TP C#1: The Case of the Hanged Man

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
tp1-sherlock.holmes/
|-- README
|-- .gitignore
|-- Case1-HangedMan/
    |-- Case1-HangedMan.sln
    |-- Case1-HangedMan/
        |-- Case1-HangedMan.csproj
        |-- Program.cs
        |-- Basics/
            |-- Loops.cs
        |-- Game/
            |-- Game.cs
            |-- Loader.cs
            |-- acdc_logins.txt
            |-- word_bank.txt
        |-- Bonuses/
            |-- Bonuses.cs
```

Do not forget to check the following requirements before submitting your work:

- You obviously replace `sherlock.holmes` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- You must respect the prototypes of the given and asked functions.
- Remove all personal tests from your code.
- The `.gitignore` file allows you to automatically ignore binary files and other trash files that may be present in your repository. Remember to configure it properly to avoid accidentally pushing those files!
- **The code MUST compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 This week's case

The terrible Professor Moriarty has struck again... He kidnapped a civilian as bait to get the great Sherlock Holmes into another one of his sick games. Tired of playing those games, the detective asked to you help him with this case. Because it is Moriarty leading the game, we should be prepared for anything, even the worst. Hence, why don't we go over loops and standard streams in C# just in case?

1.2 Objectives

By the end of this practical, you will be able to work with `while` loops as well as standard input and output via the `Console` class. This practical also contains many common and useful functions that you will often find in your future practical tasks.

2 Course

2.1 The console

The console is the easiest and the most standard way for your program to interact with the **shell**. Simply put, the **shell** is the program that allows a user to interact with the computer via text commands. When you open a terminal, a **shell** is automatically launched and waits for input. On EPITA's PIE, the **shell** used is **bash**. **bash** shows you your username, your computer, and the current directory. This is displayed in the *prompt* (see below).

```
[sherlock@bakerStreet ~/tpcs1]$
```

In this example, the user is called `sherlock`, they called their computer `bakerStreet`, and they are currently in the directory `~/tpcs1`.

2.1.1 Using the command line

After displaying the prompt, the shell waits for you to input a command. A command can have multiple forms. It is usually the name of a program followed by arguments, as shown in the following example:

```
[sherlock@bakerStreet ~]$ ls
foo  acdc  sups  tpcs1
[sherlock@bakerStreet ~]$ ls foo
bar  baz
[sherlock@bakerStreet ~]$
```

Here, the program `ls` is first executed without arguments (line 1). `ls` displays the content of a directory. If no argument is given, it displays the content of the current directory (in our case: `foo`, `acdc`, `sups` and `tpcs1`). On the third line, an argument is given to `ls`, `foo`. The content of the directory `foo` is thus displayed. As you can see, if you want to give arguments to a program, you just have to write them after the name of the program. On the fifth line, you will notice that the prompt is waiting for another command after the previous one ends.

Of course, you can give more than one argument to a program. Among these arguments, some can modify the behavior of the program. We call them "options" and they usually start with a `'-'`. Most programs provide a list of existing options when calling them with `--help`. Otherwise, you can read the program's manuals page by calling `man` followed by the name of the program (e.g. `man ls`).

2.1.2 Standard streams

Now that you know how to use the command line, let's get to something a bit more complex. Terminals use what we call *streams*. A stream is an object used by your system to read or write data such as text. For example, on UNIX systems, each file can be used as a stream. There are 3 majors streams that you should know:

- **Standard Input**: This stream allows you to send data to your programs or your shell. Everything typed in a terminal will be sent to this flow. It is also known as `stdin`.
- **Standard Output**: This stream is the opposite of `stdin`, it is the stream where your programs write data. What is displayed in your terminal is what has been sent to this stream. As you can guess, it is also known as `stdout`.

- **Standard Error Output**: This stream is similar to `stdout` and works in the same way, except that it is designed to receive error messages. It will also write in your terminal, just like `stdout` thus informing you that an error occurred. It is also known as `stderr`.

It is important for you to understand how to interact with your programs via these streams. If you have any doubts, just ask your beloved ACDCs.

2.2 The Console class

You will learn how to read and write on standard streams. In order to do this, the C# library provides the Console[1] class. This class offers a wide range of methods allowing you to write to the console, read from it, get its state, change its properties. . . These methods can be called by writing `Console.MethodName(args)` in your code. You can get more information on this class by reading its MSDN page. For now, here are some functions that you will have to use in the upcoming exercises.

2.2.1 Writing

Two basic methods allow you to write in the console:

- `Console.Write(arg)`: This method converts the parameter `arg` to a string and displays it on `stdout`. There can be more than one argument, and they can have various types. You can check the MSDN page of this method to learn more.

- `Console.WriteLine(arg)`: This method acts the same way as `Console.Write` except it adds a line break at the end of the string. On Linux, this is equivalent to:
  => `Console.WriteLine(arg) == Console.Write(arg + "\n")`
  Be careful, other systems may use characters other than `\n`[2], so make sure to always use `Console.WriteLine` if you want a string followed by a new line. You can also use `Console.WriteLine()` to just skip a line.

If you wish to write to the standard error output, you need to use `Console.Error.Write` and `Console.Error.WriteLine` instead.

[1] `https://msdn.microsoft.com/en-us/library/system.console`

[2] The characters that correspond to a line break actually depend on your platform. On UNIX and UNIX-like systems (e.g. Linux, macOS, BSD, . . .), line breaks are indicated via the `\n` character. On other operating systems such as Windows, the `\r \n` characters are used instead. You can programmatically determine the characters that correspond to a line break on your current platform in C# using the following property: `Environment.NewLine`

Here are some examples that use these methods. Don't hesitate to try them out (the output is written as a comment next to the corresponding line):

```csharp
Console.WriteLine("Sherlock");
// Sherlock

Console.Write("Hol");
Console.WriteLine("mes");
// Holmes

Console.WriteLine("a" + "b" + "c" + "d");
// abcd
Console.WriteLine("42");
// 42
Console.WriteLine(42);
// 42
Console.WriteLine(6 * 7);
// 42

string beloved = "ACDC";
bool best = !false;
Console.WriteLine("My {0}s are the {1} best ones."
                  , beloved, best);
// My ACDCs are the True best ones.
Console.Error.WriteLine("An error occurred.");
// on stderr: An error occurred.
```

Although they look similar, remember that `Console.Error` sends the result to `stderr` instead of `stdout`.

2.2.2 Reading

Just like for the writing part, we will see two methods allowing you to read from the console. Here, reading means getting data typed in the console by the user.

- `Console.Read()`: This method waits for the user to type a character. Once a character is received, the method returns the character as an `int`. The program then continues its execution.
- `Console.ReadLine()`: This method waits for the user to type a line, ending with a line break (when pressing the Return key). Once it is done, the method returns the line as a `string` without the ending line break characters. The program then continues its execution.

You can read and execute the following program to test it out. Don't hesitate to play around with the arguments:

```
static void Main(string[] args)
{
    Console.Write("First char: ");
    int c = Console.Read();
    Console.WriteLine();
    Console.WriteLine("You typed: '{0}'", (char)c);

    Console.Write("Second char: ");
    c = Console.Read();
    Console.WriteLine();
    Console.WriteLine("You typed: '{0}'", (char)c);

    Console.WriteLine("Type a line:");
    string str = Console.ReadLine();
    Console.WriteLine("You typed: \"{0}\"", str);
    Console.WriteLine("Length == " + str.Length);
}
```

Note that `Console.Error.Read()` does not exist. It would make no sense since `stderr` is an output flow.

2.2.3 Appearence

You can now read and write in the console. That's cool, but it's also a bit boring. Fortunately, the Console class can do much more, such as changing the cursor's position, changing the background color, etc.

Methods:

- `Console.Clear()`: Clears the Console. After calling this method, the text previously displayed is erased and the writing cursor is placed in the top-left corner.
- `Console.SetCursorPosition()`: Places the cursor to the given position. See the MSDN page of this method to understand how to use it.

Attributes:

- `Console.Title`: The title to display in the title section of the console.
- `Console.BackgroundColor`: The background color of the console.
- `Console.ForegroundColor`: The foreground color of the console.

You can get or set the value of these attributes just as you would get or set variables' values. Once again, you can find more methods and attributes on the Console class MSDN[3] page.

[3] `https://msdn.microsoft.com/en-us/library/system.console`

2.3 The Main function

As you may already know, the method `Main` is automatically called when your Program is executed. Let's have a look at its prototype:

```csharp
static int Main(string[] args);
```

The `static` prefix is a C# keyword that you will learn about in another lesson. `int` is the return type of the function. It is mainly used to give information on whether your program succeeded or failed after its execution. By convention, a return value of 0 means the program succeeded, any other value means that the program has failed. `Main` is the name of the method. Finally, its argument is `string[] args`. As the method `Main` is automatically called when your program is executed and not manually executed by your own code, what are its arguments? Well, when calling `Main`, your system puts the arguments given in the command line into the `args` array. In order to easily understand this concept, write this short program and compile it under the name `MyEcho.exe`:

```csharp
static void Main(string[] args)
{
    if (args.Length < 1)
        Console.WriteLine("Not enough arguments.");
    else
        Console.WriteLine(args[0]);
}
```

Let's execute it in the console:

```
1  [sherlock@bakerStreet ~]$ ./MyEcho.exe
2  Not enough arguments.
3  [sherlock@bakerStreet ~]$ ./MyEcho.exe Test.
4  Test.
5  [sherlock@bakerStreet ~]$ ./MyEcho.exe I love my ACDCs
6  I
7  [sherlock@bakerStreet ~]$ ./MyEcho.exe "I love my ACDCs"
8  I love my ACDCs
9  [sherlock@bakerStreet ~]$
```

As you can see, this program checks if it has been given arguments. If not, it will display `"Not enough arguments."` to `stdout`. Otherwise, it will display the first argument (index 0 of `args`).

Take a closer look at the lines 7 and 8 of the example above. Here, we are using double quotes ("), this way, the shell will recognize every word between the quotation marks as the one and only argument. Thus, it will be put as one string in `args`. If you don't use quotation marks, the shell will consider each word separated by spaces as an argument and each word will be put at a different index in `args`.

In a nutshell, without quotation marks, the `args` array would be `["I", "love", "my", "ACDCs"]`, whereas it would be `["I love my ACDCs"]` with the quotation marks.

2.4 Loops

Let's say we want to write a program that displays the current line number. Said program would probably look like this:

```csharp
using System;
Console.WriteLine("This is line 1");
Console.WriteLine("This is line 2");
Console.WriteLine("This is line 3");
Console.WriteLine("This is line 4");
Console.WriteLine("This is line 5");
[...]
```

Regardless of the obvious waste of time it is to write such a useless program, the problem is the time and space this code takes. In some cases, you might have to do a lot more than just printing the number of the line. You could modify some variables, write to a file, etc.

This is why we use loops, which can solve two problems.

- Repeating one or more instructions as long as a condition is satisfied.
  This is the purpose of the **while** loop.

- Repeating one or multiple instructions a certain number of times (to execute it the right amount of times, we use a variable as a counter.)
  This is the purpose of **for** loops which you will study next week.

2.4.1 The while loop

Here is how to use the while loop previously described:

```csharp
while (condition)
{
    // do things
}
```

The computer understands it this way: *as long as the "condition" is true, I have to repeat the code between the brackets.* Consequently, if the condition is false from the start, the code inside the loop will never be read.

The following code would be a fairly classic example of usage of the
textttwhile loop:

```csharp
int counter = 0;
while (counter < 42)
{
    Console.WriteLine("They were the footprints of a gigantic hound!");
    Console.WriteLine("Excellent! I cried. 'Elementary,' said he.");
    counter = counter + 1;
}
```

The condition here is **counter < 42**. As **counter** starts at 0, the program will go into the loop, display the text, then increment the counter. This will repeat 42 times.

Caution

Do not forget to add an instruction that sets the condition of the loop to false. Otherwise, the condition will always be true and the loop will run indefinitely until you force the program to stop.

Here is a little tip about the counter. As we said, we need to "increment" it, which means, add 1 to its value, in order to count how many times we entered the loop. Writing `counter = counter + 1` is a bit hideous. That is why there is some syntactic sugar for this instruction. You can simply write either `variable += 1` or `variable++`. To decrement (substract 1 from its value), it will work the same with `variable -= 1` or `variable−−`.

Going further

The first variant also exists for multiplication, division and the modulo operation[a].

- `a = a * b` is equivalent to `a *= b`
- `a = a / b` is equivalent to `a /= b`
- `a = a % b` is equivalent to `a %= b`

[a] As a reminder, the modulo of `a` and `b` is the remainder of the euclidian divison of `a` by `b`.

Finally, there is also another way of using the `while` loop, which is the **do...while** loop. It works in the same way except that the condition is checked when *exiting* the loop. Thus, the program will ALWAYS enter the `do...while` loop at least once, even if the condition is initially false. Here's the syntax for this loop:

```
var i = 0;
do {
    Console.WriteLine("Roses are red,");
    Console.WriteLine("Violets are blue,");
    Console.WriteLine("I'm coding in C#,");
    Console.WriteLine("And so will you too.");
    i++;
} while (i != 42); // Don't forget the semicolon!
```

3 Loops

Important

- You functions must have the same prototype as stated in the subject (name, type of the return value and type of the arguments).
- **Using recursion is forbidden for this TP**. You must use `while` loops when necessary.
- Only `while` and `do..while` loops are allowed. Meaning loops like `for`, `foreach` etc. are forbidden.

The functions in this section must be located in the file "Loops.cs", in the project "TP1" and at the root of the folder "Basics".
In this section, you will write functions utilizing loops and standard streams (standard input, standard output and standard error output). If you need any help concerning those notions, read the course again: here or this documentation: .NET documentation.

Warning

Do not forget that recursion is prohibited for this TP!

3.1 Factorial

```
public static ulong Factorial(ulong n);
```

This function must return $n!$, n being a natural number, positive or null.

Warning

Do not forget that $0! = 1$.

3.2 Power

```
public static int Power(int a, int b);
```

For this function, you must implement the function Power, which returns a^b with $b >= 0$. If $b < 0$, you must display on the standard error output (`stderr`) the message: `N must be positive or null` followed by a new line and return 0.

Warning

For this function, you must not use the function `Math.Pow` or any other equivalent function.

3.3 Divisor Sum

```csharp
public static int DivisorSum(int n)
```

Return the sum of divisors of `n`, `n` excluded. If n is negative or equal to 0, you must display the error message `N must be positive` followed by a new line on `stderr` and your function must return `-1`.

```csharp
DivisorSum(1); // = 0
DivisorSum(2); // = 1
DivisorSum(6); // = 6
DivisorSum(42); // = 54
DivisorSum(0); // = -1
// N must be positive
```

3.4 PerfectNumber

```
public static bool PerfectNumber(int c);
```

For this function, you must test if a natural number n is a perfect number by using the function `DivisorSum` previously written.
A number is considered perfect if the sum of its divisors is equal to itself.

3.5 Decode Binary

```
public static int DecodeBinary(string s);
```

The goal of this exercise is to decode a number in binary in a character array composed of `'0'` and `'1'` to obtain its decimal value.

```
DecodeBinary(""); // = 0
DecodeBinary("1"); // = 1
DecodeBinary("10"); // = 2
DecodeBinary("101111101111100101000"); // = 1564456
```

Warning

Loops other than `while` are still forbidden.

Tips

A few advices:

- To know the number of characters in a string you can use the attribute `String.Length`. This is how you can use it:

```
var s = "My string";
var length = s.Length; // gets the length
```

- To transform a character representing a digit in `int`, use `c - '0'`.[a]
- To get powers of 2, you can either use the function `Power` or use `bitshifting`[b].

[a]Each character is represented by a natural number in the "ASCII table" that you saw in TP0, which explains why we are doing this operation. Thanks to the characters from `'0'` to `'9'` being together in the table, we can simply take the ASCII code of the character we want to convert and subtract to it the character `'0'`. We thus get the following: `'0'` becomes `0`, `'1'` becomes `1`, etc.

[b]Here is the MSDN documentation for more information about bitshifting: `https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/bitwise-and-shift-operators`

3.6 Crack The Code

```
public static int CrackTheCode(string code);
```

This exercise will allow you to help Sherlock solve a part of the case. Moriarty left a message composed of `0s` and `1s` but Sherlock is too busy to decipher this message. He left this task for you to solve, do not let him down! You already have all the keys to decipher this enigma. Before delegating the task of solving this to you, Sherlock discovered that the message contains a number in binary and that the clue is the first perfect number inferior or equal to it.

You function will thus:

- Transform `code` in a decimal number using your function `DecodeBinary`.
- Return the biggest perfect number smaller or equal to it.

Warning

If $x < 6$, you must return 0. **Your code must not loop infinitely**.

Tips

Even if we won't test the complexity of this function, you can greatly improve the search of the first perfect number by considering that a perfect number can only be an even number.[a]

[a]In theory, it is not impossible for a perfect number to be odd. However, we know for sure that there is no odd perfect number in values of the type `int`. For more information check this out: `https://www.wikiwand.com/en/Perfect_number#/Odd_perfect_numbers`

3.7 The Case of the Hanged Man

You were of great help! After cracking the code, Sherlock used his master detective mind to deduce, only using this number, the location of the hostage! How did he do that? Magic.
You rush towards the place, only to find a man ready to be hanged, and a chalk board with the inscription `"Guess the word correctly or he dies."` written on it.

4 Hangman

You would, clever detectives that you are, have understood by now that you will now have to implement the game of the Hangman. You will be guided all throughout this section, step by step, in order to produce the expected game.

Important

- You must not change the prototypes as your functions will be evaluated automatically.
- You are not allowed to add any headers (`using ...`)
- You are prohibited from using functions that are not explicitly allowed by the subject. Please only edit functions commented by //TODO

4.1 Tools

4.1.1 Loader.cs

For this section of the TP, we provide you with a file: `Loader.cs`, containing all the functions and global variables that you will need later on in order to implement the expected functions.

Warning

This file must not be edited.

You can find in it 2 functions, a string array `Ascii` containing the ASCII arts to be displayed and 5 global variables. We are going to show you how to access and use these different tools.

The global enum variable `GameState` allows you to register the current state of the game:

```csharp
public enum GameState
{
    RUNNING = 0,
    WON = 1,
    LOST = -1
}

// Example on ways to use this enum
Loader.GameState gameState = Loader.GameState.RUNNING;
gameState = Loader.GameState.Lost;

// Here nothing will be printed on stdout
if (gameState == Loader.GameState.WON) // <=> gameState == 1
    Console.WriteLine("You won!");
```

Tips

The enumerations allow the pairing of a label to a numerical value (`int`) to ease the readability of your code. If you want more information, you can read the following page: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum`

We also provide you with a global string array called `Ascii`, containing the different ASCII art possible of the game to display:

```csharp
public static readonly string[] Ascii = { ... }
// use it like this: Loader.Ascii[index]

// for index = 0, you get:
@"
+---+
|   |
|
|
|
|
========="

// etc ...

// for the latest index, you get:
@"
+---+
|   |
|
|   O
|  \|/
|  / \
=========
phewwww!"
```

Two more global variables are present in this file in order to display the correct ASCII art in case of a victory or a defeat. These variables (representing the position of such art in the string array `Ascii`), can be used as follows:

```csharp
public static readonly int Defeat = Ascii.Length - 2;
// use it like this: Loader.Ascii[Loader.Defeat]

public static readonly int Victory = Ascii.Length - 1;
// use it like this: Loader.Ascii[Loader.Victory]
```

The last global varibale in this file you can use is Attempts. It is used to store the amount of attempts the user has until the game ends in failure.

```csharp
public static readonly int Attempts = Ascii.Length - 2;
// use it like this: Loader.Attempts
```

Finally, 2 functions are present in this file: `GetWord` and `GetEmptyDuplicate`, defined as such:

```csharp
public static string GetWord(string path = "../../../Game/word_bank.txt");
public static char[] GetEmptyDuplicate(string word);
```

The first one loads a text file (by default located at the root of the folder `Game/`, meaning you can call this function without any arguments or you can call it with one argument overwriting the default path) and returns a random word from one line of the loaded text file.

The second one gets a string `word` as argument and creates a duplicate of it, as a character array, replacing every letter by a `'_'`.

4.1.2 Game.cs

In this file, there is only one global variable, `WordToGuess` which calls the function `GetWord` of `Loader.cs` to get a random word from a text file. You shouldn't have to add an argument to the function call (unless you want to change the path to the location of the text file).

```csharp
// You can access this variable from anywhere within this file.
private static readonly string WordToGuess = Loader.GetWord();
```

The rest of the file is yours to fill by completing the functions commented with //TODO

Warning

In this section, unless told otherwise, everything displayed on the console must be followed by a line break. Thus, prefer using the `WriteLine` function in order to write on the console.

4.2 GetInput

```csharp
public static char GetInput();
```

The first function to implement is a function reading on the standard input (`stdin`), to get a line with the function `Console.ReadLine`.

You must return the character read by the function. However, in the Hangman, the input must be letter, converted to its lowercase equivalent if needed. Hence, the following inputs are invalid:

- `'null'` which can be the return value of the function `Console.ReadLine`.
- An input with more than one character.
- An input that is not a valid letter. You can use the function `char.IsLetter`.

In all of the previous cases, you must return the character 0 and display `"Invalid Argument"` **on the standard error output** (`stderr`).

Tips

In order to have a more user-friendly interface, we are here using `Console.Readline` instead of `Console.Read`.
It allows us to pause the execution of the program and wait for a user input (terminated by a new line) before clearing the console, displaying new information on top.

4.3 DisplayWord

```
public static void DisplayWord(char[] guessedWord, string usedLetters);
```

The fonction `DisplayWord` must:

- Clear the console.
- Display the following string `"Your guess:  "`
- Followed by the all the letters in the character array `guessedWord` **on the same line**
- Finally, and still on the same line, display the string `", Used letters:  "` followed by the string `usedLetters` and a new line.

The output must match the following examples:

```
// guessedWord = "_______"
// usedLetters = ""
DisplayWord(guessedWord, usedLetters);
// stdout: "Your guess _______, Used letters: "

// guessedWord = "_in__n_"
// usedLetters = "poum"
DisplayWord(guessedWord, usedLetters);
// stdout: "Your guess _in__n_, Used letters: poum"
```

4.4 DisplayHangman

```
public static void DisplayHangman(string usedLetters,
Loader.GameState gameState);
```

The function displays the correct ASCII art, matching the current number of errors, on the standard output (`stdout`). The current number of errors can be retrieved via the parameter `usedLetters`. It must display a specific ASCII art if the `gameState` indicates that the user won/lost.

4.5 ContainsLetter

```
public static bool ContainsLetter(char[] guessedWord, char letter);
```

You must implement the function `ContainsLetter`, which checks if `guessedWord` contains the character `letter`. It returns `true` if `letter` was found and `false` otherwise.

Warning

You are not allowed to use the function `String.Contains` in this function

4.6 ValidateLetter

```
public static string ValidateLetter(char[] guessedWord, string usedLetters,
char letter, Loader.GameState gameState);
```

In this function, you must check if `letter` is contained in the global string variable `WordToGuess`. As previously stated, this variable is predefined at the top of the file and gets a random word from a loaded text file. If the `letter` was found in `WordToGuess` then update the `guessedWord` character array, by replacing at the right position(s) the `'_'` character(s) by the `letter` that was found. If the `letter` was not found, then you should add it to the string `usedLetters`.

There are 2 cases that you should be able to handle:

- If `letter` was already correctly guessed, you have to print on the standard error output (`stderr`) the following string: `"This letter has already been guessed!"`
- If `letter` was already used and not present in `WordToGuess`, you have to print on the standard error output (`stderr`) the following string: `"You already used that letter..."`

In both of the previous cases, after printing the error message, you must immediately return `usedLetters`. If you did not fall in one of the previous cases, call the functions `DisplayWord` and `DisplayHangman`, then return `usedLetters`.

Warning

Do not forget to update `guessedWord` for every occurrence of `letter`

Tips

You are allowed to used the function `String.Contains` to check if a character is contained in a string but should use `ContainsLetter` to check if a letter is contained in a character array.

You may convert a char array to a string and then use the function `String.Contains` but this is not recommended as the function `ContainsLetter` will be expected and tested

4.7 GameStatus

```
public static Loader.GameState GameStatus(char[] guessedWord,
string usedLetters, int attempts);
```

The goal of this function is to return the state of the game according to the variables given as parameters. The function must return:

- `Loader.GameState.LOST` if the number of errors given by `usedLetters` exceeded `attempts`
- `Loader.GameState.WON` if `guessedWord` matches `WordToGuess`
- `Loader.GameState.RUNNING` otherwise

4.8 EndScreen

```
public static void EndScreen(char[] guessedWord, string usedLetters,
Loader.GameState gameState);
```

In this function, you simply have to call the functions `DisplayWord` and `DisplayHangman`. After that, if the user won, display: `"You won!"` on the standard output `(stdout)` else, display: `"You lost :( The answer was "` followed by the string `WordToGuess`.

4.9 LaunchGame

```
public static void LaunchGame();
```

This last function will use all the previous functions and put them together to create the expected game! You first must initialize all required variables such as `gameState`, `attempts`, `guessedWord` and `usedLetters`. Then, the game must run as follows:

1. Call `DisplayWord` and `DisplayHangman`
2. As long as the game has to run, ask for an input
3. If the output is invalid: loop again, else update the variables.
4. Upon exiting the loop, print the correct end message according to the `gameState`

Tips

You should by now have a working game! Test it using `Program.cs` by calling `LaunchGame` using `Game.Game.LaunchGame();`.

Have a go at playing the Hangman with your favorite ACDC's logins! (any non letter character was removed to follow our game's rule)
To do so, change the path of the used word bank in the `Game.cs` file!

4.10 Case closed

Hurray! You managed to guess the word correctly! I guess this word could be of importance sooner or later because Moriarty doesn't simply play games. A master criminal mind like his thinks of everything and solving this case was probably only the first piece of a very elaborate plan.

5 Bonuses

You are free to implement as many bonuses as you would like as long as they are **specified in your README** and that **they do not change the expected behaviour of the functions you were asked to implement**. All of the bonuses must be implemented in the `Bonuses.cs` file.

5.1 Hangman Reloaded

What if we made our game better and more user-friendly?. Here are a few ideas to start you off!

1. A menu capable of launching the game or any other bonus you implemented!
2. A feature allowing the user to restart a game of Hangman!
3. Personalized ASCII arts! (for this bonus **only** you are allowed to change the `Loader.cs` file in order to add your arts. This won't count as impacting the expected behaviour of your game)

Tips

You are not restricted to only 9 arts! You can add as many as you want as global variables such as `Attempts`, `Defeat` and `Victory` are dependant on the size of the `Ascii` array and not hard-coded constants!

Warning

However, you must have the defeat ASCII art in second to last position in the `Ascii` array, as well as the victory ASCII art in last position.

5.2 More bonuses!

Hungry for more? Here are some other suggestions!

- Animated ASCII arts[4]
- 221B Baker Street, display the apartment complex where the famous detective lives. Additional points are given if it takes as argument `n`, the number of floors to be displayed

Clue

indice = GetWord(1000000101010 - 2 * 11101011);

There is nothing more deceptive than an obvious fact.

[4]The infamous SL Train for example