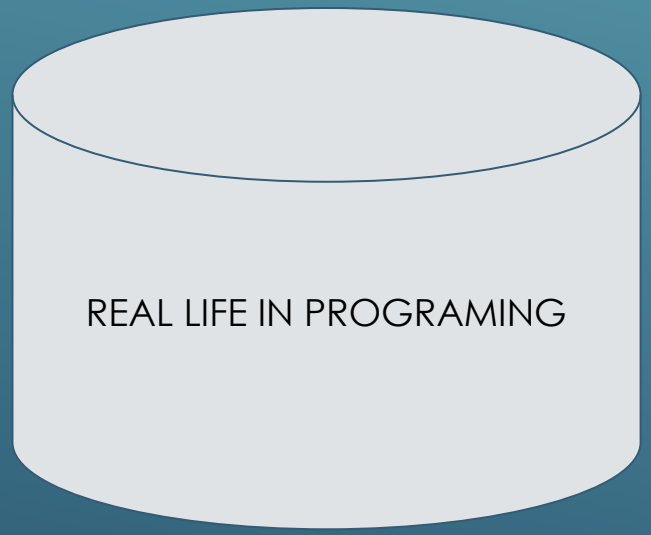
Programming 1 (C++)

Object Oriented Programming : Basic Principles

INTRODUCTION TO OOP

- C++ is a multi-paradigm language, and the paradigm that it is most strongly associated with is Object-Oriented Programming (OOP).
- The great benefit of object-oriented programming is the run-time polymorphism: which implementation of a method is called can be decided at run time.
- Represent real life object in Programming



ATTRIBUTE

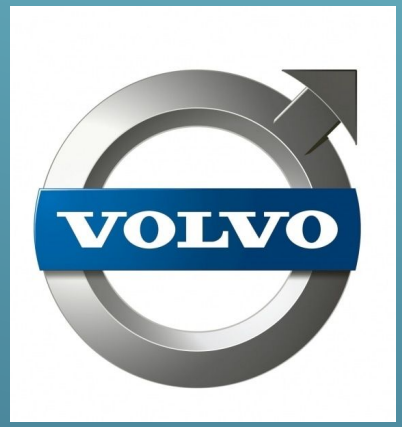
BEHAVIOUR

INTRODUCTION TO OOP

CLASSES
(REPRESENT A TEMPLATE / BLUE PRINT)

OBJECT
(REPRESENT AN EXAMPLE AND INSTANT OF THAT CLASS)

User defined Data Type



Attributes :

1. Name
2. Color
3. Price
4. Speed
5. etc

Behaviour :

1. Drive
2. Break
3. etc

BASIC PRINCIPLES

The basic principles of OOP relevant to C++ are :

- Abstraction: Classes (Chapter 2) define the attributes and methods of an object. The class can also specify invariants of attributes; e.g.,numerator and denominator shall be co-prime in a class for rational numbers. All methods must preserve these invariants.
- Encapsulation denotes the hiding of implementation details. Internal attributes cannot be accessed directly for not violating the invariants but only via the class's methods. In return, public data members are not internal attributes but part of the class interface.
- Inheritance means that derived classes contain all data and function members of their base class(es).

BASE AND DERIVED CLASSES

Base Class:

- A base class, also known as a parent class or superclass, is a class that serves as a foundation for other classes.
- It defines common attributes and behaviors that are shared by its derived classes.
- Base classes are often abstract, meaning they may contain pure virtual functions that must be implemented by derived classes.
- Base classes can be instantiated, but they are typically used as blueprints for creating derived classes.

BASE AND DERIVED CLASSES

Derived Class:

- A derived class, also known as a child class or subclass, is a class that inherits attributes and behaviors from a base class.
- It extends or modifies the functionality of the base class by adding new attributes or behaviors, or by overriding existing ones.
- Derived classes can access public and protected members of their base class.
- Multiple levels of inheritance can exist, where a derived class can itself serve as the base class for another class.

BASE AND DERIVED CLASSES

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

// Base class
class Shape {
public:
    void draw() {
        cout << "Drawing a shape" << endl;
    }
};

// Derived class 1
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle" << endl;
    }
};
```

BASE AND DERIVED CLASSES

```
// Derived class 2
class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw(); // Output: Drawing a circle
    shape2->draw(); // Output: Drawing a rectangle

    delete shape1;
    delete shape2;

    return 0;
}
```

INHERITING CONSTRUCTORS

- In C++, a derived class can inherit constructors from its base class, allowing it to use the same set of constructors without redefining them.
- This simplifies the code and reduces redundancy, especially when the base class has multiple constructors.

INHERITING CONSTRUCTORS

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

// Base class
class Shape {
public:
    Shape() {
        cout << "Shape constructor" << endl;
    }
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }
};
```

INHERITING CONSTRUCTORS

```
// Derived class 1
class Circle : public Shape {
public:
    using Shape::Shape; // Inherit constructors from Shape
    void draw() override {
        cout << "Drawing a circle" << endl;
    }
};

// Derived class 2
class Rectangle : public Shape {
public:
    using Shape::Shape; // Inherit constructors from Shape
    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }
};
```

INHERITING CONSTRUCTORS

```
int main() {
    Circle circle;
    Rectangle rectangle;

    circle.draw();    // Output: Drawing a circle
    rectangle.draw(); // Output: Drawing a rectangle

    return 0;
}
```

VIRTUAL FUNCTION AND POLYMORPHIC CLASSES

- In C++, a derived class can inherit constructors from its base class, allowing it to use the same set of constructors without redefining them.
- This simplifies the code and reduces redundancy, especially when the base class has multiple constructors.

VIRTUAL FUNCTION AND POLYMORPHIC CLASSES

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

// Base class
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }
};

// Derived class 1
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle" << endl;
    }
};
```

VIRTUAL FUNCTION AND POLYMORPHIC CLASSES

```
// Derived class 2
class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw(); // Output: Drawing a circle
    shape2->draw(); // Output: Drawing a rectangle

    delete shape1;
    delete shape2;

    return 0;
}
```

LATE BINDING AND DYNAMIC POLYMORPHISM

Late Binding and Dynamic Polymorphism:

- Late binding refers to the process of determining the appropriate function to call at runtime, rather than at compile-time.
- Dynamic polymorphism, facilitated by late binding, allows objects of different types to be treated uniformly through pointers to a common base class.
- Late binding enables dynamic polymorphism by deferring the decision about which function to call until runtime based on the actual object type.

LATE BINDING AND DYNAMIC POLYMORPHISM

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

// Base class
class Animal {
public:
    virtual void speak() {
        cout << "Animal speaks" << endl;
    }
};

// Derived class 1
class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog barks" << endl;
    }
};
```

In this example, we have a base class Animal with a virtual function speak().

We have two derived classes, Dog and Cat, which override the speak() function to provide specific behaviors.

LATE BINDING AND DYNAMIC POLYMORPHISM

```
// Derived class 2
class Cat : public Animal {
public:
    void speak() override {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal* animal1 = new Dog();
    Animal* animal2 = new Cat();

    animal1->speak(); // Output: Dog barks
    animal2->speak(); // Output: Cat meows

    delete animal1;
    delete animal2;

    return 0;
}
```

- In the main() function, objects of type Dog and Cat are created but stored in pointers of type Animal*.
- When calling the speak() function on these objects through pointers to the base class Animal, dynamic polymorphism occurs.
- The correct version of speak() (either Dog::speak() or Cat::speak()) is determined at runtime based on the actual object type, demonstrating late binding and dynamic polymorphism

DESTROYING POLYMORPHIC CLASSES

Destroying Polymorphic Classes:

- When dealing with polymorphic classes (classes with virtual functions), it's essential to ensure proper cleanup to prevent memory leaks.
- To achieve this, the destructor of the base class should be declared as virtual. This ensures that when an object of a derived class is deleted through a pointer to the base class, the destructor of the derived class is invoked.
- This ensures that all resources allocated by the derived class are properly deallocated, even if the object is deleted through a pointer to the base class.
-

DESTROYING POLYMORPHIC CLASSES

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

// Base class
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }

    virtual ~Shape() {
        cout << "Shape destructor" << endl;
    }
};
```

In this example, both the base class Shape and the derived classes (Circle and Rectangle) have virtual destructors (~Shape() and ~Circle()/~Rectangle()).

DESTROYING POLYMORPHIC CLASSES

```
// Derived class 1
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle" << endl;
    }

    ~Circle() override {
        cout << "Circle destructor" << endl;
    }
};

/
```

When an object of a derived class is deleted through a pointer to the base class (Shape*), the virtual destructor of the derived class is invoked.

This ensures that the appropriate destructor for the derived class is called, allowing for proper cleanup of resources allocated by the derived class.

DESTROYING POLYMORPHIC CLASSES

```
// Derived class 2
class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }

    ~Rectangle() override {
        cout << "Rectangle destructor" << endl;
    }
};
```

The output confirms that the destructors for Circle and Rectangle are invoked when their respective objects are deleted, demonstrating proper destruction of polymorphic objects in C++.

DESTROYING POLYMORPHIC CLASSES

```
int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw(); // Output: Drawing a circle
    shape2->draw(); // Output: Drawing a rectangle

    delete shape1; // Output: Circle destructor
    delete shape2; // Output: Rectangle destructor

    return 0;
}
```

FUNCTION CALL MECHANISMS

Function Call Mechanisms:

- In C++, function calls can be resolved using two primary mechanisms: static binding (or early binding) and dynamic binding (or late binding).
- Static binding refers to the process of determining which function to call at compile-time based on the static (or declared) type of the object or pointer.
- Dynamic binding, on the other hand, defers the decision about which function to call until runtime based on the dynamic (or actual) type of the object.

FUNCTION CALL MECHANISMS

Function Call Mechanisms:

- In C++, function calls can be resolved using two primary mechanisms: static binding (or early binding) and dynamic binding (or late binding).
- Static binding refers to the process of determining which function to call at compile-time based on the static (or declared) type of the object or pointer.
- Dynamic binding, on the other hand, defers the decision about which function to call until runtime based on the dynamic (or actual) type of the object.

FUNCTION CALL MECHANISMS

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

// Base class
class Animal {
public:
    void speak() {
        cout << "Animal speaks" << endl;
    }
};

// Derived class 1
class Dog : public Animal {
public:
    void speak() {
        cout << "Dog barks" << endl;
    }
};
```

FUNCTION CALL MECHANISMS

```
/ Derived class 2
class Cat : public Animal {
public:
    void speak() {
        cout << "Cat meows" << endl;
    }
};

int main() {
    Animal animal;
    Dog dog;
    Cat cat;

    animal.speak(); // Output: Animal speaks (Static binding)
    dog.speak();    // Output: Dog barks (Static binding)
    cat.speak();    // Output: Cat meows (Static binding)

    Animal* ptr = &dog;
    ptr->speak();   // Output: Animal speaks (Dynamic binding)

    return 0;
}
```

FUNCTORS VIA INHERITANCE

Functors via Inheritance

- In C++, functors (function objects) can be implemented using structs that define an overloaded function call operator operator().
- Inheritance with structs can also be used to create functors with different behaviors by deriving from a base functor struct and overriding the operator() function.
- This approach provides a concise and straightforward way to define functors with varying behaviors while promoting code reuse.

FUNCTORS VIA INHERITANCE

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

// Base functor struct
struct Functor {
    virtual int operator()(int x) const = 0; // Pure virtual function
};

// Functor for adding a constant value
struct AddFunctor : public Functor {
    int valueToAdd;

    AddFunctor(int value) : valueToAdd(value) {}

    // Override the function call operator
    int operator()(int x) const override {
        return x + valueToAdd;
    }
};

// Functor for multiplying by a constant value
struct MultiplyFunctor : public Functor {
    int valueToMultiply;
```

FUNCTORS VIA INHERITANCE

```
    MultiplyFunctor(int value) : valueToMultiply(value) {}

    // Override the function call operator
    int operator()(int x) const override {
        return x * valueToMultiply;
    }
};

int main() {
    // Create functors
    AddFunctor add5(5);
    MultiplyFunctor multiply3(3);

    // Use functors
    cout << "Result of add5(10): " << add5(10) << endl; // Output: 15
    cout << "Result of multiply3(10): " << multiply3(10) << endl; // Output: 30

    return 0;
}
```

DERIVED EXCEPTION CLASS

Derived Exception Class:

- In C++, exception handling allows for the creation of custom exception classes to represent specific error conditions.
- Derived exception classes can be created by inheriting from standard exception classes like std::exception.
- This allows for the creation of more specialized exception types that provide additional context or information about the error condition.

DERIVED EXCEPTION CLASS

```
#include <iostream>
#include <stdexcept> // For std::exception
using namespace std;

// Derived exception class for division by zero
class DivisionByZeroException : public std::exception {
public:
    // Override what() to provide custom exception message
    const char* what() const noexcept override {
        return "Division by zero exception";
    }
};

// Function to perform division operation
double divide(double dividend, double divisor) {
    if (divisor == 0) {
        throw DivisionByZeroException(); // Throw custom exception
    }
    return dividend / divisor;
}
```

DERIVED EXCEPTION CLASS

```
int main() {
    double dividend = 10.0;
    double divisor = 0.0;

    try {
        // Attempt division operation
        double result = divide(dividend, divisor);
        cout << "Result of division: " << result << endl;
    } catch (const DivisionByZeroException& e) {
        // Handle division by zero exception
        cout << "Exception caught: " << e.what() << endl;
    }

    return 0;
}
```

THANK YOU