OBJECT ORIENTED PROGRAMMING

LABORATORY 3

OBJECTIVES

With this laboratory, we'll step into the realm of Object-Oriented Programming (OOP). You will study how to write simple classes in C++, how to use encapsulation and abstraction, different types of constructors a class can have, and how to overload operators.

THEORETICAL NOTIONS

Features of object-oriented programming

Object-oriented programming (OOP) is a programming paradigm built on the concept of "objects", which contain both data and behavior. The main OOP features are (APIE):

- **A**bstraction - the process of identifying the essential features of an object, and ignoring the non-essential details. It simplifies the complexity of the code, making it easier to manage and maintain.
- **P**olymorphism is the ability of objects of different classes to be treated as if they are of the same type. This allows you to access objects of different types through the same interface. Will be covered in detail in a future lab.
- **I**nheritance is a mechanism that allows a new class to be based on an existing class, inheriting its properties and methods. This allows for the reuse of code, and makes it easier to manage large, complex programs. Will be covered in detail in a future lab.
- **E**ncapsulation is a mechanism for protecting the access to data and methods of an object from the outside world, and exposing only what is necessary. This helps to ensure that the data is not accidentally or intentionally modified by external code. In addition, encapsulation involves bundling data and methods that work on that data within one unit (the class).

Classes and objects

In C++, a class is a blueprint for creating objects. It defines a set of attributes (member variables) and behaviors (member functions) that are common to all objects of that class. An object, on the other hand, is an instance of a class. It has its own set of values for the attributes defined in the class and can perform several operations as defined by the member functions. Each member function of a class has access to the ***this*** pointer (similar to self in Python), which refers to the current object

instance within a member function. Unlike in python, the ***this*** pointer is passed as a hidden argument to all non-static member functions (i.e. you don't need to explicitly add it as a function parameter).

Constructors are special member functions of a class that are automatically invoked when an object of that class is created. The purpose of a constructor is to initialize the data members of the class to some initial values.

In this laboratory, we'll study about two types of constructors in C++:

> The default constructor is a special member function that takes no arguments and it is used to initialize the attributes of the class to their default values. The default constructor does not return anything (not even void) and has the same name as the class. If a class does not define any constructors, then the compiler automatically generates the default constructor.
>
> Parameterized constructors are constructors that have at least one argument. The arguments passed to the constructor are used to initialize the data members of the class.

A class can have multiple constructors with different arguments. This feature (having several functions with the same name but with different signatures) is called method overloading.

Access modifiers in C++

Access modifiers in C++ are keywords used to control the accessibility of the class members (attributes and functions). There are three access modifiers in C++: public, private, and protected:

- *Public*: Public members are accessible from anywhere in the program, including outside of the class. They are declared using the keyword "public".
- *Private*: Private members are not accessible from outside the class. They can only be accessed from within the class. They are declared using the keyword "*private*".
- *Protected*: Protected members can be accessed by derived classes (classes that inherit from the base class). They are declared using the keyword "protected" in the class definition. We'll cover this in the next laboratory session.

Specifier	**Member accessible?**		
	Within class	**In derived class**	**Outside the class and the derived class**
default	YES	NO	NO

private	YES	NO	NO
protected	YES	YES	NO
public	YES	YES	YES

Friend functions and classes

A friend function in C++ is a function that is not a member of a class, but has access to the private and protected members of the class. It is declared inside the class using the keyword "**friend**". To declare a friend your need to do the following steps:

- Declare the function (add the header of the function) inside the class and use the keyword *friend*.
- Define the friend function (write the actual implementation of the function) outside the class. Because the friend function does not belong to the class, when defining the function you don't need to use the scope resolution operator "::", as you do for class member functions.

point.h

```
#include <iostream>
#include <cmath>

class Point {
private:
    double x;
    double y;

public:
    Point(double x_val, double y_val) : x(x_val), y(y_val) {}

    // friend function declaration to calculate distance between two points
    friend double distance(const Point& p1, const Point& p2);
};
```

main.cpp

```
#include “point.h”

// the friend function is defined in the main.cpp function, without using
the scope resolution operator
double distance(const Point& p1, const Point& p2) {
     // the function has direct access to the private members of the class
    return std::sqrt(std::pow(p2.x - p1.x, 2) + std::pow(p2.y - p1.y, 2));
}

int main() {
    Point p1(1.0, 2.0);
    Point p2(4.0, 6.0);

    double dist = distance(p1, p2);
    std::cout << "Distance between points: " << dist << std::endl;

    return 0;
}
```

Similar to friend functions, a friend class is a class that is granted access to the private and protected members of another class. To make another class a “friend” of a class, you just need to add the following in the class declaration:

friend class <MyFriendClass>;

, where <MyFriendClass> is the name of the friend class (the class that gets access to the class' private and protected members).

Static members

Static members in C++ are class members (variables and functions) that are shared by all instances (objects) of the class, rather than being specific to each object of the class (to each instance). In other words, they are associated with the class itself.

Static member variables are declared using the keyword "static" and are shared by all instances of the class. Member variables of a class can be made static by using the *static* keyword in front of their type. Unlike normal member variables, static member variables are shared by all objects of the class. Static

members exist even if no objects of the class have been instantiated. Static variables need to be explicitly defined. Therefore to use a static class variable you need to do the following:

- First, we declare the static member variable inside the class using the *static* keyword. With this, we're just letting the compiler know about the static member variable, but it is not a definition.
- Secondly, you must explicitly define the static member outside of the class, in the global scope.

Static member functions are also declared using the keyword *static* and operate on the static data members of the class. They can be called without having an instance of the class, just by using the class name, followed by the scope resolution operator "::" and the name of the static function. Static member functions don't have access to *this* pointer.

Operator overloading

Operator overloading in C++ is the ability to redefine the behavior of operators (+, -, *, /, &, etc.) for user-defined types (classes and structures). By overloading operators, you can provide a more natural syntax for your class and make it easier to work with.

You can overload operators either as class members (functions) or as friend functions.

Example: A C++ class for a 2D Point, in which we illustrate all the concepts described above.

point.h

```
#pragma once
#include <fstream>
using std::ifstream;
using std::ofstream;

class Point
{
	// the default access modifier in C++ is private
public:
	// here are the public members of the class
	Point(); // default constructor
	Point(float x, float y); // parametrized constructor
```

```
      // getters for the attributes of the class
      // const -  the this-> pointer is transformed to a constat pointer
                                        // we are not allowed to change the
state of the object
      float x() const;

      // inline methods can only be defined in the header of the class
      inline float y() const { return m_y; }


      // setters for the attributes of the class
      void setX(float x);
      void setY(float y);


      // arithmetic operator overloading

      // Overloaded + operator for adding two points. this operator is a
class member
      Point operator+(const Point& other) const;

      // Overloaded - operator for subtracting two points. this operator is
overloaded using friend functions
      // i.e. it is not a class member
      friend Point operator-(const Point& p1, const Point& p2);


      // stream operators overloading. these can only be overloaded as
friend functions
    // overloaded stream insertion operator for "printing" a point
      friend std::ostream& operator<<(std::ostream& os, const Point& point);

      // Overloaded stream extraction operator for reading in a point
      friend std::istream& operator>>(std::istream& is, Point& point);

      // static function to get the number of created instances using the
parametrized constructor
      static unsigned long long numCreatedParametrized();
private:
      // these are the private attributes of the class
      float m_x; // x coordinate
      float m_y; // y coordinate

      // static member to count how many objects have been created
      // it is defined in point.cpp
      static unsigned long long CREATED_INSTANCES_PARAM;

};
```

point.cpp

```
#include "Point.h"
#include<iomanip>


// definition of the static variable (we initialize it to 0)
// we don't use the static keyword anymore, but we use the scope resolution
operator
unsigned long long Point::CREATED_INSTANCES_PARAM{ 0 };

Point::Point() : m_x{ 0.0f }, m_y{0.0f} // uniform initialization
{}

Point::Point(float x, float y)
{
	m_x = x; m_y = y;
	CREATED_INSTANCES_PARAM++; // another instance was created
}

float Point::x() const
{
	return this->m_x;
}

void Point::setX(float x)
{
	m_x = x;
}

void Point::setY(float y)
{
	m_y = y;
}

// notice that operator+ belongs to the Point class Point::operator+
// :: -> scope resolution operator
Point Point::operator+(const Point& other) const
{
	return Point(m_x+other.m_x, m_y+other.m_y);
}

unsigned long long Point::numCreatedParametrized()
{
	// we don't have access to this pointer
	return CREATED_INSTANCES_PARAM;
}

// notice that operator- is not a member of the class
// we don't use the friend keyword when defining the function
Point operator-(const Point& p1, const Point& p2)
```

```
{
	return Point(p1.m_x - p2.m_x, p1.m_y - p2.m_y);
}

// not a member of the class
// we don't use the friend keyword when defining the function
std::ostream& operator<<(std::ostream& os, const Point& point) {
	os << "(" <<std::setprecision(2)<<std::fixed << point.m_x << ", " <<
point.m_y << ")";
	return os;
}

// not a member of the class
// we don't use the friend keyword when defining the function
std::istream& operator>>(std::istream& is, Point& point) {
	is >> point.m_x>> point.m_y;
	return is;
}
```

main.cpp

```
#include "point.h"
#include <iostream>
using namespace std;
int main(void)
{
    Point p1{}; // default constructor
    // use cin to read the coordinates of a point
    cout << "Please enter the coordinates of the point separated by space:"
<< endl;
    // stream extraction operator
    cin >> p1;
    // stream insertion operator
    cout << "You entered: "<<p1<<endl;

    Point p2{ 10, 20 }; // parametrized constructor

    // add the two points
    Point p3{ p1 + p2 };
    // or, because the addition operator is a member of the class, we can do
    Point p4{ p1.operator+(p2) };

    cout << p1 << "+" << p2 << "=" << p3 << endl;
    cout << p1 << "+" << p2 << "=" << p4 << endl;

    cout << p1<<"-"<<p2<<"="<<p1 - p2<<endl;
    // calling a static method
    cout << "There were " << Point::numCreatedParametrized() << " objects
```

```
created with the parametrized constructor" << endl;
    cout << "The end" << endl;
    return 0;
}
```

std::string in C++

C++ std::string is one of the classes defined in the Standard Template Library (STL). It is a dynamic array of characters, allowing for the dynamic allocation and deallocation of memory as needed. Unlike traditional C-style strings (i.e. array of characters), std::string automatically manages memory, eliminating the need for manual memory allocation and deallocation.

```
#include <iostream>
#include <string>

int main() {
    // Creating an empty string
    std::string emptyString;

    // Initializing a string with a literal
    std::string greeting = "Hello, World!";

    // Concatenating strings
    std::string firstName = "John";
    std::string lastName = "Doe";
    std::string fullName = firstName + " " + lastName;

    // Displaying the created strings
    std::cout << "Empty String: " << emptyString << std::endl;
    std::cout << "Greeting: " << greeting << std::endl;
    std::cout << "Full Name: " << fullName << std::endl;
    return 0;
}
```

PROPOSED PROBLEMS

1. Design a simple C++ class named BankAccount to represent a basic bank account. The class should have attributes for the account holder's name, account number, and balance. Include

methods for depositing and withdrawing funds, as well as the stream insertion operator to display the account details.

a. Implement a parameterized constructor to initialize the account with the account holder's name, account number, and initial balance.
b. Include methods for depositing and withdrawing funds, adjusting the balance accordingly. Think about how you will handle the case in which the user wants to withdraw more money than the available amount.
c. In the stream insertion operator add the account details: the account holder's name, account number, and current balance.

2. Write a class to represent a complex number. This is similar to the std::complex from C++ (https://en.cppreference.com/w/cpp/numeric/complex). Use a modular programming approach! Your class should have two members, one for the real and one for the imaginary part of the number. In addition, your class should contain at least:

a. Default constructor and a constructor that initializes a complex number with the real and imaginary parts.
b. Overloads for the equal to and not equal to operators.
c. Computing the magnitude and the phase of a complex number.
d. A private static member of an enum type *DISPLAY_TYPE*. It can have two values: RECTANGULAR_FORM and POLAR_FORM. Generate getters and setters for this member.
e. Overload for the stream insertion operator (<<) and stream extraction (>>) operators. In the stream insertion operator check the value of *DISPLAY_TYPE*. If *DISPLAY_TYPE* is RECTANGULAR_FORM then the complex number will be displayed in rectangular form (re + img*i), otherwise, it will be displayed in polar form z= mag*(cosθ+i sinθ). The same applies for the *toString* method.
f. Overloads for the arithmetic operators: addition (+), subtraction (-) and multiplication(*).
g. Overload for multiplying the complex number with a scalar value.

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

- Define a variable of *Complex* type such that it is stored on the stack. Call the functions that compute the magnitude, the phase and the polar form of this complex number.
- Define a variable of *Complex* type such that is stored on the head and allocate memory for it. Call the functions that compute the magnitude, the phase and the polar form of this complex number.
- Now compute the multiplication, addition, subtraction and division between these two variables.

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

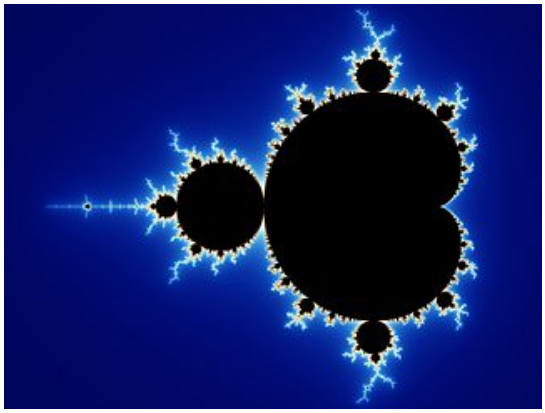
This will prove useful when implementing the stream insertion operator for the complex number: operator<<

To convert a floating point number to a string and format it to have only 2 decimal places, you can use:

```
ostringstream sstr;
sstr<<std::setprecision(2)<<std::fixed;
float v = 10.50f;
sstr<<v;
string result = sstr.str();
```

The variable result will be result.
Don't worry about the notation, you'll learn all about this in lecture 6(ish).

2. We all know the famous Mandelbrot fractal, and, as we will see, it is very easy to generate using complex numbers. Start by watching this awesome explanation of the Mandelbrot set: [The Mandelbrot Set - Numberphile - YouTube]



Formally, the *Mandelbrot set* *M* is defined as the family of complex quadratic polynomials:

$$f_c : \mathbb{C} \to \mathbb{C}, \quad f_c(z) = z^2 + c,$$

where *c* is a complex number. For each *c*, we consider the sequence < 0, $f_c(0)$, $f_c(f_c(0))$, $f_c(f_c(f_c(0)))$, .. > obtained by iterating over the function f_c starting from z = 0. The absolute value of any element in this sequence, either remains bounded or escapes to infinity. We say that a complex number *c* is a member of the Mandelbrot set if, when starting with z = 0, and applying the iteration repeatedly, the absolute value of the elements in the generated sequence remain bounded .
Use the complex class that you wrote to display the Mandelbrot fractal based on the starter code below:

```
#include <stdio.h>
#include <stdlib.h>
```

```
void display_mandelbrot( int width, int height, int max_its)
{

    const float x_start = -3.0f;
    const float x_fin = 1.0f;
    const float y_start = -1.0f;
    const float y_fin = 1.0f;

    double dx = (x_fin - x_start)/(width - 1);
    double dy = (y_fin - y_start)/(height - 1);

    for(int y = 0; y < height; ++y)
    {
        for(int x = 0; x < width; ++x)
        {
            // TODO your code here
            // create complex number z = 0 + 0i
            // create complex number c =  x_start+ x*dx + (y_start+y*dy)i

            int iteration = 0;
            // while |z| < 2 and we haven't reach max_its
            while(/* TODO your code here: |z| < 2&& */++iteration < max_its){
               // apply the rule:  z =  z*z + c
            }

            // TODO: your code here (modify the code to display the mandelbrot
fractal
            if(iteration == max_its){
                printf( "*");
            }else{
                printf( "-");
            }

        }
        printf("\n");
    }
}

int main(void)
{
    display_mandelbrot(100, 25, 100);
    getchar();
    return 0;
}
```

Then, pass a pointer to a function that takes as input an integer (the iterations) and displays the Mandelbrot points.

To print coloured characters in C++, you need to use ANSI color codes:

https://en.wikipedia.org/wiki/ANSI_escape_code

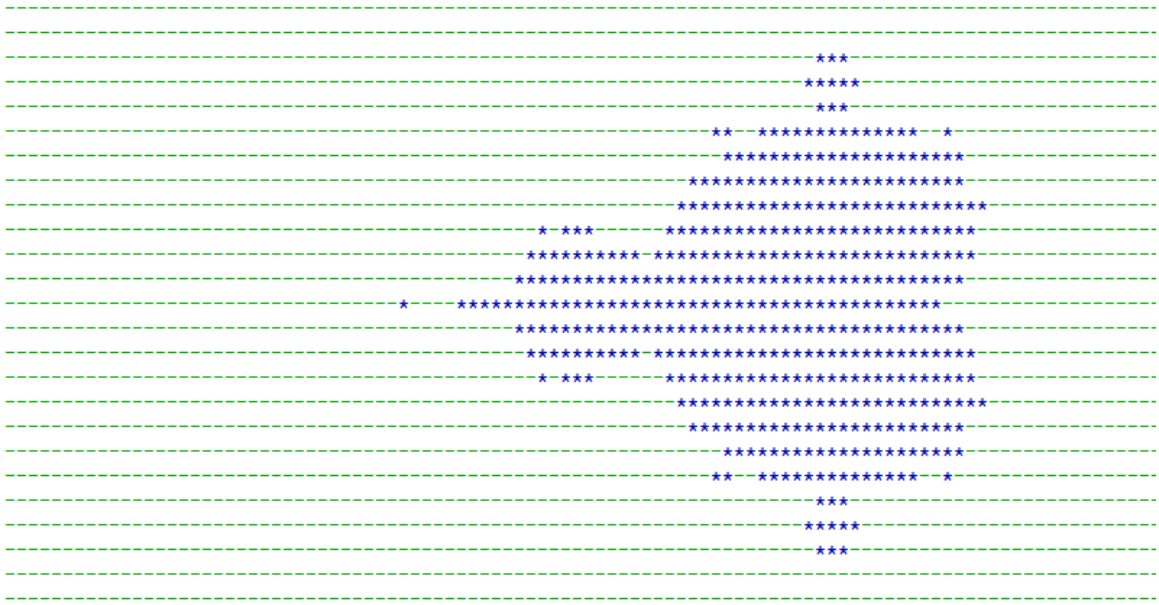
For example, this code prints a bold yellow text with a red background:

```
std::cout << "\033[1;33;41mbold yellow text with red background\033[0m\n";
```

\033 is the ESC character. It is followed by [, then several numbers separated by ;, and finally the letter m. These letters encode the color of the text:

	foreground color	**background color**
black	30	40
red	31	41
green	32	42
yellow	33	43
blue	34	44
magenta	35	45
cyan	36	46
white	37	47

Post your results on Teams:



3. Write a *Date* class in C++ to manage date-related operations in a program. Your class should have at least the following methods:

a. Initialization: Implement both a default constructor (that initializes the Date object to the standard UNIX time 1/1/1970) and a parameterized constructor that initializes the Date object with the given year, month, and day.
b. The stream insertion operator to display the date in the format: YYYY-MM-DD
c. Comparison operators (==, !=, <, <=, >, >=) to compare two Date objects.
d. A method to add a specified number of days to the current date and update the Date object accordingly.
e. Implement a static method that checks if a provided year, month, and day form a valid date:

```
static bool isValidDate(int year, int month, int day);
```

Write a separate class *DateTest* containing only public static methods to test the *Date* class.