OBJECT ORIENTED PROGRAMMING

LABORATORY 3

OBJECTIVES

In the laboratory you will learn how to define custom data types (*struct*s) in C/C++. Also, all the code written in this laboratory will follow a modular programming approach, in which the interface (*the what?)* of a module is separated of the implementation (*the how?*) of the module.

Another important topic covered in this laboratory is memory management, and you'll practice all the memory related notions (allocation, pointers) you learned about at the lecture. Finally, at the end of the laboratory, to test your skills and understanding related to memory management in C/C++, we'll organize a contest: you will pair up with a colleague and try to find all the errors in the provided code.

PROPOSED PROBLEMS

1. Define a structure for representing a complex number, something similar to the std::complex from C++ (https://en.cppreference.com/w/cpp/numeric/complex).
   Use a modular programming approach! Your *struct* should have two members, one for the real and one for the imaginary part of the number. Then write several functions for:)

   - Creating and reading from standard input a complex number:

```
complex create(float real, float imag);
complex read();
```

   - Displaying a complex number:

```
void display(complex c);
```

   - Computing the basic arithmetical operations on two complex numbers: addition, subtraction and multiplication.

```
complex add(complex c1, complex c2);
complex subtract(complex c1, complex c2);
complex multiply(complex c1, complex c2);
```

   - Checking if two complex numbers are equal:

```
bool equal(complex c1, complex c2);
```

- Computing the complex conjugate of a complex number.

```
complex conjugate(complex c);
```

- Multiplying a complex number with a scalar:

```
void multiply(complex* c, float s);
```

- Computing the magnitude and the phase of a complex number. Using these functions express the complex number in polar coordinates.

```
double magnitude(complex c);
double phase(complex c);
void toPolar(complex c, float *r, float* theta);
```

- Computing the division of two complex numbers: use the other function you wrote (conjugate, multiply, multiply by a scalar)!

```
complex divide(complex c1, complex c2);
```

Now, in another file (test.cpp), test the module that you wrote:

a. Define a variable of *complex* type such that it is stored on the stack.
   Call the functions *magnitude, phase, toPolar, conjugate* and *multiply* (with a scalar) for this variable.
b. Define a variable of *complex* type such that is stored on the head and allocate memory for it.
   Call the function *magnitude, phase, toPolar, conjugate* and *multiply* (with a scalar) for this variable.
   Now compute the multiplication, addition, subtraction and division between these two variables.

Don't forget to release the memory you allocated on the heap.

2. Finish (if you haven't already done it already) the implementation of the dynamic array module from the seminar. This implementation should also follow a modular programming approach.

3. Using the two modules that you wrote (the one for the complex numbers and the one for the dynamic array), create a dynamic array of complex numbers and populate it with the data stores in the file *complex_numbers.txt*.
   In this file, each complex number is stored on a different line, with the real and imaginary parts separated by space:
   [real_n1] [imag_n1]
   [real_n2] [imag_n2]

   Keep in mind that you don't know how many complex numbers there will be on this file!
   Then, find the complex number from this array that has the maximum magnitude.

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.

CONTEST

Find the memory errors in all the projects from the solution MemoryErrors.
Fill in the Readme.md file from *git* to describe the errors you spotted and how you fixed them. You can work in teams of 2 students for this purpose.