

Programming Principles in C

ASSIGNMENT REPORT

Lydin Camilleri | Bachelor of Science in Computing Science | CPS1000

Table of Contents

Introduction	5
Assignment Question 1	6
Part 1A.....	6
Part 1B.....	9
<i>Testing</i>	11
Part 1C.....	12
Part 1D.....	14
Part 1E.....	16
<i>Generating Random Numbers</i>	16
<i>The Insertion Sort Method</i>	18
<i>The Naïve Sort Function</i>	19
<i>The Main Method</i>	20
<i>Testing</i>	21
<i>Full-code</i>	22
Part 1F	24
<i>Generating Random Numbers</i>	24

<i>The Quick Sort Method</i>	24
<i>The Smarter Sort Function</i>	25
<i>The Main Method</i>	26
<i>Testing</i>	27
<i>Full-code</i>	28
Part1G	30
<i>Testing</i>	32
Part 1H	33
<i>Testing</i>	35
<i>Conclusion</i>	36
Part 1I	37
<i>Testing</i>	38
Part 1J	39
<i>Compute Change Function</i>	39
<i>Testing</i>	43
Assignment Question 2	44
Part 2A	44
<i>Testing</i>	47

Part 2B.....	48
<i>The Main Method</i>	49
<i>Testing</i>	51
Part 2C.....	52
<i>The postfix_evaluation() function</i>	52
<i>The Main Method</i>	53
<i>Testing</i>	54
<i>Part 2C Full code</i>	55
Assignment Question 3.....	58
Assignment Question 4.....	59
Part 4.....	59
<i>The Header File (.h)</i>	59
<i>The Source File (.c)</i>	60
<i>The Main Method</i>	62
<i>Testing</i>	63
Assignment Question 5.....	64
Creating a shared library.....	64
Linking.....	65

<i>Create Project</i>	65
<i>Directory Link</i>	66
<i>Running the project</i>	67
Part 5 Full code.....	68
SharedLibrary	69
<i>SharedADT.c Full code</i>	69
<i>SharedADT.h Full code</i>	71

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.

```
Please enter the number of hours worked during the week = 10
|
Normal hours pay = Eur 225.00
Hours worked as Overtime = 0.00 hours
Over time extra pay = Eur 0.00
Gross pay is = Eur 225.00
Tax deductions = Eur 0.00
Net pay of this week's pay is = Eur 225.00
```

```
Please enter the number of hours worked during the week = 30
|
Normal hours pay = Eur 675.00
Hours worked as Overtime = 0.00 hours
Over time extra pay = Eur 0.00
Gross pay is = Eur 675.00
Tax deductions = Eur 83.55
Net pay of this week's pay is = Eur 591.45
```

```
Please enter the number of hours worked during the week = 47
|
Normal hours pay = Eur 900.00
Hours worked as Overtime = 7.00 hours
Over time extra pay = Eur 236.25
Gross pay is = Eur 1136.25
Tax deductions = Eur 217.31
Net pay of this week's pay is = Eur 918.94
```

```
Please enter the number of hours worked during the week = 55
|
Normal hours pay = Eur 900.00
Hours worked as Overtime = 15.00 hours
Over time extra pay = Eur 506.25
Gross pay is = Eur 1406.25
Tax deductions = Eur 295.61
Net pay of this week's pay is = Eur 1110.64
```

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

```
Please Enter Sentence Here: The Quick Brown Fox Jumps Over The Lazy Dog!
Average Number of Characters in Each Word: 3.89
Upper:Lower - 9:26
Please Enter Sentence Here: The quick brown fox jumps over the lazy dog!
Average Number of Characters in Each Word: 3.89
Upper:Lower - 1:34
Please Enter Sentence Here: What IF the WORLD EnDeD toDAY!
Average Number of Characters in Each Word: 4.00
Upper:Lower - 14:10
```

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 2^y 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;
}

```

Enter a value to be converted: 102

|

This value is to be converted to its base-n equivalent
where n is expressible as 2^y .

Enter n: 2

Result: 1100110

Enter a value to be converted: 55

|

This value is to be converted to its base-n equivalent
where n is expressible as 2^y .

Enter n: 8

Result: 67

Enter a value to be converted: 68

|

This value is to be converted to its base-n equivalent
where n is expressible as 2^y .

Enter n: 16

Result: 44

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 then 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;
}

```

```

Please enter string argument: It is never too late to apologize!
The length of the string is 34
Reversed string is "!ezigolopa ot etal oot reven si tI"

```

```

Please enter string argument: And what if , ThE # WOrLD stops. Tonight
The length of the string is 40
Reversed string is "thginoT .spots DLrOW # EhT , fi tahw dnA"

```


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:

```
Please enter the number of Random Numbers to generate: 10
Random Numbers (UNSORTED):
8 58 29 32 25 58 59 84 75 3
```

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.

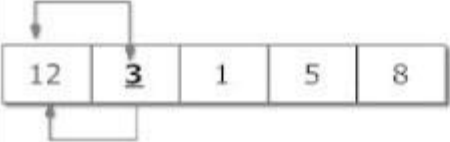
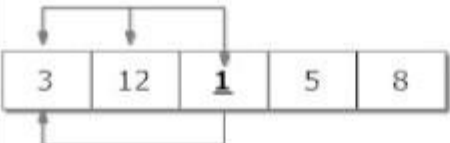
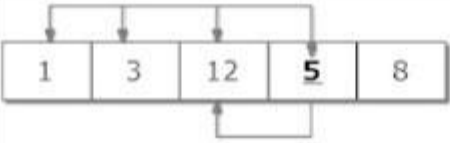
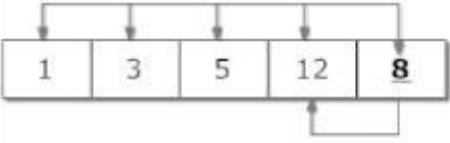

Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

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 `naive_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.

```

Please enter the number of Random Numbers to generate: 2
Random Numbers (UNSORTED):
59 32
Random Numbers (SORTED):
32 59

Please enter the number of Random Numbers to generate: 10
Random Numbers (UNSORTED):
88 50 32 11 79 57 86 52 84 4
Random Numbers (SORTED):
4 11 32 50 52 57 79 84 86 88

Please enter the number of Random Numbers to generate: 20
Random Numbers (UNSORTED):
77 45 45 61 61 74 18 52 13 46 63 13 25 97 33 61 49 37 71 29
Random Numbers (SORTED):
13 13 18 25 29 33 37 45 45 46 49 52 61 61 61 63 71 74 77 97

```

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.

```
Please enter the number of Random Numbers to generate: 2
Random Numbers (UNSORTED):
77 84
Random Numbers (SORTED):
77 84
```

```
Please enter the number of Random Numbers to generate: 10
Random Numbers (UNSORTED):
94 11 43 46 78 96 17 80 100 69
Random Numbers (SORTED):
11 17 43 46 69 78 80 94 96 100
```

```
Please enter the number of Random Numbers to generate: 20
Random Numbers (UNSORTED):
2 28 37 4 11 74 24 84 27 91 81 28 15 27 35 43 22 43 30 95
Random Numbers (SORTED):
2 4 11 15 22 24 27 27 28 28 30 35 37 43 43 74 81 84 91 95
```

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

```

Number of Random Numbers to generate: 5
Random Numbers (UNSORTED): 63 88 30 9 5
|
Random Numbers (INSERTION SORTED): 5 9 30 63 88
The insertion sort result is correctly sorted.

Random Numbers (QUICK SORTED): 5 9 30 63 88
The quick sort result is correctly sorted.

```

```

Number of Random Numbers to generate: 10
Random Numbers (UNSORTED): 84 63 44 16 28 15 16 54 80 50
|
Random Numbers (INSERTION SORTED): 15 16 16 28 44 50 54 63 80 84
The insertion sort result is correctly sorted.

Random Numbers (QUICK SORTED): 15 16 16 28 44 50 54 63 80 84
The quick sort result is correctly sorted.

```

```

Number of Random Numbers to generate: 25
Random Numbers (UNSORTED): 34 69 96 16 47 36 17 71 50 51 36 12 23 32 53 75 14 59 65 45 47 95 11 93 3
|
Random Numbers (INSERTION SORTED): 3 11 12 14 16 17 23 32 34 36 36 45 47 47 50 51 53 59 65 69 71 75 93 95 96
The insertion sort result is correctly sorted.

Random Numbers (QUICK SORTED): 3 11 12 14 16 17 23 32 34 36 36 45 47 47 50 51 53 59 65 69 71 75 93 95 96
The quick sort result is correctly sorted.

```

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.

```
Number of Random Numbers to generate: 10
Random Numbers (UNSORTED): 20 27 44 53 90 46 78 98 97 13

Random Numbers (INSERTION SORTED): 13 20 27 44 46 53 78 90 97 98
The Naive Sort result is correctly sorted.
The time taken for the Naive Sort to execute is 0.00ms.

Random Numbers (QUICK SORTED): 13 20 27 44 46 53 78 90 97 98
The Smarter Sort result is correctly sorted.
The time taken for the Smarter Sort to execute is 0.00ms.

Number of Random Numbers to generate: 100
Random Numbers (UNSORTED): 3 59 1 11 19 46 69 45 80 35 44 47 86 78 38 85 36 97 37 35 18 84 9 47 7 42 100 58 14 72 7
|
Random Numbers (INSERTION SORTED): 1 1 1 3 6 7 9 9 10 11 12 13 13 13 13 14 15 17 17 18 19 20 21 22 23 24 25 25 25 3
The Naive Sort result is correctly sorted.
The time taken for the Naive Sort to execute is 0.00ms.

Random Numbers (QUICK SORTED): 1 1 1 3 6 7 9 9 10 11 12 13 13 13 13 14 15 17 17 18 19 20 21 22 23 24 25 25 25 32 35
The Smarter Sort result is correctly sorted.
The time taken for the Smarter Sort to execute is 0.00ms.

Number of Random Numbers to generate: 1000
Random Numbers (UNSORTED): 61 15 47 69 70 78 9 66 76 58 59 27 53 18 47 38 37 98 92 76 6 94 32 44 62 50 8 17 26 60 7
|
Random Numbers (INSERTION SORTED): 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5
The Naive Sort result is correctly sorted.
The time taken for the Naive Sort to execute is 4.00ms.

Random Numbers (QUICK SORTED): 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5
The Smarter Sort result is correctly sorted.
The time taken for the Smarter Sort to execute is 1.00ms.

Number of Random Numbers to generate: 5000
Random Numbers (UNSORTED):

Random Numbers (INSERTION SORTED):
The Naive Sort result is correctly sorted.
The time taken for the Naive Sort to execute is 93.00ms.

Random Numbers (QUICK SORTED):
The Smarter Sort result is correctly sorted.
The time taken for the Smarter Sort to execute is 6.00ms.

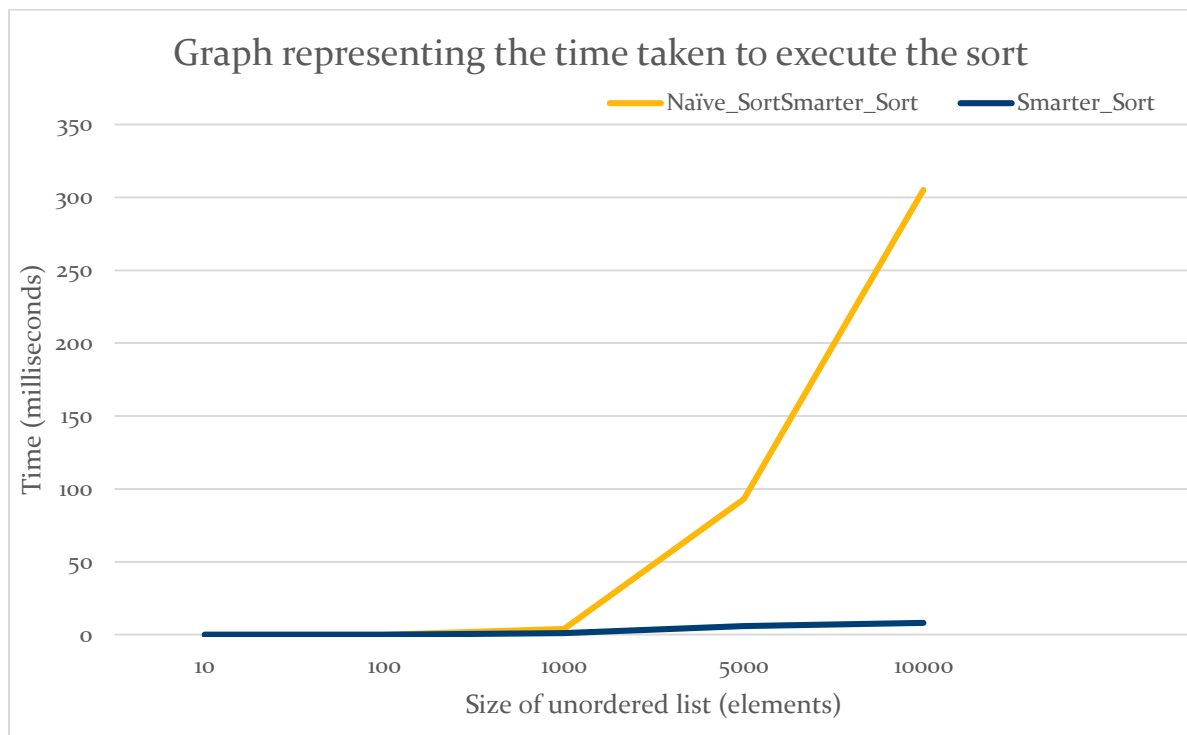
Number of Random Numbers to generate: 10000
Random Numbers (UNSORTED):

Random Numbers (INSERTION SORTED):
The Naive Sort result is correctly sorted.
The time taken for the Naive Sort to execute is 305.00ms.

Random Numbers (QUICK SORTED):
The Smarter Sort result is correctly sorted.
The time taken for the Smarter Sort to execute is 8.00ms.
```

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 (\Rightarrow 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

```

Please enter the number of bus stops during the trip: 5
Number of cats observed for bus stop 5: 20
Number of cats observed for bus stop 4: 3
Number of cats observed for bus stop 3: 9
Number of cats observed for bus stop 2: 4
Number of cats observed for bus stop 1: 0
|
The number of cats observed during the trip was 36.

```

```

Please enter the number of bus stops during the trip: 10
Number of cats observed for bus stop 10: 2
Number of cats observed for bus stop 9: 5
Number of cats observed for bus stop 8: 6
Number of cats observed for bus stop 7: 7
Number of cats observed for bus stop 6: 1
Number of cats observed for bus stop 5: 0
Number of cats observed for bus stop 4: 3
Number of cats observed for bus stop 3: 0
Number of cats observed for bus stop 2: 2
Number of cats observed for bus stop 1: 4
|
The number of cats observed during the trip was 30.

```

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");
    -----\n\n");

    } while(terminate != 0);

    return 0;
}

```

Testing

Enter the required change in cents: 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	1	2	2	3	3	4	4	5	5	6	6	7
0	1	1	2	2	1	2	2	3	3	2	3	3	4
0	1	1	2	2	1	2	2	3	3	1	2	2	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3

The least number of coins that make up the change is 3.

The coins given to the customer are 10, 2, 1.

Press any key for further change or 0 to terminate

Enter the required change in cents: 19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	1	1	2	2	1	2	2	3	3	2	3	3	4	4
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3

The least number of coins that make up the change is 4.

The coins given to the customer are 10, 5, 2, 2.

Press any key for further change or 0 to terminate

Enter the required change in cents: 589

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	1	1	2	2	1	2	2	3	3	2	3	3	4	4
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3

The least number of coins that make up the change is 9.

The coins given to the customer are 200, 200, 100, 50, 20, 10, 5, 2, 2.

Press any key for further change or 0 to terminate1

Enter the required change in cents: 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	1	2	2	3	3	4	4	5	5	6	6	7	7
0	1	1	2	2	1	2	2	3	3	2	3	3	4	4
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3
0	1	1	2	2	1	2	2	3	3	1	2	2	3	3

The least number of coins that make up the change is 3.

The coins given to the customer are 10, 5, 1.

Press any key for further change or 0 to terminate

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) && (balanced == 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.

```
Insert the Infix Expression and press 'Enter': 4+5-2
|Balanced
```

```
Insert the Infix Expression and press 'Enter': ((4*6)+5)
|Balanced
```

```
Insert the Infix Expression and press 'Enter': (((4)*3)+55)*(2+8)
|Balanced
```

```
Insert the Infix Expression and press 'Enter': )( ()
|Unbalanced
```

```
Insert the Infix Expression and press 'Enter': (4+2))
|Unbalanced
```


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

Insert the Infix Expression and press 'Enter': $((4*5)+6)$
Expression in Postfix: 4 5 * 6 +

Insert the Infix Expression and press 'Enter': $(4+(5*6))$
Expression in Postfix: 4 5 6 * +

Insert the Infix Expression and press 'Enter': $((7-2)*3)$
Expression in Postfix: 7 2 - 3 * |

Insert the Infix Expression and press 'Enter': $(9-(9/2))$
Expression in Postfix: 9 9 2 / -

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

Insert the Infix Expression and press 'Enter': $(4*(5+6))$
Expression in Postfix: 4 5 6 + *
Result after Evaluation: 44

Insert the Infix Expression and press 'Enter': $(4+(5*6))$
Expression in Postfix: 4 5 6 * +
Result after Evaluation: 34

Insert the Infix Expression and press 'Enter': $((4+5)*6)$
Expression in Postfix: 4 5 + 6 *
Result after Evaluation: 54

Insert the Infix Expression and press 'Enter': $(4+2)*5$
Expression in Postfix: 4 2 + 5 *
Result after Evaluation: 30

Insert the Infix Expression and press 'Enter': $((2*8)/4)$
Expression in Postfix: 2 8 * 4 /
Result after Evaluation: 4

Insert the Infix Expression and press 'Enter': $(9-(4+2))$
Expression in Postfix: 9 4 2 + -
Result after Evaluation: 3

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

```
Enter a word or a sentence less than 50 characters: Life is not always downhill
|
Characters Pushed: Life is not always downhill
Characters Popped: llihnwod syawla ton si efiL
```

```
Enter a word or a sentence less than 50 characters: The Quick Brown Fox Jumped Over The Lazy Dogs
|
Characters Pushed: The Quick Brown Fox Jumped Over The Lazy Dogs
Characters Popped: sgoD yzaL ehT revO depmuJ xof nworB kciuQ ehT
```

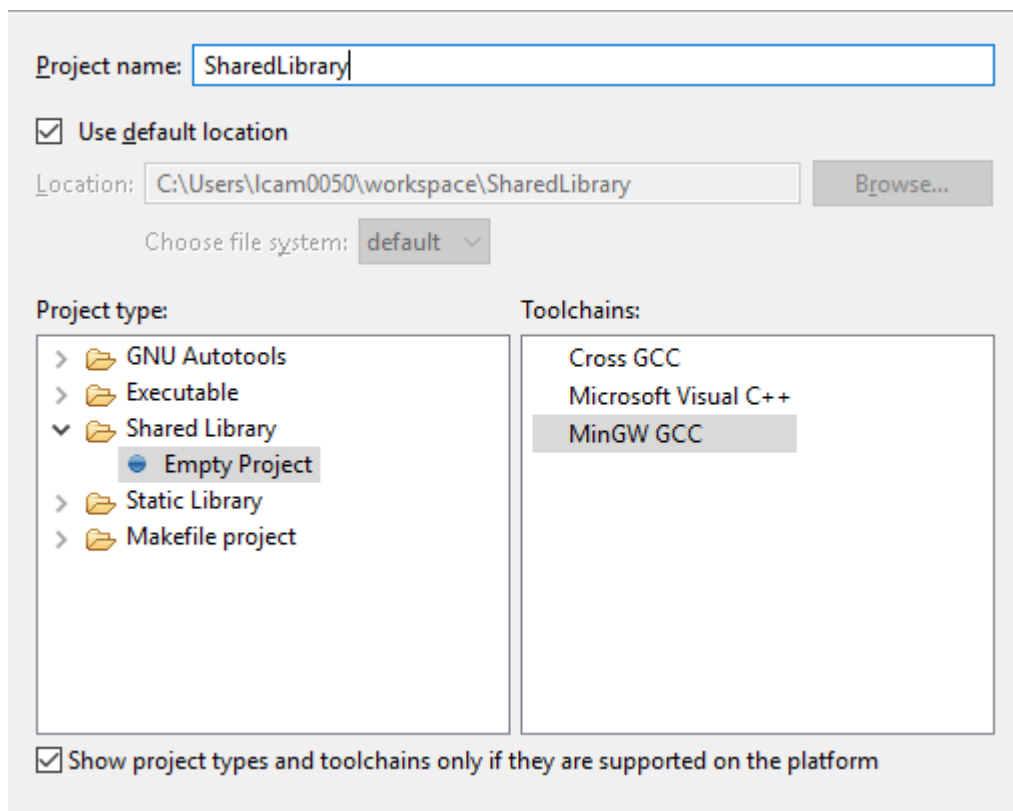
```
Enter a word or a sentence less than 50 characters: The numbers 123 are in ascending order
|
Characters Pushed: The numbers 123 are in ascending order
Characters Popped: redro gnidnecsa ni era 321 srebmun ehT
```

```
Enter a word or a sentence less than 50 characters: These characters , . & ^ % & f also work
|
Characters Pushed: These characters , . & ^ % & f also work
Characters Popped: krow osla f & % ^ & . , sretcarahc esehT
```

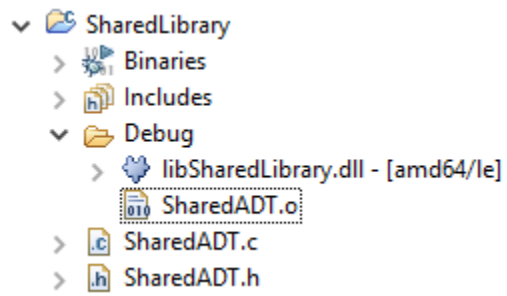

Assignment Question 5

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.



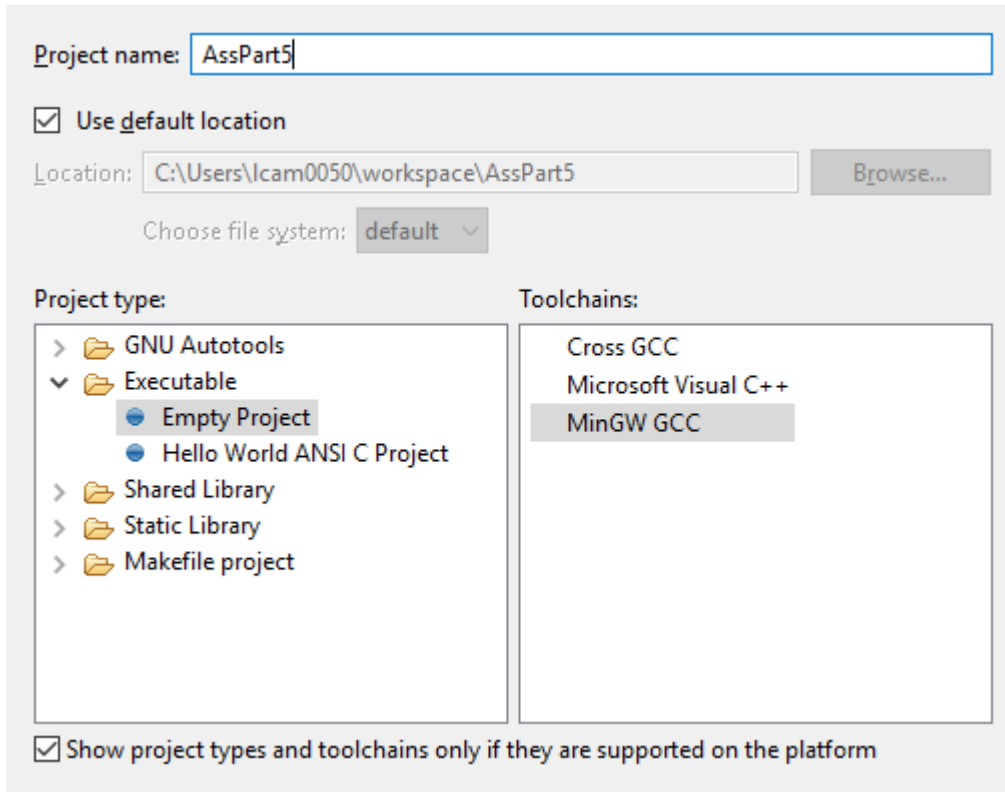
After building the project, the DLL files were created automatically and these were identified in the Debug Folder.



Linking

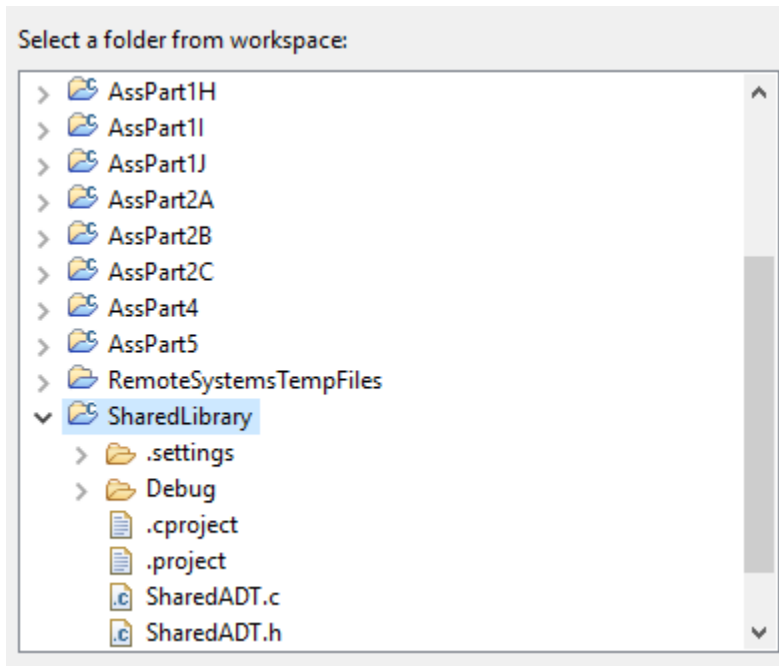
Create Project

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'.

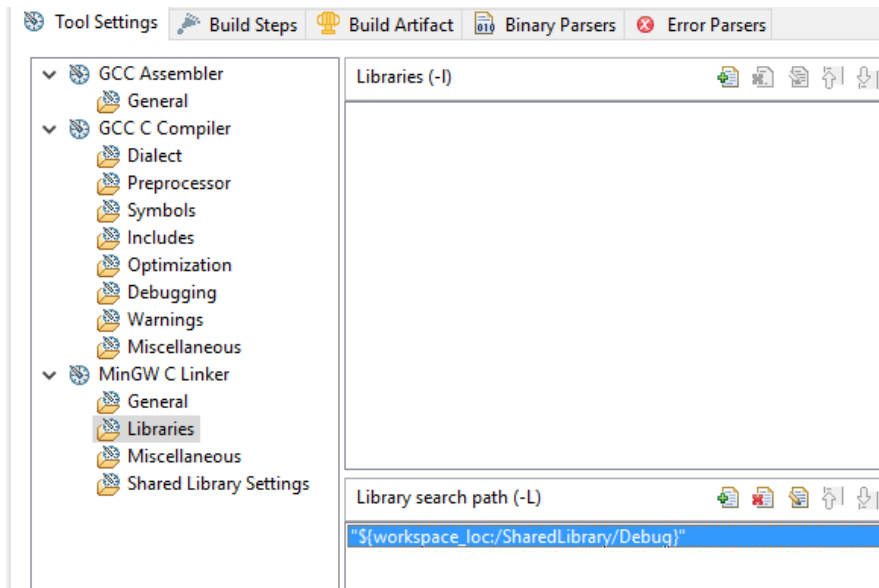


Directory Link

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.

```

AssPart5.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "SharedADT.h"
5
6  int main() {
7      stackS S;
8      char inputString[50];
9      char* pointer;
10
11     printf("Enter a word or a sentence less than 50 characters: ");
12     fflush(stdout);
13     scanf("%s[^\n]", inputString);

```

```

<terminated> AssPart5.exe [C/C++ Application] C:\Users\Icam0050\workspace\AssPart5\Debug\AssPart5.exe (2/5/16, 2:47 AM)
Enter a word or a sentence less than 50 characters: The brown fox comes to hunt the deer
|
Characters Pushed: The brown fox comes to hunt the deer
Characters Popped: reed eht tnuh ot semoc xof nworb eht

```

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
```