Lecture 13
OOP

Khola Naseem
khola.naseem@uet.edu.pk

Polymorphism:

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance lets us to use 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.
- Polymorphism in C++ means, the same entity (function or object) behaves differently in different scenarios
- 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 Dogs bark, and the cat meows, etc.)

Polymorphism:

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() {
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
int main()
{
    Animal a;
    cat c;
    Dog d;
    a.animalSound();
    c.animalSound();
    d.animalSound();
}
```

Output:

```
The animal makes a sound
The cat says: meow meow
The dog says: bow wow
```

Types of Polymorphism in C++:

- **Compile Time Polymorphism**
  - In compile-time polymorphism, a function is called at the time of program compilation. We call this type of polymorphism as early binding or Static binding
  - Function overloading and operator overloading is the type of Compile time polymorphism.
- **Runtime Polymorphism**
  - In a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.
  - Function overriding using a virtual function is a part of runtime polymorphism. In function overriding, more than one method has the same name with same types of the parameter list.
  - It is achieved by using virtual functions and pointers. It provides slow execution as it is known at the run time. Thus, It is more flexible as all the things executed at the run time

Access Overridden Function in C++

- Third method

```
class Base {
   public:
    void print() {
        cout << "Print function in Base class" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Print function in derived class" << endl;
        //Base::print();
    }
};

int main() {
    Base *b;
    Derived d1;
    b=&d1;
    b->print();
   // d1.Base::print();   //access overridden function

    return 0;
```

```
Print function in Base class
```

Virtual Function:

- If it is necessary to use a single pointer to refer to all the different classes' objects. This is because we will have to create a pointer to the base class that refers to all the derived objects.
- But, when the base class pointer contains the derived class address, the object always executes the base class function. For resolving this problem, we use the virtual function.
- When we declare a virtual function, the compiler determines which function to invoke at runtime.
- A virtual function is a member function in the base class. We can redefine it in a derived class. It is part of run time polymorphism. The declaration of the virtual function must be in the base class by using the keyword virtual. A virtual function is not static.
- The virtual function helps to tell the compiler to perform dynamic binding or late binding on the function.

Virtual Function:

- in a Runtime polymorphism, functions are called at the time the program execution. Hence, it is known as late binding or dynamic binding.
- Function overriding is a part of runtime polymorphism. In function overriding, more than one method has the same name with same type of the parameter list.
- A virtual function is declared by keyword virtual. The return type of virtual function may be int, float, void.
- Define a virtual function:

```
class Base {
   public:
    virtual void functionname() {
        // code
    }
};
```

Virtual Function:

- Example:

```cpp
#include <iostream>

using namespace std;

class Base {
   public:
    virtual void print() {
        cout << "Print function in Base class" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Print function in derived class" << endl;
        //Base::print();
    }
};

int main() {
    Base *b;
    Derived d1;
    b=&d1;
    b->print();
   // d1.Base::print();    //access overridden function

    return 0;
}
```

```
Print function in derived class
```

Virtual Function:

- Example:

```
class Add
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }

    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class Mul: public Add
{
public:
    void print ()
    {
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }

    void display ()
            {
            cout<< "display function of  derived class" <<endl;
            }
};
int main()
{
    Add *ptr;
    Mul m;
    ptr = &m;
    //binded at runtime (Runtime polymorphism) beacuse of virtual function
    ptr->print();
    //binded at compile time a  Non-virtual function
    ptr->display();
    return 0;
}
```

```
 Multiplication in drived class:50
display function of base class
```

Virtual Function:

- C++ override Identifier
- C++ 11 has given us a new identifier override that is very useful to avoid bugs while using virtual functions.

```
class Parent {
   public:
    virtual void display() {
        // code
    }
};

class Derived : public Parent {
   public:
    void display() override {
        // code
    }
};
```

Credit: Khola Naseem

Virtual Function:

- If want to define function outside the class

```
class Derived : public Parent {
   public:
    // function prototype
    void display() override;
};
// function definition
void Derived::display() {
    // code
}
```

Virtual Function:

C++ 11 has given us a new identifier override that is very useful to avoid bugs while using virtual functions.

```
class Add
{
public:
    virtual void print ()
    {
    int a=5;
    int b=10;
     cout<< " Addition in base class :"<<a+b <<endl;
      }

    void display ()
    {
    cout<< "display function of base class" <<endl;
    }
};
class Mul: public Add
{
public:
    void print () override
    {
    int x=5;
    int y=10;
    cout<< " Multiplication in drived class:"<<x*y <<endl;
    }

    void display ()
            {
            cout<< "display function of  derived class" <<endl;
            }
};
int main()
{
    Add *ptr;
    Mul m;
    ptr = &m;
    //binded at runtime (Runtime polymorphism) beacuse of virtual function
    ptr->print();
    //binded at compile time a  Non-virtual function
    ptr->display();
    return 0;
}
```

```
 Multiplication in drived class:50
display function of base class
```

Credit: Khola Naseem

Virtual Function:

- When using virtual functions, it is possible to make mistakes while declaring the member functions of the derived classes.
- Using the override identifier prompts the compiler to display error messages when these mistakes are made.
- Otherwise, the program will simply compile but the virtual function will not be overridden.
- Some of these possible mistakes are:
  - **Functions with incorrect names:**
    - Display() in derived class disply()
  - **Functions with different return types:**
    - Virtual function in parent class return type void in derived class int
  - **Functions with different parameters:**
    - If the parameters of the virtual function and the functions in the derived classes don't match.

Virtual Function:

- Example:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() {
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
int main()
{
    cat c;
    Dog d;

    Animal* ac=&c;
    Animal* ad=&d;

    ac->animalSound();
    ad->animalSound();

}
```

```
The animal makes a sound
The animal makes a sound
```

Credit: Khola Naseem

Virtual Function:

- Example:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
  public:
    virtual void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
class cat : public Animal {
  public:
    void animalSound() override{
      cout << "The cat says: meow meow \n";
    }
};
class Dog : public Animal {
  public:
    void animalSound()override {
      cout << "The dog says: bow wow \n";
    }
};


int main()
{
    cat c;
    Dog d;
    Animal* ac=&c;
    Animal* ad=&d;
    ac->animalSound();
    ad->animalSound();

}
```

```
The cat says: meow meow
The dog says: bow wow
```

Credit: Khola Naseem

Polymorphism:

- Example:

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      int area() {
         cout << "Area of parent class :" << width * height << endl;
         return width * height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
```

```
Area of parent class :6
Area of parent class :32
```

Credit: Khola Naseem

Polymorphism:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() {
         cout << "Area of parent class :" << width * height << endl;
         return width * height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
   return 0;
}
```

```
Area of rectangle class :6
Area of triangle class :16
```

Credit: Khola Naseem

Overriding vs Overloading:

Function Overloading	Function Overriding
Function overloading in C++ can occur with or without inheritance.	Function overriding in C++ can only occur in the presence of inheritance.
Overloaded functions must have different function signatures i.e., the number of parameters or the data type of parameters should be different.	Overridden functions must have the same function signature i.e., the number of parameters as well as their data type must be he same.
It represents the compile-time polymorphism or early binding as overloading occurs during compile time.	It represents the run-time polymorphism or late binding as overriding occurs during run time.
Overloading takes place within the same class	Overriding occurs in a parent class and its child class.
No special keyword is used to overload a function.	Virtual keyword in the base class and Override keyword in the derived class can be used to override a function.
Overloading is done to acquire the different behavior to the same function depending on the arguments passed to the functions.	Overriding is done when the derived class function is expected to perform differently than the base class function.

Credit: Khola Naseem

Pure virtual function:

- Pure virtual functions are used
  - if a function doesn't have any use in the base class
  - but the function must be implemented by all its derived classes
- Run time polymorphism
- Example: Suppose, we have derived Triangle and Rectangle classes from the Shape class, and we want to calculate the area of all these shapes.
- In this case, we can create a pure virtual function named area() in the Shape. Since it's a pure virtual function, all derived classes Triangle and Rectangle must include the area() function with implementation.
- A pure virtual function doesn't have the function body and it must end with = 0
- Syntax:
  - class Shape

```
{ public:
 virtual void area() = 0;
  };
```

Pure virtual function:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() =0;
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   shape = &t;
   shape->area(); //Triangle class
   return 0;
}
```

Output:

```
Area of rectangle class :6
Area of triangle class :16
```

Credit: Khola Naseem

Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).
-

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()=0;
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Shape s;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

Error

```
[Error] cannot declare variable 's' to be of abstract type 'Shape'
[Note] because the following virtual functions are pure within 'Shape':
[Note] virtual int Shape::area()
```

Credit: Khola Naseem

Abstract Class:

- A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.
- We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions) .
-

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() =0;
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get height for rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get height for Triangle"<<shape->getHeight();
   return 0;
}
```

Output:

```
Area of rectangle class :6
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Credit: Khola Naseem

Abstract Class:

- In this case, we can create a pure virtual function named area() in the Shape. Since it's a pure virtual function, all derived classes Triangle and Rectangle must include the area() function with implementation. otherwise, the derived class is also abstract.

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area() =0;
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      /*int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }*/
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

```
In function 'int main()':
[Error] cannot declare variable 'r' to be of abstract type 'Rectangle'
[Note] because the following virtual functions are pure within 'Rectangle':
[Note] virtual int Shape::area()
```

Credit: Khola Naseem

Abstract Class:

- **Example:**
- Creating Abstract Base Class Employee
- Class Employee provides functions earnings and print, in addition to various get and set functions that manipulate Employee's data members. An earnings function certainly applies generally to all employees, but each earnings calculation depends on the employee's class. So we declare earnings as pure virtual in base class Employee because a default implementation does not make sense for that function—there is not enough information to determine what amount earnings should return. Each derived class overrides earnings with an appropriate implementation. To calculate an employee's earnings.

Virtual VS pure virtual:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()
      {
        cout<<"shape area \n";
      }
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      /*int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }*/
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

Output:

```
shape area
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Credit: Khola Naseem

Virtual VS pure virtual:

- Example:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int c = 0, int d = 0){
         width = c;
         height = d;
      }
      virtual int area()
      {
        cout<<"shape area \n";
      }
      int getHeight()
      {
        return height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int c = 0, int d = 0):Shape(c, d) { }
      int area () override {
         cout << "Area of rectangle class :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int c = 0, int d = 0):Shape(c, d) { }

      int area () override {
         cout << "Area of triangle class :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
int main() {
   Shape *shape;
   Shape s;
   Rectangle r(3,2);
   Triangle  t(4,8);
   shape = &r;
   shape->area(); //rectangle class
   cout<<"Get the height of rectangle: "<<shape->getHeight()<<endl;
   shape = &t;
   shape->area(); //Triangle class
   cout<<"Get the height of Triangle: "<<shape->getHeight();
   return 0;
}
```

Output:

```
Area of rectangle class :6
Get the height of rectangle: 2
Area of triangle class :16
Get the height of Triangle: 8
```

Credit: Khola Naseem

Reference material

- **For Practice Questions, refer to these books**
  - C++ Programming From Problem Analysis To Program Design, 5th Edition, D.S.Malik. Chapter 12.
  - C++ How to Program, Deitel & Deitel, 5th Edition, Prentice Hall.
  - Object Oriented Programming in C++ by Robert Lafore.
  - Object Oriented Software Construction, Bertrand Meyer's
  - Object-Oriented Analysis and Design with applications, Grady Booch et al, 3Rd Edition, Pearson, 2007
  - Web