SCS1310: Object-Oriented Modelling and Programming

Polymorphism

Viraj Welgama

4 Basic Concepts

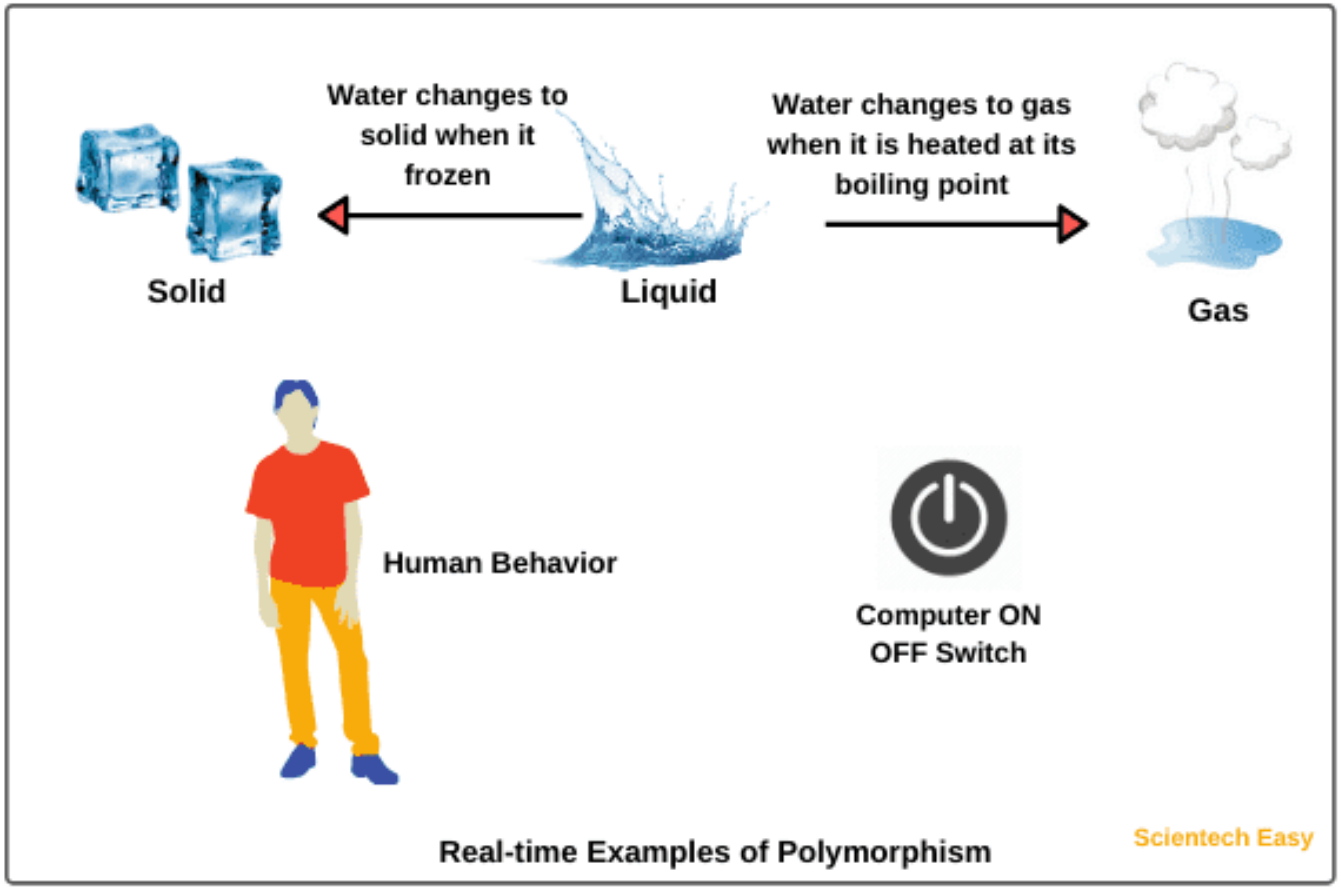
1. Inheritance
2. **Polymorphism**
3. Encapsulation (Information Hiding) and
4. Abstraction

Polymorphism

- The word polymorphism means having many forms.
  - Poly -> many
  - Morph -> shapes
  - Ism -> specific practice or theory

- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Polymorphism



Polymorphism

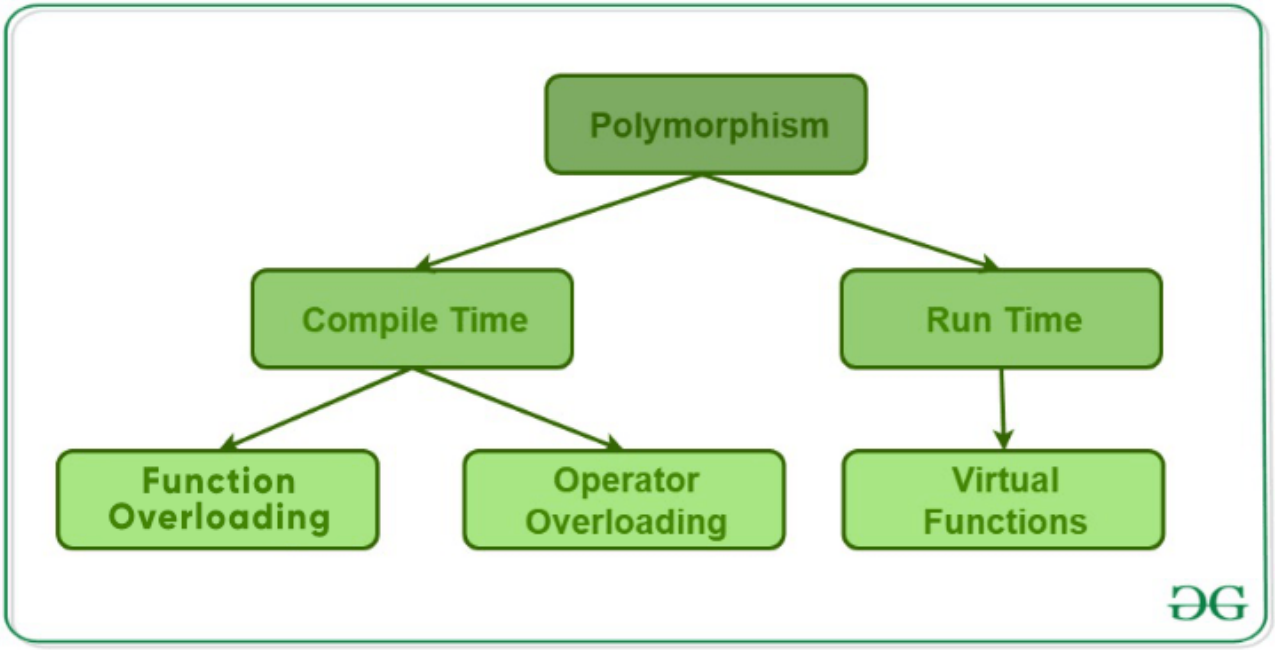
- A real-life example of polymorphism, a person at the same time can have different characteristics.
  - Like a man at the same time is a father, a husband, an employee.
  - So the same person posses different behavior in different situations.
- This is called polymorphism.
- Polymorphism is considered as one of the important features of Object Oriented Programming.

2 Types in C++

1. Compile Time Polymorphism
2. Runtime Polymorphism

2 Types in C++

1. Compile Time Polymorphism
2. Runtime Polymorphism



Compile Time Polymorphism

- This type of polymorphism is achieved by function overloading or operator overloading.
- The correct function version is determined by the compiler at compile time based on the arguments passed to the function call.
- Unlike runtime polymorphism (achieved using virtual functions), function overloading does not involve dynamic binding or late binding. Instead, it is resolved during compilation.

Function Overloading

- When there are multiple functions with same name but different parameters then these functions are said to be overloaded.
- Functions can be overloaded **if they have different function signatures**.

Signature of a Function

- A function signature typically includes:
    1. Function name
    2. Number of parameters
    3. Types of parameters
    4. Order of parameters

- Return type and the parameter names are not part of a function signature.

Overloading can be done by…

1. Same Function name but with different
2. Number of parameters
3. Types of parameters
4. Order of parameters

- But, can not be achieved by different return type and the parameter names.

Example: calcArea()

In here, a single function named ***calcArea()*** acts differently in three different situations which is the property of polymorphism.

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

class Shape {
    int height;
    int width;
    double radious;

    public:
        int calcArea(int ah) { //area of a square
            return ah*ah;
        }

        int calcArea(int ah, int aw) { // area of a rectangle
            return ah*aw;
        }

        double calcArea(double ar) { // area of a circle
            return 3.4*ar*ar;
        }

        int calcArea(char ar) { // area of a circle
            return 3.14*ar*ar;
        }
};

int main() {
    Shape obj;
    cout<<"area of the square: "<<obj.calcArea(10)<<endl;
    cout<<"area of the rectangle: "<<obj.calcArea(10,20)<<endl;
    cout<<"area of the circle: "<<obj.calcArea(10.0f)<<endl;
}
```

Runtime Polymorphism

- This type of polymorphism is achieved by **Function Overriding**.
- occurs when a derived class has a definition for one of the member functions of the base class.
- That base function is said to be **overridden**.
  - Special dish of the Italian chef…

Example: Chef

```
class Chef{
    public:
        void makeChicken(){
            cout << "The chef makes chicken" << endl;
        }

        void makeSalad(){
            cout << "The chef makes salad" << endl;
        }

        void makeSpecialDish(){
            cout << "The chef makes a special dish" << endl;
        }
};

class ItalianChef : public Chef{
    public:
        void makePasta(){
            cout << "The chef makes pasta" << endl;
        }
        // override
        void makeSpecialDish(){
            cout << "The chef makes chicken parm" << endl;
        }
};

int main(){

    Chef myChef;
    myChef.makeChicken();

    ItalianChef myItalianChef;
    myItalianChef.makeChicken();

    myChef.makeSpecialDish();
    myItalianChef.makeSpecialDish();

    return 0;
}
```

avoiding mistakes in overriding

- Function overriding is redefinition of base class function in its derived class with same signature
  - i.e name and parameters list.
- But there may be situations when a programmer makes a mistake while overriding that function.
- So, to keep track of such an error, C++11 has come up with the keyword ***override***.
- It will make the compiler to check the base class to see if there is a virtual function with this exact signature.
  - And if there is not, the compiler will show an error.

Example: the issue

Here the user intended to override the function func() in the derived class but did a small mistake and redefined the function with different signature (or maybe with some spelling mistake) which was not detected by the compiler
So, the program is not actually what the user wanted. So, to get rid of such shallow mistake to be in safe side, override keyword can be used.

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

class Base {
public:

    // user wants to override this in
    // the derived class
    virtual void func() {
        cout << "I am in base" << endl;
    }
};

class derived : public Base {
public:
    // did a silly mistake by putting
    // an argument "int a"
    void func(int a) {
        cout << "I am in derived class" << endl;
    }
};

// Driver code
int main()
{
    Base b;
    derived d;
    cout << "Compiled successfully" << endl;
    return 0;
}
```

Example: Solution

keyword *override* helps to check if :

1. There is a method with the same name in the parent class.
2. The method in the parent class is declared as “virtual” which means it was intended to be rewritten.
3. The method in the parent class has the same signature as the method in the subclass.

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

class Base {
public:

    // user wants to override this in
    // the derived class
    virtual void func() {
        cout << "I am in base" << endl;
    }
};

class derived : public Base {
public:
    // did a silly mistake by putting
    // an argument "int a"
    void func(int a) override {
        cout << "I am in derived class" << endl;
    }
};

// Driver code
int main()
{
    Base b;
    derived d;
    cout << "Compiled successfully" << endl;
    return 0;
}
```

Predict the output:

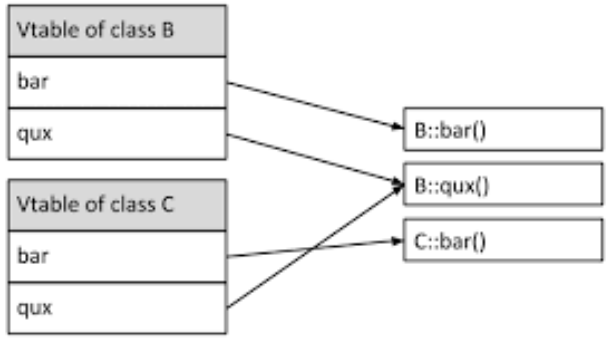
```cpp
#include <iostream>
using namespace std;
// base class
class Vehicle {
public:
    string getName() {
        return "I am a vehicle\n";
    }
    void printName() {
        cout<<getName();
    }
};
// derived class
class Car : public Vehicle {
public:
    string getName() {
        return "I am a car\n";
    }
};

// Driver code
int main()
{
    Vehicle vhe;
    Car car;
    vhe.printName();
    car.printName();
    return 0;
}
```

Virtual Functions

- virtual function is member function which declared within the base class and it re-defined (override) by the derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
  - Virtual functions ensure that the correct function is called for an object, regardless of the type of the reference used to call the function.
  - They are mainly used to achieve Runtime Polymorphism
  - Functions are declared with a virtual keyword in the base class.
  - The resolving of function call is done at run-time.

vtable

- Virtual functions introduce a concept called dynamic dispatch which typically implement using *vtable*.
  - For every class that contains virtual functions, the compiler constructs a virtual table, a.k.a vtable.
- The vtable contains an entry for each virtual function accessible by the class and stores a pointer to its definition.



cost of virtual functions

1. It requires an initial memory to store the vtable, which contains all the mappings to the virtual functions.
2. Every time we call a virtual function, it has to go through the vtable to find out which function has to call, which an additional performance penalty.

However, you can ignore these two factors if the benefit of using them is more worth than these costs.

Pure Virtual Functions

- Sometimes implementation of all the functions cannot be provided in the bass class because we don't know the implementation.
  - calcArea() function in the Shape class
- We can define these kinds of functions as pure virtual functions (or abstract functions) in the base class.
  - We do not have the implementation, but we **declare it**.
  - It is declared by assigning 0 in the declaration.

Abstract Classes

- A class is Abstract if it has **at least one** pure virtual function.
- If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.
- Abstract classes **can NOT be instantiated.**

Example:

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

// base class
class Shape {
public:
    virtual void getArea(int x=0, int y=0) = 0;
};
// derived class 1
class Circle : public Shape {
public:
    void getArea(int ar, int x=1) {
        cout<<"Area of the circle: "<<(3.14*ar*ar)<<endl;
    }
};
// derived class 2
class Rectangle : public Shape {
public:
    void getArea(int aw, int al) {
        cout<<"Area of the rectangle: "<<(aw*al)<<endl;
    }
};
// Driver code
int main()
{
    Circle cir1;
    cir1.getArea(25);
    Rectangle rec1;
    rec1.getArea(10, 20);
    return 0;
}
```

Use of overriding…

- Array of Shapes

Method Hiding in c++

What would be the outcome of this program…

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

class Base {
public:

    // user wants to override this in
    // the derived class
    virtual void func() {
        cout << "I am in base" << endl;
    }
};

class derived : public Base {
public:
    // did a silly mistake by putting
    // an argument "int a"
    void func(int a)  {
        cout << "I am in derived class" << endl;
    }
};

// Driver code
int main()
{
    Base b;
    derived d;
    d.func();
    return 0;
}
```

Method Hiding in C++

- In C++, if a derived class redefines base class member method then all the base class methods with same name become hidden in derived class.
- The above program doesn't compile because *derived* redefines Base's method func(int) and this makes func() hidden.
- Even if the signature of the derived class method is different, all the overloaded methods in base class become hidden.