Object Oriented Programming with C++
Labwork 3 – *Classes and objects*

NB: you should use the `const` qualifier everywhere it is possible to do it.

Exercise 1: Image thresholding

Restarting from first labwork exercise 3, transform your `Image` structure (or class for some of you) into a *correct* C++ class. It consists to suppress most of the independent functions (not being methods), and to use both correct constructor and destructor (for dynamic memory). Obviously, you have to follow pure OOP (in other words, hide all the data members). To simplify, you have to use the following definition (header file):

```
class Image {
  unsigned      m_width;
  unsigned      m_height;
  unsigned char m_code;   // 1 or 6
  unsigned char *m_pixels;
  Image();                // private!
 public:
  // constructor; have to allocate properly the data member "m_pixels"
  Image(const unsigned width, const unsigned height, const unsigned char code);
  // destructor
  ~Image();
  // factory pattern
  static Image read(const char*const filename);
  // others member functions
  Image convert( const float threshold ) const;
  void  writeP1( const char*const filename ) const;
};
```

NB: here the main goal is not technical, but concerns the contract binding the methods offered by your class to the users; think object!

Exercise 2: Complex numbers, round 2

Write a class to manipulate complex numbers. The internal representation can be either Cartesian or Polar, as you want. Use two methods implementing the *Factory* pattern, building a Complex instance respectively from Cartesian or Polar coordinates. Obviously, the data members have to be private (pure OOP), like any constructor having parameters (you may propose public constructor without argument, for array usage). Then write some others methods for manipulating complex numbers:

- 4 getters and 4 setters to access Cartesian and Polar coordinates.
- Addition, subtraction, product and division of two complex numbers, that return a new complex number. With these functions, you should be able to write something like:

```
Complex example( const Complex&a, const Complex&b ) {
  const Complex c = Complex::fromCartesian(1.0, -1.0); // real=1, imaginary=-1
  return a.add(b).prod(c); // return (a+b)*c
}
```

- Equality of two complex numbers, that returns a Boolean.
- Product by a real number that returns reference to current instance.

Then, build a program for testing your class (each method should be tested with some different values, and by composing operations (*e.g.* $a + b$ should not equals a nor b but $(a + b) - b$ should equals a ...).

NB: with Cartesian implementation, the angle can be obtained with the following code excerpt.

```
  if (real == 0.) {
    // + or - PI/2
    return imaginary < 0. ? -PI/2 : imaginary>0. ? PI/2 : 0.;
  }
  double at = atan (imaginary / real);
  return real > 0. ? at : (imaginary < 0. ? at-PI : at+PI);
```