

CPS 1011 Assignment

DOCUMENTATION

Tristan Oa Galea

Table of Contents

Task 1a.....	pg 2
Task 1b.....	pg 11
Task 1c.....	pg 19
Task 1c Verification.....	pg 33
Task 1 Testing.....	pg 40
Task 2a.....	pg 49
Task 2b.....	pg 81
Testing 2.....	pg104
Testing 2a.....	pg105
Testing 2b.....	pg116
Task 2c.....	pg137

Task 1a

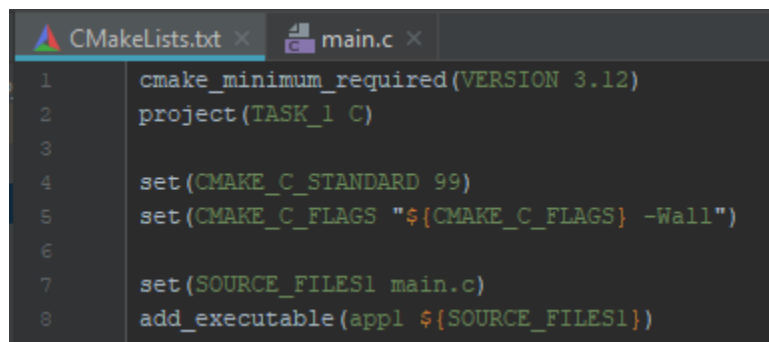
Task Specification:

In task 1a, a 3D array with dimensions as entered by the user needed to be created. The user would then be asked to populate this newly created array. The final step would be to create a function to simply copy all the contents that the user entered in the original 3D array onto a newly created empty 3D array.

Task Creation:

A New Project was created in CLion entitled Task_1 which consisted of a CMakeLists.txt file and a main.c source file.

The CMakeLists.txt file was the first to be modified. Its contents were standardized to match the cross-platform build system while also including the main.c file to be recognized as a source file. The modifications done can be seen below:

A screenshot of the CLion IDE showing the CMakeLists.txt file. The file contains the following code:

```
1 cmake_minimum_required(VERSION 3.12)
2 project(TASK_1 C)
3
4 set(CMAKE_C_STANDARD 99)
5 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall")
6
7 set(SOURCE_FILES1 main.c)
8 add_executable(appl ${SOURCE_FILES1})
```

The main.c file was now the only file required to be changed. To start off, the stdio.h library (which defines all standard input and output operations required to be used) was included at the start of the program

```
/* Including the stdio.h library which is C's Standard Input and Output */
/* Library comprising of basic functions */
#include <stdio.h>
```

The remainder of the code for task 1a was then written in the main function of the program. Three variables were declared first, each to store the 3 dimensions of the 3D array. The user was first displayed with a welcome prompt and then asked to enter the 3 dimensions required to create an empty 3D array.

```
/* The main function is the entry point of program. The program starts executing */
/* from here. This function returns an integer value of 0 if executed correctly */
int main(void) {

    /* Initialising the 3 dimensions of the array to be created as integers */
    int size1, size2, size3;
    printf("In this program a 3D array with dimensions of your choice will be created.\n");
    printf("Let's get started.\n\n");

    printf("Please enter the first dimension : ");
    //storing the value entered by the user to size1
    scanf(" %d", &size1);

    printf("Please enter the second dimension : ");
    //storing the value entered by the user to size2
    scanf(" %d", &size2);

    printf("Please enter the third dimension : ");
    //storing the value entered by the user to size3
    scanf(" %d", &size3);

    /* Program returns 0 just before finishing execution to indicate if it was */
    /* successfully implemented */
    return 0;
}
```

The method of simply using scanf to store whatever the user enters is risky as in this case, the program expects the user to input an integer value. It is for this reason that as this program progresses, validation checks are implemented to ensure that the program works as expected and does not crash.

Now that we are able to store the dimensions that the user entered, these values can be used to create the 3D array. Thus, a 3D array can now be initialized. The reason why the array is being initialized at this point in the program is because the variables size1, size2 and size3 all have a value. Initializing such an array before knowing these variables would give an error as C operators from top to bottom and would not know what variables might be coming next until those variables are actually reached. Given that we know the dimensions of the 3D original array, it would be good time to initialize the other empty copy array to be used later on.

```
/* Initialising an empty 3D array based on the dimensions entered*/  
  
/* by the user */  
  
int array3d[size1][size2][size3];  
  
  
/* Another empty array is initialised with the same dimensions */  
  
int copyArray[size1][size2][size3];
```

Now that array3d is initialized, it needs to be filled up by the user. This is done by implementing the concept of nested for loops to be able to store each individual element. A separate function was created which had to be declared as a function prototype at the top of the source file. This function contains 4 parameters, the 3 dimension sizes and the actual array itself as the final parameter. This specific order was important as the sizes were known to the array once they were passed as parameters. It was also important to stick to the order of parameters being passed through the function when calling it. Passing the array as the first parameter would have caused a major error in the program, causing it not to work because the parameter being passed would not match the prototype. The code below represents the function prototype being used to define the function we have been speaking about.

```
/* A void function which prompts the user to populate the created 3D array based */
/* on the previously inputted dimensions. To further aid the user, coordinates */
/* for the {block, row, column} are shown with every prompt for data to ensure */
/* that the user enters the correct data in the desired position. Data validation */
/* is also implemented to ensure that the user stores int values in the array. */
/* Parameters are passed through the function for the 3 dimensions of the array */
/* (s1,s2 and s3) followed by the array itself. Being a void function, nothing */
/* is returned back to the main function */
void insertContents(int s1, int s2, int s3, int ar[s1][s2][s3]);
```

The function `insertContents()`, being defined by the prototype above is currently made up of 3 for loops to loop through each single block, row and column the user has entered when defining the 3D array. The user is prompted to enter a value (being given the coordinates) that would essentially be stored in that particular position in memory and the counter pointing to that memory location would increment each time allowing the user to fill up the whole array.

```
/* When insertContents() function is called, execution continues from here */
void insertContents(int s1, int s2, int s3, int ar[s1][s2][s3]){

    /* Initialising these local variables to be used in the for loops */
    int block, row, col;

    printf("The array contents need to be inserted next.\n");

    /* For loops loop through the empty array to allow the user to fill up the array*/
```

```
for(block = 0 ; block < s1 ; block++){  
    for(row = 0 ; row < s2 ; row++) {  
        for (col = 0; col < s3; col++) {  
  
            /* Displays the coordinates of where the user will be storing      */  
            printf("Enter the contents for {%d, %d, %d}: \n", block + 1, row + 1, col + 1);  
        }  
    }  
}  
  
/* Nothing is returned back to main because this is a void function      */  
}
```

The final step would be to call the insertContents() function from the main function right after initializing the 2 arrays. For this function to work, the proper parameters matching the ones defined need to be inserted. The parameters inserted in this case would be the 3 dimensions entered by the user and the original 3D array created itself as shown:

```
/* Calls the insertContents() function while passing the 3      */  
/* dimension parameters as well as the original 3D array itself */  
  
insertContents(size1, size2, size3, array3d);
```

Once the user populates the array he created, another function needs to be called to specifically copy all the contents of one array onto another empty array(which has already been initialized with the appropriate dimensions). This function, called the copyContents() function, makes use of a very similar concept used in the previous function. It also makes use of the same local variables and nested for loops to loop through each empty element and assign it the same value found in that specific location in the original 3darray as can be seen in the snippet:

```
/* When copyContents() function is called, execution continues from here */
void copyContents(int s1, int s2, int s3, int ar[s1][s2][s3], int copy[s1][s2][s3]){

    /* Initialising these local variables to be used in the for loops */
    int block, row, col;

    /* For loops loop through the original array to copy its contents onto the */
    /* copy array */
    for(block = 0 ; block < s1 ; block++){
        for(row = 0 ; row < s2 ; row++){
            for(col = 0 ; col < s3 ; col++){

                /* Copying the contents of the original onto the copy array */
                copy[block][row][col] = ar[block][row][col];
            }
        }
    }

    /* Nothing is returned back to main because this is a void function */
}
```

As can be seen above, the value at `copy[block][row][col]` is assigned the same value of `ar[block][row][col]`. For this to be possible, the three dimensions need to be passed as parameters together with both arrays. The function prototype was also added at the top of the file right under the `insertContents()` function prototype.


```

/* A void function which simply copies the contents of the 3D array created by      */
/* the user to a new array with the same dimensions. Five arguments are passed    */
/* this time with the first 3 being the array dimensions s1, s2 and s3 and the    */
/* remaining 2 being the original array and the newly created copy array. Being   */
/* a void function, nothing is returned back to the main function                 */
void copyContents(int s1, int s2, int s3, int ar[s1][s2][s3], int copy[s1][s2][s3]);

```

This function was then called from the main function right after calling the insertContents() function.

```

/* Calls the copyContents() function while passing once again the                */
/* 3 dimensions as arguments together with the original array and                 */
/* the copy array too                                                             */
copyContents(size1, size2, size3, array3d, copyArray);

```

By this point, task 1a is complete. However it would be pointless to create 2 functions to insert elements to a 3D array created by the user and the other to copy the array onto another empty array without even being able to display or view the newly copied array. Thus it was decided to create another function to show the contents of this copied array. This function was given the name pasteContents() and had 4 arguments passed through it, the 3 dimensions of the array and the copy array itself. It's function prototype can be seen below:

```

/* A void function which displays the final version of the newly copied array.    */
/* This contains the 3 array dimensions passed as arguments as well as the copy   */
/* array itself. Being a void function, nothing is returned back to the main      */
/* function                                                                        */
void pasteContents(int s1, int s2, int s3, int copy[s1][s2][s3]);

```

This consisted of a very similar concept when compared to the other 2 functions. It also made use of the same local variables and 3 for loops were used to loop

through each individual element and print it. The blocks and columns were separated by a single '\n' whilst the rows were separated by 2 '\n'.

```
/* When pasteContents() function is called, execution continues from here */
void pasteContents(int s1, int s2, int s3, int copy[s1][s2][s3]) {

    /* Initialising these local variables to be used in the for loops */
    int block, row, col;

    /* For loops loop through the copy array to display its contents */
    for (block = 0; block < s1; block++) {
        for (row = 0; row < s2; row++) {
            for (col = 0; col < s3; col++) {

                /* Displaying the contents of the copy array */
                printf("%4d", copy[block][row][col]);

            }

            /* Puts the escape character \n onto the screen */
            putchar('\n');

        }

        /* Puts the escape characters \n onto the screen */
        putchar('\n');
        putchar('\n');

    }

    /* Puts the escape character \n onto the screen */
    putchar('\n');

    /* Nothing is returned back to main because this is a void function */
}
```

This function was then called as the last process before the *return 0;* of the main function. Being a void function, just like the other 2, it did not return anything upon being called but simply printed out the elements copied off the original array in an ordered fashion

```
/* Calls the pasteContents() function which has the 3 dimensions */  
/* and the newly copied array passed as arguments */  
pasteContents(size1, size2, size3, copyArray);
```

Task 1b

Task Specification:

The aim of task 1b is to make use of a function which reverses whole words of a string entered by a user and passed as a parameter to this function.

Task Creation:

This task was done in the same main.c file task a was done in. First and foremost, the whole chunk of characters and functions used for task 1a in the main function were commented out so as to not interfere with the newly entered code for task 1b. These commented lines were then uncommented and used again in task 1c further on.

For this particular task, another 2 libraries needed to be included at the top of the *main.c* file apart from the `<stdio.h>` library. These included the `string.h` library which defines several string related functions and the `ctype.h` library which contains functions to help classify characters.

```
/* Including the string.h library which defines several functions that */
/* manipulate C strings and arrays */
#include <string.h>

/* Including the ctype.h library which contains a set of functions to */
/* classify and even transform individual characters */
#include <ctype.h>
```

Next, the user was displayed with a welcome message and was given a brief explanation of the program's purpose through `printf`. Then the user was asked to enter a string of his choice comprising of not more than 100 characters, this time through `puts`. The main difference between these 2 functions is that upon printing out what has been placed in between parenthesis and quotation marks, the `puts` function automatically prints out a `'\n'` to the user.

```
printf("In this program a string of characters entered will have \n");  
    printf("whole individual words reversed.\n\n");  
  
/* Using the puts function instead of printf() which prints out a */  
/* new line to the screen automatically */  
puts("To get started, enter a string of not more than 100 characters:");
```

Moving on, a string to store the user's input was required to be created. It was decided that the string could have a maximum of 100 characters as already explained therefore, a constant `SIZE` was defined as a macro right under the library inclusions. This was set to 100.

```
/* Defining the macro SIZE to 100 to be used as a constant for the array size */  
#define SIZE 100
```

Once the constant `SIZE` was given a definite value, it could be called anywhere throughout the program and even in functions without the need of passing it as a parameter. Thus, the array was then created to receive user input:

```
/* Creating a string array with a capacity of 100 characters to store user */  
/* input from choice 2 */  
char str[SIZE];
```

After defining this array, the function `fgets` defined in the `stdio.h` library was used to store the string entered by the user. This function takes 3 parameters namely being the destination where the entered content was to be stored, the maximum amount of characters to be stored and from where to accept input. In this case, the input was stored in the string `str` of size `SIZE`. This gives us the second parameter to be used in this function. The third parameter would be to accept input as *stdin*

meaning standard input from the keyboard. This function can also accept input from different locations such as files when given specific keywords.

```
/* Accepts standard input from the keyboard up till 100 characters */  
  
/* and stores what is entered in the string str */  
  
fgets(str, SIZE, stdin);
```

The role of this function in only accepting input up till 100 characters is very vital as without it the program will still try its best to store the extra input into other memory locations which have not been set to store that particular data thus resulting in overwriting of information.

If the user ran the program up till now given that the only function of it is to accept input up till the 100th character, the program would simply run from start to end and return 0 immediately without allowing the user to input. The reason for this is because the hidden '\n' being printed out at the end of the *puts* function is automatically stored as input in the array *str[SIZE]*. The only way for this function to stop accepting input is for the user to enter a new line after having entered the characters he wanted to enter(i.e. *fgets* function reads '\n' and stops reading input). Thus, having this escape character being printed to the screen by another function caused the program to store it in the string and move on.

The solution to this unwanted input without removing the *puts* function was to make use of the *fflush* function to clear the output buffer. This would then allow the user to enter anything he wanted.

```
/* Clearing the output buffer from the \n automatically printed out */  
  
/* by the puts function. If this line of code is not inserted here, */  
  
/* the program assumes \n as input into the fgets() function */  
  
fflush(stdin);
```

The final task now that the user had entered input successfully was to implement a function to reverse the entered string word by word and not letter by letter. This function was the only function to be written from scratch in task 1b. This can be seen below:

```
/* When reverseString() function is called, execution continues from here */
void reverseString(char string[SIZE]){

    /* Initialising this local variable to be used in the for loop */
    int i;

    /* Initialising length to be the length of the entered string */
    int length = strlen(string);

    /* Changes the escape character \n to a \0 */
    if( string[length-1] == '\n' ){
        string[length-1] = '\0';
    }

    /* Loops throughout all the string of entered words */
    for(i = length-1 ; i >= 0 ; i--){

        /* Runs if a blank space is encountered */
        if( isblank(string[i]) ){

            /* Places the escape character \0 instead of a space */
            string[i] = '\0';

            /* Prints out words in reverse order except for the last word */
            printf(" %s", (&(string[i])+1) );
        }
    }
}
```

```
/* Prints out the last reversed word in the string which initially was the */  
/* the first word entered by the user */  
printf(" %s", string);  
  
/* Nothing is returned back to main because this is a void function */  
}
```

The string storing the user input is passed as a parameter through this function to be able to be analyzed and modified. This function was declared to a void function because it doesn't return anything, it simply converts the whole words and prints them out backwards. The local variable *i* was initialized to be used in the for loop whilst another variable *length* was set to store the amount of characters stored in the string *str[SIZE]*. The value for the length of the entered string was obtained by using the function *strlen(string)* which is the only function being used in the whole of task 1 to come out of the *string.h* library.

The next operation going to be discussed is the for loop through which whole words are reversed. This for loop however is not your regular for loop. To start with, this was implemented in a backwards (descending) approach. The variable *i* was set to the character before the last. The reason why this was done is to start from the actual final character and not from the escape character present as the last element in the string. The value of *i* was set to decrement each time as the loop was executed until *i* became equal to 0, the first character in the string.

```
/* Loops throughout all the string of entered words */  
for(i = length-1 ; i >= 0 ; i--){
```

The block of code contained within the for loop consisted of an if statement which ran only if a blank character space was detected. This was made possible by using the function *isblank(string[i])* declared in the *ctype.h* library which once again was the only time in the whole of task 1 in which this library was used. If a blank space was encountered, the if statement would execute the block of code within it causing the blank space to be converted into a string terminating character '\0'. Upon doing this, the last word in the sequence entered by the user was printed out to the user as the first word and this sequence was repeated on and on for the remaining words in the string.


```
/* Runs if a blank space is encountered */
    if( isblank(string[i]) ){
        /* Places the escape character \0 instead of a space */
        string[i] = '\0';

        /* Prints out words in reverse order except for the last word */
        printf(" %s", (&(string[i])+1) );

    }
```

If the if statement is documented as follows and right after it, the function returns back to the main function, 2 errors would rise as can be seen in the failure case below:

```
In this program a string of characters entered will have
whole individual words reversed.

To get started, enter a string of not more than 100 characters:
This sentence will be printed out backwards but incorrectly!
This sentence will be printed out backwards but incorrectly!
incorrectly!
but backwards out printed be will sentence
```

In the snippet above, a sentence was entered and printed out backwards. However, the first word to be outputted backwards was printed on a separate line than the rest of the words which have been reversed. The other reason why this program was not valid for outputting a string with words reversed is because the first word entered by the user (which would be the last word to be printed out) was never actually printed out to the screen.

To solve the first error, the following code was entered before the for loop in this function:

```
/* Changes the escape character \n to a \0 */
if( string[length-1] == '\n' ){
    string[length-1] = '\0';
}
```

Here another error related to the escape character '\n' is experienced. When the user was asked to enter the string of characters, after entering he hit enter to confirm that he is ready from writing it. This caused all the characters including the '\n' to be stored in the array str[SIZE]. Thus this '\n', being the last element in the string, can be accessed by string[length-1] and changed to a string terminating character. With this done, one of the errors was solved and the reversed words could now all be printed on the same line. However, the error of not printing the last reversed word still remained by this point. This was solved by including the following line of code right after the for loop:

```
/* Prints out the last reversed word in the string which initially was the */  
/* the first word entered by the user */  
printf(" %s", string);
```

This line of code caused the last reversed non-printed character to be printed out successfully. The reason why it was not being printed before was because the if statement would not loop one last time since it would not detect another empty character space and so would stop from printing out the last character. Adding this last line right after the loop causes the remaining string (the last word) to be printed out. The successful case without any failures can be seen below:

```
In this program a string of characters entered will have  
whole individual words reversed.  
  
To get started, enter a string of not more than 100 characters:  
This sentence will be printed out backwards correctly!  
This sentence will be printed out backwards correctly!  
correctly! backwards out printed be will sentence This
```

The function prototype for this function was defined at the top of the file just above the main function. This was the only function used in this program

```
/* This function is called if the user selects option 2 from the menu. It passes */  
/* an array of strings previously entered by the user as an argument. It then */  
/* switches the order of individual words by printing out the first entered word */  
/* at the end and vice versa. Being a void function, nothing is returned back to */  
/* the main function */  
void reverseString(char string[SIZE]);
```

This function was called from the main as the last part before execution finished and the main function returned 0. As can be seen, the array storing the user's input has been passed a parameter through this function to be able to reverse the string.

```
/* Calls the reverseString() function and passes str as argument */  
reverseString(str);
```

Task 1c

Task Specification:

A program consisting of a menu comprising both task 1a and task 1b was required to be written. This menu with the possible options would loop on end until the user decides to exit. This program was needed to be properly validated against incorrect user input and had to make use of function calls.

Task Creation:

The previous 2 tasks documented above were revised and it was decided that an extra function called *menu()* had to be written which would be called each time until the user decides to quit. It was also decided that this menu would return an integer which would determine based upon a switch case which functions to execute and if to return back to the main menu after execution is complete or not. The function prototype for this menu program can be seen below:

```
/* Displays a greeting message and prints out the available options to the user. */  
/* Prompts the user to enter his choice between 1-3. If invalid, the user is */  
/* informed and asked once again. If the entered int value is in the given range, */  
/* it is returned back to the main function from where the function was called */  
int menu();
```

The menu function is quite basic as can be seen:

```
/* When menu() function is called, execution continues from here */  
int menu(){  
  
    /* Initialising this local variable to store the user's choice */  
    int choice;  
  
    printf("\n\n\n\t\tWelcome to this 3D Array program!\n");
```

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

printf("Choose one of the options below: \n\n");

printf(" 1) Create a 3D array with dimensions and content of your choice and its\n");
printf("    contents will be copied onto another array of the same dimensions.\n");
printf("    A copy of the array created is then displayed.\n");

printf(" 2) Reverse the order of individual words entered.\n");

printf(" 3) QUIT\n\n");

printf("CHOICE: ");

/* If something other than an integer is entered, this while loop runs prompting*/
/* the user to enter the appropriate data type */
while(scanf("%d", &choice) != 1){

    printf("Invalid data!! \nPlease enter a number from 1 - 3 : \n");

    /* Clears input buffer from previously entered invalid data */
    while(getchar()!='\n');
}

/* Puts the escape character \n onto the screen */
putchar('\n');

/* Returns the value entered by the user back to the main() function */
return choice;
}
```

It simply consists of a local variable to store the user's choice and prints out the available options to the user. If the user enters anything else but an integer, a while loop was implemented to remind the user once more what the expected input is and take input from the user once again.

If the user enters a value which is an integer but not within the specified range, the while loop won't execute and the value will simply be returned back to the main function from where the function was called. However additional validation is performed there in which the user would be informed that the entered number was not in the specified range and the menu() function will be displayed once again asking for correct input. How this was done will be elaborated further on. But at the time being, let us take a look at how the value of choice is passed down from the menu() function to the main function given that the value was stored in a variable local to the menu() function only.

In the main function, now that both tasks 1a and 1b had to be made available to the user through a menu, some features had to be changed. An integer variable choice was declared right after the main to be able to act as a placeholder to store the integer being returned from the menu() function when calling it.

```
int main(void) {  
  
    /* Initialising the choice variable to use it as a placeholder when calling */  
    /* the menu() function */  
    int choice;  
  
    /* Calling the menu() function while storing the returned integer to the */  
    /* variable choice declared previously */  
    choice = menu();  
}
```

Up till this point, when running the program, the menu is simply called and whatever integer is entered by the user is simply returned and stored in the placeholder choice. Thus implementing a switch case would be ideal to make use of that value stored in choice. The switch case skeleton considered is shown below:

```
choice = menu();

/* Start of switch case */
switch (choice) {

    case 1 : {

        /* This piece of code runs if choice == 1 */

        /* Breaks out of the switch case */
        break;
    }

    case 2 : {

        /* This piece of code runs if choice == 2 */

        /* Breaks out of the switch case */
        break;
    }

    case 3: {
        /* This piece of code runs if choice == 3 */

        /* Breaks out of the switch case */
        break;
    }
}
```

```
        default: {

            /* Breaks out of the switch case */

            break;

        }

    }

    printf("\t--End of program--");

    /* Program returns 0 just before finishing execution to indicate if it was */
    /* successfully implemented */
    return 0;
```

The code represented above is referred to as a skeleton since it simply contains the separate possible cases which are empty up till now. Before these cases were individually filled up and arranged, an extra few lines of code had to be added to loop all the cases except for case 3. This could have been done in 2 possibly easy ways. The first one was to call the menu() function in all cases(including default) except for case 3 just before the break statement was reached. This implied that the placeholder choice also needed to be included to be able to retrieve the user's choice. Another method was to simply leave the whole skeleton as it is represented above and add a do while loop round the whole of the switch case which would loop each time unless choice != 3. The reason why a do while was chosen over a while is simply because it makes sure that the menu() runs at least once when it is called. Then based upon the user input, the menu() will run if the choice is not equal to 3.


```
/* Start of the do while loop */
do {

    /* Calling the menu() function while storing the returned integer to the */
    /* variable choice declared previously */
    choice = menu();

    /* Start of switch case */
    switch (choice) {

        //ALL OF SWITCH CASE

    }

    /*Condition for looping. If choice == 3, the program stops looping */
}while(choice != 3);

printf("\t--End of program--");

/* Program returns 0 just before finishing execution to indicate if it was */
/* successfully implemented */
return 0;
}
```

With this being done, it was now time to modify all the previously written functions to be able to fit them into the switch case. The code written in task 1a was tackled first. To keep the number of lines of code in the separate cases to a minimum, it was decided to convert the 3 dimension-asking lines of code into 3 separate functions which return an integer value. This required the creation of 3 new variables in the main just before the do while loop. These 3 variables called size1,

size2 and size3 were to be used as placeholders when calling the 3 dimension functions created called dimension1(), dimension2() and dimension3().

```
/* The following 3 functions are called if the user selects option '1' from the */
/* menu. Each option prompts the user to enter a single dimension of the 3D */
/* array being created. If invalid data is entered, the user is informed and */
/* asked to enter data once again. These functions all return an integer value */
/* back to the main function to be used to create the array */
int dimension1();
int dimension2();
int dimension3();
```

These 3 dimension functions were built upon what was already written in task 1a. They now each contained a local variable (s1, s2 and s3) which stored the user's choice and a while loop was also implemented for validation to ensure that the user enters an integer value only. The 3 functions can be seen below starting with dimension1():

```
/* When dimension1() function is called, execution continues from here */
int dimension1(){

    /* Initialising this local variable to store the user's choice */
    int s1;

    printf("Please enter the first dimension : ");

    /* If something other than an integer is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while(scanf("%d", &s1) != 1){
        printf("Incorrect input!! \nPlease input a number and press enter : \n");
    }
```

```
/* Clears input buffer from previously entered invalid data */
    while(getchar()!='\n');
}

/* Returns the value entered by the user back to the main() function */
return s1;
}
```

dimension2():

```
/* When dimension2() function is called, execution continues from here */
int dimension2(){

    /* Initialising this local variable to store the user's choice */
    int s2;

    printf("Please enter the second dimension : ");

    /* If something other than an integer is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while(scanf(" %d", &s2) != 1){
        printf("Incorrect input!! \nPlease input a number and press enter : \n");

        /* Clears input buffer from previously entered invalid data */
        while(getchar()!='\n');
    }

    /* Returns the value entered by the user back to the main() function */
    return s2;
}
```

dimension3():

```
/* When dimension3() function is called, execution continues from here */
int dimension3(){

    /* Initialising this local variable to store the user's choice */
    int s3;

    printf("Please enter the third dimension : ");

    /* If something other than an integer is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while(scanf(" %d", &s3) != 1){
        printf("Incorrect input!! \nPlease input a number and press enter : \n");

        /* Clears input buffer from previously entered invalid data */
        while(getchar()!='\n');
    }

    /* Returns the value entered by the user back to the main() function */
    return s3;
}
```

These methods were then called from the 1st case of the switch case as shown while storing the returned values into the placeholders:

```
/* This piece of code runs if choice == 1 */
case 1 : {

    printf("In this program a 3D array with dimensions of your choice will be created.\n");
    printf("Let's get started.\n\n");

    /* Calling all 3 dimension functions while storing the returned */
    /* result in the placeholders declared previously */
    size1 = dimension1();
    size2 = dimension2();
    size3 = dimension3();

    printf("All 3 dimensions were entered successfully.\n\n");
```

The remaining array initialisations and function calls were then inserted right after the above lines of code as they were before:

```
printf("All 3 dimensions were entered successfully.\n\n");

/* Initialising an empty 3D array based on the dimensions entered */
/* by the user */
int array3d[size1][size2][size3];

/* Another empty array is initialised with the same dimensions */
int copyArray[size1][size2][size3];

/* Calls the insertContents() function while passing the 3 */
/* dimension parameters as well as the original 3D array itself */
insertContents(size1, size2, size3, array3d);
```

```
/* Calls the copyContents() function while passing once again the */  
/* 3 dimensions as arguments together with the original array and */  
/* the copy array too */  
copyContents(size1, size2, size3, array3d, copyArray);  
  
/* Calls the pasteContents() function which has the 3 dimensions */  
/* and the newly copied array passed as arguments */  
pasteContents(size1, size2, size3, copyArray);  
  
/* Breaks out of the switch case back to the start of the do */  
/* while loop and if the condition for looping is valid, then the */  
/* loop runs again */  
break;  
}
```

Moving on to case 2, not many modifications were made. The array initialisation to store the the user's input string was added to remaining variables which were exactly at the start of the main function right before the do while loop.

```
/* Creating a string array with a capacity of 100 characters to store user */  
/* input from choice 2 */  
char str[SIZE];
```

The remaining lines of code written in 1b were simply moved into the case 2 block as can be seen:

```
/* This piece of code runs if choice == 2 */
case 2 :

    printf("In this program a string of characters entered will have \n");
    printf("whole individual words reversed.\n\n");

    /* Using the puts function instead of printf() which prints out a */
    /* new line to the screen automatically */
    puts("To get started, enter a string of not more than 100 characters:");

    /* Clearing the output buffer from the \n automatically printed out */
    /* by the puts function. If this line of code is not inserted here, */
    /* the program assumes \n as input into the fgets() function */
    fflush(stdin);

    /* Accepts standard input from the keyboard up till 100 characters */
    /* and stores what is entered in the string str */
    fgets(str, SIZE, stdin);

    /* Calls the reverseString() function and passes str as argument */
    reverseString(str);

    /* Breaks out of the switch case back to the start of the do */
    /* while loop and if the condition for looping is valid, then the */
    /* loop runs again */
    break;
}
```

Moving on to case 3, this simply outputs a simple “Bye!” message and breaks out of the switch case statement. Since the choice is now 3, the condition for looping is no longer valid and so the rest of the program continues with execution until the end of the main function.

```
/* This piece of code runs if choice == 3 */  
case 3: {  
    printf("Bye!\n");  
  
    /* Breaks out of the switch case back to the start of the do */  
    /* while loop and if the condition for looping is valid, then the */  
    /* loop runs again */  
    break;  
}
```

The final case is the default case which simply runs only if the user enters any other integer value apart from 1,2 or 3. Having performed validation in the menu() function when asking for the user's choice, this validation only clears the input buffer if the entered character/s are not integers. Thus, this default case is used as the final form of validation by letting the user know that his input was not in the specified range and prints out the menu to the user once again. This time, the input buffer is cleared automatically and not by entering the `while(getchar()!='\n')` line of code because the menu() function is called once more and user input is expected.


```
/* This piece of code runs if choice is anything else but 1,2 or 3 */
default: {

    printf("Incorrect input!\nMake sure you enter a valid number between 1-3 from the list below.\n");

    /* Breaks out of the switch case back to the start of the do */
    /* while loop and if the condition for looping is valid, then the */
    /* loop runs again */
    break;
}
```

With that being said, all cases implemented in this program were explained and we reach the end of the discussion related to task 1c.

Task 1c

Task Verification:

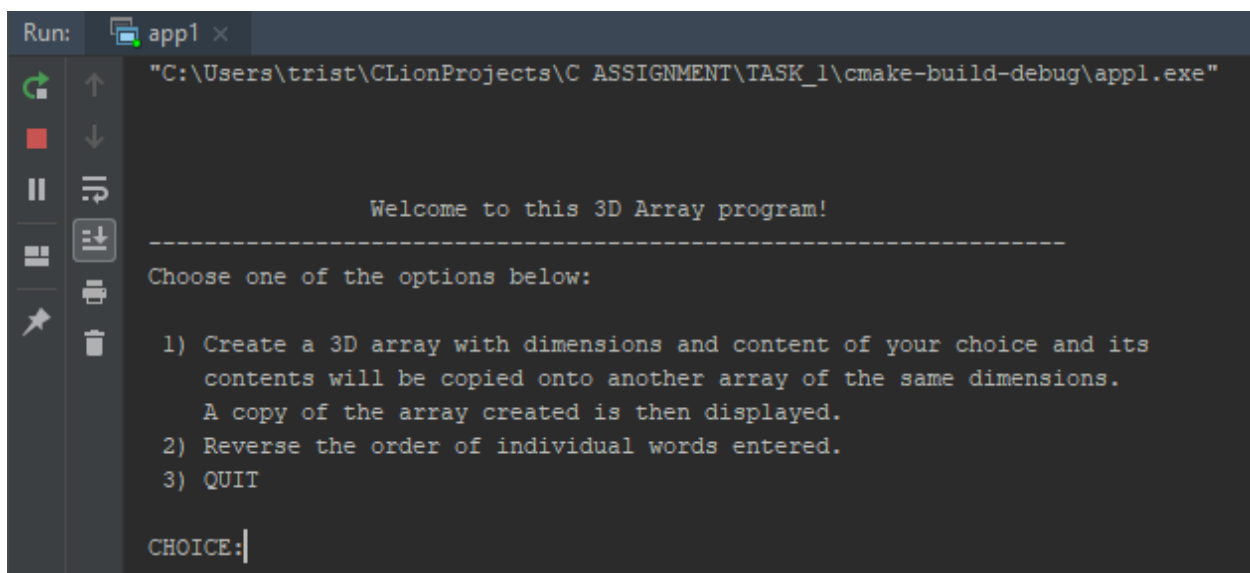
Task verification is a vital part in making sure the code written performs as expected. In this case, task verification was performed by testing the program with various valid, invalid and exaggerated inputs to check if it is working as expected.

The first test was to ensure proper compilation of the program. Upon building the program, the following successful compilation message was displayed:

```
Scanning dependencies of target appl
[ 50%] Building C object CMakeFiles/appl.dir/main.c.obj
[100%] Linking C executable appl.exe
[100%] Built target appl

Build finished
```

This meant that all the code was free of syntax errors, that's why it was able to compile. The next step was to run the program. This displayed the menu coded in task 1c perfectly as expected and asked the user to enter his choice.



```
Run: appl x
"C:\Users\trist\CLionProjects\C ASSIGNMENT\TASK_1\cmake-build-debug\appl.exe"

Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE:|
```

Upon entering a couple of characters, the program performed validation successfully as can be seen:

```
CHOICE: c
c
Invalid data!!
Please enter a number from 1 - 3 :
e
e
Invalid data!!
Please enter a number from 1 - 3 :
j
j
Invalid data!!
Please enter a number from 1 - 3 :
k
k
Invalid data!!
Please enter a number from 1 - 3 :
|
```

Next, a word was entered and the same error message was displayed:

```
Please enter a number from 1 - 3 :
Hi
Hi
Invalid data!!
Please enter a number from 1 - 3 :
Test
Test
Invalid data!!
Please enter a number from 1 - 3 :
IMPOSSIBLE
IMPOSSIBLE
Invalid data!!
Please enter a number from 1 - 3 :
|
```

To take it to the next level with testing, a couple of very long strings of characters were entered as excessive inputs and the same output was displayed back

```
Please enter a number from 1 - 3 :
Testing by entering very long sentences
Testing by entering very long sentences
Invalid data!!
Please enter a number from 1 - 3 :
This sentence is much longer than the previously entered string but still, validation works perfectly
This sentence is much longer than the previously entered string but still, validation works perfectly
Invalid data!!
Please enter a number from 1 - 3 :
|
```

Testing with different symbols and validation still works as it should:

```
Please enter a number from 1 - 3 :
$%
$$
Invalid data!!
Please enter a number from 1 - 3 :
@%
@%
Invalid data!!
Please enter a number from 1 - 3 :
)/%^%
)(%^%
Invalid data!!
Please enter a number from 1 - 3 :
|
```

Testing now with integer values much greater than the range. The value (being an integer) is returned back to the main function but the default case is chosen and so the menu is given back to the user while displaying the error message.

```
Please enter a number from 1 - 3 :
356
356

Incorrect input!
Make sure you enter a valid number between 1-3 from the list below.

-----
Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE:
```

Entering an excessively large number still goes through the default case and the menu is printed back to the user once again

```
CHOICE:1234567890123
1234567890123

Incorrect input!
Make sure you enter a valid number between 1-3 from the list below.

-----
Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE:
```

Now testing with mixed inputs. An integer value was entered followed by some characters. As can be seen below, the computer only read up till the end of the integer value and for this reason, didn't run through the while loop the first time round as expected. Instead, given that the entered number is larger than the available options, the default case was chosen once more and the menu() was displayed to the user again. However upon displaying the menu again, an error message from the while loop was displayed stating that invalid input was entered. This was because the characters entered were detected this time round and so, for the purpose of this program validation was still seen as successful.

```
CHOICE: 34mdfmdfg
34mdfmdfg

Incorrect input!
Make sure you enter a valid number between 1-3 from the list below.

-----
Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE: Ivalid data!!
Please enter a number from 1 - 3 :
|
```

Testing once again with mixed input but this time, characters and some symbols were entered first followed by some numbers. Given that the characters were entered first, the while loop was immediately triggered and the input buffer was cleared without even considering the numbers at the end. This result was as expected.

```

Please enter a number from 1 - 3 :
msdfjbl^*32
msdfjbl^*32
Ivalid data!!
Please enter a number from 1 - 3 :
illah*31
illah*31
Ivalid data!!
Please enter a number from 1 - 3 :
wowow023
wowow023
Ivalid data!!
Please enter a number from 1 - 3 :
|

```

Testing now with floats. When entering a decimal number, the program just like in the number-string case, read up till the instance where the last digit in a sequence was entered before being interrupted by a non-number character. Therefore the while loop was not activated and the value entered up till the '.' was stored and returned back to the main program activating the default case instead. This prompted the user to enter a value in the specified range and once more the menu was displayed again. Just like the number-string case, it continued reading the input from the user from where it last left off and so as it read '.', it stopped taking in input and the while loop was run once again asking the user to enter the appropriate data type.

```

Please enter a number from 1 - 3 :
34.35
34.35

Incorrect input!
Make sure you enter a valid number between 1-3 from the list below.

-----
Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE: Ivalid data!!
Please enter a number from 1 - 3 :
|

```

After finding out that the while loop doesn't really work as expected when given a floating point number as it assumes the '.' as a character instead of a floating point separator and after realizing that upon entering a string of mixed data in the form of numbers followed by strings, the while loop becomes useless during the first iteration and the characters written after the integers are only considered when returning back to the main menu, it was decided to clear the input buffer in the default case. This only solves a part of the problem but considering that the user enters a value not in the specified range followed by some characters, the characters won't be stored in the buffer upon returning back to the main menu and the problem would be partially solved in that sense as shown:

```
/*      The newly optimized default case      */
default: {

    printf("Incorrect input!\nMake sure you enter a valid number between 1-3 from the list below.\n");

    /* Clears input buffer from previously entered invalid data in menu */
    while(getchar()!='\n');

    /* Breaks out of the switch case back to the start of the do      */
    /* while loop and if the condition for looping is valid, then the  */
    /* loop runs again                                                */
    break;
}
```



```
CHOICE: 34.35
34.35

Incorrect input!
Make sure you enter a valid number between 1-3 from the list below.


Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE:|
```

Testing the program with valid data now

Testing Case 1:

If the user enters 1, the program returns this value successfully and goes into the first case of the switch statement.

```
                Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE:1
1

In this program a 3D array with dimensions of your choice will be created.
Let's get started.

Please enter the first dimension :|
```

The user is displayed a welcome message and asked instantly to enter the value of the first dimension. The data entered by the user, be it a character, string, float or integer will be validated just as demonstrated in the menu as a similar concept while loop is implemented which clears the buffer like before. For this reason it is not going to be demonstrated visually however to give a bit of a brief explanation, entering an integer value will return this value as the first dimension for the array. Entering a character will prompt the user to enter the appropriate value and clear the buffer. This is the same for entering a string. Entering a mixed string of characters followed by integers will prompt the user to enter the correct data type and clear the buffer once again. Entering an integer followed by characters will return the integer to the main menu and store it as the first dimension while calling

the dimension2() function and displaying an error since the remaining string would be read as input in this second instance. The user is asked once again to enter a value for dimension 2 and if correct input is entered, the program will proceed. The same happens if a floating point number is attempted to be entered.

Therefore considering the user enters valid data in dimension1() and dimension2() but doesn't enter valid info on the first run of dimension3(), this scenario would be displayed as follows:

```
In this program a 3D array with dimensions of your choice will be created.
Let's get started.

Please enter the first dimension :3
3
Please enter the second dimension :3
3
Please enter the third dimension :hgf
hgf
Incorrect input!!
Please input a number and press enter :
3
2
All 3 dimensions were entered successfully.
```

Now the user is asked to populate the array with the correct data.

```
The array contents need to be inserted next.
Enter the contents for {1, 1, 1}:
|
```

Validation of the same while loop system was included in the insertContents() function to ensure the user enters the correct information. This behaves as already explained before just like in the menu() and dimension functions.

```

/* If something other than an integer is entered, this while loop */

/* runs prompting the user to enter the appropriate data type */
while(scanf("%d", &ar[block][row][col]) != 1){
    printf("Invalid input!! \nPlease input an integer : \n");

    /* Clears input buffer from previously entered invalid data */
    while(getchar() != '\n');

```

Now the 3D array containing 3 blocks by 3 rows by 2 columns will be populated with some correct inputs and some invalid inputs as can be seen.

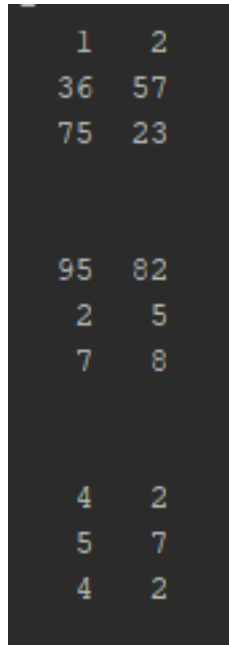
```

The array contents need to be inserted next.
Enter the contents for {1, 1, 1}:
1
1
Enter the contents for {1, 1, 2}:
2
2
Enter the contents for {1, 2, 1}:
36
36
Enter the contents for {1, 2, 2}:
87
57
Enter the contents for {1, 3, 1}:
t
t
Invalid input!!
Please input an integer :
te5
te5
Invalid input!!
Please input an integer :
75
75
Enter the contents for {1, 3, 2}:
|

```

```
Enter the contents for {1, 3, 2}:
23.2
23.2
Enter the contents for {2, 1, 1}:
Invalid input!!
Please input an integer :
095
095
Enter the contents for {2, 1, 2}:
82
82
Enter the contents for {2, 2, 1}:
input
input
Invalid input!!
Please input an integer :
2
2
Enter the contents for {2, 2, 2}:
5
5
Enter the contents for {2, 3, 1}:
7
7
Enter the contents for {2, 3, 2}:
8
8
Enter the contents for {3, 1, 1}:
4
4
Enter the contents for {3, 1, 2}:
2
2
Enter the contents for {3, 2, 1}:
5
5
Enter the contents for {3, 2, 2}:
7
7
Enter the contents for {3, 3, 1}:
4
4
Enter the contents for {3, 3, 2}:
final
final
Invalid input!!
Please input an integer :
2
2
```

All the valid values were stored and now the copyContents() function is run and the contents of the original array3d[3] [3] [2] are copied onto the empty copyArray[3] [3] [2]. The final function to be executed in case 1 is the pasteContents() function which simply prints out the copy of the array created to the user as shown :



```
1 2
36 57
75 23

95 82
2 5
7 8

4 2
5 7
4 2
```

After displaying this 3d array, the program breaks out of the switch case statement and the menu is called once again.

Testing Case 2:

If the user enters 2, the program returns this value successfully back to the main where it is stored as a placeholder in choice and evaluated in the switch case statement. This triggers the choice of case 2 and the underlying processes in this case are executed.

```
                Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE: 2

In this program a string of characters entered will have
whole individual words reversed.

To get started, enter a string of not more than 100 characters:
|
```

The user is then asked to enter a string of not more than 100 characters. No validation based on input types is performed as an entered string can be anything. The only precaution taken here, as explained before, is the using of the fgets function. This is defined to only accept input up till the 100th character. If the gets function was used instead and the user attempted to enter a string larger than required, the program will try its best to store the extra input into other memory locations which have not been reserved to store that particular input thus resulting in corruption of data due to overwriting of information.

Therefore, when using this function, any input entered would be stored up till the 100th character making it perfect for the purpose of this program. *scanf* was not even considered to be used as it stores only one entered word.

To test the duality of the fgets function, a string 110 characters long was entered and as can be seen, it was still accepted as input but the words exceeding the string were automatically cut off. The words saved in the string were then reversed back to the user and since more words were entered than requested, the reversed string reversed only the words entered up till the 100th character.

```
In this program a string of characters entered will have
whole individual words reversed.

To get started, enter a string of not more than 100 characters:
This sentence is much larger than the required amount and so it will not be stored as a whole thanks to fgets
This sentence is much larger than the required amount and so it will not be stored as a whole thanks to fgets
thank whole a as stored be not will it so and amount required the than larger much is sentence This
```

This function is now tested once again but with words fitting the requirements as shown:

```
To get started, enter a string of not more than 100 characters:
This string fits the specification stated above!!
This string fits the specification stated above!!
above!! stated specification the fits string This
```

Upon finishing, the reversing and outputting the string as should be, the program breaks out of the switch statement and the menu is displayed to the user once again.

Testing Case 3:

After returning back to the main, the menu() function is evaluated again asking the user for his preferred option. If the user enters 3, this value is returned back to the main menu and stored in the placeholder choice. This is then evaluated in the switch case statement and case 3 is selected. This will simply print out a "Bye!" message to the user and break out of the switch case. The condition for looping the do while loop is checked and in this particular case, the value 3 does not match the condition for looping and so the do while loop does not loop again. The program continues with its normal execution until it reaches the end of the program.

```
                Welcome to this 3D Array program!
-----
Choose one of the options below:

1) Create a 3D array with dimensions and content of your choice and its
   contents will be copied onto another array of the same dimensions.
   A copy of the array created is then displayed.
2) Reverse the order of individual words entered.
3) QUIT

CHOICE: 3
3

Bye!

--End of program--
Process finished with exit code 0
```

Task 2a

Task Specification:

A simplified version of a tuple data type and the functions related to it were to be implemented. This had to be given a fixed structure but had to consist of at least 3 elements. The created elements were then required to be stored within a dynamic array on the heap which would be continuously resized each time a new tuple was going to be created.

Task Creation:

A New Project was created in CLion entitled 2a which consisted of a CMakeLists.txt file and a main.c source file upon creation.

The CMakeLists.txt file was the first to be modified. Its contents were standardized to match the cross-platform build system while also including the main.c file (which was renamed to tuple2a.c) be recognized as a source file. The modifications done can be seen below:

```
1  cmake_minimum_required(VERSION 3.12)
2  project(2a C)
3
4  set(CMAKE_C_STANDARD 99)
5  set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall")
6
7  set(SOURCE_FILES1 tuple2a.c)
8  add_executable(2a ${SOURCE_FILES1})
9
```

An associated header file related to this file was also created to be able to keep hold of the global variables, the defined macros and the function prototypes. This was named *tuple2a.h*. This was then called from the *tuple2a.c* source file along with the list of libraries which had to be included in this project for it to be valid. The included libraries can be seen in the text box:

```
/*                      PREPROCESSOR DIRECTIVES                      */

/* Including the stdio.h library which is C's Standard Input and Output */
/* Library comprising of basic functions                               */
#include <stdio.h>

/* Including the string.h library which defines several functions that */
/* manipulate C strings and arrays                                     */
#include <string.h>

/* Including the stdlib.h library which contains a set of functions to general */
/* purpose functions including memory management functions             */
#include <stdlib.h>

/* Including the header file complementary to this source file         */
/* This contains the defined macros, global variables and all function prototypes */
#include "tuple2a.h"
```

Now that the libraries were included, it was time to start developing the tuple data type. This took some time and research to actually understand what a tuple was and what was its function. But after this obstacle was out of the way, everything what had to be implemented became much clearer. It was decided that this tuple data type had to be created with the help of a struct. The struct created contained a character array to store the string ID and 3 separate integer variables to store the elements (since that was the minimum requirement for this question. The keyword typedef was then used to refer to the struct simply with the keyword *tuple* rather than referring to it as a *struct tuple_t* each time. The above mentioned can be seen in this snippet:

```
/* The structure being used for the tuple being created in this task */  
typedef struct tuple_t{  
  
    char ID[LENGTH];  
  
    int element1;  
  
    int element2;  
  
    int element3;  
}
```

The value for LENGTH (as seen above) which specifies the size of the character array was then set as a macro of size 15. This was done so that it would be easier to understand the code and also so that its value can be changed independently, if needs be.

```
/* Defining the macro SIZE to 15 to be used as a constant for the array ID size */  
#define LENGTH 15
```

When planning on paper how this task had to be implemented, it was decided that a global integer variable for the amount of tuples created had to be included. This was called the counter and would be incremented by 1 each time a new tuple is added.

```
/* Initialising the global variable counter to 0 for now */  
int counter = 0;
```

The final concept which yet had to be implemented and developed is the concept of memory. These tuple variables had to be stored someplace for their contents to be retained. Thus, it was decided that to store the tuples, a dynamic array was to be used which had to be resized each time a new tuple was to be created. This was made possible through the use of pointers and the malloc and realloc functions found in the stdlib.h library. The space for the first tuple was allocated as shown:

```
/* Creating a pointer tupleArray of type tuple */  
tuple * tupleArray;  
  
/* tupleArray points to a memory location of size tuple which has just been */  
/* allocated on the heap */  
tupleArray = (tuple *) malloc(sizeof(tuple));
```

A pointer tupleArray of type tuple was initialized and was made to point to a specific location on the heap. The memory allocated for the first tuple was exactly the size of a single tuple, to waste as less memory as possible. This memory would then be required to be resized as new tuples are introduced. Therefore, a function to resize this dynamic array was written to be called each time it had to be used. The only parameter passed through this function was the pointer to the dynamic array itself. Its function prototype can be seen below:

```
/* A void function which does not return anything but makes use of the realloc */  
/* function to resize the size of the array based on the amount of tuples */  
void resizeArray(tuple * tupleArray);
```

This function made use of the realloc function to resize the dynamic array based on the counter variable which is incremented each time a new tuple is to be introduced.

```
/* When resizeArray() function is called, execution continues from here */  
void resizeArray(tuple * tupleArray) {  
    tupleArray = (tuple * ) realloc( tupleArray, counter * (sizeof(tuple)));  
}
```

Moving on, the createTuple() function had to be implemented next. This function was the basis of all the other functions as it lead to the creation of tuples. If this function was not made to work, then all the other functions would have been

useless. When thinking ahead how to create this function, it was made clear that the user had to enter the string identifier of the tuple to be created and then, he had to enter 3 integer variables, each stored in element1, element2 and element3 respectively. The counter for the amount of tuples also needed to be incremented by 1. Therefore a brief structure for this function was set as shown:

```
/* When createTuple() function is called, execution continues from here */
void createTuple(tuple * tupleArray){

    /* Increments tuple count by 1 */
    ++counter;

    printf("Creating Tuple # %d: \n\n", counter);
    printf("Please enter the tuple's string identifier (not more than 15 characters) :\n");
    //fgets is used to store the string ID entered by the user in the array up till the 15th char entered
    fgets(tupleArray[counter - 1].ID, LENGTH, stdin);

    printf("Now enter the elements going to be stored in TUPLE # %d\n", counter);
    printf("You must enter 3 elements for each tuple\n\n");

    printf("Please enter element #1: ");
    scanf(" %d", &tupleArray[counter - 1].element1);

    printf("Please enter element #2: ");
    scanf(" %d", &tupleArray[counter - 1].element2);

    printf("Please enter element #3: ");
    scanf(" %d", &tupleArray[counter - 1].element3);

}
```

As can be seen, the pointer pointing to the dynamically allocated memory on the heap was passed as an argument to this function to be able to be used it in. Next the counter was incremented by 1 as explained since a new tuple is being created. The user is then asked to enter the string ID of the tuple he wants to create and fgets is used to store the user's input up till the 15th character entered. This is done to ensure that the user sticks to the guidelines presented as well as ensuring that no memory is overwritten in the unlikely event where the program tries storing more than what was allocated. The entered string ID is stored in tupleArray[counter - 1].ID . Since arrays work on a 0-up basis, the first inputs for the first tuple would be stored in tupleArray[0]. This corresponds to the counter - 1. It is for this reason that the string ID for any tuple to be created is stored in the number of tuple we are talking about minus 1. Moving on, the user is then asked to enter the 3 elements to be stored into element1, element2 and element3 of the array. Being integers, scanf was used to take the input of the user and place it in the memory pre-allocated beforehand. The integer value for the first element was stored in &tupleArray[counter - 1].element1 just like when storing the string ID, with the only difference being that the variable name right after the dot was changed to element1.

This method as described so far works properly the first time round, however would not be suitable if run a second time. The reason for this is because the resize array function has not yet been included in this function. Thus if the user tries running it again, he won't have enough space allocated for the next set of tuples to come. For this reason, the resize array function was included with the tupleArray pointer passed as a parameter and confined in an if statement. This made sure that the function was executed only when the tuple counter was not equal to 1 so that no extra space is allocated for nothing.

```
/* Increments tuple count by 1 */
++counter;
if(counter != 1){
    resizeArray(tupleArray);

    /* Clears input buffer from previously entered invalid data */
    while(getchar()!='\n');
}
```

Enclosed in the if statement is a while loop which was included as a precaution for when creating more than one tuple. To explain it in simple terms, when modelling the approach for the creation of this tuples program, it was decided that a menu had to be displayed to the user to allow him to execute whichever functions he desired. So for example, if the user enters '1' on the keyboard and hits enter, the function enclosed in 1 which would be the createTuple() function would execute. However if the input buffer contains some information and the user is asked to enter a string ID for the tuple to be created, this would cause a massive problem as the string ID given to the tuple would not have been determined by the user. This is mainly a problem when the user types 1 on the keyboard and mistakenly enters a character or 2 right after the integer. The program then takes notice of the 1 while keeps record of the entered characters into the input buffer which might end up being read as input if the buffer is not emptied.

With the addition of this empty buffer while loop, the function becomes more and more versatile. Another way of making this function better is by including validation methods when the user enters the elements. The same concept used in task 1 was adopted and was implemented to all three elements to be entered. The code about to be seen basically ensures that an integer value is successfully passed to be stored. If not, a while loop executes letting the user know of the invalid input while clearing the buffer and asking for correct input of the data. The snippet shown below shows what was implemented for element1 only, as elements 2 and 3 rely on the same concept.

```
printf("Please enter element #1: ");

while(scanf("%d", &tupleArray[counter - 1].element1) != 1){

    printf("Invalid data!! \nPlease enter an integer : \n");

    /* Clears input buffer from previously entered invalid data */

    while(getchar()!='\n');

}
```

After implementing the above validation to all 3 inputs, the final step was to implement a way in which if the user enters the string ID of an existing tuple, the

previous tuple would be deleted to make way for the new one. This was decided to be implemented with the help of a for loop which loop through all the existing tuple IDs stored on the heap and if one with the same string ID as the one wanting to be created now is identified, then the old one is deleted through the deleteTuple() function.

```
/* Counter which loops in the for loop */  
  
int i;  
  
for(i = 0 ; i < counter - 1 ; i++){  
    if( strcmp(tupleArray[counter - 1].ID, tupleArray[i].ID) == 0){  
        deleteTuple(&tupleArray[i]);  
    }  
}
```

The local variable defined above was added to the start of the createTuple() function whilst the for loop was fitted right after the user is asked to enter the string ID. As can be seen, the strcmp() function is used which basically, after being passed 2 strings, compares their characters and returns a 0 if all characters are the same. This is being implemented by passing what the user has just entered as the first parameter and the existing string IDs which are looped each time. If a matching string ID is found, the deleteTuple() function (which has not yet been implemented) is executed and contains the address of the array to be deleted. The function prototype created for this function can be seen below:

```
/* A void function which creates a tuple based on the defined struct above */  
  
/* The user is asked to enter a string ID for the tuple being created and 3 */  
/* integer elements in all which are then all stored on the heap to be accessed */  
/* whenever needed */  
  
void createTuple(tuple * tupleArray);
```

The next function to be implemented was the getTupleByID() function. This function simply returns a tuple's pointer after being given its string ID. This set the

basis for the type of function. Unlike the previous function, this one had to be of type tuple to be able to return a memory address. Therefore, since it had to return the memory address of a specific tuple, it needed to have the pointer to the dynamic array passed as a parameter.

To start out, a method of receiving and storing input for the string ID had to be determined. It was concluded that memory had to be allocated on the heap to just fit a regular string ID. This memory would then be freed to waste as least memory as possible.

```
/* Space big enough to store a whole string ID is allocated to be freed later on */
char * tupleid = (char *) malloc(LENGTH * sizeof(char));

//more code

/* The memory allocated earlier on the heap is now freed */
free(tupleid);
```

Thus, a user would be asked to enter the string ID which would be stored at this temporary memory location and this would then be compared in a similar fashion to before with the help of a for loop and a strcmp() function as shown:

```
/* Local variable i to be used in the for loop */
int i;

printf("Please enter the tuple string ID\n");
fgets(tupleid, LENGTH, stdin);

/* for loop which loops through all existing tuple IDs until it finds a matching one */
for (i = 0; i < counter; i++) {
```

```
/* if statement which runs if the entered string matches some other tuple */  
/* string ID already stored on the dynamic array */  
if (strcmp(tupleid, tupleArray[i].ID) == 0) {  
  
    printf("Tuple #%%d found!! \n", i+1);  
  
    /* The memory allocated earlier on the heap is now freed */  
    free(tupleid);
```

The for loop loops through all the existing tuple IDs and if a matching one is found, the strcmp function returns 0, which activates the if statement. The user is then notified that a tuple has been found together with the tuple number and the memory previously stored on the heap is freed since it no longer needs to be used. The pointer can then be returned to the user right after freeing the memory as shown:

```
/* A pointer associated to the entered string ID is returned back to main */  
return &tupleArray[i];
```

Up till this point, the method is functional and works as expected given that the user enters the correct pointer immediately. But what can be done to validate this method better in a way that the user would be notified that an incorrect pointer has been entered? By implementing a simple while loop which runs indefinitely, the program will keep on expecting input from the user if a matching string ID has not been found. An error message and clear input buffer technique similar to before have also been included to make the user realize that incorrect input has been entered and to allow him to try again.

```
/* While loop runs infinitely unless the user enters the correct string ID */
/* and it returns a pointer to the main function */
while (1) {

    printf("Please enter the tuple string ID\n");
    fgets(tupleid, LENGTH, stdin);

    /* for loop which loops through all existing tuple IDs until it finds a matching one */
    for (i = 0; i < counter; i++) {

        /* if statement which runs if the entered string matches some other tuple */
        /* string ID already stored on the dynamic array */
        if (strcmp(tupleid, tupleArray[i].ID) == 0) {

            printf("Tuple #%d found!! \n", i+1);

            /* The memory allocated earlier on the heap is now freed */
            free(tupleid);

            /* A pointer associated to the entered string ID is returned back to main */
            return &tupleArray[i];
        }
    }

    printf("Tuple with string ID %s not found.\n", tupleid);
    printf("Please try again.\n\n");

    /* Clears input buffer from previously entered invalid data */
    while (getchar() != '\n');
}
```

This traps the user in an infinite loop until the correct string ID is entered and the user would be returned back to main with a corresponding pointer. The function prototype for this function can be seen below:

```
/* A function which asks the user to input an existing string ID and returns the */  
/* corresponding pointer */  
tuple * getTupleByID(tuple * tupleArray);
```

One other thing which was implemented into this function later on during testing was another input buffer clearing while loop at the start of the function as in some cases, previously entered data in the main was kept in the input buffer and dumped into the first part of the function incorrectly which asks for the string ID.

```
/* Clears input buffer from previously entered invalid data */  
while (getchar() != '\n');
```

The third function to be implemented was the exact opposite of the above function. The getTupleID() function's purpose would be to return back to the user a tuple's string ID, after being given its pointer. This set the basis for the type of this function which was set to char* as can be seen in the prototype below. Once again, the dynamic array pointer is passed as an argument to be able to retrieve information related to the tuple:

```
/* A function which returns a tuple's string ID after being given it's pointer */  
/* by the user */  
char * getTupleID(tuple * tupleArray);
```

Being of a very similar structure and format to the previous function, the general layout used for it was adopted and the pointer and string ID were simply switched round. This meant that a pointer of type tuple had to be created this time as opposed to before where memory was allocated on the heap and then freed.

```
/* When * getTupleID() function is called, execution continues from here */
char * getTupleID(tuple * tupleArray){

    /* Local variable i to be used in the for loop */
    int i;

    /* Pointer of type tuple is created */
    tuple * pTuple;

    /* While loop runs infinitely unless the user enters the correct pointer */
    /* and its corresponding string ID is returned back to the main function */
    while (1) {

        printf("Enter the pointer of the tuple to get its ID:\n");
        scanf(" %p", &pTuple);

        /* for loop which loops through all existing tuple pointers until it finds a matching one */
        for (i = 0; i < counter; i++) {

            /* if the entered pointer matches an existing one the code in the block executes */
            if ( pTuple == &tupleArray[i]) {
                printf("Tuple #%%d found!! \n", i+1);

                /* The string ID corresponding to the matching pointer is returned back to main */
                return tupleArray[i].ID;
            }
        }

        printf("Tuple with memory address %p not found.\n", &pTuple);
        printf("Please try again.\n\n");

        /* Clears input buffer from previously entered invalid data */
        while (getchar() != '\n');
    }
}
```

The function used to accept input was scanf as it made more sense to use it since a memory address was to be stored. In contrast to before, the memory address entered by the user is now compared to the existing ones and if one is found, the user is notified and returned back to main with a corresponding string ID.

The 4th function to be created is the showTuple() function. This function should be able to print out to the user the contents within a specific tuple when given the correct pointer to tuple. This function once again makes use of the pointer to the tuple array to be able to have access to the tuple contents directly. The function prototype for this function can be seen below:

```
/* A void function which simply displays the contents of a particular tuple after */  
/* the user provides it with the correct pointer */  
void showTuple(tuple * tupleArray);
```

This can be considered as one of the most basic functions in this program as its implementation requires a bit of visualizing at first, but when brainstormed becomes easy to create. Since the user is requested to enter a tuple pointer to be able to display the details, a pointer of type tuple is created. This is then used to hold the memory address entered by the user through scanf as shown.

```
/* A pointer of type tuple is declared */  
tuple *pTuple;  
  
printf("Enter the tuple's pointer for the tuple you want to display:\n");  
  
/* Stores the pointer entered by the user to be compared */  
scanf(" %p", &pTuple);
```

Once the user enters the address, a for loop is used to go through all the existing pointers and if a matching one is found, an if statement is implemented to print out the details the pointer is pointing to. A local variable i is declared at the start of this function to be used in the for loop. The function created can be seen on the next page:

```

/* When showTuple() function is called, execution continues from here */
void showTuple(tuple * tupleArray) {

    /* Local variable i to be used in the for loop */
    int i;

    /* A pointer of type tuple is declared */
    tuple *pTuple;

    printf("Enter the tuple's pointer for the tuple you want to display:\n");
    /* Stores the pointer entered by the user to be compared */
    scanf("%p", &pTuple);

    /* A for loop which loops to find if the pointer entered by the user matches*/
    /* the pointer of an existing tuple*/
    for (i = 0 ; i < counter; i++) {

        /* If the pointer the user entered matches an existing one, a tuples contents are
        displayed*/
        if (pTuple == &tupleArray[i]) {
            printf("Tuple\tString Identifier\t\t\tElements\n");
            printf("-----\n");

            printf("#%-3d \t%-17s\t\t\t\t\t(%d, %d, %d)", i+1, tupleArray[i].ID,
                tupleArray[i].element1, tupleArray[i].element2, tupleArray[i].element3);

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

```


The 5th function to be created is the deleteTuple() function which, as the name implies, proceeds to delete an existing tuple when provided with a valid string ID. To start with, the user must be asked to enter the string ID of the tuple he has in mind to delete. For this to be done, it was decided that memory would be allocated on the heap, big enough to store an entire string ID. This memory would then be freed later on in the function to preserve space. Thus, a pointer of type character is created which then allocated this said memory

```
/* A pointer of type character is created to allocate memory on the heap */  
char * del;  
  
del = (char*) malloc(LENGTH * sizeof(char));  
  
//code yet to be inserted  
  
/* The memory allocated on the heap for del is now freed */  
free(del);
```

The user would then be asked to enter the string ID of the desired tuple to be deleted and this input is stored with the fgets() function onto the memory allocated in the heap.

```
printf("Enter the string identifier of the tuple you want to delete:\n");  
  
/* User enters the string ID of a tuple which is stored up till the */  
/* 15th character entered on the heap */  
fgets(del, LENGTH, stdin);
```

It is at this stage that a for loop needs to be implemented to loop through all the possible string ID names stored for each tuple on the dynamic array. Therefore, a local variable i is declared which would be used in the for loop. Thus, to check if an entered string is exactly equal to another pre-existing string ID, the function strcmp() is used which is very similar to the strncmp() function used previously. The only difference between these 2 is that the one used now requires an extra

parameter to be passed through it. This 3rd parameter stores the amount of characters to be compared. In this case, LENGTH was used as the third parameter because the longest possibility for a string ID is 15 characters. If the first 2 parameters (strings) match in content and in size, the value 0 is returned. The user is then notified if a match was found and would also be notified of the tuple number, string ID and elements which are about to be wiped out (sort of like a small extension of the showTuple() function).

```

/* Initialises a local variable i to 0 */
int i = 0;

/* A for loop which goes through all tuple IDs created */
for (i = 0 ; i < counter ; i++) {

    /* If a compared tuple ID matches what was entered by the user, */
    /* the tuple is then deleted */
    if (strncmp(del, tupleArray[i].ID, LENGTH) == 0) {

        printf("Tuple #%d found!! \n", i+1);

        /* The tuples content to be deleted are displayed one last time */
        printf("Deleting %s containing (%d, %d, %d)\n", del, tupleArray[i].element1,
        tupleArray[i].element2, tupleArray[i].element3);
    }
}

```

After these lines of code (still in the if statement) follows a while loop which performs deletion. This method of deletion is performed by overwriting the contents stored in the tuple to be deleted. This is done by storing the contents of the next tuple into the tuple to be deleted. This process is then repeated (hence the while loop) so that all the tuples are shifted down to fill in the gap left by the moved tuples until the gap is finally at the end of the dynamic array. Up till this point, memory is still allocated for this final tuple in which a copy of its contents have been moved to the tuple below it, so to fix this, the counter is decremented

and the `resizeArray()` function is called to get rid of that unwanted space. The memory previously allocated on the heap to store the tuple's ID is given back to the system and the function returns back to main.

```
/* While loop which loops until i < counter */
while (i < counter) {

    /* All that is stored in i+1 is written over what is stored in i */
    strcpy(tupleArray[i].ID, tupleArray[i+1].ID);
    tupleArray[i].element1 = tupleArray[i+1].element1;
    tupleArray[i].element2 = tupleArray[i+1].element2;
    tupleArray[i].element3 = tupleArray[i+1].element3;

    /* The value stored in i is incremented by one */
    i++;
}

/* The number of tuples is decremented by 1 since a tuple has been deleted */
--counter;

/* Now that 1 less tuple is available, the memory allocated on the heap is resized */
resizeArray(tupleArray);

/* The memory allocated on the heap for del is now freed */
free(del);

/* The function returns nothing back to main */
return;
```

However, what has been implemented up till now for the deleteTuple() function works only considering that the user enters an actual valid ID already part of a tuple. If the user tries to test the function with other inputs, then something else might happen. Therefore, to ensure that this function works as instructed, a similar concept while loop which loops indefinitely is wrapped around the part in which the user is prompted to enter the string ID of the tuple he wants deleted and another part which is added right after the for loop which basically notifies the user that what he entered is invalid and the while loop which clears the input buffer. This can be understood better in the text box below:

```
while(1) {

    printf("Enter the string identifier of the tuple you want to delete:\n");

    //...
    //body of code already explained
    //...

    /* The function returns nothing back to main */
    return;
}

printf("Tuple with string ID %s not found.\n", del);
printf("Please try again.\n\n");

/* Clears input buffer from previously entered invalid data */
while (getchar() != '\n');

}
```

Now that validation has been performed, the user has to enter a valid string ID to be able to return back to main. Another important thing that must be mentioned is that after performing some light tests on this function, it was decided that a while loop method to clear the buffer had to be included to prevent data entered in the main from affecting the input buffer. This was entered at the very start of the function.

```
/* Clears input buffer from previously entered invalid data */  
while (getchar() != '\n');
```

Now that this function had been completed, considering that the createTuple() function had been coded properly, all that was required to be implemented in the createTuple() function was done successfully. This is because if the user enters the string ID of an existing identifier upon creating a new tuple, this function must be called to delete that other tuple and then jump back into the createTuple() function to continue adding the new tuple. Another important point to be mentioned is that, just like the other functions, this function has a pointer to the dynamic array passed through it as a parameter as can be seen in its function prototype:

```
/* A function which deletes a tuple object when the user enters it's string ID */  
void deleteTuple(tuple * tupleArray);
```

The next function to be created is the cmpTuples() function. This function aims to return an integer value based on a comparison between 2 tuples the user chooses. This is the function prototype for this function:

```
/* The user is asked to enter the string ID of 2 tuples to be compared */  
/* An in-order comparison of their elements follows and a 1 is returned if the */  
/* first non-equal element pair is larger for the first entered tuple, a -1 if */  
/* the opposite case is encountered and 0 if all elements are equal */  
int cmpTuples(tuple * tupleArray);
```

The pointer to the tupleArray is once again passed to be used to be able to compare the 2 tuples. To start off, 2 arrays of size LENGTH are created for a change rather than allocating memory on the heap. This was done to show that both ways work when implementing these functions. However, when memory is used and freed, less memory is occupied at given times and so, that would be generally preferred over the other method. After initializing these 2 character arrays, the user was asked to enter a string ID into 1 of them and a for loop was implemented to loop through all the possible string IDs stored in the dynamic array. Once again, an if statement containing a strncmp() function was used and the 3 parameters passed were the first character array cmp1, the tupleArray[i].ID which was looped each time and the LENGTH macro. If the returned value from this function was equal to 0, the block within the if statement would loop letting the user know that a tuple with that ID has been found. This is then followed by a break statement which exits the user out of the statement. The same procedure was about to be implemented to accept input for the other string which was still empty. However, it was noted that validation had to be performed so as to make the method more reliable as can be seen:

```
/* 2 arrays of size LENGTH are created*/  
  
char cmp1[LENGTH];  
char cmp2[LENGTH];  
  
/* These local variables are initialised */  
int i, j, k = 0;  
  
/* start of do while loop */  
do {  
    printf("Please enter the 1st string to compare:\n");  
    /* The string entered up till the 15th character is stored in cmp1 */  
    fgets(cmp1, LENGTH, stdin);  
  
    /* for loop loops through all the existing tuple IDs */  
    for (i = 0; i < counter; i++) {
```

```
/* If the values entered by the user matches an existing ID, the code in the block runs */
if ((strcmp(cmp1, tupleArray[i].ID, LENGTH) == 0)) {

    printf("Tuple # %d found!! \n", i + 1);

    /* The value of k is incremented by 1 */
    k++;

    /* breaks out of the statement if this part is reached */
    break;
}

/* While condition for looping is still valid, the loop continues looping */
}while(k != 1);
```

Now that what has been described in the paragraph present on the previous page can now be seen visually, the validation performed can be understood far better. The way validation was performed was by including a do while loop enclosing what had already been explained. This loop would keep on looping until the value of the newly introduced variable k (which is set to 0) becomes equal to 1. The only way this value could become equal to 1 is if the user enters an existing string ID and the if statement executes causing the value of k to increment by 1.

Another thing to notice is that this time round, no input buffer clearing method was used to dump the previously entered incorrect input. This is because this dumping of incorrect info is done automatically by the fgets function since it is used instead of scanf this time.

The same concept mentioned above was then applied for the second string changing the condition for looping in the do while loop to (k != 2). This can be seen on the next page:

```
/* start of do while loop */
do {
    printf("Please enter the 2nd string to compare:\n");
    /* The string entered up till the 15th character is stored in cmp1 */
    fgets(cmp2, LENGTH, stdin);

    /* for loop loops through all the existing tuple IDs */
    for (j = 0; j < counter; j++) {

        /* If the values entered by the user matches an existing ID, the code
        in the block runs */
        if ((strcmp(cmp2, tupleArray[j].ID, LENGTH) == 0)) {

            printf("Tuple # %d found!! \n", j + 1);

            /* The value of k is incremented by 1 */
            k++;

            /* breaks out of the statement if this part is reached */
            break;
        }
    }

    /* While condition for looping is still valid, the loop continues looping
    */
}while(k != 2);
```

After having modded both input methods as best desired, it was time to create a way to compare the elements of both tuples. A complex if statement model was

the first thing to come to mind and since it was seemed most feasible, it was implemented as shown below:

```
/* Block of code which determines what should be returned based on the values */
/* stored in the elements of the tuples being compared */
if (tupleArray[i].element1 > tupleArray[j].element1) {

    return 1;
} else if (tupleArray[i].element1 < tupleArray[j].element1) {

    return -1;
} else {

    if (tupleArray[i].element2 > tupleArray[j].element2) {

        return 1;
    } else if (tupleArray[i].element2 < tupleArray[j].element2) {

        return -1;
    } else {

        if (tupleArray[i].element3 > tupleArray[j].element3) {

            return 1;
        } else if (tupleArray[i].element3 < tupleArray[j].element3) {

            return -1;
        } else {

            return 0;
        }
    }
}
```

This basically scanned which element was bigger and which was smaller and a positive or negative value was returned respectively. If elements 1 of both tuples were of the same value, the same analysis was done for elements 2 of these same tuples. If both element 2s were the same again, the same nested if statement procedure was conducted. If in the final iteration (i.e. when element3 of tuple1 was compared to element3 of tuple2), the values were still the same, then 0 was returned back to the user.

The next function to be created is the joinTuples() function. For the purpose of this task, this function would simply expect the user to enter 2 string IDs. Then, from the first string ID entered, a copy of the whole tuple identified by that string identifier would be created and stored on the dynamic array as a new tuple. Despite this, this function is still a void function and does not return anything back to main.

```
/* A void function which asks for 2 existing string IDs and creates a copy of the*/  
/* contents stored in the first and saves this copy as a new tuple          */  
void joinTuples(tuple * tupleArray);
```

This function started out by asking the user to enter the string ID of the new tuple to be created. This meant that a character array of size LENGTH had to be initialized so that the string entered by the user could be stored there through the fgets() function. However, to prevent any unwanted input buffer values (which have not yet been dumped) from affecting this string of characters, a while loop input buffer emptying method was used again as a precaution. The code entered up till this point can be seen below:

```
/* When joinTuples() function is called, execution continues from here */  
void joinTuples(tuple * tupleArray) {  
  
    /* An array called join of size LENGTH is initialised */  
    char join[LENGTH];  
  
    /* Clears input buffer from previously entered invalid data */  
    while (getchar() != '\n');  
  
    printf("Enter the String ID of the new tuple to be created: \n");  
    fgets(join, LENGTH, stdin);
```

Then, 2 pointers of type tuple were initialized at the start of the file to store the pointers the user enters. To deal with the first case only for now, the user is then asked to store the value of the first pointer into a scanf() function which is then stored in the pointer variable. Right after, a for loop is used to go through all the existing pointers to tuples and an if statement is implemented so that, if there is a match, the user is notified, the value of i is kept and a break is used to break out of the statement. However, considering the user enters invalid data, the function would not be of much worth. Therefore, a do while loop is implemented around the part which asks the user for the pointer and the end of the for loop to make the user enter the value again, if he entered incorrectly. To make this possible, a buffer emptying method was added right after the for loop and before the end of the do while loop to ensure that the input buffer is emptied out. As per the condition in the do while loop, a new variable k was declared at the start of the function, together with the other variables, and this was also included to increment if the if statement was reached, right before the break. The condition for looping was then made to be that the loop would stop looping if the value of k becomes 1.

```
/* Three local variables are initialised */
int i, j, k=0;

//more code already explained

/* Start of do while loop */
do {
    printf("Enter the 1st tuple's pointer:\n");
    /* Stores the pointer entered by the user to be compared */
    scanf("%p", &pTuple1);

    /* Loops through all the existing tuple pointers */
    for (i = 0; i < counter; i++) {

        /* If the pointer entered matches an existing tuple pointer */
        /* the code included in this block is executed */
        if (pTuple1 == &tupleArray[i]) {

            printf("Tuple #%d found!! \n", i + 1);

            /* Increments k by 1 */
            k++;

            /* breaks out of the statement if this part is reached */
            break;
        }
    }

    /* Clears input buffer from previously entered invalid data */
    while (getchar() != '\n');
}while(k != 1);
```

The same concept is used for accepting input for the second pointer as can be seen:

```
/* Start of do while loop */
do {

    printf("Enter the 2nd tuple's pointer:\n");

    /* Stores the pointer entered by the user to be compared */
    scanf("%p", &pTuple2);

    /* Loops through all the existing tuple pointers */
    for (j = 0; j < counter; j++) {

        /* If the pointer entered matches an existing tuple pointer */
        /* the code included in this block is executed */
        if (pTuple2 == &tupleArray[j]) {

            printf("Tuple #%d found!! \n", j + 1);

            /* Increments k by 1 */
            k++;

            /* breaks out of the statement if this part is reached */
            break;
        }
    }

    /* Clears input buffer from previously entered invalid data */
    while (getchar() != '\n');

    /* Loops while k is not equal to 2 */
}while(k != 2);
```

Now that the function has received user input and knows which pointers the user entered, the code following would be to create the copy of the 1st pointer to tuple entered. The first step would be to increment the tuple counter by 1 followed by a calling of the `resizeArray()` function to allocate space for the new tuple to be stored. Once space is allocated, the string ID entered previously by the user is simply copied onto the new string ID location allocated on the dynamic array using the `strcpy()` function. Then the elements of the new tuple (which are currently at NULL) are simply made equal to the elements related to the first pointer, the user had entered. This process can be understood better below:

```
/* Increments counter by 1 */
++counter;

/* Calls resize array function now that an extra tuple is created */
resizeArray(tupleArray);

/* Copies the string stored in join as the string ID of the new tuple */
strcpy(tupleArray[counter-1].ID, join);

/* Copies the elements stored in the first entered tuple to the newly
created one*/
tupleArray[counter-1].element1 = tupleArray[i].element1;
tupleArray[counter-1].element2 = tupleArray[i].element2;
tupleArray[counter-1].element3 = tupleArray[i].element3;
}
```

The final 2 remaining functions simply save the tuples to a file and load the tuples from a file. Their function prototypes for `saveAllTuples()` and `loadAllTuples()` respectively are :

```
/* A function which stores all the created tuples in a folder */
void saveAllTuples(tuple * tupleArray);
```

```
/* A function which loads all the previously stored functions from a folder */  
void loadAllTuples();
```

Taking a look at `saveAllTuples()` first, this simply stored the saved (but not deleted) tuples to a file by first creating a file pointer. This file pointer was then instructed to open a file entitled "SaveTuples.txt" in write mode and if not existing, create it.

```
/* Creating a file pointer */  
FILE *fp;  
  
/* Opens/creates the file in write mode and if already exists, then data is overwritten */  
fp = fopen("SaveTuples.txt", "w");
```

Once successfully opened, a couple of strings were instructed to be printed into the file using `fputs` and a for loop was implemented to print all the tuples 1 by 1 in the folder. Once the for loop finished executing, another line of code was printed using `fputs` into the file to indicate that the file was now written to. And finally, the `fclose()` function was used to close the file pointer created at the start of the function. The whole function can be seen below:

```
/* When saveAllTuples() function is called, execution continues from here */
void saveAllTuples(tuple * tupleArray){

    /* Local variable i to be used in the for loop */
    int i;

    /* Creating a file pointer */
    FILE *fp;

    /* Opens/creates the file in write mode and if already exists, then data is overwritten */
    fp = fopen("SaveTuples.txt", "w");

    /* Prints out the following into the file */
    fputs("Tuple\tString Identifier\n", fp);
    fputs("-----\n", fp);

    /* Loops based on the number of tuples and stores their contents into this file */
    for(i = 0 ; i < counter ; i++){
        fprintf(fp, "%-5d\t%-10s\t%-6d\t%-6d\t%-6d\n", i+1, tupleArray[i].ID,
            tupleArray[i].element1, tupleArray[i].element2, tupleArray[i].element3);
    }

    /* Prints out the following into the file */
    fputs("\n-----\n", fp);

    /* Closes a stream (in this case a file) and all the structure associated with it */
    fclose(fp);

    printf("Tuples Saved Successfully!!\n\n");
}
```


Moving on to the loadAllTuples() function. Once again, another file pointer was created. This was to be used to open the file and view the stored items. However, first, space had to be allocated to load what was saved in this file. Therefore, space on the heap twice as much as many tuples created was allocated as allocating some extra space for extra characters not part of the tuple is important. Afterall, this space will eventually be freed up anyways.

```
/* Creating a file pointer */  
  
FILE *fp;  
  
/* Setting the local variable length the size of 2*(all the pointers) */  
int length = 2*(counter * (sizeof(tuple)));  
  
/* Allocating memory on the heap big enough to load contents from file */  
char * temp = (char*) malloc(sizeof(length));  
  
//space to be freed later on
```

Then an if statement was implemented to open the file in write mode. The condition inside the if statement clearly indicated that if a file entitled "SaveTuples.txt" does not exist, then the user is notified with some errors and the previously allocated memory is free just before the program returns back to main.

```
/* if the file doesn't exist the code in the block runs */  
if ((fp = fopen("SaveTuples.txt", "r")) == NULL)  
{  
    printf("Can't open \"SaveTuples.txt\" file.\n");  
    printf("No tuples have been created yet\n");  
  
    /* Closes a stream (in this case a file) and all the structure associated with it */  
    fclose(fp);  
}
```

```
/* Freeing the previously allocated memory from the heap */  
free(temp);  
  
/* Returns back to main */  
return;
```

Whilst if the file actually does exist, the program then opens it in read mode, stores its contents on the heap up till the last character in the file. It then displays the contents, closes the file pointer using `fclose()`, frees the memory previously allocated and returns back to main.

```
}else{  
    /* Opens the file in read mode */  
    fp = fopen("SaveTuples.txt", "r");  
  
    /* while the end of the file hasn't been reached, characters will be stored onto the heap*/  
    /* and then printed out through the puts function */  
    while(!feof(fp)){  
        fgets(temp, length, fp);  
        puts(temp);  
    }  
}  
putchar('\n');  
printf("-----END OF FILE-----\n");  
  
/* Closes a stream (in this case a file) and all the structure associated with it */  
fclose(fp);  
  
/* Freeing the previously allocated memory from the heap */  
free(temp);  
}
```

Task 2b

Task Specification:

An extension of what was done in 2(a) had to be implemented in this part to allow tuple object to take any structure. This brought about the need to internally store information about the particular structure in the form of a colon delimited string. Both the API and the implementation of the first part were challenged once again

Task Creation:

A New Project was created in CLion entitled 2b which consisted of a CMakeLists.txt file and a main.c source file upon creation.

The CMakeLists.txt file was the first to be modified. Its contents were standardized to match the cross-platform build system while also including the main.c file (which was renamed to tuple2b.c) be recognized as a source file. The modifications done can be seen below:

```
1 cmake_minimum_required(VERSION 3.12)
2 project(2b C)
3
4 set(CMAKE_C_STANDARD 99)
5 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall")
6
7 set(SOURCE_FILES1 tuple2b.c )
8 add_executable(2b ${SOURCE_FILES1})
```

An associated header file related to this file was also created to be able to keep hold of the global variables, the defined macros and the function prototypes. This was named *tuple2b.h*. This was then called from the *tuple2b.c* source file along with the list of libraries which had to be included in this project just like what was done in 2a. The reason why a separate CMake project was created for this is to ensure that nothing changed for 2b would affect 2a, given that the implementation for both had to be different. All the progress done in 2a throughout the duration of this task was looked into and any relevant pieces of code were copied onto the 2b project.

Now that a tuple could take any structure, it was even harder to draw a line over the right form of tuple to be used. After hours spent researching on what is best to be used, it was decided that a structure was not going to be used for this part. This decision was taken simply because all the elements in the structure could be different. This process was planned out on paper and seemed to be possible however was scrapped off the list shortly after when the createTuple() function seemed to be longer than 500 lines of code. Thus, the second best option on the list was implemented. This consisted of creating a tuple and a union to be used together. This option was tested on paper and it seemed to be the most feasible option, and so it was decided.

The struct defined now consisted of completely different attributes in various ways. This can be seen below alongside the struct used which was also implemented in the struct:

```
/* Creating a struct for the tuples to
be created */
typedef struct tuple_t{

    char ID[LENGTH];
    char colon[19];
    type elements[9];
    int howMany;

}tuple;
```

```
/* Creating a union of elements to be used in
the struct tuple_t */
typedef union type_t{

    char c;
    int i;
    long int l;
    char s[10];
    float f;
    double d;
```

The character array ID was the first to be changed. The LENGTH macro was defined to 80 instead of 15 to allow the user to enter more characters into the string ID

```
/* Defining the macro LENGTH to 80 to be used as a constant for the ID array size */
#define LENGTH 80
```

The second item in the list is another character array of max 19 characters. This was set to store the colon delimited string that was to be automatically generated by the program itself. It was decided prior to starting to code 2b that the maximum elements that can be entered by the user in 1 tuple is 9. This would occupy a total of 19 characters in a colon delimited string, which is exactly why this value was given to the string. Moving on, the elements of type type consisted of the amount of elements to be stored in a tuple. This meant that the user could decide to select up to 9 different types of elements from the struct list or he could simply select 1. And last but not least, the integer variable howMany stores the amount of different elements the user will be dealing with in a tuple.

The global variable referred to as 'counter' before was also changed, this time to tuplec to distinguish easily between task 2a and 2b.

```
/* Setting the global variable tuplec to be 0 when the program is started */  
int tuplec = 0;
```

The resizeArray() function was kept the same and for this reason, it will be only mentioned.

The createTuple() function, however, changed drastically. Apart from the if statement which called the resizeArray() function, and the for loop which would loop to check if there is another array with the same string ID so that it is deleted, the createTuple() function now had a whole different layout. First the user was asked to enter how many elements he intends to store in the tuple and then this value is used in a variety of different ways. Before creating these different ways however, validation was done so that no user enters a character or a value greater than the specified range.

```
printf("\n-----Creating Tuple #%-d----- \n\n", tuplec);

printf("How many elements would you like to store in this tuple? \n");

printf("(Not less than 1 and not more than 9)\n");

/* If something other than an integer is entered or a value which is larger than 9 */
/* or smaller than 1, this while loop runs prompting the user to enter the appropriate
data type and value */

while((scanf(" %d", &tupleArray[tuplec-1].howMany) != 1) || (tupleArray[tuplec-1].howMany < 1) || (tupleArray[tuplec-1].howMany > 9)){

printf("Invalid data!! \nPlease enter a number from 1 - 9 : \n");

/* Clears input buffer from previously entered invalid data */

while(getchar()!='\n');

}
```

First, the value entered in howMany is converted into an integer and stored as the first character of the colon delimited string.

```
/* A function which converts the integer value entered by the user to a null-terminated string */
itoa(tupleArray[tuplec-1].howMany, tupleArray[tuplec-1].colon, 10);
```

Then, a legend was printed out and this value was inputted inside a for loop to loop based on the amount of types a different element was to be created. At this stage, the user had to simply press a character available on the legend to be able to store an element related to that particular data type. Validation was also performed on this part so as to ensure that the user enters the correct characters available from the legend.

```

/* A for loop runs based on the value howMany entered by the user */
for(i = 0 ; i < tupleArray[tuplec-1].howMany ; i++) {
    printf("ELEMENT #%d: \n", i + 1);
    printf("What data type do you want to store?\n");
    printf("Enter (c/i/l/s/f/d)\n");

    /* Clears input buffer from previously entered data */
    while (getchar() != '\n');

    /* Validation is performed to ensure that the user enters the available options
    */
    while ((scanf(" %c", &type) != 1) || ((type != 'c') && (type != 'i') && (type != 'l')
        && (type != 's') && (type != 'f') && (type != 'd')) {

        printf("Invalid input!! \nPlease enter a character from the legend above : \n");

        /* Clears input buffer from previously entered invalid data */
        while (getchar() != '\n');
    }
}

```

After entering a valid character, the whole while loop is skipped and an if statement determines what data type the user intends to store. Proper validation has been induced on all the different data types except for that of a string as anything can be stored in a string after all. Based on the data type entered, just before exiting the for loop the strcat() function is used to concatenate that particular data type to the rest of the colon delimited string.

The following are the for loop parts for:

- A character

```
/* if user enters 'c', this part of the if statement will run */
if (type == 'c') {

    printf("Enter the character: ");

    /* Clears input buffer from previously entered data */
    while (getchar() != '\n');

    /* If something other than a character is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while (scanf(" %c", &tupleArray[tuplec-1].elements[i].c) != 1) {
        printf("\nInvalid info!! \nPlease enter a character : \n");

        /* Clears input buffer from previously entered data */
        while (getchar() != '\n');
    }

    /* Concatenates :c to the rest of the colon delimited string */
    strcat(tupleArray[tuplec-1].colon, ":c");
```


- An integer

```
/* else if user enters 'i', this part of the if statement will run */
} else if (type == 'i') {

    printf("Enter the integer: ");

    /* Clears input buffer from previously entered data */
    while (getchar() != '\n');

    /* If something other than an integer is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while (scanf(" %d", &tupleArray[tuplec-1].elements[i].i) != 1) {
        printf("\nInvalid data!! \nPlease enter an integer : \n");

        /* Clears input buffer from previously entered invalid data */
        while (getchar() != '\n');
    }

    /* Concatenates :c to the rest of the colon delimited string */
    strcat(tupleArray[tuplec-1].colon, ":i");
```

- A long integer

```
/* else if user enters 'l', this part of the if statement will run */
    } else if (type == 'l') {

        printf("Enter the long integer: ");

        /* Clears input buffer from previously entered data */
        while (getchar() != '\n');

        /* If something other than a long integer is entered, this while loop runs prompting*/
        /* the user to enter the appropriate data type */
        while (scanf("%ld", &tupleArray[tuplec-1].elements[i].l) != 1) {
            printf("\nInvalid data!! \nPlease enter a long integer : \n");

            /* Clears input buffer from previously entered invalid data */
            while (getchar() != '\n');
        }

        /* Concatenates :l to the rest of the colon delimited string */
        strcat(tupleArray[tuplec-1].colon, ":l");
```

- A string

```
} else if (type == 's') {  
  
    printf("Enter the string: \n");  
    printf("(Do not enter more than 10 characters)\n");  
  
    /* Clears input buffer from previously entered data      */  
    while (getchar() != '\n');  
  
    /* Stores what the user enters up till the 10th character to the tupleArray */  
    fgets(tupleArray[tuplec-1].elements[i].s, 10, stdin);  
  
    /* Concatenates :s to the rest of the colon delimited string */  
    strcat(tupleArray[tuplec-1].colon, ":s");
```

- A float

```
} else if (type == 'f') {

    printf("Enter the float value: ");

    /* Clears input buffer from previously entered data */
    while (getchar() != '\n');

    /* If something other than a float is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while (scanf(" %f", &tupleArray[tuplec-1].elements[i].f) != 1) {
        printf("\nInvalid data!! \nPlease enter a float : \n");

        /* Clears input buffer from previously entered invalid data */
        while (getchar() != '\n');
    }

    /* Concatenates :f to the rest of the colon delimited string */
    strcat(tupleArray[tuplec-1].colon, ":f");
```

- A double value

```
} else if (type == 'd') {

    printf("Enter the double value: ");

    /* Clears input buffer from previously entered data */
    while (getchar() != '\n');

    /* If something other than a double value is entered, this while loop runs prompting*/
    /* the user to enter the appropriate data type */
    while (scanf("%lf", &tupleArray[tuplec-1].elements[i].d) != 1) {
        printf("\nInvalid data!! \nPlease enter a double value : \n");

        /* Clears input buffer from previously entered invalid data */
        while (getchar() != '\n');
    }

    /* Concatenates :d to the rest of the colon delimited string */
    strcat(tupleArray[tuplec-1].colon, ":d");
```

Moving on to the next function, the `getTupleById()` function remained the same as before and so would only be mentioned in this case. Same goes for the `getTupleID()` function aswell.

The `showTuple()` function had to be changed for sure. Since the format of the tuple struct was different, so the format of this method had to change too. Apart from the fact that most of the variables used in it were kept the same, a char pointer called *delim* was created. Its purpose would be to point to character/s which have been separated with the help of the `strtok()` function. This function takes 2 paramaters, the first, the string (or pointer to string) whilst the second, the character/s to be used as splitting pointers. The scenario used for `showTuple()` can be seen below:

```
delim = strtok(tupleArray[i].colon, ":");  
delim = strtok(NULL, ":");
```

Here, *delim* is being set to separate the colon delimited string for `tupleArray[i]` and the colon is set as the separator. The remaining characters coming out of this colon delimited string go through a for loop which loops for all the elements in that particular tuple. Following it is an if statement which runs if all the separators have not been skipped. Following it is another nested if statement through which they are tested based on their type and based on the dereferenced value of *delim*. This is done to print out the actual value stored in `element1` of that particular tuple and so on....

```
for (j = 0; j < tupleArray[i].howMany; j++) {

    if (delim != NULL) {

        printf("%s : ", delim);

        if(*delim == 'c'){

            printf(" %c \n", tupleArray[i].elements[j].c);

        }else if(*delim == 'i'){

            printf(" %d \n", tupleArray[i].elements[j].i);

        }else if(*delim == 'l'){

            printf(" %ld \n", tupleArray[i].elements[j].l);

        }else if(*delim == 's'){

            printf(" %s \n", tupleArray[i].elements[j].s);

        }else if(*delim == 'f'){

            printf(" %f \n", tupleArray[i].elements[j].f);

        }else if(*delim == 'd') {

            printf(" %lf \n", tupleArray[i].elements[j].d);

        }

        //This make the pointer of delim skip after the next separator

        delim = strtok(NULL, ":");

    }

}
```

Moving on to the deleteTuple() function, apart from the previously created variables, another variable 'j' is initialized and a pointer 'delim' to character is created once again. Like before, in the deleteTuple() function there was an if statement which compared 2 strings through strcmp(). From that moment onwards, the code for 2b is very different when compared to 2a. Now that the tuple has a different structure, the code in the while part of the loop right after the if statement was changed to the following:

```
while (i < tuplec) {  
    strcpy(tupleArray[i].ID, tupleArray[i+1].ID);  
    strcpy(tupleArray[i].colon, tupleArray[i+1].colon);  
    tupleArray[i].howMany = tupleArray[i+1].howMany;
```

These simply copy the values and strings stored into the tuple above onto the tuple below to overwrite the previous data. Up till now everything in the deleteTuple() function was easily relatable. However, to be able to copy the actual elements of i+1 into i required more than just a simple line of code. Once again the delim character pointer is used to separate the tupleArray[i].colon into characters and colons and returns only the colons. A for loop is then implemented to loop through all the elements available in the tuple followed by an if statement loops each time up till the last colon delimited character. This also contains a nested if statement which dereferences the values stored in delim to know how to properly assign the elements of i to be equal to the elements stored in i+1. This can be understood better by taking a look at the snippet:

```
delim = strtok(tupleArray[i].colon, ":");  
  
    //loops the amount of elements stored in i+1  
    for(j = 0 ; j < tupleArray[i+1].howMany ; j++){  
        //runs only if the value pointed to by delim is != NULL  
        if (delim != NULL) {  
  
            //if the dereferenced value of delim == c run this  
            if(*delim == 'c'){
```



```
tupleArray[i].elements[j].c = tupleArray[i+1].elements[j].c;

//if the dereferenced value of delim == i run this
}else if(*delim == 'i'){

    tupleArray[i].elements[j].i = tupleArray[i+1].elements[j].i;

    //if the dereferenced value of delim == l run this
}else if(*delim == 'l'){

    tupleArray[i].elements[j].l = tupleArray[i+1].elements[j].l;

    //if the dereferenced value of delim == s run this
}else if(*delim == 's'){

    strcpy(tupleArray[i].elements[j].s, tupleArray[i+1].elements[j].s);

    //if the dereferenced value of delim == f run this
}else if(*delim == 'f'){

    tupleArray[i].elements[j].f = tupleArray[i+1].elements[j].f;

    //if the dereferenced value of delim == d run this
}else if(*delim == 'd') {

    tupleArray[i].elements[j].d = tupleArray[i+1].elements[j].d;

}

delim = strtok(NULL, ":");
```

After the above processes are executed, then the tuple count is decremented by 1 and the array is resized to a smaller size to let go of the non- used slots. Finally the memory reserved on the heap for *del* is freed and the program returns back to main.

```
/* The tuple count is decremented by 1 */  
    --tuplec;  
  
/* The resizeArray() function is called once more to remove the extra space */  
    resizeArray(tupleArray);  
  
/* The previously allocated space on the heap is now freed */  
    free(del);  
  
/* Program is returned back to the main function */  
    return;
```

Moving on to the next function, the `cmpTuples()` function which unlike before allocates memory on the heap to store what the user enters for `cmp1` and `cmp2`. This is done to show that by creating an array of size `LENGTH` or by simply allocating enough memory on the heap, both ways lead to the same outcome. This function remains the same also when taking input from the user for both `cmp1` and `cmp2`. However, upon passing this part, memory for `cmp1` and `cmp2` is freed as it is no longer needed to be used. Right after, an if statement checking for compatibility of the tuples to be compared runs if the string delimiters of both tuples being compared is the same.

```
if(strcmp(tupleArray[i].colon, tupleArray[j].colon) == 0){  
  
    printf("The tuples %s and %s are of a compatible type to be compared", tupleArray[i].ID, tupleArray[j].ID);  
  
    str1 = strtok(tupleArray[i].colon, ":");  
    for (m = 0; m < tupleArray[i].howMany; m++) {
```

Then, a similar method to before is used where a for loop, loops up to the total number of elements and the colon delimited strings are separated with `strtok`. The dereferenced value stored in `str` then determines which part of the if statement to be executed and 1 is returned if the value stored in `i > m`. If the case is the contrary, then -1 is returned and if `i == m`, nothing is returned but a variable `n` is simply incremented by 1 so that if in the end `n == total number of elements`, this means that the value of the elements stored in `i` and in `m` are the same. Only one case is shown below as the either cases are relatively similar (except for s).

```
if (str1 != NULL) {

    str1 = strtok(NULL, ":");

    if(*str1 == 'c'){

        if(tupleArray[i].elements[m].c > tupleArray[j].elements[m].c){
            return 1;
        }else if(tupleArray[i].elements[m].c < tupleArray[j].elements[m].c){
            return -1;
        }else if(tupleArray[i].elements[m].c == tupleArray[j].elements[m].c){
            n++;
            if(n == tupleArray[j].howMany) {
                return 0;
            }
            continue;
        }
    }
}
```

On the contrary, if the tuple are not of a compatible type to be compared, then a ridiculously large value is returned and the user is informed.

```
}else{

    printf("Tuples are of different types and therefore cannot be compared!\n");
    return 9999999;
```

The joinTuples() function in this case is by far the longest function in all of this program. It basically prompts the user to enter 2 pointers to existing tuples and uses validation to check their authentication. After that, an if statement checks whether joining both tuples would create a tuple having more elements than required. If so, the user is informed, the memory allocated previously freed and the user is returned back to main.

```
if((tupleArray[i].howMany + tupleArray[j].howMany) > 10 ){  
  
    free(join);  
  
    printf("The tuples cannot be joined since their elements together exceed 9, the max expected size");  
  
    return;
```

If not, the following functions are then executed:

```
}else{  
  
    ++tuplec;  
    resizeArray(tupleArray);  
    strcpy(tupleArray[tuplec-1].ID, join);  
    tupleArray[tuplec-1].howMany = tupleArray[i].howMany + tupleArray[j].howMany;  
  
    /* A function which converts the integer value for the size to a string */  
    itoa(tupleArray[tuplec-1].howMany, tupleArray[tuplec-1].colon, 10);  
  
    str1 = strtok(tupleArray[i].colon, ":");  
    str2 = strtok(tupleArray[j].colon, ":");
```

The tuple count is incremented since another tuple is going to be created out of the 2 existing ones and the `resizeArray()` function is called to make space for an extra tuple. The string ID entered by the user is automatically added as this tuples' new string ID and the `howMany` element value is equal to both the values of both tuples added. The `itoa` function is used again to convert an integer `howMany` into a string and `str1` and `str2` are pointers pointing to characters separated by colons. Then a for loop runs up till the number of elements stored in the first tuple to be joined and associated if statements run if the current dereferenced value stored in `str1` is equal to that particular character. If so, the code within the block executes which simply adds the values stored in `tupleArray[tuplec-1].elements[i]` onto the new tuple in their elements `tupleArray[tuplec-1].elements[m]`. In this if statement also, based on the variable type, a corresponding `'type'` is concatenated to the end of the colon delimited string to form a new one out of the two. Showing case c only:

```
for(m = 0 ; m < tupleArray[i].howMany ; m++){
    //doesn't loop if str1 == NULL
    if (str1 != NULL) {

        str1 = strtok(NULL, ":");
        //loops if the dereferenced value == c
        if(*str1 == 'c'){
            //adds elements in i to elements in c
            tupleArray[tuplec-1].elements[m].c = tupleArray[tuplec-1].elements[i].c;

            /* Concatenates :c to the rest of the colon delimited string */
            strcat(tupleArray[tuplec-1].colon, ":c");
```

The same is done for all the remaining elements (except for 's' which use `strcpy` instead to assign the string they are storing onto something else). This process is then repeated once again, the time for `tupleArray[j]` since this is the second tuple to be joined. This time, the for loop does not start from 0 not to overwrite the already written elements but starts from the value of `howMany` of the previous tuple as show:

```
for(m = tupleArray[i].howMany ; m < tupleArray[j].howMany ; m++){
```

Moving on to the next function, we reach the final function which has been properly modified. This is the `saveAllTuples()` function which unlike the `loadAllTuples()` function was changed in a way that the changes must be documented. Compared to task 2a, this contains the same file pointers being opened and closed. However, it makes use of a character pointer called *delim*.

```
/* A character pointer */  
char * delim;
```

When storing the tuple, the general structure of how the tuples are saved was changed in this file simply because the string ID and other elements were subject to change. In this particular case, the string ID LENGTH was changed from 15 to 80 chars, allowing the user to store more into the character array. But apart from this change, the tuples now contained more than just 3 elements of the same type. These could now contain up to 9 elements with different types. Therefore, the structure for saving these was adapted to store them in a more aesthetic manner while being orderly and neat at the same time. The first few characteristics were easily saved as:

```
/* Loops based on the number of tuples and stores their contents into this file*/  
for(i = 0 ; i < tuplec ; i++){  
  
    /* Prints out the following into the file */  
    fprintf(fp, "Tuple %d \n", i+1);  
    fprintf(fp, "String ID: %s \n", tupleArray[i].ID);  
    fprintf(fp, "Colon Delimited String : %s \n", tupleArray[i].colon);
```

However, to store the different types and the values they stored, for loop needed to be used to loop based on how many elements were present. This loop also had composed in it an if statement which also had a nested if statement too. Based on

the dereferenced value returned by the pointer `delim`, the actual data type of the element to be stored in the file is selected and the value is stored there too. Showing only up till case `i` below:

```
/* Loops as many times as howMany*/
for(j = 0 ; j < tupleArray[i].howMany ; j++){

    if (delim != NULL) {

        /* Prints out the following into the file */
        fprintf(fp, "\nElement %d : ", j+1);

        delim = strtok(NULL, ":");

        /* If the dereferenced char stored in delim is equivalent to 'c', */
        /* the code within the block is executed */
        if(*delim == 'c'){

            fprintf(fp, " %c \n", tupleArray[i].elements[j].c);

            /* If the dereferenced char stored in delim is equivalent to 'i', */
            /* the code within the block is executed */
        }else if(*delim == 'i'){

            fprintf(fp, " %d \n", tupleArray[i].elements[j].i);
```


Testing 2

In General:

Now that 2a and 2b were successfully implemented as documented, it was time to test both systems with some real data. Before testing was implemented, a testdriver source and header file was entered in each of 2a and 2b.

testdriver2a.c (for2a) or testdriver2b.c (for2b) contains the main function. It includes both tuple2a.c (for2a) (or tuple2b.c for b) and the associated header. The header files tesdriver2a.h(for 2a) and tesdriver2b.h(for 2b) only contain a main function and it's prototype which are simply called from the main each time. The testdriver source file contains a basic switch case scenario which based on the input of the user, implements the functions accordingly It simply starts off by calling the createTuple() functions twice in both cases and then displays the menu which is looped for as long as the user wants.

One final thing, the cmakelists was set to compile and run just the testdriver source file given that everything else was included into this file.

Without further redo, let's get to testing...

Testing 2a

The testdriver2a.c was compiled successfully, as can be seen:

```
Scanning dependencies of target 2a
[ 50%] Building C object CMakeFiles/2a.dir/testdriver2a.c.obj
[100%] Linking C executable 2a.exe
[100%] Built target 2a

Build finished
```

This was then executed and the following menu was displayed.

```
Creating Tuple #1:

Please enter the tuple's string identifier (not more than 15 characters) :
|
```

Some data was entered for the first tuple.

```
Please enter the tuple's string identifier (not more than 15 characters) :
Cars
Cars
Now enter the elements going to be stored in TUPLE #1
You must enter 3 elements for each tuple

Please enter element #1:23
23
Please enter element #2:34
34
Please enter element #3:45
45
Creating Tuple #2:
```

This data seemed to be entered and stored successfully. What about entering a string identifier longer than 15 characters now and testing a bit the validation tests for the elements?

```
Creating Tuple #2:

Please enter the tuple's string identifier (not more than 15 characters) :
Telecommunications
Telecommunications
Now enter the elements going to be stored in TUPLE #2
You must enter 3 elements for each tuple

Please enter element #1: Invalid data!!
Please enter an integer :
1
1
Please enter element #2: f
f
Invalid data!!
Please enter an integer :
we
we
Invalid data!!
Please enter an integer :
sf
sf
Invalid data!!
Please enter an integer :
34
34
Please enter element #3: 76
76
```

After entering “Telecommunications” (19chars long) as the 2nd tuple name, the program stored the first 15 characters(14 + \0) and kept the remaining 4 characters running in there. Since no function to clear the buffer was included, then, upon proceeding to enter the first element to be stored, an error is displayed for element 1 stating that invalid data has been entered. This was then patched for 2b. This is because the input buffer was still occupied with extra unwanted characters. Moving on, a value for element 1 is entered and stored correctly. Then element 2 was tested with 3 string inputs but all were denied and an error was given. The 4th value entered, being an integer, was saved to the storage location for element 2. Element 3 was then entered successfully.

The menu is now displayed to us for the first time.

```
Tuples Program
-----
Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 2
```

Option 2 is now selected and Cars was entered. This was successful

```
CHOICE:2
2

Please enter the tuple string ID
Cars
Cars
Tuple #1 found!!
This tuple has a pointer 00661218

Tuples Program
-----
```

Testing option 2 was again with what should be the stored string ID of Tuple #2 : "Telecommunicati" (15 chars long). This was successful once again.

```
CHOICE:2
2

Please enter the tuple string ID
Telecommunicati
Telecommunicati
Tuple #2 found!!
This tuple has a pointer 00661234

Tuples Program
-----
```

Creating a new Tuple "Dog: while testing float incorrect input and char-int input and int-char input :

```
Creating Tuple #3:

Please enter the tuple's string identifier (not more than 15 characters) :
Dog
Dog
Now enter the elements going to be stored in TUPLE #3
You must enter 3 elements for each tuple

Please enter element #1: fsdf34
fsdf34
Invalid data!!
Please enter an integer :
23sdf
23sdf
Please enter element #2: Invalid data!!
Please enter an integer :
43.33
43.33
Please enter element #3: Invalid data!!
Please enter an integer :
?
7
```

Upon entering chars-ints, the system rejects the whole string.

Upon entering ints-chars, the system accepts the ints and keeps the chars in the input buffer ultimately resulting in an error due to buffer dumping. However 23 is still stored as expected.

Upon entering a float, 43.33 is rejected as a float however 43 is stored and '.33' are carried onto the next process ultimately resulting in an error.

Testing now option 3. Twice invalid input, third time correct. This is successful.

```
CHOICE: 3
3

Enter the pointer of the tuple to get its ID:
dfwef
dfwef
Tuple with memory address 0061FEE8 not found.
Please try again.

Enter the pointer of the tuple to get its ID:
0393483494
0393483494
Tuple with memory address 0061FEE8 not found.
Please try again.

Enter the pointer of the tuple to get its ID:
00661218
00661218
Tuple #1 found!!
This tuple has an ID : Cars
```

Tuples Program

Testing option 4 with invalid input. Returns back to menu without displaying anything as expected

```
CHOICE: 4
4

Enter the tuple's pointer for the tuple you want to display:
hw23r982r
hw23r982r
```

Tuples Program

Please choose one of the following options below:

- 1) Create a Tuple
- 2) Get a Tuple's Pointer
- 3) Get a Tuple's ID
- 4) Display Tuple
- 5) Delete Tuple
- 6) Compare Tuples
- 7) Join Tuples
- 8) Save all Tuples
- 9) Load all Tuples
- 0) EXIT

Tuples created up till now: 3

Testing Option 4 now with valid input. Output is as expected.

```

Please enter a number from 0 - 9 :
4
4

Enter the tuple's pointer for the tuple you want to display:
00661218
00661218
Tuple   String Identifier      Elements
-----
#1      Cars                    (23, 34, 45)
-----

Tuples Program
-----

```

Testing Option 5. Tuple Successfully Deleted

```

Tuples created up till now: 3

CHOICE:5
5

Enter the string identifier of the tuple you want to delete:
Telecommunicati
Telecommunicati
Tuple #2 found!!
Deleting Telecommunicat containing (1, 34, 76)

Tuples Program
-----
Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 2

```

Trying to return memory address of deleted pointer. Output is successful

```
Please enter a number from 0 - 9 :  
2  
2  
  
Please enter the tuple string ID  
Telecommunicati  
Telecommunicati  
Tuple with string ID Telecommunicat not found.  
Please try again.  
  
Please enter the tuple string ID  
|
```

Testing now option 6. Tested successfully, output as expected

```
CHOICE: 6  
6  
  
Please enter the 1st string to compare:  
asdjakj  
asdjakj  
Please enter the 1st string to compare:  
223r23r  
223r23r  
Please enter the 1st string to compare:  
Dog  
Dog  
Tuple #2 found!!  
Please enter the 2nd string to compare:  
Cars  
Cars  
Tuple #1 found!!  
The value returned is 1
```

Testing now option 6 with 2 equal in elements tuples:


```
Creating Tuple #3:
```

```
Please enter the tuple's string identifier (not more than 15 characters) :
```

```
Man
```

```
Man
```

```
Now enter the elements going to be stored in TUPLE #3
```

```
You must enter 3 elements for each tuple
```

```
Please enter element #1:
```

```
1
```

```
Please enter element #2:
```

```
1
```

```
Please enter element #3:
```

```
1
```

```
Creating Tuple #4:
```

```
Please enter the tuple's string identifier (not more than 15 characters) :
```

```
Woman
```

```
Woman
```

```
Now enter the elements going to be stored in TUPLE #4
```

```
You must enter 3 elements for each tuple
```

```
Please enter element #1:
```

```
1
```

```
1
```

```
Please enter element #2:
```

```
1
```

```
Please enter element #3:
```

```
1
```

```
CHOICE:6
```

```
6
```

```
Please enter the 1st string to compare:
```

```
Man
```

```
Man
```

```
Tuple #3 found!!
```

```
Please enter the 2nd string to compare:
```

```
Woman
```

```
Woman
```

```
Tuple #4 found!!
```

```
The value returned is 0
```

Testing now option 7:

With Man(1,1,1) and Woman(1,1,1)

```
CHOICE:7
7

Enter the String ID of the new tuple to be created:
Baby
Baby
Enter the 1st tuple's pointer:
00031218
00031218
Tuple #1 found!!
Enter the 2nd tuple's pointer:
00031234
00031234
Tuple #2 found!!

Tuples Program
-----
```

Created a new tuple Baby by copying Man(1,1,1)

To check:

```
Tuples created up till now: 3

CHOICE:2
2

Please enter the tuple string ID
Baby
Baby
Tuple #3 found!!
This tuple has a pointer 00031250
```

```
CHOICE:4
4

Enter the tuple's pointer for the tuple you want to display:
00031250
00031250
```

Tuple	String Identifier	Elements
#3	Baby	(1, 1, 1)

```
-----
Tuples Program
```

Only Man, Woman and Baby present to make it simple:

Testing now option 8:

```
CHOICE: 8
8

Tuples Saved Successfully!!
```

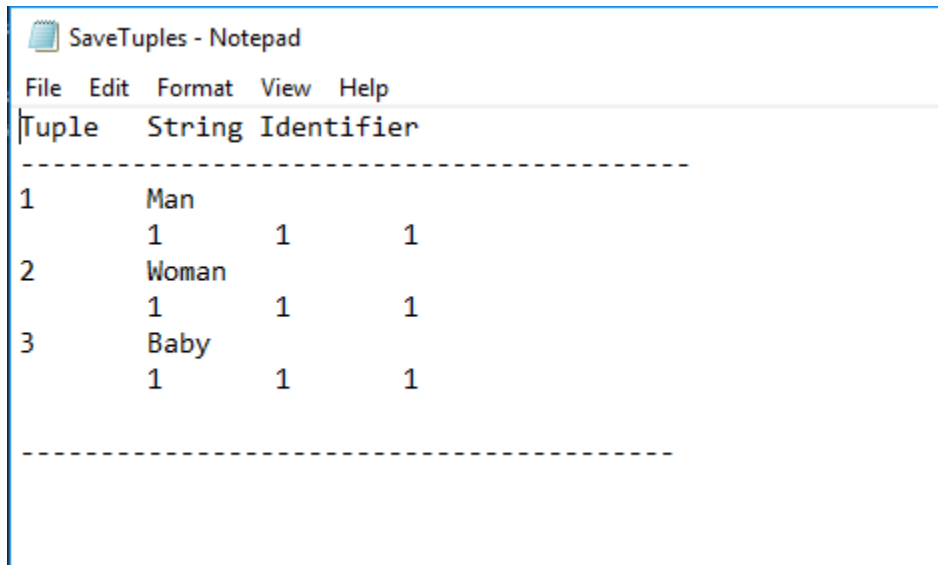
Testing now Option9:

```
Tuples created up till now: 3

CHOICE: 9
9

Tuple      String Identifier
-----,-----
1          Man
          1          1          1  □
2          Woman
          1          1          1  □
3          Baby
          1          1          1  □
-----,-----
```

The actual "SaveTuples.txt"



The screenshot shows a Notepad window with the title 'SaveTuples - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text content is as follows:

Tuple	String	Identifier
1	Man	1
2	Woman	1
3	Baby	1

Testing option 0:

```
CHOICE:0
0

Bye!
--End of program--

Process finished with exit code 0
```

Testing 2b

The testdriver2a.c was compiled successfully, as can be seen:

```
Scanning dependencies of target 2b
[ 50%] Building C object CMakeFiles/2b.dir/testdriver2b.c.obj
[100%] Linking C executable 2b.exe
[100%] Built target 2b

Build finished
```

This was then executed and the following was displayed:

```
Please enter the tuple's string identifier (not more than 80 characters) :
|
```

The following input was entered into the program:

```
Please enter the tuple's string identifier (not more than 80 characters) :
I will not enter more than 80
I will not enter more than 80

-----Creating Tuple #1-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)

3
3

-----
                LEGEND
    Enter 'c' for char
    Enter 'i' for int
    Enter 'l' for long int
    Enter 's' for string
    Enter 'f' for float
    Enter 'd' for double
-----

ELEMENT #1:
```

```

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
c
c
Enter the character:q
q
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:23
23
ELEMENT #3:
What data type do you want to store?
Enter (c/i/l/s/f/d)
l
l
Enter the long integer:456456
456456

```

```

Please enter the tuple's string identifier (not more than 80 characters) :
I will try my best to store more than 80 words so that this program is being tested to the full
I will try my best to store more than 80 words so that this program is being tested to the full

-----Creating Tuple #2-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
Invalid data!!
Please enter a number from 1 - 9 :
5
5
-----
LEGEND
Enter 'c' for char
Enter 'i' for int
Enter 'l' for long int
Enter 's' for string
Enter 'f' for float
Enter 'd' for double
-----

```

```
-----  
ELEMENT #1:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
c  
c  
Enter the character:k  
k  
ELEMENT #2:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
i  
i  
Enter the integer:1  
1  
ELEMENT #3:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
s  
s  
Enter the string:  
(Do not enter more than 10 characters)  
Man ok  
Man ok  
ELEMENT #4:
```

```
Please enter a character from the legend above :  
f  
f  
Enter the float value:345,4  
345,4  
ELEMENT #5:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
d  
d  
Enter the double value:456456456.45  
456456456.45
```

Tuples Program

```
-----
```

A menu is now displayed which is like the menu displayed for 2a

```
Tuples Program
-----
Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 2

CHOICE:
```

Another 2 Tuples are created:

```
Tuples created up till now: 2

CHOICE: 1
1

Please enter the tuple's string identifier (not more than 80 characters) :
Apricot
Apricot

-----Creating Tuple #3-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
2
2

-----
LEGEND
Enter 'c' for char
Enter 'i' for int
Enter 'l' for long int
Enter 's' for string
Enter 'f' for float
Enter 'd' for double
-----

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
l
1
Enter the long integer: 234567
234567
```



```

Enter the long integer:234567
234567
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)
c
c
Enter the character:]
j

                          Tuples Program
-----

Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 3

CHOICE:Small
Small
Ivalid data!!
Please enter a number from 0 - 9 :
j

```

```

Please enter the tuple's string identifier (not more than 80 characters) :
Small
Small

-----Creating Tuple #4-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
j
3

-----

                          LEGEND
Enter 'c' for char
Enter 'i' for int
Enter 'l' for long int
Enter 's' for string
Enter 'f' for float
Enter 'd' for double
-----

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
c
c
Enter the character:q
q
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)

```

```
-----  
  
ELEMENT #1:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
c  
c  
Enter the character:q  
q  
ELEMENT #2:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
c  
c  
Enter the character:w  
w  
ELEMENT #3:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
ce  
ce  
Enter the character:l  
l  
  
Tuples Program  
-----
```

Trying out option 2 for Tuples #3 and #4

```
Tuples Program  
-----  
Please choose one of the following options below:  
1) Create a Tuple  
2) Get a Tuple's Pointer  
3) Get a Tuple's ID  
4) Display Tuple  
5) Delete Tuple  
6) Compare Tuples  
7) Join Tuples  
8) Save all Tuples  
9) Load all Tuples  
0) EXIT  
  
Tuples created up till now: 4  
  
CHOICE:2  
2  
  
Enter the string identifier of the tuple:  
Small  
Small  
Tuple #4 found!!  
This tuple has a pointer 00A010D8
```

Writing a string ID in the menu does not confuse the program due to proper validation being enforced. Tuple pointer successfully returned.

```

Tuples Program
-----
Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 4

CHOICE: Apricot
Apricot
Invalid data!!
Please enter a number from 0 - 9 :
2
2

Enter the string identifier of the tuple:
Apricot
Apricot
Tuple #3 found!!
This tuple has a pointer 00A00FD8

```

Testing out choice 3 with Tuples #3 and #4. Validation implemented correctly!

```

Tuples created up till now: 4

CHOICE: 3
3

Enter the pointer of the tuple to get its ID:
sdfsdf
sdfsdf
Tuple with memory address 0061FEE8 not found.
Please try again.

Enter the pointer of the tuple to get its ID:
00A00FD8
00A00FD8
Tuple #3 found!!
This tuple has an ID : Apricot

Tuples Program
-----

```

Successful Once Again!

```
Tuples created up till now: 4

CHOICE:3
3

Enter the pointer of the tuple to get its ID:
00A010D8
00A010D8
Tuple #4 found!!
This tuple has an ID : Small

Tuples Program
-----
Please choose one of the following options below:
```

Testing out option 4 with Tuples #3 and #4

```
Enter the tuple's pointer for the tuple you want to display:
00A00FD8
00A00FD8
-----
Tuple # 3
String ID : Apricot

Colon Delimited String: 2:1:c
There are a total of 1 elements stored in this tuple
These are:
1 : 234567
c : j
-----
```

```
Tuples Program
-----
```

```
Tuples created up till now: 4

CHOICE: 4
4

Enter the tuple's pointer for the tuple you want to display:
00A010D8
00A010D8
-----

Tuple # 4
String ID : Small

Colon Delimited String: 3:c:c:c
There are a total of c elements stored in this tuple
These are:
c : q
c : w
c : l
-----
```

Testing option 5 for Tuple #3:

```
Tuples created up till now: 4

CHOICE: 5
5

Enter the string identifier of the tuple you want to delete:
Apricot
Apricot
Tuple #3 found!!
Deleting Apricot

-----
Tuples Program
-----

Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 3
```

Checking if Apricot still exists elsewhere:

```
Tuples created up till now: 3  
  
CHOICE: 2  
2  
  
Enter the string identifier of the tuple:  
Apricot  
Apricot  
Tuple with string ID Apricot  
not found.  
Please try again.
```

Apricot not found. Deletion successful

Checking what happened to pointer that used to belong to Apricot

```
Tuples created up till now: 3  
  
CHOICE: 3  
3  
  
Enter the pointer of the tuple to get its ID:  
00A00FD8  
00A00FD8  
Tuple #3 found!!  
This tuple has an ID : Small
```

It appears that when there was the overwriting shift, the tuple belonging to Apricot got transferred to Small instead

Creating another 2 Tuples while testing the system more for data validation:

```
Tuples created up till now: 3

CHOICE: 1
1

Please enter the tuple's string identifier (not more than 80 characters) :
New
New

-----Creating Tuple #4-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
345
345
Invalid data!!
Please enter a number from 1 - 9 :
62
62
Invalid data!!
Please enter a number from 1 - 9 :
0
0
Invalid data!!
Please enter a number from 1 - 9 :
10
10
Invalid data!!
Please enter a number from 1 - 9 :
3
3
```

```
-----  
                        LEGEND  
Enter 'c' for char  
Enter 'i' for int  
Enter 'l' for long int  
Enter 's' for string  
Enter 'f' for float  
Enter 'd' for double  
-----  
  
ELEMENT #1:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
i  
i  
Enter the integer:1  
1  
ELEMENT #2:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
2  
2  
Ivalid input!!  
Please enter a character from the legend above :  
i  
i  
Enter the integer:2  
2  
ELEMENT #3:  
What data type do you want to store?  
Enter (c/i/l/s/f/d)  
i  
i
```



```
Enter the integer: 3
3

-----
                Tuples Program
-----

Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples
7) Join Tuples
8) Save all Tuples
9) Load all Tuples
0) EXIT

Tuples created up till now: 4

CHOICE: 1
1

Please enter the tuple's string identifier (not more than 80 characters) :
Never
Newer

-----Creating Tuple #5-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
3
```

```
3
-----
                        LEGEND
Enter 'c' for char
Enter 'i' for int
Enter 'l' for long int
Enter 's' for string
Enter 'f' for float
Enter 'd' for double
-----

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:1
1
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:2
2
ELEMENT #3:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:3
3
```

Testing out option 6 with the newly created 'New' and 'Newer':
(And some data validation too)

```
CHOICE: 6
6

Please enter the 1st string ID to compare:
dfg
dfg
Please enter the 1st string ID to compare:
33t34t
33t34t
Please enter the 1st string ID to compare:
New
New
Tuple #4 found!!
Please enter the 2nd string ID to compare:
Strfgvd
5trfgvd
Please enter the 2nd string ID to compare:
Newer
Newer
Tuple #5 found!!
The tuples New
and Newer
are of a compatible type to be compared
The value returned is 0

Tuples Program
-----
```

Test Successful once again!

Testing out option 7 with 'New' and 'Newer' :

```
Tuples created up till now: 5

CHOICE:7
7

Enter the String ID of the new tuple to be created:
Newly
Newly
Enter the 1st tuple's pointer:
New
New
Enter the 1st tuple's pointer:
00A010D8
00A010D8
Tuple #4 found!!
Enter the 2nd tuple's pointer:
00A011D8
00A011D8
Tuple #5 found!!

Tuples Program
-----
```

```
Tuples created up till now: 6

CHOICE:2
2

Enter the string identifier of the tuple:
Newly
Newly
Tuple #6 found!!
This tuple has a pointer 00A012D8

Tuples Program
-----
```

Tuple #6 Newly Successfully Created!

Testing out option 0:

```
CHOICE:0
0

Bye!
      --End of program--

Process finished with exit code 0
```

Program Exited Successfully!

Creating now Dog(3,d,7.8) and Cat(2,c,);

```
Please enter the tuple's string identifier (not more than 80 characters) :
Dog
Dog

-----Creating Tuple #1-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
3
3

-----
                LEGEND
    Enter 'c' for char
    Enter 'i' for int
    Enter 'l' for long int
    Enter 's' for string
    Enter 'f' for float
    Enter 'd' for double
-----

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:3
3
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)
c
```

```

C
Enter the character:d
d
ELEMENT #3:
What data type do you want to store?
Enter (c/i/l/s/f/d)
f
f
Enter the float value:7.8
7.8

Please enter the tuple's string identifier (not more than 80 characters) :
Cat
Cat

-----Creating Tuple #2-----

How many elements would you like to store in this tuple?
(Not less than 1 and not more than 9)
2
2
-----
LEGEND
Enter 'c' for char
Enter 'i' for int
Enter 'l' for long int
Enter 's' for string
Enter 'f' for float
Enter 'd' for double
Enter 'f' for float
Enter 'd' for double
-----

ELEMENT #1:
What data type do you want to store?
Enter (c/i/l/s/f/d)
i
i
Enter the integer:2
2
ELEMENT #2:
What data type do you want to store?
Enter (c/i/l/s/f/d)
c
c
Enter the character:c
c

Tuples Program
-----
Please choose one of the following options below:
1) Create a Tuple
2) Get a Tuple's Pointer
3) Get a Tuple's ID
4) Display Tuple
5) Delete Tuple
6) Compare Tuples

```

Testing out 2 incompatible tuples in option 6:

```
CHOICE:6
6

Please enter the 1st string ID to compare:
Cat
Cat
Tuple #2 found!!
Please enter the 2nd string ID to compare:
Dog
Dog
Tuple #1 found!!
Tuples are of different types and therefore cannot be compared!

The value returned is 9999999
```

Testing out option 8:

```
Tuples created up till now: 2

CHOICE:8
8

Tuples Saved Successfully!!
```

Testing out option 9:

```
Element 2 : d

Element 3 : 7.800000

Tuple 2
String ID: Cat

Colon Delimited String : 2:i:c

Element 1 : 2

Element 2 : c

-----
```

```
SaveTuples2B - Notepad
File Edit Format View Help
The tuples which have been saved are shown below
-----
Tuple 1
String ID: Dog

Colon Delimited String : 3:i:c:f

Element 1 : 3|
Element 2 : d
Element 3 : 7.800000

Tuple 2
String ID: Cat

Colon Delimited String : 2:i:c

Element 1 : 2
Element 2 : c

-----
```


All tests performed were successful. Tuples working as expected

TASK 2C

Tasks 2a and 2b have been already been implemented in terms of an Abstract Data Type Interface. Thus simply adding

```
add_library(library SHARED "testdriver2b.c")  
add_executable(2b , "testdriver2b.c")  
target_link_libraries(2b library)
```

The following line in CMakeLists would link the test driver with the shared library.