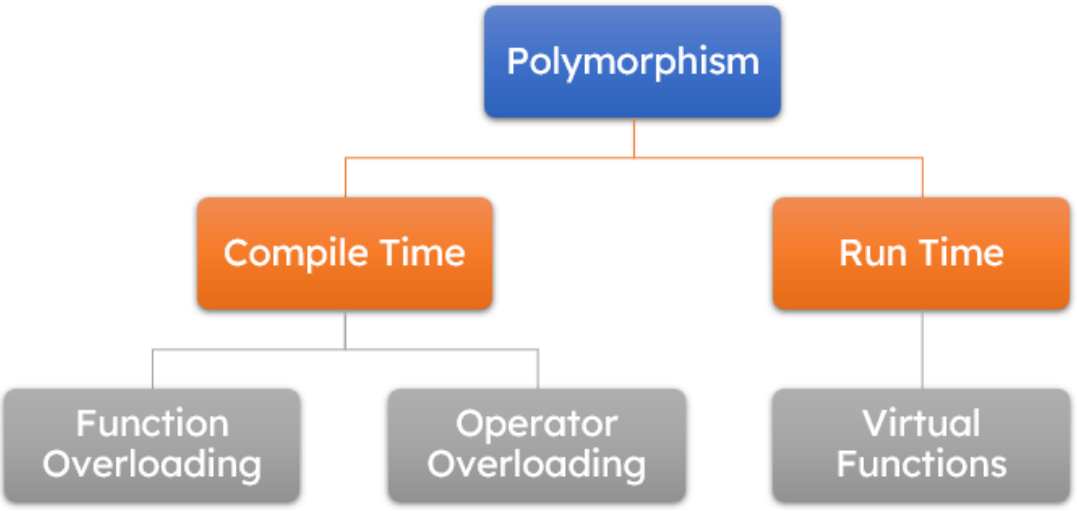
OOPC-04-Polymorphism

The word “Polymorphism” means **having many forms**. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behaviour in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming

Types of Polymorphism:

- Compile-time Polymorphism
- Runtime Polymorphism



Compile Time Polymorphism

- This type of polymorphism is achieved by **Function Overloading** or **Operator Overloading**.

Function Overloading

- When there are multiple functions with the same name but different parameters, then the functions are said to be overloaded, hence this is known as **Function Overloading**.
- Functions can be overloaded by changing the number of arguments or/and changing the type of arguments.
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.
- 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.
- There are certain Rules of Function Overloading that should be followed while overloading a function.

```cpp
// C++ program to show function overloading or compile-time polymorphism

#include <bits/stdc++.h>
using namespace std;

class Geeks {
  public:
    // Function with 1 int parameter
    void func(int x) {
      cout << "value of x is " << x << endl;
    }
  // Function with same name but
  // 1 double parameter
  void func(double x) {
    cout << "value of x is " << x << endl;
  }
  // Function with same name and
  // 2 int parameters
  void func(int x, int y) {
    cout << "value of x and y is " << x << ", " << y <<
      endl;
  }
};

// Driver code
int main() {
  Geeks obj1;
  // Function being called depends
  // on the parameters passed
  // func() is called with int value
  obj1.func(7);
  // func() is called with double value
  obj1.func(9.132);
  // func() is called with 2 int values
  obj1.func(85, 64);
  return 0;
}
```

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

```cpp
//Example of function overloading
#include <iostream>
using namespace std;

int mul(int a,int b) {
    return a*b;
}
float mul(double x, int y) {
    return x*y;
}
```

```
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    cout << "r1 is: " << r1 << endl;
    cout << "r2 is: " << r2 << endl;
    return 0;
}
```

```
r1 is: 42
r2 is 0.6
```

Operator Overloading

- C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as **Operator Overloading**.
- The advantage of Operators overloading is to perform different operations on the same operand
- For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
//Syntax of Operator overloading
return_type class_name :: operator op(argument_list) {
    // body of the function.
}
```

ⓘ Difference Between Operator Functions and Normal Functions:

- Operator functions are the same as normal functions.
- The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.

- Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded:
  `sizeof` `typeid` `Scope resolution (::)` `Class member access operators (.(dot), .*(pointer to member operator))` `Ternary or conditional (?:)`

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Unary Operator overloading

- In the unary operator function, no arguments should be passed.
- It works only with one class object. It is the overloading of an operator operating on a single operand

```
//Example to explain how minus (-) operator can be overloaded for prefix as well as

#include <iostream>
using namespace std;

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
  void operator - () {
    feet = -feet;
    inches = -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;
}
```

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

- We can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

```
// C++ Program to concatenate two string using unary operator overloading
#include <iostream>
#include <string.h>
using namespace std;

// Class to implement operator overloading function for concatenating the strings
class AddString {
  public:
    // Classes object of string
    char s1[25], s2[25];
  // Parameterized Constructor
  AddString(char str1[], char str2[]) {
    // Initialize the string to class object
    strcpy(this -> s1, str1);
    strcpy(this -> s2, str2);
  }
  // Overload Operator+ to concat the string
  void operator + () {
    cout << "\nConcatenation: " << strcat(s1, s2);
  }
};

// Driver Code
int main() {
  // Declaring two strings
  char str1[] = "Silver Oak ";
  char str2[] = "University";
  // Declaring and initializing the class with above two strings
  AddString a1(str1, str2);
  // Call operator function
  +
  a1;
  return 0;
}
```

```
Concatenation: Silver Oak University
```

Binary Operator Overloading

- An operator which contains two operands to perform a mathematical operation is called the Binary Operator Overloading.
- In the binary operator overloading function, there should be one argument to be passed.
- It is the overloading of an operator operating on two operands

- It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands from the programmer or user.
- There are multiple binary operators like `+, -, *, /` etc., that can directly manipulate or overload the object of a class.

```
// Syntax of Binary Operator Overloading
return_type :: operator binary_operator_symbol (arg){
    //Function Definition
}
```

```
// C++ program to show binary operator overloading
#include <iostream>
using namespace std;

class Distance {
  public: int feet, inch;
    Distance() {
        this -> feet = 0;
        this -> inch = 0;
    }
    Distance(int f, int i) {
        this -> feet = f;
        this -> inch = i;
    }
  // Overloading (+) operator to
  // perform addition of two distance
  // object
  // Call by reference
    Distance operator + (Distance & d2) {
        // Create an object to return
        Distance d3;
        d3.feet = this -> feet + d2.feet;
        d3.inch = this -> inch + d2.inch;
        // Return the resulting object
        return d3;
  }
};

// Driver Code
int main() {
  Distance d1(8, 9);
  Distance d2(10, 2);
  Distance d3;
  // Use overloaded operator
  d3 = d1 + d2;
  cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
  return 0;
}
```

```
Total Feet & Inches: 18'11
```

Example of plus(+) and Minus(-) binary operator overloading:

```cpp
// Write a program to demonstrate binary operator overloading
#include <iostream>
using namespace std;

class complex {
  int a, b;
  public:
    void get_data() {
      cout << "Enter the value of Complex Numbers a, b: ";
      cin >> a >> b;
    }
  // overaloded operator function +
  complex operator + (complex ob)
  {
    complex t;
    t.a = a + ob.a;
    t.b = b + ob.b;
    return (t);
  }
  // overaloded operator function -
  complex operator - (complex ob)
  {
    complex t;
    t.a = a - ob.a;
    t.b = b - ob.b;
    return (t);
  }
  void display() {
    cout << a << "+" << b << "i" << "\n";
  }
};

int main() {
  complex obj1, obj2, result, result1;
  obj1.get_data();
  obj2.get_data();
  result = obj1 + obj2;
  result1 = obj1 - obj2;
  cout << "\n\nInput Values:\n";
  obj1.display();
  obj2.display();
  cout << "\nResult:";
  result.display();
  result1.display();
  return 0;
}
```

```
-- INPUT
Enter the value of Complex Numbers a, b: 7 5
Enter the value of Complex Numbers a, b: 3 4
```

```
-- OUTPUT
Input Values:
7+5i
3+4i

Result:10+9i
4+1i
```

Runtime Polymorphism

Function Overloading

- It takes place when your derived class and base class both contain a function having the same name.
- Along with the same name, both the functions should have the same number of arguments as well as the same return type.
- The derived class inherits the member functions and data members from its base class.
- So to override a certain functionality, you must perform function overriding. It is achieved during the run time.
- Functions that are overridden acquire different scopes.
- For function overriding, **inheritance** is a must. It can only happen in a **derived class**.
- If a class is not inherited from another class, you can not achieve function overriding.
- To sum up, a function is overridden when you want to achieve a task supplementary to the base class function.

```
// Program to illustrate run-time polymorphism using function overriding

#include <iostream>
using namespace std;

// define a base class
class bird {
  public:
    // display function of the base class
    void display() {
      cout << "I am the display function of the base class";
      cout << "\n\n";
    }
};
class parrot: public bird {
  public:
    // display function of the serived class
    // this function will display()
    // of base class override at run time
    void display() {
      cout << "I am the display function of the derived class";
      cout << "\n\n";
    }
};

int main() {
```

```
  // create objects of base and child classes
  bird b;
  parrot p;
  // call the diplay() function
  b.display();
  p.display();
}
```

```
I am the display function of the base class

I am the display function of the derived class
```

Virtual Functions

- Virtual functions are the member functions of the base class which are overridden in a derived class.
- When you make a call to such a function by using a pointer or reference to the base class, the virtual function is called and the derived class's version of the function gets invoked.
  Some Key Points About Virtual Functions:
- Virtual functions are only dynamic
- They are declared within a base class by using the keyword "virtual"
- These functions are called during run time
- Virtual functions are always declared with a base class and overridden in a child class
- Virtual functions can exist in the absence of inheritance. In that case, the base class version of the function will be called.

```
// Program to illustrate run-time polymorphism using a virtual function

#include <iostream>
using namespace std;

// define a base class bird
class bird {
  public:
    // virtual function
    virtual void display() {
      cout << "This is display in bird class." << "\n\n";
    }
  void print() {
    cout << "This is show in bird class." << "\n\n";
  }
};
// define a child class parrot
class parrot: public bird {
  public: void display() {
    cout << "This is display in parrot class." << "\n\n";
  }
  void print() {
    cout << "This is show in parrot class." << "\n\n";
  }
```

```
};

int main() { // create a reference of class bird
  bird * brd;
  parrot p;
  brd = & p;
  // this will call the virtual function
  brd -> display();
  // this will call the non-virtual function
  brd -> print();
}
```

```
This is display in parrot class.

This is show in bird class.
```

Pure Virtual Functions

- A pure virtual function (or Abstract Function) in C++ is a virtual function for which we can have an implementation, but we must override that function in the derived class, otherwise, the derived class will also become an abstract class.
- A pure virtual function is declared by assigning **0** in the declaration.
- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **Abstract Class**.
- For example, let Shape be a base class. We cannot provide the implementation of function `draw()` in Shape, but we know every derived class must have an implementation of `draw()`.
- Similarly, an Animal class doesn't have the implementation of `move()` (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

```
// C++ Program to illustrate the abstract class and virtual functions

#include <iostream>
using namespace std;

class Base {
  // private member variable
  int x;
  public:
    // pure virtual function
    virtual void fun() = 0;
  // getter function to access x
  int getX() {
    return x;
  }
};

// This class inherits from Base and implements fun()
class Derived: public Base {
```

```
  // private member variable
  int y;
  public:
    // implementation of the pure virtual function
    void fun() {
      cout << "fun() called";
    }
};

int main(void) {
  // creating an object of Derived class
  Derived d;
  // calling the fun() function of Derived class
  d.fun();
  return 0;
}
```

```
fun() called
```

Difference between Compile-Time & Run-Time

Compile Time	Run Time
Compile-time polymorphism is also known as static or early binding polymorphism.	Run-time polymorphism is also known as dynamic or late binding polymorphism.
The function calls are resolved by the compiler.	The function calls are not resolved by the compiler.
Compile-time polymorphism provides less flexibility to the programmers since everything is executed during compilation.	In contrast, run-time polymorphism is more flexible since everything is executed during run-time.
It can be implemented through function overloading and operator overloading.	It can be implemented through virtual functions and function overriding.
Method overloading is an application of compile-time polymorphism where the same name can be commissioned between more than one method of functions having different arguments or signatures and the same return types.	Method overriding is an application of run time polymorphism where two or more functions with the same name, arguments, and return type accompany different classes of the same structure.
This method has a much faster execution rate since all the methods that need to be executed are called during compile time.	This method has a comparatively slower execution rate since all the methods that need to be executed are called during the run time.
This method is less preferred for handling compound problems since all the methods and details come to light only during the compile time.	This method is known to be better for dealing with compound problems since all the methods and details turn up during the run time

Questions

1. Compare and contrast Compile Vs. Runtime polymorphism.
2. Explain the concept of polymorphism.

3. Explain Function overloading with an example.
4. What is Operator overloading ? Explain with an example.
5. Explain Function overriding with an example.
6. Explain Virtual function with an example.
7. Explain Pure Virtual function with an example

Join Us on Telegram @SOU BCA