Programming Principles in C Assignment

2017/2018

Name: Valerija Holomjova
Course Code: CPS1011-SEM1-A-1718

Table of Contents

Question 1a

At the very start of the program a header file was included called '<stdio.h>' which contains various functions for performing input and output, one of which is 'printf' which sends formatted output to the text terminal. A macro called 'BASE' is defined as is set to a value of 200 which is the starting amount for each person.

```
#include <stdio.h>
#define BASE 200 //starting amount for each person
```

In the main function, two float variables 'CI' and 'SI' were delcared and will represent the total compound interest and simple interest earned respectively. The rates of each interest denoted by 'CI_rate' and 'SI_rate' are of type 'const float' as they will remain fixed throughout the code and are assigned decimal point values. The values of the rates were assigned based on the values given in the assignment sheet.

```
int main() {
   float CI, SI;
   const float CI_rate = 0.1; //Joan's interest rate compounded annually
   const float SI_rate = 0.15; //Tom's simple interest rate annually
```

Next, the compound interest and simple interest after one year is calculated. The variable 'year' was initialized as an integer as it will not hold any decimal values. The code display a 'while' loop with a condition which will terminate the loop when the value of compound interest is greater than the value of simple interest. Within the loop are two formulae constructed to calculate the values of both interests per year.

- The compounded interest is calculated by adding on the value of the previous compound interest multiplied by the compound interest rate.
- The simple interest is calculated by adding on a fixed value consisting of the base value multiplied by the simple interest rate.

The year variable is then incremented to keep track of the number of years.

```
int year = 1;
CI = BASE + (BASE*CI_rate); //Joan's CI after one year
SI = BASE + (BASE*SI_rate); //Tom's SI after one year

while(CI < SI) //loop will repeat until Joan's CI exceeds Tom's SI
{
   CI += CI * CI_rate; //calculates CI and adds onto the total of previous CI every loop(year)
   SI += BASE * SI_rate; //calculates SI and adds onto the total of previous SI every loop(year)
   year++; //keeps track of the years
```

After the loop terminates, the number of years taken from the compound interest sum to overtake the simple interest sum is displayed using the 'printf' function. The '%d' placeholder is used to represent the year variable as it is of integer type. Then, the program prints the compound interest sum and the simple interest sum after the same amount of years. At the end of the code is a curly bracket signifying the end of the main function.

```
printf("It takes %d years for Joan's invested sum to overtake Tom's.\n", year);
printf("After %d years Joan's sum is $%.2f and Tom's sum is $%.2f.\n", year,CI,SI);
}
```

Important Remarks

An important fault to note is that the dollar sign has been used instead of the euro sign as the symbol was not printed out correctly despite efforts trying to input the symbol itself and using the format string '\u20AC'. The euro sign would instead be outputted as "Ôé¼".

Below is a figure displaying the expected output and actual output produced after the code is successfully executed.

Actual Output	Expected Output
It takes 9 years for Joan's invested sum to overtake Tom's. After 9 years Joan's sum is $471.59 and Tom's sum is $470.00.	It takes 9 years for Joan's invested sum to overtake Tom's. After 9 years Joan's sum is $471.59 and Tom's sum is $470.00.

Question 1b

At the start of the code a header called '<stdlib.h>' is included which contains various general functions. In this example, the header file was used for the 'EXIT_FAILURE' macro which represents the value of the exit function which is used in cases of failure. Macros representing the price of each vegetable and the discount price were defined. Following the macros, functions were declared so that their contents could be displayed after the main function. The code within each function, including the main function, will now be explained separately.

```
#include <stdio.h>
#include <stdlib.h>
#define P_A 2.05 //price artichokes per kg
#define P_O 1.15 //price onions per kg
#define P_C 1.09 //price carrots per kg
#define DISC .05 //discount

char get_choice(); //function the reads user input for choice
float get_weight(); //function the reads user input for weight
void display(float TW_A, float TW_O, float TW_C); //function that displays a summary of all the items and costs
void get_receipt(float TW_A, float TW_O, float TW_C); //function that saves the summary in a file
```

Main Function

In the main function, a variable of type 'char' is declared called 'choice' which will store a character based on the users input which will then be used in the upcoming 'switch' statement. The total weight of each vegetable is initialized as float variables and are set to 0 to avoid potential bugs. Then, the program prints out a welcoming message using the 'printf' function.

A 'while' loop is created which will terminate when the user inputs the "q" character. Within the while loop condition is a function called 'get_choice' which will display a range of choices to the user and then return a character depending on the user's input to the variable 'choice'. This variable is used to determine the case in the 'switch' statement.

- Case 'a' – will allow the user to add weight onto the total weight of artichokes
- Case 'b' - will allow the user to add weight onto the total weight of onions
- Case 'c' - will allow the user to add weight onto the total weight of carrots

The weight is obtained using the 'get_weight' function and then added onto the total weight of the specific vegetable. The default case executes in cases of incorrect input after the 'get_choice' function terminates and displays an error message. Below is a table representing the expected outcomes and the actual outcomes from the 'switch' statement.

Input	Actual Outcome/Output	Expected Outcome/Output
'a'	"Please enter amount of artichokes in kg."	"Please enter amount of artichokes in kg."
'b'	"Please enter amount of onions in kg."	"Please enter amount of onions in kg."
'c'	"Please enter amount of carrots in kg."	"Please enter amount of carrots in kg."
'q'	Exits 'while' loop	Exits 'while' loop
'n'	"Please input a,b,c or q."	"Please input a,b,c or q."

It is important to note that the program displayed "Please input a,b,c or q" instead of "Program Error." due to the intervention with the 'get_choice' function as it prompts the user for the correct

answer. However, if the user inputs an incorrect input once more despite the warning, the “Program Error” message will display and then the user will be returned to the menu.

```
int main() {
   char choice;
   float TW_A = 0, TW_O = 0, TW_C = 0; //total weight of apple, onion and carrot

   //greats the user
   printf("Welcome to YourGreens.com.\n");
   //gets the choice of the user and loops until user enters q
   while ((choice = get_choice()) != 'q')
   {
      switch (choice)
      {
         case 'a' :
            printf("\nPlease enter amount of artichokes in kg. \n");
            //adds weight to the total weight of artichokes
            TW_A += get_weight();
            break;

         case 'b' :
            printf("\nPlease enter amount of onions in kg. \n");
            //adds weight to the total weight of onions
            TW_O += get_weight();
            break;

         case 'c' :
            printf("\nPlease enter amount of carrots in kg. \n");
            //adds weight to the total weight of carrots
            TW_C+= get_weight();
            break;

         default :
            //displays after incorrect input
            printf("\nProgram Error.\n");
            break;
      }
   }
```

After the ‘switch statement’, “while (getchar() != '\n');” is used to flush the input buffer. In other words, it loops to discard unwanted characters as well as the “\n” newline character so that next the input function has a clean stream. This is then followed up by the calling of the ‘display’ function which displays a summary of the items the user has purchased and the overall price. Using the ‘printf’ function the user is asked whether they would like a receipt. An ‘if’ statement is then used with a condition such that if the user inputs the word “yes”, the letter “y” will be considered using the ‘getchar()’ function and sets the condition to true which in turn notifies the user the receipt is being saved and calls the ‘get_receipt’ function. If the user inputs any other words that do not begin with the letter “y”, the ‘if ‘condition will be false and a message will display thanking the user for using the website and the program will terminate. The figure below displays the expected output and the actual output resulting from the ‘if’ statement.

Input	Actual Output	Expected Output
‘yes’	“Saving receipt…”	“Saving receipt...”
‘y’	“Saving receipt...”	“Saving receipt...”
‘n’	“Thank you for shopping with us! Have a nice day!”	“Thank you for shopping with us! Have a nice day!”
‘5’	“Thank you for shopping with us! Have a nice day!”	“Thank you for shopping with us! Have a nice day”

```
//removes excess possible characters
   while (getchar() != '\n');

   //function that displays a summary of all the items and costs
   display(TW_A,TW_O,TW_C);

   //asks the user if they wants a copy of the receipt
   printf("\nWould u like a copy of this receipt?.\n");
   printf("Please enter 'yes' or 'no'.\n");

   //if user enters yes it will copy of receipt/puts it in separate text file
   if ((getchar()) == ('y')) {
      //function that prints the summary into a seperate text file
      printf("Saving receipt...\n");
      get_receipt(TW_A,TW_O,TW_C);
   }

   //end of code
   printf("Thank you for shopping with us! Have a nice day!");
   return 0;
}
```

'get_choice' Function

This function is responsible for reading and validating the users input which is later used in the switch statement in the main function. The function is of type 'char' as it will be returning a single character after being called.

First, a character type variable is declared that will store a single character based on the user's input. A list of options is then displayed and the user is prompted to choose one. Using the 'getchar' function a single character is considered from the users input. It is then passed through an 'if' statement which validates the character. If the character is neither a, b, c or q, the condition is set to true. The buffer is then flushed and the user is then asked to input a valid character. Next, the character is read again and then returned. If the user inputs a valid character from the start, the 'if' condition is set to false and the character is returned immediately.

```
//function that reads the choice input from the user
char get_choice(void)
{
   char ch;
   //displays the choices that the user can pick
   printf("\nPlease choose one of the following options:\n");
   printf("a -\t Enter amount of artichokes in kg.\n");
   printf("b -\t Enter amount of onions in kg.\n");
   printf("c -\t Enter amount of carrots in kg.\n");
   printf("q -\t Proceed to checkout.\n");
   //reads users input
   ch = getchar();
   //ensures that reader inputs the correct letters
   if((ch < 'a' || ch > 'c') && ch != 'q'){
      //flushes the buffer (so that it prints error once)
      while (getchar() != '\n');
      printf("\nPlease input a,b,c or q.\n");
      //re-reads the user input if previous input was incorrect.
      ch = getchar();
   }
   //returns value to back to main
   return ch;
}
```

The figure below displays the expected outcome and actual outcome from the ‘get_choice’ function.

Input	Actual Outcome/Output	Expected Outcome/Output
‘a’	Returns ‘a’	Returns ‘a’
‘2’	“Please input a,b,c or q.”	“Please input a,b,c or q.”
‘no’	“Please input a,b,c or q.”	“Please input a,b,c or q.”

‘get_weight’ Function

The purpose of this function is to read and validate the users weight input for each vegetable. The function is of type ‘float’ as it will return the weight of the certain vegetable which can be of decimal point value. At the start a float variable is declared which will store the decimal point number the user will input.

The function ‘scanf’ is used to obtain and store the float number in the variable ‘weight’. An ‘if’ statement is used with a condition such that if the user enters a valid input such as any number, ‘scanf’ will successfully convert the item and return 1 and the test condition will be set to true. The buffer is then flushed and the weight is validated through another ‘if’ statement. The statement tests whether the weight variable is greater than 0 and less than 100 to avoid abnormal amounts to be inputted by the user. If it passes the test conditions, the weight is returned. If it doesn’t pass the conditions, the user is prompted to enter a valid number and is returned to the menu. However, if the user inputs a non-number input at the start, ‘scanf’ will return EOF and the test condition will be false. This will notify the user that they entered invalid input and then returns to the menu. The figure bellows displays the expected outcome and actual outcome from the ‘get_weight’ function.

Input	Actual Outcome/Output	Expected Outcome/Output
‘20’	Returns ‘20’	Returns ‘20’
‘0’	“Error, please input value above 0 and below 100.”	“Error, please input value above 0 and below 100.”
‘no’	“Invalid input. Try again.”	“Invalid input. Try again.”

```
//function that reads the weight input from the user
float get_weight()
{
   float weight;

   //checks if the user enters a number, else it will be considered invalid input
   if (scanf("%f", &weight) == 1) {
      //flushing buffer to disallow excess input to be considered later
      while (getchar() != '\n');
      //ensures that reader inputs appropriate weight
      if (weight > 0 && weight < 100) {
         return weight;
      } else {
         printf("Error, please input value above 0 and below 100.\n");
         return 0;
      }
   }
   else
   {
      //flushing buffer to disallow excess input to be considered later
      while (getchar() != '\n');
      printf("Invalid input. Try again.\n");
      return 0;
   }
}
```

'display' Function

This function prints a summary of all the items the user entered, their prices and the overall price after shipping and discounts were considered. The function is of type 'void' as it returns no variables. It has formal parameters which represent the total weight of each vegetable. These will be needed to calculate the price of each vegetable as well as the overall price. At the start of the 'display' function, the total price of each vegetable is calculated by multiplying its total weight by its price, which was defined as a macro previously. The total weight, 'total_w', of all the vegetables is then obtained and will be used to determine the shipping costs. The total price, 'total_p', of all the vegetables is also calculated. It is important to mention that I will be using the dollar sign instead of the euro sign as it would not display correctly. The same issue was encountered in the previous question.

```
void display(float TW_A, float TW_O, float TW_C)
{
   printf("\n---------- Checkout ----------\n");
   //calculations for the price of each vegetable
   float TP_A = TW_A * P_A;
   float TP_O = TW_O * P_O;
   float TP_C = TW_C * P_C;
   //calculations for total price and total weight
   float total_p = TP_A + TP_O + TP_C;
   float total_w = TW_A + TW_O + TW_C;
```

Next, the weight and price of each vegetable are displayed. It is important to note that they are only printed if the total price or total weight of a vegetable is greater than 0. This is done using an if statement and avoids the display of vegetables that have not been bought. The total price of all products before discounts and shipping costs are then printed.

```
//will only print products that have been bought
   if (TP_A > 0 || TW_A > 0) {
      printf("%.2f kg of Artichokes : $%.2f \n", TW_A, TP_A);
   }
   if (TP_O > 0 || TW_O > 0) {
      printf("%.2f kg of Onions : $%.2f \n", TW_O, TP_O);
   }
   if (TP_C > 0 || TW_C > 0) {
      printf("%.2f kg of Carrots : $%.2f \n", TW_C, TP_C);
   }

   //prints the total of all products before discounts and shipping costs
   printf("Total : $%.2f\n", total_p);
   printf("---------------------------------\n");
```

Subsequently, a float variable called 'discount' is initialized which will represent the amount of money that will be subtracted from the current total price after a discount is applicable. The customer is only entitled to a discounted price after spending over a $100, thus, an 'if' statement is used to ensure that the discount is applied only when the total price is over $100. The total price after the discount has been applied is then displayed.

```
float discount = 0;
   //calculate discount if the price is above 100
   if (total_p > 100.0)
   {
      discount = (total_p * DISC);
      printf("Total with 5%% discount : $%.2f\n", total_p - discount);
   }
```

Furthermore, the shipping cost was calculated according to the total weight of all the vegetables. The total weight was passed through an if statements followed by a chain of else if statements to check for the appropriate shipping cost based on the information given in the assignment. The total weight can not have a negative value due to previous precautions taken in the code.

- If the total weight was 0, signifying no items bought, there is no shipping costs given.
- If the total weight is less than or equal to 5kg, the shipping cost is $6.50.
- If the total weight is greater than 5kg but less than 20 kg, the shipping cost is $14.00
- If the total weight is greater than 20 kg, the shipping cost starts at $14.00 but $0.50 is added per extra kilogram. This is done using a for loop which loops until the index representing total weight of the products is less than 20kg. Every time it loops, $0.50 is added to the shipping cost and the index is reduced by 1 signifying the charge of an additional kilogram.

```
    //calculate shipping price according to the weight
    int index;
    float ship = 0;
    if(total_w == 0){
        ship = 0;
    } else if(total_w <= 5.0){
        ship = 6.5;
    } else if (total_w > 5.0 && total_w < 20.0) {
        ship = 14.0;
    } else if (total_w >= 20) {
        ship = 14.0;
        //function that adds 0.50 per kg for shipments of 20kg more
        for (index = total_w; index >= 20.0; index--) {
            ship += 0.5;
        }
    }
```

At the end of the function, the shipping cost is displayed using the 'printf' function as well as the final price with the discount and shipping costs applied.

```
//prints shipping cost and final overall price.
    printf("Shipping cost : $%.2f\n", ship);
    printf("-----------------------------------\n");
    printf("Overall total : $%.2f\n", total_p - discount + ship);
}
```

The figure bellows displays the expected output and actual output from the 'display' function.

Values Inputted into Function	Actual Output	Expected Output
TW_A = 70 TW_O = 0 TW_C = 50	---------- Checkout ---------- 70.00 kg of Artichokes : $143.50 50.00 kg of Carrots : $54.50 Total : $198.00 ----------------------------------- Total with 5% discount : $188.10 Shipping cost : $64.50 ----------------------------------- Overall total : $252.60	---------- Checkout ---------- 70.00 kg of Artichokes : $143.50 50.00 kg of Carrots : $54.50 Total : $198.00 ----------------------------------- Total with 5% discount : $188.10 Shipping cost : $64.50 ----------------------------------- Overall total : $252.60

'get_receipt' Function

This function has an almost identical structure to the 'display' function however, its purpose is to save the summary in a separate text file. The function is of type 'void' as it returns no variables. The formal parameters are the same as the 'display' function mentioned previously. Nonetheless, a pointer to file called "bill.txt" is defined at the start. The text file is opened as an empty file for writing use the mode "w".

Next the file is passed through an 'if' statement which checks whether the file has been opened. If it there were issues encountered when opening the file, an error message is displayed and the program is terminated using the 'exit()' function. The macro 'EXIT_FAILURE' indicates an unsuccessful execution of the program. If there no issues were encountered when opening the file, the test condition is false and the code continues.

```
void get_receipt(float TW_A, float TW_O, float TW_C)
{
   //opens a file called bill.text where the receipt will be 'printed'
   FILE *f = fopen("bill.txt", "w");
   //if cant open file
   if (f == NULL) {
      printf("Error opening file!\n");
      exit(EXIT_FAILURE);
   }

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

The remainder of the code is similar to the code of the 'display' function. The only difference is that the function 'fprintf()' is used instead of 'printf()' so that the output could be sent to the file instead of the text console. Below is an example of how the 'fprintf()' function was used to print the total weight and price of the products.

```
//will only print products that have been bought
   if (TP_A > 0 || TW_A > 0) {
      fprintf(f,"%.2f kg of Artichokes : $%.2f \n", TW_A, TP_A);
   }
   if (TP_O > 0 || TW_O > 0) {
      fprintf(f,"%.2f kg of Onions : $%.2f \n", TW_O, TP_O);
   }
   if (TP_C > 0 || TW_C > 0) {
      fprintf(f,"%.2f kg of Carrots : $%.2f \n", TW_C, TP_C);
   }
```

Important Remarks

To conclude, one improvement that could have been made is a function that calculates all of the weights and costs separately so that the 'display' and 'get_receipt' function would simply have to print out the values rather than calculate all the values in both functions. It has also been noted that in the 'get_choice' function, if the user was to input a word beginning with the letters a, b, c or q, such as 'artichoke', the letter 'a' would first be considered in the 'switch' statement, however, an error message would be displayed right after, disallowing the user from entering the weight for the vegetable.

Question 1c

At the start of the code, '<stdbool.h >' is included which permits the use of Boolean type variables. The header '<string.h>' is also included which contains several functions for manipulating arrays of characters. Two functions are declared, one of which will compare case-sensitive strings and another of which will compare case-insensitive strings. Next, the code of each function including the main function will be described separately.

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

bool compare_strings_case(char string1[],char string2[]);  //function that compares strings that are case-sensitive
bool compare_strings(char string1[],char string2[]);       //function that compares strings that are case-insensitive
```

Main Function

In the main function, several variables are defined. A pointer to a file is declared which is then opened with the name "input.txt" for reading using the mode "r". An 'if' statement is used to print an error message and exit the program in case the file did not open. A pointer to a character called 'line' is declared which will point to the address of the first character in each line read. Then a variable called 'length' is declared to store the length of characters in each line read. It is of type 'size_t' to allow a large amount of characters to be stored in a line. Three Boolean variables are initialized with the following intentions:

- 'end' – set to true if file is recognizable
- 'html_1' – set to true if <html> is present in the file
- 'html_2' – set to true if </html> is present in the file

```
int main(){
   FILE *f;                  //pointer to a file object
   char *line = NULL;        //address of first character in buffer (in the line)
   size_t length;            //length of each line (or size of buffer)
   bool end = false;         //true if at least one condition is met (file is recognizable)
   bool html_1 = false;      //true if <html> is present (at start)
   bool html_2 = false;      //true if </html> is present (at end)

   //opens a text file which it will read
   f = fopen("input.txt","r");
   //executes if there is trouble opening file
   if (f == NULL)
   {
      printf("Error opening file.\n");
      return (EXIT_FAILURE);
   }
```

Next, the 'getline()' function is used to read each line from the text file. The function is used as part of a test condition in a while loop which will terminate as soon as the 'getline()' function reaches the end of the file. The 'line' variable declared previously will point to the address of where the first character in the buffer is stored. Until it reaches the end of stream, each line from the text file will be considered in the while loop.

Next, using an ‘if’ statement, a line is checked whether it contains the string “<html>” using a function that compares two case-insensitive strings. If the function returns the Boolean value of true, the two Boolean variables ‘end’ and ‘html_1’ are set to true indicating that the file is recognizable and that “<html>” is present. If the function returns the Boolean value of false, the line is tested again through an else if statement to whether it contains the string “</html>” using the same function that compares case-insensitive strings. If it returns the Boolean value of true, ‘end’ and ‘html_2’ are set to true indicating that the file is recognizable and that “<html>” is present.

If the text file did not contain the string “<html>” which should be present at the start for every HTML file, then ‘html_1’ would remain false. The program will then check whether the line contains the string “#include” which will conclude that the file is of C type using a function that compares case-sensitive strings. If the function returns a Boolean value of true, the program will display that the file is of C type and set the Boolean variable ‘end’ to true signifying that the file type is recognizable. The ‘while’ loop is then exited using a ‘break’ statement to cease the search for a file type. If “#include” was not present in the line, the code will loop back to start of the while loop and check the next line in the text file.

The program only checks for a C file type after it has checked for the HTML type to avoid it declaring the file type as both C and HTML. This is further ensured from the ‘break’ statement once the file is recognized as a C file type.

```
//scans each line and stores it in a buffer
while ((getline(&line, &length, f)) != EOF)
{
   //checks if <html> or if </html> is present in file
   if ((compare_strings(line,"<html>")) == true) {
      end = true;
      html_1 = true;
   }
   else if ((compare_strings(line,"</html>")) == true) {
      end = true;
      html_2 = true;
   }

   //if the start of the html hasn't been detected, it will search for a C file
   //ensures that the file can't be both C and HTML type (also cause of break below)
   if( html_1 == false) {
      //checks if #include is present in file
      if ((compare_strings_case(line, "#include")) == true) {
         printf("The file is of C type.\n");
         end = true;
         //exits while loop to stop searching
         break;
      }
   }
}
```

After the ‘while’ loop terminates, both Boolean variables ‘html_1’ and ‘html_2’ are used in an ‘if’ statement to determine whether the file was of HTML type. If both variables are true, the user is told that the file is of HTML type. If only one of them was true, the program displays that the HTML file is complete as only the starting line or ending line of the HTML file was present. If both are false the program continues.

Next the ‘end’ Boolean variable is checked, if it remained false through the entire code then the user will be notified that the file type was unrecognizable. At the very end of the main function the file is then closed using the function ‘fclose()’.

```
if(html_1 == true && html_2 == true) {
   //if the file contains both </html> and <html>, it is a complete html file
   printf("The file is of HTML type.\n");
} else if (html_1 == true || html_2 == true) {
   //if it contains either </html> or <html> but not both, it is an incomplete html file
      printf("The HTML file is incomplete.");
}

//executes if file is neither of specified above
if (end == false) {
   printf("Can't recognize file type.\n");
}

//closes file
fclose(f);
return 0;
```

Please refer to the figure below which lists the expected output and actual output based on various examples of possible inputs from the text file where the line “Input text here” can consist of several lines of text.

Text File (Input)	Actual Output	Expected Output
<HTML> Input text here. </Html>	“The file is of HTML type.”	“The file is of HTML type.”
#include <stdio.h> #include <stdlib.h> Input text here.	“The file is of C type.”	“The file is of C type.”
<hTmL> #include <stdio.h>	“The HTML file is complete.”	“The HTML file is complete.”
#INCLUDE <stdio.h> #INCLUDE <stdlib.h>	“Can’t recognize file type.”	“Can’t recognize file type.”

‘compare_strings’ Function

This function compares case-insensitive strings and is of type Boolean as it returns a Boolean value of true or false based on whether the strings match or not. The formal parameter list consists of two seperate arrays of characters which will then be compared to each other. The first array ‘string1’ will represent a line that was extracted from the text file whereas ‘string2’ will be the word that is being searched for in the line. A Boolean variable is initialized called ‘contains’ and will be set to true when ‘string2’ is found in ‘string1’. A function called ‘strcasestr()’ is used to compare the two strings and

returns null if ‘string2’ is not present in ‘string1’. This function is similar to the ‘strstr()’ function but ignores the case of both arguments. If ‘string2’ is present in ‘string1’ the test condition becomes true and the Boolean variable ‘contains’ is set to true and then returned. If the string is not present, the function will return null and the Boolean variable ‘contains’ remains false and is returned indicating that the string was not found in the line.

It is important to note that ‘strcasestr()’ does not work on a Windows operating system, the code displayed was constructed on a Mac OS X operating system.

```
//function compares case-insensitive strings
bool compare_strings(char string1[],char string2[]) {
   bool contains = false;
   //if strings are identical (case-insensitive), function will return TRUE
   if (strcasestr(string1, string2) != NULL) {
      contains = true;
   }
   return contains;
}
```

‘compare_strings_case’ Function

This function is identical to the ‘compare_strings’ described previously. The only difference is that this method uses the function ‘strstr()’ instead of ‘strcasestr()’ which compares two strings but this time considers the case of both arguments. If the strings are not identical, it will also return a null value.

```
//function that compares case-sensitive strings
bool compare_strings_case(char string1[],char string2[]) {
   bool contains = false;
   //if strings are identical, function will return TRUE
   if(strstr(string1, string2) != NULL) {
      contains = true;
   }
   return contains;
}
```

Important Remarks

To conclude, it is important to mention a limitation that must be considered when using this code. The code involves the use of two functions ‘strcasestr()’ and ‘getline()’ which are not recognizable inthe Windows operating system as the code was constructed on a Mac OS X operating system. However, both functions are recognizable on a Linux operating system.

Question 1d

At the start several macros are defined such as ‘MAX’ which is of value 100 and will represent the maximum amount of characters than the program will consider. The ‘SPACE’, ‘COMMA’ and ‘DOT’ are all set to key characters that will be searched for to fix the specified spelling mistakes. A function is also defined which will be called when a word is 12 or more characters to ask the user if the word is correct. A description of the main function and the ‘word_check’ function will now follow.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
#define SPACE ' '
#define COMMA ','
#define DOT '.'


void word  check(char str[], int x, int length, FILE *f);
```

Main Function

First, an array called ‘ch’ of type ‘char’ is declared which can hold 1000 elements based on the definition of the macro MAX. This array will store all the characters read from the text file. The integer variables ‘n’ and ‘letters’ are initialized to a value of 0 to avoid potential bugs. The variable ‘n’ represents the total number of characters in the array ‘ch’ and the variable ‘letters’ is used to determine the length of each word to check whether a word is made up of 12 letters. A pointer to a file is declared which is then opened with the name “auto-correct” for reading using the mode “r”. An ‘if’ statement is used to print an error message and exit the program in case the file did not open.

```
int main() {
   char ch[MAX]; //stores the characters
   int n = 0, letters = 0; //no. of characters in array and letter counter (to count 12 or more)
   int i; //index
   FILE *f; //file pointer

   //the file is opened to reading
   f = fopen("auto-correct.txt","r");
   //executes if there is error opening file
   if (f == NULL) {
      printf("Error opening file.\n");
      return (EXIT_FAILURE);
   }
```

Subsequently, each character is scanned from the text file and store in the ‘ch’ array using the ‘getc()’ function which gets the next character from the specified stream. At the end of the file, the ‘getc()’ function returns ‘EOF’. Thus, a ‘while’ loop is used which terminate when the ‘getc()’ function reaches the end of a file, or in other words, returns ‘EOF’. The variable ‘n’ keeps track of the number of characters inputted by incrementing by a value of 1 each time a character is scanned from the text file. It is also used to access the next available space for next character to be stored in within the array. After the ‘while’ loop terminates, the last character of the array is set to null by accessing the “n^{th}” element, signifying the end of the string of characters.

```
//scans the entire file, until the end, and stores each characters in an array ch
while ((ch[n] = getc(f)) != EOF) {
   n++; //counts the number of characters inputted
}
ch[n] = '\0'; //sets the last character of the array to NULL
```

Next, the file is closed and then re-opened with mode “w+” for writing and erases all the files previous contents. This allows the program to rewrite the input text file into a corrected version after reading its contents. An ‘if’ statement is used to print an error message and exit the program in case the file did not open.

```
//closes the file
fclose(f);
//reopens the file but this time for writing and overwrites the data
f = fopen("auto-correct.txt","w+");
//executes if there is error opening file
if (f == NULL) {
   printf("Error opening file.\n");
   return (EXIT_FAILURE);
}
```

Please refer to the code below for the proceeding explanation.

```
//goes through loop for total no. of characters
for (i = 0; i < n; i++) {
   //checks if there is a space, comma or dot, indicating the end of a word
   //if there is, letters will be reset to 0, or else, if 12 letters already occurred, function will execute
   if(ch[i] == SPACE || ch[i] == COMMA || ch[i] == DOT) {
      if (letters >= 12) {
         //function that asks the user whether the input was correct
         word_check(ch, i, letters, f);
         //letter no.reset
         letters = 0;
      } else {
         letters = 0;   //if there wasn't 12 letters, letter no. reset
      }
   } else {
      letters++; //if not space, comma or dot letter no. incremented
   }

   //checks if there is a double space
   if (ch[i] == SPACE && ch[i-1] == SPACE) {
      //removes space by moving up array by one
      for(int j = i; j < n; j++) {
         ch[j] = ch[j+1];
      }
      //sets last character to null and decreases total no. of characters in array
      ch[n-1] = '\0';
      n--;
      //checks if there is a comma and space
   } else if (ch[i] == COMMA && ch[i-1] == SPACE) {
      //replaces space with comma and moves up array by one
      for(int j = i; j < n; j++)  {
         ch[j-1] = ch[j];
      }
      //sets last character to null and decreases total no. of characters in array
      ch[n-1] = '\0';
      n--;
   } else if (ch[i] == DOT && ch[i-1] == SPACE) {
      //replaces space with dot and moves up array by one
      for (int j = i; j < n; j++) {
         ch[j - 1] = ch[j];
      }
      //sets last character to null and decreases total no. of characters in array
      ch[n-1] = '\0';
      n--;
   }
}
```

The program proceeds by entering a ‘for’ loop which loops for the total number of characters. The index represents a certain character in the array, starting from the first one. At the end of each loop the next character is passed through the loop again by incrementing the index by a value of 1. This process repeats until all characters that were stored in the array ‘ch’ are individually passed through the loop.

At the start of the for loop, each character is passed through an if statement that checks whether the character is a space, comma or dot. If this condition is true, the program checks whether there have been 12 or more letters prior the current character using the variable ‘letters’. If this condition is also true, the ‘word_check’ function is called. If this condition is false, the “letters” variable is reset to 0 indicating the possible start of a new word. If, however, the character was neither a space, comma or dot, it will fail the first ‘if’ condition which will then consider the character to be a letter and increment the variable ‘letters’ by a value of 1. This part of the ‘for’ loop is important for detecting typos under the assumption that “words longer than 12 characters and not including a hyphen (-) are rare” which was given in the assignment sheet.

In the next part of the ‘for’ loop, the character is passed through an ‘if’ statement followed by two ‘else if’ statements to determine whether there are any multiple spaces, spaces before commas or spaces before full stops.

The first ‘if’ statement tests whether the current character and the previous character is a space. If the test condition is true, a ‘for’ loop is used replace the current character with the following character in the array ‘ch’ and this process is carried out for all the successive characters until the last character. This will result in the removal of one of the double spaces. Since the current character has been replaced and no longer exists in the array, the last character of the array is now the “(n-1)th” character and is set to null. The total number of characters, ‘n’, is then decreased by a value of 1. If the current character and the previous character is not a space, the code will continue to test the character in the next ‘else if’ statement.

The succeeding ‘else if’ statement tests whether the current character is a comma and whether the previous character is a space. If these test conditions are true, a character will be replaced much like the method mentioned above, however, this time the previous character will be replaced as the space needs to be replaced and not the comma. The array will then be moved up by one. This is done using a ‘for l’oop which replaces the previous character with its succeeding character and this process is carried out for the rest of the successive characters in the array until the last character. This will result in the removal of the space character before the comma character. Since the previous character has been replaced and no longer exists in the array, the last character of the array is now the “(n-1)th” character and is set to null. The total number of characters, ‘n’, is then decreased by a value of 1. If the test conditions are false, the code will proceed to test the next ‘else if’ statement.

The following ‘else if’ statement tests whether current character is a dot and whether the previous character is a space. If these test conditions are true, the previous character will be replaced using a ‘for ‘loop with the same method as the above ‘else if’ statement description. This will result in the removal of a space character before the dot character. The last character of the array is then set to null and the total number of characters denoted by ‘n’ is decreased as mentioned prior.

After the initial for loop is executed, the code prints the corrected output to the file using the ‘fprintf()’ function. The final corrected output is stored in the ‘ch’ array as a string of characters which has been modified throughout the code. The file is then closed and the program terminates.

```
//prints the corrected version of the input into file
   fprintf(f,"\nCorrected output:\n%s", ch);
   //closes file
   fclose(f);
   return 0;
}
```

Please refer to the figure below which lists the expected output and actual output of various examples based on possible inputs from the text file.

Text File (Input)	Text File (Actual Output)	Text File (Expected Output)
I am a file withmistakes , please fix me .	Are you sure you meant withmistakes? Corrected output: I am a file withmistakes, please fix me.	Are you sure you meant withmistakes? Corrected output: I am a file withmistakes, please fix me.
Double space , Double space .	 Corrected output: Double space, Double space.	 Corrected output: Double space, Double space.

'word_check' Function

This function prints out a message to the user to validate whether they meant a certain word if it contained 12 or more letters. The function is of type 'void' as it does not return any variables. I will now describe the following formal parameters of the function:

- The 'str' variable represents the address of the first character in the 'ch' array.
- The 'x' variable represents the position of the character following the last character of the word in the 'ch' array.
- The 'length' variable represents the number of letters in the word.
- The 'f' pointer will point to the file where the message will be displayed to the user.

First a message is printed to the file using the 'fprintf()' function which asks whether the word is valid. Next the word is printed character by character. By subtracting the variable 'length' from the variable 'x', the position of the first character of the word in the 'ch' array is obtained. This is used as the starting condition in the 'for' loop which will print out each character of the word individually until the very last character. This output is then displayed in the corrected version of the text file using the 'fprintf()' function.

```
void word_check(char str[], int x, int length, FILE *f) {
   //length = number of letters in the word
   //str = the address of the first character in the char array
   //x = the character after the last character of the word in the char array (will be a space, comma or dot)
   int i;

   fprintf(f, "Are you sure you meant ");
   //prints the word that is longer than 12 words
   for (i = (x - length); i < x; i++) {
      fprintf(f, "%c", str[i]);
   }
   fprintf(f, "?\n");
}
```

Important remarks

An important limitation to note is that the code will not fix multiple space consisting of more than two spaces. In other words, it will only remove a single space character even if the input contains more than two spaces in between two words.

Question 1e

At the start of the code, the 'view_stackframe' function is declared. The function will print out the value of an integer which is stored in an array followed by its memory address.

```
#include <stdio.h>
void view_stackframe(int array[], int length);
```

In the main function, a variable 'n' of integer type is declared and initialized. It represents the size of the array which is of type integer as well. The variable 'n' can be initialized to any value but in this example, it has been set to 5. Thus, the array 'value' can hold 6 elements. Each element of the array is then initialized which can also be adjusted by the programmer. In this example, random integer values were assigned to each element. The function 'view_stackframe' is then called with arguments such that the array 'value' and variable 'n' are passed through. The program then terminates.

```
int main(){
   //n represents the size of the array 'value'
   int n = 5;
   int value[n];
   //initializing elements of the array 'value'
   value[0] = 1;
   value[1] = 10;
   value[2] = 50;
   value[3] = 22;
   value[4] = 47;
   value[5] = 36;

   //calling function that will print out the values and memory address of each element
   view_stackframe(value, n);
   return 0;
}
```

The function 'view_stack' frame will display the values of each element in the array and the address of these elements in two columns. This is done using a 'for' loop with conditions which terminates the loop after the last variable and its address is printed. The placeholder '%p" is used to display the memory address of each element in the array.

```
void view_stackframe(int value[], int length){
   int i;
   //prints out value and memory address in two separate columns
   for(i = 0; i <= length; i++) {
      printf("%d\t\t%p\n", value[i], value[i]);
   }
}
```

Below is a figure displaying the expected output and actual output after the code is executed.

Actual Output/Outcome		Expected Output/Outcome	
1	00000001	1	"memory address of variable"
10	0000000A	10	"memory address of variable"
50	00000032	50	"memory address of variable"
22	00000016	22	"memory address of variable"
47	0000002F	47	"memory address of variable"
36	00000024	36	"memory address of variable"

Question 2a

At the start of the code two macros are defined. One of them is “MAX” which represents the maximum amount of characters a user can input into the data string and the key string whilst “SIZE” represents the number of rows (or hash indexes) and columns (or collisions) in the hash table. The macro “SIZE” can be adjusted by the programmer depending on how many elements they prefer in their hash table. In this example of the code, “SIZE” is assigned the value of 5, so the hash table will contain a maximum of 25 elements or data points.

A number of functions are defined which will have the identical format and parameters in the questions a, b and c for part 2 of the assignment. However, in each question the code within each function will vary to suit the different structures of the hash table. Below is a short description of each function:

- The ‘option’ function displays a list of options the user can choose from such as inputting strings into the hash table or removing them.
- The ‘get_index’ function contains the hashing function which calculates the hash index using the data string and key string. It will return the hash index which is a variable of integer type.
- The ‘insert’ function allows the user to enter a data string and key string into the hash table.
- The ‘display’ function displays the values that were inputted by the user in the hash table.
- The ‘search’ function searches for a specific data string and key string in the hash table.
- The ‘delete’ function allows the user to delete a specific data string and key string from the hash table.
- The ‘save’ function saves the contents of the hash table to a text file.

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#define MAX 100
#define SIZE 5

void option();
int get_index(char *key, char *data, size_t key_len, size_t data_len);
void insert(char *key, char *data, size_t key_len, size_t data_len);
void display();
void search(char *key, char *data, size_t key_len, size_t data_len);
void delete(char *key, char *data, size_t key_len, size_t data_len);
void save();
```

Next, a struct called “point” is created with the variables ‘key’ and ‘data’ which are character arrays. These two variables will hold the key string and data string that the users will input at each position in the hash table. Subsequently, a 2D array is defined called “TABLE” which will represent the hash table itself with a 5x5 parameter. It is of type ‘struct point’ so that it can hold a key string and data string at each element of the 2D array. The 2D array is considered a global variable as it is defined outside the main function. This allows it to be used in all functions.

```
struct point {
   char key[MAX];
   char data[MAX];
};

//array is predefined with parameters SIZE
struct point TABLE[SIZE][SIZE];
```

Main Function

At the start of the function the character arrays “key” and “data” are declared which will temporarily store the users data string and key string input before they are passed through one of the functions. A character variable ‘choice’ is also declared which stores a single character that will determine the case in the upcoming switch statement.

Next the ‘TABLE’ array which was declared previously is now initialized using two ‘for’ loop statements. These loops set the first character of the key string and data string in every point of the array ‘TABLE’ to null. This will help in determining whether a position in the hash table is empty or not.

```
int main() {
   char key[MAX], data[MAX], choice;

   //setting the first character of the strings in each elements to NULL
   for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
         TABLE[i][j].key[0] = '\0';
         TABLE[i][j].data[0] = '\0';
      }
   }
```

The following description of the main function will be identical in the main functions of the upcoming questions 2b and 2c. First. the ‘option’ function is called which displays the possible decisions the user can take. The program then scans a character from the user using the ‘getchar() ‘function and stores it in the variable ‘choice’. This is done within a ‘while’ loop test condition which will terminate when the user inputs the letter “q”. If the user enters any character besides the letter “q”, the character passes through the ‘while’ loop and considered in the ‘switch’ statement. It is important to note that the buffer is flushed to avoid excess characters from being considered in future inputs. Below is a short description of each of the cases within the ‘switch’ statement:

- Case ‘a’ allows the user to input a key and data string into the hash table.
- Case ‘b’ will display all the key and data string that were previously inputted in the text console.
- Case ‘c’ allows the user to search for a specific key and data string.
- Case ‘d’ allows the user to delete a specific key and data string.
- Case ‘s’ saves the contents of the hash table to a text file.
- The default case prints out an error message.
- If the user enters “q”, the ‘while’ loop terminates.

Below is a table representing the actual outcomes and expected outcomes from the ‘switch’ statement.

Input	Actual Output/Outcome	Expected Output/Outcome
‘a’	Enters case ‘a’	Enters case ‘a’
‘b’	Enters case ‘b’	Enters case ‘b’
‘q’	Exits ‘while’ loop	Exits ‘while’ loop
‘n’	“Error.”	“Error.”

In case ‘a’, the user is asked to input a key string and data string which is then passed through the ‘insert’ function along with the length of both strings. The insert function will input both strings into an available space in the hash table. The length of both strings is determine used the ‘strlen()’

function which calculates the length of the string excluding the terminating null character. The length of both strings is needed to calculate the hash index.

```
//displays the options the user can select
option();
//loops until the user inputs 'q'
while ((choice = getchar()) != 'q')
{
   //flushes excess characters
   while (getchar() != '\n');
   switch (choice) {
      case 'a' :
         //entering values to insert
         printf("Please enter the key to insert.\n");
         scanf("%s", key);
         printf("Please enter the data to insert.\n");
         scanf("%s", data);
         //function that enters data into array
         insert(key,data,strlen(key),strlen(data));
         //flush excess characters
         while (getchar() != '\n');
         break;
```

In case ‘b’, the ‘display’ function is called which will list the contents of the hash table that were inputted by the user. In case ‘c’, the user is asked to input a key string and data string which is then passed through the ‘search’ function so that the program locates the strings within the hash table. The length of both strings is also passed through to calculate the hash index.

```
case 'b':
   //function that displays values previously entered
   display();
   break;

case'c':
   //entering values to search
   printf("Please enter the key to search.\n");
   scanf("%s", key);
   printf("Please enter the data to search.\n");
   scanf("%s", data);
   //function that searches for elements in array
   search(key,data,strlen(key),strlen(data));
   //flush excess characters
   while (getchar() != '\n');
   break;
```

In case ‘d’ the user is asked to input a key string and data string which is then passed through the ‘delete’ function which will delete the specified strings from the hash table. The length of both strings is also passed through to calculate the hash index. In case ‘s’, the save function is called which will display the contents of the hash table in a text file.

```
case'd':
   //entering values to delete
   printf("Please enter the key to delete.\n");
   scanf("%s", key);
   printf("Please enter the data to delete.\n");
   scanf("%s", data);
   //function that searches for elements in array
   delete(key,data,strlen(key),strlen(data));
   //flush excess characters
   while (getchar() != '\n');
   break;
```

```
case's':
    save();
    break;
```

The default case will execute when the user inputs a character which isn't a, b, c, d, s or q. If it executes, it will print an error message to the user and the code will resume. After each statement is considered, the 'option' function is called to relist the options to the user and the while loop starts again. If the user inputs "q", the 'while' loop will terminate and the program will end.

```
default:
            printf("error\n");
            break;
        }
        //displays options for users
        option();
    }
}
```

'option' function

This function is identical in questions 2a, 2b and 2c. It displays the options that the user can choose from by entering a specific character.

```
//function that displays options
void option() {
    printf("\nPlease choose a,b,c,d,e or q.\n");
    printf("a - enter values.\n");
    printf("b - display values.\n");
    printf("c - searching for values.\n");
    printf("d - deleting values.\n");
    printf("s - saving values.\n");
    printf("q - quit.\n");
}
```

Below is a figure illustrating the output obtained in the text console after the function is executed.

Output
Please choose a,b,c,d,e or q. a - enter values. b - display values. c - searching for values. d - deleting values. s - saving values. q - quit.

'get_index' function

This function calculates the hash index of the inputted strings. Within the first 'for' loop, each character in the key string is scanned. The ASCII code value of each character is then stored in a variable called 'value' which adds onto the variable 'key_sum'. At the end of the loop the variable 'key_sum' will represent the total value of all the characters in the key string by adding up their ASCII code value. The same process is carried out in the next loop however this time the total value of all the characters in the 'data' array is summed up in the variable 'data_sum'. These two sums are then added together and stored in the variable 'total'. The variable 'hash_index' is then calculated by finding the remainder from the division of the variable 'total' by the macro 'SIZE'. 'SIZE' will

represent the number of rows in the hash table in all the future questions. In this question, “SIZE” holds the value 5. The hash_index is then returned.

```
//function that calculates the hash_index using the ASCII value of key and data
int get_index(char *key, char *data, size_t key_len, size_t data_len)
{
   int i, value = 0, key_sum = 0, data_sum = 0;
   //calculating ASCII value of the key
   for (i = 0; i < key_len; i++) {
      value = *key + i;
      key_sum += value;
   }
   //calculating the ASCII value of the data
   for (i = 0; i < data_len; i++) {
      value = *data + i;
      data_sum += value;
   }
   int total = key_sum + data_sum;
   int hash_index = total % SIZE;
   return hash_index;
}
```

‘insert’ Function

This function will store the user inputted key string and data string in an available position in the hash table. The pointer ‘key’ points to the address of the first character in the inputted key string and the pointer ‘data’ points to the address of the first character in the inputted data string. The variable ‘key_len’ represents the length of the key string and the variable ‘data_len’ represents the length of the data string. The function is of void type as it will not return any variables. At the start a Boolean variable is initialized which will be set to true when an empty position is found in the hash index. The integer variable ‘count’ is used to keep track of the columns (or collisions) in the hash table and will also be used to determine whether the hash index has reached the maximum number of collisions. The hash index is calculated by calling the ‘get_index’ function which is then stored in the integer variable ‘hash_index’. Next the code will check for an empty position within this hash index.

A ‘while’ loop is used with a test condition which will terminate the loop once the Boolean variable ‘empty’ is set to true. In other words, the loop will terminate once an empty position is found in the hash index for the strings to be stored in. At the start of the ‘while’ loop, an ‘if’ statement is used to check whether the variable ‘count’ has a value of 5 signifying that the total number of collisions has been reached and there are no empty positions for the strings to be stored. If this test condition is true, the user is notified that the maximum amount of collisions has been reached and a ‘break’ statement is used to exit the ‘while’ loop and the function terminates. If this condition is false, the code resumes to check whether the current collision in the hash index is empty or not.

Using an ‘if’ statement, the program checks whether a key string has already been stored in the current collision by testing whether the first character of the key string is null. If the test condition is true, this means that no strings have been stored in the particular collision. Subsequently, the key string and data string are copied character by character into hash table with a position determined by the current hash index and collision. This is done through the use of two ‘for’ loop statements. However, if the test condition was false, the variable ‘count’ will increment by a value of 1 and the loop will recommence so that the next collision in the hash index can be checked to determine whether it is empty or not.

```
void insert (char *key, char *data, size_t key_len, size_t data_len)
{
   bool empty = false;
   int count = 0, i; //count keeps track of the collisions in the index
   //calculates the hash_index using a function
   int hash_index = get_index(key,data,key_len,data_len);

   //checks if position in array is empty
   while (empty == false) {
      //executes if the total number of collisions in the hash index has been reached
      if (count == 5) {
         printf("The total number of collisions in the hash table has been reached.");
         break;
      }
      if (TABLE[hash_index][count].key[0] == '\0') {

         for (i = 0; i < key_len; i++) {
            TABLE[hash_index][count].key[i] = key[i];
         }

         for (i = 0; i < data_len; i++) {
            TABLE[hash_index][count].data[i] = data[i];
         }
         //true because an empty space was found, and strings were copied
         empty = true;
      } else {
         //increment count to keep searching for an empty space
         count++;
      }
   }
}
```

Below are the expected and actual outcomes of the 'insert' Function depending on the input of the key and string as well as the contents of the hash table. Please note that the rows represent the index of the hash table while the columns represent the collisions in the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b"	Key and String are stored in row 0 and column 0 in the 2D array.	Key and String are stored in column 0 of a certain row in the 2D array.
Key: "a", Data: "b"	Key: "a", Data: "b"	Key and String are stored in row 0 and column 1 in the 2D array.	Key and String are stored in column 1 of a certain row in the 2D array.
Key: "a", Data: "b" Key: "a", Data: "b" Key: "a", Data: "b" Key: "a", Data: "b" Key: "a", Data: "b"	Key: "a", Data: "b"	"The total number of collisions in the hash table has been reached."	"The total number of collisions in the hash table has been reached."

'display' Function

This function will display all the key strings and data strings that the user has inputted into the hash table. It will also display the hash index that they can be found in. This function was implemented into each question so that I could monitor whether each data point is being correctly stored in the hash table.

The function prints out the key string and data point using two 'for' loops which go through each row and column in the hash table respectively. If a certain position in the hash table was not initialized by the user and is empty, then it will not be displayed. This is ensured through the use of an 'if' statement which checks whether the first character of the key string in a certain position of the hash table is equivalent to null. If it is equivalent to null, the test condition is false and the code resumes as the current data point is empty and does not needs to be displayed. If it is not equivalent to null, the test condition is true, this signifies that a key string and data string has been inputted by the user and both strings are displayed as well as their hash index.

```
//function the prints values for hash table
void display() {
   printf("INDEX\tKEY\tDATA\n");
   for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
         //only prints values that were previously inputted (not NULL)
         if (TABLE[i][j].key[0] != '\0') {
            printf("%d\t%s\t%s\n", i, TABLE[i][j].key, TABLE[i][j].data);
         }
      }
   }
}
```

Below is an example of how the data points are displayed in the text console using the 'display' function based on the contents previously inputted into the hash table.

Previous Contents of hash table	Output
Key: "test", Data: "test" Key: "test2", Data: "test2" Key: "key", Data: "data" Key: "example", Data: "example2" Key: "key3", Data: "data3"	INDEX KEY DATA 0 test test 0 test2 test2 0 key data 4 example example2 4 key3 data3

'search' Function

The search function searches for the user inputted key string and data string in the hash table. The arguments of the function are identical to the arguments in the 'insert' function that were described previously. At the start of the function, a Boolean variable called "found" is initialized which Is set to true when the key string and data string are located in the hash table. The hash index of the users inputted string and data string is obtained using the 'get_index' function and stored in the variable 'hash_index'. This allows the program to narrow down the search by solely searching for the strings within their expected hash index.

A 'for' loop is used to check each collision in the calculated hash index. The 'key' and 'data' in the examined collision is compared to the users inputted key string and data string using the 'strstr()'

function which will return the value null when the strings are not identical. If the users inputted key string and data string match the ones found in the collision, the test condition will be set to true. Subsequently the Boolean variable "found" is set to true and a message is displayed to the user which shows the index in which the key and string can be found. If the test condition is set to false, the Boolean variable 'found' will remain false, and the for loop will resume until each collision in the hash index is checked. If all the collisions are tested, and the users inputted key string and data string are not located, the user will be notified that the data was not located in the hash table. This is done using an 'if' statement which checks whether the Boolean variable 'found' remained false after the termination of the 'for' loop.

```
//function that searches for certain elements in the array
void search (char *key, char *data, size_t key_len, size_t data_len) {
   //search if element is in array, true if found
   bool found = false;

   //gets index from key and data
   int hash_index = get_index(key,data,key_len,data_len);

   for(int i = 0; i < SIZE; i++) {
      //checks if key and data are present in the array
      if((strstr(TABLE[hash_index][i].key,key) != '\0') && (strstr(TABLE[hash_index][i].data,data) != '\0')) {
         found = true;
         printf("Data can be found in index %d.\n",hash_index);
         break;
      } else {
         found = false;
      }
   }
   if (found == false) {
      printf("Data cannot be found.\n");
   }
```

Below are the expected and actual outcomes of the 'search' function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b	"Data cannot be found."	"Data cannot be found."
Key: "a", Data: "b"	Key: "a", Data: "b"	"Data can be found in index 0."	"Data can be found in index X." *(where X reprents the value of 'hash_index' in the code)*

'delete' Function

The 'delete' function searches for the user inputted key string and data string in the hash table and then deletes both strings if found. The arguments of the function are identical to the arguments in the 'insert' Function and 'search' function that were described previously. A Boolean variable called 'found' is initialized which set to true when the key string and data string are located in the hash table. The hash index of the users inputted string and data string is obtained using the 'get_index'

function. Next the program will search whether the strings are present at any of the collisions in the hash index using the same methods described in the ‘search’ Function.

If the users inputted strings are located within a particular collision, the contents of the collision is replaced with the contents of the successive collision using a ‘for’ loop which copies each character from the succeeding key string and data string to the current one. This process is repeated for all the succeeding collisions. In the end, the collision where the users inputted key string and data strings were present are entirely replaced but the other data points remain the same. The only the difference is that the succeeding collisions have moved down one space in the row. The first character of the key and data strings in the last collision are then set to null. A message is then displayed to the user that the data has been successfully deleted. If the users inputted strings were not located in the hash table, the user will be notified that data could not be found.

```
//function that searches for certain elements in the array then deletes them
void delete (char *key, char *data, size_t key_len, size_t data_len) {
   //search if element is in array, true if found
   bool found = false;

   //gets index from key and data
   int hash_index = get_index(key,data,key_len,data_len);

   for (int i = 0; i < SIZE; i++) {
      //checks if key and data are present in array
      if ((strstr(TABLE[hash_index][i].key, key) != '\0') && (strstr(TABLE[hash_index][i].data, data) != '\0')) {
         //if present, sets found to true
         found = true;

         //replaces the data and increments the array by one
         for (int j = i; j < SIZE; j++) {
            //copies the strings
            for (int k = 0; k < MAX; k++) {
               TABLE[hash_index][j].key[k] = TABLE[hash_index][j+1].key[k];
               TABLE[hash_index][j].data[k] = TABLE[hash_index][j+1].data[k];
            }
            //setting the last elements in the array to 0
            TABLE[hash_index][SIZE].key[0] = '\0';
            TABLE[hash_index][SIZE].data[0] = '\0';
         }
         printf("Data successfully deleted.\n");
         break;
      } else {
         //false if there was not match in the array
         found = false;
      }
   }

if (found == false) {
      printf("Data cannot be found.\n");
   }
```

Below are the expected and actual outcomes of the 'delete' function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b	"Data cannot be found."	"Data cannot be found."
Key: "a", Data: "b"	Key: "a", Data: "b"	Key string "a" and Data string "b" are deleted from index 0 and all collisions in the index are now empty.	Key string "a" and Data string "b" are deleted from the index and all collisions in the index are now empty.
Key: "a", Data: "b" Key: "test", Data: "test"	Key: "a", Data: "b"	Key string "a" and Data string "b" are deleted from index 0 and Key string "test" and Data string "test" are now stored in collision 0 instead.	Key string "a" and Data string "b" are deleted from the index. Key string "test" and Data string "test" are stored in collision 0. *(Assuming both data points are stored in the same index).*

'save' Function

This function displays the contents of the hash table in a text file. A pointer to a file is declared which is then opened with the name "hash_table_a.txt" for writing using the mode "w". An 'if' statement is used to print an error message and exit the program in case the file did not open.

```
//function that saves to output file
void save() {
   FILE *f;
   printf("Saving...\n");
   //opening a file of writing type
   f = fopen("hash_table_a.txt", "w");
   //executes if there is problem opening the file
   if (f == NULL)
   {
      printf("Error opening file.\n");
      exit(EXIT_FAILURE);
   }

   //copies the elements of the array into a text file
   for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
         //if null is present in the string or key 'Null' will be displayed
         if (TABLE[i][j].key[0] == '\0' || TABLE[i][j].data[0] == '\0' ) {
            fprintf(f, "Null\n");
         } else {
            fprintf(f, "%s, ", TABLE[i][j].key);
            fprintf(f, "%s\n", TABLE[i][j].data);
         }
      }
   }
   printf("Saved.\n");
   fclose(f);
}
```

Next, each element of the hash table array is printed in the file using two 'for' loops which go through each row and column respectively in the 2D array. An 'if' statement is used to check whether an element of the hash table is empty by testing whether the first character of the key

string or data string is equivalent to null. If this test condition is true, the word “Null” will be inputted into the text file. If the test condition is false, the key string and data string will be printed one after another in a line in the text file. At the end of the function, the user is notified that the hash table has been saved to the text file and the file is then closed.

Below is an example of the contents of the text file using the ‘save’ function based on the contents previously inputted into the hash table.

Previous Contents of hash table	Output (In text file)
Key: “a”, Data: “b” Key: “c”, Data: “d” Key: “e”, Data: “f”	a, b Null Null Null Null Null Null Null Null Null Null Null Null Null Null e, f Null Null Null Null c, d Null Null Null Null

Important remarks

An important remark is that there are 25 lines printed in the text file. The number of lines represent the total number of elements in the hash table that were defined earlier. The first five lines represent the 5 collisions in index 0 while the next 5 lines represent the 5 collisions in index 1 and so on. From the figure, the key “a” and “b” exhibit that they are stored in Index 0, collision 0. While the key “e” and “f” are stored in index 3, collision 0. It is important to note that the key is separated from the data using a comma. I felt like this format was simple and could potentially be interpreted by the program if it was modified to interpret the contents of the text file and store it in a hash table.

It is important to note that this format was only used in question 2a as the boundaries of hash table were predefined and the following questions will have hash tables consisting of an infinite number of collisions. Thus, this format was only created specific for this question. An alternative approach that could have been taken is to print out the index number and collision number of each element as done for question 2b and 2c.

Question 2b

At the start of the code, the function declarations have the same format as the ones that were declared in question 2a. The macro “SIZE” represents the total number of characters that the user can input into the key string and data string of each element in the 2d array.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 100

void option();
int get_index(char *key, char *data, size_t key_len, size_t data_len);
void insert(char *key, char *data, size_t key_len, size_t data_len);
void display();
void search(char *key, char *data, size_t key_len, size_t data_len);
void delete(char *key, char *data, size_t key_len, size_t data_len);
void save();
```

For this question, a new struct is created which holds a pointer variable called “col” of type ‘struct point’. This pointer will point to the address of where the data of the first collision of each hash index in the hash table. It will later be used as an array so that the data in each collision as well in a specific hash index can be accessed. An integer variable ‘count’ is declared to keep track of the number of collisions in each hash index. As mentioned in the previous question, the struct ‘point’ will be used at each position in the hash table so that the users key string and data string input can be stored.

```
//struct representing the actual data points
struct point{
    char key[MAX]; //stores the users inputted key string
    char data[MAX]; //stores the users inputted data string
};

struct hash{
    struct point *col; //points to the position of the first collision in the index
    int count; //keeps track of how many data points there are in each index
};
```

In this question, the variable ‘SIZE’ is initialized to the value 0 and will later be changed by the user into a value that will represent the number of rows the hash table will contain. The hash table itself is declared as a pointer variable with the type ‘struct hash’. It will later be treated as an array with the array size of variable ‘SIZE’ so that each hash index can be accessed.

```
//initializing hash_table and SIZE outside the array so its applicable in all functions
int SIZE = 0;
struct hash *hash_table;
```

In the main function, the user is asked to input a number which will be stored in the variable ‘SIZE’. Subsequently, memory is allocated for the hash table for a number of “SIZE” elements. Thus, if the user inputs the value 5, the hash table will consist of 5 elements. In terms of the structure of a hash table, this number represents the number of rows, or indexes. The function ‘calloc()’ is used instead of ‘malloc()’ so that the allocated memory is set to 0 so that elements in the hash table are not randomly defined. Please note that although the number of rows is determined by the user’s inputs, for all examples of each function I will assume that the value of ‘SIZE’ is 5.

```
printf("Please enter the size of hash table.\n");
scanf("%d", &SIZE);
//allocating memory for a number of 'SIZE' elements for the pointer 'hash_table'
hash_table = (struct hash *) calloc(SIZE, sizeof(struct hash));
```

The remaining code of the main function is identical to the code described in Question 2a. Please refer to pages 23-25 for a detailed description of the rest of code in the main function. In this question, the code in the functions 'option' and 'get_index' are also identical in Question 2a. Please refer to pages 25-26 for a detailed description of the functions 'get_index' and 'option'. The contents of the other functions are different for each of the hash table questions and will now be described.

'insert' function

At the start of the function the hash index is calculated using the user inputted key string and data string. Next, a variable called "temp" is declared which will temporarily store the users key string and data string. It is of type 'struct point' which will allow the temporary point to be copied into an actual element of the hash table array once an available position is found. The users inputted key string and data string are copied into the temporary point using the 'strcpy()' function.

```
//function that enters values into the array
void insert(char *key, char *data, size_t key_len, size_t data_len)
{
   //calculates the hash_index using a function
   int hash_index = get_index(key,data,key_len,data_len);
   int i = 0;

   //creating a temporary data point called which will hold the key and data
   struct point temp;
   //copying the key and data into the temporary data point
   strcpy(temp.key, key);
   strcpy(temp.data, data);

   //checks if there are 0 collisions in the hash index
   if(hash_table[hash_index].count == 0)
   {
      //allocates memory for the first data point in the particular hash_index
      hash_table[hash_index].col = malloc(sizeof(struct point));
      //the first element in the hash_index is copied from the temporary node
      hash_table[hash_index].col[0] = temp;
      hash_table[hash_index].count = 1;
   } else {
      //executes if there were previous data points in the hash_index
      i = hash_table[hash_index].count;
      //i is used to deduce how much memory is to be reallocated to fit in an additional data point
      hash_table[hash_index].col = realloc(hash_table[hash_index].col, sizeof(struct point)*(i+1));
      //stores the information from the temporary node into the new data point(collision)
      hash_table[hash_index].col[i] = temp;
      hash_table[hash_index].count++;

   }
}
```

Next, an 'if' statement is used to check whether the 'count' variable in the hash index is equivalent to 0. If this condition is true, this means that there are 0 collisions in the current hash index, thus, the key string and data sting can be stored in the first collision of the hash index. This is done by allocating memory for the 'col' variable in the hash index which is considered the array of collisions

in the hash index. The size of the memory block is of size 'struct point' which is obtained using the function 'sizeof()'. This allows one collision to be stored in the 'col' array. The first element in the 'col' array is then set to the variable 'temp' which means that the users inputted key string and data string are now stored in the first collision of the calculated hash index. The count of the current hash index is then set to a value of 1 signifying that the hash index contains 1 data point.

If the 'count' variable of the hash index is not equivalent to 0, the variable will then be used to determine the collision in which the strings will be stored in. A variable 'I' is set to the value of the 'count' variable. Since there are "i" number of collisions already present in the 'col' array of the current hash index, memory is allocated for a new additional collision by multiplying the memory block size of the 'struct point' by "(i+1)" elements. This will allow an additional collision to be stored in the 'col' array. The newly allocated memory for the additional collision is then initialized to hold the data points within the variable 'temp'. This means that the users inputted strings have been stored as an additional collision in the hash index and the 'count' variable is then incremented to signify an additional data point added to the index.

Below are the expected and actual outcomes of the 'insert' function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b"	Key and String are stored in index 0 and collision 0 in the hash table.	Key and String are stored collision 0 of a particular index.
Key: "a", Data: "b"	Key: "a", Data: "b"	Key and String are stored in index 0 and collision 1 in the hash table.	Key and String are stored in collision 1 of a particular index.

'display' Function

At the start of the function, a pointer 'temp' is declared of type 'struct point'. The variable is declared as a pointer so that it could be set to the pointer variable 'col' which is also of type 'struct point'. Thus, 'temp' will be used to access each collision in a particular hash index.

```
//function the prints values for hash_table
void display() {
   //creating a temporary pointer to a data point
   struct point *temp;
   printf("INDEX\tKEY\tDATA\n");
   for (int i = 0; i < SIZE; i++) {
      //if there are 0 collisions in the index, checks the next index
      if (hash_table[i].count == 0)
         continue;

      //points to the first element in the hash_index
      temp = hash_table[i].col;
      //prints out the values in each collision of the index
      for(int j = 0; j < hash_table[i].count; j++) {
         printf("%d\t%s\t%s\n", i, temp[j].key, temp[j].data);
      }
   }
}
```

Using a ‘for’ loop, each hash index of the hash table is tested. If the hash index contains no collisions, the ‘count’ variable will have a value of 0, signifying that no data points are present. The ‘continue’ statement is used to skip to the end of the loop and check the next hash index. If the ‘count’ variable is not 0, this signifies that there are data points or a single data point present in the hash index. Thus, a ‘for’ loop is used to print out the key strings and data strings of all the collisions in the hash index.

Below is an example of how the data points are displayed in the text console using the ‘display’ function based on the contents previously inputted into the hash table.

Previous Contents of hash table	Output
Key: “test”, Data: “test” Key: “a”, Data: “b” Key: “c”, Data: “d”	INDEX KEY DATA 0 test test 0 a b 4 c d

‘search’ Function

At the start the hash index is calculated a Boolean variable called ‘found’ is initialized which will be set to true when the users inputted key strings and data strings are found in the hash table. A temporary pointer ‘temp’ is declared and then set to the pointer ‘col’ which represents the collisions of the calculated hash index in the hash table. The variable count is then initilaized to hold the total number of data points (or collisions) in the hash index.

```
//function that searches for certain elements in the array
void search(char *key, char *data, size_t key_len, size_t data_len) {
   //calculates the hash_index
   int hash_index = get_index(key,data,key_len,data_len);
   //true when values are found
   bool found = false;

   struct point *temp;
   //pointer to the first element in the index
   temp = hash_table[hash_index].col;
   //count represents the number of data points in the array
   int count = hash_table[hash_index].count;

   //checks if the index has 0 collisions or not
   if(hash_table[hash_index].count == 0) {
      //data not found if there are no collisions to search
      found = false;
   } else {
      //loops until the entire index is searched
      for (int i = 0; i < count; i++) {
         //checks if key and data are present in the current collision
         if((strstr(temp[i].key, key) != NULL) && (strstr(temp[i].data, data) != NULL)) {
            found = true;
            break;
         }
      }
   }
   //displays whether the values were found our not
   if(found == false) {
      printf("Data not found.\n");
   } else {
      printf("Data can be found in index %d.\n",hash_index);
   }
```

Using an ‘if’ statement, the program checks whether the ‘count’ variable is equivalent to 0, signifying that there are no data points in the index. If this condition is true, the Boolean variable ‘found’ remains to false which will signals the program to notify the user that the data could not be found. If the ‘count’ variable is not equivalent to 0, a ‘for’ loop is used to access each individual collision in the index. Inside the loop, the strings in each collision are compared to the users inputted key strings and data strings. If they are the same, a message is displayed to the user that the strings can be found in a certain index. If no strings in each collision of the index match the users inputted strings, then the Boolean variable ‘found’ remains false and the user is notified at the end of the function that the data is not found.

Below are the expected and actual outcomes of the ‘search’ Function depending on the input of the key and string as well as the contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: “a”, Data: “b	“Data not found.”	“Data not found.”
Key: “a”, Data: “b”	Key: “a”, Data: “b”	“Data can be found in index 0”.	“Data can be found in index 0”.
Key: “a”, Data: “b”	Key: “Test”, Data: “Test”	“Data not found.”	“Data not found.”

‘delete’ Function

Please refer to the code below for the proceeding description of the function.

```
void delete(char *key, char *data, size_t key_len, size_t data_len){
  //calculates hash_index
  int hash_index = get_index(key,data,key_len,data_len);
  bool found = false; //true when data points are found in array

  struct point *temp;
  //points to the first element in the index
  temp = hash_table[hash_index].col;
  //checks if there are 0 collisions in the hash index
  if(hash_table[hash_index].count == 0) {
    found = false;
  } else {
    //goes through each data-point in the index
    for (int i = 0; i < hash_table[hash_index].count; i++) {
      //executes if the key and data matches in the collision
      if((strstr(temp[i].key, key) != NULL) && (strstr(temp[i].data, data) != NULL)) {
        found = true;
        for (int j = i; j < hash_table[hash_index].count; j++) {
          temp[j] = temp[j+1];
        }
        //count decreases as one data point is removed
        (hash_table[hash_index].count)--;
        //reallocate the memory to adjust to one less node
        hash_table[hash_index].col = realloc(hash_table[hash_index].col, sizeof(struct point)*(hash_table[hash_index].count));
        break;
      }
    }
  }
  if(found == false) {
    printf("Data not found.\n");
  }
}
```

At the start the hash index is calculated and a Boolean variable called “found” is initialized which will be set to true when the users inputted key strings and data strings are found in the hash table. A temporary pointer ‘temp’ is declared and assigned the pointer ‘col’ which represents the collisions of the calculated hash index in the hash table.

The users inputted strings are then searched for in the hash table using the same method mentioned in the ‘search’ function described previously. If the strings are found, the collision holding the two strings is replaced with the proceeding one. This process is repeated for all successive collisions. The ‘count’ variable in the index is then reduced by a value of 1 since the collision containing the two strings has been entirely replaced, and the position of the succeeding collisions were adjusted. Since one collision has been removed, memory is reallocated in the ‘col’ array using the function ‘realloc()’ to reserve memory for one less element.

Below are the expected and actual outcomes of the ‘delete’ function depending on the input of the key and string as well as the contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: “a”, Data: “b	“Data cannot be found.”	“Data cannot be found.”
Key: “a”, Data: “b”	Key: “a”, Data: “b”	Key string “a” and Data string “b” are deleted from index 0 and all collisions in the index are now empty.	Key string “a” and Data string “b” are deleted from the index and all collisions in the index are now empty.
Key: “a”, Data: “b” Key: “test”, Data: “test”	Key: “a”, Data: “b”	Key string “a” and Data string “b” are deleted from index 0 and Key string “test” and Data string “test” are now stored in collision 0 instead.	Key string “a” and Data string “b” are deleted from the index. Key string “test” and Data string “test” are stored in collision 0. *(Assuming both data points are stored in the same index).*

‘save’ Function

This function displays the contents of the hash table in a text file. A pointer to a file is declared which is then opened with the name “hash_table_b.txt” for writing using the mode “w”. An if statement is used to print an error message and exit the program in case the file did not open.

Subsequently, each occupied element of the hash table is printed out using the method explained in the ‘display’ function. However, it is important to note that in this function, the output is printed out in a text file instead of the text console. The collision of each element is also printed. The user is then notified that the contents have been saved and the file is closed.

```c
void save() {
   FILE *f;
   printf("Saving...\n");
   //opening a file of writing type
   f = fopen("hash_table_b.txt", "w");
   //executes if there is problem opening the file
   if (f == NULL)
   {
      printf("Error opening file.\n");
      exit(EXIT_FAILURE);
   }

   struct point *temp;

   for (int i = 0; i < SIZE; i++) {
      //if there are 0 collisions in the index, check next index
      if (hash_table[i].count == 0)
         continue;

      //points to the first element in the hash_index
      temp = hash_table[i].col;

      //prints out the values in each collision in the index
      for(int j = 0; j < hash_table[i].count; j++) {
         fprintf(f,"Index:%d\tCollision:%d\tKey:%s\tData:%s\n", i, j, temp[j].key, temp[j].data);
      }
   }
   printf("Saved.\n");
   fclose(f);
}
```

Below is an example of the contents of the text file using the 'save' function based on the contents previously inputted into the hash table.

Previous Contents of hash table	Output (In text file)
Key: "a", Data: "b" Key: "test", Data: "test" Key: "c", Data: "d"	Index:0 Collision:0 Key:a Data:b Index:0 Collision:1 Key:test Data:test Index:4 Collision:0 Key:c Data:d

Question 2c

In this question, an additional function was declared called “create” which will point to a node that will temporarily store the users inputted key string and data string before it is placed into the hash table.

```
void option();
int get_index(char *key, char *data, size_t key_len, size_t data_len);
void insert(char *key, char *data, size_t key_len, size_t data_len);
struct node *create(char *key, char *data);
void display();
void search(char *key, char *data, size_t key_len, size_t data_len);
void delete(char *key, char *data, size_t key_len, size_t data_len);
void save();
```

The struct ‘node’ is defined with the same variables as the struct ‘point’ in the previous question, however, in this question another pointer is defined of type ‘struct node’. This will point to the address of the next node. The struct ‘hash’ has a variable ‘count’ which will keep track of the number of nodes in each index. A pointer called “head” is also declared with type ‘struct node’ which will point to the address of the first node in the index.

```
//actual data will be stored here
struct node
{
   char data[MAX];
   char key[MAX];
   struct node *next; //points to the next node
};

//point in each index
struct hash
{
   struct node *head; //position of the first node in the list
   int count; //keeps track of the number of collisions
};
```

The pointer ‘hash_table’ is declared outside the main along with the variable ‘SIZE’ which will represent the number of elements in the ‘hash_table’. They are declared outside of the main function as global variables so that they can be accessed in every function.

```
//declaring it outside main so it can be used in all functions
struct hash *hash_table;
int SIZE = 0;
```

In the main function, the user is asked to input a number which will determine the number of indexes in the main function. This number will be stored in the variable ‘SIZE’. Subsequently, memory is allocated for the hash table for a number of “SIZE” elements.

```
//allocating memory for a number of 'SIZE' elements for the pointer 'hash_table'
printf("Please enter the size of hash table.\n");
scanf("%d", &SIZE);
hash_table = (struct hash *) calloc(SIZE, sizeof(struct hash));
```

The remaining code of the main function is identical to the code described in Question 2a. Please refer to pages 23-25 for a detailed description of the rest of code in the main function. In this question, the code in the functions 'option' and 'get_index' are also identical in Question 2a. Please refer to pages 25-26 for a detailed description of the functions 'get_index' and 'option'. The contents of the other functions are different for each of the hash table questions and will now be described. Please note that although the number of rows is determined by the user's inputs, for all examples of each function I will assume that the value of "SIZE" is 5.

'insert' function

First, the hash index of the users inputted key string and data string is calculated. A new node is created using the 'create' function. The function allocates a block of memory of size 'struct node' so that the key string and data string can be stored within this new node using the 'strcpy()' function. The 'next' pointer is set to null and can later point to another node depending where the node will be placed in the hash index. It then returns the address of this node back to the 'insert' function. The variable "new node" will now point to the address of the node in which the strings are stored.

```
struct node *create(char *key, char *data)
{
   struct node *newnode;
   //creates a new node
   newnode = (struct node *) malloc(sizeof(struct node));
   //stores value in the new node
   strcpy(newnode->key, key);
   strcpy(newnode->data, data);
   //sets the next node to null
   newnode->next = NULL;
   return newnode;
}
```

```
void insert(char *key, char *data, size_t key_len, size_t data_len)
{
   int hash_index = get_index(key,data,key_len,data_len);

   //points to a new node where the key string and data string is stored
   struct node *newnode = create(key,data);

   //checks if there are 0 collisions in index
   if (hash_table[hash_index].count == 0)
   {
      //sets the head of the index to the new node
      hash_table[hash_index].head = newnode;
      hash_table[hash_index].count = 1;

   } else {
      //adds new node to the list
      //the next node of the new node becomes the data from the previous node
      newnode->next = (hash_table[hash_index].head);
      //updates the head of list, it becomes the new node
      hash_table[hash_index].head = newnode;
      hash_table[hash_index].count++;
   }
}
```

An ‘if’ statement is used to check whether there are nodes that have already been stored in the hash index. If the variable “count” of the hash index is equivalent to null, the head of the index will point to the new node and the “count” variable will be set to 1 indicating that there is now one node in the hash index. The pointer ‘next’ in the new node does not have to be changed as there are no succeeding nodes.

Alternatively, if the first test condition is false, the pointer ‘next’ of the new node will point to the current head of the hash index. Then, the head of the hash index will now point towards the new node indicating that the start of the index is now the new node and the succeeding node was the previous head of the index. This method inserts elements to the beginning of the list, contrary to the methods of the previous questions which appended elements to the end of the list.

Below are the expected and actual outcomes of the ‘insert’ function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: “a”, Data: “b”	New Key and String are stored in index 0 and collision 0 in the hash table.	Key and String are stored in collision 0 in a particular index.
Key: “a”, Data: “b”	Key: “a”, Data: “b”	New Key and String are stored in index 0 and collision 0 in the hash table. The previous Key and String are now stored in index 0 and collision 1.	Key and String are stored in collision 0 of a particular index. The previous Key and String are now stored collision 1 in the particular index.

‘display’ function

First a pointer is declared called “this_node”. Using a ‘for’ loop each index in the hash table is accessed. Using an ‘if’ statement the index is checked whether there are any nodes. If the variable ‘count’ is equivalent to 0, this signifies that no nodes are present in the current index and a ‘continue’ statement will be used to check the next index.

```
void display() {
   struct node *this_node;

   printf("INDEX\t\tKEY\t\tDATA\n");

   for (int i = 0; i < SIZE; i++) {
      //if there are 0 collisions in the index, check next index
      if (hash_table[i].count == 0)
         continue;

      this_node = hash_table[i].head;

      //works until reaches NULL in the next node, meaning end of node
      while (this_node != NULL) {
         printf("%d\t\t%s\t\t%s\n", i, this_node->key, this_node->data);
         this_node = this_node->next;
      }
   }
```

Next the pointer “this_node” is set to point to the head of the current index. A ‘while’ loop is used to print the index, key string and data string of each node in the index. At the end of each loop, the node is then set to next node in the index. If the next node is equivalent to null, this signifies that the last node at the end of the list has been reached which terminates the ‘while’ loop.

Below is an example of how the data points are displayed in the text console using the ‘display’ function based on the contents previously inputted into the hash table.

Previous Contents of hash table	Output
Key: “test”, Data: “test” Key: “a”, Data: “b” Key: “c”, Data: “d”	INDEX KEY DATA 0 a b 0 test test 4 c d

‘search’ function

At the start the hash index is calculated and a Boolean variable called “found” is initialized which will be set to true when the users inputted key strings and data strings are found in the hash table.

Next a pointer called “this_node” is set to point to the head of the determined hash index. An ‘if’ statement is used to check whether the hash index has 0 nodes, if this condition Is true, the Boolean variable remains false which in turn will notify the user that the Data was not found in the hash table. If there are nodes present in the index, the first test condition will fail proceeding to the ‘else’ statement.

```
void search(char *key, char *data, size_t key_len, size_t data_len) {

    int hash_index = get_index(key, data, key_len, data_len);
    bool found = false;

    struct node *this_node;
    //setting the current node to the head of the list in the index
    this_node = hash_table[hash_index].head;
    //checks if there are 0 collisions
    if (hash_table[hash_index].count == 0) {
        found = false;
    } else {
        //until the last node is checked(NULL)
        while (this_node != NULL) {
            //checks if the strings are present in the node
            if ((strstr(this_node->key, key) != NULL) && (strstr(this_node->data, data) != NULL)) {
                printf("Data is located in index %d.", hash_index);
                found = true;
                break;
            }
            this_node = this_node->next;
        }
    }
    //if data can not be found
    if (found == false)
        printf("Data not found.\n");
}
```

Next, the program will check whether the strings in each node of the index match the users inputted key string and data string. This is done through the use of a ‘while’ loop which terminates after each

node is scanned. If one of the nodes contain the users inputted strings, the user will be notified that their data can be found in the specified hash index. If they are not found in the node, the current node will point to the next node and the loop will resume until the last node is null signifying the end of the node list. If neither of the nodes in the index contain the strings, the Boolean variable will remain false and the user will be notified that the data was not found.

Below are the expected and actual outcomes of the 'search' function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b	"Data not found."	"Data not found."
Key: "a", Data: "b"	Key: "a", Data: "b"	"Data is located in index 0".	"Data is located in index 0".
Key: "a", Data: "b"	Key: "Test", Data: "Test"	"Data not found."	"Data not found."

'delete' function

At the start the hash index is calculated a Boolean variable called "found" is initialized which will be set to true when the users inputted key strings and data strings are found in the hash table.

Next a pointer called "this_node" is set to point to the head of the determined hash index. An 'if 'statement is used to check whether the hash index has 0 nodes, if this condition Is true, the Boolean variable "found" remains false which in turn will notify the user that the Data was not found in the hash table. If there are nodes in the index, the first test condition will fail proceeding to the 'else' statement.

At the start of the 'else' statement a pointer to a node called "temp" node is set to point to "this_node" which points to the head of the index. Next, the program will check whether the strings in each node of the index match the users inputted key string and data string. This is done using a 'while' loop which terminates after each node is scanned. It is important to note that the at the end of the loop, before the next node is scanned, the temporary node will be set to represent the current node in the next loop, while the current node will be set to the successive node in the next loop. In other words, the "temp" node will always represent the node before the current node at the start of each loop. If the users inputted strings match the ones found within a node, two different courses of actions can be taken to delete the node.

- If the node containing the identical is strings is located in the head of the index (the first node in the index). The head of the index is replaced with the next node.
- If the node containing the identical is strings is located in a different position in the index, the node following the "temp" node is set to the node following the "this_node" node. Then the "this_node" node is deallocated using the 'free()' function, or removed from the hash table.

At the end, the user is notified whether the data has been deleted or not found by testing the Boolean variable "found" in an 'if' statement.

```
void delete(char *key, char *data, size_t key_len, size_t data_len){
   int hash_index = get_index(key,data,key_len,data_len);
   bool found = false;
   struct node *temp, *this_node;
   //sets the position of the node to the head of the index
   this_node = hash_table[hash_index].head;

   //checks if there 0 collisions
   if (hash_table[hash_index].count == 0) {
     found = false;
   } else {
      //if data is not found in the head of the index, temp will represent the previous node
      temp = this_node;

      //loops until it checks the last node
      while (this_node != NULL) {
         //checks if the key and data matches
         if ((strstr(this_node->key, key) != NULL) && (strstr(this_node->data, data) != NULL)) {
            found = true;
            //checks if the node is at the head of the index
            if (this_node == hash_table[hash_index].head)
               //sets the head to NULL
               hash_table[hash_index].head = this_node->next;
            else
               //the following nodes of temp are set to the following nodes of next
               //this_node is no longer needed in the list
               temp->next = this_node->next;

            //reduces amount of collisions in the table
            hash_table[hash_index].count--;
            //deallocates memory of this_node
            free(this_node);
            break;
         }
         //if the key and data does not match, check the next node
         //in the next loop, temp will represent the node before this_node as this_node is assigned to the next node
         temp = this_node;
         this_node = this_node->next;
      }
   }
   if (found == true)
      printf("Data deleted.\n");
   else
      printf("Data not found.\n");
}
```

Below are the expected and actual outcomes of the 'delete' function depending on the input of the key and string as well as the previous contents of the hash table.

Previous Contents of hash table	Input	Actual Outcome	Expected Outcome
No contents	Key: "a", Data: "b	"Data not found."	"Data not found."
Key: "a", Data: "b"	Key: "a", Data: "b"	The node containing Key string "a" and Data string "b" is removed from the index.	The node containing Key string "a" and Data string "b" is removed from the index.
Key: "test", Data: "test" Key: "a", Data: "b"	Key: "a", Data: "b"	The node containing Key string "a" and Data string "b" is removed from the index. The node containing "test" and "test" is now the head of the node list in the index.	The node containing Key string "a" and Data string "b" is removed from the index. The node containing "test" and "test" is now the head of the node list in the index.

'save' function

This function prints the contents of the hash table in a text file. A pointer to a file is declared which is then opened with the name "hash_table_c.txt" for writing using the mode "w". An 'if' statement is used to print an error message and exit the program in case the file did not open.

Subsequently, each occupied element of the hash table is printed out using the method explained previously in the 'display' function of this question. However, it is important to note that in this function, the output is printed out in a text file instead of the text console. In addition, a variable 'j' has been added to represent each collision. As each node from the start of the index (or the head) is printed, the value is incremented and then resets back to 0 when the next index is being checked. The collision of each element is also printed. The user is then notified that the contents have been saved and the file is closed.

```
void save(){
   FILE *f;
   printf("Saving...\n");
   //opening a file of writing type
   f = fopen("hash_table_c.txt", "w");
   //executes if there is problem opening the file
   if (f == NULL){
      printf("Error opening file.\n");
      exit(EXIT_FAILURE);
   }

   struct node *this_node;

   for (int i = 0; i < SIZE; i++) {
      //if there are 0 collisions in the index, check next index
      if (hash_table[i].count == 0)
         continue;

      this_node = hash_table[i].head;
      int j = 0;
      //loops until reaches NULL in the next node, meaning the last node has been checked
      while (this_node != NULL) {
         fprintf(f,"Index:%d\tCollision:%d\tKey:%s\tData:%s\n", i, j, this_node->key, this_node->data);
         this_node = this_node->next;
         j++;
      }
   }
   printf("Saved.\n");
   fclose(f);
}
```

Below is an example of the contents of the text file after using the 'save' function based on the contents previously inputted into the hash table. It is important to note how the new strings are stored in the start of the index rather than the end of the index like in the previous questions.

Previous Contents of hash table	Output (In text file)
Key: "a", Data: "b" Key: "test", Data: "test" Key: "c", Data: "d"	Index:0 Collision:0 Key:test Data:test Index:0 Collision:1 Key:a Data:b Index:4 Collision:0 Key:c Data:d