

CPS1000 - Programming Principles in C Assignment

Miguel Dingli
B.Sc. (Hons) in Computing Science

Contents

1	Problem Solving	3
1a	3
1b	7
1c	11
1d	16
1e	19
1f	24
1g	29
1h	35
1i	40
1j	43
2	Data Structures	49
2a	49
2b	55
2c	62
3	Allocated Memory and I/O	72
4	Abstract Data Types (ADT)	80
4a	80
4b	84
4c	90
5	Shared Libraries	94
6	Source code listing	96
6a	Task1a	96
6b	Task1b	97
6c	Task1c	98
6d	Task1d	100
6e	Task1e	101
6f	Task1f	102
6g	Task1g	104
6h	Task1h	108
6i	Task1i	111
6j	Task1j	112
6k	Task2a	114
6l	Task2b	117
6m	Task2c	122
6n	Task3	130
6o	Task4a	139

6p	Task4b	141
6q	Task4c	144
6r	Task5	154
7	Reference List	156

Introduction to the assignment

As an introduction to the assignment, a few notes about the assignment are given below.

- The IDE initially used to write the code was the Code::Blocks IDE. Any sample outputs of the execution shown will have been produced by Code::Blocks.
- The code was then transferred to projects in the Eclipse IDE so that the makefiles could easily be generated using the GNU Make Builder.
- Throughout the assignment, the order of code explanations and the corresponding code segment is such that code segments are always found after their respective explanation.
- Every task or subtask follows the same structure. An introduction contains the layout of the functions within the files, any limitations, and any assumptions made for the respective task. The code is then explained and includes code segments. At the end of a subtask, sample outputs are provided, when seen as necessary.
- Limitations of the code are not limited to the limitations listed since the code might include unidentified problems.

Task 1 Problem Solving

Task 1a

Introduction

The code for this task consists of two functions within a single source file. Below is a list showing the layout of the functions within the respective files.

- Task1a_source.c
 - int main(void)
 - void emptyInputBuffer(void)

A list of assumptions made for this task is found below.

1. Any input that is accepted by the ‘scanf’ function as used in the implementation is a valid input.
2. The ‘float’ data type is suitable for any input that the user will provide.

Header files included in Task1a_source.c

Starting off, the ‘stdio.h’ header file was included and the necessary symbolic constants were defined using ‘#define’ to be used by their symbolic name later on. These were written with ease of maintenance in mind as suggested in the task and are self-explained by their name.

```
1 #include <stdio.h> //puts(), scanf(), printf(), getchar()
2 #define TAXRATE1 (float) 15/100
3 #define TAXRATE2 (float) 29/100
4 #define PAYRATE 22.50
5 #define OVERTIMERATE 1.5
```

Function prototypes and ‘main’ function in Task1a_source.c

Next, a function ‘emptyInputBuffer’ was declared. This function will be used to clear the buffer if an invalid input was entered. In the ‘main’ function, variables were declared as floats for fractions of hours such as 50.5 hours to be accepted and for corresponding results to also have a fractional part. The first output for the user is now presented.

```
7 /* operation: clears the input buffer */
8 /* preconditions: input buffer contains unneeded characters */
9 /* postconditions: input buffer is clear */
10 void emptyInputBuffer(void);
11
12 int main(void)
13 {
14     /*Declarations and initialization of variables*/
15     float hoursWorked=0; //amount of hours worked
```

```

16     float grossPay=0;           //calculated grossPay
17     float tax=0;               //calculated tax
18     float netPay=0;           //calculated net pay
19
20     /*First output*/
21     puts("Please insert the number of hours worked in a week: ");

```

Next, a loop controls the simple input validation. When the return value of ‘scanf’ is zero (indicating that no value was read), the loop condition is satisfied and the user is asked to enter another value after clearing the input buffer. Otherwise, the loop terminates and an output indicates that the input was accepted.

```

23     /*Loop until the input is accepted*/
24     while(!scanf("%f", &hoursWorked))
25     {
26         emptyInputBuffer(); //dump the input
27         puts("Invalid input; Insert the hours worked in a week: ");
28     }
29
30     /*Output indicating the input that was accepted*/
31     printf("\nInput \".2f\" accepted\n", hoursWorked);

```

Next up is the gross pay calculation. If the hours worked exceeds 40 hours, the remaining hours worked (hoursWorked-40) are considered to count as 1.5 hours for every 1 hour by multiplying them to the over-time rate (OVERTIMERATE*(hoursWorked-40)) so that overtime will effectively be paid at x1.5 the basic rate in the next step. The total number of hours (40+(OVERTIMERATE*(hoursWorked-40))) is multiplied by the PAYRATE to get the grossPay.

On the other hand, if hours worked does not exceed 40 hours, the hours worked are simply multiplied by the PAYRATE.

```

33     /*Calculate gross pay*/
34     if(hoursWorked > 40)
35         grossPay = PAYRATE*(40+OVERTIMERATE*(hoursWorked-40));
36     else
37         grossPay = PAYRATE*hoursWorked;

```

Next is the calculation of the tax. If the gross pay does not exceed 300 hours, tax is left at zero. If gross pay exceeds 480 hours, the first 300 hours are ignored, the next 180 hours use TAXRATE1, and the remaining pay (grossPay-480) uses TAXRATE2.

Otherwise, if gross pay does not exceed 480 but exceeds 300, the first 300 hours are ignored, and the remaining pay (grossPay-300) uses TAXRATE1.

```

39     /*Calculate tax*/
40     if(grossPay > 300)
41     {
42         if(grossPay > 480)
43             tax = (TAXRATE1*180)+(TAXRATE2*(grossPay-480));
44         else
45             tax = TAXRATE1*(grossPay-300);
46     }

```

The net pay is then calculated by a simple difference formula and all of the calculated values are output to the user in an organized manner by using maximum field size of 10 characters and rounding values to 2 decimal places.

```
48  /*netPay calculated by subtracting tax from grossPay*/
49  netPay = grossPay - tax;
50
51  /*Outputs to the user rounded to two decimal places*/
52  printf("-----\n");
53  printf("Gross pay: %10.2f Eur\n", grossPay);
54  printf("Tax pay:   %10.2f Eur\n", tax);
55  printf("-----\n");
56  printf("Net pay:   %10.2f Eur\n", netPay);
57  printf("-----\n");
58
59  return 0;
60 }
```

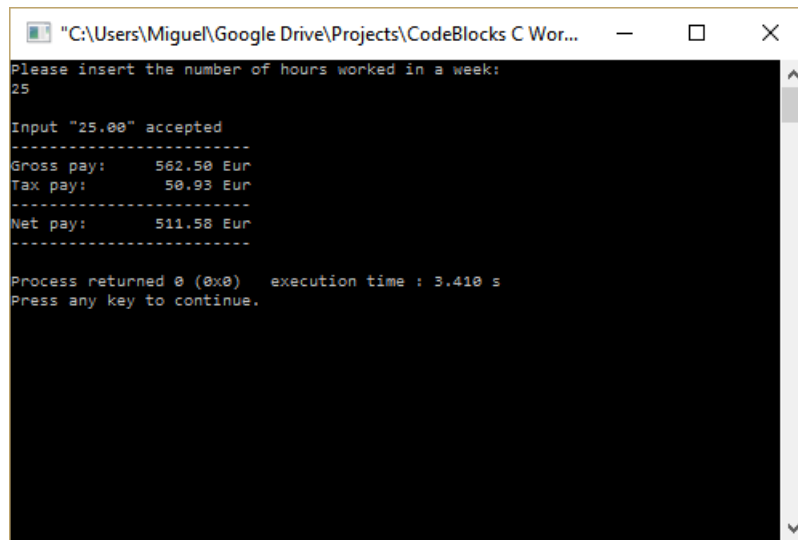
‘emptyInputBuffer’ function in Task1a.source.c

The implementation for the function ‘emptyInputBuffer’ will now be discussed. The loop in the function loops until the newline character (present due to the user pressing the ENTER key) is found, indicating that all characters in the input buffer including the newline character were cleared. The point of this function is to clear the input buffer for the next user input.

```
63 void emptyInputBuffer(void)
64 {
65     while(getchar() != '\n');
66 }
```

Testing

For testing purposes, two sample outputs will be shown. The input given to the program can be seen from the screenshots themselves. Kindly proceed to the next page for the sample outputs.

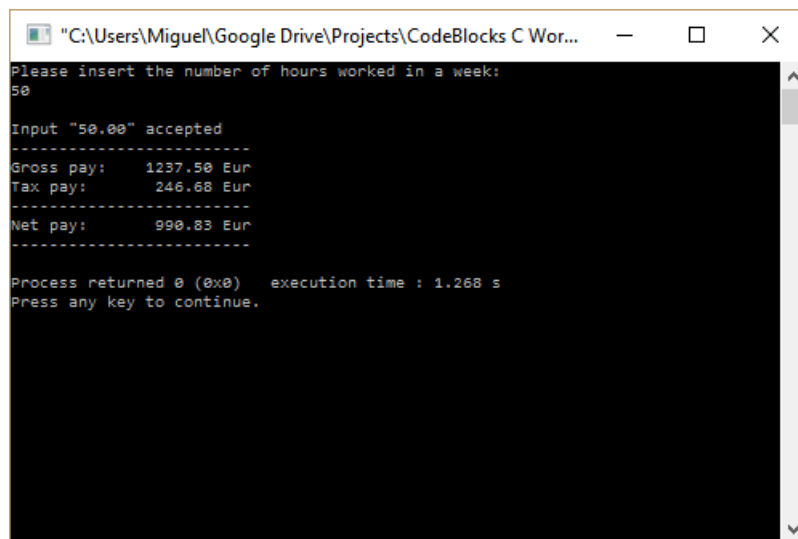


```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Please insert the number of hours worked in a week:
25

Input "25.00" accepted
-----
Gross pay:      562.50 Eur
Tax pay:        50.93 Eur
-----
Net pay:        511.58 Eur
-----

Process returned 0 (0x0)   execution time : 3.410 s
Press any key to continue.
```

Figure 1: Task 1a sample output 1



```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Please insert the number of hours worked in a week:
50

Input "50.00" accepted
-----
Gross pay:      1237.50 Eur
Tax pay:         246.68 Eur
-----
Net pay:         990.83 Eur
-----

Process returned 0 (0x0)   execution time : 1.268 s
Press any key to continue.
```

Figure 2: Task 1a sample output 2

The results can easily be verified. Consider the first sample output; for 25 hours, the gross pay is $25 \times 22.50 = 562.5$ Euro. Tax is 0% for the first 300 Euro. It is 15% for the next 180 Euro, so the tax so far is $180 \times 15\% = 27$ Euro and the tax is 29% for the remaining 82.5 Euro, so the total tax is $27 + (82.5 \times 29\%) = 50.93$ Euro. Hence the net pay is $562.5 - 50.93 = 511.58$ Euro. The second sample output can be confirmed using a similar process but where 10 of the 50 hours need to be paid at the overtime rate.

Task 1b

Introduction

The code for this task consists of two functions within a single source file. Below is a list showing the layout of the functions within the respective files.

- Task1b_source.c
 - int main(void)
 - int findHCF(const int num1, const int num2)

A list of assumptions made for this task is found below.

1. A character that is not alphanumeric does not count as part of a word.
2. Words made up of digits only (i.e. numbers) still count as words.

Header files, function prototypes, and 'main' function in Task1b_source.c

The 'stdio.h', 'stdbool.h', and 'ctype.h' header files were included. The 'stdbool.h' header file includes a 'bool' type and the values 'true' and 'false' which were used for more easily readable code. The 'ctype.h' header file provides functions to check the characteristics of characters used while going through the user's input.

A function 'findHCF' was declared. This function will be used to find the highest common factor of the upper-case and lower-case letter counts so that it can be used to reduce their ratio to its simplest form. In the 'main' function, the necessary variables were declared and initialized followed by the first output.

```
1 #include <stdio.h>    //printf(), getchar()
2 #include <stdbool.h>  //bool, true, false
3 #include <ctype.h>    //isalnum(), isalpha(), isupper()
4
5 /* operation: finds and returns the Highest Common Factor of two numbers*/
6 /* preconditions: num1 and num2 are two integers */
7 /* postconditions: highest common factor is returned */
8 int findHCF(const int num1, const int num2);
9
10 int main(void)
11 {
12     /*Declaration and initialization of variables*/
13     char ch;                //stores input characters
14     int letters=0, words=0; //store amount of letters and words
15     int uppers=0, lowers=0; //store amount of upper/lower-case letters
16     int HCF=0;              //stores the HCF of uppers and lowers
17     bool needNextWord = true; //indicates whether a word is being expected
18
19     /*First output*/
20     printf("Insert text (insert an EOF on a line by itself to stop):\n");
```


A loop now goes through the input character-by-character. It loops until an end-of-file (EOF) at the start of a line is read, indicating that the end of the user input was reached. If the character is not an EOF, it goes through a checking process.

Firstly, the 'isalnum' function is used to check if the character is alphanumeric (a letter or a digit). If so, 'needNextWord' is set to false, indicating that the beginning of a word was found (i.e. next word not needed since one was just found). Now, a further condition checks if the character is alphabetic using the 'isalpha' function. If so, 'letters' is incremented. Now another if...else statement checks if the character is an upper-case or lower-case letter and the respective variable is incremented. None of the above-mentioned variables are incremented if the character is a digit. Despite this, a word partly or fully made up of digits is still considered to be a normal word.

On the other hand, if the character is not alphanumeric (including spaces and special characters), 'needNextWord' is checked. If 'needNextWord' is false, this indicates that a word was previously found and the end of it was just discovered since a character which is not considered to be a part of a word was found. Hence, a complete word was formed and so the 'words' variable is incremented. 'needNextWord' is reset to 'true' so that the search for a new word starts (i.e. a next word is needed again since one just ended).

```
22  /*Loop until an EOF is read*/
23  while((ch = getchar()) != EOF)
24  {
25      /*if character is alphanumeric*/
26      if(isalnum(ch))
27      {
28          /*if a word was being expected, it is not exp-*/
29          /*ected anymore since a word's start was found*/
30          if(needNextWord)
31              needNextWord = false;
32
33          /*if character is alphabetic (letter)*/
34          if(isalpha(ch))
35          {
36              letters++;          //letter found
37
38              /*Check case*/
39              if(isupper(ch))
40                  uppers++;      //upper-case letter found
41              else
42                  lowers++;       //lower-case letter found
43          }
44      }
45      else if(!needNextWord)
46      {
47          words++;                //complete word formed
48          needNextWord = true;    //next word needed
49      }
50  }
```

After the loop, the highest common factor (HCF) of the letter case counts is found using the ‘findHCF’ function found in the same source file.

```
52  /*Calculate HCF*/
53  HCF = findHCF(upper, lower);
```

Outputs to the user now show all of the counts: letters, words, upper-case letters and lower-case letters. For the last two outputs, the average letters per word and ratio of upper-case to lower-case letters are calculated. A conditional statement for the average letters per word handles the case where words is zero, preventing a division by zero. This value is rounded to 3 decimal places. For the last output, the ratio is reduced to its simplest form by dividing the upper-case and lower-case letter counts by the HCF.

```
55  /*Outputs*/
56  printf("Letters: %d\n", letters);
57  printf("Words:   %d\n", words);
58  printf("Uppers:  %d\n", upper);
59  printf("Lower:   %d\n", lower);
60  printf("Average letters per word: %.3f\n", words>0 ? ((float)letters/
61  words) : 0);
62  printf("Ratio of upper- to lower-case letters: %d:%d\n", upper/HCF,
63  lower/HCF);
64  return 0;
```

‘findHCF’ function in Task1b_source.c

Now, the ‘findHCF’ function was implemented. The HCF is calculated by finding the first number that is a factor of both num1 and num2. If both num1 and num2 are zero, 1 is returned since there is no point in finding the HCF and the ratio in this case will be 0:0. Otherwise, if num1 and num2 are not both zero, values from num1 down to 1 are checked. The first number that does not leave a remainder when num1 and num2 are divided by it is the HCF. The value i=1 serves as the base case since 1 is a factor of any integer.

If num2 is equal to zero, the condition in the loop is immediately satisfied and num1 is returned since it is the initial value of ‘i’. If num1 is equal to zero, the for loop condition ‘i>0’ is not satisfied and num2 is returned from outside the loop. This means that the ratio will be simplified to 1:0 or 0:1 if either num1 or num2 are zero.

```
66  int findHCF(const int num1, const int num2)
67  {
68      int i=0; //loop counter
69
70      /*If both numbers are zero, HCF is 1. */
71      /*Otherwise, the HCF is calculated.  */
72      if(num1==0 && num2==0)
73          return 1;
74      else
```

```

75     {
76         /*Loop from num1 down to 1*/
77         for(i=num1; i>0; i--)
78         {
79             /*if 'i' is a factor of both numbers*/
80             if (num1%i==0 && num2%i==0)
81                 return i; //return common factor
82         }
83     }
84
85     /*if num1 is 0, num2 is the HCF*/
86     return num2;
87 }

```

Testing

For testing purposes, one sample output will be shown. The input given to the program can be seen from the screenshot itself found below.

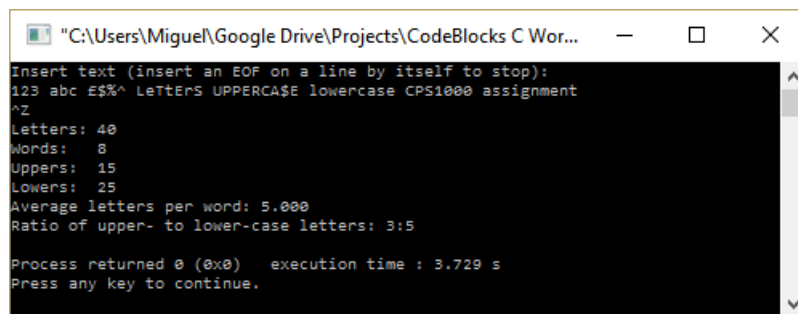
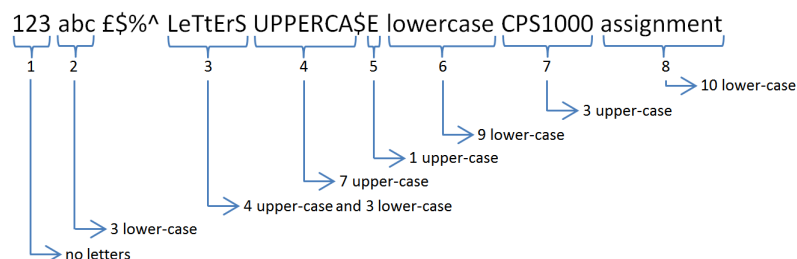


Figure 3: Task 1b sample output

This result can easily be verified by going through the input character by character. In the diagram below, the words are grouped and the amount of lower-case and upper-case letters in each word is listed. The digit below each word is the word count up to the respective word.



Task 1c

Introduction

The code for this task consists of three functions within a single source file. Below is a list showing the layout of the functions within the respective files.

- Task1c_source.c
 - int main(void)
 - void to_base_n(int num, const int base)
 - void emptyInputBuffer(void)

A list of assumptions made for this task is found below.

1. Any input that is accepted by the ‘scanf’ function is a valid input.
2. The user will not ask for values larger than the base-64 representation of 35_{10} since such values would use complicated symbols.
3. The user will not ask for the base-n representation of negative values.

A list of identified limitations of the code is found below.

1. If the input base is floating-point, it is still accepted, but only the integer part is read. When the program expects the next input, ‘scanf’ will attempt to read the fractional part left in the input buffer but the input is immediately not accepted, even though the user never got the chance to input a different number and base. This results in an awkward output.

Header files and function prototypes in Task1c_source.c

Starting off, the ‘stdio.h’ and ‘math.h’ header files were included. The ‘math.h’ header file provides a \log_2 function which will be used to check that the user’s input for the base is a power of two integer.

```
1 #include <stdio.h> //printf(), scanf(), putchar(), getchar()
2 #include <math.h> //log2()
```

Two functions ‘to_base_n’ and ‘emptyInputBuffer’ were now declared. The ‘to_base_n’ function will be used to convert a number into its base-n equivalent. The ‘emptyInputBuffer’ function was obtained from task 1a.

```
4 /* operation: transforms num to its base-n equivalent and outputs result */
5 /* preconditions: num is a base-10 number and n is a power-of-2 integer */
6 /* postconditions: the base-n equivalent of num is output to the user */
7 void to_base_n(int num, const int n);
8
9 /* operation: clears the input buffer */
10 /* preconditions: input buffer contains unneeded characters */
11 /* postconditions: input buffer is clear */
12 void emptyInputBuffer(void);
```

‘main’ function in Task1c_source.c

In the ‘main’ function, the required variables were declared and initialized.

```

14 int main(void)
15 {
16     /*Declaration and initialization of variables*/
17     int inputVerif=0;    //stores return value of inputs
18     double log2base=0;  //stores log2 of the base
19     int number=0;       //stores the input decimal number
20     int base=0;         //stores the input integer base

```

A while loop now starts with a condition which is always satisfied. The loop will repeat until the number or base input are negative (in which case the loop is terminated by a ‘break’). The return value of ‘scanf’ is stored in ‘inputVerif’ to be checked.

```

22     /*infinitely loops until any input is negative*/
23     while(1)
24     {
25         /*Output and input (of number and base)*/
26         printf("Insert a decimal number and a power-of-2 integer ");
27         printf("base (or insert a negative number or base to quit): \n");
28         inputVerif = scanf("%d %d", &number, &base);
29
30         /*negative number or base taken as a 'quit'*/
31         if(number<0 || base<0)
32             break;

```

Since the base has to be a power of 2 integer (i.e. 1, 2, 4, 8, 16 etc.), the \log_2 of the base is calculated. In the next if statement condition, it is checked whether \log_2 of the base is an integer by checking if it is equal to the integer-equivalent of itself by using typecasting (this method was used instead of checking if $\text{base} \% 1$ is zero since base is a double and the modulus operator only works for integers). If \log_2 of the base is an integer, this means that the base can be represented in the form 2^x , where x is an integer, making it a valid base for the program. Within the same condition, ‘inputVerif’ is also checked to verify that two integers were accepted in the previous ‘scanf’. If the conditions are both satisfied, the number is either left as zero (if it is zero) or converted to base-n. Otherwise, an output indicates that one or both of the inputs were invalid. In this case, the input buffer is cleared using the ‘emptyInputBuffer’ function to accept a new pair of integers.

Kindly proceed to the next page for the code corresponding to the above description.

```

34     /*log2 of base is calculated*/
35     log2base = log2(base);
36
37     /*if input was accepted and if baseVerif is an integer*/
38     if(inputVerif==2 && (log2base) == (int)log2base)
39     {
40         printf("The base-%d representation is: ", base);
41
42         /*If number is zero, it remains zero */
43         /*Otherwise, it is converted to base-n*/
44         if(number == 0)
45             putchar('0');           //result is zero
46         else
47             to_base_n(number, base); //conversion to base_n
48     }
49     else //else if the input was rejected
50     {
51         printf("Invalid input; One/both of the inputs were invalid.");
52         emptyInputBuffer(); //clears input buffer
53     }
54     printf("\n\n");
55 }
56
57 return 0;
58 }

```

‘to_base_n’ function in Task1c_source.c

The implementation of the ‘to_base_n’ function will be discussed next. To start with, a condition checks if the base is 1. In this case, since only one character can represent the result, a stream of a ‘num’ amount of 1’s is printed and execution is returned to the function call in the ‘main’ function. If the base is not 1 then the conversion to base-n begins.

The conversion process can follow a set of three general steps that are repeated. Firstly, the decimal number is divided by the base. Secondly, the remainder from this division is added as the most significant digit to the expression being generated. And thirdly, the above two steps are repeated, considering the quotient from the division to be the decimal number in each round of the steps until the decimal number reaches zero. [1]

Following the above method, the ‘to_base_n’ function is recursively called, each time dividing the number by the base until the decimal number loses all of its value (i.e. it gets rounded to zero) in which case the condition ‘num==0’ acts as the base case. Before each recursive call of the function, the remainder of dividing the number by the base is calculated using the modulus operator and will be in the range of 0 to ‘base-1’. For a more in-depth understanding as to how and why this method works, the workings were seen from a different interpretation. The remainder can be seen as signifying how many counts of $base^x$ are needed to represent the decimal number, where x is the x^{th} digit in the final expression, starting from $x = 0$, the least-significant digit. This is further explained in the following example.

Example: consider 6_{10} to be converted to base-2. 6 divided by 2 is 3 remainder **0**, so zero 2^0 values are required. 3 divided by 2 is 1 remainder **1**, so one 2^1 value is required. Finally, 1 divided by 2 is 0 remainder **1**, so one 2^2 value is required. The number (6_{10}) lost its value, so no more digits are found. Reading the remainders in reverse results in 110_2 which is the base-2 equivalent of 6_{10} .

After the base case 'num==0' is reached, as execution backtracks through the function calls, the remainders are output (in reverse order) which is actually outputting the base-n representation of the number since digits of higher significance were calculated later. The characters output are based on the value of the remainders. For 0-9, a digit is output, but for 10-35, the characters A-Z are output. Assuming that the user does not ask for the base-64 representation of 36_{10} , or larger values, letters A-Z will suffice. Otherwise, the characters following Z in the American Standard Code for Information Interchange (ASCII) table are used, which may not produce an easily-understandable or user-friendly result. In fact, the first character after the letter 'Z' is the open square parenthesis '['.

```

60 void to_base_n(int num, const int base)
61 {
62     /*If base is 1, print a stream of 1's*/
63     if(base == 1)
64     {
65         while(num-- != 0)
66             printf("%d", 1);
67         return;
68     }
69     else if(num == 0) //If no value left, return
70         return;
71
72     int remainder = num%base; //remainder calculated
73     to_base_n(num/base, base); //recursive call
74
75     /*If remainder is larger than 10, letters are used instead of digits*/
76     printf("%c", remainder < 10 ? remainder+48 : remainder+55);
77 }

```

‘emptyInputBuffer’ function in Task1c_source.c

The implementation for the ‘emptyInputBuffer’ function is identical to that found in task 1a. Basically, the loop in the function loops until the line character (present due to the user pressing the ENTER key) is found, indicating that all characters in the input buffer were cleared.

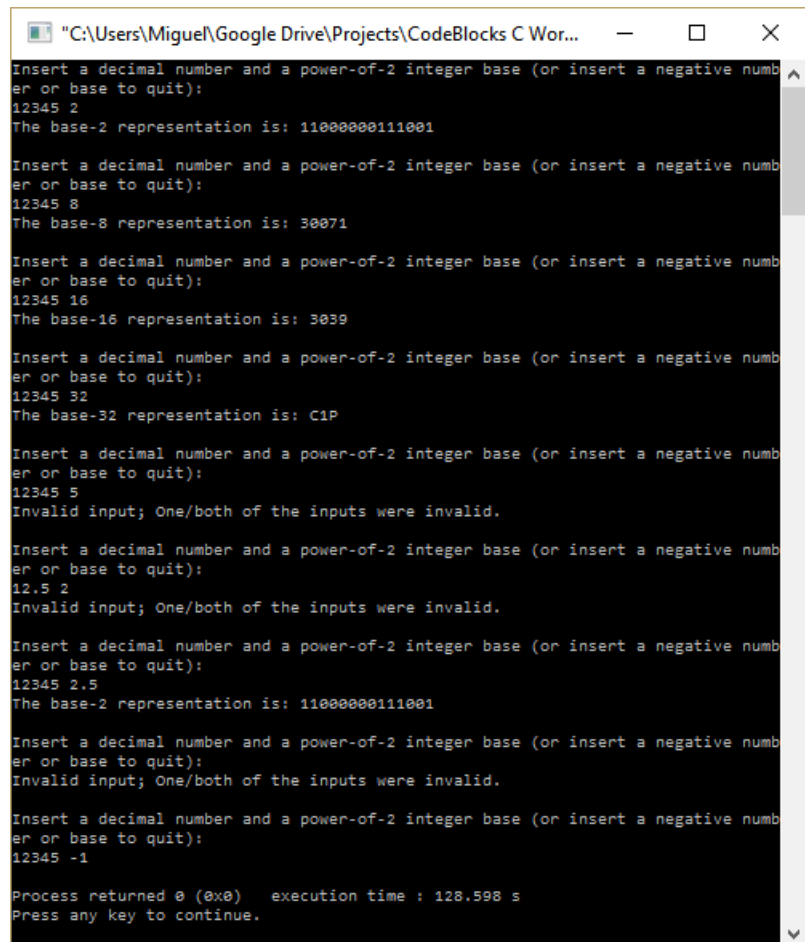
```

79 /*This function clears the input buffer*/
80 void emptyInputBuffer(void)
81 {
82     while(getchar() != '\n');
83 }

```

Testing

For testing purposes, one sample output will be shown. The inputs given to the program can be seen from the screenshot itself found below. Adjacent to the screenshot are indicators for what the inputs accepted by 'scanf' were in each case. It is interesting to observe the results for invalid inputs. When the decimal number input was 12.5, the input was not accepted, but when the base was 2.5, the integer part 2 of the base was accepted and the 0.5 was left in the input buffer. In fact, the next output is triggered by the 'scanf' scanning the remaining 0.5 as the decimal number and realizing that this value is not an integer that it was expecting, but a floating-point number. This occurrence corresponds to the limitation presented at the start of this subtask. The final input includes a negative base, so the program quits.



Input	Output
12345 2	Decimal Number: 12345 Base: 2
12345 8	Decimal Number: 12345 Base: 8
12345 16	Decimal Number: 12345 Base: 16
12345 32	Decimal Number: 12345 Base: 32
12345 5	Decimal Number: 12345 Base: 5
12.5 2	Decimal Number: 12.5 Base: 2
12345 2.5	Decimal Number: 12345 Base: 2.5
0.5 /	Decimal Number: 0.5 Base: /
12345 -1	Decimal Number: 12345 Base: -1

Figure 4: Task 1c sample output

Task 1d

Introduction

The code for this task consists of two functions within a single source file. Below is a list showing the layout of the functions within the respective files.

- Task1d_source.c
 - int main(void)
 - void str_reverse(char[MAXLEN])

Header files, constants, and function prototypes in Task1d_source.c

To start with, the ‘stdio.h’ and ‘string.h’ header files were included. The ‘string.h’ header file provides a function ‘strlen’ which will be used to measure the input string’s length. A constant ‘MAXLEN’ was defined as 100. It signifies the maximum length of the string to be reversed. A hundred characters maximum for the input string was assumed to be enough. A function ‘str_reverse’ which will serve as the string reverser was declared.

```
1 #include <stdio.h> //printf(), fgets()
2 #include <string.h> //strlen()
3
4 #define MAXLEN 100 //maximum input size
5
6 /* operation: reverses the string passed to the function */
7 /* preconditions: a character array is passed to the function */
8 /* postconditions: the character array is modified in a way */
9 /*                  that the characters inside it are reversed */
10 void str_reverse(char[MAXLEN]);
```

‘main’ function in Task1d_source.c

In the ‘main’ function, a character array ‘input’ of size ‘MAXLEN’ was declared store the user’s input. Next is the first output for the user followed by the code for string input. The input function used was ‘fgets’ since it is a safer approach towards string input. The ‘MAXLEN’ argument for ‘fgets’ indicates that the maximum amount of characters read from ‘stdin’ will be ‘MAXLEN’, meaning that the array size limit will not be exceeded and hence memory found after the array cannot be overwritten by mistake. Since the ‘fgets’ function keeps the newline character introduced when the user presses ENTER, the element containing the newline is set to a null character. Hence, the string is shortened by one character. The input string is output to the user for easier comparison with the reversed string that is output later on.

Kindly proceed to the next page for the code corresponding to the above description.

```

12 int main(void) {
13     /*Declaration of variables*/
14     char input[MAXLEN]; //stores input string
15
16     /*User inputs a string*/
17     printf("Insert a string: ");
18     fgets(input, MAXLEN, stdin);
19     input[strlen(input)-1]='\0'; //removing the extra newline due to fgets
20     printf("Input:   \"%s\"\n", input);

```

For string reversal, the ‘input’ array is passed to the ‘str_reverse’ function. The string is modified by reference in the function, so the original string is lost. The resultant string is output to the user.

```

22     /*Reversing the string*/
23     str_reverse(input);
24     printf("Reversed: \"%s\"", input);
25
26     return 0;
27 }

```

‘str_reverse’ function in Task1d_source.c

In the ‘str_reverse’ function implementation, the necessary variables were declared. The ‘strlen’ function is used on ‘string’ to obtain its length on the go. A for loop for string reversal loops through ‘string’ from the first character up to the midpoint of the array indicated by ‘size/2’. Each of these elements is swapped with the corresponding opposite element going from ‘size-1’ down to the midpoint ‘size/2’ as ‘index’ increases. When the array size is odd, the middle character is not swapped with any other character.

Swapping takes place by storing one character in a temporary char variable, copying the second character to the first character’s location, and copying the first character (stored in the temporary char variable) to the second character’s location. Without the temporary character storage, the characters would overwrite each other.

Since the array was modified by reference, nothing needs to be returned from the function. The original array now stores the reversed string.

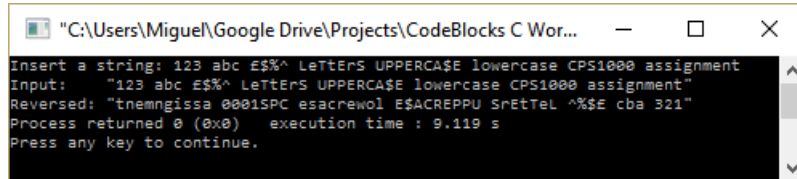
```

29 void str_reverse(char string[MAXLEN])
30 {
31     /*Declaration of variables*/
32     int size = strlen(string); //stores the string length
33     int index; //stores loop counter/index
34     char temp; //stores a char while swapping
35
36     /*String is reversed*/
37     for(index = 0; index < size/2; index++){
38         temp = string[index];
39         string[index] = string[size-1-index];
40         string[size-1-index] = temp;
41     }
42 }

```

Testing

For testing purposes, one sample output will be shown. The input given to the program can be seen from the screenshot itself found below.

A screenshot of a Windows command prompt window. The title bar reads "C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...". The window contains the following text:

```
Insert a string: 123 abc E$%^ LeTtErS UPPERCASE lowercase CPS1000 assignment
Input:  "123 abc E$%^ LeTtErS UPPERCASE lowercase CPS1000 assignment"
Reversed: "tnemngissa 00015PC esacrewol E$ACREPPU SreTtEl ^%$E cba 321"
Process returned 0 (0x0)   execution time : 9.119 s
Press any key to continue.
```

Figure 5: Task 1d sample output

Task 1e

Introduction

The code for this task consists of four functions within a single source file. Below is a list showing the layout of the functions within the respective files.

- Task1e_source.c
 - int main(void)
 - void randomArray(const int size, int nums[size])
 - void naive_sort(const int size, int nums[size])
 - void printArray(const int size, const int nums[size])

Header files and function prototypes in Task1e_source.c

The ‘stdio.h’, ‘stdlib.h’, and ‘time.h’ header files were included. Header file ‘stdlib.h’ provides functions ‘srand’ and ‘rand’ for generating random numbers. The ‘time.h’ header file will provide a function ‘time’ to obtain the current time which is used as a seed for the random number generator.

Three functions ‘randomArray’, ‘naive_sort’, and ‘printArray’ were now declared. The ‘randomArray’ function will generate a set of random integers, the ‘naive_sort’ will sort an array’s values using an insertion sort algorithm, and ‘printArray’ will be used to print the contents of an array.

```
1 #include <stdio.h> //printf(), put()
2 #include <stdlib.h> //srand(), rand()
3 #include <time.h> //time()
4
5 /* operation: fills the 'nums' array with random integers. */
6 /* preconditions: an integer array and a variable which shows */
7 /* its size are passed to the function. */
8 /* postconditions: the 'nums' array contains a set of random */
9 /* integers generated by the function. */
10 void randomArray(const int size, int nums[size]);
11
12 /* operation: sorts the integers in the 'nums' array using */
13 /* an insertion sort algorithm. */
14 /* preconditions: an integer array containing a set of poss- */
15 /* ibly unsorted integers and the array's size */
16 /* are passed to the function. */
17 /* postconditions: the integer array will now contain a set */
18 /* of sorted integer values. */
19 void naive_sort(const int size, int nums[size]);
20
21 /* operation: outputs the integers found in an integer array. */
22 /* preconditions: an integer array and its size are passed to */
23 /* the function. */
24 /* postconditions: the integers in the array are output to */
25 /* the user in an easy to read form. */
26 void printArray(const int size, const int nums[size]);
```

‘main’ function in Task1e_source.c

In the ‘main’ function, a random number generator is initialized using the ‘srand’ function. ”The pseudo-random number generator is initialized using the argument passed as seed [2].” Also, ”For every different seed value used in a call to srand, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to rand [2].” Since the ‘srand’ function takes an integer as an argument, the seed is hence an integer value used by the number generator algorithm to decide which random number to start with.

A constant ‘ARRAYSIZE’ was defined to be a random integer value from 1 to 30 using the ‘rand’ function which returns a random integer that can be scaled down using the modulus operator. The ‘+1’ changes the random integer range from 0-29 to 1-30, which was seen as a suitable range of array sizes without having too many elements. An array ‘numbers’ was declared with the size specified by the randomly-generated ‘ARRAYSIZE’ during runtime.

```
28 int main(void)
29 {
30     /*Declaration and initialization*/
31     srand(time(NULL)); //number generator seeded
32     const int ARRAYSIZE = (rand()%30) + 1; //random size between 1 and 30
33     int numbers[ARRAYSIZE]; //holds numbers to be sorted
```

The ‘numbers’ array is now filled with random integer values by passing it to the ‘randomArray’ function along with its size ‘ARRAYSIZE’.

```
35     /*Array given random integers*/
36     randomArray(ARRAYSIZE, numbers); //array given random integers
```

The array’s values now have to be sorted. The array is passed into ‘printArray’ along with its size, which outputs the array’s values. ‘printArray’ is invoked before and after the sort for easier comparison between the unsorted array and the sorted array. The array is then passed to the ‘naive_sort’ function which will sort the integers in the array.

```
38     puts("Original: ");
39     printArray(ARRAYSIZE, numbers); //output array
40     naive_sort(ARRAYSIZE, numbers); //sort array
41     puts("Sorted: ");
42     printArray(ARRAYSIZE, numbers); //output array
43
44     return 0;
45 }
```

‘randomArray’ function in Task1e_source.c

In the ‘randomArray’ function implementation, a loop traverses the array and uses the ‘rand’ function to set the array’s elements to random integers in the range 0-999 (indicated by the `rand()%1000` operation, which leaves a minimum remainder of 0 and maximum remainder of 999).

```
47 void randomArray(const int size, int nums[size])
48 {
49     int index; //stores array index in the loop
50
51     /*Each element is given a random integer value*/
52     for(index=0; index < size; index++)
53         nums[index] = rand() % 1000; //random integer between 0 and 999
54 }
```

‘naive_sort’ function in Task1e_source.c

The ‘naive_sort’ function implementation uses an insertion sort algorithm.

The insertion sort algorithm repeatedly traverses the list of numbers, sorting one number with each traversal. The part of the array consisting of sorted numbers is considered to be a sub-list of the whole list of numbers. The algorithm starts by assuming that the first item in the list is sorted, meaning that the sub-list initially contains the first item. In each traversal, the first item found after the sub-list is compared with the sub-list. Any number in the sub-list that is larger than this item is shifted forwards. The vacated spot is then occupied by the item in consideration and the sub-list’s size is increased by 1. In general, after the n^{th} traversal of the list, the sub-list will contain $n + 1$ items. The process is repeated until the last item in the list has been considered. [3]

In the function, the necessary variables were declared. A for loop starts from the second element of the array at index 1 (since the first element is initially considered to be sorted). In each iteration, the current element and its location are copied to ‘tempInt’ and ‘index2’ respectively to back up the value at ‘index1’ and to leave ‘index1’ itself unchanged.

Another loop (a while loop) loops through the elements starting from the one indicated by ‘index1’ downwards to the first element. Any elements found to the left (smaller index) of ‘index1’ which are larger than ‘tempInt’ are shifted one space to the right (larger index). The inner loop terminates when the index reaches zero or when a value smaller than the one indicated by ‘index1’ is found, indicating that no further shifting is required. The vacated element is then taken up by ‘tempInt’. Hence, the section of the array from the first element up to ‘index1’ is now sorted. During the lifetime of the main loop ‘index1’ is set to every element in the array from left to right (excluding index 0) to sort the array.

Kindly proceed to the next page for the code corresponding to the above description.

```
56 void naive_sort(const int size, int nums[size])
57 {
58     /*Declaration of variables*/
59     int index1;    //outer loop index
60     int index2;    //inner loop index
61     int tempInt;    //stores value of index1'th element
62
63     /*Main sort loop*/
64     for(index1=1; index1 < size; index1++)
65     {
66         tempInt = nums[index1]; //current element stored
67         index2 = index1;        //index copied to leave index1 unchanged
68
69         /*Inner sort loop*/
70         while(index2 > 0 && nums[index2-1] > tempInt)
71         {
72             nums[index2] = nums[index2-1]; //element shifted to the right
73             index2--;
74         }
75         nums[index2]=tempInt; //vacated element taken up by tempInt
76     }
77 }
```

‘printArray’ function in Task1e_source.c

Next was the implementation of the ‘printArray’ function. This function takes an integer array and its size as arguments and outputs all the array’s values to the user. The loop iterates through all of the elements, printing each one individually. The end result is a list of integers separated by commas and grouped by curly brackets for better presentation.

```
79 void printArray(const int size, const int nums[size])
80 {
81     /*Declaration of variable*/
82     int index; //stores index for loop
83
84     /*Output*/
85     printf("{");
86     for(index=0; index < size-1; index++)
87         printf("%d, ", nums[index]);
88     printf("%d}\n", nums[size-1]);
89 }
```

Testing

For testing purposes, two sample outputs will be shown. The arrays of numbers to be sorted can be seen in the screenshots themselves. An interesting fact to note is that the second sample output contains two identical numbers (two 275’s) in the array to be sorted. These two numbers were successfully placed adjacent to each other. Kindly proceed to the next page for the sample outputs.

```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...  
Original:  
{642, 604, 699, 780, 515, 38, 824, 484, 360, 685, 153, 713, 625, 449, 444, 561,  
775}  
Sorted:  
{38, 153, 360, 444, 449, 484, 515, 561, 604, 625, 642, 685, 699, 713, 775, 780,  
824}  
  
Process returned 0 (0x0)   execution time : 0.012 s  
Press any key to continue.
```

Figure 6: Task 1e sample output 1

```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...  
Original:  
{638, 951, 604, 275, 389, 308, 377, 160, 275, 628, 148, 829, 663, 188, 786, 176,  
506, 230, 191, 928, 192, 426, 842, 96, 894, 541, 750, 654, 581}  
Sorted:  
{96, 148, 160, 176, 188, 191, 192, 230, 275, 275, 308, 377, 389, 426, 506, 541,  
581, 604, 628, 638, 654, 663, 750, 786, 829, 842, 894, 928, 951}  
  
Process returned 0 (0x0)   execution time : 0.014 s  
Press any key to continue.
```

Figure 7: Task 1e sample output 2

Task 1f

Introduction

The code for this task consists of four functions within a single source file. Since this task's source code is similar to that of the previous task, the majority of the detail will be excluded to avoid repetition. The major difference is the sorting function. Below is a list showing the layout of the functions within the respective files.

- Task1f_source.c
 - int main(void)
 - void randomArray(const int size, int nums[size])
 - void smarter_sort(int nums[], const int start, const int end)
 - void printArray(const int size, const int nums[size])

Header files and function prototypes in Task1f_source.c

At the start, a new function 'smarter_sort' was declared to replace the 'naive_sort' function found in the previous task.

```
1 #include <stdio.h> //printf()
2 #include <stdlib.h> //srand(), rand()
3 #include <time.h> //time()
4
5 /* operation: fills the 'nums' array with random integers. */
6 /* preconditions: an integer array and a variable which shows */
7 /* its size are passed to the function. */
8 /* postconditions: the 'nums' array contains a set of random */
9 /* integers generated by the function. */
10 void randomArray(const int size, int nums[size]);
11
12 /* operation: sorts the integers in the 'nums' array using */
13 /* a quick sort algorithm. */
14 /* preconditions: an integer array containing a set of poss- */
15 /* ibly unsorted integers and the array's size */
16 /* are passed to the function. */
17 /* postconditions: the integer array will now contain a set */
18 /* of sorted integer values. */
19 void smarter_sort(int nums[], const int start, const int end);
20
21 /* operation: outputs the integers found in an integer array */
22 /* preconditions: an integer array and its size are passed to */
23 /* the function. */
24 /* postconditions: the integers in the array are output to */
25 /* the user in an easy to read form. */
26 void printArray(const int size, const int nums[size]);
```

‘main’ function in Task1f_source.c

In the ‘main’ function, the ‘smarter_sort’ function is called instead of the ‘naive_sort’ function at line 40. The values ‘0’ and ‘ARRAYSIZE-1’ passed to the function will indicate to the function that the whole array (first to last element) is to be considered at first.

```
28 int main(void)
29 {
30     /*Declaration and initialization*/
31     srand(time(NULL)); //number generator seeded
32     const int ARRAYSIZE = (rand()%30) + 5; //random size between 1 and 30
33     int numbers[ARRAYSIZE]; //array of numbers to be sorted
34
35     /*Array given random integers*/
36     randomArray(ARRAYSIZE, numbers); //array given random integers
37
38     printf("Original:\n");
39     printArray(ARRAYSIZE, numbers); //output array
40     smarter_sort(numbers, 0, ARRAYSIZE-1); //sort array
41     printf("Sorted:\n");
42     printArray(ARRAYSIZE, numbers); //output array
43
44     return 0;
45 }
```

‘randomArray’ and ‘printArray’ functions in Task1f_source.c

The ‘randomArray’ and ‘printArray’ functions are identical to the ones implemented in the previous task.

‘smarter_sort’ function in Task1f_source.c

The quick-sort algorithm will now be discussed.

The quicksort algorithm works by applying an procedure to pairs of sub-lists where each sub-list is then further split into a pair of sub-lists, until the sub-lists are partitioned enough times for all of the sub-lists to have only 1 number, at which point the list is considered to be sorted. The main procedure in the quicksort algorithm involves ordering the numbers in a section of the array based on a specific reference number, which may be called the pivot. The pivot can be selected to be any of the numbers in the section of the list being considered and is commonly chosen to be the value of the middle element or the first element. In an ordering, the numbers to the left of the pivot must be smaller than the pivot value, while the numbers to the right must be larger. Once the list is ordered in this manner, the procedure is performed on the sections of the array to the left and to the right of the reference number, given that these contain at least two numbers. [4,5]

The actual implementation of the ‘smarter_sort’ function will now be discussed. This function uses a recursive implementation approach to the quick-sort algorithm. The ‘start’ and ‘end’ values passed in are set to ‘left’ and ‘right’ respectively. Values ‘start’ and ‘end’ are indices that indicate the range of the array that is to be operated on in a specific call of the function. A value for the ‘pivot’ variable is calculated by taking the element found at the mid-point of the range from ‘start’ to ‘end’. This range gets smaller and smaller with each call of the function.

The main loop now starts with the condition that ‘left’ cannot be larger than ‘right’. The reason for this is more clear when the loop has iterated at least once. Within the loop, two further loops are encountered. The first loop increments the ‘left’ index until a number larger than or equal to the pivot is found. The second loop decrements the ‘right’ index until an element smaller than or equal to the pivot is found. The point is to find two values which are out of order (one smaller or equal and one larger or equal) with respect to the value of ‘pivot’.

```

56 void smarter_sort(int nums[], const int start, const int end)
57 {
58     /*Declaration and initialization of variables*/
59     int left = start; //index of element smaller than pivot
60     int right = end; //index of element larger than pivot
61     int pivot = nums[(end+start)/2]; //reference number
62     int temp; //used when swapping two elements
63
64     /*Main loop*/
65     while(left <= right)
66     {
67         /*Move to the right until value larger */
68         /*than or equal to pivot is found. */
69         while(nums[left] < pivot)
70             left++;
71         /*Move to the left until value smaller */
72         /*than or equal to pivot is found. */
73         while(nums[right] > pivot)
74             right--;

```

Next, a condition checks if ‘left’ has exceeded ‘right’. If yes, then this indicates that a pair of values, one larger than the pivot and another smaller than the pivot, was not found and the ‘left’ and ‘right’ indices overlapped. This means that for the range of the array from ‘start’ till ‘end’, any value to the left of the pivot is smaller than the pivot and any value to the right of the pivot is larger than the pivot.

```

76         /*If 'left' exceeds 'right', then this */
77         /*section of the array has been ordered*/
78         if(left > right)
79             break;

```

If ‘left’ has not exceeded ‘right’, then the values at these two indices swapped, making use of the ‘temp’ integer. Since the values at ‘left’ and ‘right’ were smaller (or equal) and larger (or equal) to the pivot, respectively, then this swap effectively orders these two values with respect to the pivot.

In the cases where the value at 'left' or that at 'right' is equal to the pivot, the pivot will be one of the values getting swapped. When the pivot gets swapped, it is effectively creating more space for values smaller than it or larger than it, depending on where space is needed. If the other value in the swap is larger than the pivot (and hence to the left of it), then the pivot is being moved to the left, making space for more values that are larger than it. This also applies oppositely when the other value in the swap is smaller than the pivot.

After the swap, the indices 'left' and 'right' are incremented and decremented, respectively, so that they will now store the index of the next numbers to be compared in an effort to find the next pair of values that are out of order. The main loop's condition is satisfied only if the latest change to the indices did not cause them to overlap. Overlapping of the indices would mean that no more values out of order are present in the range of the list of numbers being considered and hence no more swaps are needed, so the loop is not terminated.

```
81      /*Swapping numbers at 'left' and 'right'*/
82      temp = nums[left];
83      nums[left] = nums[right];
84      nums[right] = temp;
85
86      /*Moving on to the next elements */
87      /*in the list to be considered. */
88      left++;
89      right--;
90  }
```

Once the loop is broken, two conditions decide what is to be done next. The first condition checks whether at least two elements are found to the left of the 'left' index, in which case the 'smarter_sort' is applied to the range of the array starting from 'start' up to the element before 'left'. The second condition checks whether at least one element is found to the right of the 'left' index, in which case a recursive call is made with the range of the array starting from 'left' itself up to the end of this section, 'end'. The minimum size of a 'section' of an array is two elements, since array sections of one or zero elements are considered to be ordered within themselves.

As such, the whole array will go through the function again, but only sections of it will be considered in the value ordering process. The ranges will divide into further ranges repeatedly. By the end of the ordering process, all of the elements will have been sorted.

```
91      /*If more than one value to the left of 'left'*/
92      if((left-1) > start)
93          smarter_sort(nums, start, left-1); //sort left part
94      /*If at least one value to the right of 'left'*/
95      if(left < end)
96          smarter_sort(nums, left, end);      //sort right part
97  }
```

Testing

For testing purposes, two sample outputs will be shown. The arrays of numbers to be sorted can be seen in the screenshots themselves. An interesting fact to note is that the second sample output contains two identical numbers (two 435's) in the array to be sorted. These two numbers were successfully placed adjacent to each other.

```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...  
Original:  
{555, 304, 738, 246, 206, 54, 139, 110, 982, 403, 754, 345, 940, 793, 646, 16, 8  
, 283, 864, 965, 34, 152, 905, 394, 519, 704, 163, 680, 270, 126, 666, 171, 612,  
415}  
Sorted:  
{0, 16, 34, 54, 110, 126, 139, 152, 163, 171, 206, 246, 270, 283, 304, 345, 394,  
403, 415, 519, 555, 612, 646, 666, 680, 704, 738, 754, 793, 864, 905, 940, 965,  
982}  
Process returned 0 (0x0) execution time : 0.020 s  
Press any key to continue.
```

Figure 8: Task 1f sample output 1

```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...  
Original:  
{508, 975, 998, 845, 435, 248, 120, 213, 435, 487, 747, 164, 433, 744, 42}  
Sorted:  
{42, 120, 164, 213, 248, 433, 435, 435, 487, 508, 744, 747, 845, 975, 998}  
Process returned 0 (0x0) execution time : 0.024 s  
Press any key to continue.
```

Figure 9: Task 1f sample output 2

Task 1g

Introduction

The code for this task consists of six functions spread out across three source file. For this task, the two major functions ‘naive_sort’ and ‘smarter_sort’ implemented in the previous tasks were copied to individual source files and included in the main source file using a header file for each sort type. At the top of each of the two sort function source files, the respective header file containing the respective sorting function prototype was included. Below is a list showing the layout of the functions within the respective files.

- Task1g_source.c
 - int main(void)
 - void randomArray(const int size, int nums[size])
 - void printArray(const int size, const int nums[size])
 - void tester(const int size, const int nums[size])
- Insertion.c
 - void naive_sort(const int size, int nums[size])
- Quick.c
 - void smarter_sort(int nums[], const int start, const int end)
- Insertion.h
- Quick.h

Insertion.c

```
1 #include "Insertion.h"
2
3 void naive_sort(const int size, int nums[size])
4 {
5     /*Declaration of variables*/
6     int index1;    //outer loop index
7     int index2;    //inner loop index
8     int tempInt;   //stores value of index1'th element
9
10    /*Main sort loop*/
11    for(index1=1; index1 < size; index1++)
12    {
13        tempInt = nums[index1]; //current element stored
14        index2 = index1;
15
16        /*Inner sort loop*/
17        while(index2 > 0 && nums[index2-1] > tempInt)
18        {
19            nums[index2] = nums[index2-1]; //element shifted to the right
```

```
20         index2--;  
21     }  
22     nums[index2]=tempInt; //vacated element taken up by tempInt  
23 }  
24 }
```

Quick.c

```
1 #include "Quick.h"  
2  
3 void smarter_sort(int nums[], const int start, const int end)  
4 {  
5     /*Declaration and initialization of variables*/  
6     int left = start; //index of element smaller than pivot  
7     int right = end; //index of element larger than pivot  
8     int pivot = nums[(end+start)/2]; //reference number  
9     int temp; //used when swapping two elements  
10  
11     /*Main loop*/  
12     while(left <= right)  
13     {  
14         /*Move to the right until value larger */  
15         /*than or equal to pivot is found. */  
16         while(nums[left] < pivot)  
17             left++;  
18         /*Move to the left until value smaller */  
19         /*than or equal to pivot is found. */  
20         while(nums[right] > pivot)  
21             right--;  
22  
23         /*If 'left' exceeds 'right', then this */  
24         /*section of the array has been ordered*/  
25         if(left > right)  
26             break;  
27  
28         /*Swapping numbers at 'left' and 'right'*/  
29         temp = nums[left];  
30         nums[left] = nums[right];  
31         nums[right] = temp;  
32  
33         /*Moving on to the next elements */  
34         /*in the list to be considered. */  
35         left++;  
36         right--;  
37     }  
38     /*If more than one value to the left of 'left'*/  
39     if((left-1) > start)  
40         smarter_sort(nums, start, left-1); //sort left part  
41     /*If at least one value to the right of 'left'*/  
42     if(left < end)  
43         smarter_sort(nums, left, end); //sort right part  
44 }
```

Insertion.h

```

1 #ifndef INSERTION_H_INCLUDED
2 #define INSERTION_H_INCLUDED
3
4 /* operation: sorts the integers in the 'nums' array using      */
5 /*              an insertion sort algorithm.                    */
6 /* preconditions: an integer array containing a set of poss-    */
7 /*              ibly unsorted integers and the array's size    */
8 /*              are passed to the function.                    */
9 /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */
11 void naive_sort(const int size, int nums[size]);
12
13 #endif //INSERTION_H_INCLUDED

```

Quick.h

```

1 #ifndef QUICK_H_INCLUDED
2 #define QUICK_H_INCLUDED
3
4 /* operation: sorts the integers in the 'nums' array using      */
5 /*              a quick sort algorithm.                          */
6 /* preconditions: an integer array containing a set of poss-    */
7 /*              ibly unsorted integers and the array's size    */
8 /*              are passed to the function.                    */
9 /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */
11 void smarter_sort(int nums[], const int start, const int end);
12
13 #endif //QUICK_H_INCLUDED

```

Header files and function prototypes in Task1g_source.c

Many features of the main source file are copies from previous tasks, so extra detail will be excluded. The required standard header files along with the quick-sort and insertion-sort header files were all included. Three functions were now declared. The new 'tester' function will check that the integers in the array passed to the function are sorted in an ascending order to verify that the sorting functions successfully sorted the values.

```

1 #include <stdio.h> //printf(), puts(), getchar()
2 #include <stdlib.h> //srand(), rand()
3 #include <time.h> //time()
4
5 #include "Quick.h"
6 #include "Insertion.h"
7
8 /* operation: fills the 'nums' array with random integers.      */
9 /* preconditions: an integer array and a variable which shows   */
10 /*              its size are passed to the function.            */
11 /* postconditions: the 'nums' array contains a set of random   */
12 /*              integers generated by the function.             */
13 void randomArray(const int size, int nums[size]);
14

```



```

15 /* operation: outputs the integers found in an integer array. */
16 /* preconditions: an integer array and its size are passed to */
17 /* the function. */
18 /* postconditions: the integers in the array are output to */
19 /* the user in an easy to read form. */
20 void printArray(const int size, const int nums[size]);
21
22 /* operation: checks that the integers in an array are sorted */
23 /* in an ascending order. */
24 /* preconditions: an integer array and its size are passed to */
25 /* the function. */
26 /* postconditions: an output indicates whether the sort was */
27 /* verified or found to be incorrect. */
28 void tester(const int size, const int nums[size]);

```

‘main’ function in Task1g_source.c

The ‘main’ function consists of a do-while loop which will loop until the character entered at the end is a ‘q’, meaning that the user wants to quit. In each iteration of the loop, the random number generated is given a seed based on the time. Following this, the ‘ARRAYSIZE’ value is calculated and ‘numbers’ array is declared with size ‘ARRAYSIZE’.

For each type of sort, the array is given a set of random numbers by the ‘randomArray’ function and is sorted using the respective sorting function. Outputs of the arrays are placed before and after the sorts for easier comparison between the original and sorted arrays. Additionally, for each type of sort, the sorted array is passed to the ‘tester’ function along with its size to check whether the values were indeed sorted. The function outputs an indication of whether the arrays are sorted.

At the end of each loop iteration, the user is asked to press ENTER to perform another pair of sorts or type in ‘q’ to quit. The post-tested loop’s condition reads a character and terminates the loop if the character is a ‘q’. Otherwise, a newline character is read and the loop condition is satisfied. This functionality assumes that the user will not enter extra characters. For each extra character entered, the loop condition is repeatedly satisfied until the last character, the newline, is read. Functions to clear the input buffer were avoided to avoid unnecessary complications.

```

30 int main(void)
31 {
32     do
33     {
34         /*Declaration and initialization*/
35         srand(time(NULL)); //number generator seeded
36         const int ARRAYSIZE = (rand()%30)+1; //random size from 1 to 30
37         int numbers[ARRAYSIZE]; //array to be sorted
38
39         /*Insertion sort*/
40         randomArray(ARRAYSIZE, numbers); //array given random integers
41         printf("Original before insertion sort:\n");
42         printArray(ARRAYSIZE, numbers); //print array
43         naive_sort(ARRAYSIZE, numbers); //sort array

```

```

44     printf("Sorted by insertion sort:\n");
45     printArray(ARRAYSIZE, numbers);    //print array
46     tester(ARRAYSIZE, numbers);        //verify sort
47
48     printf("\n");
49
50     /*Quick sort*/
51     randomArray(ARRAYSIZE, numbers);    //array given random
52     integers
53     printf("Original before quick sort:\n");
54     printArray(ARRAYSIZE, numbers);    //print array
55     smarter_sort(numbers, 0, ARRAYSIZE-1); //sort array
56     printf("Sorted by quick sort:\n");
57     printArray(ARRAYSIZE, numbers);    //print array
58     tester(ARRAYSIZE, numbers);        //verify sort
59
60     puts("\nPress ENTER to do another pair of sorts or type 'q' to
61     quit...");
62 }
63 while(getchar() != 'q');
64 return 0;
65 }

```

‘randomArray’ and ‘printArray’ functions in Task1g_source.c

The ‘randomArray’ and ‘printArray’ functions are identical to the ones implemented in the previous tasks. The new function is the ‘tester’ function.

‘tester’ function in Task1g_source.c

The ‘tester’ function simply goes through the elements of the array passed in, checking whether any number in the array is followed by a smaller number, in which case the two values are not in order and the array is hence not sorted. An output to the user indicates this and the function returns. Otherwise, an output at the end indicates that the array is sorted.

```

87 void tester(const int size, const int nums[size])
88 {
89     /*Declaration of variable*/
90     int index; //stores array counter/index
91
92     /*Tester loop*/
93     for(index=0; index < size-1; index++)
94     {
95         /*Check if a number is followed by a smaller one*/
96         if(nums[index] > nums[index+1])
97         {
98             printf("Sort incorrect.\n");
99             return;
100         }
101     }
102     /*Since no number was followed by a smaller one */
103     /*the array is hence sorted and was verified. */
104     printf("Sort verified.\n");
105 }

```

Testing

For testing purposes, one sample output will be shown. The arrays of numbers to be sorted can be seen in the screenshots themselves found below. Since the program allows for multiple subsequent sorts, the ENTER key was pressed for the second pair of sorts. The second input given was a 'q' to quit from the program.

```

"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Original before insertion sort:
{139, 835, 63, 687, 184, 845, 236, 14, 155, 330, 857, 369, 495, 362, 412, 319, 7
51, 963, 696, 544, 996, 937, 70, 698, 985, 269, 465, 719, 844, 522}
Sorted by insertion sort:
{14, 63, 70, 139, 155, 184, 236, 269, 319, 330, 362, 369, 412, 465, 495, 522, 54
4, 687, 696, 698, 719, 751, 835, 844, 845, 857, 937, 963, 985, 996}
Sort verified.

Original before quick sort:
{430, 956, 328, 795, 339, 541, 569, 24, 391, 352, 600, 59, 979, 455, 568, 174, 5
76, 538, 80, 72, 665, 452, 370, 160, 697, 52, 939, 184, 520, 776}
Sorted by quick sort:
{24, 52, 59, 72, 80, 160, 174, 184, 328, 339, 352, 370, 391, 430, 452, 455, 520,
538, 541, 568, 569, 576, 600, 665, 697, 776, 795, 939, 956, 979}
Sort verified.

Press ENTER to do another pair of sorts or type 'q' to quit...

Original before insertion sort:
{570, 772, 256, 215, 756, 89, 658, 778, 358, 999, 253, 11, 340, 484, 615, 305, 7
38, 440, 42, 979, 252, 231, 665, 52, 210, 438, 674, 351, 470}
Sorted by insertion sort:
{11, 42, 52, 89, 210, 215, 231, 252, 253, 256, 305, 340, 351, 358, 438, 440, 470
, 484, 570, 615, 658, 665, 674, 738, 756, 772, 778, 979, 999}
Sort verified.

Original before quick sort:
{762, 821, 263, 424, 526, 188, 275, 628, 468, 331, 888, 154, 190, 697, 214, 359,
772, 351, 999, 834, 496, 593, 62, 794, 769, 587, 373, 30, 589}
Sorted by quick sort:
{30, 62, 154, 188, 190, 214, 263, 275, 331, 351, 359, 373, 424, 468, 496, 526, 5
87, 589, 593, 628, 697, 762, 769, 772, 794, 821, 834, 888, 999}
Sort verified.

Press ENTER to do another pair of sorts or type 'q' to quit...
q

Process returned 0 (0x0)   execution time : 13.246 s
Press any key to continue.

```

Figure 10: Task 1g sample output

Task 1h

Introduction

The code for this task consists of six functions spread out across three source file. The code and files in this task are similar to the previous task but there were some changes the main source file. Below is a list showing the layout of the functions within the respective files.

- Task1h_source.c
 - int main(void)
 - void randomArray(const int size, int nums[size])
 - void tester(const int size, const int nums[size])
 - void calcTime(const clock_t start)
- Insertion.c
 - void naive_sort(const int size, int nums[size])
- Quick.c
 - void smarter_sort(int nums[], const int start, const int end)
- Insertion.h
- Quick.h

New function prototype in Task1h_source.c

A new function ‘calcTime’ was declared. This function will calculate and output the time passed from its starting time parameter and will be used to time the sorting algorithms.

```
22 /* operation: calculates and outputs the time passed from a */
23 /*           starting time indicated by the argument and the */
24 /*           current time. */
25 /* preconditions: a pre-calculated starting time is passed to */
26 /*           the function. */
27 /* postconditions: an output indicates the time passed from */
28 /*           'start' till the current time. */
29 void calcTime(const clock_t start);
```

‘main’ function in Task1h_source.c

The ‘main’ function is similar to that of the previous task but sees some changes. A new variable ‘start’ will hold the starting time for timing of the sorts, and a new constants array of array sizes will hold five sizes for five tests. The ‘clock_t’ type is an “Alias of a fundamental arithmetic type

capable of representing clock tick counts [6].” It is hence suitable for timing processes.

A for loop loops through indices 0 to 4. For each index, a pointer to an integer ‘numbers’ is declared and memory is allocated to it using ‘malloc’. The memory allocation function ‘malloc’ is passed the standard size of an integer ‘sizeof(int)’ multiplied by the amount of numbers that the pointer will be needed to access, which is the corresponding array size in the ‘SIZE’ array. The pointer will from this point onwards be considered to be an array of integers using array notation instead of pointer notation.

For each type of sort, the array is given a random set of integers, then it is sorted and tested. What is new is that for each type of sort, the starting time is recorded in ‘start’ using the function ‘clock’ from header file ‘time.h’. The ‘clock’ function returns the “The number of clock ticks elapsed since an epoch related to the particular program execution [7].” After the sort, this time in clock ticks is passed to function ‘calcTime’ to calculate the time taken to sort the values in the array by calculating the difference between the time passed by then, with the time calculated before the sorting starts.

At the end of each loop iteration, unless this was the final pair of sorts (i.e. unless index is 4), the user is asked to press ENTER for the next pair of sorts. This gives the user some time to observe the timing and verification outputs of the latest pair of sorts before moving on. The user is not given the option to quit given that the amount of sorts to be performed is limited. Extra inserted characters up to the newline are dumped by the one-line loop. The memory allocated to the pointer at the start of the loop iteration is freed so that the pointer may be allocated the next amount of memory for the sorting of a larger array. This is done by passing the pointer into the ‘free’ function.

```

31 int main(void)
32 {
33     /*Declaration and initialization*/
34     clock_t start;           //holds the starting time for process timings
35     srand ( time(NULL) );   //random number generator seeded
36     const int SIZE[] = {10,100,1000,5000,10000};
37
38     int index;
39     for(index = 0; index < 5; index++)
40     {
41         /*Pointer to an integer is allocated memory for elements*/
42         int *numbers = malloc(SIZE[index]*sizeof(int));
43
44         /*Insertion sort*/
45         printf("Insertion sort with array of size %d;\n", SIZE[index]);
46         randomArray(SIZE[index], numbers); //array given random integers
47         start = clock();                   //record starting time
48         naive_sort(SIZE[index], numbers); //sort array
49         calcTime(start);                   //calculate time taken
50         tester(SIZE[index], numbers);     //verify sort
51
52         printf("\n");
53     }

```

```
54      /*Quick sort*/
55      printf("Quick sort with array of size %d;\n", SIZE[index]);
56      randomArray(SIZE[index], numbers); //array given random integers
57      start = clock();                  //record starting time
58      smarter_sort(numbers, 0, SIZE[index]-1); //sort array
59      calcTime(start);                  //calculate time taken
60      tester(SIZE[index], numbers);     //verify sort
61
62      if(index != 4)
63      {
64          puts("\nPress ENTER to do the next pair of sorts...");
65          while(getchar() != '\n'); //remove all extra characters
66      }
67
68      /*Memory allocated is freed*/
69      free(numbers);
70  }
71  return 0;
72 }
```

‘randomArray’, ‘printArray’, and ‘tester’ functions in Task1h_source.c

The ‘randomArray’, ‘printArray’, and ‘tester’ functions are identical to the ones implemented in the previous tasks.

‘calcTime’ function in Task1h_source.c

The new function is the ‘calcTime’ function. This function takes a clock_t type argument ‘start’ which indicates the starting point for timing. Another clock_t type variable ‘diff’ is declared and is set to the difference between ‘start’ and the time obtained using the ‘clock’ function.

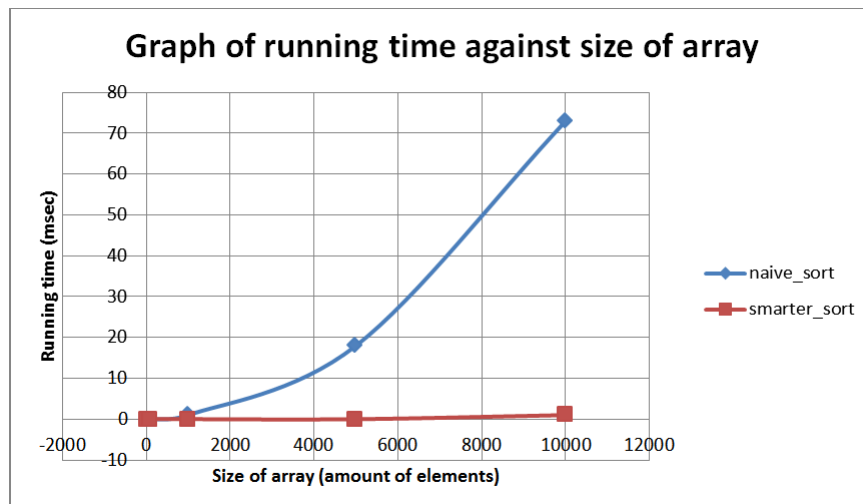
This difference is then output in milliseconds using a calculation. The value of ‘CLOCKS_PER_SEC’ from the ‘time.h’ header file is “the number of clock ticks per second [8].” Hence, if ‘diff’ is divided by this value, the time in seconds is obtained. This value is multiplied by 1000.0 to get the time taken in milliseconds since seconds are too long to measure short-duration processes. Despite this, most of the measurements resulted in zero milliseconds. This is either due to a problem in the way that the time is measured or due to the fact that the processes may be too fast to measure a reading.

Kindly proceed to the next page for the code corresponding to the above description.

```
103 void calcTime(const clock_t start)
104 {
105     /*Calculating the difference*/
106     clock_t diff = clock() - start;
107     /*Outputting the difference*/
108     printf("Time taken: %lf msec. ", diff*1000.0/CLOCKS_PER_SEC);
109 }
```

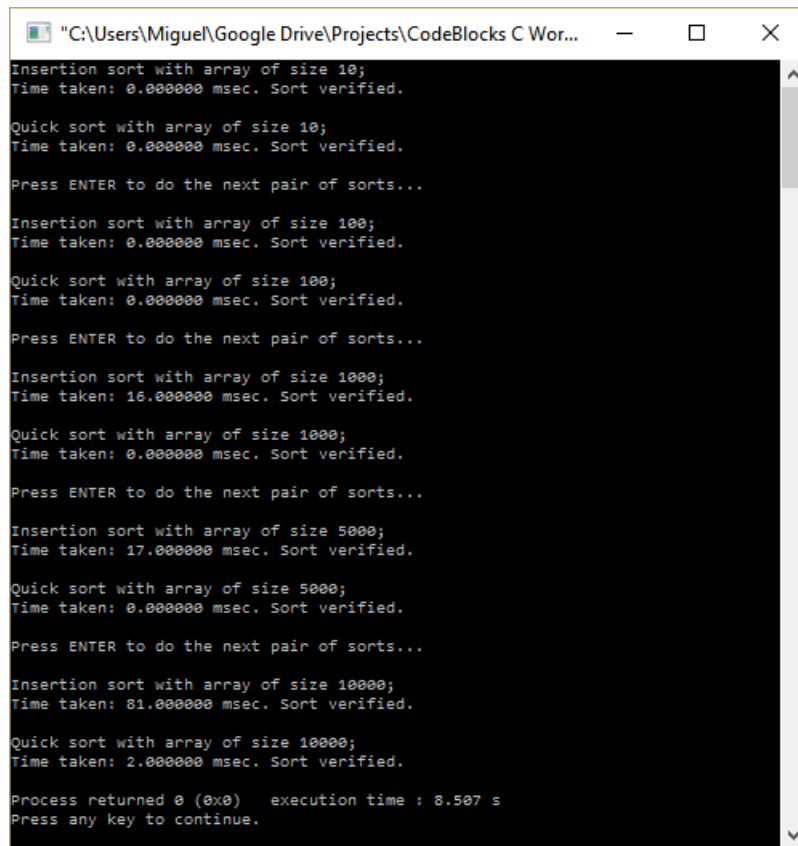
Graph for comparison of running times

A graph for the running time against the size of array for the two types of sorts is shown below. Despite the lack of detailed results, the graph still gives a clear indication that the 'smarter_sort' function has a much shorter running time than the 'naive_sort' function, especially when sorting arrays of large sizes.



Testing

For testing purposes, one sample output will be shown. The ENTER key was pressed after each sort to move on to the next pair of sorts. Note that the results in the sample output are not the same results used to plot the graph above. Each execution of the program gives slightly different (but relatively similar) results. Kindly proceed to the next page for the sample outputs.



```
"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Insertion sort with array of size 10;
Time taken: 0.000000 msec. Sort verified.

Quick sort with array of size 10;
Time taken: 0.000000 msec. Sort verified.

Press ENTER to do the next pair of sorts...

Insertion sort with array of size 100;
Time taken: 0.000000 msec. Sort verified.

Quick sort with array of size 100;
Time taken: 0.000000 msec. Sort verified.

Press ENTER to do the next pair of sorts...

Insertion sort with array of size 1000;
Time taken: 16.000000 msec. Sort verified.

Quick sort with array of size 1000;
Time taken: 0.000000 msec. Sort verified.

Press ENTER to do the next pair of sorts...

Insertion sort with array of size 5000;
Time taken: 17.000000 msec. Sort verified.

Quick sort with array of size 5000;
Time taken: 0.000000 msec. Sort verified.

Press ENTER to do the next pair of sorts...

Insertion sort with array of size 10000;
Time taken: 81.000000 msec. Sort verified.

Quick sort with array of size 10000;
Time taken: 2.000000 msec. Sort verified.

Process returned 0 (0x0)   execution time : 8.507 s
Press any key to continue.
```

Figure 11: Task 1h sample output

Task 1i

Introduction

The code for this task consists of two functions within one source file. Below is a list showing the layout of the functions within the respective files.

- Task1i_source.c
 - int main(void)
 - int catCount(const int strayCats[STOPS], const int stops)

Header files, constants, and function prototypes in Task1i_source.c

At the start, the ‘stdio.h’ header file was included and a symbolic constant ‘STOPS’ was defined. This constant will indicate the total amount of stops there will be, and hence the total counts of stray cats. A function ‘catCount’ was defined. This function will make use of the constant ‘STOPS’ and will count the total of stray cats observed through the bus windows during a trip.

```
1 #include <stdio.h> //printf()
2 #define STOPS 10 //maximum stops
3
4 /* operation: comes up with a total stray cats values from the*/
5 /* cat counts in the array passed in. */
6 /* preconditions: array of stray cats along with the stop to */
7 /* start counting back from are passed in. */
8 /* postconditions: the return value is the total amount of */
9 /* stray cats up to the stop indicated by the */
10 /* value of 'stops' passed in the first call. */
11 int catCount(const int strayCats[STOPS], const int stops);
```

‘main’ function in ‘Task1i_source.c’

In the ‘main’ function, an array of cat counts of size ‘STOPS’ is declared and initialized. An output will indicate the total cats observed, using the return value of the ‘catCount’ function. The ‘catCount’ function takes the array of counts and the stop number to count back from. Passing the value ‘STOPS’ as the stop to start from will start from the final stop.

```
13 int main(void)
14 {
15     /*Declaration and initialization*/
16     int strayCats[STOPS] = {12, 5, 3, 20, 15, 6, 7, 1, 19, 30};
17     /*Final Output*/
18     printf("Hence, a total of %d stray cats were observed.\n", catCount(
19         strayCats, STOPS));
20     return 0;
21 }
```

‘catCount’ function in ‘Task1i_source.c’

The ‘catCount’ function implementation will now be discussed. This function will recursively count the total amount of cats up to the ‘stop’ passed to the function. For the purpose of easier explanation, the outputs found in the function were removed. A version of the same function with all the outputs is found below the next listing. The outputs were used for better understanding of what is happening during runtime.

In the function, a variable ‘count’ was declared. It will hold the total stray cats up to the ‘stop’ parameter. An if statement checks whether the total up to the first stop is required, which is where counting will start from. Otherwise, the count is set to the current stop count plus the counts of previous stops by a recursive call. This means that it does not know the total up to it until the totals up to previous stops are calculated.

Since the ‘stop’ variable will range from 1 to ‘STOPS’ and does not start from zero, accesses to the array are done using ‘stop-1’. This range was used with the outputs in mind since a 0th stop would make less sense.

In each case, the count up to the *stop*th stop is returned.

```

23 int catCount(const int strayCats[STOPS], const int stop)
24 {
25     /*Declaration of variables*/
26     int count; //holds the total up to a stop
27
28     /*Base case - If this is the first stop, value is first element*/
29     if(stop == 1)
30         return strayCats[0];
31     else /*else, request value of previous stop*/
32     {
33         /*Add current value with count up to previous stop*/
34         count = strayCats[stop-1] + catCount(strayCats, stop-1);
35
36         /*The total up to this stop is returned*/
37         return count;
38     }
39 }

```

Listing 1: catCount without outputs

```

23 int catCount(const int strayCats[STOPS], const int stop)
24 {
25     /*Declaration of variables*/
26     int count; //holds the total up to a stop
27
28     /*Base case - If this is the first stop, value is first element*/
29     if(stop == 1)
30     {
31         printf("\nThe total up to stop 1 is: %d; ", strayCats[0]);
32         printf("Sending this data to stop 2...\n");
33         return strayCats[0];
34     }
35     else /*else, request value of previous stop*/
36     {
37         /*If this is not the final stop*/
38         if(stop != STOPS)

```

```

39     {
40         printf("For the total up to stop %d, the ", stop);
41         printf("total up to stop %d is required.\n", stop-1);
42     }
43     else
44     {
45         printf("For the total up to the final stop, ");
46         printf("the total up to stop %d is required.\n", stop-1);
47     }
48
49     /*Add current value with count up to previous stop*/
50     count = strayCats[stop-1] + catCount(strayCats, stop-1);
51
52     /*If this is not the final stop*/
53     if(stop != STOPS)
54     {
55         printf("The total up to stop %d is: %d; ", stop, count);
56         printf("Sending this data to stop %d...\n", stop+1);
57     }
58     else
59     {
60         printf("The total up to the final stop ");
61         printf("at the terminus is: %d.\n", count);
62     }
63
64     /*The total up to this stop is returned*/
65     return count;
66 }
67 }

```

Listing 2: catCount with outputs

Testing

For testing purposes, one sample output will be shown.

```

"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
For the total up to the final stop, the total up to stop 9 is required.
For the total up to stop 9, the total up to stop 8 is required.
For the total up to stop 8, the total up to stop 7 is required.
For the total up to stop 7, the total up to stop 6 is required.
For the total up to stop 6, the total up to stop 5 is required.
For the total up to stop 5, the total up to stop 4 is required.
For the total up to stop 4, the total up to stop 3 is required.
For the total up to stop 3, the total up to stop 2 is required.
For the total up to stop 2, the total up to stop 1 is required.

The total up to stop 1 is: 12; Sending this data to stop 2...
The total up to stop 2 is: 17; Sending this data to stop 3...
The total up to stop 3 is: 20; Sending this data to stop 4...
The total up to stop 4 is: 40; Sending this data to stop 5...
The total up to stop 5 is: 55; Sending this data to stop 6...
The total up to stop 6 is: 61; Sending this data to stop 7...
The total up to stop 7 is: 68; Sending this data to stop 8...
The total up to stop 8 is: 69; Sending this data to stop 9...
The total up to stop 9 is: 88; Sending this data to stop 10...
The total up to the final stop at the terminus is: 118.
Hence, a total of 118 stray cats were observed.

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.

```

Figure 12: Task 1i sample output

Task 1j

Introduction

The code for this task consists of two functions within one source file. Below is a list showing the layout of the functions within the respective files.

- Task1j_source.c
 - int main(void)
 - int computeChange(const int coins[COINS], const int change)

A list of assumptions made for this task is found below.

1. Providing the amount of coins only, rather than the amount of each type of coin, is enough.
2. The user will not ask for the minimum amount of coins of excessively large changes.

A list of identified limitations of the code is found below.

1. Changes input exceeding around 520,000 cents will crash the program. This is most probably due to the fact that an array of that size would have to be created. This could have been avoided by keeping less values stored in the ‘pastValues’ array, even though the algorithm would then have a longer running time.
2. If the change input is floating-point, it is still accepted, but only the integer part is read. When the program expects the next input, ‘scanf’ will attempt to read the fractional part left in the input buffer but this input is not accepted, even though the user never got the chance to input a different change. This results in an awkward output.

This task was done with generalisation in mind, meaning that the program will work with other sets of coins if these were to be changed. Also, the symbolic constants defined at the top of the source file can easily be modified if a different cache size or more types of coins are required. Caching done both outside the ‘computeChange’ function and in the function itself also ensures a very short running time.

Header files, constants, and function prototypes in ‘Task1j_source.c’

At the start, the ‘stdio.h’ header file was included and two symbolic constants ‘COINS’ and ‘CACHE_SIZE’ were defined. ‘COINS’ indicates the amount of unique coins that may make up the change while ‘CACHE_SIZE’

indicates the maximum amount of cached results. A cache of size 50 was assumed to be enough. A function ‘computeChange’ was defined. This function will be used to calculate the minimum coins that will make up the change.

```

1 #include <stdio.h> //printf(), puts(), scanf(), getchar()
2 #define COINS 8 //amount of unique coins
3 #define CACHESIZE 50 //maximum cached results
4
5 /* operation: calculates the the minimum number of coins that */
6 /* make up the 'change' passed in. */
7 /* preconditions: the set of different types of coins and the */
8 /* change to be given are passed in. */
9 /* postconditions: minimum amount of coins that make up the */
10 /* change is returned. */
11 int computeChange(const int coins[COINS], const int change);

```

‘main’ function in ‘Task1j_source.c’

The main function starts off with the declaration of an array of constants and a set of necessary variables variable. The array ‘coins’ is defined as the set of different types of coins (in cents) that may be used by the ‘computeChange’ function. Another array ‘cache’ is a two-dimensional array which will store past results in the form of pairs of values, one value indicating the change and the other value indicating the amount of coins. The variable ‘cacheCounter’ stores the index of the space in the cache that the next result will occupy.

In the first loop encountered, the change part of each pair of values in the cache is cleared.

```

13 int main(void)
14 {
15     /*Declaration and initialization*/
16     const int coins[COINS] = {1,2,5,10,20,50,100,200};
17     int index=0; //stores array index in loop
18     int changeInput=0; //stores the user input
19     int coinsResult=0; //stores result for a specific change
20     int cacheUsed=0; //indicates whether cache was used
21     int cacheCounter=0; //location of next cache space to use
22     int cache[CACHESIZE][2]; //storage of past results
23
24     /*Clearing the cache*/
25     for(index=0; index < CACHESIZE; index++)
26         cache[index][0] = -1;

```

Next, a do-while loop will loop until the user inputs a negative change, to indicate a ‘quit’. The user is now asked for a change input in cents. A while loop will loop until ‘scanf’ returns 1 indicating that a value was read successfully. For every unsuccessful input, the user is asked to enter another value and the input buffer is cleared using a loop which reads and discards characters in the input buffer until the newline is reached, indicating the end of the input. Once the input is accepted, the while loop terminates and a condition checks if the value is negative and breaks the main loop (i.e. quits) if it is so.

```

28  /*Loops until change input is negative*/
29  do
30  {
31      /*Input of change*/
32      puts("What is the required change in cents? (-ve to quit)");
33      while(!scanf("%d", &changeInput))
34      {
35          printf("Invalid value; insert an integer change...\n");
36          while(getchar()!='\n'); //clear the input buffer
37      }
38
39      /*If changeInput is negative, this is taken as a 'quit'*/
40      if(changeInput < 0)
41          break;

```

Otherwise, if the input is positive, the cache will now be checked. Before the loop, 'cacheUsed' is set to 0, assuming that the cache will not be used. A loop then iterates over the cache, comparing the first value in each pair (corresponding to the past change inputs) with the user's latest change input. If a match is found, the second value in the pair of values is copied to the 'coinsResult' variable and the 'coinsUsed' variable is set to 1 to indicate that the cache was indeed used. The loop is then broken using a 'break' statement since the rest of the cache need not be checked if the cache was used.

```

43      /*Check cache for past results*/
44      cacheUsed = 0;
45      for(index=0; index < CACHESIZE; index++)
46      {
47          if(cache[index][0]==changeInput)
48          {
49              coinsResult = cache[index][1];
50              cacheUsed = 1;
51              break;
52          }
53      }

```

Next, a condition checks if the cache was not used. If so, then the minimum amount of coins for the user's change input needs to be calculated. The change along with the 'coins' array are passed in to the function 'computeChange' to compute the minimum amount of coins required. The return value (i.e. the coins result) is stored in variable 'coinsResult'.

The user's input along with the result are now stored in the cache at the location indicated by the 'cacheCounter' variable. This variable is then incremented and a condition checks if the limit for the counter was reached. If so, the counter is reset to 0 so that the next pair of values to be stored in the cache will be stored at index zero, considering the first dimension in the two-dimensional array.

After the amount of coins was either obtained from the cache or calculated using the function 'computeChange', an output at the end of the loop outputs the minimum amount of coins required for the change input by the user.

```

55     /*Compute the amount of coins for 'changeInput'*/
56     if(!cacheUsed)
57     {
58         /*Change is calculated*/
59         coinsResult = computeChange(coins, changeInput);
60
61         /*Storing in cache*/
62         cache[cacheCounter][0] = changeInput;
63         cache[cacheCounter][1] = coinsResult;
64
65         /*Cache counter incremented and reset */
66         /*if the cache limit was reached.      */
67         cacheCounter++;
68         if(cacheCounter == CACHESIZE)
69             cacheCounter=0;
70     }
71
72     /*Final output*/
73     printf("For %d cents change, a minimum of %d coin/s is needed.\n\n",
74           changeInput, coinsResult);
75 }
76 while(1);
77
78 return 0;
79 }

```

‘computeChange’ function in ‘Task1j_source.c’

The algorithm used to come up with optimal results will now be discussed.

Consider a set of unique coins and a change for which the minimum number of coins to represent it needs to be discovered. Consider if the value of a coin from the set of coins is taken from the value of the change and the minimum number of coins required for the resultant value is found, then one of the ways in which the change can be represented is by this minimum count of the resultant value plus 1, to compensate for the coin taken out. If this process is applied for all coins in the set that are smaller than or equal to the change, this results in a set of different possible counts of coins that can make up the change. Taking the minimum of these counts, the minimum number of coins that represent the change is found. [9]

The ‘computeChange’ function implementation will be discussed next. Four necessary variables were declared. For a dynamic programming approach, the ‘subChange’ variable will progressively store the changes from zero to ‘change’ to calculate the minimum amount of coins required in each case. These minimums are stored in the ‘coinCount’ variable and the ‘past-Values’ array stores the different minimum coin counts from zero to ‘change’. It will be used as a cache internal to the function to make it easier to calculate the minimum number of coins required for larger changes.

Kindly proceed to the next page for the code corresponding to the above description.

```

81 int computeChange(const int coins[COINS], const int change)
82 {
83     /*Declaration of variables*/
84     int index;        //used as a loop index/counter
85     int subChange;    //stores change from 0 to 'change'
86     int coinCount;    //stores the amount of coins
87     int pastValues[change+1]; //used as a cache

```

The function's main loop now starts. To take on a dynamic approach, with each iteration of the loop, the 'subChange' has a different value, starting from zero and going up to 'change'. The minimum amount of coins is calculated for each 'subChange' and stored in the cache 'pastValues' for future calculations to use to speed up calculations.

At the start of each loop, 'coinCount' is set to its maximum, which would be a 'subChange'-worth of 1 cent coins. Now another loop will be used to calculate the minimum coin count. The for loop will iterate through all of the available coins. For each coin, it is first checked whether the 'subChange' is equal to a coin, in which case only 1 coin would be required to represent the sub change. Otherwise, if the change is not equal to the coin, it is first checked whether the sub change is larger than the coin currently being considered. For coins smaller than 'subChange', the amount of coins required for the 'subChange' minus the coin's value is checked (adding +1 to the count to represent the coin being subtracted). If this count is less than the current 'coinCount', then the current coin count is set to this smaller count.

Hence, after the for loop is done, the minimum coin count for the 'subChange' will have been obtained. This count is cached in the 'pastValues' array and the main loops proceeds to the next 'subChange' value. The final 'subChange' value will be the original 'change' passed in. The minimum coins for this change will be stored in the last element of the 'pastValues' array and the main loop terminates.

```

89     /*Main loop*/
90     for(subChange=0; subChange <= change; subChange++)
91     {
92         /*Maximum amount of coins is 'subChange' 1-cent coins*/
93         coinCount = subChange;
94
95         /*Calculating the minimum coin count*/
96         for(index=0; index<COINS; index++)
97         {
98             /*If the change is equal to a coin only one coin is required*/
99             if(subChange==coins[index])
100             {
101                 coinCount = 1;
102                 break;
103             }
104             else if(subChange>coins[index]
105                 && 1+pastValues[subChange-coins[index]] < coinCount)
106             {
107                 coinCount = 1+pastValues[subChange-coins[index]];
108             }

```



```
109     }
110     /*Storing the count in the cache*/
111     pastValues[subChange] = coinCount;
112 }
```

After the loop is terminated, the final minimum coin count for the change passed in is returned.

```
113 /*Return result of minimum amount of coins*/
114 return pastValues[change];
115 }
```

Testing

For testing purposes, one sample output will be shown. Four different change values were input. Note that for the non-integer input, the 'scanf' function accepted the integer '12' from '12.5' and when the program was ready for input again, the remaining '0.5' was read and an output indicated that this input was invalid. This phenomenon corresponds to the last limitation in the list of limitations presented at the start of the task.

Figure 13: Task 1j sample output

Task 2 Data Structures

Task 2a

Introduction

The code for this task consists of four functions spread out across three source files. Below is a list showing the layout of the functions within the respective files.

- Task2_source.c
 - int main(void)
- StackFunctions.c
 - void push(struct stack *theStack, const float item)
 - float pop(struct stack *theStack)
- Validation.c
 - int parenthesesVal(const char express[])
- Stack.h
- Validation.h

A list of identified limitations of the code is found below.

1. No checks were implemented to prevent the array in the stack implemented to overflow. Despite this, the standard expression length ‘EXPLEN’ should prevent such a case from happening.

‘Stack.h’

The first part of this subtask will focus on the stack and stack function implementations.

To start with, a header file ‘Stack.h’ for the stack data structure implementation and prototypes for the push and pop functions was created. A constant for the maximum expression length ‘EXPLEN’ was defined as 100. This sets a limit on the input expression’s size. It was placed in this header file to be accessible to the stack struct implementation, functions in the main source file and functions in the stack functions source file to be shown later on. Hence, errors caused due to illegal memory access when using arrays are highly reduced since the limit used is common.

The stack data structure was now implemented. The stack struct includes an integer stack pointer and an array of floats with maximum size of

‘EXPLEN’ since, at this point, it will never hold more items than the maximum expression size. The ‘float’ datatype for the array was chosen due to the fact that, looking ahead in the tasks, both characters and numbers will be pushed onto the stack. For characters, a float can store the corresponding American Standard Code for Information Interchange (ASCII) decimal value, while for numbers, the float can store the actual value, also allowing for the possibility of pushing floating-point numbers and not just integers. Due to the fact that a single stack will not be used for storing both characters and numbers simultaneously in the next tasks, there will be no mix-ups between ASCII values and operands. The ‘double’ data type was not used since it is assumed that the user will not be making use of excessively large numbers. Hence, memory is saved since ‘float’ takes up half the size of a ‘double’ in memory.

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #define EXPLEN 100
4
5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[EXPLEN];
10 };

```

Function prototypes for the ‘push’ and ‘pop’ functions were now created. The functions will take pointers to the new stack struct as arguments so that any changes done to the stacks passed in will reflect to the original ones passed to the function.

```

12 /* operation: pushes the float passed in onto the specified */
13 /* stack. The stack pointer is incremented. */
14 /* preconditions: a pointer to a predeclared stack and a float*/
15 /* are passed to the function. */
16 /* postconditions: stack will contain the float 'item' and the*/
17 /* stackPointer within the stack holds the new*/
18 /* float item's index in the 'items' array. */
19 void push(struct stack *theStack, const float item);
20
21 /* operation: pops and returns a float from the specified */
22 /* stack. The stack pointer is decremented. */
23 /* preconditions: a pointer to a predeclared stack from which */
24 /* a float will be obtained is passed in. */
25 /* postconditions: float obtained is returned and stack */
26 /* pointer will point to next item, if any. */
27 /* If stack was empty when 'pop' was called, */
28 /* a null character '\0' is returned instead. */
29 float pop(struct stack *theStack);
30
31 #endif // STACK_H_INCLUDED

```

‘StackFunctions.c’

A source file ‘StackFunctions.c’ for the stack functions implementations was created. The ‘Stack.h’ header file was included for the prototypes, stack struct, and constant ‘EXPLEN’.

The ‘push’ function takes a pointer to a stack and the item of type float to be pushed onto the stack. The function does not return anything, with the assumption that the stack will never be considered to be full. This is reinforced by the fact that the limit ‘EXPLEN’ used in the main source file is identical to that used in the stack definition. The stack’s stack pointer is incremented to store a new index for the stack’s array. The new item is then copied to the ‘items’ array at the index stored in the stack pointer.

On the other hand, the ‘pop’ function only takes a pointer to a stack as an argument and returns the float popped from the top of the stack. Firstly, it is checked whether the stack pointer is equal to -1 , which is the initial value for any stack pointer. This would indicate that the stack is empty. If the stack pointer is not -1 , the float at the index stored in the stack pointer is returned, followed by the decrementation of the stack pointer. Having the decrementation postfix to the stack pointer rather than prefixed is important since the stack pointer initially points to the item to be popped, and is then decremented to point to the new stack top, after being used in the expression. If the stack pointer is -1 , a null character is returned instead, which would indicate that the stack is empty.

```
1 #include "Stack.h"
2
3 void push(struct stack *theStack, const float item)
4 {
5     /*Stack pointer incremented and item stored at new location*/
6     (*theStack).stackPointer++;
7     (*theStack).items[(*theStack).stackPointer] = item;
8 }
9
10 float pop(struct stack *theStack)
11 {
12     /*If stack not empty, return item at location indicated*/
13     /*by stack pointer. Stack pointer is then decremented */
14     if((*theStack).stackPointer != -1)
15         return (*theStack).items[(*theStack).stackPointer--];
16     else
17         return '\0'; //stack is empty
18 }
```

‘Validation.h’

Now, focus will shift towards the parentheses validation function and its implementation.

A header file ‘Validation.h’ was created. This header file only includes a function prototype for the parentheses validation.

```

1 #ifndef VALANDCOMP_H_INCLUDED
2 #define VALANDCOMP_H_INCLUDED
3
4 /* operation: the parentheses, if any, in the expression found*/
5 /*           in 'express' are checked for mismatches. */
6 /* preconditions: array containing expression to be checked is*/
7 /*               passed to the function. */
8 /* postconditions: '0' is returned in the case of a mismatch. */
9 /*               '1' returned if the expression is valid. */
10 int parenthesesVal(const char express[]);
11
12 #endif // VALANDCOMP_H_INCLUDED

```

‘Validation.c’

A source file ‘Validation.c’ for the function implementations was created. Necessary header files were included. The ‘parenthesesVal’ function takes the expression to be checked as an argument and returns 0 and 1 if the expression is invalid or valid, respectively. The expression’s length is obtained using the ‘strlen’ function and is subtracted by 1 to exclude the newline character which is introduced when the expression is input in the main source file. A variable to store the array’s index was also declared.

Next, a stack to store the parentheses was declared and its stack pointer was initialised to -1.

```

1 #include <stdio.h> //for puts()
2 #include <string.h> //for strlen()
3 #include "Stack.h" //for stack
4
5 int parenthesesVal(const char express[])
6 {
7     /*Declaration of variables*/
8     int expressLength = strlen(express)-1; // -1 to exclude newline
9     int index=0; //array index for expression
10
11     /*declaring a stack*/
12     struct stack parenStack;
13     parenStack.stackPointer = -1; //initializing the stackPointer

```

For the actual validation, a loop goes through each character that makes up the expression passed to the function. When an open parenthesis is found, it is pushed onto the stack ‘parenStack’ using the ‘push’ function. On the other hand, when a close parenthesis is found, the ‘pop’ function is called to pop an open parenthesis off the stack. Since both functions expect a pointer to a stack, the address of the stack is passed by using the ‘&’ symbol. It is worth emphasizing that even though the stack stores float

values in its items array and that the 'push' and 'pop' functions accept and return floats respectively, characters are being pushed and popped on or from the stack instead. By implicit type-casts, the characters' ASCII value is actually being pushed and popped, not the characters themselves.

Since the 'pop' function returns '\0' when the stack is empty, the condition in the if statement at line 26 is satisfied if the stack is empty. In this case an output indicates that the expression is invalid since no match was made for the parenthesis and 0 is returned.

```
15  /*pushing and popping parentheses*/
16  for(index=0; index < expressLength; index++)
17  {
18      /*Push any open parentheses and pop an open      */
19      /*parenthesis when a close parenthesis is found */
20      if(express[index] == '(')
21          push(&parenStack, '(');
22      else if(express[index] == ')')
23      {
24          /*If stack is empty, function returns.      */
25          /*Otherwise, an open parentheses is popped*/
26          if(pop(&parenStack) == '\0')
27          {
28              puts("Invalid expression; parentheses mismatch.");
29              return 0;
30          }
31      }
32  }
```

When the loop terminates, the value of the stack's stack pointer is checked. If the stack pointer's value is not equal to -1 , then the stack still has open parentheses on it, indicating that the parentheses were not balanced and that the expression is invalid. In this case, an output indicates that the expression is invalid and 0 is returned. Otherwise, the function returns a 1 indicating that the expression passed the validation check.

```
34  /*Checking if stack still has parentheses*/
35  if(parenStack.stackPointer != -1)
36  {
37      puts("Invalid expression; parentheses mismatch.");
38      return 0;
39  }
40
41  /*Expression passed parentheses check*/
42  return 1;
43 }
```

‘Task2_source.c’

Finally, the main source file ‘Task2_source.c’ was created. The required header files were included. An array to hold the input infix expression was declared with size ‘EXPLEN’ from ‘Stack.h’. Next, a do-while loop was set up to loop until the input is valid or the user quits. The ‘fgets’ function was used for a safer approach to string input where ‘EXPLEN’ is passed to the function to indicate the maximum amount of characters to read. The input infix expression is read into ‘express’. If the first character is a ‘q’, this is taken as a quit, and so the program exits by invoking ‘exit()’ with an argument 0 to indicate normal termination. Validity of the expression is determined in the post-tested loop’s condition which is satisfied if the function ‘parenthesesVal’ returns a 0, indicating an invalid expression, with respect to the parentheses.

Once the loop terminates, an output indicates that the expression was valid, with respect to the parentheses.

```
1 #include <stdio.h>           //for puts() and fgets()
2 #include <stdlib.h>          //for exit()
3 #include "Validation.h"      //for parentheses validation
4 #include "Stack.h"           //for EXPLEN
5
6 int main(void)
7 {
8     /*Declaration of array to hold input infix expression*/
9     char express[EXPLEN];
10
11     /*Loops until expression is valid or user quits*/
12     do
13     {
14         /*Input of expression*/
15         puts("Insert an arithmetic expression (q to quit): ");
16         fgets(express, EXPLEN, stdin); //input of infix expression
17
18         /*Check if user inserted a 'q'*/
19         if(express[0] == 'q')
20             exit(0);                //user inserted a 'q' so quit
21
22     } while(!parenthesesVal(express)); //repeat until expression is valid
23     puts("Expression parentheses validated.");
24
25     return 0;
26 }
```

Testing

Testing of the code found in this subtask will take place within the testing of task 2c.

Task 2b

Introduction

The code for this task consists of seven functions spread out across three source files. Since this subtask continues from the previous subtask, only the changes between the two subtasks will be documented. Below is a list showing the layout of the functions within the respective files.

- Task2_source.c
 - int main(void)
 - void infixToPostfix(const char infixExp[], char postfixExp[])
 - int isOperator(const char a)
- StackFunctions.c
 - void push(struct stack *theStack, const float item)
 - float pop(struct stack *theStack)
- ValAndComp.c
 - int parenthesesVal(const char express[])
 - int precedenceComp(const char a, const char b)
- Stack.h
- ValAndComp.h

‘Task2_source.c’

In the main source file ‘Task2_source.c’, two new header files ‘string.h’ and ‘ctype.h’ were included. Two function prototypes were also added. It is worth noticing that the ‘Validation.h’ header file was renamed to ‘ValAndComp.h’ since the header file will now contain function prototypes for comparison of operator precedence as well.

```

1 #include <stdio.h>           //for puts() and fgets()
2 #include <stdlib.h>          //for exit()
3 #include <ctype.h>           //for isdigit()
4 #include <string.h>          //for strlen()
5 #include "ValAndComp.h"      //for parentheses validation
6 #include "Stack.h"           //for stack, stack functions, and EXPLEN
7
8 /* operation: Converts the infix expression passed in to a      */
9 /*                  postfix expression.                          */
10 /* preconditions: The infix expression is stored in 'infixExp' */
11 /*                  and an additional array 'postfixExp' is      */
12 /*                  passed to store the resultant postfix exp-  */
13 /*                  resion without having to return anything.   */
14 /* postconditions: The 'postfixExp' array will now hold the      */

```



```

15  /* postfix equivalent of the passed expression*/
16  void infixToPostfix(const char infixExp[], char postfixExp[]);
17
18  /* operation: Checks whether the character passed to the      */
19  /* function is an operator (+, -, /, or *).                  */
20  /* preconditions: the character to be checked is passed to    */
21  /* the function.                                              */
22  /* postconditions: 1 is returned the 'a' is an operator.      */
23  /* 0 is returned the 'a' is not an operator.                 */
24  int isOperator(const char a);

```

The new function 'infixToPostfix' takes two character arrays as arguments. Array 'infixExp' stores the infix expression input by the user in the 'main' function, while 'postfixExp' will store the resultant postfix expression generated by the conversion to be implemented.

The necessary variables were declared and initialized. Although the stack struct created earlier on takes floating point numbers as its items, a character variable 'ch' will be used to store popped values from the stack since throughout this function, only operators and parentheses will be stored on the stack, and not floating-point numbers. The expression's length is calculated using 'strlen' and is decremented by 1 to exclude the newline introduced by the 'fgets' function in the 'main' function.

```

52  void infixToPostfix(const char infixExp[], char postfixExp[])
53  {
54      /*Declaration and initialization*/
55      int infixIndex=0; //array index for infixExp and loop counter
56      int pofxIndex=0; //array index for postfixExp
57      char ch; //stores popped stack items
58      int expressLength = strlen(infixExp)-1; //-1 to exclude newline

```

Next, a stack 'opStack' is declared. This stack will store operators and parentheses in the conversion process found next.

```

60      /*Declaring a stack*/
61      struct stack opStack;
62      opStack.stackPointer = -1; //initializing the stackPointer

```

The main loop now starts. The variable 'infxIndex' doubles as the loop's index and as an index for the 'infixExp' array. The loop will go through the expression character by character, left-to-right.

The first condition handles operands. It checks whether the current character is a digit. If so, an operand was found and simply needs to be added to the postfix expression. A do-while loop was set up to read any further digits or decimal points and loops until a character which is neither is found. Every digit or decimal point read is copied from the 'infixExp' array to the 'postfixExp' array, incrementing the indices using postfix incrementations. The 'infxIndex' now stores the index of the next character to be read while 'pofxIndex' now stores the index of the next free location in 'postfixArray'. The loop condition checks the next character. It is assumed that the user will only input one decimal point per operand.

After the loop terminates, a space is added to the postfix expression to separate the operand from any item that will follow, also incrementing the 'pofxIndex' index in the process. Since the index 'infxIndex' already points to the first character found after the operand, it is decremented by 1 to counteract the incrementation which will take place due to the for loop once the if statement is exited. Hence, when the incrementation takes place, 'infxIndex' will store the index of the first character found after the operand.

```

64  /*Main loop - goes through expression one character at a time*/
65  for(infxIndex=0; infxIndex < expressLength; infxIndex++)
66  {
67      /*If digit, read all further digits*/
68      if(isdigit(infxExp[infxIndex]))
69      {
70          /*Loop while the next character is a digit or decimal point*/
71          do
72          {
73              postfixExp[pofxIndex++] = infxExp[infxIndex++];
74          }while(isdigit(infxExp[infxIndex])
75                || infxExp[infxIndex]=='.'');
76
77          /*Insert a space for a new postfix element to come after*/
78          postfixExp[pofxIndex++] = ' ';
79          /*Index adjusted since an extra character was read*/
80          infxIndex--; //since an extra character was read
81      }

```

If the current character is not a digit, the second condition checks if it is an open parenthesis. Following the instructions given in the assignment, open parentheses are simply pushed onto the stack 'opStack'.

```

82      else if(infxExp[infxIndex] == '(')
83      {
84          /*Left parentheses simply pushed*/
85          push(&opStack, '(');
86      }

```

The next condition checks if the current character is a close parenthesis. Following the instructions provided, when a close parentheses is found, operators are popped from 'opStack' and added to the postfix expression 'postfixExp' along with a space until the popped item is the corresponding open parentheses.

```

87      else if(infxExp[infxIndex] == ')')
88      {
89          /*For right parentheses, append operators to */
90          /*the postfix expression. */
91          while((ch=pop(&opStack)) != '(')
92          {
93              /*Insert popped operator and a space*/
94              postfixExp[pofxIndex++] = ch;
95              postfixExp[pofxIndex++] = ' ';
96          }
97      }

```

If the character is neither an operand nor a parentheses, the next condition checks if the character read is an operator using a function 'isOperator' implemented further on. Intuitively, the condition is satisfied if the character is an operator, which includes the binary operators '+', '-', '*', '/', as well as the unary minus operator '-' for negative numbers. In the final postfix representation, unary minuses will be placed directly after their operand (with a space in between), symbolising a subtraction of the value from an implicit zero.

If this condition 'isOperator' is satisfied, before appending the found operator to the postfix expression, any operators with equal or higher precedence need to be popped and appended to 'postfixExp'. A while loop is used to pop operators until an open parenthesis is found or until the stack is emptied (indicated by the character returned from 'pop' being a null character '\n'). For each iteration of the loop, the popped operator's precedence is compared to the one found at 'infixIndex'.

The function 'precedenceComp' takes two operators as arguments and returns '-1' if the first operator's precedence is less than that of the second one. Hence, the if statement checks whether the popped operator's precedence is not less than that of the one at 'infixIndex', effectively checking if the popped operator's precedence is higher than or equal to that of the other operator. If the condition is satisfied, the popped operator is added to the postfix expression at index 'pofxIndex', followed by a space. In each case, 'pofxIndex' is incremented by a postfix increment operation so that it stores the index of the next space in the 'postfixExp' array after being used in each statement.

On the other hand, if the precedence of the popped operator is less than that of the operator at 'pofxIndex', then the popped operator stored in 'ch' is pushed back onto the stack, using the '&' symbol to pass the address of the stack 'opStack' along with the character to be pushed. Since an operator of lower precedence was found and cannot be added to the postfix expression, no further operators will be popped, so the loop is broken by a 'break' statement.

After the loop, if the last character popped from 'opStack' was an open parenthesis, it is pushed back onto the stack since open parentheses are only pushed out permanently when their corresponding close parenthesis is found. Finally, the operator encountered at the start is pushed onto 'opStack'.

```

98         else if(isOperator(infixExp[infixIndex]))
99         {
100             /*if operator was found, add all higher-or-equal- */
101             /*precedence operators to postfix expression      */
102             while((ch=pop(&opStack)) != '(' && ch!='\0')
103             {
104                 /*If not less precedence (i.e if higher or equal)*/
105                 if(precedenceComp(ch, infixExp[infixIndex]) != -1)
106                 {
107                     /*Insert popped operator and a space*/

```

```

108         postfixExp[pofxIndex++] = ch;
109         postfixExp[pofxIndex++] = ' ';
110     }
111     else
112     {
113         /*Push operator with less precedence*/
114         push(&opStack, ch);
115         /*Append no more operators to the list*/
116         break;
117     }
118 }
119
120 /*Open parenthesis pushed back*/
121 if(ch == '(')
122     push(&opStack, '(');
123
124 /*Original operator now pushed*/
125 push(&opStack, infixExp[infxIndex]);
126 }

```

If character at 'infxIndex' is not an operand, parenthesis, or an operator, then it is considered to be an extra character which can be ignored. This includes spaces but also includes any possible unnecessary characters such as letters or other symbols considered to be illegal characters.

```

128     /*If character did not satisfy a condition, the character*/
129     /*is simply skipped since it is not a useful character. */
130 }

```

After the loop has terminated, any remaining operators have to be popped and added to the postfix expression. A while loop pops characters until the stack is emptied. Each popped character 'ch' is copied to the postfix expression, followed by a space after each operator.

Additionally, a null character replaces the last space inserted after the last element in the postfix expression so that the postfix expression can now be treated as a string, allowing for use of string functions such as 'strlen' to obtain the expression's length.

```

132 /*Pop remaining operators*/
133 while((ch=pop(&opStack)) != '\0')
134 {
135     /*Insert popped operator and a space*/
136     postfixExp[pofxIndex++] = ch;
137     postfixExp[pofxIndex++] = ' ';
138 }
139
140 /*Last character set to a null character */
141 postfixExp[pofxIndex-1] = '\0'; //exclude last space
142 }

```

The 'isOperator' function implementation will now be discussed. The function simply takes a character argument and returns the result of a condition. The condition returns zero if the character passed to the function is not an operator, i.e. '+', '-', '*', or '/'. Otherwise, it returns a 1, indicating that the character is indeed an operator.

```

144 int isOperator(const char a)
145 {
146     /* 1 is returned the 'a' is an operator.    */
147     /* 0 is returned the 'a' is not an operator. */
148     return (a == '+' || a == '-' || a == '*' || a == '/');
149 }

```

Now, changes in the 'main' function will be pointed out. If the expression passed from the parentheses check, then this code section is next. A character array is declared to store the postfix expression to be generated. The size of this array was set to twice that of the infix expression since a postfix expression may have a space for each character of the infix expression such that it ends up being twice the size of the infix expression (minus 1, if the last space is excluded). This is the case if only operators and single-digit operands are used, without parentheses. Example: '1 + 2 + 3 + 4 + 5' of length 9 characters would result in '1 2 + 3 + 4 + 5 +' of length 17 characters. Next, the infix expression and the newly declared array are passed to the 'infixToPostfix' function to generate the postfix expression which is then output to the user.

```

44     /*Infix to postfix conversion*/
45     char postfixExp[2*strlen(express)]; //array declared
46     infixToPostfix(express, postfixExp); //expression converted
47     printf("Postfix: \"%s\\n\"", postfixExp);

```

'StackFunctions.c' and 'Stack.h'

Next, changes in the stack definition and functions part of the code will be indicated. The 'StackFunctions.c' source file was not modified. In the 'Stack.h' header file, the stack struct definition was slightly modified. The size of the maximum stack storage was doubled to cater for the case recently described where the infix expression reaches the expression limit and the postfix expression is twice its size. Since the expression limit is common between all files, the stack will theoretically never exceed its size limit unless a logical error were to be present.

```

5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[2*EXPLEN];
10 };

```

‘ValAndComp.c’ and ‘ValAndComp.h’

The validations portion of the code was also modified. First of all, the header and source files ‘Validation.h’ and ‘Validation.c’ were renamed to ‘ValAndComp.h’ and ‘ValAndComp.c’. In the ‘ValAndComp.h’ header file, a new prototype for the function ‘precedenceComp’ was added.

```

12 /* operation: compares the precedence of two operators from */
13 /*          '+', '-', '*', and '/' */
14 /* preconditions: characters 'a' and 'b' each contain an */
15 /*          an operator. These will be compared together */
16 /* postconditions: '1' is returned if precedence of 'a' is */
17 /*          greater than that of 'b' */
18 /*          '0' is returned if precedence of 'a' is */
19 /*          equal to that of 'b' */
20 /*          '-1' is returned if precedence of 'a' is */
21 /*          less than that of 'b' */
22 int precedenceComp(const char a, const char b);

```

The ‘ValAndComp.c’ source file contains the implementation for the newly added ‘precedenceComp’ function. The function takes two characters as arguments. It is assumed that these two characters will always be ‘+’, ‘-’, ‘*’, or ‘/’. Due to the fact that the infix to postfix conversion gets rid of any extra characters, it is highly unlikely that a character which is not an operator from this list is passed to this function, so this seems to be a healthy assumption to make.

The function simply returns the result of the subtraction of two terms. The terms are 1 when their condition is satisfied and 0 otherwise. Hence, the possible return values are 1 (when first term is 1 and second term is 0), 0 (when the terms are both 1 or 0), -1 (when the first term is zero and the second term is 1). Since ‘*’ and ‘/’ have higher precedence than ‘+’ and ‘-’, the return value will indicate whether operator ‘a’ has higher, equal, or lower precedence compared to operator ‘b’.

```

45 int precedenceComp(const char a, const char b)
46 {
47     /*If 'a' is * or /... */
48     /*...if 'b' is * or /, operators have equal precedence */
49     /*...if 'b' is + or -, then 'a' has higher precedence */
50     /*If 'a' is + or -... */
51     /*...if 'b' is * or /, then 'a' has lower precedence */
52     /*...if 'b' is + or -, operators have equal precedence */
53     return (a=='*' || a=='/') - (b=='*' || b=='/');
54 }

```

Testing

Testing of the code found in this subtask will take place within the testing of task 2c.

Task 2c

Introduction

The code for this task consists of eleven functions spread out across three source files. Similar to as was done in the previous subtask, since this subtask continues from the previous subtask, only the changes between the two subtasks will be documented. Below is a list showing the layout of the functions within the respective files.

- Task2_source.c
 - int main(void)
 - void infixToPostfix(const char infixExp[], char postfixExp[])
 - float evaluateExpress(char postFixExp[])
 - int isOperator(const char a)
- StackFunctions.c
 - void push(struct stack *theStack, const float item)
 - float pop(struct stack *theStack)
 - int pushNumber(struct stack *theStack, char express[], int index)
- ValAndComp.c
 - int expressionVal(const char express[])
 - int parenthesesVal(const char express[])
 - char charactersVal(const char express[])
 - int precedenceComp(const char a, const char b)
- Stack.h
- ValAndComp.h

A list of identified limitations of the code is found below. This list excludes limitations already mentioned in other subtasks in task 2.

1. Some types of invalid inputs still pass all validation checks and were not catered for. This includes (but is not limited to) divisions by zero and double decimal points within a single number.
2. The unary minus operator cannot be used with the guarantee that it will work, even though it does work for a variety of inputs. The user is advised to simulate a unary minus by using $(0 - A)$ instead of a direct $-A$. In the program, the unary minus uses the precedence of a binary minus, which is logically incorrect.

3. Spaces in the infix expression are forbidden, meaning that inputting an expression may be less user friendly, but this effectively reduces the possible invalid expressions.

‘Task2_source.c’

Starting off from the changes in the main source file ‘Task2_source.c’, header file ‘stdbool.h’ was added.

```

1 #include <stdio.h>           //for puts() and fgets()
2 #include <stdlib.h>          //for exit()
3 #include <ctype.h>           //for isdigit()
4 #include <string.h>          //for strlen()
5 #include <stdbool.h>         //for bool, true, and false
6 #include "ValAndComp.h"      //for expression validation
7 #include "Stack.h"           //for stack, stack functions, and EXPLEN

```

A function prototype for a new function ‘evaluateExpress’ was also added to the source file.

```

19 /* operation: Evaluates the postfix expression passed in and */
20 /*           returns the final answer.                        */
21 /* preconditions: The array passed in contains an expression */
22 /*               in postfix notation.                        */
23 /* postconditions: The answer of type float is returned.      */
24 float evaluateExpress(char postFixExp[]);

```

In the ‘main’ function, line 54 was modified to invoke a new function ‘expressionVal’ instead of ‘parenthesesVal’. This function from ‘ValAndComp.c’ will perform a set of validation checks and will not only validate parentheses. Line 62 was added to invoke the new ‘evaluateExpress’ function and use its return value, a value of type ‘float’, to output the result of the evaluation. A loop was also added around all of the code, excluding the ‘express’ array declaration since the array will not be changing size. The loop will loop until the user inserts a ‘q’ to indicate a request to quit followed by the calling of the ‘exit’ function at line 51.

```

34 int main(void)
35 {
36     /*Declaration of array to hold input infix expression*/
37     char express[EXPLEN];
38
39     /*Loops until user quits*/
40     do
41     {
42         /*Loops until expression is valid or user quits*/
43         do
44         {
45             /*Input of expression*/
46             puts("Insert an arithmetic expression (q to quit): ");
47             fgets(express, EXPLEN, stdin); //input of infix expression
48
49             /*Check if user inserted a 'q'*/
50             if(express[0] == 'q')
51                 exit(0); //user inserted a 'q' so quit
52

```



```

53     }
54     while(!expressionVal(express)); //repeat until expression is valid
55
56     /*Infix to postfix conversion*/
57     char postfixExp[2*strlen(express)]; //array declared
58     infixToPostfix(express, postfixExp); //expression converted
59     printf("Postfix: \"%s\"\n", postfixExp);
60
61     /*Evaluation and postfix expression and output of answer*/
62     printf("Answer: %.4f\n\n", evaluateExpress(postfixExp));
63 }
64 while(1);
65
66 return 0;
67 }

```

Now, the function implementation for the ‘evaluateExpress’ function will be discussed. This function will evaluate the postfix expression and come up with an answer which it then returns. The necessary variables were declared, among which is a variable ‘operatorFound’. This variable is of type ‘bool’ obtained from ‘stdbool.h’. It is set to ‘true’ once the first operator is found. If no operator is found, this means that the user only input a single operand.

A stack ‘postfixStack’ to hold operands is now declared and its stack pointer was initialized to -1 , indicating an empty stack.

```

162 float evaluateExpress(char postfixExp[])
163 {
164     /*Declaration and initialization*/
165     int index=0; //array index for postfixExp
166     float op1, op2; //store values in arithmetic operations
167     float answer=0.0; //stores answer in arithmetic operations
168     bool operatorFound=false; //indicates that at least one operator found
169
170     /*Declaring a stack*/
171     struct stack postfixStack;
172     postfixStack.stackPointer = -1; //initializing the stackPointer

```

The main loop now starts. This loop will go through the postfix expression and will keep repeating until the end of the postfix expression, indicated by a ‘\0’ is reached. The null character was introduced by the ‘infixToPostfix’ function to easily keep track of the end of the expression.

Since ‘index’ was initialized to 0, the first iteration of the loop will check the first character. The first condition checks whether the character is a space. Spaces are skipped, so the index is simply incremented to store the index of the next character and a new iteration of the loop starts.

If the character is not a space, the character must either be an operand or an operator, otherwise the expression is invalid. In the else branch, the first condition checks if the character is a digit. If so, then the start of an operand was found. The ‘pushNumber’ function from ‘Stack.h’ is invoked to go through the array ‘postfixExp’ starting from the first digit at ‘index’ to form the operand, and pushes the operand into the specified stack ‘postfixStack’. The return value of the function is the index of the first character found

after the operand, so that the loop may continue going through the postfix expression from after the operand.

The second condition in the else branch checks if the character is an operator. If so, the variable 'operatorFound' is set to 'true' if this is the first operator found. Next, two operands to perform the operation on are popped from the stack 'postfixStack'. Since operands are popped in a 'first-in-first-out' fashion, the first operand popped is set to 'op2' and the second operand popped is set to 'op1'. In the case of a unary minus, the result may be invalid in certain cases. If the stack only has one operand in it, 'op2' will be set to this operand and 'op1' will be set to zero since the 'pop' will return a '\0'. Hence the operation $0 - op2$ will be performed, simulating a unary minus. In other cases, the unary minus has a higher chance of working when the negative number is not involved in multiplications or divisions.

```

174  /*Loops until the end of the postfix expression is reached*/
175  while(postfixExp[index] != '\0')
176  {
177      /*Spaces are skipped*/
178      if(postfixExp[index] == ' ')
179          index++;
180      else
181      {
182          /*If first digit of a number was found, pushNumber invoked.*/
183          /*Else if an operator is found, perform an operation.      */
184          if(isdigit(postfixExp[index]))
185          {
186              /*Index set to first character after the number*/
187              index = pushNumber(&postfixStack, postfixExp, index);
188          }
189          else if (isOperator(postfixExp[index]))
190          {
191              /*If this is the first operator found, set to true*/
192              if(!operatorFound)
193                  operatorFound=true;
194
195              /*Two operands popped from stack*/
196              op2 = pop(&postfixStack);
197              op1 = pop(&postfixStack);

```

Moving on to the operation, a switch statement now checks the operator to see which operation will be performed. For the specific operator, the operation performed will use 'op1' and 'op2' in that order and the result is stored in 'answer'. After the operation has been performed, the result is pushed back onto the stack 'postfixStack' to be used as an operand in any following operations. An output indicates the operation performed and the index is incremented so that it stores the index of the next character in the postfix expression.

The else branch found at line 222 within the main else branch is used in the case that the character is neither an operator nor an operand. This would indicate that a forbidden character is present in the expression and that the expression is invalid. An output indicates this and the function returns a dummy result of zero.

```

199     /*Performing operation according to operator*/
200     switch(postfixExp[index])
201     {
202     case '+':
203         answer = op1+op2;
204         break;
205     case '-':
206         answer = op1-op2;
207         break;
208     case '*':
209         answer = op1*op2;
210         break;
211     case '/':
212         answer = op1/op2;
213         break;
214     }
215
216     /*sub-answer pushed back onto the stack*/
217     push(&postfixStack, answer);
218     printf("Sub-expression: %10.4f %c %10.4f = %10.4f\n",
219         op1, postfixExp[index], op2, answer);
220     index++;
221 }
222 else //character is neither a digit nor an operand
223 {
224     puts("Error in evaluation");
225     return 0;
226 }
227 }
228 }

```

Once the loop terminates, the variable ‘operatorFound’ is checked. If no operator was found, this means that, unless the input was not in a suitable form, only one operand was present in the expression and is still in the stack ‘postfixStack’. Hence, the value is popped and returned. Otherwise, the latest result stored in ‘answer’ is returned.

```

230     /*If no operators were found, answer is the number in the stack*/
231     /*Otherwise, answer of last sub-expression is returned*/
232     if(!operatorFound)
233         return pop(&postfixStack);
234     else
235         return answer;
236 }

```

‘Stack.h’

Moving on to the ‘Stack.h’ header file, a new function prototype for function ‘pushNumber’ was added.

```

31 /* operation: pushes a number found in the postfixExp array */
32 /*           starting from the location indicated by 'index'. */
33 /*           Characters from this location onwards are pushed */
34 /*           onto the specified stack until a character which */
35 /*           is not a digit, decimal point, or negative sign */
36 /*           is reached in the array. */
37 /* preconditions: a stack to push the number onto, the array */

```

```

38 /*          containing the number, and the index at          */
39 /*          which the number starts are all passed in.      */
40 /* postconditions: the stack now contains the number. The index */
41 /*          of the array location containing the first      */
42 /*          character found after the number is returned.*/
43 int pushNumber(struct stack *theStack, char postfixExp[], int index);

```

‘StackFunctions.c’

In the ‘StackFunctions.c’, two header files ‘ctype.h’ and ‘stdlib.h’ were added to the source file.

```

1 #include <ctype.h> //for isdigit()
2 #include <stdlib.h> //for atof()
3 #include "Stack.h"

```

The implementation for the ‘pushNumber’ function was also added. This function pushes a number found in ‘express’ and starting at ‘index’ onto the ‘theStack’. A necessary variable and a stack are declared and the stack’s stack pointer is initialized.

```

22 int pushNumber(struct stack *theStack, char express[], int index)
23 {
24     /*Declaration of variables*/
25     int tempIndex; //stores array index
26     int numCount=0; //stores amount of number characters

```

A loop starts from ‘index’ and pushes all characters that make up the number onto the stack ‘numStack’, incrementing ‘tempIndex’ with each iteration to move on to the next character. The loop stops once a character which is not part of the number is found. The stack now contains the characters that make up the number, hence the stack pointer’s value will give an indication of the number’s length. The ‘numLength’ is set to the value of the stack pointer, plus an extra 2 elements, one for the fact that the stack pointer starts from zero, and one to leave a space for a null character ‘\0’ to be included. Now that the size of the operand is known, an array ‘number’ of that size was created to store the operand.

```

29     /*until a character which is not a digit, a decimal point,*/
30     /*or a negative sign is found.*/
31     tempIndex = index;
32     while(isdigit(express[tempIndex])
33         || express[tempIndex]=='.'
34         || express[tempIndex]=='-')
35     {
36         numCount++;
37         tempIndex++;
38     }
39
40     /*Declaring array for number*/

```

A second loop now uses ‘index’ and ‘tempIndex’ to iterate over the two arrays ‘number’ and ‘express’, starting from 0 for the ‘number’ array and starting from ‘index’ for the ‘express’ array, since that is the index that

the operand starts from within the array 'express'. The operand is copied from 'express' to 'number', character by character, incrementing 'index' with each iteration using a postfix increment operation so that the next character can be copied. By the end of the loop, 'index' will store the index of the character found exactly after the operand.

Once the loop terminates, 'tempIndex' has the value 'numLength-1', which is the index of the last element in 'number'. A null element '\0' is copied to the last element in the array 'number' so that it can be treated as a string.

Since the operand is now a string, the function 'atof' can be used to convert it into a floating-point number. The function 'push' is invoked to push the resultant floating-point number onto the stack originally passed to the 'pushNumber' function.

Finally, 'index' is returned to indicate the location of the character found exactly after the operand read, so that the expression evaluation can continue scanning characters from after the operand.

```

42
43     /*Sub-array for operand created*/
44     for(tempIndex=0; tempIndex < numCount; tempIndex++)
45         number[tempIndex] = express[index++];
46     number[tempIndex] = '\0';
47
48     /*Push number onto stack*/
49     push(theStack, atof(number));
50     /*Index of first character found after the number is returned*/
51     return index;
52 }

```

'ValAndComp.h'

Now, changes in the validation and comparison sections of the code will be pointed out. Starting from the 'ValAndComp.h' header file, a new function prototype 'expressionVal' was added to this header file, and the function prototype for 'parenthesesVal' was moved to the 'ValAndComp.c' source file. This is due to the fact that the implementation of 'expressionVal' in 'ValAndComp.c' will invoke multiple validation checks within 'ValAndComp.c', including the 'parenthesesVal' function.

```

18 /* operation: compares the precedence of two operators from */
19 /*          '+', '-', '*', and '/' */
20 /* preconditions: characters 'a' and 'b' each contain an */
21 /*          an operator. These will be compared together*/
22 /* postconditions: '1' is returned if precedence of 'a' is */
23 /*          greater than that of 'b' */
24 /*          '0' is returned if precedence of 'a' is */
25 /*          equal to that of 'b' */
26 /*          '-1' is returned if precedence of 'a' is */
27 /*          less than that of 'b' */
28 int precedenceComp(char a, char b);

```

‘ValAndComp.c’

In the ‘ValAndComp.c’ source file, a new header file ‘ctype.h’ was included.

```
1 #include <stdio.h> //for puts()
2 #include <string.h> //for strlen()
3 #include <ctype.h> //for isdigit()
4 #include "Stack.h" //for stack
```

A new function prototype for a function ‘charactersVal’ was also added to this source file.

```
14 /* operation: the characters which make up the expression in */
15 /*          'express' are checked for illegal characters. */
16 /*          Legal: '+', '-', '*', '/', '.', '(', and ')' */
17 /* preconditions: array containing expression to be checked is */
18 /*               passed to the function. */
19 /* postconditions: First illegal character found is returned. */
20 /*               Otherwise, a null character '\0' returned. */
21 char charactersVal(const char express[]);
```

The implementation of the ‘charactersVal’ function starts off with a declaration of required variables. The expression length is set such that it excludes the newline character introduced by the ‘fgets’ function when the infix expression gets originally input in the ‘main’ function.

```
90 char charactersVal(const char express[])
91 {
92     /*Declaration of variables*/
93     int expressLength = strlen(express)-1; // -1 to exclude newline
94     int index=0; //array index for expression
```

A loop now iterates over all the characters that make up the infix expression ‘express’ and check that all characters are either a digit or one of the characters from ‘+’, ‘-’, ‘*’, ‘/’, ‘.’, ‘(’, or ‘)’. It is worth noting that spaces are not allowed in the input. This will be held as a rule to reduce any errors related to spaces such as when the user only inputs spaces. If a forbidden character is found, it is immediately returned. Otherwise, once the loop terminates, a null character ‘\0’ is returned.

```
96     /*Loops through all characters of the expression*/
97     for(index=0; index < expressLength; index++)
98     {
99         /*If a character is not any of these characters, */
100        /*then it is considered to be forbidden. */
101        if( !isdigit(express[index]) && express[index] != '+'
102            && express[index] != '-' && express[index] != '*'
103            && express[index] != '/' && express[index] != '.'
104            && express[index] != '(' && express[index] != ')')
105        {
106            /*Illegal character returned*/
107            return express[index];
108        }
109    }
110    /*No illegal characters found - expression passed character check*/
111    return '\0';
112 }
```

Next, the implementation of the function ‘expressionVal’ will be discussed. A variable ‘ch’ was declared. The expression now goes through three validation checks: parenthesis validation by ‘parenthesesVal’, a characters validation by ‘charactersVal’, and a presence check which will check if the first character in the expression is the newline introduced by the ‘fgets’ function when the infix expression was originally input. If the expression fails a validation check, an output indicates this and the function returns a ‘0’. The outputs previously found in ‘parenthesesVal’ were moved to the ‘expressionVal’ function.

```

23 int expressionVal(const char express[])
24 {
25     /*Character used for return of function charactersVal*/
26     char ch;
27
28     /*A series of checks run*/
29     /*0 returned : expression failed at least one check*/
30     /*1 returned : expression passed all checks*/
31     if( !parenthesesVal(express) ) //parentheses check
32     {
33         puts("Invalid expression; parentheses mismatch.");
34         return 0;
35     }
36     else if( (ch=charactersVal(express)) ) //character check
37     {
38         printf("Invalid expression; illegal character '%c'.\n", ch);
39         return 0;
40     }
41     else if(express[0] == '\n') //expression presence check
42     {
43         puts("Invalid expression; nothing was input.");
44         return 0;
45     }

```

If the expression passes all validation checks, a final condition checks whether the size of the expression reached the limit imposed by ‘EXPLEN’ in ‘Stack.h’. An output warns the user that the results may be incorrect if this limit was reached. The function returns a ‘1’ at the end, indicating that the expression passed all of the validation checks.

```

47     /*Expression still considered to be valid if expression limit*/
48     /*was reached. Program might still crash if limit reached.  */
49     if(strlen(express) >= EXPLEN-1) //expression length limit check
50         puts("Warning: answer might be incorrect; expression length limit
51         reached.");
52
53     /*Expression passed all checks*/
54     return 1;

```

Testing

For testing purposes, one sample output will be shown. The inputs given to the program can be seen from the screenshots themselves.

```

"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Insert an arithmetic expression (q to quit):
(1+2)*3+4/5-6*((0.5/2)+(3*5))
Postfix: "1 2 + 3 * 4 5 / + 6 0.5 2 / 3 5 * + * ."
Sub-expression: 1.0000 + 2.0000 = 3.0000
Sub-expression: 3.0000 * 3.0000 = 9.0000
Sub-expression: 4.0000 / 5.0000 = 0.8000
Sub-expression: 9.0000 + 0.8000 = 9.8000
Sub-expression: 0.5000 / 2.0000 = 0.2500
Sub-expression: 3.0000 * 5.0000 = 15.0000
Sub-expression: 0.2500 + 15.0000 = 15.2500
Sub-expression: 6.0000 * 15.2500 = 91.5000
Sub-expression: 9.8000 - 91.5000 = -81.7000
Answer: -81.7000

Insert an arithmetic expression (q to quit):
0
Postfix: "0"
Answer: 0.0000

Insert an arithmetic expression (q to quit):
InvalidInputTest
Invalid expression; illegal character 'I'.
Insert an arithmetic expression (q to quit):
)(1+1)+1(
Invalid expression; parentheses mismatch.
Insert an arithmetic expression (q to quit):
123.456*789.10/11.12-13+14
Postfix: "123.456 789.10 * 11.12 / 13 - 14 +"
Sub-expression: 123.4560 * 789.1000 = 97419.1250
Sub-expression: 97419.1250 / 11.1200 = 8760.7129
Sub-expression: 8760.7129 - 13.0000 = 8747.7129
Sub-expression: 8747.7129 + 14.0000 = 8761.7129
Answer: 8761.7129

Insert an arithmetic expression (q to quit):
(((2*2)*2)*2)*2
Postfix: "2 2 * 2 * 2 * 2 * 2"
Sub-expression: 2.0000 * 2.0000 = 4.0000
Sub-expression: 4.0000 * 2.0000 = 8.0000
Sub-expression: 8.0000 * 2.0000 = 16.0000
Sub-expression: 16.0000 * 2.0000 = 32.0000
Answer: 32.0000

Insert an arithmetic expression (q to quit):
3*(3*(3*(0+1)))
Postfix: "3 3 3 0 1 + * * *"
Sub-expression: 0.0000 + 1.0000 = 1.0000
Sub-expression: 3.0000 * 1.0000 = 3.0000
Sub-expression: 3.0000 * 3.0000 = 9.0000
Sub-expression: 3.0000 * 9.0000 = 27.0000
Answer: 27.0000

Insert an arithmetic expression (q to quit):

```

Figure 14: Task 2c sample output

Task 3 Allocated Memory and I/O

Introduction

The code for this task consists of eleven functions spread out across three source files. The code for this task continues from task 2c and documentation will only focus on the modified code. Below is a list showing the layout of the functions within the respective files.

- Task3_source.c
 - int main(void)
 - void infixToPostfix(const char infixExp[], char postfixExp[])
 - float evaluateExpress(char postfixExp[])
 - int isOperator(const char a)
- StackFunctions.c
 - void push(struct stack *theStack, const float item)
 - float pop(struct stack *theStack)
 - int pushNumber(struct stack *theStack, char express[], int index)
- ValAndComp.c
 - int expressionVal(const char express[])
 - int parenthesesVal(const char express[])
 - char charactersVal(const char express[])
 - int precedenceComp(const char a, const char b)
- Stack.h
- ValAndComp.h
- expressions.txt

A list of identified limitations of the code is found below.

1. The program could have been made more memory efficient by having better more advanced checks to predict exactly how many memory needs to be allocated.
2. The primary input is still stored in an array of fixed size, making it not memory efficient and imposing a limit on the length of the expression. A possible solution to this was proposed in the ‘Task3_source.c’ subsection within the documentation for this sub-task but was not implemented.

3. The file `expressions.txt` is assumed to exist. Cases where the file does not exist were not catered for.
4. Running the program from the Eclipse IDE required files to be in the task's folder, while running the program from Command Prompt required files to be in the Debug folder within the task's folder. Hence, the `'expressions.txt'` was placed both in the task's folder and the Debug folder.
5. If the file contains a blank line within the set of expressions, the program has a high chance of not successfully evaluating the expression.
6. In some cases, the last expression does not get successfully evaluated and the program terminates with an error. The cause of this problem was not identified but due to the fact that this problem is not present in task 4c, the source of the problem is most probably the implementation of the stack.

`'expressions.txt'` and `'answers.txt'`

The file `'expressions.txt'` contains a set of expressions to be evaluated. These were selected to test many of the possibilities of the program. The answers of the expressions' evaluation are stored in the file `'answers.txt'`, which may or may not exist before a file pointer is declared to it in the `'main'` function. The contents of the file `'expressions.txt'` are shown below:

```
1+1
(1+2)*3+4/5-6*((0.5/2)+(3*5))
0
InvalidInputTest
)(1+1)+1(
3/3/3/3/3/3/3/3/3/3
123.456*789.10/11.12-13+14
(((2*2)*2)*2)*2
3*(3*(3*(0+1)))
```

‘StackFunctions.c’ and ‘Stack.h’

The source file ‘StackFunctions.c’ was not modified, but the corresponding header file was. Starting off with the definition of the stack in ‘Stack.h’, the array size specification for the stack’s ‘items’ array was removed so that the array now becomes a dynamic array. Memory for this array is allocated at runtime where a custom size can be used as needed.

```
5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[];
10 };
```

‘Task3_source.c’

Moving on to the ‘Task3_source.c’ source file, the ‘main’ function was modified in many ways and will be fully documented. First of all, two pointers to files are declared and two files are opened using ‘fopen’. The file ‘expressions.txt’ is opened in read mode, while the file ‘answers.txt’ is opened in write mode. This mode caters for the case where the file ‘answers.txt’ does not exist at the starting point of execution. In such cases, it is created automatically when a file pointer is declared for it using the write mode. On the other hand, if it already existed, any present contents are discarded in preparation for the new contents.

An array of size ‘EXPLEN’ was also declared to store expressions input from the file. This part of the program was not made dynamic since this would involve having to figure out the size of the input and to allocate memory accordingly. Hence, the input would somehow have to be traversed before actually reading it so that the size of the input is known. One solution could have been to push the input into a stack and check the stack pointer for the size, but then the expression would have to be reversed after being popped out, because of the ‘last-in-first-out’ nature of the stack. Another solution could have been to read and count the characters that make up the input to find the length, and then setting the file pointer back to before the expression, where it can now be read and its size would now be known.

```
34 int main(void)
35 {
36     /*Declaration of file pointers and opening files*/
37     FILE *expFp = fopen("expressions.txt", "r");
38     FILE *ansFp = fopen("answers.txt", "w");
39
40     /*Declaration of array to hold input infix expression*/
41     char express[EXPLEN];
```

Next, a loop was set up to take inputs from the file pointer 'expFp' pointing to the file containing the infix expressions. The loop will repeat until the 'fgets' function returns a null pointer, marking the end of the file, and meaning that the loop condition will not be satisfied.

Since the last expression read will not include a newline character, and considering that the code is built with the 'fgets' function in mind, the newline character is manually added. Next, an output shows the expression read as a confirmation to the user. An if statement now invokes the 'expressionVal' function, passing the expression so that it goes through validation checks. If the condition is satisfied, the infix expression will now be converted to postfix. An array of double the size of the infix expression is declared to store the postfix expression, and the 'infixToPostfix' function is invoked with 'express' and 'postfixExp' as arguments.

```

43     while(fgets(express, EXPLEN, expFp))
44     {
45         /*Newline character added to last expression*/
46         if(express[strlen(express)-1] != '\n')
47             strcat(express, "\n");
48
49         /*Output indicates expression read*/
50         printf("\nExpression read: %s", express);
51
52         /*If expression is validated, condition is satisfied*/
53         if(expressionVal(express))
54         {
55             /*Infix to postfix conversion*/
56             char postfixExp[2*strlen(express)]; //array declared
57             infixToPostfix(express, postfixExp); //expression converted

```

After conversion to postfix, now evaluation has to take place. A null character is added to the end of the expression 'express' so that it can be used as a string in the following output. Function 'fprintf' is invoked to output the expression and its answer which will be obtained through calling the 'evaluateExpress' function. An output to the user indicates that evaluation was successfully completed.

After termination of the while loop, the file pointers previously declared are passed to the 'fclose' function so that the opened files are closed.

```

59         /*Evaluation of expressions and output to file*/
60         express[strlen(express)-1] = '\0'; //overwrite newline
61         fprintf(ansFp, "%s = %.4f\n", //output to file
62             express, evaluateExpress(postfixExp));
63         puts("Evaluation completed.");
64     }
65 }
66 /*Files closed*/
67 fclose(expFp);
68 fclose(ansFp);
69
70 return 0;
71 }

```

The ‘infixToPostfix’ function was also modified. A new variable ‘operatorCount’ was added. In the following loop, the amount of operators in the expression is calculated by checking each character and using function ‘isOperator’. The stack declaration was changed to a declaration of a pointer to a stack whose memory is allocated using ‘malloc’. As an argument, the ‘malloc’ function is provided with the size of the struct itself (which is basically just the size of the integer ‘stackPointer’) added to the size of a float variable multiplied by the amount of operators found in the expression. The maximum amount of operators that the stack can hold is hence the amount of operators found in the expression. After memory has been allocated, the stack pointer for the newly declared pointer to a stack is initialized to -1 . Dereferencing is used to be able to access the stack and its ‘stackPointer’.

```

80     int operatorCount=0; //stores the count of operators
81
82     /*Counting the operators in the expression*/
83     for(infixIndex=0; infixIndex < expressLength; infixIndex++)
84         if(isOperator(infixExp[infixIndex]))
85             operatorCount++;
86
87     /*Declaring a pointer to a stack*/
88     struct stack *opStack = malloc(sizeof(struct stack)
89                                   + operatorCount*sizeof(float));
89     (*opStack).stackPointer = -1; //initializing the stackPointer
90

```

Throughout this function and also in other functions, since pointers to stacks will now be used instead of declarations of stacks directly, the ‘pop’ and ‘push’ functions will not be invoked with the use of the ‘&’ symbol. This symbol was used to obtain the address of a the particular stack being used, but now pointers to stacks are being used, so using the pointer’s name will suffice since this is identical to obtaining the address of the stack. An example from line 114 is shown below. In the previous task, the address of ‘opStack’ would not be obtained through its name, but using the ‘&’ symbol.

```

114     push(opStack, '(');

```

The last change in the ‘infixToPostfix’ function is the use of the ‘free’ function. The pointer to the stack that was used throughout the function is passed to the ‘free’ function so that the memory allocated to it is freed and is available to be used by other functions.

```

171     /*Frees the memory space taken up by opStack*/
172     free(opStack);

```

Changes in the ‘evaluateExpress’ include changes similar to those in the previous function. A variable ‘operatorCount’ was added. A loop now iterates over the expression and calculates the amount of operators in the expression using the ‘isOperator’ function. The declaration of a stack was changed to a declaration of a pointer to a stack using ‘malloc’. The ‘malloc’ function is passed the size of the stack struct added to the size of a float variable multiplied by the amount of operators plus 1. The extra 1 is added

since the amount of operands is always one plus the amount of operators, provided that binary operators are being used. In the case where unary operators are used, the amount of operands will be greater, so the memory allocated will still be enough to store all of the operands in the expression. After memory has been allocated, the stack pointer for the newly declared pointer to a stack is initialized to -1 using dereferencing to access the stack and its 'stackPointer'.

```

182     int operatorCount;           //stores the count of operators
183
184     /*Counting the operators in the expression*/
185     for(index=0; index < strlen(postfixExp); index++)
186         if(isOperator(postfixExp[index]))
187             operatorCount++;
188
189     /*Declaring a pointer to a stack*/
190     struct stack *postfixStack = malloc(sizeof(struct stack)
191                                         + (1+operatorCount)*sizeof(float));
192     (*postfixStack).stackPointer = -1; //initializing the stackPointer

```

In the last part of the same function 'evaluateExpress', in the case where no operator was found, the answer is no longer returned directly from the 'pop' function, but is stored in the 'answer' variable first. In any case, after the if statement, the memory occupied by the stack is then freed using the 'free' function which takes the pointer to the stack as an argument. Finally, the answer is returned.

```

249     /*If no operators were found, answer is the number in the stack*/
250     /*Otherwise, answer of last sub-expression is returned */
251     if(!operatorFound)
252         answer = pop(postfixStack);
253     /*Free the memory space occupied by the postfixStack */
254     /*and return the result of the arithmetic evaluation. */
255     free(postfixStack);
256     return answer;
257 }

```

'ValAndComp.h' and 'ValAndComp.c'

In the validation and comparison part of the code, the header file 'ValAndComp.h' remained unchanged but the source file 'ValAndComp.c' was modified. In this source file, a new header file 'stdlib.h' was added.

```

1 #include <stdio.h> //for puts()
2 #include <string.h> //for strlen()
3 #include <ctype.h> //for isdigit()
4 #include <stdlib.h> //for malloc(), free()
5 #include "Stack.h" //for stack

```

Changes were also done in the 'parenthesesVal' function, since it previously made use of arrays whose size was determined by the constant 'EXPLEN'. A new variable 'openParCount' will store the count of open parentheses. The changes found next are very similar to the changes done in the

previous two functions discussed. A loop iterates over the expression and counts the open parentheses in 'openParCount'. Next, memory is allocated for the pointer to a stack 'parenStack'. The value in bytes passed to the 'malloc' function is the size in bytes of the struct itself, added to the size of a 'float' variable multiplied by the amount of open parentheses present in the expression. The stack's stack pointer is then initialized to -1.

```

57 int parenthesesVal(const char express[])
58 {
59     /*Declaration of variables*/
60     int expressLength = strlen(express)-1; //-1 to exclude newline
61     int index=0; //array index for expression
62     int openParCount=0; //stores the count of open parentheses
63
64     /*Count parentheses for use in malloc*/
65     for(index=0; index < expressLength; index++)
66         if(express[index] == '(')
67             openParCount++;
68
69     /*Declaring a pointer to a stack*/
70     struct stack *parenStack = malloc(sizeof(struct stack)
71                                     + openParCount*sizeof(float));
72     (*parenStack).stackPointer = -1; //initializing the stackPointer

```

The remaining changes in the 'parenthesesVal' function deal with freeing the allocated memory. At each point where the function returns, the memory that was allocated is freed beforehand using the function 'free' and passing the pointer to the stack as an argument. The three changes are shown below.

```

87     free(parenStack);
88     return 0;

```

```

96     free(parenStack);
97     return 0;

```

```

101     free(parenStack);
102     return 1;

```

Testing

For testing purposes, one sample output and the corresponding 'answers.txt' contents will be shown. The inputs read from the file 'expressions.txt' can be seen from the screenshot itself found below. Kindly proceed to the next page for the sample output and the contents of the 'answers.txt' file.

```

"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Expression read: 1+1
Evaluation completed.

Expression read: (1+2)*3+4/5-6*((0.5/2)+(3*5))
Evaluation completed.

Expression read: 0
Evaluation completed.

Expression read: InvalidInputTest
Invalid expression; illegal character 'I'.

Expression read: )(1+1)+1(
Invalid expression; parentheses mismatch.

Expression read: 3/3/3/3/3/3/3/3/3/3
Evaluation completed.

Expression read: 123.456*789.10/11.12-13+14
Evaluation completed.

Expression read: ((2*2)*2)*2
Evaluation completed.

Expression read: 3*(3*(3*(0+1)))
Evaluation completed.

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.

```

Figure 15: Task 3 sample output

The results were written to the file 'answers.txt'. For the above set of accepted expressions, the contents of the 'answers.txt' file after execution are the lines shown below. For invalid inputs, the file does not contain any corresponding result.

```

1+1 = 2.0000
(1+2)*3+4/5-6*((0.5/2)+(3*5)) = -81.7000
0 = 0.0000
3/3/3/3/3/3/3/3/3/3 = 0.0002
123.456*789.10/11.12-13+14 = 8761.7129
((2*2)*2)*2 = 32.0000
3*(3*(3*(0+1))) = 27.0000

```


Task 4 Abstract Data Types (ADT)

Task 4a

Introduction

The code for this task consists of six functions spread out across two source files. For the first two subtasks of task 4, the main function will not be used and so only contains a ‘return 0’ line. The points of focus of the first two subtasks will be the header file ‘Stack.h’ and the source file ‘StackFunctions.c’. Below is a list showing the layout of the functions within the respective files.

- Task4_source.c
 - int main(void)
- StackFunctions.c
 - Stack *newStack(const int max_size)
 - void push(struct stack *theStack, const float item)
 - float pop(struct stack *theStack)
 - float top(Stack *theStack)
 - bool is_empty(Stack *theStack)
- Stack.h

‘Stack.h’

The header file ‘Stack.h’ will be discussed first. The definition of the stack struct is identical to that found in the previous tasks. A typedef was added so that an alias for ‘struct stack’ is created. “Stack” can now be used in place of ‘struct stack’ and so declarations of stacks were all replaced to use this simpler form. Examples may be observed in the parameter lists of the functions to be discussed.

```
1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3
4 /*stack structure definition*/
5 struct stack
6 {
7     int stackPointer;
8     float items[];
9 };
10
11 /*An alias for the stack struct*/
12 typedef struct stack Stack; // "Stack" is an alias for the struct
```

This header file also contains five function prototypes. The ‘push’ and ‘pop’ function prototypes are identical to those in previous tasks, but another three new function prototypes were added for functions ‘newStack’, ‘top’, and ‘is_empty’.

```

14 /* operation: declares a pointer to a stack and allocates      */
15 /*           enough memory so that the array 'items' in the  */
16 /*           stack will have 'max_size' usable elements.      */
17 /* preconditions: maximum amount of items that the stack will */
18 /*           be able to fit is passed to the function.        */
19 /* postconditions: the pointer to the stack is returned.       */
20 Stack *newStack(int max_size);
21
22 /* operation: pushes the float passed in onto the specified   */
23 /*           stack. The stack pointer is incremented.         */
24 /* preconditions: a pointer to a predeclared stack and a float*/
25 /*           are passed to the function.                      */
26 /* postconditions: stack will contain the float 'item' and the*/
27 /*           stackPointer within the stack holds the new*/
28 /*           float item's index in the 'items' array.         */
29 void push(Stack *theStack, const float item);
30
31 /* operation: pops and returns a float from the specified     */
32 /*           stack. The stack pointer is decremented.         */
33 /* preconditions: a pointer to a predeclared stack from which */
34 /*           a float will be obtained is passed in.           */
35 /* postconditions: float obtained is returned and stack        */
36 /*           pointer will point to next item, if any.          */
37 /*           If stack was empty when 'pop' is called,          */
38 /*           a null character '\0' is returned instead.        */
39 float pop(Stack *theStack);
40
41 /* operation: similar to the operation of the pop function    */
42 /*           but does not decrement the stack pointer.         */
43 /* preconditions: a pointer to a predeclared stack whose top   */
44 /*           item will be returned is passed in.               */
45 /* postconditions: float at the top of the stack (the last     */
46 /*           item pushed) is returned. If stack is             */
47 /*           empty, a null character '\0' is returned.         */
48 float top(Stack *theStack);
49
50 /* operation: checks the specified stack's stack pointer to   */
51 /*           see if the stack is empty and returns a value     */
52 /*           indicating the state of the stack.                */
53 /* preconditions: a pointer to a predeclared stack that needs  */
54 /*           to be checked is passed in.                      */
55 /* postconditions: 1 is returned if the stack is empty.        */
56 /*           0 is returned if the stack is not empty.          */
57 bool is_empty(Stack *theStack);
58
59 #endif // STACK_H_INCLUDED

```

‘StackFunctions.c’

The implementations for the functions in the header file are found in the ‘StackFunctions.c’ source file. At the top, two header files ‘stdlib.h’ and ‘stdbool.h’ were included for the indicated reasons and the stack header file ‘Stack.h’ was also included.

```
1 #include <stdlib.h> //for malloc()
2 #include <stdbool.h> //for bool, true, false
3 #include "Stack.h"
```

Once again, the function implementations for ‘push’ and ‘pop’ are identical to those found in previous tasks and so will not be discussed. The new function implementations are those for functions ‘newStack’, ‘top’, and ‘is.empty’.

The function ‘newStack’ declares a pointer to a ‘Stack’ and allocates memory using ‘malloc’. The argument for the memory allocation function is the size of the ‘Stack’ struct itself, which is made up of its stack pointer only, added to the standard size of a ‘float’ multiplied by the amount of items that the stack will be needed to store, at max. This means that enough memory is allocated so that the array ‘items’ within the stack struct will be capable of storing ‘max_size’ items. Next, the new stack’s stack pointer is initialized to -1. Finally, the declared pointer to a stack is returned.

```
5 Stack *newStack(const int max_size)
6 {
7     /*Pointer to a stack declared, memory allocated,*/
8     /*and the stack pointer is initialized to -1. */
9     Stack *theStack = malloc(sizeof(Stack) + max_size*sizeof(float));
10    (*theStack).stackPointer = -1;
11
12    /*Pointer to the stack is returned*/
13    return theStack;
14 }
```

The next new function implementation is that of the ‘top’ function, which has the task of returning the last item pushed on the stack, without modifying the stack pointer. The stack pointer of the stack passed to the function is checked to see if the stack is empty. If the stack is not empty, the item found at the index stored in the stack’s ‘stackPointer’ is returned. Otherwise, a null character ‘\0’ is returned, indicating that the stack is empty. Dereferencing using the ‘*’ symbol makes it possible to access the stack’s members from the pointer to a stack provided as the function parameter.

```
33 float top(Stack *theStack)
34 {
35     /*If stack not empty, return item at location indicated*/
36     /*by stack pointer. Stack pointer is not decremented. */
37     if((*theStack).stackPointer != -1)
38         return (*theStack).items[(*theStack).stackPointer];
39     else
40         return '\0'; //stack is empty
41 }
```

The final new function implementation for function 'is_empty' will now be discussed. This function makes use of the 'bool' type and the values 'true' and 'false' to indicate whether the stack passed to the function is empty or not. A conditional statement checks the value of the stack's 'stackPointer'. If this value is -1 , then the stack is empty and 'true' (which corresponds to the value 1 or any other non-zero value) is returned. Otherwise, 'false' (which corresponds to zero) is returned.

```
43 bool is_empty(Stack *theStack)
44 {
45     /*If stack pointer is -1, then the stack is empty, so*/
46     /*'true' is returned. Otherwise, 'false' is returned.*/
47     return (*theStack).stackPointer == -1 ? true : false;
48 }
```

Testing

No testing outputs will be presented for this subtask.

Task 4b

Introduction

The code for this task consists of six functions spread out across two source files. In this subtask, changes to the header file ‘Stack.h’ and source file ‘StackFunctions.c’ will be discussed. The knowledge to be able to implement the stack as a linked list was obtained through [10]. Below is a list showing the layout of the functions within the respective files.

- Task4_source.c
 - int main(void)
- StackFunctions.c
 - Stack *newStack(void)
 - void push(Stack *theStack, const float item)
 - float pop(Stack *theStack)
 - float top(Stack *theStack)
 - bool is_empty(Stack *theStack)
- Stack.h

Throughout the documentation of tasks 4b and 4c, since a stack was implemented as a linked list, the terms “stack” and “list” will be used interchangeably since they will be referring to the same structure, i.e. the stack implemented as a linked list.

‘Stack.h’

Starting off from the header file ‘Stack.h’, the ‘stack’ struct definition was fully changed and a new ‘node’ struct was defined. Firstly, it can be noted that ‘typedef’ was used with the actual definitions for the structs to immediately give the structs an alias. “Node” may be used instead of ‘struct node’ while “Stack” may be used instead of ‘struct node’. The stack struct now consists of two pointers to nodes, corresponding to the head and tail of the linked list. On the other hand, the new node struct consists of a ‘float’ item, and a pointer to the next node found in the list. One may appreciate the fact that the node structure has a pointer to a structure of the same type within it.

Kindly proceed to the next page for the code corresponding to the above description.

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #include <stdbool.h>
4
5  /*node structure definition with typedef*/
6  typedef struct node
7  {
8      float item;           //this node's item
9      struct node *nextNode; //pointer to next node
10 } Node;
11
12 /*stack structure definition with typedef*/
13 typedef struct stack
14 {
15     Node *listHead; //pointer to list start
16     Node *listTail; //pointer to list end
17 } Stack;

```

The function prototypes for four of the five functions remained unchanged from the previous subtask. The changed function prototype is that of 'newStack'. This function will now take no arguments since a maximum size will not be specified when a stack is being created. Rather, memory allocation will now be performed in the 'push' function for each item pushed. Changes in the prototype descriptions reflect the changes done in the function implementations to be discussed next.

```

19 /* operation: declares a pointer to a stack and allocates      */
20 /*                  memory for the head and tail pointers to nodes. */
21 /* preconditions: the function takes no arguments.             */
22 /* postconditions: the pointer to the stack is returned.       */
23 Stack *newStack(void);
24
25 /* operation: pushes the float passed in onto the specified   */
26 /*                  stack. The list will have a new tail node. */
27 /* preconditions: a pointer to a predeclared stack and a float*/
28 /*                  are passed to the function.               */
29 /* postconditions: stack will contain the float 'item' and the */
30 /*                  stackPointer within the stack holds the new */
31 /*                  float item's index in the 'items' array.   */
32 void push(Stack *theStack, const float item);
33
34 /* operation: pops and returns a float from the specified     */
35 /*                  stack. List's tail node is removed from the list*/
36 /* preconditions: a pointer to a predeclared stack from which */
37 /*                  a float will be obtained is passed in.    */
38 /* postconditions: float obtained is returned and stack       */
39 /*                  pointer will point to next item, if any.   */
40 /*                  If stack was empty when 'pop' is called,   */
41 /*                  a null character '\0' is returned instead. */
42 float pop(Stack *theStack);
43
44 /* operation: similar to the operation of the pop function   */
45 /*                  but does not remove items from the stack. */
46 /* preconditions: a pointer to a predeclared stack whose top  */
47 /*                  item will be returned is passed in.       */
48 /* postconditions: float at the tail of the list (the last    */
49 /*                  item pushed) is returned. If stack is     */
50 /*                  empty, a null character '\0' is returned.  */
51 float top(Stack *theStack);

```

```

52
53 /* operation: checks the specified stack's head node to see */
54 /*          see if the stack is empty and returns a value */
55 /*          indicating the state of the stack. */
56 /* preconditions: a pointer to a predeclared stack that needs */
57 /*          to be checked is passed in. */
58 /* postconditions: 1 is returned if the stack is empty. */
59 /*          0 is returned if the stack is not empty. */
60 bool is_empty(Stack *theStack);
61
62 #endif // STACK_H_INCLUDED

```

‘StackFunctions.c’

In the ‘StackFunctions.c’ source file, the functions were highly modified to reflect the new implementation of the stack as a linked list. The arguments and return values of all of the functions except for the ‘newStack’ function were unchanged from the previous tasks.

The ‘newStack’ function will no longer take an argument to specify the maximum size, in an aim to improve memory efficiency and reduce limits set by the program. In the function, a pointer to a stack is first declared and memory is allocated. The ‘malloc’ function is passed the size of the stack struct itself, in bytes, corresponding to the size of the two pointers found in the definition of this struct. Next, the ‘listHead’ and ‘listTail’ pointers in the stack declared are set to ‘NULL’ since the stack will initially be empty. The pointer to the declared stack is then returned.

```

1 #include <stdlib.h> //for malloc(), free(), and NULL
2 #include <stdbool.h> //for bool, true, false
3 #include "Stack.h"
4
5 Stack *newStack(void)
6 {
7     /*Pointer to a stack declared and memory allocated*/
8     Stack *theStack = malloc(sizeof(Stack));
9
10    /*List head and tail initialized to NULL -- since no items yet*/
11    (*theStack).listHead = NULL;
12    (*theStack).listTail = NULL;
13
14    /*Pointer to the stack is returned*/
15    return theStack;
16 }

```

The implementation for the ‘push’ function is next. A pointer to a new node is declared and memory is allocated based on the size of the node struct. Next, a condition checks if the stack argument is empty using the ‘is_empty’ function discussed further on. If the stack is empty, the stack’s list head is set to the new node. Otherwise, if the stack is not empty, the ‘nextNode’ of the node at the list tail is set to the new node. This is done by using the ‘*’ symbol twice for two dereferences. The first dereference is used to obtain ‘theStack’ and access the pointer ‘listTail’ stored in it. Since the

‘listTail’ is a pointer, a further dereference is used to access the node that ‘listTail’ points to (i.e. the tail node), so that the tail node’s ‘nextNode’ pointer can be accessed and set to the new node pointer.

The remaining steps in the function are common for both if the stack was empty and if it was not empty.

The stack’s list tail is set to the new node declared. This means that the list now has a new tail node. The ‘item’ variable of the new node is set to the item passed to the function, while the new node’s ‘nextNode’ is set to ‘NULL’ since this node is, at this point, the last node in the list.

```

18 void push(Stack *theStack, const float item)
19 {
20     /*Pointer to a new node declared*/
21     Node *newNode = malloc(sizeof(Node));
22
23     /*If the list is empty*/
24     if(is_empty(theStack))
25     {
26         /*New node set as head since the only item*/
27         (*theStack).listHead = newNode;
28     }
29     else //if the list is not empty
30     {
31         /*Current tail node's next node set to the new node.*/
32         ((*theStack).listTail).nextNode = newNode;
33     }
34     /*New node set as the list tail*/
35     (*theStack).listTail = newNode;
36
37     /*New node given an item and its next node set to NULL*/
38     (*newNode).item = item;
39     (*newNode).nextNode = NULL;
40 }

```

The function implementation for ‘pop’ will now be discussed. A variable is declared to store the item of type ‘float’ to be popped from the stack. Next, an if...else statement checks if the stack is empty, in which case a null character ‘\0’ will be returned to indicate this. Otherwise, a different procedure is taken. The item to be popped is obtained from the ‘item’ variable of the node found at the linked list’s tail (a pointer to which is found in the stack).

Next, continuing off from the fact that the stack is not empty, a nested if statement first checks if the list’s tail node is equal to the list’s head node. This would indicate that the list’s head and tail are the same node, and hence that the list (i.e. the stack) only contains one item. In this case, the pointer to the list’s tail found in ‘theStack’ is passed to the ‘free’ function to deallocate any memory previously allocated for this node. Note that since the head and tail are the same node, and hence they occupy the same memory, the ‘free’ can also be seen as implicitly freeing the memory of the head node. Next, the pointers in ‘theStack’ are set to ‘NULL’ to return the stack to the empty state since the pointers will no longer be pointing to nodes for the time being.


```

42 float pop(Stack *theStack)
43 {
44     /*Declaration of variable*/
45     float item; //stores popped item
46
47     /*If list is empty, return null character*/
48     if(is_empty(theStack))
49         return '\0';

```

Alternatively, if the list's tail node is not equal to the list's head node, two pointers to nodes are first declared. 'tailNode' will simply be used to make the code easier to follow and will point to the list's tail node, while the 'tempNode' will be used to store the pointer to the node found exactly before the tail node and will initially point to the list's head node. In fact, a loop was set up to loop until the node before the tail node is found. The condition checks if the 'nextNode' of 'tempNode' is not the 'tailNode'. If the condition is satisfied, 'tempNode' is set to the next node using the 'nextNode' pointer found within itself. Once the 'nextNode' is the 'tailNode', then the 'tempNode' is the node exactly before it and the loop terminates. The pointer 'tailNode' and the list's tail are now set to the node found exactly before the tail such that the list has effectively been shortened by one node. Hence, the 'nextNode' of the 'tailNode' is passed to the 'free' function to free the memory that was previously allocated for the list's tail. Finally, the 'nextNode' pointer of the 'tailNode' is set to 'NULL' since the new tail will not point to any further nodes.

At the end of the function, the item of type float popped is finally returned.

```

50     else
51     {
52         /*Obtain last node's item*/
53         item = ((*theStack).listTail).item;
54
55         /*If head is tail, stack only has one item*/
56         if((*theStack).listHead == (*theStack).listTail)
57         {
58             /*Free last node and make nodes NULL since no more nodes*/
59             free((*theStack).listTail); //head implicitly freed
60             (*theStack).listHead = (*theStack).listTail = NULL;
61         }
62         else //stack does not only have one item
63         {
64             /*Declaration of pointers to nodes*/
65             Node *tailNode = (*theStack).listTail; //makes code simpler
66             Node *tempNode = (*theStack).listHead;
67
68             /*Starting from the list head, and setting tempNode */
69             /*to next node until the next node is the tail node. */
70             while((*tempNode).nextNode != tailNode)
71                 tempNode = (*tempNode).nextNode;
72
73             /*Tail node set to node before it*/
74             (*theStack).listTail = tailNode = tempNode;
75

```

```
76         /*free last node and make it NULL*/
77         free((*tailNode).nextNode);
78         (*tailNode).nextNode = NULL;
79     }
80 }
81 /*Return item if stack was not empty*/
82 return item;
83 }
```

The next function to be discussed is the ‘top’ function. This function consists only of an if...else statement. A condition checks if the stack passed to the function is empty. If the stack is empty, a null character ‘\0’ is returned as an indication of this. Otherwise the item stored in the list tail’s node is returned, using a double dereferencing method also used in previously discussed functions. This involves first obtaining the stack, then using its ‘tailNode’ pointer to obtain the tail node stored at the location pointed to by ‘tailNode’, allowing the tail node’s ‘item’ to be accessed.

```
85 float top(Stack *theStack)
86 {
87     /*If list is empty, return null character*/
88     if(is_empty(theStack))
89         return '\0';
90     else //otherwise, return last node's item
91         return ((*theStack).listTail).item;
92 }
```

The final function implemented in the source file is the ‘is_empty’ function. This function returns the result of a condition which checks whether the list’s head node points to a node or not. If the ‘listHead’ is ‘NULL’, then the stack is empty and 1 is returned by the condition itself. Otherwise, the stack is not empty, and 0 is returned.

```
94 bool is_empty(Stack *theStack)
95 {
96     /*If list head is NULL, list is empty. */
97     /*Otherwise, the list is not empty.    */
98     return ((*theStack).listHead == NULL);
99 }
```

Testing

No testing outputs will be presented for this subtask.

Task 4c

Introduction

The code for this task consists of fourteen functions spread out across three source files. Most of the code in this task was obtained from tasks 3 and 4b. The files ‘Stack.h’ and ‘StackFunctions.c’ were obtained from task 4b. The files ‘ValAndComp.h’, ‘ValAndComp.c’, and ‘Task3_Source.c’ (renamed to ‘Task4_Source.c’) were obtained from task 3. Certainly, these files were modified to make use of the newly available functions and type definitions.

Considering the text file ‘expressions.txt’ required, similar to task 3, running the program from the Eclipse IDE required files to be in the task’s folder, while running the program from Command Prompt required files to be in the Debug folder within the task’s folder. Hence, the ‘expressions.txt’ file was placed both in the task’s folder and the Debug folder.

The ‘expressions.txt’ file was also copied to this task’s directory and contains the same expressions as used in task 3. Below is a list showing the layout of the functions within the respective files.

- Task4_source.c
 - int main(void)
 - void infixToPostfix(const char infixExp[], char postfixExp[])
 - float evaluateExpress(char postfixExp[])
 - int isOperator(const char a)
- StackFunctions.c
 - Stack *newStack(void)
 - void push(Stack *theStack, const float item)
 - float pop(Stack *theStack)
 - float top(Stack *theStack)
 - bool is_empty(Stack *theStack)
 - int pushNumber(struct stack *theStack, char express[], int index)
- ValAndComp.c
 - int expressionVal(const char express[])
 - int parenthesesVal(const char express[])
 - char charactersVal(const char express[])
 - int precedenceComp(const char a, const char b)
- Stack.h
- ValAndComp.h
- expressions.txt

‘Stack.h’ and ‘StackFunctions.c’

In the ‘Stack.h’ header file, the function prototype for function ‘pushNumber’ was copied from the task 3 ‘Stack.h’ header file and will not be discussed since it was not modified. Similarly, in the ‘StackFunctions.c’ source file, the corresponding implementation of the function ‘pushNumber’ was copied from the task 3 ‘StackFunctions.c’ source file, and will also not be discussed since no changes were done, given that this function does not make use of a ‘Stack’.

‘ValAndComp.h’ and ‘ValAndComp.c’

Moving on to the validation and comparison files, the ‘ValAndComp.h’ header file was not changed, but changes were done to the ‘parenthesesVal’ function in the ‘ValAndComp.c’ source file. Since memory allocation is handled by the stack functions and that there is no limit on the size of the stack due to dynamic memory allocation, the counting of open parentheses was removed and replaced by a simple declaration of a pointer to a ‘Stack’ using the ‘createStack’ function.

```
63  /*Declaring a pointer to a stack*/  
64  Stack *parenStack = newStack();
```

The second and last modification is the condition of an if statement at line 86. Previously, the stack pointer’s value for a stack would have been checked to see whether the stack is empty or not. Since a function ‘is_empty’ was implemented, it was used to check if the stack of parentheses is not empty.

```
85  /*Checking if stack still has parentheses*/  
86  if(!is_empty(parenStack))
```

‘Task4_source.c’

In the main source file ‘Task4_source.c’, similar modifications were done in the ‘infixToPostfix’ and ‘evaluateExpress’ functions. In both functions, the operators were previously counted to come up with a maximum amount of elements that the stack would be able to hold. Also, memory used to be allocated using ‘malloc’. Due to the memory allocation now handled by the ‘createStack’ function and the dynamism introduced to memory allocation, a simple pointer to a stack declaration replaced having to count operators and allocating memory explicitly from the main source file.

In the ‘infixToPostfix’ function, the new declaration is found at line 82...

```
81  /*Declaring a pointer to a stack*/  
82  Stack *opStack = newStack();
```

In the 'evaluateExpress' function, the new and similar declaration is found at line 172...

```
171  /*Declaring a pointer to a stack*/  
172  Stack *postfixStack = newStack();
```

The last modification is found in the 'infixToPostfix' function. The while loop starting from line 123 now uses the 'top' function from 'StackFunction.c' in the condition. Contrary to as was previously done, the character is not popped in the condition itself, but is popped in the first line in the loop after the condition verifies that the top of the stack is not a close parenthesis. In the final test of the loop's condition, if the character is a close parenthesis, it will not be popped since it is not needed. This means that when the loop terminates, the close parenthesis does not have to be pushed back since it was never popped.

```
121  /*if operator was found, add all higher-or-equal- */  
122  /*precedence operators to postfix expression      */  
123  while((ch=top(opStack)) != '(' && ch!='\0')  
124  {  
125      /*Pop the character that was checked*/  
126      pop(opStack);  
127      /*If not less precedence (i.e if higher or equal)*/  
128      if (precedenceComp(ch, infixExp[infixIndex]) != -1)  
129      {  
130          /*Insert popped operator and a space*/  
131          postfixExp[pofxIndex++] = ch;  
132          postfixExp[pofxIndex++] = ' ';  
133      }  
134      else  
135      {  
136          /*Push operator with less precedence*/  
137          push(opStack, ch);  
138          /*Append no more operators to the list*/  
139          break;  
140      }  
141  }
```

Testing

For testing purposes, one sample output and the corresponding 'answers.txt' contents will be shown. The inputs read from the file 'expressions.txt' can be seen from the screenshot itself found below. Kindly proceed to the next page for the sample output and the contents of the 'answers.txt' file.

```

"C:\Users\Miguel\Google Drive\Projects\CodeBlocks C Wor...
Expression read: 1+1
Evaluation completed.

Expression read: (1+2)*3+4/5-6*((0.5/2)+(3*5))
Evaluation completed.

Expression read: 0
Evaluation completed.

Expression read: InvalidInputTest
Invalid expression; illegal character 'I'.

Expression read: )(1+1)+1(
Invalid expression; parentheses mismatch.

Expression read: 3/3/3/3/3/3/3/3/3/3
Evaluation completed.

Expression read: 123.456*789.10/11.12-13+14
Evaluation completed.

Expression read: (((2*2)*2)*2)*2
Evaluation completed.

Expression read: 3*(3*(3*(0+1)))
Evaluation completed.

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.

```

Figure 16: Task 4c sample output

The results were written to the file 'answers.txt'. For the above set of accepted expressions, the contents of the 'answers.txt' file after execution are the lines shown below. For invalid inputs, the file does not contain any corresponding result.

```

1+1 = 2.0000
(1+2)*3+4/5-6*((0.5/2)+(3*5)) = -81.7000
0 = 0.0000
3/3/3/3/3/3/3/3/3/3 = 0.0002
123.456*789.10/11.12-13+14 = 8761.7129
(((2*2)*2)*2)*2 = 32.0000
3*(3*(3*(0+1))) = 27.0000

```

Task 5 Shared Libraries

Creating the shared library

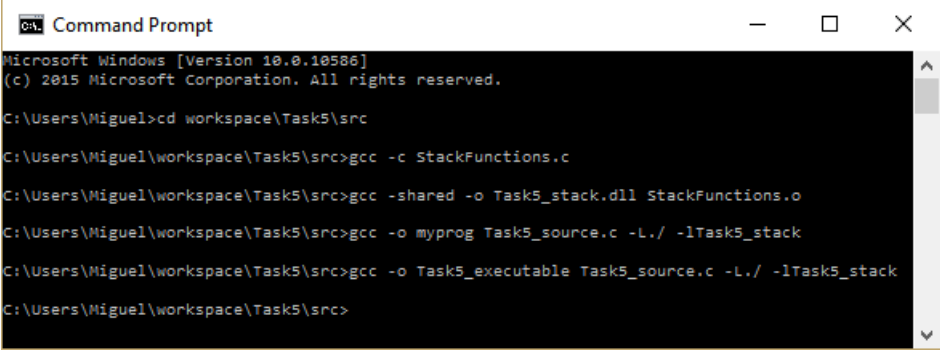
Below is a list of steps followed based on [11] to create the shared library.

1. A copy of the 'Task4b' folder was created and renamed to 'Task5'.
2. Using Command Prompt, the source folder 'src' in 'Task5' was navigated to.
3. The source file 'StackFunctions.c' was now compiled to object code using the command shown below. This resulted in a file 'StackFunctions.o' in the 'src' folder.

`"gcc -c Task5_source.c".`

4. The shared library was now built using the command shown below. This resulted in a file 'Task5_stack.dll' in the 'src' folder in 'Task5'. The Dynamically Linked Library was hence successfully created.

`"gcc -shared -o Task5_stack.dll Task5_source.o".`



```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Miguel>cd workspace\Task5\src

C:\Users\Miguel\workspace\Task5\src>gcc -c StackFunctions.c

C:\Users\Miguel\workspace\Task5\src>gcc -shared -o Task5_stack.dll StackFunctions.o

C:\Users\Miguel\workspace\Task5\src>gcc -o myprog Task5_source.c -L. -lTask5_stack

C:\Users\Miguel\workspace\Task5\src>gcc -o Task5_executable Task5_source.c -L. -lTask5_stack

C:\Users\Miguel\workspace\Task5\src>
```

Figure 17: Task 5 - Creating the shared library

Linking to the library and using the library

Next, below is a list of steps followed based on [11] to link to the shared library and to test the library using a simple ‘main’ function.

1. The source file ‘StackFunctions.c’ was deleted so that it is more clear that the ‘.dll’ file is being used.
2. A simple ‘main’ function in ‘Task5_source.c’ was implemented which includes use of the ‘createStack’, ‘push’, and ‘pop’ functions.
3. Linking to the ‘.dll’ file was done using the command below. This resulted in a file ‘Task5_executable.exe’ in the ‘src’ folder.

“gcc -o Task5_executable Task5_source.c -L./ -lTask5_stack”.

4. The executable file ‘Task5_executable.exe’ was now executed and the program successfully worked, as can be seen from the output obtained.

```

Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Miguel>cd workspace\Task5\src

C:\Users\Miguel\workspace\Task5\src>gcc -o Task5_executable Task5_source.c -L./ -lTask5_stack

C:\Users\Miguel\workspace\Task5\src>Task5_executable.exe
Pushing into stack...
Popping from stack...
Character popped: 'C'
C:\Users\Miguel\workspace\Task5\src>

```

Figure 18: Task 5 - Linking to and using the shared library

```

1 #include <stdio.h>
2 #include "Stack.h"
3 int main(void)
4 {
5     /*Declaring a pointer to a stack*/
6     Stack *testStack = newStack();
7
8     /*Pushing the character 'C' onto the stack*/
9     puts("Pushing into stack...");
10    push(testStack, 'C');
11
12    /*Popping the character 'C' from the stack*/
13    puts("Popping from stack...");
14    printf("Character popped: \'%c\'", (char) pop(testStack));
15
16    return 0;
17 }

```

Listing 3: Task 5 - ‘Task5_source.c’

6 Source code listing

6a Task1a

```

1 #include <stdio.h> //puts(), scanf(), printf(), getchar()
2 #define TAXRATE1 (float) 15/100
3 #define TAXRATE2 (float) 29/100
4 #define PAYRATE 22.50
5 #define OVERTIMERATE 1.5
6
7 /* operation: clears the input buffer */
8 /* preconditions: input buffer contains unneeded characters */
9 /* postconditions: input buffer is clear */
10 void emptyInputBuffer(void);
11
12 int main(void)
13 {
14     /*Declarations and initialization of variables*/
15     float hoursWorked=0; //amount of hours worked
16     float grossPay=0; //calculated grossPay
17     float tax=0; //calculated tax
18     float netPay=0; //calculated net pay
19
20     /*First output*/
21     puts("Please insert the number of hours worked in a week: ");
22
23     /*Loop until the input is accepted*/
24     while(!scanf("%f", &hoursWorked))
25     {
26         emptyInputBuffer(); //dump the input
27         puts("Invalid input; Insert the hours worked in a week: ");
28     }
29
30     /*Output indicating the input that was accepted*/
31     printf("\nInput \".2f\" accepted\n", hoursWorked);
32
33     /*Calculate gross pay*/
34     if(hoursWorked > 40)
35         grossPay = PAYRATE*(40+OVERTIMERATE*(hoursWorked-40));
36     else
37         grossPay = PAYRATE*hoursWorked;
38
39     /*Calculate tax*/
40     if(grossPay > 300)
41     {
42         if(grossPay > 480)
43             tax = (TAXRATE1*180)+(TAXRATE2*(grossPay-480));
44         else
45             tax = TAXRATE1*(grossPay-300);
46     }
47
48     /*netPay calculated by subtracting tax from grossPay*/
49     netPay = grossPay - tax;
50
51     /*Outputs to the user rounded to two decimal places*/
52     printf("-----\n");
53     printf("Gross pay: %10.2f Eur\n", grossPay);
54     printf("Tax pay: %10.2f Eur\n", tax);
55     printf("-----\n");
56     printf("Net pay: %10.2f Eur\n", netPay);
57     printf("-----\n");

```

```
58
59     return 0;
60 }
61
62 /*This function clears the input buffer*/
63 void emptyInputBuffer(void)
64 {
65     while(getchar() != '\n');
66 }
```

Listing 4: Task1a_source.c

6b Task1b

```
1 #include <stdio.h>    //printf(), getchar()
2 #include <stdbool.h>  //bool, true, false
3 #include <ctype.h>    //isalnum(), isalpha(), isupper()
4
5 /* operation: finds and returns the Highest Common Factor of two numbers*/
6 /* preconditions: num1 and num2 are two integers */
7 /* postconditions: highest common factor is returned */
8 int findHCF(const int num1, const int num2);
9
10 int main(void)
11 {
12     /*Declaration and initialization of variables*/
13     char ch;                //stores input characters
14     int letters=0, words=0;  //store amount of letters and words
15     int uppers=0, lowers=0;  //store amount of upper/lower-case letters
16     int HCF=0;              //stores the HCF of uppers and lowers
17     bool needNextWord = true; //indicates whether a word is being expected
18
19     /*First output*/
20     printf("Insert text (insert an EOF on a line by itself to stop):\n");
21
22     /*Loop until an EOF is read*/
23     while((ch = getchar()) != EOF)
24     {
25         /*if character is alphanumeric*/
26         if(isalnum(ch))
27         {
28             /*if a word was being expected, it is not exp-*/
29             /*ected anymore since a word's start was found*/
30             if(needNextWord)
31                 needNextWord = false;
32
33             /*if character is alphabetic (letter)*/
34             if(isalpha(ch))
35             {
36                 letters++;        //letter found
37
38                 /*Check case*/
39                 if(isupper(ch))
40                     uppers++;    //upper-case letter found
41                 else
42                     lowers++;    //lower-case letter found
43             }
44         }
45         else if(!needNextWord)
46         {
```

```

47         words++;           //complete word formed
48         needNextWord = true; //next word needed
49     }
50 }
51
52 /*Calculate HCF*/
53 HCF = findHCF(upper, lower);
54
55 /*Outputs*/
56 printf("Letters: %d\n", letters);
57 printf("Words:   %d\n", words);
58 printf("Uppers:  %d\n", upper);
59 printf("Lower:   %d\n", lower);
60 printf("Average letters per word: %.3f\n", words>0 ? ((float)letters/
61   words) : 0);
62 printf("Ratio of upper- to lower-case letters: %d:%d\n", upper/HCF,
63   lower/HCF);
64
65 return 0;
66 }
67
68 int findHCF(const int num1, const int num2)
69 {
70     int i=0; //loop counter
71
72     /*If both numbers are zero, HCF is 1. */
73     /*Otherwise, the HCF is calculated. */
74     if(num1==0 && num2==0)
75         return 1;
76     else
77     {
78         /*Loop from num1 down to 1*/
79         for(i=num1; i>0; i--)
80         {
81             /*if 'i' is a factor of both numbers*/
82             if(num1%i==0 && num2%i==0)
83                 return i; //return common factor
84         }
85     }
86
87     /*if num1 is 0, num2 is the HCF*/
88     return num2;
89 }

```

Listing 5: Task1b_source.c

6c Task1c

```

1 #include <stdio.h> //printf(), scanf(), putchar(), getchar()
2 #include <math.h> //log2()
3
4 /* operation: transforms num to its base-n equivalent and outputs result*/
5 /* preconditions: num is a base-10 number and n is a power-of-2 integer */
6 /* postconditions: the base-n equivalent of num is output to the user */
7 void to_base_n(int num, const int n);
8
9 /* operation: clears the input buffer */
10 /* preconditions: input buffer contains unneeded characters */
11 /* postconditions: input buffer is clear */
12 void emptyInputBuffer(void);

```

```

13
14 int main(void)
15 {
16     /*Declaration and initialization of variables*/
17     int inputVerif=0;    //stores return value of inputs
18     double log2base=0;  //stores log2 of the base
19     int number=0;       //stores the input decimal number
20     int base=0;         //stores the input integer base
21
22     /*infinitely loops until any input is negative*/
23     while(1)
24     {
25         /*Output and input (of number and base)*/
26         printf("Insert a decimal number and a power-of-2 integer ");
27         printf("base (or insert a negative number or base to quit): \n");
28         inputVerif = scanf("%d %d", &number, &base);
29
30         /*negative number or base taken as a 'quit'*/
31         if(number<0 || base<0)
32             break;
33
34         /*log2 of base is calculated*/
35         log2base = log2(base);
36
37         /*if input was accepted and if baseVerif is an integer*/
38         if(inputVerif==2 && (log2base) == (int)log2base)
39         {
40             printf("The base-%d representation is: ", base);
41
42             /*If number is zero, it remains zero */
43             /*Otherwise, it is converted to base-n*/
44             if(number == 0)
45                 putchar('0');           //result is zero
46             else
47                 to_base_n(number, base); //conversion to base_n
48         }
49         else //else if the input was rejected
50         {
51             printf("Invalid input; One/both of the inputs were invalid.");
52             emptyInputBuffer(); //clears input buffer
53         }
54         printf("\n\n");
55     }
56
57     return 0;
58 }
59
60 void to_base_n(int num, const int base)
61 {
62     /*If base is 1, print a stream of 1's*/
63     if(base == 1)
64     {
65         while(num-- != 0)
66             printf("%d", 1);
67         return;
68     }
69     else if(num == 0) //If no value left, return
70         return;
71
72     int remainder = num%base; //remainder calculated
73     to_base_n(num/base, base); //recursive call
74

```

```

75  /*If remainder is larger than 10, letters are used instead of digits*/
76  printf("%c", remainder < 10 ? remainder+48 : remainder+55);
77  }
78
79  /*This function clears the input buffer*/
80  void emptyInputBuffer(void)
81  {
82      while(getchar() != '\n');
83  }

```

Listing 6: Task1c_source.c

6d Task1d

```

1  #include <stdio.h> //printf(), fgets()
2  #include <string.h> //strlen()
3
4  #define MAXLEN 100 //maximum input size
5
6  /* operation: reverses the string passed to the function */
7  /* preconditions: a character array is passed to the function */
8  /* postconditions: the character array is modified in a way */
9  /* that the characters inside it are reversed */
10 void str_reverse(char[MAXLEN]);
11
12 int main(void) {
13     /*Declaration of variables*/
14     char input[MAXLEN]; //stores input string
15
16     /*User inputs a string*/
17     printf("Insert a string: ");
18     fgets(input, MAXLEN, stdin);
19     input[strlen(input)-1]='\0'; //removing the extra newline due to fgets
20     printf("Input:   \"%s\"\n", input);
21
22     /*Reversing the string*/
23     str_reverse(input);
24     printf("Reversed: \"%s\"", input);
25
26     return 0;
27 }
28
29 void str_reverse(char string[MAXLEN])
30 {
31     /*Declaration of variables*/
32     int size = strlen(string); //stores the string length
33     int index; //stores loop counter/index
34     char temp; //stores a char while swapping
35
36     /*String is reversed*/
37     for(index = 0; index < size/2; index++){
38         temp = string[index];
39         string[index] = string[size-1-index];
40         string[size-1-index] = temp;
41     }
42 }

```

Listing 7: Task1d_source.c

6e Task1e

```

1 #include <stdio.h> //printf(), put()
2 #include <stdlib.h> //srand(), rand()
3 #include <time.h> //time()
4
5 /* operation: fills the 'nums' array with random integers. */
6 /* preconditions: an integer array and a variable which shows */
7 /* its size are passed to the function. */
8 /* postconditions: the 'nums' array contains a set of random */
9 /* integers generated by the function. */
10 void randomArray(const int size, int nums[size]);
11
12 /* operation: sorts the integers in the 'nums' array using */
13 /* an insertion sort algorithm. */
14 /* preconditions: an integer array containing a set of poss- */
15 /* ibly unsorted integers and the array's size */
16 /* are passed to the function. */
17 /* postconditions: the integer array will now contain a set */
18 /* of sorted integer values. */
19 void naive_sort(const int size, int nums[size]);
20
21 /* operation: outputs the integers found in an integer array. */
22 /* preconditions: an integer array and its size are passed to */
23 /* the function. */
24 /* postconditions: the integers in the array are output to */
25 /* the user in an easy to read form. */
26 void printArray(const int size, const int nums[size]);
27
28 int main(void)
29 {
30     /*Declaration and initialization*/
31     srand(time(NULL)); //number generator seeded
32     const int ARRAYSIZE = (rand()%30) + 1; //random size between 1 and 30
33     int numbers[ARRAYSIZE]; //holds numbers to be sorted
34
35     /*Array given random integers*/
36     randomArray(ARRAYSIZE, numbers); //array given random integers
37
38     puts("Original: ");
39     printArray(ARRAYSIZE, numbers); //output array
40     naive_sort(ARRAYSIZE, numbers); //sort array
41     puts("Sorted: ");
42     printArray(ARRAYSIZE, numbers); //output array
43
44     return 0;
45 }
46
47 void randomArray(const int size, int nums[size])
48 {
49     int index; //stores array index in the loop
50
51     /*Each element is given a random integer value*/
52     for(index=0; index < size; index++)
53         nums[index] = rand() % 1000; //random integer between 0 and 999
54 }
55
56 void naive_sort(const int size, int nums[size])
57 {
58     /*Declaration of variables*/
59     int index1; //outer loop index
60     int index2; //inner loop index

```

```

61     int tempInt;        //stores value of index1'th element
62
63     /*Main sort loop*/
64     for(index1=1; index1 < size; index1++)
65     {
66         tempInt = nums[index1]; //current element stored
67         index2 = index1;        //index copied to leave index1 unchanged
68
69         /*Inner sort loop*/
70         while(index2 > 0 && nums[index2-1] > tempInt)
71         {
72             nums[index2] = nums[index2-1]; //element shifted to the right
73             index2--;
74         }
75         nums[index2]=tempInt; //vacated element taken up by tempInt
76     }
77 }
78
79 void printArray(const int size, const int nums[size])
80 {
81     /*Declaration of variable*/
82     int index; //stores index for loop
83
84     /*Output*/
85     printf("{");
86     for(index=0; index < size-1; index++)
87         printf("%d, ", nums[index]);
88     printf("%d}\n", nums[size-1]);
89 }

```

Listing 8: Task1e_source.c

6f Task1f

```

1  #include <stdio.h>    //printf()
2  #include <stdlib.h>   //srand(), rand()
3  #include <time.h>     //time()
4
5  /* operation: fills the 'nums' array with random integers.    */
6  /* preconditions: an integer array and a variable which shows */
7  /*                its size are passed to the function.        */
8  /* postconditions: the 'nums' array contains a set of random   */
9  /*                integers generated by the function.         */
10 void randomArray(const int size, int nums[size]);
11
12 /* operation: sorts the integers in the 'nums' array using    */
13 /*                a quick sort algorithm.                      */
14 /* preconditions: an integer array containing a set of poss-   */
15 /*                ibly unsorted integers and the array's size  */
16 /*                are passed to the function.                  */
17 /* postconditions: the integer array will now contain a set   */
18 /*                of sorted integer values.                    */
19 void smarter_sort(int nums[], const int start, const int end);
20
21 /* operation: outputs the integers found in an integer array  */
22 /* preconditions: an integer array and its size are passed to */
23 /*                the function.                                  */
24 /* postconditions: the integers in the array are output to    */
25 /*                the user in an easy to read form.          */
26 void printArray(const int size, const int nums[size]);

```

```
27
28 int main(void)
29 {
30     /*Declaration and initialization*/
31     srand(time(NULL)); //number generator seeded
32     const int ARRAYSIZE = (rand()%30) + 5; //random size between 1 and 30
33     int numbers[ARRAYSIZE]; //array of numbers to be sorted
34
35     /*Array given random integers*/
36     randomArray(ARRAYSIZE, numbers); //array given random integers
37
38     printf("Original:\n");
39     printArray(ARRAYSIZE, numbers); //output array
40     smarter_sort(numbers, 0, ARRAYSIZE-1); //sort array
41     printf("Sorted:\n");
42     printArray(ARRAYSIZE, numbers); //output array
43
44     return 0;
45 }
46
47 void randomArray(const int size, int nums[size])
48 {
49     int index; //stores index/counter for the loop
50
51     /*Each element is given a random integer value*/
52     for(index=0; index < size; index++)
53         nums[index] = rand() % 1000; //random integer between 0 and 999
54 }
55
56 void smarter_sort(int nums[], const int start, const int end)
57 {
58     /*Declaration and initialization of variables*/
59     int left = start; //index of element smaller than pivot
60     int right = end; //index of element larger than pivot
61     int pivot = nums[(end+start)/2]; //reference number
62     int temp; //used when swapping two elements
63
64     /*Main loop*/
65     while(left <= right)
66     {
67         /*Move to the right until value larger */
68         /*than or equal to pivot is found. */
69         while(nums[left] < pivot)
70             left++;
71         /*Move to the left until value smaller */
72         /*than or equal to pivot is found. */
73         while(nums[right] > pivot)
74             right--;
75
76         /*If 'left' exceeds 'right', then this */
77         /*section of the array has been ordered*/
78         if(left > right)
79             break;
80
81         /*Swapping numbers at 'left' and 'right'*/
82         temp = nums[left];
83         nums[left] = nums[right];
84         nums[right] = temp;
85
86         /*Moving on to the next elements */
87         /*in the list to be considered. */
88         left++;
```



```

89     right--;
90 }
91 /*If more than one value to the left of 'left'*/
92 if((left-1) > start)
93     smarter_sort(nums, start, left-1); //sort left part
94 /*If at least one value to the right of 'left'*/
95 if(left < end)
96     smarter_sort(nums, left, end);    //sort right part
97 }
98
99 void printArray(const int size, const int nums[size])
100 {
101     /*Declaration of variable*/
102     int index; //stores index for loop
103
104     /*Output*/
105     printf("{");
106     for(index=0; index < size-1; index++)
107         printf("%d, ", nums[index]);
108     printf("%d}\n", nums[size-1]);
109 }

```

Listing 9: Task1f_source.c

6g Task1g

```

1  #include <stdio.h> //printf(), puts(), getchar()
2  #include <stdlib.h> //srand(), rand()
3  #include <time.h> //time()
4
5  #include "Quick.h"
6  #include "Insertion.h"
7
8  /* operation: fills the 'nums' array with random integers. */
9  /* preconditions: an integer array and a variable which shows */
10 /* its size are passed to the function. */
11 /* postconditions: the 'nums' array contains a set of random */
12 /* integers generated by the function. */
13 void randomArray(const int size, int nums[size]);
14
15 /* operation: outputs the integers found in an integer array. */
16 /* preconditions: an integer array and its size are passed to */
17 /* the function. */
18 /* postconditions: the integers in the array are output to */
19 /* the user in an easy to read form. */
20 void printArray(const int size, const int nums[size]);
21
22 /* operation: checks that the integers in an array are sorted */
23 /* in an ascending order. */
24 /* preconditions: an integer array and its size are passed to */
25 /* the function. */
26 /* postconditions: an output indicates whether the sort was */
27 /* verified or found to be incorrect. */
28 void tester(const int size, const int nums[size]);
29
30 int main(void)
31 {
32     do
33     {
34         /*Declaration and initialization*/

```

```

35     srand(time(NULL)); //number generator seeded
36     const int ARRAYSIZE = (rand()%30)+1; //random size from 1 to 30
37     int numbers[ARRAYSIZE]; //array to be sorted
38
39     /*Insertion sort*/
40     randomArray(ARRAYSIZE, numbers); //array given random integers
41     printf("Original before insertion sort:\n");
42     printArray(ARRAYSIZE, numbers); //print array
43     naive_sort(ARRAYSIZE, numbers); //sort array
44     printf("Sorted by insertion sort:\n");
45     printArray(ARRAYSIZE, numbers); //print array
46     tester(ARRAYSIZE, numbers); //verify sort
47
48     printf("\n");
49
50     /*Quick sort*/
51     randomArray(ARRAYSIZE, numbers); //array given random
    integers
52     printf("Original before quick sort:\n");
53     printArray(ARRAYSIZE, numbers); //print array
54     smarter_sort(numbers, 0, ARRAYSIZE-1); //sort array
55     printf("Sorted by quick sort:\n");
56     printArray(ARRAYSIZE, numbers); //print array
57     tester(ARRAYSIZE, numbers); //verify sort
58
59     puts("\nPress ENTER to do another pair of sorts or type 'q' to
    quit...");
60 }
61 while(getchar() != 'q');
62
63 return 0;
64 }
65
66 void randomArray(const int size, int nums[size])
67 {
68     int index; //stores index/counter for the loop
69
70     /*Each element is given a random integer value*/
71     for(index=0; index < size; index++)
72         nums[index] = rand() % 1000; //random integer between 0 and 999
73 }
74
75 void printArray(const int size, const int nums[size])
76 {
77     /*Declaration of variable*/
78     int index; //stores index for loop
79
80     /*Output*/
81     printf("(");
82     for(index=0; index < size-1; index++)
83         printf("%d, ", nums[index]);
84     printf("%d)\n", nums[size-1]);
85 }
86
87 void tester(const int size, const int nums[size])
88 {
89     /*Declaration of variable*/
90     int index; //stores array counter/index
91
92     /*Tester loop*/
93     for(index=0; index < size-1; index++)
94     {

```

```

95     /*Check if a number is followed by a smaller one*/
96     if(nums[index] > nums[index+1])
97     {
98         printf("Sort incorrect.\n");
99         return;
100     }
101 }
102 /*Since no number was followed by a smaller one */
103 /*the array is hence sorted and was verified.  */
104 printf("Sort verified.\n");
105 }

```

Listing 10: Task1g_source.c

```

1  #ifndef INSERTION_H_INCLUDED
2  #define INSERTION_H_INCLUDED
3
4  /* operation: sorts the integers in the 'nums' array using      */
5  /*              an insertion sort algorithm.                    */
6  /* preconditions: an integer array containing a set of poss-    */
7  /*              ibly unsorted integers and the array's size    */
8  /*              are passed to the function.                    */
9  /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */
11 void naive_sort(const int size, int nums[size]);
12
13 #endif //INSERTION_H_INCLUDED

```

Listing 11: Insertion.h

```

1  #ifndef QUICK_H_INCLUDED
2  #define QUICK_H_INCLUDED
3
4  /* operation: sorts the integers in the 'nums' array using      */
5  /*              a quick sort algorithm.                          */
6  /* preconditions: an integer array containing a set of poss-    */
7  /*              ibly unsorted integers and the array's size    */
8  /*              are passed to the function.                    */
9  /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */
11 void smarter_sort(int nums[], const int start, const int end);
12
13 #endif //QUICK_H_INCLUDED

```

Listing 12: Quick.h

```

1  #include "Insertion.h"
2
3  void naive_sort(const int size, int nums[size])
4  {
5      /*Declaration of variables*/
6      int index1;        //outer loop index
7      int index2;        //inner loop index
8      int tempInt;       //stores value of index1'th element
9
10     /*Main sort loop*/
11     for(index1=1; index1 < size; index1++)
12     {
13         tempInt = nums[index1]; //current element stored
14         index2 = index1;

```

```

15
16     /*Inner sort loop*/
17     while(index2 > 0 && nums[index2-1] > tempInt)
18     {
19         nums[index2] = nums[index2-1]; //element shifted to the right
20         index2--;
21     }
22     nums[index2]=tempInt; //vacated element taken up by tempInt
23 }
24

```

Listing 13: Insertion.c

```

1  #include "Quick.h"
2
3  void smarter_sort(int nums[], const int start, const int end)
4  {
5      /*Declaration and initialization of variables*/
6      int left = start; //index of element smaller than pivot
7      int right = end;  //index of element larger than pivot
8      int pivot = nums[(end+start)/2]; //reference number
9      int temp;         //used when swapping two elements
10
11     /*Main loop*/
12     while(left <= right)
13     {
14         /*Move to the right until value larger */
15         /*than or equal to pivot is found.    */
16         while(nums[left] < pivot)
17             left++;
18         /*Move to the left until value smaller */
19         /*than or equal to pivot is found.    */
20         while(nums[right] > pivot)
21             right--;
22
23         /*If 'left' exceeds 'right', then this */
24         /*section of the array has been ordered*/
25         if(left > right)
26             break;
27
28         /*Swapping numbers at 'left' and 'right'*/
29         temp = nums[left];
30         nums[left] = nums[right];
31         nums[right] = temp;
32
33         /*Moving on to the next elements */
34         /*in the list to be considered. */
35         left++;
36         right--;
37     }
38     /*If more than one value to the left of 'left'*/
39     if((left-1) > start)
40         smarter_sort(nums, start, left-1); //sort left part
41     /*If at least one value to the right of 'left'*/
42     if(left < end)
43         smarter_sort(nums, left, end);     //sort right part
44 }

```

Listing 14: Quick.c

6h Task1h

```

1 #include <stdio.h> //printf(), puts(), getchar()
2 #include <stdlib.h> //srand(), rand(), malloc(), free()
3 #include <time.h> //clock(), clock_t, CLOCKS_PER_SEC
4
5 #include "Quick.h"
6 #include "Insertion.h"
7
8 /* operation: fills the 'nums' array with random integers. */
9 /* preconditions: an integer array and a variable which shows */
10 /* its size are passed to the function. */
11 /* postconditions: the 'nums' array contains a set of random */
12 /* integers generated by the function. */
13 void randomArray(const int size, int nums[size]);
14
15 /* operation: checks that the integers in an array are sorted */
16 /* preconditions: an integer array and its size are passed to */
17 /* the function. */
18 /* postconditions: an output indicates whether the sort was */
19 /* verified or found to be incorrect. */
20 void tester(const int size, const int nums[size]);
21
22 /* operation: calculates and outputs the time passed from a */
23 /* starting time indicated by the argument and the */
24 /* current time. */
25 /* preconditions: a pre-calculated starting time is passed to */
26 /* the function. */
27 /* postconditions: an output indicates the time passed from */
28 /* 'start' till the current time. */
29 void calcTime(const clock_t start);
30
31 int main(void)
32 {
33     /*Declaration and initialization*/
34     clock_t start; //holds the starting time for process timings
35     srand ( time(NULL) ); //random number generator seeded
36     const int SIZE[] = {10,100,1000,5000,10000};
37
38     int index;
39     for(index = 0; index < 5; index++)
40     {
41         /*Pointer to an integer is allocated memory for elements*/
42         int *numbers = malloc(SIZE[index]*sizeof(int));
43
44         /*Insertion sort*/
45         printf("Insertion sort with array of size %d;\n", SIZE[index]);
46         randomArray(SIZE[index], numbers); //array given random integers
47         start = clock(); //record starting time
48         naive_sort(SIZE[index], numbers); //sort array
49         calcTime(start); //calculate time taken
50         tester(SIZE[index], numbers); //verify sort
51
52         printf("\n");
53
54         /*Quick sort*/
55         printf("Quick sort with array of size %d;\n", SIZE[index]);
56         randomArray(SIZE[index], numbers); //array given random integers
57         start = clock(); //record starting time
58         smarter_sort(numbers, 0, SIZE[index]-1); //sort array
59         calcTime(start); //calculate time taken
60         tester(SIZE[index], numbers); //verify sort

```

```

61
62     if(index != 4)
63     {
64         puts("\nPress ENTER to do the next pair of sorts...");
65         while(getchar() != '\n'); //remove all extra characters
66     }
67
68     /*Memory allocated is freed*/
69     free(numbers);
70 }
71 return 0;
72 }
73
74 void randomArray(const int size, int nums[size])
75 {
76     int index; //stores index/counter for the loop
77
78     /*Each element is given a random integer value*/
79     for(index=0; index < size; index++)
80         nums[index] = rand() % 1000; //random integer between 0 and 999
81 }
82
83 void tester(const int size, const int nums[size])
84 {
85     /*Declaration of variable*/
86     int index; //stores array counter/index
87
88     /*Tester loop*/
89     for(index=0; index < size-1; index++)
90     {
91         /*Check if a number is followed by a smaller one*/
92         if(nums[index] > nums[index+1])
93         {
94             printf("Sort incorrect.\n");
95             return;
96         }
97     }
98     /*Since no number was followed by a smaller one */
99     /*the array is hence sorted and was verified. */
100    printf("Sort verified.\n");
101 }
102
103 void calcTime(const clock_t start)
104 {
105     /*Calculating the difference*/
106     clock_t diff = clock() - start;
107     /*Outputting the difference*/
108     printf("Time taken: %lf msec. ", diff*1000.0/CLOCKS_PER_SEC);
109 }

```

Listing 15: Task1h_source.c

```

1  #ifndef INSERTION_H_INCLUDED
2  #define INSERTION_H_INCLUDED
3
4  /* operation: sorts the integers in the 'nums' array using      */
5  /*              an insertion sort algorithm.                    */
6  /* preconditions: an integer array containing a set of poss-    */
7  /*              ibly unsorted integers and the array's size */
8  /*              are passed to the function.                    */
9  /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */

```

```

11 void naive_sort(const int size, int nums[size]);
12
13 #endif //INSERTION_H_INCLUDED

```

Listing 16: Insertion.h

```

1 #ifndef QUICK_H_INCLUDED
2 #define QUICK_H_INCLUDED
3
4 /* operation: sorts the integers in the 'nums' array using      */
5 /*              a quick sort algorithm                          */
6 /* preconditions: an integer array containing a set of poss-    */
7 /*              ibly unsorted integers and the array's size */
8 /*              are passed to the function.                    */
9 /* postconditions: the integer array will now contain a set    */
10 /*              of sorted integer values.                      */
11 void smarter_sort(int nums[], const int start, const int end);
12
13 #endif //QUICK_H_INCLUDED

```

Listing 17: Quick.h

```

1 #include "Insertion.h"
2
3 void naive_sort(const int size, int nums[size])
4 {
5     /*Declaration of variables*/
6     int index1;    //outer loop index
7     int index2;    //inner loop index
8     int tempInt;   //stores value of index1'th element
9
10    /*Main sort loop*/
11    for(index1=1; index1 < size; index1++)
12    {
13        tempInt = nums[index1]; //current element stored
14        index2 = index1;
15
16        /*Inner sort loop*/
17        while(index2 > 0 && nums[index2-1] > tempInt)
18        {
19            nums[index2] = nums[index2-1]; //element shifted to the right
20            index2--;
21        }
22        nums[index2]=tempInt; //vacated element taken up by tempInt
23    }
24 }

```

Listing 18: Insertion.c

```

1 #include "Quick.h"
2
3 #include "Quick.h"
4
5 void smarter_sort(int nums[], const int start, const int end)
6 {
7     /*Declaration and initialization of variables*/
8     int left = start; //index of element smaller than pivot
9     int right = end;  //index of element larger than pivot
10    int pivot = nums[(end+start)/2]; //reference number
11    int temp;           //used when swapping two elements

```

```

12
13  /*Main loop*/
14  while(left <= right)
15  {
16      /*Move to the right until value larger */
17      /*than or equal to pivot is found.    */
18      while(nums[left] < pivot)
19          left++;
20      /*Move to the left until value smaller */
21      /*than or equal to pivot is found.    */
22      while(nums[right] > pivot)
23          right--;
24
25      /*If 'left' exceeds 'right', then this */
26      /*section of the array has been ordered*/
27      if(left > right)
28          break;
29
30      /*Swapping numbers at 'left' and 'right'*/
31      temp = nums[left];
32      nums[left] = nums[right];
33      nums[right] = temp;
34
35      /*Moving on to the next elements */
36      /*in the list to be considered.  */
37      left++;
38      right--;
39  }
40  /*If more than one value to the left of 'left'*/
41  if((left-1) > start)
42      smarter_sort(nums, start, left-1); //sort left part
43  /*If at least one value to the right of 'left'*/
44  if(left < end)
45      smarter_sort(nums, left, end);    //sort right part
46  }

```

Listing 19: Quick.c

6i Task1i

```

1  #include <stdio.h> //printf()
2  #define STOPS 10   //maximum stops
3
4  /* operation: comes up with a total stray cats values from the*/
5  /*              cat counts in the array passed in.          */
6  /* preconditions: array of stray cats along with the stop to */
7  /*              start counting back from are passed in.      */
8  /* postconditions: the return value is the total amount of    */
9  /*              stray cats up to the stop indicated by the */
10 /*              value of 'stops' passed in the first call. */
11 int catCount(const int strayCats[STOPS], const int stops);
12
13 int main(void)
14 {
15     /*Declaration and initialization*/
16     int strayCats[STOPS] = {12, 5, 3, 20, 15, 6, 7, 1, 19, 30};
17     /*Final Output*/
18     printf("Hence, a total of %d stray cats were observed.\n", catCount(
19         strayCats, STOPS));

```



```

20     return 0;
21 }
22
23 int catCount(const int strayCats[STOPS], const int stop)
24 {
25     /*Declaration of variables*/
26     int count; //holds the total up to a stop
27
28     /*Base case - If this is the first stop, value is first element*/
29     if(stop == 1)
30     {
31         printf("\nThe total up to stop 1 is: %d; ", strayCats[0]);
32         printf("Sending this data to stop 2...\n");
33         return strayCats[0];
34     }
35     else /*else, request value of previous stop*/
36     {
37         /*If this is not the final stop*/
38         if(stop != STOPS)
39         {
40             printf("For the total up to stop %d, the ", stop);
41             printf("total up to stop %d is required.\n", stop-1);
42         }
43         else
44         {
45             printf("For the total up to the final stop, ");
46             printf("the total up to stop %d is required.\n", stop-1);
47         }
48
49         /*Add current value with count up to previous stop*/
50         count = strayCats[stop-1] + catCount(strayCats, stop-1);
51
52         /*If this is not the final stop*/
53         if(stop != STOPS)
54         {
55             printf("The total up to stop %d is: %d; ", stop, count);
56             printf("Sending this data to stop %d...\n", stop+1);
57         }
58         else
59         {
60             printf("The total up to the final stop ");
61             printf("at the terminus is: %d.\n", count);
62         }
63
64         /*The total up to this stop is returned*/
65         return count;
66     }
67 }

```

Listing 20: Task1i.source.c

6j Task1j

```

1 #include <stdio.h> //printf(), puts(), scanf(), getchar()
2 #define COINS 8 //amount of unique coins
3 #define CACHESIZE 50 //maximum cached results
4
5 /* operation: calculates the the minimum number of coins that */
6 /* make up the 'change' passed in. */
7 /* preconditions: the set of different types of coins and the */

```

```

8  /*          change to be given are passed in.          */
9  /* postconditions: minimum amount of coins that make up the */
10 /*          change is returned.          */
11 int computeChange(const int coins[COINS], const int change);
12
13 int main(void)
14 {
15     /*Declaration and initialization*/
16     const int coins[COINS] = {1,2,5,10,20,50,100,200};
17     int index=0;           //stores array index in loop
18     int changeInput=0;     //stores the user input
19     int coinsResult=0;     //stores result for a specific change
20     int cacheUsed=0;       //indicates whether cache was used
21     int cacheCounter=0;    //location of next cache space to use
22     int cache[CACHESIZE][2]; //storage of past results
23
24     /*Clearing the cache*/
25     for(index=0; index < CACHESIZE; index++)
26         cache[index][0] = -1;
27
28     /*Loops until change input is negative*/
29     do
30     {
31         /*Input of change*/
32         puts("What is the required change in cents? (-ve to quit)");
33         while(!scanf("%d", &changeInput))
34         {
35             printf("Invalid value; insert an integer change...\n");
36             while(getchar()!='\n'); //clear the input buffer
37         }
38
39         /*If changeInput is negative, this is taken as a 'quit'*/
40         if(changeInput < 0)
41             break;
42
43         /*Check cache for past results*/
44         cacheUsed = 0;
45         for(index=0; index < CACHESIZE; index++)
46         {
47             if(cache[index][0]==changeInput)
48             {
49                 coinsResult = cache[index][1];
50                 cacheUsed = 1;
51                 break;
52             }
53         }
54
55         /*Compute the amount of coins for 'changeInput'*/
56         if(!cacheUsed)
57         {
58             /*Change is calculated*/
59             coinsResult = computeChange(coins, changeInput);
60
61             /*Storing in cache*/
62             cache[cacheCounter][0] = changeInput;
63             cache[cacheCounter][1] = coinsResult;
64
65             /*Cache counter incremented and reset */
66             /*if the cache limit was reached.      */
67             cacheCounter++;
68             if(cacheCounter == CACHESIZE)
69                 cacheCounter=0;

```

```

70     }
71
72     /*Final output*/
73     printf("For %d cents change, a minimum of %d coin/s is needed.\n\n",
74           changeInput, coinsResult);
75 }
76 while(1);
77
78 return 0;
79 }
80
81 int computeChange(const int coins[COINS], const int change)
82 {
83     /*Declaration of variables*/
84     int index;        //used as a loop index/counter
85     int subChange;    //stores change from 0 to 'change'
86     int coinCount;    //stores the amount of coins
87     int pastValues[change+1]; //used as a cache
88
89     /*Main loop*/
90     for(subChange=0; subChange <= change; subChange++)
91     {
92         /*Maximum amount of coins is 'subChange' 1-cent coins*/
93         coinCount = subChange;
94
95         /*Calculating the minimum coin count*/
96         for(index=0; index<COINS; index++)
97         {
98             /*If the change is equal to a coin only one coin is required*/
99             if(subChange==coins[index])
100             {
101                 coinCount = 1;
102                 break;
103             }
104             else if(subChange>coins[index]
105                     && 1+pastValues[subChange-coins[index]] < coinCount)
106             {
107                 coinCount = 1+pastValues[subChange-coins[index]];
108             }
109         }
110         /*Storing the count in the cache*/
111         pastValues[subChange] = coinCount;
112     }
113     /*Return result of minimum amount of coins*/
114     return pastValues[change];
115 }

```

Listing 21: Task1j_source.c

6k Task2a

```

1 #include <stdio.h>        //for puts() and fgets()
2 #include <stdlib.h>       //for exit()
3 #include "Validation.h"   //for parentheses validation
4 #include "Stack.h"        //for EXPLEN
5
6 int main(void)
7 {
8     /*Declaration of array to hold input infix expression*/

```

```

9   char express[EXPLEN];
10
11  /*Loops until expression is valid or user quits*/
12  do
13  {
14      /*Input of expression*/
15      puts("Insert an arithmetic expression (q to quit): ");
16      fgets(express, EXPLEN, stdin); //input of infix expression
17
18      /*Check if user inserted a 'q'*/
19      if(express[0] == 'q')
20          exit(0); //user inserted a 'q' so quit
21
22  } while(!parenthesesVal(express)); //repeat until expression is valid
23  puts("Expression parentheses validated.");
24
25  return 0;
26 }

```

Listing 22: Task2_source.c

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #define EXPLEN 100
4
5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[EXPLEN];
10 };
11
12 /* operation: pushes the float passed in onto the specified
13    stack. The stack pointer is incremented. */
14 /* preconditions: a pointer to a predeclared stack and a float*/
15 /*                  are passed to the function. */
16 /* postconditions: stack will contain the float 'item' and the*/
17 /*                  stackPointer within the stack holds the new*/
18 /*                  float item's index in the 'items' array. */
19 void push(struct stack *theStack, const float item);
20
21 /* operation: pops and returns a float from the specified
22    stack. The stack pointer is decremented. */
23 /* preconditions: a pointer to a predeclared stack from which
24    a float will be obtained is passed in. */
25 /* postconditions: float obtained is returned and stack
26    pointer will point to next item, if any. */
27 /*                  If stack was empty when 'pop' was called,
28    a null character '\0' is returned instead. */
29 float pop(struct stack *theStack);
30
31 #endif // STACK_H_INCLUDED

```

Listing 23: Stack.h

```

1  #ifndef VALANDCOMP_H_INCLUDED
2  #define VALANDCOMP_H_INCLUDED
3
4  /* operation: the parentheses, if any, in the expression found*/
5  /*                  in 'express' are checked for mismatches. */
6  /* preconditions: array containing expression to be checked is*/

```

```

7  /*          passed to the function.          */
8  /* postconditions: '0' is returned in the case of a mismatch. */
9  /*          '1' returned if the expression is valid.      */
10 int parenthesesVal(const char express[]);
11
12 #endif // VALANDCOMP_H_INCLUDED

```

Listing 24: Validation.h

```

1  #include "Stack.h"
2
3  void push(struct stack *theStack, const float item)
4  {
5      /*Stack pointer incremented and item stored at new location*/
6      (*theStack).stackPointer++;
7      (*theStack).items[(*theStack).stackPointer] = item;
8  }
9
10 float pop(struct stack *theStack)
11 {
12     /*If stack not empty, return item at location indicated*/
13     /*by stack pointer. Stack pointer is then decremented */
14     if((*theStack).stackPointer != -1)
15         return (*theStack).items[(*theStack).stackPointer--];
16     else
17         return '\0'; //stack is empty
18 }

```

Listing 25: StackFunctions.c

```

1  #include <stdio.h> //for puts()
2  #include <string.h> //for strlen()
3  #include "Stack.h" //for stack
4
5  int parenthesesVal(const char express[])
6  {
7      /*Declaration of variables*/
8      int expressLength = strlen(express)-1; // -1 to exclude newline
9      int index=0; //array index for expression
10
11     /*declaring a stack*/
12     struct stack parenStack;
13     parenStack.stackPointer = -1; //initializing the stackPointer
14
15     /*pushing and popping parentheses*/
16     for(index=0; index < expressLength; index++)
17     {
18         /*Push any open parentheses and pop an open */
19         /*parenthesis when a close parenthesis is found */
20         if(express[index] == '(')
21             push(&parenStack, '(');
22         else if(express[index] == ')')
23         {
24             /*If stack is empty, function returns. */
25             /*Otherwise, an open parentheses is popped*/
26             if(pop(&parenStack) == '\0')
27             {
28                 puts("Invalid expression; parentheses mismatch.");
29                 return 0;
30             }
31         }
32     }
33 }

```

```

32     }
33
34     /*Checking if stack still has parentheses*/
35     if(parenStack.stackPointer != -1)
36     {
37         puts("Invalid expression; parentheses mismatch.");
38         return 0;
39     }
40
41     /*Expression passed parentheses check*/
42     return 1;
43 }

```

Listing 26: Validation.c

61 Task2b

```

1  #include <stdio.h>           //for puts() and fgets()
2  #include <stdlib.h>         //for exit()
3  #include <ctype.h>          //for isdigit()
4  #include <string.h>         //for strlen()
5  #include "ValAndComp.h"     //for parentheses validation
6  #include "Stack.h"          //for stack, stack functions, and EXPLEN
7
8  /* operation: Converts the infix expression passed in to a      */
9  /* postfix expression.                                           */
10 /* preconditions: The infix expression is stored in 'infixExp' */
11 /*                and an additional array 'postfixExp' is      */
12 /*                passed to store the resultant postfix exp-   */
13 /*                resion without having to return anything.    */
14 /* postconditions: The 'postfixExp' array will now hold the     */
15 /*                postfix equivalent of the passed expression */
16 void infixToPostfix(const char infixExp[], char postfixExp[]);
17
18 /* operation: Checks whether the character passed to the      */
19 /* function is an operator (+, -, /, or *).                  */
20 /* preconditions: the character to be checked is passed to    */
21 /* the function.                                              */
22 /* postconditions: 1 is returned the 'a' is an operator.      */
23 /*                0 is returned the 'a' is not an operator.  */
24 int isOperator(const char a);
25
26 int main(void)
27 {
28     /*Declaration of array to hold input infix expression*/
29     char express[EXPLEN];
30
31     /*Loops until expression is valid or user quits*/
32     do
33     {
34         /*Input of expression*/
35         puts("Insert an arithmetic expression (q to quit): ");
36         fgets(express, EXPLEN, stdin); //input of infix expression
37
38         /*Check if user inserted a 'q'*/
39         if(express[0] == 'q')
40             exit(0); //user inserted a 'q' so quit
41
42     } while(!parenthesesVal(express)); //repeat until expression is valid
43

```

```

44  /*Infix to postfix conversion*/
45  char postfixExp[2*strlen(express)]; //array declared
46  infixToPostfix(express, postfixExp); //expression converted
47  printf("Postfix: \"%s\\n\"", postfixExp);
48
49  return 0;
50 }
51
52 void infixToPostfix(const char infixExp[], char postfixExp[])
53 {
54     /*Declaration and initialization*/
55     int infixIndex=0; //array index for infixExp and loop counter
56     int pofxIndex=0; //array index for postfixExp
57     char ch; //stores popped stack items
58     int expressLength = strlen(infixExp)-1; //-1 to exclude newline
59
60     /*Declaring a stack*/
61     struct stack opStack;
62     opStack.stackPointer = -1; //initializing the stackPointer
63
64     /*Main loop - goes through expression one character at a time*/
65     for(infixIndex=0; infixIndex < expressLength; infixIndex++)
66     {
67         /*If digit, read all further digits*/
68         if(isdigit(infixExp[infixIndex]))
69         {
70             /*Loop while the next character is a digit or decimal point*/
71             do
72             {
73                 postfixExp[pofxIndex++] = infixExp[infixIndex++];
74             }while(isdigit(infixExp[infixIndex])
75                 || infixExp[infixIndex]=='.'');
76
77             /*Insert a space for a new postfix element to come after*/
78             postfixExp[pofxIndex++] = ' ';
79             /*Index adjusted since an extra character was read*/
80             infixIndex--; //since an extra character was read
81         }
82         else if(infixExp[infixIndex] == '(')
83         {
84             /*Left parentheses simply pushed*/
85             push(&opStack, '(');
86         }
87         else if(infixExp[infixIndex] == ')')
88         {
89             /*For right parentheses, append operators to */
90             /*the postfix expression. */
91             while((ch=pop(&opStack)) != '(')
92             {
93                 /*Insert popped operator and a space*/
94                 postfixExp[pofxIndex++] = ch;
95                 postfixExp[pofxIndex++] = ' ';
96             }
97         }
98         else if(isOperator(infixExp[infixIndex]))
99         {
100             /*if operator was found, add all higher-or-equal- */
101             /*precedence operators to postfix expression */
102             while((ch=pop(&opStack)) != '(' && ch!='\0')
103             {
104                 /*If not less precedence (i.e if higher or equal)*/
105                 if(precedenceComp(ch, infixExp[infixIndex]) != -1)

```

```

106         {
107             /*Insert popped operator and a space*/
108             postfixExp[pofxIndex++] = ch;
109             postfixExp[pofxIndex++] = ' ';
110         }
111         else
112         {
113             /*Push operator with less precedence*/
114             push(&opStack, ch);
115             /*Append no more operators to the list*/
116             break;
117         }
118     }
119
120     /*Open parenthesis pushed back*/
121     if(ch == '(')
122         push(&opStack, '(');
123
124     /*Original operator now pushed*/
125     push(&opStack, infixExp[infxIndex]);
126 }
127
128 /*If character did not satisfy a condition, the character*/
129 /*is simply skipped since it is not a useful character. */
130 }
131
132 /*Pop remaining operators*/
133 while((ch=pop(&opStack)) != '\0')
134 {
135     /*Insert popped operator and a space*/
136     postfixExp[pofxIndex++] = ch;
137     postfixExp[pofxIndex++] = ' ';
138 }
139
140 /*Last character set to a null character */
141 postfixExp[pofxIndex-1] = '\0'; //exclude last space
142 }
143
144 int isOperator(const char a)
145 {
146     /* 1 is returned the 'a' is an operator.      */
147     /* 0 is returned the 'a' is not an operator.  */
148     return (a == '+' || a == '-' || a == '*' || a == '/');
149 }

```

Listing 27: Task2_source.c

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #define EXPLEN 100
4
5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[2*EXPLEN];
10 };
11
12 /* operation: pushes the float passed in onto the specified */
13 /*              stack. The stack pointer is incremented.      */
14 /* preconditions: a pointer to a predeclared stack and a float*/
15 /*              are passed to the function.                  */

```



```

16 /* postconditions: stack will contain the float 'item' and the*/
17 /*                stackPointer within the stack holds the new*/
18 /*                float item's index in the 'items' array. */
19 void push(struct stack *theStack, const float item);
20
21 /* operation: pops and returns a float from the specified */
22 /*                stack. The stack pointer is decremented. */
23 /* preconditions: a pointer to a predeclared stack from which */
24 /*                a float will be obtained is passed in. */
25 /* postconditions: float obtained is returned and stack */
26 /*                pointer will point to next item, if any. */
27 /*                If stack was empty when 'pop' was called, */
28 /*                a null character '\0' is returned instead. */
29 float pop(struct stack *theStack);
30
31 #endif // STACK_H_INCLUDED

```

Listing 28: Stack.h

```

1 #ifndef VALANDCOMP_H_INCLUDED
2 #define VALANDCOMP_H_INCLUDED
3
4 /* operation: the parentheses, if any, in the expression found*/
5 /*                in 'express' are checked for mismatches. */
6 /* preconditions: array containing expression to be checked is*/
7 /*                passed to the function. */
8 /* postconditions: '0' is returned in the case of a mismatch. */
9 /*                '1' returned if the expression is valid. */
10 int parenthesesVal(const char express[]);
11
12 /* operation: compares the precedence of two operators from */
13 /*                '+', '-', '*', and '/' */
14 /* preconditions: characters 'a' and 'b' each contain an */
15 /*                an operator. These will be compared together*/
16 /* postconditions: '1' is returned if precedence of 'a' is */
17 /*                greater than that of 'b' */
18 /*                '0' is returned if precedence of 'a' is */
19 /*                equal to that of 'b' */
20 /*                '-1' is returned if precedence of 'a' is */
21 /*                less than that of 'b' */
22 int precedenceComp(const char a, const char b);
23
24 #endif // VALANDCOMP_H_INCLUDED

```

Listing 29: ValAndComp.h

```

1 #include "Stack.h"
2
3 void push(struct stack *theStack, const float item)
4 {
5     /*Stack pointer incremented and item stored at new location*/
6     (*theStack).stackPointer++;
7     (*theStack).items[(*theStack).stackPointer] = item;
8 }
9
10 float pop(struct stack *theStack)
11 {
12     /*If stack not empty, return item at location indicated*/
13     /*by stack pointer. Stack pointer is then decremented */
14     if((*theStack).stackPointer != -1)
15         return (*theStack).items[(*theStack).stackPointer--];

```

```

16     else
17         return '\0'; //stack is empty
18 }

```

Listing 30: StackFunctions.c

```

1  #include <stdio.h> //for puts()
2  #include <string.h> //for strlen()
3  #include "Stack.h" //for stack
4
5  int parenthesesVal(const char express[])
6  {
7      /*Declaration of variables*/
8      int expressLength = strlen(express)-1; //-1 to exclude newline
9      int index=0; //array index for expression
10
11     /*declaring a stack*/
12     struct stack parenStack;
13     parenStack.stackPointer = -1; //initializing the stackPointer
14
15     /*pushing and popping parentheses*/
16     for(index=0; index < expressLength; index++)
17     {
18         /*Push any open parentheses and pop an open */
19         /*parenthesis when a close parenthesis is found */
20         if(express[index] == '(')
21             push(&parenStack, '(');
22         else if(express[index] == ')')
23         {
24             /*If stack is empty, function returns. */
25             /*Otherwise, an open parentheses is popped*/
26             if(pop(&parenStack) == '\0')
27             {
28                 puts("Invalid expression; parentheses mismatch.");
29                 return 0;
30             }
31         }
32     }
33
34     /*Checking if stack still has parentheses*/
35     if(parenStack.stackPointer != -1)
36     {
37         puts("Invalid expression; parentheses mismatch.");
38         return 0;
39     }
40
41     /*Expression passed parentheses check*/
42     return 1;
43 }
44
45 int precedenceComp(const char a, const char b)
46 {
47     /*If 'a' is * or /... */
48     /*...if 'b' is * or /, operators have equal precedence */
49     /*...if 'b' is + or -, then 'a' has higher precedence */
50     /*If 'a' is + or -... */
51     /*...if 'b' is * or /, then 'a' has lower precedence */
52     /*...if 'b' is + or -, operators have equal precedence */
53     return (a=='*' || a=='/') - (b=='*' || b=='/');
54 }

```

Listing 31: Validation.c

6m Task2c

```

1 #include <stdio.h>           //for puts() and fgets()
2 #include <stdlib.h>          //for exit()
3 #include <ctype.h>           //for isdigit()
4 #include <string.h>          //for strlen()
5 #include <stdbool.h>         //for bool, true, and false
6 #include "ValAndComp.h"      //for expression validation
7 #include "Stack.h"           //for stack, stack functions, and EXPLEN
8
9 /* operation: Converts the infix expression passed in to a      */
10 /* postfix expression.                                          */
11 /* preconditions: The infix expression is stored in 'infixExp' */
12 /* and an additional array 'postfixExp' is                      */
13 /* passed to store the resultant postfix exp-                  */
14 /* resion without having to return anything.                    */
15 /* postconditions: The 'postfixExp' array will now hold the     */
16 /* postfix equivalent of the passed expression*/
17 void infixToPostfix(const char infixExp[], char postfixExp[]);
18
19 /* operation: Evaluates the postfix expression passed in and   */
20 /* returns the final answer.                                    */
21 /* preconditions: The array passed in contains an expression   */
22 /* in postfix notation.                                         */
23 /* postconditions: The answer of type float is returned.       */
24 float evaluateExpress(char postFixExp[]);
25
26 /* operation: Checks whether the character passed to the      */
27 /* function is an operator (+, -, /, or *).                    */
28 /* preconditions: the character to be checked is passed to    */
29 /* the function.                                               */
30 /* postconditions: 1 is returned the 'a' is an operator.      */
31 /* 0 is returned the 'a' is not an operator.                  */
32 int isOperator(const char a);
33
34 int main(void)
35 {
36     /*Declaration of array to hold input infix expression*/
37     char express[EXPLEN];
38
39     /*Loops until user quits*/
40     do
41     {
42         /*Loops until expression is valid or user quits*/
43         do
44         {
45             /*Input of expression*/
46             puts("Insert an arithmetic expression (q to quit): ");
47             fgets(express, EXPLEN, stdin); //input of infix expression
48
49             /*Check if user inserted a 'q'*/
50             if(express[0] == 'q')
51                 exit(0); //user inserted a 'q' so quit
52         }
53         while(!expressionVal(express)); //repeat until expression is valid
54
55         /*Infix to postfix conversion*/
56         char postfixExp[2*strlen(express)]; //array declared
57         infixToPostfix(express, postfixExp); //expression converted
58         printf("Postfix: \"%s\"\n", postfixExp);
59     }
60 }

```

```

61     /*Evaluation and postfix expression and output of answer*/
62     printf("Answer: %.4f\n\n", evaluateExpress(postfixExp));
63 }
64 while(1);
65
66 return 0;
67 }
68
69 void infixToPostfix(const char infixExp[], char postfixExp[])
70 {
71     /*Declaration and initialization*/
72     int infxIndex=0; //array index for infixExp and loop counter
73     int pofxIndex=0; //array index for postfixExp
74     char ch; //stores popped stack items
75     int expressLength = strlen(infixExp)-1; //-1 to exclude newline
76
77     /*Declaring a stack*/
78     struct stack opStack;
79     opStack.stackPointer = -1; //initializing the stackPointer
80
81     /*Main loop - goes through expression one character at a time*/
82     for(infxIndex=0; infxIndex < expressLength; infxIndex++)
83     {
84         /*If digit, read all further digits*/
85         if(isdigit(infixExp[infxIndex]))
86         {
87             /*Loop while the next character is a digit or decimal point*/
88             do
89             {
90                 postfixExp[pofxIndex++] = infixExp[infxIndex++];
91             }
92             while(isdigit(infixExp[infxIndex])
93                 || infixExp[infxIndex]=='.'');
94
95             /*Insert a space for a new postfix element to come after*/
96             postfixExp[pofxIndex++] = ' ';
97             /*Index adjusted since an extra character was read*/
98             infxIndex--; //since an extra character was read
99         }
100         else if(infixExp[infxIndex] == '(')
101         {
102             /*Left parentheses simply pushed*/
103             push(&opStack, '(');
104         }
105         else if(infixExp[infxIndex] == ')')
106         {
107             /*For right parentheses, append operators to */
108             /*the postfix expression. */
109             while((ch=pop(&opStack)) != '(')
110             {
111                 /*Insert popped operator and a space*/
112                 postfixExp[pofxIndex++] = ch;
113                 postfixExp[pofxIndex++] = ' ';
114             }
115         }
116         else if(isOperator(infixExp[infxIndex]))
117         {
118             /*if operator was found, add all higher-or-equal- */
119             /*precedence operators to postfix expression */
120             while((ch=pop(&opStack)) != '(' && ch!='\0')
121             {
122                 /*If not less precedence (i.e if higher or equal)*/

```

```

123         if (precedenceComp(ch, infixExp[infixIndex]) != -1)
124         {
125             /*Insert popped operator and a space*/
126             postfixExp[pofxIndex++] = ch;
127             postfixExp[pofxIndex++] = ' ';
128         }
129         else
130         {
131             /*Push operator with less precedence*/
132             push(&opStack, ch);
133             /*Append no more operators to the list*/
134             break;
135         }
136     }
137
138     /*Open parenthesis pushed back*/
139     if (ch == '(')
140         push(&opStack, '(');
141
142     /*Original operator now pushed*/
143     push(&opStack, infixExp[infixIndex]);
144 }
145
146 /*If character did not satisfy a condition, the character*/
147 /*is simply skipped since it is not a useful character. */
148 }
149
150 /*Pop remaining operators*/
151 while ((ch=pop(&opStack)) != '\0')
152 {
153     /*Insert popped operator and a space*/
154     postfixExp[pofxIndex++] = ch;
155     postfixExp[pofxIndex++] = ' ';
156 }
157
158 /*Last character set to a null character */
159 postfixExp[pofxIndex-1] = '\0'; //exclude last space
160 }
161
162 float evaluateExpress(char postfixExp[])
163 {
164     /*Declaration and initialization*/
165     int index=0;           //array index for postfixExp
166     float op1, op2;        //store values in arithmetic operations
167     float answer=0.0;      //stores answer in arithmetic operations
168     bool operatorFound=false; //indicates that at least one operator found
169
170     /*Declaring a stack*/
171     struct stack postfixStack;
172     postfixStack.stackPointer = -1; //initializing the stackPointer
173
174     /*Loops until the end of the postfix expression is reached*/
175     while (postfixExp[index] != '\0')
176     {
177         /*Spaces are skipped*/
178         if (postfixExp[index] == ' ')
179             index++;
180         else
181         {
182             /*If first digit of a number was found, pushNumber invoked.*/
183             /*Else if an operator is found, perform an operation. */
184             if (isdigit(postfixExp[index]))

```

```

185     {
186         /*Index set to first character after the number*/
187         index = pushNumber(&postfixStack, postfixExp, index);
188     }
189     else if (isOperator(postfixExp[index]))
190     {
191         /*If this is the first operator found, set to true*/
192         if (!operatorFound)
193             operatorFound=true;
194
195         /*Two operands popped from stack*/
196         op2 = pop(&postfixStack);
197         op1 = pop(&postfixStack);
198
199         /*Performing operation according to operator*/
200         switch(postfixExp[index])
201         {
202             case '+':
203                 answer = op1+op2;
204                 break;
205             case '-':
206                 answer = op1-op2;
207                 break;
208             case '*':
209                 answer = op1*op2;
210                 break;
211             case '/':
212                 answer = op1/op2;
213                 break;
214         }
215
216         /*sub-answer pushed back onto the stack*/
217         push(&postfixStack, answer);
218         printf("Sub-expression: %10.4f %c %10.4f = %10.4f\n",
219             op1, postfixExp[index], op2, answer);
220         index++;
221     }
222     else //character is neither a digit nor an operand
223     {
224         puts("Error in evaluation");
225         return 0;
226     }
227 }
228 }
229
230 /*If no operators were found, answer is the number in the stack*/
231 /*Otherwise, answer of last sub-expression is returned*/
232 if (!operatorFound)
233     return pop(&postfixStack);
234 else
235     return answer;
236 }
237
238 int isOperator(const char a)
239 {
240     /* 1 is returned the 'a' is an operator.      */
241     /* 0 is returned the 'a' is not an operator.  */
242     return (a == '+' || a == '-' || a == '*' || a == '/');
243 }

```

Listing 32: Task2_source.c

```

1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3 #define EXPLEN 100
4
5 /*stack structure definition*/
6 struct stack
7 {
8     int stackPointer;
9     float items[2*EXPLEN];
10 };
11
12 /* operation: pushes the float passed in onto the specified */
13 /* stack. The stack pointer is incremented. */
14 /* preconditions: a pointer to a predeclared stack and a float */
15 /* are passed to the function. */
16 /* postconditions: stack will contain the float 'item' and the */
17 /* stackPointer within the stack holds the new */
18 /* float item's index in the 'items' array. */
19 void push(struct stack *theStack, const float item);
20
21 /* operation: pops and returns a float from the specified */
22 /* stack. The stack pointer is decremented. */
23 /* preconditions: a pointer to a predeclared stack from which */
24 /* a float will be obtained is passed in. */
25 /* postconditions: float obtained is returned and stack */
26 /* pointer will point to next item, if any. */
27 /* If stack was empty when 'pop' was called, */
28 /* a null character '\0' is returned instead. */
29 float pop(struct stack *theStack);
30
31 /* operation: pushes a number found in the postfixExp array */
32 /* starting from the location indicated by 'index'. */
33 /* Characters from this location onwards are pushed */
34 /* onto the specified stack until a character which */
35 /* is not a digit, decimal point, or negative sign */
36 /* is reached in the array. */
37 /* preconditions: a stack to push the number onto, the array */
38 /* containing the number, and the index at */
39 /* which the number starts are all passed in. */
40 /* postconditions: the stack now contains the number. The index */
41 /* of the array location containing the first */
42 /* character found after the number is returned.*/
43 int pushNumber(struct stack *theStack, char postfixExp[], int index);
44
45 #endif // STACK_H_INCLUDED

```

Listing 33: Stack.h

```

1 #ifndef VALANDCOMP_H_INCLUDED
2 #define VALANDCOMP_H_INCLUDED
3
4 /* operation: the infix expression 'express' is checked for */
5 /* mismatches in parentheses, illegal characters, */
6 /* and for the case where the user input an empty */
7 /* expression. Also, if expression size limit was */
8 /* reached, user is warned about incorrect results.*/
9 /* preconditions: array containing expression to be checked is*/
10 /* passed to the function. */
11 /* postconditions: '0' is returned if the expression broke one*/
12 /* of the guidelines specified above; '1' is */
13 /* returned if the expression is valid. Not */

```

```

14 /*          Note that if the size limit was exceeded, */
15 /*          expression is still considered to be valid.*/
16 int expressionVal(char express[]);
17
18 /* operation: compares the precedence of two operators from */
19 /*          '+', '-', '*', and '/' */
20 /* preconditions: characters 'a' and 'b' each contain an */
21 /*          an operator. These will be compared together*/
22 /* postconditions: '1' is returned if precedence of 'a' is */
23 /*          greater than that of 'b' */
24 /*          '0' is returned if precedence of 'a' is */
25 /*          equal to that of 'b' */
26 /*          '-1' is returned if precedence of 'a' is */
27 /*          less than that of 'b' */
28 int precedenceComp(char a, char b);
29
30 #endif // VALANDCOMP_H_INCLUDED

```

Listing 34: ValAndComp.h

```

1 #include <ctype.h> //for isdigit()
2 #include <stdlib.h> //for atof()
3 #include "Stack.h"
4
5 void push(struct stack *theStack, const float item)
6 {
7     /*Stack pointer incremented and item stored at new location*/
8     (*theStack).stackPointer++;
9     (*theStack).items[(*)theStack).stackPointer] = item;
10 }
11
12 float pop(struct stack *theStack)
13 {
14     /*If stack not empty, return item at location indicated*/
15     /*by stack pointer. Stack pointer is then decremented */
16     if((*theStack).stackPointer != -1)
17         return (*theStack).items[(*)theStack).stackPointer--];
18     else
19         return '\0'; //stack is empty
20 }
21
22 int pushNumber(struct stack *theStack, char express[], int index)
23 {
24     /*Declaration of variables*/
25     int tempIndex; //stores array index
26     int numCount=0; //stores amount of number characters
27
28     /*Push all characters in 'express' starting from 'index' */
29     /*until a character which is not a digit, a decimal point,*/
30     /*or a negative sign is found.*/
31     tempIndex = index;
32     while(isdigit(express[tempIndex])
33         || express[tempIndex]=='.'
34         || express[tempIndex]=='-')
35     {
36         numCount++;
37         tempIndex++;
38     }
39
40     /*Declaring array for number*/
41     char number[numCount+1];
42

```



```

43  /*Sub-array for operand created*/
44  for(tempIndex=0; tempIndex < numCount; tempIndex++)
45      number[tempIndex] = express[index++];
46  number[tempIndex] = '\0';
47
48  /*Push number onto stack*/
49  push(theStack, atof(number));
50  /*Index of first character found after the number is returned*/
51  return index;
52 }

```

Listing 35: StackFunctions.c

```

1  #include <stdio.h> //for puts()
2  #include <string.h> //for strlen()
3  #include <ctype.h> //for isdigit()
4  #include "Stack.h" //for stack
5
6  /* operation: the parentheses, if any, in the expression found*/
7  /*          in 'express' are checked for mismatches.          */
8  /* preconditions: array containing expression to be checked is*/
9  /*                passed to the function.                      */
10 /* postconditions: '0' is returned in the case of a mismatch. */
11 /*                '1' returned if the expression is valid.    */
12 int parenthesesVal(const char express[]);
13
14 /* operation: the characters which make up the expression in */
15 /*          'express' are checked for illegal characters.      */
16 /*          Legal: '+', '-', '*', '/', '.', '(', and ')'       */
17 /* preconditions: array containing expression to be checked is*/
18 /*                passed to the function.                      */
19 /* postconditions: First illegal character found is returned. */
20 /*                Otherwise, a null character '\0' returned. */
21 char charactersVal(const char express[]);
22
23 int expressionVal(const char express[])
24 {
25     /*Character used for return of function charactersVal*/
26     char ch;
27
28     /*A series of checks run*/
29     /*0 returned : expression failed at least one check*/
30     /*1 returned : expression passed all checks*/
31     if( !parenthesesVal(express) ) //parentheses check
32     {
33         puts("Invalid expression; parentheses mismatch.");
34         return 0;
35     }
36     else if( (ch=charactersVal(express)) ) //character check
37     {
38         printf("Invalid expression; illegal character '%c'.\n", ch);
39         return 0;
40     }
41     else if(express[0] == '\n') //expression presence check
42     {
43         puts("Invalid expression; nothing was input.");
44         return 0;
45     }
46
47     /*Expression still considered to be valid if expression limit*/
48     /*was reached. Program might still crash if limit reached. */
49     if(strlen(express) >= EXPLEN-1) //expression length limit check

```

```

50     puts("Warning: answer might be incorrect; expression length limit
    reached.");
51
52     /*Expression passed all checks*/
53     return 1;
54 }
55
56 int parenthesesVal(const char express[])
57 {
58     /*Declaration of variables*/
59     int expressLength = strlen(express)-1; //-1 to exclude newline
60     int index=0; //array index for expression
61
62     /*Declaring a stack*/
63     struct stack parenStack;
64     parenStack.stackPointer = -1; //initializing the stackPointer
65
66     /*Pushing and popping parentheses*/
67     for(index=0; index < expressLength; index++)
68     {
69         /*Push any open parentheses and pop an open */
70         /*parenthesis when a close parenthesis is found */
71         if(express[index] == '(')
72             push(&parenStack, '(');
73         else if(express[index] == ')')
74         {
75             /*If stack is empty, function returns. */
76             /*Otherwise, an open parentheses is popped*/
77             if(pop(&parenStack) == '\0')
78                 return 0;
79         }
80     }
81
82     /*Checking if stack still has parentheses*/
83     if(parenStack.stackPointer != -1)
84         return 0;
85
86     /*Expression passed parentheses check*/
87     return 1;
88 }
89
90 char charactersVal(const char express[])
91 {
92     /*Declaration of variables*/
93     int expressLength = strlen(express)-1; //-1 to exclude newline
94     int index=0; //array index for expression
95
96     /*Loops through all characters of the expression*/
97     for(index=0; index < expressLength; index++)
98     {
99         /*If a character is not any of these characters, */
100         /*then it is considered to be forbidden. */
101         if( !isdigit(express[index]) && express[index] != '+'
102             && express[index] != '-' && express[index] != '*'
103             && express[index] != '/' && express[index] != '.'
104             && express[index] != '(' && express[index] != ')')
105         {
106             /*Illegal character returned*/
107             return express[index];
108         }
109     }
110     /*No illegal characters found - expression passed character check*/

```

```

111     return '\0';
112 }
113
114 int precedenceComp(const char a, const char b)
115 {
116     /*If 'a' is * or /... */
117     /*...if 'b' is * or /, operators have equal precedence */
118     /*...if 'b' is + or -, then 'a' has higher precedence */
119     /*If 'a' is + or -... */
120     /*...if 'b' is * or /, then 'a' has lower precedence */
121     /*...if 'b' is + or -, operators have equal precedence */
122     return (a=='*' || a=='/') - (b=='*' || b=='/');
123 }

```

Listing 36: Validation.c

6n Task3

```

1 #include <stdio.h>           //for puts(), fgets(), fprintf(), fopen(), fclose
    ()
2 #include <stdlib.h>         //for exit(), malloc(), and free()
3 #include <ctype.h>          //for isdigit()
4 #include <string.h>         //for strlen(), strcat()
5 #include <stdbool.h>        //for bool, true, false
6 #include "ValAndComp.h"    //for expression validation
7 #include "Stack.h"         //for stack, stack functions, and EXPLEN
8
9 /* operation: Converts the infix expression passed in to a      */
10 /* postfix expression.                                          */
11 /* preconditions: The infix expression is stored in 'infixExp' */
12 /* and an additional array 'postfixExp' is                      */
13 /* passed to store the resultant postfix exp-                  */
14 /* resion without having to return anything.                  */
15 /* postconditions: The 'postfixExp' array will now hold the     */
16 /* postfix equivalent of the passed expression*/
17 void infixToPostfix(const char infixExp[], char postfixExp[]);
18
19 /* operation: Evaluates the postfix expression passed in and   */
20 /* returns the final answer.                                    */
21 /* preconditions: The array passed in contains an expression   */
22 /* in postfix notation.                                         */
23 /* postconditions: The answer of type float is returned.       */
24 float evaluateExpress(char postfixExp[]);
25
26 /* operation: Checks whether the character passed to the      */
27 /* function is an operator (+, -, /, or *).                   */
28 /* preconditions: the character to be checked is passed to    */
29 /* the function.                                              */
30 /* postconditions: 1 is returned the 'a' is an operator.      */
31 /* 0 is returned the 'a' is not an operator.                  */
32 int isOperator(const char a);
33
34 int main(void)
35 {
36     /*Declaration of file pointers and opening files*/
37     FILE *expFp = fopen("expressions.txt", "r");
38     FILE *ansFp = fopen("answers.txt", "w");
39
40     /*Declaration of array to hold input infix expression*/
41     char express[EXPLEN];

```

```

42
43 while(fgets(express, EXPLEN, expFp))
44 {
45     /*Newline character added to last expression*/
46     if(express[strlen(express)-1] != '\n')
47         strcat(express, "\n");
48
49     /*Output indicates expression read*/
50     printf("\nExpression read: %s", express);
51
52     /*If expression is validated, condition is satisfied*/
53     if(expressionVal(express))
54     {
55         /*Infix to postfix conversion*/
56         char postfixExp[2*strlen(express)]; //array declared
57         infixToPostfix(express, postfixExp); //expression converted
58
59         /*Evaluation of expressions and output to file*/
60         express[strlen(express)-1] = '\0'; //overwrite newline
61         fprintf(ansFp, "%s = %.4f\n", //output to file
62             express, evaluateExpression(postfixExp));
63         puts("Evaluation completed.");
64     }
65 }
66 /*Files closed*/
67 fclose(expFp);
68 fclose(ansFp);
69
70 return 0;
71 }
72
73 void infixToPostfix(const char infixExp[], char postfixExp[])
74 {
75     /*Declaration and initialization*/
76     int infixIndex=0; //array index for infixExp and loop counter
77     int pofxIndex=0; //array index for postfixExp
78     char ch; //stores popped stack items
79     int expressLength = strlen(infixExp)-1; //-1 to exclude newline
80     int operatorCount=0; //stores the count of operators
81
82     /*Counting the operators in the expression*/
83     for(infixIndex=0; infixIndex < expressLength; infixIndex++)
84         if(isOperator(infixExp[infixIndex]))
85             operatorCount++;
86
87     /*Declaring a pointer to a stack*/
88     struct stack *opStack = malloc(sizeof(struct stack)
89         + operatorCount*sizeof(float));
90     (*opStack).stackPointer = -1; //initializing the stackPointer
91
92     /*Main loop - goes through expression one character at a time*/
93     for(infixIndex=0; infixIndex < expressLength; infixIndex++)
94     {
95         /*If digit, read all further digits*/
96         if(isdigit(infixExp[infixIndex]))
97         {
98             /*Loop while the next character is a digit or decimal point*/
99             do
100             {
101                 postfixExp[pofxIndex++] = infixExp[infixIndex++];
102             }
103             while(isdigit(infixExp[infixIndex])

```

```

104         || infixExp[infixIndex]=='.'');
105
106         /*Insert a space for a new postfix element to come after*/
107         postfixExp[pofxIndex++] = ' ';
108         /*Index adjusted since an extra character was read*/
109         infixIndex--; //since an extra character was read
110     }
111     else if(infixExp[infixIndex] == '(')
112     {
113         /*Left parentheses simply pushed*/
114         push(opStack, '(');
115     }
116     else if(infixExp[infixIndex] == ')')
117     {
118         /*For right parentheses, append operators to */
119         /*the postfix expression.                */
120         while((ch=pop(opStack)) != '(')
121         {
122             /*Insert popped operator and a space*/
123             postfixExp[pofxIndex++] = ch;
124             postfixExp[pofxIndex++] = ' ';
125         }
126     }
127     else if(isOperator(infixExp[infixIndex]))
128     {
129         /*if operator was found, add all higher-or-equal- */
130         /*precedence operators to postfix expression        */
131         while((ch=pop(opStack)) != '(' && ch!='\0')
132         {
133             /*If not less precedence (i.e if higher or equal)*/
134             if(precedenceComp(ch, infixExp[infixIndex]) != -1)
135             {
136                 /*Insert popped operator and a space*/
137                 postfixExp[pofxIndex++] = ch;
138                 postfixExp[pofxIndex++] = ' ';
139             }
140             else
141             {
142                 /*Push operator with less precedence*/
143                 push(opStack, ch);
144                 /*Append no more operators to the list*/
145                 break;
146             }
147         }
148
149         /*Open parenthesis pushed back*/
150         if(ch == '(')
151             push(opStack, '(');
152
153         /*Original operator now pushed*/
154         push(opStack, infixExp[infixIndex]);
155     }
156
157     /*If character did not satisfy a condition, the character*/
158     /*is simply skipped since it is not a useful character. */
159 }
160
161 /*Pop remaining operators*/
162 while((ch=pop(opStack)) != '\0')
163 {
164     /*Insert popped operator and a space*/
165     postfixExp[pofxIndex++] = ch;

```

```

166     postfixExp[pofxIndex++] = ' ';
167 }
168
169 /*Last character set to a null character */
170 postfixExp[pofxIndex-1] = '\0'; //exclude last space
171 /*Frees the memory space taken up by opStack*/
172 free(opStack);
173 }
174
175 float evaluateExpress(char postfixExp[])
176 {
177     /*Declaration and initialization*/
178     int index=0;                //array index for postfixExp
179     float op1, op2;            //store values in arithmetic operations
180     float answer=0.0;          //stores answer in arithmetic operations
181     bool operatorFound=false;  //indicates that at least one operator found
182     int operatorCount;          //stores the count of operators
183
184     /*Counting the operators in the expression*/
185     for(index=0; index < strlen(postfixExp); index++)
186         if(isOperator(postfixExp[index]))
187             operatorCount++;
188
189     /*Declaring a pointer to a stack*/
190     struct stack *postfixStack = malloc(sizeof(struct stack)
191                                         + (1+operatorCount)*sizeof(float));
192     (*postfixStack).stackPointer = -1; //initializing the stackPointer
193
194     /*Loops until the end of the postfix expression is reached*/
195     index=0;
196     while(postfixExp[index] != '\0')
197     {
198         /*Spaces are skipped*/
199         if(postfixExp[index] == ' ')
200             index++;
201         else
202         {
203             /*If first digit of a number was found, pushNumber invoked.*/
204             /*Else if an operator is found, perform an operation. */
205             if(isdigit(postfixExp[index]))
206             {
207                 /*Index set to first character after the number*/
208                 index = pushNumber(postfixStack, postfixExp, index);
209             }
210             else if (isOperator(postfixExp[index]))
211             {
212                 /*If this is the first operator found, set to true*/
213                 if(!operatorFound)
214                     operatorFound=true;
215
216                 /*Two operands popped from stack*/
217                 op2 = pop(postfixStack);
218                 op1 = pop(postfixStack);
219
220                 /*Performing operation according to operator*/
221                 switch(postfixExp[index])
222                 {
223                     case '+':
224                         answer = op1+op2;
225                         break;
226                     case '-':
227                         answer = op1-op2;

```

```

228         break;
229     case '*':
230         answer = op1*op2;
231         break;
232     case '/':
233         answer = op1/op2;
234         break;
235     }
236
237     /*sub-answer pushed back onto the stack*/
238     push(postfixStack, answer);
239     index++;
240 }
241 else //character is neither a digit nor an operand
242 {
243     puts("Error in evaluation");
244     return 0;
245 }
246 }
247 }
248
249 /*If no operators were found, answer is the number in the stack*/
250 /*Otherwise, answer of last sub-expression is returned */
251 if(!operatorFound)
252     answer = pop(postfixStack);
253 /*Free the memory space occupied by the postfixStack */
254 /*and return the result of the arithmetic evaluation. */
255 free(postfixStack);
256 return answer;
257 }
258
259 int isOperator(const char a)
260 {
261     /* 1 is returned the 'a' is an operator. */
262     /* 0 is returned the 'a' is not an operator. */
263     return (a == '+' || a == '-' || a == '*' || a == '/');
264 }

```

Listing 37: Task3_source.c

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #define EXPLEN 100
4
5  /*stack structure definition*/
6  struct stack
7  {
8      int stackPointer;
9      float items[];
10 };
11
12 /* operation: pushes the float passed in onto the specified */
13 /* stack. The stack pointer is incremented. */
14 /* preconditions: a pointer to a predeclared stack and a float */
15 /* are passed to the function. */
16 /* postconditions: stack will contain the float 'item' and the */
17 /* stackPointer within the stack holds the new */
18 /* float item's index in the 'items' array. */
19 void push(struct stack *theStack, const float item);
20
21 /* operation: pops and returns a float from the specified */
22 /* stack. The stack pointer is decremented. */

```

```

23 /* preconditions: a pointer to a predeclared stack from which */
24 /*               a float will be obtained is passed in.      */
25 /* postconditions: float obtained is returned and stack        */
26 /*               pointer will point to next item, if any.     */
27 /*               If stack was empty when 'pop' was called,     */
28 /*               a null character '\0' is returned instead.    */
29 float pop(struct stack *theStack);
30
31 /* operation: pushes a number found in the postfixExp array   */
32 /*               starting from the location indicated by 'index'. */
33 /*               Characters from this location onwards are pushed */
34 /*               onto the specified stack until a character which */
35 /*               is not a digit, decimal point, or negative sign */
36 /*               is reached in the array.                      */
37 /* preconditions: a stack to push the number onto, the array   */
38 /*               containing the number, and the index at        */
39 /*               which the number starts are all passed in.     */
40 /* postconditions: the stack now contains the number. The index */
41 /*               of the array location containing the first     */
42 /*               character found after the number is returned.*/
43 int pushNumber(struct stack *theStack, char postfixExp[], int index);
44
45 #endif // STACK_H_INCLUDED

```

Listing 38: Stack.h

```

1 #ifndef VALANDCOMP_H_INCLUDED
2 #define VALANDCOMP_H_INCLUDED
3
4 /* operation: the infix expression 'express' is checked for */
5 /*               mismatches in parentheses, illegal characters, */
6 /*               and for the case where the user input an empty */
7 /*               expression. Also, if expression size limit was */
8 /*               reached, user is warned about incorrect results.*/
9 /* preconditions: array containing expression to be checked is */
10 /*               passed to the function.                        */
11 /* postconditions: '0' is returned if the expression broke one */
12 /*               of the guidelines specified above; '1' is     */
13 /*               returned if the expression is valid. Not      */
14 /*               Note that if the size limit was exceeded,    */
15 /*               expression is still considered to be valid.*/
16 int expressionVal(const char express[]);
17
18 /* operation: compares the precedence of two operators from */
19 /*               '+', '-', '*', and '/'                      */
20 /* preconditions: characters 'a' and 'b' each contain an     */
21 /*               an operator. These will be compared together*/
22 /* postconditions: '1' is returned if precedence of 'a' is   */
23 /*               greater than that of 'b'                    */
24 /*               '0' is returned if precedence of 'a' is     */
25 /*               equal to that of 'b'                        */
26 /*               '-1' is returned if precedence of 'a' is    */
27 /*               less than that of 'b'                       */
28 int precedenceComp(const char a, const char b);
29
30 #endif // VALANDCOMP_H_INCLUDED

```

Listing 39: ValAndComp.h


```

1 #include <ctype.h> //for isdigit()
2 #include <stdlib.h> //for atof()
3 #include "Stack.h"
4
5 void push(struct stack *theStack, const float item)
6 {
7     /*Stack pointer incremented and item stored at new location*/
8     (*theStack).stackPointer++;
9     (*theStack).items[(*theStack).stackPointer] = item;
10 }
11
12 float pop(struct stack *theStack)
13 {
14     /*If stack not empty, return item at location indicated*/
15     /*by stack pointer. Stack pointer is then decremented */
16     if((*theStack).stackPointer != -1)
17         return (*theStack).items[(*theStack).stackPointer--];
18     else
19         return '\0'; //stack is empty
20 }
21
22 int pushNumber(struct stack *theStack, char express[], int index)
23 {
24     /*Declaration of variables*/
25     int tempIndex; //stores array index
26     int numCount=0; //stores amount of number characters
27
28     /*Push all characters in 'express' starting from 'index' */
29     /*until a character which is not a digit, a decimal point,*/
30     /*or a negative sign is found.*/
31     tempIndex = index;
32     while(isdigit(express[tempIndex])
33         || express[tempIndex]=='.'
34         || express[tempIndex]=='-')
35     {
36         numCount++;
37         tempIndex++;
38     }
39
40     /*Declaring array for number*/
41     char number[numCount+1];
42
43     /*Sub-array for operand created*/
44     for(tempIndex=0; tempIndex < numCount; tempIndex++)
45         number[tempIndex] = express[index++];
46     number[tempIndex] = '\0';
47
48     /*Push number onto stack*/
49     push(theStack, atof(number));
50     /*Index of first character found after the number is returned*/
51     return index;
52 }

```

Listing 40: StackFunctions.c

```

1 #include <stdio.h> //for puts()
2 #include <string.h> //for strlen()
3 #include <ctype.h> //for isdigit()
4 #include <stdlib.h> //for malloc(), free()
5 #include "Stack.h" //for stack
6

```

```

7  /* operation: the parentheses, if any, in the expression found*/
8  /*           in 'express' are checked for mismatches.      */
9  /* preconditions: array containing expression to be checked is*/
10 /*              passed to the function.                      */
11 /* postconditions: '0' is returned in the case of a mismatch. */
12 /*              '1' returned if the expression is valid.    */
13 int parenthesesVal(const char express[]);
14
15 /* operation: the characters which make up the expression in */
16 /*           'express' are checked for illegal characters.    */
17 /*           Legal: '+', '-', '*', '/', '.', '(', and ')'     */
18 /* preconditions: array containing expression to be checked is*/
19 /*              passed to the function.                      */
20 /* postconditions: First illegal character found is returned. */
21 /*              Otherwise, a null character '\0' returned.   */
22 char charactersVal(const char express[]);
23
24 int expressionVal(const char express[])
25 {
26     /*Character used for return of function charactersVal*/
27     char ch;
28
29     /*A series of checks run*/
30     /*0 returned : expression failed at least one check*/
31     /*1 returned : expression passed all checks*/
32     if( !parenthesesVal(express) ) //parentheses check
33     {
34         puts("Invalid expression; parentheses mismatch.");
35         return 0;
36     }
37     else if( (ch=charactersVal(express)) ) //character check
38     {
39         printf("Invalid expression; illegal character '%c'.\n", ch);
40         return 0;
41     }
42     else if(express[0] == '\n') //expression presence check
43     {
44         puts("Invalid expression; nothing was input.");
45         return 0;
46     }
47
48     /*Expression still considered to be valid if expression limit*/
49     /*was reached. Program might still crash if limit reached.  */
50     if(strlen(express) >= EXPLEN-1) //expression length limit check
51         puts("Warning: answer might be incorrect; expression length limit
52 reached.");
53
54     /*Expression passed all checks*/
55     return 1;
56 }
57
58 int parenthesesVal(const char express[])
59 {
60     /*Declaration of variables*/
61     int expressLength = strlen(express)-1; //-1 to exclude newline
62     int index=0;        //array index for expression
63     int openParCount=0; //stores the count of open parentheses
64
65     /*Count parentheses for use in malloc*/
66     for(index=0; index < expressLength; index++)
67         if(express[index] == '(')
68             openParCount++;

```

```

68
69  /*Declaring a pointer to a stack*/
70  struct stack *parenStack = malloc(sizeof(struct stack)
71                                  + openParCount*sizeof(float));
72  (*parenStack).stackPointer = -1; //initializing the stackPointer
73
74  /*Pushing and popping parentheses*/
75  for(index=0; index < expressLength; index++)
76  {
77      /*Push any open parentheses and pop an open      */
78      /*parenthesis when a close parenthesis is found */
79      if(express[index] == '(')
80          push(parenStack, '(');
81      else if(express[index] == ')')
82      {
83          /*If stack is empty, function returns.      */
84          /*Otherwise, an open parentheses is popped. */
85          if(pop(parenStack) == '\0')
86          {
87              free(parenStack);
88              return 0;
89          }
90      }
91  }
92
93  /*Checking if stack still has parentheses*/
94  if((*parenStack).stackPointer != -1)
95  {
96      free(parenStack);
97      return 0;
98  }
99
100  /*Expression passed parentheses check*/
101  free(parenStack);
102  return 1;
103 }
104
105 char charactersVal(const char express[])
106 {
107     /*Declaration of variables*/
108     int expressLength = strlen(express)-1; // -1 to exclude newline
109     int index=0;                          //array index for expression
110
111     /*Loops through all characters of the expression*/
112     for(index=0; index < expressLength; index++)
113     {
114         /*If a character is not any of these characters, */
115         /*then it is considered to be forbidden.          */
116         if( !isdigit(express[index])    && express[index] != '+'
117            && express[index] != '-' && express[index] != '*'
118            && express[index] != '/' && express[index] != '.'
119            && express[index] != '(' && express[index] != ')')
120         {
121             /*Illegal character returned*/
122             return express[index];
123         }
124     }
125     /*No illegal characters found - expression passed character check*/
126     return '\0';
127 }
128
129 int precedenceComp(const char a, const char b)

```

```

130 {
131     /*If 'a' is * or /... */
132     /*...if 'b' is * or /, operators have equal precedence */
133     /*...if 'b' is + or -, then 'a' has higher precedence */
134     /*If 'a' is + or -... */
135     /*...if 'b' is * or /, then 'a' has lower precedence */
136     /*...if 'b' is + or -, operators have equal precedence */
137     return (a=='*' || a=='/') - (b=='*' || b=='/');
138 }

```

Listing 41: Validation.c

6o Task4a

```

1 int main(void)
2 {
3     return 0;
4 }

```

Listing 42: Task4_source.c

```

1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3
4 /*stack structure definition*/
5 struct stack
6 {
7     int stackPointer;
8     float items[];
9 };
10
11 /*An alias for the stack struct*/
12 typedef struct stack Stack; // "Stack" is an alias for the struct
13
14 /* operation: declares a pointer to a stack and allocates */
15 /* enough memory so that the array 'items' in the */
16 /* stack will have 'max_size' usable elements. */
17 /* preconditions: maximum amount of items that the stack will */
18 /* be able to fit is passed to the function. */
19 /* postconditions: the pointer to the stack is returned. */
20 Stack *newStack(int max_size);
21
22 /* operation: pushes the float passed in onto the specified */
23 /* stack. The stack pointer is incremented. */
24 /* preconditions: a pointer to a predeclared stack and a float */
25 /* are passed to the function. */
26 /* postconditions: stack will contain the float 'item' and the */
27 /* stackPointer within the stack holds the new */
28 /* float item's index in the 'items' array. */
29 void push(Stack *theStack, const float item);
30
31 /* operation: pops and returns a float from the specified */
32 /* stack. The stack pointer is decremented. */
33 /* preconditions: a pointer to a predeclared stack from which */
34 /* a float will be obtained is passed in. */
35 /* postconditions: float obtained is returned and stack */
36 /* pointer will point to next item, if any. */
37 /* If stack was empty when 'pop' is called, */
38 /* a null character '\0' is returned instead. */

```

```

39 float pop(Stack *theStack);
40
41 /* operation: similar to the operation of the pop function */
42 /* but does not decrement the stack pointer. */
43 /* preconditions: a pointer to a predeclared stack whose top */
44 /* item will be returned is passed in. */
45 /* postconditions: float at the top of the stack (the last */
46 /* item pushed) is returned. If stack is */
47 /* empty, a null character '\0' is returned. */
48 float top(Stack *theStack);
49
50 /* operation: checks the specified stack's stack pointer to */
51 /* see if the stack is empty and returns a value */
52 /* indicating the state of the stack. */
53 /* preconditions: a pointer to a predeclared stack that needs */
54 /* to be checked is passed in. */
55 /* postconditions: 1 is returned if the stack is empty. */
56 /* 0 is returned if the stack is not empty. */
57 bool is_empty(Stack *theStack);
58
59 #endif // STACK_H_INCLUDED

```

Listing 43: Stack.h

```

1 #include <stdlib.h> //for malloc()
2 #include <stdbool.h> //for bool, true, false
3 #include "Stack.h"
4
5 Stack *newStack(const int max_size)
6 {
7     /*Pointer to a stack declared, memory allocated,*/
8     /*and the stack pointer is initialized to -1. */
9     Stack *theStack = malloc(sizeof(Stack) + max_size*sizeof(float));
10    (*theStack).stackPointer = -1;
11
12    /*Pointer to the stack is returned*/
13    return theStack;
14 }
15
16 void push(struct stack *theStack, const float item)
17 {
18     /*Stack pointer incremented and item stored at new location*/
19     (*theStack).stackPointer++;
20     (*theStack).items[(*theStack).stackPointer] = item;
21 }
22
23 float pop(struct stack *theStack)
24 {
25     /*If stack not empty, return item at location indicated*/
26     /*by stack pointer. Stack pointer is then decremented */
27     if((*theStack).stackPointer != -1)
28         return (*theStack).items[(*theStack).stackPointer--];
29     else
30         return '\0'; //stack is empty
31 }
32
33 float top(Stack *theStack)
34 {
35     /*If stack not empty, return item at location indicated*/
36     /*by stack pointer. Stack pointer is not decremented. */
37     if((*theStack).stackPointer != -1)
38         return (*theStack).items[(*theStack).stackPointer];

```

```

39     else
40         return '\0'; //stack is empty
41 }
42
43 bool is_empty(Stack *theStack)
44 {
45     /*If stack pointer is -1, then the stack is empty, so*/
46     /*'true' is returned. Otherwise, 'false' is returned.*/
47     return (*theStack).stackPointer == -1 ? true : false;
48 }

```

Listing 44: StackFunctions.c

6p Task4b

```

1 int main(void)
2 {
3     return 0;
4 }

```

Listing 45: Task4_source.c

```

1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3 #include <stdbool.h>
4
5 /*node structure definition with typedef*/
6 typedef struct node
7 {
8     float item;           //this node's item
9     struct node *nextNode; //pointer to next node
10 } Node;
11
12 /*stack structure definition with typedef*/
13 typedef struct stack
14 {
15     Node *listHead; //pointer to list start
16     Node *listTail; //pointer to list end
17 } Stack;
18
19 /* operation: declares a pointer to a stack and allocates      */
20 /* memory for the head and tail pointers to nodes. */
21 /* preconditions: the function takes no arguments. */
22 /* postconditions: the pointer to the stack is returned. */
23 Stack *newStack(void);
24
25 /* operation: pushes the float passed in onto the specified */
26 /* stack. The list will have a new tail node. */
27 /* preconditions: a pointer to a predeclared stack and a float*/
28 /* are passed to the function. */
29 /* postconditions: stack will contain the float 'item' and the*/
30 /* stackPointer within the stack holds the new*/
31 /* float item's index in the 'items' array. */
32 void push(Stack *theStack, const float item);
33
34 /* operation: pops and returns a float from the specified */
35 /* stack. List's tail node is removed from the list*/
36 /* preconditions: a pointer to a predeclared stack from which */
37 /* a float will be obtained is passed in. */

```

```

38 /* postconditions: float obtained is returned and stack */
39 /*                pointer will point to next item, if any. */
40 /*                If stack was empty when 'pop' is called, */
41 /*                a null character '\0' is returned instead. */
42 float pop(Stack *theStack);
43
44 /* operation: similar to the operation of the pop function */
45 /*                but does not remove items from the stack. */
46 /* preconditions: a pointer to a predeclared stack whose top */
47 /*                item will be returned is passed in. */
48 /* postconditions: float at the tail of the list (the last */
49 /*                item pushed) is returned. If stack is */
50 /*                empty, a null character '\0' is returned. */
51 float top(Stack *theStack);
52
53 /* operation: checks the specified stack's head node to see */
54 /*                see if the stack is empty and returns a value */
55 /*                indicating the state of the stack. */
56 /* preconditions: a pointer to a predeclared stack that needs */
57 /*                to be checked is passed in. */
58 /* postconditions: 1 is returned if the stack is empty. */
59 /*                0 is returned if the stack is not empty. */
60 bool is_empty(Stack *theStack);
61
62 #endif // STACK_H_INCLUDED

```

Listing 46: Stack.h

```

1 #include <stdlib.h> //for malloc(), free(), and NULL
2 #include <stdbool.h> //for bool, true, false
3 #include "Stack.h"
4
5 Stack *newStack(void)
6 {
7     /*Pointer to a stack declared and memory allocated*/
8     Stack *theStack = malloc(sizeof(Stack));
9
10    /*List head and tail initialized to NULL -- since no items yet*/
11    (*theStack).listHead = NULL;
12    (*theStack).listTail = NULL;
13
14    /*Pointer to the stack is returned*/
15    return theStack;
16 }
17
18 void push(Stack *theStack, const float item)
19 {
20     /*Pointer to a new node declared*/
21     Node *newNode = malloc(sizeof(Node));
22
23     /*If the list is empty*/
24     if(is_empty(theStack))
25     {
26         /*New node set as head since the only item*/
27         (*theStack).listHead = newNode;
28     }
29     else //if the list is not empty
30     {
31         /*Current tail node's next node set to the new node.*/
32         ((*theStack).listTail).nextNode = newNode;
33     }
34     /*New node set as the list tail*/

```

```

35     (*theStack).listTail = newNode;
36
37     /*New node given an item and its next node set to NULL*/
38     (*newNode).item = item;
39     (*newNode).nextNode = NULL;
40 }
41
42 float pop(Stack *theStack)
43 {
44     /*Declaration of variable*/
45     float item; //stores popped item
46
47     /*If list is empty, return null character*/
48     if(is_empty(theStack))
49         return '\0';
50     else
51     {
52         /*Obtain last node's item*/
53         item = ((*theStack).listTail).item;
54
55         /*If head is tail, stack only has one item*/
56         if((*theStack).listHead == (*theStack).listTail)
57         {
58             /*Free last node and make nodes NULL since no more nodes*/
59             free((*theStack).listTail); //head implicitly freed
60             (*theStack).listHead = (*theStack).listTail = NULL;
61         }
62         else //stack does not only have one item
63         {
64             /*Declaration of pointers to nodes*/
65             Node *tailNode = (*theStack).listTail; //makes code simpler
66             Node *tempNode = (*theStack).listHead;
67
68             /*Starting from the list head, and setting tempNode */
69             /*to next node until the next node is the tail node. */
70             while((*tempNode).nextNode != tailNode)
71                 tempNode = (*tempNode).nextNode;
72
73             /*Tail node set to node before it*/
74             (*theStack).listTail = tailNode = tempNode;
75
76             /*free last node and make it NULL*/
77             free((*tailNode).nextNode);
78             (*tailNode).nextNode = NULL;
79         }
80     }
81     /*Return item if stack was not empty*/
82     return item;
83 }
84
85 float top(Stack *theStack)
86 {
87     /*If list is empty, return null character*/
88     if(is_empty(theStack))
89         return '\0';
90     else //otherwise, return last node's item
91         return ((*theStack).listTail).item;
92 }
93
94 bool is_empty(Stack *theStack)
95 {
96     /*If list head is NULL, list is empty. */

```



```

97  /*Otherwise, the list is not empty.  */
98  return ((*theStack).listHead == NULL);
99  }

```

Listing 47: StackFunctions.c

6q Task4c

```

1  #include <stdio.h>           //for puts(), fgets(), fprintf(), fopen(), fclose
    ()
2  #include <stdlib.h>         //for exit(), malloc(), and free()
3  #include <ctype.h>          //for isdigit()
4  #include <string.h>         //for strlen(), strcat()
5  #include <stdbool.h>        //for bool, true, false
6  #include "ValAndComp.h"     //for expression validation
7  #include "Stack.h"          //for stack and stack functions
8
9  /* operation: Converts the infix expression passed in to a      */
10 /*      postfix expression.                                     */
11 /* preconditions: The infix expression is stored in 'infixExp' */
12 /*      and an additional array 'postfixExp' is                */
13 /*      passed to store the resultant postfix exp-            */
14 /*      resion without having to return anything.              */
15 /* postconditions: The 'postfixExp' array will now hold the     */
16 /*      postfix equivalent of the passed expression*/
17 void infixToPostfix(const char infixExp[], char postfixExp[]);
18
19 /* operation: Evaluates the postfix expression passed in and    */
20 /*      returns the final answer.                                */
21 /* preconditions: The array passed in contains an expression    */
22 /*      in postfix notation.                                     */
23 /* postconditions: The answer of type float is returned.        */
24 float evaluateExpress(char postfixExp[]);
25
26 /* operation: Checks whether the character passed to the        */
27 /*      function is an operator (+, -, /, or *).                */
28 /* preconditions: the character to be checked is passed to      */
29 /*      the function.                                           */
30 /* postconditions: 1 is returned the 'a' is an operator.        */
31 /*      0 is returned the 'a' is not an operator.              */
32 int isOperator(const char a);
33
34 int main(void)
35 {
36     /*Declaration of file pointers and opening files*/
37     FILE *expFp = fopen("expressions.txt", "r");
38     FILE *ansFp = fopen("answers.txt", "w");
39
40     /*Declaration of array to hold input infix expression*/
41     char express[EXPLEN];
42
43     while(fgets(express, EXPLEN, expFp))
44     {
45         /*Newline character added to last expression*/
46         if(express[strlen(express)-1] != '\n')
47             strcat(express, "\n");
48
49         /*Output indicates expression read*/
50         printf("\nExpression read: %s", express);
51

```

```

52     /*If expression is validated, condition is satisfied*/
53     if(expressionVal(express))
54     {
55         /*Infix to postfix conversion*/
56         char postfixExp[2*strlen(express)]; //array declared
57         infixToPostfix(express, postfixExp); //expression converted
58
59         /*Evaluation of expressions and output to file*/
60         express[strlen(express)-1] = '\0'; //overwrite newline
61         fprintf(ansFp, "%s = %.4f\n", //output to file
62             express, evaluateExpress(postfixExp));
63         puts("Evaluation completed.");
64     }
65 }
66 /*Files closed*/
67 fclose(expFp);
68 fclose(ansFp);
69
70 return 0;
71 }
72
73 void infixToPostfix(const char infixExp[], char postfixExp[])
74 {
75     /*Declaration and initialization*/
76     int infxIndex=0; //array index for infixExp and loop counter
77     int pofxIndex=0; //array index for postfixExp
78     char ch; //stores popped stack items
79     int expressLength = strlen(infixExp)-1; //-1 to exclude newline
80
81     /*Declaring a pointer to a stack*/
82     Stack *opStack = newStack();
83
84     /*Main loop - goes through expression one character at a time*/
85     for(infxIndex=0; infxIndex < expressLength; infxIndex++)
86     {
87         /*If digit, read all further digits*/
88         if(isdigit(infixExp[infxIndex]))
89         {
90             /*Loop while the next character is a digit or decimal point*/
91             do
92             {
93                 postfixExp[pofxIndex++] = infixExp[infxIndex++];
94             }
95             while(isdigit(infixExp[infxIndex])
96                 || infixExp[infxIndex]=='.' );
97
98             /*Insert a space for a new postfix element to come after*/
99             postfixExp[pofxIndex++] = ' ';
100             /*Index adjusted since an extra character was read*/
101             infxIndex--; //since an extra character was read
102         }
103         else if(infixExp[infxIndex] == '(')
104         {
105             /*Left parentheses simply pushed*/
106             push(opStack, '(');
107         }
108         else if(infixExp[infxIndex] == ')')
109         {
110             /*For right parentheses, append operators to */
111             /*the postfix expression. */
112             while((ch=pop(opStack)) != '(')
113             {

```

```

114         /*Insert popped operator and a space*/
115         postfixExp[pofxIndex++] = ch;
116         postfixExp[pofxIndex++] = ' ';
117     }
118 }
119 else if(isOperator(infixExp[infxIndex]))
120 {
121     /*if operator was found, add all higher-or-equal- */
122     /*precedence operators to postfix expression */
123     while((ch=top(opStack)) != '(' && ch!='\0')
124     {
125         /*Pop the character that was checked*/
126         pop(opStack);
127         /*If not less precedence (i.e if higher or equal)*/
128         if(precedenceComp(ch, infixExp[infxIndex]) != -1)
129         {
130             /*Insert popped operator and a space*/
131             postfixExp[pofxIndex++] = ch;
132             postfixExp[pofxIndex++] = ' ';
133         }
134         else
135         {
136             /*Push operator with less precedence*/
137             push(opStack, ch);
138             /*Append no more operators to the list*/
139             break;
140         }
141     }
142     /*Original operator now pushed*/
143     push(opStack, infixExp[infxIndex]);
144 }
145 /*If character did not satisfy a condition, the character*/
146 /*is simply skipped since it is not a useful character. */
147 }
148
149 /*Pop remaining operators*/
150 while((ch=pop(opStack)) != '\0')
151 {
152     /*Insert popped operator and a space*/
153     postfixExp[pofxIndex++] = ch;
154     postfixExp[pofxIndex++] = ' ';
155 }
156
157 /*Last character set to a null character */
158 postfixExp[pofxIndex-1] = '\0'; //exclude last space
159 /*Frees the memory space taken up by opStack*/
160 free(opStack);
161 }
162
163 float evaluateExpress(char postfixExp[])
164 {
165     /*Declaration and initialization*/
166     int index=0;           //array index for postfixExp
167     float op1, op2;        //store values in arithmetic operations
168     float answer=0.0;      //stores answer in arithmetic operations
169     bool operatorFound=false; //indicates that at least one operator found
170
171     /*Declaring a pointer to a stack*/
172     Stack *postfixStack = newStack();
173
174     /*Loops until the end of the postfix expression is reached*/
175     while(postfixExp[index] != '\0')

```

```

176 {
177     /*Spaces are skipped*/
178     if(postfixExp[index] == ' ')
179         index++;
180     else
181     {
182         /*If first digit of a number was found, pushNumber invoked.*/
183         /*Else if an operator is found, perform an operation. */
184         if(isdigit(postfixExp[index]))
185         {
186             /*Index set to first character after the number*/
187             index = pushNumber(postfixStack, postfixExp, index);
188         }
189         else if (isOperator(postfixExp[index]))
190         {
191             /*If this is the first operator found, set to true*/
192             if(!operatorFound)
193                 operatorFound=true;
194
195             /*Two operands popped from stack*/
196             op2 = pop(postfixStack);
197             op1 = pop(postfixStack);
198
199             /*Performing operation according to operator*/
200             switch(postfixExp[index])
201             {
202                 case '+':
203                     answer = op1+op2;
204                     break;
205                 case '-':
206                     answer = op1-op2;
207                     break;
208                 case '*':
209                     answer = op1*op2;
210                     break;
211                 case '/':
212                     answer = op1/op2;
213                     break;
214             }
215
216             /*sub-answer pushed back onto the stack*/
217             push(postfixStack, answer);
218             index++;
219         }
220         else //character is neither a digit nor an operand
221         {
222             puts("Error in evaluation");
223             return 0;
224         }
225     }
226 }
227
228 /*If no operators were found, answer is the number in the stack*/
229 /*Otherwise, answer of last sub-expression is returned */
230 if(!operatorFound)
231     answer = pop(postfixStack);
232 /*Free the memory space occupied by the postfixStack */
233 /*and return the result of the arithmetic evaluation. */
234 free(postfixStack);
235 return answer;
236 }
237

```

```

238 int isOperator(const char a)
239 {
240     /* 1 is returned the 'a' is an operator.      */
241     /* 0 is returned the 'a' is not an operator.  */
242     return (a == '+' || a == '-' || a == '*' || a == '/');
243 }

```

Listing 48: Task4_source.c

```

1  #ifndef STACK_H_INCLUDED
2  #define STACK_H_INCLUDED
3  #include <stdbool.h>
4  #define EXPLEN 100
5
6  /*node structure definition with typedef*/
7  typedef struct node
8  {
9      float item;           //this node's item
10     struct node *nextNode; //pointer to next node
11 } Node;
12
13 /*stack structure definition with typedef*/
14 typedef struct stack
15 {
16     Node *listHead; //pointer to list start
17     Node *listTail; //pointer to list end
18 } Stack;
19
20 /* operation: declares a pointer to a stack and allocates      */
21 /*               memory for the head and tail pointers to nodes. */
22 /* preconditions: the function takes no arguments.             */
23 /* postconditions: the pointer to the stack is returned.       */
24 Stack *newStack(void);
25
26 /* operation: pushes the float passed in onto the specified    */
27 /*               stack. The list will have a new tail node.    */
28 /* preconditions: a pointer to a predeclared stack and a float*/
29 /*               are passed to the function.                   */
30 /* postconditions: stack will contain the float 'item' and the */
31 /*               stackPointer within the stack holds the new*/
32 /*               float item's index in the 'items' array.      */
33 void push(Stack *theStack, const float item);
34
35 /* operation: pops and returns a float from the specified      */
36 /*               stack. List's tail node is removed from the list*/
37 /* preconditions: a pointer to a predeclared stack from which */
38 /*               a float will be obtained is passed in.       */
39 /* postconditions: float obtained is returned and stack        */
40 /*               pointer will point to next item, if any.      */
41 /*               If stack was empty when 'pop' is called,      */
42 /*               a null character '\0' is returned instead.    */
43 float pop(Stack *theStack);
44
45 /* operation: similar to the operation of the pop function     */
46 /*               but does not remove items from the stack.     */
47 /* preconditions: a pointer to a predeclared stack whose top   */
48 /*               item will be returned is passed in.          */
49 /* postconditions: float at the tail of the list (the last     */
50 /*               item pushed) is returned. If stack is         */
51 /*               empty, a null character '\0' is returned.     */
52 float top(Stack *theStack);
53

```

```

54 /* operation: checks the specified stack's head node to see */
55 /*      see if the stack is empty and returns a value */
56 /*      indicating the state of the stack. */
57 /* preconditions: a pointer to a predeclared stack that needs */
58 /*      to be checked is passed in. */
59 /* postconditions: 1 is returned if the stack is empty. */
60 /*      0 is returned if the stack is not empty. */
61 bool is_empty(Stack *theStack);
62
63 /* operation: pushes a number found in the postfixExp array */
64 /*      starting from the location indicated by 'index'. */
65 /*      Characters from this location onwards are pushed */
66 /*      onto the specified stack until a character which */
67 /*      is not a digit, decimal point, or negative sign */
68 /*      is reached in the array. */
69 /* preconditions: a stack to push the number onto, the array */
70 /*      containing the number, and the index at */
71 /*      which the number starts are all passed in. */
72 /* postconditions: the stack now contains the number. The index */
73 /*      of the array location containing the first */
74 /*      character found after the number is returned.*/
75 int pushNumber(struct stack *theStack, char postfixExp[], int index);
76
77 #endif // STACK_H_INCLUDED

```

Listing 49: Stack.h

```

1 #ifndef VALANDCOMP_H_INCLUDED
2 #define VALANDCOMP_H_INCLUDED
3
4 /* operation: the infix expression 'express' is checked for */
5 /*      mismatches in parentheses, illegal characters, */
6 /*      and for the case where the user input an empty */
7 /*      expression. Also, if expression size limit was */
8 /*      reached, user is warned about incorrect results.*/
9 /* preconditions: array containing expression to be checked is*/
10 /*      passed to the function. */
11 /* postconditions: '0' is returned if the expression broke one*/
12 /*      of the guidelines specified above; '1' is */
13 /*      returned if the expression is valid. Not */
14 /*      Note that if the size limit was exceeded, */
15 /*      expression is still considered to be valid.*/
16 int expressionVal(const char express[]);
17
18 /* operation: compares the precedence of two operators from */
19 /*      '+', '-', '*', and '/' */
20 /* preconditions: characters 'a' and 'b' each contain an */
21 /*      an operator. These will be compared together*/
22 /* postconditions: '1' is returned if precedence of 'a' is */
23 /*      greater than that of 'b' */
24 /*      '0' is returned if precedence of 'a' is */
25 /*      equal to that of 'b' */
26 /*      '-1' is returned if precedence of 'a' is */
27 /*      less than that of 'b' */
28 int precedenceComp(const char a, const char b);
29
30 #endif // VALANDCOMP_H_INCLUDED

```

Listing 50: ValAndComp.h

```

1 #include <stdlib.h> //for malloc(), free(), and NULL

```

```

2 #include <stdbool.h> //for bool, true, false
3 #include <ctype.h>    //for isdigit()
4 #include "Stack.h"
5
6 Stack *newStack(void)
7 {
8     /*Pointer to a stack declared and memory allocated*/
9     Stack *theStack = malloc(sizeof(Stack));
10
11     /*List head and tail initialized to NULL -- since no items yet*/
12     (*theStack).listHead = NULL;
13     (*theStack).listTail = NULL;
14
15     /*Pointer to the stack is returned*/
16     return theStack;
17 }
18
19 void push(Stack *theStack, const float item)
20 {
21     /*Pointer to a new node declared*/
22     Node *newNode = malloc(sizeof(Node));
23
24     /*If the list is empty*/
25     if(is_empty(theStack))
26     {
27         /*New node set as head since the only item*/
28         (*theStack).listHead = newNode;
29     }
30     else //if the list is not empty
31     {
32         /*Current tail node's next node set to the new node.*/
33         ((*theStack).listTail).nextNode = newNode;
34     }
35     /*New node set as the list tail*/
36     (*theStack).listTail = newNode;
37
38     /*New node given an item and its next node set to NULL*/
39     (*newNode).item = item;
40     (*newNode).nextNode = NULL;
41 }
42
43 float pop(Stack *theStack)
44 {
45     /*Declaration of variable*/
46     float item; //stores popped item
47
48     /*If list is empty, return null character*/
49     if(is_empty(theStack))
50         return '\0';
51     else
52     {
53         /*Obtain last node's item*/
54         item = ((*theStack).listTail).item;
55
56         /*If head is tail, stack only has one item*/
57         if((*theStack).listHead == (*theStack).listTail)
58         {
59             /*Free last node and make nodes NULL since no more nodes*/
60             free((*theStack).listTail); //head implicitly freed
61             (*theStack).listHead = (*theStack).listTail = NULL;
62         }
63         else //stack does not only have one item

```

```

64     {
65         /*Declaration of pointers to nodes*/
66         Node *tailNode = (*theStack).listTail; //makes code simpler
67         Node *tempNode = (*theStack).listHead;
68
69         /*Starting from the list head, and setting tempNode */
70         /*to next node until the next node is the tail node. */
71         while((*tempNode).nextNode != tailNode)
72             tempNode = (*tempNode).nextNode;
73
74         /*Tail node set to node before it*/
75         (*theStack).listTail = tailNode = tempNode;
76
77         /*free last node and make it NULL*/
78         free((*tailNode).nextNode);
79         (*tailNode).nextNode = NULL;
80     }
81 }
82 /*Return item if stack was not empty*/
83 return item;
84 }
85
86 float top(Stack *theStack)
87 {
88     /*If list is empty, return null character*/
89     if(is_empty(theStack))
90         return '\0';
91     else //otherwise, return last node's item
92         return ((*theStack).listTail).item;
93 }
94
95 bool is_empty(Stack *theStack)
96 {
97     /*If list head is NULL, list is empty. */
98     /*Otherwise, the list is not empty. */
99     return ((*theStack).listHead == NULL);
100 }
101
102 int pushNumber(struct stack *theStack, char express[], int index)
103 {
104     /*Declaration of variables*/
105     int tempIndex; //stores array index
106     int numCount=0; //stores amount of number characters
107
108     /*Push all characters in 'express' starting from 'index' */
109     /*until a character which is not a digit, a decimal point,*/
110     /*or a negative sign is found.*/
111     tempIndex = index;
112     while(isdigit(express[tempIndex])
113           || express[tempIndex]=='.'
114           || express[tempIndex]=='-')
115     {
116         numCount++;
117         tempIndex++;
118     }
119
120     /*Declaring array for number*/
121     char number[numCount+1];
122
123     /*Sub-array for operand created*/
124     for(tempIndex=0; tempIndex < numCount; tempIndex++)
125         number[tempIndex] = express[index++];

```



```

126     number[tempIndex] = '\0';
127
128     /*Push number onto stack*/
129     push(theStack, atof(number));
130     /*Index of first character found after the number is returned*/
131     return index;
132 }

```

Listing 51: StackFunctions.c

```

1  #include <stdio.h> //for puts()
2  #include <string.h> //for strlen()
3  #include <ctype.h> //for isdigit()
4  #include <stdlib.h> //for malloc(), free()
5  #include "Stack.h" //for stack
6
7  /* operation: the parentheses, if any, in the expression found*/
8  /*           in 'express' are checked for mismatches. */
9  /* preconditions: array containing expression to be checked is*/
10 /*                passed to the function. */
11 /* postconditions: '0' is returned in the case of a mismatch. */
12 /*                '1' returned if the expression is valid. */
13 int parenthesesVal(const char express[]);
14
15 /* operation: the characters which make up the expression in */
16 /*           'express' are checked for illegal characters. */
17 /*           Legal: '+', '-', '*', '/', '.', '(', and ')' */
18 /* preconditions: array containing expression to be checked is*/
19 /*                passed to the function. */
20 /* postconditions: First illegal character found is returned. */
21 /*                Otherwise, a null character '\0' returned. */
22 char charactersVal(const char express[]);
23
24 int expressionVal(const char express[])
25 {
26     /*Character used for return of function charactersVal*/
27     char ch;
28
29     /*A series of checks run*/
30     /*0 returned : expression failed at least one check*/
31     /*1 returned : expression passed all checks*/
32     if( !parenthesesVal(express) ) //parentheses check
33     {
34         puts("Invalid expression; parentheses mismatch.");
35         return 0;
36     }
37     else if( (ch=charactersVal(express)) ) //character check
38     {
39         printf("Invalid expression; illegal character '%c'.\n", ch);
40         return 0;
41     }
42     else if(express[0] == '\n') //expression presence check
43     {
44         puts("Invalid expression; nothing was input.");
45         return 0;
46     }
47
48     /*Expression still considered to be valid if expression limit*/
49     /*was reached. Program might still crash if limit reached. */
50     if(strlen(express) >= EXPLEN-1) //expression length limit check
51         puts("Warning: answer might be incorrect; expression length limit
reached.");

```

```

52
53     /*Expression passed all checks*/
54     return 1;
55 }
56
57 int parenthesesVal(const char express[])
58 {
59     /*Declaration of variables*/
60     int expressLength = strlen(express)-1; //-1 to exclude newline
61     int index=0;          //array index for expression
62
63     /*Declaring a pointer to a stack*/
64     Stack *parenStack = newStack();
65
66     /*Pushing and popping parentheses*/
67     for(index=0; index < expressLength; index++)
68     {
69         /*Push any open parentheses and pop an open      */
70         /*parenthesis when a close parenthesis is found */
71         if(express[index] == '(')
72             push(parenStack, '(');
73         else if(express[index] == ')')
74         {
75             /*If stack is empty, function returns.      */
76             /*Otherwise, an open parentheses is popped*/
77             if(pop(parenStack) == '\0')
78             {
79                 free(parenStack);
80                 return 0;
81             }
82         }
83     }
84
85     /*Checking if stack still has parentheses*/
86     if(!is_empty(parenStack))
87     {
88         free(parenStack);
89         return 0;
90     }
91
92     /*Expression passed parentheses check*/
93     free(parenStack);
94     return 1;
95 }
96
97 char charactersVal(const char express[])
98 {
99     /*Declaration of variables*/
100     int expressLength = strlen(express)-1; //-1 to exclude newline
101     int index=0;          //array index for expression
102
103     /*Loops through all characters of the expression*/
104     for(index=0; index < expressLength; index++)
105     {
106         /*If a character is not any of these characters, */
107         /*then it is considered to be forbidden.          */
108         if( !isdigit(express[index])    && express[index] != '+'
109            && express[index] != '-' && express[index] != '*'
110            && express[index] != '/' && express[index] != '.'
111            && express[index] != '(' && express[index] != ')')
112         {
113             /*Illegal character returned*/

```

```

114         return express[index];
115     }
116 }
117 /*No illegal characters found - expression passed character check*/
118 return '\0';
119 }
120
121 int precedenceComp(const char a, const char b)
122 {
123     /*If 'a' is * or /... */
124     /*...if 'b' is * or /, operators have equal precedence */
125     /*...if 'b' is + or -, then 'a' has higher precedence */
126     /*If 'a' is + or -... */
127     /*...if 'b' is * or /, then 'a' has lower precedence */
128     /*...if 'b' is + or -, operators have equal precedence */
129     return (a=='*' || a=='/') - (b=='*' || b=='/');
130 }

```

Listing 52: Validation.c

6r Task5

```

1 #include <stdio.h>
2 #include "Stack.h"
3 int main(void)
4 {
5     /*Declaring a pointer to a stack*/
6     Stack *testStack = newStack();
7
8     /*Pushing the character 'C' onto the stack*/
9     puts("Pushing into stack...");
10    push(testStack, 'C');
11
12    /*Popping the character 'C' from the stack*/
13    puts("Popping from stack...");
14    printf("Character popped: \'%c\'", (char) pop(testStack));
15
16    return 0;
17 }

```

Listing 53: Task5_source.c

```

1 #ifndef STACK_H_INCLUDED
2 #define STACK_H_INCLUDED
3 #include <stdbool.h>
4
5 /*node structure definition with typedef*/
6 typedef struct node
7 {
8     float item;           //this node's item
9     struct node *nextNode; //pointer to next node
10 } Node;
11
12 /*stack structure definition with typedef*/
13 typedef struct stack
14 {
15     Node *listHead; //pointer to list start
16     Node *listTail; //pointer to list end
17 } Stack;

```

```
18
19 /* operation: declares a pointer to a stack and allocates */
20 /* memory for the head and tail pointers to nodes. */
21 /* preconditions: the function takes no arguments. */
22 /* postconditions: the pointer to the stack is returned. */
23 Stack *newStack(void);
24
25 /* operation: pushes the float passed in onto the specified */
26 /* stack. The list will have a new tail node. */
27 /* preconditions: a pointer to a predeclared stack and a float*/
28 /* are passed to the function. */
29 /* postconditions: stack will contain the float 'item' and the*/
30 /* stackPointer within the stack holds the new*/
31 /* float item's index in the 'items' array. */
32 void push(Stack *theStack, const float item);
33
34 /* operation: pops and returns a float from the specified */
35 /* stack. List's tail node is removed from the list*/
36 /* preconditions: a pointer to a predeclared stack from which */
37 /* a float will be obtained is passed in. */
38 /* postconditions: float obtained is returned and stack */
39 /* pointer will point to next item, if any. */
40 /* If stack was empty when 'pop' is called, */
41 /* a null character '\0' is returned instead. */
42 float pop(Stack *theStack);
43
44 /* operation: similar to the operation of the pop function */
45 /* but does not remove items from the stack. */
46 /* preconditions: a pointer to a predeclared stack whose top */
47 /* item will be returned is passed in. */
48 /* postconditions: float at the tail of the list (the last */
49 /* item pushed) is returned. If stack is */
50 /* empty, a null character '\0' is returned. */
51 float top(Stack *theStack);
52
53 /* operation: checks the specified stack's head node to see */
54 /* see if the stack is empty and returns a value */
55 /* indicating the state of the stack. */
56 /* preconditions: a pointer to a predeclared stack that needs */
57 /* to be checked is passed in. */
58 /* postconditions: 1 is returned if the stack is empty. */
59 /* 0 is returned if the stack is not empty. */
60 bool is_empty(Stack *theStack);
61
62 #endif // STACK_H_INCLUDED
```

Listing 54: Stack.h

7 Reference List

- [1] D. Cook. Decimal Number Conversion. Available:
<http://www.robotroom.com/NumberSystems3.html>.
- [2] srand. Available:
<http://www.cplusplus.com/reference/cstdlib/srand/>.
- [3] R. Bradley, "Insertion sort," in New Understanding Computer Science for Advanced Level, 4th ed. United Kingdom: Stanley Thornes, 1999, pp. 402-403.
- [4] R. Bradley, "Quick sort (partition sort)," in New Understanding Computer Science for Advanced Level, 4th ed. United Kingdom: Stanley Thornes, 1999, pp. 408-410.
- [5] Quicksort. Available:
<http://www.algolist.net/Algorithms/Sorting/Quicksort>.
- [6] clock_t. Available:
http://www.cplusplus.com/reference/ctime/clock_t/.
- [7] clock. Available: <http://www.cplusplus.com/reference/ctime/clock/>.
- [8] CLOCKS_PER_SEC. Available:
http://www.cplusplus.com/reference/ctime/CLOCKS_PER_SEC/.
- [9] Dynamic Programming. Available:
<http://interactivepython.org/runestone/static/pythonds/Recursion/DynamicProgramming.html>.
- [10] S. Prata, C Primer Plus. Sams, 2004.
- [11] Building and Using DLLs. Available:
<https://cygwin.com/cygwin-ug-net/dll.html>.