# Unit III - Polymorphism

Presented by,
Prof. Rupali Shende

# Introduction

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.
- In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.
- It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios.

# Real-life example

- A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

# For example

- The + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.
- int a = 5; int b = 6;
- int sum = a + b;
- // sum = 11
- And when we use the + operator with strings, it performs string concatenation.
- For example,
- string firstName = "abc ";
- string lastName = "xyz";
- // name = "abc xyz"
- string name = firstName + lastName;

# Inheritance v/s Polymorphism

- **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.)

# Inheritance v/s Polymorphism Example

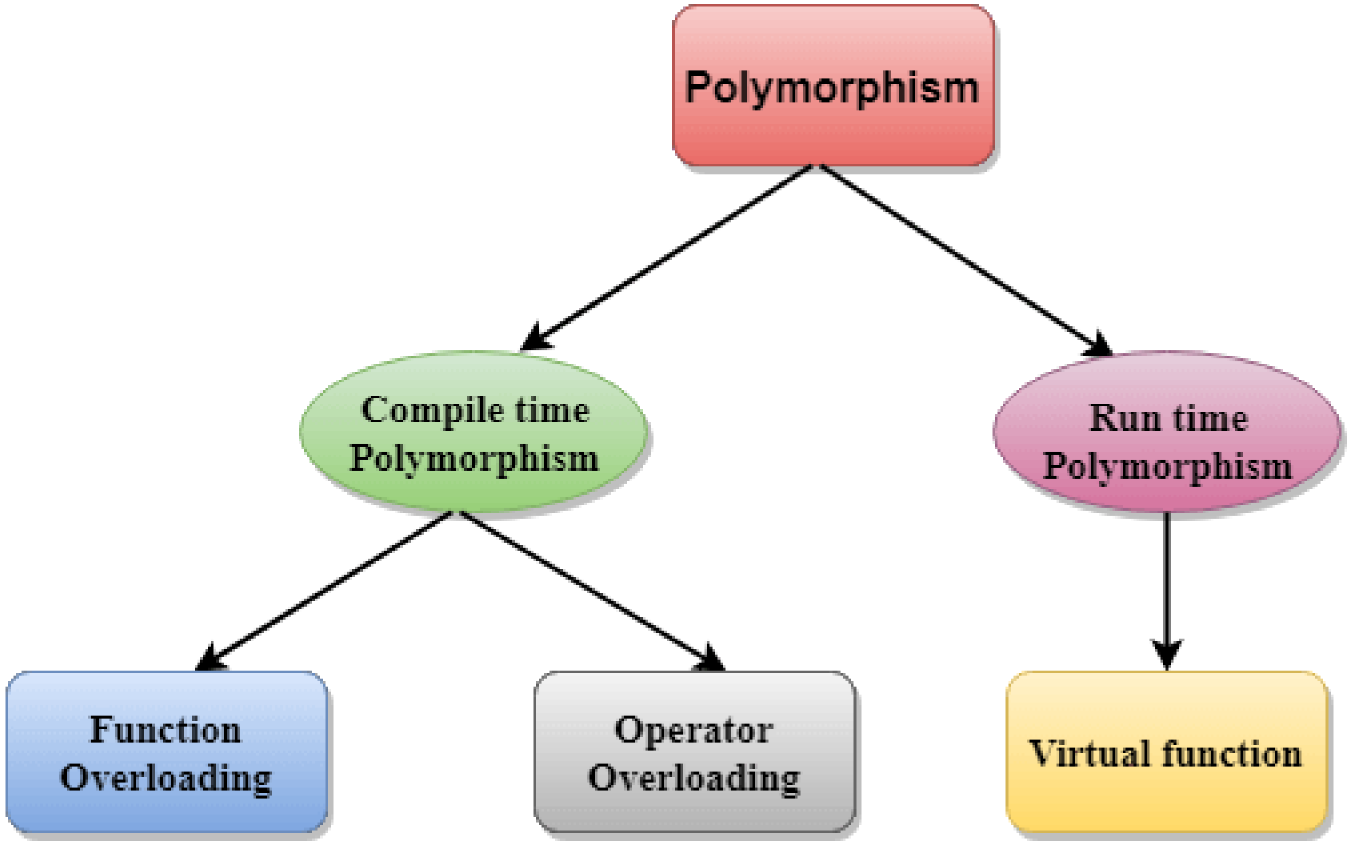
```
// Base class
class Animal {
  public:
    void animalSound() {
    cout << "The animal makes a sound \n" ;
  }
};
// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
    cout << "The pig says: wee wee \n" ;
  }
};
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
    cout << "The dog says: bow wow \n" ;
  }
};
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

# Why Polymorphism?

- Polymorphism allows us to create consistent code. For example,
- Suppose we need to calculate the area of a circle and a square. To do so, we can create a Shape class and derive two classes Circle and Square from it.
- In this case, it makes sense to create a function having the same name calculateArea() in both the derived classes rather than creating functions with different names, thus making our code more consistent.

# Types of polymorphism in C++

- Compile time Polymorphism
- Runtime Polymorphism



# Compile Time Polymorphism

- You invoke the overloaded functions by matching the number and type of arguments. The information is present during compile-time. This means the C++ compiler will select the right function at compile time.
- Compile-time polymorphism is achieved through function overloading and operator overloading.
- Compile-time polymorphism is also known as early binding and Static polymorphism

# Runtime Polymorphism

- This happens when an object's method is invoked/called during runtime rather than during compile time.
- Runtime polymorphism is achieved through function overriding. The function to be called/invoked is established during runtime.
- Run-time polymorphism is also known as late binding and Dynamic polymorphism.

# Concept of Overloading

- Creating two or more members that have the same name but are different in number or type of parameter is known as **overloading**.

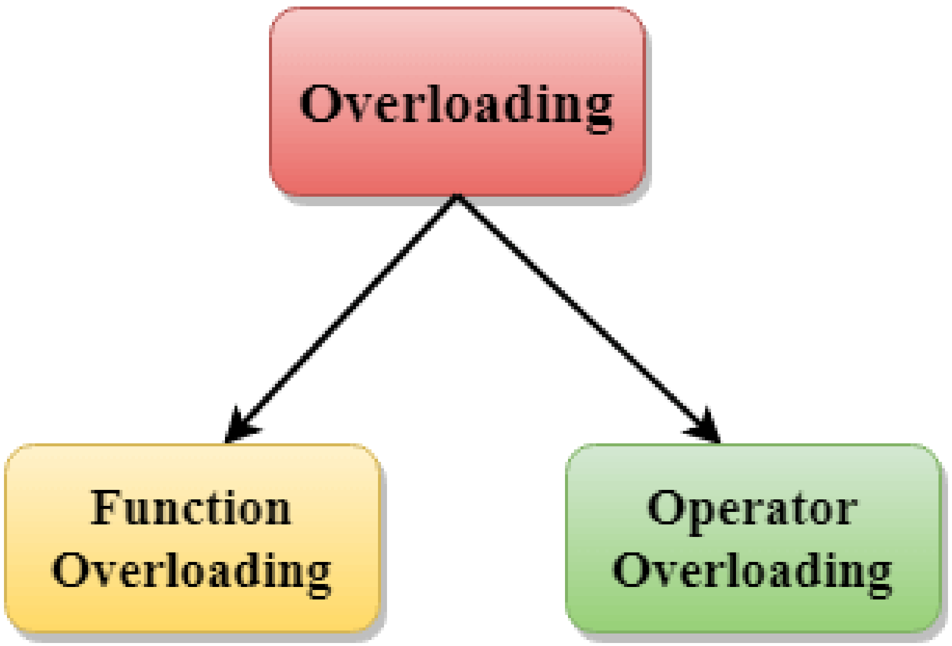
- An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

- In C++, we can overload:

✓ Methods

✓ Constructors

✓ Indexed Properties



# C++ Function Overloading

- Function overloading occurs when we have many functions with similar names but different arguments. The arguments may differ in terms of number or type.
- In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).
- And, depending upon the number/type of arguments, different functions are called.

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.
- In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.
- The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

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

// Function with 2 int parameters
int sum(int num1, int num2) {
   return num1 + num2;
}

// Function with 2 double parameters
double sum(double num1, double num2) {
   return num1 + num2;
}

// Function with 3 int parameters
int sum(int num1, int num2, int num3) {
   return num1 + num2 + num3;
}
```

```
int main() {
   // Call function with 2 int parameters
   cout << "Sum 1 = " << sum(5, 6) << endl;

   // Call function with 2 double parameters
   cout << "Sum 2 = " << sum(5.5, 6.6) <<
endl;

   // Call function with 3 int parameters
   cout << "Sum 3 = " << sum(5, 6, 7) << endl;

   return 0;
}
```

**Output**

```
Sum 1 = 11
Sum 2 = 12.1
 Sum 3 = 18
```

# C++ Operator Overloading

- Another example of static polymorphism is Operator overloading. Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.
- We cannot use operator overloading for basic types such as int, double, etc.
- Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands.
- And, depending on the operands, different operator functions are executed.

# EXAMPLE

```
class Count {
  private:
   int value;

  public:

   // Constructor to initialize count to 5
   Count() : value(5) {}

   // Overload ++ when used as prefix
   void operator ++() {
       value = value + 1;
   }

   void display() {
       cout << "Count: " << value << endl;
   }
};
```

```
int main() {
    Count count1;

    // Call the "void operator ++()" function
    ++count1;

    count1.display();
    return 0;
}
```

**Output**

Count: 6

Here, we have overloaded the ++ operator, which operates on objects of Count class (object count1 in this case).

We have used this overloaded operator to directly increment the value variable of count1 object by 1.

# Unary Operators Overloading

- The unary operators operate on a single operand and following are the examples of Unary operators −
- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.
- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

# Example - minus (-) operator overloaded for prefix & postfix usage

```
class Distance {
  private:
    int feet;          // 0 to infinite
    int inches;        // 0 to 12

  public:
    // required constructors
    Distance() {
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i) {
      feet = f;
      inches = i;
    }

    // method to display distance
    void displayDistance() {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
// overloaded minus (-) operator
    Distance operator- () {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                    // apply negation
  D1.displayDistance();   // display D1

  -D2;                    // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

Output:

```
F: -11 I:-10
F: 5 I:-11
```

# Binary Operators Overloading

- The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.
- Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

# Example

```
class Box {
  double length;     // Length of a box
  double breadth;    // Breadth of a box
  double height;     // Height of a box

  public:
   double getVolume(void) {
     return length * breadth * height; }
  void setLength( double len ) {
     length = len; }
   void setBreadth( double bre ) {
     breadth = bre;
  }
   void setHeight( double hei ) {
     height = hei;
  }
     // Overload + operator to add two Box objects.
  Box operator+(const Box& b) {
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
  }
};
```

```
// Main function for the program
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   Box Box3;              // Declare Box3 of type Box
   double volume = 0.0;    // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
```

# Compile-Time Vs. Run-Time Polymorphism

| Compile-time polymorphism | Run-time polymorphism |
| --- | --- |
| It's also called early binding or static polymorphism | It's also called late/dynamic binding or dynamic polymorphism |
| The method is called/invoked during compile time | The method is called/invoked during run time |
| Implemented via function overloading and operator overloading | Implemented via method overriding and virtual functions |
| Example, method overloading. Many methods may have similar names but different number or types of arguments | Example, method overriding. Many methods may have a similar name and the same prototype. |
| Faster execution since the methods discovery is done during compile time | Slower execution since method discoverer is done during runtime. |
| Less flexibility for problem-solving is provided since everything is known during compile time. | Much flexibility is provided for solving complex problems since methods are discovered during runtime. |

Thank You!