uc3m | Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

**SYSTEMS ARCHITECTURE**

**LAST NAME:** ..........................................................................................................................
**NAME:** ..........................................................................................................................

**LABORATORY SESSION 3**

**PART 1: DYNAMIC MEMORY**
**PROBLEM 1**
The Cylons have a plan.

But in order to carry it out, they need to keep **information about all the humans** that are on the 12 colonies (planets where these humans live). Since each human can move freely between planets, they cannot organize it by planets, so they have decided to have a **single array to store all the information**.

The **array is of fixed length** as the Cylons are still learning C.

When running your program, it should **not have any memory leaks.**

**Section 1 (2 points)** Define a **data type** to store all the information about a human:
- Name of the human (a string of undefined length).
- Hazard: harmless, common or dangerous.
- Probability of recruitment (number between 0 and 1).

**Section 2 (4 points)** Implement a **function** that, given the set of humans to be considered, the size of that set, a probability of recruitment and a given hazard, returns how many humans of that particular hazard have a **higher or equal probability of recruitment** than the passed as parameter.

**Section 3 (4 points)** Implement a **main program**, which creates a fixed-size array of size 4 humans, initializes it (with the values you want), calls the function in the previous section and prints out the value returned by that function. The name of the human should be stored in dynamic memory.

**PROBLEM 2**
We have been entrusted with the task of implementing a completely new question and answer PC game based on the well-known travel guide "The Hitchhiker's Guide to the Galaxy". Once our board game has been developed, it will be released in a pack that will include a Babel fish and a towel.

The set of questions will be stored in a **fixed-length array**.

When running our program, **it should not have any memory leaks**

**Section 1 (2 points)** Define a **data type** to store all the information about a question:
- The question itself (a string of undefined length).
- Its answer (a chain of length 30)
- Type of question: corporations, civilizations, art, history, literature or general.
- Points (may have decimals) obtained by getting the question right.

**Section 2 (4 points)** Implement a **function** that, given a set of questions, the number of questions and a question type, returns the maximum score that can be obtained from getting all questions of that type right. Additionally, if the question type is ANY, it will return the maximum score that can be obtained from getting all questions right.

**Section 3 (4 points)** Implement a **main program**, which creates a fixed-length array of size 4 questions, initializes it (with the values you want), calls the function of the previous section and prints the value returned by that function. The sentence of the question itself should be stored in dynamic memory.

**PART 2: VARIADIC FUNCTIONS**
**PROBLEM 3**
Implement a variadic function called `count` that counts and returns the number of odd or even numbers within a set of `N` ints.
It will receive an `int` with value `0` if you want to count even numbers and and `int` with value 1, if you want to count odd numbers. It will also receive the number of ints within the set.

Examples:
- `count(0, 2, 10, 14)` returns 2

- `count(1, 2, 10,14)` returns 0
- `count(0, 5, 10, 14,2,3,1)` returns 3
- `count(1, 5, 10, 14,2,3,1)` returns 2

**PROBLEM 4**
Implement in C a variadic function called `print_and_sum`. The input parameters of the function will be a format string, a number `num` and `num` doubles.

The function should calculate the sum of the given doubles and it should print it as it is shown in the following example.

If in a main program we perform the following calls to the function:

```
print_and_sum("%3.1f ", 1, 1.5);
print_and_sum("%3.1f + %3.1f ", 2, 1.5, 2.5);
print_and_sum("%3.1f + %3.1f + %3.1f ", 3, 1.5, 2.5, 3.5);
```

It should print:

```
./apartado1
1.5 = 1.500
1.5 + 2.5 = 4.000
1.5 + 2.5 + 3.5 = 7.500
```

**PART 3: DYNAMIC MEMORY (higher difficulty than Part 1)**
**PROBLEM 5**
As everybody knows, we are at the middle of a horrible war against the burgers. In preparation for the anticipated third invasion, children, as Enger Wiggin are trained at the Battle School, a military space station situated in Earth's orbit. The military, wise of Ender's potential, lead him to exhaustion, assigning him not only military tasks, but also programming tasks. However, his friend Bean (that is in the shadows, looking after Ender) is the one that implements all the programming tasks, and this time he asked us for help.

Mazer Rackham (a military instructor of Ender and Bean) is really obsessed with the scores obtained by the different child at the different simulators. In fact, he is scoring all the information about the score of each game played by each child withing a dynamic array named `partidas`. This is a dynamic array of variables of type `struct partida`:

```
struct partida{
    float puntos;
    char *nombre;
};
```

Right now, he has stored within `partidas` (of length `numpartidas`) (both local variables of the `main`) the scores of all the played games of his different students. Now, he wants Ender to implement a function that generates another dynamic array with the summary for each child, that is, her/his name, the number of played games and her/his total obtained score. This array sould not share memory with other students.

```
struct puntuacion{
    float total;
    char *nombre;
    int num_partidas;
};
```

**Section 1.1**
Implement a function `agrego_puntuacion` that, given a dynamic array of variables of type `struct partida` or a given length (that it is also passed as input parameter), creates another dynamic array of variables of type `struct puntuacion` of length the number of children that have played. Each position of the new array will correspond to a child and it should store the name of the child, the number of games s/he has played and her/his total score (as sum of the obtained points).

The function should work for any number of children. Thus, the function should also return in some way the total number of children. The function should inform the user of an error if there is any (using an input or output parameter, you are not allowed to print any information within the function).

- You should define the header of the function (including the output parameter and as many input parameters as you may need).
- **You cannot use global variables.**
- The new array of variables of type `struct puntuacion` should be allocated in dynamic memory.
- You **should not modify** the array of the variables of type `struct partida`.
- The function should work for any length of array, including the empty array (NULL).
- You can assume that the filed `nombre` within each struct of type `struct partida` is never NULL.
- You can define (and implement) all the functions that you need.
- The function **should not print any information**.

**Section 1.2**

Implement a main where you should define the variables `partidas` and `numpartidas`, and where you should invoke the function `agrego_puntuacion`. After the invocation, you should print the number of children, and for each child her/his name, the number of played games and her/his total score.

**PROBLEM 6**

Implement using dynamic arrays a data structure that behaves as a stack of `float`.

In order to achieve this, you should:

- Define the data structure `struct stack`.
- Implement the `init` function that will receive as parameters at least:
    - a variable of type `struct stack`;
    - the initial capacity of the stack, a positive integer;
    - factor, a positive integer, that will be the factor by which the stack will augment if it is filled.

  This function should initialize the stack to the given initial capacity.
- Implement the function `delete_stack` that will free the dynamic memory used by the stack.
- Implement the functions `push` and `pop`.
    - You can define the input and output parameters as you consider.
    - Within the function `push`, if the stack is full, you should augment it in order to be able to store the new element (by the factor given when the stack was initialized).
    - The *pop* function should return a `float`. If there are no data in the stack, you should inform the user that an error happened.
- Implement the function `size` that returns the number of elements stored in the stack. The function should return a -1 if there are no elements in the stack.
- You should use the need mechanisms to inform the user if an error happened during the execution of the function or if the input parameters are invalid.
- **None of these functions is allowed to print at the terminal**.