PROGRAMMING PRINCIPLES IN C

UNIT ASSIGNMENT REPORT

Russell Sammut-Bonnici
CPS1011
January 1, 2018

Contents

Question 1 – Problem Solving

Task 1a – Interest and Investments 1
Task 1b –Grocery Shopping System 4
Task 1c –Language Detection 12
Task 1d – Typo Detection and Correction 16
Task 1e – View Stack Frame 26

Question 2 – Hash Tables

Task 2a – Fixed 2D Array Version 28
Task 2b – Dynamic 2D Array Version 35
Task 2c – Linked List Version 42
Task 2d – Test Driver 43

Question 1 – Problem Solving

Task 1a – Interest and Investments

In this task, a program was required to find the amount of years it takes for Joan's invested sum at 10% interest to overtake Tom's invested sum at 15%. Both have invested €200, and Joan's investment earns a compound interest annually whereas Tom's earns a simple interest.

Functions Declared

- calcInterest(double amount, int rate)
  This method accepts the amount in euros and the rate as an integer, it would then return the compound or simple interest for one year by finding the product of the amount and rate divided by a hundred. (divided by a hundred as rate is converted from an integer to a percentage)

Variables Declared

- double x
  This variable is used to store Tom's simple interest, starting at the principle that is 200. Then, through a for loop it accumulates the interest for each year with the calcInterest() by having the initial investment inserted as the amount for each year.

- double y
  This variable is used to store Joan's compound interest, starting at the principle that is 200. Then, through a for loop it accumulates the interest for each year with the calcInterest() by having the previous year's accumulated interest inserted as the amount for each year.

- int year
  This variable is used to initialize the year counter i and then it becomes set to the value of the year when Joan's invested sum overtakes Tom's.

- int i
  This variable is used as a counter in the for loop, since it cannot be accessed outside the local scope year is set to its value each iteration, and finally what is printed is the final successful year

How the problem was solved

The problem was solved by first initializing x and y as 200. A for loop was then used to loop until y exceeded x. The counter i was incremented every iteration. Inside the loop Tom's yearly interest was calculated with calcInterest() and accumulated to x and Joan's interest was also calculated with calcInterest() and accumulated to y.

To keep a track of each year, information of the variables was printed out each iteration. Finally, when exiting the loop, the program prints when Joan's interest exceeds Tom's.

Source code (task_a.c)

```
// Exercise 1a. Problem Solving
// Created by Russell Sammut-Bonnici on 08/11/2017.
// CPS1011

#include <stdio.h>

/* function declaration */
double calcInterest(double amount,int rate);

//main method to evaluate different interests through time and compare
int main(){
    int year = 1;
    double x = 200; //x stores Tom's simple interest, starts at principle
    double y = 200; //y stores Joan's compound interest, starts at principle

    //for loop that calculates interest every year until Joan overtakes Tom
    for(int i = year;y<=x;i++){

        //Tom
        x += calcInterest(200,15); //accumulates simple interest per year

        //Joan
        y += calcInterest(y,10); //accumulates compound interest per year


        //structurally outprints interests each year
        printf("----------Year %d----------\nTom\t\tJoan\n%.2lf\t\t%.2lf\n\n",i,x,y);


        //keeps track of years each iteration so that years can be read from outside the loop
        year = i;
    }

    //prints result
    printf("Joan's interest exceeds Tom's after %d years when Tom's is %.2lf and Joan's is %.2lf\n",year,x,y);

    return 0;
}

//method to calculate compound or simple interest for one year
double calcInterest(double amount,int rate){
    return amount * rate/100;
}
```

Output Listing

```
C:\Users\rsamm\CLionProjects\assignment\question_1\cmake-build-debug\task_a.exe

----------Year 1----------

Tom             Joan

230.00          220.00


----------Year 2----------

Tom             Joan

260.00          242.00
```

```
----------Year 3----------
Tom             Joan
290.00          266.20

----------Year 4----------
Tom             Joan
320.00          292.82

----------Year 5----------
Tom             Joan
350.00          322.10

----------Year 6----------
Tom             Joan
380.00          354.31

----------Year 7----------
Tom             Joan
410.00          389.74

----------Year 8----------
Tom             Joan
440.00          428.72

----------Year 9----------
Tom             Joan
470.00          471.59

Joan's interest exceeds Tom's after 9 years when Tom's is 470.00 and Joan's is 471.59

Process finished with exit code 0
```

Task 1b –Grocery Shopping System

In this task, a program was required to function as a specified grocery shopping system. The system would stay prompting the user to choose between three specified vegetables and input how much kilograms wanted. The program would then stop when the user enters q, and prompts whether the user wants a text file copy of the receipt or not. (The system includes shipping costs and discounts)

Constants defined

- DISCOUNT 0.05
  Constant DISCOUNT defined to store the percentage discount to be applied to the cost when specifications are met.

Functions Declared

- double calcCost(int amount,double cost_per_kg)
  This method accepts the amount in kilograms of a vegetable ordered and the initial cost per kilogram of the said vegetable. It then returns the product of the amount and cost per kilogram.

- char getCharValidation()
  This method validates character input and returns a validated character. In it it makes use of the library function getchar() to get user input and then passes it through a list of error checks to reinforce input validation. These error checks include, checking whitespaces after the character, checking if it is alphanumeric and checking if it is not one character. Finally, it returns the inputted character if valid. If invalid it loops until user input is valid.

- int getIntValidation()
  This method scans an integer and puts it through a list of error checks and returns an error when invalid, whereas when valid it returns the validated integer. It checks to see if the input is in fact an integer and if it is a positive number.

- double calcShipping(int total_amount)
  This method calculates the shipping cost by checking the value of the total_amount. If less than or equal to 5, it returns 6.50. If in between the range of 5 and 20 it returns 14. If more than or equal to 20 it returns 14 + ½ * total_amount.

- printReceipt(.....)
  This method accepts all the variables related to the cost and prints out a structured receipt to the output panel. It only prints out the discount if discount is detected as true.

- printReceiptToText(.....)
  This method works similarly to the previous except it print the receipt to a text file "receipt.txt" instead. This works using FILE pointer *f, setting it to library function fopen() to write, and then having an error case if something goes wrong during writing.

Variables Declared

- char input
  This variable is used to store the user input after validated by calling getCharValidation(). Here it is used for the menu for the user to inbut 'a','b','c' or 'q'. In the program the character inputted is then changed to uppercase using the library function toupper() for case in-sensitivity.

- char receipt
  This variable is used for character input 'Y' or 'N' when the user is prompted whether they would like a copy of the receipt. It also makes use of getCharValidation().

- int amount_a, amount_b, amount_c
  Amounts of vegatables in kg all firstly initialized to 0. It is accumulated to each time the user selects vegetable a,b or c and inputs a valid integer amount. This amount is temporarily stored in temp then added to these amounts. This allows the program to keep track of the total kg required after each iteration.

- int temp
  This variable is temporarily stores the amount in kg to then be accumulated to amount_a, b or c dependent on what the user selected in the switch case.

- double cost_a, cost_b, cost_c
  The final price of the vegetables. Stores the calculated cost from calcCost() after the user enters 'q' breaking the while loop surrounding the switch case.

- double cost_reduction
  This variable stores the cost reduction after a discount is applicable. With an if statement, the discount of 5% off is only applicable when the total_cost is more than or equal to 100.

- int total_amount
  This variable stores the total amount in kg of vegetables, this is used when calculating the shipping cost.

- double total_cost
  This variable stores the total order cost in euros. It gets reduced if a discount applies, and added to with the sipping cost.

- double shipping_cost
  Stores the shipping cost which is calculated using the calcShipping() function which accepts the total amount kg as a parameter and returns the shipping cost.

How the problem was solved

Firstly, the user is prompted with a welcome message and a list of which characters refer to which vegetable and that 'q' refers to proceeding to check out. A while loop was used around a switch case to handle each case after user input. User validation was included to maximize functionality.

When 'q' is inputted, preferably after amounts in kg of vegetables are requested by the user, the total cost is then calculated and any discounts and shipping costs then modify the final price. A receipt displaying all the costs and possible discounts are then printed to the user, and the user is then prompted whether they would like a copy of the receipt or not.

If no the program just ends, if yes then the receipt is printed out to the text file "receipt.txt" which is saved in the "cmake_debug" folder. Also, user input validation is applied to the receipt prompt as well as when prompted to enter the wanted amount of kg of a vegetable and when selecting a vegetable.

Source Code (task 1b.c)

```
// Exercise 1b. Problem Solving
// Created by Russell Sammut-Bonnici on 09/11/2017.
// CPS1011

#include <stdio.h> //used for scanf() and printf()
#include <stdbool.h> //used for boolean
#include <ctype.h> //used for character validation
#include <mem.h> //used for strlen() function

#define DISCOUNT 0.05 //constant defining percentage discount

/* function declaration */
double calcCost(int amount,double cost_per_kg);
char getCharValidation();
int getIntValidation();
double calcShipping(int total_amount);

void printReceipt(int amount_a, int amount_b, int amount_c, double cost_a,
                  double cost_b, double cost_c, int total_amount, double total_cost,
                  double shipping_cost, bool discount, double cost_reduction);

int printReceiptToText(int amount_a, int amount_b, int amount_c, double cost_a,
                   double cost_b, double cost_c, int total_amount, double total_cost,
                   double shipping_cost, bool discount, double cost_reduction);

int main(){

    char input='x'; //input initially set as 0 so it isn't equal to 'q'
    char receipt; //input for if user wants a copy of the receipt or not

    int amount_a = 0;
    int amount_b = 0; //amounts of vegetables in kg initialised to 0
    int amount_c = 0;

    int temp; //temporarily stores inputted kg to accumulate to amount a,b or c

    double cost_a;
    double cost_b; //final price of vegetables
    double cost_c;

    double cost_reduction = 0; //cost reduction for when discount applies

    int total_amount; //stores total vegetable amount in kg
    double total_cost; //stores total order cost in euros
    double shipping_cost = 0; //shipping and handling cost

    bool discount = false; //boolean discount initialised for condition

    printf("Welcome to YourGreens.com!\n"); //welcome message
    printf("a) Artichokes\nb) Onions\nc) Carrots\nq) Proceed to checkout\n"); //displays menu

    //loop while input isn't 'q'
    while(input!='Q'){

        printf("Input a character to select an option:\n"); //prompt
        input = getCharValidation(); //scans char input, ignoring whitespaces after character
```

```
        input = (char)toupper(input); //changes to uppercase fo case in-sensitivity


        //using switch cases for choices
        switch(input){
            case 'A':
                printf("Enter the amount of artichokes desired in kg:\n"); //prompt

                //validates integer input and returns int to temp
                temp = getIntValidation();

                amount_a += temp; //accumulates total amount
                printf("You have %dkg of artichokes in your cart\n", amount_a);//outprints updated amount

                //breaks outside of switch case
                break;


            case 'B':
                printf("Enter the amount of onions desired in kg:\n");//prompt

                //validates integer input and returns int to temp
                temp = getIntValidation();

                amount_b += temp; //accumulates total amount
                printf("You have %dkg of onions in your cart\n", amount_b);//outprints updated amount

                //breaks outside of switch case
                break;

            case 'C':
                printf("Enter the amount of carrots desired in kg:\n");//prompt

                //validates integer input and returns int to temp
                temp = getIntValidation();

                amount_c += temp; //accumulates total amount
                printf("You have %dkg of onions in your cart\n", amount_c);//outprints updated amount

                //breaks outside of switch case
                break;

            case 'Q':
                printf("\nThank you for shopping with YourGreens.com!\n\n");
                break;//breaks outside of switch case

            default: //default case
                printf("Input not recognised! Select from the given list\n");//prompt
        }

    }

    //calculates total kg
    total_amount = amount_a + amount_b + amount_c;

    //calculates costs of each vegetable type
    cost_a = calcCost(amount_a,2.05);
    cost_b = calcCost(amount_b,1.15);
    cost_c = calcCost(amount_c,1.09);

    //calculates total vegetable cost
    total_cost = cost_a + cost_b + cost_c;

    //user gets 5% off Discount when order is 100 or more prior to shipping cost
    if(total_cost>=100) {
        discount = true; //set to true for condition when printing
        cost_reduction = total_cost * DISCOUNT;
        total_cost -= cost_reduction;
    }

    //calculates shipping cost and accumulates to total cost
    shipping_cost = calcShipping(total_amount);
    total_cost += shipping_cost;

    //structurally outprints receipt
    printReceipt(amount_a, amount_b, amount_c, cost_a, cost_b, cost_c, total_amount, total_cost,
shipping_cost, discount, cost_reduction);
```

```
        printf("Would you like a copy of this receipt? (Y/N)\n"); //prompts user

    do {
        receipt = getCharValidation();

        if (receipt == 'Y' || receipt == 'y') { //user wants a copy of the receipt

            //structurally prints to receipt.txt file
            printReceiptToText(amount_a, amount_b, amount_c, cost_a, cost_b, cost_c, total_amount, total_cost,
                               shipping_cost, discount, cost_reduction);

            printf("Receipt printed.\n"); //confirms receipt was printed.

        } else if (receipt == 'N' || receipt == 'n') { //user doesn't want a copy of the receipt
            return 0;
        } else {
            printf("Input not recognised!\n");
            printf("Please enter 'Y' or 'N':\n");
        }
    }while(receipt != 'Y' && receipt != 'y' );

    return 0;
}

//method that calculates cost of vegetable type a,b or c
double calcCost(int amount,double cost_per_kg){
    return amount*cost_per_kg;
}

//method that validates character input and returns validated character
char getCharValidation() {

    int i; //used as counter for characters in input
    char ch; //inputted character

    do {
        //scans char input, ignoring whitespaces after character
        ch = (char)getchar();
        i=1; //i initialised to 1

        //checks if input is character
        if (isalpha(ch) == 0 || ch == '\n') {
            printf("Error: Input must be an alphanumeric character\n");
        }

        //counts inputted character stream in buffer
        while (getchar() != '\n') {
            i++;
        }

        //checks if input is not one character
        if (i != 1) {
            printf("Error: Input must be one character\n");
        }

    }while(isalpha(ch) != 0 && ch != '\n' && i != 1);

    return ch;
}

//method that gets integer after user validation process
int getIntValidation(){

    char term; //used for user validation for temp
    int temp; //used for temporarily storing amount

    if(scanf("%d%c", &temp, &term) != 2 || term != '\n'){//scans amount in kg and checks if int
        printf("Error: Amount must be an integer\n");
        return 0;
    }
    else if(temp<=0){ //checks if negative or nothing
        printf("Error: Amount exceeds possible range. Please enter a positive number\n");
        return 0;
    }
    else{
        return temp;
    }
```

```
}

//method that calculates shipping cost and returns it
double calcShipping(int total_amount){

    double shipping_cost = 0; //shipping and handling cost

    if(total_amount<=5) { //shipping cost is 6.50 eu when under 5 kg
        shipping_cost = 6.50;
        return shipping_cost;
    }
    else if(total_amount>5 && total_amount<20){ //shipping cost is 14 eu when over 5 kg but under 20 kg
        shipping_cost = 14;
        return shipping_cost;
    }
    else if(total_amount>=20){ //shipping cost is 14 + 0.50 Eur/kg when over 20 kg
        shipping_cost = 14 + 0.5*total_amount;
        return shipping_cost;
    }
}

//method that prints receipt requiring arguments for all the needed variables for doing so
void printReceipt(int amount_a, int amount_b, int amount_c, double cost_a, double cost_b, double cost_c,
                  int total_amount, double total_cost, double shipping_cost, bool discount, double
cost_reduction){

    printf("---------Receipt-----------\n\t\tWEIGHT\tPRICE\n");

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

    if(amount_a>0) //prints artichokes if added to cart
        printf("Artichokes:\t%dkg  \t%.2lfeu\n",amount_a,cost_a);
    if(amount_b>0) //prints onions if added to cart
        printf("Onions:    \t%dkg  \t%.2lfeu\n",amount_b,cost_b);
    if(amount_c>0) //prints carrots if added to cart
        printf("Carrots:   \t%dkg  \t%.2lfeu\n",amount_c,cost_c);

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

    printf("Shipping:\t%dkg  \t%.2lfeu\n",total_amount,shipping_cost); //prints shipping cost

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

    if(discount==true) //prints out discount when value is true i.e. cost >= 100 (used ASCII for %)
        printf("%c5 off\nDISCOUNT:\t\t%.2lfeu\n", 37, cost_reduction);

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

    printf("Total:\t\t%dkg  \t%.2lfeu\n",total_amount,total_cost); //prints total
}

//method that prints receipt to text file, requiring all the needed variables for doing so
int printReceiptToText(int amount_a, int amount_b, int amount_c, double cost_a, double cost_b, double cost_c,
                       int total_amount, double total_cost, double shipping_cost, bool discount, double
cost_reduction) {

    //declares file pointers and uses fopen() to open text file to write
    FILE *f;
    f = fopen("receipt.txt", "w");

    if (f == NULL)
    {
        printf("Error opening file.\n");
        return 0;
    }

    fprintf(f, "---------Receipt-----------\n\t\tWEIGHT\tPRICE\n");

    fprintf(f,"----------------------------\n");

    if(amount_a>0) //prints artichokes if added to cart
        fprintf(f, "Artichokes:\t%dkg  \t€%.2lf\n",amount_a,cost_a);
    if(amount_b>0) //prints onions if added to cart
        fprintf(f, "Onions:    \t%dkg  \t€%.2lf\n",amount_b,cost_b);
    if(amount_c>0) //prints carrots if added to cart
        fprintf(f, "Carrots:   \t%dkg  \t€%.2lf\n",amount_c,cost_c);

    fprintf(f,"----------------------------\n");
```

```
    fprintf(f, "Shipping:\t%dkg  \t€%.2lf\n",total_amount,shipping_cost); //prints shipping cost

    fprintf(f,"----------------------------\n");

    if(discount==true) //prints out discount when value is true i.e. cost >= 100 (used ASCII for %)
        fprintf(f, "%c5 off\nDISCOUNT:\t\t€%.2lf\n", 37, cost_reduction);

    fprintf(f,"----------------------------\n");

    fprintf(f, "Total:\t\t%dkg  \t€%.2lf\n",total_amount,total_cost); //prints total

    //closes file
    fclose(f);

}
```

Output Listing

```
C:\Users\rsamm\CLionProjects\assignment\question_1\cmake-build-debug\task_b.exe

Welcome to YourGreens.com!

a) Artichokes

b) Onions

c) Carrots

q) Proceed to checkout

Input a character to select an option:

a

a

Enter the amount of artichokes desired in kg:

50

50

You have 50kg of artichokes in your cart

Input a character to select an option:

b

b

Enter the amount of onions desired in kg:

50

50

You have 50kg of onions in your cart

Input a character to select an option:

c

c

Enter the amount of carrots desired in kg:

50
```

```
50

You have 50kg of onions in your cart

Input a character to select an option:

q

q


Thank you for shopping with YourGreens.com!


---------Receipt-----------

                WEIGHT  PRICE

------------------------------

Artichokes:     50kg    102.50eu

Onions:         50kg    57.50eu

Carrots:        50kg    54.50eu

------------------------------

Shipping:       150kg   89.00eu

------------------------------

%5 off

DISCOUNT:               10.73eu

------------------------------

Total:          150kg   292.77eu

Would you like a copy of this receipt? (Y/N)

Y

Y

Receipt printed.


Process finished with exit code 0
```

Task 1c –Language Detection

In this task, a program was required to read an input text file and differentiate between whether it is a C source file (by the case sensitive phrase "#include") or whether it is an HTML file (by the case in-sensitive opening phrase "<html>" and closing phrase "<\html>"). A margin of imprecision was also permitted, so in the solution it wouldn't detect if it was an "incomplete" html file (example when it would include the opening tag but not the closing tag) also it would still detect the file as HTML if </html> came before <html>.

Constants Defined

- INPUTFILE "inputText_2.txt"
  Constant INPUTFILE defined as either "inputText_1.txt", "inputText_2.txt" or "inputText_3.txt". The first contains C source code for printing "Hello World", the second contains an HTML alternative and the third a Python alternative. These are all used to test if the program can distinguish between C, HTML and other.

- PHRASE_C "#include"
  Phrase used as needle into the strstr() method to check if it is found with case sensitivity in the haystack that is the character array, char string[].

- PHRASE H1 "<html>"
  Phrase used as needle into the strstri() method to check if it is found with case insensitivity in the haystack that is the character array, char string[].

- PHRASE H2 "</html>"
  Phrase used as needle into the strstri() method to check if it is found with case insensitivity in the haystack that is the character array, char string[].

- MAX_L 1000
  Constant MAX_L used to define the max array size of the string that stores the input file.

Functions Declared

- int strstri(const char *haystackIn,const char *needleIn)
  This method accepts two string literals. One is the haystack and the other is the needle. The method then copies the string literal from a character pointer to a character array using the library function strcpy(). This is done so that the library function toupper() can be used to turn both string literals into uppercase, allowing case insensitivity. It then passes these two capitalized strings into the library function strstr() which finds a needle in the haystack.

Variables Declared

- char string[MAX_L]
  Char array, string used to store the string from the inputted to text file, this way string functions strstr() and strstri() can be used to search for a case-sensitive and case-insensitive match respectively.

- int i
  This integer is used as a counter for the array string when copying char by char from the text file to char string. This is done using a while loop, FILE pointer *f and the library functions fopen() and fgetc().

- char ch
  This variable stores the current character retrieved from the text file with fgetc(). It is then stored to char string[] and repeated until the end of file denoted by the keyword EOF.

How the problem was solved

The program loads up one of the three created text files that store code of type C, HTML or 'other'. It loads the text file character by character and stores each character into char string[]. It then proceeds to print the string after the loop reaching the end of file of the text file.

It then using the string library function strstr() to find the phrase "#include". If found then the program prints that the text file is a C source file.

If the if statement fails it goes to the If else statement after it which check if its an HTML by using the declared function strstri(). It works similarly to strstr() but is case insensitive in result of its switching to uppercase no matter the input. If it succeeds with both finding "<html>" and "</html>" in the haystack, the program prints that the inputted text file is an HTML file.

The else statement then prints that the text file is of type other (which in this test case is Python).

Source Code (task_c.c)

```
//Exercise 1c. Problem Solving
//Created by Russell Sammut-Bonnici on 13/11/2017.
//CPS1011

#include <stdio.h>
#include <string.h> //for finding substring in string
#include <ctype.h> //for toupper() and isalpha()

#define INPUTFILE "inputText_2.txt" //constant defining file name

#define PHRASE_C "#include"
#define PHRASE_H1 "<html>"
#define PHRASE_H2 "</html>"

#define MAX_L 1000 //Max length of text file defined as 1000 characters

/* function declaration*/
int strstri(const char *haystackIn,const char *needleIn);
```

```
int main(){

    char string[MAX_L]; //char array called string stores text file
    int i=0; //counter for array string

    //we copy from const pointer to array to be able to use isupper()

    //declaring inputFile names and file pointers
    //character arrays store each character after reading the textFiles
    char inputFile[] = INPUTFILE;
    char ch;
    FILE *f;

    //Reading input text file using file pointer and methods from <string.h>
    if((f = fopen(inputFile, "r")) != NULL)
    {
        //while read character isn't at the end of file, loop
        //scanning every character until the end of file
        while((ch = fgetc(f)) != EOF) {
            string[i]=ch;
            i++;
        }
    }
    else //exception of retrieval failure
    {
        printf("Error: %s was not found.\n", inputFile);
        return 1;
    }

    //prints string, that is the text file
    printf("%s\n",string);
    printf("----------------------------------------\n");

    //strstr() returns true when it finds a substring in string (needle in a haystack)
    //strstri() works like strstr() except it is case-insensitive

    if(strstr(string, PHRASE_C) ) //if match is found at the beginning of string, it prints
        printf("The text file %s is a C source file.\n",inputFile);
    else if( strstri(string,PHRASE_H1) && strstri(string, PHRASE_H2) ) //if both function statements for HTML
returns true, it prints
        printf("The text file %s is an HTML file.\n",inputFile);
    else //if conditions returns false, it prints
        printf("The text file %s is an unknown 'other type' of file.\n",inputFile);

    //closes file
    fclose(f);

    return(0);
}

//method searches for case insensitive match, returns 1 (true) when match is found
int strstri(const char *haystackIn,const char *needleIn){

    //declare arrays
    char haystack[MAX_L];
    char needle[10];

    //copy string from pointer to arrays
    strcpy(haystack,haystackIn);
    strcpy(needle,needleIn);

    //changes needle to uppercase
    for(int i=0;i<strlen(needle);i++) {
        //converts character if alphanumeric
        if(isalpha(needle[i])!=0) {
            needle[i] = (char) toupper(needle[i]);
        }
    }

    //changes haystack to uppercase
    for(int i=0;i<strlen(haystack);i++) {
        //converts character if alphanumeric
        if(isalpha(haystack[i])!=0) {
            haystack[i] = (char) toupper(haystack[i]);
        }
    }

    //scans for case-insensitive match
```

```
    if(strstr(haystack,needle) )
        return 1;
}
```

Output Listing

```
C:\Users\rsamm\CLionProjects\assignment\question_1\cmake-build-debug\task_c.exe

<HtML>

<header><title>This is title</title></header>

<body>

Hello world

</body>

</htMl>

----------------------------------------

The text file inputText_2.txt is an HTML file.


Process finished with exit code 0
```

Task 1d – Typo Detection and Correction

In this task, a program was required to read an input file and detect specified typing errors that are space before comma or full-stop, multiple spaces as opposed to single and missing spaces. A specific heuristic was given so that the typo is only detected when a word has more than 12 characters and is not hyphenated. The user is then prompted to confirm auto-correction for just these cases.

Constants Defined

- INPUTFILE "inputText_d.txt"
  Constant INPUTFILE defined as "inputText_d.txt", this text file contains a paragraph filled with the specified typos for testing.

- OUTPUTFILE "outputText_d.txt"
  Constant OUTPUT defined as "outputText_d.txt" to save the corrected text after all the typo corrections are agreed to by the user.

- MAX_L 1000
  Constant MAX_L used to define the max array size of the string that stores the input file.

Functions Declared

- void replace(char * string, char * typo, char * correction)
  This method replaces phrase typo with phrase correction in the haystack string. It makes use of the string library functions strncpy(), strcpy(), sprint(), strlen() and recursion. Its base case is set when the string literal pointer returns null when set to strstr(string, typo). So when all the instances are replaced it stops performing recursion.

- char getCharValidation()
  This method validates character input and returns a validated character. In it it makes use of the library function getchar() to get user input and then passes it through a list of error checks to reinforce input validation. These error checks include, checking whitespaces after the character, checking if it is alphanumeric and checking if it is not one character. Finally, it returns the inputted character if valid. If invalid it loops until user input is valid.

- int confirmPrompt()
  Method that loops prompt confirmation with Y or N. This is used when the user is asked whether they would like the typo corrected or not. It returns 1 when the user agrees, if the user disagrees then the typo is left as is. If an unrecognized character is entered then the program prompts the user to enter a recognized character. Case in-sensitivity is also accounted for.

- void printToTextFile(char *string)
  Method used to print autocorrected string to text file after all the typos are detected and the user chooses to fix or not fix them. The method uses a FILE pointer *f and library functions fopen(), fprintf() and fclose().

- void correctSpaceCommaFullstop(char * string)
  Method detects the space before comma or fullstop typo and prompts the user whether it would like it correct. If confirmPrompt() returns 1, which is when the user agrees, then it uses the replace() method to replace all instances of " ," with "," and " ." with ".". After this it then prints the current state of the string, that is, after the correction.

- void correctMultipleSpaces(char * string)
  Method detects multiple spaces typo and prompts the user whether it would like it correct. If confirmPrompt() returns 1, which is when the user agrees, then it uses locally declared pointers *from and *to and int spc. Unification is established by setting *to to *from to *string. A while loop is then used to replace all multiple spaces with a single space. After this it then prints the current state of the string, that is, after the correction.

- void correctMissingSpaces(char *string)
  Method detects the missing space typo. It does this by scanning every word in the string (using loops and string functions memset() and strcopy()) and storing any detected words which are more than 12 characters and not hyphenated into string array typo[][]. The rows store the first letter, whereas the columns store the other letters.

  After a typo is detected and stored, it prompts the user whether it would like it to correct or not, along with the detected misspelt words. If confirmPrompt() returns 1, which is when the user agrees, then it tells the user to input the typo with spaces in the places they were missing. The replace() function is then called to replace the typo with the correction. After this it then prints the current state of the string, that is, after the correction.

Variables Declared

- char string[MAX_L]
  Char array, string used to store the string from the inputted to text file, this way string functions strstr() and strstri() can be used to search for a case-sensitive and case-insensitive match respectively.

- int i
  This integer is used as a counter for the array string when copying char by char from the text file to char string. This is done using a while loop, FILE pointer *f and the library functions fopen() and fgetc().

- char ch
  This variable stores the current character retrieved from the text file with fgetc(). It is then stored to char string[] and repeated until the end of file denoted by the keyword EOF.

- char temp[MAX_L]
  Locally declared in replace(). Used to temporarily store string before the first occurrence of typo. It is then appended using sprint() and copied into string using strcpy(). The variable is declared at multiple levels due to recursion until the base case is reached.

- char * pointer
  Locally declared in replace(). Used to store what is in the base case, set to strstr(string, typo). Then is used when rising back through the levels after the base case to help replace all instances.

- char word[50]
  Locally declared in correctMissingSpaces(). Used for temporarily storing words from the text.

- char correction[50]
  Locally declared in correctMissingSpaces(). Used for temporarily storing input correction which is then to replace the typo in the text.

- char typo[50][50]
  Locally declared in correctMissingSpaces(). Used for storing strings in an array which are in this case typos. An if statement stores into this when the word follows the heuristic that it contains 12 characters and has no hyphen.

- int i
  Locally declared in correctMissingSpaces(). Used as a counter for string array index (for the text to be scanned then corrected)

- int j
  Locally declared in correctMissingSpaces(). Used as a counter for storing the current word's length. This is used to then detect whether it has more than 12 characters.

- int n
  Locally declared in correctMissingSpaces(). Used as a counter for copying typos into and from the 2D string array typo[][].

How the problem was solved

The program loads up the input text file "InputText_d.txt". In it the text file had the requested typos needed to be fixed. After copying the text file contents to a string it then proceeds to call the three correction methods correctSpaceCommaFullstop(), correctMultipleSpaces() and correctMissingSpaces(). The string is passed through each of them and then returned fixed if the user agrees to fixing the specific type of typo detected. After going through each of these methods, the program then prints the fixed string to an output text file "outputText_d.txt".

Source Code (task_d.c)

```
// Exercise 1d. Problem Solving
// Created by Russell Sammut-Bonnici on 21/11/2017.
// CPS1011

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

#define INPUTFILE "inputText_d.txt" //constant defining input file
#define OUTPUTFILE "outputText_d.txt" //constant defining output file
#define MAX_L 1000 //sets max length for string

/* function declaration */
void replace(char * string, char * typo, char * correction);
char getCharValidation();
int confirmPrompt();
void printToTextFile(char *string);
void correctSpaceCommaFullstop(char * string);
void correctMultipleSpaces(char * string);
void correctMissingSpaces(char *string);

int main() {

    char string[MAX_L] = ""; //char array called string stores text file, initialised to empty to avoid
garbage characters
    int i = 0; //counter for array string

    //INPUTFILE name and file pointers
    //character arrays store each character after reading the textFiles
    char ch;
    FILE *f;
    f = fopen(INPUTFILE, "r");

    //Reading input text file using file pointer and methods from <string.h>
    if (f != NULL) {
        //while read character isn't at the end of file, loop
        //scanning every character until the end of file
        while ((ch = (char) fgetc(f)) != EOF) {
            string[i] = ch;
            i++;
        }
    } else //exception of retrieval failure
    {
        printf("Error: \"%s\" was not found.\n", INPUTFILE);
        return 1;
    }

    fclose(f);

    //prints string, that is the text file
    printf("%s\n", string);
    printf("----------------------------------------\n");

    //calls auto-correct methods
    correctSpaceCommaFullstop(string);
    correctMultipleSpaces(string);
    correctMissingSpaces(string);

    //print to text file
    printToTextFile(string);
    printf("The corrected string has been saved to \"%s\"\n",OUTPUTFILE);


    return 0;
}

//method that searches and replaces substring in string string
void replace(char * string, char * typo, char * correction) {

    //a temp variable to do all replace things
    char temp[MAX_L];
    //to store what is returned from base case strstr()
    char * pointer;

    //base case
```

```
    if(!(pointer = strstr(string, typo)))
        return;

    //copy all the content to temp before the first occurrence of the typo string
    strncpy(temp, string, pointer-string);

    //prepare the temp for appending by adding a null to the end of it
    temp[pointer-string] = 0;

    //append using sprintf() function
    sprintf(temp+(pointer - string), "%s%s", correction, pointer + strlen(typo));

    //empty string for copying
    string[0] = 0;
    strcpy(string, temp);

    //pass recursively to replace other occurrences
    return replace(string, typo, correction);
}

//method that validates character input and returns validated character
char getCharValidation() {

    int i; //used as counter for characters in input
    char ch; //inputted character

    do {
        //scans char input, ignoring whitespaces after character
        ch = (char)getchar();
        i=1; //i initialised to 1

        //checks if input is character
        if (isalpha(ch) == 0 || ch == '\n') {
            printf("Error: Input must be an alphanumeric character\n");
        }

        //counts inputted character stream in buffer
        while (getchar() != '\n') {
            i++;
        }

        //checks if input is not one character
        if (i != 1) {
            printf("Error: Input must be one character\n");
        }

    }while(isalpha(ch) != 0 && ch != '\n' && i != 1);

    return ch;
}

//method that returns 1 when confirmed
int confirmPrompt(){

    char confirmation;

    printf("Would you like to auto-correct this typo? (Y/N):\n");

    //do while loop so that it loops until character is recognised
    do {

        confirmation = getCharValidation();

        if (confirmation == 'y' || confirmation == 'Y') {
            return 1;
        } else if (confirmation == 'n' || confirmation == 'N') {
            printf("Confirmed. The typo has been left as before.\n");
        } else {
            printf("Error: input not recognised.\n");
            printf("Please enter a recognised character (Y/N).\n");
        }
    }while(confirmation != 'n' && confirmation != 'N');
}

void printToTextFile(char *string){
    //opens output text file, prepared for writing
    FILE *f;
    f = fopen(OUTPUTFILE, "w");
```

```
    if (f == NULL)
    {
        printf("Error opening file.\n");
        return;
    }

    //prints auto-corrected string to text file
    fprintf(f, string);

    //closes file
    fclose(f);
}

//method that auto-corrects space before commas and fullstops
void correctSpaceCommaFullstop(char * string){

    //if statement checks for space before comma or a full stop
    if( strstr(string," ,") || strstr(string," .") ){
        printf("Error: Space before comma or a full-stop\n");

        //if confirmed program fixes typo
        if(confirmPrompt()==1){

            //replaces typos with correction
            replace(string," ,",",");
            replace(string," .",".");

            printf("Confirmed. The typos have been auto-corrected.\n");
            printf("----------------------------------------\n");
            //prints string, that is the text file
            printf("%s\n",string);
            printf("----------------------------------------\n");
        }
    }
}

//method that auto-corrects multiple spaces
void correctMultipleSpaces(char * string) {

    //if statement checks for multiple spaces
    if (strstr(string, "  ")) { //finds more than one space
        printf("Error: Multiple spaces as opposed to a single space\n");

        //if confirmed program fixes typo
        if (confirmPrompt() == 1) {

            //declaring variables
            char *from, *to;
            int spc = 0;

            //establishing unification
            to = from = string;

            //loop to auto-correct multiple spaces
            while (1) {
                if (spc && *from == ' ' && to[-1] == ' ')
                    ++from;
                else {
                    spc = (*from == ' ') ? 1 : 0;
                    *to++ = *from++;
                    if (!to[-1])
                        break;
                }
            }

            printf("Confirmed. The typos have been auto-corrected.\n");
            printf("----------------------------------------\n");
            //prints string, that is the text file
            printf("%s\n", string);
            printf("----------------------------------------\n");
        }
    }
}

void correctMissingSpaces(char *string) {

    char word[50] = ""; //char array for storing words from the text
```

```
    char correction[50]; //input for replacement of word

    //initialise 2D char array (String array) to store typos when found
    char typo[50][50] = {'\0'};

    int i = 0; //counter used for string array index
    int j = 0; //counter used for storing current words length
    int n = 0; //counter used for typo array

    //while loop scans for typo until the end of the text string (the end of its length)
    while (i < strlen(string)) {

        //while loop builds up word from characters until scanning a space/punctuation
        while (string[i] != ' ' && string[i] != '.' && string[i] != ',' && string[i] != '!' && string[i] !=
'\n') {
            word[j] = string[i];
            j++;
            i++;
        }


        //if word is more than 12 characters and no '-' then word typo is found so it is stored
        if (j > 12 && (strstr(word, "-") == 0)) {

            //stores typo from word into array
            strcpy(typo[n], word);
            n++;
        }

        //clear contents within word and iterates location to scan next word
        memset(word, 0, strlen(word));
        j = 0;
        i++;

    }

    //detect error when 1 or more typos are found
    if (n > 0) {

        printf("Error: Missing spaces\n");
        printf("Words longer than 12 characters and not including a hyphen (-) are rare\n");

        if (n == 1) //singular
            printf("%d typo of this case was found.\n", n);
        else //plural
            printf("%d typos of this case were found.\n", n);

        //loop for each typo
        for (i = 0; i < n; i++) {
            printf("Typo %d: ", i + 1);
            printf("The detected word was: \"%s\"\n", typo[i]);

            //if confirmed program fixes typo
            if (confirmPrompt() == 1) {

                printf("Input below the correction for \"%s\"\n", typo[i]);

                //scans so that spaces corrected between words are accepted
                scanf("%[^\n]%*c", correction);

                //replaces typo with correction
                replace(string, typo[i], correction);

                printf("Confirmed. The typo has been auto-corrected.\n");
                printf("----------------------------------------\n");
                //prints string, that is the text file
                printf("%s\n", string);

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

        }
    }
}
```

Output Listing

```
C:\Users\rsamm\CLionProjects\assignment\question_1\cmake-build-debug\task_d.exe
John was running down the street. He realised he was   being followed ,
speeding up his pace , he started running . The manbehindhimstarted running too.
A memory startedrushingstraight to his   brain. The stress from the chase  triggered
his post-traumatic stress disorder. His  palmsweredrowning in sweat .
----------------------------------------
Error: Space before comma or a full-stop
Would you like to auto-correct this typo? (Y/N):
k
k
Error: input not recognised.
Please enter a recognised character (Y/N).
y
y
Confirmed. The typos have been auto-corrected.
----------------------------------------
John was running down the street. He realised he was   being followed,
speeding up his pace, he started running. The manbehindhimstarted running too.
A memory startedrushingstraight to his   brain. The stress from the chase  triggered
his post-traumatic stress disorder. His  palmsweredrowning in sweat.
----------------------------------------
Error: Multiple spaces as opposed to a single space
Would you like to auto-correct this typo? (Y/N):
y
y
Confirmed. The typos have been auto-corrected.
----------------------------------------
John was running down the street. He realised he was being followed,
speeding up his pace, he started running. The manbehindhimstarted running too.
A memory startedrushingstraight to his brain. The stress from the chase triggered
his post-traumatic stress disorder. His palmsweredrowning in sweat.
----------------------------------------
Error: Missing spaces
Words longer than 12 characters and not including a hyphen (-) are rare
```

```
3 typos of this case were found.
Typo 1: The detected word was: "manbehindhimstarted"
Would you like to auto-correct this typo? (Y/N):
y
y
Input below the correction for "manbehindhimstarted"
man behind him started
man behind him started
Confirmed. The typo has been auto-corrected.
----------------------------------------
John was running down the street. He realised he was being followed,
speeding up his pace, he started running. The man behind him started running too.
A memory startedrushingstraight to his brain. The stress from the chase triggered
his post-traumatic stress disorder. His palmsweredrowning in sweat.
----------------------------------------
Typo 2: The detected word was: "startedrushingstraight"
Would you like to auto-correct this typo? (Y/N):
y
y
Input below the correction for "startedrushingstraight"
started rushing straight
started rushing straight
Confirmed. The typo has been auto-corrected.
----------------------------------------
John was running down the street. He realised he was being followed,
speeding up his pace, he started running. The man behind him started running too.
A memory started rushing straight to his brain. The stress from the chase triggered
his post-traumatic stress disorder. His palmsweredrowning in sweat.
----------------------------------------
Typo 3: The detected word was: "palmsweredrowning"
Would you like to auto-correct this typo? (Y/N):
y
y
Input below the correction for "palmsweredrowning"
palms were drowning
```

```
palms were drowning

Confirmed. The typo has been auto-corrected.

----------------------------------------

John was running down the street. He realised he was being followed,

speeding up his pace, he started running. The man behind him started running too.

A memory started rushing straight to his brain. The stress from the chase triggered

his post-traumatic stress disorder. His palms were drowning in sweat.

----------------------------------------

The corrected string has been saved to "outputText_d.txt"


Process finished with exit code 0
```

Task 1e – View Stack Frame

In this task, a program was required to call a declared memory inspection function "view_stack_frame()." This method then displays the addresses of the passed variables then their corresponding values in two columns.

Constants Defined

- ARRAY_SIZE 10
  Constant ARRAY SIZE defined as 10 to set the size of the integer array collection, which is then passed into view_stack_frame().

Functions Declared

- void view_stack_frame(int array[])
  This method accepts an integer array and then it prints out the all the addresses and values of the variables within the integer array using a for loop. The layout it prints these out are in two columns.

Variables Declared

- int collection[ARRAY_SIZE]
  This array then stores integer variables which are automatically inputted through a for loop and a locally declared variable counter i used to set the value.

- int i
  There are two locally declared counter variables in this program. One within the main method, used to set the values to each position in the array. The other within the view_stack_frame() method used to help print out each position in the array.

How the problem was solved

The problem was solved by declaring an array called collection and setting it's size to a predefined constant ARRAY_SIZE. The array is the filled with a consecutive sequence made from a for loop. The array collection is then passed through view_stack_frame and its address and value for each array location is printed out using yet another for loop.

Source Code (task_e.c)

```
// Exercise 1e. Problem Solving
// Created by Russell Sammut-Bonnici on 07/12/2017.
// CPS1011
```

```
#include <stdio.h>

#define ARRAY_SIZE 10 //constant defining array size

/* function declaration */
void view_stack_frame(int array[]);

int main(){

    //declare array storing integer variables
    int collection[ARRAY_SIZE];

    //loop to automatically store varying values in the variables
    for(int i=0;i<ARRAY_SIZE;i++){
        collection[i] = i;
    }

    view_stack_frame(collection);
    return 0;
}

//view stack frame accepts array as an argument
void view_stack_frame(int array[]){

    //prints two columns
    printf("Address\t\tValue\n");
    printf("=======\t\t=====\n");

    //loops address and value contents of array
    for(int i=0;i<ARRAY_SIZE;i++){
        printf("%p\t%d\n",&array[i],array[i]);
    }

}
```

Output Listing

```
C:\Users\rsamm\CLionProjects\assignment\question_1\cmake-build-debug\task_e.exe

Address         Value

=======         =====

0061FF04        0

0061FF08        1

0061FF0C        2

0061FF10        3

0061FF14        4

0061FF18        5

0061FF1C        6

0061FF20        7

0061FF24        8

0061FF28        9


Process finished with exit code 0
```

Question 2 – Hash Tables

Task 2a – Fixed 2D Array Version

In this task, a program was required to be able to insert, delete, lookup, save and load a hash table, of which was a fixed 2D array version. The hash space and max collisions were requested to be pre-defined.

Structs Defined

- Pair
  The struct Pair stores each pair's corresponding key and value. Each key and value is stored as a string literal using constant character pointers.

- Bucket
  The struct bucket stores each row in the hash table. It points to the first pair in each row, or any pair after it.

- HashTable
  The struct hash table usually stores the hash space, but since in this version the hash space is pre-defined there is no need. It points to every row to access the data.

Constants Defined

- HASH_SPACE 20
  Constant HASH_SPACE defined as 20, to be used when defining how many buckets are to be allocated for the hash table.

- MAX_COLL 6
  Constant MAX_COLL defined as 6, used when defining the maximum amount of collisions in each bucket. If the max collision is reached then an error is printed.

Functions Declared

- HashTable * createHashTable_S()
  This method is used for creating static hash tables, using pre-defined constants. It does not require any arguments for the hash space. It allocates memory for the hash table that points to each bucket and it makes use of the pre-fixed HASH_SPACE to allocate memory for the amount of buckets.

- **int** hashFunction_S(**const char** * keyIn)
  This method is used for getting the hashValue for this version as it makes use of the pre-fixed HASH_SPACE, therefore it does not require any other arguments apart from the key. The hash value is calculated by getting the some of the characters' ASCII value and modulating it by the hash space of the hash table.

- **int** insertPair(HashTable * hashTable,**const char** * key, **const char** * value)
  This method inserts a pair into the hash table created. The hash value of the key inputted is first calculated, then it is inserted into the previously-existing bucket or newly-created bucket. If previously-existing then collision is checked and if collision is detected it is placed after the last pair, provided it does not exceed pre-fixed MAX_COLL.

- **int** deletePair(HashTable * hashTable, **const char** * key)
  This method deletes a pair from the hash table. It first checks if the bucket of the hashValue of the key exists. If it does not exist an error is printed. If it does it proceeds to scan the bucket for a pair with matching keys to the one requested. If it is found at the head then the whole bucket is deleted provided there are no following elements. If it is found at the head and there are more elements, the head is deleted and the following are shifted to the left. If it is found in the middle then the pair is deleted and all the following elements are shifted to the left. If it is found at the end then the element at the end is just freed.

- **int** checkExists(HashTable * hashTable, **const char** * key)
  This method looks up a pair in the hash table and checks if it exists or not. It returns 1 when it exists and 0 when it does not. It checks every column of pair until the end for a match with the inputted key.

- **int** saveHashTableAs(HashTable * hashTable,**const char** * fileName)
  This method saves the hash table as a structured text file. Two loops are used for printing the rows and columns to the text file. A "," is used in between keys and values, A "\t" is used in between different pairs and a "\n" is used in between different buckets.

- **int** loadHashTableTo(**const char** *fileName,HashTable * hashTable)
  This method loads the previously saved hashTable and adds every pair from it to the created hash table. It works with scanning the structure for pairs with linked keys and value (by ',') and it then calls insertPair().

- **int** freeHashTable(HashTable * hashTable)
  This method frees the allocated memory of the created hash table. It uses a for loop to scan for existing buckets and free them if existing. After which it frees the hash table itself that points to the buckets.

Source Code (task_2a.c)

```
// Exercise 2a. Hash Tables - Fixed 2D Array version
// Created by Russell Sammut-Bonnici on 08/12/2017.
```

```
// CPS1011

#include "hashTable.h"

//creating a struct Pair to store HashTable's values and keys
typedef struct pair{
    const char * key; //the key to be inputted
    const char * value; //the value to be inputted
}Pair;

//creating struct for bucket including its head
typedef struct bucket{
    Pair ** pair; //pointer to data that is in pair
}Bucket;

//creating struct for hashTable including pointer to row
typedef struct hashTable{
    Bucket ** bucket; //double pointer to the beginning of the row
}HashTable;

//initializes hashTable and allocates requested space from hashSpace
HashTable * createHashTable_S(){

    //allocating space for hashTable which is just 1 item
    HashTable * hashTable = calloc(1,sizeof(hashTable));

    //allocating space for number of rows, which are equal to the hashSpace
    hashTable->bucket= calloc(HASH_SPACE, sizeof(Bucket));

    return hashTable; //returns created hash Table
}

//method that inserts pair into created hashTable
int insertPair(HashTable * hashTable,const char * key, const char * value){

    int hashValue = hashFunction_S(key); //calculate hashValue

    if(hashTable->bucket[hashValue]==NULL){//if bucket doesnt exist

        hashTable->bucket[hashValue] = malloc(sizeof(Bucket)); //allocate mem for new bucket

        if (hashTable->bucket[hashValue] == NULL) { //error check
            printf("Failed to allocate memory for new bucket at hashValue: %d.",hashValue);
            return 1;//fail
        }

        hashTable->bucket[hashValue]->pair = calloc(MAX_COLL, sizeof(Pair)); //allocate mem for three pairs in
bucket

        if (hashTable->bucket[hashValue]->pair == NULL) { //error check
            printf("Failed to allocate memory for array of pairs at hashValue: %d.",hashValue);
            return 1;//fail
        }

        hashTable->bucket[hashValue]->pair[0] = malloc(sizeof(Pair)); //allocate mem for first pair
        hashTable->bucket[hashValue]->pair[0]->key = key; //set key of first pair
        hashTable->bucket[hashValue]->pair[0]->value = value; //set value of first pair

    }else if(hashTable->bucket[hashValue]!=NULL){ //if bucket exists

        int c=0; //collision counter to count amount of collisions in bucket when inserting

        while(hashTable->bucket[hashValue]->pair[c]!=NULL){ //while pair exists
            c++; //increment collision counter
        }

        if(c<MAX_COLL){ //if c is less than max initial collisions
            hashTable->bucket[hashValue]->pair[c] = malloc(sizeof(Pair)); //allocate mem for pair
            hashTable->bucket[hashValue]->pair[c]->key = key; //set key after last collision
            hashTable->bucket[hashValue]->pair[c]->value = value; //set value after last collision

        }else{ //if c is equal to the ax collisions, no space for new pair, produce error
            printf("Error: Max_Collisions %d exceeded.",MAX_COLL);
            return 1;
        }

    }
```

```
    printf("Pair of key: \"%s\" and value: \"%s\" was successfully inserted into the hashTable, hashValue:
%d.\n",key,value,hashValue);

}

//method for hashFunction_S, accepts key and value as arguments
int hashFunction_S(const char * keyIn){

    int charSumKey = 0; //sum of the ASCII code of characters in key
    int x; //variable to be returned

    //accumulates sum of ASCII code of each character
    for(int i=0;i<strlen(keyIn);i++) {
        charSumKey += keyIn[i];
    }

    x = charSumKey % HASH_SPACE; //invented formula for hash function
    return x;
}

int deletePair(HashTable * hashTable, const char * key){

    int hashValue = hashFunction_S(key); //get hashValue

    if(checkExists(hashTable,key)==1){ //if exists, delete pair

        int c = 0; //counter for collision index
        int stop = 0; //to break from loop from if statement
        Bucket *scan = hashTable->bucket[hashValue];

        while(scan!=NULL && stop!=1){ //scans through list until match to delete is found

            if(strcmp(hashTable->bucket[hashValue]->pair[0]->key,key)==0){ //if match found at head

                if(hashTable->bucket[hashValue]->pair[1]==NULL){ //if head to delete is the only node in the
bucket delete head then bucket

                    hashTable->bucket[hashValue]->pair[0]->key=NULL;
                    hashTable->bucket[hashValue]->pair[0]->value=NULL;
                    free(hashTable->bucket[hashValue]->pair[0]);
                    free(hashTable->bucket[hashValue]->pair);
                    free(hashTable->bucket[hashValue]);

                }else if(hashTable->bucket[hashValue]->pair[1]!=NULL){ //if there is more than just one node
in the list, shift rest to left then delete last

                    while(scan->pair[c]!=NULL) { //while current node exists

                        if (scan->pair[c + 1] != NULL) { //if next node exists

                            scan->pair[c]->key = scan->pair[c + 1]->key; //copy, from next node to key
                            scan->pair[c]->value = scan->pair[c + 1]->value; //copy, from next node to value

                        }else{ //if next node doesn't exist (at end of list)

                            hashTable->bucket[hashValue]->pair[c]->key=NULL;
                            hashTable->bucket[hashValue]->pair[c]->value=NULL;
                            free(scan->pair[c]); //delete last element

                        }

                        c++; //increment counter

                    }
                }

                printf("Deletion of key: \"%s\" was successful at head.\n",key);
                stop=1; //stop scanning for match

            }else if(strcmp(scan->pair[c]->key,key)==0 && scan->pair[c+1]!=NULL){//if match found in middle,
shift everything after it to the left

                while(scan->pair[c]!=NULL) { //while current node exists

                    if (scan->pair[c + 1] != NULL) { //if next node exists

                        scan->pair[c]->key = scan->pair[c + 1]->key; //copy, from next node to key
                        scan->pair[c]->value = scan->pair[c + 1]->value; //copy, from next node to value
```

```
                    }else{ //if next node doesn't exist (at end of list)

                        hashTable->bucket[hashValue]->pair[c]->key=NULL;
                        hashTable->bucket[hashValue]->pair[c]->value=NULL;
                        free(scan->pair[c]); //delete last element

                    }

                    c++; //increment counter

                }

                printf("Deletion of key: \"%s\" was successful in middle.\n",key);
                stop=1; //stop scanning for match

            }else if(strcmp(scan->pair[c]->key,key)==0 && scan->pair[c+1]==NULL){//if match found at end

                hashTable->bucket[hashValue]->pair[c]->key=NULL;
                hashTable->bucket[hashValue]->pair[c]->value=NULL;
                free(scan->pair[c]); //delete last element

                printf("Deletion of key: \"%s\" was successful.\n",key);
                stop=1; //stop scanning for match

            }else { //if match not found
                c++; //increment c
            }

        }

    }


}

//method that looks up key and checks if it exists in the hashTable, returns 1 when exists
int checkExists(HashTable * hashTable, const char * key){

    int hashValue = hashFunction_S(key); //get hashValue

    if(hashTable->bucket[hashValue]==NULL){//checks if bucket of hashValue exists
        printf("Error: the key's hashValue does not own any existing bucket.\n");
        return 0;
    }else{ //goes through bucket

        int c = 0; //collision index counter
        Pair *scan = hashTable->bucket[hashValue]->pair[c];

        while(scan!=NULL){ //loops to last node in the bucket (which can be the first pair)

            if(strcmp(scan->key,key)==0){

                return 1; //exists

            }

            c++; //increment c
            scan = hashTable->bucket[hashValue]->pair[c]; //goes to next column

        }

        //at this point it has reached the end of the bucket without finding any match
        printf("Error: the key's hashValue does not own an existing list.\n");
        return 0;

    }

}

//method that saves hashTable to structured text file
int saveHashTableAs(HashTable * hashTable,const char * fileName){

    FILE *f;
    f = fopen(fileName,"w"); //opens new file to write to
    if(f!=NULL) {

        //loop by row
```

```
        for (int i = 0; i < HASH_SPACE; i++) {

            if(hashTable->bucket[i]!=NULL) { //if bucket exists

                Bucket *scan = hashTable->bucket[i];
                int c=0; //collision counter index initialised

                //navigate to the end of the bucket (can be the head)
                while (scan->pair[c]!=NULL) {

                    fprintf(f, "%s,%s", scan->pair[c]->key, scan->pair[c]->value);
                    fprintf(f, "\t"); //new column

                    c++; //increment c
                }
            }

            fprintf(f,"\n"); //new row
        }


        printf("Hash table saved successfully to \"%s\"\n", fileName);
    }
}

//method that loads hashTable setting it to the current hashTable
int loadHashTableTo(const char *fileName,HashTable * hashTable){

    char ch; //temporarily stores character from file
    char string[1000]=""; //initialized string to temporarily hold text for scanning to max 1000 characters

    char word[30] = {'\0'};

    char key[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30
    char value[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30

    int i=0; //counts characters for string
    int j = 0; //counts characters for word

    FILE *f;
    f = fopen(fileName,"r"); //opens new file to write to

    //Reading input text file using file pointer and methods from <string.h>
    if(f != NULL)
    {
        //while read character isn't at the end of file, loop
        //scanning every character until the end of file
        while((ch = (char)fgetc(f)) != EOF) {
            string[i]=ch;
            i++;
        }
    }
    else //exception of retrieval failure
    {
        printf("Error: %s was not found.\n", fileName);
        return 1;
    }

    fclose(f); //fclose(f) to avoid memory leak

    i=0; //refreshing counter to be re-used in another while loop

    //scans string for words till the end, which are then sorted into keys and values
    while(i<strlen(string)){

        //nested while loop builds up word
        while(string[i]!=','&&string[i]!='\t'&&string[i]!='\n'){
            word[j]=string[i];
            j++; //increment char counter for word
            i++; //increment char counter for string
        }

        if(string[i]==','){
            for(int n=0;n<strlen(word);n++)
                key[n]=word[n];

        }else if(string[i]=='\t'){ //when after TAB it goes to next column or row
            //printf("value:%s",word);
```

```
            //printf("\tcolumn:%d\n",c);
            for(int n=0;n<strlen(word);n++)
                value[n]=word[n];

            if(strcmp(key,"(null)")!=0 && strcmp(value,"(null)")!=0) //only inserts when not null
                insertPair(hashTable,key,value);

            memset(key,0,strlen(key)); //refresh current key for next key
            memset(value,0,strlen(value)); //refresh current value for next value

        }

        memset(word,0,strlen(word)); //refresh current word for next word
        j = 0; //refresh char counter for word
        i++; //increment char counter for string
    }

    printf("\"%s\" loaded successfully to the current hashTable\n",fileName);

}

//method that frees the hashTable since memory was allocated to it when creating
int freeHashTable(HashTable * hashTable){

    for(int i=0;i<HASH_SPACE;i++) { //frees every bucket if exists
        if(hashTable->bucket[i]!=NULL)
            free(hashTable->bucket[i]);
    }

    free(hashTable); //finally, frees hashTable

    printf("Current hashTable has been successfully freed.\n");

}
```

Task 2b – Dynamic 2D Array Version

In this task, a program was required to be able to insert, delete, lookup, save and load a hash table, of which was a dynamic 2D array version. The hash space was to be fully parametrizable and the maximum collisions were to be dependent on the memory of the computer.

Structs Defined

- Pair
  The struct Pair stores each pair's corresponding key and value. Each key and value is stored as a string literal using constant character pointers.

- Bucket
  The struct bucket stores each row in the hash table. It points to the first pair in each row, or any pair after it. Stores the maximum collisions for each bucket.

- HashTable
  The struct hash table stores the hash space and points to every row to access the data.

Constants Defined

- INITIAL_MAX_C 3
  Constant INITIAL_MAX_C is defined as 3 in the header file "hashTable.h". It is used to initialize the maximum collision for the struct Bucket. When exceeded, instead of out printing an error like in the previous version, it grows the maximum collisions in the struct Bucket dynamically by one, making space. It makes use of the memory allocation functions to achieve this.

Functions Declared

- HashTable * createHashTable_D(unsinged int hashSpace)
  This method is used for creating dynamic hash tables, using hash space as an argument rather than pre-fixed like in the previous static version. It allocates memory for the hash space and hash table that points to each bucket.

- **int** hashFunction_D(**HashTable * hashTable, const char * keyIn**)
  This method is used for getting the hashValue for dynamic hash tables. Hash space is not pre-defined as a constant like for static, so it is retrieved from the hash table itself. The hash value is calculated by getting the some of the characters' ASCII value and modulating it by the hash space of the hash table.

- **int** insertPair(HashTable * hashTable,**const char** * key, **const char** * value)
  This method inserts a pair into the hash table created. The hash value of the key inputted

is first calculated, then it is inserted into the previously-existing bucket or newly-created bucket. If previously-existing then collision is checked and if collision is detected it is placed after the last pair, provided it does not exceed the maximum collisions. If it does then more memory is allocated to dynamically grow the bucket by one element.

- **int** deletePair(HashTable * hashTable, **const char** * key)
  This method deletes a pair from the hash table. It first checks if the bucket of the hashValue of the key exists. If it does not exist an error is printed. If it does it proceeds to scan the bucket for a pair with matching keys to the one requested. If it is found at the head then the whole bucket is deleted provided there are no following elements.

  If it is found at the head and there are more elements, the head is deleted and the following are shifted to the left. If it is found in the middle then the pair is deleted and all the following elements are shifted to the left. If it is found at the end then the element at the end is just freed. Every time an empty space is then found at the end then the bucket deallocates memory by 1 element for efficiency.

- **int** checkExists(HashTable * hashTable, **const char** * key)
  This method looks up a pair in the hash table and checks if it exists or not. It returns 1 when it exists and 0 when it does not. It checks every column of pair until the end for a match with the inputted key.

- **int** saveHashTableAs(HashTable * hashTable,**const char** * fileName)
  This method saves the hash table as a structured text file. Two loops are used for printing the rows and columns to the text file. A "," is used in between keys and values, A "\t" is used in between different pairs and a "\n" is used in between different buckets.

- **int** loadHashTableTo(**const char** *fileName,HashTable * hashTable)
  This method loads the previously saved hashTable and adds every pair from it to the created hash table. It works with scanning the structure for pairs with linked keys and value (by ',') and it then calls insertPair().

- **int** freeHashTable(HashTable * hashTable)
  This method frees the allocated memory of the created hash table. It uses a for loop to scan for existing buckets and free them if existing. After which it frees the hash table itself that points to the buckets.

Source Code (task_2b.c)

```
// Exercise 2b. Hash Tables - Dynamic 2D Array version
// Created by Russell Sammut-Bonnici on 23/12/2017.
// CPS1011

#include "hashTable.h"

//creating a struct Pair to store HashTable's values and keys
typedef struct pair{
    const char * key; //the key to be inputted
    const char * value; //the value to be inputted
```

```
}Pair;

//creating struct for linked list including its head
typedef struct bucket{
    Pair ** pair; //pointer to data that is in pair
    int max_c; //max collision aka no. of columns  (grows dynamically)
}Bucket;

//creating struct for hashTable including pointer to row and hashSpace size
typedef struct hashTable{
    Bucket ** bucket; //double pointer to the beginning of the row
    int hashSpace; //size of the hashTable
}HashTable;

//initializes hashTable and allocates requested space from hashSpace
HashTable * createHashTable_D(unsigned int hashSpace){

    //allocating space for hashTable which is just 1 item
    HashTable * hashTable = calloc(1,sizeof(hashTable));

    //allocating space for number of rows, which are equal to the hashSpace
    hashTable->bucket= calloc(hashSpace, sizeof(Bucket));
    hashTable->hashSpace = hashSpace; //store hashSpace in hashTable

    return hashTable; //returns created hash Table
}

//method that inserts pair into created hashTable
int insertPair(HashTable * hashTable,const char * key, const char * value){

    int hashValue = hashFunction_D(hashTable,key); //calculate hashValue

    if(hashTable->bucket[hashValue]==NULL){//if bucket doesnt exist

        hashTable->bucket[hashValue] = malloc(sizeof(Bucket)); //allocate mem for new bucket

        if (hashTable->bucket[hashValue] == NULL) { //error check
            printf("Failed to allocate memory for new bucket at hashValue: %d.",hashValue);
            return 1;//fail
        }

        hashTable->bucket[hashValue]->max_c = INITIAL_MAX_C; //max collisions set initially to 3
        hashTable->bucket[hashValue]->pair = calloc(INITIAL_MAX_C, sizeof(Pair)); //allocate mem for three
pairs in bucket

        if (hashTable->bucket[hashValue]->pair == NULL) { //error check
            printf("Failed to allocate memory for array of pairs at hashValue: %d.",hashValue);
            return 1;//fail
        }

        hashTable->bucket[hashValue]->pair[0] = malloc(sizeof(Pair)); //allocate mem for first pair
        hashTable->bucket[hashValue]->pair[0]->key = key; //set key of first pair
        hashTable->bucket[hashValue]->pair[0]->value = value; //set value of first pair

    }else if(hashTable->bucket[hashValue]!=NULL){ //if bucket exists

        int c=0; //collision counter to count amount of collisions in bucket when inserting

        while(hashTable->bucket[hashValue]->pair[c]!=NULL){ //while pair exists
            c++; //increment collision counter
        }

        if(c<INITIAL_MAX_C){ //if c is less than max initial collisions
            hashTable->bucket[hashValue]->pair[c] = malloc(sizeof(Pair)); //allocate mem for pair
            hashTable->bucket[hashValue]->pair[c]->key = key; //set key after last collision
            hashTable->bucket[hashValue]->pair[c]->value = value; //set value after last collision

        }else{ //if c is equal to the initial max collisions, no space for new pair, bucket needs to
dynamically grow

            //reallocate to add one new column to the bucket for the new pair
            hashTable->bucket[hashValue] = realloc(hashTable->bucket[hashValue],sizeof(Bucket)*hashTable-
>bucket[hashValue]->max_c+1);
            hashTable->bucket[hashValue]->max_c++; //max collision grows by 1
            hashTable->bucket[hashValue]->pair[c] = malloc(sizeof(Pair)); //allocate mem for pair
            hashTable->bucket[hashValue]->pair[c]->key = key; //set key at newly added column
            hashTable->bucket[hashValue]->pair[c]->value = value; //set value at newly added column
        }
```

```
    }

    printf("Pair of key: \"%s\" and value: \"%s\" was successfully inserted into the hashTable, hashValue:
%d.\n",key,value,hashValue);

}

//method for hashFunction_D, accepts key and value as arguments
int hashFunction_D(HashTable * hashTable, const char * keyIn){

    int charSumKey = 0; //sum of the ASCII code of characters in key
    int x; //variable to be returned
    int size = hashTable->hashSpace; //sets size to hashSpace of table

    //accumulates sum of ASCII code of each character
    for(int i=0;i<strlen(keyIn);i++) {
        charSumKey += keyIn[i];
    }

    x = charSumKey % size; //invented formula for hash function
    return x;
}

int deletePair(HashTable * hashTable, const char * key){

    int hashValue = hashFunction_D(hashTable,key); //get hashValue

    if(checkExists(hashTable,key)==1){ //if exists, delete pair

        int c = 0; //counter for collision index
        int stop = 0; //to break from loop from if statement
        Bucket *scan = hashTable->bucket[hashValue];

        while(scan!=NULL && stop!=1){ //scans through list until match to delete is found

            if(strcmp(hashTable->bucket[hashValue]->pair[0]->key,key)==0){ //if match found at head

                if(hashTable->bucket[hashValue]->pair[1]==NULL){ //if head to delete is the only node in the
bucket delete head then bucket

                    hashTable->bucket[hashValue]->pair[0]->key=NULL;
                    hashTable->bucket[hashValue]->pair[0]->value=NULL;
                    free(hashTable->bucket[hashValue]->pair[0]);
                    free(hashTable->bucket[hashValue]->pair);
                    free(hashTable->bucket[hashValue]);

                }else if(hashTable->bucket[hashValue]->pair[1]!=NULL){ //if there is more than just one node
in the list, shift rest to left then delete last

                    while(scan->pair[c]!=NULL) { //while current node exists

                        if (scan->pair[c + 1] != NULL) { //if next node exists

                            scan->pair[c]->key = scan->pair[c + 1]->key; //copy, from next node to key
                            scan->pair[c]->value = scan->pair[c + 1]->value; //copy, from next node to value

                        }else{ //if next node doesn't exist (at end of list)

                            hashTable->bucket[hashValue]->pair[c]->key=NULL;
                            hashTable->bucket[hashValue]->pair[c]->value=NULL;
                            free(scan->pair[c]); //delete last element

                            //reallocate to shorten bucket
                            hashTable->bucket[hashValue] = realloc(hashTable-
>bucket[hashValue],sizeof(Bucket)*hashTable->bucket[hashValue]->max_c-1);
                            hashTable->bucket[hashValue]->max_c--; //max collision shrinks by 1

                        }

                        c++; //increment counter

                    }
                }

                printf("Deletion of key: \"%s\" was successful at head.\n",key);
                stop=1; //stop scanning for match
```

```
            }else if(strcmp(scan->pair[c]->key,key)==0 && scan->pair[c+1]!=NULL){//if match found in middle,
shift everything after it to the left

                while(scan->pair[c]!=NULL) { //while current node exists

                    if (scan->pair[c + 1] != NULL) { //if next node exists

                        scan->pair[c]->key = scan->pair[c + 1]->key; //copy, from next node to key
                        scan->pair[c]->value = scan->pair[c + 1]->value; //copy, from next node to value

                    }else{ //if next node doesn't exist (at end of list)

                        hashTable->bucket[hashValue]->pair[c]->key=NULL;
                        hashTable->bucket[hashValue]->pair[c]->value=NULL;
                        free(scan->pair[c]); //delete last element

                        //reallocate to shorten bucket
                        hashTable->bucket[hashValue] = realloc(hashTable-
>bucket[hashValue],sizeof(Bucket)*hashTable->bucket[hashValue]->max_c-1);
                        hashTable->bucket[hashValue]->max_c--; //max collision shrinks by 1

                    }

                    c++; //increment counter

                }

                printf("Deletion of key: \"%s\" was successful in middle.\n",key);
                stop=1; //stop scanning for match

            }else if(strcmp(scan->pair[c]->key,key)==0 && scan->pair[c+1]==NULL){//if match found at end

                hashTable->bucket[hashValue]->pair[c]->key=NULL;
                hashTable->bucket[hashValue]->pair[c]->value=NULL;
                free(scan->pair[c]); //delete last element

                //reallocate to shorten bucket
                hashTable->bucket[hashValue] = realloc(hashTable->bucket[hashValue],sizeof(Bucket)*hashTable-
>bucket[hashValue]->max_c-1);
                hashTable->bucket[hashValue]->max_c--; //max collision shrinks by 1

                printf("Deletion of key: \"%s\" was successful.\n",key);
                stop=1; //stop scanning for match

            }else { //if match not found
                c++; //increment c
            }

        }

    }


}

//method that looks up key and checks if it exists in the hashTable, returns 1 when exists
int checkExists(HashTable * hashTable, const char * key){

    int hashValue = hashFunction_D(hashTable,key); //get hashValue

    if(hashTable->bucket[hashValue]==NULL){//checks if bucket of hashValue exists
        printf("Error: the key's hashValue does not own any existing bucket.\n");
        return 0;
    }else{ //goes through bucket

        int c = 0; //collision index counter
        Pair *scan = hashTable->bucket[hashValue]->pair[c];

        while(scan!=NULL){ //loops to last node in the bucket (which can be the first pair)

            if(strcmp(scan->key,key)==0){

                return 1; //exists

            }

            c++; //increment c
            scan = hashTable->bucket[hashValue]->pair[c]; //goes to next column
```

```
        }

        //at this point it has reached the end of the bucket without finding any match
        printf("Error: the key's hashValue does not own an existing list.\n");
        return 0;

    }

}

//method that saves hashTable to structured text file
int saveHashTableAs(HashTable * hashTable,const char * fileName){

    FILE *f;
    f = fopen(fileName,"w"); //opens new file to write to
    if(f!=NULL) {

        //loop by row
        for (int i = 0; i < hashTable->hashSpace; i++) {

            if(hashTable->bucket[i]!=NULL) { //if bucket exists

                Bucket *scan = hashTable->bucket[i];
                int c=0; //collision counter index initialised

                //navigate to the end of the bucket (can be the head)
                while (scan->pair[c]!=NULL) {

                    fprintf(f, "%s,%s", scan->pair[c]->key, scan->pair[c]->value);
                    fprintf(f, "\t"); //new column

                    c++; //increment c
                }
            }

            fprintf(f,"\n"); //new row
        }


        printf("Hash table saved successfully to \"%s\"\n", fileName);
    }
}

//method that loads hashTable setting it to the current hashTable
int loadHashTableTo(const char *fileName,HashTable * hashTable){

    char ch; //temporarily stores character from file
    char string[1000]=""; //initialized string to temporarily hold text for scanning to max 1000 characters

    char word[30] = {'\0'};

    char key[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30
    char value[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30

    int i=0; //counts characters for string
    int j = 0; //counts characters for word

    FILE *f;
    f = fopen(fileName,"r"); //opens new file to write to

    //Reading input text file using file pointer and methods from <string.h>
    if(f != NULL)
    {
        //while read character isn't at the end of file, loop
        //scanning every character until the end of file
        while((ch = (char)fgetc(f)) != EOF) {
            string[i]=ch;
            i++;
        }
    }
    else //exception of retrieval failure
    {
        printf("Error: %s was not found.\n", fileName);
        return 1;
    }

    fclose(f); //fclose(f) to avoid memory leak
```

```
    i=0; //refreshing counter to be re-used in another while loop

    //scans string for words till the end, which are then sorted into keys and values
    while(i<strlen(string)){

        //nested while loop builds up word
        while(string[i]!=','&&string[i]!='\t'&&string[i]!='\n'){
            word[j]=string[i];
            j++; //increment char counter for word
            i++; //increment char counter for string
        }

        if(string[i]==','){
            for(int n=0;n<strlen(word);n++)
                key[n]=word[n];

        }else if(string[i]=='\t'){ //when after TAB it goes to next column or row
            //printf("value:%s",word);
            //printf("\tcolumn:%d\n",c);
            for(int n=0;n<strlen(word);n++)
                value[n]=word[n];

            if(strcmp(key,"(null)")!=0 && strcmp(value,"(null)")!=0) //only inserts when not null
                insertPair(hashTable,key,value);

            memset(key,0,strlen(key)); //refresh current key for next key
            memset(value,0,strlen(value)); //refresh current value for next value

        }

        memset(word,0,strlen(word)); //refresh current word for next word
        j = 0; //refresh char counter for word
        i++; //increment char counter for string
    }

    printf("\"%s\" loaded successfully to the current hashTable\n",fileName);

}

//method that frees the hashTable since memory was allocated to it when creating
int freeHashTable(HashTable * hashTable){

    for(int i=0;i<hashTable->hashSpace;i++) { //frees every bucket if exists
        if(hashTable->bucket[i]!=NULL)
            free(hashTable->bucket[i]);
    }

    free(hashTable); //finally, frees hashTable

    printf("Current hashTable has been successfully freed.\n");

}
```

Task 2c – Linked List Version

In this task, a program was required to be able to insert, delete, lookup, save and load a hash table, of which was a linked list version.

Structs Defined

- Pair
  The struct Pair stores each pair's corresponding key and value. Each key and value is stored as a string literal using constant character pointers.

- Node
  The struct Node stores the data pointer and the next pointer. The data pointer points to the pair, whereas the next pointer points to the next node in the linked list.

- List
  The struct bucket stores each list/row in the hash table. It points to the first node of the list that is the head.

- HashTable
  The struct hash table stores the hash space and points to every list to access the data.

Functions Declared

- HashTable * createHashTable_D(unsinged int hashSpace)
  This method is used for creating dynamic hash tables, using hash space as an argument rather than pre-fixed like in the previous version. It allocates memory for the hash space and hash table that points to each bucket.

- **int** hashFunction_D(**HashTable * hashTable, const char * keyIn**)
  This method is used for getting the hashValue for dynamic hash tables. Hash space is not pre-defined as a constant like for static, so it is retrieved from the hash table itself. The hash value is calculated by getting the some of the characters' ASCII value and modulating it by the hash space of the hash table.

- **int** insertPair(HashTable * hashTable,**const char** * key, **const char** * value)
  This method inserts a pair into the hash table created. The hash value of the key inputted is first calculated, then it is inserted into the previously-existing bucket or newly-created bucket. If previously-existing then collision is checked and if collision is detected it is placed after the last pair.

- **int** deletePair(HashTable * hashTable, **const char** * key)
  This method deletes a node from the hash table. It first checks if the bucket of the hashValue of the key exists. If it does not exist an error is printed.

If it does it proceeds to scan the bucket for a pair with matching keys to the one requested. If it is found at the head then the whole bucket is deleted provided there are no following elements. If it is found at the head and there are more elements, the head is deleted and the head is set as the node that was previously after it. If it is found in the middle then the next pointer of the node before it is set to point to the node after it and the node is freed. If it is found at the end then the element at the end is just freed and the next pointer of the one previous to it is set to NULL.

- **int** checkExists(HashTable * hashTable, **const char** * key)
  This method looks up a pair in the hash table and checks if it exists or not. It returns 1 when it exists and 0 when it does not. It makes use of the pointer variable scan and a for loop to traverse through the linked list. It checks each pair in each node of the bucket until next is found to be null.

- **int** saveHashTableAs(HashTable * hashTable,**const char** * fileName)
  This method saves the hash table as a structured text file. Two loops are used for printing the rows and columns to the text file. A "," is used in between keys and values, A "\t" is used in between different pairs and a "\n" is used in between different buckets.

- **int** loadHashTableTo(**const char** *fileName,HashTable * hashTable)
  This method loads the previously saved hashTable and adds every pair from it to the created hash table. It works with scanning the structure for pairs with linked keys and value (by ',') and it then calls insertPair().

- **int** freeHashTable(HashTable * hashTable)
  This method frees the allocated memory of the created hash table. It uses a for loop to scan for existing buckets and free them if existing. After which it frees the hash table itself that points to the buckets.

Source Code (task_2c.c)

```
// Exercise 2c. Hash Tables - linked list version
// Created by Russell Sammut-Bonnici on 20/12/2017.
// CPS1011

#include "hashTable.h"

//creating a struct Pair to store HashTable's values and keys
typedef struct pair{
    const char * key; //the key to be inputted
    const char * value; //the value to be inputted
}Pair;

//creating a struct for elements and their pointers inside singly linked lists
typedef struct node{
    Pair * data; //points to node's key and value
    struct node * next; //points to next node
}Node;

//creating struct for linked list including its head
typedef struct list{
    Node * head; //pointer to the head of the list
}List;

//creating struct for hashTable including pointer to head and hashSpace size
```

```c
typedef struct hashTable{
    List **list; //double pointer to the beginning of the linked list
    int hashSpace; //size of the hashTable
}HashTable;

//initializes hashTable and allocates requested space from hashSpace
HashTable * createHashTable_D(unsigned int hashSpace){

    //allocating space for hashTable which is just 1 item
    HashTable * hashTable = calloc(1,sizeof(hashTable));

    //allocating space for number of lists, which are equal to the hashSpace
    hashTable->list = calloc(hashSpace, sizeof(Node));
    hashTable->hashSpace = hashSpace; //store hashSpace in hashTable

    return hashTable; //returns created hash Table
}


//method creates node that holds pair and adds to the list at head or tail
int insertPair(HashTable *hashTable, const char *key, const char *value) {

    int hashValue = hashFunction_D(hashTable, key); //hashValue for current insertion

    //If there is no list cause of no head, add head
    if (!hashTable->list[hashValue]) {

        hashTable->list[hashValue] = malloc(sizeof(List)); //allocate mem for new list

        if (hashTable->list[hashValue] == NULL) { //error check
            printf("Failed to allocate memory for new list at hashValue: %d.",hashValue);
            return 1;//fail
        }

        hashTable->list[hashValue]->head = malloc(sizeof(Node)); //allocate mem for new head

        if (hashTable->list[hashValue]->head->next == NULL) { //error check
            printf("Failed to allocate memory for new node at head.");
            return 1;//fail
        }

        hashTable->list[hashValue]->head->data = malloc(sizeof(Pair)); //allocate mem for new pair

        if (hashTable->list[hashValue]->head->data == NULL) { //error check
            printf("Failed to allocate memory for new pair at head.");
            return 1;//fail
        }

        hashTable->list[hashValue]->head->data->key = key; //set head key
        hashTable->list[hashValue]->head->data->value = value; //set head value
        hashTable->list[hashValue]->head->next = NULL; //set next of head to null for now

    } else { //if at tail aka when collision is detected

        Node *scan;//pointer scan used for navigating through the list
        scan = hashTable->list[hashValue]->head; //start scanning from head

        //navigate to the node at the end of the linked list (even at head)
        while(scan->next!=NULL){
            scan = scan->next; //point to next
        }

        //now we add new node to the end of the list
        scan->next = malloc(sizeof(Node)); //allocates memory for new node

        if (scan->next == NULL) { //error check
            printf("Failed to allocate memory for new node at tail.");
            return 1;//fail
        }

        scan->next->data = malloc(sizeof(Pair)); //allocate mem for new pair

        if (scan->next->data == NULL) { //error check
            printf("Failed to allocate memory for new pair at tail.");
            return 1;//fail
        }

        scan->next->data->key = key; //sets data key of new node
```

```
        scan->next->data->value = value; //sets data value of new node
        scan->next->next = NULL; //since last node, next node after it is set to null

    }

    printf("Pair of key: \"%s\" and value: \"%s\" was successfully inserted into the hashTable, hashValue:
%d.\n",key,value,hashValue);
}

//method deletes pair held in linked list by key
int deletePair(HashTable * hashTable, const char * key){

    int r = hashFunction_D(hashTable,key); //keeps track of hashValue aka list number for pair deletion
    Node *scan = hashTable->list[r]->head; //set currNode to scan list, checking for match
    int stop = 0; //variable for stopping loop when set to 1 - when pair is deleted

    if(checkExists(hashTable,key)==1) { //deletes if requested key exists

        //navigate to the node at the end of the linked list checking for match and stop when match is found
(even at head)
        while(scan!=NULL && stop!=1){

            if(strcmp(hashTable->list[r]->head->data->key,key)==0){ //delete at head

                if(hashTable->list[r]->head->next!=NULL){ //if list has more than head, need set head to node
after it

                hashTable->list[r]->head = hashTable->list[r]->head->next; //remove head by overwriting it
with next node

                }else { //if list just contains head, delete head then list
                    free(hashTable->list[r]->head); //deletes head where match is found
                    free(hashTable->list[r]); //free list

                }

                printf("Deletion of key: \"%s\" was successful at head.\n",key);

                stop=1; //exits loop

            }else if(strcmp(scan->next->data->key,key)==0&&scan->next->next!=NULL){ //delete in the middle

                scan->next = scan->next->next; //skips over deleted node
                free(scan->next); //deletes element where match is found

                printf("Deletion of key: \"%s\" was successful in middle.\n",key);

                stop=1; //exits loop

            }else if(strcmp(scan->next->data->key,key)==0&&scan->next->next==NULL){ //delete at tail

                scan->next = NULL; //sets pointer to deleted node to NULL
                free(scan->next); //deletes tail where match is found

                printf("Deletion of key: \"%s\" was successful at tail.\n",key);

                stop=1; //exits loop
            }

            if(stop!=1)
                scan = scan->next; //point to next
        }

    }

}

//method that goes through link checking if pair of entered key exists. returns 1 when exists
int checkExists(HashTable *hashTable,const char *key){

    int hashValue = hashFunction_D(hashTable,key);

    if(hashTable->list[hashValue]->head==NULL){//checks if list of hashValue exists
        printf("Error: the key's hashValue does not own any existing list.");
        return 0;
    }else{ //goes through list
        Node *scan = hashTable->list[hashValue]->head;
```

```
        while(scan!=NULL){ //loops to last node in the list (which can be the head)

            if(strcmp(scan->data->key,key)==0){

                return 1; //exists

            }

            scan = scan->next; //point to next

        }

        //at this point it has reached the end of the list without finding any match
        printf("Error: the key's hashValue does not own an existing list.");
        return 0;

    }

}

//method for hashFunction_D, accepts key and value as arguments
int hashFunction_D(HashTable * hashTable, const char * keyIn){

    int charSumKey = 0; //sum of the ASCII code of characters in key
    int x; //variable to be returned
    int size = hashTable->hashSpace; //sets size to hashSpace of table

    //accumulates sum of ASCII code of each character
    for(int i=0;i<strlen(keyIn);i++) {
        charSumKey += keyIn[i];
    }

    x = charSumKey % size; //invented formula for hash function
    return x;
}

int saveHashTableAs(HashTable * hashTable,const char * fileName){

    Node *scan;//pointer scan used for navigating through the list

    FILE *f;
    f = fopen(fileName,"w"); //opens new file to write to
    if(f!=NULL) {

        //loop by row
        for (int i = 0; i < hashTable->hashSpace; i++) {

            if(hashTable->list[i]!=NULL) { //if list exists

                scan = hashTable->list[i]->head; //start scanning from head

                //navigate to the end of the linked list (can be the head)
                while (scan!=NULL) {

                    fprintf(f, "%s,%s", scan->data->key, scan->data->value);
                    fprintf(f, "\t"); //new column
                    scan = scan->next; //point to next

                }
            }

            fprintf(f,"\n"); //new row
        }


        printf("Hash table saved successfully to \"%s\"\n", fileName);
    }
}

//method that loads hashTable setting it to the current hashTable
int loadHashTableTo(const char *fileName,HashTable * hashTable){

    char ch; //temporarily stores character from file
    char string[1000]=""; //initialized string to temporarily hold text for scanning to max 1000 characters

    char word[30] = {'\0'};

    char key[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30
```

```
    char value[30] = {'\0'}; //initialized to temporarily store key, max characters set to 30

    int i=0; //counts characters for string
    int j = 0; //counts characters for word

    FILE *f;
    f = fopen(fileName,"r"); //opens new file to write to

    //Reading input text file using file pointer and methods from <string.h>
    if(f != NULL)
    {
        //while read character isn't at the end of file, loop
        //scanning every character until the end of file
        while((ch = (char)fgetc(f)) != EOF) {
            string[i]=ch;
            i++;
        }
    }
    else //exception of retrieval failure
    {
        printf("Error: %s was not found.\n", fileName);
        return 1;
    }

    fclose(f); //fclose(f) to avoid memory leak

    i=0; //refreshing counter to be re-used in another while loop

    //scans string for words till the end, which are then sorted into keys and values
    while(i<strlen(string)){

        //nested while loop builds up word
        while(string[i]!=','&&string[i]!='\t'&&string[i]!='\n'){
            word[j]=string[i];
            j++; //increment char counter for word
            i++; //increment char counter for string
        }

        if(string[i]==','){
            for(int n=0;n<strlen(word);n++)
                key[n]=word[n];

        }else if(string[i]=='\t'){ //when after TAB it goes to next column or row
            //printf("value:%s",word);
            //printf("\tcolumn:%d\n",c);
            for(int n=0;n<strlen(word);n++)
                value[n]=word[n];

            if(strcmp(key,"(null)")!=0 && strcmp(value,"(null)")!=0) //only inserts when not null
                insertPair(hashTable,key,value);

            memset(key,0,strlen(key)); //refresh current key for next key
            memset(value,0,strlen(value)); //refresh current value for next value

        }

        memset(word,0,strlen(word)); //refresh current word for next word
        j = 0; //refresh char counter for word
        i++; //increment char counter for string
    }

    printf("\"%s\" loaded successfully to the current hashTable\n",fileName);

}

//method that frees the hashTable since memory was allocated to it when creating
int freeHashTable(HashTable * hashTable){

    for(int i=0;i<hashTable->hashSpace;i++) { //frees every list if exists
        if(hashTable->list[i]!=NULL)
            free(hashTable->list[i]);
    }

    free(hashTable); //finally, frees hashTable

    printf("Current hashTable has been successfully freed.\n");

}
```

Task 2d – Test Driver

In this task, a program was required to act as a test driver for all the previous hash table versions. A header file was meant to have all the function, constant and struct declarations of the previous versions. By including the header file in the test driver all the methods are then able to be called to test each version.

Constants Defined

- INPUT_AMOUNT 20
  Constant INPUT_AMOUNT defined as 20 for the number of key-value pair inputs inserted in to a newly created hash table for testing. This constant is used in the for loop when inserting into the table, and when declaring the string arrays storing keys and values.

Variables Defined

- HashTable * hashTable1
  In this case HashTable * hashTable1 is created to test the first version of hash tables. The method createHashTable_S() is called rather than createHashTable_D() as the static version is being tested as opposed to the other two dynamic versions.

  Each version has been tested already and each file has been saved as hashTable1, hashTable2 and hashTable3 for the first, second and third version accordingly. The text file names match the HashTable pointer variable names.

- **char** *key1[INPUT_AMOUNT]
  The string array key1 stores ID card numbers to be inputted as keys into the hash table. This is used for testing purposes.

- **char** *value1[INPUT_AMOUNT]
  The string array value1 stores mobile numbers to be inputted as values into the hash table. It corresponds to the keys in the previous string array, and when put into the hash table they are inserted together, forming a pair. This array instance is also used for testing purposes.

Header File (hashTable.h)

```
// Exercise 2d. Hash Tables - Library linking and API
// Created by Russell Sammut-Bonnici on 25/12/2017.
// CPS1011

#ifndef HASHTABLE_H
#define HASHTABLE_H

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

```
#include <stdlib.h>

/* constant declaration */
#define INPUT_AMOUNT 20 //amount of pairs to be inputted to test the hashMap
#define HASH_SPACE 10 //pre-fixed hash_space defined as 10
#define MAX_COLL 6 //pre-fixed max_collisions defined as 6
#define INITIAL_MAX_C 3 //maximum collisions initially set to 3 (then grows dynamically)

/* struct declaration*/
typedef struct pair Pair; //creating struct for storing key-value pair
typedef struct bucket Bucket; //creating struct for linked list including its head
typedef struct hashTable HashTable; //creating struct for hashTable including pointer to row and hashSpace
size
typedef struct node Node; //creating a struct for elements and their pointers inside singly linked lists
typedef struct list List; //creating struct for linked list including its head

/* function declaration for hashTable*/
HashTable * createHashTable_S(); //used for making static hashTable, using pre-defined constants
HashTable * createHashTable_D(unsigned int hashSpace); //used for making dynamic hashTable, initialising
pointer hashTable by hashSpace as a parameter
int hashFunction_S(const char * keyIn); //used for returning hash value with pre-defined HASH_SPACE in static
2D array version
int hashFunction_D(HashTable * hashTable, const char * keyIn); //returns calculated hash value with relation
to hash space
int insertPair(HashTable * hashTable,const char * key, const char * value); //inserts a pair to dynamic
hashTable
int deletePair(HashTable * hashTable, const char * key); //deletes a pair from dynamic hashTable
int checkExists(HashTable * hashTable, const char * key); //looks up to see if an inputted key pair exists in
dynamic hashTable
int saveHashTableAs(HashTable * hashTable,const char * fileName); //saves dynamic hashTable
int loadHashTableTo(const char *fileName,HashTable * hashTable); //loads dynamic hashTable
int freeHashTable(HashTable * hashTable); //frees space allocated during creation

#endif //HASHTABLE_H
```

Source Code (task_2d.c)

```
// Exercise 2d. Hash Tables - Test Driver
// Created by Russell Sammut-Bonnici on 25/12/2017.
// CPS1011

#include "hashTable.h"

//main method
int main(){

    //create and initialize hashTable1 with the pre-defined hash space 10
    HashTable * hashTable1 = createHashTable_S();

    //initializing array to store string key inputs (in this case IDs)
    char *key1[INPUT_AMOUNT]={"0426298M","0326288M","0134566M","0987654M","0234211M",
                              "0423458M","0234288M","0134098M","0456654M","0214211M",
                              "2345698M","0678348M","0223366M","0999954M","0121211M",
                              "0010108M","0234018M","0014098M","0972654M","0211111M"};

    //initializing array to store string value inputs (in this case Mobile No.s)
    char *value1[INPUT_AMOUNT]={"79835334","99887766","79856342","99223344","79887766",
                                "79567834","99333366","79111112","99113114","79000000",
                                "79835335","79835337","79835336","99222332","79563366",
                                "79874534","95692366","79109422","92376314","79000943"};

    //inserts keys and values into hashTable demonstrating insertion
    for(int i=0;i<INPUT_AMOUNT;i++) //loops for every pair input
        insertPair(hashTable1,key1[i],value1[i]);

    printf("\n");

    //deletes a specified pair in hashTable by scanning through list demonstrating deletion
    deletePair(hashTable1,"0972654M"); //deletes pair (which is "0972654M","92376314")

    printf("\n");

    //checks if key "0326288M" exists, returns 1 since it exists, demonstrating look up
    printf("Look up of key: \"%s\". Return value: %d\n","0326288M",checkExists(hashTable1,"0326288M"));
```

```
    printf("\n");

    //saves hashTable to disk as fileName
    saveHashTableAs(hashTable1,"hashTable1.dat");

    printf("\n");

    //loads "hashTable2" from disk by fileName and adds to hashTable
    loadHashTableTo("hashTable2.dat",hashTable1);

    printf("\n");

    //frees data in hashTable
    freeHashTable(hashTable1);

    return 0;
}
```

Output Listing for testing version 2a.c

```
C:\Users\rsamm\CLionProjects\assignment\question_2\cmake-build-debug\2a_test.exe

Pair of key: "0426298M" and value: "79835334" was successfully inserted into the hashTable, hashValue: 4.

Pair of key: "0326288M" and value: "99887766" was successfully inserted into the hashTable, hashValue: 2.

Pair of key: "0134566M" and value: "79856342" was successfully inserted into the hashTable, hashValue: 8.

Pair of key: "0987654M" and value: "99223344" was successfully inserted into the hashTable, hashValue: 2.

Pair of key: "0234211M" and value: "79887766" was successfully inserted into the hashTable, hashValue: 6.

Pair of key: "0423458M" and value: "79567834" was successfully inserted into the hashTable, hashValue: 9.

Pair of key: "0234288M" and value: "99333366" was successfully inserted into the hashTable, hashValue: 0.

Pair of key: "0134098M" and value: "79111112" was successfully inserted into the hashTable, hashValue: 8.

Pair of key: "0456654M" and value: "99113114" was successfully inserted into the hashTable, hashValue: 3.

Pair of key: "0214211M" and value: "79000000" was successfully inserted into the hashTable, hashValue: 4.

Pair of key: "2345698M" and value: "79835335" was successfully inserted into the hashTable, hashValue: 0.

Pair of key: "0678348M" and value: "79835337" was successfully inserted into the hashTable, hashValue: 9.

Pair of key: "0223366M" and value: "79835336" was successfully inserted into the hashTable, hashValue: 5.

Pair of key: "0999954M" and value: "99222332" was successfully inserted into the hashTable, hashValue: 8.

Pair of key: "0121211M" and value: "79563366" was successfully inserted into the hashTable, hashValue: 1.

Pair of key: "0010108M" and value: "79874534" was successfully inserted into the hashTable, hashValue: 3.

Pair of key: "0234018M" and value: "95692366" was successfully inserted into the hashTable, hashValue: 1.

Pair of key: "0014098M" and value: "79109422" was successfully inserted into the hashTable, hashValue: 5.

Pair of key: "0972654M" and value: "92376314" was successfully inserted into the hashTable, hashValue: 6.

Pair of key: "0211111M" and value: "79000943" was successfully inserted into the hashTable, hashValue: 0.


Deletion of key: "0972654M" was successful.
```

```
Look up of key: "0326288M". Return value: 1


Hash table saved successfully to "hashTable1.dat"


Pair of key: "0234288M" and value: "99333366" was successfully inserted into the hashTable, hashValue: 0.
Pair of key: "2345698M" and value: "79835335" was successfully inserted into the hashTable, hashValue: 0.
Pair of key: "0211111M" and value: "79000943" was successfully inserted into the hashTable, hashValue: 0.
Pair of key: "0121211M" and value: "79563366" was successfully inserted into the hashTable, hashValue: 1.
Pair of key: "0234018M" and value: "95692366" was successfully inserted into the hashTable, hashValue: 1.
Pair of key: "0326288M" and value: "99887766" was successfully inserted into the hashTable, hashValue: 2.
Pair of key: "0987654M" and value: "99223344" was successfully inserted into the hashTable, hashValue: 2.
Pair of key: "0456654M" and value: "99113114" was successfully inserted into the hashTable, hashValue: 3.
Pair of key: "0010108M" and value: "79874534" was successfully inserted into the hashTable, hashValue: 3.
Pair of key: "0426298M" and value: "79835334" was successfully inserted into the hashTable, hashValue: 4.
Pair of key: "0214211M" and value: "79000000" was successfully inserted into the hashTable, hashValue: 4.
Pair of key: "0223366M" and value: "79835336" was successfully inserted into the hashTable, hashValue: 5.
Pair of key: "0014098M" and value: "79109422" was successfully inserted into the hashTable, hashValue: 5.
Pair of key: "0234211M" and value: "79887766" was successfully inserted into the hashTable, hashValue: 6.
Pair of key: "0134566M" and value: "79856342" was successfully inserted into the hashTable, hashValue: 8.
Pair of key: "0134098M" and value: "79111112" was successfully inserted into the hashTable, hashValue: 8.
Pair of key: "0999954M" and value: "99222332" was successfully inserted into the hashTable, hashValue: 8.
Pair of key: "0423458M" and value: "79567834" was successfully inserted into the hashTable, hashValue: 9.
Pair of key: "0678348M" and value: "79835337" was successfully inserted into the hashTable, hashValue: 9.
"hashTable2.dat" loaded successfully to the current hashTable


Current hashTable has been successfully freed.


Process finished with exit code 0
```