

Recap Inheritance and Polymorphism

By, Mahrukh Khan

Recap-Inheritance

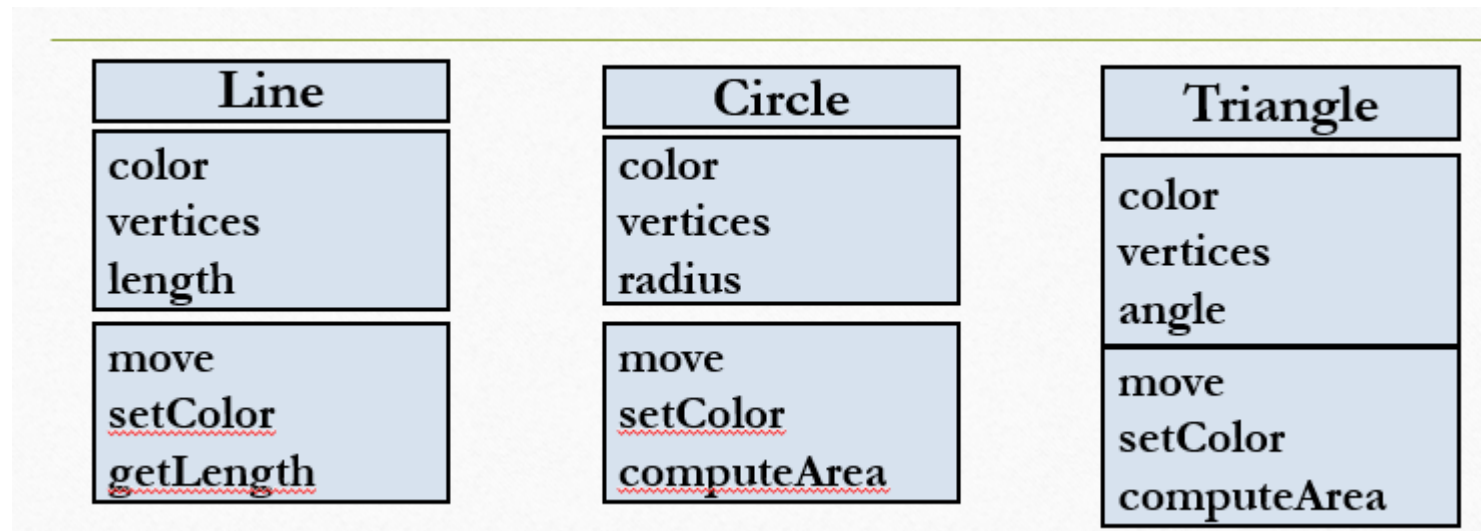
- ▶ Derived class inherits all the characteristics of the base class
- ▶ Besides inherited characteristics, derived class may have its own unique characteristics
- ▶ Major benefit of inheritance is reuse

Concepts related with Inheritance

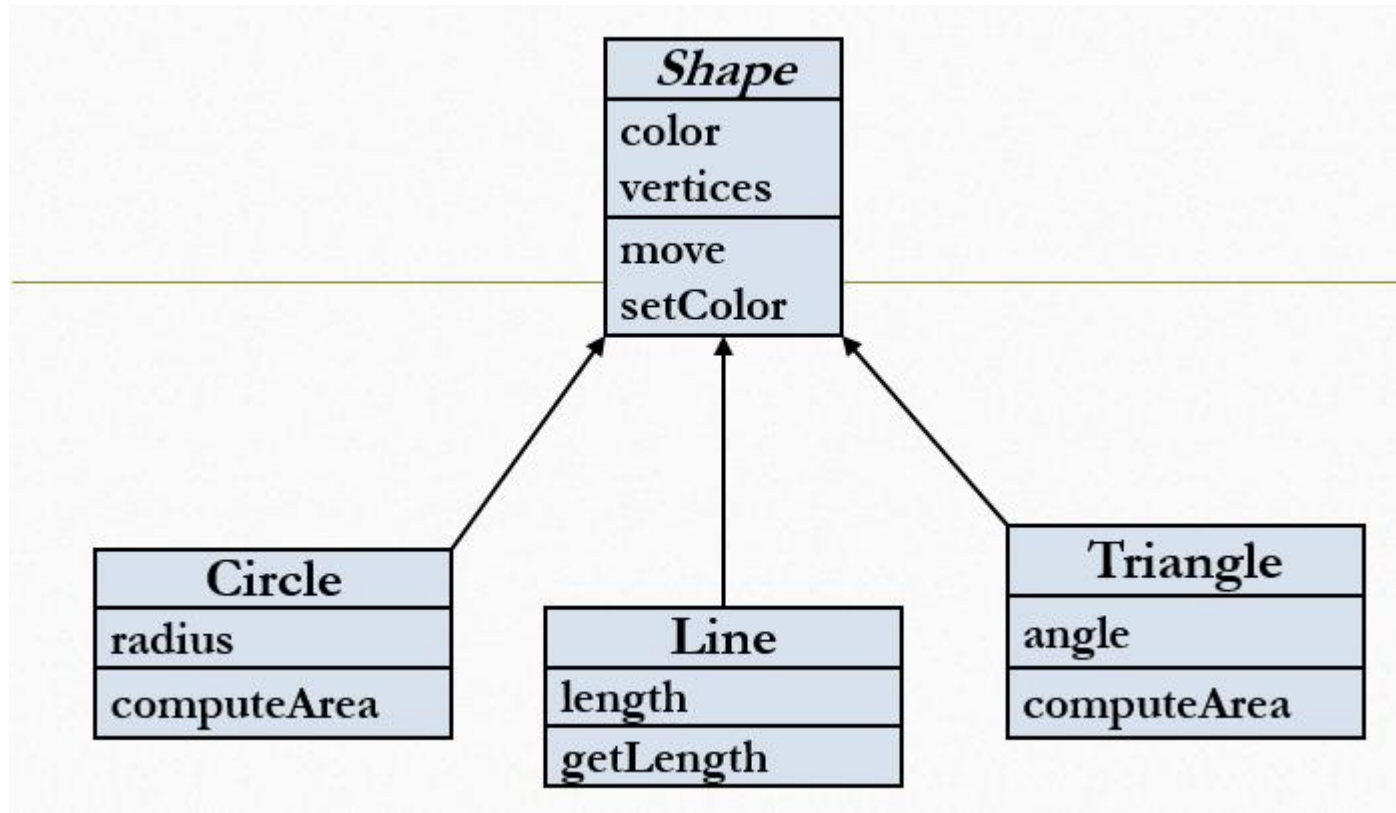
- ▶ Generalization
- ▶ Subtyping (extension)
- ▶ Specialization (restriction)

Generalization

- ▶ In OO models, some classes may have common characteristics
- ▶ We extract these features into a new class and inherit original classes from this new class
- ▶ This concept is known as Generalization

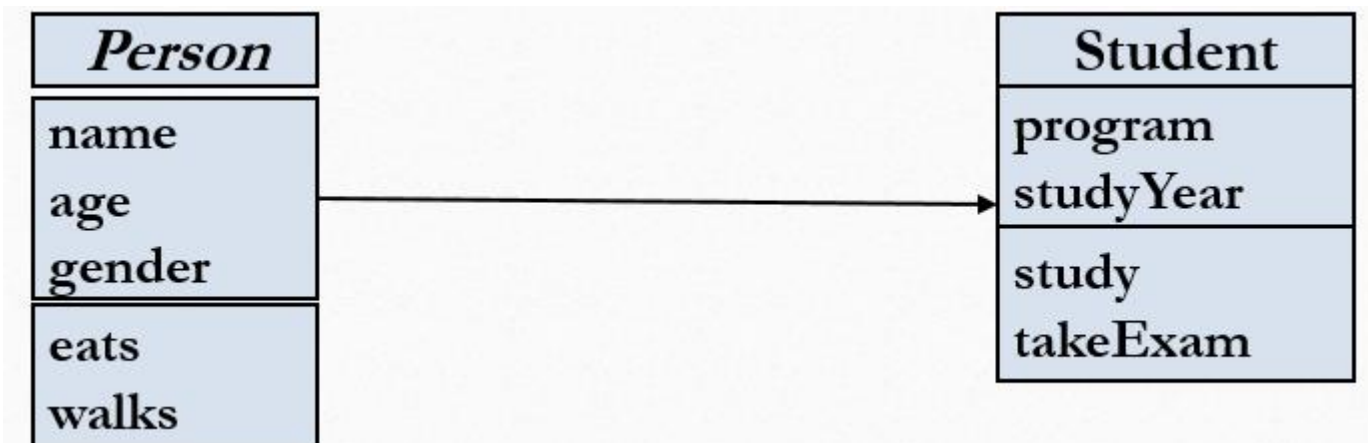


Example Generalization



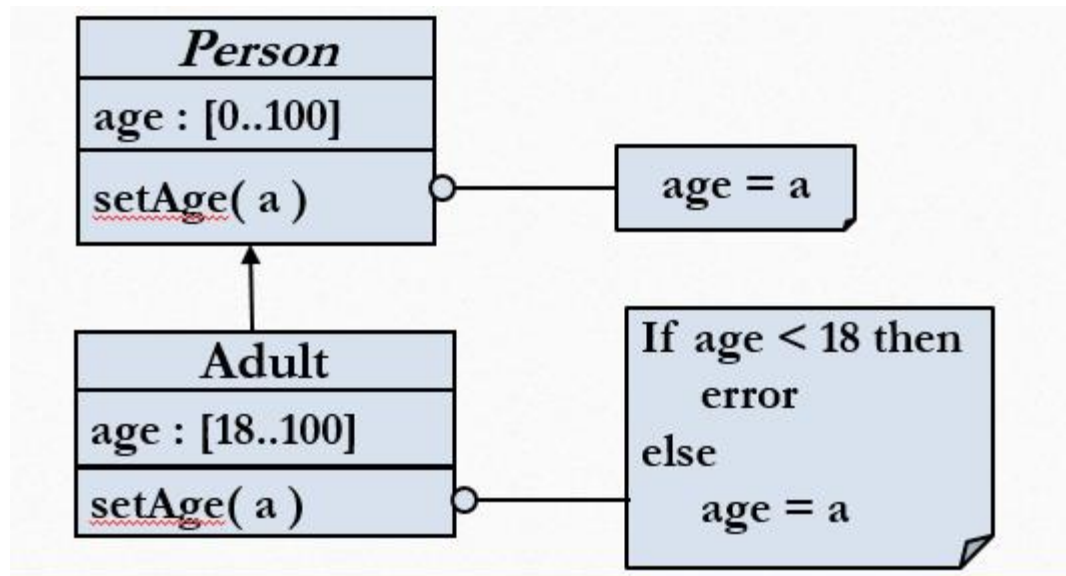
Subtyping and Specialization

- ▶ We want to add a new class to an existing model
- ▶ Find an existing class that already implements some of the desired state and behavior
- ▶ Inherit the new class from this class and add unique behavior to the new class
- ▶ Sub-typing means that derived class is behaviorally compatible with the base class



Specialization(Restriction)

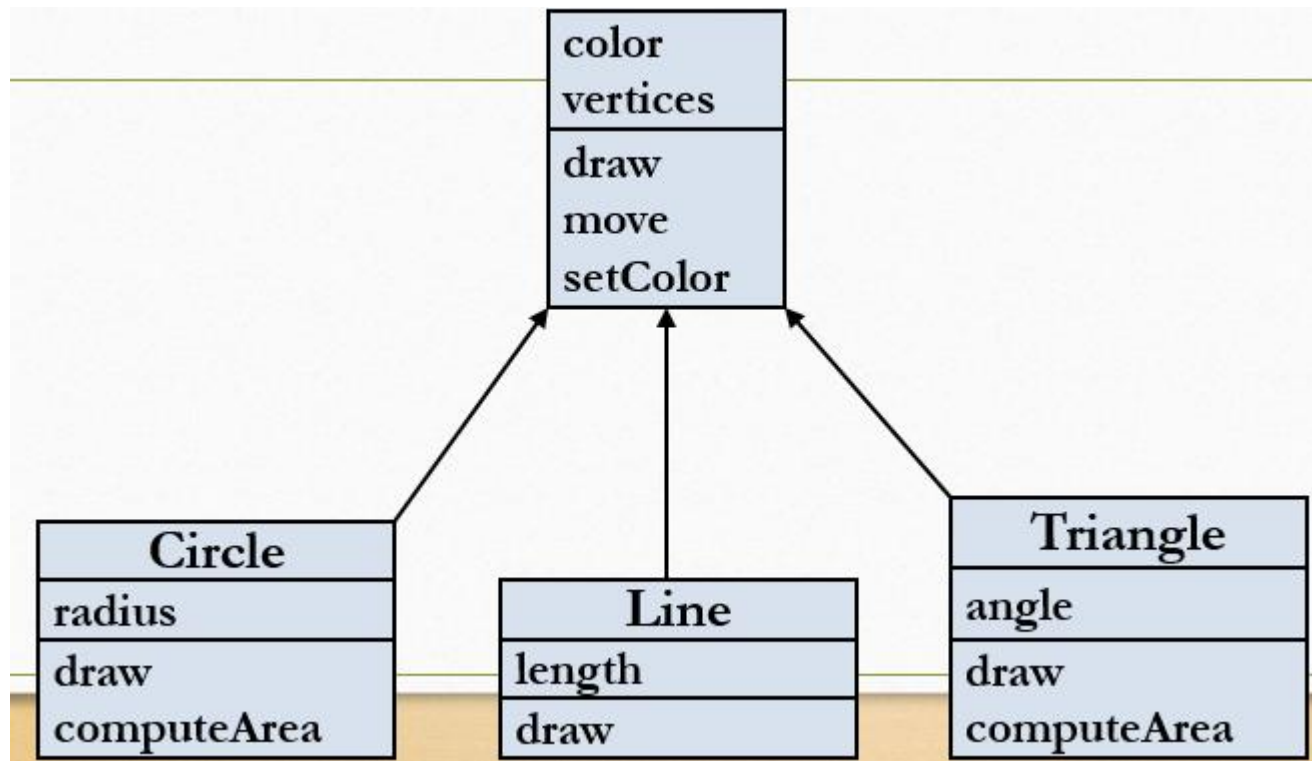
- Specialization means that derived class is behaviourally incompatible with the base class
- Behaviorally incompatible means that base class can't always be replaced by the derived class



Overriding

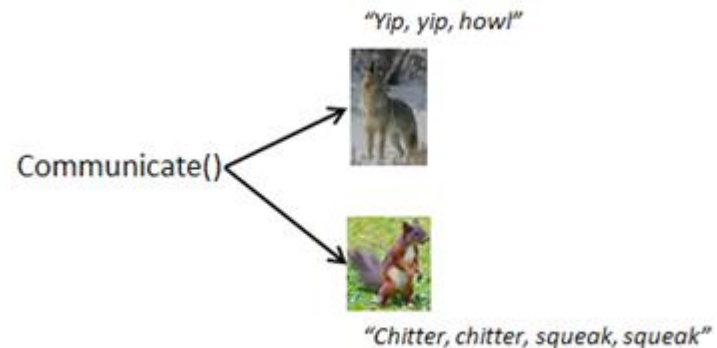
- ▶ A class may need to override the default behavior provided by its base class
- ▶ Reasons for overriding
 - ▶ Provide behavior specific to a derived class
 - ▶ Extend the default behavior
 - ▶ Restrict the default behavior
 - ▶ Improve performance

Example Overriding



Polymorphism

- ▶ Poly means many, morph means form
- ▶ Polymorphic means having many forms
- ▶ Hence Polymorphism is:
- ▶ Ability for the same code to be used with different types of objects and behave differently with each.



Types Of Polymorphism

- ▶ Compile-Time Polymorphism/ Overloading
- ▶ Run-time Polymorphism/ Over-riding

Compile Time Polymorphism

- ▶ It is also called Early Binding
- ▶ It happens where more than one methods share the same name with different parameters or signature and different return type.
- ▶ It is **known** as Early Binding because the **compiler** is aware of the functions with same name and also which overloaded function is to be **called** is known at **compile time**.
- ▶ **Overloading**
 - ▶ Function Overloading
 - ▶ Constructor Overloading
 - ▶ Operator Overloading (TO BE DISCUSSED)

Runtime Polymorphism

- ▶ Method Overriding uses runtime Polymorphism.
- ▶ It is also called Late Binding.
- ▶ Runtime Polymorphism is done using virtual and inheritance.
- ▶ When overriding a method, the behavior of the method is changed for the derived class.

```

class Shape {
    protected:
        int width, height;

    public:
        Shape( int a = 0, int b = 0){
            width = a;
            height = b;
        }
        int area() {
            cout << "Parent class area :" <<endl;
            return 0;
        }
};

```

```

class Triangle: public Shape {
    public:
        Triangle( int a = 0, int b = 0):Shape(a, b) { }

        int area () {
            cout << "Triangle class area :" <<endl;
            return (width * height / 2);
        }
};

```

```

class Rectangle: public Shape {
    public:
        Rectangle( int a = 0, int b = 0):Shape(a, b) { }

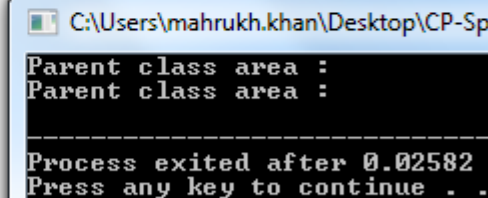
        int area () {
            cout << "Rectangle class area :" <<endl;
            return (width * height);
        }
};

```

```

// Main function for the program
int main() {
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);
    // store the address of Rectangle
    shape = &rec;
    // call rectangle area.
    shape->area();
    // store the address of Triangle
    shape = &tri;
    // call triangle area.
    shape->area();
    return 0;
}

```



```

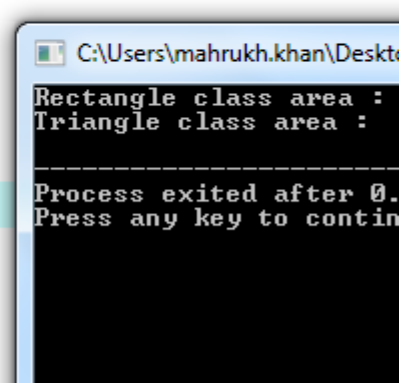
C:\Users\mahrukh.khan\Desktop\CP-Sp
Parent class area :
Parent class area :
-----
Process exited after 0.02582
Press any key to continue . .

```

Virtual Functions

- ▶ A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
- ▶ Defining in a base class a virtual function, with another version in a derived class, signals to 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**.
- ▶ When you make a function **virtual**, you are telling the compiler, “I do not know how this function is implemented. Wait until it is used in a program, and then get the implementation from the object instance.” The technique of waiting until run-time to determine the implementation of a procedure is called **late binding** or **dynamic binding**.

```
class Shape {  
    protected:  
        int width, height;  
  
    public:  
        Shape( int a = 0, int b = 0){  
            width = a;  
            height = b;  
        }  
        virtual int area() {  
            cout << "Parent class area :" <<endl;  
            return 0;  
        }  
};
```

A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Users\mahrukh.khan\Desktop". The console output displays "Rectangle class area :" followed by "Triangle class area :". Below these, a dashed line separates the output from the termination message "Process exited after 0." and the instruction "Press any key to continue".

```
C:\Users\mahrukh.khan\Desktop  
Rectangle class area :  
Triangle class area :  
-----  
Process exited after 0.  
Press any key to continue
```



```
class base {
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }

    void show ()
    { cout<< "show base class" <<endl; }
};
class derived:public base {
public:
    void print ()
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};
```

```
int main() {
    base *bptr;
    derived d;
    bptr = &d;
    //virtual function, binded at runtime (Runtime polymorphism)
    bptr->print();
    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}
```

C:\Users\mahrukh.khan\Desktop\CP-Spring2019\Lessons\Week 8\pol.exe

```
print derived class
show base class
```

Pure Virtual Function

- 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.

```
}  
virtual int area() =0; // pure virtual function|
```

- The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Example

Employee example will represent 2 types of employees as classes in C++:

- a *generic employee* (class Employee)
- a *manager* (class Manager)

For these employees, we'll store *data*, like their:

- name
- pay rate

And...we'll require some *functionality*, like being able to:

- initialize the employee
- get the employee's fields
- calculate the employee's pay

For Manager class that is inherited from Employee Class and an extra data member of type bool salaried. Define the method pay in manager class also.