OBJECT ORIENTED PROGRAMMING

LABORATORY 2

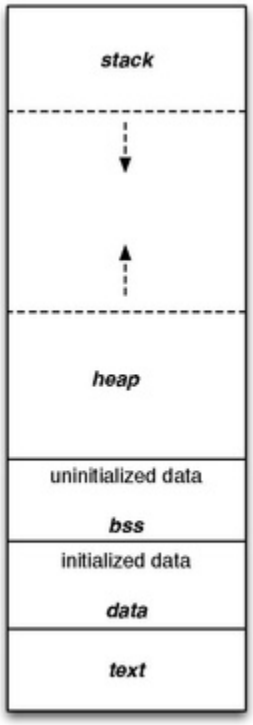
OBJECTIVES

The main objective of this laboratory is to teach students about pointers, parameter passing, memory management and strings in C.

THEORETICAL NOTIONS

Memory layout of a C program

In C/C++, a program's memory layout can be divided into four main section:



- The ***code/text section***: contains the compiled machine instructions of the program. This section is read-only, meaning that the instructions stored here cannot be modified at runtime;
- The ***data section*** contains initialized global and static variables, as well as constant values. This section is typically located after the code section; this area is separated into initialized and uninitialized data.
- The ***heap*** is a dynamic memory area that is allocated and managed by the program at runtime. The heap is used to store dynamically allocated data (with malloc, calloc, realloc, or new in C++). Unlike the stack, the heap is allocated explicitly by programmers and it won't be deallocated until it is explicitly freed.
- The **stack** is a special area of memory that is used to store function calls and local variables. The stack grows and shrinks as functions are called and the control returns from these functions, and each function call creates a new stack frame, which contains information such as local variables, function arguments, and the return address. The stack is limited in size and it is automatically managed by the compiler.

Pointers

Pointers are a powerful feature in C/C++ that allow you to directly manipulate memory addresses. A pointer is simply a variable that stores a memory address of another variable.

The most important aspects when working with pointers are:

- *How to declare a pointer?* To declare a pointer variable, just need to put an asterisk after the datatype.

- *How to assign a value to a pointer?* To assign a value to a pointer, you need to use the & (address of operator) operator to get the memory address of another variable.
- *How to access the value stored at a memory address?* You need to the use * operator (dereferencing operator) to get the value stored at the memory address it points to.

Declaring a pointer	Getting the address of a variable	Dereferencing a pointer

`type* variable;`
`int* pi;`
`float* pf;`
`chat* pc;`
	`&var_name;`
	`int x; int* px = &x;`
	`float f; float* pf = &f;`
	`chat c; char* pc = &c;`
		`*pointer_var_name`
		`*px -> value stored at address px`
		`*pf -> value stored at address pf`
		`*pc -> value stored at address pc`

Parameters passing in C

In C programming language, parameter passing refers to the way in which arguments are passed to a function during a function call. There are two ways in which parameters can be passed in C:

Pass by Value: (This is the default way of passing the arguments to a function) a copy of the value of the argument is passed to the function. The function works on the copy of the value, and any changes made to the value inside the function do not affect the original argument in the calling function.

For example, in the code:

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    printf("x = %d, y = %d", x, y);
    return 0;
}
```

The output will still be x=10, y = 20 .

In the above example, the swap function takes two integer arguments a and b and swaps their values using a temporary variable. However, since the arguments are passed by value, the original values of x and y remain unchanged

Pass by Address/Pointer: In this method, a pointer to the memory location of the argument is passed to the function. The function works on the original argument using the pointer, and any changes made to the argument inside the function affect the original argument in the calling function.

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    printf("x = %d, y = %d", x, y);
    return 0;
}
```

In the above example, the swap function takes two integer pointers a and b and swaps their values using a temporary variable. Since the arguments are passed by address using the & operator, the original values of x and y are swapped. The changes from the previous example are:

- `void swap(int *a, int *b)` - the function takes as parameters pointer int: int*
- `*a, *b` - we are using the dereferencing operator inside the function to access the value stored at the memory address
- `swap(&x, &y)` - when calling the function we need to use the & (address of) operator to pass the address of x and y to the function

Pointers and arrays

In C, pointers and arrays are closely related concepts, and an array name can be used interchangeably with a pointer to the first element of the array. Also, we can use pointer arithmetics to access the other elements of the array.

An array is a collection of elements of the same data type, stored in contiguous memory locations. The elements of an array can be accessed using the indexing operation ([]). For example, in the following array declaration, we have created an array of integers with 5 elements:

```
int arr[5] = {1, 2, 3, 4, 5};
// use the indexing operation to get the second elements in the array
int second = arr[1];
```

A pointer is a variable that stores the memory address of another variable. **In C, an array name is a pointer to the first element of the array**. For example, in the following code, we create a pointer variable *p* that points to the first element of the array *arr*:

```
int *p = arr;
```

The name of the array *arr* can be used interchangeably with a pointer to the first element of the array. So, the above statement is equivalent to :

```
int *p = &arr[0]; // & - gets the address of arr[0]
```

Both statements assign the memory address of the first element of the array to the pointer variable *p*. Once we have a pointer to the first element of an array, we can use pointer arithmetic to access the other elements of the array. For example, the second element of the array can be accessed using:

```
int second = *(p + 1); // this is equivalent to arr[1]
       // * is the dereferencing operator, so we get the value stored at
p+1
       // p + 1 is the memory address of the second element in the array
```

Dynamic memory allocation in C

In the earlier examples, all the memory requirements (size) were known before the program execution. For example, the compiler knows how much memory is needed for an *int* variable, and when we used arrays we specified their size: *char name[32]*. But there are some cases in which we don't know the memory needs of a program at compile time and they can only be determined at runtime. For example, when we need to allocate an array depending on the user input.

Dynamic memory allocation is a way to achieve this. It allows memory to be allocated and deallocated as needed during the execution of a program. This strategy allocates memory on the heap and relies on pointers.

In C, dynamic memory allocation is accomplished using three functions: malloc(), calloc(), and realloc().

malloc: *malloc* function is used to allocate a block of memory of a specified size in **bytes**. It takes one argument, which is the number of **bytes** to allocate. If the allocation is successful, malloc() returns a pointer to the beginning of the block of memory, otherwise it returns NULL.

For example, if we want to dynamically allocate an array to store 100 integer elements, we would use:

```
int* arr = (int*) malloc(100*sizeof(int));
if(!arr){
  printf("Error! Failed to allocate memory!");
  exit(-1);
}
```

Notice that to allocate memory we need to specify the size in bytes. So, we use `sizeof(int)` which gives us the number of bytes required to store a single integer, and we multiply it with 100 (because we want to create an array of 100 integers). malloc returns a *void** , so you always need to cast it. In our example, we cast it to a pointer to int (*int**), because we allocated memory to store an integer array.

calloc is similar to malloc(), but it initializes all the bits in the allocated memory to zero. It takes two arguments: the number of elements to allocate, and the size of each element in bytes. To allocate an array of 100 integers using *calloc,* we would use:

```
int* arr = (int*) calloc (100, sizeof(int));
if(!arr){
  printf("Error! Failed to allocate memory!");
  exit(-1);
}
```

realloc() function is used to resize an existing block of memory. It takes two arguments: a pointer to the existing block of memory, and the new size of the block in bytes. The function may move the memory block to a new location (whose address is returned by the function). The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location.

Below you have an example that uses *realloc* . The problem is the following: you need to read numbers typed by the user until the user types the value -1. All the numbers that you read must be stored in an array. You don't know in advance how many numbers the user will type, so you start with an array of 10 elements, and if needed, you double the capacity of the array. The capacity of the array is the maximum elements that you can store in the array, while the size/length of the array is how many elements you actually have in the array.

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

int main()
{

    int nr = 0;
    unsigned int cap = 10; // capacity of the array (the maximum number of
elements that we can store in the array)
    unsigned int sz = 0; // the number of elements that we have in the
array
    int *arr = (int*) malloc(sizeof(int)*cap);

    if(!arr){
        printf("Failed to allocate memory! App will now stop\n");
        exit(-1);
    }

    printf("Please type some numbers. When you want to stop, type -1\n");
```

```
    while(nr != -1){
        scanf("%d", &nr);
        // reallocate the array is needed
        if(sz >= cap){
            cap *= 2;
            arr = (int*)realloc(arr, cap*sizeof(int));
            if(!arr){
                printf("Failed to allocate memory! App will now stop\n");
                exit(-1);
            }
        }
        if(nr != -1)
            arr[sz++] = nr;
    }

    printf("The numbers are: ");
    for(unsigned int i = 0; i < sz; i++)
        printf("%d ", arr[i] );
    printf("\n");

    return 0;
}
```

As when using dynamic memory allocation, the memory is allocated on the heap, it is your responsibility to deallocate it. So, the memory allocated using malloc(), calloc(), or realloc() must be freed when it is no longer needed, using the **free()** function. Otherwise, this will lead to memory leaks.

```
int* arr = (int*) malloc(100*sizeof(int));
if(!arr){
  printf("Error! Failed to allocate memory!");
  exit(-1);
}
// ... use arr
// when you are done with arr, you must free the memory
free(arr);
```

Using command line arguments

Command-line arguments are values that are passed to a C program when it is run in a command-line interface. These values are passed as strings and can be used by the program to modify its behavior according to the user commands. The information related to command-line arguments is stored in the *argc* and *argv* parameters of the main function:

- the *argc* variable is an integer that represents the number of command-line arguments(arguments count)
- the *argv* variable is an array of strings that contains the actual arguments. (char *argv[] - the datatype of argv is an array of character pointers (char*) where each pointer points to the first character of the string).

Always, the first command line argument of a program is the name of the executable.

Below you have an example on how you can work with command line arguments in C:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    printf("The executable name is %s\n", argv[0]);

    for (i = 1; i < argc; i++) {
        printf("Argument %d has the value %s\n", i, argv[i]);
    }

    return 0;
}
```

Setting command line arguments in Visual Studio:

https://cs-people.bu.edu/deht/CS585/VSTutorial/#CommandLineArgs

Menu based console applications A menu-based command-line application is a program that allows users to interact with it through a series of menu options presented in the command line (terminal). The user is presented with a list of options, and they can choose one by entering a corresponding number or letter. These types of applications are often used to provide a user-friendly interface for performing complex tasks that require multiple steps or options.

Below you have a simple menu based console application, that allows the user to type two real numbers and then perform some operations on numbers based on the option she/he chooses.

```
#define _CRT_SECURE_NO_WARNINGS
#include <math.h>
#include <stdio.h>

void displayMenu(){
    printf("\nPlease select an option:\n");
    printf("a/A - Addition\n");
    printf("s/S - Subtraction\n");
    printf("m/M - Multiplication\n");
    printf("d/D - Division\n");
    printf("e/E - Exit\n\n");
}

int main() {
    float num1, num2, result;
    char option;
```

```
    printf("Simple command line menu based app that allows you to perform
basic arithmetic operations between two numbers");
    do{
        printf("\n\n--------------------------------\n");
        printf("Enter 1st number: ");
        scanf("%f", &num1);

        printf("Enter 2nd number: ");
        scanf("%f", &num2);

        displayMenu();

        scanf(" %c", &option);
        switch(option) {
        case 'a':
        case 'A':
            result = num1 + num2;
            printf("\n%.2f + %.2f = %.2f\n", num1, num2, result);
            break;

        case 's':
        case 'S':
            result = num1 - num2;
            printf("\n%.2f - %.2f = %.2f\n", num1, num2, result);
            break;

        case 'm':
        case 'M':
            result = num1 * num2;
            printf("\n%.2f * %.2f = %.2f\n", num1, num2, result);
            break;

        case 'd':
        case 'D':
            if(fabs(num2 - 0.0) < 0.00001) {
                printf("\nError: division by zero\n");
            } else {
                result = num1 / num2;
                printf("\n%.2f / %.2f = %.2f\n", num1, num2, result);
            }
            break;

        case 'e':
        case 'E':
            printf("\nExiting program...\n");
            break;
        default:
            printf("\nInvalid option. Please try again.\n");
            break;
        }
    }while(option!='e' && option != 'E');

    return 0;
```

```
}
```

PROPOSED PROBLEMS

1. Write a function that takes as an input an array of integer numbers (both positive and negative numbers) and returns the value of the triplet with the maximum product, as well as the elements that form this triplet (in increasing order). Use pass by pointer/address to return the elements that form that triplet. Think about the appropriate data type for the result. If the size of the array is less than 3, you should return the minimum representable value of the data type and the elements that form the triplet should be set to 0.

If the size of the array is less than 3 display the message: *The array has less than 3 elements. Application will now stop.*

To read the input data, you will first read the size of the array *n,* and then *n* integers.

For example, if the user types: 5, -3, 10, 200, 4, -900, then the size of the array is $n = 5$ and the array elements are {-3, 10, 200, 4, -900}.

You should display *"The maximum triplet is (-900, -3, 200) with a product of 540000".* (the elements of the triplet should be displayed in increasing order).

2. In C, the *strchr* function is used to locate a character in a string. More specifically, it returns a pointer to the first occurrence of the character in the string and NULL otherwise.

```
const char * strchr ( const char * str, char character );
```

The parameters of the function are:

- *str* - a C string in which we will search a character;
- *character* - the character to be searched.

Write a function *my_strchr*, with the same parameters and return value as *strchr*.

```
const char * my_strchr ( const char * str, char character );
```

You are not allowed to pass the length of the string as a parameter, nor to use the *strlen* function to determine the length of the string. Also, don't use the indexing operator [].

3. Using the function that you wrote for problem *2* , write a function that computes and returns an array with all the positions of the occurrences of a character in a string.

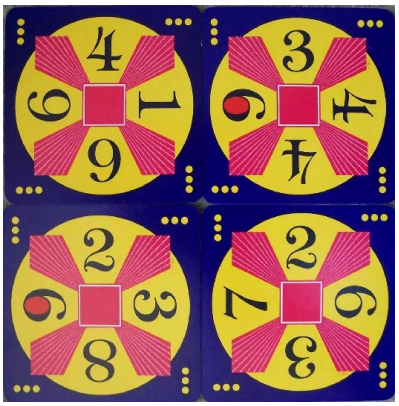
```
void find_all( const char * str, char character, int*
positions, unsigned int cap, unsigned int * l);
```

The parameters of the function are:

- *str* - a C string in which we will search the character;
- *character* - the character to be searched;
- *positions* - [in/out] param. Array containing all the positions on which the character *character* occurs in the string *str*. First fill this array with -1!
- *cap* - the capacity of the *positions* array (the maximum number of elements that you can add in the array). If there are more occurrences of *character* in *str* you can and should only store the positions for the first *cap* ones.
- *l* - [in/out] parameter. The length of the *positions* array (the number of elements currently in the array - i.e. the number of times *character* occurs in *str*).

4. Check your programs for memory leaks using the C runtime library: https://docs.microsoft.com/en-us/visualstudio/debugger/finding-memory-leaks-using-the-crt-library?view=vs-2019.

5. The ***24 puzzle*** is math game in which the player is given 4 digits and he/she must figure out a way of combing these digits and the addition (+), subtraction (-), multiplication (*), and division (/) operators such that the result is 24.



The result is computed in sequential order on the digits. For example, if the digits are 3, 4, 4, 9 and the user enters the three operations as +, -, * then the calculation is done as follows:

3 + 4 = 7.
7 - 4 = 3.
3 * 9 = 21.

Of course, this is not a valid solution for the game.

Write a menu-based application with command line arguments to play this game. The game can be played in easy mode or full mode. For the easy mode, the digits displayed to the user are randomly selected from the following combinations:

- 1 1 4 6
- 1 1 3 8
- 1 2 2 6
- 1 2 3 4
- 1 1 3 9
- 4 4 4 6
- 1 8 8 8

For the full mode, you need to generate all the 3188 combinations for the game that can be used to generate a valid solution, and randomly select one of them. Use *double* when performing the computations. Don't try to be a hero for this, a brute-force solution with many imbricated for loops will do. Consider the numbers in increasing order and the operators in the order +-*/.

Write a function *generateAllCombinations* that takes as parameters two dynamically allocated arrays (one for the operators and one for the operands, and fills in all the 3188 solutions).

*void generateAllCombinations(int** numbers, char** operators);*

The *numbers* parameter is a 2D matrix, with 3188 rows and 4 columns, that will store the numbers (operators), while the *operators* parameter is a 2D matrix, with 3188 rows and 3 columns, that will store the operators used to combine the numbers.

The memory will be dynamically allocated in the main function. Make sure to free the memory at the end of the program!

First generate the 4 digits, then generate all the operators. Use double when computing the results.

The program will have the following command line arguments:

- -h: display information about how the game is played and exit.
- -s: save all the possible solutions for the game in a text file called "solution.txt" and then exit.
- -e: play the game in easy mode.

If no command line arguments are specified, the game is played in the "full" mode

The algorithm that you need to implement is the following:

- A set of digits is randomly chosen and the four digits are displayed to the user.
- After reading in the operators that the user chooses, the result is computed.

- If the user enters an invalid character (something different than +, -, *, or /), the following error message is displayed: "Error: you typed an invalid operator. Please retry!", and a new set of digits is randomly selected and displayed to the user.
- If the user enters a number of characters different than 3, the following error message is displayed: "Error: you need to enter exactly three operators. Please retry!", and a new set of digits is randomly selected and displayed to the user.

- The program will display a step-by-step illustration of how the result is computed. If the result is equal to 24, at the end the program will display: "Well done!", otherwise "You lost! the result is not 24."
- The user will be asked whether he/she wants to continue the game. If the user types *y* or *Y* the game continues (a new set of digits is selected and displayed to the user), otherwise (if any other character is entered) the game stops.
- At the end of the game (the user chooses to exit), the message "Thanks for playing! See you soon!" is displayed.

Example run:

```
Welcome to the game of TwentyFour.
Use each of the four numbers shown below exactly once, combining them somehow with the basic mathematical operators (+,-
,*, /) to yield the value of twenty-four.

The numbers to use are: 8, 5, 8, 1.
Enter the three operators to be used, in order (+,-,* or \):-**
8 - 5 = 3.
3 * 8 = 24.
24 * 1 = 24.
Well done!

Would you like to play again? Enter N for no and any other character for yes.
y
The numbers to use are: 6, 2, 8, 1.
Enter the three operators to be used, in order (+,-,* or \):+++
6 + 2 = 8.
8 + 8 = 16.
16 + 1 = 17.
You lost! the result is not 24.

Would you like to play again? Enter N for no and any other character for yes.
N
Thanks for playing! See you soon!
Press <RETURN> to close this window...
```

EXTRA CREDIT I

Use frequency analysis to break Caesar's cipher. This is a brute force method that tests all the possible displacements and returns the most likely decryption of a text.
Frequency analysis relies on the fact that some letters (or combination of letters) occur more in a language, regardless of the text size. For example, in English the letters E, A are the most frequent, while the Z and Q are the least frequent; miscellaneous fact: see *Etaion shrdlu*[1]). The distribution of all the characters in English is depicted in the figure below:

E	T	A	O	I	N	S	H	R	D	L	U	C
12.7	9.1	8.2	7.5	7.0	6.7	6.3	6.1	6.0	4.3	4.0	2.8	2.8
M	**W**	**F**	**Y**	**G**	**P**	**B**	**V**	**K**	**X**	**J**	**Q**	**Z**
2.4	2.4	2.2	2.0	2.0	1.9	1.5	1.0	0.8	0.2	0.2	0.1	0.1

Figure 1. *Distribution letters in English (image source: https://ibmathsresources.com/tag/vigenere-cipher/)*

The idea of this method is to compare the frequency of the letters with the Chi-Squared distance:

$$\chi^2(C, E) = \sum_{i='a'}^{i='z'} \frac{(C_i - E_i)^2}{E_i}$$

, where C_i represents the occurrence of the i[th] character, and E_i is the expected count of the i[th] character of the alphabet.

The formula seems complicated at a first glance, but it is really not that complicated. Basically, for each possible character (*i* goes from 'a' to 'z'), we measure the discrepancy between how often it appeared in the encrypted text (C_i) and how often it is expected to appear in English texts (E_i); the difference $C_i - E_i$ is squared such that we remove negative signs. The division by E_i is simply a normalization factor.

The lower the Chi-square distance $\chi^2(C, E)$, the more similar the histograms C and E are.
As this algorithm is a brute force method to break the cipher, you should compute the histogram for all possible shifts, and compute the Chi Squared distance between these histograms and the average distribution of the characters in English. The shift with the lowest Chi Squared distance is the solution.

You can find more information about this algorithm here:
https://ibmathsresources.com/2014/06/15/using-chi-squared-to-crack-codes/ .

To sum up, to solve this problem you need to:

- Write a function that reads the distribution of the letters from a file (*distribution.txt*) and stores it into an array. The frequency of the letter 'a' is stored on the first line of the file (as

[1] https://en.wikipedia.org/wiki/Etaoin_shrdlu

a floating point number), the frequency of the letter 'b' is stored on the second line of the file etc.

- Write a function that computes the normalized frequency of each character (a histogram) in a text.
- Write a function that computes the Chi-square distance between two histograms.
- Write a function that breaks the Caesar's cipher using frequency analysis: iteratively shifts the encrypted code through all the possible permutations, computes the Chi-square distance between each permutation and the approximate distribution of letters in English, and returns the permutation with the least Chi squared distance as the solution.
- Finally, create a user-friendly menu based application for Caesar's cipher related tasks.

Break the message:

> *Uf ime ftq nqef ar fuyqe, uf ime ftq iadef ar fuyqe, uf ime ftq msq ar iuepay, uf ime ftq msq ar raaxuetzgee, uf ime ftq qbaot ar nqxuqr, uf ime ftq qbaot ar uzodqpgxufk, uf ime ftq eqmeaz ar xustf, uf ime ftq eqmeaz ar pmdwzqee, uf ime ftq ebduzs ar tabq, uf ime ftq iuzfqd ar pqebmud.*

The extra credit should be uploaded on github, and you should also have a Readme.md file. https://www.makeareadme.com/