Lydin Camilleri | Bachelor of Science in Computing Science | CPS1000

Programming Principles in C

Assignment Report

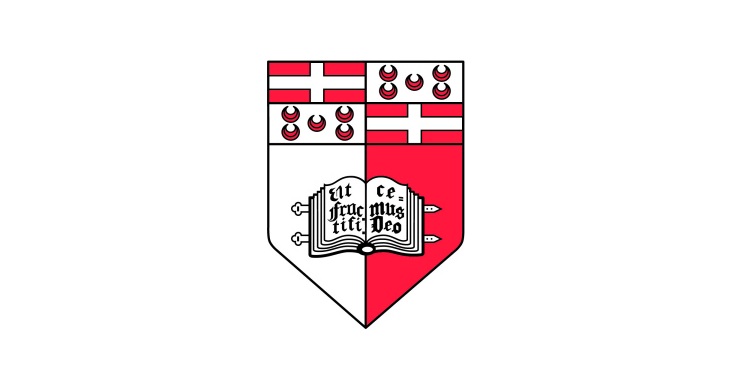


Table of Contents

[Introduction 5](#_Toc442419274)

[Assignment Question 1 6](#_Toc442419275)

[Part 1A 6](#_Toc442419276)

[Part 1B 9](#_Toc442419277)

[Testing 11](#_Toc442419278)

[Part 1C 12](#_Toc442419279)

[Part 1D 14](#_Toc442419280)

[Part 1E 16](#_Toc442419281)

[Generating Random Numbers 16](#_Toc442419282)

[The Insertion Sort Method 18](#_Toc442419283)

[The Naïve Sort Function 19](#_Toc442419284)

[The Main Method 20](#_Toc442419285)

[Testing 21](#_Toc442419286)

[Full-code 22](#_Toc442419287)

[Part 1F 24](#_Toc442419288)

[Generating Random Numbers 24](#_Toc442419289)

[The Quick Sort Method 24](#_Toc442419290)

[The Smarter Sort Function 25](#_Toc442419291)

[The Main Method 26](#_Toc442419292)

[Testing 27](#_Toc442419293)

[Full-code 28](#_Toc442419294)

[Part1G 30](#_Toc442419295)

[Testing 32](#_Toc442419296)

[Part 1H 33](#_Toc442419297)

[Testing 35](#_Toc442419298)

[Conclusion 36](#_Toc442419299)

[Part 1I 37](#_Toc442419300)

[Testing 38](#_Toc442419301)

[Part 1J 39](#_Toc442419302)

[Compute Change Function 39](#_Toc442419303)

[Testing 43](#_Toc442419304)

[Assignment Question 2 44](#_Toc442419305)

[Part 2A 44](#_Toc442419306)

[Testing 47](#_Toc442419307)

[Part 2B 48](#_Toc442419308)

[The Main Method 49](#_Toc442419309)

[Testing 51](#_Toc442419310)

[Part 2C 52](#_Toc442419311)

[The postfix\_evaluation() function 52](#_Toc442419312)

[The Main Method 53](#_Toc442419313)

[Testing 54](#_Toc442419314)

[Part 2C Full code 55](#_Toc442419315)

[Assignment Question 3 58](#_Toc442419316)

[Assignment Question 4 59](#_Toc442419317)

[Part 4 59](#_Toc442419318)

[The Header File (.h) 59](#_Toc442419319)

[The Source File (.c) 60](#_Toc442419320)

[The Main Method 62](#_Toc442419321)

[Testing 63](#_Toc442419322)

[Assignment Question 5 64](#_Toc442419323)

[Creating a shared library 64](#_Toc442419324)

[Linking 65](#_Toc442419325)

[Create Project 65](#_Toc442419326)

[Directory Link 66](#_Toc442419327)

[Running the project 67](#_Toc442419328)

[Part 5 Full code 68](#_Toc442419329)

[SharedLibrary 69](#_Toc442419330)

[SharedADT.c Full code 69](#_Toc442419331)

[SharedADT.h Full code 71](#_Toc442419332)

# Introduction

The general-purpose high level language was developed in by Dennis Ritchie for the Unix Operating System. It was then first implemented on a PDP-11 computer in 1972. Eventually, the C Programming Language has become widely used by various professional programmers and this is due to the facts that it is very easy to learn, produces efficient programs, provides ease for cross-platform programming and can handle low-level activities.

The C Programming Language is classified in the middle level between the high level such as Java and Python and the low level such as the assembler. The middle level language does not provide all the built-in functions that one can find in high level languages but provides enough to achieve several aims and objectives.

This language is also classified under Structure-Oriented Languages which means that large programs are divided into smaller programs called functions, which also attract all the focus of the programmer since these functions work on the data whilst the data moves freely between them.



# Assignment Question 1

## Part 1A

For the first question, a program that calculates the taxes paid for a particular week including the overtime hours worked and the ease of maintenance in mind was requested. To begin with, a rough structure of the program was sketched on a piece of paper before starting to implement code. The first part of the program defines the Tax Rates as stated in the question. These rates were converted to satisfy the purpose of being multiplied with the number of hours worked and automatically the tax deductions are reduced. The Breakpoint definitions were then defined identifying the amount of money on which the taxes apply. Also, some calculations were defined for a better code structure.

**#include** <stdio.h>

**#define** TaxRate1 1.00

**#define** TaxRate2 0.85

**#define** TaxRate3 0.71

**#define** Breakpoint1 300.00

**#define** Breakpoint2 480.00

**#define** Calculation1 (TaxRate1 \* Breakpoint1)

**#define** Calculation2 (Calculation1 + (TaxRate2 \* (Breakpoint2 - Breakpoint1)))

The program starts by requesting the number of hours worked during the week and the value entered by the user is stored in a float variable named ‘HoursWorked’. The float type was used to hold a more precise value than integer (including decimal places (for example: half an hour, 0.5)).

Two if statements were then used to calculate the total wage of the employee according to the given number of hours worked. Overtime was also calculated above 40 hours of work. The addition of the normal hours wage and the overtime wage, if any, were added and displayed as the Gross wage.

**float** BasicPayRate = 22.50;

**float** HoursWorked=0, NormalHoursWage=0;

**float** Overtime=0, OvertimeWageAddition=0, OvertimeMultiplier=1.5;

**float** GrossPay=0, NetPay=0;

**int** **main**(**void**) {

**printf**("Please enter the number of hours worked during the week = ");

**fflush**(stdout);

**scanf**("%f", &HoursWorked);

//Calculating Wage when Hours Worked is Less Than or Equal to 40 hours

**if**(HoursWorked <= 40)

NormalHoursWage = (BasicPayRate \* HoursWorked);

**else**

NormalHoursWage = (BasicPayRate \* 40);

**printf**("\nNormal hours pay = Eur %.2f \n", NormalHoursWage);

//Calculating Overtime over 40 hours

**if** (HoursWorked > 40)

Overtime = HoursWorked - 40;

**printf**("Hours worked as Overtime = %.2f hours", Overtime);

OvertimeWageAddition = Overtime \* (OvertimeMultiplier \* BasicPayRate);

**printf**("\nOver time extra pay = Eur %.2f", OvertimeWageAddition);

//Gross Pay Calculation

GrossPay = (NormalHoursWage + OvertimeWageAddition);

**printf**("\nGross pay is = Eur %.2f", (GrossPay));

The three cases that could occur were treated separately using IF ELSE conditions. Firstly, the gross pay calculated earlier was checked whether it skips the breakpoint defined in the beginning of the program (which refers to the amount of money on which the respective tax applies). Then, the net pay was calculated by multiplying the amount with the TaxRates so that the tax percentages are deducted. Whenever a breakpoint is exceeded, the calculations defined on top are used as an addition to the calculation on the remainder money in a particular tax gap. The total amount of tax deductions was calculated by subtracting the net pay from the gross and this was displayed together with the final net pay.

//Tax Calculations

**if** (GrossPay <= Breakpoint1)

NetPay = GrossPay \* TaxRate1;

**else** **if** (GrossPay <= Breakpoint2)

NetPay = Calculation1 + ((GrossPay - Breakpoint1) \* TaxRate2);

**else** **if** (GrossPay > Breakpoint2)

NetPay = Calculation2 + ((GrossPay - Breakpoint2) \* TaxRate3);

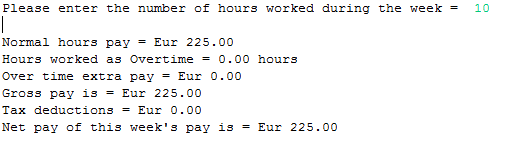
**printf**("\nTax deductions = Eur %.2f", (GrossPay - NetPay));

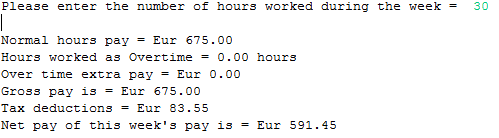
**printf**("\nNet pay of this week's pay is = Eur %.2f", NetPay);

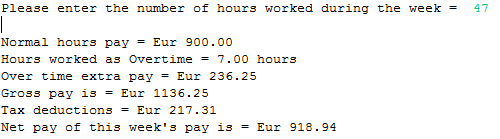
**return** 0 ;

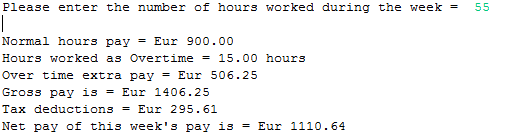
}

Different cases were tested including the use and the absence of overtime and also different tax usages.









## Part 1B

This question requested a program that reads the user input character by character until reaching end of file (EOF) and then prints the average number of letters per word and gives a ratio of uppercase letters with lowercase letters.

Variables were initialized to hold several values such as the number of white spaces, the lower and upper case counter and the number of characters that we encounter along the reading of the input. Another variable of type float was declared so that the calculation of the average can be stored into.

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <ctype.h>

**int** **main**(**void**) {

**char** ch;

**int** lowercaseCounter = 0;

**int** uppercaseCounter = 0;

**int** whiteSpaces = 0;

**int** noOfCharacters = 0;

**float** averageCharacters = 0;

The user input was then read character by character. These characters were stored in variable ch of type character. Then, using functions such as isspace(), isalpha() and islower() were used to reach the objective of the program. These are included using the **#include** <ctype.h> command. The isspace() built-in function was used to count the number of spaces between each word. For the same character isalpha() is used to make sure that it is alphabetical and not numeric or a punctuation to increase the number of characters variable. If this condition is satisfied then either the lower case or upper case counters are incremented according to the result returned by islower().

//Examples:

//The Quick Brown Fox Jumps Over The Lazy Dog!

//The quick brown fox jumps over the lazy dog!

**printf**("Please Enter Sentence Here: ");

**fflush**(stdout);

**while**((ch = **getchar**()) != EOF){

**if**(**isspace**(ch))

whiteSpaces++;

**if**(**isalpha**(ch)) {

noOfCharacters++;

**if**(**islower**(ch) != 0)

lowercaseCounter++;

**else**

uppercaseCounter++;

}}

Upon reaching the EOF, the program breaks for the while loop and the calculations are initiated. The average number of characters per word (averageCharacters) was set to be equal to the number of characters divided by the number of white spaces. Since this can be a decimal integer, the float type was used. Parsing the integer type variables to float was required. This average was then printed on screen to two decimal places.

//Parsing Values and Calculating Average Number of Characters in Each Word

averageCharacters = (**float**)noOfCharacters / ((**float**)(whiteSpaces));

**printf**("Average Number of Characters in Each Word: %.2f \n", averageCharacters);

For the ratio of upper:lower case letter, the two variables(uppercaseCounter & lowercaseCounter) were printed with a colon in between.

**printf**("Upper:Lower - %d:%d", uppercaseCounter, lowercaseCounter);

**return** 0;

}

### Testing



## Part 1C

The aim of this task was to create a function that returns a particular input decimal number to its base-n equivalent where n is a number that can be expressed as 2y such as 2,4,8,16,32… The function to\_base\_n was coded in such a way that it is passed both integer inputs from the user. An integer variable was declared to store the result of the value%toBaseNumber function which gives the remainder as a result of the value to be converted divided by the base number. Then, by recursion the same function is re-called with different parameters as can be seen in the code below. The remainders are also printed.  
The main method asks the user to input the values and then calls the function to\_base\_n which prints the result itself.

**#include** <stdio.h>

**void** **to\_base\_n**(**int** value, **int** toBaseNumber) {

**if**(value > 0) {

**int** equivalentNumber = (value%toBaseNumber);

to\_base\_n((value/toBaseNumber), toBaseNumber);

**printf**("%d", equivalentNumber );

} **else** {

**if**(value < 0)

**printf**("Number is invalid");

}

}

**int** **main**() {

**int** value;

**int** toBaseNumber;

**printf**("Enter a value to be converted: ");

**fflush**(stdout);

**scanf**("%d",&value);

**printf**("\nThis value is to be converted to its base-n equivalent \nwhere n is expressible as 2^y.\nEnter n: ");

**fflush**(stdout);

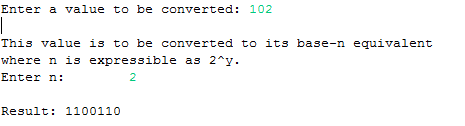
**scanf**("%d",&toBaseNumber);

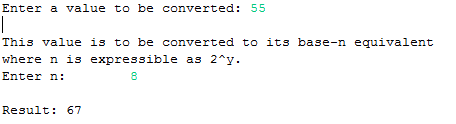
**printf**("\nResult: ");

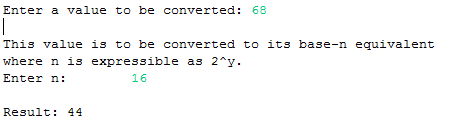
to\_base\_n(value, toBaseNumber);

**printf**("\n");

**return** 0;  
}







## Part 1D

This task requested to build a **function** that reverses any string that is passed to it as an argument. This was done using several functions that are included using the first three lines of code of our program.

**#include** <stdio.h>

**#include** <string.h>

**#include** <ctype.h>

An array of type char and size 50 was set. Fifty was chosen to be the maximum size the array holds for this program but it could have been chosen to be bigger without any problems. Some variables were initiated and the program asks the user to input the sentence to be reversed. It’s length is then calculated using strlen() function and its result is displayed.

In order to reverse the string a loop was generated for all characters according to the length found before. In this loop, an if statement checks whether the character is not alphabetical and not numerical or if it is the last character. If these conditions are obeyed, another if condition states that if the character being conditioned is not the last letter thank variable K is set to the previous character position or else (if it is the last character) K is set to that character position.

Finally, a while loop iterating from position 0 until reaching ‘pointer’ K in the array is used to flip the characters using a temporary variable in order not to lose any data. The array is then displayed as a result.

**#define** SIZE 50

**char** inputString[SIZE];

**int** length = 0;

**void** **str\_reverse**() {

//Set a temporary variable to hold character values

**char** temporary;

**int** i = 0;

**int** k = 0;

**int** j = 0;

//Reverse the string

**for** (j = 0; j < length; j++) {

**if** (!**isalnum**(inputString[j]) || (j == length - 1)) {

**if** (j < length - 1)

k = j - 1;

**else**

k = j;

}

}

**while** (i < k) {

temporary = inputString[i];

inputString[i] = inputString[k];

inputString[k] = temporary;

i++;

k--;

}

//Print the reversed string

**printf**("The reversed string is \"%s\"", inputString);

}

**int** **main**(**void**) {

**printf**("Please enter a string argument: ");

**fflush**(stdout);

**scanf**("%[^\n]", inputString);

length = **strlen**(inputString);

**printf**("The length of the string is %d \n", length);

str\_reverse(inputString);

**return** 0;

}





## Part 1E

This part of the assignment was related to sorting algorithms. In this particular task, a function, named naïve\_sort, was requested. This function had to take an array of random generated numbers and returns them sorted using the insertion sort method.

### Generating Random Numbers

The first task to start with was building a generator for random numbers lists according to the size inputted by the user.

//function declarations

**int**\* **generate\_random\_numbers**(**int**);

//Generating an array of random numbers

**int**\* **generate\_random\_numbers**(**int** numberOfRandomNumbers) {

**int**\* randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

**int** counter, randomNumber, i;

**for**(counter = 0; counter < numberOfRandomNumbers; counter++) {

randomNumber = (**rand**()% 100) + 1;

randomNumbersArray[counter] = randomNumber;

}

**printf**("Random Numbers (UNSORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

**return** randomNumbersArray;

}

Above we can see the random number generator function that takes the number of random numbers requested by the user as an argument and returns an array. The function firstly allocates enough memory space for the randomNumbersArray to be created according to the numberOfRandomNumbers requested.

**int**\* randomNumbersArray =(**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

Then, using the rand() function it generates random numbers from 1 up to 100 using the following line of code:

randomNumber = (**rand**()% 100) + 1;

and putting it in the array with location [counter] which increments until the required size of the random numbers list is achieved.

randomNumbersArray[counter] = randomNumber;

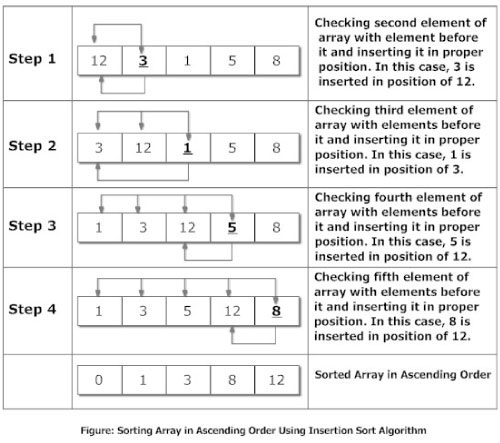
Then, the array of random numbers generated is displayed on screen using a loop.

Example for generating 10 random numbers using this function:



### The Insertion Sort Method

The insertion sort method is a method that builds the sorted list one item at a time. As researched, it is not efficient on large lists compared to other sorting methods but it is very stable and does not require a lot of memory space. This method starts of by the first unsorted element (in position 2 in a list) and inserting it accordingly on the list on the left hand side of its original position by moving the necessary elements. This is then repeated for the next elements until reaching the last element of the list.



### The Naïve Sort Function

As explained above, the naïve sort function was then built in order to sort the list of random numbers generated by the previous function. Therefore, the unsorted array from generate\_random\_numbers is passed as an argument together with the number of elements the list has. Using the same logic explained as the insertion sort method approaches, each element in the list starting from position 1 and not 0 is checked with the previous elements and if it is smaller it is moved to its correct position and moving the other elements to the right. This function then returns the sorted array.

//Insertion Sort Method

**int**\* **naive\_sort**(**int** numberOfRandomNumbers, **int**\* unsortedArray) {

**int** i, n;

**int** tempStore;

**for** (i = 1; i < numberOfRandomNumbers; i++) {

tempStore = unsortedArray[i];

n = i - 1;

**while** ((tempStore < unsortedArray[n]) && (n >= 0)) {

unsortedArray[n + 1] = unsortedArray[n];

n = n - 1;

}

unsortedArray[n + 1] = tempStore;

}

//even though it is named unsorted array, this function returns

//the sorted array

**return** unsortedArray;

}

### The Main Method

In the main method, the user is requested to enter the number of random numbers to be generated by the program and this is stored in a variable named numberOfRandomNumbers. The malloc() function was then used to allocate enough memory space for the randomNumbersArray which will be holding the list of numbers generated by the generator and eventually the sorted list. This array is then filled by calling the function generate\_random\_numbers as follows:

randomNumbersArray = generate\_random\_numbers(numberOfRandomNumbers);

This array is then passed as an argument in the naïve\_sort function that does the necessary sorting and returns the sorted list which is then printed on screen with a for loop.

naive\_sort(numberOfRandomNumbers, randomNumbersArray);

At the very end, the memory allocation for the randomNumbersArray is freed as part of the structure when using malloc().

Note: The srand(time(NULL)) function is used to eliminate the error of generating the same random numbers when the program is executed again using the rand() function.

**int** **main**() {

**int** numberOfRandomNumbers;

**int** i;

//Used to seed the rand() function to generate different random numbers

**srand**(**time**(NULL));

**printf**("Please enter the number of Random Numbers to generate: ");

**fflush**(stdout);

**scanf**("%d", &numberOfRandomNumbers);

**int**\* randomNumbersArray;

randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

randomNumbersArray = generate\_random\_numbers(numberOfRandomNumbers);

naive\_sort(numberOfRandomNumbers, randomNumbersArray);

**printf**("\n");

**printf**("Random Numbers (SORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

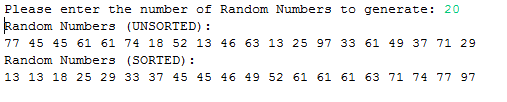
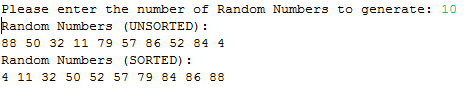
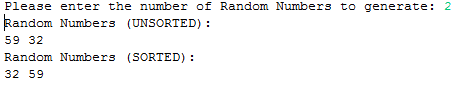
**free**(randomNumbersArray);

**return** 0;

}

### Testing

Different number of values were tested and their correct operation was manually checked. This was done for several times.



### Full-code

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <time.h>

**#include** <malloc.h>

//function declarations

**int**\* **generate\_random\_numbers**(**int**);

**int**\* **naive\_sort**(**int**, **int**\*);

//Generating an array of random numbers

**int**\* **generate\_random\_numbers**(**int** numberOfRandomNumbers) {

**int**\* randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

**int** counter, randomNumber, i;

**for**(counter = 0; counter < numberOfRandomNumbers; counter++) {

randomNumber = (**rand**()% 100) + 1;

randomNumbersArray[counter] = randomNumber;

}

**printf**("Random Numbers (UNSORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

**return** randomNumbersArray;

}

//Insertion Sort Method

**int**\* **naive\_sort**(**int** numberOfRandomNumbers, **int**\* unsortedArray) {

**int** i, n;

**int** tempStore;

**for** (i = 1; i < numberOfRandomNumbers; i++) {

tempStore = unsortedArray[i];

n = i - 1;

**while** ((tempStore < unsortedArray[n]) && (n >= 0)) {

unsortedArray[n + 1] = unsortedArray[n];

n = n - 1;

}

unsortedArray[n + 1] = tempStore;

}

//even though it is named unsorted array, this function returns

//the sorted array

**return** unsortedArray;

}

**int** **main**() {

**int** numberOfRandomNumbers;

**int** i;

//Used to seed the rand() function to generate different random numbers

**srand**(**time**(NULL));

**printf**("Please enter the number of Random Numbers to generate: ");

**fflush**(stdout);

**scanf**("%d", &numberOfRandomNumbers);

**int**\* randomNumbersArray;

randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

randomNumbersArray = generate\_random\_numbers(numberOfRandomNumbers);

naive\_sort(numberOfRandomNumbers, randomNumbersArray);

**printf**("\n");

**printf**("Random Numbers (SORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

**free**(randomNumbersArray);

**return** 0;

}

## Part 1F

This task was similar to Part 1E with the single difference being the implementation of the quick sort method in the smarter\_sort function requested rather than the insertion sort method.

### Generating Random Numbers

The same random number generator was used in this task as the one explained in Part 1E. Please refer.

### The Quick Sort Method

Also called the comparison sort, the quick sort is an efficient algorithm that can sort many elements in quite a few time. Developed by Tony Hoare in 1959, the quick sort method is unstable and does not require much additional memory to perform. The algorithm itself divides and conquers by pivoting an element and re-ordering the values to be less than it on the left and bigger on its right ; concluding that it will be in its correct place after this re-ordering is done. **Recursively**, this approach is taken on the left hand side of the pivot and on the right hand side of the pivot until the list is sorted.

### The Smarter Sort Function

The smarter\_sort function was built using the quick sort method concluding that the partition and conquer approach was taken. Therefore, two functions were actually implemented. Firstly, the smarter\_sort function, that takes an unsorted array of numbers together with the first and last element as arguments, was coded. This function is used to call the other complementary function named partition and recursively calls itself for the divided sub-lists.

In addition, the partition function is used to sort the elements of the array. Pointer like variables ‘I’ and ‘j’ are used to hold the first and last elements of the list. The pivot is always set to the first element of the list. Using do while loops, the pointers are incremented and decremented until the elements at those positions should be swapped. An extra if condition was implemented to check that the left pointer is not larger or equal to the right pointer before swapping elements. Variable ‘t’ was used as a temporary variable not to lose any data during the swap.

**void** **smarter\_sort**(**int**\* unsortedArray, **int** left, **int** right) {

**int** j = 0;

**if**(left < right) {

//Divide and conquer

j = partition(unsortedArray, left, right);

smarter\_sort(unsortedArray, left, j-1);

smarter\_sort(unsortedArray, j+1, right);

}

}

**int** **partition**(**int**\* unsortedArray, **int** left, **int** right) {

**int** pivot, i, j, t;

pivot = unsortedArray[left];

i = left;

j = right+1;

**while**(1) {

**do** i++;

**while**(unsortedArray[i] <= pivot && i <= right);

**do** j--;

**while**(unsortedArray[j] > pivot);

**if**(i >= j)

**break**;

t = unsortedArray[i];

unsortedArray[i] = unsortedArray[j];

unsortedArray[j] = t;

}

t = unsortedArray[left];

unsortedArray[left] = unsortedArray[j];

unsortedArray[j] = t;

**return** j;

}

### The Main Method

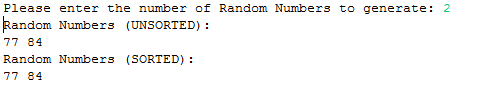
For the main program, the same as part 1E was used with the only difference that the smarter\_sort function was called instead of the naïve\_sort. The parameters passed to this function are the randomNumbersArray generated by the random number generator together with the first element position of the list which is always 0, and the last element position of this list which is always found to be the number of random numbers generated -1.

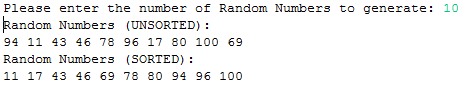
smarter\_sort(randomNumbersArray, 0, (numberOfRandomNumbers-1));

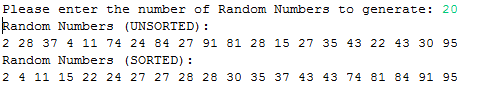
**printf**("\n");

### Testing

Different number of values were tested and their correct operation was manually checked. This was done for several times.







### Full-code

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <time.h>

**#include** <malloc.h>

**int**\* **generate\_random\_numbers**(**int**);

**void** **smarter\_sort**(**int**\*, **int**, **int**);

**int** **partition**(**int**\*, **int**, **int**);

**int**\* **generate\_random\_numbers**(**int** numberOfRandomNumbers) {

**int**\* randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

**int** counter, randomNumber, i;

**for**(counter = 0; counter < numberOfRandomNumbers; counter++) {

randomNumber = (**rand**()% 100) + 1;

randomNumbersArray[counter] = randomNumber;

}

**printf**("Random Numbers (UNSORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

**return** randomNumbersArray;

}

**void** **smarter\_sort**(**int**\* unsortedArray, **int** left, **int** right) {

**int** j = 0;

**if**(left < right) {

//Divide and conquer

j = partition(unsortedArray, left, right);

smarter\_sort(unsortedArray, left, j-1);

smarter\_sort(unsortedArray, j+1, right);

}

}

**int** **partition**(**int**\* unsortedArray, **int** left, **int** right) {

**int** pivot, i, j, t;

pivot = unsortedArray[left];

i = left;

j = right+1;

**while**(1) {

**do** i++;

**while**(unsortedArray[i] <= pivot && i <= right);

**do** j--;

**while**(unsortedArray[j] > pivot);

**if**(i >= j)

**break**;

t = unsortedArray[i];

unsortedArray[i] = unsortedArray[j];

unsortedArray[j] = t;

}

t = unsortedArray[left];

unsortedArray[left] = unsortedArray[j];

unsortedArray[j] = t;

**return** j;

}

**int** **main**() {

**int** numberOfRandomNumbers;

**int** i;

//Used to seed the rand() function to generate different random numbers

**srand**(**time**(NULL));

**printf**("Please enter the number of Random Numbers to generate: ");

**fflush**(stdout);

**scanf**("%d", &numberOfRandomNumbers);

**int**\* randomNumbersArray;

randomNumbersArray = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

randomNumbersArray = generate\_random\_numbers(numberOfRandomNumbers);

smarter\_sort(randomNumbersArray, 0, (numberOfRandomNumbers-1));

**printf**("\n");

**printf**("Random Numbers (SORTED): \n");

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray[i]);

}

**free**(randomNumbersArray);

**return** 0;

}

## Part1G

In this section a test application was requested that takes care of generating the random numbers and the correct sorted outputs for both the naïve\_sort and the smarter\_sort function.

For this task, both the previous tasks were amalgamated together with some minor additions to satisfy the aim of this task. Therefore, all the functions were put in code (the naïve\_sort, the smarter\_sort and the partition function).

Some additions were made in the main program. Firstly, the program asks the user to input the wanted number of random numbers to be generated. Then, the appropriate memory allocations were set using the malloc() function for randomNumbersArray1 and randomNumbersArray2 which would eventually be the two arrays that are passed as arguments to be sorted by the naïve\_sort and the smarter\_sort. A set of random numbers generated by the generate\_random\_numbers() function fill the first array whilst the second is filled with the same elements as the first by using the memcpy() built-in function.

**int**\* randomNumbersArray1 = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

randomNumbersArray1 = generate\_random\_numbers(numberOfRandomNumbers);

**int**\* randomNumbersArray2 = (**int**\*)**malloc**(**sizeof**(**int**)\*numberOfRandomNumbers);

**memcpy**(randomNumbersArray2, randomNumbersArray1,**sizeof**(**int**)\*numberOfRandomNumbers);

The first generated random array is passed to the naïve sort function and then the sorted array is printed. During the same loop, a test function was implemented. This check includes the condition that if the number in position, let’s say [i] in the array is greater than the previous element, then, the array is not sorted. This result is then printed to state whether the naïve\_sort function works correctly.

naive\_sort(numberOfRandomNumbers, randomNumbersArray1);

**printf**("\n\n");

**printf**("Random Numbers (INSERTION SORTED): ");

**int** sorted = 1;

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray1[i]);

**if**(&randomNumbersArray1[i] < &randomNumbersArray1[i-1])

sorted = 0;

}

**printf**("\n");

**if**(sorted == 1)

**printf**("The insertion sort result is correctly sorted.");

**else**

**printf**("The insertion sort result is incorrect.");

**free**(randomNumbersArray1);

The same approach was taken to test the functionality of the smarter\_sort function using randomNumbersArray2.

smarter\_sort(randomNumbersArray2, 0, (numberOfRandomNumbers-1));

**printf**("\n\n");

**printf**("Random Numbers (QUICK SORTED): ");

sorted = 1;

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray2[i]);

**if**(&randomNumbersArray2[i] < &randomNumbersArray2[i-1])

sorted = 0;

}

**printf**("\n");

**if**(sorted == 1)

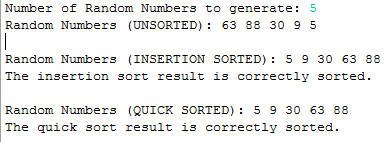
**printf**("The quick sort result is correctly sorted.");

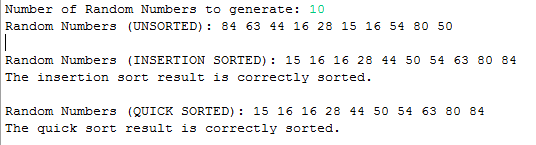
**else**

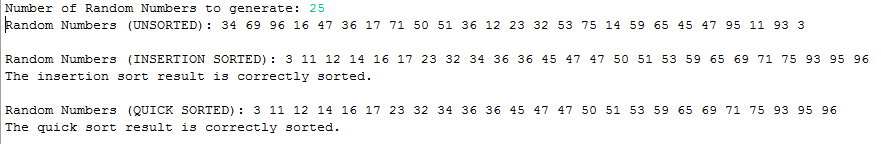
**printf**("The quick sort result is incorrect.");

**free**(randomNumbersArray2);

### Testing







## Part 1H

An enhancement on the previous task was requested in this part of the assignment. This involves the varying of array sizes and the time complexity of each sort function.

The clock() function included in <time.h> library was used to reach this objective. Two variables named startTime and endTime of type clock\_t were declared together with timeTaken of type double. These variables were then used to store the number of clock ticks elapsed since the program was launched up till a certain point. Thus, in the program the startTime was set to have a count before calling the naïve sort and afterwards for the endTime. The startTime was then subtracted from the endTime and divided by CLOCKS\_PER\_SEC to get the number of seconds used by the CPU. This result was stored in timeTaken and it is also printed in milliseconds.

clock\_t startTime, endTime;

**double** timeTaken;

startTime = **clock**();

naive\_sort(numberOfRandomNumbers, randomNumbersArray1);

endTime = **clock**();

timeTaken = (**double**)(endTime - startTime) / CLOCKS\_PER\_SEC;

**printf**("Random Numbers (INSERTION SORTED): ");

**int** sorted = 1;

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray1[i]);

**if**(&randomNumbersArray1[i] < &randomNumbersArray1[i-1])

sorted = 0;

}

**if**(sorted == 1)

**printf**("The Naive Sort result is correctly sorted.");

**else**

**printf**("The Naive Sort result is incorrect.");

**printf**("\n");

**printf**("The time taken for the Naive Sort to execute is %0.2fms.", (timeTaken\*1000));

**free**(randomNumbersArray1);

The same approach was taken on the smarter\_sort function.

startTime = **clock**();

smarter\_sort(randomNumbersArray2, 0, (numberOfRandomNumbers-1));

endTime = **clock**();

timeTaken = (**double**)(endTime - startTime) / CLOCKS\_PER\_SEC;

**printf**("Random Numbers (QUICK SORTED): ");

sorted = 1;

**for**(i = 0; i < numberOfRandomNumbers; i++) {

**printf**("%d ", randomNumbersArray2[i]);

**if**(&randomNumbersArray2[i] < &randomNumbersArray2[i-1])

sorted = 0;

}

**printf**("\n");

**if**(sorted==1)

**printf**("The Smarter Sort result is correctly sorted.");

**else**

**printf**("The Smarter Sort result is incorrect.");

**printf**("\n");

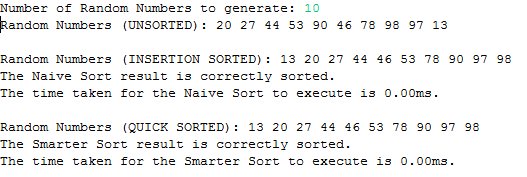
**printf**("The time taken for the Smarter Sort to execute is %0.2fms.", (timeTaken\*1000));

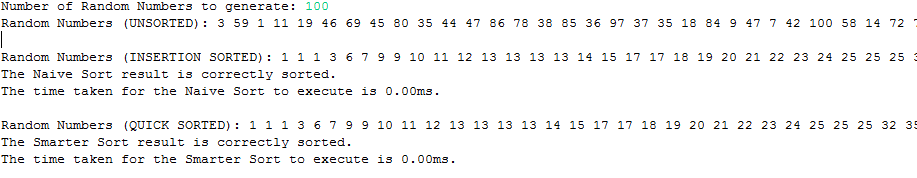
**printf**("\n\n");

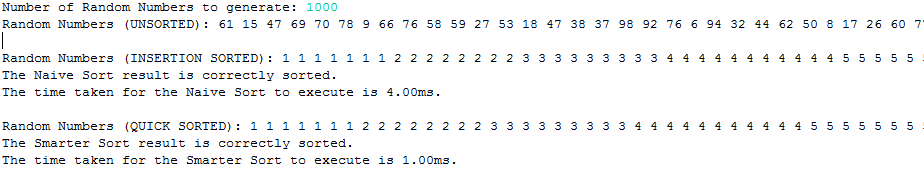
**free**(randomNumbersArray2);

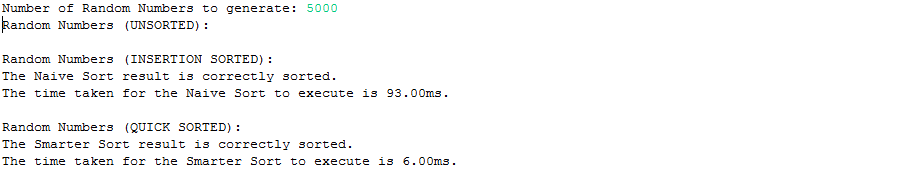
### Testing

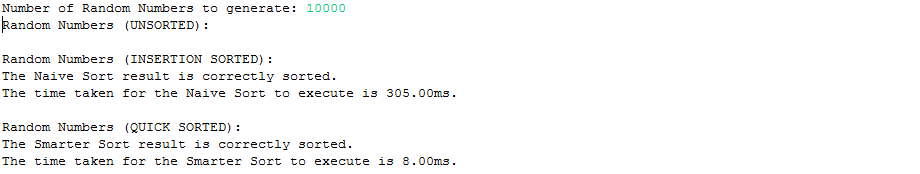
For testing, as the task requested, various input array sizes created according to the user inputs were tested and the execution time of each sort algorithm were noted. These are represented in the screenshots and graph below.











### Conclusion

From the research done before completing these last four tasks, many sources have mentioned the complexity of these algorithms. The **quick sort** is much faster on long lists having a complexity average of **O(n log n)**, whilst the **insertion** **sort** is inefficient on long lists **(O(n^2))** while it is approximately efficient as the quick sort and other similar sorting algorithms when it comes to short lists. This can be proven by the following graph where the y-axis represents the time taken whilst the x-axis represents the size of the list that was ordered:

From the results, the program confirms that the research found was correct. Therefore, concluding that the quick sort and the insertion sort are non-comparative as with respect to time for short lists, but then comparatively different when treating large lists of unordered elements. The smarter\_sort is much more efficient than the naïve\_sort.

## Part 1I

This task requested the *use of recursion* to count the total of cats observed during a bus trip.

A function called count\_cats(int) was implemented. This function takes the number of stops of the trip and returns the total number of cats observed by different employees throughout the journey. Two variables were declared to hold the data given by the user (number of cats observed) and the tally of these. Thus, for a particular stop) the data inputted by the user is incremented with the tally. The function then decrements the number of stops left, and calls the same function with the number of stops left until there are no stops ( => stops=0). When this state is reached, the function returns the totalCats variable which holds the tally of the cats observed.

The main program simply asks the user to input the number of stops during the trip and then calls the recursive function. Finally, it prints the return value of the function as a result.

**#include** <stdio.h>

**int** **count\_cats**(**int** stops);

**int** **count\_cats**(stops) {

**static** **int** totalCats;

**int** noOfCats;

**if**(stops > 0){

**printf**("Number of cats observed for bus stop %d: ", stops);

**fflush**(stdout);

**scanf**("%d", &noOfCats);

totalCats = totalCats + noOfCats;

stops--;

count\_cats(stops);

}

**return** totalCats;

}

**int** **main**(**void**) {

**int** busStops;

**int** totalNumberOfCats = 0;

**printf**("Please enter the number of bus stops during the trip: ");

**fflush**(stdout);

**scanf**("%d", &busStops);

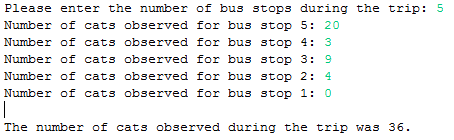
totalNumberOfCats = count\_cats(busStops);

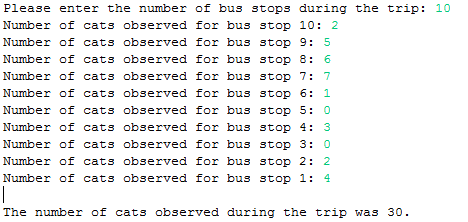
**printf**("\nThe number of cats observed during the trip was %d.", totalNumberOfCats);

**return** 0;

}

### Testing





## Part 1J

Using dynamic programming, we were requested to build a change computation that computes the minimum number of coins required to make up the change as requested by the user.

### Compute Change Function

To start off with, a 2D array was declared with 8 rows representing the number of the valid coins ( { 1, 2, 5, 10, 20, 50, 100, 200} ) and width columns equal to the number of change required + 1. The first row is then filled with incrementing numbers from 0 to the amount of change required using a for-loop.

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>

**#include** <malloc.h>

**void** **compute\_change**(**int**\*, **int**);

**void** **compute\_change**(**int**\* coins, **int** changeRequired) {

**int** i = 0;

**int** j = 0;

**int** least\_num = 0;

**int** array[8][changeRequired+1];

//Fill [0][1] with the change required (ascending order from 0)

**for**(j = 0; j <= (changeRequired); j++) {

array[i][j] = j;

}

Then, it was important to calculate the least number of coins required that amount to the change. The following pseudo code explains clearly the function of the next set of lines:

**for**(i = 1; i < 8; i++) {

**for**(j = 0; j <= (changeRequired); j++) {

**if**(j >= coins[i]) {

**if**(array[i-1][j] > (1+array[i][j-coins[i]]))

array[i][j] = 1 + array[i][j-coins[i]];

**else**

array[i][j] = array[i-1][j];

} **else** {

array[i][j] = array[i-1][j];

}

}

}

**For** every row starting from row 1 up till row 8  
 **For** every column starting from column 0 until j=ChangeRequired

**If** variable ‘j’ is greater or equal to stack coins at position [i]

**If** the 2D array value at the column above is greater than the (2D array position at [i][j-coins[i])+1

The value of the comparator is stored in the current location in the 2D array (ie:array[i][j])

**Else**

The value of the row above in the same column is copied to the current location in the 2D array

**Else:** The value of the row above in the same column is copied to the current location in the 2D array

The 2D array is then printed on screen using the following code:

**for**(i = 0; i < 8; i++) {

**for**(j = 0; j<= (changeRequired); j++)

**printf**("%d\t", array[i][j]);

**printf**("\n");

}

In order to get the least number of coins that make up to the change required, the bottom right corner value had to be fetched from the 2D array. Therefore, row 8 was selected by setting i=7 and the last column was selected by setting j to changeRequired. This value was then displayed.

i = 7;

j = changeRequired;

//Value in the bottom right corner

least\_num = array[i][j];

**printf**("\n");

**printf**("The least number of coins that make up the change is %d.\n", least\_num);

**int** arr2 [least\_num];

**int** count = 0;

**while**(array[i][j] != 0) {

**if**(array[i][j] == array[i-1][j]) {

i--;

} **else** {

j = j - coins[i];

arr2[count] = coins[i];

count ++;

}

}

**printf**("The coins given to the customer are ");

**for**(count = 0; count < least\_num; count ++) {

**if** (count == least\_num - 1)

**printf**("%d. ", arr2[count]);

**else**

**printf**("%d, ", arr2[count]);

}

}

**int** **main**(**void**) {

**int** terminate;

//int changeInserted = 0;

**do** {

**int** changeRequired = 0;

**int** coins[8] = {1,2,5,10,20,50,100,200};

**printf**("Enter the required change in cents: ");

**fflush**(stdout);

**scanf**("%d", &changeRequired);

**printf**("\n");

compute\_change(coins, changeRequired);

**printf**("\n\n");

**printf**("Press any key for further change or 0 to terminate");

**fflush**(stdout);

**scanf**("%d", &terminate);

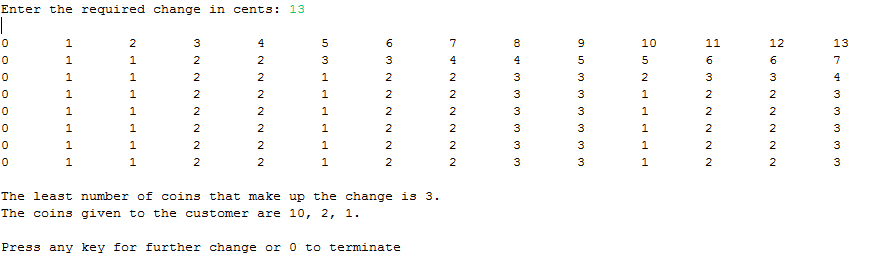
**printf**("\n--------------------------------------------------------------\n\n");

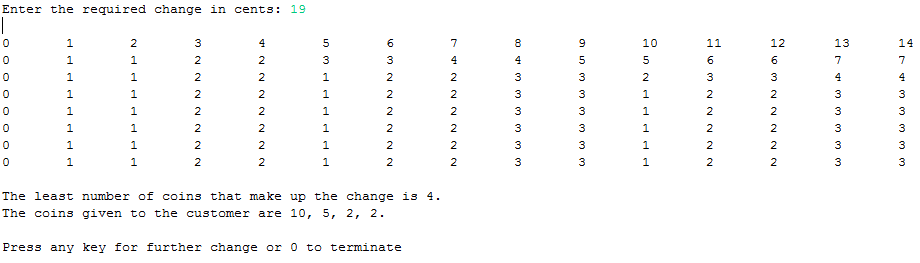
} **while**(terminate != 0);

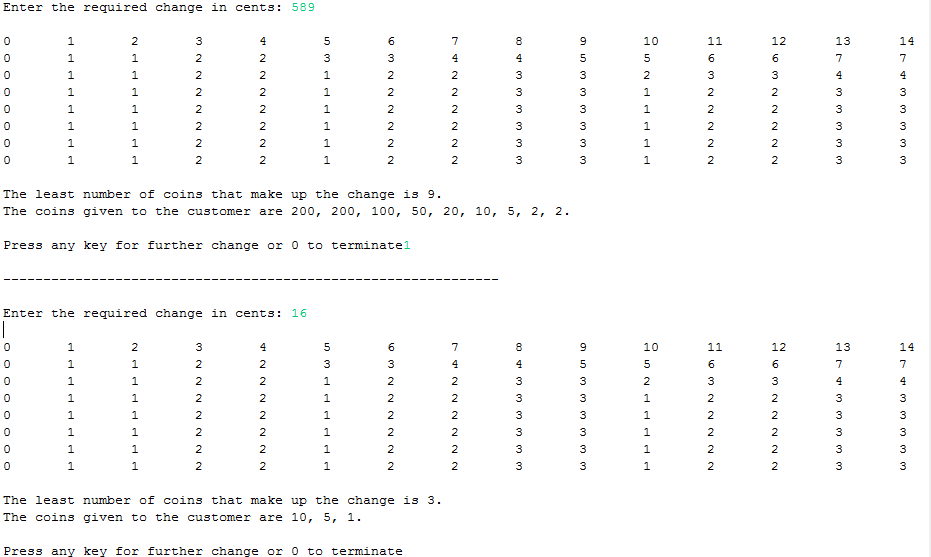
**return** 0;

}

### Testing







# Assignment Question 2

## Part 2A

In this task we were requested to check whether an inputted expression is balanced in respect to parenthesis. After several researches, the approach to complete this task was by implementing the push and pop functions.

A limit of 256 was defined as SIZE. This size is used to limit the stack to that size. Using a stack, the **stack pointer** was implemented by the variable topElement that was initiated as -1 since the stack is initialized empty.

**#include** <stdio.h>

**#define** SIZE 256

**int** topElement = -1;

**int** balanced = 1;

For both functions the stack had to be passed as arguments. Then the push and pop functions were implemented separately. The push function was used to push an element onto the stack. Therefore, firstly an if condition was used to check whether the stack is full by making sure that the topElement is not big as SIZE-1. If the stack is not full, the topElement is incremented and the element (passed as an argument to this function) is pushed to this position(topElement) in the stack.

**void** **push** (**char** stack[], **char** item);

**int** **pop** (**char** stack[]);

**void** **push** (**char** stack[], **char** item) {

**if** (topElement == (SIZE-1)){

//Stack is Full

balanced = 0;

} **else** {

//Push is Successful

balanced = 1;

++topElement;

stack [topElement] = item;

}}

On the contrary, the pop function was used to pop out the last pushed item on stack. Therefore, a check whether the stack is empty was firstly implemented and if it is not empty, the value of the stack at the topElement position is stored in a variable value (of type integer) and then decrement topElement. This decrement is used to stack point the top element after the pop elimination. This function returns the variable value.

**int** **pop** (**char** stack[]) {

**int** value;

**if** (topElement == -1){

//Stack is Empty

//There is nothing to Pop

value = 0;

balanced = 0;

} **else** {

//Pop is Successful

balanced = 1;

value = stack [topElement];

--topElement;

}

**return** value;

}

In both the above functions, the ‘balanced’ variable is used to hold a binary value of whether the push/ or the pop functions were successful. Obviously, this value was then used to check that the parentheses are equal.

In the main program, the user is asked to enter the expression. Using the getchar() function, the input is read character by character until reaching the [Enter] key and stored in an input variable. For each left parenthesis ‘(‘, the push function is called to push this item on stack whilst for every right parenthesis ‘)’, the pop function is called to remove this element from stack. If one of these functions set the ‘balanced’ value to 0, the while-loop breaks and the result automatically is concluded to be negative (ie: not balanced). Upon reaching the end of the expression, if the stack pointer (implemented as topElement) is equal to -1 and the ‘balanced’ variable is equal to 1, then the expression is balanced. Otherwise, the expression is also unbalanced.

**int** **main**(**void**) {

**char** stack[SIZE];

**char** input;

**printf**("Insert the Infix Expression and press 'Enter': ");

**fflush**(stdout);

**while**(input!= '\n') {

input = **getchar**();

**if**(input == '(')

push(stack, input);

**else** **if**(input == ')')

pop(stack);

//Check whether Right Parenthesis Matched

**if**(balanced == 0)

**break**;

}

**if**((topElement == -1) && (nalanced == 1))

**printf**("Balanced");

**else**

**printf**("Unbalanced");

**return** 0;

}

### Testing

Different cases were tested varying from no parenthesis, to multiple good use of the parenthesis and to cases where a right parenthesis is firstly found in the expression etc.. The following are some cases that were tested.











## Part 2B

This question was rather a straight forward task. The aim was building an expression converter from infix to post-fix notations. With the help of some guidelines in the question itself and research about these notations the following approach was taken to reach the objective.

Part 2A was utilized in order to use the POP() and PUSH() functions as they were required in the implementation of the converter. These functions are explained earlier, please refer.

**#include** <ctype.h>

**#include** <stdio.h>

**#define** SIZE 256

**int** topElement = -1;

**int** balanced;

**void** **push** (**char** stack[], **char** item);

**char** **pop** (**char** stack[]);

**void** **push** (**char** stack[], **char** item) {

**if** (topElement == (SIZE-1)){

balanced = 0;

} **else** {

balanced = 1;

++topElement;

stack[topElement] = item;

}

}

**char** **pop** (**char** stack[]) {

**char** value;

**if** (topElement == -1){

value = 0;

balanced = 0;

} **else** {

balanced = 1;

value = stack[topElement];

--topElement;

}

**return** value;  
}

### The Main Method

In the main method, the program asks the user to input the infix expression to be converted. This input is read character by character using the getchar() built-in function until the ‘Enter’ key is entered.

**int** **main**(**void**) {

**char** stack[SIZE];

**char** string[SIZE];

**char** character;

**char** x;

**int** position = 0;

**int** i = 0;

**printf**("Insert the Infix Expression and press 'Enter': ");

**fflush**(stdout);

**while**(character!= '\n') {

character = **getchar**();

Each character is analyzed to convert the order of the elements as required. Therefore, if a left parenthesis or any binary operator is encountered, this is pushed on to stack by calling the function push() and pass the character as argument. Else, if the character is a digit, this is stored in an array and the variable position is incremented. If a right parenthesis is encountered, the pop() function is called and the popped value is stored in a variable called ‘x’ that if it is not a left matching parenthesis, is also stored in the array with the digits and position is also incremented. This was implemented by the following code:

**if**((character == '(') || (character == '+') || (character == '-') || (character == '\*') || (character == '/')) {

//Push elements to the stack

push(stack, character);

} **else** **if**(**isdigit**(character)) {

string[position] = character;

position++;

} **else** **if**(character == ')') {

//Pop elements from the stack

x = pop(stack);

**if**(x != '(') {

//Add elements to the array

string[position] = x;

position++;

}

}

The expression till now is divided on the stack and in the string array by the preceding algorithm. Therefore, to join back these lists together the following algorithm was coded ,

**if**(topElement != -1) {

//Pop elements from the stack

x = pop(stack);

**if**(x != '(') {

//Add the operators to the array

string[position] = x;

position++;

}

}

Where if the stack is not empty, the string array is filled with the elements found in the stack in the next free positions, excluding the left parenthesis. (These elements are mostly the binary operators)

The postfix expression is then printed by printing the elements found in the array.

**printf**("Expression in Postfix: ");

**for**(i = 0; i < position; i++)

**printf**("%c ", string[i]);

**return** 0;

}

### Testing









## Part 2C

The evaluation of post-fix expressions was the objective of this task. What was implemented in Part2A and Part2B was utilized again. Though, a new function was built to evaluate the result and send it to the console.

### The postfix\_evaluation() function

This function evaluates the post-fix expression in order to give a result to the user. Therefore for each character that is not a NULL element, if it is a digit (checked by the isdigit() built-in function included in <ctype.h>), the digit is pushed onto the stack. Please **note** that the character (stored in variable z) is decremented by 48 due to the **ASCII** code characterization. Else, if the character is not a digit, two digits (operands) are popped and stored in two similar separate variable names ‘operand1’ and ‘operand2’.

**void** **postfix\_evaluation** () {

**char** z;

**int** i = 0;

**int** operand1;

**int** operand2;

**while**((z = string[i++]) != '\0') {

**if**(**isdigit**(z)) {

//Push operand

push(stack, (z - 48));

} **else** {

//Pop 2 operands and evaluate

operand2 = pop(stack);

operand1 = pop(stack);

By logic reasoning, if in the post-fix expression the character analyzed is not a digit, it is obviously one of the binary operators. Thus, for each case the two operands undergo the respective operation and this is pushed onto the stack again. This was implemented using the **switch-case statement** as follows:

**switch**(z) {

**case** '+': push(stack, (operand1 + operand2));

**break**;

**case** '-': push(stack, (operand1 - operand2));

**break**;

**case** '\*': push(stack, (operand1 \* operand2));

**break**;

**case** '/': push(stack, (operand1 / operand2));

**break**;

}

}

}

**printf**("\n");

**printf**("Result after Evaluation: %d", stack[topElement]);

}

Finally, the result is printed on the console.

### The Main Method

The same method of Part2B was used with some minor changes. The main change was the extra line of code that calls the function postfix\_evaluation() after printing the post-fix expression.

postfix\_evaluation();

### Testing













### Part 2C Full code

**#include** <ctype.h>

**#include** <stdio.h>

**#define** SIZE 256

**char** stack[SIZE];

**char** string[SIZE];

**int** topElement = -1;

**int** balanced;

**void** **push** (**char** stack[], **char** item);

**char** **pop** (**char** stack[]);

**void** **postfix\_evaluation** ();

**void** **push** (**char** stack[], **char** item) {

**if** (topElement == (SIZE-1)){

balanced = 0;

} **else** {

balanced = 1;

++topElement;

stack[topElement] = item;

}

}

**char** **pop** (**char** stack[]) {

**char** value;

**if** (topElement == -1){

value = 0;

balanced = 0;

} **else** {

balanced = 1;

value = stack[topElement];

--topElement;

}

**return** value;

}

**void** **postfix\_evaluation** () {

**char** z;

**int** i = 0;

**int** operand1;

**int** operand2;

**while**((z = string[i++]) != '\0') {

**if**(**isdigit**(z)) {

//Push operand

push(stack, (z - 48));

} **else** {

//Pop 2 operands and evaluate

operand2 = pop(stack);

operand1 = pop(stack);

**switch**(z) {

**case** '+': push(stack, (operand1 + operand2));

**break**;

**case** '-': push(stack, (operand1 - operand2));

**break**;

**case** '\*': push(stack, (operand1 \* operand2));

**break**;

**case** '/': push(stack, (operand1 / operand2));

**break**;

}

}

}

**printf**("\n");

**printf**("Result after Evaluation: %d", stack[topElement]);

}

**int** **main**(**void**) {

**char** character;

**char** x;

**int** position = 0;

**int** i = 0;

**printf**("Insert the Infix Expression and press 'Enter': ");

**fflush**(stdout);

**while**(character!= '\n') {

character = **getchar**();

**if**((character == '(') || (character == '+') || (character == '-') || (character == '\*') || (character == '/')) {

//Push elements to the stack

push(stack, character);

} **else** **if**(**isdigit**(character)) {

string[position] = character;

position++;

} **else** **if**(character == ')') {

//Pop elements from the stack

x = pop(stack);

**if**(x != '(') {

//Add elements to the array

string[position] = x;

position++;

}

}

}

**if**(topElement != -1) {

//Pop elements from the stack

x = pop(stack);

**if**(x != '(') {

//Add the operators to the array

string[position] = x;

position++;

}

}

**printf**("Expression in Postfix: ");

**for**(i = 0; i < position; i++)

**printf**("%c ", string[i]);

postfix\_evaluation();

**return** 0;

}

# Assignment Question 3

This task requests the use of dynamic arrays in order to make the previous program more memory efficient. This approach was already taken during the implementation of previous programs. Even though I did not have enough time to dedicate on making the program function with input and output files, **fopen()** and **fclose()** functions together with **FILE \*pointers** should have been used to reach the objective.

# Assignment Question 4

## Part 4

The objective of this task was to re-implement the stack data structure as an Abstract Data Type that exposes a particular interface. Also, to further improve memory efficiency the stack had to be implemented as a linked list. The previous program was then implemented using this ADT.

### The Header File (.h)

In the header file the interface of our program was coded. Using the **typedef** keyword, the **stackElement type** was defined together with the definition of **stackS using typedef struct**. As in the following code, stackS includes the variables ‘elements’ of type stackElement and ‘maxSize’ and ‘topElement’ of type Integer.

**#ifndef** ASSPART4\_H\_

**#define** ASSPART4\_H\_

**typedef** **char** stackElement;

**typedef** **struct** {

stackElement \*elements;

**int** maxSize;

**int** topElement;

} stackS;

The interface of the stack was then implemented using the following function declarations to satisfy the requested interface in the question: (**note** that is\_full() was also defined as part of the result when planning the source file)

**void** **stack**(stackS \*S, **int** maxSize);

**void** **push**(stackS \*S, stackElement p);

stackElement **pop**(stackS \*S);

stackElement **top**(stackS \*S);

**int** **is\_full**(stackS \*S);

**int** **is\_empty**(stackS \*S);

**#endif**

### The Source File (.c)

The source file, as usual, includes the libraries that include the built-in functions. When using header files, the following declaration is necessary to include the information in that file.

**#include** "AssPart4.h"

Also, the source file implementation includes the functions of the interface. Each function (stack(max\_size), push(S, p), pop(S), top(S), is\_empty(S) and is\_full(S)) was developed according to their functional aim.

For the stack function, firstly, enough memory allocations are set to variable ‘contents; of type stackElement. Then, the struct stackS is passed ‘contents’ as elements, ‘maxSize’ as maxSize and ‘-1’ as the topElement.

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>  
**#include** "AssPart4.h"

**void** **stack**(stackS \*S, **int** maxSize) {

stackElement \*contents;

contents = (stackElement\*)**malloc**(**sizeof**(stackElement) \* maxSize);

**if** (contents == NULL) {

**printf**("Not enough memory to initialize stack!");

} **else** {

S -> elements = contents;

S -> maxSize = maxSize;

S -> topElement = -1;

}

}

The push and pop functions were similar in logic to the ones used earlier. The push function though uses the is\_full() function to check whether the stack is full by checking whether the topElement of stackS is greater or equal to the maximum size of the stack-1.

**void** **push**(stackS \*S, stackElement p) {

**if**(is\_full(S)) {

**printf**("Stack is Full!");

} **else** {

S -> elements[++S -> topElement] = p;

}

}

stackElement **pop**(stackS \*S) {

stackElement p;

**if**(is\_empty(S)) {

**printf**("Stack is Empty!");

} **else** {

p = S -> elements[S -> topElement--];

}

**return** p;

}

**int** **is\_full**(stackS \*S) {

**return** S -> topElement >= S -> maxSize - 1;

}

Lastly, the is\_empty() function is implemented by checking whether the topElement is smaller than 0.

**int** **is\_empty**(stackS \*S) {

**return** S -> topElement < 0;}

### The Main Method

The main method was the most important part of this task. Here the previous program was implemented with the abstract data types created. The program first asks the user to input a sentence and the stack is initialized by calling the ‘stack’ function. The ‘stack’ was created to be of equal length to the length of the sentence inputted.

The push() function was then set into a for loop that repeats until it reaches a NULL element while also printing the \*pointer (therefore, the element pushed).

Similarly the pop() function was set in a while loop until the stack is not empty and printing the popping elements.

**int** **main**() {

stackS S;

**char** inputString[50];

**char**\* pointer;

**printf**("Enter a word or a sentence less than 50 characters: ");

**fflush**(stdout);

**scanf**("%[^\n]", inputString);

//Initialize the stack

stack(&S, **strlen**(inputString));

**printf**("\n");

**printf**("Characters Pushed: ");

**for**(pointer = inputString; \*pointer != '\0'; pointer++) {

push(&S, \*pointer);

**printf**("%c", \*pointer);

}

**printf**("\n");

**printf**("Characters Popped: ");

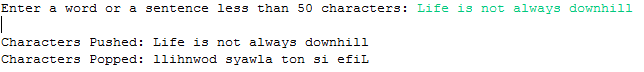
**while**(!is\_empty(&S)) {

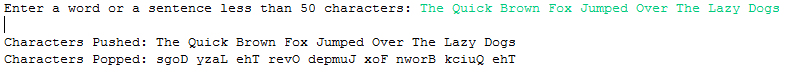
**printf**("%c", pop(&S));

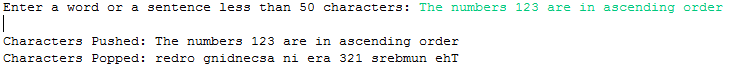
}

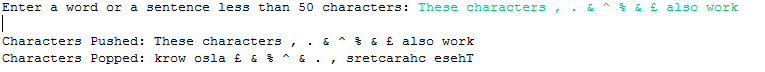
**return** 0;}

### Testing





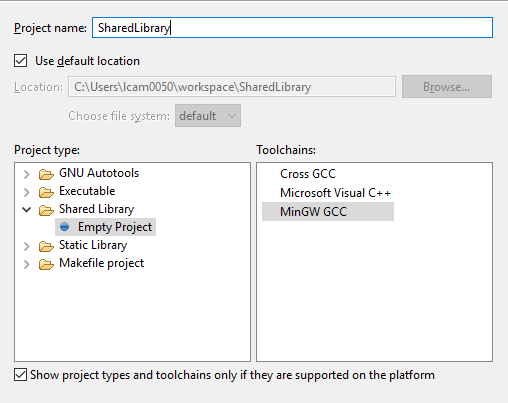




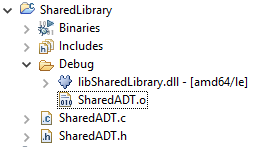
# Assignment Question 5

## Creating a shared library

In Eclipse, a new C project was created. This Project should be of type ‘Shared Library’ and the ‘Toolchain’ used should be ‘MinGW GCC’. After inputting the project name and finishing the setup, the files were added to this project.

****

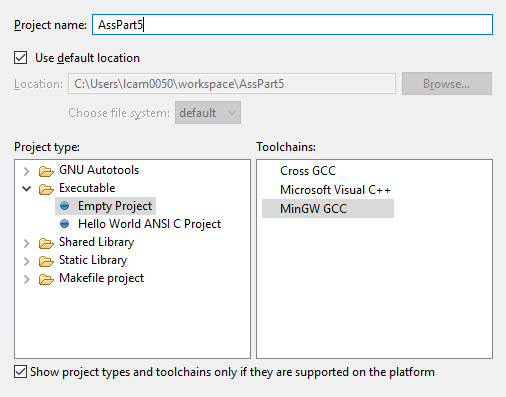
After building the project, the DLL files were created automatically and these were identified in the Debug Folder.



## Linking

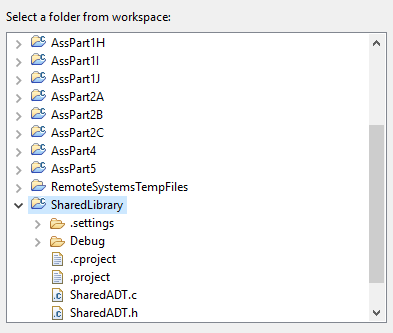
### Create Project

To initialize, a C project was first created. This project was set to be of Executable Type and the ‘Toolchain’S was set to ‘MinGW GCC’.

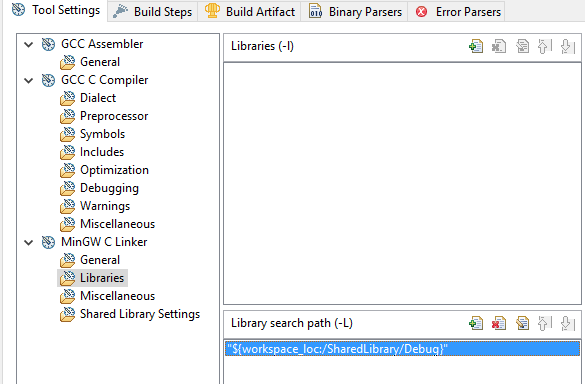
****

### Directory Link

From the project ‘Properties’, the ‘Paths and Symbols’ tab was selected. The GNU C was then selected and the Shared Library Project was added as a workspace in the directory path window.

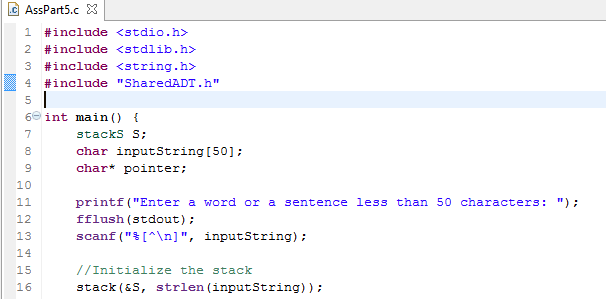


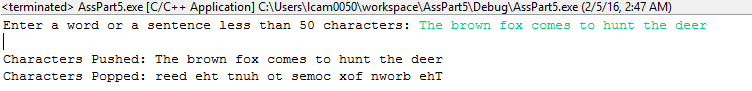
From the Built Tab, project settings were opened. The MinGW C Linker tab was selected and the Debug folder in which the DLL files were created was added to the Libraries. This was named the same as the shared library excluding the file type (.dll).



### Running the project

The project was then built. Before running, the DLL file from the Shared Library Debug Folder was copied to the Project Debug Folder. The Project was then run.





## Part 5 Full code

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>

**#include** "SharedADT.h"

**int** **main**() {

stackS S;

**char** inputString[50];

**char**\* pointer;

**printf**("Enter a word or a sentence less than 50 characters: ");

**fflush**(stdout);

**scanf**("%[^\n]", inputString);

//Initialize the stack

stack(&S, **strlen**(inputString));

**printf**("\n");

**printf**("Characters Pushed: ");

**for**(pointer = inputString; \*pointer != '\0'; pointer++) {

push(&S, \*pointer);

**printf**("%c", \*pointer);

}

**printf**("\n");

**printf**("Characters Popped: ");

**while**(!is\_empty(&S)) {

**printf**("%c", pop(&S));

}

**return** 0;

}

## SharedLibrary

### SharedADT.c Full code

**#include** "SharedADT.h"

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <string.h>

**void** **stack**(stackS \*S, **int** maxSize) {

stackElement \*contents;

contents = (stackElement\*)**malloc**(**sizeof**(stackElement) \* maxSize);

**if** (contents == NULL) {

**printf**("Not enough memory to initialize stack!");

} **else** {

S -> elements = contents;

S -> maxSize = maxSize;

S -> topElement = -1;

}

}

**void** **push**(stackS \*S, stackElement p) {

**if**(is\_full(S)) {

**printf**("Stack is Full!");

} **else** {

S -> elements[++S -> topElement] = p;

}

}

stackElement **pop**(stackS \*S) {

stackElement p;

**if**(is\_empty(S)) {

**printf**("Stack is Empty!");

} **else** {

p = S -> elements[S -> topElement--];

}

**return** p;

}

**int** **is\_full**(stackS \*S) {

**return** S -> topElement >= S -> maxSize - 1;

}

**int** **is\_empty**(stackS \*S) {

**return** S -> topElement < 0;

}

**int** **main**() {

stackS S;

**char** inputString[50];

**char**\* pointer;

**printf**("Enter a word or a sentence less than 50 characters: ");

**fflush**(stdout);

**scanf**("%[^\n]", inputString);

//Initialize the stack

stack(&S, **strlen**(inputString));

**printf**("\n");

**printf**("Characters Pushed: ");

**for**(pointer = inputString; \*pointer != '\0'; pointer++) {

push(&S, \*pointer);

**printf**("%c", \*pointer);

}

**printf**("\n");

**printf**("Characters Popped: ");

**while**(!is\_empty(&S)) {

**printf**("%c", pop(&S));

}

**return** 0;

}

### SharedADT.h Full code

**#ifndef** SHAREDADT\_H\_

**#define** SHAREDADT\_H\_

**typedef** **char** stackElement;

**typedef** **struct** {

stackElement \*elements;

**int** maxSize;

**int** topElement;

} stackS;

**void** **stack**(stackS \*S, **int** maxSize);

**void** **push**(stackS \*S, stackElement p);

stackElement **pop**(stackS \*S);

stackElement **top**(stackS \*S);

**int** **is\_full**(stackS \*S);

**int** **is\_empty**(stackS \*S);

**#endif**