Dr. Vishwanath Karad

MIT WORLD PEACE UNIVERSITY | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

MIT-WPU

।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Inheritance and Polymorphism

Contents

Inheritance

- Types of inheritance
- Virtual base class

Polymorphism

- Introduction to Polymorphism
- Types to Polymorphism: Static & Dynamic
- Virtual Function
- Abstract base Class
- Interfaces

Introduction to Inheritance

Definition

- The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.
- Ability to extend the functionality from base entity to new entity belonging to same group.
  - This will help us to reuse the functionality which is defined before.

Important points

- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
- The class which inherits the properties of other is known as subclass (derived class, child class)
- The class from which the properties are inherited is known as superclass (parent class, base class)

Why use Inheritance ?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Introduction to Inheritance

Advantages of inheritance

- Application development time is less.
- Application take less memory.
- Application execution time is less.
- Application performance is enhanced (improved).
- Redundancy (repetition) of the code is reduced or minimized so that we get consistent results and less storage cost.

Class Derivation

Any class can serve as a base class…………Thus a derived class can also be a base class

Syntax

class DerivedClassName : specification BaseClassName

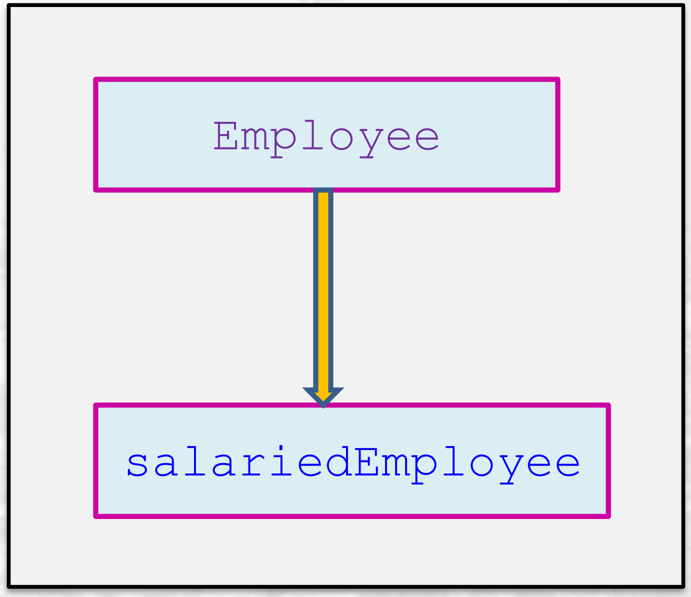
DerivedClassName - the class being derived

specification - specifies access to the base class members

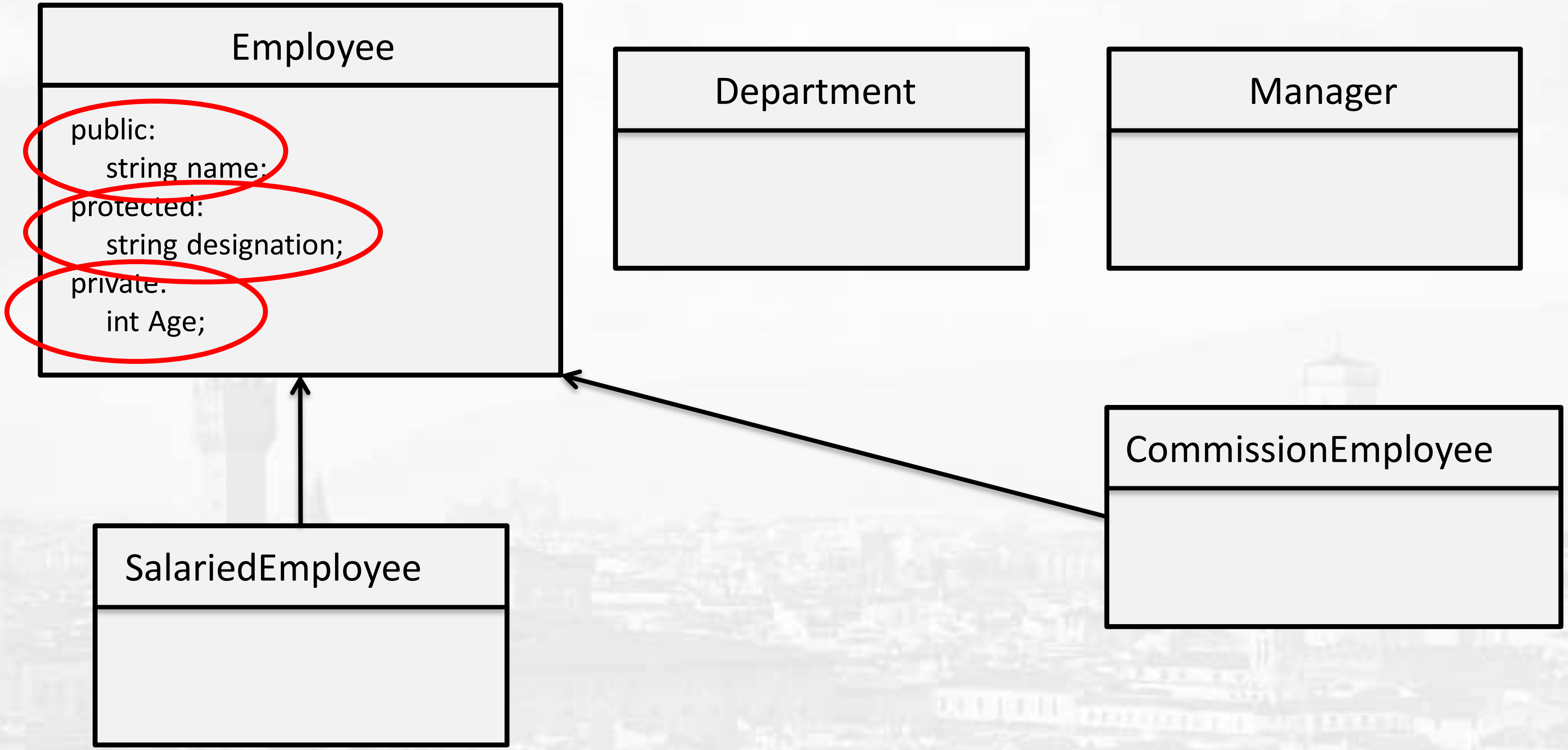
public / protected / private - private by default

Example:

```
class Employee                  // base class
{
        . . .
};
class salariedEmployee : public Employee
{                                          // derived class
        . . .
};
```



Access Specifiers



Mode of Inheritance

Public inheritance		Protected inheritance		Private inheritance	

class D: public B
{
//members of D
}
		class D : protected B
		{
		//members of D
		}
				class D : B
				{
				//members of D
				}	

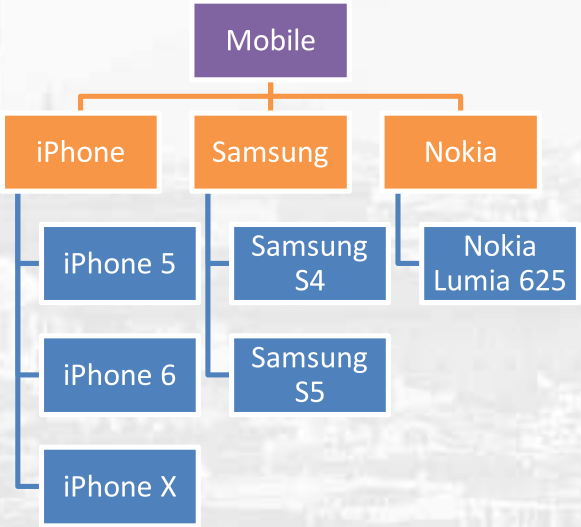
Base class	DerClass	Base class	DerClass	Base class	Der Class
public	public	public	protected	public	private
protected	protected	protected	protected	protected	private

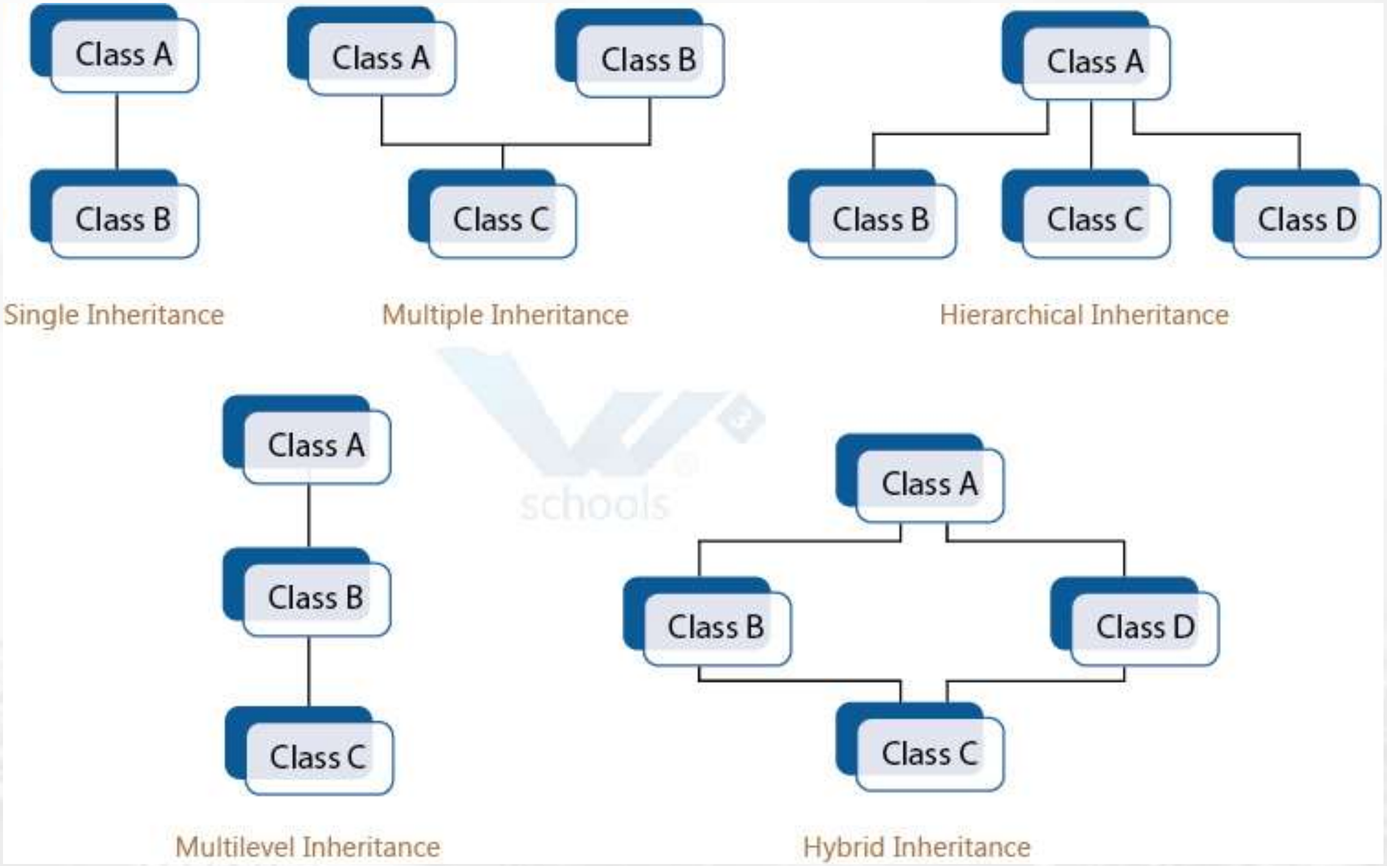
The significance of visibility modes

- Private :-When the derived class require to use some attributes of the base class and these inherited features are intended not to be inherited further .
- Public :-When the situation demands the derived class to have all the attributes of the base class,plus some extra attributes.
- Protected:- When the features are required to be hidden from the outside world and at same time required to be inheritable.

Types of Inheritance

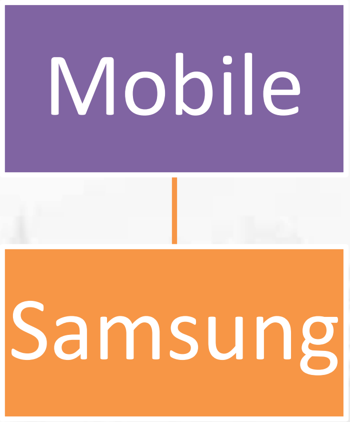
- Single level inheritance
- Multi-level inheritance
- Hierarchical inheritance
- Hybrid inheritance
- Multiple inheritance



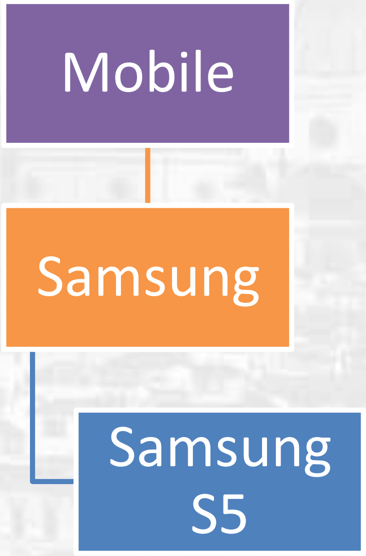


Types of Inheritance

- Single level inheritance
  - Single base class & a single derived class i.e. - A base mobile features are extended by Samsung brand.

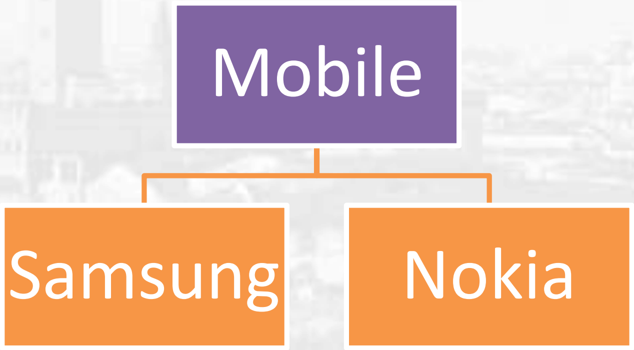


- Multi level inheritance
  - In Multilevel inheritance, there is more than one single level of derivation.
  - E.g. After base features are extended by Samsung brand, a new model is launched with latest Android OS

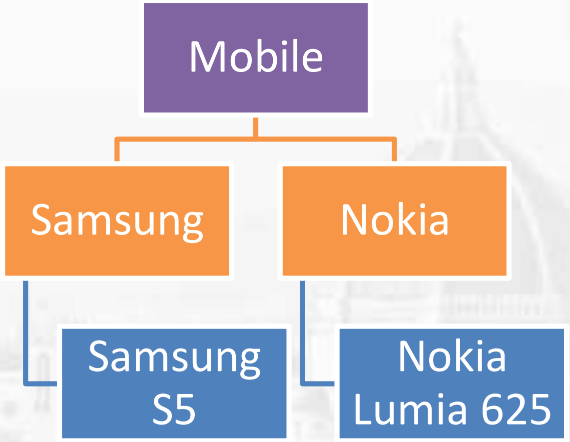


Types Inheritance

- Hierarchical inheritance
  - Multiple derived class would be extended from base class
  - It’s similar to single level inheritance but this time along with Samsung, Nokia is also taking part in inheritance.

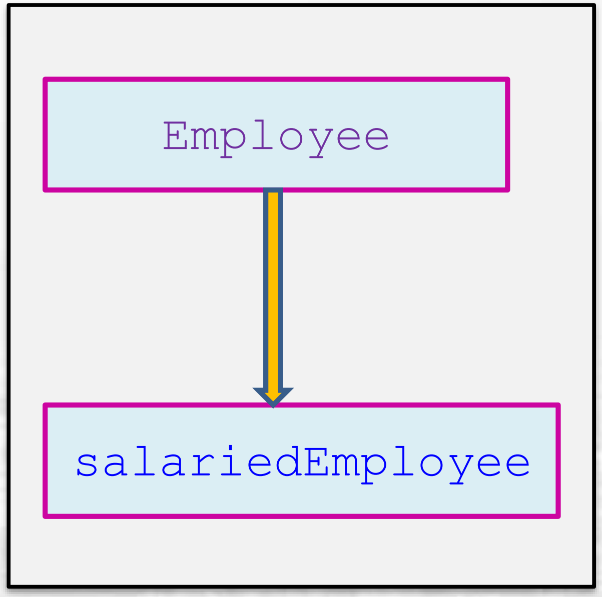


- Hybrid inheritance
  - Single, Multilevel, & hierarchal inheritance all together construct a hybrid inheritance.



Single Inheritance

```
class Employee   {  // Employee superclass
protected
              string  firstname,lastname,socialsecuritynunmber;
public:
      Employee(string first,string last,string ssn)       {
         firstName=first;
         lastName=last;
         socialSecurityNumber=ssn;
       } // end three-argument Employee constructor
    three getters &setters,one print function and one earning function
  };
class SalariedEmployee : public Employee        // SalariedEmployee subclass inherits class Employee    {
        protected:
                    double weeklySalary;
        public:
              one getter & setter,one print function and one earning function
        };
```



Example

Members of Employee Class

- Protected Members
  - Firstname
  - Lastname
  - SocialSecurityNumber

- Public Members
  - three getters &setters
  - one print function
  - one earning function

Members of Salaried Employee :Private Mode of Inheritance

- Private Members
  - Firstname
  - Lastname
  - SocialSecurityNumber
  - three getters &setters

- Protected Members
  - weeklysalary

- Public Members
  - one getter & setter(for weekly salary)
  - one print function
  - one earning function

Single Inheritance Example

```
// multiple inheritance
#include <iostream>
using namespace std;
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };
class CRectangle: public Cpolygon
 {
  public:
    int area ()
      { return (width * height); }
  };
```

```
class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);

  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- In case of multiple inheritances, the base classes are constructed in the order in which they appear in the declaration of the derived class
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Execution of base class constructor

Method of inheritance	Order of execution
class B : public A { };	A(); base constructor B(); derived constructor
class A : public B, public C	B();base (first) C();base (second) A();derived constructor

What is inherited from the base class?

- In principle, a derived class inherits every member of a base class except:
  - its constructor and its destructor
  - its overloaded operators
  - its friends

- Default constructor and destructor of base class are always called when a new object of a derived class is created or destroyed

```
derived_constructor_name (parameters):base_constructor_name (parameters)
        { …
        }
```

Constructor of base and derived class with arguments

- Pass all necessary arguments to the derived class’s constructor
- Then pass the appropriate arguments along to the base class

```
// Program to demonstrate constructor functions of both the base class and derived class with arguments
class base
{
    int i;
public:
   base(int n)
   {
     cout<<"Constructing Base \n ";
     i = n;
   }
~ base()
   {
    cout<<"Destructing Base \n ";
    }
    };
```

```
class derived : public base
{
 int j;
public:
derived(int n, int m) : base(m)
{
  cout << "Constructing derived\n";
  j = n;
}
~ derived()
 {
  cout << "Destructing derived\n";
 }
   };
int main()
{ derived obj1(10,20);
  return 0;
}
```

Output:

```
constructing base
constructing derived
destructing derived
destructing base
```

Constructor of base and derived class with arguments

```
class SalariedEmployee: public Employee{
  double weeklySalary;

public:
        SalariedEmployee(string first,string last,string ssn,double salary):Employee(first,last,ssn){
        weeklySalary=salary
}

..
};
int main(){
        SalariedEmployee salariedEmployee("John","Smith","111-11-1111",800.00);
        return 0;
}
```

Constructor of base and derived class with arguments

```
// constructors and derived classes
#include <iostream>
class mother {
  public:
    mother () {
            cout << "mother: no parameters\n"; }
    mother (int a)   {
            cout << "mother: int parameter\n";
     }
};

class daughter : public mother {
  public:
    daughter (int a)     {
            cout << "daughter: int parameter\n\n"; }
};
```

```
class son : public mother {
  public:
    son (int a) : mother (a)    {
            cout << "son: int  parameter\n\n";
     }
};

int main () {
        daughter cynthia (0);
        son daniel(0);
        return 0;
}
```

Output:

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Multiple Inheritance

One derived class with multiple base classes

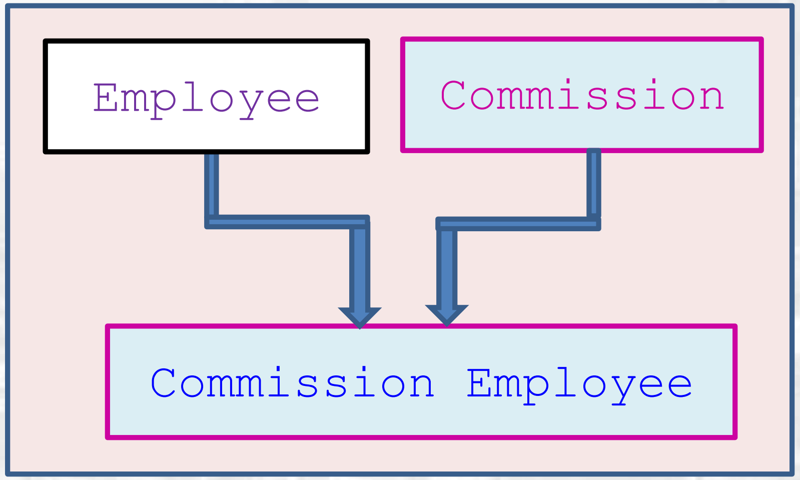
Syntax class DerivedClassName: access BaseClassName-1, access BaseClassName-2, …….

```
class Employee   // Employee superclass
{
      private:
                  string firstName, string lastName;
       public:
               . . . . .
  };
class Commission     // Commission superclass
{
        public:
 void setCommissionRate(double rate)
     {

commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
      }
         };
```

```
class CommissionEmployee : public Employee, public Commission
{
     . . . . . // derived class definition

  };
```



Multiple Inheritance

```
// multiple inheritance
#include <iostream>
using namespace std;
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };
class COutput {
  public:
    void output (int i);
  };
void COutput::output (int i) {
  cout << i << endl;
  }
```

```
class CRectangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height); }
  };
class CTriangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);

  trgl.set_values (4,5);
  rect.output (rect.area());//20
  trgl.output (trgl.area());//10
  return 0;
}
```

Constructor with Multiple Inheritance

```
// First base class
class B1
  {
       int a;
public:
       B1(int x) { a = x; }
       int geta(){ return a; }
  };

//  Second base class
class B2
  {
        int b;
public:
       B2(int x) { b = x; }
       int getb(){ return b; }
};
```

```
// Directly inherit two base classes

class D : public B1, public B2
  {
      int c;
  public:

  D(int x, int y, int z): B1(z),
B2(y)
  {
      c = x;
  }

void show()
  {
  cout << geta() << getb() << c;
   }
  };
```

```
int main()
{
  D obj1(10,20,30);
  obj1.show();
  return 0;
}
```

Output:

30 20 10

Ambiguity in Multiple Inheritance

Suppose class A and class B both have method show( )

```
class C : pubic A, public B

{

 * * *

 };

C obj1;

obj1.show();

//which of the two is called?.
```

Resolution of Ambiguity

Use the resolution operator to specify a particular method:

```
obj1.B::show();
```

- Override show( ) method in `class C` to call either one or both base class methods:

```
void C::show()
{
        B::show();
        A::show();
}
```

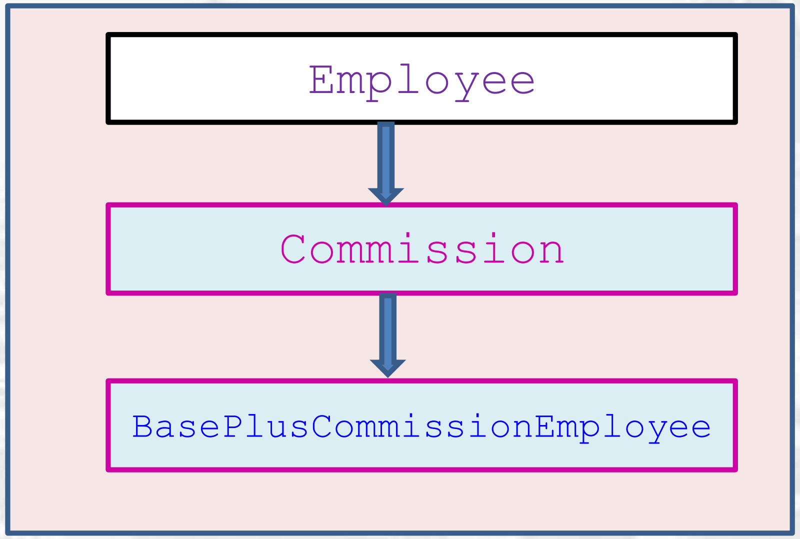
Multilevel Inheritance

Subclass can be created from another intermediate subclass

```
class Employee   // Employee superclass
{
      private:
               string firstName, string lastName;
       public:
              . . . . .
  };
class Commission: public Employee      // Commission Subclass
{
        public:
 void setCommissionRate(double rate)
     {
       commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
      }
          };
```

```
Class BasePlusCommissionEmployee : public Commission
{
    . . . . . // derived class definition

  };
```



Base Class

Intermediate Base Class

Derived Class

Overriding Member Functions

- If a base and derived class have member functions with same name and arguments then method is said to be overridden and it is called as “function overriding” or “method overriding” .
- The child class provides alternative implementation for parent class method specific to a particular subclass type.

```
class Car
{
 public:
 void maxspeed()
 {
 cout<<“Max speed is 60 mph \n”
 }
};
```

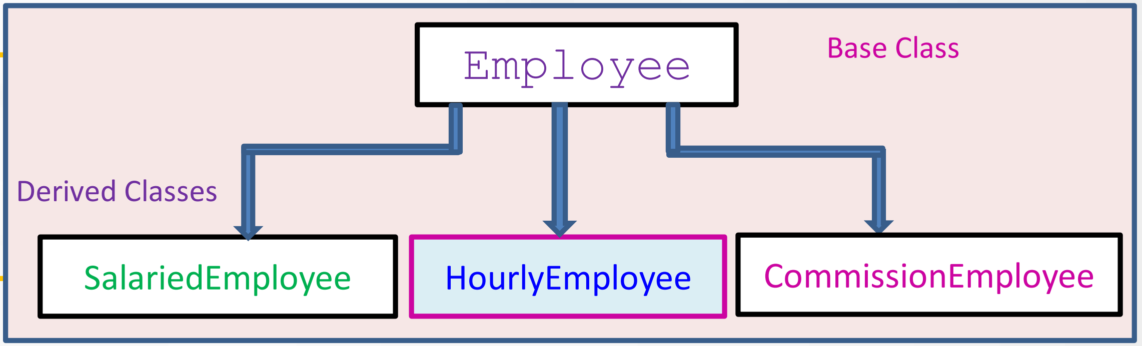
```
class Ferrari: public Car
{
 public:
 void maxspeed()
 {
 cout<<“Max speed is 120 mph \n”
 }
 public void msc(){    }
 };
```

```
int main()
{
 Ferrari f;
  f.maxspeed();
  f.Car::maxspeed();
  return 0;
}
```

Output:

```
Max speed is 120 mph
Max speed is 60 mph
```

Hierarchical Inheritance

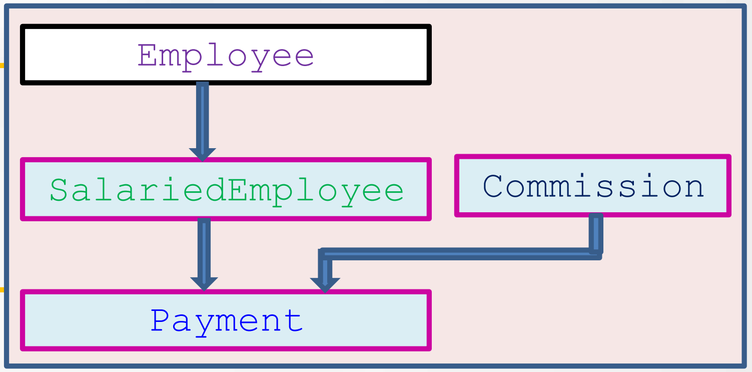


Multiple subclasses have only one base class

```
class Employee   // Employee superclass
{
      private:
                  string firstName, string lastName;
       public:
               . . . . .
   };
class SalariedEmployee : public Employee          // Subclass
{
       public:
 void setWeeklySalary (double salary)
    {
       weeklySalary=salary<0.0?0.0:salary;
      }
          };
```

```
class HourlyEmployee : public Employee        // Subclass
{    public:
 void setHours(double hoursWorked)
{      . . . . . .
  }
   };
 class CommissionEmployee: public Employee     //Subclass
{
       public:
 void setCommissionRate(double rate)
    {
       commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
     }
         };
```

Hybrid Inheritance



Any legal combination of other four types of inheritance

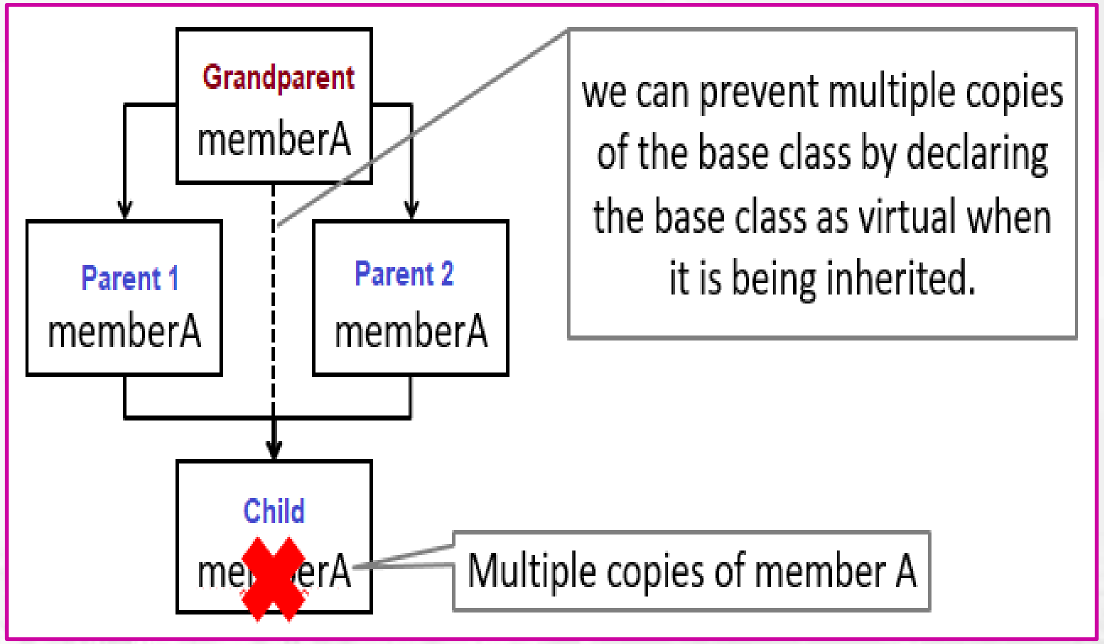
```
class Employee   // Employee superclass
{
      private:
                    string firstName, string
lastName;
       public:
                  . . . . .
  };
class SalariedEmployee : public Employee
{
        public:
 void setWeeklySalary (double salary)
     {
        weeklySalary=salary<0.0?0.0:salary;
      }
           };
```

```
class Commission              // Commission Subclass
{
         public:
 void setCommissionRate(double rate)
      {
         commissionRate=(rate>0.0&&rate<1.0)?rate:0.0;
       }
             };

class Payment : public SalariedEmployee, public Commission
 {
       double earnings()
         {  . . . . . }
   };
```

Virtual Base Class

- In hybrid inheritance child class has two direct parents which themselves have a common base class.
- So, the child class inherits the grandparent via two separate paths. It is also called as indirect parent class.
- All the public and protected members of grandparent are inherited twice into child.
- Virtual base class is used to prevent the duplication / ambiguity by making common base class as **virtual base class** while declaring the direct or intermediate base classes.



Multipath inheritance

Example: Virtual Base Class

```
class DataCompressor
{
public:
 void CompressStream();
 void DecompressStream();
//...
};
class AudioPlayer :public
DataCompressor
{
//...
};
```

```
class VideoPlayer :public
DataCompressor
{
//...
};
class MediaPlayer: public
AudioPlayer,public VideoPlayer
{
public:
 int Play();
 //...
};
```

```
int main()
{
   MediaPlayer player;
//..load a clip
player.DecompressStream();
// ambiguous call
   return 0;
}
```

Output:

Ambiguity erro

Example: Virtual Base Class

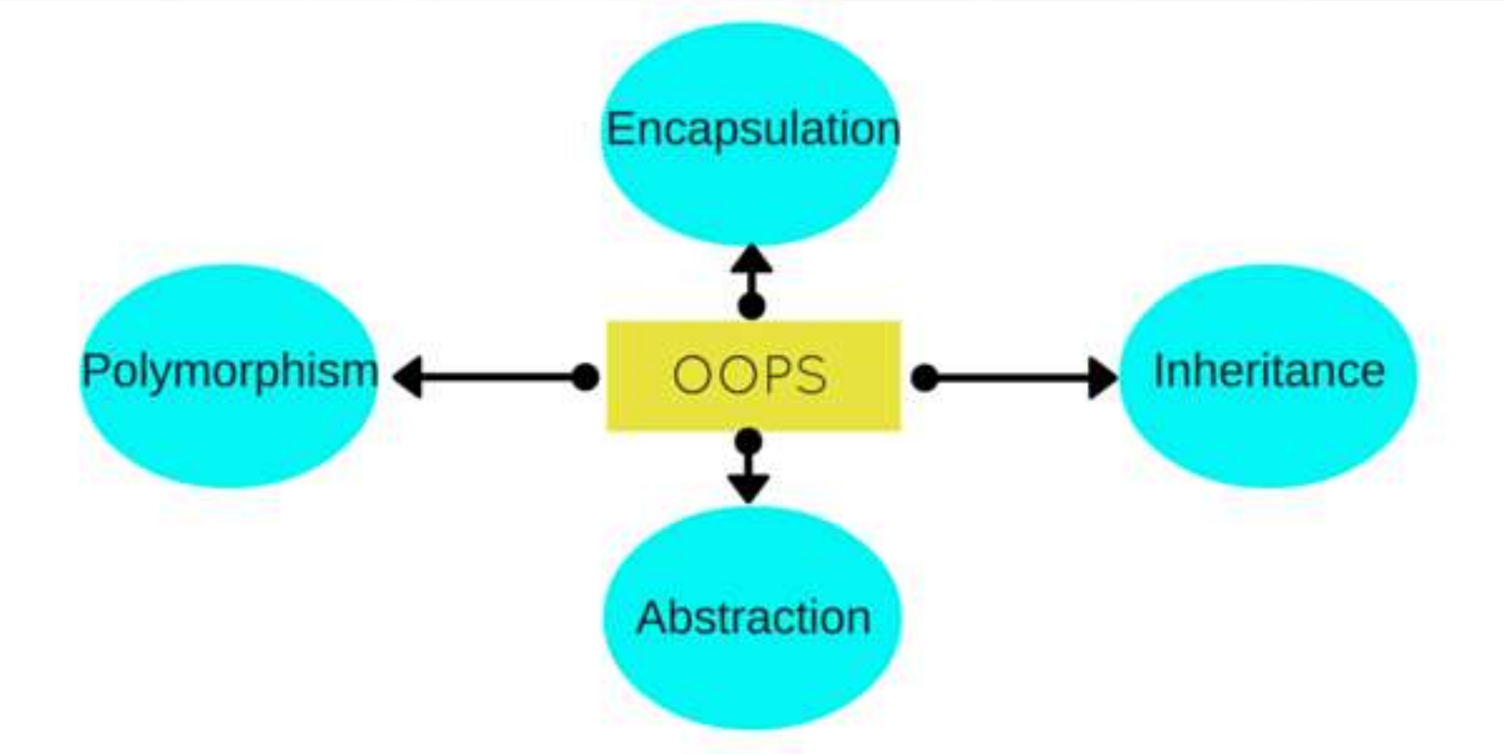
```
class DataCompressor
{
public:
 void CompressStream();
 void DecompressStream();
//...
};
class AudioPlayer :virtual public
DataCompressor
{
//...
};
```

```
class VideoPlayer : virtual public
DataCompressor
{
//...
};
class MediaPlayer: public
AudioPlayer,public VideoPlayer
{
public:
 int Play();
 //...
};
```

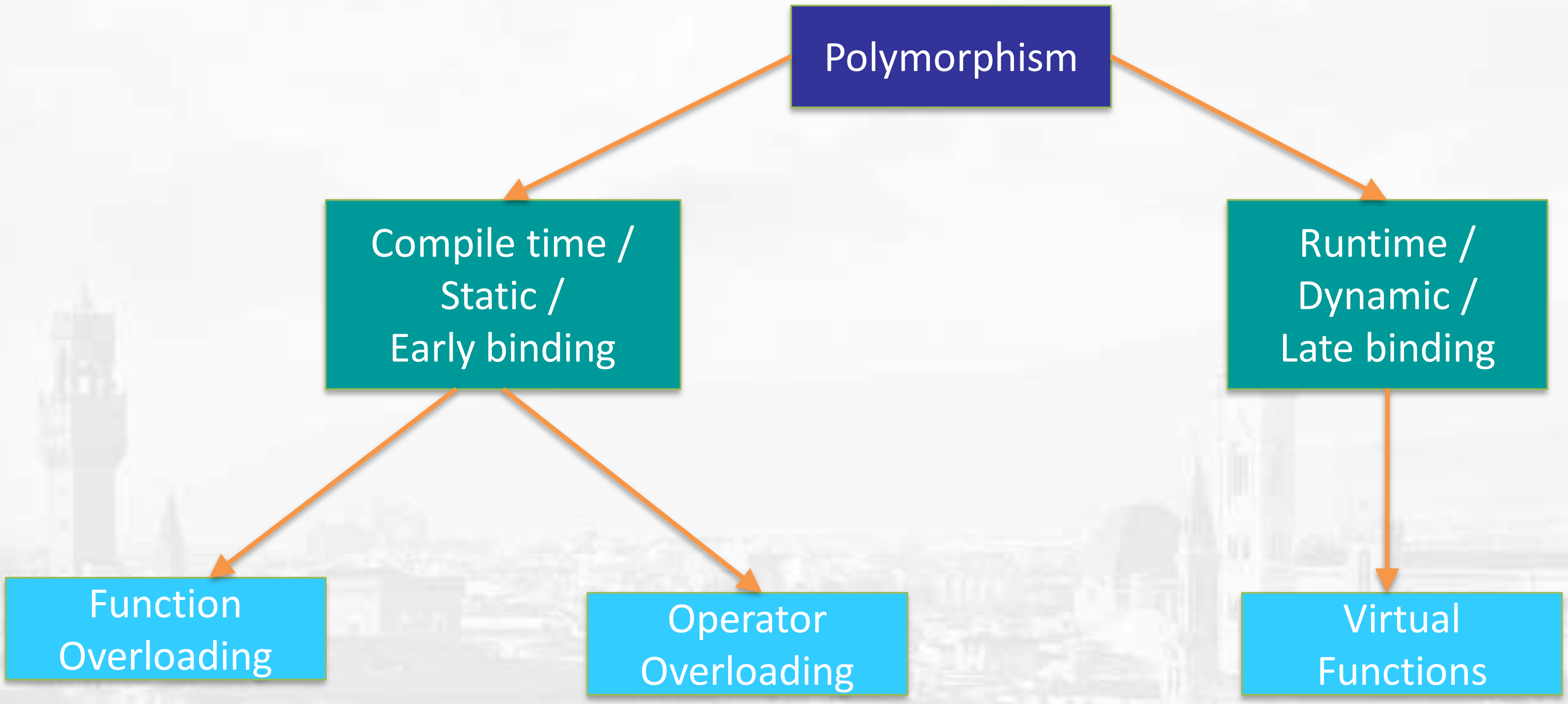
```
int main()
{
   MediaPlayer player;
//..load a clip
player.DecompressStream();
//unambiguous call
   return 0;
}
```

```
Output:
Decompress Stream
```

OOP Features



Polymorphism



Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.

```
#include<iostream>
using namespace std;
double calc_Gross_Pay(float basic, float da, float hra)
        {
                return basic + da/100*basic + hra;
        }
double calc_Gross_Pay(float hr, float wg)
        {
                return hr * wg;.
        }
double calc_Gross_Pay(float p)
        {
                return p;
        }
int main()
{
        cout << calc_Gross_Pay(basic, da, hra);
        gross = basic + da/100*basic + hra;

        cout << calc_Gross_Pay (hours, wages_Hr);

        cout << calc_Gross_Pay(pay);
        }
}
```

Operator Overloading

- C++ programming feature that allows programmer to redefine the meaning of an existing operator when they operate on class objects.
- Closely related to function overloading.
- Allows existing operators to be redefined (overloaded) to have new meaning for a specific class objects.
- Already used the + and - in overloaded fashion when add or subtract ints, floats, doubles, etc.

Operator Overloading

- Overloading of operators are achieved by creating **operator function**
- An **operator function** defines the operations that the overloaded operator can perform relative to the class
- An operator function is created using the keyword **operator**
- Operator functions can be either **members** or **nonmembers** of a class
- **Non-member** operator functions are always **friend functions** of the class

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Syntax of Operator Overloading

Overloading an operator

- Write function definition as normal
- Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
- **`operator+`** used to overload the addition operator (**+**)

General form of an operator function

keyword **Operator to be overloaded**

```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Example

```
void space :: operator – ()
{
        x = -x ;
        y = -y;
}
```

Example 1: Overloading Binary + Operator

```
class Rectangle{
        int length, breadth;
public:
   Rectangle(){ length=0; breadth=0; }
   Rectangle(int l, int b)
        {length= l; breadth= b;}

        //Binary operator overloading function
Rectangle operator +(Rectangle rec)
{
    Rectangle R;
    R.length= length + rec.length;
    R.breadth= breadth + rec.breadth;
    return(R);
}
void display(void);
};
```

```
void Rectangle :: display(void)
{
cout<<"\n Length ="<<length;
cout<<"\n Breadth="<<breadth;
}
int main()
{
Rectangle R1, R2, R3; //Creating Objects
R1 = Rectangle(2, 5);
R2 = Rectangle(3, 4);
R3 = R1 + R2; // R1 will invoke operator+()
                // R2 is passing as argument
cout<<"\n Rectangle:1  "; R1.display();
cout<<"\n Rectangle:2  "; R2.display();
cout<<"\n Rectangle:3  "; R3.display();
return 0;
}
```

```
Output:

  Rectangle:1
  Length  = 2
  Breadth = 5

  Rectangle:2
  Length  = 3
  Breadth = 4

  Rectangle:3
  Length  = 5
  Breadth = 9
```

Example 2: Overloading Binary + Operator

```
class complex{
        float x;        //real part
        float y;        //imaginary part
public:
   complex() {     }
   complex(float real, float imag)
        {x=real; y= imag;}
complex operator +(complex);
void display(void);
};
complex complex :: operator+(complex c)
{       complex temp;
        temp.x= x + c.x;
        temp.y= y + c.y;
        return(temp);
}
```

```
void complex :: display(void)
{
cout<<x<<" + j"<<y<<"\n";
}
int main()
{
complex C1, C2, C3;
C1 = complex(2.5, 3.5);
C2 = complex(1.6, 2.7);
C3 = C1 + C2;
cout<<"C1 = "; C1.display();
cout<<"C2 = "; C2.display();
cout<<"C3 = "; C3.display();
return 0;
}
```

```
Output:

  C1 = 2.5 + j3.5
  C2 = 1.6 + j2.7
  C3 = 4.1 + j6.2
```

Example 3: Overloading Binary + Operator using Friend Function

```
class complex{
        float real;   //real part
        float imag;   //imaginary part
public:
    complex() { }
    complex(float x, float y){real=x; imag=y;}
    friend complex operator+(complex& c1, complex& c2);
    void display(void);
};
complex operator+(complex& ca, complex& cb) {
        complex tmp;
        tmp.real = ca.real + cb.real;
        tmp.imag = ca.imag + cb.imag;
        return tmp;
}
void complex :: display(void){
        cout<<real<<" + j"<<imag<<"\n";
}
```

```
int main(){
complex C1, C2, C3;
C1 = complex(2.5, 3.5);
C2 = complex(1.6, 2.7);
C3 = C1 + C2;
cout<<"C1 = "; C1.display();
cout<<"C2 = "; C2.display();
cout<<"C3 = "; C3.display();
return 0;
}
```

```
Output:

  C1 = 2.5 + j3.5
  C2 = 1.6 + j2.7
  C3 = 4.1 + j6.2
```

Friend Functions/Classes

- friends allow functions/classes access to private data of other classes.
- Friend functions
  - A 'friend' function has access to all private and protected members (variables and functions) of the class for which it is a 'friend'.
  - friend function is not the actual member of the class.
  - To declare a 'friend' function, include its prototype within the class, preceding it with the C++ keyword 'friend'.

Example

```
#include <iostream>
using namespace std;
class beta; //needed for frifunc declaration
class alpha{
private:
    int data;
public:
  alpha() { data=3;} //no-arg constructor
  friend int frifunc(alpha, beta); //friend function
};
class beta{
private:
  int data;
public:
  beta()  { data=7;} //no-arg constructor
  friend int frifunc(alpha, beta); //friend function
};
```

```
int frifunc(alpha a, beta b) //function definition
{
return( a.data + b.data );
}
//-------------------------------------------------------------
int main()
{
alpha aa;
beta bb;
cout << frifunc(aa, bb) << endl; //call the function
return 0;
}
```

```
Output:
  10
```

Increment/Decrement Operator(++,--) Overloading

- int y;
- y = ++x; // Value of x is incremented, then x is assigned to y.
- y = x++; // x is assigned to y, then value of x is incremented.

```
Ret-type operator++()       {
    //body of pre-increment operator
      }
```

```
Ret-type operator++(int) {
  //body of post-increment operator
  }
//int inside () indicates postfix operator
```

```
class Complex {
            double real, imag;
 public:
            Complex() {
                        real = imag = 0;
}
  Complex& operator++(void); // Pre-increment
Complex operator++(int);   // Post-increment operator
  };
```

Restrictions on Operator Overloading

- Precedence or associativity of an operator cannot be changed by overloading
    - Use parentheses to force order of overloaded operators in an expression
- C++ does not allow new operators (the symbols themselves) to be created
- Number of operands an operator takes cannot be changed
    - Unary operators remain unary, and binary operators remain binary
- Cannot overload the meaning of operators if all arguments are primitive data types
    - i.e. No overloading operators for built-in types
        - Cannot change how two integers are added

Operators that cannot be overloaded

::	scope resolution operator
.	direct member access operator
.*	direct pointer to member access operator
sizeof	size of object operator

Accessing Members of Base and Derived Classes using an object

```
class Shape {
        public: void rotate() {
                cout << “shape:rotate" << endl;
        }
        void draw() {
                cout << “shape:draw" << endl;
        }
};
class Circle: public Shape {
        public: void rotate() {
                cout << “circle:rotate" << endl;
        }
        void scale() {
                cout << “circle:scale" << endl;
        }
};
```

	Output
Shape s;	
s.rotate();	shape:rotate
s.draw();	shape:draw
Circle c;	
c.rotate();	circle:rotate
c.draw();	shape:draw
c.scale();	circle:scale

Accessing Members of Base and Derived Classes using a pointer

```
class Shape {
        public: void rotate() {
                cout << “shape:rotate" << endl;
        }
        void draw() {
                cout << “shape:draw" << endl;
        }
};
class Circle: public Shape {
        public: void rotate() {
                cout << “circle:rotate" << endl;
        }
        void scale() {
                cout << “circle:scale" << endl;
        }
};
```

```
Shape *sptr;
sptr-> rotate();
sptr-> draw();

Circle *cptr;
cptr-> rotate();
cptr-> draw();
cptr-> scale();
```

Output

```
shape:rotate
shape:draw

circle:rotate
shape:draw
circle:scale
```

Early Binding

- Early binding refers to events that occur at compile time.
- Occurs when all information needed to call a function is known at compile time.
- Examples : Standard library functions, overloaded function calls, and overloaded operators.
- The main advantage to early binding is efficiency.

Early Binding- Example

```
class Animals {
    public: void sound() {
        cout << "This is parent class" << endl;
    }
};

class Dogs : public Animals {
    public: void sound() {
        cout << "Dogs bark" << endl;
    }
};
```

```
Animals  *a;
Dogs  d;
a = &d;
a -> sound();  // early binding
```

Output:

This is parent class

Late Binding

- Late binding refers to function calls that are not resolved until run time.
- Virtual functions are used to achieve late binding.
- The main advantage to late binding is flexibility.
- As a function call is not resolved until run time, late binding has slower execution times

Late Binding- Example

```
class Animals {
    public: virtual void sound() {
        cout << "This is parent class" << endl;
    }
};

class Dogs : public Animals {
    public: void sound() {
        cout << "Dogs bark" << endl;
    }
};
```

```
Animals  *a;
Dogs  d;
a = &d;
a -> sound();  // late binding
```

- access to methods is determined at run time by the *type of the object*

Output

Dogs bark

Early Binding & Late Binding

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Virtual Functions

- **Virtual Function** is a member function of the base class which is overridden in the derived class.
- Compiler performs **late binding** on this function.
- To make a function virtual, we write the keyword **virtual** before the function definition.
- A virtual member function in a base class automatically becomes virtual in all of its derived classes.
- A class that declares or inherits a virtual function is called a *polymorphic class*.

Virtual Function Example

```
class Animals {
    public: virtual void sound() {
        cout << "This is parent class" << endl;
    }
};

class Dogs : public Animals {
    private: virtual void sound() {
        cout << "Dogs bark" << endl;
    }
};
```

```
Animals  *a;
Dogs  d;
a = &d;
a -> sound();  // late
    binding
```

Output

Dogs bark

We can also call private function of derived class from a base class pointer by declaring that function in the base class as virtual.

Pure Virtual Function

- **Pure virtual function** is a virtual function which has no definition.
- Also called **abstract functions**.
- To create a pure virtual function, we assign a value **0** to the function.
- **Eg: virtual void sound() = 0;**
- Tells compiler that there *is no* implementation.

Abstract Class

- Abstract class is also known as **Interface**.
- An **abstract class** is a class whose instances (objects) can't be made.
- Objects of subclass can be made if they are not abstract.
- **An abstract class has at least one abstract function (pure virtual function).**
- Abstract class can have normal functions and variables along with a pure virtual function.
- If even one pure virtual function is not overridden, the derived-class will also be abstract
- Compiler will refuse to create any objects of the class
- Cannot call a constructor

Abstract Class Example

```
class Employee // abstract base class
{
    virtual int getSalary() = 0; // pure virtual function
};
class Developer : public Employee {
    int salary;
    public: Developer(int s) { salary = s; }
    int getSalary() { return salary; }
};
class Driver : public Employee {
    int salary;
    public: Driver(int t) { salary = t; }
    int getSalary() { return salary; }
};
```

```
int main() {
    Developer d1(5000); Driver d2(3000);
    int sal1, sal2;
    sal1 = d1.getSalary();
    sal2 = d2.getSalary();
    cout << "Salary of Developer : " << sal1 << endl;
    cout << "Salary of Driver : " << sal2 << endl;
    return 0;
}
```

Output

Salary of Developer : 5000
Salary of Driver : 3000

Example Payroll System

```
class Employee{              //abstract base class Employee
    protected: string name;
    public:
        Employee(string);
        void setName(string);
        string getName();
        virtual void display();            //virtual function
        virtual double earnings()=0;                //pure virtual function
};
```

Example Payroll System contd…

```
Employee::Employee(string sname){
    name=sname;
}
void Employee::setName(string sname){
   name=sname;
}
string Employee::getName(){
    return name;
}
void Employee::display(){
    cout<<"name is:"<<name;
}
```

Example Payroll System contd…

```
class SalariedEmployee: public Employee { //inherited class SalariedEmployee
    protected: double salary;
    public:
        SalariedEmployee(string,double);
        void setSalary(double);
        double getSalary();
        void display();
        double earnings();
};
```

Example Payroll System contd...

```
SalariedEmployee::SalariedEmployee(string sname,double sal):Employee(sname){  ///calling base class constructor
    salary=sal;
}
void SalariedEmployee::setSalary(double sal){
    salary=sal;
}
double SalariedEmployee::getSalary(){
    return salary;
}
void SalariedEmployee::display(){  //method overriding
    Employee::display();
    cout<<"Earnings is:"<<earnings();
}
double SalariedEmployee::earnings(){  //method overriding
    return getSalary();
}
```

Example Payroll System contd…

```
int main() {
    SalariedEmployee s(“John",25000);
    s.display();
    return 0;
}
```

Virtual Destructor

- Calling the destructor of base class, does not destruct the memory of derived class.
- This problem can be fixed up by making the base class destructor virtual.
- We can ensure that the derived class destructor gets called before the base class destructor.

Destructor-Example

```
class a{
     public:
          a(){ printf("\nBase Constructor"); }
          ~a(){ printf("\nBase Destructor"); }
};

class b : public a {
     public:
          b(){ printf("\nDerived Constructor"); }
          ~b(){ printf("\nDerived Destructor"); }
};
```

```
int main()
{
   a* obj=new b;
   delete obj;
   return 0;
}
```

Output:

Base Constructor
Derived Constructor
Base Destructor

Virtual Destructor-Example

```
class a{
     public:
          a(){ printf("\nBase Constructor"); }
          virtual ~a(){ printf("\nBase Destructor"); }
};

class b : public a {
     public:
          b(){ printf("\nDerived Constructor"); }
          ~b(){ printf("\nDerived Destructor"); }
};
```

```
int main()
{
   a* obj=new b;
   delete obj;
   return 0;
}
```

Output:

Base Constructor
Derived Constructor
Derived Destructor
Base Destructor

Summary

- Inheritance is the mechanism that provides the power of **reusability** and **extendibility**.
- Polymorphism makes systems extensible and maintainable.
- With single inheritance, a class derived from one base class. With multiple inheritance, a class is derived from more than one direct base class.
- A derived class is more specific than its base class and represents a smaller group of objects.
- Every object of a derived class is also an object of that class's base class. However, a base-class object is not an object of that class's derived classes.
- A derived class cannot access the private members of its base class directly; allowing this would violate the encapsulation of the base class. A derived class can, however, access the public and protected members of its base class directly.

References

- Reference Books:

1. Herbert Schildt, ‘C++ The Complete Reference’, Fourth Edition, McGraw Hill Professional, 2011, ISBN-13: 978-0072226805
2. Robert Lafore, ‘Object-Oriented Programming in C++’, Fourth Edition, Sams Publishing, ISBN: 0672323087, ISBN-13: 978-8131722824
3. Bjarne Stroustrup, ‘The C++ Programming language’, Third Edition, Pearson Education. ISBN: 9788131705216
4. K. R. Venugopal, Rajkumar Buyya, T. Ravishankar, ‘Mastering C++’, Tata McGraw-Hill, ISBN 13: 9780074634547

- Weblinks:

1. http://nptel.ac.in/syllabus/106106110/
2. http://ocw.mit.edu

- MOOCs:

1. https://www.coursera.org/learn/c-plus-plus-a
2. https://www.mooc-list.com/tags/c-1