenges follow broadly a 'top down' approach. In this sense, the student is given a program to study and see working – then they are asked to change it in some way to produce different behaviour and results – or to add to it. Hopefully, we then remove the challenges of getting working code in the first place, and just concentrate on the syntax and elements of the program.

Practical Exercises

Contents

Session 1: Introduction to C

Problem 1

Write, compile and run the "`Hello Lincoln!`" program which you saw in the slides.

Use your editor/compiler of choice – however, the slides provide instruction on using VS Code/gcc.

Make sure it compiles and runs without warnings or errors.

This is a way to ensure that your build and compile toolchain is working properly.

Problem 2

The following lines of code have been jumbled up. Can you put them in the proper order in your editor and make them run?

```
int main(int argc, char **argv)
#include <stdio.h>
return 0;
}
printf(txt);
#define TEXT "I am getting on with C programming!"
{
const char txt[] = TEXT;
```

Problem 3

This snippet of code has caused errors and will not compile.

Identify what is wrong and fix it in your editor – checking that it is ok by compiling and running.

```
int items = 60;
int cost_per_item = 19.95;
int total_cost = items * cost_per_item;

printf("Number of items: %d\n", items);
printf("Cost per item: %.2f\n", cost_per_item);
printf("Total cost = %.2c\n", total_cost);
```

Challenge 1

The code in session1.c uses a preprocessor define for an 'add' function (we will look more closely at more 'usual' functions later on).

Look at this code and see it working. Try changing the parameters in 'add(.. , ..)' on line 10.

```c
#include <stdio.h>
#include <stdlib.h>

//Use a preprocessor define for a 'function'
#define add(x, y) (x+y)


int main(int argc, char **argv)
{
    printf("%d", add(4, 5));
    return 0;
}
```

Your challenge is to add some more preprocessor defines which subtract, multiply and divide two numbers.

Use '`printf`' to output the result.

Be careful with the divide, as you may have to refactor some other bits of the code as well.

Session 2: Variables and Datatypes in C

Problem 1

session2.c is a program which detects whether an inputted character is either:

- A lower case letter
- An upper case letter
- A number
- Whitespace

Ascii values are used to determine which category they fall in. The ascii reference chart is here: https://en.cppreference.com/w/c/language/ascii c represents chars using their ascii value.

Your task is to take this code:

- Check it works
- Modify the '`printf`' statements so that they output the actual character as well as its ascii value.
- Modify the code further so that it 'guards' the `if` selection statements and filters out characters that are not detected. Make the program more robust and resistant to runtime errors.

Problem 2

An int value is 5.

a) Use a left bitwise shift to double the number. Use printf to check.
b) Use bitwise shifts to:
    a. Multiply the number by 8
    b. Multiply the number by 9

Challenge: Can you implement some code to multiply any value by any value – using only bitwise shifts?

Problem 3

The Problem 3 code shows the algorithm to convert a decimal number to a binary number. Some basic code is set up – complete the while loop to convert the value.

In addition complete the code to output the contents of the array in the correct order.

Values to test:

```
Input: 18    Output: 10010
```

```
Input: 21    Output: 10101
```

Challenge 2

We saw in Problem 3 that you can convert from a decimal to a binary value. Add some code now which will convert from a binary value to a decimal.

I'll leave you to determine the algorithm for this, however:

- The binary value (eg; 1010) is stored as an int
- You should try to extract the last digit of the binary value by using the mod (%) operator
- Remove the last digit of the binary value by dividing by 10.

Values to test:

```
Input:  11001     Output: 25

Input:  11110    Output: 30
```

Session 3: Program Flow and Loops in C

Problem 1

In session3.c 'Problem 1', you need to fill an array with integers that are inputted by the user.

Use a `for` loop to loop through the slots of the array and insert the value which is entered.

Hint: Use `scanf(..)` to get the input from the user and put the input into the array.

Problem 2

Now, with our filled array, you need to calculate the average of all the elements.

So, we need to know:

- the number of elements we are adding (the size of the array)
- the sum of all the elements in the array.

We know the size of the array, from the #define label 'arraySize'.

Loop though the array to get the total of all the elements, by adding each one to a running total.

Calculate the average by calculating (`total / arraySize`).

Problem 3

We also need to calculate the maximum and minimum values – imagine the array is a collection of daily temperature values, and we want to determine the temperature range.

In session3.c Problem 3, three integer variables are declared. Loop through the array to determine the maximum and minimum values. (Could both of these things be done with just one loop?)

From these values, determine the range.

Challenge 3

The 'bubble sort' algorithm is a quick and easy way to sort the contents of a small array. There are many more efficient sort algorithms, but this is the easiest to implement.

Your challenge is to do two things:

1. Output the contents of the array as they are (i.e. unsorted)
2. Apply the bubble sort algorithm to the array and output the resulting sorted array. We will sort from the largest to the smallest.

The bubble sort algorithm is:

- Take the first element of the array and compare it with the next.
- If the first element is smaller than the second, then swap.
- Repeat this process until the end of the array is reached.
- Repeat the process until no values are swapped in a pass

Session 4: Functions, Input/Output in C

Problem 1

In this problem, you will use '`scanf`' to gather user input and use a function to encapsulate the functionality required.

The `<string.h>` header file is used to give us string functionality. '%s' is the format specifier for a string variable.

In session4.c there is a function which asks the user to enter a name and an age. There are two variables declared, 'name' and 'age' to hold the inputs. The name array has '+1' because C appends the EOF ('end of file') character to the end of a string.

Complete the function '`processInput()`' so that it takes input and puts that input into 'name' and 'age'. Call this function in '`main`', and then write out the values of 'name' and 'age'.

Problem 2

Modify your code solution, so that 'guards' are used to protect the variables 'name' and 'age'. Ensure that:

- The name entered cannot exceed 20 characters
- The age variable cannot be lower than 0 or greater than 100.
- Neither should be 'NULL'

If they are, prompt the input again – with a suitable message.

Return a 'bool' from the function so that it returns 'TRUE' when it is complete.

Problem 3

Modify your code still further, so that you call `processInput()` repeatedly in a loop until 'FALSE' is returned, indicating the user wants to stop entering data.

Challenge 4

We can now enter multiple names and ages. Use an array to hold all of the names and a separate array to hold all of the ages.

HINT: char names[10][STR_MAX_LEN+1]; is an array of 10 character strings up to STR_MAX_LEN in length.

Call processInput() multiple times, and then output the contents of the names and ages arrays in main.

Session 5: Pointers and Strings in C

Problem 1

In this problem, you are making use of pointers and array in C. We can create an array as a stack memory implementation by, for example, `int myArray[5];`, or create a collection that we can use as a dynamic array – i.e. the size of the array is decided while the program is running, not at compile time.

We can use:

malloc(size) [Memory allocate at runtime, not initialised]

calloc(number of elements, size of one element) ['contiguous memory allocation', all elements initialised to 0]

realloc(oldPtr, newSize) [Resize previously allocated memory, returns a new ptr]

free(ptr) [release dynamically allocated memory above back to the OS]

Take a minute, and read https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/ for a full explanation.

In session5.c, you are going to get a size from the user, the program then allocates that memory to hold that number of integers, the memory is filled (like filling an array), then resized dynamically:

- Get a size input from the user.
- Dynamically allocate the memory for that amount of integers.
- Fill that memory.
- Resize the memory to hold 5 more integers.
- Refill that memory.
- Make sure to use free()

Problem 2

Modify your program so that you can see the difference on the allocated memory between malloc, calloc and realloc.

Hint: Fill the allocated memory with less than the size number of integers, and output the whole memory which was allocated.

Challenge 5

`calloc` creates memory in a slightly different way to `malloc`. Create a function called `myCalloc` which uses only malloc and returns a pointer to the newly created memory.

Session 6: `structs` in C

Problem 1

The 'Point' struct code is shown. It contains two float variables – it models a simple 2D point.

Create a Point variable, allocate values to the two floats, and output them.

Problem 2

Create a new 'Triangle' struct which contains three Point structs.

Put the structs in a header file called 'Shapes.h' – make sure this is included in the main c file.

Create a utility function which calculates the area of the triangle.

Challenge 6

Either:

If you are interested in games programming, then modify the Point struct to become 3D – use 4 components per point (x, y, z, w). Create another utility function which calculates the distance between the two points.

Or:

Create a struct called 'Node', with two members – an int (the data) and an int pointer. A 'linked list' links these nodes together in that the pointer in one points to the next node.

Review this: https://www.geeksforgeeks.org/linked-list-in-c/ then demonstrate the creation of a singly liked list.

Show that it has been created by 'traversing' the linked list to output the data at each node.