# OOP ASG#3

Q11. Design a base class Animal with a virtual function speak(), and derive classes Dog and Cat that override this function. Demonstrate polymorphism by creating objects of derived classes and calling speak() through a base class pointer.

```
#include <iostream>

using namespace std;


class Animal {

public:

   virtual void speak() { cout << "Animal speaks" << endl; }

   virtual ~Animal() {}

};


class Dog : public Animal {

public:

   void speak() override { cout << "Dog barks" << endl; }

};


class Cat : public Animal {

public:

   void speak() override { cout << "Cat meows" << endl; }

};


int main() {

   Animal* a1 = new Dog();

   Animal* a2 = new Cat();


   a1->speak();  // Dog barks

   a2->speak();  // Cat meows
```

```
    delete a1;

    delete a2;

    return 0;

}
```

**Explanation:**

- **Polymorphism** enables the use of a base class pointer (`Animal*`) to access the derived class methods (`Dog` and `Cat`), ensuring flexibility and dynamic behavior at runtime.
- This approach allows us to extend the program easily by adding new animals (like `Bird`) without modifying existing code. The `speak()` function in the base class ensures all derived classes share the same interface.

Q12. Create an abstract class Shape with a pure virtual function draw(). Derive classes Circle and Rectangle that implement draw(). Explain how abstract classes enable extensibility in software design.

```
#include <iostream>

using namespace std;


class Shape {

public:

    virtual void draw() = 0;  // Pure virtual function

    virtual ~Shape() {}

};


class Circle : public Shape {

public:

    void draw() override { cout << "Drawing Circle" << endl; }

};


class Rectangle : public Shape {
```

```
public:
    void draw() override { cout << "Drawing Rectangle" << endl; }
};

int main() {
    Shape* s1 = new Circle();
    Shape* s2 = new Rectangle();

    s1->draw();  // Drawing Circle
    s2->draw();  // Drawing Rectangle

    delete s1;
    delete s2;
    return 0;
}
```

### Explanation:

- Abstract classes act as a blueprint with at least one pure virtual function (e.g., `draw()` in `Shape`).
- Derived classes (`Circle` and `Rectangle`) must implement the abstract function, which ensures uniformity across all shapes.
- **Extensibility:** Developers can add new shapes (like `Triangle`) without altering the `Shape` class or existing code. This reduces dependency and makes the design scalable.

Q13. Research and design a Movable interface with a pure virtual function move() and a Vehicle class hierarchy implementing this interface. Discuss how interfaces provide flexibility in OOP design.

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

class Movable {
```

```
public:
    virtual void move() = 0;  // Pure virtual function
    virtual ~Movable() {}
};

class Car : public Movable {
public:
    void move() override { cout << "Car moves on roads" << endl; }
};

class Boat : public Movable {
public:
    void move() override { cout << "Boat moves on water" << endl; }
};

int main() {
    Movable* v1 = new Car();
    Movable* v2 = new Boat();

    v1->move();  // Car moves on roads
    v2->move();  // Boat moves on water

    delete v1;
    delete v2;
    return 0;
}
```

**Explanation:**

- **Interfaces** are classes with only pure virtual functions. They enforce a contract for derived classes to implement specific behavior (e.g., `move()`).
- The `Movable` interface allows different types of vehicles (e.g., `Car`, `Boat`) to define their unique implementation of `move()` without inheriting from a common base class.
- **Flexibility in OOP Design:**
  - Interfaces promote **loose coupling**, meaning classes are less dependent on specific implementations.
  - You can create new types (like `Plane`) by implementing the `Movable` interface without changing the existing `Car` or `Boat` classes.
  - This design supports **multiple inheritance**, as a class can implement multiple interfaces without the restrictions of single inheritance in C++.

Q14. Design a function template findMax that accepts an array of any data type and returns the maximum element. Discuss the benefits of templates in reducing code duplication and enhancing code reusability.

```
#include <iostream>

using namespace std;


template <typename T>

T findMax(T arr[], int size) {

  T max = arr[0];

  for (int i = 1; i < size; i++) {

    if (arr[i] > max) max = arr[i];

  }

  return max;

}


int main() {

  int intArr[] = {1, 5, 3, 9, 2};

  cout << "Max int: " << findMax(intArr, 5) << endl;


  double doubleArr[] = {1.1, 3.5, 2.2, 4.8};

  cout << "Max double: " << findMax(doubleArr, 4) << endl;
```

```
    return 0;
}
```

**Explanation:**

- **Templates** allow writing generic functions that work with different data types without duplicating code.
- For example, `findMax` works with both integers and doubles (or even strings).
- **Benefits of Templates:**
    - **Reduced Code Duplication:** Instead of writing separate functions for each type (e.g., `int`, `double`), we use a single template.
    - **Code Reusability:** The same function can handle different data types, enhancing flexibility.
    - **Type Safety:** Templates ensure the function operates on consistent types during compilation, reducing runtime errors.

---

Q15. Implement a class template for a Stack data structure that can handle any data type. Test the stack with integer and string data types and discuss how class templates support generic programming.

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

template <typename T>
class Stack {
    vector<T> elements;

public:
    void push(T value) { elements.push_back(value); }
    void pop() { if (!elements.empty()) elements.pop_back(); }
    T top() { return elements.back(); }
    bool empty() { return elements.empty(); }
```

```
};
int main() {
  Stack<int> intStack;
  intStack.push(10);
  intStack.push(20);
  cout << "Top of intStack: " << intStack.top() << endl;  // 20

  Stack<string> stringStack;
  stringStack.push("Hello");
  stringStack.push("World");
  cout << "Top of stringStack: " << stringStack.top() << endl;  // World

  return 0;
}
```

### Explanation:

- A **class template** defines a blueprint for classes that work with any data type.
- The `Stack` template works for `int`, `string`, or any custom data type, ensuring code reuse.
- **Generic Programming with Class Templates:**
  - Instead of writing separate stack implementations for `int`, `string`, etc., a single template supports all types.
  - Templates ensure the stack is type-safe, meaning we can't mix data types in the same stack.
  - This design makes the code more modular, reusable, and easy to maintain.