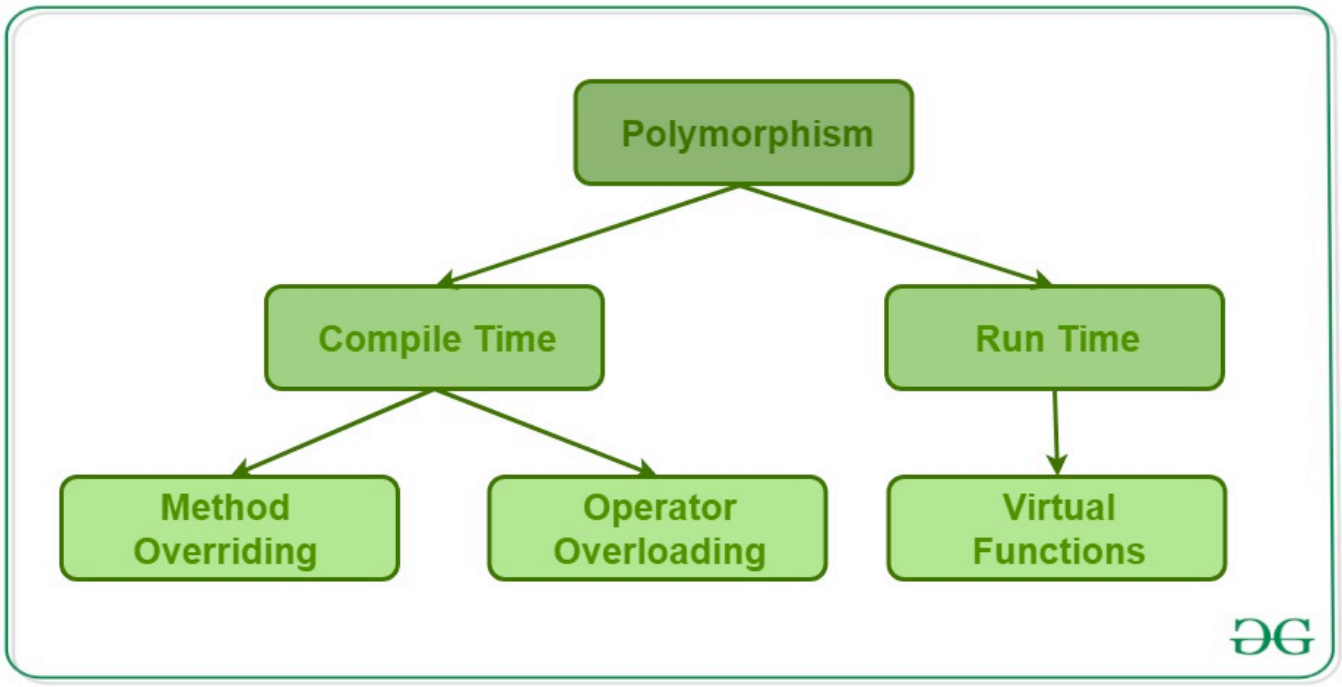
Polymorphism in C++



The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by three other classes –

```
class Shape {
public:
    Shape(float wid = 0, float hgt = 0) : width(wid), height(hgt) {}

    float area() const {
        float result = 0;
        cout << "Parent class area : " << result << endl;
        return result;
    }
protected:
    float width;
    float height;
};

class Rectangle : public Shape {
```

```
public:
    Rectangle(float wid = 0, float hgt = 0) : Shape(wid, hgt) {}

    float area() const {
        float result = width * height;
        cout << "Rectangle class area : " << result << endl;
        return result;
    }
};

class Triangle : public Shape {
public:
    Triangle(float wid = 0, float hgt = 0) : Shape(wid, hgt) {}

    float area() const {
        float result = width * height / 2;
        cout << "Triangle class area : " << result << endl;
        return result;
    }
};

class Circle : public Shape {
public:
    Circle(float rad = 0) : Shape(rad) {}

    float area() const {
        float result = static_cast<float>(width * width * M_PI);
        cout << "Circle class area : " << result << endl;
        return result;
    }
};

// Main function for the program
int main() {
    Rectangle rec(10, 5);
    Triangle  tri(10, 5);
    Circle    cir(10);

    rec.area();
    tri.area();
    cir.area();

    return 0;
}
```

When the above code is compiled and executed, it produces the following result --

```
Rectangle class area : 50
Triangle class area : 25
Circle class area : 314.159
```

As expected.

Remember: inheritance is an IS-A relationship. Since a Rectangle IS-A Shape, we can assign the address of a

Rectangle object to a pointer to the base class, Shape. Let's make the following changes to the main function:

```
// Main function for the program
int main() {
    Shape*    shape = nullptr;
    Rectangle rec(10, 5);
    Triangle  tri(10, 5);
    Circle    cir(10);

    // store the address of Rectangle
    shape = &rec;

    // call rectangle area.
    shape->area();

    // store the address of Triangle
    shape = &tri;

    // call triangle area.
    shape->area();

    // store the address of Circle
    shape = &cir;

    // call circle area.
    shape->area();

    return 0;
}
```

When the above code is compiled and executed, it produces the following result --

```
Parent class area : 0
Parent class area : 0
Parent class area : 0
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** or the function call, or **static linkage** -- the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification to our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it now looks like this --

```
class Shape {
public:
    Shape(float wid = 0, float hgt = 0) : width(wid), height(hgt) {}

    virtual float area() const {
        float result = 0;
        cout << "Parent class area : " << result << endl;
        return result;
```

```
    }
protected:
    float width;
    float height;
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result --

```
Rectangle class area : 50
Triangle class area : 25
Circle class area : 314.159
```

This time, the compiler looks at the contents of the pointer instead of its type. Hence, since addresses of objects of Rectangle, Triangle, and Circle classes are stored in *shape, the respecitve area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function with the same signature (same number and type of arguments), but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining a virtual function in a base class with overridden functions in derived classes signals the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage** or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following –

```
    virtual float area() const = 0;
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**. A class containing abstract functions cannot be instantiated. Any derived class must implement the abstract functions, or it, too, is abstract. Only derived classes that implement all abstract functions may be instantiated.