OBJECT-ORIENTED PROGRAMMING

LABORATORY 5

OBJECTIVES

In this laboratory session, you will learn about an important OOP feature, inheritance.

THEORETICAL NOTIONS

Inheritance

Inheritance is one of the main features of Object-Oriented Programming (along with encapsulation, abstraction, and polymorphism). Inheritance is a mechanism that allows creating new classes (derived classes) based on existing (base classes); the derived class "inherits" the public and protected members (attributes and methods) of the base class. Therefore, the main advantage of using inheritance is code reuse.

Let's take the example of an application like *MSPaint* which allows you to draw and move around the screen basic shapes like squares, rectangles, circles, etc. All the shapes in the application have a center (described by the shape's x and y coordinates), and share a set of common behaviors: display information about a shape, draw a shape, translate a shape, etc.. In addition, a shape can have additional attributes and behaviors, such as the width and the height in the case of a *Rectangle,* the side in the case of a *Square,* and the radius in the case of a *Circle*. In this case, it makes sense to use inheritance.

You can create a base class for a "generic" *Shape* that has as attributes the center of the shape, and as behaviors, methods to move (translate) the shape and to display the position (center) of the shape. Then you can create subclasses of *Shape* for the *Rectangle, Square* and *Circle*. These subclasses inherit the center attribute of the *Shape* class, and its behavior: translate and display information about the shape.

Below you have an inheritance example between the *Shape* and *Circle* classes. *Shape* is called the parent, the superclass, or the base class, while *Circle* is called the child, the subclass, or the derived class.

```
#include <iostream>
using namespace std;

class Shape {

protected:
    // coordinates of the center of the shape
    // protected - can be accessed by the derived classes
    double m_x;
    double m_y;
public:
    Shape(double x, double y) {
        m_x = x;
        m_y = y;
    }

    // this method is inherited by the derived class
    void translate(double dx, double dy) {
        m_x += dx;
        m_y += dy;
    }

    void displayInfo(){
        cout<<"Centroid: ("<<m_x<<", "<<m_y<<")"<<endl;
    }
};

class Circle : public Shape
{
private:
    double m_radius;
public:
    // we call the base class constructor Shape{x, y}
    // this will initialize the inherited m_x and m_y attributes
    Circle(double x, double y, double r) : Shape{x, y} {
        m_radius = r;
    }

    // redefine the displayInfo method
    void displayInfo(){
        cout<<"Circle:"<<endl<<"\t";
        Shape::displayInfo(); // call the displayInfo method from the parent class
        cout<<"\tRadius: "<<m_radius<<endl;
        cout<<endl;
    }


};

class Rectangle : public Shape {
private:
    double m_width;
    double m_height;

public:
    Rectangle(double x, double y, double w, double h) : Shape{x, y} {
        m_width = w; m_height = h;
    }

    // redefine the displayInfo method
    void displayInfo(){
        cout<<"Rectangle:"<<endl<<"\t";
        Shape::displayInfo(); // call the displayInfo method from the parent class
        cout<<"\tSize: ("<<m_width<<", "<<m_height<<")"<<endl;
        cout<<endl;
    }
};


int main(){
    Circle c{10, 20, 4};
    Rectangle r{0, 0, 20, 30};
    c.displayInfo();
    r.displayInfo();

    cout<<"Translate circle with (20, -4)"<<endl;
    c.translate(20, -4);
    c.displayInfo();
    return 0;
}
```

To implement inheritance example in C++ the following syntax is used:

```
class <derived_class_name>: [inheritance_access_modifier] <base_class_name>{
};
```

, where <base_class_name> is the name of the parent class (Shape in our example), and <derived_class_name> is the name of the derived class (Circle) in our example.

The derived class has direct access to the public and protected members (attributes and functions) of the base class, but it does not have direct access to the private members.

In addition, in C++ there are three types of ways in which a child class can inherit from the parent class: private (the default inheritance type), protected, and public. This is specified by *[inheritance_access_modifier]*. These methods of inheritance can change the access specifiers of the inherited members, as illustrated in the table below:

Public inheritance		**Protected inheritance**		**Private inheritance**	
Access modifier in base class	**Access modifier in derived class**	**Access modifier in base class**	**Access modifier in derived class**	**Access modifier in base class**	**Access modifier in derived class**
public	public	public	protected	public	private
protected	protected	protected	protected	protected	private
private	inaccessible	private	inaccessible	private	inaccessible

Public inheritance is the most widely used inheritance type, and it is the easiest to understand. Basically, the inherited members with public access in the base class specifier remain public in the derived class, the inherited members with protected access in the base class specifier remain protected in the derived class, while the private member in the base class are inaccessible in the derived class.

Protected inheritance is basically never used in practice. The main idea of protected inheritance is that the public and the protected inherited member from the base class become protected in the derived class.

Finally, in private inheritance, all the public and protected members are inherited as private. Private inheritance is used when there isn't an "*IS A*" relationship between the base class and the derived class, but the derived class uses the implementation of the base class internally. In such cases, it is not a good practice to expose the public interface of the base class through objects of the derived class (as in the case of public inheritance).

Method overloading and method overriding

Method overloading and method overriding are two important concepts in object-oriented programming.

Method Overloading occurs when you have multiple methods with the same name but with different signatures (different parameters). An example of overloading occurs when you define several constructors for your class (the default constructor, the parameterized constructor, copy constructor): they all have the same name, but their number and type of parameters differ. When you call an overloaded method, the compiler decides which of these methods should be called based on the parameters you pass.

Method Overriding: only makes sense in the context of inheritance, and it refers to the case in which a subclass provides a specific implementation for a method that is already defined in its super-class. To override a method, you need to create a method with the exact same signature as in the superclass. It's a good practice to mark the overridden function from the derived class with the ***override*** specifier.

PROPOSED PROBLEMS

---

1. Implement a C++ class hierarchy for an animal classification system. The base class is the *Animal* class with the following attributes protected commonName (string), scientificName (string), and a method that displays information about the animal. Create a parameterized constructor which initializes these attributes to the passed parameters.
   Then create three subclasses: *Mammal*, *Bird*, and *Reptile*. Add specific attributes to each derived class: the *Mammal* class should have an attribute *isAquatic* (bool) and *gestationPeriod* (unsigned char); the *Bird* class should have an attribute *wingSpan* (unsigned int); and *the* Reptile class should have an attribute *isVenomous* (bool). For each of the subclasses, re-implement the method to display the information about the animal and also display the values for the new attributes.
   In the main function create an object from all the classes that you defined and call the display method on all of them.

2. Implement a simplified version of the chess game that checks the correctness of moves on the chessboard of different pieces, and makes the move only the move is valid. The base class shoule be *ChessPiece* with the following protected members: *positionX*, *positionY, name*. The class should also have a private attribute *color*. The class should have method *bool movePiece(unsinged int newX, unsigned int newY);* which does nothing in the function body and returns *false*. Create getters and setters for all the attributed.

Then create a class for each chess piece (*Bishop*, *Rook*, *Pawn*, *Queen, King*) that inherits the *ChessPieces* class and implements the *movePiece* method according to the move rules of each piece. If the move is not valid, than the function will return *false* and the position of the piece will not be changed. If the move is valid, the function will return *true* and the position of the piece will be updated.
In the main function, create objects from all the derived classes. Then create a menu-based console application that prompts the user for which piece to move and then for the coordinates of where to move the piece. The program will display if the attempt to move the piece was correct or not and the current position of the piece.

3. Write a class hierarchy for a system that manages two types of events: *Conferences* and *Concerts*. The base class is *Event* with attributes *eventName*, *eventDate*, *location*, and a method to display information about the event *void displayEventDetails()*. Derive classes *Conference* and *Concert* from the *Event* class. *Conference* should include another attribute *keynoteSpeaker* to store the name of the keynote speaker, while the *Workshop* class should have an additional attribute *workshopTopic*. For each derived re-implement the method *displayEventDetails* to display specific details about the event type.

4. A video game needs to handle two types of characters: *Wizards and* Knights. Create a simple class hierarchy in C++ for this scenario. The base class is the *Character* class that contains the common attributes: *name*, *health*, and *level*; create getters and setters for these attributes.
Then create two sub-classes of the Character class: Wizard and Knight..
The Wizard class should have the following attributes and behaviors:
- *mana*: an integer value representing the amount of magical power the wizard has.
- *spells*: a std::vector of spells that the wizard is able to cast. Each spell is represented by its name (an std::string).
- *spellPower*: an integer value representing the strength of the wizard's spells.
- *bool castSpell(std::string spell)*: a method that allows the wizard to cast a spell, using their spellPower and reducing their mana (if the wizard is able to cast the spell). Each spell requires 10 mana to cast. If the wizard does not know the requested spell, or he does not have enough mana, the function will return false and the mana will remain unchanged.

The Knight class should have the following attributes and behaviors:
- *armor*: a real value between [0, 1] value representing the level of protection the knight has.
- *swordDamage*: an integer value representing the strength of the knight's sword.
- *void takeDamage(int damage)*: a method that reduces the knight's armor by the specified amount. If the armor gets negative, the health of the knight is set to 0.

5. For all the classes that you wrote for the previous problem create your own tests. The test should be written in other classes that contain only ***static*** methods and that are friend classes of the tested class. All the test classes should have a static method *void runTests()*; which runs all the tests you wrote.
For example, for the Character class, the tester class should be *CharacterTest*.

```
#include<string>
using namespace std;

class Character{

   // TODO Character implementation
private:
   string m_name;
   // other attributes here

   // the tester class is a friend of the Character class
   friend class CharacterTest;
};

class CharacterTest{

   static void testSetName(){
       // TODO your code here
   }
   // TODO other test methods
   static void runAll(){
       // TODO call all the other methods
       testSetName();
   }

}
```