Polymorphism in C++

By:
freeCSIT

C++ course with PDF Notes

OOPs in C++

Classes and objects are the two main aspects of object-oriented programming.

Features of OOP

1. Classes
2. Object
3. Inheritance
4. Encapsulation
5. Polymorphism

OOPs in C++

Classes and objects are the two main aspects of object-oriented programming.

Features of OOP

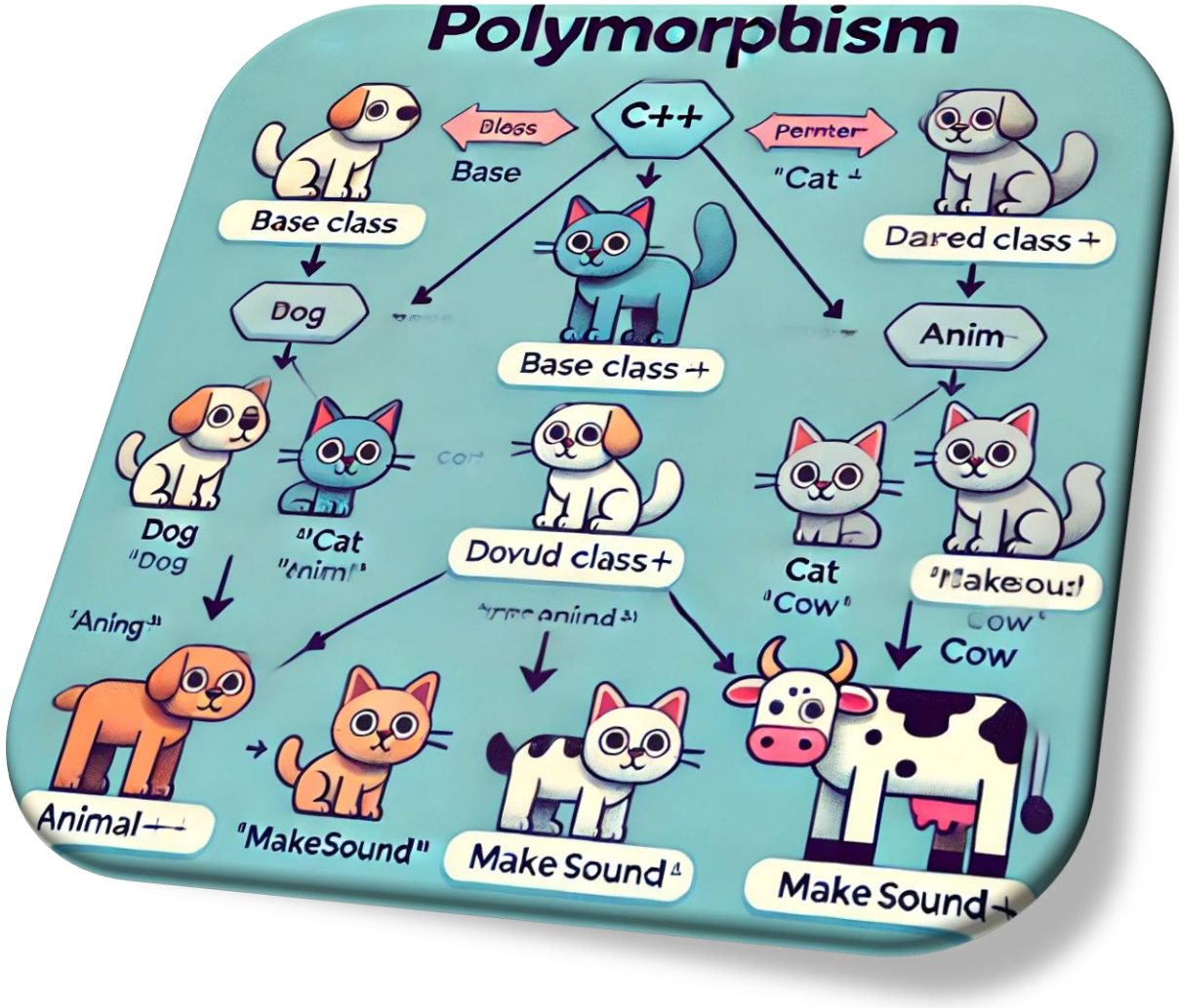
1. Classes
2. Object
3. Inheritance
4. Encapsulation
5. Polymorphism

OOPs in C++

Classes and objects are the two main aspects of object-oriented programming.

Features of OOP

5 **Polymorphism**



OOPs in C++

Classes and objects are the two main aspects of object-oriented programming.

Features of OOP

5 Polymorphism

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) in C++. It allows objects of different classes to be treated as objects of a common base class. Polymorphism enables a single interface to be used for different types, improving code reusability and flexibility.

OOPs in C++

Classes and objects are the two main aspects of object-oriented programming.

Features of OOP

5 Polymorphism

Types:

1. Compile-time Polymorphism (Static Binding)
   1. Function Overloading
   2. Operator Overloading
2. Run-time Polymorphism(Dynamic Binding)
   1. Function Overriding(Using Virtual Functions)

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

1 Compile-time Polymorphism (Static Binding)

Compile-time polymorphism is achieved through function overloading and operator overloading. The function to be executed is determined at **compile-time**.

1 Function Overloading

Function overloading allows multiple functions to have the **same name** but with different parameters (different type or number of arguments).

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

1 Function Overloading - Example

Function overloading allows multiple functions to have the **same name** but with different parameters (different type or number of arguments).

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

class MathOperations {
public:
  // Function to add two integers
  int add(int a, int b) {
    return a + b;
  }

  // Function to add three integers
  int add(int a, int b, int c) {
    return a + b + c;
  }

  // Function to add two double numbers
  double add(double a, double b) {
    return a + b;
  }
};

int main() {
  MathOperations obj;

  cout << "Sum of 2 and 3: " << obj.add(2, 3) << endl;
  cout << "Sum of 2, 3 and 4: " << obj.add(2, 3, 4) << endl;
  cout << "Sum of 2.5 and 3.5: " << obj.add(2.5, 3.5) << endl;

  return 0;
}
```

Output:

```
Sum of 2 and 3:  5
Sum of 2, 3 and 4:  9
Sum of 2.5 and 3.5:  6
```

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

1 Compile-time Polymorphism (Static Binding)

Compile-time polymorphism is achieved through function overloading and operator overloading. The function to be executed is determined at **compile-time**.

2 Operator Overloading

Operator overloading allows operators like +, -, *, etc., to work on user-defined data types.

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

2 Operator Overloading - Example

Operator overloading allows operators like +, -, *, etc., to work on user-defined data types.

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

class Complex {
private:
   double real, imag;

public:
   Complex(double r = 0, double i = 0) : real(r), imag(i) {}

   // Overloading the '+' operator
   Complex operator+(const Complex &obj) {
      return Complex(real + obj.real, imag + obj.imag);
   }

   // Display function
   void display() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(3.2, 4.5), c2(1.5, 2.5);
   Complex c3 = c1 + c2;  // Uses overloaded '+' operator

   cout << "Result of addition: ";
   c3.display();

   return 0;
}
```

Output:
Result of addition: 4.7 + 7i

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

2 Run-time Polymorphism (Dynamic Binding)

Run-time polymorphism is achieved using **function overriding** and **virtual functions**. The function to be executed is determined at **run-time**, based on the object type.

1 Function Overriding

Function overriding occurs when a derived class provides a **specific implementation** of a function that is already defined in its base class.

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

1 Function Overriding - Example

Function overriding occurs when a derived class provides a **specific implementation** of a function that is already defined in its base class.

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

class Animal {
public:
  virtual void makeSound() {  // Virtual function
    cout << "Animal makes a sound!" << endl;
  }
};

class Dog : public Animal {
public:
  void makeSound() override {  // Overriding base class function
    cout << "Dog barks!" << endl;
  }
};

class Cat : public Animal {
public:
  void makeSound() override {  // Overriding base class function
    cout << "Cat meows!" << endl;
  }
};

int main() {
  Animal *animal1 = new Dog();  // Base class pointer to derived class object
  Animal *animal2 = new Cat();

  animal1->makeSound();  // Calls Dog's makeSound() due to dynamic binding
  animal2->makeSound();  // Calls Cat's makeSound() due to dynamic binding

  delete animal1;
  delete animal2;

  return 0;
}
```

Output:
Dog barks!
Cat meows!

Polymorphism

Classes and objects are the two main aspects of object-oriented programming.

1 Function Overriding - Example

Function overriding occurs when a derived class provides a **specific implementation** of a function that is already defined in its base class.

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

class Animal {
public:
  virtual void makeSound() {  // Virtual function
    cout << "Animal makes a sound!" << endl;
  }
};

class Dog : public Animal {
public:
  void makeSound() override {  // Overriding base class function
    cout << "Dog barks!" << endl;
  }
};

class Cat : public Animal {
public:
  void makeSound() override {  // Overriding base class function
    cout << "Cat meows!" << endl;
  }
};

int main() {
  Animal *animal1 = new Dog();  // Base class pointer to derived class object
  Animal *animal2 = new Cat();

  animal1->makeSound();  // Calls Dog's makeSound() due to dynamic binding
  animal2->makeSound();  // Calls Cat's makeSound() due to dynamic binding

  delete animal1;
  delete animal2;

  return 0;
}
```

Output:
Dog barks!
Cat meows!

Thanks for watching

Please Subscribe