OBJECT-ORIENTED PROGRAMMING

LABORATORY 6

OBJECTIVES

In this laboratory session, you will apply one of the most important OOP concepts – inheritance and polymorphism by designing and implementing some basic class hierarchies.

THEORETICAL NOTIONS

Polymorphism in C++

Binding refers to the process in which identifiers, such as variables or method names, are converted into addresses. Static /early binding occurs when the compiler can directly determine the identifier with a machine address. On the other hand, in C++, late /dynamic binding occurs when the machine address is not known at compile time and is resolved at run-time. In the remainder of this laboratory, we will focus on dynamic polymorphism and late binding.

Polymorphism is one of the main features of object-oriented programming. Etymologically, "*polymorphism*" comes from the Greek words: "polys" (many) and "morphe" (shapes). In object-oriented programming, this feature refers to the ability to adapt the behavior of an object to the context; in other words, the function to be called is decided based on the actual type of the object. In C++, you can implement two types of polymorphism, depending on when the types are resolved:

- static or compile-time polymorphism, which can be achieved by method overloading or by templates (covered in the next laboratories);
- dynamic or run-time polymorphism, which can be achieved by inheritance and virtual functions.

Virtual methods

In the previous laboratory, we learned about the inheritance relationship ("IS A" relationship) between classes, and we saw that when we create a derived class it "comprises" two parts: one part for the base class and one part for the derived class. In other words, when we construct an object belonging to a derived class, first the base class constructor is called (to initialize the members that the object inherits from the base class) and then the derived class constructor is called (to initialize the members belonging to the derived class).

Because of this (since a derived class has a base class part), C++ allows you to set a base class reference or a pointer to a derived class object.

```
#include <iostream>
#include <exception>
```

```
using namespace std;


class Animal{
public:
  void speak(){ cout<<"???"<<endl;}
};

class Cat: public Animal{
public:
  void speak() { cout<<"Meow!"<<endl;}
};


class Dog: public Animal{
public:
  void speak(){ cout<<"Woof!"<<endl;}
};

int main(){
  Dog d{};
  Cat *c{new Cat{}};
  Animal* aCat{c};
  Animal aDog{d};

  aCat->speak();
  aDog.speak();

  delete c;
  return 0;
}
```

```
Output:
???
???
```

In the example above, we use *aCat* (base class pointer) and *aDog* (base class reference) to refer to the objects *c* and *d* belonging to the derived classes. However, if we run this code, we see that when calling the *speak()* method on these pointers (*aCat*) and references (*aDog*) the *speak()* function from the base class is called. And this might not be what you expected. Because *c* and *d* are actually *Cat* and *Dog* instances, one would expect that the *speak()* function from the *Cat* and *Dog* classes to be called.

To achieve this (i.e. dynamic polymorphism and late binding) in C++ you need to use virtual functions. A *virtual* function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.

To mark a method as virtual in C++, you just need to use the *virtual* keyword.

Let's modify the previous example to use virtual functions!

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


class Animal{
public:
  virtual void speak(){ cout<<"???"<<endl;} // the function is marked as virtual
};

class Cat: public Animal{
public:
  void speak() override{ cout<<"Meow!"<<endl;} // good practice: use the override specifier
};


class Dog: public Animal{
public:
  void speak() override{ cout<<"Woof!"<<endl;}
};

int main(){
  Dog d{};
  Cat *c{new Cat{}};
  Animal* aCat{c};
  Animal& aDog{d};

  aCat->speak();
  aDog.speak();

  delete c;
  return 0;
}
```

```
Output:
Meow!
Woof!
```

We just added the *virtual* keyword in front of the *speak()* function inside the base class. Also, it's a good practice to mark the overridden functions with the *override* specifier in the derived classes: `void speak() override{ cout<<"Meow!"<<endl;}`. Now, the code works just as you expected. The derived versions of the *speak()* method are being called.

Keep in mind that if a function is marked as virtual, all the matching function overrides in derived classes are implicitly virtual.

It is a good practice to mark the destructors of the base class as virtual. If you don't actually need to implement the destructor, you can use the default implementation provided by the compiler by using the *default* keyword:

```
virtual ~<class_name>() = default;
```

, where you should replace <class_name> with the name of your class.

In order to implement virtual functions and dynamic polymorphism, C++ uses a virtual table. The virtual table is just a lookup table that contains pointers to functions and it is used for dynamic binding (i.e. to resolve function calls).

Every class that has at least one virtual function is automatically assigned a virtual table. This table contains an entry for each virtual function of the class; an entry in the table is just a function pointer that points to the most-derived function accessible by that class. The compiler automatically adds a hidden member variable to the base class (*vptr*) which points to the virtual table of the class. Take note that this hidden member is also inherited by the derived classes.

tl;dr
In order for dynamic polymorphism to work, two conditions must be met:
- you need to work with ***pointers*** or ***references*** to the base class;
- the methods that are overridden in the derived classes must be marked as ***virtual*** in the base class.

Object slicing

We saw that in order to use dynamic polymorphism in C++ we need to use non-value types (i.e. pointers or references) to the base class. Let's see what happens if we simply assign a derived object to a base object. Because the derived object has a base part and a derived part, when we use simple assignment between a base class object and a derived class object, only the base part gets copied, while the derived part is "sliced off". This is called object slicing.

```
Dog d{};
Animal a{d};
  a.speak();
```

This code will call the *speak()* method from the animal class because this instruction Animal **a**{d}; performs object slicing. Therefore the variable *a* does not have a *Dog* part, only the *Animal* part, so the *speak()* method from the *Animal* class is called.

Abstract methods. Abstract classes. Interfaces.

An abstract method is a function declared within a class or an interface, but it does not have any definition (implementation). Abstract methods in C++ are the methods declared as pure virtual functions. An abstract class is a class that contains at least one abstract function.

To define a pure virtual method in C++ you need to use the following syntax in the class declaration (header):

```
class Shape {
public:
  virtual double area() = 0;
};
```

You just need to put the *virtual* function in front of the function declaration (i.e. virtual <return_type> <function_name>({<param_type> <param_name>}).

The method *area* is a pure virtual function (notice **= 0;** at the end of the method declaration). The *area* function does not have an implementation (a body).

Interfaces: An interface is a class with no members and only abstract methods. Unlike other programming languages such as Java or C#, C++ does not provide an explicit interface type. An interface in C++ is just a class with only pure virtual functions.

The *Shape* class from the example above is an interface.

Abstract classes: An abstract class is a class that has at least one abstract method. Abstract classes cannot be instantiated (i.e. you cannot create objects belonging to an abstract class), instead they can be used as a base class for other classes. Note that the derived classes need to provide implementations for all the abstract methods, otherwise they will be considered abstract classes as well.

Now let's transform the *Shape* interface into an abstract class:

```
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;
  }
  virtual ~Shape() = default;

  virtual double area() = 0;
  // this method is inherited by the derived class

 void translate(double dx, double dy) {
     m_x += dx;
     m_y += dy;
  }

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

```
  }
};
```

In the example above, *Shape* is an abstract class because it has a pure virtual function (the area function). However, it is no longer an interface because it also has a state (defined by the *m_x* and *m_y* attributes), and non-abstract methods.

Abstract classes cannot be instantiated (i.e. you cannot create objects belonging to abstract classes). However, you can define subclasses of abstract classes, implement all the pure virtual functions in the base class, and then instantiate the derived class.

If a class derives from an abstract class and does not implement all the pure virtual functions from the base class, then the derived class is also an abstract class.

Upcasting and downcasting

Generally speaking, casting is the process in which a data type is converted into another datatype. Upcasting and downcasting refer to casting operations on class hierarchies (in the context of inheritance).

Upcasing (i.e. "moving up" in the inheritance tree) refers to the process of converting a pointer (or a reference) of a derived class into a pointer (or a reference) of the base class. In C++ upcasting is implicit (because inheritance implements an "IS-A" relationship, therefore the derived class has all the attributes and behaviors of the base class).

Downcasting (i.e. "moving down" in the inheritance tree) refers to the process of converting a pointer (or a reference) of a base class into a pointer (or a reference) of the derived class. As opposed to upcasting, this operation must be performed explicitly (the base class may not have all the members of the derived class; you should explicitly perform this operation when you know that the pointer or reference of the base class is a pointer or reference to an object of the derived class).

To perform downcasting in C++, the best practice is to use *dynamic_cast* (https://en.cppreference.com/w/cpp/language/dynamic_cast). *dynamic_cast* "notifies" you if the downcasting operation was successful:

- if you are trying to convert a pointer of the base class to a pointer of the derived class and the operation fails, the function returns *nullptr*;
- If you are trying to convert a reference of the base class to a reference of the derived class and the operation fails, the function throws a std::bad_cast exception. Exceptions will be covered in the next laboratories.

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

class Shape {

protected:
  // coordinates of the center of the shape
  // protected - can be accessed by the derived classes
```

```
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
```

```
  double m_x;
  double m_y;
public:
  Shape(double x, double y) {
     m_x = x;
     m_y = y;
  }
  virtual ~Shape() = default;

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

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

```
  Circle(double x, double y, double r) : Shape{x, y} {
     m_radius = r;
  }

  double diameter() const{
     return 2*m_radius;
  }

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


};
```

```
int main(){

  Circle c1{0, 0, 5};
  Circle c2{-10, 10, 100};

  Shape *s{new Shape{10, 10}};
  Shape *s1 {&c1}; // upcasting (this is implicit). s1 is a base class pointer
  Shape &s2 {c2};  // upcasting (this is implicit). s2 is a base class reference

  // downcasting
  // this will fail because s does not point to a Circle
  if(!dynamic_cast<Circle*>(s)){
     cout<<"Failed to convert object s to a Circle"<<endl;
  }

  // the dynamic_casts below will work, because s1 and s2 actually refer to Circle(s)
  Circle* cp{dynamic_cast<Circle*>(s1)};
  if(cp != nullptr){
     cout<<"Successfully converted s1 to a Circle pointer"<<endl;
     cout<<"Diameter of s1 "<<cp->diameter()<<endl;
  }else{
     cout<<"Failed to convert s1 to a Circle pointer"<<endl;
  }

  try{
     Circle &cr{dynamic_cast<Circle&>(s2)};
     // exceptions will be covered in the next labs
     cout<<"Successfully converted s2 to a Circle reference"<<endl;
```

```
    cout<<"Diameter of s1 "<<cr.diameter()<<endl;
  }catch(std::bad_cast& e){
    cout<<"Failed to convert s2 to a Circle reference"<<e.what()<<endl;
  }

  delete s;
  return 0;

}
```

In the example below, you have a base class Shape and a derived class Circle. In the main function, the variable *s* is a pointer to a Shape object. However, the variables *s1* and *s2* are a pointer and a reference, respectively, to a Shape, but they point to circle objects. Therefore *s1* and *s2* can be downcasted to a Circle pointer and a Circle reference, respectively, but *s* cannot be downcasted to a Circle pointer. When you try to downcast *s,* the dynamic_cast method will return *nullptr*.

PROPOSED PROBLEMS

---

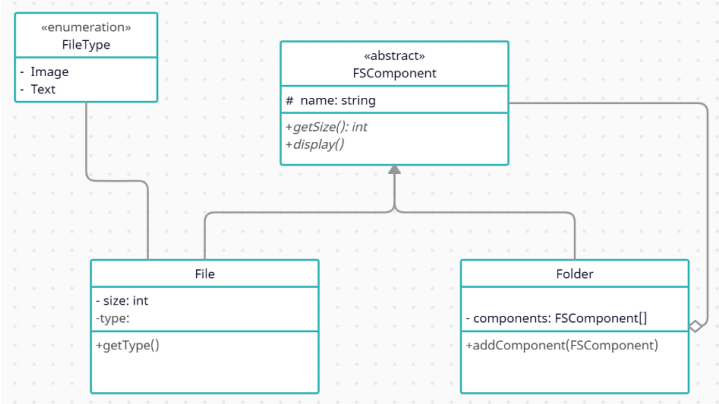
1. Design and implement a vehicle management system in C++ that simulates a garage for maintenance and display of different types of vehicles. Create an abstract base class *Vehicle*, which represents individual vehicle items within the system. This class should have a string attribute *registrationNumber* and a virtual function *void display()* to present the vehicle's specifications and status.

   The system supports two specific types of vehicles, implemented in two concrete classes: *Car* for personal vehicles and *Truck* for commercial heavy vehicles. Override *display*() to provide vehicle-specific information. For *Car*, add an additional attribute *bodyStyle* of type string to represent the car's body type (e.g., sedan, hatchback); implement getters and setters for this field. For the *Truck* class, add an attribute *payloadCapacity* of type double to represent the weight that the truck can carry; also implement getters and setters for this field.
   Implement a Garage class which maintains a collection of *Vehicle* objects and can accommodate both Car and Truck instances (i.e., it stores a polymorphic collection of pointers to *Vehicle* items). This class should implement the *display*() method to list all the vehicles' details in the garage. Moreover, an *addVehicle*() method should be implemented to allow the insertion of new *Vehicle* objects into the *Garage*.

2. Develop a simple file system (FS) simulation in C++ using the principles of object-oriented programming. The simulation must include an abstract class *FSComponent*, which represents components within the file system. The *FSComponent* has a single attribute *path* and two abstract functions: *unsigned int getSize() const;* to retrieve the size of the component and *void display();* to print the file's details.
   The file system should have two concrete classes, *TextFile* and *Folder*. The *TextFile* class should override the *getSize*() method to return its size.

The *Folder* class will act as a composite object that can contain any number of *FSComponent* objects, allowing folders to contain *TextFile* or other *Folder*s. The *Folder* class should implement the *getSize*() method, which returns the size of all its components, and a *display*() method to print details of all its contents. As attributes, the *Folder* class should have a polymorphic container of *FSComponent*s objects which stores its files and subfolders. Additionally, it should include an *addComponent*() method to add new *FSComponent* objects to the folder.

Implement this file system with the ability to add files and folders, display their contents, and report the size of individual files and the total size of folders.



3. Prey-Predator simulation

In the next two laboratories, you will implement a prey-predator simulation application. The simulation will take place over several cycles *C* in an environment represented as an *NxM* grid. Each cell in this grid can be occupied by the following: Fox (F), Gopher (G), Plant (P) or Empty (E).

When performing the simulation, you need to analyze the neighborhood of each cell; this neighborhood is defined as a 3 × 3 square centered in the current cell (in the right figure, the neighbors of the current cell, depicted in blue, are marked in red). If you are dealing with a cell that is located on the margin of the grid, you need to ignore the neighbor cells that are outside the grid area ($x < 0$ || $x >= M, y < 0, y > N$).

	NW	N	NE	
	W	C	E	
	SW	S	SE	

Survival rules

Empty	In the next iteration, an Empty cell will be replaced by: a) Gopher, if more than one neighboring Gopher; b) otherwise, Fox, if more than one neighboring Fox; d) otherwise, Plant, if at least one neighboring Plant; e) otherwise, Empty.
Plant	In the next iteration, the life form on a Plant cell will be transformed into: a) Empty if there are twice as many Gophers as Plant cells in the neighborhood; b) otherwise, Gopher if there are at least three Gophers in the neighborhood; c) otherwise, Plant.
Gopher	In the next iteration, the life form on a Gopher cell will be transformed into: a) Empty if the Gopher's current age is 4 (the gopher dies of old age); b) otherwise, Empty if there is no Plant in the neighborhood (the gopher starves to death); c) otherwise, Fox if in the neighborhood there are at least as many Foxes as Gophers; d) otherwise, Gopher(the gopher will survive).
Fox	In the next iteration, the life form on a Fox cell will be transformed into: a) Empty if the Fox has an age greater or equal to 5 (the fox dies of old age); b) otherwise, Empty, if there isn't any Gopher in the neighborhood (the fox starves to death); c) otherwise, Fox (the fox will live on).

1. Create an enum class called *EntityType* with the following values: EMPTY, FOX, GOPHER, PLANT.
2. Implement the abstract class Entity to represent a generic life form. The class should have at least:
    a. two **protected** variables *row* and *column* of type *int* that store the location on which the life form resides;
    b. **a pure virtual** method: *EntityType what() = 0;* that returns the type of the entity.
    c. **an abstract method** *std::string toString() const;* that returns a string corresponding to the entity type ("E" for empty, "F" for fox, "G" for gopher and "P" for plant).
    d. a function *void demographics(unsigned int population[], const SimulationGrid &g);* that takes as input an input-output array *population* of size 4 and returns the number of entities in the neighborhood (number of *Empty* - position 0, number of *Fox* - position 1, number of *Gopher* - position 2, number of *Plant* - position 3)

Helper code for the *Entity::demographics* method:

```
// set the population array to 0; we have 4 types of elements : Empty, Fox, Gopher, Plant
```

```
        std::fill(population, population + 4, 0);
        // offsets for the neighbors' coordinates
        int dx[]{0, 0, 1, 1, 1, -1, -1, -1};
        int dy[]{1, -1, -1, 0, 1,  -1, 0, 1};
// row : row+1, row-1
// col: col+0, col+0
        unsigned int numNeigh{ sizeof(dy) / sizeof(dy[0]) };

        for (unsigned int i{ 0 }; i < numNeigh; ++i) {
                int r{ row + dy[i] };
                int c{ col + dx[i] };
                // TODO check that r and c are within bounds
                EntityType et = g.grid[r][c]->what();
                // convert the enum class to int, you can use to_underlying
                population[to_underlying(et)]++;

        }
```

e. a pure virtual method ***Entity* next(const SimulationGrid &g);*** that takes an input a *const Grid&* object (see below) and returns the new Entity that will occupy that same cell in the next iteration. This function will be implemented in all the subclasses, as follows:
   i. compute the histogram of life forms in the 3x3 neighborhood of the class object;
   ii. applies the survival rules for the entity (see point 3);
   iii. generates and returns the new life form at the same location based on the life form.

3. Implement the *Plant* and *Empty* subclasses of the *Entity* classes. *Empty* and *Plant* subclasses should implement at least:
   a. all the abstract methods from the *Entity* class;
   b. overload for the stream insertion operator (it just inserts the result of the *toString()* method in the stream).

4. Implement another subclass of the ***Entity*** class called ***Animal***. The ***Animal*** class is an abstract class (i.e. it does not implement the abstract methods in the ***Entity*** class) that contains a protected member variable *age,* and getters and setters for this member.

5. Implement the ***Fox*** and ***Gopher*** classes as subclasses of the ***Animal*** class.
   The ***Fox*** and ***Gopher*** subclasses should implement at least:

   a. all the abstract methods from the ***Entity*** class;

b. overload for the stream insertion operator (it just inserts the result of the *toString()* method in the stream).

6. Implement the ***SimulationGrid*** class. This class should have at least the following members:
   a. a matrix of Entity objects: Entity* grid[MAX_ROWS][MAX_COLS];
   b. overload for the stream insertion operator.
   c. Implement the rule of three;
   d. Implement the stream insertion operator (to display the simulation grid);
   e. Implement the stream extraction operator (to read the simulation grid - from a file).

      Example

      3 3

      E P E

      F1 G1 F2

      P P P

   f. Implement a function *Entity* createEntity(string s);* that will create an entity based on the string value passed as a parameter:
      i. "P" - create and return a Plant object;
      ii. "E" - create and return an Empty object;
      iii. "F<*i*>" - create and return a Fox object with the age <*i*>. Where <*i*> is a digit*;*
      iv. "G<*i*>" - create and return a Gopher object with the age <*i*>. Where <*i*> is a digit.
   g. Implement three constructors:
      - The default constructor (the size of the grid should be 0, 0).
      - A parameterized constructor that takes as input two integers: the number of rows and the number of columns in the grid. The grid with this size will be randomly generated.
      - A parameterized constructor that takes as input the path to a file. The grid will be created based on the contents of the file. Use the function *Entity* createEntity(string s);*
   h. Implement any other getters, setters, and functions that you think are necessary (for any of the classes).

7. Implement the class ***Simulation*** that computes the evolutions of some input simulation grids, which are either randomly generated or read from input files. In each iteration, it prompts the user on how the starting grid should be generated and for *N ,* the number of cycles to simulate. The program will print the initial grid and final grid after *N* cycles.

0. prompt the user on how the initial grid should be generated
   - If the grid should be randomly generated, prompt the user for the size of the grid
   - If the grid should be read from a file, prompt the user for the filepath. The file will contain on the first row the height and width of the simulation grid, and then the elements of the grid

1. prompt the user for the number of simulation cycles *N*

```
// you may define two plains for odd and even simulation cycles
SimulationGrid even;      // the grid after an even number of cycles
SimulationGrid odd;       // the grid after an odd number of cycles
```

2. run the simulation for *N* cycles

3. Display the initial and final grids only.
No intermediate grids should appear in the standard output. (Of course, when debugging your program, you can print intermediate plains.)

8. Draw the class diagram for your solution. You can draw the diagram by hand.

9. Create test classes for all of the classes that you wrote (except the *EntityType* class). As in lab 5, each of the test classes contains only static methods.