**1st recitation**

**Introduction to Unit Tests and CLion**
**Cycles, invariants and manipulation of integers, strings and vectors**

The main objective of this first practical lesson is to introduce you to the unit testing model (using GoogleTest) that will be used in all the lessons (and also in the practical tests). **At the end of this lesson you should make sure that you have downloaded the given code, that you have compiled and run it, and that you have understood how the tests are being done**.

**Note on the development environment:** we advise you to use **CLion** as the IDE for this course. If you prefer, you can use another environment/editor under your own responsibility.

- CLion: A cross-platform IDE for C and C++ (installation guide) (free academic license)
- C++ Compiler: Configure CLion on Windows (on Linux/Mac you can simply use GCC)

**Instructions:**

- Download the file `aed2324_p01.zip` from the course page and unzip it (it contains the `lib` folder, the `Tests` folder with the files `funWithCycles.cpp`, `funWithCycles.h` and `tests.cpp`, and the files `CmakeLists.txt` and `main.cpp`);
- In CLion, open a project by selecting the folder containing the files from the previous point;
- If you can't compile, perform "Reload CMake Project" on the `CMakeLists.txt` file;
- Implement it in the file `funWithCycles.cpp;`
- Note that all the tests are uncommented. They should fail when run for the first time (before implementation). As you solve the exercises, the respective tests should pass.

The base exercises in this lesson involve the creation of static functions of the `FunWithCycles` class. They are intended to "unleash" your C++ and allow you to exercise the use of cycles (e.g., `for` or `while`) while simultaneously checking that you have understood the notion of **invariant** given in the previous week's theory lessons.

Unless explicitly stated, you can assume that the test cases made to your programs are "small" and that correctness is your main concern (we will talk about efficiency in the following lesson).

**1. Palindromes**. Function to implement:

```
bool FunWithCycles::palindrome(const std::string & s)
```

This function should return `true` if the string `s` is a palindrome and false otherwise. A string is a palindrome if it is exactly the same as its inverse (read backward).

*Example call and expected output:*

```
cout << FunWithCycles::palindrome("abba") << endl;
cout << FunWithCycles::palindrome("abcdba") << endl;
cout << FunWithCycles::palindrome("reviver") << endl;
cout << FunWithCycles::palindrome("revive") << endl;
1
0
1
0
```

*Explanation*: "*abba*" and "reviver" are palindromes; "*abcdba*" and "*revive*" are not.

**a)** Start by compiling and running the project and check that there are tests that fail (familiarize yourself with the interface); take a look at `funWithCycles.cpp` to see how the code only returns `false` and, therefore, fails the cases where it should give `true` and take a look at `test.cpp` to see how the tests are implemented.

**b)** Place the following code inside the `palindrome` function and check that the tests are now correct (note: there were other ways of doing the exercise).

```
unsigned length = s.size();
for (unsigned i = 0; i < length/2; i++)
   if (s[i] != s[length-i-1])
       return false;
return true;
```

**c)** As explained in the lectures, an **invariant** is a condition that is always true immediately before (and after) each iteration of the cycle and captures the essence of what the algorithm does (it is the "what", while the instructions are the "how"), and its veracity at the end of the cycle must necessarily imply the correctness of the algorithm. Can you identify which invariant is maintained in this piece of code?

**d)** A loop **variant** allows us to reason about the termination of a loop. It should be a non-negative integer expression whose value decreases with each loop execution. Can you identify a loop variant in this piece of code?

**2. Palindrome sentences**.

Function to implement:

```
bool FunWithCycles::palindromeSentence(const std::string & s)
```

This function should return `true` if the string `s` is a palindrome sentence and `false` otherwise. A string is a palindrome sentence if it is a palindrome when we ignore all non-alphabetic characters (such as spaces, punctuation, and numbers) and if we ignore the capitalization of letters (e.g., for example, 'a' should be considered the same character as 'A').

*Example call and expected output:*

```
cout << FunWithCycles::palindromeSentence("Madam, I'm Adam") << endl;
cout << FunWithCycles::palindromeSentence("O bolo do  lobo") << endl;
cout << FunWithCycles::palindromeSentence("o bolo d lobo") << endl;
cout << FunWithCycles::palindromeSentence("Anotaram a data da maratona") << endl;
```

```
1
1
0
1
```

*Tip*: Start by creating a new `string` containing the sentence with only lowercase letters ("ignoring") everything that is not a letter, and converting all the letters to lowercase (e.g., "Madam, I'm Adam" would be "madamimadam"). To do this, you can use the methods in the `<cctype>` library (see the documentation on cpluplus.com or cppreference.com), which is already in your project's includes. Then, all you have to do is... call the `palindrome` method from exercise 1 with this string you have created!

*Note*: If you used a loop in the code, can you identify the invariant?

**3. Interesting numbers.**

Pedro noticed that the number 95726184 was written under his laptop. Curious as he was, he noticed that the sum of its digits was 42. In fact, 9+5+7+2+6+1+8+4=42, which made Pedro think that this was a really "interesting" number (because 42 is the meaning of life).

Pedro and Vanessa thought they could create a game to challenge Ana, António, Filipa, Bernardo and João, which consists of finding the first "interesting" number greater than a given number. To prevent them from simply trying to memorize answers, they decided they were interested in numbers whose sum of the digits was also a number of their choice, not just 42.

Function to implement:

```
int FunWithCycles::nextInterestingNumber(int n, int sum)
```

This function should return the smallest number greater than `n` whose sum of digits is exactly the `sum`.

*Restrictions*: in this exercise, in all the tests carried out it is guaranteed that $1 \leq n \leq 10^9$, that $1 \leq sum \leq 50$, and that the number to look for will never be more than 100,000 numbers above `n`.

*Example call and expected output:*

```
cout << FunWithCycles::nextInterestingNumber(95726184, 42) << endl;
cout << FunWithCycles::nextInterestingNumber(424242, 42) << endl;
cout << FunWithCycles::nextInterestingNumber(1, 25) << endl;
cout << FunWithCycles::nextInterestingNumber(299, 13) << endl;
```

```
95726193
429999
799
319
```

*Explanation*: 95726193 is the smallest number greater than 95726184 whose digits add up to 42.
429999 is the smallest number greater than 424242 whose digits add up to 42.
799 is the smallest number greater than 1 whose digits add up to 25.
319 is the smallest number greater than 299 whose digit sum is 13.

*Tip*: Start by implementing the helper function `int FunWithCycles::digitSum(int n)` so that it returns the sum of the digits of `n` (e.g. `digitSum(123)` should return 1+2+3=6). Then just run a cycle that tries all the consecutive numbers after `n`, stopping when it finds one that has the desired sum of digits (using the auxiliary function on each number).

*Note*: If you used a loop(s) in the code, can you identify the invariant?

**4. Winter is coming.**


Boromir has noticed that temperatures are dropping in Gondor. Looking back over the last 10 days, the temperatures have been as follows:

```
23 24 22 21 18 17 17 22 14 13
```

He calculated the temperature differences between consecutive days and obtained the following result:

```
+1 -2 -1 -3 -1 0 +5 -8 -1
```

Boromir then wanted to determine the longest sequence of days in a row when the temperature always fell. For these 10 days, this sequence is size 4 (`-2 -1 -3 -1`), while there is another sequence of size only 2 (`-8 -1`).

Function to implement:

```
int FunWithCycles::winter(const vector<int> & v)
```

Given a vector `v` with the temperatures of the last few days, this function should return the longest length of a consecutive sequence of temperature drops.

*Example call and expected output:*

```
cout << FunWithCycles::winter({23,24,22,21,18,17,17,22,14,13}) << endl;
4
```

## Extra Exercises

The following exercises allow you to consolidate and refresh the knowledge acquired in previous courses.

**5. Playing with vectors**.

**a)** Function to be implemented:

```
int FunWithCycles::count(const vector<int> & v, int n)
```

Given a vector `v` and an integer `n`, this function should return the number of occurrences of `n` in the vector.

*Example call and expected output:*

```
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 1) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 2) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 3) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 4) << endl;
```

```
2
4
1
0
```

*Explanation*: in the indicated vector exist:

- 2 occurrences of 1
- 4 occurrences of 2
- 1 occurrence of 3
- 0 occurrences of 4

**b)** Function to be implemented:

```
bool FunWithCycles::hasDuplicates(const vector<int> & v)
```

Given a vector `v`, this function should return `false` if all the integers in the vector occur only once and `true` otherwise (i.e. if there is at least one duplicate element).

*Example call and expected output:*

```
cout << FunWithCycles::hasDuplicates({1,2,3,2,6,1,2,2}) << endl;
cout << FunWithCycles::hasDuplicates({2,4,6,8,3,5,7}) << endl;
cout << FunWithCycles::hasDuplicates({10,20,30,10}) << endl;
```

```
1
0
1
```

*Explanation*: the first and third vectors have duplicate elements; the second vector does not.

**c)** Function to be implemented:

```
void FunWithCycles::removeDuplicates(vector<int> & v)
```

Given a vector `v`, it should remove all duplicate occurrences of an integer, keeping only the first occurrence of each.

*Example call and expected output:*

```
vector<int> v = {1,2,3,2,6,1,2,2};
FunWithCycles::removeDuplicates(v);
for (int i : v) cout << i << " ";
cout << endl;
```

```
1 2 3 6
```

*Explanation*: the occurrences indicated in red are duplicated {1,2,3,**2**,6,**1**,**2**,**2**}

**d)** Function to be implemented:

```
vector<int> FunWithCycles::merge(const vector<int> & v1, const vector<int> & v2)
```

Given two vectors `v1` and `v2` sorted in ascending order, this function should return a new vector also sorted in ascending order containing all the integers in `v1` and `v2`.

*Example call and expected output:*

```
vector<int> v = FunWithCycles::merge({1,5,5,7}, {2,3,6});
for (int i : v) cout << i << " ";
cout << endl;
```

```
1 2 3 5 5 6 7
```

*Explanation*: the resulting vector is {**1**,**2**,**3**,**5**,**5**,**6**,**7**} and is sorted in ascending order, containing all the elements of {1,5,5,7} (in **red**) and {2,3,6} (in **green**).

**6. Playing with prime numbers.**

**a)** Function to be implemented:

```
bool FunWithCycles::isPrime(int n)
```

Given an integer n > 1, this function should return `true` if the number is prime and `false` otherwise. An integer greater than 1 is prime if it is only divisible by 1 and itself.

*Example call and expected output:*

```
cout << FunWithCycles::isPrime(13) << endl;
cout << FunWithCycles::isPrime(15) << endl;
cout << FunWithCycles::isPrime(49) << endl;
cout << FunWithCycles::isPrime(89) << endl;
```

```
1
0
0
1
```

*Explanation*: 13 and 89 are prime numbers; 15 and 49 are not.

**b)** Function to be implemented:

```
vector<int> FunWithCycles::factorize(int n)
```

Given an integer `n` > 1, this function should return a vector containing the prime numbers that make up the factorization of the number. The prime factors must come in ascending order.

*Example call and expected output:*

```
vector<int> v1 = FunWithCycles::factorize(1176);
for (auto i : v1) cout << i << " ";
cout << endl;
vector<int> v2 = FunWithCycles::factorize(3037);
for (auto i : v2) cout << i << " ";
cout << endl;
```

```
2 2 2 3 7 7
3037
```

*Explanation*: 1176 = 2 * 2 * 2 * 3 * 7 * 7 and 3037 is a prime number.

**c)** Function to be implemented:

```
vector<int> FunWithCycles::listPrimes(int n)
```

Given an integer `n` > 1, this function should return a vector containing all the prime numbers less than or equal to `n`.

*Example call and expected output:*

```
vector<int> v = FunWithCycles::listPrimes(100);
for (auto i : v) cout << i << " ";
cout << endl;
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89
97
```

*Explanation*: all primes less than or equal to 100 are listed.

*Note*: To calculate prime numbers efficiently, you should not use a naive strategy. We suggest that you read up on and implement a Sieve of Eratosthenes.