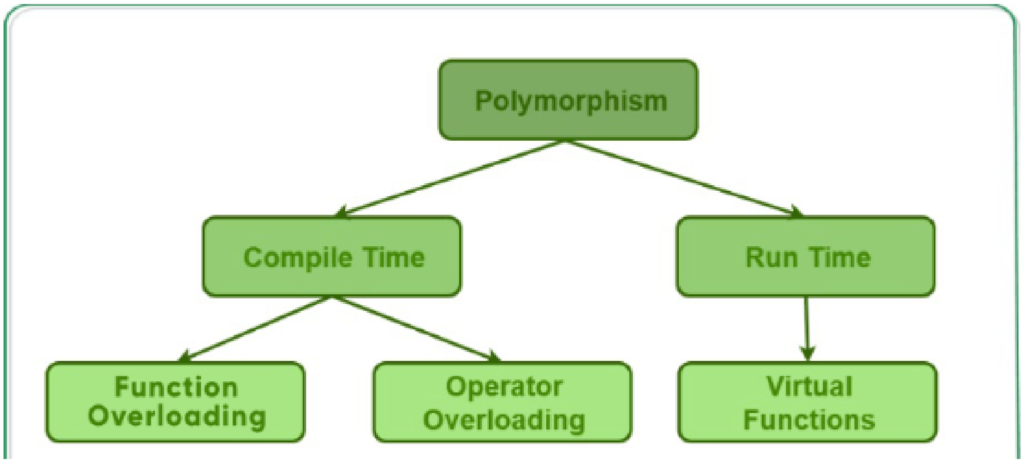
Polymorphism

Introduction

- Polymorphism is one of the pillars of OOPs. It means to provide the ability of a functionality to be displayed in more than one form.
- Types:
    - Compile Time Polymorphism
    - Runtime Polymorphism



Compile Time Polymorphism

- This is a type of polymorphism that is resolved during the compilation process.
- Also known as static polymorphism.
- This can be achieved using:
    - Function Overloading
    - Operator Overloading
    - Templates
        - Function Template
        - Class Template

A. Function Overloading

- Function Overloading allows multiple functions with the same name to be defined as long as they have different parameter lists (either in number / type of parameter)
- **Syntax:**

```
void display(double d) {
        std::cout << "Displaying double: " << d << std::endl;
}
void display(const std::string& str) {
        std::cout << "Displaying string: " << str << std::endl;
}
```

B. Operator Overloading

- Operator overloading allows to redefine the way operators work for user defined types (classes).
- **Syntax:**

```
// Overload the + operator
Complex operator+(const Complex& other) const
{
        return Complex(real + other.real, imag + other.image);
}
```

C. Templates

- Templates allow functions and classes to operate with generic types. This enables code reusability and type safety.
- **Syntax (Function Template):**

  ```
  T add(T a, T b)
  {
          return a + b;
  }
  ```
- **Syntax (Class Template):**

  ```
  template <typename T>
  class Calculator
  {
    public:
          T add(T a, T b)
          {
                  return a + b;
          }
          T subtract(T a, T b)
          {
                  return a - b;
          }
  };
  ```

- Compile-time polymorphism is powerful in C++ as it provides flexibility and enhances code reusability while ensuring type safety.

Runtime Polymorphism

- Run-time polymorphism in C++ is a type of polymorphism that is resolved during the program's execution.
- It is also known as **dynamic** polymorphism.
- The most common way to achieve run-time polymorphism in C++ is through **inheritance** and **virtual functions**. This allows a base class pointer or reference to call derived class methods, enabling polymorphic behavior.

Ways To Achieve:

1. **Inheritance And Virtual Functions**

    - To achieve run-time polymorphism, you define a virtual function in the base class, and then override this function in derived classes.
    - A pointer or reference to the base class can then be used to call the overridden function in the derived class.

Syntax:

```
class Animal
{
   public:
        virtual void makeSound() const
        {
              // Virtual function
               std::cout << "Some generic animal sound" <<
std::endl;
        }
};

class Dog : public Animal
{
   public:
       void makeSound() const override
       {
              // Override in derived class
             std::cout << "Bark" << std::endl;
        }
};

class Cat : public Animal
{
      public:
           void makeSound() const override
           {
                 // Override in derived class
                std::cout << "Meow" << std::endl;
            }
```

```
int main()
{
        Animal* animal2 = new Dog();
        Animal* animal2 = new Cat();
        animal1->makeSound();
              // Calls Dog's makeSound
        animal2->makeSound();
             // Calls Cat's makeSound
        delete animal1;
        delete animal2;
        return 0;
}
```

};	

What is Virtual Function?

- A virtual function is a member function in a base class that you expect to be overridden in derived classes.
- When a base class pointer or reference calls a virtual function, C++ determines at run time which version of the function to invoke based on the actual type of the object pointed to.
- **Syntax:**

```
class Base {
    public:
        virtual void someFunction() {
                // Base class implementation
        }
};
```

What is the Override Keyword ?

- The `override` keyword is used in a derived class to explicitly indicate that a function is meant to override a virtual function in the base class.
- This helps prevent errors if the base class function signature changes or if the derived class function doesn't match exactly.
- **Syntax:**

```
class Derived : public Base {
        public:
            void someFunction() override {
                    // Derived class implementation
            }
 };
```

Abstract Classes and Interfaces:

- Abstract classes and interfaces are used in many languages (like Java, C#, etc.) to define methods that must be implemented by any derived class.
- These methods provide a mechanism to enforce runtime polymorphism, as different classes can provide different implementations of the same method signature.

Applications

- **Design Patterns**: Many design patterns like Strategy, State, and Command heavily rely on runtime polymorphism.
- **Frameworks**: Many frameworks use runtime polymorphism to allow developers to extend and customize their behavior.